.Net Core

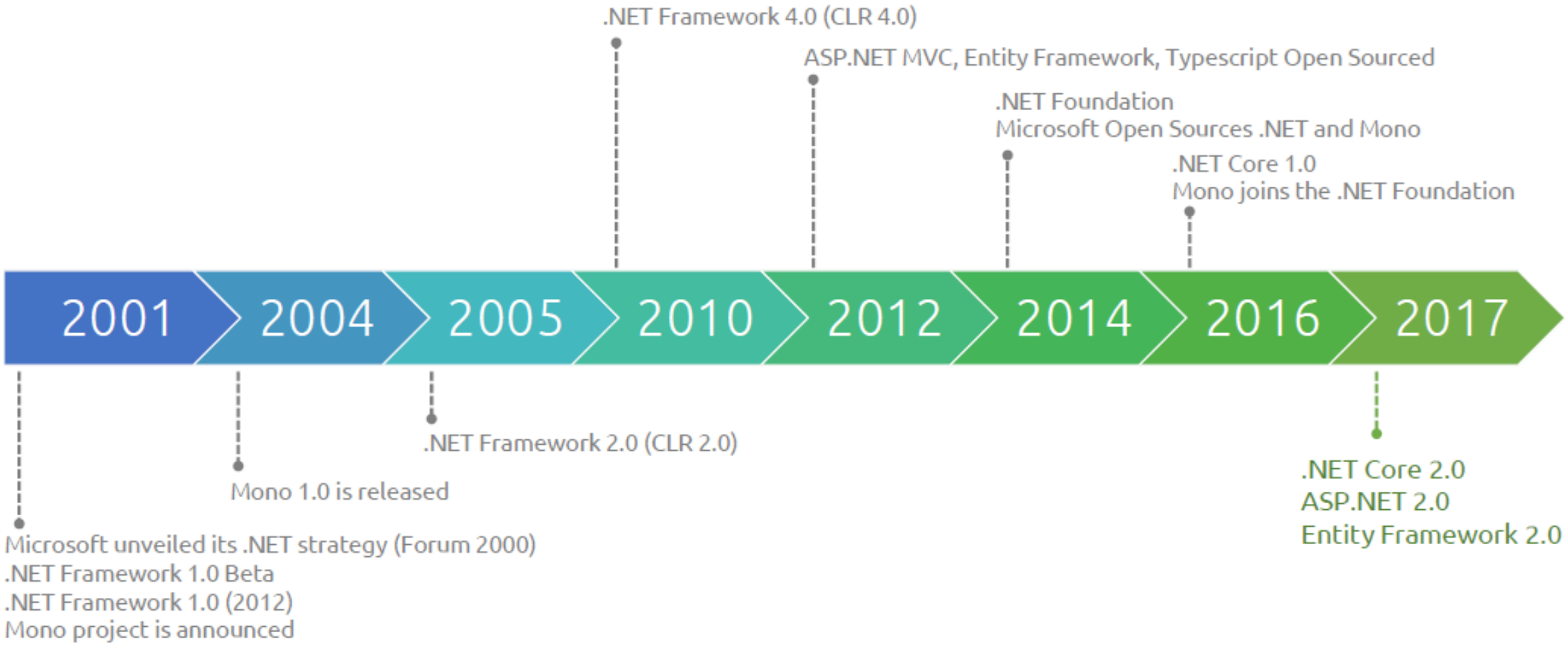
Presented by
Venu
Technology Evangelist
18 + Years of IT Experience

Togaf Certified Enterprise Architect
Microsoft Certified Azure Solution Architect
CTO @ TritechSamvit Pvt Ltd
Passion – Training ,Motoring

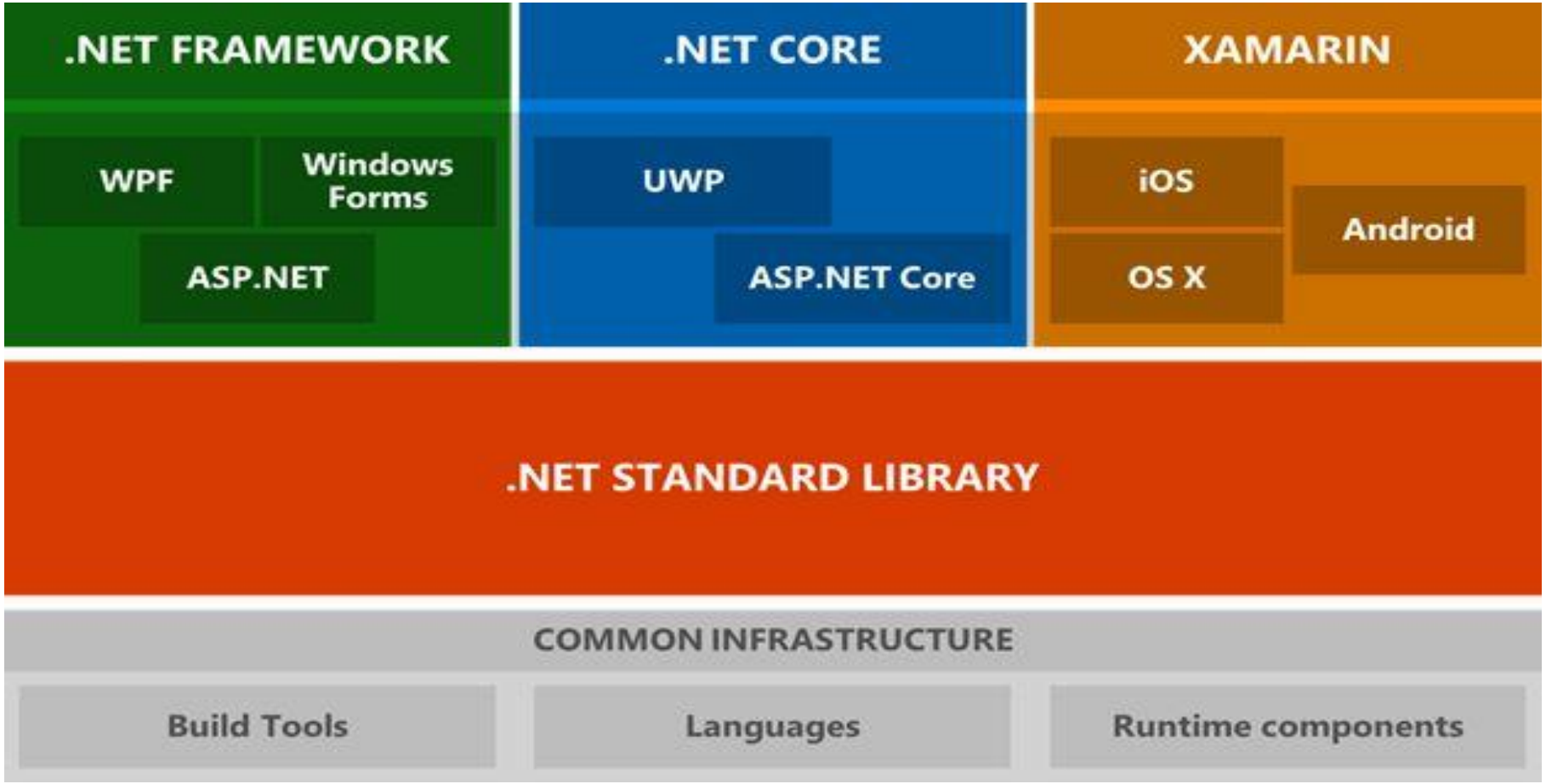
ASP.NET Core – What is it?

A open-source and cross-platform framework for building modern cloud-based Web applications using .NET

History of a Framework



.NET Ecosystem



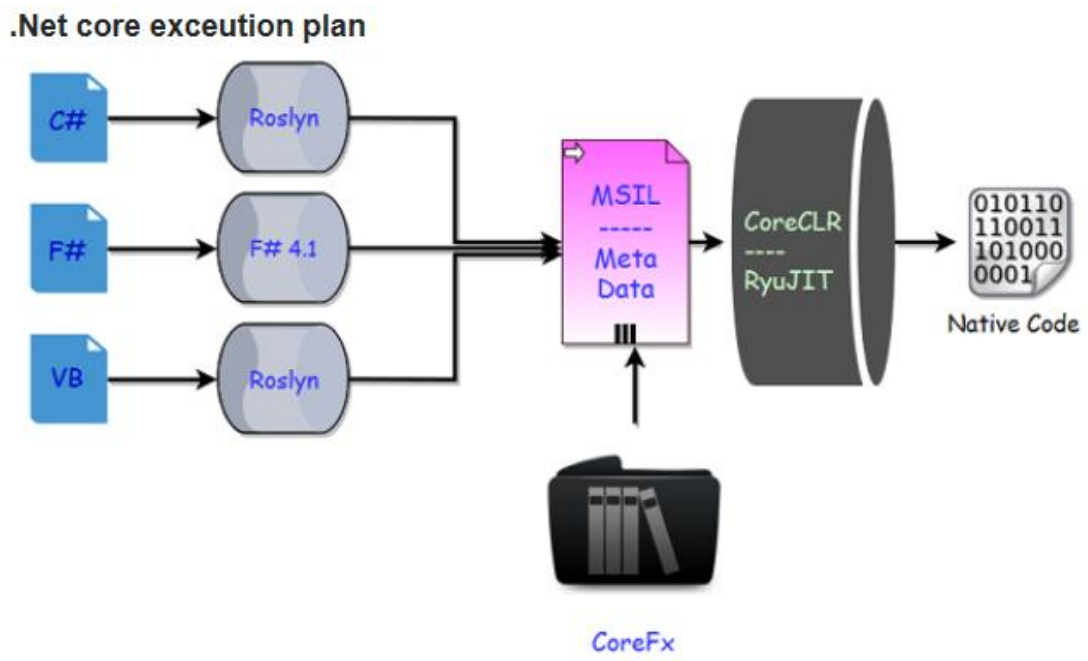
Introduction to .Net Core

Cross-platform
Open source
Microservices architecture
Containers
Modern Architecture
Modular Design
Various development tools

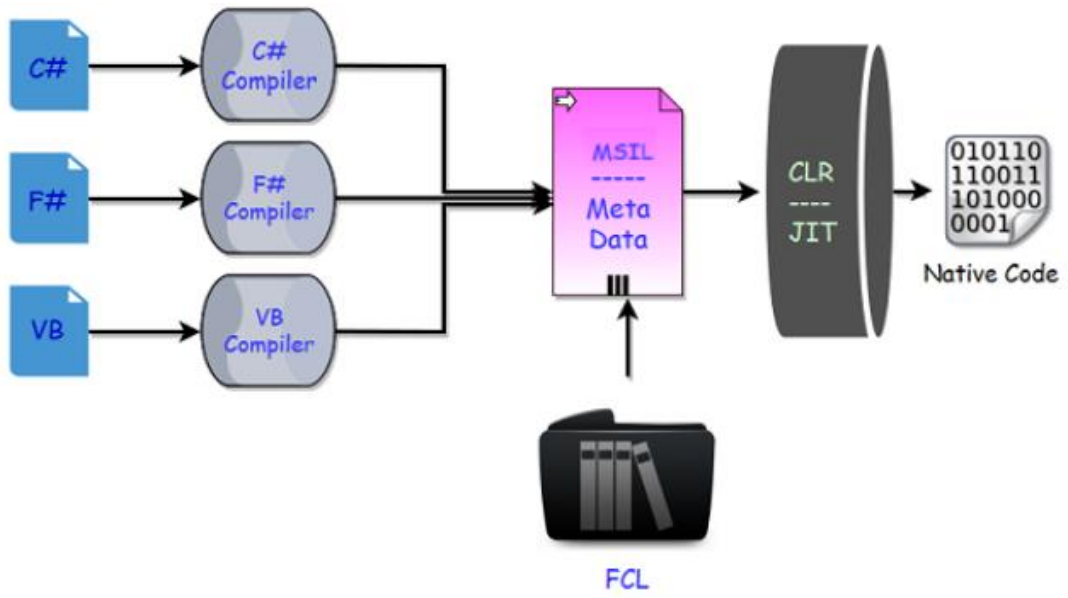
A need for high-performance and scalable systems
A need for side by side of .NET versions per application level

.Net Core / .Net Fx Excecution

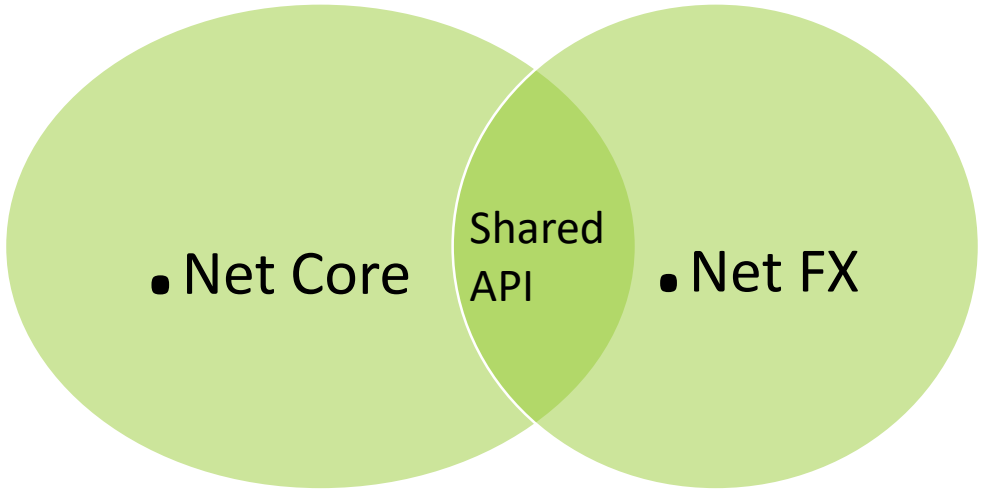
.Net Core



.Net Fx



.Net Core v/s .Net Fx

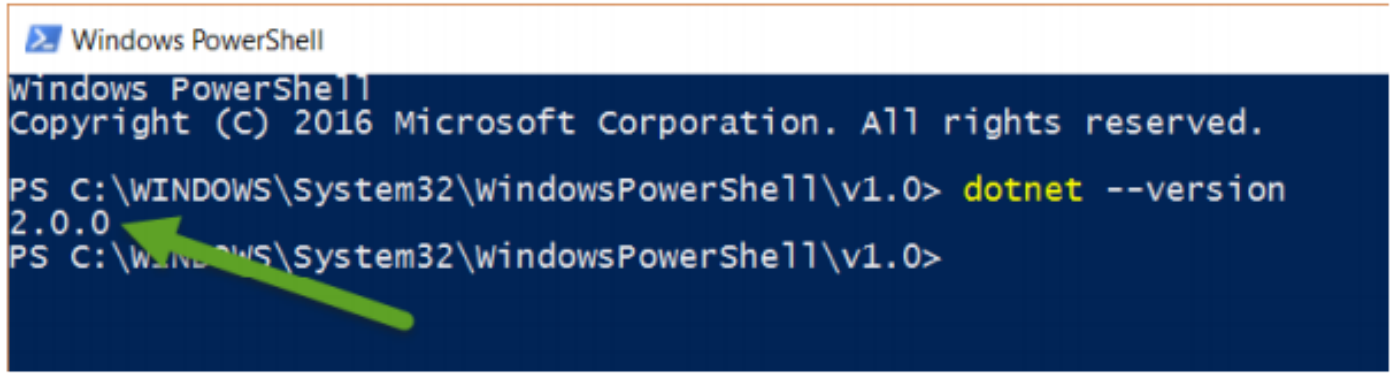


https://stackify.com/net-core-vs-net-framework/

.Net Core	.Net Fx
Modular and Open Source	A Whole Framework and Proprietary
Cross Platform	Windows Only
Targeting Micro Services	Targeting SOA
Container Based Delivery(Docker)	You run app in old fashion (thick deployment)
High-performance and Scalable systems	Speed is not a concern (RAD and Windows OS Portability)
New Environment, Tools	Already have a pre-configured environment and systems
Cloud Ready Configuration	N.A

Getting the Bits

- Install SDK
  - https://www.microsoft.com/net/download/core
- Optional: Install Visual Studio 2017
  - Install VS2017 .NET Core 2.0 Tools
- Confirm Version:



