Contents

[Generic host (HostBuilder) //TODO 1](#_Toc158826339)

[Configuration //TODO 1](#_Toc158826340)

[Dependency injection //TODO 1](#_Toc158826341)

[Options pattern 3](#_Toc158826342)

[Logging //TODO 3](#_Toc158826343)

[Hosted service // TODO 3](#_Toc158826344)

[ASP .NET Core 3](#_Toc158826345)

[ASP .NET Core vs ASP .NET Core MVC 3](#_Toc158826346)

[Kestrel vs IIS 4](#_Toc158826347)

[Фильтры 4](#_Toc158826348)

[Свой фильтр 5](#_Toc158826349)

[Глобальные фильтры 5](#_Toc158826350)

[Установка отдельно для контроллеров: 5](#_Toc158826351)

[Filter vs Middleware 6](#_Toc158826352)

[Отличия MVC от ASP .NET 6](#_Toc158826353)

Generic host (HostBuilder) //TODO

Просто все отсюда: <https://docs.microsoft.com/en-us/dotnet/core/extensions/generic-host>

Configuration //TODO

<https://docs.microsoft.com/en-us/dotnet/core/extensions/configuration>

Appsettings - <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/?view=aspnetcore-6.0>

Options pattern

<https://docs.microsoft.com/en-us/dotnet/core/extensions/options>

Logging //TODO

<https://docs.microsoft.com/en-us/dotnet/core/extensions/logging?tabs=command-line>

Hosted service // TODO

<https://docs.microsoft.com/en-us/aspnet/core/fundamentals/host/hosted-services?view=aspnetcore-6.0&tabs=visual-studio>

ASP .NET Core

ASP .NET Core vs ASP .NET Core MVC

<https://docs.microsoft.com/en-us/dotnet/architecture/porting-existing-aspnet-apps/middleware-modules-handlers>

You must remember how you differentiate between a normal MVC Controller and a Web API Controller...

WebAPI Controllers enforces this annotation [ApiController] and must inherits from ControllerBase:

[ApiController]

public class PeopleController : ControllerBase {

//Your API methods here

}

A normal MVC Controller only inherits from Controller base class:

public class PeopleController : Controller {

//Your Action methods here...

}

Those already create configuration for your APP which becomes easier for you Hosting environment to know what is going and what to return when.

Kestrel vs IIS

<https://www.tutorialspoint.com/what-is-kestrel-and-how-does-it-differ-from-iis-asp-net>

Dependency injection //TODO

**Construction Injection**

Зависимости нужны на этапе создания, чтобы проинициализировать объекты. Т.е. конструктор использует параметры для внедрения зависимостей.

Плюсы/минусы

зависимость можно сделать immutable (неизменяемой), чтобы предотвратить повторное внедрение зависимости

все зависимости должны быть переданы в конструкторе

**Setter Injection**

Для внедрения сложных зависимостей в указанный момент

Плюсы/минусы

не нужно все прописывать в конструкторе

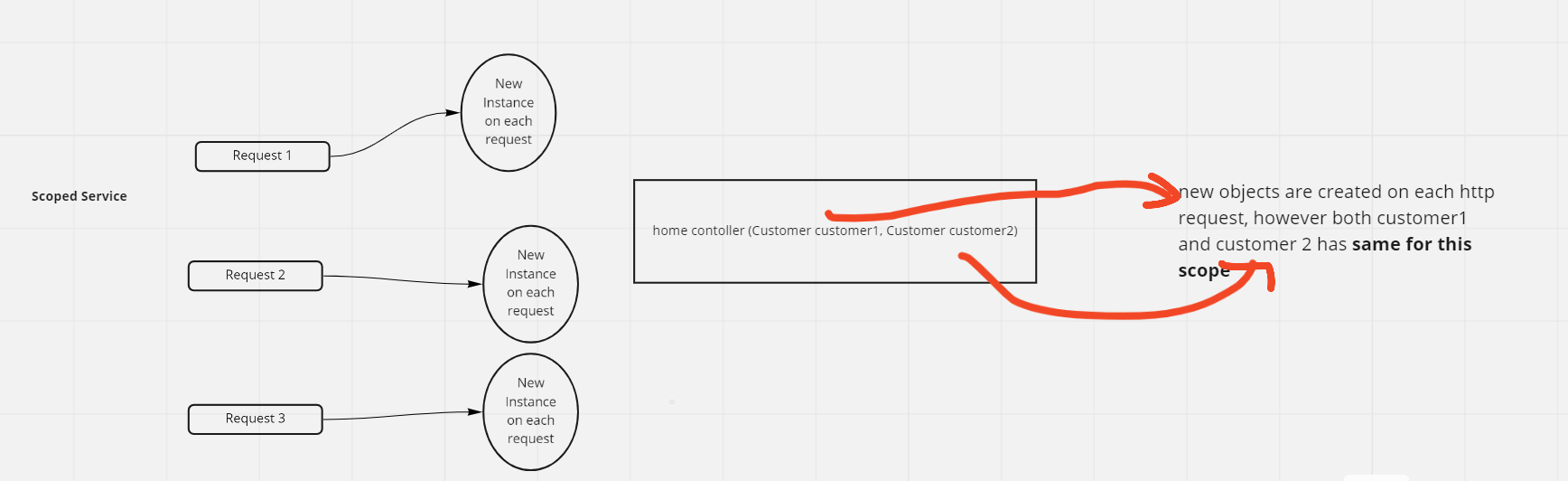
всегда надо отслеживать, какая зависимость нужна, чтоб подгрузить ее

