Quick Notes

State Management in ASP.NET

- **Client side State Management**
  - View State
  - Control State
  - Hidden fields
  - Cookies
  - Query String
- **Server Side State Management**
  - Application State
  - Session State
  - Profile Properties

In ASP.NET MVC request flows in general moves as follows:

1. The first hit comes to the controller
2. Depending on the action, controller creates the object of model. Model in turn calls the data access layer which fetches data in the model.
3. This data filled model is then passed to the view for display purpose.

Two things are very important:

1. How we name something?
2. Where we keep something?

Routing

- ASP.Net routing system has two functions:
  - Examine an incoming URL and figure out for which controller and action the request is intended.
  - Generate outgoing URLs.
- There are two ways to create routes in MVC framework application:
  - Convention based routing
  - Attribute routing
- Define a route:
  - Map between URLs and action methods

- At the time of defining routes in the RouteConfig.cs file the order matters. That's why most specific routes must be defined first.
- URLs can be broken down into Segments. These are the parts of the URL, excluding the hostname and query string, that are separated by / character.
- The segment variables are expressed using the braces (the { and } characters).
- The routing system does not have any special knowledge of the controllers and actions. It just extracts values for the segment variables.
- By default, a URL pattern will match any URL that has the correct number of segments.

- Two Key behavior of the URL patterns:
  - URL patterns are conservative, and will match only URLs that have the same number of segments as the pattern.
  - URL patterns are liberal. If a URL does have the correct number of segments, the pattern will extract the value for the segment variable, whatever it might be.
- Allow URL segments to be omitted by using default values for segment variables
- Static segments:
  - Match URL segments that don't have corresponding routing variables
  - Static URL segments can be used to create an alias for the a specific URL. This can be useful if you have published your URL schema publicly and it has formed a contract with your user. If you refactor an application in this situation, you need to preserve the previous URL format so that any URL favorites, macros, and scripts that user has created continue to work.
- Custom segment variable
  - Pass URL segments to action methods
  - To access any segment variable in the action method, we can use RouteData.Values property.
  - If we define parameters to the action method with names that match the URL pattern variable, the framework will pass the value from the URL to the action method parameter.
  - I can define a segment variable as optional by specifying the value as Url.Parameter.Optional, if no value is supplied to that optional segment variable then null will be assigned. Means if we are using any primitive type then we have to make it nullable.
  - If you want to enforce separation of concerns in MVC then you should not define the default values for the segments in the route, you should define the default value to the action method parameter.
- Allow URL segments for which there is no default value to be omitted: define optional segment
- Define routes that match any number of URL segments: use a catchall segment
  - To accept a variable number of URL segments, we have to designate one of the segments as catchall variable by prefixing it with an asterisk.
  - The segments captured by the catchall are presented in the form *segment/segment/segment*, and we have to break the string into individual segments.

- Avoid controller name confusion: specify priority namespaces in a route
    - When an incoming URL matches a route, the framework takes the value for the controller variable and looks for the appropriate name. but this is an unqualified class name, which means framework does not know how to resolve when there are more than one controller with the same name in different namespaces.
    - To prioritize the namespaces to resolve we can specify the namespaces in string array.
    - To disable searching for controller in other namespaces, take the Route object and set UseNamespaceFallback key in DataTokens collection property to false.
- The component that is responsible for finding controller is known as **controller factory**.
- Limit the URLs that a route can match: Apply a route constraint
- Constraining Routes using a Regular Expression:
    - Constraints are defined by passing them as a parameter to MapRoute method. It is passed as an anonymous object, where the properties of the type correspond to the names of the segment variable they constrain.
    - Default values are applied before the constraints are checked.
    - We can also constrain a route for specific HTTP methods by passing a value of HttpMethodConstraint() object to the annonymous object property.
- Using Type and value constraint:
    - The framework contains a number of built-in constraints that can be used to restrict URLs that a route matches based on the type and value of the segment variables.
    - We can also combine different constraints for a single segment variable by using the CompoundRouteConstraint class, which accepts an array of constraints as its constructor argument.
    - If the standard constraints are not sufficiant for your needs, we can define our own constraint by implementing IRouteConstraint interface.
- Enabling the attribute routing:
    - Call MapMvcAttributeRoutes() extension method, which is called on the RouteCollection object in RouteConfig.cs file.
    - The most important attribute is the *Route*.
    - When an action method is decorated with Route attribute, it is no longer be accessed through the convention based routing.
- Define a route within the controller: apply the Route attribute to the action methods
- Constrain an attribute route: Apply a constraint to the segment variable in the route pattern.
- Define a common prefix for all the attribute routes in a controller: apply the RoutePrefix attribute to the controller class.

Advanced Routing Features

- Generating Outgoing URLs in action methods:
    - Difference between UrlAction() and UrlRoute()
    - Difference between RedirectToAction() and RedirectToRoute()
- Provide values to the segment variables:
    - Pass an anonymous object to the ActionLink helper whose properties correspond to the segment variable names.
- Define attributes for the link element:
    - Pass an anonymous object to the ActionLink helper whose properties correspond to the attribute names.
- Url.Action() helper method:
    - Generate an outgoing URL without the link element
- Generate a URL from a specific route
    - Specify the route name when calling the helper.
- Customizing the Routing System:
    - Derive a class from RouteBase, that has implemented these two methods:
        - GetRouteData (HttpContextBase httpContext) this is the mechanism by which inbound URL matching works. The framework calls this method on each RouteTable.Routes entry in turn until one of them returns a non-null value.
        - GetVirtualPath (RequestContext requestContext, RouteValueDictionary values) this is the mechanism by which outbound URL generation works.
- IRouteHandler interface:
    - Implement this interface to create a custom mapping between URLs and action methods
- Routing requests for disk files:
    - By default, the routing system checks to see if a URL matches a disk file before evaluating the application's routes.
- IgnoreRoute method: prevents the URLs from being evaluated against the routes.
- URL schema best practices:
    - Make URLs clean and human friendly
    - Design URL to describe their content, not the implementation of the application.
    - Prefer content title over ID numbers. If you must need an ID number, then use both.
    - Do not use file extensions for HTML pages, but do use them for specialized file types (such as .jpg, .pdf, .zip)
    - Create a sense of hierarchy, like, /Products/Menswear/Shirts/Red
    - Be case-insensitive. The ASP.Net routing system is case-insensitive by default.
    - Avoid symbols, codes, and character sequences. If you want a word separator use a dash. Underscore is unfriendly. Replace the spaces in the route data to dash.
    - Do not change URLs.
    - Be consistent. Adopt one URL format across the entire application.
    - URLs should be short, easy to type, hackable (human-enditable), and persistent, and they should visualize the site structure.
    - GET and POST pick the right one.

