# 2. ASP.NET Core Request Processing Pipeline

## **Request Pipeline Overview**

Every .NET core web app uses a Startup class to bootstrap the application. Startup class has two methods:

* **Configure Services**, to configure the dependencies
* **Configure**, to configure the request processing pipeline. This is the part which decides which middlewares would be invoked.

The middlewares are invoked in the order in which they are configured in Configure method.

Below diagram shows overview of recommended order of middlewares for .NET Core web application. It includes some middlewares which we may not want to use with .NET core web API app.

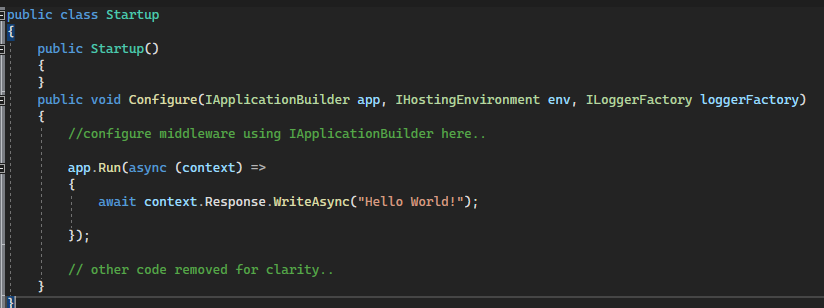
# **Middleware**

* ASP.NET Core introduced a new concept called Middleware.
* A middleware is nothing but a component (class) which is executed on every request in ASP.NET Core application.
* In the classic ASP.NET, HttpHandlers and HttpModules were part of request pipeline.
* Middleware is similar to HttpHandlers and HttpModules where both needs to be configured and executed in each request.
* there will be multiple middleware in ASP.NET Core web application. It can be either framework provided middleware, added via NuGet or your own custom middleware.
* We can set the order of middleware execution in the request pipeline.
* Each middleware adds or modifies http request and optionally passes control to the next middleware component.
* But a middleware component can decide not to call the next piece of middleware in the pipeline. This is called **short-circuiting** or terminate the request pipeline.
* Short-circuiting is often desirable because it avoids unnecessary work.



## **Configure Middleware**

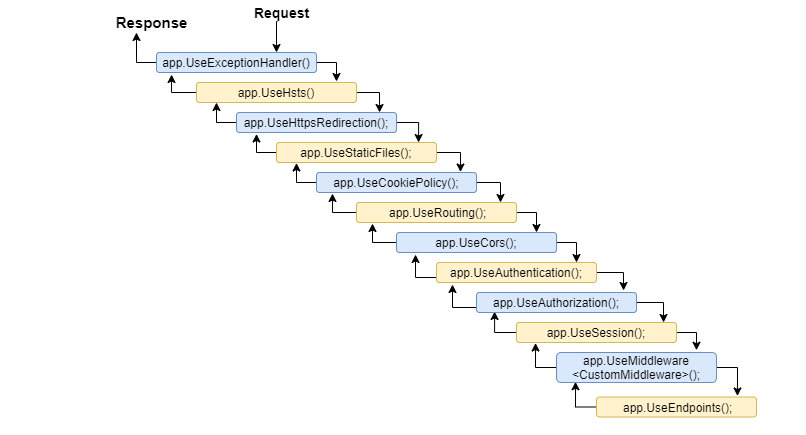
* We can configure middleware in the Configure method of the Startup class using IApplicationBuilder instance.
* The following example adds a single middleware using Run method which returns a string "Hello World!" on each request.



* In the above example, Run() is an extension method on IApplicationBuilder instance which adds a terminal middleware to the application's request pipeline.
* The above configured middleware returns a response with a string "Hello World!" for each request.

## **Middleware Ordering**

* Middleware components are executed in the order they are added to the pipeline and care should be taken to add the middleware in the right order otherwise the application may not function as expected.
* This ordering is critical for security, performance, and functionality.
* The first configured middleware has received the request, modify it (if required), and passes control to the next middleware.
* Similarly, the first middleware is executed at the last while processing a response if the echo comes back down the tube.
* That’s why Exception-handling delegates need to be called early in the pipeline, so they can validate the result and displays a possible exception in a browser and client-friendly way.



## **Run Method**

* We used Run extension method to add middleware.

