.NET Core

- Minhazul Hayat Khan

What is a framework?

And the advantages of using frameworks…

Advantages of using a framework

Better programming practices and design patterns

Secured codes, proper coding structures.

Reduce duplication and redundant coding.

Easier to work and consistent codes with fewer bugs.

Many prebuilt functionality and availability of 3rd party plugins/extensions.

Examples of frameworks

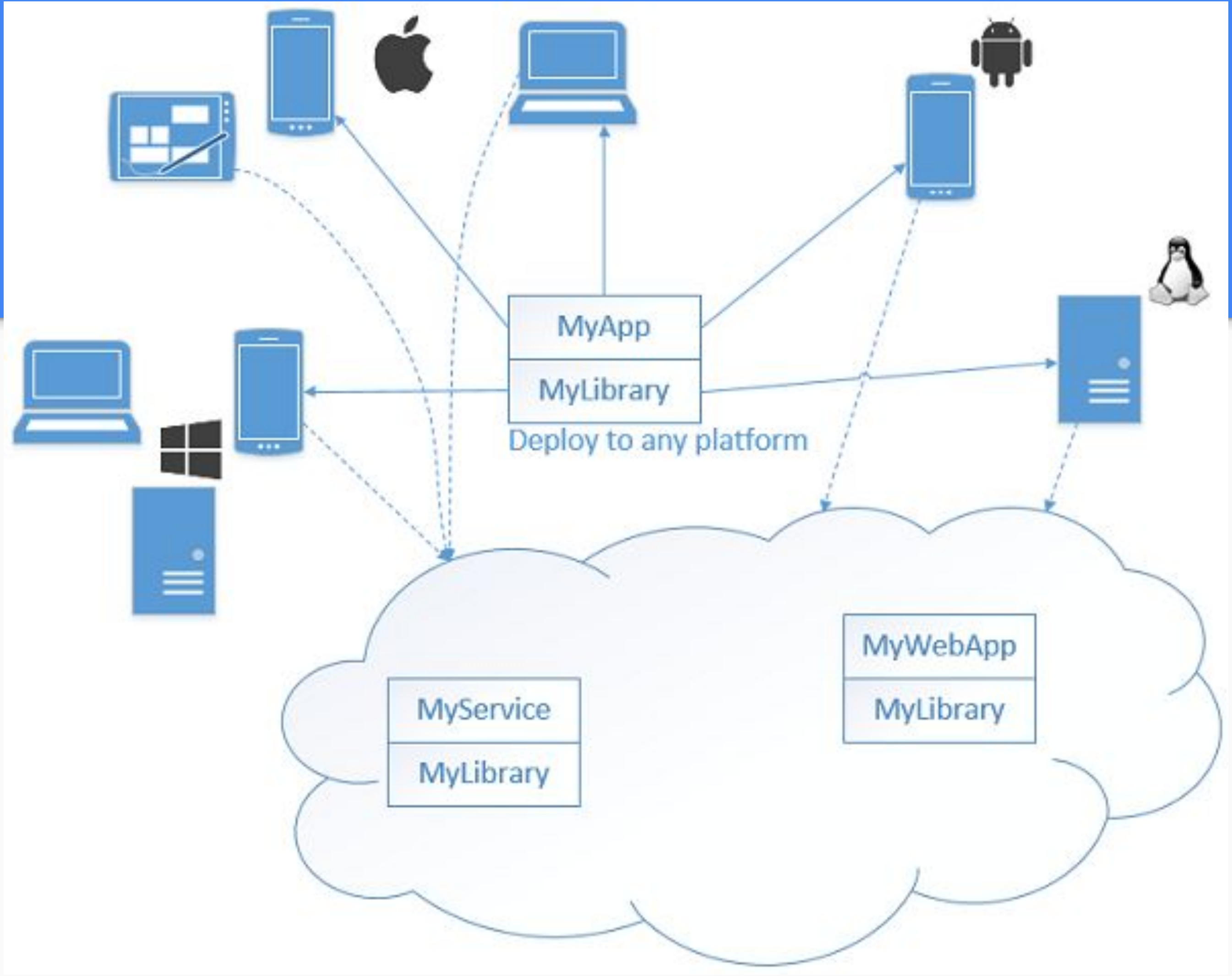
- For front end (mostly)
  a. Angular
  b. React
  c. Vue
  d. Servlet
- Backend (mostly + FE)
  e. .net framework (c#)
  f. Spring boot (java)
  g. Laravel (php)
  h. Django (python)

- Mobile app
  a. Flutter (dart)
  b. React native (js)
  c. Xamarin (c#)
  d. Ionic (js)
- Data Science
  a. Tensorflow
  b. PyTorch
  c. Scikit learn

The need of a new framework...

.NET Core

- Cross Platform
- Open-source
- gRPC, ML.NET for Machine Learning
- Building blocks of .NET Core are CoreCLR and CoreFX
- Maintained by Microsoft and .NET community
- IIS, nginx, Apache, Docker or self hosted


