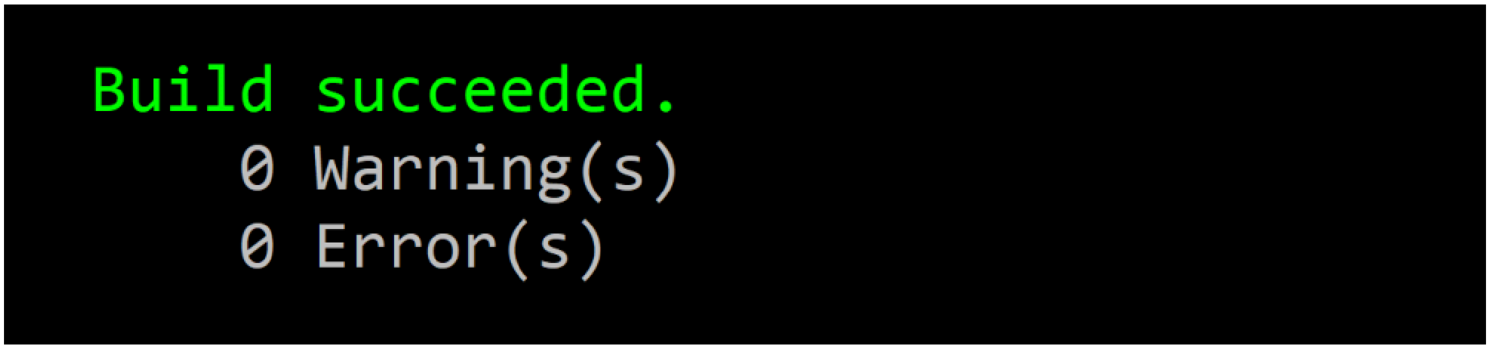
David's Blog



```
Build succeeded.
    0 Warning(s)
    0 Error(s)
```

← Previous Post Next Post →

Building a Console App with .NET Generic Host

Aug 24, 2020 • Posted in .NET

The .NET Generic Host is a feature which sets up some convenient patterns for an application including those for dependency injection (DI), logging, and configuration. It was originally named Web Host and intended for Web scenarios like ASP.NET Core applications but has since been generalized (hence the rename to *Generic* Host) to support other scenarios, such as Windows services, Linux daemon services, or even a console app.

A working example can be found at dfederm/GenericHostConsoleApp.

Basics

You will first need to create a new console application and add a `PackageReference` to `Microsoft.Extensions.Hosting`.

```
dotnet new console
dotnet add package Microsoft.Extensions.Hosting
```

Now for the `Main` method. Typically the `Main` method for console apps just immediately jump into the application’s core logic, but when using the .NET Generic Host, instead the host is set up. This should look familiar if you’ve developed web applications using the Web Host or Generic Host before.

```
internal sealed class Program
{
    private static async Task Main(string[] args)
    {
        await Host.CreateDefaultBuilder(args)
            .RunConsoleAsync();
    }
}
```

Running that alone will start up the host, but without any logic it will do nothing and never exit. The generic host simply sets up some reasonable defaults around configuration and logging, as well as provides a few services in the DI container which handle the the application lifetime. Calling `RunConsoleAsync` will run start the host and wait for a `Ctrl+C` or `SIGTERM` to exit, which means without explicitly telling the app to exit, it will not exit.