The following is the signature of the Run method:

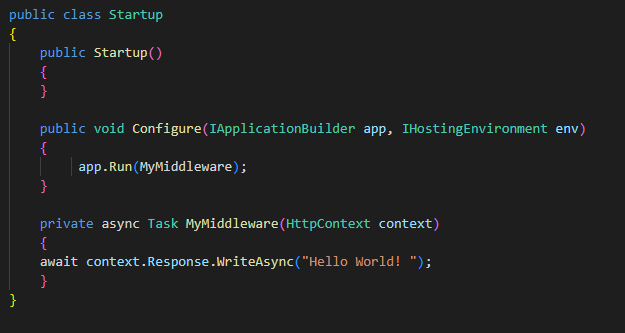


* The Run method is an extension method on IApplicationBuilder and accepts a parameter of RequestDelegate.
* The RequestDelegate is a delegate method which handles the request.

The following is a RequestDelegate signature.

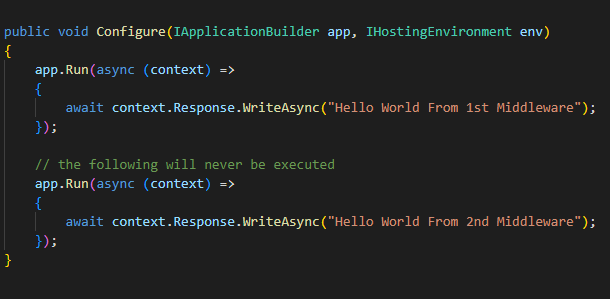


* As you can see above, the Run method accepts a method as a parameter whose signature should match with RequestDelegate. Therefore, the method should accept the HttpContext parameter and return Task.
* So, you can either specify a lambda expression or specify a function in the Run method.
* The lambda expression specified in the Run method above is similar to the one in the example shown below.



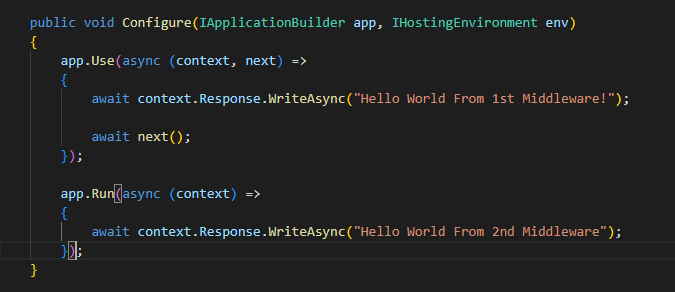
## **Configure Multiple Middleware**

* Mostly there will be multiple middleware components in ASP.NET Core application which will be executed sequentially.
* The Run method adds a terminal middleware so it cannot call next middleware as it would be the last middleware in a sequence.
* The following will always execute the first Run method and will never reach the second Run method.



To configure multiple middleware, use Use() extension method. It is similar to Run() method except that it includes next parameter to invoke next middleware in the sequence. Consider the following example.

This example will display **Hello World From 1st Middleware! Hello World From 2nd Middleware!** in the browser.



## 

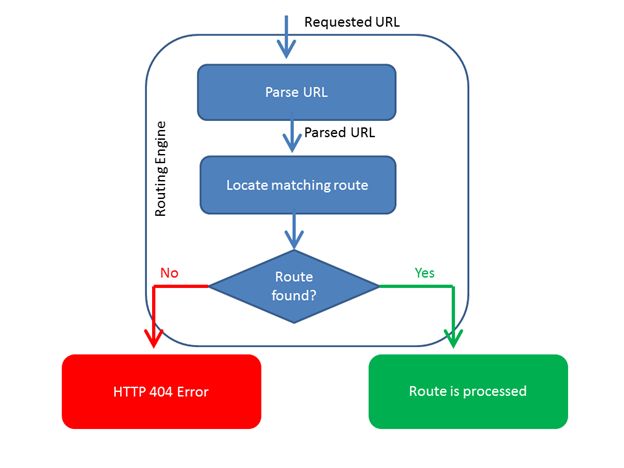
## **Built-in Middleware Via NuGet**

ASP.NET Core is a modular framework. We can add server side features we need in our application by installing different plug-ins via NuGet. There are many middleware plug-ins available which can be used in our application.

| Middleware | Description |
| --- | --- |
| Authentication | Adds authentication support. |
| CORS | Configures Cross-Origin Resource Sharing. |
| Routing | Adds routing capabilities for MVC or web form |
| Session | Adds support for user session. |
| Static Files | Adds support for serving static files and directory browsing. |
| Diagnostics | Adds support for reporting and handling exceptions and errors. |

# **Routing**

* Routing is a pattern matching system that monitors the incoming request and figures out what to do with that request.
* Typically, it is a way to serve the user request.
* When a user request URLs from the server then URLs are handled by the routing system.
* The Routing system try to find out the matching route pattern of requested URL with already registered routes which are map to controller, actions, files, or other items.
* If there is a matching route entry, then it processes the request i.e. serve the resource, otherwise it returns 404 error.