MyApp
MyLibrary
Deploy to any platform
MyService
MyLibrary
MyWebApp
MyLibrary

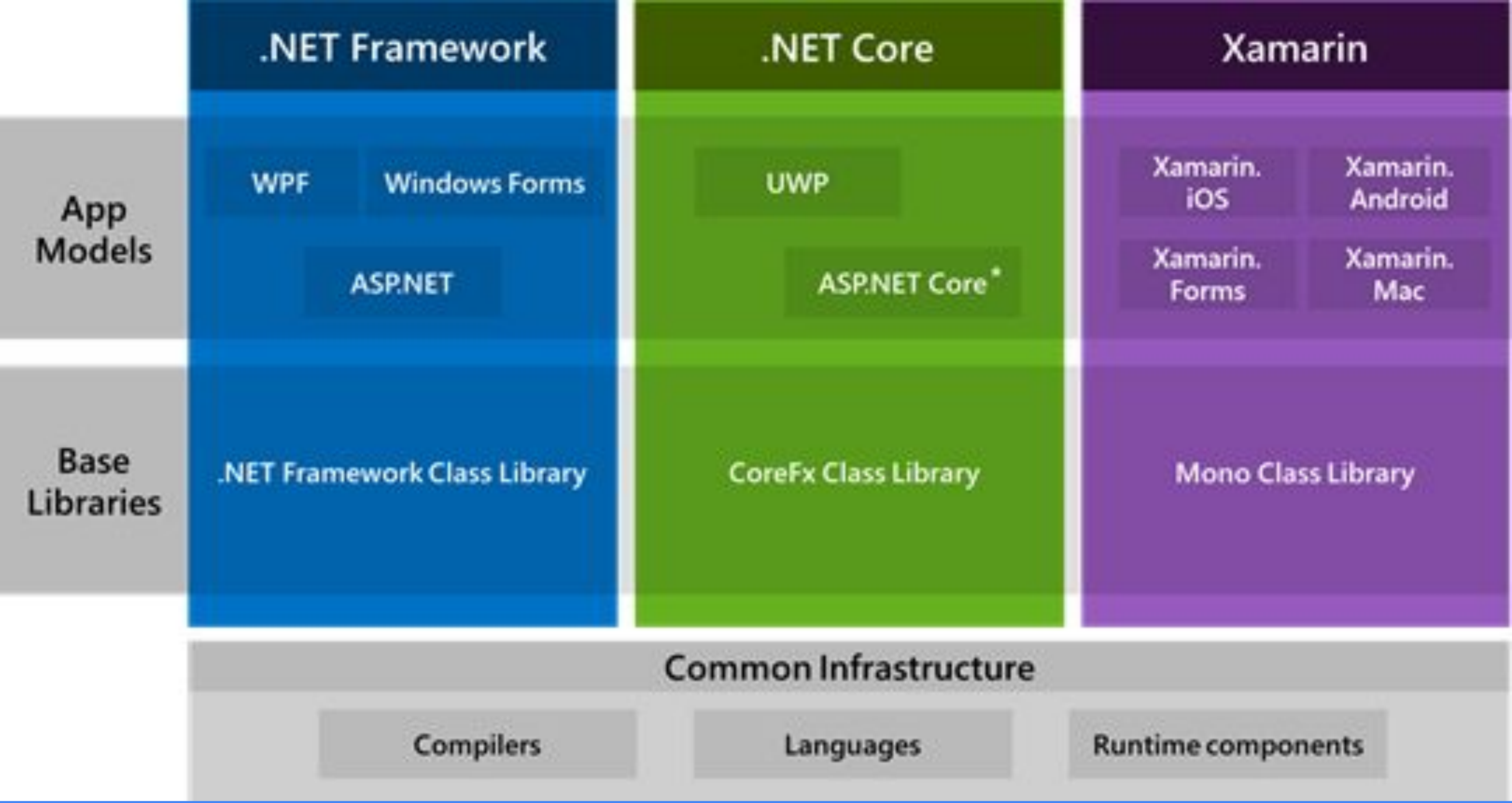
.NET Framework vs .NET Core

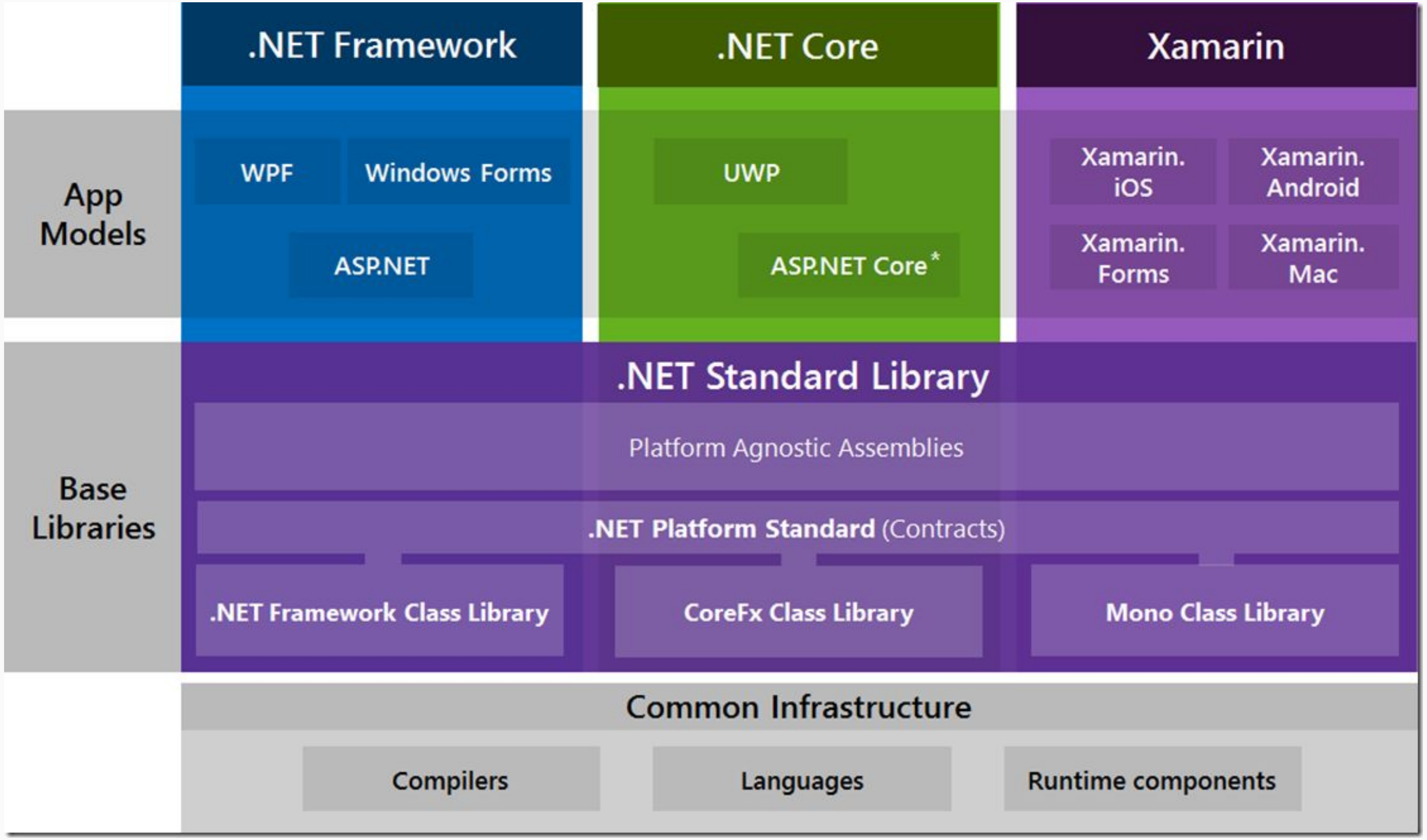
- Windows only
- Windows Form, WPF
- All the libraries are bundled

