

Корректный ASP.NET Core

Специально для любителей книг из серии "C++ за 24 часа" решил написать статью про ASP.NET Core.

Если вы раньше не разрабатывали под .NET или под какую-то аналогичную платформу, то смысла заходить под кат для вас нет. А вот если вам интересно узнать что такое IoC, DI, DIP, Interceptors, Middleware, Filters (то есть все то, чем отличается Core от классического .NET), то вам определенно есть смысл прочитать данную статью, так как заниматься разработкой без понимания всего этого явно не корректно.

IoC, DI, DIP

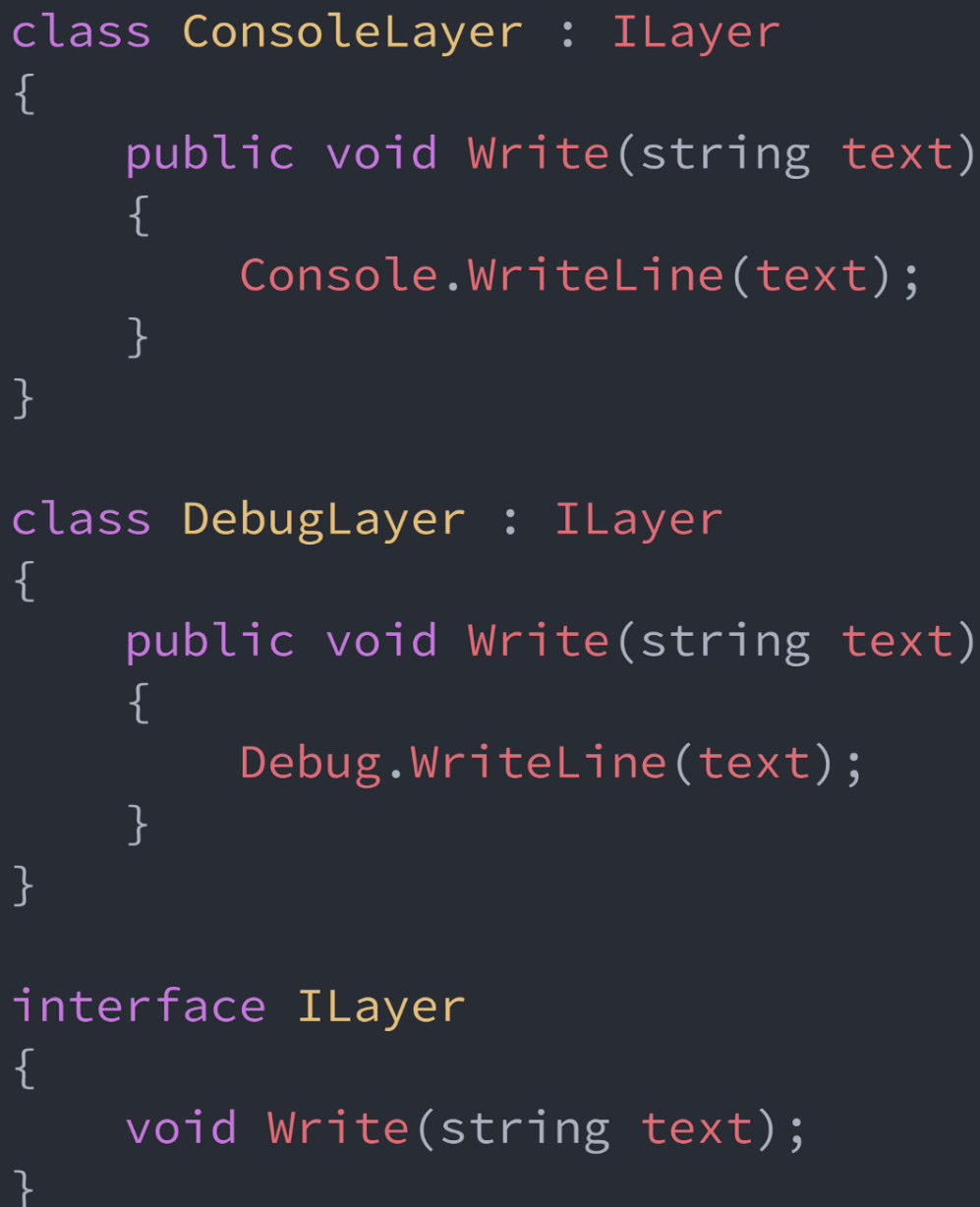
Если театр начинается с вешалки, то ASP.NET Core начинается с Dependency Injection. Для того, чтобы разобраться с DI нужно понять, что такое IoC.

