Dependency Injection in action filters in ASP.NET Core

It is quite common to decorate ASP.NET MVC controller actions with filter attributes to separate cross cutting concerns from the main concern of the action. Sometimes these filters need to use other components but attributes are quite limited in their functionality and dependency injection into an attribute is not directly possible. This post looks at a few different techniques for injecting dependencies into action filters in ASP.NET Core. We discuss when each method should be used before taking a step back and examining if we can approach the problem in a different way for a cleaner solution.

DI and Action Filter Attributes

This article discusses action filters as an example because they are the most widely used type of filter in ASP.NET MVC. Most of what is discussed applies equally well to other types of filter such as Authorization, Resource or Result.

Attributes in C# are very simple. You can pass static values into a constructor and/or set public properties directly.

Because attribute parameters are evaluated at compile time, they have to be compile time constants. Therefore injecting dependencies from an IoC container is not an option. In situations where we want an attribute to have access to another component, we must use a workaround. Let's look at a few options:

The RequestServices.GetService Service Locator

Note that this technique is not recommended but we will discuss it here because it is by far the most widely used option, particularly in previous versions of ASP.NET MVC.

If we cannot inject a component into our attribute, it seems as though the next best option is to request the component from our IoC container (either directly or via a wrapper).

In .NET Core, we can use service location to resolve components from the built-in IoC container by using RequestServices.GetService:

```
    public void OnActionExecuting(ActionExecutingContext context)
    {
        var cache = context.HttpContext.RequestServices.GetService<IDistributedCache>
();
        ...
    }
    ...
}
```

Note that you need to add the following using statement in order to use the generic version of GetService:

```
using Microsoft.Extensions.DependencyInjection;
```

This will work but it is not recommended. Unlike constructor injection, it can be much harder to work out your dependencies with the service locator (anti)pattern. The beauty of constructor injection is its simplicity. Just by looking at a constructor definition, I immediately know all classes on which the class depends. Service location also makes unit testing harder requiring you to have far more knowledge about the internals of the subject under test.

In previous version of ASP.NET MVC, service location inside filters was more acceptable because of the lack of a viable alternative. .NET Core addresses this with several new alternatives:

The ServiceFilter attribute

The ServiceFilter attribute can be used at the action or controller level. Usage is very straightforward:

```
[ServiceFilter(typeof(ThrottleFilter))]
```

The ServiceFilter attribute allows us to specify the type of our action filter and have it automatically resolve the class from the built-in IoC container. This means that we can change our action filter to accept dependencies directly via the constructor:

```
public class ThrottleFilter : IActionFilter
{
    private readonly IDistributedCache _cache;

    public ThrottleFilter(IDistributedCache cache)
    {
        _cache = cache ?? throw new ArgumentNullException(nameof(cache));
    }
```

Obviously, as we are resolving our filter from the IoC container, we need to register it:

```
public void ConfigureServices(IServiceCollection services)
{
    ...
    services.AddScoped<ThrottleFilter>();
    ...
}
```

This looks much more like the DI that we are used to but it is not perfect. The main problem with this approach is that it is not possible to specify any configuration for the filter, other than any configuation registered with the container. The great thing about attributes is the ability to specify values on a case by case basis. If we take a caching filter as an example, we probably want to specify different cache durations for different actions. Using ServiceFilter does not allow this.

The TypeFilter attribute

ServiceFilter is very useful for attributes which have dependencies that need to be resolved from the IoC container but the lack of property support is a major limitation. If we modify our ThrottleFilter example to add a configuration property (while retaining the IDistributedCache dependency) then ServiceFilter is no longer useful to us. We can however use TypeFilter.

```
public class ThrottleFilter : IActionFilter
{
    private readonly IDistributedCache _cache;

    public int MaxRequestPerSecond { get; set; }

    public ThrottleFilter(IDistributedCache cache, int maxRequestPerSecond)
    {
        _cache = cache ?? throw new ArgumentNullException(nameof(cache));
        MaxRequestPerSecond = maxRequestPerSecond;
    }
}
```

TypeFilter is very similar to ServiceFilter but has two notable differences:

- The type being resolved does not need to be registered with the IoC container (but any dependencies do)
- Arguments can be supplied which are used when constructing the filter

constructor. Unfortunately due to the generic nature of the filter, the syntax for doing so is terrible and you have no type safety:

```
[TypeFilter(typeof(ThrottleFilter), Arguments = new object[] { 10 })]
```

I therefore tend to avoid this option and cannot recommend it.

