1. **MVC:**
   1. **Model:** Set of classes that describes the data we are working with and the business rules for how the data can be changed and manipulated
   2. **View:** Applications UI
   3. **Controller:** handles communication from the user, overall application flow and application-specific logic
2. **MVC features history:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **MVC1** | **MVC2** | **MVC3** | **MVC4** | **MVC5** | **MVC6** |
|  | UI helpers | Razor view engine | ASP.NET Web API | One ASP.NET |  |
|  | Scaffolding (automatic code generation) | Data annotations | Enhancements to default project templates | New web project experience |  |
|  | Attribute based model validation | Improved model validation | Mobile project template using jQuery mobile | ASP.NET identity |  |
|  | Strongly typed HTML helpers | Dependency resolution | Display modes | Bootstrap templates |  |
|  | Asynchronous controllers | Global action filters | Task support for asynchronous controllers | Attribute routing |  |
|  | Rendering subsections of a page | Unobstructive javascript | Bundling and minification | ASP.NET scaffolding |  |
|  |  | jQuery validation |  | Authentication filters |  |
|  |  | JSON binding |  | Filter overrides |  |
|  |  | NuGet |  |  |  |

1. **Important MVC features:**
   1. **One ASP.NET:** This eliminates the choice of choosing MVC or web forms before starting a project. You can add MVC to your project at any time.
   2. **ASP.NET identity system:** This facilitates membership login to MVC application. MVC5 has simplified this identity system so that it suits with all ASP.NET web applications (web forms, web pages, MVC applications, web API, SignalR etc)
   3. **Attribute routing:** To specify routes by placing annotations on your controller classes or action methods.
   4. **ASP.NET Scaffolding:** used to generate boilerplate code based on your model classes.
   5. **Authentication filters:** To control access by role or other custom logic. Difference between authentication and authorization
   6. **Filter override:** Filter overrides mean that you can exclude a controller or action from executing a global filter.
2. **Installing MVC 5 and creating applications:**
   1. **Requirements:**
      1. .NET 4.5
   2. Project templates
      1. Empty
      2. Web forms
      3. MVC
      4. Web API
      5. Single page application
      6. Facebook
      7. Azure mobile service
   3. Folder and core references
      1. Web forms
      2. MVC
      3. Web API
   4. Enabling unit testing
   5. Configuring authentication
      1. No authentication 🡪 for public websites
      2. Individual user accounts
      3. Organizational accounts 🡪 Used for accounts that authenticate via some form of Active Directory.
      4. Windows authentication 🡪 For intranet applications
3. **MVC Application structure:**

|  |  |  |
| --- | --- | --- |
| **S.NO** | **Directory** | **Description** |
| 1 | /Controllers |  |
| 2 | /Models |  |
| 3 | /Views |  |
| 4 | /Scripts | To store javascript files |
| 5 | /fonts | To store bootstrap fonts |
| 6 | /Content | CSS, images and other project files |
| 7 | /App\_Data | Data files we read or write |
| 8 | /App\_Start | Configuration files |

The above folder structure is not a mandatory or required one. It’s just a convention. Big projects split code among different projects to provide more flexibility.

1. **ASP.NET MVC and conventions:**
   1. **Controllers** ends with Controller suffix and are stored in Controllers directory
   2. **Views:** single views directory for all views of your application with controller as the folder name. **Ex:** ProductController has a corresponding view in /Views/Product directory.

**Chapter 2: Controllers**

1. **Controllers:** Controllers in MVC pattern are responsible for responding to user input, often making changes to the model in response to user input. In this way, Controllers are concerned with the flow of the application, working with data coming in, and providing data going out to the relevant view.

Chapter 3: Views

1. Supplying different view

*Public ActionResult About(){*

*return View(“MyNewAboutUs”);*

*}*

*Public ActionResult About(){*

*return View(“~/Views/Example/AboutUs.cshtml”);*

*}*

1. Strongly typed views:
   1. Strongly typed views allow us to set a Model type for a view.
   2. This allows us to send model object from controller to the view that’s strongly typed on both the ends, so we get the benefit of intellisense, compiler checking etc.
   3. We specify the model in the controller by the overloaded method of view()

**Method 1:**

In controller:

Public ActionResult List(){

var albums = new List<Album>();

for(int i=0;i<10;i++){

albums.Add(new Album{Title=”Album ”+i});

}

**return view(albums);**

}

In View:

@model IEnumerable<MVCMusicStore.Models.Album>

<ul>

@foreach(Album a in **Model**){

<li>@a.Title</li>

}

</ul>

**Method 2:**

In views we can use @using namespace to avoid fully qualified name

@using MVCMusicStore.Models

@model IEnumerable<Album>

**Method 3:**

Declare the name spaces which you often use in several views in web.config file within the views directory.