Говоря о IoC очень часто вспоминают голливудский принцип "Don't call us, we'll call you". Что означает "Не нужно звонить нам мы позвоним вам сами".

Различные источники приводят различные паттерны, к которым может быть применен IoC. И скорее всего они все правы и просто дополняют друг друга. Вот некоторые из этих паттернов: factory, service locator, template method, observer, strategy.

Давайте разберем IoC на примере простого консольного приложения.

Допустим у нас есть два простых класса, реализующих интерфейс с одним методом:



```
class ConsoleLayer : ILayer
{
    public void Write(string text)
    {
        Console.WriteLine(text);
    }
}

class DebugLayer : ILayer
{
    public void Write(string text)
    {
        Debug.WriteLine(text);
    }
}

interface ILayer
{
    void Write(string text);
}
```

Они оба зависят от абстракции (в данном случае в виде абстракции выступает интерфейс).

И, допустим, у нас есть объект более высокого уровня, использующий эти классы:

```
class Logging {  
    private ILayer _instance;  
  
    public Logging (int i) {  
        if (i == 1) {  
            _instance = new ConsoleLayer ();  
        } else {  
            _instance = new DebugLayer ();  
        }  
    }  
  
    public void Write (string text) {  
        _instance.Write (text);  
    }  
}
```

В зависимости от параметра конструктора переменная `_instance` инициализируется определенным классом. Ну и далее при вызове `Write` будет совершен вывод на консоль или в `Debug`. Все вроде бы неплохо и даже, казалось бы, соответствует первой части принципа `Dependency Inversion`

Объекты более высокого уровня не зависят от объектов более низкого уровня. И те, и те зависят от абстракций.

В качестве абстракции в нашем случае выступает `ILayer`.

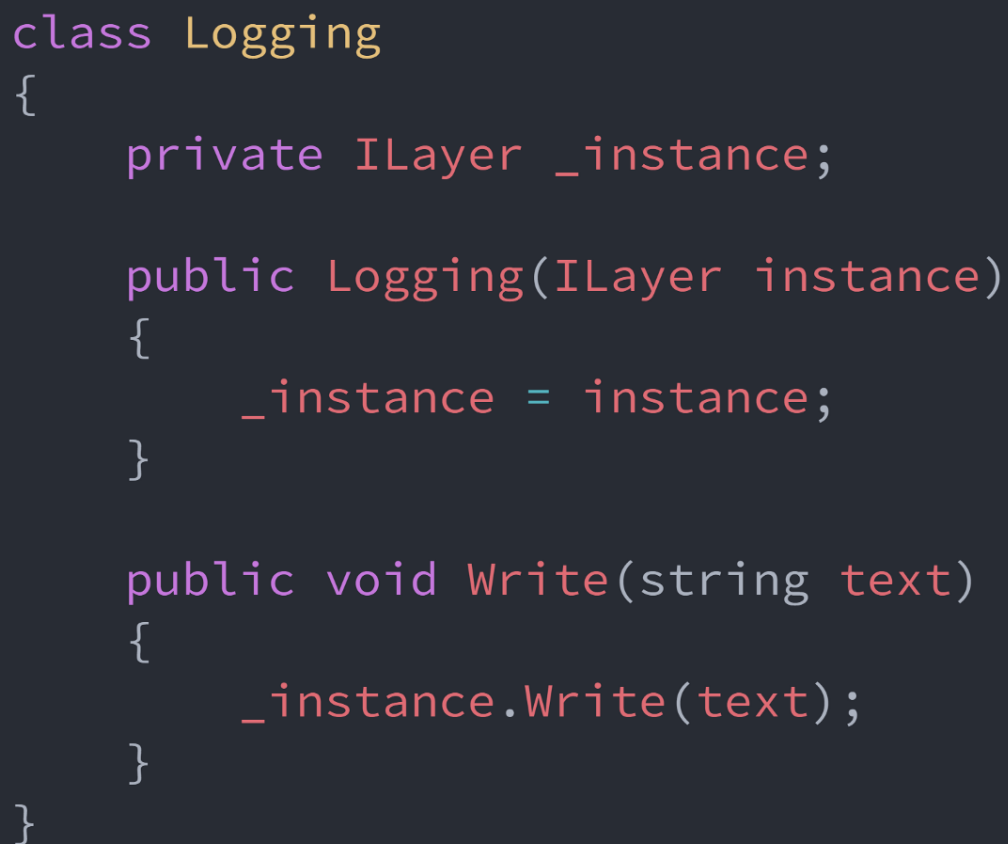
Но у нас должен быть еще и объект еще более высокого уровня. Тот, который использует класс `Logging`.

```
static void Main(string[] args)
{
    var log = new Logging(1);
    log.Write("Hello!");
    Console.Read();
}
```

