Asp.net MVC Overview

Benefits of ASP.NET MVC

ASP.NET MVC – Pattern

ASP.NET MVC - Life Cycle

ASP.NET MVC – Routing, Attribute routing, Custom Routing, Route Constraints

Types of Action

Action Selector

ASP.NET MVC - Model Binding, Value Provider, Model Binder

## ASP.NET MVC OVERVIEW:

ASP.NET MVC is basically a web development framework from Microsoft, which combines the features of MVC (Model-View-Controller) architecture, the most up-to-date ideas and techniques from Agile development, and the best parts of the existing ASP.NET platform.

ASP.NET MVC is not something, which is built from ground zero. It is a complete alternative to traditional ASP.NET Web Forms.

ASP.NET MVC was announced in 2007, the MVC pattern was becoming one of the most popular ways of building web frameworks.

In April 2009, the ASP.NET MVC source code was released under the Microsoft Public License (MS-PL). "ASP.NET MVC framework is a lightweight, highly testable presentation framework that is integrated with the existing ASP.NET features.

Some of these integrated features are master pages and membership-based authentication. The MVC framework is defined in the System.Web.Mvc assembly.

## Benefits of ASP.NET MVC

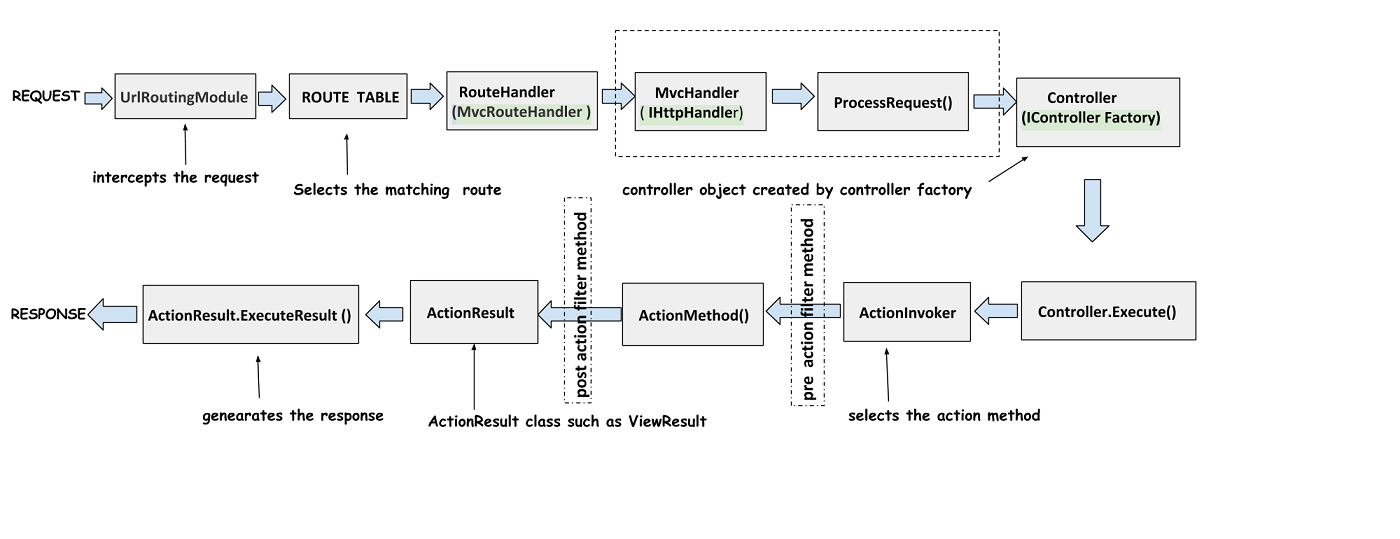
Following are the benefits of using ASP.NET MVC −

* It makes easier to manage complexity by dividing an application into the model, the view, and the controller.
* Enables full control over the rendered HTML and provides a clean separation of concerns.
* Direct control over HTML also means better accessibility for implementing compliance with evolving Web standards.
* Facilitates adding more interactivity and responsiveness to existing apps.
* Provides better support for test-driven development (TDD).
* Works well for Web applications that are supported by large teams of developers and for Web designers who need a high degree of control over the application behavior.