Custom Filter Factories

Both ServiceFilter and TypeFilter are examples of a built-in IFilterFactory implementation but you can easily write your own. This is ideal for situations where you require both component based dependencies and custom configuration of the attribute. It can also be used for situations where you do not want to register filters and/or dependencies with the IoC container.

In the previous section, we added maxRequestPerSecond to the ThrottleFilter constructor in order to use the TypeFilter attribute but this is not required for our own filter factory implementation so we'll remove it and use the MaxRequestPerSecond property to override a default value:

```
public class ThrottleFilter : IActionFilter
{
    private const int DefaultMaxRequestPerSecond = 3;

    private readonly IDistributedCache _cache;

    public int MaxRequestPerSecond { get; set; } = DefaultMaxRequestPerSecond;

    public ThrottleFilter(IDistributedCache cache)
    {
        _cache = cache ?? throw new ArgumentNullException(nameof(cache));
    }
    ...
}
```

The filter factory simply resolves the filter from the IoC container and then conditionally sets the configuration property if it has a value:

```
public class ThrottleFilterFactory : Attribute, IFilterFactory
{
    public int MaxRequestPerSecond { get; set; }

    public bool IsReusable => false;

    public IFilterMetadata CreateInstance(IServiceProvider serviceProvider)
```

```
        if (MaxRequestPerSecond > 0)
        {
            filter.MaxRequestPerSecond = MaxRequestPerSecond.Value;
        }

        return filter;
    }
}
```

If the code above looks suspiciously like a service locator then you are not wrong but the difference here is that the filter factory can be thought of as infrastructure code, leaving the filter itself clean. This is debatable however.

The usage of the filter factory is very clean and simple:

```
[ThrottleFilterFactory(MaxRequestPerSecond = 10)]
public IActionResult Example()
```

Global action filters

So far we have only talked about filters implemented as attributes but you can also apply filters globally:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
        options.Filters.Add<ThrottleFilter>();
    });
}
```

This allows us to resolve the filter from the IoC container in the same way as using the ServiceFilter attribute on the action or controller but it will be applied to every action in every controller instead. If you need to be able to configure individual actions however then you can't use global filters in isolation.

Passive Attributes

If we think about it, it doesn't really make sense for action filter attributes to house complex logic and interactions with other components. Attributes are great for identification - choosing which actions or controllers. They should probably not be full of complex code.

Perhaps a better approach would be to restrict attributes to what they are good at and use a global (non-attribute) action filter to deal with the actual implementation. This is nothing new but I rarely see it implemented. I came across the concept a few years ago where Mark Seeman termed it passive attributes.

The idea is that you use attributes to opt in to (or out of) the functionality of a global filter. The attributes can also contain configuration which can be read by the global filter. Using our example, we would have a simple ThrottleAttribute which is applied to actions and a ThrottleFilter which is registered globally.

The presence of the ThrottleAttribute on an action indicates that it should be throttled. This is an opt-in approach but we could just as easily use an opt-out approach where an attribute indicates that we do not want to throttle the action.

We also want to be able to optionally specify a max requests per second. The implementation is trivial:

```
public class ThrottleAttribute : Attribute, IFilterMetadata
{
    public int MaxRequestPerSecond { get; set; }
}
```

Note that we implement IFilterMetadata. This is a marker interface that makes it much easier to read our attribute from the global filter.

The global action filter is registered with MVC in exactly the same way as we have shown at the start of the section. All dependencies are automatically resolved via the built-in IoC container. The skeleton ThrottleFilter would look similar to:

