ASP.NET MVC3 Points

Contents

CH 0 - JQuery

Basic jQuery Theory - jQuery("DIV.MyClass").hide() makes all the matching div elements that have the CSS class MyClass suddenly vanish. - jQuery provides a shorthand syntax $() - jQuery is extremely concise same effects would take many lines of JavaScript.

Understanding jQuery Selectors - jQuery selectors are greedy, meaning they select as many elements as they can in the HTML DOM - One exception to this is $('#id') selector which selects the element with the specified ID; - Also @Html.TextBox("pledge.Amount") will render id="pledge_Amount" name="pledge.Amount" - $('td, th') selects all the td and th elements in the DOM - $('td input') selects all the input elements contained within td elements - *Using Attribute Selectors* - $('[attr]="val"') Selects elements that have an attr attribute whose value is val - $('[type][value="Delete"]') selects those elements that have a type attribute (with any value such as submit, reset etc) and a value attribute whose value is Delete.

Using jQuery Filters – these are a convenient means for narrowing the range of elements that we select. – takes the format of :function(n) – Example $('td:eq(8)') selects only the ninth item in the array of elements matched by the selector - The filters can be used in conjunction with selectors - Example: $('td:odd:eq(1)') – selects the second odd td element - *Using Content Filters* - These filters are focused on the content of an element, both in terms of text and other elements – Example: $(':contains("K2")') selects six elements: the td element that contains the text and all of this element's parents - *Using Form Filters* - convenient for selecting elements related to HTML forms. Example: :button - Selects button elements and input elements whose type is button, :password - Selects password elements.

Waiting for the DOM - $(document).ready(function () {} - JQUERY METHOD OVERLOADS – Example: $('tr').css('background-color', $('tr:first').css('background-color')) sets the background-color for all tr elements to match that of the first tr element (note that the first .css() is a method that takes 2 arguments) - we can chain jQuery methods together – Example: $(':submit[value="Reset"]').css('float', 'left').css('margin', '5px'); - Method chaining is one of the key characteristics of a fluent API, which can make coding simpler and code easier to read.

Working with the DOM - jQuery's support for manipulating the DOM is very comprehensive - jQuery DOM capabilities requires diligent perusal of the API reference and some careful experimentation. A special selector, $(this), creates a selection that contains the element currently being processed.

Using jQuery Visual Effects -

CH 1 – Introduction

MVC Architecture (Improved separation of concerns, User interaction with an actions and responses) – Extensibility (for each MVC Framework Component - Use the default implementation - Derive a subclass of the default implementation and change behavior - Replace the component entirely) - Tight Control over HTML and HTTP - (clean, standards-compliant markup, pages don't contain any View State data hence page size is small) – Testability (both unit testing and automation testing) - Powerful Routing System (improved URL Styles).

CH 2 – First MVC Application

Controller (derive from System.Web.Controller class- incoming requests are handled by controllers - Each public method in a controller is known as an action method – by convention the names we give to controllers should be descriptive and end with Controller – Routing system (we can enter /Home or /Home/Index – both will point to our action method called Index) – Rendering the View (ViewResult and Controller.View() method) – Showing Views (by convention view has the name of the action method and is contained in a folder named after the controller) - Adding Dynamic Output (using ViewBag).

Designing a Data Model (importance) - Adding a Strongly Typed View (@Model, `@using (Html.BeginForm())` , `@Html.TextBoxFor(x => x.Name)`, `@Html.DropDownListFor()`,[HttpGet] and [HttpPost] attributes for Action Methods

Using Model Binding - the names of the input elements are used to set the values of the properties in an instance of the model class, which is then passed to our POST-enabled action method.

Adding Validation – Attributes specified in the Model like [Required(ErrorMessage="Please enter your name")] and ModelState.IsValid property in our controller class, Validation Summary.

CH 3 – The MVC Pattern

The History of MVC - MVC forces a separation of concerns – inspired from Ruby on Rails - Understanding the MVC Pattern (Models represent the data that users work with – can be simple "view models" or "domain models" which contain the data in a business domain, Views - used to render some part of the model as a UI., Controllers - process incoming requests, perform operations on the model and select views to render to the user)

Understanding the Domain Model: A domain model is the single, authoritative definition of the business data and processes within your application. A persistent domain model is also the authoritative definition of the state of your domain representation

- Good practice to place the model in a separate C# assembly (In this way, you can create references to the domain model from other parts of the application but ensure that there are no references in the other direction.)

The ASP.NET Implementation of MVC - In MVC, controllers are C# classes – Each public method in a class derived from Controller is called an action method, which is associated with a configurable URL through the ASP.NET routing system. When a request is sent to the URL associated with an action method, the statements in the controller class are executed in order to perform some operation on the domain model and then select a view to display to the client <- *Important to note this point.*

Comparing MVC to Other Patterns - *Smart UI Pattern* (monolithic, similar to Winforms) - *Model-View Pattern* (UI is separate and the business logic is put into a separate domain model - impossible to perform automated unit tests). *Three-Tier Architectures* (UI, Model and data access layer (DAL) are introduced, still impossible to perform automated unit tests) – Others are MVP and MVVM.

Applying Domain-Driven Development (DDD) - The domain model is the heart of an MVC application. Everything else, including views and controllers, is just a means to interact with the domain model. *Ubiquitous Language* - adopt the domain terminology when it already exists. – For example call the domain models as "agents" and "clearances" in the domain instead of as "users" and "roles". And when modeling concepts that the domain experts don't have terms for, you should come to a common agreement about how you will refer to them, creating a ubiquitous language that runs throughout the domain model. DDD experts suggest that any change to the ubiquitous language should result in a change to the model – Aggregates (see book for example)

Defining Repositories - Persistence is not part of our domain model. It is an independent or orthogonal concern in our separation of concerns pattern. This means that we don't want to mix the code that handles persistence with the code that defines the domain model. Repositories are concerned only with loading and saving data; they don't contain any domain logic at all. We can complete the repository classes by adding to each method statements that perform the store and retrieve operations for the appropriate persistence mechanism.

Building Loosely Coupled Components - Using Dependency Injection (meaning if a class called PasswordResetHelper depends on an instance of IEMailSender for resetting the password, during runtime "some" object that implements this interface will be injected into the constructor of the PasswordResetHelper (called *constructor injection)* or to a property (called *setter injection)* – Dependency injecting a Repository to in a controller's constructor.
Good DI containers such as Ninject typically manage **Dependency chain resolution** (satisfying the request of a component that has its own dependencies), **Object life-cycle management**: (allowing you to select from singleton (the same instance each time), transient (a new instance each time), instance-per-thread, instance-per-HTTP-request, instance-from-a-pool, and many others), **Configuration of constructor parameter values**.

Getting Started with Automated Testing - Unit tests are simple to create and run, are brilliantly precise when you are working on algorithms, business logic etc. Integration testing can model how a user will interact with the UI, and can cover the entire technology stack that your application uses, including the web server and database.
Integration testing tends to be better at detecting new bugs that have arisen in old features; this is known as regression testing. Unit Tests follow the pattern known as arrange/act/assert (A/A/A). The most common approach to integration testing is UI automation (ex: Selenium RC, WatiN).

CH 4 – Essential Language Features

Automatically Implemented Properties (Example: public int ProductID { get; set; }) - Object Initializer Feature - Extension methods (Example: public static decimal TotalPrices(this ShoppingCart cartParam)) - Applying Extension Methods to an Interface (say IEnumerable<Product>) – defined in a separate class - (public static decimal TotalPrices(this IEnumerable<Product> productEnum)) - Creating Filtering Extension Methods - public static IEnumerable<Product> FilterByCategory(this IEnumerable<Product> productEnum, string categoryParam) {} - use a **yield** return statement to return each element one at a time (example yield return matchedProduct;) – helps in chaining methods together.
Using Lambda Expressions – Generally specified for Func<T,R>) - The following expression is lambda expression Func<Product, bool> categoryFilter = prod => prod.Category == "Soccer"; Here the => characters are read aloud as "goes to" and links the parameter to the result of the lambda expression - lambda expressions with method calls (prod => EvaluateProduct(prod)) - lambda expressions with multiple parameters (wrap the parameters in parentheses as (prod, count) => prod.Price > 20 && count > 0) - lambda expressions with multiple statements (using braces ({}).

Using Automatic Type Inference - C# var keyword - By combining object initializers and type inference we can create Anonymous Types as
var myAnonType = new { Name = "MVC", Category = "Pattern" };

LINQ - *query syntax (*from product in products etc*)* and *dot-notation syntax.* Each of the LINQ extension methods is applied to an IEnumerable<T> and returns an IEnumerable<T>, which allows us to chain the methods together to form complex queries – Projection (with Select) - Understanding Deferred LINQ Queries (the query isn't evaluated until the results are enumerated typically in a foreach loop) – Hence in LINQ some extension methods (such are *Sum())* are executed immediately whereas *OrderByDescending*() is executed as deferred)- Repeatedly Using a Deferred Query - A Deferred query is executed every time the results are enumerated - The IQueryable<T> interface is derived from IEnumerable<T> and is used to signify the result of a query executed against a specific data source.

Razor syntax - centered on the **@** symbol - **@:** means consider the following as text as textual content.