Areas

- Break an application into sections
- Area will create a mini MVC project within the Areas folder and name of the area.
- The MapRoute method in the AreaRegistrationContext class automatically limit the routes you register to the namespace that contains for the area. This means that when we create a controller in an area, we must leave it in its default namespace; otherwise, the routing system will not be able to find it.
- If we assign name to the routes, we must ensure that they are unique across the entire application and not just the area for which they are intended.
- RouteArea:
  - We can also create an area using the RouteArea attribute above the controller. The RouteArea attribute moves all of the routes defined by the Route attribute into the specified area.
  - The RouteArea attribute doesn't affect routes that are defined by the Route attribute that starts with ~/ mean the root.
  - The RouteArea has no effect on action methods to which the Route attribute has not been defined.
  - To create a link to an action in a different area, you must create a variable called area and use it to specify the name of the area you want. Or if you want to link an action on one of the top-level controllers, then you specify the area as an empty string.

Controllers and Actions

- Create a controller:
  - Implement the IController interface or derive from the Controller class.
  - There is only one method in the IController interface and that is Execute(RequestContext requestContext)
  - RequestContext has two properties:
    - HttpContextBase HttpContext: describes the current request.
    - RouteData RouteData: describes the route that matched the request.
      - RouteBase Route: returns the RouteBase implementation that matched the request.
      - IRouteHandler RouteHandler: returns the IRouteHandler that handled the route.
      - RouteValueDictionary Values: returns a collection of segment values, indexed by name.
- Get information about the request:
  - Use the context objects and properties or define action method parameters.
  - Three main ways to that data:
    - Extract it from the set of context objects.
    - Have the data passed as the parameters to action method.

        - Explicitly invoke the framework's model binding feature.

- The properties that are available in the controller that are derived from the Controller base class (convenience properties):
    - Request
        - NameValueCollection Request.QueryString : GET variables sent with the request
        - NameValueCollection Request.Form : POST variables sent with the request
        - HttpCookieCollection Request.Cookies : cookies sent by the browser with this request.
        - string Request.HttpMethod
        - NameValueCollection Request.Headers : the full set of Http Headers sent with this request.
        - Uri Request.Url
        - string Request.UserHostAddress : the IP address of the user making this request.
    - Response
    - RouteData
        - RouteBase RouteData.Route : the chosen RouteTable.Routes entry for this request.
        - RouteValueDictionary RouteData.Values : active route parameters (either extracted from the URL or default values)
    - HttpContext
        - HttpApplicationStateBase HttpContext.Application : application state store
        - Cache HttpContext.Cache : application cache store
        - IDictionary HttpContext.Items : state store for the current request
        - HttpSessionStateBase HttpContext.Session : state store for the visitor's session
    - Server:
        - string Server.MachineName
    - IPrincipal User: authentication information about the logged-in user.
    - TempDataDictionary TempData temporary data items stored for the current user.
    - ViewBag
    - ViewData
- Sending response to the client:
    - HttpResponse context object:
    - Generate a response from controller that directly implements the IController interface.
    - Generate response from the controller derived from the Controller class
    - Use an action result:
    - The Controller class has created convenient methods that we can use to generate proper response, These methods can be used to return response by the action method:
        - ViewResult View() : renders the specified or default view template
        - PartialViewResult PartialView() : renders the specified or default partial view template
        - RedirectToRouteResult RedirectToAction(), RedirectToActionPermanent(), RedirectToRoute(), RedirectToRoutePermanent() : Issues an HTTP 301 or 302 redirection to an action method or specific route entry, generating a URL according to routing configuration.

        - RedirectResult Redirect(), RedirectPermanent() : issues an HTTP 301 or 302 redirection to a specific URL.
        - ContentResult Content() : returns raw textual data to the browser, optionally setting a content type header.
        - FileResult File() : transmits binary data (such as a file from disk or a byte array in memory) directly to the browser.
        - JsonResult Json() : serializes a .Net object in JSON format and sends it as response. More frequently used in Web API.
        - JavaScriptResult JavaScript() : Sends a snippet of JavaScript source code that should be executed by the browser.
        - HttpUnauthorizedResult : status code 403
        - HttpNotFoundResult HttpNotFound() : 404
        - HttpStatusCodeResult : returns a specific HTTP code
        - EmptyResult : done nothing

- ViewResult:
    - Tell the MVC framework to render a view.
- Pass data from the controller to View:
    - Use a ViewModel object ot the ViewBag.
- Redirection:
    - When we perform a redirect, we send one of the two HTTP codes to the browser:
        - HTTP code 302 : temporary redirection
        - HTTP code 301 : permanent redirection
    - RedirectResult Redirect() method is used to temporary redirect to a literal URL, there is also an overload that accepts bool to specify whether redirection is temporary or permanent..
    - RedirectResult RedirectPermanent() method is used to permanent redirect to a literal URL.
    - RedirectToRouteResult RedirectToRoute() : is used to redirect using the system routing schema.
    - RedirectToRouteResult RedirectToRoutePermanent()
    - RedirectToRouteResult RedirectToAction() and RedirectToRouteResult RedirectToActionPermanent() are just the wrapper around the RedirectToRoute()
- Preserving Data across the redirection (TempData):
    - If you want to pass data from one rewqest to the next, you can use the TempData feature.
    - TempData is similar to Session data, except that TempData values are marked for deletion when they are read, and removed when the request has been processed.
    - We can also read the TempData values using the Peek(), without marking it for removal.
    - We can also preserve the value that would otherwise be deleted by using the Kee() method.
- Send an HTTP result code to the browser:
    - Return an HttpStatusCodeResult object or use one of the convenience methods such as HttpNotFound.
    - HttpUnauthorizedResult

Filters

- Filters inject extra logic to the MVC framework request processing.
- Exception Filters:
  - These filters are run only if an unhandled exception has been thrown executing:
    1. Another kind of filters (authorization, action or result)
    2. The action method itself
    3. When the action result is executed.
  - IExceptionFilter:
    1. OnException (ExceptionContext filterContext)
  - ExceptionContext is derived from ControllerContext and has the following useful properties:
    1. ActionDescriptor ActionDescriptor: provides details of the action method.
    2. ActionResult Result: the result for the action method, a filter can cancel the request by setting this property to a non-null value. How a result could be there if there is any Exception.
    3. Exception Exception: the unhandled exception.
    4. Bool ExceptionHandled: returns true if another filter has marked the exception as handled.
    5. ControllerBase Controller: returns the Controller object for this request.
    6. HttpContextBase HttpContext: Provides access to the details of the request and access to the response.
    7. Bool IsChildAction
    8. RequestContext RequestContext: Provides access to the HttpContext and the routing data, both of which are available through other properties
    9. RouteData RouteData: Returns the routing data for this request
  - HandleErrorAttribute:
    1. The HandleErrorAttribute filter works only when custom errors are enabled in web.config, which is done adding a CustomErrors attribute inside <system.web> node.
    2. Type ExceptionTypeThe exception type handled by this filter. It will also handle exception types that inherit from the specified value, but will ignore all others. The default value is System.Exception, which means that, by default, it will handle all standard exceptions.
    3. String View: The name of the view template that this filter renders.
    4. String Master: The name of the layout used when rendering this filter's view. If you do not specify a value, the view uses its default layout page.
    5. When rendering a View, the HandleErrorAttribute filter passes a HandleErrorInfo view model object, which is a wrapper around the exception that provides additional information that can be used in View.
    6. Useful properties of the HandleErrorInfo class:
       - String ActionName