Инициализируя `Logging` с помощью `1` мы получаем в классе `Logging` экземпляр класса, выводящего данные на консоль. Если мы инициализируем `Logging` любым другим числом, то `log.Write` будет выводить данные в `Debug`. Все, казалось бы, работает, но работает плохо.

Наш объект более высокого уровня `Main` зависит от деталей кода объекта более низкого уровня — класса `Logging`. Если мы в этом классе что-то изменим, то нам необходимо будет изменять и код класса `Main`. Чтобы это не происходило мы сделаем инверсию контроля — `Inversion of Control`.

Сделаем так чтобы класс `Main` контролировал то, что происходит в классе `Logging`. Класс `Logging` будет получать в виде параметра конструктора экземпляр класса, реализующего интерфейс `ILayer`.



```
class Logging
{
    private ILayer _instance;

    public Logging(ILayer instance)
    {
        _instance = instance;
    }

    public void Write(string text)
    {
        _instance.Write(text);
    }
}
```

И теперь нас класс Main будет выглядеть таким образом:

```
static void Main(string[] args)
{
    var log = new Logging(new DebugLayer());
    log.Write("Hello!");
    Console.Read();
}
```

Фактически мы декорируем наш объект Logging с помощью необходимого для нас объекта.

Теперь наше приложение соответствует и второй части принципа Dependency Inversion:

*Абстракции не зависят от деталей. Детали зависят от абстракций.
Т.е. мы не знаем деталей того, что происходит в классе Logging, мы просто передаем туда класс, реализующий необходимую абстракцию.*

Есть такой термин tight coupling — тесная связь. Чем слабее связи между компонентами в приложении, тем лучше. Хотелось бы заметить, что данный пример простого приложения немного не дотягивает до идеала. Почему? Да потому что в классе самого высокого уровня в Main у нас дважды используется создание экземпляров класса с помощью new. А есть такая мнемоническая фраза «New is a clue» — что означает чем меньше вы используете new, тем меньше тесных связей компонентов в приложении и тем лучше. В идеале мы не должны были использовать new DebugLayer, а должны были получить DebugLayer каким-нибудь другим способом. Каким? Например, из IoC контейнера или с помощью рефлексии из параметра передаваемого Main.

Теперь мы разобрались с тем, что такое Inversion of Control (IoC) и что такое принцип Dependency Inversion (DIP). Осталось разобраться с тем, что такое Dependency Injection (DI). IoC представляет собой парадигму дизайна. Dependency Injection это паттерн. Это то, что у нас теперь происходит в конструкторе класса Logging. Мы получаем экземпляр определенной зависимости (dependency). Класс Logging зависит от экземпляра класса, реализующего ILayer. И это экземпляр внедряется (injected) через конструктор.

IoC container

IoC контейнер это такой объект, который содержит в себе множество каких-то определенных зависимостей (dependency). Зависимость можно иначе назвать сервисом — как правило это класс с определенным функционалом. При необходимости из контейнера можно получить зависимость необходимого типа. Внедрение dependency в контейнер — это Inject. Извлечение — Resolve. Приведу пример самого простого самостоятельно написанного IoC контейнера:

t.me/webb_dev

```
public static class IoCContainer
{
    private static readonly Dictionary<Type, Type> _registeredObjects = new
    Dictionary<Type, Type>();

    public static dynamic Resolve<TKey>()
    {
        return Activator.CreateInstance(_registeredObjects[typeof(TKey)]);
    }

    public static void Register<TKey, TConcrete>() where TConcrete : TKey
    {
        _registeredObjects[typeof(TKey)] = typeof(TConcrete);
    }
}
```

Всего дюжина строчек кода, а уже можно использовать (не в продакшн, разумеется, а в учебных целях).