*<add namespace=”MVCMusicStore.Models”/>*

1. ViewBag, ViewData and ViewDatadictionary
   1. Both ViewBag and strongly typed models are passed to the view via ViewDataDictionary
   2. Technically all data is passed to the view from controller via a ViewDataDictionary(a specialized dictionary class) called ViewData.

ViewData[“CurrentTime”]=Datetime.Now;

* 1. ViewBag is a dynamic wrapper around ViewData.

ViewBag.CurrentTime=DateTime.Now;

Is same as

ViewData[“CurrentTime”]=DateTime.Now;

1. ViewModel (View soecific model)
   1. ViewModels are models designed to cater the requirement of a view usually made by combining more than one model with normal data types or even other models.

**Model:**

Public class ShoppingCartViewModel{

Public IEnumerable<Product> products {get;set;}

Public decimal CartTotal {get; set;}

Public string Message{get;set;}

}

**Controller:**

Public ActionResult Index(){

var products = new List<Product>();

for(int i=0; i<10 ; i++){

products.Add(new Product{Title=”Product ”+i, Price=1.13M\*i});

}

Var model=new ShoppingCartViewModel{

Products = products,

CartTotal=products.sum(p=>p.price),

Message= “Thanks for your business!”

};

return View(model);

}

**View:**

@model ShoppingCartViewModel

1. Adding a view:
   1. Rt.click in ActionMethod 🡪 Add View
   2. Templates: Create, delete, details, edit, empty, empty(without model), list
   3. Reference script libraries: inserts jQuery validation, unobtrusive jQuery validation library
   4. Layout page: By default the layout page is set in \_ViewStart.cshtml file. We can use this option to change the layout.
2. Razor view engine:

|  |  |  |  |
| --- | --- | --- | --- |
| Razor | Syntax | Description | Example |
| Implicit code expression | @ |  | @items.Length |
| Explicit code expression | @() |  | @(rootNamespace).Models |
| Escape sequence | @@ |  | I am going @@9:00 AM  Prints  Iam going @9:00 AM |
| HTML encoding |  | Razor expressions are automatically HTML encoded |  |
| Server side comment | @\*  \*@ | This is not printed to the HTML | @\*  I am a comment and am not visible in HTML (even in view page source)  \*@ |
|  |  |  |  |

1. Layouts
   1. Layouts are the template of a page which views use

Example:

**SiteLayout.cshtml**

<html>

<head><title**>@ViewBag.Title**</title></head>

<body>

<h1**>@ViewBag.Title**</h1>

<div class=”main-content**”>@RenderBody()**</div>

</body>

</html>

In the place of @RenderBody() the content in the views that use the layout will be replaced.

**Index.cshtml (View)**

@\*Specifying layout in the view\*@

@{

**Layout = “~Views/Shared/siteLayout.cshtml”**

ViewBag.Title =”Index”

}

* 1. A layout may have multiple sections. These sections are to be provided by the views that use this layout. We can make it optional by specifying required:false attribute to the @renderSection method.

**Example:**

**SiteLayout.cshtml**

<html>

<head><title**>@ViewBag.Title**</title></head>

<body>

<h1**>@ViewBag.Title**</h1>

<div class=”main-content**”>@RenderBody()**</div>

<div class=”footer”>**@RenderSection(“Footer”)**</div>

</body>

</html>

**Index.cshtml (View):**

@{

**Layout =”~Views/Shared/SiteLayout.cshtml”**

}

<p>this goes in place of @RenderBody()</p>

**@section Footer{**

<p>This is the footer supplied from Index</p>

}

**Note:** If we do not specify Footer section in the view, it will result in an exception stating that the section named “Footer” was not defined.

Optionally you can mention reuired:false attribute to the @RenderSection method to avoid this.

