**program.cs**

It is the entry point of our app, where we configure the Web host. The program class configures the application infrastructure like Web host, logging, Dependency injection container, IIS integration, etc. It is created, configured, built using the createdefaultbuilder method of the Main method in the **program class**.

## What is Program.cs?

The Program class contains the method Main, which is the entry point of the ASP.NET Core applications.

The Main method is similar to the Main method of a console Applications. That is because all the .NET Core applications basically are console applications.

We build other types of applications like MVC Web Application or Razor page application over the console app.

The main purpose of the Program class is to configure the applications infrastructure.

The program class creates the **Web Host** at the startup.

it then configures the logging, [Dependency Injection Container](https://www.tektutorialshub.com/asp-net-core/asp-net-core-dependency-injection/#di-container). configuration system, [Kestrel Web server](https://www.tektutorialshub.com/asp-net-core/asp-net-core-kestrel-web-server/), IIS Integration, etc.

It also adds the framework services to the [DI container](https://www.tektutorialshub.com/asp-net-core/asp-net-core-middleware-request-pipeline/#what-is-middleware) so that we can use it. The code for the program class is automatically generated for us and most probably enough for most of the projects.

## What is Web Host?

The Web host is responsible for starting the app and running it. It is created when the application starts. Web host creates a server, which listens for the HTTP requests. It configures the [request pipeline](https://www.tektutorialshub.com/asp-net-core/asp-net-core-middleware-request-pipeline/) (or [middleware pipeline](https://www.tektutorialshub.com/asp-net-core/asp-net-core-middleware-request-pipeline/)). Also, It sets up DI container, where we add our services. Managing the lifetime of the services is also a task of Webhost.

Basically, Web Host ready’s our app to receive the requests. But, the Web Host needs to be created and configured. We do that in Main method.

## Main method

Open the Applications we built in the [Hello World Example](https://www.tektutorialshub.com/asp-net-core/asp-net-core-getting-started/) and open the program.cs

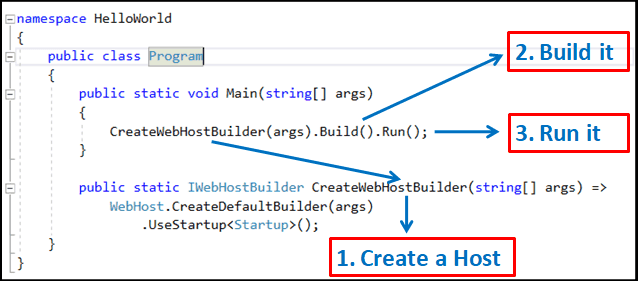
This class has a single method Main

|  |  |
| --- | --- |
|  | using System;  using System.Collections.Generic;  using System.IO;  using System.Linq;  using System.Threading.Tasks;  using Microsoft.AspNetCore;  using Microsoft.AspNetCore.Hosting;  using Microsoft.Extensions.Configuration;  using Microsoft.Extensions.Logging;  namespace HelloWorld  {  public class Program  {  public static void Main(string[] args)  {  CreateWebHostBuilder(args).Build().Run();  }  public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>  WebHost.CreateDefaultBuilder(args).UseStartup<Startup>();  }  } |

This is the entry point to our application.

The following image shows how the web host is created.

1. CreateWebHostBuilder creates the host and provides the configuration to it.
2. The build method builds it using the provided configuration
3. The Run method runs it and it then listens for HTTP requests.

