# App startup in ASP.NET Core

12/05/2019 • 13 minutes to read • +10

**In this article**



By Rick Anderson, Tom Dykstra, and Steve Smith

The `Startup` class configures services and the app's request pipeline.

## The Startup class

ASP.NET Core apps use a `Startup` class, which is named `Startup` by convention. The `Startup` class:

- Optionally includes a ConfigureServices method to configure the app's *services*. A service is a reusable component that provides app functionality. Services are *registered* in `ConfigureServices` and consumed across the app via dependency injection (DI) or ApplicationServices.
- Includes a Configure method to create the app's request processing pipeline.

`ConfigureServices` and `Configure` are called by the ASP.NET Core runtime when the app starts:

C# Copy

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddRazorPages();
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseExceptionHandler("/Error");
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();

        app.UseRouting();

        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapRazorPages();
```

```
        });
    }
}
```

The preceding sample is for Razor Pages; the MVC version is similar.

The `Startup` class is specified when the app's host is built. The `Startup` class is typically specified by calling the WebHostBuilderExtensions.UseStartup<TStartup> method on the host builder:

C# Copy

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

The host provides services that are available to the `Startup` class constructor. The app adds additional services via `ConfigureServices`. Both the host and app services are available in `Configure` and throughout the app.

Only the following service types can be injected into the `Startup` constructor when using the Generic Host (IHostBuilder):

- IWebHostEnvironment
- IHostEnvironment
- IConfiguration

C# Copy

```
public class Startup
{
    private readonly IWebHostEnvironment _env;

    public Startup(IConfiguration configuration, IWebHostEnvironment env)
    {
        Configuration = configuration;
        _env = env;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        if (_env.IsDevelopment())
        {
        }
        else
        {
        }
    }
}
```

Most services are not available until the `Configure` method is called.

## Multiple Startup

When the app defines separate `Startup` classes for different environments (for example, `StartupDevelopment`), the appropriate `Startup` class is selected at runtime. The class whose name suffix matches the current environment is prioritized. If the app is run in the Development environment and includes both a `Startup` class and a `StartupDevelopment` class, the `StartupDevelopment` class is used. For more information, see Use multiple environments.

See The host for more information on the host. For information on handling errors during startup, see Startup exception handling.

# The ConfigureServices method

The ConfigureServices method is:

- Optional.
- Called by the host before the `Configure` method to configure the app's services.
- Where configuration options are set by convention.

The host may configure some services before `Startup` methods are called. For more information, see The host.

For features that require substantial setup, there are `Add{Service}` extension methods on IServiceCollection. For example, **Add**DbContext, **Add**DefaultIdentity, **Add**EntityFrameworkStores, and **Add**RazorPages:

C# Copy

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {

        services.AddDbContext<ApplicationDbContext>(options =>
            options.UseSqlServer(
                Configuration.GetConnectionString("DefaultConnection")));
        services.AddDefaultIdentity<IdentityUser>(
```

```
            options => options.SignIn.RequireConfirmedAccount = true)
            .AddEntityFrameworkStores<ApplicationDbContext>();

        services.AddRazorPages();
    }
```

Adding services to the service container makes them available within the app and in the `Configure` method. The services are resolved via dependency injection or from ApplicationServices.

# The Configure method

The Configure method is used to specify how the app responds to HTTP requests. The request pipeline is configured by adding middleware components to an IApplicationBuilder instance. `IApplicationBuilder` is available to the `Configure` method, but it isn't registered in the service container. Hosting creates an `IApplicationBuilder` and passes it directly to `Configure`.

The ASP.NET Core templates configure the pipeline with support for:

- Developer Exception Page
- Exception handler
- HTTP Strict Transport Security (HSTS)
- HTTPS redirection
- Static files
- ASP.NET Core MVC and Razor Pages

C# Copy

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
```

```
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddRazorPages();
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseExceptionHandler("/Error");
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();

        app.UseRouting();