New Web Projects with dotnet

- dotnet new web
  - Empty Hello World Project
- dotnet new mvc
  - MVC Template
  - `--auth Individual` adds identity support
  - `--use-local-db true` uses localdb instead of SQLite
- dotnet new webapi
- dotnet new razorpages
- dotnet new angular | react | reactredux

- Use `--help` to view options in CLI

Introduction

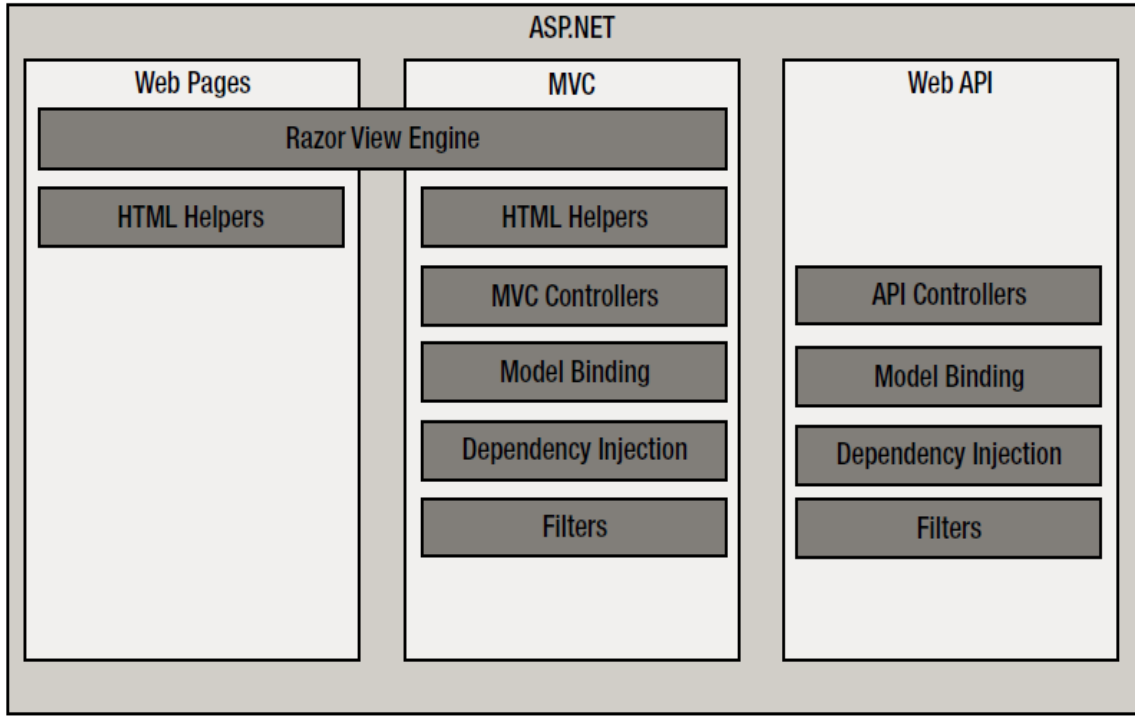
- Why ASP.NET Core?
- Using the CLI
- Startup
- Managing Dependencies
- Managing Middleware
- ASP.NET Core MVC and Web APIs
- Tag Helpers
- Razor Pages
- Testing ASP.NET Core
- What’s New in 2.0

Choose between ASP.NET and ASP.NET Core

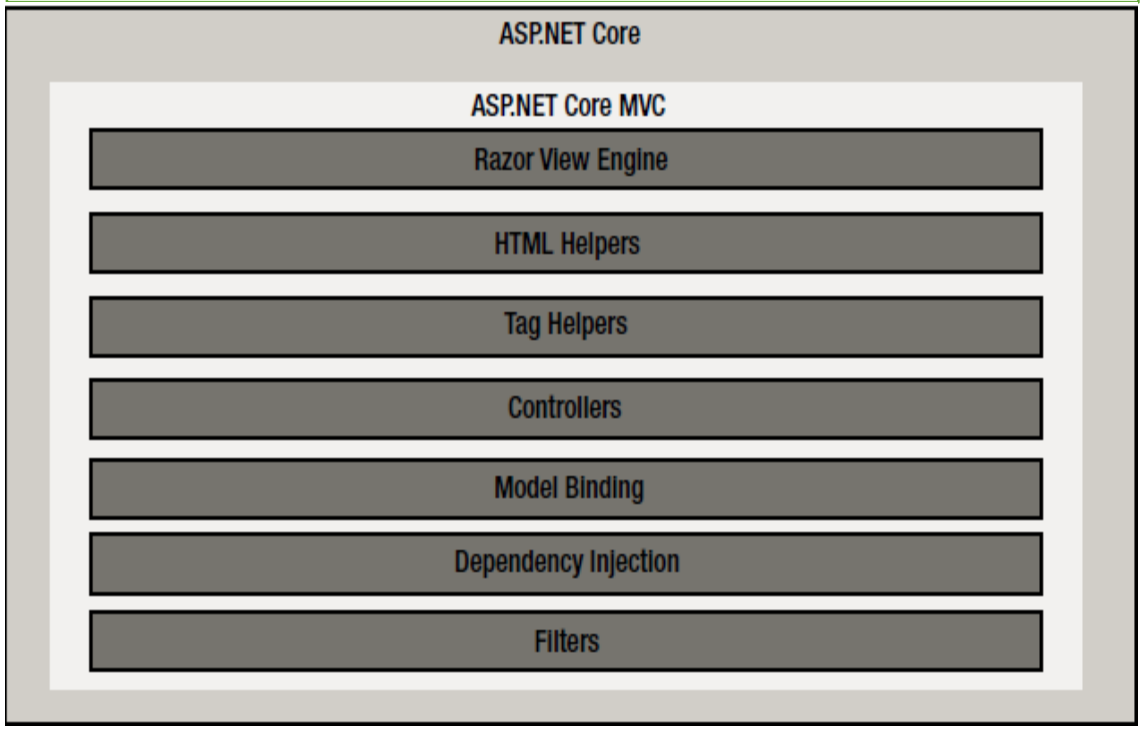
ASP.Net Core	ASP.Net
ASP.NET Core is an open-source, cross-platform framework for building modern, cloud-based web apps on Windows, macOS, or Linux.	ASP.NET is a mature framework that provides all the services needed to build enterprise-grade, server-based web apps on Windows
Build for Windows, macOS, or Linux	Build for Windows(IIS Coupled)
Multiple versions per machine	One version per machine
Higher performance than ASP.NET	Good performance
Choose .NET Framework or .NET Core runtime	Use .NET Framework runtime
Architected to provide an optimized development framework for Cloud or run On-Premise apps.	Architected to Build Server Centric WebSites with the rich set of server side controls (RAD)