The Main method of Program.cs class in ASP.NET Core

## Create the Host

### CreateWebHostBuilder

The CreateWebHostBuilder is a static method, which creates and configures the **Host** and then returns it.

|  |  |
| --- | --- |
|  | public static IWebHostBuilder CreateWebHostBuilder() => string[] args          WebHost.CreateDefaultBuilder(args)          .UseStartup<Startup>(); |

The above syntax is known as “[expression bodied function member](https://lostechies.com/jimmybogard/2015/12/17/c-6-feature-review-expression-bodied-function-members/).” feature that was introduced in C#

The above code is the same as the following function

|  |  |
| --- | --- |
|  | public static IWebHostBuilder CreateWebHostBuilder(string[] args)  {    return WebHost.CreateDefaultBuilder(args)                  .UseStartup<Startup>();  } |

The method above uses the WebHost helper **static** class

The CreateDefaultBuilder method of the WebHost helper class is responsible for creating the IWebHostBuilder instance, with the desired configuration

### CreateDefaultBuilder

The CreateDefaultBuilder performs the following tasks.

1. Sets the content root to Directory.GetCurrentDirectory.
2. Loads optional configuration from
   1. Appsettings.json
   2. Appsettings.{Environment}.json.
   3. User secrets when the app runs in the Development environment.
   4. Environment variables
   5. Command-line arguments.
3. Enable logging
4. Sets up the [Dependency Injection Container](https://www.tektutorialshub.com/asp-net-core/asp-net-core-dependency-injection/#di-container).
5. Configures [Kestre](https://www.tektutorialshub.com/asp-net-core/asp-net-core-kestrel-web-server/)l as the webserver
6. Adds Framework Services to the DI Container
7. Integrates the Kestrel run with IIS

You can peek into the [source code](https://github.com/dotnet/aspnetcore/blob/1480b998660d2f77d0605376eefab6a83474ce07/src/DefaultBuilder/src/WebHost.cs) for the CreateDefaultBuilder from the ASP.NET Core meta package

|  |  |
| --- | --- |
|  | public static IWebHostBuilder CreateDefaultBuilder(string[] args)  {      var builder = new WebHostBuilder();        if (string.IsNullOrEmpty(builder.GetSetting(WebHostDefaults.ContentRootKey)))      {          builder.UseContentRoot(Directory.GetCurrentDirectory());      }     if (args != null)      {          builder.UseConfiguration(new ConfigurationBuilder().AddCommandLine(args).Build());      }        builder.ConfigureAppConfiguration((hostingContext, config) =>      {          var env = hostingContext.HostingEnvironment;            config.AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)                .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true, reloadOnChange: true);            if (env.IsDevelopment())          {              var appAssembly = Assembly.Load(new AssemblyName(env.ApplicationName));              if (appAssembly != null)              {                  config.AddUserSecrets(appAssembly, optional: true);              }          }            config.AddEnvironmentVariables();            if (args != null)          {              config.AddCommandLine(args);          }      })            .ConfigureLogging((hostingContext, logging) =>      {          logging.AddConfiguration(hostingContext.Configuration.GetSection("Logging"));          logging.AddConsole();          logging.AddDebug();          logging.AddEventSourceLogger();      }).          UseDefaultServiceProvider((context, options) =>      {          options.ValidateScopes = context.HostingEnvironment.IsDevelopment();      });          ConfigureWebDefaults(builder);        return builder;  } |

There is a lot of code there. Let us look at each line

#### Setting the content Root

|  |  |
| --- | --- |
|  | builder.UseContentRoot(Directory.GetCurrentDirectory()); |

This sets the content root to Directory.GetCurrentDirectory. This basically sets the current directory as the root of the application.

#### Loading Configuration file

The next section of the code loads the configuration from the various sources and in the following order

1. Appsettings.json.
2. appsettings.{Environment}.json.
3. User secrets when the app runs in the Development environment.
4. Environment variables.
5. Command-line arguments.

|  |  |
| --- | --- |
|  | if (args != null)  {    builder.UseConfiguration(new ConfigurationBuilder().AddCommandLine(args).Build());  }    builder.ConfigureAppConfiguration((hostingContext, config) =>  {    var env = hostingContext.HostingEnvironment;      config.AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)           .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true, reloadOnChange: true);      if (env.IsDevelopment())    {      var appAssembly = Assembly.Load(new AssemblyName(env.ApplicationName));      if (appAssembly != null)      {        config.AddUserSecrets(appAssembly, optional: true);      }    }      config.AddEnvironmentVariables();      if (args != null)    {      config.AddCommandLine(args);    }  }) |