Зарегистрировать зависимость (допустим, ConsoleLayer или DebugLayer которые мы использовали в прошлом примере) можно так:

t.me/webb_dev

```
IoCContainer.Register<ILayer, ConsoleLayer>();
```

А извлечь из контейнера в необходимом месте программы так:

```
ILayer layer = IoCContainer.Resolve<ILayer>();  
layer.Write("Hello from IoC!");
```

В реальных контейнерах еще реализуется и `Dispose()`, позволяющий уничтожать ставшие ненужными ресурсы.

Кстати, имя IoC контейнер не совсем точно передает смысл, так как термин IoC гораздо шире по применению. Поэтому в последнее время все чаще применяется термин DI контейнер (так как все-таки применяется *dependency injection*).

Service lifetimes + various extension methods in Composition Root

Приложения ASP.NET Core содержат файл `Startup.cs` который является отправной точкой приложения, позволяющей настроить DI. Настраивается DI в методе `ConfigureServices`.

```
public void ConfigureServices(IServiceCollection services)  
{  
    services.AddScoped<ISomeRepository, SomeRepository>();  
}
```

Этот код добавит в DI контейнер класс `SomeRepository`, реализующий интерфейс `ISomeRepository`. То, что сервис добавлен в контейнер с помощью `AddScoped` означает, что экземпляр класса будет создаваться при каждом запросе страницы.

Добавить сервис в контейнер можно и без указания интерфейса.

Но такой способ не рекомендуется, так как у вашего приложения теряется гибкость и появляются тесные связи. Рекомендуется всегда указывать интерфейс, так как в таком

случае в любой момент времени можно заменить одну реализацию интерфейса другой. И если реализации поддерживают принцип Liskov substitution, то «легким движением руки» сменив название класса реализации вы измените и функционал всего приложения.

Есть еще 2 варианта добавить сервис — `AddSingleton` и `AddTransient`.

При использовании `AddSingleton` сервис создается один раз и при использовании приложения обращение идет к одному и тому же экземпляру. Использовать этот способ нужно особенно осторожно, так как возможны утечки памяти и проблемы с многопоточностью.

У `AddSingleton` есть небольшая особенность. Он может быть инициализирован либо при первом обращении к нему.

t.me/webb_dev

```
services.AddSingleton<IYourService, YourService>();
```

либо сразу же при добавлении в конструктор

t.me/webb_dev

```
services.AddSingleton<IYourService>(new YourService(param));
```

Вторым способом можно даже добавить параметр в конструктор.

Если хочется добавить параметр в конструктор сервиса, добавленного не только с помощью `AddSingleton`, но и с помощью `AddTransient`/`AddScoped`, то можно использовать лямбда выражение:

```
services.AddTransient<IYourService>(o => new YourService(param));
```

Ну и наконец, при использовании `AddTransient` сервис создается каждый раз при обращении к нему. Отлично подходит для легких сервисов, не потребляющих память и ресурсы.

Если с `AddSingleton` и `AddScoped` все должно быть более-менее понятно, то `AddTransient` требует разъяснений. Официальная документация приводит пример, в котором определенный сервис добавлен в DI контейнер и в качестве параметра конструктора другого сервиса и отдельно самостоятельно. И вот в случае, если он добавлен отдельно с помощью `AddTransient`, он создает свой экземпляр 2 раза. Приведу очень-очень упрощенный пример. В реальной жизни к применению не рекомендуется, т.к. классы для упрощения не наследуют интерфейсы. Допустим у нас есть простой класс:

t.me/webb_dev

```
public class Operation
{
    public Guid OperationId { get; private set; }

    public Operation()
    {
        OperationId = Guid.NewGuid();
    }
}
```

И есть второй класс, который содержит первый как зависимый сервис и получает эту зависимость в качестве параметра конструктора:

```
public class OperationService
{
    public Operation Operation { get; }

    public OperationService (Operation operation)
    {
        Operation = operation;
    }
}
```

Теперь совершаем inject двух сервисов:

```
services.AddTransient<Operation>();
services.AddScoped<OperationService>();
```

И в каком-нибудь контроллере в Action добавим получение наших зависимостей и вывод значений в окно Debug.

```

public IActionResult Index([FromServices] Operation operation,
[FromServices] OperationService operationService)
{
    Debug.WriteLine(operation.OperationId);
    Debug.WriteLine(operationService.Operation.OperationId);

    return View();
}

```

Так вот в результате мы получим 2 разных значения Guid. А вот если мы заменим AddTransient на AddScoped, то в результате мы получим 2 одинаковых значения.