To actually implement your console app's logic, and get the application to exit after it's done, you'll want to implement and register an `IHostedService` , as well as interact with the `IHostApplicationLifetime` from the DI container.

```
internal sealed class Program
{
    private static async Task Main(string[] args)
    {
        await Host.CreateDefaultBuilder(args)
            .ConfigureServices((hostContext, services) =>
            {
                services.AddHostedService<ConsoleHostedService>();
            })
            .RunConsoleAsync();
    }
}

internal sealed class ConsoleHostedService : IHostedService
{
    private readonly ILogger _logger;
    private readonly IHostApplicationLifetime _appLifetime;

    public ConsoleHostedService(
        ILogger<ConsoleHostedService> logger,
        IHostApplicationLifetime appLifetime)
    {
        _logger = logger;
        _appLifetime = appLifetime;
    }

    public Task StartAsync(CancellationToken cancellationToken)
    {
        _logger.LogDebug($"Starting with arguments: {string.Join(" ", Environment.G

        _appLifetime.ApplicationStarted.Register(() =>
        {
            Task.Run(async () =>
            {
                try
                {
                    _logger.LogInformation("Hello World!");

                    // Simulate real work is being done
                    await Task.Delay(1000);
                }
                catch (Exception ex)
                {
                    _logger.LogError(ex, "Unhandled exception!");
                }
                finally
                {
                    // Stop the application once the work is done
                    _appLifetime.StopApplication();
                }
            });
        });

        return Task.CompletedTask;
    }

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

Other useful events to subscribe to on the `IHostApplicationLifetime` are `ApplicationStopping` and `ApplicationStopped`.

Note that you can registrer multiple `IHostedService` implementations and each of them will have their `StartAsync` and `StopAsync` methods called. However, personally I find that to be a bit confusing for a console application, so I would just stick to a single `IHostedService` .

Dependency Injection

As you may have noticed, your `IHostedService` implementation has full access to the DI container, so you can register services in `Main` and then use them in your `IHostedService` .

In `Main` :

```
await Host.CreateDefaultBuilder(args)
    .ConfigureServices((hostContext, services) =>
    {
        services
            .AddHostedService<ConsoleHostedService>();
            .AddSingleton<IWeatherService, WeatherService>();
    })
    .RunConsoleAsync();
```

In `ConsoleHostedService`

```
private readonly ILogger _logger;
private readonly IHostApplicationLifetime _appLifetime;
private readonly IWeatherService _weatherService;

public ConsoleHostedService(
    ILogger<ConsoleHostedService> logger,
    IHostApplicationLifetime appLifetime,
    IWeatherService weatherService)
{
    _logger = logger;
    _appLifetime = appLifetime;
    _weatherService = weatherService;
}

public Task StartAsync(CancellationToken cancellationToken)
{
    _appLifetime.ApplicationStarted.Register(() =>
    {
        Task.Run(async () =>
        {
            try
            {
                IReadOnlyList<int> temperatures = await _weatherService.GetFiveDayT
                for (int i = 0; i < temperatures.Count; i++)
                {
                    _logger.LogInformation($"{DateTime.Today.AddDays(i).DayOfWeek}:
                }
            }
            catch (Exception ex)
            {
                _logger.LogError(ex, "Unhandled exception!");
            }
            finally
            {
                // Stop the application once the work is done
                _appLifetime.StopApplication();
            }
        });
    });

    return Task.CompletedTask;
}
```

Exit Code

To return a non-zero exit code, your `IHostedService` implementation can set `Environment.ExitCode` when the host is stopping.

```
internal sealed class ConsoleHostedService : IHostedService
{
    private int? _exitCode;

    // ...

    public Task StartAsync(CancellationToken cancellationToken)
    {
        // ...

        _appLifetime.ApplicationStarted.Register(() =>
        {
            Task.Run(async () =>
            {
                try
                {
                    // ...
```

```
                        _exitCode = 0;
                    }
                    catch (Exception ex)
                    {
                        _logger.LogError(ex, "Unhandled exception!");
                        _exitCode = 1;
                    }
                    finally
                    {
                        // Stop the application once the work is done
                        _appLifetime.StopApplication();
                    }
                });
            });

            return Task.CompletedTask;
        }

        public Task StopAsync(CancellationToken cancellationToken)
        {
            _logger.LogDebug($"Exiting with return code: {_exitCode}");

            // Exit code may be null if the user cancelled via Ctrl+C/SIGTERM
            Environment.ExitCode = _exitCode.GetValueOrDefault(-1);
            return Task.CompletedTask;
        }
    }
```

Logging and Configuration

Logging and configuration work mostly just as they do in Web applications. The one caveat is that because `appsettings.json` is loaded from the content root, which for the Generic Host is the current working directory by default, the content root needs to be set to the same directly the executable is in using `UseContentRoot` .

In `Main` :