- Cross-Platform needs
- Command line style development
- Lightweight

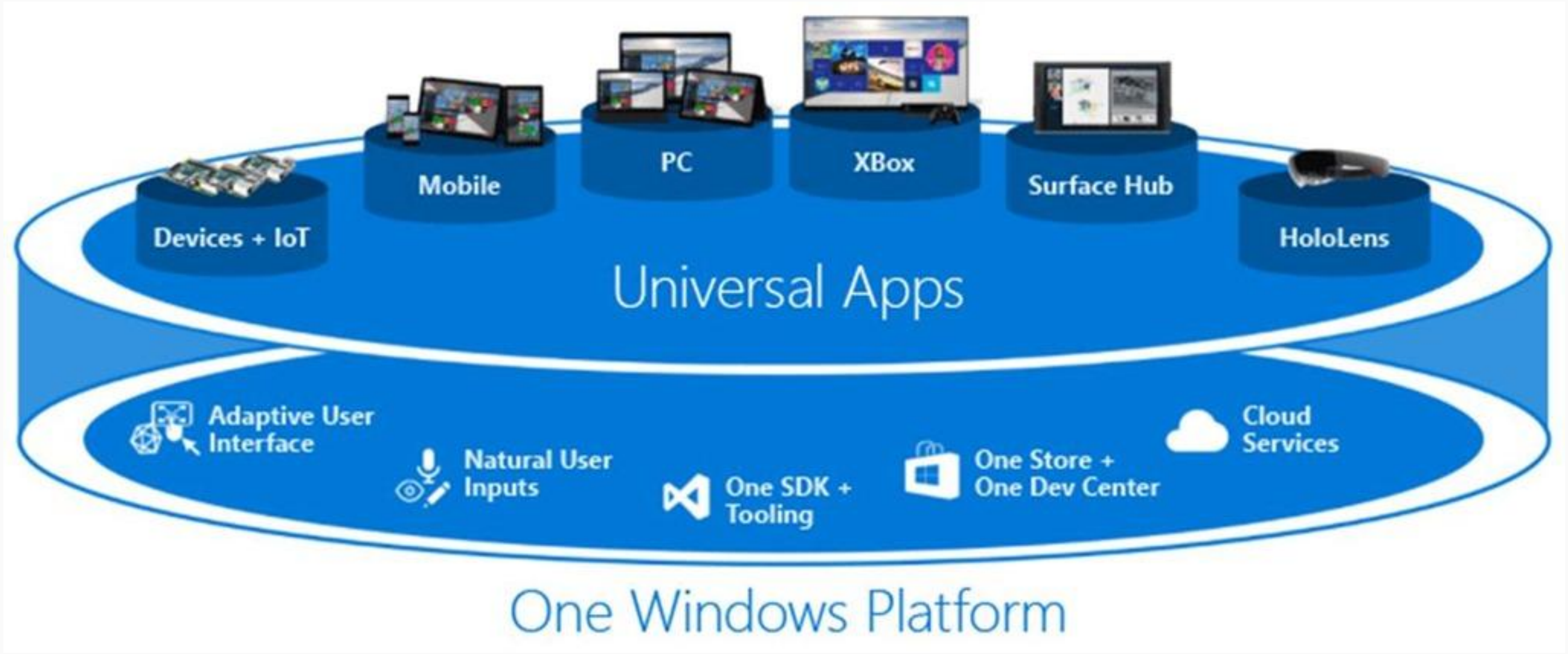
.NET Standard

- .NET Standard is a specification
- Used for developing library projects


.NET Framework
.NET Core
Xamarin
App Models
WPF
Windows Forms
UWP
Xamarin. iOS
Xamarin. Android
ASP.NET
ASP.NET Core*
Xamarin. Forms
Xamarin. Mac
Base Libraries
.NET Framework Class Library
CoreFx Class Library
Mono Class Library
Common Infrastructure
Compilers
Languages
Runtime components



https://devblogs.microsoft.com/cesardelatorre/net-core-1-0-net-framework-xamarin-the-whatand-when-to-use-it/



UWP apps are compiled to native code for their target platforms through .NET Native. For more information, see Guide to Universal Windows Platform Apps (UWP).

Resources

- Differences
- DotNet standard
- New to c# 10
- Asp.net 6 announcement.

.NET Core environment setup

- https://dotnet.microsoft.com/en-us/download
- VsCode or Visual Studio 2022
- .NET Core 6.0
- Location => C:\Program Files\dotnet\sdk
- dotnet --version

.NET Core CLI

- dotnet [verb] [arguments]
- dotnet new
- dotnet restore
- dotnet run
- dotnet build
- dotnet clean
- dotnet --listsdks

Let's check out the codes

Solution Structure

- Connected Services (Azure Storage or Application Insights)
- Dependencies
  - Packages
  - Projects etc.
- Properties
  - launchSettings.json
- Root Folder Files
  - Program.cs, application.settings, Startup.cs

launchSettings.json

- Launch profiles
- Environment variables
- IIS Express, IIS, Projects, Executables

appsettings.json

- The appsettings stores the configuration values in name-value pairs using the JSON format.
- The values of appsettings.json are overwritten by the appsettings.development.json
- IConfiguration config
- var result = _config.GetValue<string>("LogLevel");