## ASP.NET MVC – Pattern

* **The Model** − A set of classes that describes the data you are working with as well as the business logic.
* **The View** − Defines how the application’s UI will be displayed. It is a pure HTML, which decides how the UI is going to look like.
* **The Controller** − A set of classes that handles communication from the user, overall application flow, and application-specific logic.

## ASP.NET MVC - Life Cycle



### **UrlRoutingModule**

The request is first intercepted by the UrlRoutingModule which is a HTTP Module. *This module that decides whether the request would be handled by our MVC application*. UrlRoutingModule selects the first matching route.

How does the UrlRoutingModule matches the request with the routes in the application?.

If you look into the RegisterRoutes method called from the global.asax you will notice that we add routes to the routes RouteCollection. This method is called from the application\_start event handler of the global.asax

It is this RegisterRoutes method which registers all the routes in the application

RouteConfig.RegisterRoutes(RouteTable.Routes);

public static void RegisterRoutes(RouteCollection routes)

{

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

routes.MapRoute(

name: "Default",

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

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

);

}

### **RouteHandler**

Whenever a request is received by **MVC**, it is the job of the routing engine to match the request URL with the registered **routes**. After finding the matched **route**, the**route handler** for the **route** is being called. Each **route** can have its own **route handler**. A **route handler** is a class that implements IRouteHandler interface.

### **MvcHandler (initiate mvc controller factory)**

**MvcHandler**. This **handler** is responsible for initiating the ASP.NET pipeline for an **MVC** application. It receives a Controller instance from the **MVC** controller factory; this controller handles further processing of the request.

### **ControllerFactory**

The **controller** is the glue between your View and Logic. The **controller** object is created by the default **controller factory** object using the constructor with no parameters.

### **Controller**

So we have seen that the ControllerFactory creates the Controller object in the ProcessRequest() method of the MvcHandler.

As we know a controller contains the action methods.An action method gets called when we request a URL in the browser.Instead of explicitly implementing the IController interface we create our controllers using the Controller class which provides many features for us.

After the controller object is created using the controller factory following happens :

* The Execute() method of the controllerbase is called
* This Execute() method calls the ExecuteCore() method which is declared abstract and is defined by Controller class.
* The Controller class's implementation of the ExecuteCore() method retrieves the action name from the RouteData
* ExecuteCore() method calls ActionInvoker's InvokeAction() method.

### **ActionInvoker**

The ActionInvoker class has some of the most important responsibilities of finding the action method in the controller and then invoking the action method.

### **ActionResult**

One of the characteristics of the action method is that instead of returning different data types it always returns the ActionResult type

*Below are some of these methods and the type of ActionResult they return*

|  |  |  |
| --- | --- | --- |
| **ActionResult Class** | **Helper Method** | **ReturnType** |
| ViewResult | View | web page |
| JsonResult | Json | Retuns a serialized JSON object |
| RedirectResult | Redirect | Redirects to another action method |
| ContentResult | Content | Returns a user-defined content type |

## ASP.NET MVC - Routing

Routing is the process of directing an HTTP request to a controller and the functionality of this processing is implemented in System.Web.Routing.

using System.Web.Routing;

namespace MVCFirstApp {

public class RouteConfig {

public static void RegisterRoutes(RouteCollection routes){

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

routes.MapRoute(

name: "Default",

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

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

}

}

}

A **route** is a URL pattern that is mapped to a handler. A handler can be a class that processes the request, such as a controller in an **MVC** application.