Building Block Differences

Asp.Net MVC

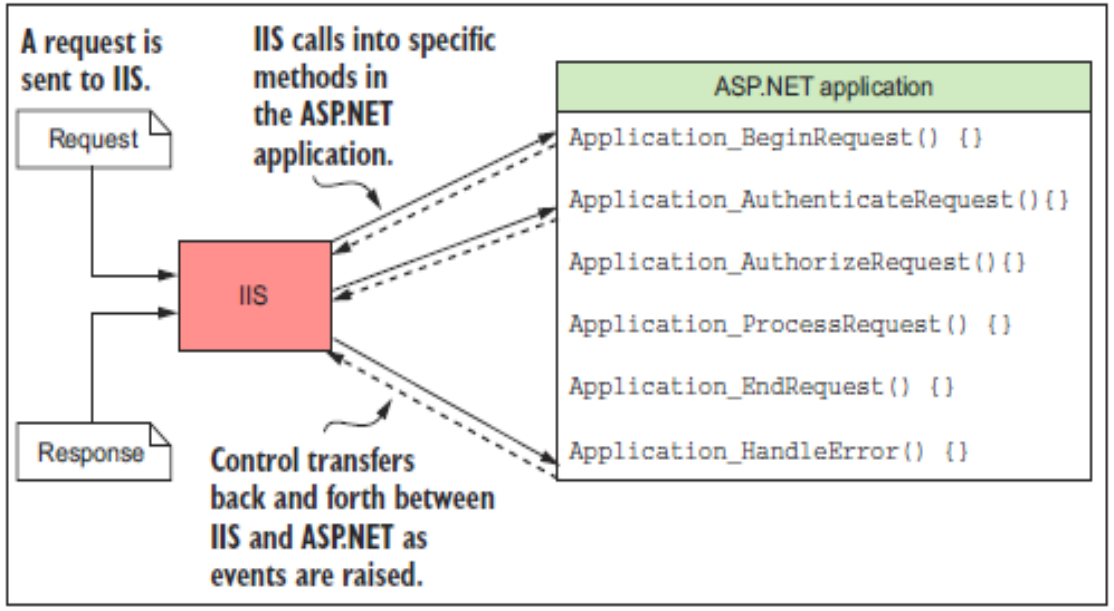


ASP.Net Core

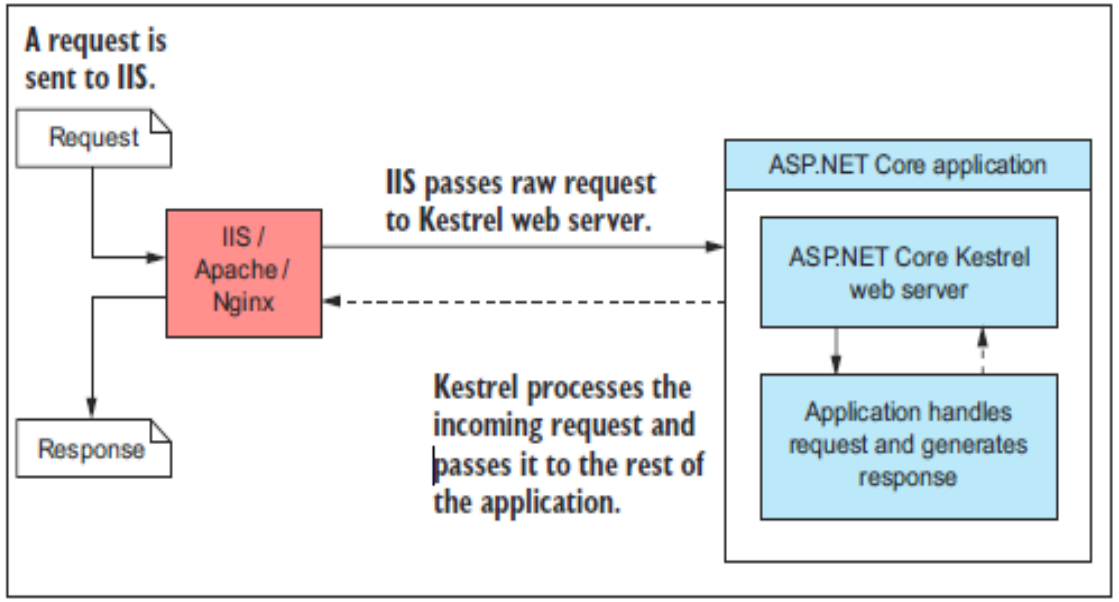


Hosting Differences

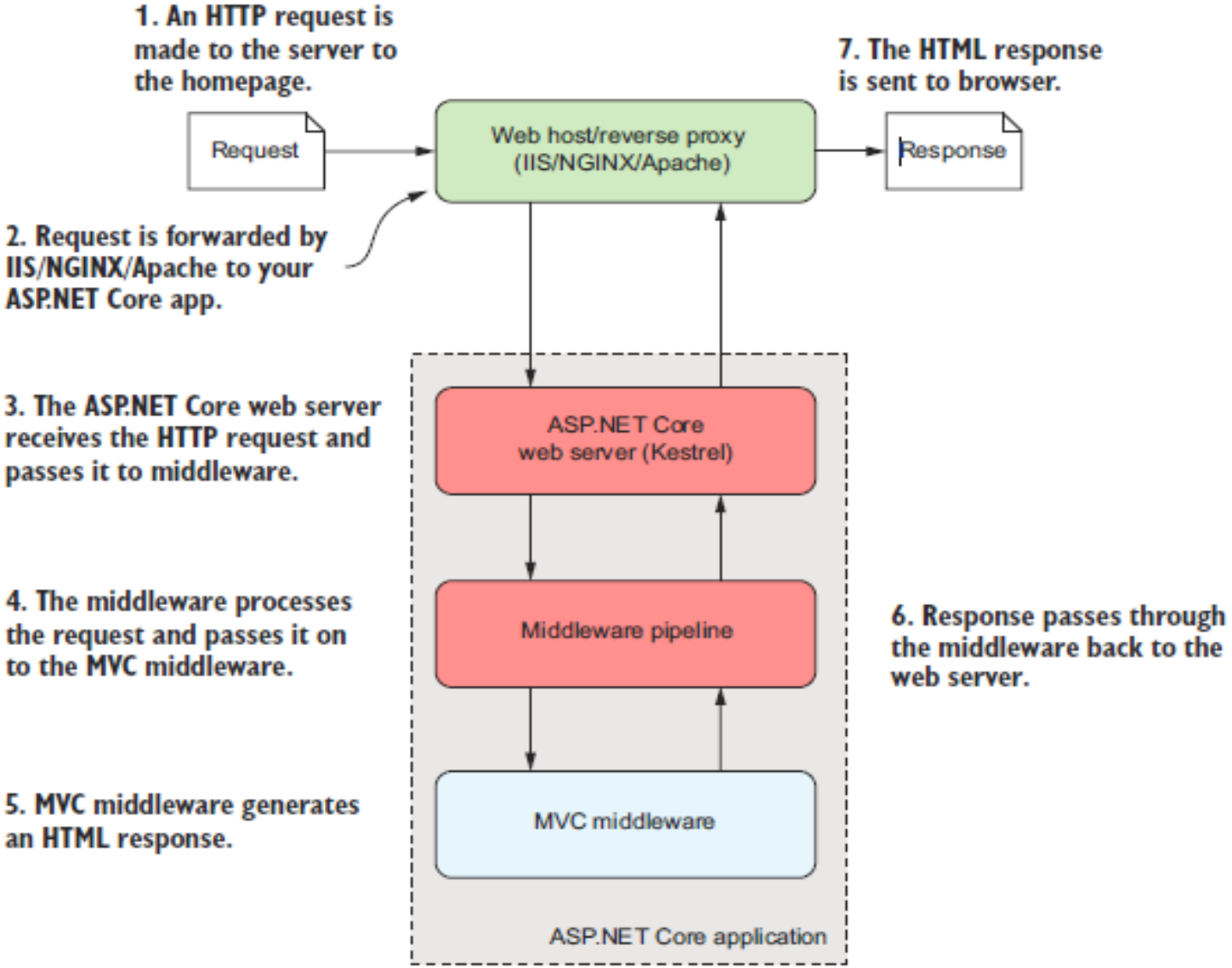
ASP.net



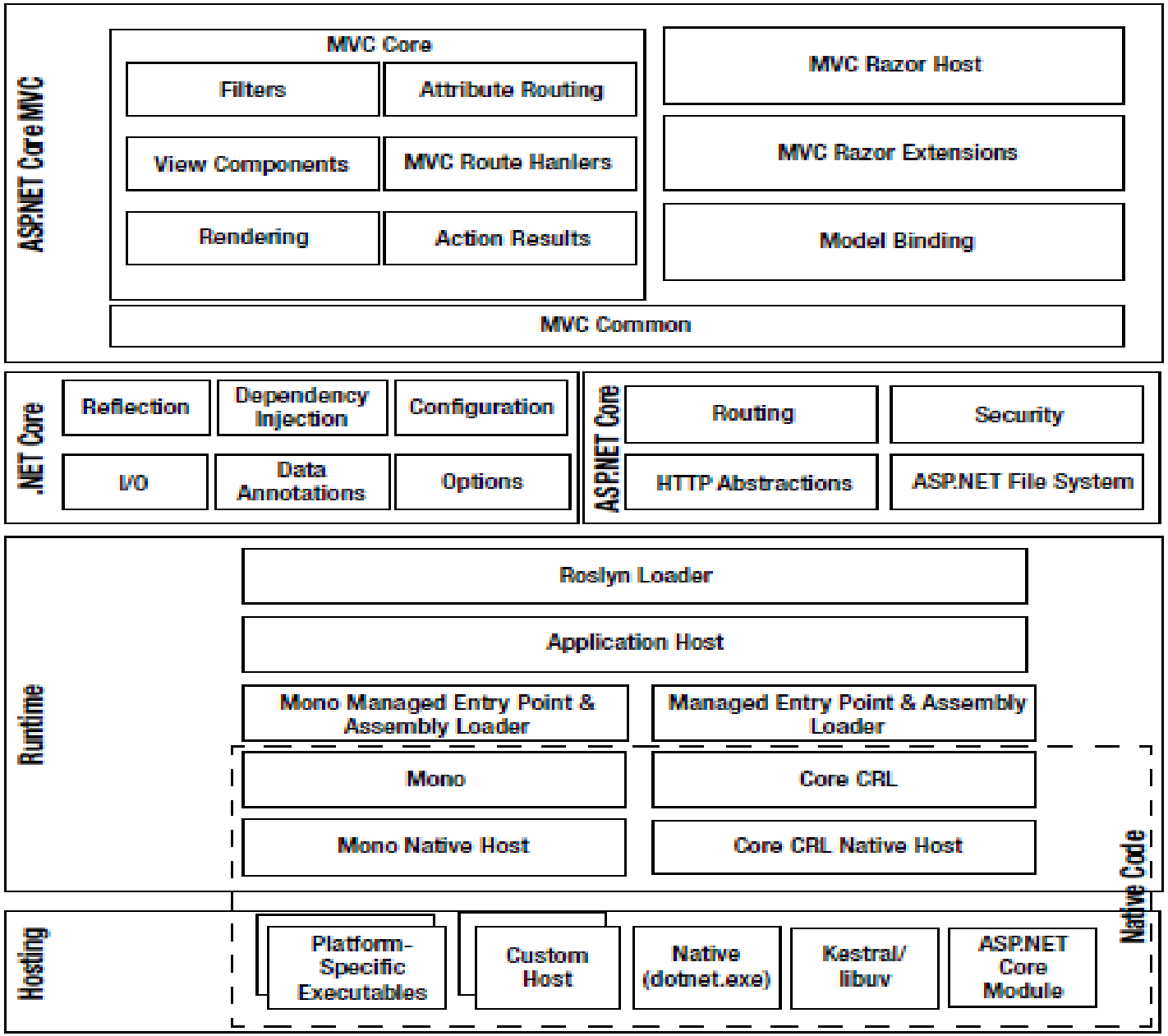
ASP.net Core



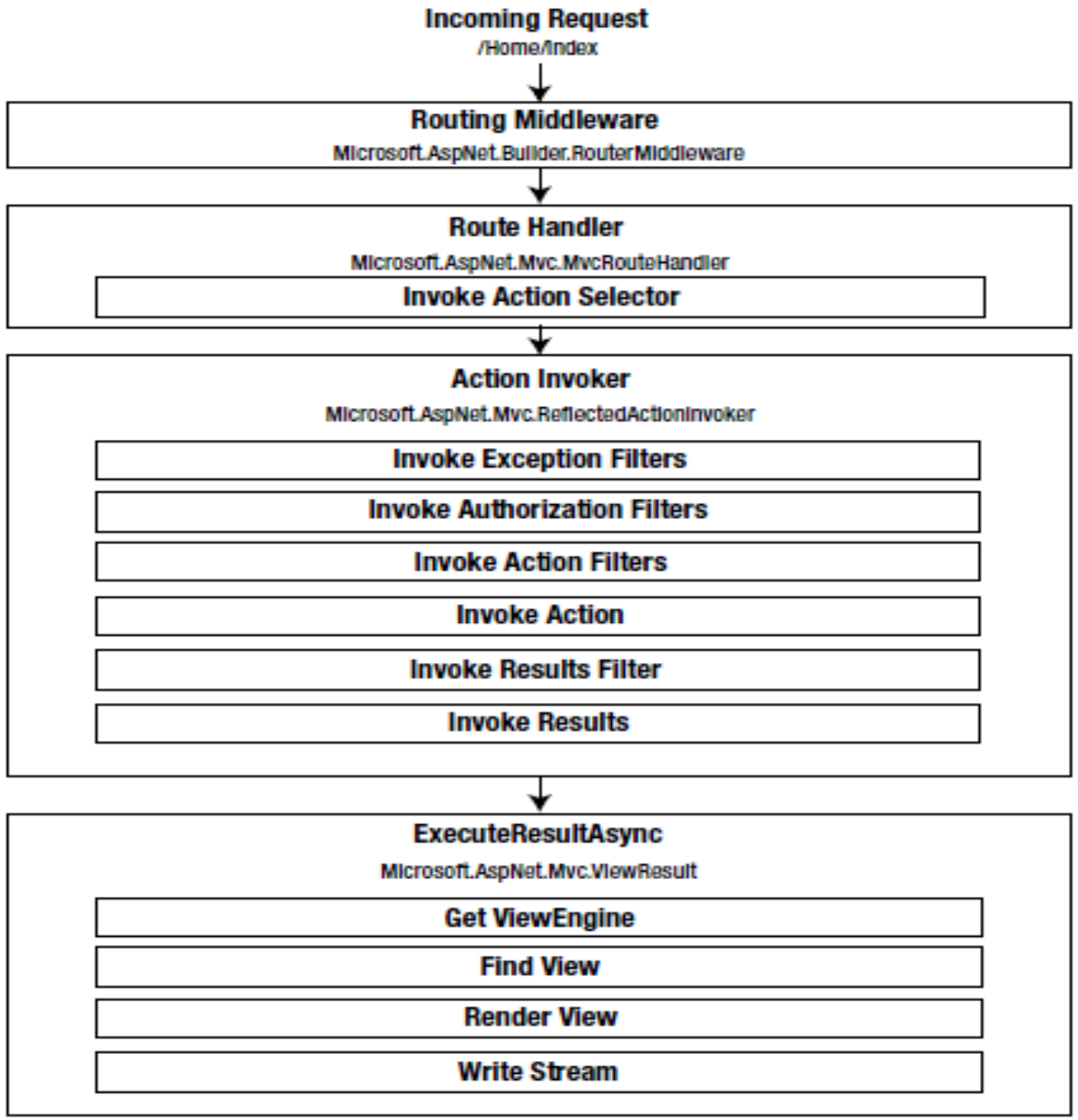
An overview of an ASP.NET Core Application



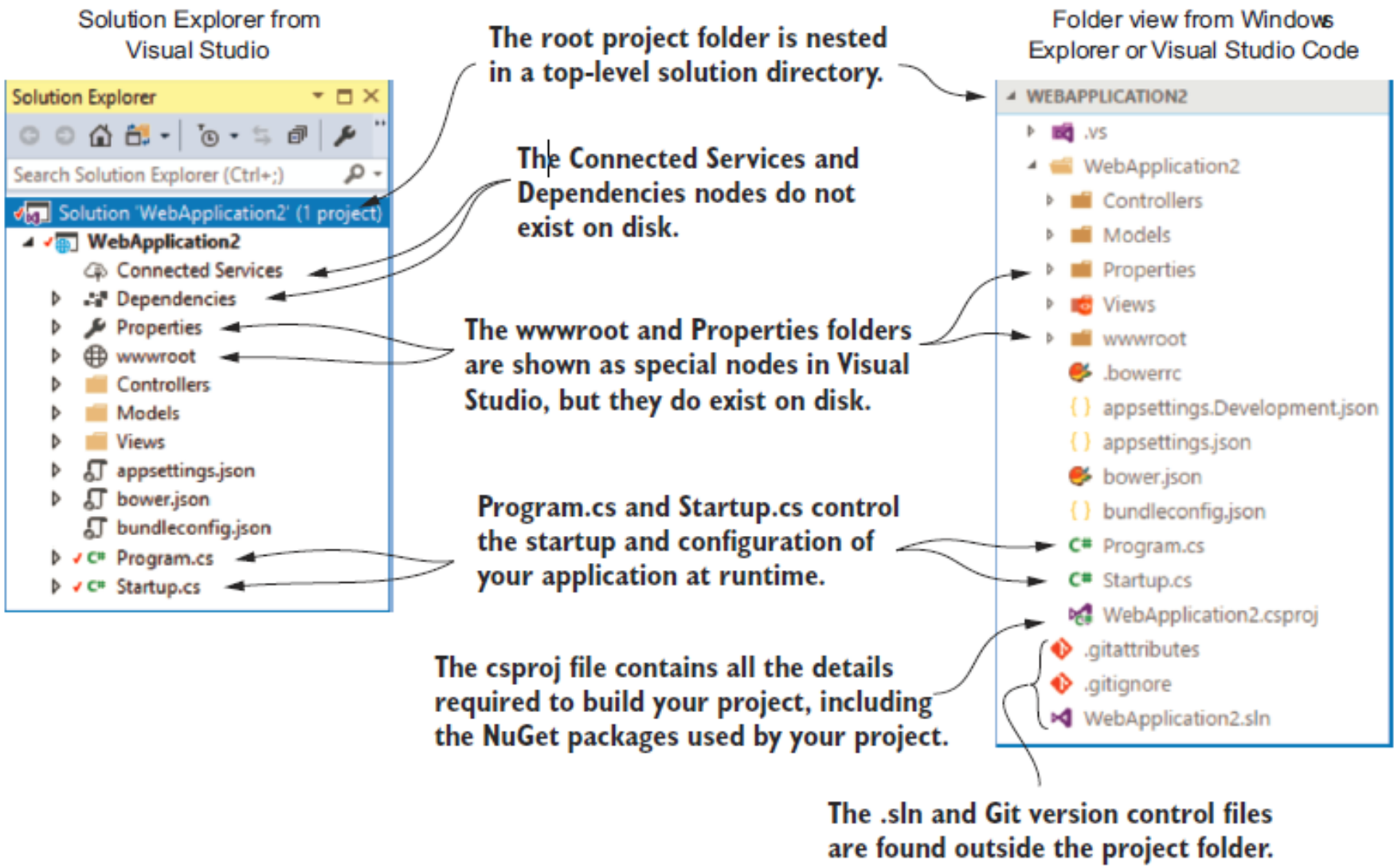
ASP.net Core Architecture



ASP.NET Core MVC framework request processing pipeline



Project Structure



The csproj project file, showing SDK, target framework, and references

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All"
      Version="2.0.0" />
  </ItemGroup>

  <ItemGroup>
    <DotNetCliToolReference Version="2.0.0"
      Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools" />
  </ItemGroup>

</Project>
```

The SDK attribute specifies the type of project you're building.

The TargetFramework is the framework you'll run on, in this case, .NET Core 2.0.

You reference NuGet packages with the PackageReference element.

Additional tools used by Visual Studio to generate controllers and views at design time

The default Program.cs configures and runs an IWebHost

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

    public static IWebHost BuildWebHost(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>()
            .Build();
}
```

Create an IWebHost using the BuildWebHost method.

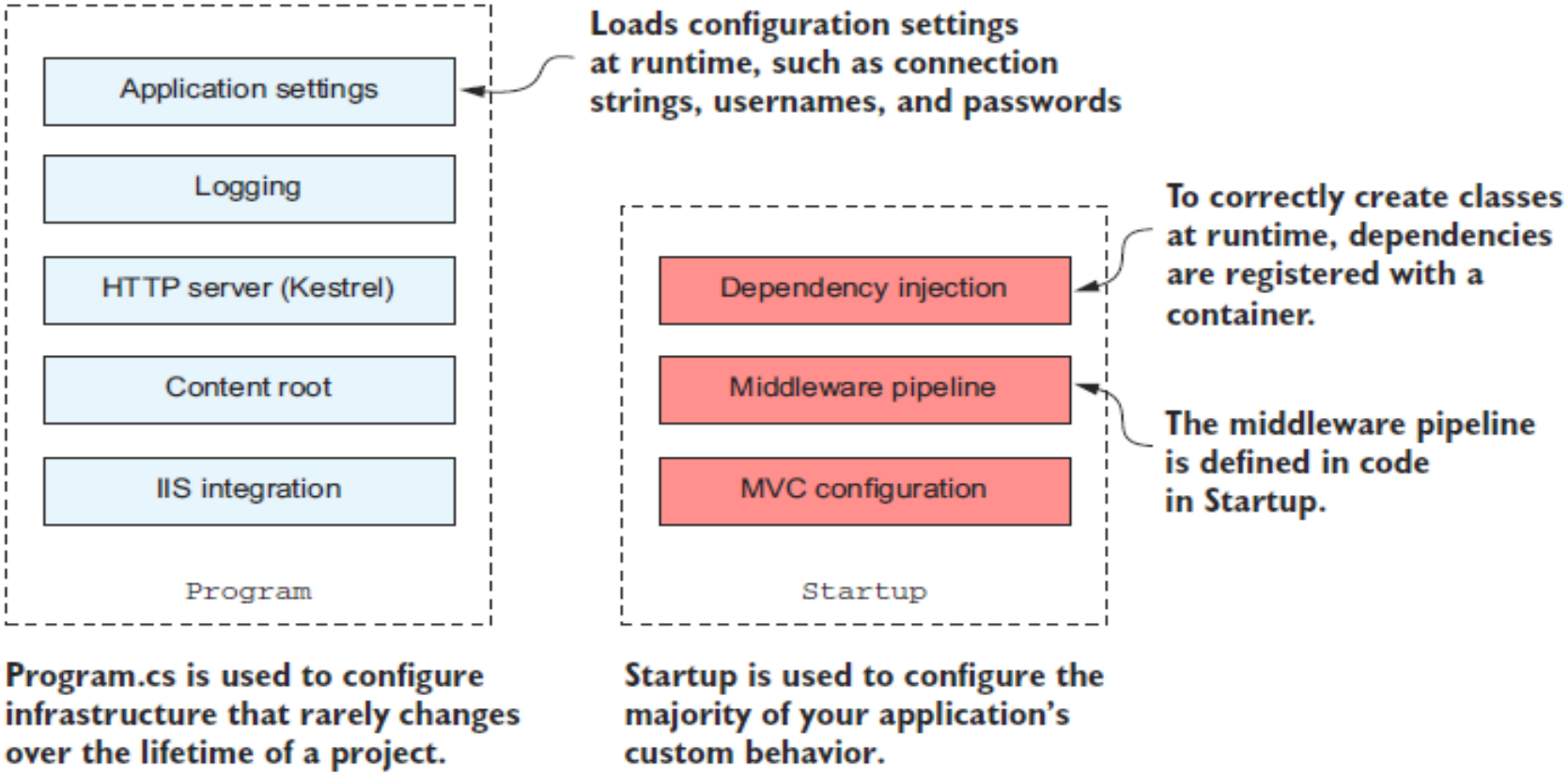
Run the IWebHost, start listening for requests and generating responses.

Create a WebHostBuilder using the default configuration.

The Startup class defines most of your application's configuration.

Build and return an instance of IWebHost from the WebHostBuilder.