        - String ControllerName
        - Exception Exception
- Action Filters:
    - Action filters are the filters that can be used for any purpose.
    - The interface for creating this kind of filters is IActionFilter, it has two methods:
        1. OnActionExecuting(ActionExcutingContext filterContext): before the action method is executed
        2. OnActionExecuted(ActionExecutedContext filterContext)
        3. Useful Properties of the ActionExecutedContext:
            - ActionDescriptor ActionDescriptor
            - Bool Canceled: returns true if the action has been canceled by another filter
            - Exception Exception
            - Bool ExceptionHandled
            - Result ActionResult
- Result filters:
    - Result filters are the general-purpose filters which operate on the result produced by the action methods.
    - The interface for creating this kind of filters is IResultFilter, it has two methods:
        1. OnResultExecuting(ResultExecutingContext filterContext)
        2. OnResultExecuted(ResultExecutedContext filterContext)
    - The parameters has the same properties as the action filter counterparts.
- ActionFilterAttribute
    - The built-in filter class that implements both IActionFilter and IResultFilter interfaces.
- Filtering without attributes:
    - The Controller class implements all these Filter interfaces and also provides the default empty implementation of all the methods.
    - This technique is useful when you are trying to create a base class which multiple controllers in project are derived.
- Global filters
- Ordering filter execution:
    - The default sequence is:
        1. Authentication filters
        2. Authorization filters
        3. Action filters
        4. Result filters
        5. Exception filters are executed whenever any unhandled exception is thrown.
    - However within each type, we can control the order in which the individual filters are executed.
    - Global filters are executed first, then Controller filters and then at last action method filters
- Filter overrides
    - Implement the IOverrideFilter

- The MVC framework has also provided some built-in override filters like OverrideAuthenticationAttribute, OverrideActionFilterAttribute and so on.

Controller Extensibility

- The process:
- Request → Routing → Controller Factory → Controller → Action Invoker → Action Method → Response
- The **controller factory** is responsible for creating instances of controllers to service a request.
- The **action invoker** is responsible for finding and invoking the action method in the controller class.
- IControllerFactory:
    - Implement this interface to create a custom implementation of the controller factory.
    - interface IControllerFactory:
        - IController CreateController (RequestContext requestContext, string controllerName)
        - SessionStateBehavior GetControllerSessionBehavior (RequestContext requestContext, string controllerName) : to determine if the session data should be maintained for the controller.
            - SessionStateBehavior.Default
            - SessionStateBehavior.Required
            - SessionStateBehavior.ReadOnly
            - SessionStateBehavior.Disabled
        - void ReleaseController (IController controller)
- Built-in implementation of the IControllerFactory is DefaultControllerFactory:
    - When DefaultControllerFactory recieves a request from the routing system, this factory looks at the routing data to find the value of the controller property and tries to find a class in the web application that meets the following criteria:
        - The class must be public.
        - The class must be concrete.
        - The class must not take generic parameters.
        - The name of the class must end with controller.
        - The class must implement the IController interface.
    - The DefaultControllerFactory maintains a list of such classes, so that it does not have to perform this search every time.
    - If a suitable class is found then an instance is created otherwise the request cannot be proccessed further.
- DefaultNamespaces collection:
    - Prioritize namespaces in the default controller factory.
    - To set the global namespace priority for searching controller class is sepecified using ControllerBuilder.Current.DefaultNamespaces.Add() method.
- IControllerActivator interface:
    - We can introduce DI into controllers by creating a controller activator.

  - IController Create (RequestContext requetContext, Type controllerType)
- IActionInvoker:
  - Implement this interface to create a custom implementation for the action invoking mechanism.
  - Bool InvokeAction (ControllerContext controllerContext, string actionName)
- Built-in implementation of the IActionInvoker is ControllerActionInvoker.
- Specify the action name that is different from the action method using **ActionName** attribute.
- **NonAction** attribute Prevent a method from being used as an action.
- Action method selectors are derived from the ActionMethodSelectorAttribute class.
  - Abstract bool IsValidForRequet (ControllerContext controllerContext, MethodInfo methodInfo)
- Session less controller:
  - By defining the method GetControllerSessionBehavior() in custom IControllerFactory implementation.
  - Setting the SessionState attribute for the controller that uses the DefaultControllerFactory

Views

- Create a custom view engine:
  - View engines implement the IViewEngine interface:
    - ViewEngineResult FindPartialView (ControllerContext controllerContext, string partialViewName, bool useCache)
    - ViewEngineResult FindView (ControllerContext controllerContext, string viewName, string masterName, bool useCache)
    - Void ReleaseView (COntrollerContext, IView view)
  - The role of view engine is to translate requests for view into ViewEngineResult objects.
  - An IView implementation is passed to the constructor of a ViewEngineResult object, which is then returned from the view engine methods.
  - IView interface:
    - Void Render (ViewContext viewContext, TextWriter writer)
      - ViewContext.Controller
      - ViewContext.RequetContext
      - ViewContext.RouteData
      - ViewContext.TempData
      - ViewContext.View
      - ViewContext.ViewBag
      - ViewDataDictionary ViewContext.ViewData
        - Keys
        - Model
        - ModelMetaData
        - ModelState
  - The MVC framework then calls the Render method.

