.NET Generic Host

Article • 12/14/2023

In this article, you learn about the various patterns for configuring and building a .NET Generic Host available in the Microsoft.Extensions.Hosting NuGet package. The .NET Generic Host is responsible for app startup and lifetime management. The Worker Service templates create a .NET Generic Host, HostApplicationBuilder. The Generic Host can be used with other types of .NET applications, such as Console apps.

A *host* is an object that encapsulates an app's resources and lifetime functionality, such as:

- Dependency injection (DI)
- Logging
- Configuration
- App shutdown
- `IHostedService` implementations

When a host starts, it calls IHostedService.StartAsync on each implementation of IHostedService registered in the service container's collection of hosted services. In a worker service app, all `IHostedService` implementations that contain BackgroundService instances have their BackgroundService.ExecuteAsync methods called.

The main reason for including all of the app's interdependent resources in one object is lifetime management: control over app startup and graceful shutdown.

Set up a host

The host is typically configured, built, and run by code in the `Program` class. The `Main` method:

IHostApplicationBuilder

- Calls a CreateApplicationBuilder method to create and configure a builder object.
- Calls Build() to create an IHost instance.
- Calls Run or RunAsync method on the host object.

The .NET Worker Service templates generate the following code to create a Generic Host:

C#

```
using Example.WorkerService;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);
builder.Services.AddHostedService<Worker>();

IHost host = builder.Build();
host.Run();
```

For more information on Worker Services, see Worker Services in .NET.

Host builder settings

IHostApplicationBuilder

The CreateApplicationBuilder method:

- Sets the content root to the path returned by GetCurrentDirectory().
- Loads host configuration from:
  - Environment variables prefixed with `DOTNET_`.
  - Command-line arguments.
- Loads app configuration from:
  - *appsettings.json*.
  - *appsettings.{Environment}.json*.
  - Secret Manager when the app runs in the `Development` environment.
  - Environment variables.
  - Command-line arguments.
- Adds the following logging providers:
  - Console
  - Debug
  - EventSource
  - EventLog (only when running on Windows)
- Enables scope validation and dependency validation when the environment is `Development`.

The HostApplicationBuilder.Services is an Microsoft.Extensions.DependencyInjection.IServiceCollection instance. These services are used to build an IServiceProvider that's used with dependency injection to resolve the registered services.

Framework-provided services

When you call either IHostBuilder.Build() or HostApplicationBuilder.Build(), the following services are registered automatically:

- IHostApplicationLifetime
- IHostLifetime
- IHostEnvironment

IHostApplicationLifetime

Inject the IHostApplicationLifetime service into any class to handle post-startup and graceful shutdown tasks. Three properties on the interface are cancellation tokens used to register app start and app stop event handler methods. The interface also includes a StopApplication() method.

The following example is an `IHostedService` implementation that registers `IHostApplicationLifetime` events:

C#

```
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

namespace AppLifetime.Example;

public sealed class ExampleHostedService : IHostedService
{
    private readonly ILogger _logger;

    public ExampleHostedService(
        ILogger<ExampleHostedService> logger,
        IHostApplicationLifetime appLifetime)
    {
        _logger = logger;

        appLifetime.ApplicationStarted.Register(OnStarted);
        appLifetime.ApplicationStopping.Register(OnStopping);
        appLifetime.ApplicationStopped.Register(OnStopped);
    }

    public Task StartAsync(CancellationToken cancellationToken)
    {
        _logger.LogInformation("1. StartAsync has been called.");

        return Task.CompletedTask;
    }
```

```
    public Task StopAsync(CancellationToken cancellationToken)
    {
        _logger.LogInformation("4. StopAsync has been called.");

        return Task.CompletedTask;
    }

    private void OnStarted()
    {
        _logger.LogInformation("2. OnStarted has been called.");
    }

    private void OnStopping()
    {
        _logger.LogInformation("3. OnStopping has been called.");
    }

    private void OnStopped()
    {
        _logger.LogInformation("5. OnStopped has been called.");
    }
}
```

The Worker Service template could be modified to add the `ExampleHostedService` implementation:

C#

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

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Services.AddHostedService<ExampleHostedService>();
using IHost host = builder.Build();

await host.RunAsync();
```

The application would write the following sample output:

C#

```
// Sample output:
//     info: ExampleHostedService[0]
//           1. StartAsync has been called.
//     info: ExampleHostedService[0]
//           2. OnStarted has been called.
//     info: Microsoft.Hosting.Lifetime[0]
//           Application started.Press Ctrl+C to shut down.
//     info: Microsoft.Hosting.Lifetime[0]
```

```
//           Hosting environment: Production
//     info: Microsoft.Hosting.Lifetime[0]
//           Content root path: ..\app-lifetime\bin\Debug\net7.0
//     info: ExampleHostedService[0]
//           3. OnStopping has been called.
//     info: Microsoft.Hosting.Lifetime[0]
//           Application is shutting down...
//     info: ExampleHostedService[0]
//           4. StopAsync has been called.
//     info: ExampleHostedService[0]
//           5. OnStopped has been called.
```

IHostLifetime

The IHostLifetime implementation controls when the host starts and when it stops. The last implementation registered is used. `Microsoft.Extensions.Hosting.Internal.ConsoleLifetime` is the default `IHostLifetime` implementation. For more information on the lifetime mechanics of shutdown, see Host shutdown.

IHostEnvironment

Inject the IHostEnvironment service into a class to get information about the following settings:

- IHostEnvironment.ApplicationName
- IHostEnvironment.ContentRootFileProvider
- IHostEnvironment.ContentRootPath
- IHostEnvironment.EnvironmentName

Additionally, the `IHostEnvironment` service exposes the ability to evaluate the environment with the help of these extension methods:

- HostingEnvironmentExtensions.IsDevelopment
- HostingEnvironmentExtensions.IsEnvironment
- HostingEnvironmentExtensions.IsProduction
- HostingEnvironmentExtensions.IsStaging

Host configuration

Host configuration is used to configure properties of the IHostEnvironment implementation.

IHostApplicationBuilder

The host configuration is available in IHostApplicationBuilder.Configuration property and the environment implementation is available in IHostApplicationBuilder.Environment property. To configure the host, access the `Configuration` property and call any of the available extension methods.

To add host configuration, consider the following example:

C#

```
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Hosting;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Environment.ContentRootPath = Directory.GetCurrentDirectory();
builder.Configuration.AddJsonFile("hostsettings.json", optional: true);
builder.Configuration.AddEnvironmentVariables(prefix: "PREFIX_");
builder.Configuration.AddCommandLine(args);

using IHost host = builder.Build();

// Application code should start here.

