Agenda: ASP.NET Core Startup Process

- IHost and IHostBuilder
- The Startup Class
- Configure and ConfigureService Methods
- Registering App Services using Dependency Injection
- Example of Dependency Injection (DI)

IHost and IHostBuilder

.NET apps configure and launch a *host*. The host is responsible for app startup and lifetime management.

Two host APIs are available for use:

- **Web Host –** Suitable for hosting web apps.(which remains available only for backward compatibility)
- **Generic Host** – Suitable for hosting all types of apps. (From 3.1, Generic Host is recommended for all app types)

Program.cs

Specify the **Startup** class with the WebHostBuilderExtensions **UseStartup**<TStartup> method:

```
public class Program
{
   public static void Main(string[] args)
   {
      IHostBuilder builder = CreateHostBuilder(args);
      IHost host = builder.Build();
      host.Run();
      //CreateHostBuilder(args).Build().Run();
   }
   //For an HTTP workload, the CreateHostBuilder calls ConfigureWebHostDefaults:
   public static IHostBuilder CreateHostBuilder(string[] args)
   {
        var hostbuilder = Host.CreateDefaultBuilder(args);
        return hostbuilder.ConfigureWebHostDefaults(webBuilder =>
        {
           webBuilder.UseStartup<Startup>();
        });
   }
}
```

- **IHost**: Represents a configured host.
- **IHostBuilder**: A builder for IHost
- IHostBuilder.**Build**(): Builds an IHost which hosts a web application.
- IHost.**Run**(): Runs a web application and block the calling thread until host shutdown.
- **Host:** Provides **convenience** methods for creating instances of **IHost** and **IHostBuilder** with pre-configured defaults.
- Host.**CreateDefaultBuilder**(args): returns a new instance of the **HostBuilder** class.
- **ConfigureWebHostDefaults** Initializes a new instance of the **IWebHostBuilder** with **pre-configured defaults**. The following **defaults** are applied to the returned HostBuilder:
  a. Uses **Kestrel** as the web server and configure it using the application's configuration providers.
  b. Set the content root to the path returned by **Directory**.**GetCurrentDirectory**()
  c. Loads **host configuration** from:
    a. Environment variables prefixed with `ASPNETCORE_` (for example, `ASPNETCORE_ENVIRONMENT`).
    b. Command-line arguments.
  d. Loads **app configuration** in the following order from:
    a. *appsettings.json*.
    b. *appsettings.{Environment}.json*.
    c. Environment variables.
    d. Command-line arguments.
  e. Configure the ILoggerFactory to log to the console and debug output, and
  f. When running behind IIS, Enables IIS integration which configures the app's base address and port.
- WebBuilder.**UseStartup**<TStartup>: Returns IWebHostBuilder and specifies the **startup type** to be used by the web host.

The Startup Class

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

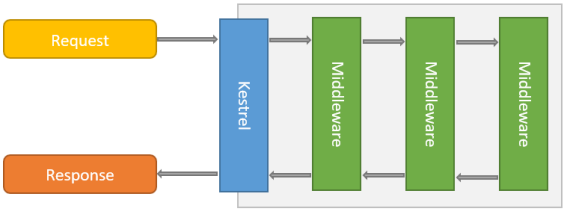
- The Startup class constructor accepts **dependencies** defined by the **host**.
- **ConfigureServices** method is used to **configure the app's services**. Services are configured here and consumed across the app via **dependency injection (DI)** or ApplicationServices.
- **Configure** method is used to create the **app's request processing pipeline**.

```
public class Startup
{
```

```
    public IConfiguration Configuration { get; }
    //The Startup class constructor accepts dependencies defined by the host.
    public Startup(IWebHostEnvironment env, IConfiguration configuration, ILoggerFactory loggerFactory)
    {
        Configuration = configuration;
    }
    // This method gets called by the runtime. Use this method to add services to the container.
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllersWithViews ();
    }
    // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
            app.UseBrowserLink();
        }
        else
        {
            app.UseExceptionHandler("/Error");
        }
        app.UseStaticFiles();
        app.UseRoute();
        app. UseEndpoints (endpoints =>
        {
            endpoints.MapControllerRoute (
                name: "default",
                pattern: "{controller}/{action=Index}/{id?}");
        });
    }
}
```

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.



ASP.NET Core Pipeline

- Each **Use** extension method adds a middleware component to the request pipeline. For instance, the **UseEndpoints and UseRoute** extension methods adds the routing middleware to the request pipeline and configures MVC as the default handler.
- The ASP.NET Core templates configure the pipeline with support for a developer exception page, BrowserLink, error pages, static files, and ASP.NET MVC

The ConfigureService Method

- It's Optional.
- It's called by the web host **before** the **Configure** method to configure the app's services.
- It's where **configuration options** are set by convention.
- **The web host provides some services that are available to the Startup class constructor. The app adds additional services via ConfigureServices. Both the host and app services are then available in Configure and throughout the application**.

For features that require substantial setup, there are **Add[Service]** extension methods on IServiceCollection.

