PhotoShare

- Sign in/Sign out of your account
- Create your user profile
- Create a new album, and upload your photos to it
- Tag those photos
- Comment on photos
- Search for photos by tag, album, user

Passing a model to a view

In your action, create a model object, and do return View(model). Then in your view, put this at the top:

@model <Application>.Models.<ModelName>

And then access the properties of this model in your view via @Model

Unit Testing

Specify to create a unit testing project at the 'Type of Application' tab when you're creating a new asp.net mvc application

See the tests in <Application>.Tests projects, and Run tests by Ctrl+R, A

[TestClass], [TestMethod] attributes

In each test method, you need to make an instance of the controller, an instance of the action that you're testing as a ViewResult, and then do some assertions with that ViewResult

Routing

Route table is in App_Start/RouteConfig.cs

RouteData data structure can be used to access parts of the url. They're stored by key value pairs.

For ex: RouteData['controller'] gets you the name of the controller in the URL

To define your own routes, you wanna place them above the Default route, because the routes are handled in the order in which they are placed

Actions

If you just wanna send a string back to the browser in an action, just do return Content("<your string>")

If you've defined some parameter in your RouteConfig for a url, then you can just access that parameter by passing an argument to your action. So for example, in the photos/search/{tag} , you can pass string tag as an argument to your action method, and its value will be available

If you define an argument in your action, it'll be available as a querystring key. So you could do photos/search?tag=vacation and the value of tag will still be available in your action

Action Results

- If you wanna redirect your action to another action in a controller, you can use
    - return RedirectToAction(action,controller,new {parameter1= value,…})
- If you wanna redirect to a route from the RouteConfig, you can use RedirectToRoute
    - return RedirectToAction(route name, new {controller=”controller”, action=”Action”})
- If you wanna return a file, do
    - return File(Server.MapPath(“~/<path from root folder of website>”), file type)
        - NOTE: ~ represents the root folder of the website.
- If you wanna return a json result, do
    - return Json(some object, JsonRequestBehavior.AllowGet);
    - NOTE: first parameter can also be an anonymous object. For ex:
        - return Json({name=”nirav”,age=21,school=”GT”},JsonRequestBehavior.AllowGet)

Accept Verbs

- If you wanna specify controller actions with the same name but corresponding to different types of http requests, like if you want a Home/Index action to respond differently in case of a GET and a POST request, you can use the Accept Verbs [HttpGet] and [HttpPost]. You just put them right above your action, and ASP.NET will figure out which one to go to in case of a get or a post request. If you don’t use these and specify two different actions with the same name, then ASP.NET will throw a “ambiguous actions” error.

Action Filters

- They are fired before an action.
- You can also put them above the controller and then the filter will be applied before every action in the controller
- Examples:
    - [Authorize] – redirects the user to the login page if the user isn’t logged in
- If you want some filter to be applied to every action in every controller without having to put the filter at the top of every controller, you can use a global filter for this purpose. Global filters are registered in the Global.asax file, and are added to the list of filters in App_Start/FilterConfig.cs file.
    - The HandleErrorAttribute filter here basically shows friendly error messages to the user when something goes wrong.
    - To show pretty errors to the end users instead of the usual “Server Error” (Yellow screen of death) error message showing code pieces, put this in the system.web section of Web.config file:
        - <customErrors mode="On"/>
            - If the mode is set to RemoteOnly, then during localhost, you’ll see the usual yellow screen of death, but the end user will see the pretty page
    - If you wanna change how the above error message looks, you can edit the Error.cshtml file in Views/Shared folder

- You can define your own filters in Filters folder, by creating a class that extends the ActionFilterAttribute and ends in “Attribute”. So the format is:
    - public class <Filter name>Attribute : ActionFilterAttribute
- There are four methods that you can override in your custom filter:
    - OnActionExecuting
    - OnActionExecuted
    - OnResultExecuting
    - OnResultExecuted

Creating a Model

Right click the Models folder and add a new class

Creating a Controller

Right click the Controller and click ‘Add Controller’. If you wanna add basic read/write actions but not generate views yet, then choose the option “MVC controller with empty read/write actions”

Generating a View for a certain action in your controller

Right click inside the action and click ‘Add View’