The difference in configuration scope for Program and Startup



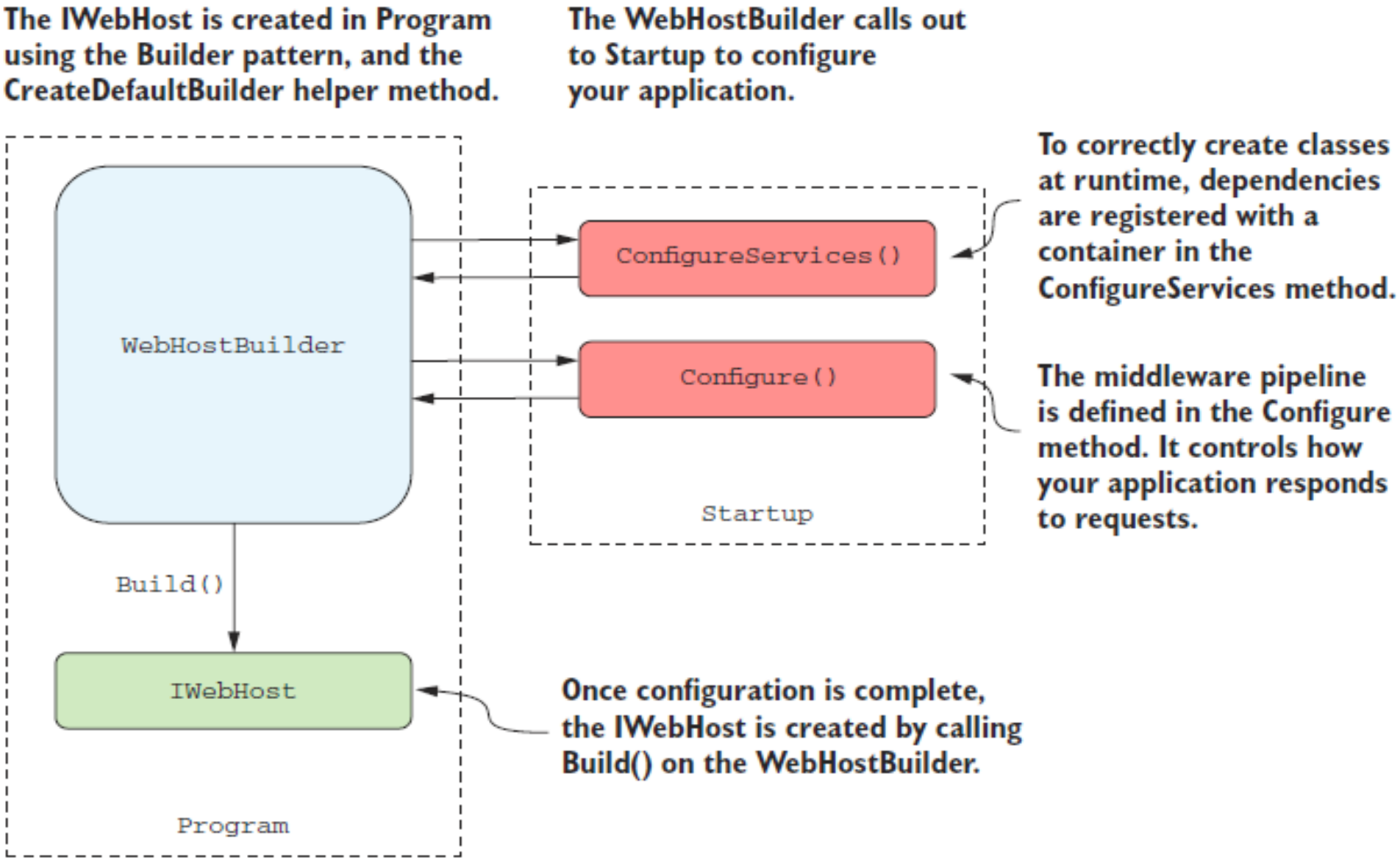
An outline of Startup.cs

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        // method details
    }

    public void Configure(IApplicationBuilder app)
    {
        // method details
    }
}
```

Configure services by registering services with the IServiceCollection.

Configure the middleware pipeline for handling HTTP requests.

The Startup class: configuring your application



Startup.Configure:
Defining the middleware pipeline

```
public class Startup
{
    public void Configure(
        IApplicationBuilder app,
        IHostingEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
            app.UseBrowserLink();
        }
        else
        {
            app.UseExceptionHandler("/Home/Error");
        }

        app.UseStaticFiles();

        app.UseMvc(routes =>
        {

            routes.MapRoute(
                name: "default",
                template: "{controller=Home}/{action=Index}/{id?}");
        });
    }
}
```

The IApplicationBuilder is used to build the middleware pipeline.

Other services can be accepted as parameters.

Different behavior when in development or production

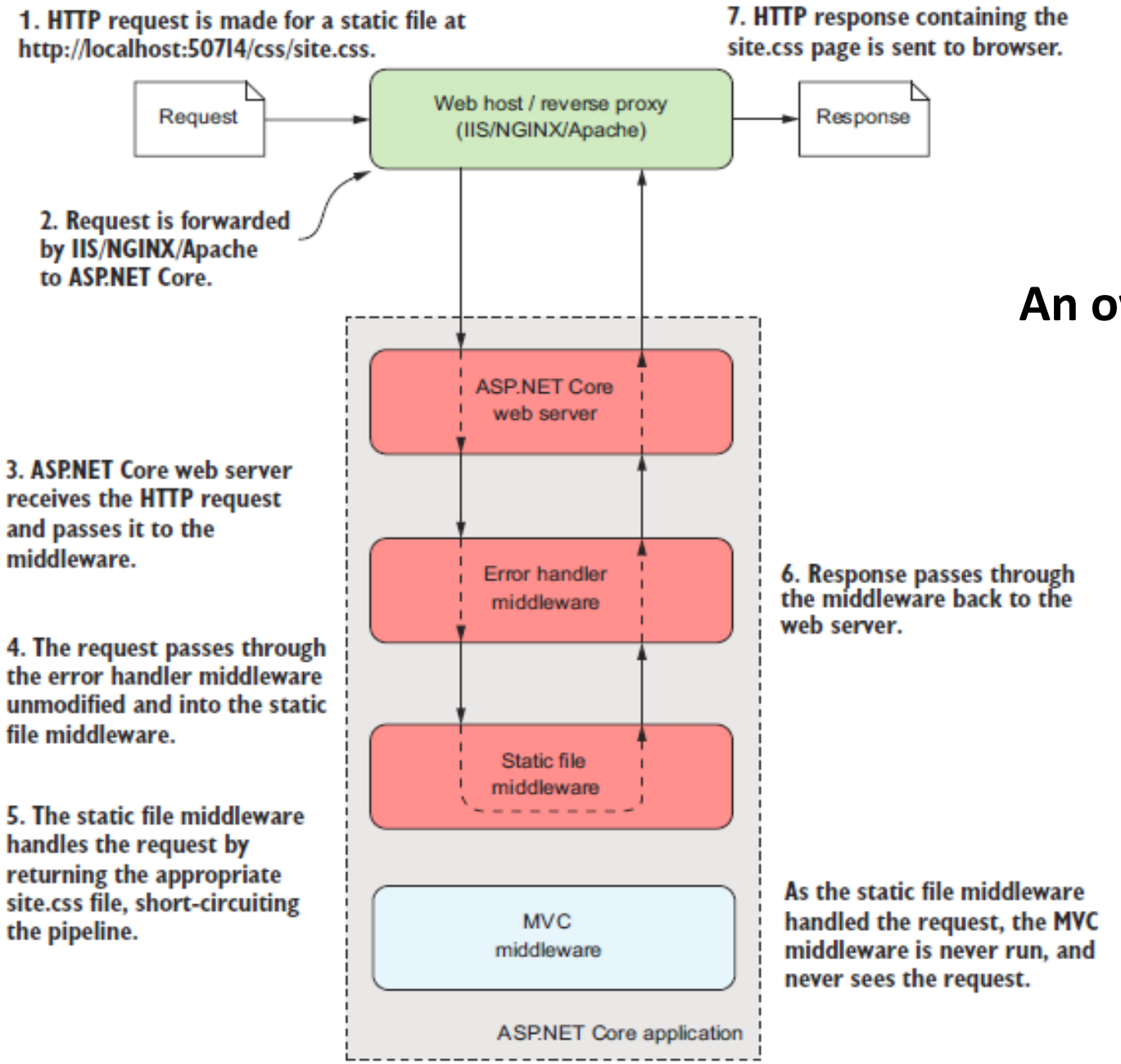
Only runs in a development environment

Only runs in a production environment

Adds the MVC middleware

Adds the static file middleware

An overview of a request for a static file



ASP.NET Core configuration model

- Loading settings from multiple configuration providers
- Storing sensitive settings safely
- Using strongly typed settings objects
- Using different settings in different hosting environments

Loading settings from multiple configuration providers

- ASP.NET Core uses *configuration providers* to load key-value pairs from a variety of sources
- Applications can use many different configuration providers
  - Environment Variables
  - Command-line arguments
  - Database
  - Remote Service
  - Custom *Configuration Provider*
- Format
  - Json
  - Xml
  - Yaml
- overriding settings
  - Each configuration provider can define its own settings, or it can overwrite settings from a previous provider

Creating an instance of WebHostBuilder

Kestrel is the default HTTP server in ASP.NET Core.

```
public static IWebHostBuilder CreateDefaultBuilder(string[] args)
{
  var builder = new WebHostBuilder()
    .UseKestrel()
    .UseContentRoot(Directory.GetCurrentDirectory())
    .ConfigureAppConfiguration((hostingContext, config) =>
    {
        // Configuration provider setup
    })
```

The content root defines the directory where configuration files can be found.

Configures application settings,

```
    {
        logging.AddConfiguration(
          hostingContext.Configuration.GetSection("Logging"));
        logging.AddConsole();
        logging.AddDebug();
    })
    .UseIISIntegration()
    .UseDefaultServiceProvider((context, options) =>
    {
        options.ValidateScopes =
          context.HostingEnvironment.IsDevelopment();
    });

  return builder;
}
```

Sets up the logging infrastructure

Configures the DI container. The ValidateScopes option checks for captured dependencies.

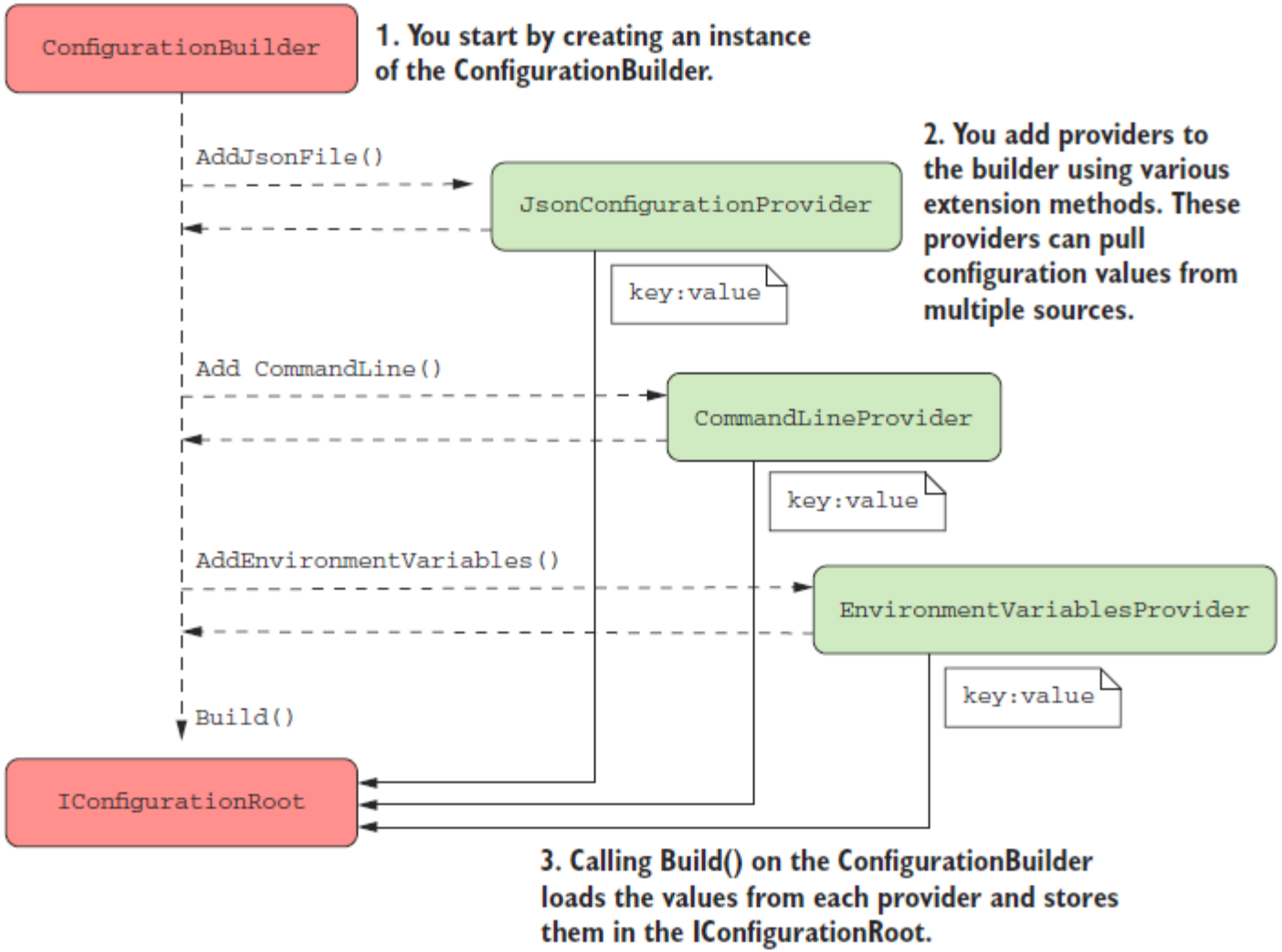
Returns WebHostBuilder for further configuration before calling Build()

When running on windows, IIS integration is automatically configured.

The WebHost.CreateDefaultBuilder method

Building a configuration object for your app

- Main Constructs
  - Configuration-Builder
    - Describes how to construct the final configuration representation for your app
  - IConfigurationRoot
    - Holds the configuration values themselves



Microsoft.AspNetCore.All Package

- JSON files—Microsoft.Extensions.Configuration.Json
- XML files—Microsoft.Extensions.Configuration.Xml
- Environment variables—Microsoft.Extensions.Configuration.EnvironmentVariables
- Command-line arguments—Microsoft.Extensions.Configuration.CommandLine
- Azure Key Vault—Microsoft.Extensions.Configuration.AzureKeyVault

Loading appsettings.json using a custom WebHostBuilder

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

    public static IWebHost BuildWebHost(string[] args) =>
        new WebHostBuilder()
            .UseKestrel()
            .UseContentRoot(Directory.GetCurrentDirectory())
            .ConfigureAppConfiguration(AddAppConfiguration)
            .ConfigureLogging(
              (hostingContext, logging) => { /* Detail not shown */ })
            .UseIISIntegration()
            .UseDefaultServiceProvider(
              (context, options) =>{ /* Detail not shown */ })
            .UseStartup<Startup>()
            .Build();

    public static void AddAppConfiguration(
        WebHostBuilderContext hostingContext,
        IConfigurationBuilder config)
    {
        config.AddJsonFile("appsettings.json", optional: true);
    }
}
```

Adds the configuration setup function to WebHostBuilder

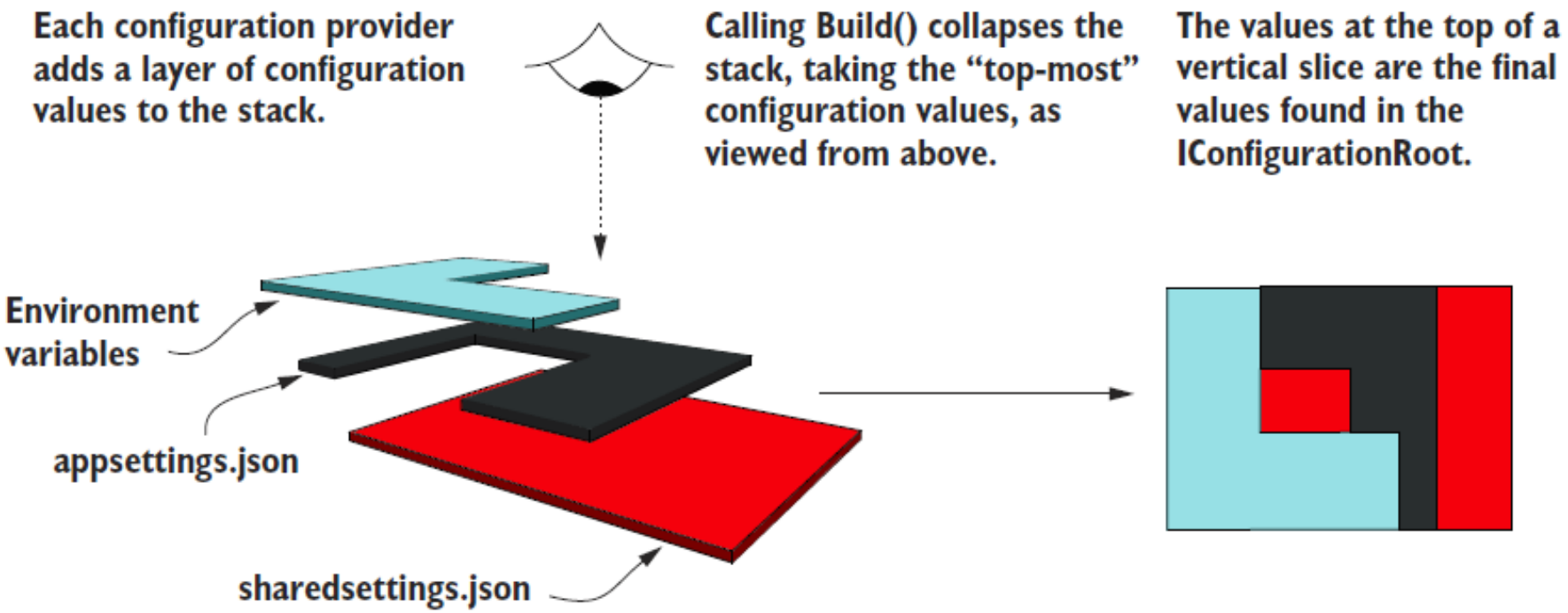
WebHostBuilder provides a hosting context and an instance of ConfigurationBuilder.

