NET Core Background Services

> NOTE: Refer to this page for how to register and consume a background service: Consuming NET Core Background Service with DI

General

Lots of the ceremony of a background service has been wrapped up and tested, in this class: BackgroundService_Base

> NOTE: As of 20250504, the latest BackgroundService_Base type is in the MOVEELSEWHERE folder of OGA.Tasking.CommonShared_SP. Need to create a library just for it, to centralize the copies of its type.
> The previous golden version was found in the MOVEELSEWHERE project of the Chat Message Service solution.

It provides the following features:

- Can be registered as a HostedService. See: Consuming NET Core Background Service with DI
- Capable of performing periodic action via looping override
- Periodic service loop can be set to a fixed delay between loop iterations
- Overrides for startup and shutdown, allowing setup and cleanup

Minimal Derived Service

A minimal derived backgroundservice class needs only a constructor, like below:

```
public class Derived_BackgroundService : BackgroundService_Base
{
```

```
    public Derived_BackgroundService(IServiceProvider serviceProvider) : base(serviceProvider)
    {
    }
}
```

The above example shows a simple constructor that does the following:

- Accepts a service provider instance, required for a singleton service to access DI resources
- Sets the local classname variable, for log messages

Setup and Teardown Overrides

Obviously, a background service that does real work must do include setup and teardown.

Setup Override

Override the `DoStartupActivities, below, with any setup logic your service needs.`

You don't need to call the base method. Any preceding setup is performed before this override.

> NOTE: This override must return '1', or the service fails to startup.

```
protected override int DoStartupActivities(CancellationToken token)
{
  // Do some startup magic, here...
  return 1;
}
```

Teardown Override

Override the `DoShutdownActivities, below, with any teardown logic your service needs.`

You don't need to call the base method. Any mandatory teardown setup, is performed before and after this override.

This teardown method is called by the dispose method of the base class, to ensure everything gets shutdown.

```
protected override void DoShutdownActivities()
{
  // Do any cleanup, here...
  return;
}
```

Derived Dispose

If your derived service type has no need to dispose or release resources it owns, then you can rely on the Dispose methods of the background service base, and don’t have to include any Dispose overrides.

But. If your derived service class needs to release resources, you will have to follow a cascade Dispose, and override the protected Dispose method of the base service class.

> BE AWARE: The BackgroundService class (from Microsoft) that our Background service base inherits from, has only a public Dispose method, and no protected virtual Dispose. This is NOT the sanctioned Dispose pattern that developers are accustomed to.
> So, follow the Dispose guidelines in this article carefully, to ensure that the protected Dispose(true) method of the background service base actually gets called.
> What this means, is our background service base class is attempting to correct for the pattern discrepancy, and cannot 100% follow follow the normal pattern of cascaded Dispose, because there is a lingering virtual public Dispose() of the base class, that can be mistakenly overridden.

However. Our background service base class does correct for this, by creating the normal Dispose implementation pattern for derived classes, so that any derived service types can also follow the cascaded Dispose pattern.

See this article for how to implement cascaded dispose in derived types: C# Disposed in Derived Types

Implementation

Getting back to what we were doing...
If your derived service class needs to release resources, in a Dispose, here are the required elements:

1. You will need to include your own private bool flag that your derived type is disposed. This a normal requirement for any class that implements IDisposable, or derives from a class that does.

   See this: C# Disposed in Derived Types
2. You will need to override the protected virtual void Dispose(bool disposing) method of the background service base class. This is a requirement, so that your private is disposed flag can be checked and set when Dispose runs.
3. Your protected dispose override method body will need to call the base.dispose(disposing) method at the end of its logic (right before setting your disposed flag).
4. Your protected dispose override method body will need to set the disposed flag of your derived type.
   This needs to be the last thing your protected Dispose method does.

> NOTE: Your derived type will have its own is disposed flag, which is separate from the is disposed flag of the base class. This is on purpose, to ensure the base can successfully handle its own disposing needs, without being affected by any logic flaw of your code.

> NOTE: Your derived type will NOT need to override the public Dispose() method, as this is ONLY by our background base service, to correct for the incorrect pattern usage by the Microsoft backgroundservice class.

Here are the pieces that your derived service type will need:

```
public class BackgroundService_Base : BackgroundService, IDisposable
{
    private bool _disposecalled = false;

    override protected void Dispose(bool disposing)
    {
        if (!this._disposecalled)
        {
            if (disposing)
            {
                // TODO: dispose managed state (managed objects)
            }

            // Free your additional resources...

            // Call the protected Dispose method of the base...
            base.Dispose(disposing);
```

```
            // Set your own is disposed flag...
            this._disposecalled = true;
        }
    }
}
```

> NOTE: Our background service base has no unmanaged resources to release. So, it doesn’t have a destructor (finalizer) override.
> But, if our derived type includes unmanaged resources that need to be released, it will require a finalizer override that can call the protected Dispose() method.

Here’s what that finalizer would look like:

```
    ~BackgroundService_Base()
    {
        // Do not change this code. Put cleanup code in 'Dispose(bool disposing)' method
        Dispose(disposing: false);
    }
```

Revision #9
Created 30 April 2025 01:40:48 by glwhite
Updated 15 September 2025 22:51:25 by glwhite