CH 4 – Essential Tools for MVC

Getting Started with Ninject - Getting Started with Ninject – StandardKernel object - two stages to working with Ninject - The first is to bind the types you want associated with the interfaces you've created. - Bind and To methods (ninjectKernel.Bind<IValueCalculator>().To<LinqValueCalculator<();) - The second stage is to use the Ninject Get method to create an object (IValueCalculator calcImpl = ninjectKernel.Get<IValueCalculator>();) - Creating Chains of Dependency - examines the couplings between that type and other types. - Specifying Property and Parameter Values – WithPropertyValue() method – this method allows chaining, so we can pass multiple property values ninjectKernel.Bind<IDiscountHelper>().To<DefaultDiscountHelper>().**WithPropertyValue**("DiscountSize", 40M); We can do the same thing with constructor parameters - <DefaultDiscountHelper>().**WithConstructorArgument**("discountParam", 50M) - Using Self-Binding - where a concrete class can be requested (and therefore instantiated) from the Ninject kernel. - Binding to a Derived Type (ninjectKernel.Bind<ShoppingCart>() .**To<LimitShoppingCart>()** .WithPropertyValue("ItemLimit", 200M);) (note here that LimitShoppingCart is derived from ShoppingCart)- Using Conditional Binding (ninjectKernel.Bind<IValueCalculator>().To<IterativeValueCalculator>().**WhenInjectedInto**<LimitShoppingCart>();)

CH 5,6,7 8 – SportsStore

Domain Assembly (contains the Model)

- Define the interfaces.
- Define the EntityFramework DbContext implementation.
- Entity classes with properties and methods of entities such as Product, Cart etc. The properties can also contain [Required] attribute annotations.
- `public IQueryable<Product> Products` : Return type of `IQueryable<T>`

Unit Tests Assembly

- Define the Unit tests against both the Web Project and the Domain Assembly.

Web Assembly

- `IModelBinder` Defined to create C# objects from HTTP requests in order to pass them as parameter values to action methods.
- Controller classes

- Generally take repository objects as parameters in their constructor.
  - Each action method returns a ViewResult, RedirectToRouteResult, PartialViewResult etc.
  - Call **RedirectToAction()** to execute a different action Ex: return RedirectToAction("Index", new { returnUrl }); This happens when we return RedirectToRouteResult and usually used for showing a different view when executing an existing action.
  - Call **PartialView()** to return a PartialViewResult
  - Contains [HttpPost] actions for Postback and if(ModelState.IsValid) checks. Use ModelState.AddModelError to raise any errors in the model state.
  - Action Methods connect to Views to display UI.
  - Use anonymous objects when required.
- Views (.cshtml)
  - Html.BeginForm() can be used with or without parameters. Html.BeginForm("AddToCart", "Cart") invokes a specific action (here AddToCart action of controller Cart).
  - @Html.EditorFor() will find the best suitable UI element depending on the data type.
  - We can use either Hidden or HiddenFor to embed a hidden field.
  - Use @Html.ActionLink("Checkout now", "Checkout") to embed a hyperlink that when clicked will execute the Checkout action. Similarly @Html.RouteLink() does a similar functionality but the latter will allow us to specify route values (controller name, action name and action parameters) and html attribute values.
  - Use @Html.RenderAction("Menu", "Nav"); to embed a child view in line with a parent view.
  - Use @Html.Action("ChildAction") to embed the output of a child action in a parent view.

CH 11 – URLs, Routing, and Areas

Introducing the Routing System - The routing system has two functions: 1. Examine an incoming URL 2. Generate outgoing URLs - The routing system classes are in the System.Web assembly (and not in System.Web.Mvc). – The Global.asax file contains Application_Start() that calls RegisterRoutes(RouteTable.Routes) method – Note that Routes property is of type RouteCollection.

Introducing URL Patterns - URLs can be broken down into segments. For example: **http://mysite.com/Admin/Index** contains the URL Pattern {controller}/{action} – where Admin is the first segment and Index is the second segment. An MVC application will usually have several routes, and the routing system will compare the incoming URL to the URL pattern of each route until it can find a match. *Note: The routing system doesn't have any special knowledge of controllers and actions. It just extracts values for the segment variables and passes them along the request pipeline. It is later in the pipeline, when the request reaches the MVC Framework proper, that meaning is assigned to the controller and action variables. This is why the routing system can be used with Web Forms and how we are able to create our own variables.* Two key behaviors of URL patterns - URL patterns are **conservative** (http://mysite.com/Admin will mean *No match—too few segments)* and **liberal -**

http://mysite.com/Index/Admin will mean an exact match for controller = Index and action = Admin) . A convenient way of registering routes is to use the MapRoute method () of RouteCollection class. : Example: routes.MapRoute("MyRoute", "{controller}/{action}");
Note: It is recommend to unit test your routes to make sure they process incoming URLs as expected.(For unit testing routes see book).

Defining Default Values for Routing - A default value is applied when the URL doesn't contain a segment that can be matched to the value. Example: routes.MapRoute("MyRoute", "{controller}/{action}", new { action = "Index" }); here we have provided a default value of Index for the action variable. Now it will match Urls like http://mydomain.com/Home. Note: We have to unit test Default routing values also.

Using Static URL Segments - Suppose we want to match a URL like this to support URLs that are prefixed with *Public* like http://mydomain.com/Public/Home/Index. We can do so by using a pattern like routes.MapRoute("", "Public/{controller}/{action}", new { controller = "Home", action = "Index" });This URL pattern will match only URLs that contain three segments, the first of which must be Public. We can also create URL patterns that have segments containing both static and variable elements. Example: routes.MapRoute("", "X{controller}/{action}"); will match urls like http://mydomain.com/XHome/Index. Important: routes are applied in the order in which they appear in the RouteCollection object. The MapRoute method adds a route to the end of the collection, which means that routes are generally applied in the order in which we add them - We can combine static URL segments and default values to create an alias for a specific URL, for example: if we used to have a controller called Shop, which has now been replaced by the Home controller, we can create a route to preserve the old URL schema as routes.MapRoute("ShopSchema", "Shop/{action}", new { controller = "Home" }); The action value is taken from the second URL segment. The URL pattern doesn't contain a variable segment for controller, so the default value we have supplied (Home as Controller) is used.

Defining Custom Segment Variables - We can also define our own segment variables: routes.MapRoute("MyRoute", "{controller}/{action}/**{id}**", new { controller = "Home", action = "Index", **id = "DefaultId"** }); This route will match any zero-to-three-segment URL. The contents of the third segment will be assigned to the id variable, and if there is no third segment, the default value will be used. We can access any of the segment variables in an action method by using the RouteData.Values property: For example: ViewBag.CustomVariable = RouteData.Values["id"]; Another elegant way will be to define parameters to our action method with names that match the URL pattern variables, such as public ViewResult Index(string id) {}.

Defining Optional URL Segments - An optional URL segment is one that the user does not need to specify, but for which no default value is specified - routes.MapRoute("MyRoute", "{controller}/{action}/{id}", new { controller = "Home", action = "Index", id **= UrlParameter.Optional** });

Defining Variable-Length Routes – We can define support for variable segments by designating one of the segment variables as a *catchall* as routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{*catchall}", new { controller = "Home", action = "Index", id = UrlParameter.Optional });
Hence for a URL like **mydomain.com/Customer/List/All/Delete/Perm** we get controller = Customer, action = List, id = All, catchall = Delete/Perm

Prioritizing Controllers by Namespaces – If two different namespaces contain controllers with the same name (say AccountController) then accessing a route with that controller will cause an error. To address this problem, we can tell the MVC Framework to give preference to certain namespaces when attempting to resolve the name of a controller class as routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{*catchall}", new { controller = "Home", action = "Index", id = UrlParameter.Optional }, **new[] { "URLsAndRoutes.Controllers"}**); The namespaces added to a route are given equal priority – For example routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{*catchall}", new { controller = "Home", action = "Index", id = UrlParameter.Optional }, new[] { "URLsAndRoutes.Controllers", "AdditionalControllers"}); Here the MVC Framework doesn't check the first namespace before moving on to the second and so forth, on the other hand it is simply going to return an error. If we want to give preference to a single controller in one namespace, but have all other controllers resolved in another namespace, we need to create multiple routes by calling MapRoute again with the second namespace.

We can tell the MVC Framework to look only in the namespaces that we specify. If a matching controller cannot be found, then the framework won't search elsewhere. We do this with : myRoute.DataTokens["UseNamespaceFallback"] = false; where myRoute data is the reference we get from our MapRoute() call.

Constraining Routes – We can restrict the set of URLs that a route will match against - ***Constraining a Route Using a Regular Expression –*** For example, use a constraint with a regular expression that matches URLs only where the value of the controller variable begins with the letter H as routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{*catchall}", new { controller = "Home", action = "Index", id = UrlParameter.Optional }, **new { controller = "^H.*"}**, new[] { "URLsAndRoutes.Controllers"}); Important: Default values are used before constraints are checked.For example in the above if we request the URL /, the default value for controller, which is Home, is applied. The constraints are then checked, and since the controller value beings with H, the default URL will match the route. If we use a constraint like **new { controller = "^H.*", action = "^Index$|^About$"},** this will match URLs only when the controller variable begins with the letter H and the action variable is Index or About. ***Constraining a Route Using HTTP Methods -*** We can constrain routes so that they match a URL only when it is requested using a specific HTTP method (such as Get, Post etc) – Example: routes.MapRoute("MyRoute", "{controller}/{action}/{id}",httpMethod = new HttpMethodConstraint("GET") }); ***Defining a Custom Constraint*** – We can define your own custom constraints by implementing the IRouteConstraint interface that defines the **Match** method, which an implementation can use to indicate to the routing system if its constraint has been satisfied. (See example in book).