Adds a JSON configuration provider, providing the filename of the configuration file

The **`WebHostBuilder`** instance takes care of calling `Build()`, which generates **`IConfigurationRoot`** which represents your configuration object. This is then registered as an **IConfiguration** instance with the DI container

Using multiple providers to override configuration values

- The order of adding configuration providers to ConfigurationBuilder is important



Reloading appsettings.json when the file changes

```
public class Program
{
    /* Additional Program configuration*/

    public static void AddAppConfiguration(
        WebHostBuilderContext hostingContext,
        IConfigurationBuilder config)
    {
        config.AddJsonFile(
            "appsettings.json",
            optional: true
            reloadOnChange: true);
    }

}
```

IConfigurationRoot will be rebuilt if the appsettings.json file changes.

Configuring an application for multiple environments

- ***Identifying the hosting environment***
  - ASPNETCORE_ENVIRONMENT - magic environment variable
  - IHostingEnvironment object
    - EnvironmentName
      - Development
      - Staging
      - Production

```
IHostingEnvironment.IsDevelopment()
IHostingEnvironment.IsStaging()
IHostingEnvironment.IsProduction()
IHostingEnvironment.IsEnvironment(string environmentName)
```

Loading environment-specific configuration files

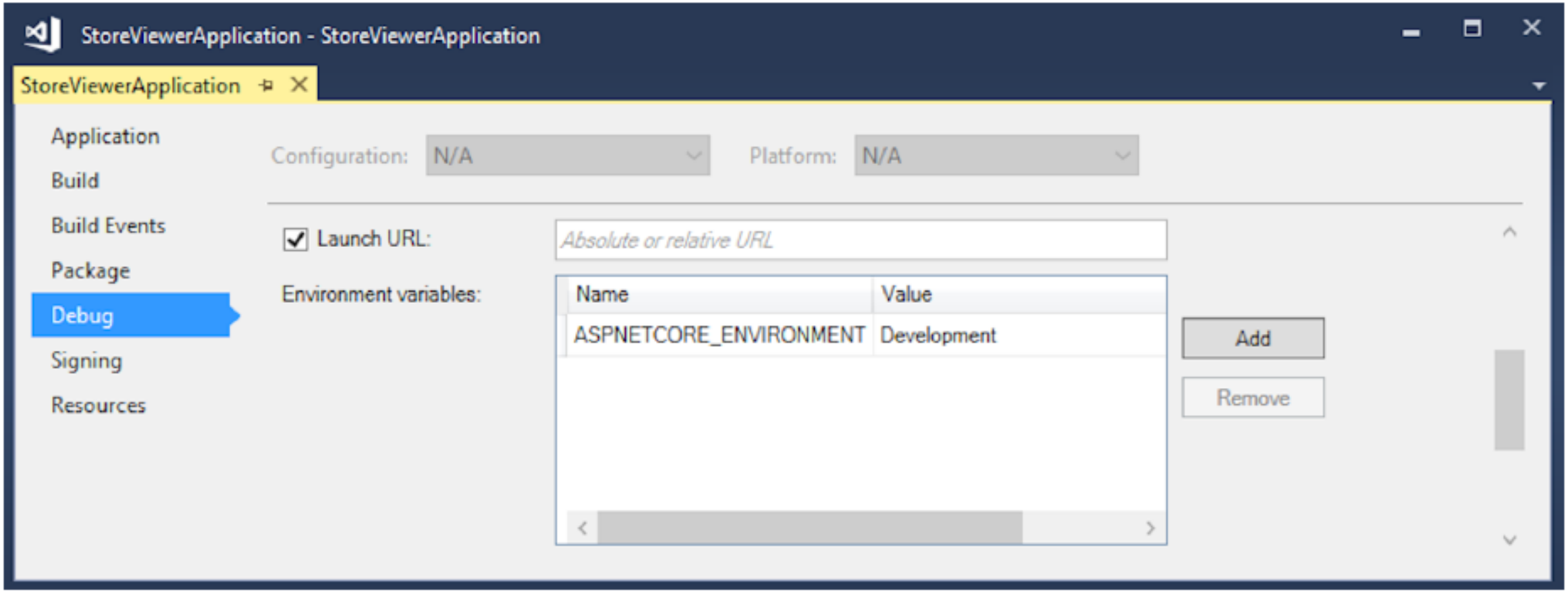
```
public class Program
{
    /* Additional Program configuration*/

    public static void AddAppConfiguration(
        WebHostBuilderContext hostingContext,
        IConfigurationBuilder config)
    {
        var env = hostingContext.HostingEnvironment;
        config

            .AddJsonFile(
                "appsettings.json",
                optional: false)
            .AddJsonFile
                $"appsettings.{env.EnvironmentName}.json",
                optional: true);
    }
}
```

The current IHostingEnvironment is available on WebHostBuilderContext.

It's common to make the base appsettings.json compulsory.

Adds an optional environment-specific JSON file where the filename varies with the environment

Setting the hosting environment



Hardcoding **EnvironmentName** with **UseEnvironment**

```
public static IWebHost BuildWebHost(string[] args) =>
    new WebHostBuilder()
        .UseEnvironment(EnvironmentName.Production)
        .UseKestrel()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .ConfigureAppConfiguration( /* Not shown*/)
        .UseIISIntegration()
        .UseStartup<Startup>()
        .Build();
```

This ignores ASPNETCORE_ENVIRONMENT and hardcodes the environment to Production.

Using strongly typed settings with the ***options pattern***

- The ASP.NET Core configuration system includes a *binder*, which can take a collection of configuration values and *bind* them to a strongly typed object, called an *options class*

```
services.Configure<HomePageSettings>(
    Configuration.GetSection("HomePageSettings"));
```

Binds the HomePageSettings section to the POCO options class HomePageSettings

```
"HomePageSettings": {
  "Title": "Visa Home Page ",
  "ShowCopyright": true
}
```

General settings related to the app's homepage

```
public HomeController(IOptions<HomePageSettings> options)
{
    HomePageSettings settings = options.Value;
    var title = settings.Title;
    var showCopyright = settings.ShowCopyright;
}
```

You can inject a strongly typed options class using the IOptions<> wrapper interface.

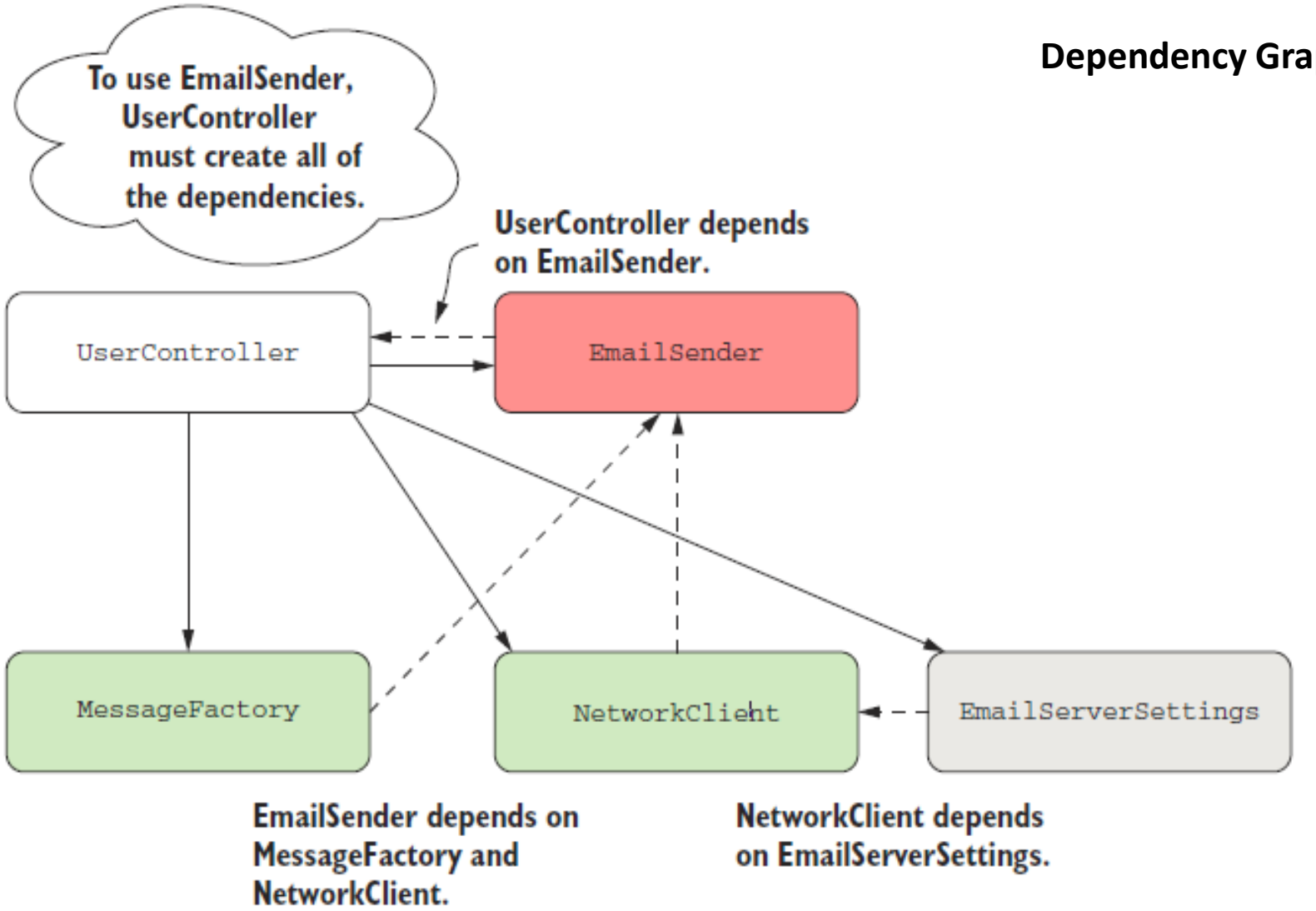
Designing ***options classes*** *for* ***automatic binding***

- Be non-abstract
- Have a default (public parameterless) constructor
- Is public
- Has a getter—the binder won’t write set-only properties
- Has a setter or a non-null value
- Is not an indexer

Service configuration with ***dependency injection***

- Understanding the benefits of dependency injection
- How ASP.NET Core uses dependency injection
- Configuring your services to work with dependency injection
- Choosing the correct lifetime for your services

Dependency Graph



```
public IActionResult RegisterUser(string username)
{
    var emailSender = new EmailSender(
        new MessageFactory(),
        new NetworkClient(
            new EmailServerSettings
            (
                host: "smtp.server.com",
                port: 25
            ))
```

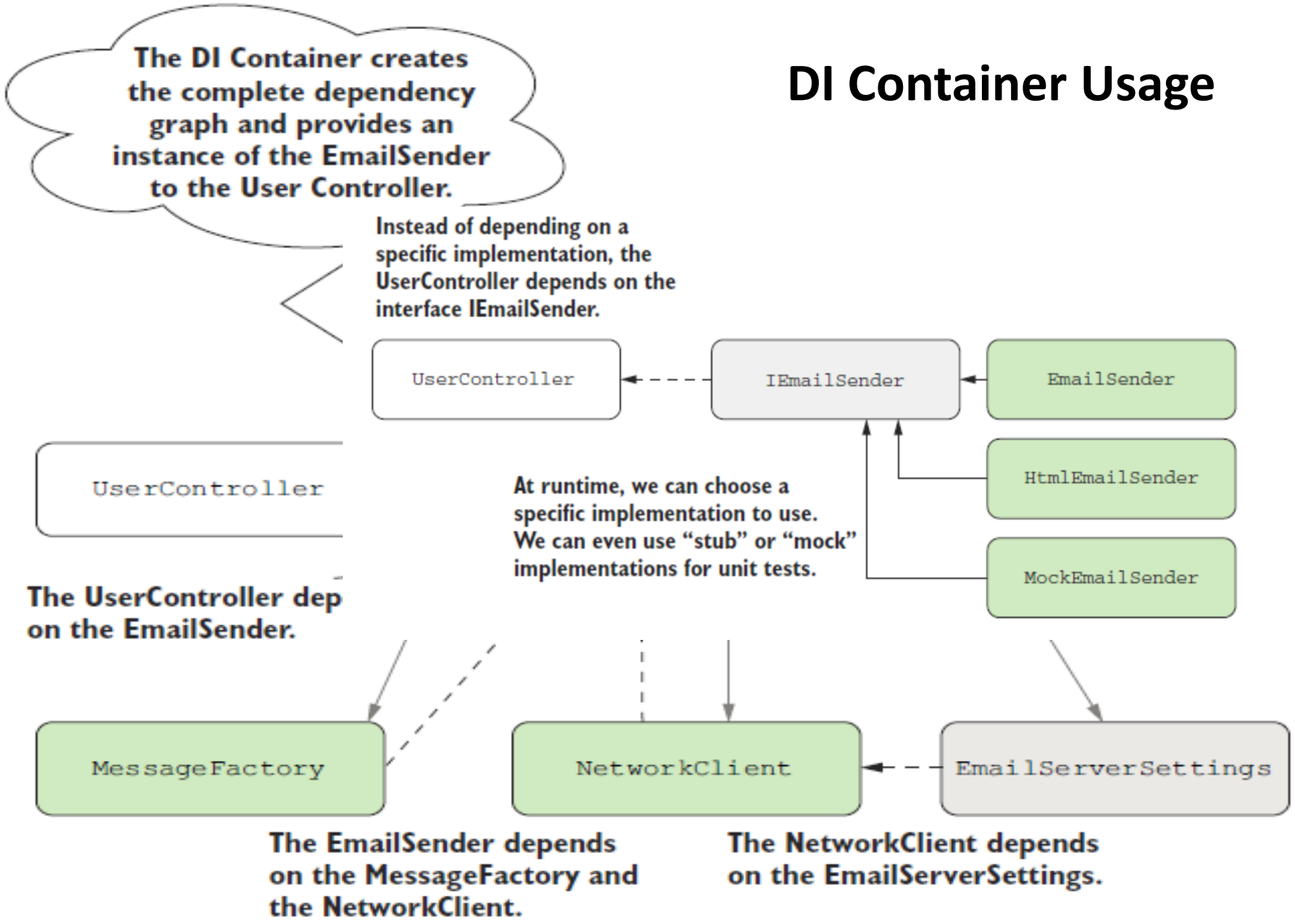
To create EmailSender, you also have to create all of its dependencies.

You need a new MessageFactory.