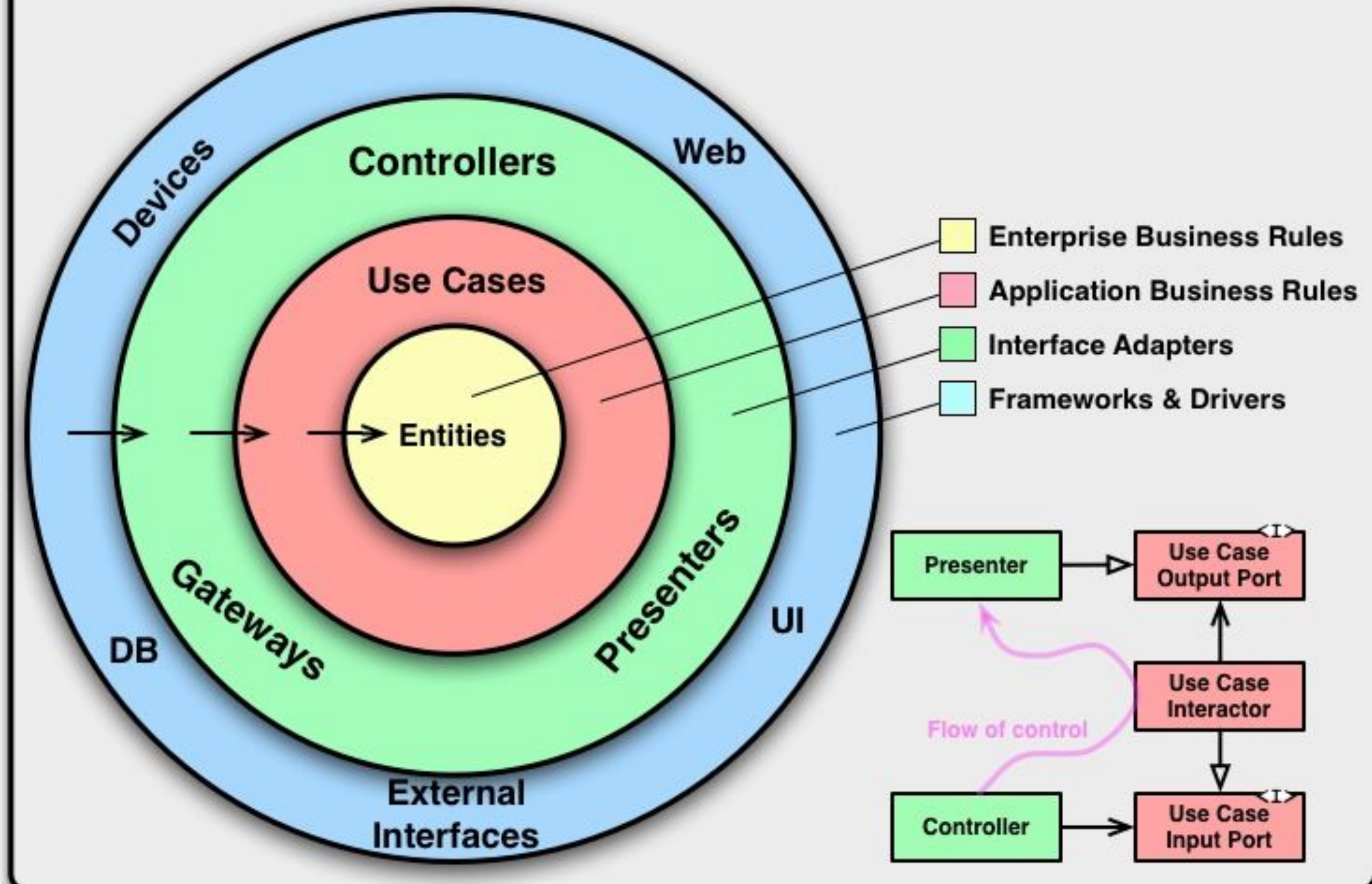
Program.cs

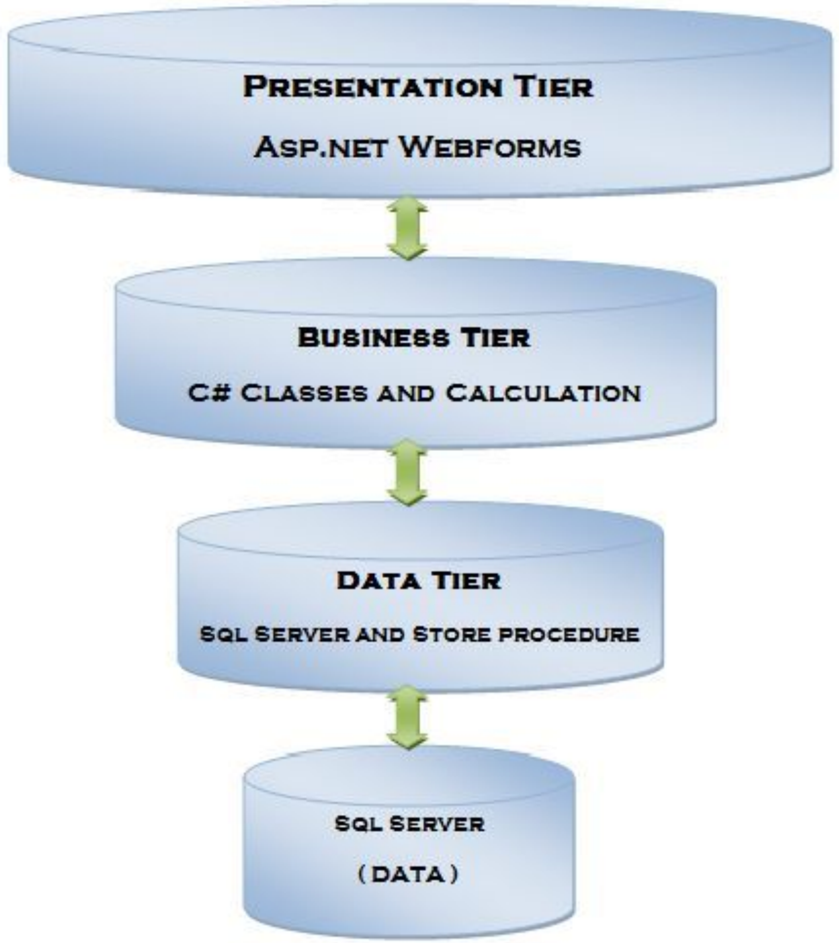
- The Program class contains the method Main, which is the entry point of the ASP.NET Core applications.
- The Main method is similar to the Main method of a console Applications.
- That is because all the .NET Core applications basically are console applications.
- The main purpose of the Program class is to configure the applications infrastructure.

What is Web Host?