Routing Requests for Disk Files - The routing system provides integrated support for serving content such as static Html files, images etc . If you start the application and request the URL such as /Content/StaticContent.html, you will see the contents of this HTML file (if it exists). By default, the routing system checks to see if a URL matches a disk file before evaluating the application's routes. If there is a match, then the disk file is served, and the routes are never used. We can reverse this behavior so that our routes are evaluated before disk files are checked by setting the **RouteExistingFiles** property of the RouteCollection to true. Example routes.RouteExistingFiles = true; Routing requests intended for disk files requires careful thought because a request for /Content/StaticContent.html will be matched by a URL pattern such as {controller}/{action} and the user will be sent a 404 – Not Found error because there is no controller by name Content and Action by name StaticContent.html.

Bypassing the Routing System – The method routes.**IgnoreRoute**("Content/{filename}.html"); makes the routing system less inclusive and prevents URLs from being evaluated against our routes. We can use segment variables like **{filename}** to match a range of URLs. In this case, the URL pattern will match any two-segment URL where the first segment is Content and the second content has the .html extension. Note: Where we place the call to the **IgnoreRoute** method is important.

Generating Outgoing URLs - We also need to be able use our URL schema to generate outgoing URLs we can embed in our views, so that users can click links and submit forms back to our application in a way that will target the correct controller and action. Important: In ASP.NET MVC, we must never manually define outgoing URLs like **<a href="/Home/About">** targetting the Home Controller's About action. This is because every time you change the URL schema for the application, you break all of your outgoing URLs. ***Generating Outgoing URLs in Views -*** The simplest way to generate an outgoing URL in a view is to call the Html.ActionLink method within a view as @Html.ActionLink("About this application", "About"). Changing the routes that define your URL schema changes the way that outgoing URLs are generated – for example the above action link will get you this Html <a href="/Home/About">About this application</a> if the following is defined : routes.MapRoute("MyRoute", "{controller}/{action}/{id}", new { controller = "Home", action = "Index", id = UrlParameter.Optional }); On the other hand we get this HTML: <a href="/App/DoAbout">About this application</a> if the following route is defined first: routes.MapRoute("NewRoute", "App/Do{action}", new { controller = "Home" }); To be very clear: the routing system doesn't try to find the route that provides the *best* matching route. It finds only the *first* match, at which point it uses the route to generate the URL; any subsequent routes are ignored. For this reason, you should define your most specific routes first. ***Targeting Other Controllers***: To create an outgoing URL that targets a different controller (remember, by default it will target the same controller that rendered the view), we can use for example: @Html.ActionLink("About this application", "About", "MyController") ***Passing Extra Values:*** You can pass values for segment variables using an anonymous type, with properties representing the segments: for example: @Html.ActionLink("About this application", "About", new { id = "MyID" }). In this example, we have supplied a value for a segment variable called id. Important: It is recommend that you supply values for all of the segment variables in a URL pattern – for example: For the route routes.MapRoute("MyRoute", "{controller}/{action}/{color}/{page}"); we must generate the outbound URL as @Html.ActionLink("Click me", "List", "Catalog", new {color=green}, new {page=789}) Also when we supply values for properties that do not correspond with segment variables, the values are appended to the outgoing URL as the query string: for Example @Html.ActionLink("About this application", "About", new { id = "MyID", myVariable = "MyValue" }) – this generates the following HTML: <a href="/Home/About/MyID?myVariable=MyValue">About this application</a>
Specifying HTML Attributes – For example to set an id attribute and assigns a CSS class to the HTML element we can use anonymous types as @Html.ActionLink("About this application", "Index", "Home", null, **new {id = "myAnchorID", @class = "myCSSClass"}**) - ***Generating Fully Qualified URLs in Links*** - we can also use the ActionLink helper method to generate fully qualified URLs as @Html.ActionLink("About this application", "Index", "Home", "https", "myserver.mydomain.com", " myFragmentName", new { id = "MyId"}, new { id = "myAnchorID", @class = "myCSSClass"}) that renders <a class="myCSSClass" href="https://myserver.mydomain.com/Home/Index/MyId#myFragmentName" id="myAnchorID">About this application</a>. ***Generating URLs (and Not Links)*** The Html.ActionLink helper method generates complete HTML <a> elements. The Url.Action method works in the same way as the Html.ActionLink method, except that it generates only the URL. For example: @Url.Action("Index", "Home", new { id = "MyId" }) generates /Home/Index/MyId. ***Generating Links and URLs from Routing Data***: Sometimes it is useful to treat controller and action just like any other variables, and to generate a link or URL by providing a collection of name/value pairs. We can do this by using helper methods such as @Html.RouteLink("Routed Link", new { controller = "Home", action = "About", id="MyID"}) that renders <a href="/Home/About/MyID">Routed Link</a> . We can use the Url.RouteUrl helper method to generate just the URL, as @Url.RouteUrl(new { controller = "Home", action = "About", id = "MyID" }). ***Generating Outgoing URLs in Action Methods*** - If we just need to generate a URL in our Action methods, we can use the same helper method that we used in the view as string myActionUrl = Url.Action("Index", new { id = "MyID" }); Good Practice: We tend to avoid naming our routes (in the MapRoute method by specifying null for the route name parameter). We prefer to use code comments to remind ourselves of what each route is intended to do.

Redirection Basics: The **RedirectToAction()** method is used to instruct the MVC Framework to issue a redirect instruction to a URL that will invoke the specified action. For example public ActionResult MyActionMethod() { return RedirectToAction("Index"); }. This invokes the Index Action. If you want to send a redirect using a URL generated from just object properties, you can use the **RedirectToRoute()** method, as public ActionResult MyOtherActionMethod() { return RedirectToRoute(new { controller = "Home", action = "Index", id = "MyID" }); }. Both methods return a RedirectToRouteResult object.

Customizing the Routing System - Creating a Custom RouteBase Implementation - If you don't like the way that standard Route objects match URLs, or want to implement something unusual, you can derive an alternative class from **RouteBase** and override **GetRouteData**(HttpContextBase httpContext) for inbound URL matching and **GetVirtualPath** () for outbound URL generation (see book for more details) ***Creating a Custom Route Handler* - MvcRouteHandler** is the main class that connects the routing system to the MVC Framework. The routing system also lets us define our own route handler by implementing the **IRouteHandler** interface. This interface contains a method that returns a IHttpHandler instance as public IHttpHandler GetHttpHandler(); The purpose of the **IRouteHandler** interface is to provide a means to generate implementations of the IHttpHandler interface, which is responsible for processing requests. In the MVC implementation of these interfaces, controllers are found, action methods are invoked, views are rendered, and the results are written to the response. (see book for more details)

Working with Areas - The MVC Framework supports organizing a web application into areas, where each area represents a functional segment of the application, such as administration, billing etc – Every area contains a class that is derived from the AreaRegistration class which is used to register the area with MVC and this class overrides the RegisterArea() method where the routes are specified - During Area Registration, it is important to ensure that the route names are unique across the entire application and not just the area for which they are intended - working inside an area is pretty much the same as working in the main part of an MVC project - While working with areas it is important to note that any ambiguous controller issues (multiple controllers with the same name) are resolved by explicitly specifying controller within the appropriate namespaces to be given preference. ***Generating Links to Actions in Areas -*** You don't need to take any special steps to create links that refer to actions in the same MVC area that the user is already on. The MVC Framework detects that the current request relates to a particular area, and then outbound URL generation will find a match only among routes defined for that area. For example, this addition to the view in our Admin area: @Html.ActionLink("Click me", "About") generates <a href="/Admin/Home/About">Click me</a> where Home is the controller. To create a link to an action in a different area, or no area at all, you must create a variable called **area** (this is a reserved work) and use it to specify the name of the area you want, like this: @Html.ActionLink("Click me to go to another area", "Index", new { area = "Support" }) If you want to link to an action on one of the top-level controllers (then you should specify the area as an empty string, like this: @Html.ActionLink("Click me to go to another area", "Index", new { area = "" })

URL Schema Best Practices - *Make Your URLs Clean and Human-Friendly* (Example: http://www.amazon.com/books/pro-aspnet-mvc3-framework) - Use file name extensions only for specialized file types (such as .jpg, .pdf, and .zip) and <u>not</u> for .html files. *GET and POST: Pick the Right One* - The rule of thumb is that GET requests should be used for all read-only information retrieval, while POST requests should be used for any operation that changes the application state. In standards-compliance terms, GET requests are for safe interactions (having no side effects besides information retrieval), and POST requests are for unsafe interactions (making a decision or changing something). These conventions are set by the World Wide Web Consortium (W3C).