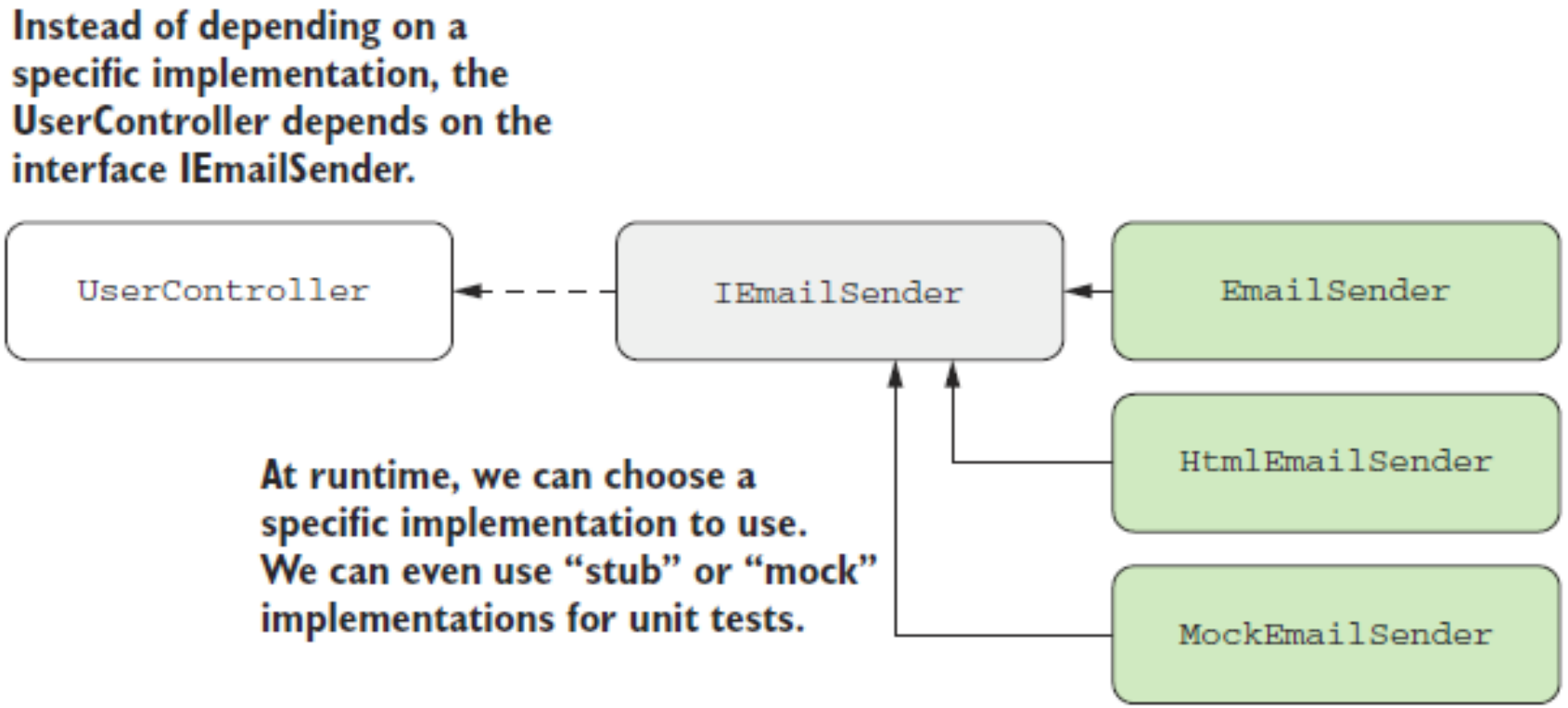
The NetworkClient also has dependencies.

You're already two layers deep, but there could feasibly be more.

DI Container Usage

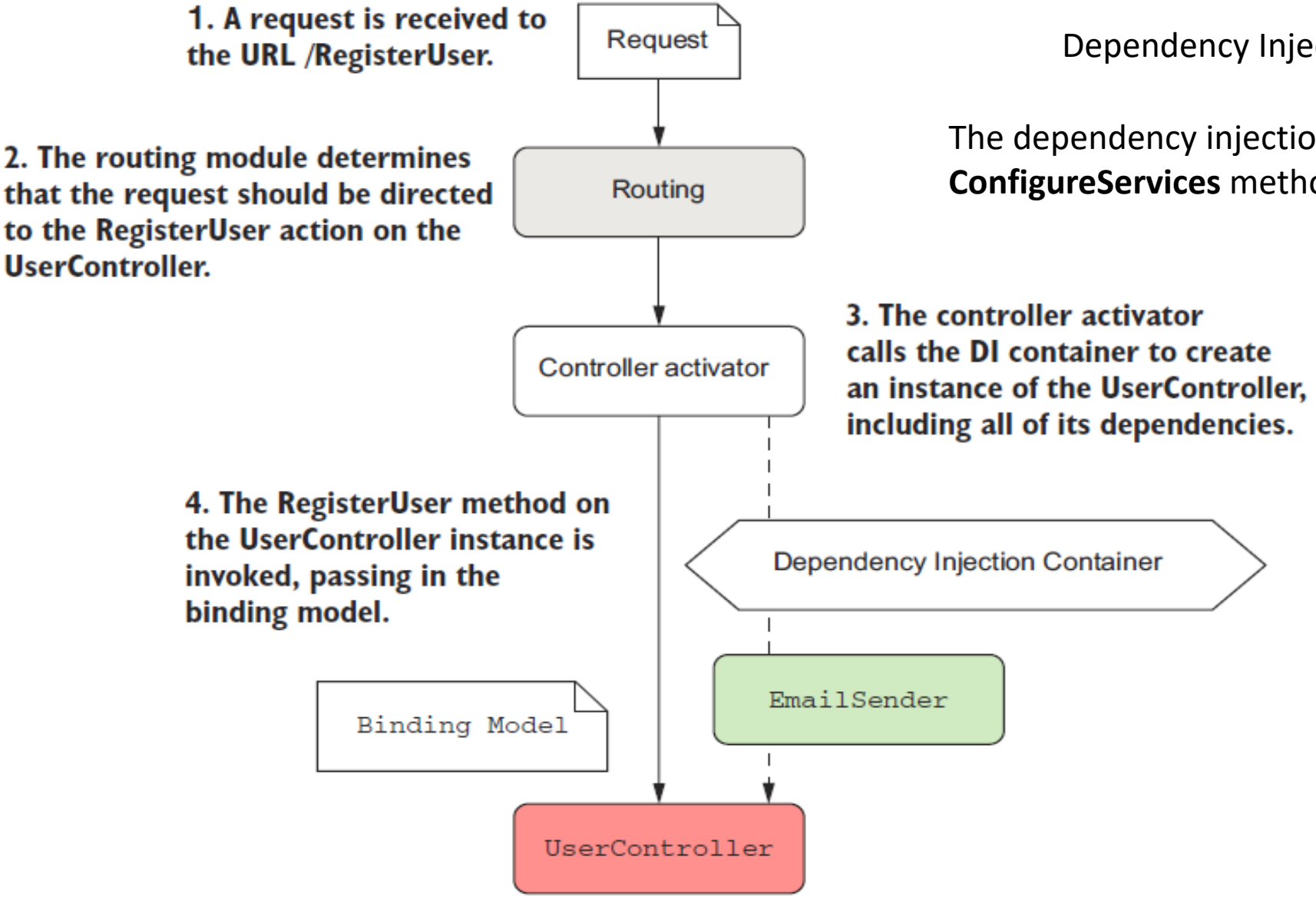


Loosely Coupled Code



Dependency Injection in Asp.net Core

The dependency injection container is set up in the **ConfigureServices** method of your Startup class in Startup.cs



Registering dependencies

- New instance "per call"

```
services.AddTransient<IMyCustomService, MyCustomService>();
```

- New instance per HTTP request

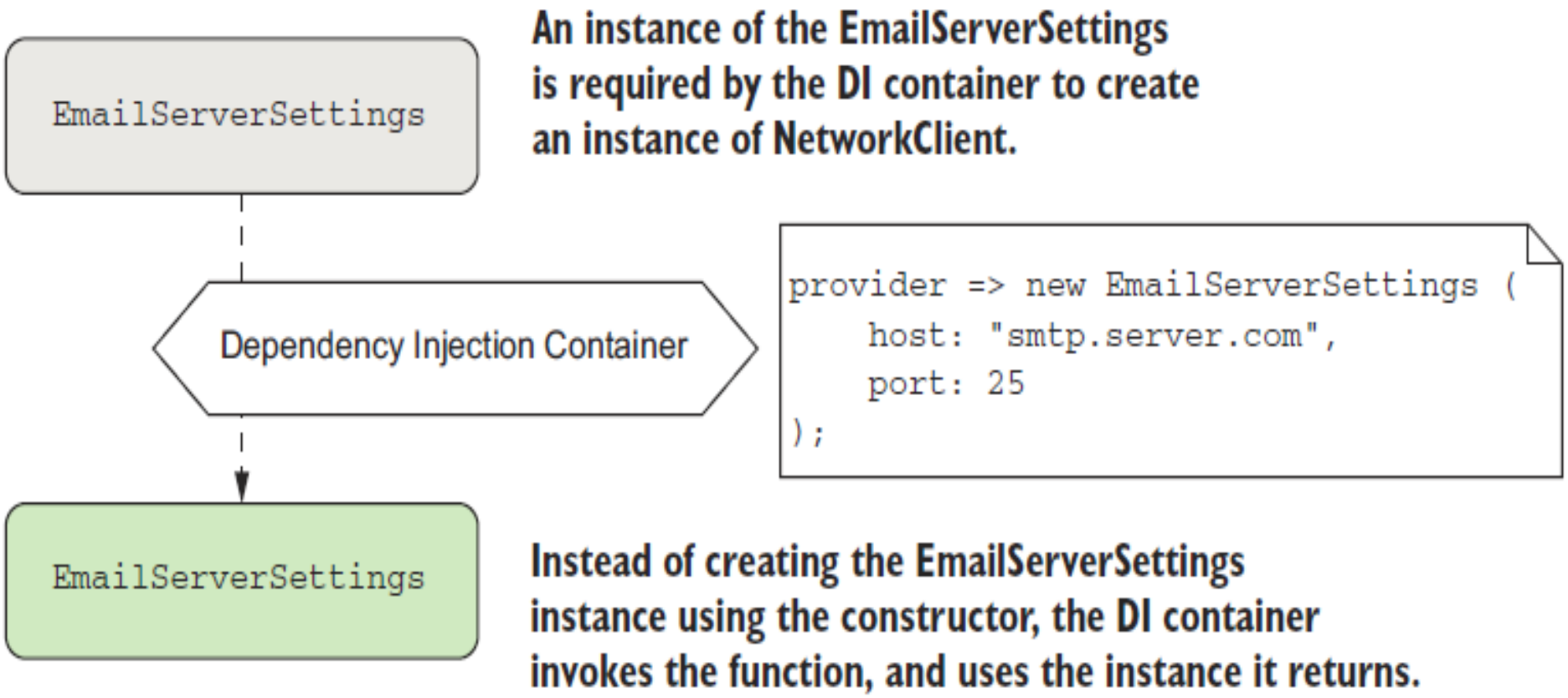
```
services.AddScoped<IMyCustomService, MyCustomService>();
```

- Singleton

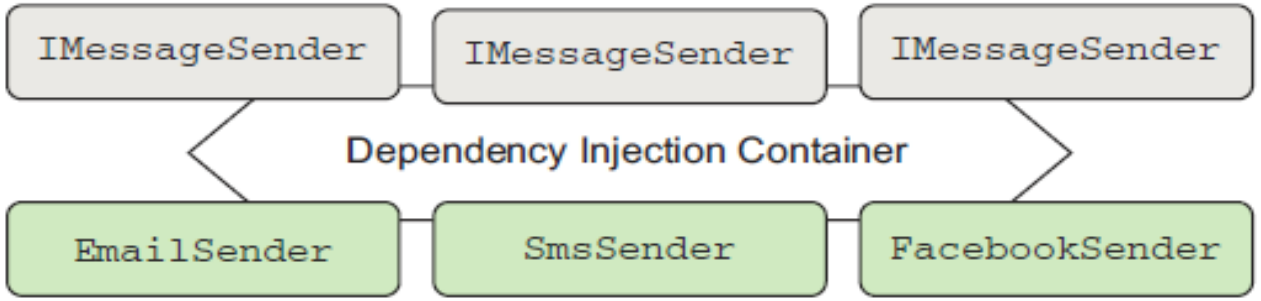
```
services.AddSingleton<IMyCustomService, MyCustomService>();
```

Add* methods limitation

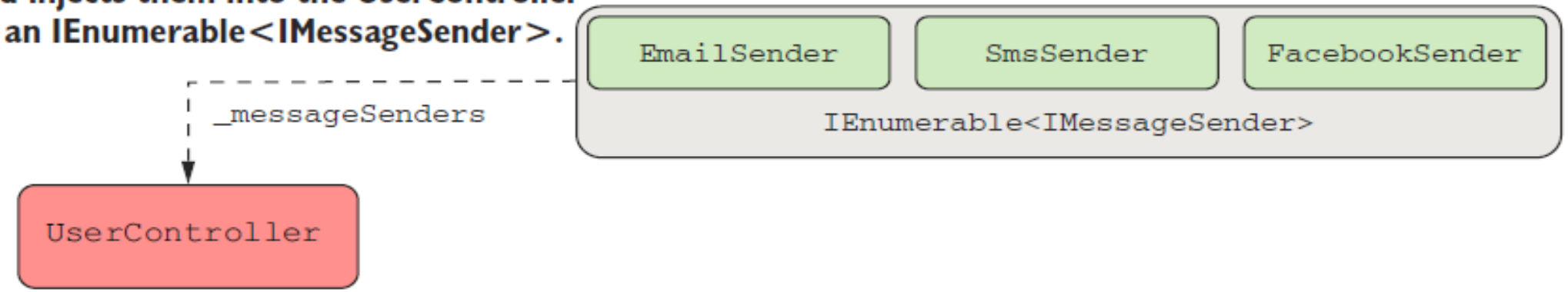
- The class must be a concrete type.
- The class must have only a single “valid” constructor that the container can use.
- For a constructor to be “valid,” all constructor arguments must be registered with the container, or must be an argument with a default value
- Under the hood, the built-in ASP.NET Core DI container uses reflection to create dependencies


EmailServerSettings
An instance of the EmailServerSettings is required by the DI container to create an instance of NetworkClient.
Dependency Injection Container
provider => new EmailServerSettings (
host: "smtp.server.com",
port: 25
);
EmailServerSettings
Instead of creating the EmailServerSettings instance using the constructor, the DI container invokes the function, and uses the instance it returns.

1. During Startup, multiple implementations of IMessageSender are registered with the DI container using the normal Add* methods.



2. The DI container creates one of each IMessageSender implementation, and injects them into the UserController as an IEnumerable<IMessageSender>.




```
foreach (var messageSender in _messageSenders)
{
  messageSender.SendMessage(username);
}
```