В IoC контейнере приложений ASP.NET Core по умолчанию содержатся уже некоторые сервисы. Например, IConfiguration — сервис с помощью которого можно получить настройки приложения из файлов appsettings.json и appsettings.Development.json. IHostingEnvironment и ILoggerFactory с помощью которых можно получить текущую конфигурацию и вспомогательный класс, позволяющий проводить логирование.

Извлекают классы из контейнера с помощью следующей типичной конструкции (самый банальный пример):

```

private readonly IConfiguration _configuration;

public SomePageController(IConfiguration configuration)
{
    _configuration = configuration;
}

public async Task<IActionResult> Index()
{
    string connectionString = _configuration["connectionString"];
}

```

В области видимости контроллера создается переменная с модификаторами доступа `private readonly`. Зависимость получается из контейнера в конструкторе

класса и присваивается приватной переменной. Далее эту переменную можно использовать в любых методах или Action контроллера.

Иногда не хочется создавать переменную для того, чтобы использовать ее только в одном Action. Тогда можно использовать атрибут `[FromServices]`. Пример:

t.me/webb_dev

```
public IActionResult About([FromServices] IDateTime dateTime)
{
    ViewData["Message"] = «Московское время " + dateTime.Now;
    return View();
}
```

Выглядит странно, но для того, чтобы в коде не вызывать метод статического класса `DateTime.Now()` иногда делают так, что значение времени получается из сервиса в качестве параметра. Таким образом появляется возможность передать любое время в качестве параметра, а значит становится легче писать тесты и, как правило, становится проще вносить изменения в приложение.

Нельзя сказать, что `static` — это зло. Статические методы выполняются быстрее. И скорее всего `static` может использоваться где-то в самом IoC контейнере. Но если мы избавим наше приложение от всего статического и `new`, то получим большую гибкость.

Сторонние DI контейнеры

То, что мы рассматривали и то, что фактически реализует ASP.NET Core DI контейнер по умолчанию — `constructor injection`. Имеется еще возможность внедрить зависимость в `property` с помощью так называемого `property injection`, но эта возможность отсутствует у встроенного в ASP.NET Core контейнера. Например, у нас может быть какой-то класс, который мы внедряем как зависимость, и у этого класса есть какое-то `public property`. Теперь представьте себе, что во время или после того как мы внедряем зависимость, нам нужно задать значение `property`. Вернемся к примеру похожему на пример, который мы недавно рассматривали.

Если у нас есть такой вот класс:

```
public class Operation
{
    public Guid OperationId { get; set; }
    public Operation() {}
}
```

который мы можем внедрить как зависимость,

```
services.AddTransient<Operation>();
```

то используя стандартный контейнер задать значение для свойства мы не можем.

Если вы захотите использовать такую возможность задать значение для свойства `OperationId`, то вы можете использовать какой-то сторонний DI контейнер, поддерживающий **property injection**. К слову сказать **property injection** не особо рекомендуется использовать. Однако, существуют еще **Method Injection** и **Setter** **Method Injection**, которые вполне могут вам пригодиться и которые также не поддерживаются стандартным контейнером.

У сторонних контейнеров могут быть и другие очень полезные возможности. Например, с помощью стороннего контейнера можно внедрять зависимость только в контролеры, у которых в названии присутствует определенное слово. И довольно часто используемый кейс — DI контейнеры, оптимизированные на быстродействие. Вот список некоторых сторонних DI контейнеров, поддерживаемых ASP.NET Core: Autofac, Castle Windsor, LightInject, DryIoC, StructureMap, Unity.

Хоть при использовании стандартного DI контейнера и нельзя использовать property/method injection, но зато можно внедрить зависимый сервис в качестве параметра конструктора реализовав паттерн «Фабрика» следующим образом:

t.me/webb_dev

```
services.AddTransient<IDataService, DataService>((dsvc) =>
{
    IOtherService svc = dsvc.GetService<IOtherService>();
    return new DataService(svc);
});
```

В данном случае GetService вернет null если зависимый сервис не найден. Есть вариация GetRequiredService, которая выбросит исключение в случае, если зависимый сервис не найден.