- Web host creates a server, which listens for the HTTP requests.
- It configures the request pipeline (or middleware pipeline).
- Also, It sets up DI container, where we add our services.
- Managing the lifetime of the services is also a task of Webhost.

The Clean Architecture

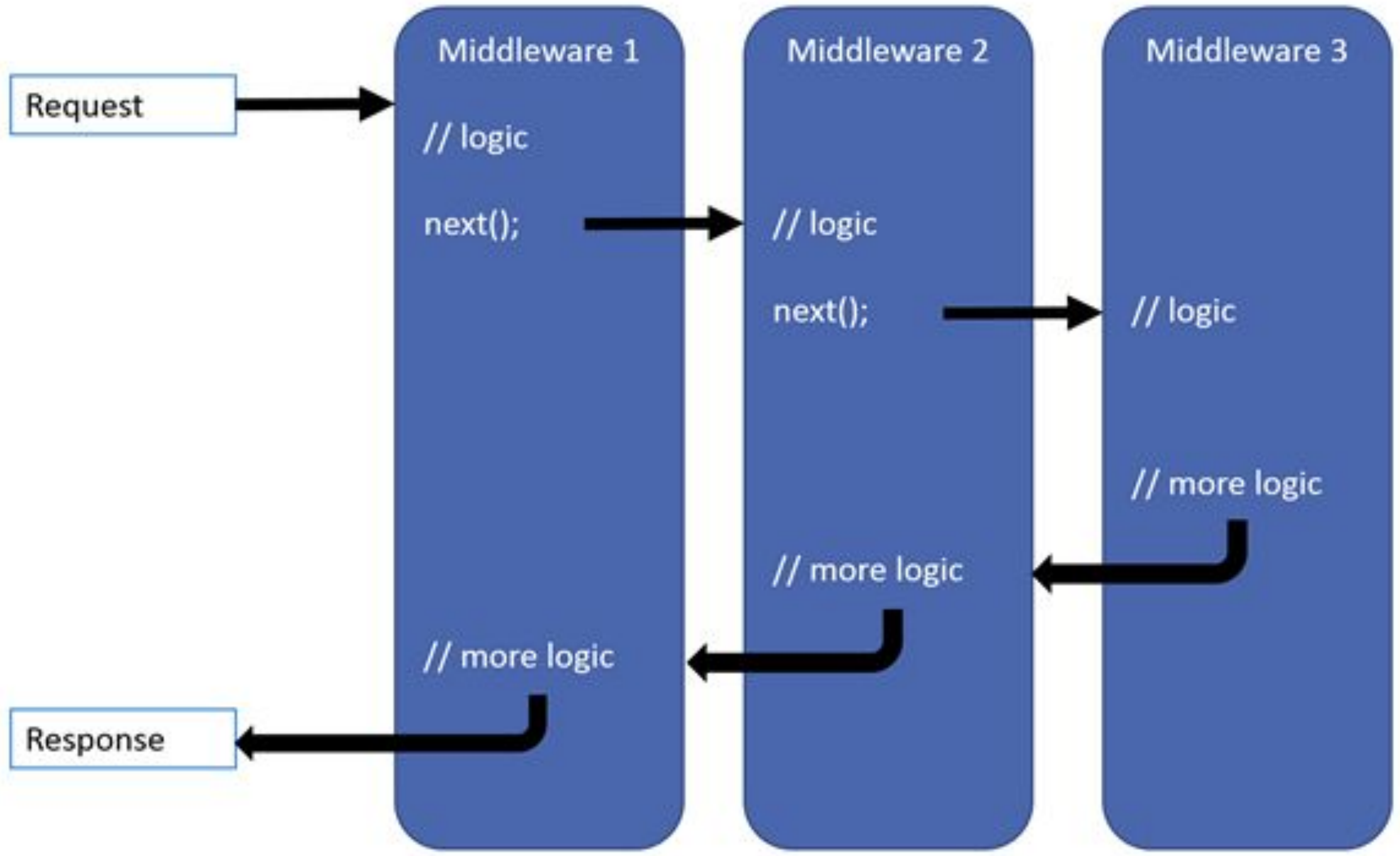




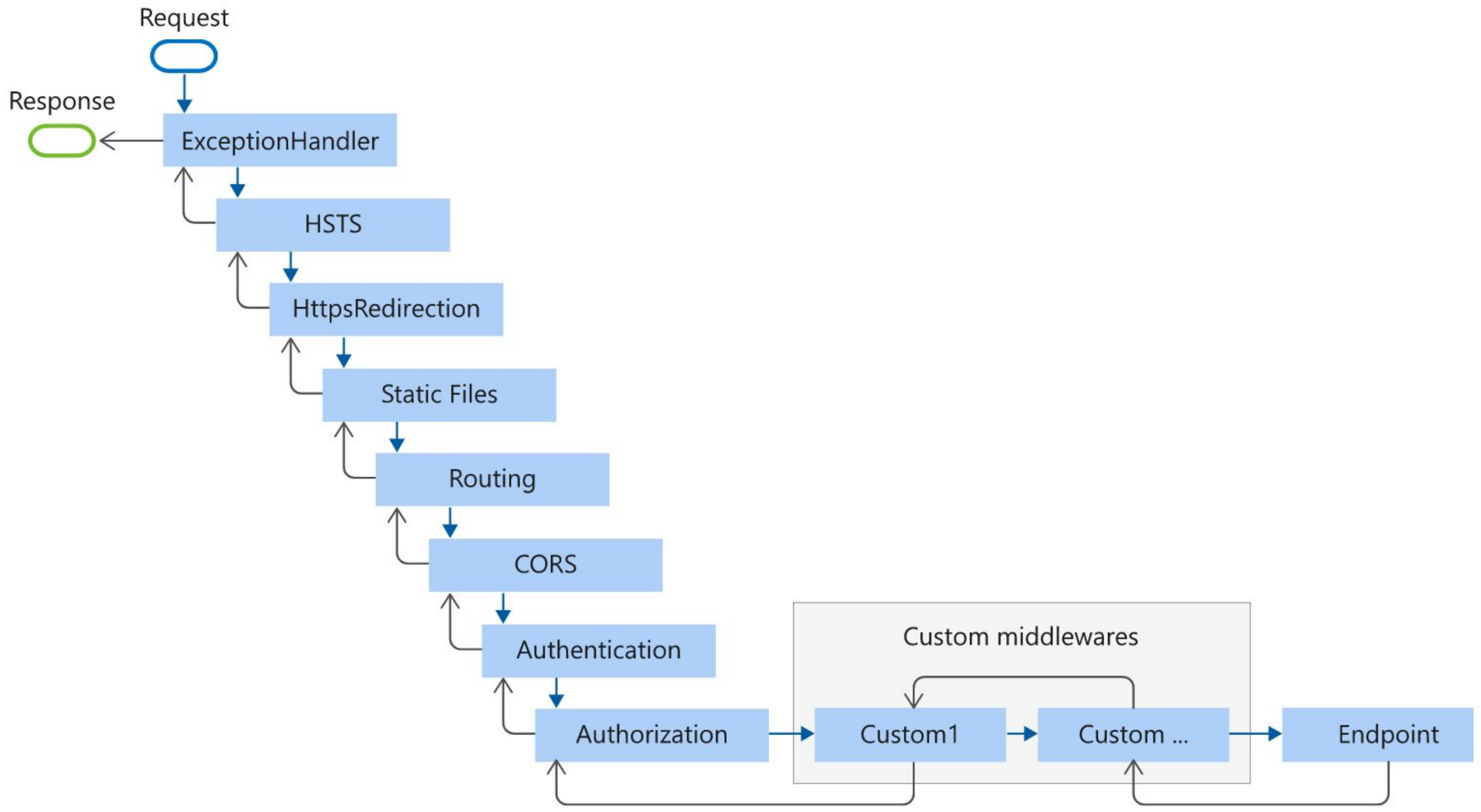
3 layer architecture

Resources

- 1. Build api with dotnet 6
- Rest API guidelines
- Clean architecture Uncle Bob
- 2. Middleware
- ** Advance Resource
- 3. Dependency Injection
- Middleware Doc
- 4. Attribute



Middleware pipeline


Request
Response
ExceptionHandler
HSTS
HttpsRedirection
Static Files
Routing
CORS
Authentication
Authorization
Custom middlewares
Custom1
Custom ...
Endpoint

Middleware

CreateDefaultBuilder

- Sets the content root to Directory.GetCurrentDirectory.
- Loads optional configuration from
- Appsettings.json
- Appsettings.{Environment}.json.
- User secrets when the app runs in the Development environment.
- Environment variables

CreateDefaultBuilder

- Command-line arguments.
- Enable logging
- Sets up the Dependency Injection Container.
- Configures Kestrel as the webserver
- Adds Framework Services to the DI Container
- Integrates the Kestrel run with IIS

CreateDefaultBuilder

Source Code:
https://github.com/dotnet/aspnetcore/blob/1480b998660d2f77d0605376eefab6a83474ce07/src/DefaultBuilder/src/WebHost.cs

Build & Run it

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

Startup.cs

- The startup class is conventionally named as Startup.cs.
- It is located in the project root.
- Its location is configured in the Main method of the program.cs.

Startup.cs

- The Host is created and configured, but before building & running we need to further configure with the application.