3. The RegisterUser method on the UserController loops over the IMessageSender instances and calls SendMessage on each.

Conditionally adding a service using **TryAddScoped**

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddScoped<IMessageSender, EmailSender>();
    services.TryAddScoped<IMessageSender, SmsSender>();
}
```

EmailSender is registered with the container.

There's already an IMessageSender implementation, so SmsSender isn't registered.

Understanding lifetimes: when are services created?

- The lifetime of a service is how long an instance of a service should ***live*** in a container before it creates a new instance
- *Transient*—Every single time a service is requested, a new instance is created. This means you can potentially have different instances of the same class within the same dependency graph.
- *Scoped*—Within a *scope*, all requests for a service will give you the same object. For different scopes you'll get different objects. In ASP.NET Core, each web request gets its own scope. anything that you want to share across your services ***within*** a single request, but that needs to change ***between*** requests

- *Singleton*—You'll always get the same instance of the service, no matter which scope.

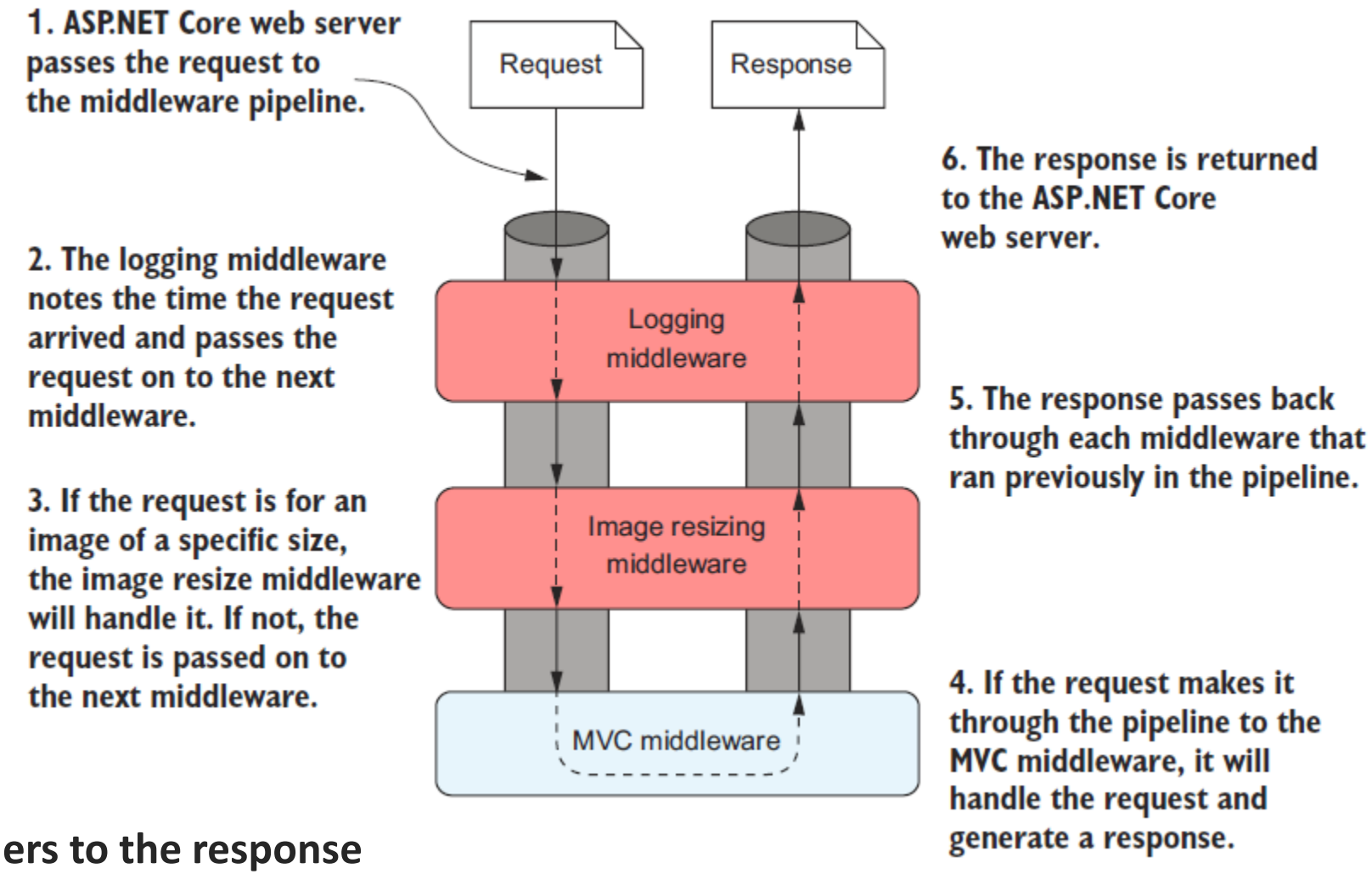
Captured Dependencies

- injecting a *scoped* object, **DataContext**, into a *singleton* ***DataServices*** *causes Captured Dependencies*
- Captured dependencies can cause subtle bugs that are hard to root out
- *Best Practices*
  - A service should only use dependencies with a lifetime longer than or equal to the lifetime of the service.
  - A service registered as a singleton can only use singleton dependencies.
  - A service registered as scoped can use scoped or singleton dependencies.
  - A transient service can use dependencies with any lifetime.

Understanding ASP.NET Middleware

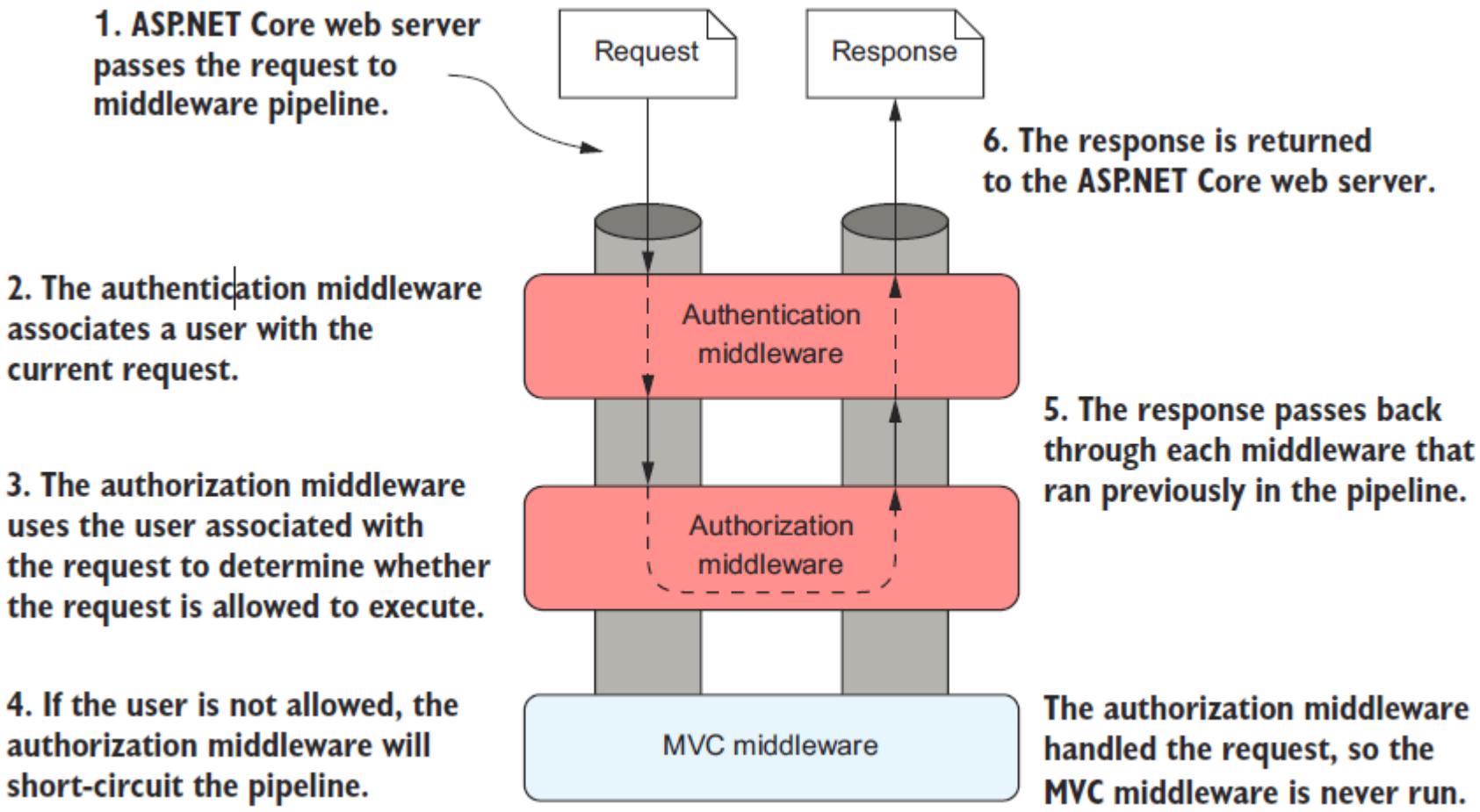
- The term used for the components that are combined to form the **request pipeline**
- The **request** pipeline is arranged like a chain
- Middleware components don’t implement an interface or derive from a common base class
  - Define a constructor that takes a RequestDelegate object and define an Invoke method
  - **RequestDelegate** object represents the **next** middleware component in the chain
  - **Invoke** method is called when ASP.NET **receives** an **HTTP request**
- **Register** Middleware using **app.UseMiddleware() in startup Config()**
- Middleware Usecases
  - Content-Generation
  - Request- Editing
  - Response- Editing

Middleware Usecases



- Logging each request
- Adding standard security headers to the response
- Associating a request with the relevant user
- Setting the language for the current request

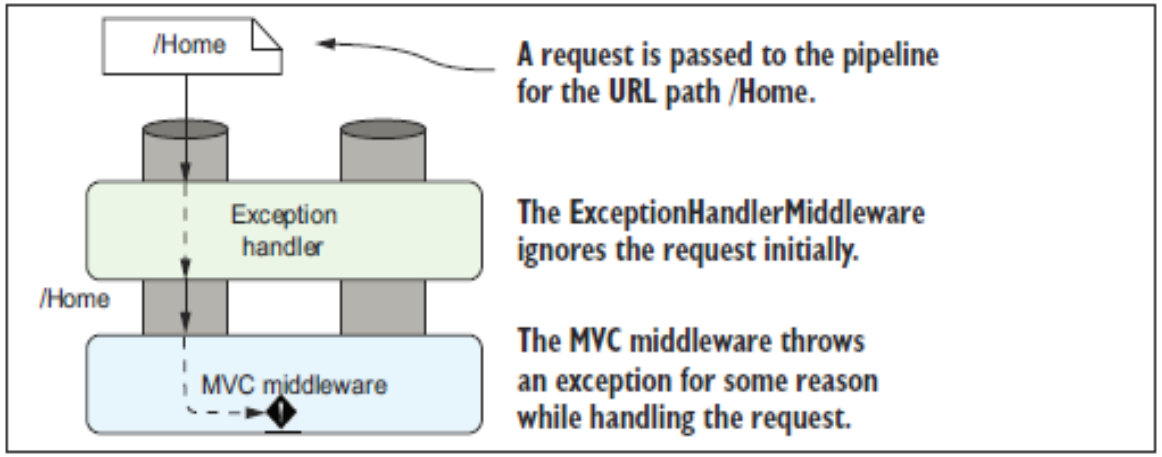
ShortCircuit Pipeline

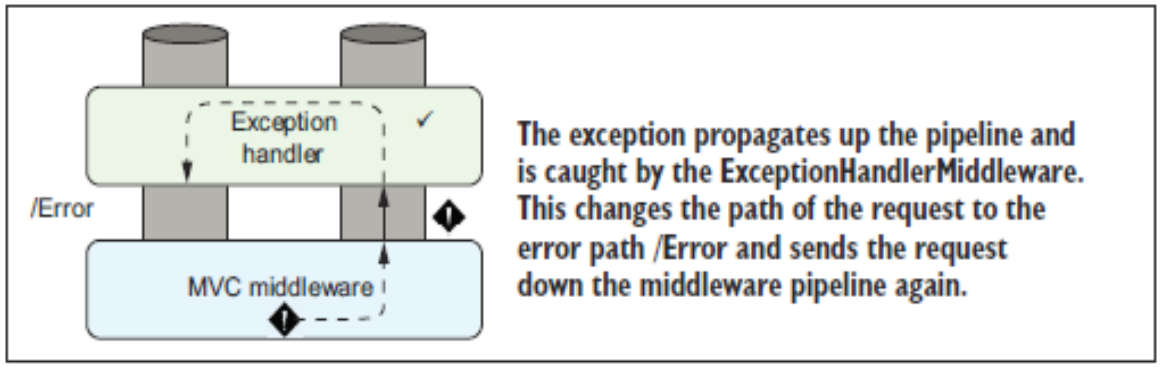


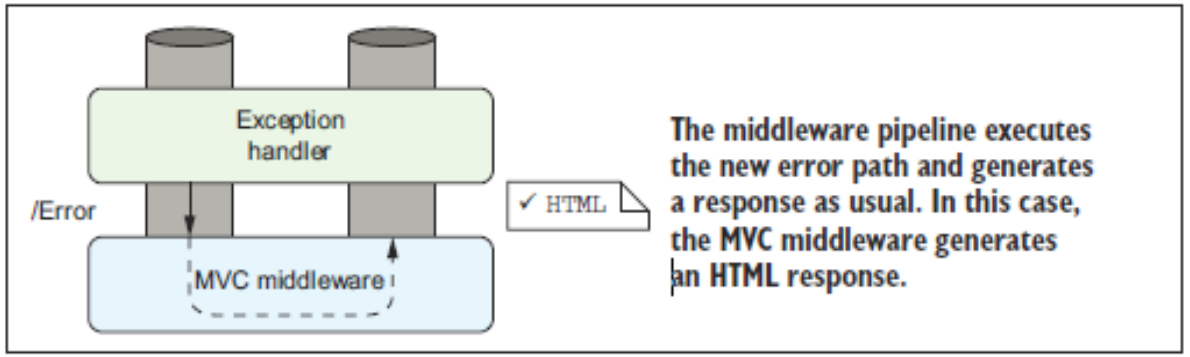
Handling errors using middleware

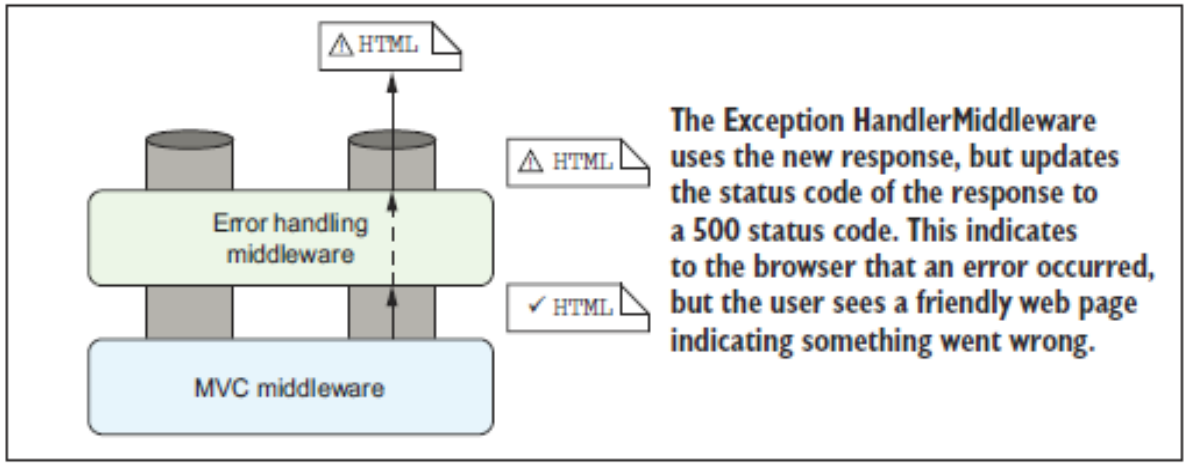
- Exceptions
  - Development Mode
    - DeveloperExceptionPageMiddleware
    - app.UseDeveloperExceptionPage();
  - Production Mode
    - ExceptionHandlerMiddleware
    - app.UseExceptionHandler("/home/error");
- StatusCode Errors
  - StatusCodePagesMiddleware
  - app.UseStatusCodePages();
  - app.UseStatusCodePagesWithReExecute("/error/{0}");

How ExceptionHandlerMiddleware Works









Creating simple middleware using the **Run** extension

Run based middleware must always place it at the end of the pipeline, as no middleware placed after it will execute

```
public void Configure(IApplicationBuilder app)
{
    app.Run(async (context) =>
    {
        context.Response.ContentType = "text/plain";
        await context.Response.WriteAsync(
            DateTime.UtcNow.ToString());
    });

    app.UseMvc();
}
```

Uses the Run extension to create a simple middleware that always returns a response

You should set the content-type of the response you're generating.

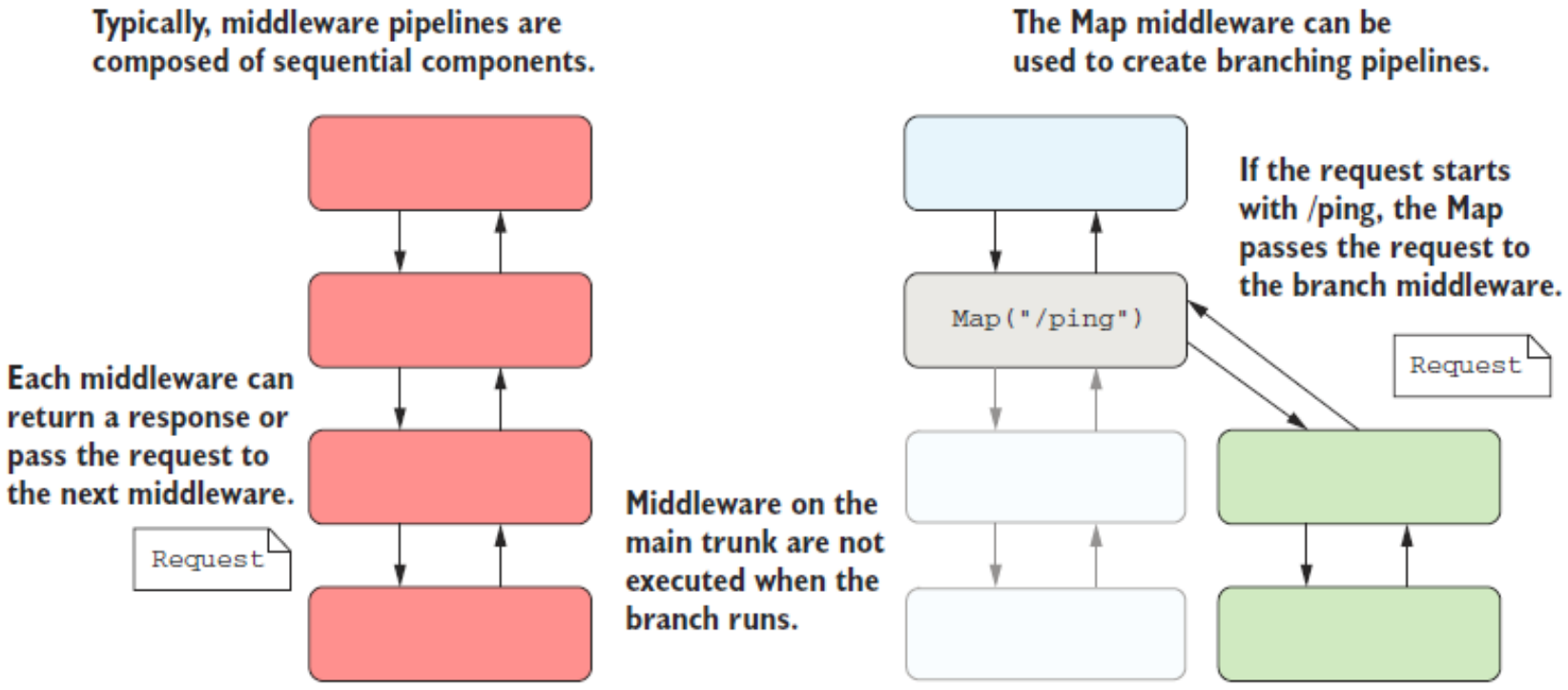
Returns the time as a string in the response. The 200 OK status code is used if not explicitly set.

Any middleware added after the Run extension will never execute.

Branching middleware pipelines with the Map extension

The `Map` extension method lets you change that simple pipeline into a branching structure

Each branch of the pipeline is independent



Traditional sequential middleware pipeline

Branching middleware pipeline

Using the **Map** extension to create branching middleware pipelines