CH 12 – Controllers and Actions

Introducing the Controller - Every request that comes to your application is handled by a controller. In the ASP.NET MVC Framework, controllers are .NET classes that contain the logic required to handle a request. The role of the controller is to encapsulate your application logic. This means that controllers are responsible for processing incoming requests, performing operations on the domain model, and selecting views to render to the user. ***Creating a Controller with IController -*** In the MVC Framework, controller classes must implement the IController interface from the System.Web.Mvc namespace and define the method: void Execute(RequestContext requestContext); You can implement the IController interface to create any kind of request handling and result generation you require. Don't like action methods? Don't care for rendered views? Then you can just take matters in your own hands and write a better, faster, and more elegant way of handling requests. ***Creating a Controller by Deriving from the Controller Class -*** System.Web.Mvc.Controller is the class that provides the request handling support that most MVC developers will be familiar with. The Controller class provides three key features: **Action methods, Action results, Filters.** Our job, as a derivation of the Controller class, is to implement action methods, obtain whatever input we need to process a request, and generate a suitable response. ***Receiving Input -*** When you create a controller by deriving from the Controller base class, you get access to a set of **convenience properties** to access information about the request. These properties include **Request, Response, RouteData, HttpContext**, and **Server**. Example: string serverName = Server.MachineName; string clientIP = Request.UserHostAddres; ***Using Action Method Parameters*** - Action methods can take parameters. This is a neater way to receive incoming data than extracting it manually from context objects, and it makes you action methods easier to read. Example: public ActionResult ShowWeatherForecast(string city, DateTime forDate){ // ... implement weather forecast here ... }
Understanding How Parameters Objects Are Instantiated : The base Controller class obtains values for your action method parameters using MVC Framework components called value providers and model binders. Value providers represent the set of data items available to your controller. There are built-in value providers that fetch items from Request.Form, Request.QueryString, Request.Files, and RouteData.Values. The values are then passed to model binders that try to map them to the types that your action methods require as parameters. The default model binders can create and populate objects of any .NET type, including collections and project-specific custom types. **Understanding Optional and Compulsory Parameters** :*Value-type parameters are compulsory*. To make them optional, either specify a default value (see the next section) or change the parameter type to a nullable type (such as int? or DateTime?), so the MVC Framework can pass null if no value is available. *Reference-type parameters are optional.* To make them compulsory (to ensure that a non-null value is passed), add some code to the top of the action method to reject null values. For example, if the value equals null, throw an ArgumentNullException. ***Specifying Default Parameter Values* -** If you want to process requests that don't contain values for action method parameters, but you would rather not check for null values in your code or have exceptions thrown, you can use the C# optional parameter feature instead. Example: public ActionResult Search(string query= "all", int page = 1) {// ..}.

Producing Output
Understanding Action Results - The Instead of working directly with the Response object, we return an object derived from the ActionResult class that describes what we want the response from our controller to be, such as rendering a view or redirecting to another URL or action method. When the MVC Framework receives an ActionResult object from an action method, it calls the ExecuteResult method defined by that class (meaning ActionResult class). The action result implementation then deals with the Response object for you, generating the output that corresponds to your intention. (For example of our sample RedirectActionResult that derives from ActionResult see book).

There is nothing in the action result system that is very complex, and you end up with simpler, cleaner and more consistent code. Also, you can unit test your action methods very easily.(Note: It is important to know the various ActionResults. Please see book for Built-in ActionResult Types). ***Returning HTML by Rendering a View -*** The most common kind of response from an action method is to generate HTML and send it to the browser. When using the action result system, you do this by creating an instance of the ViewResult class that specifies the view you want rendered in order to generate the HTML Example: public ViewResult Index() { return View("Homepage"); } When the MVC Framework calls the ExecuteResult method of the ViewResult object, a search will begin for the view that you have specified.
Passing Data from an Action Method to a View - Providing a View Model Object - You can send an object to the view by passing it as a parameter to the View method. Example : public ViewResult Index() { DateTime date = DateTime.Now; return View(date); } We can access the object in the view using the Razor Model keyword as The day is: @(((DateTime)Model).DayOfWeek). Such a view is known as an untyped or weakly typed view. The view doesn't know anything about the view model object, and treats it as an instance of object. To get the value of the DayOfWeek property, we need to cast the object to an instance of DateTime. This works, but produces messy views. We can tidy this up by creating strongly typed views, in which we tell the view what the type of the view model object will be. For example : @model DateTime and later accessing the property as The day is: @Model.DayOfWeek. Notice that we use a lowercase m when we specify the model type and an uppercase M when we read the value. ***Passing Data with the ViewBag –*** This feature allows you to define arbitrary properties on a dynamic object and access them in a view. Setting a value is done as (for example): ViewBag.Message = "Hello"; We simply get the same properties that we set in the action method, as The message is: @ViewBag.Message ***Passing Data with View Data:*** The View Bag feature was introduced with MVC version 3. Prior to this, the main alternative to using view model objects was view data. The View Data feature similarly to the View Bag feature, but it is implemented using the ViewDataDictionary class rather than a dynamic object. The ViewDataDictionary class is like a regular key/value collection and is accessed through the ViewData property of the Controller class. Example: ViewData["Message"] = "Hello"; You read the values back in the view as you would for any key/value collection: Example: The message is: @ViewData["Message"]. ***Performing Redirections:*** A common result from an action method is not to produce any output directly, but to redirect the user's browser to another URL. *THE POST/REDIRECT/GET PATTERN:* During Http POST, in order to minimize the risk that the user will click the browser's reload button and resubmit the form a second time, causing unexpected and undesirable results we can follow the pattern called Post/Redirect/Get. In this pattern, you receive a POST request, process it, and then redirect the browser so that a GET request is made by the browser for another URL. When you perform a redirect, you send one of two HTTP codes to the browser: Send the HTTP code 301, which indicates a permanent redirection (meaning it instructs the recipient of the HTTP code not to request the original URL ever again and to use the new URL that is included alongside the redirection code) or Send the HTTP code 302, which is a temporary redirection (Most frequently used type of redirection). ***Redirecting to a Literal URL -*** The most basic way to redirect a browser is to call the Redirect method, which returns an instance of the RedirectResult class, as public RedirectResult Redirect() { return Redirect("/Example/Index"); } This sends a temporary redirection. You can send a permanent redirection using the RedirectPermanent() method. ***Redirecting to a Routing System URL***- The problem with using literal URLs for redirection is that any change in your routing schema means that you need to go through your code and update the URLs. As an alternative, you can use the routing system to generate valid URLs with the RedirectToRoute method, which creates an instance of the RedirectToRouteResult. For example: public RedirectToRouteResult Redirect() { return RedirectToRoute(new { controller = "Example", action = "Index", ID = "MyID" }); } The RedirectToRoute method issues a temporary redirection. Use the RedirectToRoutePermanent method for permanent redirections. ***Redirecting to an Action Method-*** You can redirect to an action method more elegantly by using the RedirectToAction method. This is just a wrapper around the RedirectToRoute method that lets you specify values for the action method and the controller without needing to create an anonymous type. Example: public RedirectToRouteResult Redirect() { return RedirectToAction("Index"); } If you want to redirect to another controller, you need to

provide the name as a parameter, like this: return RedirectToAction("Index", "MyController"); *PRESERVING DATA ACROSS A REDIRECTION:* A redirection causes the browser to submit an entirely new HTTP request, which means that you don't have access to the details of the original request. If you want to pass data from one request to the next, you can use the Temp Data feature. TempData is similar to Session data, except that TempData values are marked for deletion when they are read, and they are removed when the request has been processed. TempData["Message"] = "Hello"; You can get a value from TempData without marking it for removal by using the Peek method, like this: DateTime time = (DateTime)TempData.Peek("Date"); You can preserve a value that would otherwise be deleted by using the Keep method, like this: TempData.Keep("Date"); The Keep method doesn't protect a value forever. If the value is read again, it will be marked for removal once more. ***Returning Text Data -*** For all of the text data types (except JSON), we can use the general-purpose ContentResult action result as public ContentResult Index() { string message = "This is plain text"; return Content(message, "text/plain", Encoding.Default); } ***Returning XML Data -*** Returning XML data from an action method is very simple, especially when you are using LINQ to XML and the XDocument API to generate XML from objects. We use the same Controller.Content helper method to return an Xml string. ***Returning JSON Data –*** In recent years, the use of XML documents and XML fragments in web applications has waned, in favor of the JavaScript Object Notation (JSON) format. JSON is a lightweight, text-based format that describes hierarchical data structures. JSON is most commonly used to send data from the server to a client in response to AJAX queries. You can create JsonResult objects using the Controller.Json convenience method, as [HttpPost] public JsonResult JsonData() {StoryLink[] stories = GetAllStories(); return Json(stories); } (For security reasons, the JsonResult object will generate a response only for HTTP POST requests). ***Returning Files and Binary Data***- FileResult is the abstract base class for all action results concerned with sending binary data to the browser. Controller.File helper method is used for this purpose. This will return the appropriate type of FileResult (FilePathResult sends a file directly from the server file system, FileContentResult sends the contents of an in-memory byte array and FileStreamResult sends the contents of a System.IO.Stream object that is already open). Example public FileResult AnnualReport() { return File("C:\AnnualReport2011.pdf", "application/pdf", AnnualReport2011.pdf); } Here the last parameter is the download name***. Returning Errors and HTTP Codes*** - You can send a specific HTTP status code to the browser using the HttpStatusCodeResult class. There is no controller helper method for this, so you must instantiate the class directly, as public HttpStatusCodeResult StatusCode() { return new HttpStatusCodeResult(404, "URL cannot be serviced"); } - This is the same as return HttpNotFound(); - Sending a 401 Result – HttpUnauthorizedResult class, returns the 401 code as return new HttpUnauthorizedResult(); ***Creating a Custom Action Result*** - The built-in action result classes are sufficient for most situations and applications, but you can create your own custom action results for those occasions when you need something special. Here we derive our class from ActionResult class and override the method: public override void ExecuteResult(ControllerContext context).