**Interface Based Injection**

…

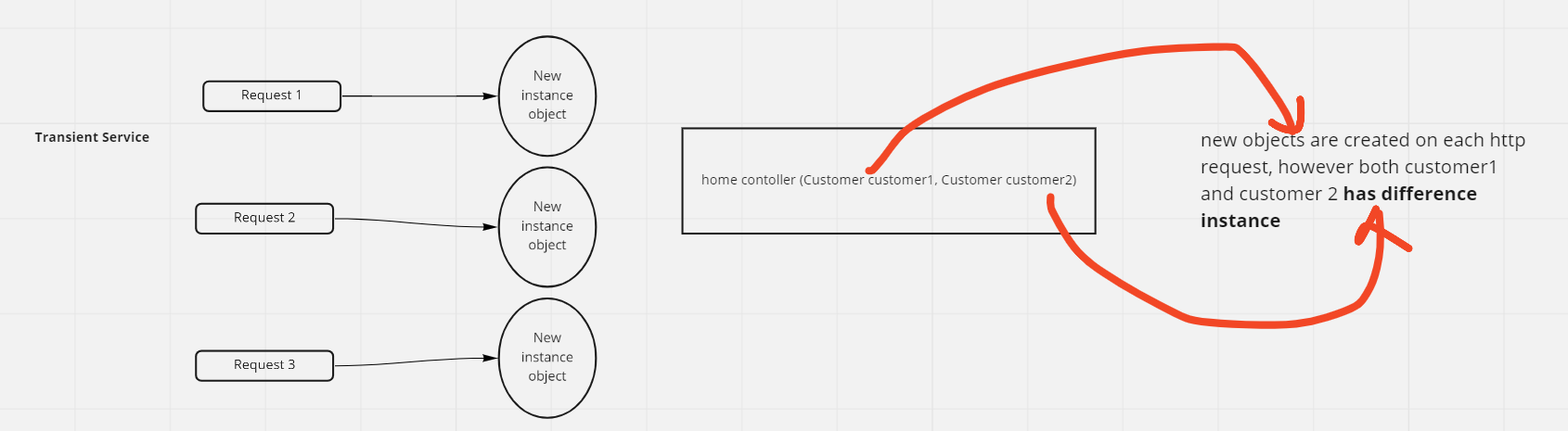
## **Scoped**

* In this service, with every HTTP request, we get a new instance.
* The same instance is provided for the entire scope of that request.
  + eg., if we have a couple of parameter in the controller, both object contains the same instance across the request
* This is a better option when you want to maintain a state within a request.