- UseStartup<Startup>()
- The startup class contains two methods. One is ConfigureServices (which is optional) and the other one is Configure.

Startup.cs

- 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.

Services Available in Startup

- IApplicationBuilder => We use this service to the application request pipeline.
- IHostingEnvironment => This service provides the current EnvironmentName, ContentRootPath, WebRootPath, and web root file provider.
- IServiceCollection => This is a DI container. We add services into this container.

Availability of Services

- IApplicationBuilder => Configure
- IHostingEnvironment => Startup Constructor
- IServiceCollection => ConfigureServices

Startup.cs - ConfigureServices

- IServiceCollection is a **DI container**.
- We add services into this container.
- The following code is an example of how we use AddMvc() extension method to add MVC related services to the IServiceCollection collection
- services.AddMvc();

Startup.cs - ConfigureServices

- Adding services to the **Dependency Injection container** will make them available for **dependency injection**.
- That means we can inject those services anywhere in our application.
- Dependency injection added by default in .net core, unlike .net framework with ninject etc
- The ASP.NET Core uses the dependency injection extensively.

Startup.cs - Startup.cs - ConfigureServices

```
public void Configure(IApplicationBuilder app, IHostingEnvironment
env) {
      if (env.IsDevelopment()) {
       app.UseDeveloperExceptionPage();
     }
}
```

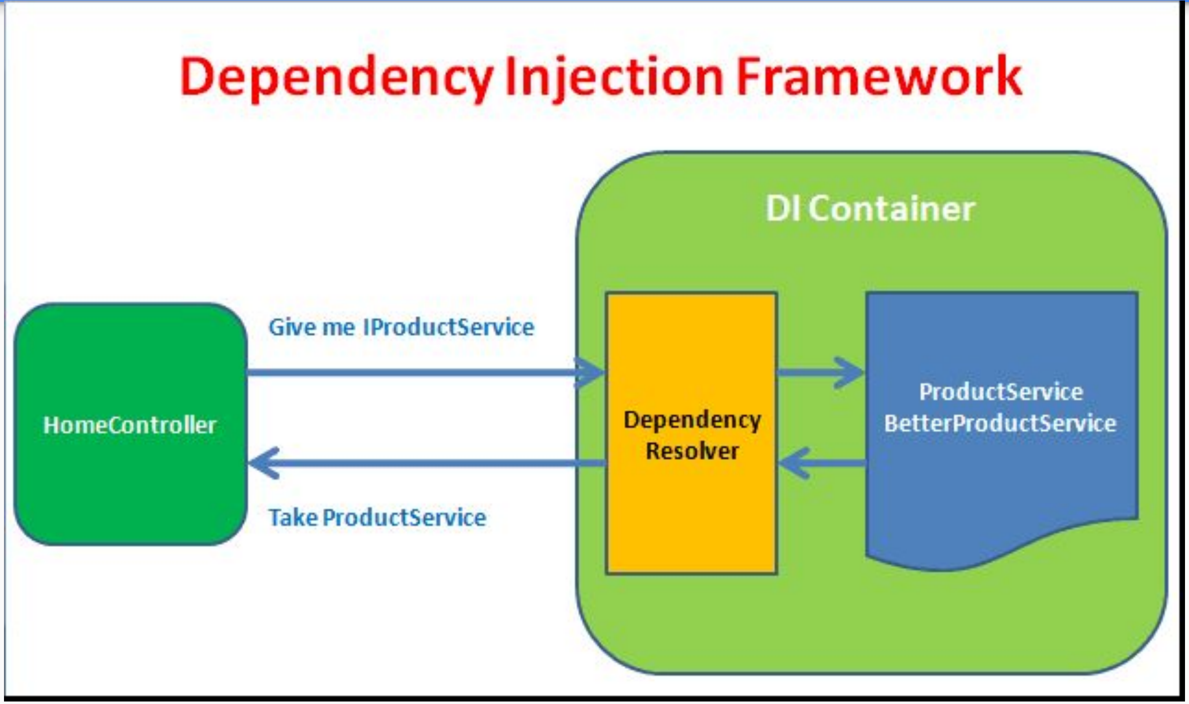
Startup.cs - Startup.cs - ConfigureServices

```
app.UseStaticFiles();

app.UseCors("AllowAll");

app.UseAuthentication();
```