CH 13 – Filters

Filters inject extra logic into the request processing pipeline. They provide a simple and elegant way to implement cross-cutting concerns. This term refers to functionality that is used all over an application and doesn't fit neatly into any one place, so it would break the separation of concerns pattern. Classic examples of cross-cutting concerns are logging, authorization, and caching.

Introducing the Four Basic Types of Filters - The MVC Framework supports four different types of filters. They are:

Filter Type	Interface	Default Implementation	Description
Authorization	IAuthorizationFilter	AuthorizeAttribute	Runs first, before any other filters or the action method
Action	IActionFilter	ActionFilterAttribute(abstract)	Runs before and after the action method
Result	IResultFilter	ActionFilterAttribute (abstract)	Runs before and after the action result is executed
Exception	IExceptionFilter	HandleErrorAttribute	Runs only if another filter, the action method, or the action result throws an exception

Before the MVC Framework invokes an action, it inspects the method definition to see if it has attributes that implement the interfaces listed in Table 13-1. If so, then at the appropriate point in the request pipeline, the methods defined by these interfaces are invoked - Filters can be applied to individual action methods or to an entire controller. Important: If you have defined a custom base class for your controllers, any filters applied to the base class will affect the derived classes.

Using Authorization Filters - Let's set the scene. The MVC Framework has received a request from a browser. The routing system has processed the requested URL, and extracted the name of the controller and action that are targeted. A new instance of the controller class is created, but before the action method is called, the MVC Framework checks to see if there are any authorization filters applied to the action method. If there are, then the sole method defined by the IAuthorizationFilter interface, OnAuthorization, is invoked. If the authentication filter approves the request, then the next stage in the processing pipeline is performed. If not, then the request is rejected.

Creating an Authentication Filter : The simplest way to create an authorization filter is to subclass the AuthorizeAttribute class and override the AuthorizeCore method. This ensures that we benefit from the features built in to AuthorizeAttribute. (See the code for this useful filter in the book).

The interesting part of AuthorizeAttribute class is the implementation of the OnAuthorization method. The parameter that is passed to this method is an instance of the AuthorizationContext class, which is derived from ControllerContext. ControllerContext gives us access to some useful objects, not least of which is HttpContextBase. The AuthorizationContext defines two additional properties: ActionDescriptor and Result. ActionDescriptor, returns an instance of System.Web.Mvc.ActionDescriptor, which you can use to get information about the action to which your filter has been applied. The second property, Result, is the key to making your filter work. If you are happy to authorize a request for an action method, then you do nothing in the OnAuthorization method. Your silence is interpreted by the MVC Framework as agreement that the request should proceed. However, if you set the Result property of the context object to be an ActionResult object, the MVC Framework will use this as the result for the entire request. The remaining steps in the pipeline are not performed, and the result you have provided is executed to produce output for the user. Note that we above have been given to just show how the filters work. We generally never implement our own AuthorizeAttribute and are happy to use the built in AuthorizeAttribute class provided by the MVC framework.

Using the Built-in Authorization Filter : For most applications, the authorization policy that AuthorizeAttribute provides is sufficient. If you want to implement something special, you can derive from this class. This is much less risky than implementing the IAuthorizationFilter interface directly, but you should still be very careful to think through the impact of your policy and test it thoroughly. (A good implementation of a custom Authorization filter that derives from FilterAttribute and implementing IAuthorizationFilter attribute is given in the article of item no: 3 of Project Essentials below).

The AuthorizeAttribute class provides two different points of customization:

- The AuthorizeCore method, which is called from the AuthorizeAttribute implementation of OnAuthorization and implements the authorization check.
- The HandleUnauthorizedRequest method, which is called when an authorization check fails. (for both examples see book).

Using Exception Filters - Runs only if another filter, the action method, or the action result throws an exception.

Creating an Exception Filter - Exception filters must implement the IExceptionFilter interface that defines the OnException() method as public interface IExceptionFilter { void OnException(ExceptionContext filterContext); } The OnException method is called when an unhandled exception arises. The parameter for this method is an ExceptionContext object that derives from the ControllerContext class.This parameter in addition to ActionDescriptor and Result, contains Exception and ExceptionHandled properties which are important. The exception that has been thrown is available through the Exception property. An exception filter can report that it has handled the exception by setting the ExceptionHandled property to true. All of the exception filters applied to an action are invoked even if this property is set to true, so it is good practice to check whether another filter has already dealt with the problem, to avoid attempting to recover from a problem that another filter has resolved. For complete example of “Implementing an Exception Filter” see book.

Using the Built-In Exception Filter: The HandleErrorAttribute is the built-in implementation of the IExceptionFilter interface and makes it easier to create exception filters. With it, you can specify an exception and the names of a view and layout. Example: [HandleError(ExceptionType=typeof(NullReferenceException), View="SpecialError")] public ActionResult Index() {……} Here when a NullReferenceException is encountered, this filter will set the HTTP result code to 500 (meaning server error) and render the view specified by the View property (using the default layout or the one specified by Master). When rendering a view, the HandleErrorAttribute filter passes a HandleErrorInfo view model object, which means that you can include details about the exception in the message displayed to the user. This is done using @Model HandleErrorInfo in the view (see book).

Using Action and Result Filters - Action and result filters are general-purpose filters that can be used for any purpose. Both kinds follow a common pattern. The built-in class for creating these types of filters,implements IActionFilter, that defines methods OnActionExecuting and OnActionExecuted. The MVC Framework calls the OnActionExecuting method before the action method is invoked. It calls the OnActionExecuted method after the action method has been invoked.

Implementing the OnActionExecuting Method - You can use this opportunity to inspect the request and elect to cancel the request, modify the request, or start some activity that will span the invocation of the action. The parameter to this method is an ActionExecutingContext object, which subclasses the ControllerContext class and defines the same two properties you have seen in the other context objects (ActionDescriptor and Result)

Implementing the OnActionExecuted Method - You can also use the filter to perform some task that spans the execution of the action. The parameter to this method is the ActionExecutedContext which contains 5 properties.(see example in book).

Implementing a Result Filter: Result filters are to action results what action filters are to action methods. Result filters implement the IResultFilter interface containing the OnResultExecuting() and OnResultExecuted() methods. Important: If we define one ActionFilter and one ResultFilter and implement the Executing and Executed method of each, the ActionExecuting and ActionExecuted will fire first and second respectively. Then the ResultExecuting event will fire, followed by the contents of the view followed by the ResultExecuted event.

Using the Built-In Action and Result Filter Class – The ActionFilterAttribute,is an abstract class that implements IActionFilter, IResultFilter interaces. The only benefit to using this class is that you don't need to implement the methods that you don't intend to use. Hence we could just derive our Filter attribute from ActionFilterAttribute and override only the required methods. For example: public class ProfileAllAttribute : ActionFilterAttribute { public override void OnActionExecuting(ActionExecutingContext filterContext) { …code goes here….} }

Using Other Filter Features

Filtering Without Attributes -The Controller class implements the IAuthorizationFilter, IActionFilter, IResultFilter, and IExceptionFilter interfaces. It also provides empty virtual implementations of each of the OnXXX methods you have already seen, such as OnAuthorization and OnException. This technique is most useful when you are creating a base class from which multiple controllers in your project are derived. But it is preferred to use attributes. We like the separation between the controller logic and the filter logic.

Using Global Filters - Global filters are applied to all of the action methods in your application. We make a regular filter into a global filter through the RegisterGlobalFilters method in Global.asax like: public static void RegisterGlobalFilters(GlobalFilterCollection filters) { filters.Add(new ProfileAllAttribute()); } The RegisterGlobalFilters method is called from the Application_Start method, which ensures that the filters are registered when your MVC application starts.

Ordering Filter Execution - Filters are executed by type. The sequence is - authorization filters, action filters, and then result filters. The framework executes your exception filters at any stage if there is an unhandled exception. However, within each type category (say action filters), you can take control of the order in which individual filters are used. This is done using the order attribute. For example:

```
[SimpleMessage(Message="A", Order=2)]
[SimpleMessage(Message="B", Order=1)]
public ActionResult Index() {..}
```

The Order parameter takes an int value and the MVC Framework executes the filters in ascending order. Also it is important to know that Global filters are executed first, then filters applied to the controller class, and then filters applied to the action method.

Using the Built-in Filters - The MVC Framework supplies some built-in filters, which are ready to be used in applications.
Using the RequireHttps Filter - This is an authorization filter and not an action filter. It allows you to enforce the use of the HTTPS protocol for actions. It redirects the user's browser to the same action, but using the https:// protocol prefix.

Using the OutputCache Filter **-** The OutputCache filter tells the MVC Framework to cache the output from an action method so that the same content can be reused to service subsequent requests for the same URL. Caching action output can offer a significant increase in performance, because most of the time-consuming activities required to process a request (such as querying a database) are avoided. Of course, the downside of caching is that you are limited to producing the exact same response to all requests, which isn't suitable for all action methods.

One of the nice features of the OutputCache filter is that you can apply it to child actions. A child action is invoked from within a view using the Html.Action helper method. This is a new feature in MVC 3 that allows you to be selective about which parts of a response are cached and which are generated dynamically. Example: public class ExampleController : Controller {[**OutputCache(Duration = 30)**] public ActionResult ChildAction() {..} }.This code specifies the no: of seconds (30 here) the output from an action method will be cached.