await host.RunAsync();
```

The preceding code:

- Sets the content root to the path returned by GetCurrentDirectory().
- Loads host configuration from:
  - *hostsettings.json*.
  - Environment variables prefixed with `PREFIX_`.
  - Command-line arguments.

App configuration

IHostApplicationBuilder

App configuration is created by calling ConfigureAppConfiguration on an IHostApplicationBuilder. The publicIHostApplicationBuilder.Configuration property

allows consumers to read from or make changes to the existing configuration using available extension methods.

For more information, see Configuration in .NET.

Host shutdown

There are several ways in which a hosted process is stopped. Most commonly, a hosted process can be stopped in the following ways:

- If someone doesn't call Run or HostingAbstractionsHostExtensions.WaitForShutdown and the app exits normally with `Main` completing.
- If the app crashes.
- If the app is forcefully shut down using SIGKILL (or CTRL+Z).

The hosting code isn't responsible for handling these scenarios. The owner of the process needs to deal with them the same as any other app. There are several other ways in which a hosted service process can be stopped:

- If `ConsoleLifetime` is used (UseConsoleLifetime), it listens for the following signals and attempts to stop the host gracefully.
  - SIGINT (or CTRL+C).
  - SIGQUIT (or CTRL+BREAK on Windows, CTRL+\ on Unix).
  - SIGTERM (sent by other apps, such as `docker stop`).
- If the app calls Environment.Exit.

The built-in hosting logic handles these scenarios, specifically the `ConsoleLifetime` class. `ConsoleLifetime` tries to handle the "shutdown" signals SIGINT, SIGQUIT, and SIGTERM to allow for a graceful exit to the application.

Before .NET 6, there wasn't a way for .NET code to gracefully handle SIGTERM. To work around this limitation, `ConsoleLifetime` would subscribe to System.AppDomain.ProcessExit. When `ProcessExit` was raised, `ConsoleLifetime` would signal the host to stop and block the `ProcessExit` thread, waiting for the host to stop.

The process exit handler would allow for the clean-up code in the application to run—for example, IHost.StopAsync and code after HostingAbstractionsHostExtensions.Run in the `Main` method.

However, there were other issues with this approach because SIGTERM wasn't the only way `ProcessExit` was raised. SIGTERM is also raised when app code calls

`Environment.Exit`. `Environment.Exit` isn't a graceful way of shutting down a process in the `Microsoft.Extensions.Hosting` app model. It raises the `ProcessExit` event and then exits the process. The end of the `Main` method doesn't get executed. Background and foreground threads are terminated, and `finally` blocks *aren't* executed.

Since `ConsoleLifetime` blocked `ProcessExit` while waiting for the host to shut down, this behavior led to deadlocks from `Environment.Exit` also blocks waiting for the call to `ProcessExit`. Additionally, since the SIGTERM handling was attempting to gracefully shut down the process, `ConsoleLifetime` would set the ExitCode to `0`, which clobbered the user's exit code passed to `Environment.Exit`.

In .NET 6, POSIX signals are supported and handled. The `ConsoleLifetime` handles SIGTERM gracefully, and no longer gets involved when `Environment.Exit` is invoked.

> **Tip**
>
> For .NET 6+, `ConsoleLifetime` no longer has logic to handle scenario `Environment.Exit`. Apps that call `Environment.Exit` and need to perform clean-up logic can subscribe to `ProcessExit` themselves. Hosting will no longer attempt to gracefully stop the host in these scenarios.

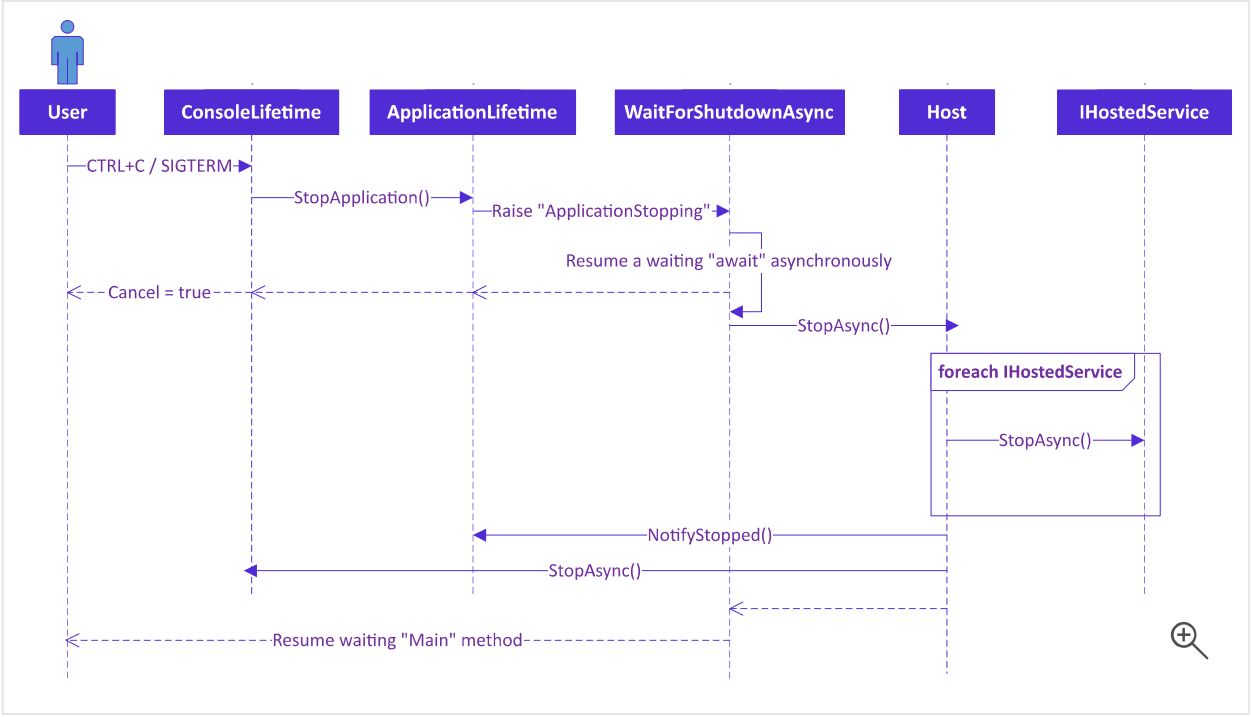