**Example:**

<div class=”footer”>@RenderSection(“Footer”, reuired:false)</div>

You can also mention default section if the section is not found in the included view as below

<div class=”footer”>

@if(IsSectionDefined(“Footer”)){

RenderSection(“Footer”);

}else{

<p>This is a default footer from SiteLayout.cshtml</p>

}

</div>

1. ViewStart
   1. ViewStart is a file which is placed in views directory or in sub directories of Views directory to execute common code.
   2. This is used to include common @Layout for set of views in the directory and in the subdirectories.

**Example:**

**\_ViewStart.cshtml**

@{

**Layout =”~Views/Shared/\_Layout.cshtml”**

}

* 1. Wherever it is placed the views in the directory and sub directories get the layout \_Layout.cshtml.
  2. Each view can override this by specifying the layout inside the view itself.

1. PartialView
   1. A partial view is just like a view which is rendered by the action method to update a part of the page.
   2. The main difference between a view and a partial view is that PartialView does not specify Layout. Even if the layout is specified in \_ViewStart.cshtml it is ignored.
   3. Action method calls partial view through **PartialView()** method.

Example:

Public ActionResult Message(){

ViewBag.Message=”this is a partial view!”;

return **PartialView()**;

}

* 1. Rendering partial view through

<div id=”result”></div>

@section scripts{

<script type=”text/javascript”>

$(function(){

**$(‘#result’).load(‘/home/message’);**

});

</script>

}

Chapter 4: Models

1. Scaffolding: Scaffolding can generate boilerplate code to create, read, update and delete (CRUD) functionality in an application. It can examine type definition for a model and then generate a controller, the controller’s associated views and in some cases data access classes as well.

Chapter 5: Forms and HTML Helpers

1. HTML helpers are useful to write HTML code (tags) wherein we maintain coordination between the model and the view that is used in the form; they help in giving correct names to the HTML elements.

Chapter 6: Data Annotations and Validation

1. Data annotations are stored in System.ComponentModel.DataAnnotations class.
2. Data validations

|  |  |  |  |
| --- | --- | --- | --- |
| S.NO | Attribute | Description | Example |
| 1 | Required | To make a property mandatory | [Required]  public string LastName { get; set; } |
| 2 | StringLength | To specify restrictions on a property value’s length | [StringLength(50)]  public string FirstName { get; set; } |
|  | MinimumLength | To specify minimum length for a string | [StringLength(50,MinimumLength =3)]  public string FirstName { get; set; } |
|  | ReglarExpression | To validate the string acc to thee given format | [RegularExpression(@"[A-Za-z0-9.\_%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}")]  public string Email { get; set; } |
|  | Range | To specify min and max values to the property | [Range(35,44)]  public int Age { get; set; } |
|  |  | To specify the type | [Range(typeof(decimal), "0.00","0.50")]  public decimal Total { get; set; } |
|  | Compare | To compare two properties of a model object | **[Compare("Email")]**  public string EmailConfirm { get; set; } |
|  | Remote | TO perform client-side validation with a server call-back.  In System.Web.MVC | **[Remote(“CheckUserName”,”Account”)]**  **Public string UserName{ get ; set;}** |
|  | ErrorMessage  (Parameter) | To specify custom error message | [StringLength(50,MinimumLength =3,ErrorMessage ="Minimum length for your first name is 3")]  public string FirstName { get; set; } |
| Display annotations | | | |
|  | Display | To display friendly name for a property | public int OrderId { get; set; }  [Display(Name ="Orer Date")] |
|  |  | Order attribute | [Display(Name ="Orer Date",Order =5)]  public DateTime OrderDate { get; set; } |
|  | ScaffoldColumn | Enables or disable column display in the View | [ScaffoldColumn(false)] |
|  | DisplayFormat | To format the input in the view | [DisplayFormat(ApplyFormatInEditMode=true, DataFormatString="{0:c}")]  public decimal Total { get; set; } |
|  | ReadOnly | To make a field read only | [ReadOnly(true)]  public decimal Total { get; set; } |
|  | DataType | To specify the data type of a property | [DataType(DataType.Password)]  public string Password { get; set; } |
|  | UIHint | To give the ASP.NET MVC runtime the name of a template to use when rendering output with the template helpers(such as Displayfor and EditorFor) |  |
|  | HiddenInput | To generate hidden input tag |  |