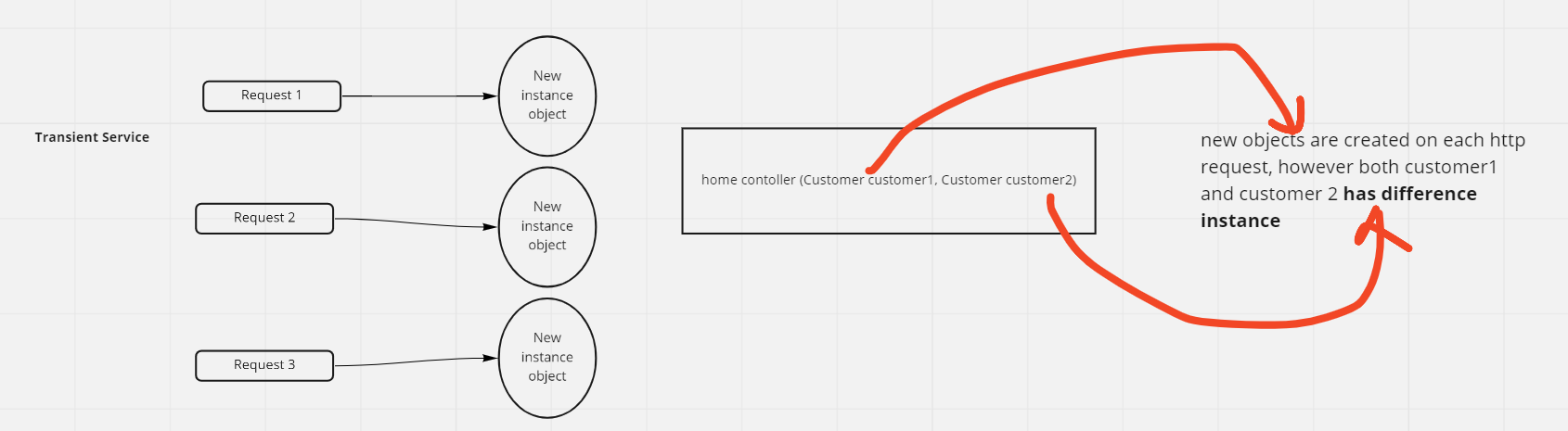
## **Transient**

* A new service instance is created for each object in the HTTP request.
* This is a good approach for the multithreading approach because both objects are independent of one another.
* The instance is created every time they will use **more memory** and **resources**and can have a **negative** impact on performance
* Utilize for the **lightweight** service with little or **no state**.



## **Singleton**

* Only one service instance was created throughout the lifetime.
* Reused the same instance in future, wherever the service is required
* Since it's a single lifetime service creation, memory leaks in these services will build up over time.
* Also, it has memory efficient as they are created once reused everywhere.



DI - <https://docs.microsoft.com/en-us/dotnet/core/extensions/dependency-injection>

Типы регистрации сервисов:

<https://docs.microsoft.com/en-us/dotnet/core/extensions/dependency-injection#service-registration-methods>

| **Method** | **Automatic object disposal** | **Multiple implementations** | **Pass args** |
| --- | --- | --- | --- |
| Add{LIFETIME}<{SERVICE}, {IMPLEMENTATION}>()  Example:  services.AddSingleton<IMyDep, MyDep>(); | Yes | Yes | No |
| Add{LIFETIME}<{SERVICE}>(sp => new {IMPLEMENTATION})  Examples:  services.AddSingleton<IMyDep>(sp => new MyDep()); services.AddSingleton<IMyDep>(sp => new MyDep(99)); | Yes | Yes | Yes |
| Add{LIFETIME}<{IMPLEMENTATION}>()  Example:  services.AddSingleton<MyDep>(); | Yes | No | No |
| AddSingleton<{SERVICE}>(new {IMPLEMENTATION})  Examples:  services.AddSingleton<IMyDep>(new MyDep()); services.AddSingleton<IMyDep>(new MyDep(99)); | No | Yes | Yes |
| AddSingleton(new {IMPLEMENTATION})  Examples:  services.AddSingleton(new MyDep()); services.AddSingleton(new MyDep(99)); |  |  |  |

IOC - <https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/architectural-principles#dependency-inversion>

* [TryAdd](https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.dependencyinjection.extensions.servicecollectiondescriptorextensions.tryadd)
* [TryAddTransient](https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.dependencyinjection.extensions.servicecollectiondescriptorextensions.tryaddtransient)
* [TryAddScoped](https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.dependencyinjection.extensions.servicecollectiondescriptorextensions.tryaddscoped)
* [TryAddSingleton](https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.dependencyinjection.extensions.servicecollectiondescriptorextensions.tryaddsingleton)

один интерфейс можно регистрировать для разных сервисов

ASP .NET MVC

Routing

ASP.NET Routing is setup in **two** places.