If your application uses hosting, and you want to gracefully stop the host, you can call IHostApplicationLifetime.StopApplication instead of `Environment.Exit`.

Hosting shutdown process

The following sequence diagram shows how the signals are handled internally in the hosting code. Most users don't need to understand this process. But for developers that need a deep understanding, a good visual may help you get started.

After the host has been started, when a user calls `Run` or `WaitForShutdown`, a handler gets registered for IApplicationLifetime.ApplicationStopping. Execution is paused in `WaitForShutdown`, waiting for the `ApplicationStopping` event to be raised. The `Main` method doesn't return right away, and the app stays running until `Run` or `WaitForShutdown` returns.

When a signal is sent to the process, it initiates the following sequence:



1. The control flows from `ConsoleLifetime` to `ApplicationLifetime` to raise the `ApplicationStopping` event. This signals `WaitForShutdownAsync` to unblock the `Main` execution code. In the meantime, the POSIX signal handler returns with `Cancel = true` since the POSIX signal has been handled.
2. The `Main` execution code starts executing again and tells the host to `StopAsync()`, which in turn stops all the hosted services, and raises any other stopped events.
3. Finally, `WaitForShutdown` exits, allowing for any application clean-up code to execute, and for the `Main` method to exit gracefully.

Host shutdown in web server scenarios

There are various other common scenarios in which graceful shutdown works in Kestrel for both HTTP/1.1 and HTTP/2 protocols, and how you can configure it in different environments with a load balancer to drain traffic smoothly. While web server configuration is beyond the scope of this article, you can find more information on Configure options for the ASP.NET Core Kestrel web server documentation.

When the Host receives a shutdown signal (for example, CTL+C or `StopAsync`), it notifies the application by signaling ApplicationStopping. You should subscribe to this event if you have any long-running operations that need to finish gracefully.

Next, the Host calls IServer.StopAsync with a shutdown timeout that you can configure (default 30s). Kestrel (and Http.Sys) close their port bindings and stop accepting new connections. They also tell the current connections to stop processing new requests. For HTTP/2 and HTTP/3, a preliminary `GOAWAY` message is sent to the client. For HTTP/1.1,

they stop the connection loop because requests are processed in order. IIS behaves differently, by rejecting new requests with a 503 status code.

The active requests have until the shutdown timeout to complete. If they're all complete before the timeout, the server returns control to the host sooner. If the timeout expires, the pending connections and requests are aborted forcefully, which can cause errors in the logs and to the clients.

Load balancer considerations

To ensure a smooth transition of clients to a new destination when working with a load balancer, you can follow these steps:

- Bring up the new instance and start balancing traffic to it (you may already have several instances for scaling purposes).
- Disable or remove the old instance in the load balancer configuration so it stops receiving new traffic.
- Signal the old instance to shut down.
- Wait for it to drain or time out.

See also

- Dependency injection in .NET
- Logging in .NET
- Configuration in .NET
- Worker Services in .NET
- ASP.NET Core Web Host
- ASP.NET Core Kestrel web server configuration
- Generic host bugs should be created in the github.com/dotnet/runtime repo