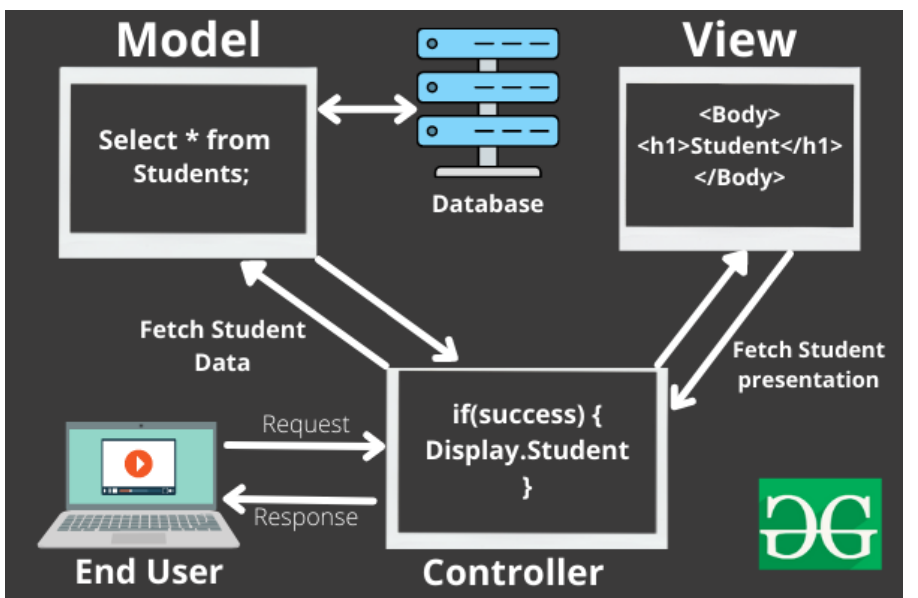
**Dot Net Document:**

**MVC Framework Introduction**

The [**Model-View-Controller (MVC)**](https://www.geeksforgeeks.org/mvc-design-pattern/) framework is an architectural/design pattern that separates an application into three main logical components **Model**, **View**, and **Controller**. Each architectural component is built to handle specific development aspects of an application



**Features of MVC :**

* It provides a **clear separation** of business logic, Ul logic, and input logic.
* It supports **Test Driven Development (TDD).**
* It offers full control over your HTML and URLs which makes it easy to design web application architecture.

**Disadvantages of MVC:**

* Increased complexity and Inefficiency of data
* It is not suitable for building small applications.

**MVC Fundamentals:**

// Add services to the container.

builder.Services.AddRazorPages();

builder.Services.AddControllersWithViews();

var app = builder.Build();

// Configure the HTTP request pipeline.

if (!app.Environment.IsDevelopment())

{

app.UseExceptionHandler("/Error");

app.UseHsts();

}

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseAuthorization();

app.MapDefaultControllerRoute();

app.MapRazorPages();

**Controller:**

The controller is the component that enables the interconnection between the views and the model so it acts as an intermediary.  It process all the business logic and incoming requests, manipulate data using the **Model**component and interact with the **View**to render the final output.

**View:**

The **View**component is used for all the UI logic of the application. It generates a user interface for the user. Views are created by the data which is collected by the model component but these data aren’t taken directly but through the controller.

**Model:**

The **Model**component corresponds to all the data-related logic that the user works with. This can represent either the data that is being transferred between the View and Controller components or any other business logic-related data.

**Razer View:**

Razor View engine is a markup syntax which helps us to write HTML and server-side code in web pages using C# or VB.NET.

Razor is a templating engine and ASP.NET MVC has implemented a view engine which allows us to use Razor inside of an MVC application to produce HTML. However, Razor does not have any ties with ASP.NET MVC.

**Explanation about IIS Express**

@{

var price = 101;

}

@{

if(price == 100)

{

<p>It's hundred</p>

}

else

{

<p>It's not hundred</p>

}

}

**ASP.NET MVC Core Action Methods**

Every public method of the controller is ActionMethod except methods marked as [NonAction]. Action methods are similar to normal methods however ActionMethod has limitations as

* Action method must be public, it can not be private.
* Action method can not be static or extension method.
* Action method can not be getter or setter.