- The views are translated into C# classes, and then compiled.
- The rendered class is derived from the WebViewPage<T> where T is model type.
- We can change the view files that razor searches for by creating a subclass of RazorViewEngine and adding the location in one of the following lists:
    - ViewLocationFormats
    - MasterLocationFormats
    - PartialViewLocationFormats

    - AreaViewLocationFormats
    - AreaMasterLocationFormats
    - AreaParticalViewLocationFormats
- Adding dynamic contents to the View:
    - Inline Code: if or foreach like statements.
    - HTML Helper methods
    - Sections
        - Use for creating sections of content that will be inserted into layout at specific locations.
        - Sections are defined using the @section tag followed by the name of the section. In View files.
        - Sections are defined in the View, but applied in a Layout with @RenderSection helper method.
    - Partial Views
        - Use for sharing subsections of view markup between Views. Partial views do not invoke an action method, so they cannot be used to perform business logic.
        - A partial view is consumed by calling the Html.Partial() helper method from views.
        - We can also create strongly typed partial views.
    - Child Actions
        - Use for creating reusable UI controls or widgets that need to contain business logic.
        - Child actions are the actions invoked within the view.
        - One can use child actions whenever there is need of displaying some data driven widgets that appear on multiple pages and contains data unrelated to the main action that is running.
        - ChildActionOnly attribute prevents the action methods from being invoked as a result of user request.
        - Child actions are typically associated with partial views, although this is not compulsory
        - Child actions are invoked using the Html.Action() helper method.

Helper Methods

- Helper methods allows up to package chunks of code markup so that they can be reused throughout an MVC framework application.
- Inline helpers:
  - Create a region of reusable markup within a view.
  - We can create an inline helper method using the @helper keyword followed by the regular method definition but without return type and return statement.
- External helper:
  - Create markups that can be used within multiple views.
  - External helper methods are expressed as a C# extension method.
  - The HtmlHelper provides access to information that can be useful when creating content:
    - RouteCollection
    - ViewBag
    - ViewContext
      - Controller
      - HttpContext
      - IsChildAction
      - RouteData
      - View
  - TagBilder helps to create HTML elements within the helper method:
    - The TagBuilder class is part of the System.Web.WebPages.Mvc assembly, but uses a feature called ***type forwarding*** to appear as though it is part of System.Web.Mvc assembly.
    - String InnerHtml: set the content of the element as an HTML string.
    - SetInnerText(string) : sets the text content of the HTML element. The string patameter is encoded to make it sage to display.
    - AddCssClass(string)
    - MergeAttribute(string key, string value, bool replaceExisting)
- Helper methods should be used to reduce the amount of duplication in views, and to create small HTML markups. For more complex markup and content, partial views should be used and if there is any need to manipulate the model data, then child actions should be used.
- Html.BeginForm() and Html.EndForm()
- Html.BeginRouteForm():
  - Generates a form element using a specific route
  - If we want to ensure that a particular route is used, then we can use the BeginRouteForm() helper method.
- Generate input elements
  - We can generate input elements through model property also. In which we have to pass only one string argument that specifies the name of the property. The MVC framework tries to find the item of data that corresponds with the key, and the following locations are checked:
    - ViewBag

        - Model
    - We can also use strongly typed helper methods, which takes the lambda expression as the first argument and in that the Model will be passed as the argument.
- Generate select elements

Templated Helper methods

- With templated helper methods, we specify the property we want to display and let the MVC framework figure out what HTML elements are required.
- Html.Editor() and Html.EditorFor()
- Html.Label() and Html.LabelFor()
- Html.Display() and Html.DisplayFor()
- Generate elements for a complete model object:
    - DisplayForModel()
    - EditorForModel(}
    - LabelFormModel()
- HiddenInput attribute:
    - Hide an element from the user when generating elements using a whole-model helper or prevent it from being edited.
    - To completely exclude a property from the generated HTML, we can use the ScaffoldColumn attribute.
- DisplayName and Display attribute:
    - Set the name that will be used to display model properties.
- DataType attribute:
    - Specify the way in which model properties are displayed.
    - DateTime
    - Date
    - Time
    - Text
    - PhoneNumber
    - MultilineText
    - Password
    - Url
    - EmailAddress
- UIHint attribute:
    - Specify the template used to display a model property.
    - Boolean: Renders a checkbox for bool values. For nullable bool? Values, a select element is created with options for True, False and Not Set.
    - Collection: renders the appropriate template for each of the elements in an IEnumerable sequence. The items in the sequence do Not have to be of the same type.
    - Decimal: renders a single line textbox input element and formats the value to display two decimal places.
    - DateTime

  - Date
  - EmailAddress
  - HiddenInput
  - Html: renders a link using an HTML a element.
  - MultilineText
  - Number
  - Object: it is the template used by the scaffolding helpers to generate HTML for a view model object. This template examines each of the properties of an object and select the most suitable template for the property type.
  - Password
  - String
  - Text
  - Tel
  - Time
  - Url
- Define model metadata separately from the model type:
  - Create a buddy class and use the MetadataType attribute.
  - To use this feature define the model class as a partial class and create a second partial class that contains the metadata.
  - Use the MetadataType attribute on the model class to specify the metadata buddy class.
  - The buddy class must be in the same namespace as the Model class.
- Create custom template:
  - Change the elements that are generated for a model property.
  - The MVC framework looks for the custom editor template in the /Views/Shared/EditorTemplates folder.

URL and Ajax helper methods

- Generate links and URLs:
  - Url.Content(): used to create the application relative URL
  - Url.Action(): only the URL for the action
  - Url.RouteUrl(): URL using the route data and named routes
  - Html.ActionLink(): link to the named action/controller, will generate anchor tag
  - Html.RouteLink(): link using the route data and named routes
- Ajax.BeginForm()
  - The MVC framework relies on the Microsoft Unobtrusive Ajax package to make and process Ajax request, and for that install these packages:
    - Install-Package jQuery -version 1.10.2
    - Install-Package Microsoft.jQuery.Unobtrusive.Ajax -version 3.0.0
  - In Web.config the configuration/appSettings element contains an entry for the **UnobtrusiveJavaScriptEnabled** property, which must be set to true.
  - At the heart of the MVC framework support for Ajax forms is the Ajax.BeginForm() helper method, which takes an AjaxOptions object as its argument.

    - The AjaxOptions class which is in the namespace System.Web.Mvc.Ajax namespace, defines properties that let me configure how the synchronous request to the server is made and what happens to the data that comes back.
        - String Confirm: sets a message to be displayed to the use in a confirmation window before making the Ajax request.
        - HttpMethod HttpMethod
        - InsertionMode InsertionMode:
            - InsertAfter
            - InsertBefore
            - Replace (default)
        - LoadingElementId
        - LoadingElementDuration
        - UpdateTargetId
        - Url: sets the URL that will be requested from the server. Use this property instead of specifying the route to the Ajax.BeginForm() method to restrict the wrong redirection for the JavaScript disabled users.
    -
- Ensure that non-JavaScript browsers do not display HTML fragments
    - Set the Url Ajax option
- Provide the user with feedback during an Ajax request:
    - LoadingElementId
    - LoadingElementDuration
- Prompt a user before making an Ajax request:
    - Use the Confirm ajax option
- Ajax.ActionLink()
    - Create an ajax enabled link
- Ajax callback options:
    - Receive notifications about the progress and outcome of Ajax requests.
    - OnBegin → jQuery event beforeSend: called immediately before the request being sent
    - OnComplete → jQuery event complete: called if the request is successful.
    - OnFailure → jQuery event error: called if the request fails
    - OnSuccess → jQuery success: called when the request has completed, irrespective of whether the request succeeded or failed.
- JsonResult
- Request.IsAjaxRequest:
    - Detect ajax request in the controller