## **Types of Routing**

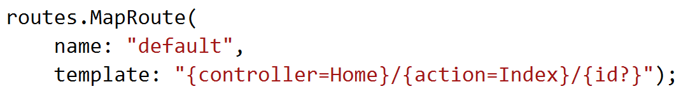
There are two main ways to define routes in ASP.NET Core:

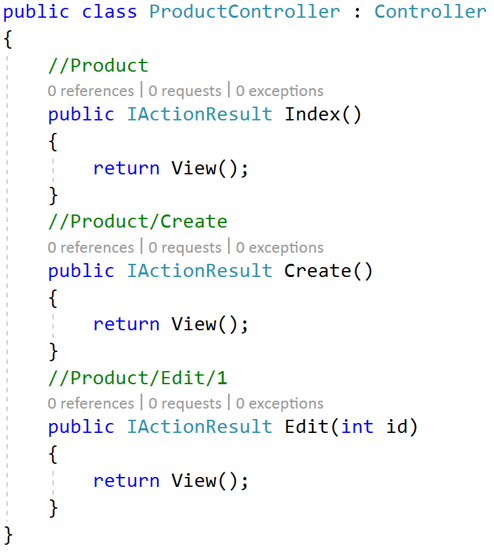
## Convention-based Routing

## Attribute Routing

## **Convention-based Routing**

It creates routes based on a series of conventions which represent all the possible routes in your system. Convention-based are defined in the Startup.cs file.





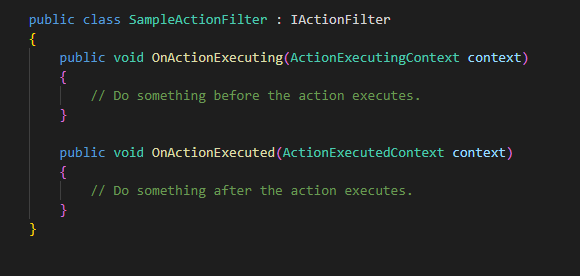
## **Attribute Routing**

It creates routes based on attributes placed on controller or action level. Attribute routing provides us more control over the URLs generation patterns which helps us in SEO.

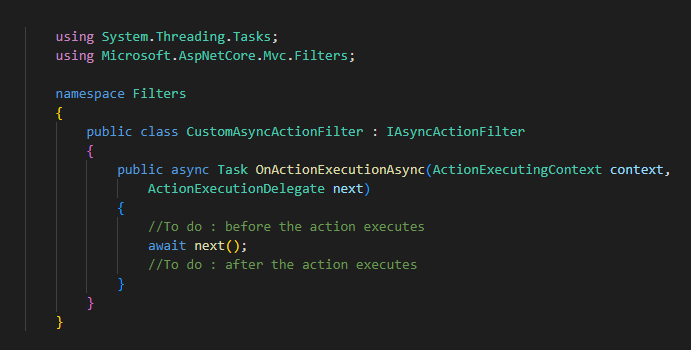


* **Filters**
* Filters allow us to run custom code before or after executing the action method. They provide ways to do common repetitive tasks on our action method.
* The filters are invoked on certain stages in the request processing pipeline.
* we can create custom filters as well.
* Filters help us to remove duplicate codes in our application.
* **Filter Types**
* Every filter type is executed at a different stage in the filter pipeline. Following are the filter types.
* Filter supports two types of implementation: synchronous and asynchronous; Both the implementations use different interface definitions.
* The Synchronous filters run the code before and after their pipeline stage defines OnStageExecuting and OnStageExecuted. For example, ActionFilter. The OnActionExecuting method is called before the action method and OnActionExecuted method is called after the action method.