#### Enable Logging

The code which enables & configures logging follows next.

|  |  |
| --- | --- |
|  | .ConfigureLogging((hostingContext, logging) =>  {      logging.AddConfiguration(hostingContext.Configuration.GetSection("Logging"));      logging.AddConsole();      logging.AddDebug();      logging.AddEventSourceLogger();  }). |

This reads the configuration rules specified in the “Logging” section of the configuration files and configures logging for console and debug output.

#### Sets up the DI Container

The UseDefaultServiceProvider method sets up the Dependency Injection container and configures it.

|  |  |
| --- | --- |
|  | UseDefaultServiceProvider((context, options) =>  {    options.ValidateScopes = context.HostingEnvironment.IsDevelopment();  }); |

#### Configures WebHost

Finally, the code calls the ConfigureWebDefaults method. This method configures the Web Host. The [source code is available on GitHub](https://github.com/dotnet/aspnetcore/blob/1480b998660d2f77d0605376eefab6a83474ce07/src/DefaultBuilder/src/WebHost.cs).

|  |  |
| --- | --- |
|  | internal static void ConfigureWebDefaults(IWebHostBuilder builder)  {        builder.ConfigureAppConfiguration((ctx, cb) =>      {          if (ctx.HostingEnvironment.IsDevelopment())          {              StaticWebAssetsLoader.UseStaticWebAssets(ctx.HostingEnvironment, ctx.Configuration);          }      });          builder.UseKestrel((builderContext, options) =>      {          options.Configure(builderContext.Configuration.GetSection("Kestrel"));      })        .ConfigureServices((hostingContext, services) =>      {          // Fallback          services.PostConfigure<HostFilteringOptions>(options =>          {              if (options.AllowedHosts == null || options.AllowedHosts.Count == 0)              {                 // "AllowedHosts": "localhost;127.0.0.1;[::1]"                 var hosts = hostingContext.Configuration["AllowedHosts"]?.Split(new[] { ';' }, StringSplitOptions.RemoveEmptyEntries);                 // Fall back to "\*" to disable.                 options.AllowedHosts = (hosts?.Length > 0 ? hosts : new[] { "\*" });              }          });          // Change notification          services.AddSingleton<IOptionsChangeTokenSource<HostFilteringOptions>>(                      new ConfigurationChangeTokenSource<HostFilteringOptions>(hostingContext.Configuration));            services.AddTransient<IStartupFilter, HostFilteringStartupFilter>();            if (string.Equals("true", hostingContext.Configuration["ForwardedHeaders\_Enabled"], StringComparison.OrdinalIgnoreCase))          {              services.Configure<ForwardedHeadersOptions>(options =>              {                  options.ForwardedHeaders = ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwardedProto;                          // Only loopback proxies are allowed by default. Clear that restriction because forwarders are                          // being enabled by explicit configuration.                          options.KnownNetworks.Clear();                  options.KnownProxies.Clear();              });                services.AddTransient<IStartupFilter, ForwardedHeadersStartupFilter>();          }            services.AddRouting();      })        .UseIIS()      .UseIISIntegration();  } |

#### Kestral

The First method tells the host to use the[Kestrel Web server](https://www.tektutorialshub.com/asp-net-core/asp-net-core-kestrel-web-server/). Kestrel is a cross-platform Managed HTTP server-based. This Server allows ASP.NET Core application runs on OS other than Windows.

|  |  |
| --- | --- |
|  | builder.UseKestrel((builderContext, options) =>  {    options.Configure(builderContext.Configuration.GetSection("Kestrel"));  }) |

#### Configure services

This section configures the important services & adds them to DI Container.

|  |  |
| --- | --- |
|  | .ConfigureServices((hostingContext, services) =>  {     //Services added here  } |

#### Use IIS Or Integration

Configures IIS Server to Host the App. There are two ways in which you can host the App. One is in process and the other one is out of process.