[ActionName("GetProductNameByID")]

public string GetProductName(int ProductID)

{

return "Product name is ABC.";

}

[ActionName("GetProductNameByProductCode")]

public string GetProductName(string code)

{

return "Product name is ABC.";

}

**ActionResult**

MVC framework includes various Result classes, which can be returned from an action method. The result classes represent different types of responses, such as HTML, file, string, JSON, javascript, etc. The following table lists all the result classes available in ASP.NET MVC.

| Result Class | Description |
| --- | --- |
| ViewResult | Represents HTML and markup. |
| EmptyResult | Represents No response. |
| ContentResult | Represents string literal. |
| FileContentResult/ FilePathResult/ FileStreamResult | Represents the content of a file. |
| JavaScriptResult | Represent a JavaScript script. |
| JsonResult | Represent JSON that can be used in AJAX. |
| RedirectResult | Represents a redirection to a new URL. |
| RedirectToRouteResult | Represent another action of same or other controller. |
| PartialViewResult | Returns HTML from Partial view. |
| HttpUnauthorizedResult | Returns HTTP 403 status. |

**Routing in ASP.NET Core MVC**

Routing is the process through which the application matches an incoming URL path and executes the corresponding action methods. ASP.NET Core MVC uses a routing middleware to match the URLs of incoming requests and map them to specific action methods.

There are two types of routing for action methods:

* [Conventional Routing](https://code-maze.com/routing-asp-net-core-mvc/#conventionalrouting)
* [Attribute Routing](https://code-maze.com/routing-asp-net-core-mvc/#attributerouting)

**Conventional Routing**

Configure routing in Configure method in startup.cs class

app.UseEndpoints(endpoints =>

{

endpoints.MapControllerRoute(

name: "default",

pattern: "{controller=Home}/{action=Index}/{id?}");

endpoints.MapControllerRoute(

name : "employee",

pattern: "EmployeeDetails",

defaults: new { controller = "Employee", action = "Index" });

});

**Attribute Routing**

By placing a route on the controller or the action method, we can make use of the Attribute Routing feature.

Let’s modify the Configure() method in the startup.cs class and remove the default routes that we had defined earlier.

[Route("[controller]/[action]")]

[Route("[controller]")]

public class TestController : Controller

{

[Route("")] // Matches 'Test'

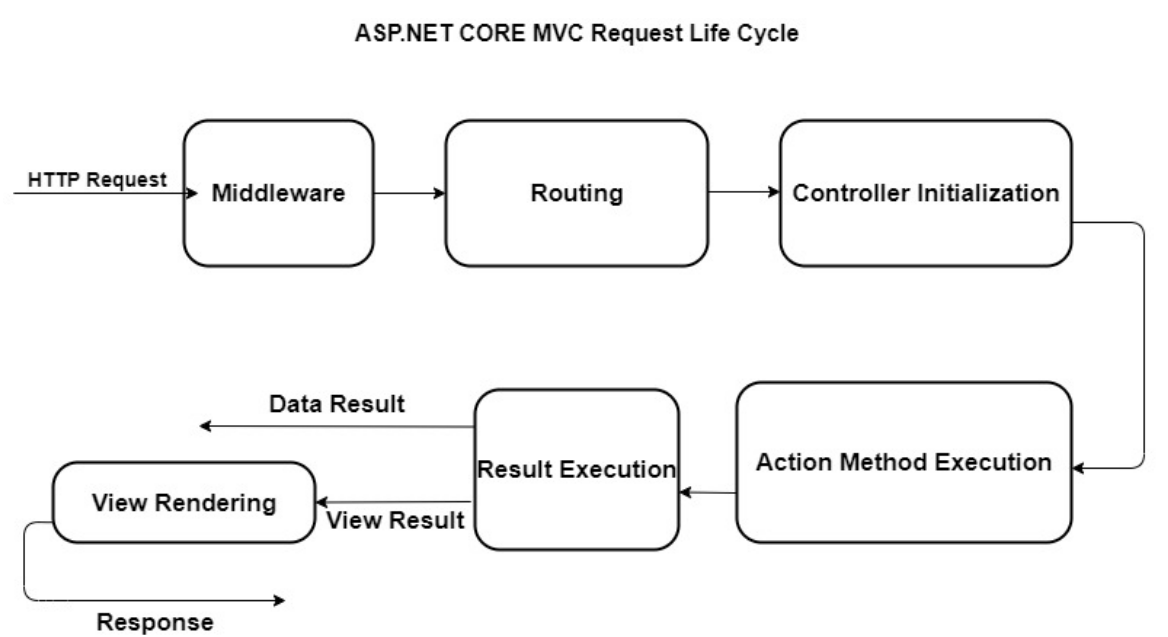
[Route("Index")] // Matches 'Test/Index'

public IActionResult Index()