## Attribute routing

Attribute routing provides you more control over the URIs by defining routes directly on actions and controllers in your ASP.NET MVC. When **convention-based routing** is complex to support certain URI patterns than this type of routing is used in RESTful APIs

**1. Controller Level Attribute routing**

**2. Action Level Attribute routing**

1. **[RoutePrefix("MyHome")]**
2. **[Route("{action=index}")] *//default action***
3. **public class HomeController : Controller**
4. **{**
5. ***//new route: /MyHome/Index OR Home/Index***
6. **public ActionResult Index()**
7. **{**
8. **return View();**
9. **}**
10. **[Route("MyHome/{id:int:min(100)}")] *//route: / MyHome/100***
11. **public ActionResult Index(int id)**
12. **{**
13. ***//TO DO:***
14. **return View();**
15. **}**

## Custom Routing

It is possible to develop an MVC application without creating a Custom Route. But sometimes, there is a situation to change the pattern of URL in which it makes sense to create Custom Routes.

namespace MVCFirstApp{

public class RouteConfig{

public static void RegisterRoutes(RouteCollection routes){

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

routes.MapRoute(

"Process", "Processing/{action}/{id}",

defaults: new{

controller = "Process", action = "List ", id =

UrlParameter.Optional});

routes.MapRoute(

name: "Default",

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

defaults: new{

controller = "Home", action = "Index", id =

UrlParameter.Optional});

}

}

}

## Route Constraints

routes.MapRoute(

"Blog",

"Archive/{entrydate}",

new { Controller = "Blog", action = "Archive" },

new { entryDate = @"\d{2}-\d{2}-\d{4}" },*//Custom Route With Regular Expression Constraints*

new { method = new HttpMethodConstraint("GET") }*//Custom Route With HttpMethod Constraint*

);

## Types of Action

Actions basically return different types of action results. The ActionResult class is the base for all action results. Following is the list of different kind of action results and its behavior.

|  |  |
| --- | --- |
| **Sr.No.** | **Name and Behavior** |
| 1 | **ContentResult**  Returns a string |
| 2 | **FileContentResult**  Returns file content |
| 3 | **FilePathResult**  Returns file content |
| 4 | **FileStreamResult**  Returns file content |
| 5 | **EmptyResult**  Returns nothing |
| 6 | **JavaScriptResult**  Returns script for execution |
| 7 | **JsonResult**  Returns JSON formatted data |
| 8 | **RedirectToResult**  Redirects to the specified URL |
| 9 | **HttpUnauthorizedResult**  Returns 403 HTTP Status code |
| 10 | **RedirectToRouteResult**  Redirects to different action/different controller action |
| 11 | **ViewResult**  Received as a response for view engine |
| 12 | **PartialViewResult**  Received as a response for view engine |

## Action Selector

There are three types of Action Selector attributes −

* ActionName
* NonAction
* ActionVerbs

## ASP.NET MVC - Model Binding

ASP.NET MVC model binding allows you to map HTTP request data with a model. It is the process of creating .NET objects using the data sent by the browser in an HTTP request.

**Value Provider:** ValueProvider is like parsed source of dictionary that help model binder to get appropriate value for its model. **Value providers are responsible for collecting the values from the current request**

**Model Binder:** Model binders are responsible for querying/populate the values from value providers and building up object and bind with model in the post action method parameter.

## ASP.NET MVC – Partial Views

## Partial view is special view which renders a portion of view content. It is just like a user control web form application. Partial can be reusable in multipleviews. It helps us to reduce code duplication. In other word a partial view enables us to render a viewwithin the parent view

Use **Partial** when you have a model to send to the view and there will be a little bit of text that needs to be stored in a variable.

* + Returns MvcHtmlString, You can Modify it latter.
  + When you have little text and **model in main view to send** to partial view use this
  + Syntax is @Html.Partial(“viewname”, model);

Use **RenderPartial** when you have a model to send to the view and there will be a lot of html that doesn't need to be stored in a variable.

* + Return void, internaly calls write method of http response, same TextWriter object as used
  + When you have lot of html and **model in main view to send** partial view use this
  + Syntax is @{ Html.RenderPartial("ViewName", model); }

Use **Action** when you do not have a model in correspondence view to send to the partial view and have a little bit of text to bring back that needs to be stored in a variable.