Razor

- A statement that starts with @ is basically C# code
    - @VirtualPath gives you the path from the root folder of the website to the view’s file. So if you’re in Photos/Index.cshtml view, and you put @VirtualPath, then you’ll be returned a string ~/Views/photos/Index.cshtml
- Razor’s automatically encodes the text output, so that xss attacks can be prevented. If you have a script tag in your string from the controller, then the < will be treated as < and not as <. If you didn’t want Razor to encode the output, you can use @Html.Raw(<your string>)
- Comments: @* this is a comment *@
- Explicit code expression:
    - You surround the code after @ in parenthesis to make it evaluate the code individually instead of evaluating in the context of what’s after and before the expression
        - Doing R@item.Rating is treated as an email address, but doing R@(item.Rating) is treated as R concatenated with the rating of the item, for ex, R10
        - Doing @item.Rating / 10 is evaluated as: “display the rating of them and then / 10”, but @(item.Rating/10) is evaluated as: “divide the rating of the item by 10 and then display it
- If you just put @something, Razor will throw an error because something is not a valid C# code expression. But if you put @@something, then the @ will be escaped. So if you literally wanna put an @ sign and not a code expression, you’d have to do @@
- Code Blocks
    - Code blocks are denoted by curly braces @{ <code> }. Variables you declare here are visible throughout the View
    - @foreach(<expression>){ } can be used to iterate through a list

    - Inside of a code block, you can't just put anything. You can only put HTML tags or c# code using @. If you wanna put some literal stuff, like if you put @foreach(<expression>){ Photo }, this would throw an error. So if you wanna put literal text in C#, you can use @: So to literally put Photo, we can do @:Photo. But you can't put @: inside an HTML tag. For some reason that causes an error
- @Url.Content(<some path relative to root>) can be used in hrefs or src fields of html tags to get the right path of the file or the link you want. Read more about it here: http://blog.webactivedirectory.com/2011/09/23/asp-net-use-url-content-from-razor-to-resolve-relative-urls/
- Layouts:
    - Layouts for **all** views are set in _ViewStart.cshtml in Views folder. In this file, there's just a code block that sets the "Layout" property. By default, this property is set to ~/Views/Shared/_Layout.cshtml
    - However, if you wanted the views of a certain controller to use a different layout, you can just copy the _ViewStart.cshtml file and put it in the Views folder corresponding to that controller
    - If you wanna set the Layout only for a specific view file, then you can specify the Layout property in the code block at the top. You can just do: @{Layout = "whatever"}.
    - If you don't want something to use a layout, just set Layout = null
    - Sections
        - You can define required or optional "sections" at a certain position in your layout to be displayed in your views using:
            - @RenderSection("name of section",required: <false or true>)
        - Then, in your view, you can do
            - @section name of section { code / html }
        - If you set required:false, then it won't matter if you don't include the section in your view. If you set required:true, then ASP.NET MVC will throw an error if you don't.
    - @RenderBody
        - The @RenderBody() in your layout is where the code from your view is injected.
    - HTML Helpers
        - @Html.ActionLink("link text", "action name", "controller"), @Html.ActionLink("link text", "action name", new { id = 2, …})
            - The second one is used within a view file, and the controller is set to the controller whose action this view is for
        - @Html.BeginForm()
            - Creates a form tag with default method = post and action = the url of current page. You can pass in the url and method you want.
            - @Html.LabelFor, @Html.HiddenFor, @Html.EditorFor (for this one, Razor figures out which kind of textbox (input type="text" or textarea) will be the most appropriate.)
            - When you submit this form, you can use TryUpdateModel in the HttpPost version of the action corresponding to this view and pass in the model you wanna update, and it'll automatically grab the data

submitted in this form and return True if the updating was successful. If not, it'll return False.

- Partials
    - You can create partials to re-use code (just as in Rails) by creating them either in the views folder corresponding to your controller, or in the Shared folder. In the latter approach, the partial is available in every view in the application. In the former approach, the partial is only available in other views corresponding to the controller.
    - You create a partial by Add > View > 'Create as a partial view'
    - Refer to code in Photos/_Photo.cshtml and Photos/Index.cshtml to see how to deploy partials and how to pass information to them
- @Html.Action
    - Placing this somewhere causes ASP.NET MVC to issue a child request to the action of a controller and place the output of that action wherever the @Html.Action is called. Again, refer to code in Photos/LatestPhoto action and Shared/_Layout.cshtml
    - Note: Like any other action, if this action is accessed via the url Photos/LatestPhoto, you'll get the result of that action to the browser. But if you don't want that to happen and you only want to display the output of this action in your view and not individually, you can put the [ChildActionOnly] attribute over your action.