```
await Host.CreateDefaultBuilder(args)
    .UseContentRoot(Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location)
    .ConfigureLogging(logging =>
    {
        // Add any 3rd party loggers like NLog or Serilog
    })
    .ConfigureServices((hostContext, services) =>
    {
        services
            .AddHostedService<ConsoleHostedService>()
            .AddSingleton<IWeatherService, WeatherService>();

        services.AddOptions<WeatherSettings>().Bind(hostContext.Configuration.GetSe
    })
    .RunConsoleAsync();
```

In `WeatherService.cs`:

```
internal sealed class WeatherService : IWeatherService
{
    private readonly IOptions<WeatherSettings> _weatherSettings;

    public WeatherService(IOptions<WeatherSettings> weatherSettings)
    {
        _weatherSettings = weatherSettings;
    }

    public Task<IReadOnlyList<int>> GetFiveDayTemperaturesAsync()
    {
        int[] temperatures = new[] { 76, 76, 77, 79, 78 };
        if (_weatherSettings.Value.Unit.Equals("C", StringComparison.OrdinalIgnoreC
        {
            for (int i = 0; i < temperatures.Length; i++)
            {
                temperatures[i] = (int)Math.Round((temperatures[i] - 32) / 1.8);
            }
        }

        return Task.FromResult<IReadOnlyList<int>>(temperatures);
    }
}
```

In `WeatherSettings.cs`:

```
internal sealed class WeatherSettings
{
    public string Unit { get; set; }
}
```

In the project file:

```
  <ItemGroup>
    <Content Include="appsettings.json">
      <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
    </Content>
  </ItemGroup>
```

In `appsettings.json`:

```
{
    "Logging": {
        "LogLevel": {
            "Default": "Information",
            // Avoid logging lifetime events
            "Microsoft.Hosting.Lifetime": "Warning"
        }
    },
    "Weather": {
        "Unit": "C"
    }
}
```

43 Comments 1 Login

G

Join the discussion…

LOG IN WITH OR SIGN UP WITH DISQUS ?

Name

7 Share Best Newest Oldest

R

Rich Ellis
3 years ago

This was a lifesaver. Could not find much of it in the Microsoft docs.

4 0 Reply

Andrey Rodin
a year ago

What if I need to run everything from Generic Host in the background (services which are used elsewhere in my solution), but I need a separate console app which accepts console input. In other words I would like a normal console application with user input/app output which have access to all services in Generic Host?

0 0 Reply

M

Michael von Fleisch
2 years ago

Hi, I found your blog trough a intensive google search.

Following Problem:
I am an experienced .net-Framework dev and have my issues understanding the new service Architecture.

I usually builded my services in company that I can start them as .Console-App for debugging purpose but being able to be installed as service

Just like this mini garbage project of mine https://bitbucket.org/da-le...

If I understand your article rights its to build an .exe which hosts a service, (maybe I understand it wrong) which hosts a service.

Do you have an Idea how I can write like in good old times an App which can work as Console-App or/and as service with same codebase and same deployment?

Best Regards,
Michael Fleischer

0 0 Reply

David Federman **Mod** Michael von Fleisch

2 years ago edited

You may be interested in .NET Worker Services: https://learn.microsoft.com...

0 0 Reply

M

Michael von Fleisch David Federman

2 years ago

thanks for the answer, so just replace the .net framework start stop thing with that and I will still be able to install software as console app/or and as service?

0 0 Reply

S

Some Coder

2 years ago

There's some stuff here which is really hard to follow. I'll start with a simple question.
Why is StartAsync (ostensibly an async method) run synchronously?

0 0 Reply

David Federman **Mod** Some Coder

2 years ago

`StartAsync` needs to return for the application to actually finish "starting". It's async to allow some async initialization, but if you put the entire application's logic in `StartAsync`, then it'll always be "starting" and never "started".

1 0 Reply

S

Some Coder David Federman

2 years ago

OK. I think I understand that. It'll return quickly if it is synchronous and you can use Task.Run to kick off the application logic (in the registered callback of the _applicationLifetime). Thank you.

0 0 Reply

E

Eduardo Garcia-Prieto

2 years ago

Hi David,

Great post!

I took the liberty of forking your GitHub repo and added some extra features that I plan on using in future console apps.

I would love any feedback from you (or anyone on here). :)

egarcia74/GenericHostConsoleApp

* Separating the main application logic from the boilerplate startup code. There is now a separate MainService class, which can be easily modified to implement custom logic without having to worry about all the application plumbing required to wire up the application hosted service.

* Moved classes into subdirectories corresponding to the class' area of concern. E.g. Configuration, Services, Interfaces.

* Using an ExitCode enum to define return values.

* Compile-time logging source generation .

* Validation of configuration options .

* Serilog as the logging provider.

0 0 Reply

D **Dave Jones** Eduardo Garcia-Prieto

2 years ago edited

Some great code in there. But there is a problem. If you add a second service, you will sometimes get a TaskCanceledException. I cloned your repo, duplicated your **ApplicationHostedService** (named it something else) and on my first run I got that exception. Been getting that exception in my experiments with this as well. I think there's something wrong with the token handling in the code.

0 0 Reply

David Federman Mod Dave Jones

2 years ago

Can you share a repro?

I suspect it's related to the StopApplication call in the finally block. When that service is done it would tear down the entire application. Multiple services would need to coordinate on that.

3 0 Reply

E **Eduardo Garcia-Prieto** David Federman

2 years ago

I think that's correct. The ApplicationHostedService will tear down the process after the MainService.Main method completes.

1 0 Reply

D **Dave Jones** ↪ David Federman

2 years ago