```
public class ThrottleFilter : IActionFilter
{
    private readonly IDistributedCache _cache;

    public ThrottleFilter(IDistributedCache cache)
    {
        _cache = cache ?? throw new ArgumentNullException(nameof(cache));
    }

    public void OnActionExecuted(ActionExecutedContext context)
    {
        //noop
    public void OnActionExecuting(ActionExecutingContext context)
    {
        var throttleAttribute = context.ActionDescriptor.FilterDescriptors
            .Select(x => x.Filter).OfType<ThrottleAttribute>().FirstOrDefault();

        if (throttleAttribute != null)
        {
            // implementation with access to throttleAttribute.MaxRequestPerSecond
value
        }
    }
}
```

I really like this approach. Our attribute is small and typesafe and does not require any DI. Our global action filter is also typesafe and can be automatically resolved from the container. The only slight negative is the fact that the action filter runs for all actions which many involve a tiny performance penalty for those actions which do not require throttling.

Middleware

In some situations it can be beneficial to take it one level higher and use middleware instead of a global filter. We used this approach in the last post when we implemented custom response caching. Dependency injection into middleware happens out of the box and all you need to do is register any dependencies with the built-in container.

Our example caching middleware can short-circuit the request pipeline so MVC is not even invoked. We return the cached page directly from the middleware for maximum performance. If the page is not cached however then we need to let the MVC action execute and capture the response.

The slight difficulty with using middleware over MVC filters is the fact that because we are outside the MVC pipeline, we do not have access to any attributes on the actions or controllers so configuring caching on individual actions requires a little work. We could potentially use reflection for this but a simpler method is to take advantage of the HttpContext.Items collection. The last article explains this more throughly but in summary, you can use very basic action filters to write configuration values to the HttpContext.Items collection. These values can then be read easily from your middleware:

```
// write to the items collection in our action filter attribute
public override void OnActionExecuting(ActionExecutingContext context)
{
    context.HttpContext.Items["Example"] = "Value from attribute";
}
```