Data

- Code First
    - You create a class in your Models folder that derives from DbContext in System.Data.Entity namespace, and you list DbSets of each of the models you specified in that folder.
    - Then, you create an instance of the database in your controller, and when that controller executes, a database will be created for you automatically.
    - To connect to this database, go to View > Database Explorer > Data Connections. Right click on Data Connections and click Add Connection. Click Change and choose Microsoft SQL Server, and specify the name of the server and choose the <Application>.Models.<ApplicationDb> database. (In your case, the server is NBDEV\sqlexpress)
- Migrations
    - http://msdn.microsoft.com/en-us/data/jj591621
    - To enable migrations, go to Tools > Library Package Manager > Package Manager Console, and type:
        1. Enable-Migrations -ContextTypeName <Application>.Models.<ApplicationDb>
        2. The above will create a folder called Migrations and put a Configuration file in it.
    - To create a new migration, run Add-Migration <MigrationName> from the Package Manager Console. To apply this migration, run Update-Database from the Package Manager Console
    - You can also enable Automatic Migrations in Configuration.cs by setting AutomaticMigrationsEnabled to true. Now every time you make a change to your models, you won't explicitly have to add new migrations. You can just run Update-Database, and the changes will automatically be applied to the database.

    - You can also provide some seed data to go every time you update your database using the Seed method. This is usually used for populating database with some basic static data like postal codes or city names etc. Check out the code in Seed method of Migrations/Configuration.cs to see an example
- View Models
    - Sometimes, if you want to pass information combined from multiple models into your view, you can create a ViewModel in your Models folder. Check out the code in HomeController.cs/Index action and Models/AlbumsListViewModel as an example
    - If you want to pass your ViewModel to your view, you'll have to declare the type of the viewmodel at the top.
- Loading Related Entities in your view:
    - When you, for example, define your Album model and put a field for ICollection of Photos, the only way you'll be able to do @Model.Photos in your view is if you declare it as **virtual**, because by default, whenever MVC creates a new instance of the model, it'll keep that Photos property as null. Refer to the Album model for the example.
- Uploading a file
    - The following things must be done for file uploading to work:
        1. Make Html.BeginForm include enctype
        2. Put a input file field with a certain name, for example, "file"
        3. Pass "file" as a parameter in your [HttpPost] method, with type HttpPostedFileBase
        4. If the file argument is not null, move the file to the appropriate directory on your server.
    - To see the above in action, refer to the post versions of Create and Edit methods in PhotosController and Create and Edit views
- Preventing Mass Assignment
    - You can put [Bind(Exclude="comma separated fields")] or [Bind(Include="comma separated fields")] right before the object parameter in your HttpPost method to prevent hackers from modifying values that shouldn't be modified in a POST request
- Validations Annotations
    - You can validate data using data annotations in your model. Data annotations are available through the System.ComponentModel.DataAnnotations namespace.
    - Some examples of data annotations are:
        1. [Range(m,n)] → used for numbers. Numbers can only be between m and n inclusive
        2. [StringLength(m)] →the string can only be as long as m characters
        3. [Required] → the field cannot be left blank. This is redundant for fields that are numbers or date based because C# cannot assign null for these values, so you can basically not include this for an integer or date field and MVC will still require you to specify the value for the field.
        4. [Display(Name="some name")] → If you want a different name to be displayed for a field in your views, for example, you want the field Description to be shown as Blurb everywhere, you can use this one.

    - NOTE: you must update your database after putting these annotations! You can use the –Force flag with Update-Database in order to make MVC ignore the fact that data might be truncated.
- Custom Validations
    - There are two ways to write your own custom validations.
        1. This one is if you wanna write custom validation annotations for specific attributes. In this one, you create a new class right above your model, make it extend the ValidationAttribute class, and override the IsValid method inside of that class. Refer to Models/Photo.cs in r29.
        2. This one is you wanna write validations concerning multiple fields. In this one, you basically make your model extend the IValidatable object and implement a Validate method. Refer to Models/Photo.cs in r30.