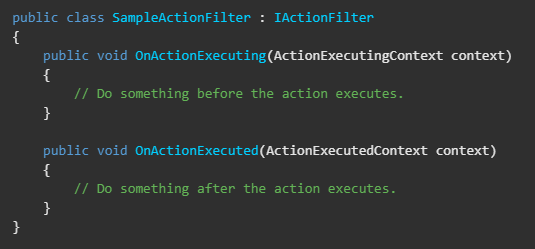
**Synchronous Filter Example**



**Asynchronous filter example**



* **Authorization filters**
* Are the first filters run in the filter pipeline.
* Control access to action methods.
* Have a before method, but no after method.
* Custom authorization filters require a custom authorization framework.
* Prefer configuring the authorization policies or writing a custom authorization policy over writing a custom filter.
* The built-in authorization filter:
  + Calls the authorization system.
  + Does not authorize requests.
  + Do not throw exceptions within authorization filters:
  + The exception will not be handled.
  + Exception filters will not handle the exception.
* **Resource filters**
* Implement either the IResourceFilter or IAsyncResourceFilter interface.
* Execution wraps most of the filter pipeline.
* Only Authorization filters run before resource filters.
* Resource filters are useful to short-circuit most of the pipeline. For example, a caching filter can avoid the rest of the pipeline on a cache hit.
* **Action filters**
* Implement either the IActionFilter or IAsyncActionFilter interface.
* Their execution surrounds the execution of action methods.
* The following code shows a sample action filter:



The ActionExecutingContext provides the following properties:

* Action Arguments - enables reading the inputs to an action method.
* Controller - enables manipulating the controller instance.
* Result - setting Result short-circuits execution of the action method and subsequent action filters.

Throwing an exception in an action method:

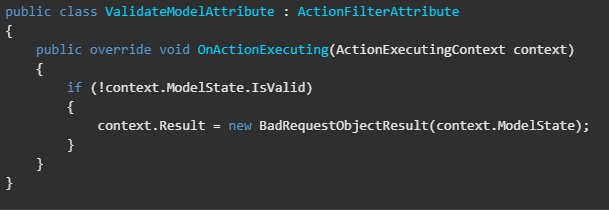
* Prevents running of subsequent filters.
* Unlike setting Result, is treated as a failure instead of a successful result.

The ActionExecutedContext provides Controller and Result plus the following properties:

* Cancelled - True if the action execution was short-circuited by another filter.
* Exception - Non-null if the action or a previously run action filter threw an exception. Setting this property to null:
* Effectively handles the exception.
* Result is executed as if it was returned from the action method.

The OnActionExecuting action filter can be used to:

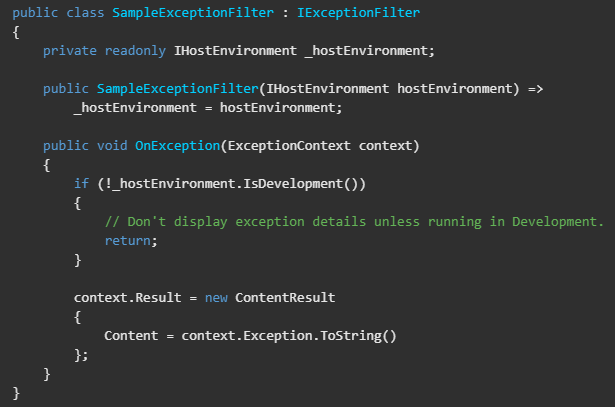
* Validate model state.
* Return an error if the state is invalid.



The OnActionExecuted method runs after the action method:

* And can see and manipulate the results of the action through the Result property.
* Cancelled is set to true if the action execution was short-circuited by another filter.
* Exception is set to a non-null value if the action or a subsequent action filter threw an exception. Setting Exception to null:
  + Effectively handles an exception.
  + ActionExecutedContext.Result is executed as if it were returned normally from the action method.
* **Exception filters**
* Implement IExceptionFilter or IAsyncExceptionFilter.
* Can be used to implement common error handling policies.

The following sample exception filter displays details about exceptions that occur when the app is in development:

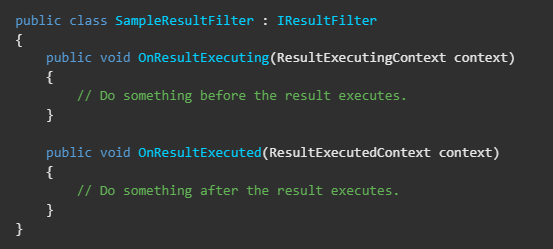


Exception filters:

* Don't have before and after events.
* Implement OnException or OnExceptionAsync.
* Handle unhandled exceptions that occur in Razor Page or controller creation, model binding, action filters, or action methods.
* Do not catch exceptions that occur in resource filters, result filters, or MVC result execution.