Процесс получения зависимого сервиса с помощью GetService фактически применяет паттерн Service locator.

Middleware

В ASP.NET существует определенная цепочка вызовов кода, которая происходит при каждом request. Еще до того, как загрузился UI/MVC выполняются определенные действия.

То есть, например, если мы добавим в начало метода `Configure` класса `Startup.cs` код:

t.me/webb_dev

```
app.Use(async (context, next) =>
{
    Debug.WriteLine(context.Request.Path);
    await next.Invoke();
});
```

То мы сможем посмотреть в консоли дебага какие файлы запрашивает наше приложение. Фактически мы получаем возможности AOP "out of box". Немного useless, но понятный и познавательный пример использования `middleware` я вам сейчас покажу:

t.me/webb_dev

```
public void Configure(IApplicationBuilder app)
{
    app.Use(async (context, next) =>
    {
        await context.Response.WriteAsync("Hello!" + Environment.NewLine);
        await next.Invoke();
    });

    app.Run(async context =>
    {
        await context.Response.WriteAsync("Hello again.");
    });
}
```

При каждом запросе начинает выполняться цепочка вызовов. Из каждого `app.Use()` после вызова `next.invoke()` совершается переход ко следующему вызову. И все завершается после того как отработает `app.Run()`.

Можно выполнять какой-то код только при обращении к определенному `route`.

Сделать это можно с помощью `app.Map()`:


```

private static void Goodbye(IApplicationBuilder app)
{
    app.Run(async context =>
    {
        await context.Response.WriteAsync("Goodbye!");
    });
}

public void Configure(IApplicationBuilder app)
{
    app.Map("/goodbye", Goodbye);
    app.Run(async context =>
    {
        await context.Response.WriteAsync("Hello!");
    });
}

```

Теперь если просто перейти на страницу сайта, то можно будет увидеть текст "Hello!", а если добавить к строке адреса /Goodbye, то вам будет отображено Goodbye.

Кроме `Use` и `Map` можно использовать `UseWhen` или `MapWhen` для того, чтобы добавлять код в цепочку middleware только при каких-то определенных условиях.

До сих пор были все еще useless примеры, правда? Вот вам нормальный пример:

```

app.Use(async (context, next) =>
{
    context.Response.Headers.Add("X-Frame-Options", "DENY");
    context.Response.Headers.Add("X-Content-Type-Options", "nosniff");
    context.Response.Headers.Add("X-Xss-Protection", "1");
    await next();
});

```

Здесь мы добавляем к каждому запросу заголовки, помогающие обезопасить страницу от хакерских атак.

Или же вот пример локализации:

t.me/webb_dev

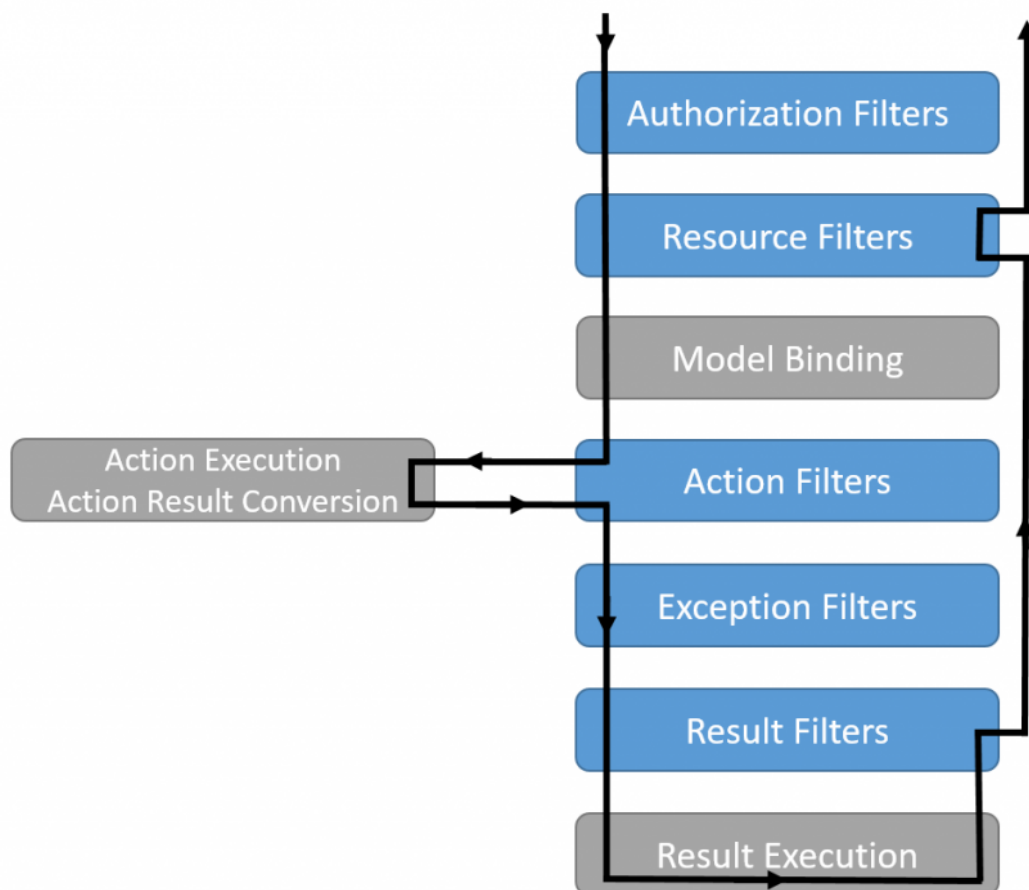
```
var supportedCultures = new[]
{
    new CultureInfo("ru"),
    new CultureInfo("fr")
};

app.UseRequestLocalization(new RequestLocalizationOptions
{
    DefaultRequestCulture = new RequestCulture("ru"),
    SupportedCultures = supportedCultures,
    SupportedUICultures = supportedCultures
});
```