CH 14 – Controller Extensibility - skipped

CH 15 – Views

Creating a Custom View Engine - The MVC Framework includes two built-in view engines The Razor engine and the legacy ASPX engine (Web Forms view engine). View engines implement the IViewEngine interface, that contain FindView, FindPartialView and ReleaseView methods. Both FindView(), FindPartialView() methods return ViewEngineResult. If your view engine is able to provide a view for a request, then you create a ViewEngineResult using this constructor: public ViewEngineResult(IView view, IViewEngine viewEngine) .If your view engine can't provide a view for a request, then you use this constructor: public ViewEngineResult(IEnumerable<string> searchedLocations) The IView interface contains the single method Render ().The simplest way to see how this works—how IViewEngine, IView, and ViewEngineResult fit together—is to create a view engine. The steps to create a Custom View Engine is: 1. Create the IView implementation by implementing the Render() method. 2. Implement the IViewEngine implementation to implement the methods FindView(), FindPartialView() methods that return ViewEngineResult. Also implement the ReleaseView () method if required. Once done we can register the View Engine with the MVC framework as ViewEngines.Engines.Add(new DebugDataViewEngine()); where DebugDataViewEngine is our implementation of IViewEngine.

The MVC Framework supports the idea of there being several engines installed in a single application. When a ViewResult is being processed, the action invoker obtains the set of installed view engines and calls their FindView methods in turn. The action invoker stops calling FindView methods as soon as it receives a ViewEngineResult object that contains an IView. This means that the order in which engines are added to the ViewEngines.Engines collection is significant if two or more engines are able to service a request for the same view name.

Working with the Razor Engine - Razor is the view engine that was introduced with MVC 3 and replaces the previous view engine (known as the ASPX or Web Forms engine).

Understanding Razor View Rendering - The Razor View Engine compiles the views in your applications to improve performance. The views are translated into C# classes, and then compiled, which is why you are able to include C# code fragments so easily. The views in an MVC application are not compiled until the application is started, so to see the classes that are created by Razor, you need to start the application and navigate to an action method on the browser. The generated classes are written to the disk as C# code files and then compiled, You can find the generated files in the AppData folder. The code fragments that we prefixed with the @ symbol are expressed directly as C# statements. The HTML elements are handled with the WriteLiteral method, which writes the contents of the parameter to the result as they are given.

Adding Dependency Injection to Razor Views - Every part of the MVC request processing pipeline supports DI, and the Razor View Engine is no exception. For this create an interface (say call this ICalculator). Now create an abstract class that is derived from **WebViewPage** and contains a property of this interface say decorate it with the Inject attribute as [Inject] public ICalculator Calulator { get; set; } Now we can use the @inherits to specify the base class that gives us access to the Calculator property, which we are then able to get in order to receive an implementation of the ICalculator interface, all without creating a dependency between the view and the ICalculator implementation. Of course we need to add the binding beforehand as Bind<ICalculator>().To<SimpleCalculator>(); so as to use the injected Calculator propery.

Configuring the View Search Locations - You can change the view files that Razor searches for by creating a subclass of RazorViewEngine. This class is the Razor IViewEngine implementation. It builds on a series of base classes that define a set of properties that determine which view files are searched for. The properties **ViewLocationFormats**, **MasterLocationFormats**, **PartialViewLocationFormats** specify locations to look for views, partial views, and layouts.Similarly **AreaViewLocationFormats**, **AreaMasterLocationFormats** and **AreaPartialViewLocationFormats** properties specify locations to look for views, partial views, and layouts for an area. We can change them so that the our Razor engine searches a different set of locations.

Adding Dynamic Content to a Razor View - The whole purpose of views is to allow you to render parts of your domain model as a user interface. To do that, you need to be able to add dynamic content to views. You can add dynamic content to views in the four ways: 1. Inline code 2. HTML helper methods 3. Partial views 4. Child actions.

Using Inline Code: The simplest and easiest way to generate dynamic content is to use inline code—one or more C# statements prefixed with the @ symbol. This is the heart of the Razor View Engine. *Importing Namespaces into a View*: The easiest way to import a namespace is to use the @using tag in the Razor view. Example: @using DynamicData.Infrastructure. If you want to import a namespace to all of the views in a project, you can add the namespace to the Views/Web.config file, as <add namespace="DynamicData.Infrastructure"/> (Caution: This is the Web.config file that is in the Views folder, and *not* the main Web.config file for the project.) *Understanding Razor HTML String Encoding:* In Razor, when we use the @ tag to call a method or read a variable, the result is encoded. You don't need to take any explicit steps to ensure that your data is encoded, because Razor does it by default. If you want to disable automatic encoding, you can return an instance of MvcHtmlString as the result of the method or property or you can call @Html.Raw(outputstring).

Using HTML Helpers: A common need is to generate the same block of HTML repeatedly in a view. Duplicating the same HTML elements and Razor tags over and over again is tedious and prone to errors. Furthermore, if you need to make changes, you must do so in multiple places. Fortunately, the MVC Framework provides a mechanism for solving the problem, called HTML helpers. *Creating an Inline HTML Helper:* The most direct way to create a helper is to do so in the view, using the Razor @helper tag.(see book for example). *Creating an External Helper Method:* Inline helpers are convenient, but they can be used only from the view in which they are declared. If they involve too much code, they can take over that view and make it hard to read. The alternative is to create an external HTML helper method, which is expressed as a C# extension method.Example: public static class CustomHtmlHelpers { public static MvcHtmlString List(this HtmlHelper html, string[] listItems) {….} }
In the above example the second parameter is passed from the view that defines a string array parameter that will contain the list items to be rendered. The easiest way to create HTML in a helper method is to use the TagBuilder class, which allows you to build up HTML strings without needing to deal with all of the escaping and special characters.(see book for example).

Using the Built-in HTML Helpers - The MVC Framework includes a selection of built-in HTML helper methods that generate often-required HTML fragments or perform common tasks. **Creating Forms** - Html.BeginForm(),EndForm(), **Using Input Helpers** - Html.CheckBox(), Html.Hidden etc **Using Strongly Typed Input Helpers -** Html.CheckBoxFor(x => x.IsApproved), Html.HiddenFor(x => x.SomeProperty) - **Creating Select Elements -** Html.DropDownList(),Html.DropDownListFor() - **Creating Links and URLs -** Url.Content(),Html.ActionLink() -.

Using Sections - The Razor engine supports the concept of sections, which allow you to provide regions of content within a layout. You create sections using the Razor @section tag followed by a name for the section. The body of a section is just a fragment of Razor syntax and static markup.Example: `@section Footer { <h4>This is the footer</h4> }` We insert the contents of a section into a layout using the RenderSection helper (@RenderSection(sectionName)), passing in the name of the section. The parts of the view that are not contained in a section are available through the RenderBody helper. This helper inserts content from anywhere in the view that is not contained in a @section block,

Testing For Sections - You can check to see if a view has defined a specific section from the layout using the IsSectionDefined helper. Example: `@if (IsSectionDefined("Footer")) { …}`

Rendering Optional Sections - The overloaded version of RenderSection takes a parameter that specifies whether a section is required. By passing in false, we make the section optional, so that the layout will include the contents of the Footer section if the view defines it, but <u>will not</u> throw an exception if it doesn’t. For example: `@RenderSection("Footer", false)`

CH 16 – Model Templates

Using Templated View Helpers - The idea of the templated view helpers is that they are more flexible. We don't have to worry about specifying the HTML element we want to represent a model property—we just say which property we want displayed and leave the details to the MVC Framework to figure out. For example if our model looks like **public class Person { public int PersonId { get; set; } public string FirstName { get; set; }** then MVC Templated HTML Helpers such as Html.Display("FirstName") renders a read-only view of the specified model property, choosing an HTML element according to the property's type and metadata. Similarly Html.DisplayFor(x =>x.FirstName) is a strongly typed version of the previous helper. Others are Html.Editor("FirstName") and Html.EditorFor(x =>x.FirstName) that render an editor for the specified model property, Html.Label("FirstName") and Html.LabelFor(x => x.FirstName) render an HTML <label> element referring to the specified model property and Html.DisplayText("FirstName") and Html.DisplayText("FirstName") bypass all templates and renders a simple string representation of the specified model property.

Scaffolding: These helpers generate HTML for an individual model property, but the MVC helper also includes helpers that will generate HTML for all the properties in a model object. This process is known as ***scaffolding***, whereby a complete view or editor is created for our model object without us having to explicitly deal with individual elements of the display. For example: Html.DisplayForModel() renders a read-only view of the entire model object. Similarly Html.EditorForModel() and Html.LabelForModel() render editor elements and HTML <label> element referring to the entire model object. The templated helpers are a useful and convenient feature. However, as we'll see, we often have to tweak the way they operate in order to get precisely the results we require—this is particularly true when working with the scaffolding helpers.

Styling Generated HTML - When we use the templated helpers to create editors, the HTML elements that are generated contain useful values for the class attribute, which we can use to style the output. For example, this helper: @Html.EditorFor(m => m.BirthDate) generates the following HTML: <input **class="text-box single-line"** id="BirthDate" name="BirthDate" type="text" value="25/02/1975 00:00:00" /> Using these class attributes, it is a simple matter to apply styles to the generated HTML. The same is applicable for elements generated thru scaffolding.

Using Model Metadata - In our Person class, the PersonId property is one that we don't want the user to be able to see or edit. We can use the HiddenInput attribute, which causes the helper to render a hidden input field, Example: public class Person { [**HiddenInput**] public int PersonId { get; set; } public string FirstName { get; set; } When this attribute has been applied, the Html.EditorFor and Html.EditorForModel helpers will render a read-only view of the decorated property along with a hidden input field. If we want to hide a property entirely, then we can set the value of the DisplayValue property in the DisplayName attribute to false as [HiddenInput(DisplayValue=false)] Here when we use the Html.EditorForModel helper on a Person object, a hidden input will be created so that the value for the PersonId property will be included in any form submissions.