Model Binding

- Model Binding is the process of create .NET objects using data sent by the browser in an HTTP request.
- Bind to a simple type or a collection:

- Add a parameter to an action method.
- DefaultModelBinder is a buit-in model binder class. By default, this model binder searched four locations for data matching the name of the parameter being bound:
  - Request.Form
  - RouteData.Values
  - Request.QueryString
  - Request.Files
- The DefaultModelBinder uses the culture-specific settings to perform type conversions from different areas of the request data. The values that are obtained from the URLs (the routing and query string data) are converted using culture-insensitive parsing, but values obtained from form data are converted taking culture into account.
- To bind coming values to the *collection of the model object*, it must be in the form *[index].<PropertyName>*

* Provide a fallback value for model binding:
  - Use a nullable type for the action method parameter or use default value.
* Bind to a complex type:
  - Ensure that the HTML generated by the view contains nested property values.
* Override the default approach to locatiing nested complex types:
  - Check for the HomeAddress apropety generated HTML by the EditorForModel().
  - Use the prefix property Bind attribute applied to the action method parameter.
* Selectively binding properties:
  - Use the Include or Exclude properties of the Bind attribute, applied either to action method parameter or to the model class.
  - When the Bind attribute is applied to both model class and the action method parameter, then a property will be included in the bind process only if neither of the attribute excludes it. This means that the policy applied to the model class cannot be overridden by applying a less restrictive policy to the action method parameter.
* Manually invoke model binding:
  - UpdateModel() or TryUpdateModel()
  - By using one these methods we can restrict the model binding for specific model objects and we can also restrict the source also in which the data will be searched. It must be given in the form of IValueProvider implementation.
  - The built-in IValueProvider impelementations:
    - FormValueProvider Requet.Form
    - RouteDataValueProvider RouteData.Values
    - QueryStringValueProvider Request.QueryString
    - HttpFileCollectionValueProvider Request.Files
  - Each of these classes takes a ControllerContext constructor parameter.
  - There are some overloaded versions of these methods that specify a prefix to seach for and which model properties should be included in the binding process.
* Creating a custom value provider:
  - By defining the custom value provider, we can add own source of data to the model binding process.
  - Value provider implements the IValueProvider interface.

        - Bool ContainsPrefix(string prefix): to determine if the value provider can resolve the data for a given prefix.
        - ValueProviderResult GetValue (string key): returns a value for the given data key, or null if the provider doesn't have any suitable data.
    - To register a value provider with the application, we need to create a factory class that will create instances of the provider when they are required by the MVC framework. The Factory class must be derived from the ValueProviderFactory:
        - IValueProvider GetValueProvider(ControllerContext controllerContext)
- Custom model binder:
    - IModelBinder interface:
        - Object BindModel(COntrollerContext controllerContext, ModelBindingContext bindingContext): MVC framework will call the this method when it wants an instance of the model type that the binder supports.
            - ControllerContext
            - ModelBindingContext: provides details of the model object that is sought.
                - Model: returns the model object passed to the UpdateModel() method if binding has been invoked manually.
                - ModelName: returns the name of the model being bound or If there is a prefix need to append to the model object property.
                - ModelType
                - ValueProvider
    - Register the custom model binder through the ModelBinders.Binders.Add() or we can also use the ModelBinder attribute.

Model Validation

- Model validation is the process of ensuring that data received by the application is suitable for binding to the model and, when this is not the case, providing useful information to the user that will help explain the problem.
- Explicitly validate a model:
    - Use the ModelState object to record validation errors.
- DefaultModelBidnder:
    - The built-in default model binder class, provides some useful methods that can be overridden to add validation to a binder.
        - OnModelUpdated(): called when the binder has tried to assign values to all of the properties in the model object. Applies the validation rules defined by the model metadata and registers any errors with ModelState.
        - SetProperty(): called when the binder wants to apply a value to a specific property. If the propety cannot hold null value and there was no value to apply, or there is a value that cannot be parsed, then it will register error with ModelState.
- Validation rules within the model class:

- Apply attributes to the properties of the model class.
- The built-in validation attributes:
    - Compare: two properties must have the same value.
    - Range: a numeric value (or any property type that implements IComparable) must not lie beyond the specified minimum and maximum values.
    - RegularExpression: make it case insensitive use (?i) modifier.
    - Required: the value that must not be empty or be a string containing only white spaces. If you want to treat whitespace as valid, use [Required(AllowEmptyStrings = true)]
    - StringLength: a string value must not be longer that the specified maximum length. You can also specify minimum length.
- The DataType attribute cannot be used to validate user input, only to provide hints for rendering values using templated helpers. So, do not expect the DataType.EmialAddress attribute to enforce a specific format.

- Custom validation attribute:
    - Derive a class from ValidationAttribute class.
        - bool IsValid(object value)
    - We can also extend the attributes that are provided to use and use the base.IsValid() method with value to check the base validations.
    - We can also create attributes that validate the complete model instead of the property. Model level attributes will not be used when a property level problem is detected.
- Define a self-validating model:
    - Implement IValidatableObject interface:
        - IEnumerable<ValidationResult> Validate(ValidationContext validationContext)
    - One benefit of this approach is that the model- and property-level validation is combined in one place, which means all of the errors are displayed togather.
    - IValidationObject will have no effect on the client-side library.
- Client-side validation:
    - Add Microsoft unobtrusive validation package to the application.
    - The term unobtrusive means that validation rules are expressed using attributes added to the HTML elements that views generate.
    - Most MVC applications use client-side validation for property-level issues and rely on server-side validation for the overall model validation.
    - Client side validation is controlled by two settings in Web.config file:
        - ClientValidationEnabled = true
        - UnobtrusiveJavaScriptEnabled = true
    - To set the client-side validation per view base by setting:
        - HtmlHelper.ClientvalidationEnabled
        - HtmlHelper.UnobtrusiveJavaScriptEnabled
    - Install the following packages to get the required JS libraries:
        - Install-Package jQuery -version 1.10.2
        - Install-Package jQuery.Validation -version 1.11.1
        - Install-Package Microsoft.jQuery.Unobtrusive.Validation -version 3.0.0
- **EpressiveAnnotations** library for annotation based conditional validation.