}

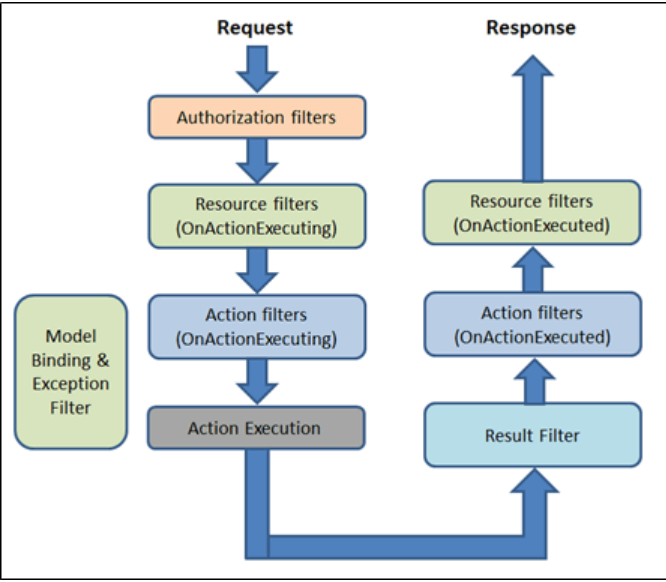
**ASP.NET Core MVC Request Life Cycle/Pipeline**

The ASP.NET Core MVC Request Life Cycle is a sequence of events, stages or components that interact with each other to process an HTTP request and generate a response that goes back to the client. In this article, we will discuss each and every stage of ASP.NET Core MVC Request Life Cycle in detail.



**Filters in MVC**

Every filter type is executed at a different stage in the filter pipeline. Following are the filter types.



***Authorization filters***  
The Authorization filters are executed first. This filter helps us to determine whether the user is authorized for the current request. It can short-circuit a pipeline if a user is unauthorized for the current request. We can also create custom authorization filter.

public class AuthorizeActionFilter : Attribute, IAuthorizationFilter

{

public void OnAuthorization(AuthorizationFilterContext context)

{

//Write you code here to authorize or unauthorize the user

}

}

***Resource filters***

The Resource filters handle the request after authorization. It can run the code before and after the rest of the filter is executed. This executes before the model binding happens. It can be used to implement caching.

public class CustomResourceFilterAttribute : Attribute, IResourceFilter

{

public void OnResourceExecuting(ResourceExecutingContext context)

{

context.Result = new ContentResult()

{

Content = "This is a Resource filter."

};

}

public void OnResourceExecuted(ResourceExecutedContext context)

{

}

}

***Action filters***  
The Action filters run the code immediately before and after the controller action method is called. It can be used to perform any action before or after execution of the controller action method. We can also manipulate the arguments passed into an action.

public class CustomActionFilter : IActionFilter

{

public void OnActionExecuting(ActionExecutingContext context)

{

// Executed before execution of an action method

}

public void OnActionExecuted(ActionExecutedContext context)

{

// Executed after execution of an action method

}

}

***Exception filters***  
The Exception filters are used to handle exception that occurred before anything written to the response body.

public class CustomExceptionFilter : Attribute, IExceptionFilter

{

public void OnException(ExceptionContext context)

{

context.Result = new ViewResult()

{

StatusCode = (int)HttpStatusCode.BadRequest,

ViewName = "Error"

};

context.ExceptionHandled = true;

}

}

***Result filters***  
The Result filters are used to run code before or after the execution of controller action results. They are executed only if the controller action method has been executed successfully.

public class CustomResultFilter : Attribute, IResultFilter

{

public void OnResultExecuting(ResultExecutingContext context)

{

context.Result = new ViewResult

{

ViewName = "Hello"

};

}

public void OnResultExecuted(ResultExecutedContext context)

{

}

}

**Transport data from controller to view**

@model Employee

<h2>Employee Detail:</h2>

<ul>

<li>Student Id: @Model.EmployeeId</li>

<li>Student Name: @Model. EmployeeName</li>

<li>Age: @Model.Age</li>

</ul>

**Authentication And Authorization In ASP.NET Core MVC**

https://www.c-sharpcorner.com/article/authentication-and-authorization-in-asp-net-core-mvc-using-cookie/#:~:text=Authentication%20And%20Authorization%20In%20ASP.NET%20Core%20MVC%20Using%20Cookie,-Mukesh%20Kumar&text=Security%20is%20the%20main%20concern,think%20about%20Authentication%20and%20Authorization.