1. **Localization of error messages:** If we provide error messages through ErrorMessage attribute of validations we can’t achieve localization.
2. **How does it work?**
   1. The **model binder** runs in the controller when the form is submitted.
   2. When Model binder runs it updates the model properties with the new values.
   3. The model builder uses the current models metadata and obtains all the validators for the model.
   4. The MVC runtime provides a validator to work with data annotations
   5. This model validator can find all the validation attributes and execute the validation logic inside.
   6. The model binder catches all the failed validation rules and places them into model state.
   7. When the model is submitted to the Controller’s action it will check if the ModelState is valid or not. If yes we will do the necessary logic. Otherwise we return the same view with the same model to show the user the errors in the form.
3. Custom validation logic:
   1. Custom validation can be done in two ways
      1. Packaging validation logic into a custom data annotation
      2. Packaging validation logic into a model object itself
   2. All the validations annotations are derived from **ValidationAttribute** base class in System.ComponentModel.DataAnnotations namespace.
   3. To implement validation logic we need to override the base class’s IsValid method.

Code:

public class MaxWordsAttribute: ValidationAttribute

{

private readonly int \_maxWords; //private variable to which max words count is set

//constructor

public MaxWordsAttribute(int maxWords):base("Too many words for {0}") {//This constructor initiate max words permitted and default error

\_maxWords = maxWords;

}

protected override ValidationResult IsValid(object value, ValidationContext validationContext)

{

if (value != null) {

string Name = value.ToString();

if (Name.Split(' ').Length > \_maxWords) {

var errorMessage = FormatErrorMessage(validationContext.DisplayName);

return new ValidationResult(errorMessage);

}

}

return ValidationResult.Success;

}

}

Explanation:

* ValidationAttribute is the base class
* We override the IsValid method of the base class by passing the value and ValidationContext.
* ValidationContext gives access to model type, model object instance, and friendly display name of the property you are validating.
* FormatErrorMessage formats the error message with the display name

**IValidatableObject:**

public IEnumerable<ValidationResult> Validate(

ValidationContext validationContext)

{

if (LastName != null &&

LastName.Split(' ').Length > 10)

{

yield return new ValidationResult("The last name has too many words!",

new[] { "LastName" });

}

}

Chapter 7: Membership, authorization and security

1. Authentication Vs Authorization:
   1. Authentication: Authentication is the one through which user says this who I am through some form of login mechanism (username and password or OpenID or OAuth)
   2. Authorization: Authorization is through which the user is verified if he can do something what he wants through role-based or claim-based system
2. **Authorize attribute**

Authorizing an action method

[Authorize]

Public ActionResult Index(){

Return View();

}

Authorizing a controller:

[Authorize]

public class CheckoutController : Controller

Authorizing an entire application:

Add global filter in App\_start/Filterconfig.cs

public static void RegisterGlobalFilters(GlobalFilterCollection filters) {

**filters.Add(new System.Web.Mvc.AuthorizeAttribute());**

filters.Add(new HandleErrorAttribute());

}

You can use **AllowAnonymous** attribute to allow some action or controller to show pages like home and aboutus.

**Chapter 7 is not completed**

Chapter 8: AJAX

1. ASP.NET MVC framework supports AJAX through jQuery
2. jQuery is pre built into our project in scripts folder.
3. Theory about jQuery and its support for AJAX
4. **Unobtrusive javascript:** Unobtrusive javascript is a practice of keeping javascript code separate from markup. Advantages of unobtrusive javascript include improved performance as browser can cache script files and progressive enhancement.
5. We place custom scripts in /Scripts/App directory
6. Razor’s script section helps to render scripts which are page specific only after jQuery reference is included into the page.

@section scripts{

<script src=”~/scripts/App/MusicScripts.js”></script>

}

1. Some pre included files:
   1. Bootstrap.js
   2. Respond.js 🡪 used by bootstrap.js to make older browsers understand advanced features like HTML5
   3. Modernizr.js 🡪 helps you build modern applications by modernizing older browsers
2. AJAX Helpers:
   1. Ajax helpers reduce the asynchrony work that we need to do like writing own jQuery code of posting data and displaying the output.
   2. To use Ajax helpers we need to install the **jquery.unobtrusive-ajax.js** file in the project and add script reference in the view.