Теперь если вы к адресу страницы добавите параметр `?culture=fr` то вы сможете переключить язык приложения на французский (если в ваше приложение добавлена локализация, то все сработает).

Filters

Если цепочка middleware относятся к процессам до MVC, то фильтры работают вместе с MVC. Следующее схематическое изображение показывает, как отрабатывают фильтры.



Сначала отработывают фильтры авторизации. Т.е. можно создать какой-то фильтр или несколько фильтров и вписать в них какой-то свой код авторизации, который будет отработывать при запросах.

Затем отработывают фильтры ресурсов. С помощью этих фильтров можно, например, вернуть какую-то информацию из кеша.

Затем происходит привязка данных и выполняются Action фильтры. С их помощью можно манипулировать параметрами передаваемыми Action и возвращаемым результатом.

Exception фильтры как намекает название позволяют добавить какую-то общую обработку ошибок для приложения. Должно быть довольно удобно обрабатывать ошибки везде одинаково. Эдакий AOP-шный плюс.

Result фильтры позволяют совершить какие-то действия до выполнения Action контроллера или после. Они довольно похожи на Action фильтры, но выполняются только в случае отсутствия ошибок. Подходят для логики завязанной на View.

Все должно быть более понятно на примере. А у меня под рукой как раз есть пример упрощенного авторизационного фильтра:

```
public class YourCustomFilter : Attribute, IAuthorizationFilter
{
    public async void OnAuthorization(AuthorizationFilterContext context)
    {
        // какая-то логика и в случае, если у
        // пользователя нет прав, можно сделать следующее
        context.Result = new ContentResult()
        {
            Content = "У вас нет прав для доступа к этому ресурсу"
        };
    }
}
```

Добавляете этот класс в DI контейнер (как обычно в Startup.cs).

```
services.AddScoped<YourCustomFilter>();
```

И теперь становится возможным добавить какую-то свою авторизацию любому Action добавив следующий атрибут:

```
[ServiceFilter(typeof(YourCustomFilter))]
```

Забавная штука — можно создать свое middleware и добавлять его каким-то action в качестве фильтра. Для того чтобы сделать так нужно создать класс с произвольным

названием и методом `Configure`.

t.me/webb_dev

```
public class MyMiddlewareFilter
{
    public void Configure(IApplicationBuilder applicationBuilder)
    {
        applicationBuilder.Use(async (context, next) =>
        {
            Debug.WriteLine("Привет от middleware!");
            await next.Invoke();
        });
    }
}
```

Теперь этот класс можно добавлять Action-ам с помощью следующего атрибута.

t.me/webb_dev

```
[MiddlewareFilter(typeof(MyMiddlewareFilter))]
```

[#DI](#), [#IOC](#), [#DIP](#), [#asp](#), [#зависимость](#), [#инжекция](#)