**First**, ASP.NET Routing is enabled in your application's Web configuration file (Web.config file). There are four sections in the configuration file that are relevant to routing: the system.web.httpModules section, the system.web.httpHandlers section, the system.webserver.modules section, and the system.webserver.handlers section.

**Second**, and more importantly, a route table is created in the application's Global.asax file. The Global.asax file is a special file that contains event handlers for ASP.NET application lifecycle events. The route table is created during the Application Start event.

namespace MvcApplication1

{

// Note: For instructions on enabling IIS6 or IIS7 classic mode,

// visit https://go.microsoft.com/?LinkId=9394801

public class MvcApplication : System.Web.HttpApplication

{

public static void RegisterRoutes(RouteCollection routes)

{

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

routes.MapRoute(

"Default", // Route name

"{controller}/{action}/{id}", // URL with parameters

new { controller = "Home", action = "Index", id = "" } // Parameter defaults

);

}

protected void Application\_Start()

{

RegisterRoutes(RouteTable.Routes);

}

}

}

Because of the Default route parameter defaults, entering this URL will cause the Index() method of the HomeController class in Listing 2 to be called.  
Listing 2 - HomeController.cs

using System.Web.Mvc;

namespace MvcApplication1.Controllers

{

[HandleError]

public class HomeController : Controller

{

public ActionResult Index(string id)

{

return View();

}

}

}

Types of Routing

**Convention-Based Routing**

The route is determined based on conventions that are defined in route templates that, at runtime, will map requests to controllers and actions (methods).

app.UseRouting();

app.MapControllerRoute(

name: "default",

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

**Attribute-Based Routing**

The route is determined based on attributes that you set on your controllers and methods. These will define the mapping to the controller’s actions.

public class HomeController : Controller

{

[Route("")]

[Route("Home")]

[Route("Home/Index")]

public string Index()

{

return "Index() // Action Method of HomeController";

}

[Route("Home/Details/{id}")]

public string Details(int id)

{

return "Details() // Action Method of HomeController, ID Value = " + id;

}

}

Result types

1. **View Results**:

ViewResult: Renders an HTML view using a specified model. This is the most common way to return web pages in ASP.NET MVC.

PartialViewResult: Similar to ViewResult, but renders a partial view, often used for dynamically updating sections of a page.

2. **Redirect Results**:

RedirectResult: Redirects the user to a specified URL.

RedirectToActionResult: Redirects to another action within the same or a different controller.

RedirectToRouteResult: Redirects to a URL generated based on a route.

3. **Data Results**:

JsonResult: Returns data in JSON format, commonly used for AJAX requests and APIs.

ContentResult: Returns plain text or other content types, useful for simple text responses.

FileResult: Returns a file for download, like a PDF or image.

3.1 **FileResult**

**FileResult is the parent of all file-related action results. It is a base class that is used to send binary file content to the response. It represents an ActionResult that when executed will write a file as the response.**

**FileContentResult**

**FileStreamResult**

**VirtualFileResult**

**PhysicalFileResult**

4. **Status Results**:

EmptyResult: Represents no result, sometimes used for actions that perform an operation but don't need to return anything.

StatusCodeResult: Returns an HTTP status code, like 404 (Not Found) or 500 (Internal Server Error).

5. **Other Results**:

JavaScriptResult: Returns JavaScript code to be executed on the client-side.

ObjectResult: Returns a serialized object (e.g., JSON, XML) based on the request's Accept header.

Views

public ActionResult Index()

{

List<Product> p = new List<Product>() {

new Product { Id = 1, Name = "Pen", Price = 300 },

new Product { Id = 2, Name = "Pencil", Price = 100 }

};

return View(p);

}

View

<table class="table">

<tr>

<th>

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

th>

<th>

@Html.DisplayNameFor(model => model.Price)

th>

<th>th>

tr>

@foreach (var item in Model) {

<tr>

<td>

@Html.DisplayFor(modelItem => item.Name)

td>

<td>

@Html.DisplayFor(modelItem => item.Price)

td>

<td>

@Html.ActionLink("Edit", "Edit", new { id=item.Id }) |

@Html.ActionLink("Details", "Details", new { id=item.Id }) |

@Html.ActionLink("Delete", "Delete", new { id=item.Id })

td>

tr>

}

</table>

Now we have a requirement to pass data (other than a model) to the view from the controller. There are two possible ways data can be passed.