Example:

View

<div id="DailyDeal">

@Ajax.ActionLink("Click here to know today's deal!",

"DailyDeal",null,

new AjaxOptions {

UpdateTargetId="DailyDeal",

InsertionMode=InsertionMode.Replace,

AllowCache=true,

HttpMethod="get"

},

new { @class="btn btn-primary"})

</div>

Controller:

public ActionResult DailyDeal() {

var album = GetDailyDeal();

return PartialView("\_DailyDeal",album);

}

private Album GetDailyDeal() {

var musicStoreDb = new MusicStoreDB();

var album = musicStoreDb.Albums.OrderBy(a => a.Price).First();

album.Price \*= 0.5m;

return album;

}

Partial view(\_DailyDeal.cshtml)

@model MVCMusicStore.Models.Album

<div>

@Html.DisplayNameFor(model => model.Artist.Name)

</div>

1. **BeginForm example**

<div id="AlbumSearchResult"></div>

<div>

@using (@Ajax.BeginForm("AlbumSearch", new AjaxOptions

{

HttpMethod = "get",

InsertionMode = InsertionMode.Replace,

LoadingElementId = "AjaxLoad",

UpdateTargetId = "AlbumSearchResult",

OnFailure = "AlbumSearchFailure"

})) {

<input type="text" name="album"/>

<input type="submit" value="search"/>

<img src="~/Content/loading.gif" id="AjaxLoad" />

}

</div>

1. **Client side validation of custom data validation attribute**
   1. In general custom data validations are validated only at server side. To make them client side validated the attribute need to implement the interface **IClientValidatable**. This interface has a method called **GetClientValidationrules** which is to be implemented by the custom attribute**.**
   2. For more description on this see Professional ASP.NET MVC 5 (pg.no: 236-241)
2. **jQuery UI**
   1. A javascript UI to tweak the UI and to give some widgets
   2. Add it through NuGet package manager
   3. jQueryUI auto complete

**View:**

<input type="text" name="album" data-autocomplete-source="@Url.Action("QuickSearch", "Home")" />

**Js:**

$("input[data-autocomplete-source]").each(function () {

var target = $(this);

target.autocomplete({ source: target.attr("data-autocomplete-source") });

});

**Controller:**

public JsonResult QuickSearch(string term) {

var musicStoreDB = new MusicStoreDB();

var albums = musicStoreDB.Albums.Where(a => a.Title.Contains(term)).

ToList().

Select(a => new { value = a.Title });

return Json(albums,JsonRequestBehavior.AllowGet);

}

**Note:** JsonRequestBehavior.AllowGet is necessary as by default .NET MVC does not allow to send json response on HTTP GET request. There is to take care when sending sensitive data in json format from **json hijacking.**

1. **Improving AJAX performance**
   1. YSlow for firefox helps in checking where we can improve performance
   2. Bundling and minification features
      1. To see the effect of bundling and minification set debug=”false” in web.config file.
   3. BundleConfig.cs
   4. Including bundles in the view
      1. @Scripts.Render(“~/Scripts/jQuery”)
      2. @Styles.Render(“~Styles/css”)

Chapter 9: Routing

1. Routing in ASP.NET MVC framework serves two main purposes
   1. It matches the incoming requests which does not match a file on the file system to a controller action
   2. It constructs outgoing URL that corresponds to controller action
2. URL rewriting vs routing: URL rewriting just concentrates on mapping one URL to a friendly readable URL. Routing additionally maps the URL to a resource.
3. With routing it does not mean which file to fetch, it means which piece of code to execute for incoming URL request.
4. Routing types in ASP.MET MVC
   1. Traditional routing: which ASP.NET supports by default
   2. Attribute routing: which is introduced in MVC5 using declarative attributes on controllers and actions.
5. **Attribute routing:**
   1. In attribute routing we specify what action to be executed based on incoming URL by placing route attribute on actions and controllers.
   2. For this to work you have to replace the content of RegisterRoutes method of /App\_start/RouteConfig.cs as below

public static void RegisterRoutes(RouteCollection routes)

{

**routes.MapMvcAttributeRoutes();**

}

After this you will place route attributes on actions and controllers.