- Perform Remote validation:
    - Define an action method that returns a JsonResult and add the Remote attribute to the model property to be validated.
- Classes used for while validation fails:
    -

Actions

- Actions basically return different type of action results:
    - ContentResult: returns a string
    - FileContentResult, FilePathResult , FileStreamResult: returns file content
    - EmptyResult: returns nothing
    - JavaScriptResult: returns script for execution
    - JsonResult: returns JSON formatted data
    - RedirectToResult: redirects to specified URL
    - HttpUnauthorizedResult: returns HTTP 403 start code
    - RedirectToRouteResult: redirects to different action/controller action
    - ViewResult, PartialViewResult: returns the response from the view engine

Difference between ViewResult and PartialViewResult

ViewResult and ActionResult

```
ActionResult

        |- ViewResultBase

        |        |- ViewResult (HTML)

        |- ContentResult (scalar text response)
```

Passing data (Model) from Controller to View:

- ViewData is a dictionary, which will contain data to be passed between controller and views. Controller will add items to this dictionary and view reads from it.
- ViewBag is just a syntactic sugar for ViewData. ViewBag uses the dynamic feature of C# 4.0 and makes the ViewData more dynamic.
- Problems with using ViewData or ViewBag:
    - Performance issues: cast from object to proper type
    - No type safety and compile time errors
    - No proper connection between data sent and data received.
- Strongly types views.
- ViewModel fits between the Model and the View and acts as data container for View. We can also include Lists as the property of the ViewModel.

Data Access Layer:

- Entity Framework is an ORM (Object Relational Mapping) tool.
- In Entity Framework, we can follow one of the three approaches:
    - Database first approach
    - Model first approach
    - Code first approach
- **DbSet:** DbSet simply represents the collection of all the entities that can be queried from the database. When we write a **LINQ query** against the DbSet object, it internally converted to query and fired against the database.
- Model binder

Model Binder

- To explicitly call the model binder we can use UpdateModel() or TryUpdateModel() method.

ViewModel

DataAccessLayer

Data attributes for server side validation:

- Required
- StringLength: max length
- DataType: make sure that the data is of particular type like email, url, credit-card number, etc.
- EnumDataTypeAttribute
- Range
- RegularExpression

Creating custom validation

- To enable client side validation, we can use the JavaScript unobtrusive validation package and reference them in our view.
- Install-Package Microsoft.jQuery.Unobtrusive.Validation – version 3.0.0 -projectname <name-of-the-project>
- We can enable or disable client side validation by using HtmlHelper.ClientvalidationEnabled and HtmlHelper.UnobtrusiveJavaScriptEnabled property on view or setting it in Web.config

Authentication in ASP.Net (froms authentication)

Styling the HTML that are generated through ASP.NET MVC helper elements

Microsoft jQuery Unobtrusive validation for client side auto validation

Partial Views

Layouts just like master pages in ASP.Net web forms

Role based security using the ActionFilter

HTML Helper methods:

- Html.ActionLink(linkText, actionName)
- Html.BeginForm()
- Html.TextBoxFor()
- Html.DropDownListFor()
- Html.ValidationSummary()
- Html.ValidationMessage()
- Html.ValidationMessageFor()

C# essentials

String Interpolation:

```
String value = $“Name = {Name}, Price = {Price}”;
```

Automatically implemented properties:

- Simplify C# properties
- We don’t need to create the private field every time to create a property, if there is no any special function going on inside the getter or setter.
- The backed field will be automatically created.

Object or collection initializer:

- Create an object and sets its properties in a single step

```
Product myProduct = new Product {
    ProductID = 100, Name = “Kayak”, Description = “A boat for one person”, Price = 202M
}
```

Extension method:

- Add functionality to a class which cannot be modified
- We can also add functionality to interfaces using this.

Lambda expressions:

- Simplify the use of delegates
- We can simplify the delegate using this lambda expression feature.

Implicit typing:

- Var keyword
- By combining object initializer and type inference, we can create simple data-storage objects without needing to define the corresponding class or struct.
- The C# compiler generates a class based on the name and the type of the parameter in the initializer. Two anonymously types objects that have the same property and name and types will be assigned to the same automatically generated class. So by this feature, we can create arrays of anonymously typed objects.

Anonymous type:

- Create objects without defining type

LINQ:

- Query collections of objects as though there were a database
- A query that contained all the deferred methods is not executed until the results are enumerated. Means queries are evaluated from the scratch every time the results are enumerated, so we can perform the query repeatedly as the source data for the changes and get results that reflect the current state of the source data.

Simplify the use of asynchronous methods:

- Use the async and wait keyword

Razor

- Razor views are compiled into C# classes.

Define and access the model type

- Use the @model and @Model expressions

Using Layout

- Reduce duplication in views
- Files in the View folder whose name begins with an _ (underscore) are not returned to the user, which allows file name to differentiate between views that we want to render and the views that support them. Layouts, which are support files are prefixed with an underscore.
- The call to the @RenderBody() method inserts the content of the view specified by the action method into the layout markup.

View start view

- Specify the default layout
- When the engine renders a View, it will look for a file called _ViewStart.cshtml. The content of the file will be treated as though they were contained in the view file itself and we can use this feature to automatically set the Layout property.

Data passing from controller to view

- Pass a view model object or the view bag

Razor conditional statements

- Generate different content based on the data values

Enumerate an array or collection

- Use @foreach expression

Add namespace to the view

- Use a @using expression

Essential Tools

- Dependency Injection (DI) or Inversion of Control (IoC) container: Ninject, Unity from Microsoft
- A unit test framework: NUnit
- Mocking tool: Moq, Rhino Mocks

Decouple classes:

- Introduce interfaces a declare dependencies on them in the class constructor
- The idea of dependency injection is to decouple the components in an MVC application, with a combination of interfaces and DI container that creates instances of objects by creating implementations of the interfaces they depend on and injecting them into the constructor.

Automatically resolve dependencies expressed using Interfaces:

- Use Ninject or another dependency injection framework.

Integrate Ninject into an MVC application:

- Create an implementation of the IDependencyResolver interface that calls the Ninject kernel and register it as a resolver by calling the System.Web.Mvc.DependencyResolver.SetResolver() method.
- Commands to install Ninject:
  - Install-Package Ninject -version 3.0.1.10
  - Install-Package Ninject.Web.Common -version 3.0.0.7

  - Install-Package Ninject.MVC3 -version 3.0.0.6
  - The first command installs the Ninject core package and the others install the extension to the core that makes Ninject work nicely with ASP.NET application.
- There are three stages to get the basic functionality of the Ninject working:
  - Create an instance of the Ninject Kernel, which is the object that is responsible for resolving dependencies and creating new objects. When we need an object we will use Ninject kernel instead of using new keyword.
  - Configure the Ninject so that it understands which implementation objects I want to use for each interface I am working with.
  - Use Ninject Kernel to create an object.