Dependency injection (DI)

- Dependency injection (also known as DI) is a design pattern in which an object does not create it dependent classes, but asks for it.
- Dependency injection has now become the first class citizen in ASP.NET Core. It helps us to build loosely coupled, easily readable and maintainable code.

Dependency injection (DI)



```
services.AddTransient<IProductService, BetterProductService>();

services.AddTransient<IProductService, ProductService>();
```

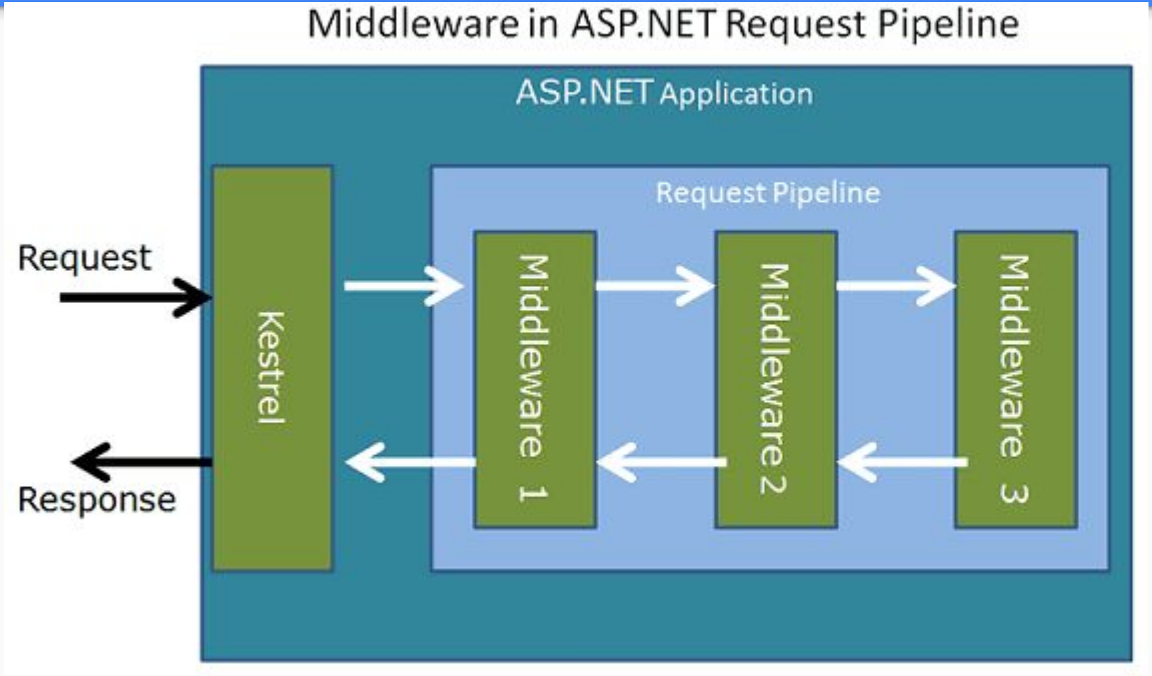
Dependency injection (DI)

- 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.

What is Middleware

- Middleware is a software component that hooks into the request pipeline to handle web requests and generate responses.
- Each middleware Process and manipulates the request as it is received from the previous middleware.
- It may decide to call the next middleware in the pipeline or send the response back to the previous middleware (terminating the pipeline)

How Middleware works



How Middleware works

- First, the Http Request arrives (directly or via External web server) at the Application
- The Kestrel Web server picks up the Request and creates the httpContext and passes it to the First Middleware in the request pipeline.
- The First Middleware then takes over, process the request and passes it to the next Middleware. This goes on until it reaches the last middleware

How Middleware works

- The last middleware returns the request back to the previous middleware, effectively terminating the request pipeline.
- Each middleware in the sequence gets a second chance to inspect the request and modify the response on its way back.
- Finally, the response reaches kestrel, which returns the response back to the client.Any of the middleware in the request pipeline can terminate the request pipeline by simply not passing the request to the next middleware.

Configuring the Request Pipeline

- The Use and Run method extensions allow us to register the Inline Middlewares to the Request pipeline.
- The Run method adds the terminating middleware.
- The Use method adds the middleware, which may call the next middleware in the pipeline.
- Middleware is executed in the same order in which they are added in the pipeline.

Let's check out the codes

Let's make a custom middleware

Custom Middleware

- The middleware class is not required to implement any interface or inherit from any class. However, there are two specific rules that you must follow.
- **Rule 1:** The middleware class must declare a non-static public constructor with at least one parameter of type RequestDelegate.What actually you get here is the reference to the next middleware in the pipeline.

Custom Middleware

- When you invoke this RequestDelegate you are actually invoking the next middleware in the pipeline
- **Rule 2:** The middleware class must define a public method named Invoke that takes an HttpContext and returns a Task.This is the method that gets invoked when the request arrives at the middleware.

Kestrel: Web Server for ASP.NET Core

- ASP.NET Core has gone through some drastic change compared to the previous version of ASP.NET.
- 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.