```
// read from items collection in our middleware
context.Items.TryGetValue("Example", out string whatever)
```

Conclusion

Injecting components into action filter attributes directly is not possible but there are various workarounds to allow us to effectively accomplish the same thing. Using ServiceFilter is a relatively clean way to allow dependency injection into individual action filters. Specifying the type for the filter this way does mean that invalid types can be entered and will not be discovered until runtime however.

If you need to use attribute values to configure the action filter then ServiceFilter cannot be used. The TypeFilter alternative that does allow arbitrary values to be passed in is not recommended. The way that values are passed into the attribute (as an object array) is too error prone and likely to break during refactoring. Instead if you must take this approach then consider a custom IFilterFactory.

Before implementing any of these techniques, you should consider the passive attribute approach. In my opinion this is by far the best solution to the problem. To summarise, this method requires two classes. We have a very simple marker attribute applied to actions and this is paired with a global action filter (or middleware) which houses the actual logic.

Useful or Interesting?

If you liked the article, I would really appreciate it if you could share it with your Twitter followers.

SHARE ON TWITTER

Comments

WHOEVER WROTE ON 07 SEP 2017

In pre-core MVC Attribute IoC, it was claimed

"Web API caches filter attribute instances indefinitely per action, effectively making them singletons. This makes them unsuited for dependency injection, since the attribute's dependencies

Do you happen to know if that's still the case with Mvc Core 2?

Thanks.

PAUL HILES WROTE ON 11 OCT 2017

@Whoever - good question. Any attribute based filters will always be singletons but all the other techniques outlined in the article can be used to create per-request filters. This includes the use of ServiceFilter, TypeFilter, custom filter factories and global filters.

ANDREW RADFORD WROTE ON 01 AUG 2018

Thanks for the article. I was trying to do dependency injection for filters and this article did a great job of explaining all of the different strategies.

FEDERICO CALDAS WROTE ON 21 DEC 2018

Excellent article, explaining different options clearly.

YEHUDA RINGLER WROTE ON 08 JAN 2019

The global filters with passive attributes is very graceful, but there's one potential problem not mentioned:
If you have a large project, with 1000s of dependencies and filters, and, between all the filters, all or most of the dependencies are used, those dependencies will be created for every request.
If the dependency creation is heavy, such as potentially DbContext, I suspect that would make a noticeable impact on performance.

attribute, and the other takes all the dependencies and sets them to private properties.
Then, in the method which needs the dependency, I'll use null coalescing to get it, e.g.
var injectedFactory = this.clientIdentityFactory ??
context.HttpContext.RequestServices.GetService<IClientIdentityFactory>();

This allows unit testing and explicit documentative dependencies in the constructor.

If there are multiple public methods, this may get even more clunky, but for e.g an IAuthorizationFilter with one OnAuthorization method, I find it's an ok compromise.

VERY HELPFUL ! WROTE ON 01 JUL 2019

Thanks for the post. Very heplfull information!

RODRIGO WROTE ON 06 SEP 2019

Man, you just saved my life, thank you very very much

TONY DAVIS WROTE ON 10 FEB 2020

Great post!

>IFilterMetadata... makes it much easier to read our attribute from the global filter

Arguably, it's just as easy to query the EndpointMetadata:

var throttleAttribute = context
.ActionDescriptor
.EndpointMetadata
.OfType<ThrottleAttribute>()

Is there any disadvantage to this approach?

PAUL HILES WROTE ON 10 FEB 2020

@Tony - Thanks. I think EndpointMetadata was added in .NET Core 2.2 (after this article was written) but going forward, it looks like a much better alternative.

YUSUF WROTE ON 21 FEB 2020

Thank you :))

SAITEJA WROTE ON 17 MAR 2020

I really found this article helpful,thanks.

PAWEL GORALCZYK WROTE ON 02 JUL 2020

Tank You !

JAY WROTE ON 08 NOV 2020

I prefer the GetService() way by far the most. It leaves the constructor free for passing in your own parameters.

Really nice article! This is a sticky area. I needed a custom authorization attribute and luckily only need the HttpContext which is already available to the filter. However, I did need a gRPC client which I had to get using the service locator. It's bad for testing but at the same time what are you going to do about it?

Oh well, at least it works. The more I work with attributes the less I like them, debugging them is very frustrating because it's all done automagically.

LOGAN LAMOUAR WROTE ON 08 SEP 2021

It was incredibly usefull for me and my mate Christopher, thanks.

NAME

EMAIL

Used for Gravatar and never shared

COMMENT

ADD COMMENT

DevTrends BLOG ABOUT CONTACT

DevTrends is owned by Paul Hiles who has over 15 years of experience developing Microsoft based applications. MORE »

Contact

Email me at **paul@devtrends.co.uk**

Search

enter search term

SEARCH

Latest Blog Posts

Fast Free Geolocation in .NET with freegeoip.net

3 Ways To Avoid An Anemic Domain Model In Entity Framework

Installing the ASP.NET Core 2.0 runtime store on Linux

Dependency Injection in action filters in ASP.NET Core

Custom response caching in ASP.NET Core (with cache invalidation)

A guide to caching in ASP.NET Core

Hashing, Encryption and Random in ASP.NET Core

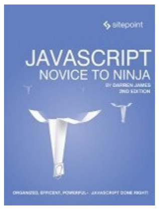
Create a Free Private NuGet Server with Continuous Deployment using VSTS

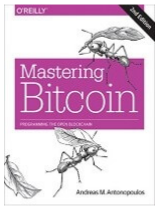
Creating your first shared library in .NET Core

Conditional Middleware based on request in ASP.NET Core

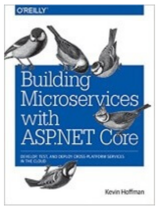
VIEW ALL BLOG POSTS »

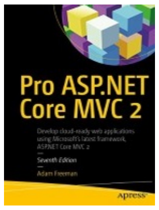
Reading List


JavaScript: Novice to Ninja 2nd Edition (2017) - Darren Jones


Mastering Bitcoin: Programming the Open Blockchain - Andreas M. Antonopoulos

Clean Architecture: A Craftsman's Guide to Software Structure and Design - Robert C. Martin


Building Microservices with ASP.NET Core - Kevin Hoffman


Pro ASP.NET Core MVC 2 - Adam Freeman