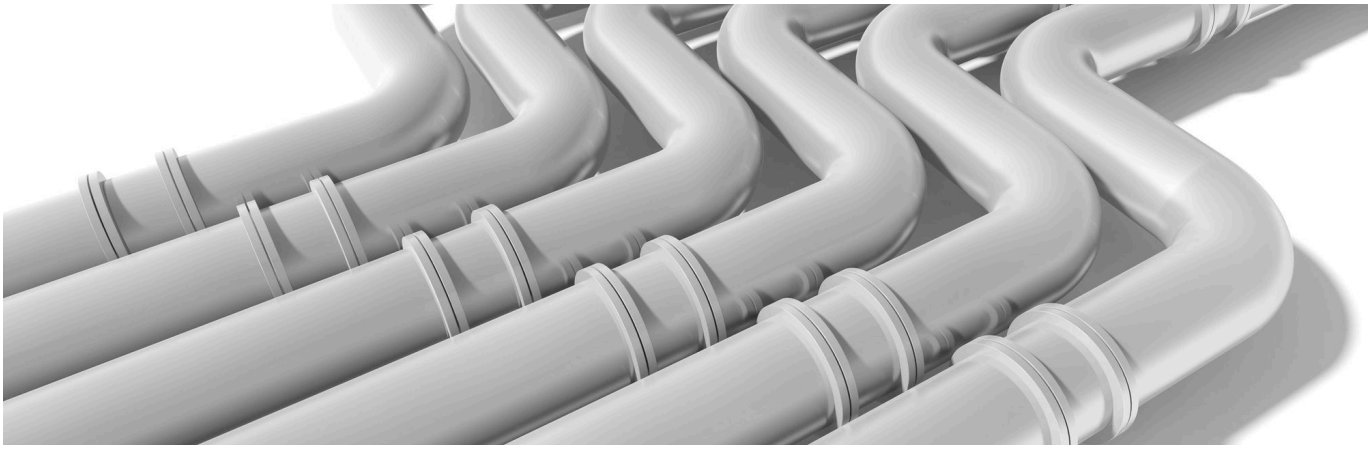
1 OCTOBER 2018 / DOTNET

ASP.NET Core: Middleware by Abstraction



With ASP.NET Core the primary way of creating middleware components is by using **convention-based** activation logic. A secondary way is to use factory-based implementation that was added with two interfaces, `IMiddlewareFactory` and `IMiddleware`.

An example of convention-based middleware class looks like this:

```
// You may need to install the Microsoft.AspNetCore.Http.Abstractions package into your project
public class ConventionalMiddleware
{
    private readonly RequestDelegate _next;

    public ConventionalMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public Task Invoke(HttpContext httpContext)
    {

        return _next(httpContext);
    }
}
```

And the factory-based implementation could look like this:

```
public class FactoryActivatedMiddleware : IMiddleware
{
    public Task InvokeAsync(HttpContext context, RequestDelegate next)
    {
        throw new NotImplementedException();
    }
}
```

However, although both ways are sleak and dandy, I wanted to improve the **convention-based** activation with two main abstractions; one for middleware components that does not need configuration - and one that requires additional setup.

Middleware with zero to five dependency injected parameters

For the first middleware abstraction, I decided to create an abstract class named `Middleware` and five additional implementations. The difference lies in the generic type parameters covering dependency injection.

Following our examples from before, we are ending up with a class like this:

```
public class ConventionalMiddleware : Middleware
{
    public ConventionalMiddleware(RequestDelegate next) : base(next)
    {
    }

    public override Task InvokeAsync(HttpContext context)
    {

        return Next(context);
    }
}
```

And if we we have one dependency on `ITest`, it could look like this:

```
public class ConventionalMiddleware : Middleware<ITest>
{
    public ConventionalMiddleware(RequestDelegate next) : base(next)
    {
    }

    public override Task InvokeAsync(HttpContext context, ITest test)
    {

        return Next(context);
    }
}
```

ConfigurableMiddleware with zero to five dependency injected parameters

For the second - and probably the most relevant middleware abstraction - I went for an abstract class named `ConfigurableMiddleware` that without dependency injection has one generic type parameter, `TOptions`.