A typical web app registers services for Entity Framework, Identity, and MVC:

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddDbContext<ApplicationDbContext>(options =>
                        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
   services.AddIdentity<ApplicationUser, IdentityRole>()
                        .AddEntityFrameworkStores<ApplicationDbContext>()
                        .AddDefaultTokenProviders();
   // Add framework services.
   services. AddControllersWithViews();
   // Add application services.
   services.AddTransient<IEmailSender, AuthMessageSender>();
   services.AddTransient<ISmsSender, AuthMessageSender>();
```

```
}
```

Registering App Services using Dependency Injection

Dependency injection (DI) is a technique for achieving loose coupling between objects and their collaborators, or dependencies.

Most often, classes will declare their dependencies via their constructor, allowing them to follow the Explicit Dependencies Principle. This approach is known as "**constructor injection**".

Dependency Injection and Registering Services

You can register your own application services as follows.

```
services.AddTransient<IEmailSender, EmailMessageSender>();
services.AddTransient<ISmsSender, SmsMessageSender>()
```

The first generic type represents the type (typically an interface) that will be requested from the container. The second generic type represents the concrete type that will be instantiated by the container and used to fulfill such requests.

Controlling Service Lifetime:

- **Transient**: Transient lifetime services are created each time they're requested. This lifetime works best for lightweight, stateless services.
- **Scoped:** Scoped lifetime services are created once per request.
- **Singleton:** Singleton lifetime services are created the first time they're requested (or when `ConfigureServices` is run if you specify an instance there) and then every subsequent request will use the same instance.

```
public interface IOperation
{
   Guid OperationId { get; }
}
public interface IOperationTransient : IOperation
{ }
public interface IOperationScoped : IOperation
{ }
public interface IOperationSingleton : IOperation
{ }
public interface IOperationSingletonInstance : IOperation
```

```
{ }

public class Operation : IOperationTransient, IOperationScoped, IOperationSingleton, IOperationSingletonInstance
{
   Guid _OperationId;
   public Guid OperationId { get => _OperationId; }

   public Operation()
   {
      _OperationId = Guid.NewGuid();
   }

   public Operation(Guid operationId)
   {
      _OperationId = operationId;
   }
}

public class OperationService
{
   public IOperationTransient TransientOperation { get; }
   public IOperationScoped ScopedOperation { get; }
   public IOperationSingleton SingletonOperation { get; }
   public IOperationSingletonInstance SingletonInstanceOperation { get; }

   public OperationService(IOperationTransient transientOperation,
      IOperationScoped scopedOperation,
      IOperationSingleton singletonOperation,
      IOperationSingletonInstance instanceOperation)
   {
      TransientOperation = transientOperation;
      ScopedOperation = scopedOperation;
      SingletonOperation = singletonOperation;
      SingletonInstanceOperation = instanceOperation;
   }
```

```
}

public class ContactModel : PageModel
{
   private readonly OperationService _operationService;
   private readonly IOperationTransient _transientOperation;
   private readonly IOperationScoped _scopedOperation;
   private readonly IOperationSingleton _singletonOperation;
   private readonly IOperationSingletonInstance _singletonInstanceOperation;
   private OperationService _operationServices;
   public ContactModel(OperationService operationService,
      IOperationTransient transientOperation,
      IOperationScoped scopedOperation,
      IOperationSingleton singletonOperation,
      IOperationSingletonInstance singletonInstanceOperation)
   {
      _operationService = operationService;
      _transientOperation = transientOperation;
      _scopedOperation = scopedOperation;
      _singletonOperation = singletonOperation;
      _singletonInstanceOperation = singletonInstanceOperation;
   }

   public string Transient { get; set; }
   public string Scoped { get; set; }
   public string Singleton { get; set; }
   public string SingletonInstance { get; set; }
   public string TransSingletonInstanceient { get; set; }

   public OperationService Service { get; set; }
   public void OnGet()
   {
     Transient = _transientOperation.OperationId.ToString();
     Scoped = _scopedOperation.OperationId.ToString();
     Singleton = _singletonOperation.OperationId.ToString();
```

```
    SingletonInstance = _singletonInstanceOperation.OperationId.ToString();

    Service = _operationService;

  }
}
```

Add the following to **Startup.ConfigureServices**

```
services.AddTransient<IOperationTransient, Operation>();
services.AddScoped<IOperationScoped, Operation>();
services.AddSingleton<IOperationSingleton, Operation>();
services.AddSingleton<IOperationSingletonInstance>(new Operation(Guid.Empty));
services.AddTransient<OperationService>();
```

Observe which of the `OperationId` values vary within a request, and between requests.

- ***Transient*** objects are always different; a new instance is provided to every controller and every service.
- ***Scoped*** objects are the same within a request, but different across different requests
- ***Singleton*** objects are the same for every object and every request (regardless of whether an instance is provided in `ConfigureServices`)

Thread safety

Singleton services need to be thread safe. If a singleton service has a dependency on a transient service, the transient service may also need to be thread safe depending how it's used by the singleton.

Action Injection with FromServices

Sometimes you don't need a service for more than one action within your controller. In this case, it may make sense to inject the service as a parameter to the action method. This is done by marking the parameter with the attribute `[FromServices]` as shown here:

```
public IActionResult About([FromServices] IMathService srv)
{
  . . .
  return View();
```

```
}
```