Use ASP.NET Core hosted services to run a background task

Pal Chouhan · Follow
4 min read · May 9, 2022

Hosted services were introduced in ASP.NET Core 3.1, and are an excellent way of running background tasks.

They can be ran in an ASP.NET Core web application. This is ideal if we need to update something that runs in the background that could effect all users.

Alternatively, they can be run using a Worker Service template. This is designed to run a background service and can be set up as a Windows Service.

In this article we will be focusing on adding a hosted service to an ASP.NET Core application. We'll take a look at how to create one, how to add it to the `IServiceCollection` instance and how we can use **dependency injection** within it.

Creating a hosted service

There are a couple of ways of how we can created a hosted service. The first way we can do it is to inherit the `IHostedService` interface. Within that, we must implement the `StartAsync` and `StopAsync` methods into our class.

```
using Microsoft.Extensions.Hosting;

namespace RoundTheCode.HostedServiceExample
{
    public class MyHostedService : IHostedService
    {
        public Task StartAsync(CancellationToken cancellationToken)
        {
            return Task.CompletedTask;
        }

        public Task StopAsync(CancellationToken cancellationToken)
        {
            return Task.CompletedTask;
        }
    }
}
```

We can then kick off our tasks in the `StartAsync` method. However, one thing to note is that we would have to have to fire off our tasks as separate tasks using `Task.Run`.

If we have a infinite task and we decide to run it directly in the `StartAsync` method, the task would never complete. A ASP.NET Core web application relies on the `StartAsync` method to complete before it can start the web application.

Here is how we can set off a separate task in the `StartAsync` without delaying the web application from launching.

```
using Microsoft.Extensions.Hosting;

namespace RoundTheCode.HostedServiceExample
{
    public class MyHostedService : IHostedService
    {
        public Task StartAsync(CancellationToken cancellationToken)
        {
            Task.Run(async () =>
            {
                while (!cancellationToken.IsCancellationRequested)
                {
                    await Task.Delay(new TimeSpan(0, 0, 5)); // 5 second delay
                }
            });

            return Task.CompletedTask;
        }

        public Task StopAsync(CancellationToken cancellationToken)
        {
            return Task.CompletedTask;
        }
    }
}
```

Inheriting the BackgroundService class

The alternative is to inherit the `BackgroundService` class. The `BackgroundService` class is an abstract class that also inherits the `IHostedService` interface.

And unlike running a task in `StartAsync`, we don't have to run a separate task and risk the web application not starting. We can make the `ExecuteAsync` task asynchronous by using the `async` modifier, and run our task within it.

```
using Microsoft.Extensions.Hosting;

namespace RoundTheCode.HostedServiceExample
{
    public class MyBackgroundService : BackgroundService
    {
        protected override async Task ExecuteAsync(CancellationToken stoppingToken)
        {
            await Task.CompletedTask;
        }
    }
}
```

Add the hosted service to IServiceCollection

When configuring services, there is an extension method called `AddHostedService` in the `IServiceCollection` interface.

As the `AddHostedService` extension method is expecting a class that inherits `IHostedService`, we don't specify the interface when adding the hosted service.

If the web application has a `Startup.cs` file, which will be the case if using .NET 5 or lower, it can be added using the `IServiceCollection` type parameter that is passed into the `ConfigureServices` method.

```
namespace RoundTheCode.HostedServiceExample
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        // This method gets called by the runtime. Use this method to add services t
        public void ConfigureServices(IServiceCollection services)
        {
            // Add dependency injection.
            services.AddHostedService<MyBackgroundService>();

            ...
        }

        // This method gets called by the runtime. Use this method to configure the
        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            ...
        }
    }
}
```

If the web application is using .NET 6 and was created using the .NET 6 template, then we can add in the `Program.cs` file like this:

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddHostedService<MyBackgroundService();

var app = builder.Build();

...