Excluding a property from scaffolding - If we want to exclude a property from the generated HTML use [ScaffoldColumn (false)] .

Using Metadata for Labels - By default, the Label, LabelFor, LabelForModel, and EditorForModel helpers use the names of properties as the content for the label elements they generate. Of course, the names we give to our properties are often not what we want to be displayed to the user. To that end, we can apply the DisplayName attribute from the System.ComponentModel namespace, passing in the value we want as a parameter as [Display(Name="Date of Birth")] When the label helpers render a label element for the BirthDate property, they will detect the Display attribute and use the value of the Name parameter for the inner text, like this: <label for="BirthDate">Date of Birth</label>. Also an attribute called the **DisplayName** can be applied to classes, which allows us to use the Html.LabelForModel helper. We can decorate this attribute on top of the class as [DisplayName("Person Details")] public class Person { } . This puts a label on top for the entire form.

Using Metadata for Data Values - We can also use metadata to provide instructions about how a model property should be displayed. For example: [DataType(DataType.Date)] public DateTime BirthDate { get; set; }. In the example we have specified the DataType.Date value, which causes the templated helpers to render the value of the BirthDate property as a date without the associated time. See book for useful values of the DataType enumeration.

Using Metadata to Select a Display Template - As their name suggests, templated helpers use display templates to generate HTML. The template that is used is based on the type of the property being processed and the kind of helper being used. We can use the UIHint attribute to specify the template we want to use to render HTML for a property as [UIHint("MultilineText")] public string FirstName { get; set; }. This renders an HTML textarea element for the FirstName property when used with one of the editor helpers, such as EditorFor or EditorForModel. For a list of useful view templates refer book. Care must be taken when using the UIHint attribute. We will receive an exception if we select a template that cannot operate on the type of the property we have applied it to—for example, applying the Boolean template to a string property. The Object template is a special case—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 selects the most suitable template for the property type. The Object template takes metadata such as the UIHint and DataType attributes into account.

Applying Metadata to a Buddy Class: It isn't always possible to apply metadata to an entity model class. This is usually the case when the model classes are generated automatically, like sometimes with ORM tools such as the Entity Framework. The solution to this problem is to ensure that the model class is defined as partial and to create a second partial class that contains the metadata. This second partial class would be an empty class and will simple contain the **MetadataType** attribute as **[MetadataType(typeof(PersonMetadataSource))]** public partial class Person { }. Now define the metadata for the Person class in a class called PersonMetadataSource as
class PersonMetadataSource { [HiddenInput(DisplayValue=false)] public int PersonId { get; set; } } This buddy class only needs to contain properties that we want to apply metadata to—we don't have to replicate all of the properties of the Person class, for example.

Working with Complex Type Parameters - The templating process relies on the Object template that we described in the previous section. Each property is inspected, and a template is selected to render HTML to represent the property and its data value. But the Object template operates only on simple types such as int, string, bool, DateTime etc. Complex types such as Address within our Person type will be ignored. If we want to render HTML for a complex property, we have to do it explicitly, as @Html.EditorFor(m => m.HomeAddress). The HomeAddress property is typed to return an Address object, and we can apply all of the same metadata to the Address class as we did to the Person class.

Customizing the Templated View Helper System - There are some advanced options that let us customize the template helpers entirely.
Creating a Custom Editor Template - One of the easiest ways of customizing the templated helpers is to create a custom template. For example, let us say our Person class contains an Enum property called Role which can take the following values: Admin, User, and Guest. If we put a template helper such as @Html.EditorFor(m => m.Role), this is going to display a simple textbox which is not what we want. Instead, we are going to create a custom editor view, which in essence means creating a strongly typed partial view called Role.cshtml (with model as Role) in a particular location called EditorTemplates in the ~/Views/Shared folder of our project. Here we can freely create our partial view using a mix of static HTML elements and Razor tags. Now after this once we use Html.EditorForModel helper (actually it doesn't matter which of the built-in editor helpers we use)—they will all find and use our Role.cshtml template.

Creating a Custom Display Template - The process for creating a custom display template is similar to that for creating a custom editor, except that we place the templates in the DisplayTemplates folder.

Creating a Generic Template - We are not limited to creating type-specific templates. We can, for example, create a template that works for all enumerations and then specify that this template be selected using the **UIHint** attribute. Thus we can define a template called Enum.cshtml in the ~/Views/Shared/EditorTemplates folder. This template is a more general treatment for C# enumerations. For example:
[UIHint("Enum")] public Role Role { get; set; }

Replacing the Built-in Templates - If we create a custom template that has the same name as one of the built-in templates, the MVC Framework will use the custom version in preference to the built-in one. For example we could provide a custom template for bool?. Here in this case the custom template will always be given preference when we call @Html.EditorFor(m => m.IsCertified) which is a Boolean? property of our Person object. For this bool? (nullable bool type) we define the template's model using @model bool?.

Using the ViewData.TemplateInfo Property - The MVC Framework provides the **ViewData.TemplateInfo** property to make writing custom view templates easier. The property returns a **TemplateInfo** object. The most useful members of TemplateInfo are: FormattedModelValue, GetFullHtmlFieldId(), GetFullHmlFieldName(), HtmlFieldPrefix.

Respecting Data Formatting - Perhaps the most useful of the TemplateInfo properties is FormattedModelValue, which allows us to respect formatting metadata without having to detect and process the attributes ourselves. For example, a custom template called DateTime.cshtml that generates an editor for DateTime objects is specified using **@Html.TextBox("", ViewData.TemplateInfo.FormattedModelValue).** If we apply the DataType attribute to the BirthDate property from our Person model class, like this: **[DataType(DataType.Date)]** public DateTime BirthDate { get; set; } then the value returned from the FormattedModelValue property will be just the date component of the DateTime value.

Passing Additional Metadata to a Template - On occasions, we want to provide additional information to our templates that we cannot express using the built-in attributes. We can do this using the AdditionalMetadata attribute. For example: [**AdditionalMetadata("RenderList", "true")]**

`public bool IsApproved { get; set; }` . This attribute takes key/value parameters for the information we want to pass along. We detect these values through the ViewData property in our template, as the editor template Boolean.cshtml as:

```
if(ViewData.ModelMetadata.AdditionalValues.ContainsKey("RenderList")) {
        renderList = bool.Parse(ViewData.ModelMetadata.AdditionalValues["RenderList"].ToString());
}
```

It is important not to assume that the additional values you are checking for are present. This is especially true when replacing a built-in template, because it can be hard to determine in advance which model objects your template will be used for.

Understanding the Metadata Provider System - The metadata examples we have shown you so far have relied on the **DataAnnotationsModelMetadataProvider** class, which is the class that has been detecting and processing the attributes we have been adding to our classes so that the templates and formatting options are taken care of. Underlying the model metadata system is the **ModelMetadata** class, which contains a number of properties that specify how a model or property should be rendered. The DataAnnotationsModelMetadata processes the attributes we have applied and sets value for the properties in a ModelMetadata object, which is then passed to the template system for processing.See full list of attributes in book.

Creating a Custom Model Metadata Provider - We can create a custom model metadata provider if the data annotations system doesn't suit our needs. Providers must be derived from the abstract ModelMetadataProvider class. Here we can implement the GetMetadataForProperties(), GetMetadataForProperty(), GetMetadataForType() methods. A simpler approach is to derive a class from AssociatedMetadataProvider, which takes care of the reflection for us and requires us only to implement a single method called CreateMetadata().The recommended way to implement a custom policy is to take advantage of the data annotations attributes, so we have to derive our custom metadata provider from DataAnnotationsModelMetadataProvider. This class is derived from AssociatedMetadataProvider, so we only have to override the CreateMetadata method.(See useful example in book).

CH 17 – Model Binding

Model binding is the process of creating .NET objects using the data sent by the browser in an HTTP request. We have been relying on the model binding process each time we have defined an action method that takes a parameter—the parameter objects are created by model binding.

Understanding Model Binding - Imagine that we have defined an action method in a controller as public ViewResult Person(int id) { .. }. Suppose that our action method is defined in the HomeController class, which means the default route that Visual Studio creates for us will let us invoke our action method. When we receive a request for a URL such as /Home/Person/23, the MVC Framework has to map the details of the request in such a way that it can pass appropriate values or objects as parameters to our action method. The action invoker, the component that invokes action methods, is responsible for obtaining values for parameters before it can invoke the action method. The default action invoker, ControllerActionInvoker relies on model binders, which are defined by the IModelBinder interface given below:

public interface IModelBinder { object BindModel(ControllerContext controllerContext, ModelBindingContext bindingContext);}

There can be multiple model binders in an MVC application, and each binder can be responsible for binding one or more model types. When the action invoker needs to call an action method, it looks at the parameters that the method defines and finds the responsible model binder for each parameter type. In our case, the action invoker would find that the action method had one ***int*** parameter, so it would locate the binder responsible for binding ***int*** values and call its BindModel method. If there is no binder that will operate on ***int*** values, then the default model binder is used. **A model binder is responsible for generating suitable action method parameter values**, and this usually means transforming some element of the request data (such as form or query string values), but the MVC Framework doesn't put any limits on how the data is obtained.