ViewBag

We can pass data using the ViewBag as shown in the listing below:

public ActionResult Index()

{

ViewBag.data1 = "I am ViewBag data";

return View(p);

}

On the view, ViewBag data can be read as the property of the ViewBag as shown in the listing below:

<h2>@ViewBag.data1</h2>

* It’s a property of the dynamic type.
* Data is passed as a property of the object.
* There is no need of typecasting to read the data.
* There is no need for null checking.
* Life of ViewBag is restricted to the current request and becomes Null on redirection.
* ViewBag is a property of ControllerBase class.

ViewData

We can pass data using the ViewData as shown in the listing below:

public ActionResult Index()

{

ViewData["data1"] = "I am ViewBag data";

return View(p);

}

On the view, ViewData data can be read as the string value pair of the ViewData as shown in the listing below:

<h2>@ViewData["data1"]h2>

Let us examine the differences between ViewData and ViewBag. ViewBag is a dynamic property which is based on the dynamic type, whereas ViewData is a dictionary object. We can read data from ViewBag as a property and from ViewData as a key-value pair. Some bullet points about both are as follows:

* It’s a property of type ViewDataDictionary class.
* Data can be passed in the form of a key-value pair.
* To read the complex type data on the view, typecasting is required.
* To avoid the exception, null checking is required.
* Life of ViewData is restricted to the current request and becomes Null on redirection.
* ViewData is a property of the ControllerBase class

TempData

One of the major attributes of both ViewData and ViewBag are that their lifecycle is limited to one HTTP request. On redirection, they lose the data. We may have another scenario to pass data from one HTTP request to the next HTTP request; for example, passing data from one controller to another controller or one action to other action. TempData is used to pass data from one request to the next request.

Let us say that we want to navigate to Read action from Index action and while navigating, pass data to the Read action from the Index action.  So in the Index action, we can assign a value to TempData as shown in the listing below:

public ActionResult Index()

{

TempData["data1"] = "I am from different action";

return RedirectToAction("Read");

}

We can read TempData as a key-value pair. In the Read action, TempData can be read as shown in the listing below:

public string Read()

{

string str;

str = TempData["data1"].ToString();

return str;

}

Like ViewData, TempData is also a dictionary object and to read the data, typecasting and null checking is required. Keep in mind that TempData can persist data only to the subsequent HTTP request. When you are very sure about the redirection, then use TempData to pass the data.

Some points about TempData are as follows:

* TempData is used to pass data from one HTTP request to next HTTP request.
* In other words, TempData is used to pass data from one controller to another controller or action to another action.
* TempData is a property of BaseController class.
* TempData stores data in a session object
* TempData is a property of ControllerBase class
* To read data,  Typecasting and null checking are required.
* Type of TempData is TempDataDictionary.
* TempData works with HTTP redirection like HTTP 302/303 status code

Фильтры

IAuthorizationFilter

Запускается первым из всех. Проверяет, есть ли у юзера все права для доступа к методу. (проверяет авторизацию)

IResourceFilter

wЗапускается вторым.

Методы

OnResourceExecuting - код в нем запустится после AuthorizationFilter, но до других фильтров.

OnResourceExecuted - выполнится последним, после выполнения всего пайплайна приложения.

IActionFilter

OnActionExecuting - вызвется во время выполнения Action-а

OnActionExecuted - после выполнения

IPageFilter

IExceptionFilter

Фильтрует ошибки до того, как они заносятся в тело ответа.

Может отловить все ошибки кроме авторизационных.

IResultFilter

Запускается сразу до и после Action метода.