* + Returns MvcHtmlString, You can Modify it latter.
  + When you have little text and **not** **model in main view to send** to partial view use this
  + Syntax is @Html.Partial(“viewname”);

Use **RenderAction** when you do not have a model to send to the view and have a lot of html to bring back that doesn't need to be stored in a variable.

* + Return void, internaly calls write method of http response, same TextWriter object as used
  + When you have lot of html and **not have model in main view** to send partial view use this
  + Syntax is @{ Html.RenderAction("ViewName"); }

## ASP.NET MVC – Child Action Only

The [ChildActionOnly](https://msdn.microsoft.com/en-us/library/system.web.mvc.childactiononlyattribute(v=vs.118).aspx) attribute ensures that an action method can be called only as a child method from within a view. An action method doesn’t need to have this attribute to be used as a child action, but we tend to use this attribute to prevent the action methods from being invoked as a result of a user request. Having defined an action method, we need to create what will be rendered when the action is invoked. Child actions are typically associated with partial views, although this is not compulsory.

1. [ChildActionOnly] allowing restricted access via code in View
2. State Information implementation for specific page URL. Example: Payment Page URL (paying only once) razor syntax allows to call specific actions conditional

**ChildActionOnly vs NoAction**

ChildActionOnly -- can only be called by another action and not directly from an external call (via routing). Permitted actions include Action/RenderAction extension methods.

NonAction -- Like marking a method "private" in regards to keeping it from being accessible from either an external call or as a child action. Good for protecting actions you don't need/want created as a direct view. Worth mentioning this is only necessary on public methods (as private/protected aren't considered "actionable").

**Ignore authorization and authentication for a specific action**

**[AllowAnonymous]**

**How to skip validation on textbox if model property is set as required in model annotation**

**@Html.Textboxfor(model=>model.UserName, new { data\_val=false})**

**XSS Attacks (Cross Site Scripting):**

XSS (cross-site scripting) is a security attack where the attacker injects malicious code while doing data entry. But the good thing is that XSS is by default prevented in MVC. So if any one tries to post JavaScript or HTML code then MVC shows the “Potentially Dangerous Request” error.

**ValidateInput:** ValidateInput and AllowHTML are directly connected with [XSS](http://en.wikipedia.org/wiki/Cross-site_scripting) security issues. There is some scenarios where we need to send HTML as input to action methods in that case We can allowed HTML as input on the complete action but that is dangerous.

[ValidateInput(false)]

public ActionResult PostProduct(Product obj)

{

return View(obj);

}

**AllowHTML:** AllowHTML will allow to send HTML as input to action method through model property

public class Product

{

public string ProductName { get; set; }

[AllowHtml]

public string ProductDescription { get; set; }

}

**ValidateAntiForgeryToken:**

**@Html.**

**How to map model property names in action method parametor**

**Ans:**

**How to exclude property validation in action method.**

public ActionResult Index([Bind(Exclude = "Id")] SomeDomainModel model)

**What is \_ViewStart?**

The \_ViewStart.cshtml file executes before the rendering of each view. This means that any code within this file will execute before any code in the view. This page helps us when we want to set or apply something before the view render. \_ViewStart.cshtml page used to define the default layout page for the MVC application. 

**It is common place to call layout page for all views**

Using the \_ViewStart.cshtml page, we can change the **layout page** based on a condition.

**What is Remote Validation in MVC**

To check instantly whether a record already exists in the database or not

[Remote("IsAlreadySigned", "Register",HttpMethod ="POST", ErrorMessage = "EmailId already exists in database.")]

**public** **string** UserEmailId { **get**; **set**; }

[HttpPost]

**public** JsonResult IsAlreadySigned(**string** UserEmailId)

{

**return** Json(true/false);

}

**What is Action Filter in MVC/Web Api**

Action filter used to log action name, controller name, execution time and error occurred in action method