Using the Default Model Binder - Although an application can have multiple binders, most just rely on the built-in binder class, DefaultModelBinder. This is the binder that is used by the action invoker when it can't find a custom binder to bind the type. By default, this model binder searches four locations for data matching the name of the parameter being bound. The order is 1. Request.Form 2. RouteData.Values 3. Request.QueryString 4. Request.Files . In the case of the above action method shown in the DefaultModelBinder class examines our action method and finds that there is one parameter called id. The form data and route data values are searched, but since a routing segment with the name id will be found in the second location, the query string and uploaded files (Request.Files) will not be searched at all. You can see how important the name of the action method parameter is—the name of the parameter and the name of the request data item must match in order for the DefaultModelBinder class to find and use the data.

Binding to Simple Types - When dealing with simple parameter types, the DefaultModelBinder tries to convert the string value that has been obtained from the request data into the parameter type using the System.ComponentModel.TypeDescriptor class. If the value cannot be converted—for example, if we have supplied a value of "apple" for a parameter that requires an int value—then the DefaultModelBinder won't be able to bind to the model. We can always make our parameter optional by supplying a default value to be used when there is no data available, like this: public ViewResult RegisterPerson(int id = 23) { …}

CULTURE-SENSITIVE PARSING - The DefaultModelBinder class uses different culture settings to perform type conversions from different areas of the request data. The values obtained from 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. A date value won't be converted if it isn't in the right format. This means we must make sure that all dates included in the URL are expressed in the universal format (yyyy-mm-dd). We must also be careful when processing date values that users provide. The default binder assumes that the user will express dates using the format of the server culture, something that is unlikely to always happen in an MVC application that has international users.

Binding to Complex Types **-** When the action method parameter is a complex type (in other words, any type that cannot be converted using the TypeConverter class), then the DefaultModelBinder class uses reflection to obtain the set of public properties and then binds to each of them in turn. For example, when we call @Html.EditorFor(m => m.HomeAddress.Line1) ; here the default model binder checks the class properties to see whether they are simple types, since here it is HomeAddress which is a complex type, the process is repeated for this type; the set of public properties are obtained, and the binder tries to find values for all of them. Here it finds a match for Line1 since it is property of HomeAddress and it is a simple type.

Binding to Arrays and Collections - One elegant feature of the default model binder is how it deals with multiple data items that have the same name. For example if our view contains the following, @using (Html.BeginForm()) { @Html.TextBox("movies") @Html.TextBox("movies") @Html.TextBox("movies") <input type=submit /> }. We have used the Html.TextBox helper to create three input elements. We can receive the values that the user enters with an action method like [HttpPost] public ViewResult Movies(**List<string> movies**) { …}. The model binder will find all the values supplied by the user and pass them to the Movies action method in a List<string>.

Binding to Collections of Custom Types - The multiple-value binding trick is very nice, but if we want it to work on custom types, then we have to produce HTML in a certain format. Listing below shows how we can do this with an array of Person objects.

```
@model List<MvcApp.Models.Person>
@for (int i = 0; i < Model.Count; i++) {
        <h4>Person Number: @i</h4>
        @:First Name: @Html.EditorFor(m => m[i].FirstName)
        @:Last Name: @Html.EditorFor(m => m[i].LastName)
}
```

Here the templated helper generates HTML that prefixes the name of each property with the index of the object in the collection as: <input class="text-box single-line" name="[0].FirstName" type="text" etc. To bind to this data, we just have to define an action method that takes a collection parameter of the view model type, as public ViewResult Register(List<Person> people) {..}.

Binding to Collections with Nonsequential Indices - An alternative to sequential numeric index values is to use arbitrary string keys to define collection items. To use this option, we need to define a hidden input element called index that specifies the key for the item, as <input type="hidden" name="index" value="firstPerson"/> First Name: @Html.TextBox("[firstPerson].FirstName") and <input type="hidden" name="index"

value="secondPerson"/>First Name: @Html.TextBox("[secondPerson].FirstName") etc. We have prefixed the names of the input elements to match the value of the hidden index element. The model binder detects the index and uses it to associate data values together during the binding process.

Binding to a Dictionary - The default model binder is capable of binding to a dictionary, but only if we follow a very specific naming sequence as <input type="hidden" name="[0].key" value="firstPerson"/> First Name: @Html.TextBox("[0].value.FirstName") Last Name: @Html.TextBox("[0].value.LastName") and <input type="hidden" name="[1].key" value="secondPerson"/> First Name: @Html.TextBox("[1].value.FirstName") Last Name: @Html.TextBox("[1].value.LastName") etc. We can receive the data with an action method like this one: [HttpPost] public ViewResult Register(IDictionary<string, Person> people) {..}

Manually Invoking Model Binding - The model binding process is performed automatically when an action method defines parameters, but we can take direct control of the process if we want. This gives us more explicit control over how model objects are instantiated, where data values are obtained from, and how data parsing errors are handled. For example: [HttpPost] public ActionResult RegisterMember() { Person myPerson = new Person(); **UpdateModel(myPerson);** return View(myPerson); }. The UpdateModel (it is actually derived from base controller) method takes a model object we previously created as a parameter and tries to obtain values for its public properties using the standard binding process. One reason for invoking model binding manually is to support dependency injection (DI) in model objects. If we used dependency injection we can alternatively call as UpdateModel((Person)DependencyResolver.Current.GetService(typeof(Person)));

Restricting Binding to a Specific Data Source - When we manually invoke the binding process, we can restrict the binding process to a single source of data. By default, the binder looks in four places: form data, route data, the query string, and any uploaded files. Let us see how we can restrict the binder to search for data in a single location—in this case, the form data. For example: we can say UpdateModel(myPerson, **new FormValueProvider(ControllerContext)**); This version of the UpdateModel method takes an implementation of the IValueProvider interface which becomes the sole source of data values for the binding process. Another neat mechanism is to use the FormCollection class that implements the IValueProvider interface, and if we define the action method to take a parameter of this type, the model binder will provide us with an object that we can pass directly to the UpdateModel method. For example: public ActionResult RegisterMember(**FormCollection** formData) {…UpdateModel(myPerson, formData);…}

Dealing with Binding Errors - Users will inevitably supply values that cannot be bound to the corresponding model properties— invalid dates, or text for numeric values, for example. When we invoke model binding explicitly, we are responsible for dealing with any such errors. The model binder expresses binding errors by throwing an InvalidOperationException. Details of the errors can be found through the ModelState feature. As an alternative approach, we can use the TryUpdateModel method, which returns true if the model binding process is successful and false if there are errors, as if (**TryUpdateModel**(myPerson, formData)) { //...proceed as normal } else { //...provide UI feedback based on ModelState }. The only reason to favor TryUpdateModel over UpdateModel is if you don’t like catching and dealing with exceptions; there is no functional difference in the model binding process.

Using Model Binding to Receive File Uploads - All we have to do to receive uploaded files is to define an action method that takes a parameter of the HttpPostedFileBase type. The model binder will populate it with the data corresponding to an uploaded file. Also it is important to set the value of the enctype attribute to multipart/form-data. If we don't do this, the browser will just send the name of the file and not the file itself. Example: <form action="@Url.Action("Upload")" method="post" enctype="multipart/form-data">

Project Essentials

Take care of the following:

1. Dependency Injection container like Ninject:
2. Unit Testing mechanisms like Moq for:
    a. Unit testing Controllers
    b. Unit Testing Routing (see CH 11).
    c. Unit Testing of Default Routing values. (see CH 11).
    d. Unit Testing of Static segments if required. (see CH 11).
    e. Unit Testing of Custom segment variables, CatchAll and Optional Arguments if required. (see CH 11).
    f. Unit Testing of Route Constraints
    g. UNIT TEST: TESTING OUTGOING URLS (use UrlHelper.GenerateUrl())
    a. UNIT TESTING CONTROLLERS AND ACTIONS (important) To test the view that an action method renders, you can inspect the ViewResult object that it returns. This is not quite the same thing—after all, you are not following the process through to check the final HTML that is generated—but it is close enough, as long as you have reasonable confidence that the MVC Framework view system works properly.
    b. UNIT TEST: VIEW MODEL OBJECTS

3. Website Security mechanism
    a. Good link: Implement secure ASP.NET MVC applications: http://www.codeproject.com/Articles/288631/Secure-ASP-NET-MVC3-applications
4. Localization of ASP.NET MVC
5. Themes in ASP.NET MVC
6. Create areas with unique route names. Register Area routes before top level routes (this is default behavior anyway).
7. From Action Methods it is always recommended to return the specific object such as ViewResult, PartialViewResult etc than returning the base ActionResult.
8. It is always a good practice to specify a BaseController class for our project and specify any base filters (authorization, exception filters) there.
9. Explore MEF at http://msdn.microsoft.com/en-us/magazine/ee291628.aspx
10. Use HtmlHelpers for view specific extensions.
11. Try to use Model Templates especially EditorForModel() kind of templates.
    a. Customize the Templated View Helper System if required – Good Idea.

12. Try Multiple Deployment platforms with PhoneGap:
    a. http://msdn.microsoft.com/en-us/magazine/hh975345.aspx
    b. http://www.codeproject.com/Articles/445361/Property-Finder-a-Cross-Platform-HTML5-Mobile-App
    c.

Useful Links

MongoDB and .NET

http://www.primaryobjects.com/cms/article137.aspx

http://blog.endjin.com/2010/12/a-step-by-step-guide-to-mongodb-for-net-developers/