Сам фильтр запускается только если Action метод выполнен успешно

Глобальные фильтры выполняются первыми (заданные в Startup),

затем фильтры контроллера,

затем фильтры Action-а

Для смены очередности фильтров, необходимо в фильтре реализовать интерфейс IOrderedFilter

В нем public int Order => <номер>

Свой фильтр

public class SimpleResourceFilter : Attribute, IResourceFilter

{

public void OnResourceExecuting(ResourceExecutingContext context)

{

var a = context.ActionArguments["name"];

if (a == "test")

<do something>

context.HttpContext.Response.Cookies.Append("LastVisit", DateTime.Now.ToString("dd/MM/yyyy hh-mm-ss"));

}

public void OnResourceExecuted(ResourceExecutedContext context)

{

// реализация отсутствует

}

}

Глобальные фильтры

public void ConfigureServices(IServiceCollection services)

{

// глобально - все сервисы MVC - и контроллеры, и Razor Page

services.AddMvc(options =>

{

options.Filters.Add(typeof(SimpleResourceFilter)); // подключение по типу

// альтернативный вариант подключения

//options.Filters.Add(new SimpleResourceFilter()); // подключение по объекту

});

}

Установка отдельно для контроллеров:

public void ConfigureServices(IServiceCollection services)

{

services.AddControllersWithViews(options=>

{

options.Filters.Add(new SimpleResourceFilter()); // подключение по объекту

options.Filters.Add(typeof(SimpleResourceFilter)); // подключение по типу

});

}

Чтобы прервать пайплайн фильтров, надо присвоить значение Result

public void OnResourceExecuting(ResourceExecutingContext context)

{

context.Result = new ContentResult { Content = "Ресурс не найден" };

}

Конвейер обработки запроса и middleware

Компоненты конвейера, которые отвечают за обработку запроса, называются middleware.

Для подключения компонентов middleware используется метод Configure из класса Startup.

Компоненты middleware конфигурируются с помощью методов расширений Run, Map и Use объекта IApplicationBuilder, который передается в метод Configure() класса Startup

Свой middleware

Класс middleware должен иметь конструктор, который принимает параметр типа RequestDelegate. Через этот параметр можно получить ссылку на тот делегат запроса, который стоит следующим в конвейере обработки запроса.

Также в классе должен быть определен метод, который должен называться либо Invoke, либо InvokeAsync. Причем этот метод должен возвращать объект Task и принимать в качестве параметра контекст запроса - объект HttpContext. Данный метод собственно и будет обрабатывать запрос.

public class TokenMiddleware

{

private readonly RequestDelegate \_next;

public TokenMiddleware(RequestDelegate next)

{

this.\_next = next;

}

public async Task InvokeAsync(HttpContext context)

{

var token = context.Request.Query["token"];

if (token!="12345678")

{

context.Response.StatusCode = 403;

await context.Response.WriteAsync("Token is invalid");

}

else

{

await \_next.Invoke(context); <---------- Вызов следующего мидлвеера

}

}

}

public class Startup

{

public void Configure(IApplicationBuilder app)

{

app.UseMiddleware<TokenMiddleware>(); <------------- добавление в пайплайн

app.Run(async (context) =>

{

await context.Response.WriteAsync("Hello World");

});

}

}

Filter vs Middleware

Middleware operate on the level of ASP.NET Core and can act on every single request that comes in to the application.

Filters only run for requests that come to the endpoint that has the filter attached to it.

Отличия MVC от ASP .NET

The HttpRequestMessage exists only inside the "hosting" environment, web browser client does not know anything about that.

In the "hosting" world, IIS app pool is running the code you have built and deployed which knows very well wich framewok you are using as your code also contains the using assemblies you listed, System.Web... or System.Net...

Consider that even if you have shown separation between hosting, Controller and Action, all of that is running in same App Pool in IIS which, again, runs your code so knows what it is about as your IL assemblies were built from your specific source code.

IActionResult

Объекты типа IActionResult непосредственно предназначены для генерации результата действия.

Интерфейс IActionResult находится в пространстве имен Microsoft.AspNetCore.Mvc и определяет один метод:

public interface IActionResult

{

Task ExecuteResultAsync(ActionContext context);

}

Свой ActionResult

public class HtmlResult : IActionResult

{

string htmlCode;

public HtmlResult(string html)

{

htmlCode = html;

}

public async Task ExecuteResultAsync(ActionContext context)

{

string fullHtmlCode = "<!DOCTYPE html><html><head>";

fullHtmlCode += "<title>Главная страница</title>";

fullHtmlCode += "<meta charset=utf-8 />";

fullHtmlCode += "</head> <body>";

fullHtmlCode += htmlCode;

fullHtmlCode += "</body></html>";

await context.HttpContext.Response.WriteAsync(fullHtmlCode);

}

}