I have forked your repo and added a 2nd service. That fork is here. I've put a small delay in the "Other" service to demonstrate. The only solution I can think of, in the absence of some kind of tricky orchestration, is to ensure that the slowest service calls StopApplication. But it will have to be ensured to be the slowest and is not really an optimum solution. The thing I don't get is that when you call IsCancellationRequested in the other service, it returns false! I guess that can't be updated across hosted services, despite the linked token source. Cheers.

0 0 Reply

D **Dave Jones** ↪ David Federman

2 years ago

I believe that is exactly right. I'll put together a sample when I get back to a PC. Not sure how to stop the host other than StopApplication and the HostApplicationLifetime abstraction does not really help, despite being a singleton.

0 0 Reply

J **Jean-Francois Cloutier**

3 years ago

Thanks David. This was so helpful !!!

I cannot comprehend that Microsoft does not have an example as clear as this. If they do, it is well buried on their site and it should be promoted as one of their top documentation.

In any case (and with all due respect), I think you have one small bug in your app. You should store the Task object returned by Task.Run when you register your application and await that task in StopAsync.

At least, in my case, I was getting a problem on shutdown about the CancellationToken. I figured it had to do with my background task and as soon as I started awaiting for that Task obejct, my application was now shutting down perfectly and cleanly.

Let me know if this makes sense. It is also possible that there is a subtle detail that I missed from your example.

0 0 Reply

Bob ↪ Jean-Francois Cloutier

6 months ago

You could add to your try catch so you catch and swallow the cancelled exception before you unhandled exception.

so

catch when (applicationLifeTime.ApplicationStopping.IsCancellationRequested) {
// ignore the error
} catch (Exception ex) {
...

0 0 Reply

David Federman Mod Jean-Francois Cloutier
2 years ago edited

I know this is a super late reply, but I think you're right. I'll correct the code in the GitHub repo.

It's actually a little more involved and requires you to maintain your own cancellation token source to cancel your application logic properly.

0 0 Reply

D **Dave Jones** David Federman
2 years ago

I have been experimenting with this and the Github code does not seem to work where a 2nd hosted service is included. I'm getting a TaskCanceledException. It is always thrown by the slower service (experimented with Task.Delay).

However, if you do not call cancellationTokenSource.Cancel(), it all works as expected. Cancelling in the ApplicationStopping event causes the problem.

0 0 Reply

S **Some Coder** Dave Jones
2 years ago

Pretty sure this happens because Task.Delay throws an exception when a Task is cancelled. The code is prevaricated on the notion that the async operation you call does NOT throw an exception. If it does, your would need to change the code to handle that (I think).

0 0 Reply

D **Dave Jones** Some Coder
2 years ago

I tested your theory and it still does not work. So, I replaced my call to Task.Delay with a service that checks for cancelation and returns without an exception. It does not work. There does not seem to be a way of signalling cancelation from one hosted service to another. The linked CancellationTokenSource may do that within one hosted service, but it does not achieve it across two hosted services. Something is missing here.

1 0 Reply

N

NB

3 years ago

Thanks David for this very useful post. Why did you need to register the task in startasync to IApplicationLiffetime CancellationToken called applicationstarted? I didn't do this I just called the task directly in background so StartAsync returns immediately and all works well. Can you explain the difference thanks

0 0 Reply

David Federman Mod ➦ NB

3 years ago

The differences are subtle but approximately the same. In my example, the actual "service" startup (your logic) happens once the application is started, ie after any internal stuff the generic host does. If you kick off that logic in a task immediately, then the generic host's application startup happens concurrently with your logic, so the application isn't "started" yet (from the host's perspective). I can't think of any specific issues your approach would cause, but my example follows the application lifetime events of the host.

0 0 Reply

N

NB ➦ David Federman

3 years ago edited

Thanks David for your explanation which could be included to explain the code content :)

Do you have links showing samples where application lifetime events are handled from IHostedService

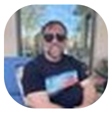
I found this one for example https://voidnish.wordpress.com/2018/07/21/handling-application-lifetime-events-for-a-hosted-service/

Here what I did, in my case I needed to close down the application once the process is done so _hosttApplicationLifeTime.StopApplication(); is called