The constructor defer a little from the first implementation; not only does it support `TOptions`, it also comes in two flavors: one that is supported by interface `IOptions<TOptions>` and one that is supported by delegate `Action<TOptions>`.

When creating the recommended extension method for `IApplicationBuilder`, do consider using the static `ApplicationBuilderFactory` class to insure correct invokation of either constructor.

To make the usage of this middleware implementation more clear, I have included a working example that is found in the `Cuemon.AspNetCore` assembly; it is the `HostingEnvironmentMiddleware` class:

```
/// <summary>
/// Provides a hosting environment middleware implementation for ASP.NET Core.
/// </summary>
public class HostingEnvironmentMiddleware : ConfigurableMiddleware<IHostingEnvironment, HostingE
nvironmentOptions>
{
    /// <summary>
    /// Initializes a new instance of the <see cref="HostingEnvironmentMiddleware"/> class.
    /// </summary>
    /// <param name="next">The delegate of the request pipeline to invoke.</param>
    /// <param name="setup">The <see cref="HostingEnvironmentOptions" /> which need to be config
ured.</param>
    public HostingEnvironmentMiddleware(RequestDelegate next, IOptions<HostingEnvironmentOptions
> setup) : base(next, setup)
    {
    }

    /// <summary>
    /// Initializes a new instance of the <see cref="HostingEnvironmentMiddleware"/> class.
    /// </summary>
    /// <param name="next">The delegate of the request pipeline to invoke.</param>
    /// <param name="setup">The <see cref="HostingEnvironmentOptions" /> which need to be config
ured.</param>
    public HostingEnvironmentMiddleware(RequestDelegate next, Action<HostingEnvironmentOptions>
setup) : base(next, setup)
    {
    }

    /// <summary>
    /// Executes the <see cref="HostingEnvironmentMiddleware" />.
    /// </summary>
    /// <param name="context">The context of the current request.</param>
    /// <param name="he">The dependency injected <see cref="IHostingEnvironment"/> of <see cref=
"InvokeAsync"/>.</param>
    /// <returns>A task that represents the execution of this middleware.</returns>
    public override Task InvokeAsync(HttpContext context, IHostingEnvironment he)
    {
        context.Response.OnStarting(() =>
        {
            if (!Options?.SuppressHeaderPredicate(he) ?? false)
            {
                context.Response.Headers.AddIfNotContainsKey(Options.HeaderName, he.EnvironmentN
ame);
            }
            return Task.CompletedTask;
        });
        return Next(context);
    }
}

/// <summary>
/// This is a factory implementation of the <see cref="HostingEnvironmentMiddleware"/> class.
/// </summary>
public static class HostingEnvironmentBuilderExtension
{
    /// <summary>
    /// Adds a hosting environment HTTP header to the <see cref="IApplicationBuilder"/> request
```

```
execution pipeline.
    /// </summary>
    /// <param name="builder">The type that provides the mechanisms to configure an application'
s request pipeline.</param>
    /// <param name="setup">The <see cref="CorrelationIdentifierOptions"/> middleware which need
 to be configured.</param>
    /// <returns>A reference to this instance after the operation has completed.</returns>
    public static IApplicationBuilder UseHostingEnvironmentHeader(this IApplicationBuilder build
er, Action<HostingEnvironmentOptions> setup = null)
    {
        return ApplicationBuilderFactory.UseMiddlewareConfigurable<HostingEnvironmentMiddleware,
 HostingEnvironmentOptions>(builder, setup);
    }
}
```

As you can see, this implementation has two generic type parameters; the first is for dependency injection (in this case, we depend on an implementation of `IHostingEnvironment`) and the other is the required setup of `HostingEnvironmentOptions`.

The rest of the code should be rather self-explanatory, but do take note of the extension method associated with this middleware.

Closing Words

This was some insights to Middleware and ConfigurableMiddleware. I hope it brought something new for you to enjoy and triggered your curiosity to try it out in your projects.

Code with passion 🔥