* 1. Examples:

To match single route template:

[Route("About")]

public ActionResult About()

{

ViewBag.Message = "Your application description page.";

return View();

}

To match multiple route templates:

[Route("")]

[Route("Home")]

[Route("Home/Index")]

public ActionResult Index()

{

return View();

}

Specifying route values:

[Route("person/{id}")]

public ActionResult Contact()

{

ViewBag.Message = "Your contact page.";

return View();

}

Controller routes:

[Route("home/{action}")]

public class HomeController : Controller

{

public ActionResult Index()

{

return View();

}

public ActionResult About()

{

return View();

}

public ActionResult Contact()

{

return View();

}

Overriding controller route by action routes:

[Route("home/{action}")]

public class HomeController : Controller

{

[Route("home")]

[Route("home/index")]

public ActionResult Index()

{

return View();

}

public ActionResult About()

{

return View();

}

public ActionResult Contact()

{

return View();

}

}

Specifying RoutePrefix: This eliminates the necessity of specifying controller name each time you override.

[RoutePrefix("home")]

[Route("{action}")]

public class HomeController : Controller

{

[Route("")]

[Route("index")]

public ActionResult Index()

{

return View();

}

}

Specifying route constraint: A route constraint is useful in distinguishing incoming parameter by its type.

[Route("person/{id:int}")]

public ActionResult Details(int id)

{

// Do some work

return View();

}

[Route("person/{name}")]

public ActionResult Details(string name)

{

// Do some work

return View();

}

Specifying route defaults: Route defaults are helpful in specifying default values for a route parameter.

[Route("home/{action=Index}")]

For route parameters: This makes the id parameter optional for actions that does not have parameter.

[RoutePrefix("contacts")]

[Route("{action}/{id?}")]

Providig multiple defaults:

[Route("{action=Index}/{id?}")]

1. **Traditional routing:** Traditional routing specifies the rout template in the route config file. It matches the controller name and action name instead of giving own names.

**Examples:**

public static void RegisterRoutes(RouteCollection routes)

{

routes.MapRoute(

name: "Default",

url: "{controller}/{action}/{id}",

defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }

);

}

Specifying regular expression matching for route templates using **constraints**:

routes.MapRoute("blog", "{year}/{month}/{day}",

new { controller = "blog", action = "index" },

new { year = @"\d{4}", month = @"\d{2}", day = @"\d{2}" });

The above route matches the URL of the form /blog/2016/01/26

1. Routes are evaluated in the same order that they are mentioned in RegisterRoutes method. So place your maps carefully in the order you want to execute. Also it is preferable to place attribute routing before traditional routing.
2. Choose attribute routes or traditional routes?
   1. Consider choosing traditional routes when:
      1. You want centralized configuration of all your routes.
      2. You use custom constraint objects.
      3. You have an existing working application you don’t want to change.
   2. Consider choosing attribute routes when:
      1. You want to keep your routes together with your action’s code.
      2. You are creating a new application or making significant changes to an existing one.
3. **MVC Areas:**
   1. Areas allow us to divide models, controllers and views into separate sections which can make them lot easier to manage.
   2. More on areas see
      1. <http://www.codeproject.com/Articles/714356/Areas-in-ASP-NET-MVC>
      2. <http://www.dotnet-tricks.com/Tutorial/mvc/a9P1010113-MVC-Areas-with-example.html>
4. **Catch-all parameter:** A catch-all parameter allows for a route to match part of a URL with an arbitrary number of segments. The value put in the parameter is the rest of the URL path.

**Example:**

public static void RegisterRoutes(RouteCollection routes)

{

routes.MapRoute("catchallroute", "query/{query-name}/{\*extrastuff}");

}

The above URL matches URL like ‘/query/select/a/b/c’, ‘/query/select/a/b/c/’, ‘/query/select/’

1. StopRoutind and IgnoreRoute methods

routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");

The string we pass here is called **route template**

Chapter 11: ASP.NET Web API

1. ASP.NET is a framework for building and consuming HTTP services that can reach a broad range of clients including browsers, mobiles and tablets.
2. Difference between web services, WCF and Web API: <http://www.dotnet-tricks.com/Tutorial/webapi/JI2X050413-Difference-between-WCF-and-Web-API-and-WCF-REST-and-Web-Service.html>