Inject property and constructor values into newly created objects:

- Use the WithPropertyValue() and WithConstructorArgument() method.

Dynamically select an implementation class for an interface

- Use Ninject conditional binding.

Object scope:

- Control the lifecycle of the objects that Ninject creates.
- Available scope related extension methods are:
  - InTransientScope()
  - InSingletoneScope(), ToConstant(object)
  - InThreadScope()
  - InRequestScope()

Create a Unit test:

- Add a unit test project to the solution and annotate a class file with TestClass and TestMethod attributes.

Check for the expected outcomes in a unit test:

- Use the Assert class

Focus a unit test on a single feature or component:

- Isolate the test target using the mock objects.

Steps to create ASP.Net MVC application:

- Create a blank solution
- Three projects within that:
  - Domain (simple class library)
  - WebUI (ASP.Net MVC)
  - UnitTests (If you want the facility of unit testing)

- Install tool packages:
  - Ninject in WebUI, UnitTests
  - Ninject.Web.Common in WebUI, UnitTests
  - Ninject.MVC3 in WebUI, UnitTests
  - Moq in Web, UnitTests
  - Microsoft.Aspnet.Mvc in Domain, UnitTests
- Add references between the projects
  - Domain -> System.ComponentModel.DataAnnotations
  - WebUI -> Domain
  - UnitTests -> Domain, WebUI, System.Web, Microsoft.CSharp
- Set up DI container:
  - In WebUI project create Infrastructure > NinjectDependencyResolver.cs and implement IDependencyResolver and AddBinding().
  - Create a bridge between NinjectDependencyResolver and MVC by registering it into App_Start > NinjectWebCommon.cs
- Model repository and domain model:
  - All MVC projects starts with domain model because everything in an MVC framework resolves around it.
  - Create Entities folder in Domain project and create all the classes in that.
  - Create Abstract folder in Domain project and create interfaces there.
- Connecting to Database;
  - Install Entity Framework in Domain and WebUI
  - Create Concrete folder and create EFDbContext class that derives System.Data.Entity.DbContext class and create the properties for the all the table names, that will return DbSet and the type parameter of the DbSet result specifies the model type.
  - Define the connection string in Web.config
  - Create Concrete class for the repository in the Concrete folder and add the binding of that concrete class to the AddBindins of NinjectDependencyResolver.
- Add Controllers and action methods within that
  - Controllers should declare the dependencies on the repository interfaces (mostly through constructor injection).
- Add Models and ViewModels in WebUI if required
- Add Views, Layouts, and Partial Layouts as per the requirement.
- Add HtmlHelpers as per the requirement.
- Set the routes in App_Start > RouteConfig.cs

Model binder:

- Model binder is used to map the action method parameters with the data coming from the request.
- We can also create our custom model binder by implementing the System.Web.Mvc.IModelBinder interface.

Filters

- Sometimes, we want to perform some logic either before or after the action method runs. To support this, AP.Net MVC provides Filters.
- Filters are custom classes that provide both declarative and programmatic means to add pre-action and post-action behavior to controller action methods.
- An action filter is an attribute that you can apply to a controller action or an entire controller that modifies the way in which the action is executed.
- The ASP.Net MVC includes several action filters:
    - OutputCache
    - HandleError
    - Authorize
- Framework provides four different types filters (in order):
    1. Authorization Filters: implements IAuthorizationFilter attribute
    2. Action Filters: implements IActionFilter attribute
    3. Result Filters: implements IResultFilter attribute
    4. Exception Filters: implements IExceptionFilter attribute
- To create a custom filter, framework provides a base class which is known as ActionFilterAttribute. This class implements both IActionFilter and IResultFilter interfaces.

Selectors

- Selectors help routing engine to select the correct action method to handle a particular request.
- There are three action selectors:
    - ActionName
    - NonAction
    - Action Verbs
        - HttpGet
        - HttpPost
        - HttpPut
        - HttpDelete
        - HttpOption
        - HttpPatch

HtmlHelper

- ASP.Net MVC provides HtmlHelper class which contains different methods that help in creating HTML controls programmatically.
- All Html helper methods generate HTML and return the result as a string.
- There are different types of helper methods:
    - Create inputs

- GenerateIdFromName (string name): generates an HTML element Id using the specified element name.
- GenerateIdFromName (string name, string idAttributeDotReplacement): generates an HTML element Id using the specified element name and a string that replaces dots in the name.
- Checkbox (string name)
- Checkbox (string name, bool isChecked)
- Checkbox (string name, bool isChecked, IDictionary<string, object> htmlAttributes)
- Checkbox (string name, bool isChecked, object htmlAttributes)
- Checkbox (string name, IDIctionary<string, object> htmlAttributes)
- Checkbox (string name, object htmlAttributes)

- Display (string expression)
- Display (string expression, object addiionalViewData)
- Display (string expression, string templateName)
- Display (string expression, string templateName, object addiionalViewData)
- Display (string expression, string templateName, string htmlFieldName)
- Display (string expression, string templateName, string htmlFieldName, object addiionalViewData)

- DisplayForModel() and overloads
- DisplayName (string expression)
- DisplayNameForModel ()
- DisplayText(string name)

- DropDownList (string name)
- DropDownList (string name, IEnumerable<SelectListItem> selectList)
- DropDownList (string name, IEnumerable<SelectListItem> slelectList, IDictionary htmlAttributes)
- DropDownList (string name, IEnumerable<SelectListItem> slelectList, object htmlAttributes)
- DropDownList (string name, IEnumerable<SelectListItem> selectList, string optionLabel)
- DropDownList (string name, IEnumerable<SelectListItem> selectList, string optionLabel, IDictionary<string, object> htmlAttributes)
- DropDownList (string name, IEnumerable<SelectListItem> selectList, string optionLabel, object htmlAttributes)
- DropDownList (string name, string optionLabel)

- Editor (string expression) and overloads
- EditorForModel() and overloads
- Hidden() and overloads: returns the hidden input element
- Id()

  - IdForModel()
  - Label() and overloads
  - LabelForModel() and overloads
  - ListBox() and overloads
  - Name(): gets the full HTML filed name for the object that is represented by the expression.
  - NameForModel()
  - Partial(string viewName) and overloads: renders the specified partial view as an HTML-encoded string.
  - Password() and overloads
  - RadioButton() and overloads
  - RenderAction() and overloads
  - RenderPartial()
  - RouteLink()
  - TextArea() and overloads
  - TextBox() and overloads
  - Validate(string modelName): retrieves the validation metadata for the specified model and applies each rule to the data field, client validation must be enabled.
  - ValidationMessage(string modelName): displays the validation error message if there is any error.
  - ValidationSummary()
  - Value() and overloads
  - ValueForModel() and overloads
- Create links
  - Action (String actionName): invokes the specified child action method and returns the result as an HTML string.
  - Action (string actionName, object routeValues)
  - Action (string actionName, RouteValueDictionary routeValues)
  - Action (string actionName, string controllerName)
  - Action (string actionName, string controllerName, object routeValues)
  - Action (string actionName, string controllerName, RouteValueDictionary routeValues)

