**ASP.NET MVC NOTES**

* **App\_Start:**
* FilterConfig: Filters added to all controllers
* RouteConfig: Routing for the application. We can define default and custom routes/URLs
  + - * BundleConfig: Packages for static resources such as JS, CSS etc for faster performance
* **Global.asax:** Contains all methods that fire up when the application starts and points to the App\_Start folder and files

**Setup Entity framework**

* Add NuGet package
* DataAccessFolder or Project
* Add ***DataContext*** Class
* Inherit from ***DbContext*** class
* Route to the SQL server by adding the ***connection string name*** to the constructor to match the ***WebConfig*** file
* Add the data models as ***DbSet*** collection

**WebApi**

* Create a model in the project or associate to another
* Add controller (empty)
* Create a collection of the model or get them from another source

**Allow connection to external APIs (Cross-Origin Resource Sharing, CORS)**

* NuGet package **Microsoft.AspNet.WebApi.Cors**
* WebApiConfig.cs > Register > Add ***config.EnableCors()*** method on the top of the body. This enables the api to be connected to externally
* Controller > Add attribute [**EnableCors**(**origins**:””, **headers**:”\*”, **methods**:”\*”). The domain that will connect to the API should go in the string, e.g. <http://localhost:2345>.

**Main App**

* Add empty controller and associate with view, create the layout etc
* Add an AJAX call to the api

Run both by setting them to run both from **Set Startup project**

**Controller Action Results/Returns**

1. **View:**
2. **Empty:** 200 OK, Content-Length: 0 (Not commonly used)
3. **Redirect:** 300 Redirect, redirect to another controller specified in the arguments
4. **Json:** 200 OK, returns JSON
5. **JavaScript:** 200 OK, streams JS file or string to the client
6. **Content:** 200 OK, any string with any content type
7. **FileContent,**

**FilePath,**

**FileStream:** 200 OK, stream files. Works with byte arrays, streams and file path strings

**Attribute routing**

Can be used with and without default routing. It specifies routing for each controller individually to avoid conflicts etc.

* RouteConfig.cs: add ***routes.MapMvcAttributeRoutes();***
* (Optional) Remove ***routes.MapRoute()***
* Use ***[Route(“RouteName”]*** .
* **Custom Parameters:** add them in curly braces after the ***RouteName*** and **/** and match them to the parameters of the controller.

e.g. ***[Route(“RouteName/{username}”]*** will be matched to ***string*** *username* in the controller

* **Constraints:** add them after the parameter following **: .**

e.g. ***[Route(“RouteName/{userId:int}”]***

* **Custom constraints:** new class that implements the ***IRouteContstraint*** interface. Add them to the ***RouteConfig.cs*** by creating an instance of the ***DefaultInlineConstraintResolver***class and adding the custom constraint to the ***ConstraintMap*** list. Use them as you will in the Route Attributes
* **Optional:** add **?** after the parameter both in the attribute and the controller
* Use ***[RoutePrefix(“ControllerName”)]*** before the class declaration so that the controller prefixes all the strings used in the attribute. e.g. **[Route(“Courses”]** from the **Course** controller, if prefixed with ***[RoutePrefix(“Course”)]*** will result to /**Course/Courses** in the url.
* Override prefix by adding ***~/***  before the string in the ***[Route(“”)]*** .

Route attribute can be used on the controller level

**Filters**

1. **Authentication:** Runs first and verifies identity via a cookie
2. **Authorization:** allows access to parts of the website(controllers) if the user is verified

**[Authorize]**:

* + - **WebConfig: <System.web>**-> **<authentication** mode**=”**Forms**”>** -> **<forms** loginUrl**=”**e.g. /login**”** defaultUrl**=”/”>**

Redirects to login page or the specified url or to the site root

* Create controller in the Login class that has **[HttpPost]** attribute and takes **string** *username* and **string** *password* as parameters
* Check parameters if valid
* Call the **FormsAuthentication.SetAuthCookie(username, false);**

This authorizes the user to access locked controllers

* **return Redirect(FormsAuthentication.GetRedirectUrl(username,false));**

This will redirect to the specified controller if the credentials are correct.