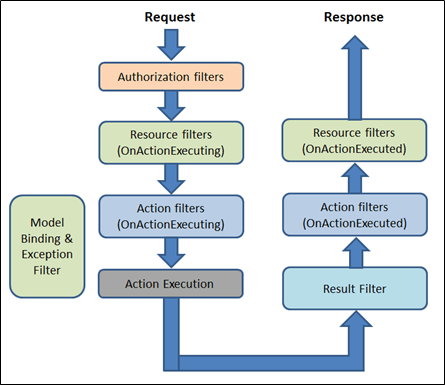
To handle an exception, set the ExceptionHandled property to true or assign the Result property. This stops propagation of the exception. An exception filter can't turn an exception into a "success". Only an action filter can do that.

* **Result filters**
* Implement an interface:
  + IResultFilter or IAsyncResultFilter
  + IAlwaysRunResultFilter or IAsyncAlwaysRunResultFilter
* Their execution surrounds the execution of action results.



The kind of result being executed depends on the action. An API method might perform some serialization as part of the execution of the result. Result filters are only executed when an action or action filter produces an action result. Result filters are not executed when:

* + An authorization filter or resource filter short-circuits the pipeline.
  + An exception filter handles an exception by producing an action result.



* **Controller Initialization**

Controllers, actions, and action results are a fundamental part of how developers build apps using ASP.NET Core MVC.

* **What is a Controller**?
* A controller is used to define and group a set of actions.
* An action (or action method) is a method on a controller which handles requests. Controllers logically group similar actions together.
* This aggregation of actions allows common sets of rules, such as routing, caching, and authorization, to be applied collectively.
* Requests are mapped to actions through routing.

A controller is an instantiable class, usually public, in which at least one of the following conditions is true:

* The class name is suffixed with Controller.
* The class inherits from a class whose name is suffixed with Controller.
* The [Controller] attribute is applied to the class.
* A controller class must not have an associated [NonController] attribute.
* The controller is a UI-level abstraction. Its responsibilities are to ensure request data is valid and to choose which view (or result for an API) should be returned.
* In well-factored apps, it doesn't directly include data access or business logic. Instead, the controller delegates to services handling these responsibilities.
* Every controller must be derived from this **abstract** Controller class. This base Controller class contains helper methods that can be used for various purposes.
* **Controller Helper Methods**

Controllers usually inherit from Controller, although this isn't required. Deriving from Controller provides access to three categories of helper methods:

* **Methods resulting in an empty response body**

No Content-Type HTTP response header is included, since the response body lacks content to describe.

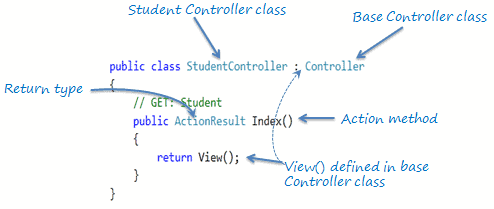
There are two result types within this category: Redirect and HTTP Status Code.

* **HTTP Status Code**
* This type returns an HTTP status code.
* A couple of helper methods of this type are BadRequest, NotFound, and Ok.
* For example, return BadRequest(); produces a 400 status code when executed. When methods such as **BadRequest**, **NotFound** and 200 for **Ok** are overloaded, they no longer qualify as HTTP Status Code responders, since content negotiation is taking place.
* **Redirect**
* This type returns a redirect to an action or destination (using Redirect, LocalRedirect, RedirectToAction, or RedirectToRoute).
* For example, return RedirectToAction("Complete", new {id = 123}); redirects to Complete, passing an anonymous object.
* The Redirect result type differs from the HTTP Status Code type primarily in the addition of a Location HTTP response header.

# **Action method**

All the public methods of the Controller class are called Action methods. They are like any other normal methods with the following restrictions:

* Action method must be public.
* Action method cannot be overloaded, a static method and private or protected.



the Index() method is public, and it returns the ActionResult using the View() method. The View() method (use in MVC Model) is defined in the Controller base class, which returns the appropriate ActionResult.

* All the public methods in the Controller class are called Action methods.
* The Action method has the following restrictions.
  1. Action method must be public. It cannot be private or protected.
  2. Action method cannot be overloaded.
  3. Action method cannot be a static method.
* ActionResult is a base class of all the result type which returns from Action method.
* The base Controller class contains methods that returns appropriate result type e.g. View(), Content(), File(), JavaScript() etc.
* The Action method can include Nullable type parameters.