Kestrel: Web Server for ASP.NET Core

- 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

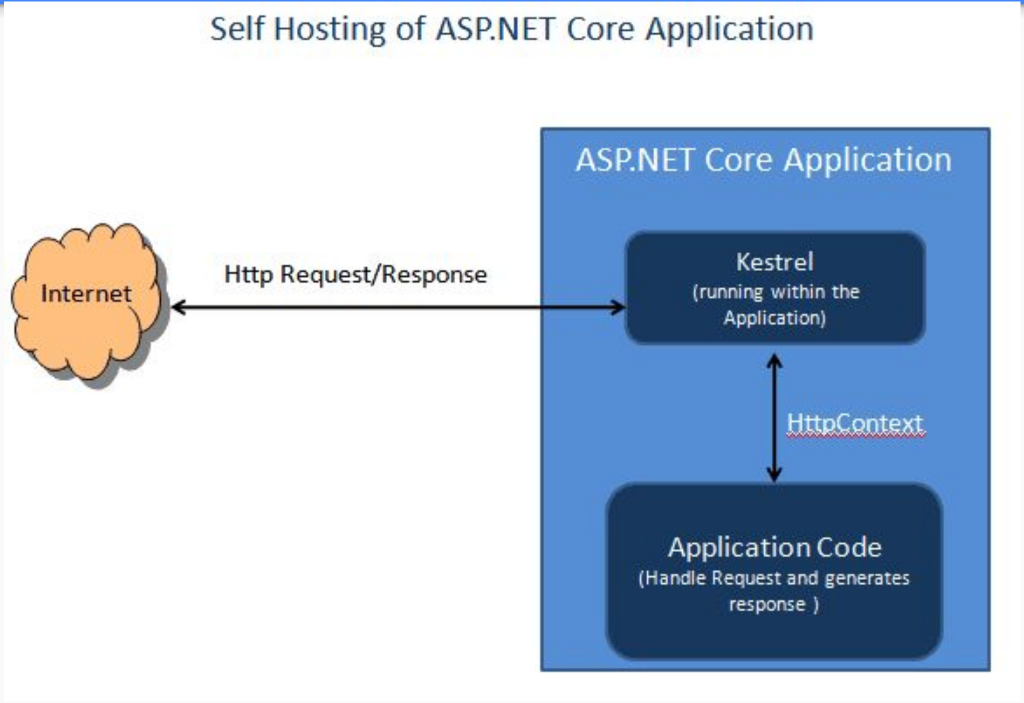
Why Kestrel

- 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.

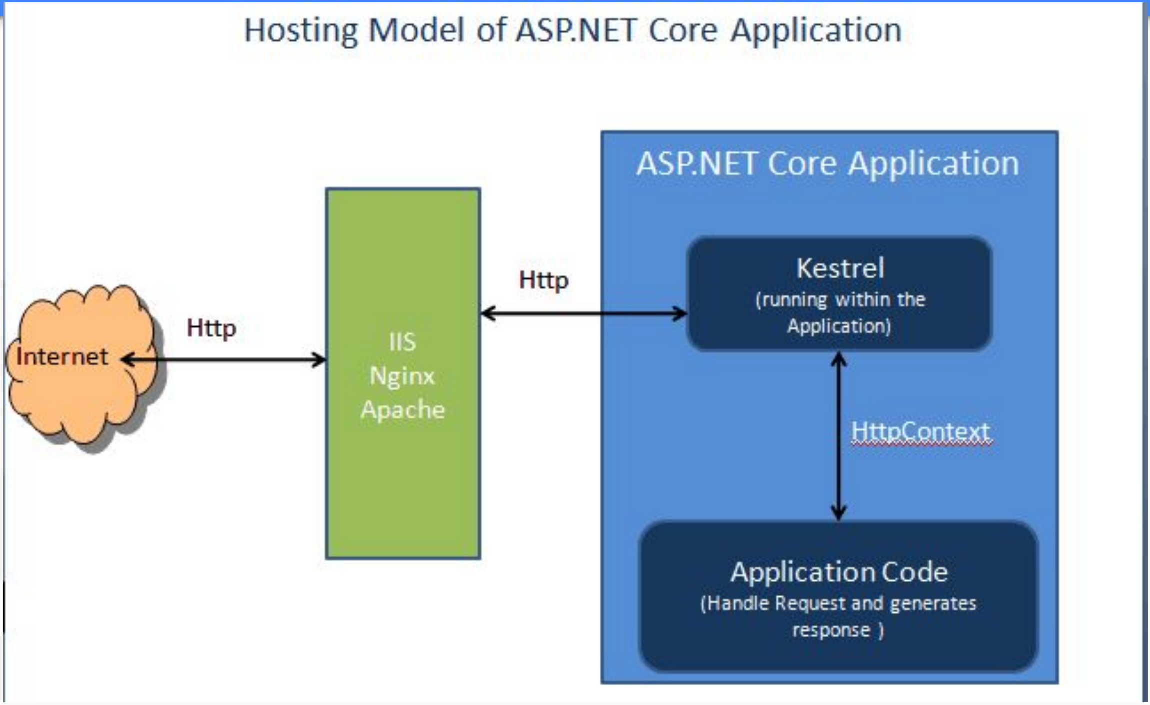
Using Kestrel

- The Main method of Program.cs invokes CreateDefaultBuilder, which is responsible to create the web application host.
- The CreateDefaultBuilder is a helper method (click here for the source code)
- It calls the UseKestrel method to registers the Kestrel as the server that will be used to host our application

Self Hosting


Self Hosting of ASP.NET Core Application
Internet
Http Request/Response
ASP.NET Core Application
Kestrel
(running within the Application)
HttpContext
Application Code
(Handle Request and generates response)

Behind another Webserver


Hosting Model of ASP.NET Core Application
Internet
Http
IIS
Nginx
Apache
Http
ASP.NET Core Application
Kestrel
(running within the Application)
HttpContext
Application Code
(Handle Request and generates response)

Behind another Webserver : How it works

- 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.
- Web server can perform some useful processing like logging, request filtering before passing the request to Kestrel.

Behind another Webserver : Benefits

- Security
- It can limit your exposed surface area. It provides an optional additional layer of configuration and defense.
- Simplifies Load Balancing
- SSL Setup
- Sharing Single IP with Multiple addresses
- Request FIltering, logging & URL Rewrites, etc.
- It can make sure that the application restarts if it is crashes

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

Thank you. :)