        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapRazorPages();
        });
    }
}
```

The preceding sample is for Razor Pages; the MVC version is similar.

Each `Use` extension method adds one or more middleware components to the request pipeline. For instance, UseStaticFiles configures middleware to serve static files.

Each middleware component in the request pipeline is responsible for invoking the next component in the pipeline or short-circuiting the chain, if appropriate.

Additional services, such as `IWebHostEnvironment`, `ILoggerFactory`, or anything defined in `ConfigureServices`, can be specified in the `Configure` method signature. These services are injected if they're available.

For more information on how to use `IApplicationBuilder` and the order of middleware processing, see ASP.NET Core Middleware.

# Configure services without Startup

To configure services and the request processing pipeline without using a `Startup` class, call `ConfigureServices` and `Configure` convenience methods on the host builder. Multiple calls to `ConfigureServices` append to one another. If multiple `Configure` method calls exist, the last `Configure` call is used.

C# Copy

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureAppConfiguration((hostingContext, config) =>
            {
            })
            .ConfigureWebHostDefaults(webBuilder =>
```

```
            {
                webBuilder.ConfigureServices(services =>
                {
                    services.AddControllersWithViews();
                })
                .Configure(app =>
                {
                    var loggerFactory = app.ApplicationServices
                        .GetRequiredService<ILoggerFactory>();
                    var logger = loggerFactory.CreateLogger<Program>();
                    var env = app.ApplicationServices.GetRequiredService<IWebHostEnvironment>();
                    var config = app.ApplicationServices.GetRequiredService<IConfiguration>();

                    logger.LogInformation("Logged in Configure");

                    if (env.IsDevelopment())
                    {
                        app.UseDeveloperExceptionPage();
                    }
                    else
                    {
                        app.UseExceptionHandler("/Home/Error");
                        app.UseHsts();
                    }

                    var configValue = config["MyConfigKey"];
                });
            });
        });
}
```

# Extend Startup with startup filters

Use IStartupFilter:

- To configure middleware at the beginning or end of an app's Configure middleware pipeline without an explicit call to `Use{Middleware}`. `IStartupFilter` is used by ASP.NET Core to add defaults to the beginning of the pipeline without having to make the app author explicitly register the default middleware. `IStartupFilter` allows a different component call `Use{Middleware}` on behalf of the app author.
- To create a pipeline of `Configure` methods. IStartupFilter.Configure can set a middleware to run before or after middleware added by libraries.

`IStartupFilter` implements Configure, which receives and returns an `Action<IApplicationBuilder>`. An IApplicationBuilder defines a class to configure an app's request pipeline. For more information, see Create a middleware pipeline with IApplicationBuilder.

Each `IStartupFilter` can add one or more middlewares in the request pipeline. The filters are invoked in the order they were added to the service container. Filters may add middleware before or after passing control to the next filter, thus they append to the beginning or end of the app pipeline.

The following example demonstrates how to register a middleware with `IStartupFilter`. The `RequestSetOptionsMiddleware` middleware sets an options value from a query string parameter:

C# Copy

```
public class RequestSetOptionsMiddleware
{
    private readonly RequestDelegate _next;

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

    // Test with https://localhost:5001/Privacy/?option=Hello
    public async Task Invoke(HttpContext httpContext)
    {
        var option = httpContext.Request.Query["option"];
```

```
        if (!string.IsNullOrWhiteSpace(option))
        {
            httpContext.Items["option"] = WebUtility.HtmlEncode(option);
        }

        await _next(httpContext);
    }
}
```

The `RequestSetOptionsMiddleware` is configured in the `RequestSetOptionsStartupFilter` class:

C# Copy

```
public class RequestSetOptionsStartupFilter : IStartupFilter
{
    public Action<IApplicationBuilder> Configure(Action<IApplicationBuilder> next)
    {
        return builder =>
        {
            builder.UseMiddleware<RequestSetOptionsMiddleware>();
            next(builder);
        };
    }
}
```

The `IStartupFilter` is registered in the service container in ConfigureServices.

C# Copy

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
```

```
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
           .ConfigureAppConfiguration((hostingContext, config) =>
           {
           })
         .ConfigureWebHostDefaults(webBuilder =>
         {
             webBuilder.UseStartup<Startup>();
         })
        .ConfigureServices(services =>
        {
            services.AddTransient<IStartupFilter,
                      RequestSetOptionsStartupFilter>();
        });
}
```

When a query string parameter for `option` is provided, the middleware processes the value assignment before the ASP.NET Core middleware renders the response.

Middleware execution order is set by the order of `IStartupFilter` registrations:

- Multiple `IStartupFilter` implementations may interact with the same objects. If ordering is important, order their `IStartupFilter` service registrations to match the order that their middlewares should run.
- Libraries may add middleware with one or more `IStartupFilter` implementations that run before or after other app middleware registered with `IStartupFilter`. To invoke an `IStartupFilter` middleware before a middleware added by a library's `IStartupFilter`:
  - Position the service registration before the library is added to the service container.
  - To invoke afterward, position the service registration after the library is added.

# Add configuration at startup from an external assembly

An IHostingStartup implementation allows adding enhancements to an app at startup from an external assembly outside of the app's `Startup` class. For more information, see Use hosting startup assemblies in ASP.NET Core.

# Additional resources

- The host
- Use multiple environments in ASP.NET Core
- ASP.NET Core Middleware
- Logging in .NET Core and ASP.NET Core
- Configuration in ASP.NET Core

---

**Is this page helpful?**

Yes No

---