app.Run();
```

Which ever template is being used, the hosted service will work in the same way.

How dependency injection works

Hosted services are added to the DI container using the singleton service lifetime. As a result, it can only resolve singleton and transient service lifetime classes that are injected into it.

If we were to inject a scoped service lifetime class into the hosted service, it would not know which scope it belongs to and would throw a runtime error when the hosted background service started.

But we can create our own scope when running a background task and resolve a scoped service from that scope.

To do that, we need to inject the `IServiceProvider` instance into the hosted service. Within that interface, we can use the `CreateScope` method to create a new scope instance. From there, we can use our new scope instance to resolve a scoped service into our background task.

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

namespace RoundTheCode.HostedServiceExample
{
    public class MyBackgroundService : BackgroundService
    {
        private readonly IServiceProvider _serviceProvider;
        public MyBackgroundService(IServiceProvider serviceProvider)
        {
            _serviceProvider = serviceProvider;
        }

        protected override async Task ExecuteAsync(CancellationToken cancellationTok
        {
            while (!cancellationToken.IsCancellationRequested)
            {
                using (var scope = _serviceProvider.CreateScope())
                {
                    var myScopedService = scope.ServiceProvider.GetRequiredService<I
                }

                await Task.Delay(new TimeSpan(0, 1, 0));
            }
        }
    }
}
```

See how a hosted service works

Check out our video where we demonstrate how an hosted service works in ASP.NET Core web application. Learn how to create and configure a hosted service and how it behaves with dependency injection.

Worker Service template

Should a background service be in an ASP.NET Core web application? It's good if we need to feed data to all users of the website. However, if a background service has performance issues, it can affect the web application's stability.

A better alternative is to use a Worker Service Template which is solely a background service application. And the benefit is that it can be used as a Windows service.

Net Core Host Aspnet

11

Written by Pal Chouhan

23 Followers · 2 Following

Microsoft Developer

Follow

No responses yet

What are your thoughts? Respond

More from Pal Chouhan

Pal Chouhan

Upgrading a WCF service to .NET 6 with CoreWCF

Almost three years ago, I posted a blog walking through the process of migrating a...

May 9, 2022 6

Pal Chouhan

Using PathBase with .NET 6's WebApplicationBuilder

In this post I describe the difficulties of adding calls to UsePathBase with .NET 6...

Jun 12, 2022 3 1

Pal Chouhan

D365FO Infolog call stack

Infolog is a primary message display system in Dynamics 365 FO. And 'infolog.add()' is...

May 7, 2022

Pal Chouhan

Working with LedgerDimension values from posted transactions i...

Recently we faced and fixed an interesting bug. We had a custom procedure that...

May 7, 2022

See all from Pal Chouhan

Recommended from Medium

Ravi Patel

Dependency Injection and Services in ASP.NET Core: A Comprehensi...

Dependency Injection (DI) is a cornerstone of modern software development, enabling...

Sep 30, 2024 43

In Level Up Coding by Jacob Bennett

The 5 paid subscriptions I actually use in 2025 as a Staff Software...

Tools I use that are cheaper than Netflix

Jan 8 5K 110

Lists

Staff picks
800 stories · 1569 saves

Stories to Help You Level-Up at Work
19 stories · 920 saves

Self-Improvement 101
20 stories · 3227 saves

Productivity 101
20 stories · 2725 saves

Harendra

How I Am Using a Lifetime 100% Free Server

Get a server with 24 GB RAM + 4 CPU + 200 GB Storage + Always Free

Oct 26, 2024 8.5K 132

Jessica Stillman

Jeff Bezos Says the 1-Hour Rule Makes Him Smarter. New...

Jeff Bezos's morning routine has long included the one-hour rule. New...

Oct 30, 2024 20K 502

Kenji Elzerman

C# Is Dead

Is C# on it's decline? Are other languages taken over? What some people are telling m...

Sep 20, 2024 1K 62

In CodeX by Lakhveer Singh Rajput

SQL Queries That Will Surprise You! 🚀💡

SQL is the backbone of data manipulation, but some queries are not your everyday...

Dec 3, 2024 773 15

See more recommendations