The in Process runs the app inside the IIS Process and is configured by UseIIS().

The Out of process runs in a separate process and uses the [Kestrel server](https://www.tektutorialshub.com/asp-net-core/asp-net-core-kestrel-web-server/). The IIS then acts as a reverse proxy forwarding the requests to the [Kestrel](https://www.tektutorialshub.com/asp-net-core/asp-net-core-kestrel-web-server/). This is configured by the method UseIISIntegration().

|  |  |
| --- | --- |
| 1  2  3  4 | .UseIIS()  .UseIISIntegration(); |

#### Startup class

The Host is created and configured, but before building & running we need to further configure with the application. We do this in the startup class. By default, the class is named as the startup class. We let the builder know the location of the startup class by using the UseStartup method.

|  |  |
| --- | --- |
|  | .UseStartup<Startup>() |

The startup class contains two methods. One is ConfigureServices (which is optional) and the other one is Configure. The ConfigureServices is where we configure services we created in the app and adds them to the DI Container. In the Configure method, we create the request pipeline by adding the middlewares.

The CreateWebHostBuilder will invoke the ConfigureServices & configure method from the startup class and configure the host further

## Build & Run it

CreateWebHostBuilder creates the Web Host and returns it back. The Build & Run methods are invoked and the app starts and begins listening for HTTP requests.

|  |  |
| --- | --- |
|  | CreateWebHostBuilder(args).Build().Run(); |

## Summary

The Main method of the program.cs class is the entry point of our application. It configures & builds the **Web host**. The web host is responsible for running our app. Most of the plumbing required to configure host is already done for us in the createdefaultbuilder method, which is invoked in the Main method. We can further add our custom configuration is **startup class**, which we cover in the next tutorial.

## The Kestrel is the new default web server that is included in the ASP.NET Core project templates. It runs within the application process making it completely self-contained. What is Kestrel?

The Kestrel is open-source, cross-platform, event-driven, asynchronous I/O based HTTP server. It is developed to host ASP.NET Core applications on any platform. It is included by default in the ASP.NET Core applications.

It is based on [libuv](https://github.com/libuv/libuv)

Kestel is an Open source library and is available at [GitHub](https://github.com/aspnet/KestrelHttpServer)

## Why Kestrel

The older ASP.NET applications are tightly coupled to the Internet Information Services or IIS.

The IIS is a complete web server with all the features that you require out of a Web Server. Over the period it has grown into a matured web server and along the way, it has added a lot of weight and bloat. It has become one of the best Web servers around and at the same time, it is one of the slowest.

ASP.NET being tightly coupled with IIS carried the burden of the IIS

The newly designed ASP.NET Core applications are now completely decoupled from the IIS. This decoupling makes ASP.NET Core run on any platform making it truly cross-platform

But, it still needs to have the ability to listen to HTTP requests and send the response back to the Client. That is where Kestrel comes in

## Using Kestrel

The Kestrel runs in in-process in the ASP.NET Core Applications. Hence, It runs independently of the environment in which it lives. The kestrel Web server is available in the namespace Microsoft.AspNetCore.Server.Kestrel

We looked at the program and startup class in the tutorial [Application startup](https://www.tektutorialshub.com/asp-net-core/asp-net-core-startup-class/) in ASP.NET Core

The Program class contains a static void Main function, which is the entry point to our Application

public class Program

{

    public static void Main(string[] args)

    {

        BuildWebHost(args).Run();

    }

    public static IWebHost BuildWebHost(string[] args) =>

        WebHost.CreateDefaultBuilder(args)

        .UseStartup<Startup>()

        .Build();

}

The Main method invokes CreateDefaultBuilder, which is responsible to create the web application host.