1. **Action:** they run before, after or both when an Action Method is executed. This is specified by the OnActionExecuting or OnActionExecuted handlers.
   * Create new class
   * Inherit from the **ActionFilterAttribute**
   * Override **OnActionExecuting/Executed**
   * The **filterContext** contains various info to be used accordingly.
   * Add filter to the FilterConfig or to a specific controller

e.g. filters.Add(new FilterName());

1. **Result:** they run after the action method and after the view result is rendered.
2. **Exception:** they run if an exception is thrown and is not caught earlier

* **WebConfig: <System.web>**-> <**customErrors** mode=”On”>

This tells the handler to redirect to the default error page if there is an uncaught exception.

The default page is in **Shared/Error.cshtml**

* Create new class
* Inherit from the **FilterAttribute** and **IExceptionFilter**
* Implement the **OnException** method
* filterContext contains info. **Result** and **ExceptionHandled** properties need to be set
* add filter to filter list

**STRONGLY TYPED MODELS**

1. Add model object to the parameters of the Controller that will use it and return it in the View()
2. If we are using Entity Framework, the model should be bound in the controller and then be passed to the View. Binding should happen either in the controller class or a View Model.
3. @model <modelName> on top of the html that will use it
4. Use the Model inside the Html

**Model Validations/Data annotations**

1. using System.ComponentModel.DataAnnotations
2. Add attributes before each property that should be validated

e.g. [Required],

[MinLength()],

[MaxLength()],

[EmailAddress],

[DataType],

[Compare],

[Range],

[Regex]

Etc

1. **Custom Validations:**
   1. new class which inherits from *ValidationAttribute*
   2. override *IsValid* method with needed logic
   3. override *FormatErrorMessage* method with custom message
   4. use custom validation as all others

1. **ModelState** validation: ModelState.IsValid -> return sth
2. **HTML:** *@Html.ValidationMessageFor(lambda for model property)* for each required field. It will display the message specified in the data annotation. *@Html.ValidationSummary()* displays all error messages.

**HTML HELPERS**: c# methods to be used inside cshtml pages in order to create html elements

Add html properties with ***new { @class = “” etc}*** in the specified constructor position

1. @**Html.Label**
2. @**Html.LabelFor**(lambda to select property from the Model, etc)
3. @**Html.TextBox**
4. @**Html.TextBoxFor**
5. @**Url.Action**(“controllerAction”,”ControllerName”) -> html anchor
6. @**using (Html.BeginForm**(“controllerAction”,”ControllerName”, returnUrl(??), FormMethod.Post(Http request), htmlProperties ))
7. @Html.ValidationMessageFor(lambda for model property

*Classes that implement the IDisposable interface can be used in a using code block. It is instantiated in the using() starting point and the dispose method is created and executed automatically in the end of the code block.*

**Custom helpers:**

A)

1. **Top side: @helper** *HelperName(parameters){ body }*
2. **Inside: @**HelperName

B)

**1.** Create folder for extensions

**2.** Create static class ***HtmlHelperExtensions***

**3.**  Put class in the **System.Web.Mvc** namespace where the helper class comes from

**4.** Create extension method to the HtmlHelper

public static output <MethodName>(this HtmlHelper helper){ body }

*Strings are always encoded to HTML in MVC 5, so, if the return is of type string and is to be used as an element, change it to* ***IHtmlString*** *and the return should be helper.Raw(“string”) which also returns IHtmlString. This also applies to cshtml. When we need to write an element, we use @Html.Raw(“html code”)*

1. Use extension method in the cshtml: @Html.<HelperName>()

**Shared Layouts and Partial Views**

*The layout is structured like a regular html page (Header, styles, body,footer, scripts). In the body section, there is the* ***@RenderBody()*** *which renders each html page through the controller.*

1. Create or use an existing layout in the shared folder
2. Add it to the top of the cshtml that will use it

e.g. @{

Layout = “~/Views/Shared/\_LayoutName.cshtml”;

}