```
public Task StartAsync(CancellationToken cancellationToken)
{
try
{
```

see more

0 0 Reply



Michael Wasson

3 years ago

Program.cs Line 17 says to add 3rd Party Loggers but those require info like a Token that you would store in app.config. Can you get Configuration at that point? Should ConfigureLogging() be moved below ConfigureServices()?

0 0 Reply

David Federman Mod ↪ Michael Wasson

3 years ago edited

For NLog, I usually just do something like (apologies for bad formatting):

```
hostBuilder.ConfigureLogging((hostingContext, loggingBuilder) =>
{
// configure Logging with NLog
loggingBuilder.ClearProviders();
loggingBuilder.AddConfiguration(hostingContext.Configuration.GetSection("Logging"
));
loggingBuilder.AddNLog(GetLoggingConfiguration());
});
```

0 0 Reply

Glen

3 years ago

(Sorry, formatting in Disqus is not very nice.)

0 0 Reply

Glen

3 years ago

This seems to be working for me.

```
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using MyApp.MainCode;

namespace MyApp
{
class Program
{
static void Main(string[] args)
{
var host = Host.CreateDefaultBuilder()
 .ConfigureAppConfiguration(app =>
{
app.AddJsonFile("appsettings.json");
```

see more

0 0 Reply

David Federman Mod ➦ Glen
3 years ago edited

Based on this, seems like you're missing the RunConsoleAsync call which is what sets up the Cyril+C handler. I think that's the issue you were facing?

I also don't think any of the lifetime events are set up in your case. You're more or less just using the Service collection directly and not actually using the host

0 0 Reply

Glen ➦ David Federman
3 years ago

Yes, my goal is to simply have a working console app... The Main routine is using the host to do configuration and DI.

0 0 Reply

Glen
3 years ago

Here's another approach: https://wildermuth.com/2020...

0 0 Reply

Glen
3 years ago

With this approach, it looks like using Ctrl+C to stop the application does not work. Here's what the log shows:

> info: Microsoft.Hosting.Lifetime[0]
> Application is shutting down...

but the application continues.

0 0 Reply

Dennis Rongo ➦ Glen
2 years ago

My guess is, you're probably in debug mode?

0 0 Reply

David Federman Mod ➦ Glen

3 years ago

Do you have a repro to share? It seems to work in the example code I used.

0 0 Reply

W

Wouter Van Ranst

3 years ago edited

This is an EXCELLENT post!

How would you tackle unit testing/integration testing, specifically how to test the components that are generated by the DI? I got the IServiceProvider reference after the run, but (obviously) all the objects are disposed when the application is completed.

Additionally/FYI in the context for unit testing, the Environment.GetCommandLineArgs() call doesn't work as expected (it returns the arguments that were passed into xUnit) - after a lot of browsing the best practice seems to be to set a custom/short lived environment variable, but that somewhat messes up the cleanliness of the code.

0 0 Reply

David Federman Mod ➦ Wouter Van Ranst

3 years ago edited

For testing components which are creating by the DI container, I would just test them "normally". ie just `new` the object up and provide mock implementations for all dependencies (after all, these are *unit* tests).

Regarding the command line args, you could create a service like `IEnvironment` where the implementation uses `Environment.GetCommandLineArgs()`, Env vars, and whatever other statics which are part of the environment and hard to mock. That way in unit tests you can provide a MockEnvironment. Similarly you see people use this pattern and create an IClock which abstracts `DateTime.Now`.

0 0 Reply

Chris Walsh

3 years ago edited

Great article. And nice to stumble across one that is "up to date" rather than lots of examples of "older" versions of .NET Core async/DI/Configuration from the last 5 years.
I think I understand what you are saying in your reply to **@Oleksii Nikiforov** about using `_appLifetime.ApplicationStarted.Register` but why wrap the implementation code in an `Task.Run(async() => {})`? Surely the actual program (`Task.Delay(1000)` or similar) can be awaited on directly within the Register() callback?
Thanks.

0 0 Reply

David Federman Mod ➦ Chris Walsh

3 years ago

I believe you want to wrap it in a `Task.Run` since it's in an event callback. Generally you don't want event callbacks to block for a long time (in this case the entire lifetime of the application basically), as events are dispatched sequentially. Imagine if there were multiple registrations to the event.

To be fair, I didn't test not wrapping it, so they very well may call these in a non-blocking way.

1 0 Reply

vkradio David Federman

2 years ago

But if it is an event, isn't async-await created exactly to not block it, without introducing any additional threads? 🤔

1 0 Reply

David Federman **Mod** vkradio

2 years ago

It's not the same thing as this `Task.Run` is *not* awaited, so it doesn't block at all. Also, keep in mind that we're not in an async context anyway at this point, we're in a synchronous lambda passed to `_appLifetime.ApplicationStarted.Register`.

1 0 Reply

Chris Walsh David Federman

3 years ago

Thanks. Hadn't considered that.

0 0 Reply

O **Oleksii Nikiforov**

4 years ago

Thanks for post.
Why do you need to invoke `_appLifetime.ApplicationStarted.Register` rather than doing your code directly in *StartAsync*. What is the benefit of doing that?

0 0 Reply

David Federman **Mod** Oleksii Nikiforov

4 years ago

According to the docs, `IHostedService.StartAsync` is "triggered when the application host is ready to start the service", while `IApplicationLifetime.ApplicationStarted` is "triggered when the application host has fully started and is about to wait for a graceful shutdown." The differences are subtle, and I suspect in practice you won't have much of an issue using either one. However, `IHostedService.StartAsync` is called when the host isn't completely started up yet, so I suspect there might be race conditions if your code finishes (and calls `StopApplication`) faster than the host finishing starting up.

0 0 Reply

Oleksii Nikiforov David Federman

David's Blog

Some guy's programming blog.

Privacy Policy • License