  - ActionLink (string linkText, string actionName): creates a hyperlink on View page.
  - ActionLink (string linkText, string actionName, object routeValues)
  - ActionLink (string linkText, string actionName, object routeValues, object htmlAttributes) try to escape the routeValues using named parameters
  - ActionLink (string linkText, string actionName, RouteValueDictionary routeValues)
  - ActionLink (string linkText, string actionName, RouteValueDictionary routeValues, IDictionary<string, object> htmlAttributes)
  - ActionLink (string linkText, string actionName, string controllerName)

        - ActionLink (string linkText, string actionName, string controllerName, object routeValues, object htmlAttributes)
        - ActionLink (string linkText, string actionName, string controllerName, RouteValueDictionary routeValues, IDictionary<string, object> htmlAttributes)
        - ActionLink (string linkText, string actionName, string controllerName, string protocol, string hostname, string fragment, object routeValues, object htmlAttributes)
        - ActionLink (string linkText, string actionName, string controllerName, string protocol, string hostname, string fragment, IDictionary<string, object> htmlAttributes)

- Create forms
    - BeginForm(): writes an opening form tag to the response. The form uses the POST method and the request is processed by the action method for the view.
    - BeginForm (object routeValues)
    - BeginForm (RouteValueDictionary routeValues)
    - BeginForm (string actionName, string controllerName)
    - BeginForm (string actionName, string controllerName, FormMethod method)
    - BeginForm (string actionName, string controllerName, FormMethod method, IDctionary<string, object> htmlAttributes)
    - BeginForm (string actionName, string controllerName, FormMethod method, object htmlAttributes)
    - BeginForm (string actionName, string controllerName, object routeValues)
    - BeginForm (string actionName, string controllerName, object routeValues, FormMethod method)
    - BeginForm (string actionName, string controllerName, object routeValues, FormMethod method, object htmlAttributes)
    - BeginForm (string actionName, string controllerName, RouteValueDictionary routeValues)
    - BeginForm (string actionName, string controllerName, RouteValueDictionary routeValues, FormMethod method)
    - BeginForm (string actionName, string controllerName, RouteValueDictionary routeValues, FormMethod method, IDictionary<string, object> htmlAttributes)

    - BeginRouteForm (object routeValues)
    - BeginRouteForm (RouteValueDictionary routeValues)
    - BeginRouteForm (string routeName)
    - BeginRouetForm (string routeName, FormMethod method)
    - BeginRouteForm (string routeName, FormMethod method, IDictionary<string, object> htmlAttributes)
    - BeginRouteForm (string routeName, FormMethod method, object htmlAttributes)
    - BeginRouteForm (string routeName, object roureValues)
    - BeginRouteForm (string routeName, object roureValues, FormMethod method)

            - BeginRouteForm (string routeName, object roureValues, FormMethod method, object htmlAttributes)
            - BeginRouteForm (string routeName, RouteValueDictionary routeValues)
            - BeginRouteForm (string routeName, RouteValueDictionary routeValues, FormMethod method)
            - BeginRouteForm (string routeName, RouteValueDictionary routeValues, FormMethod method, IDictionary<string, object> htmlAttributes)

            - EndForm()
    - Others
        - AnonymousObjectToHtmlAttributes(object): replaces underscore characters (_) with the hyphens (-) in specified HTML attributes.
        - AntiForgeryToken(): generates a hidden filed that is validated when the form is submitted.

Web API

- **SPA (Single Page Applications):**
    - The most consistently used definition of SPA (Single Page Applications), is a web application whose initial content is delivered as a combination of HTML and JavaScript and whose subsequent operations are performed using a RESTful web service that delivers data via JSON in response to Ajax requests.
    - The advantages of a SPA are that less bandwidth is required and that the user receives a smoother experience.
    - The disadvantages are that the smoother experience can be hard to achieve and that the complexity of the JavaScript code is required for an SPA demands careful design and testing.
- RTA (Round-Trip Applications)
- The most applications mix and match the SPA and RTA techniques, where each major functional area of the application is delivered as an SPA, and navigation between functional areas is managed using standard HTTP requests that create a new HTML document.

- **Create a RESTful web service:**
    - Add a Web API controller to an MVC Framework application
    - The Web API is based on adding a special kind of controller to an MVC Framework application. This kind of controller called API Controller, has two distinctive characteristics:
        1. Action methods return model, rather than ActionResult objects.
        2. Action methods are selected based on the HTTP methods used in the request.

    - The model objects that are returned from an API controller action method are encoded as JSON and sent to the client.
    - API controllers are designed to deliver web data services, so they do not support views, layouts, or any of the other features that is used to generate HTML.
    - Web API controller is a derivation of the ApiController
    - The return type of the API controller action methods may differ because it uses the HTTP Accept header contained in the request wo work out what data type the client would prefer to work with. It gives preference to JSON.
- Map between HTTP methods and action names in a Web API controller:
    - Apply attributes such as HttpPut and HttpPost to the methods.
- Create a Single Page Application:
    - Use Knowkout and jQuery to obtain data view Ajax and bind it to HTML elements.
- Using Web API:

Questions and to learn:

- What to do if we don't want to add the Controller key word in the Controller creation. Custom name without controller word.
- Dynamic object to which we can assign arbitrary properties
- Razor
- Lambda expressions
- Difference between MvcHTMLString and simple string
- Difference between assembly and namespace
- Decimal data type
- How extension methods are discovered throughout the Namespace.
- How Array of Product is created so that it can use extension methods defined for IEnumerable<Product>.
- Yield keyword
- Func and delegate() in c#
- How var keyword infers the type from the value or if it is not found then will it create a new definition for the anonymous type?
- How HtmlHelper methods or any other extension methods (methods added to the classes that cannot be modified) are found?
- Url.Action() and route.MapRoute()
- Mischief with hidden field value
- How Entity Framework actually works? At the time of updating any record we first have to fetch that and then update it, is there any other elegant way of doing this?
- Asynchronous programming in .NET MVC

Important Notes:

- Handle the display block for User information message or error message (mostly using TempData) in the template file. The benefit of dealing with the message in the template file is that user will see it displayed on whatever page is rendered after they have saved the change (as long as the next view also uses the same Layout).