The CreateDefaultBuilder is a helper method ( click here for the [source code](https://github.com/aspnet/MetaPackages/blob/master/src/Microsoft.AspNetCore/WebHost.cs))

It calls the UseKestrel method to registers the Kestrel as the server that will be used to host our application

### Behind another Webserver

Kestrel is not a fully-featured web server. But that is what makes it fast.

It is not advisable to run Kestrel as a standalone web server in the Production environment. It is recommended to run it behind a Fully Featured Web server like IIS, Nginx, Apache, etc. In such a scenario, the Web server acts as a reverse proxy server

The reverse proxy server takes the HTTP request from the internet and passes it to the kestrel server just the way it is received.

The IIS can take the HTTP request and perform some useful processing like logging, request filtering, URL rewrites before passing the request to Kestrel.

The CreateDefaultBuilder method calls the UseIISIntegration, which tells ASP.NET that the application will be using the IIS as a reverse proxy in front of Kestrel

**Benefits of Kestrel Web Server**

1. Kestrel is fast. It is not a fully-featured website and does not provide the many features that you expect from a standard web server. Hence it makes it lightweight in design and fast.
2. Is supports all the versions of .NET Core
3. It is cross-platform. You can run it on Windows/Linux or Mac. This is something the ASP.NET missed always
4. It is very simple to configure and run. In fact, it is already configured, when you create a [new ASP.NET Core project](https://www.tektutorialshub.com/asp-net-core/asp-net-core-getting-started/) in Visual Studio
5. Supports HTTPS
6. Supports HTTP/2 (except on macOS. But will be supported in a future release).

## Alternatives to Kestrel

The Kestrel is not the only way to host ASP.NET Core applications. There is another webserver implementation available in Windows known as HTTP.SYS

The [HTTP.sys](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/httpsys?view=aspnetcore-2.2)  is a Windows-only HTTP server based on the [Http.Sys kernel driver](https://docs.microsoft.com/en-us/windows/win32/http/http-api-start-page).

ASP.**NET Core** supports the **dependency injection** (DI) software design pattern, which is a technique for achieving Inversion of Control (IoC) between classes and their **dependencies**. For more information specific to **dependency injection** within MVC controllers, see **Dependency injection** into controllers in ASP.**NET Core**.

The **appsettings**. **json** file is an application configuration file used to store configuration settings such as database connections strings, any application scope global variables, etc. If you open the ASP.**NET Core appsettings**. **json** file, then you see the following code by default which is created by visual studio.

{

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft": "Warning",

"Microsoft.Hosting.Lifetime": "Information"

}

},

"AllowedHosts": "\*",

**"MyCustomKey": "MyCustomKey Value coming from appsettings.json"**

}

##### **How to access the configuration information in the ASP.NET Core application?**

To access the configuration information within the Startup class, you need to use the IConfiguration service which is provided by the ASP.NET Core Framework. So, what you need to do is just inject the IConfiguration service through the constructor of the Startup class. To do so modify the Startup class which is present in the Startup.cs file as shown below.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.AspNetCore.Mvc;

using Microsoft.Extensions.Configuration;

namespace WebApplication3.Controllers

{

public class First : Controller

{

IConfiguration configuration;

public First(IConfiguration iConfig)

{

configuration = iConfig;

}

public IActionResult Index()

{

string dbConn = configuration.GetSection("MyCustomKey").Value;

ViewBag.va = dbConn;

return View();

}

}

}

We have added the following JSON object.

1. "MySettings": {
2. "DbConnection": "abc",
3. "Email": "abc@domain.com",
4. "SMTPPort": "5605"
5. }
6. string dbConn = configuration.GetSection("MySettings").GetSection("DbConnection").Value;

In the first method, we are getting our section from the configuration object. Inside this, we are getting another section that contains the value.

1. string dbConn2 = configuration.GetValue<string>("MySettings:DbConnection");