```
public void Configure(IApplicationBuilder app)
{
    app.UseDeveloperExceptionPage();

    app.Map("/ping", branch =>
    {
        branch.UseExceptionHandler();

        branch.Run(async (context) =>
        {
            context.Response.ContentType = "text/plain";
            await context.Response.WriteAsync("pong");
        });
    });

    app.UseMvc();
}
```

Every request will pass though this middleware.

The Map extension method will branch if a request starts with /ping.

This middleware will only run for requests matching the /ping branch.

The MvcMiddleware will run for requests that don't match the /ping branch.

The Run extension always returns a response, but only on the /ping branch.

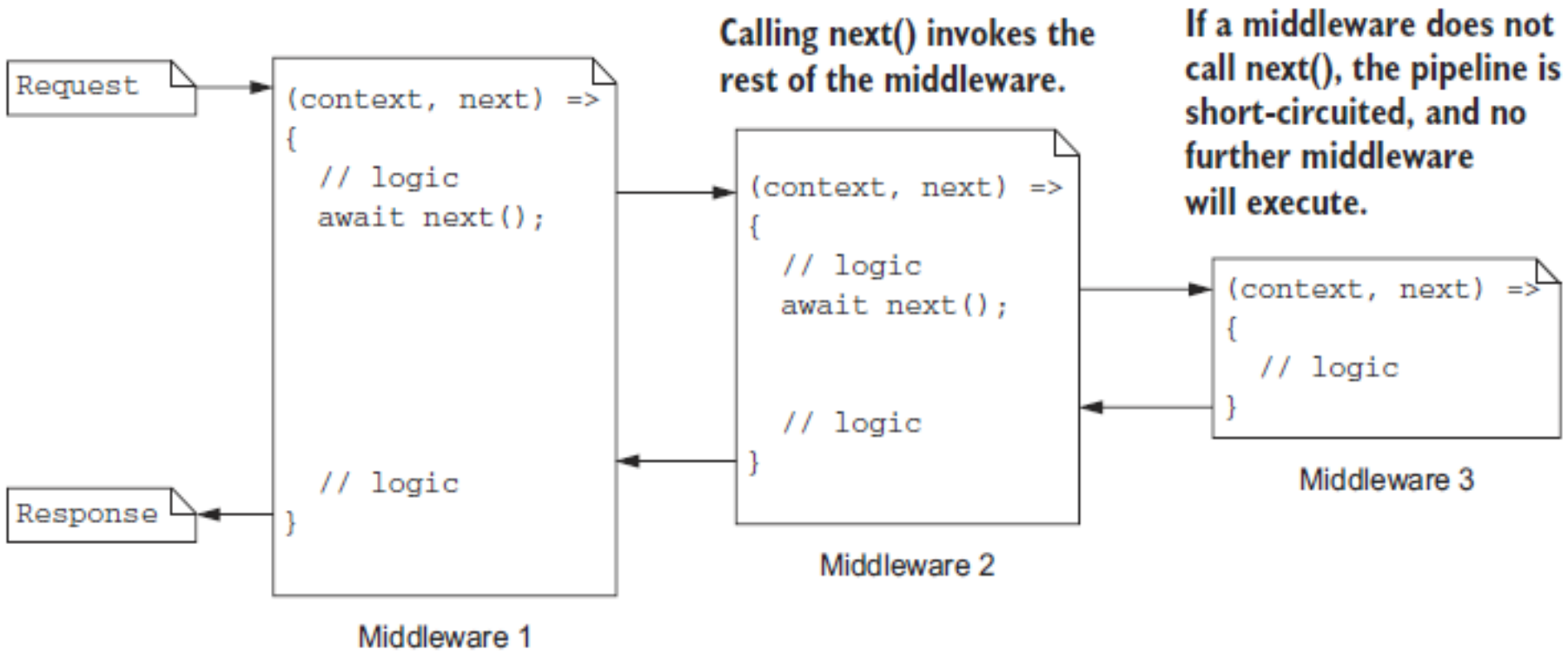
Middleware with the **Use extension**

- Allows to Add a general-purpose piece of middleware
- Use Cases
  - View and modify requests as they arrive
  - Generate a response
  - Pass the request on to subsequent middleware in the pipeline
- Syntax

```
app.Use(async (context, next) => { await next.Invoke(); });
```

- The HttpContext representing the current request and response
- **Next is a** pointer to the rest of the pipeline as a Func<Task>.

Use in Action



Adding headers to a response with the **Use** extension

```
public void Configure(IApplicationBuilder app)
{
    app.Use(async (context, next) =>
    {
        context.Response.OnStarting(() =>
        {
            context.Response.Headers["Strict-Transport-Security"] =
                "max-age=60";
            return Task.CompletedTask;
        });
        await next();
    }

    app.UseMvc();
}
```

Adds the middleware at the start of the pipeline

Sets a function that should be called before the response is sent to the browser

Adds an HSTS header. For 60 seconds the browser will only send HTTPS requests to your app.

The function passed to OnStarting must return a Task.

Invokes the rest of the middleware pipeline

No matter what response the MvcMiddleware generates, it'll have the HSTS header.

```
app.UseMiddleware<HeadersMiddleware>();
```

Building a custom middleware component

```
public class HeadersMiddleware
{
    private readonly RequestDelegate _next;
    public HeadersMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task Invoke(HttpContext context)
    {
        context.Response.OnStarting(() =>
        {
            context.Response.Headers["Strict-Transport-Security"] =
                "max-age=60";
            return Task.CompletedTask;
        });

        await _next(context);
    }
}
```

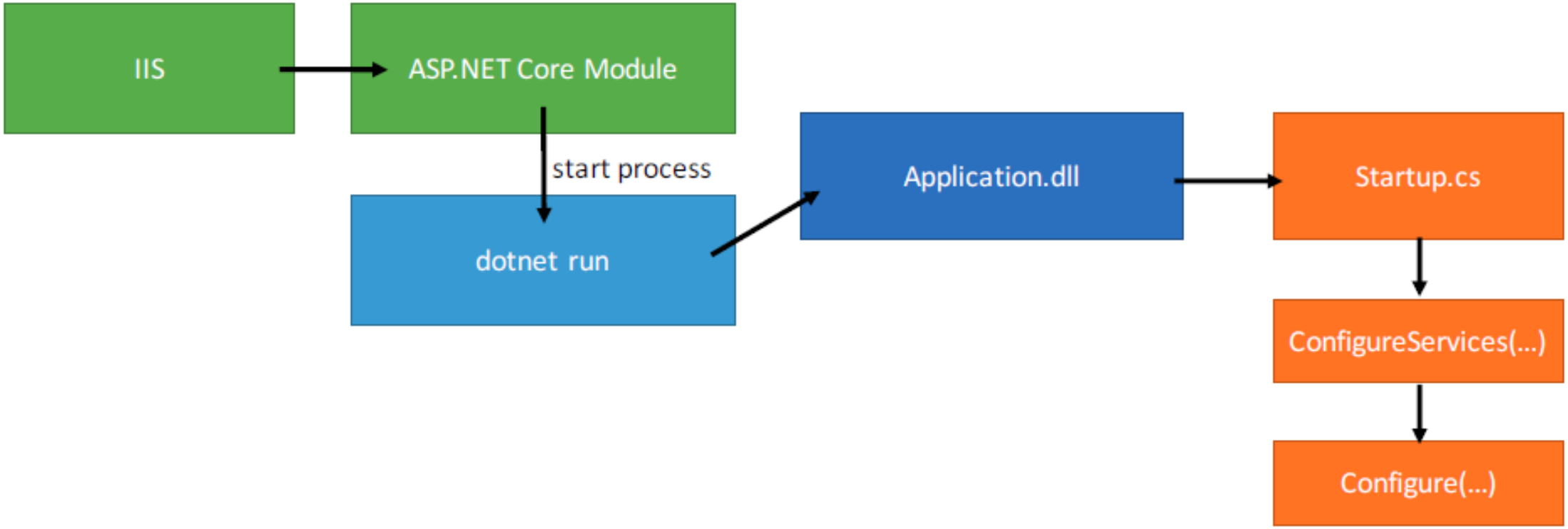
The RequestDelegate represents the rest of the middleware pipeline

The Invoke method is called with HttpContext when a request is received.

Adds the HSTS header response as before

Invokes the rest of the middleware pipeline. Note that you must pass in the provided HttpContext.

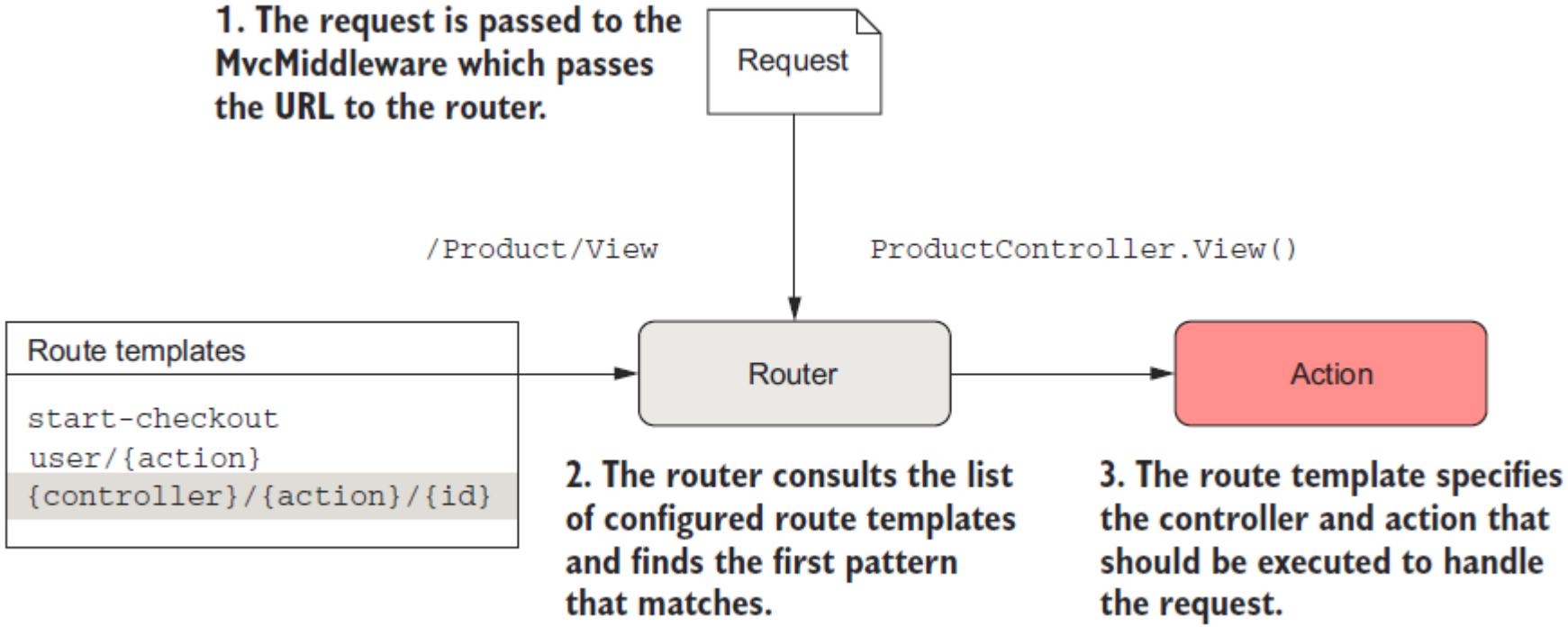
How ASP.NET Core Applications start



Dependency Injection in Asp.Net Core

- Preparing for Dependency Injection
- Configuring the Service Provider
- Using Dependency Chains
- Using Dependency Injection for Concrete Types
- Using Factory Function
- Understanding Service Life Cycles
  - Transient
    - service provider create a new instance of the implementation type whenever it needs to resolve a dependency
  - Scoped
    - Service provider creates a single object from the implementation class that is used to resolve all of the dependencies associated with a single scope, which generally means a single HTTP request
  - Singleton
    - single object is used to resolve all the dependencies for a given service type
- Using Action Injection
  - dependencies to be declared through parameters to action methods
  - Action injection is performed using the FromServices attribute
- Manually Requesting an Implementation Object
  - HttpContext.RequestServices.GetService<IRepository>();
- Using the Property Injection Attributes

What Router Does



URL Routing

```
app.UseMvc(routes => {
    routes.MapRoute(name: "default", template: "{controller}/{action}");
});
```

- The routing system is responsible for processing incoming requests and selecting controllers and action methods to process them
- *outgoing URLs*.
  - Generate Routes from View
- Convention Based Routing
  - The mapping between URLs and the controllers and action methods is defined in the Startup.cs
- Attribute Based Routing
  - Applying the Route attribute to controllers

Using configuration

```
public class Startup
{
    IConfiguration _configuration;

    public Startup()
    {
        _configuration = new ConfigurationBuilder()
          ...
          .Build();
    }

    public void Configure(IApplicationBuilder app)
    {
        var copyright = new Copyright
        {
            Company = _configuration.Get("copyright_company"),
            Year = _configuration.Get("copyright_year")
        };

        app.Run(async (context) =>
        {
            await context.Response.WriteAsync($"Copyright {copyright.Year}, {copyright.Company}");
        });
    }
}
```

```
{
  "copyright": {
    "year": "2015",
    "company": "Foo Industries"
  }
}
```

- Microsoft.AspNetCore.Routing
- IRouteBuilder
- Defining Default Values
- Defining Inline Default Values
- Static URL Segments

```
routes.MapRoute(
    name: "default",
    template: "{controller}/{action}",
    defaults: new { action = "Index" });
});
```

```
template: "{controller}/{action=Index}");
```

```
routes.MapRoute(name: "",
    template: "Public/{controller=Home}/{action=Index}");
```

```
routes.MapRoute("", "X{controller}/{action}");
```

Defining Custom Segment Variables

```
template: "{controller=Home}/{action=Index}/{id?}");
```

```
template: "{controller=Home}/{action=Index}/{id?}/{*catchall}");
```

Constraining Routes

```
template: "{controller=Home}/{action=Index}/{id:int?}");
```

```
template: "{controller}/{action}/{id?}",
defaults: new { controller = "Home", action = "Index" },
constraints: new { id = new IntRouteConstraint() });
```

Microsoft.AspNetCore.Routing.Constraints

- AlphaRouteConstraint()
- BoolRouteConstraint()
- DateTimeRouteConstraint()
- DecimalRouteConstraint()
- DoubleRouteConstraint()
- FloatRouteConstraint()
- GuidRouteConstraint()
- MaxLengthRouteConstraint(len)

……..

Constraining a Route Using a Regular Expression

```
template: "{controller:regex(^H.*)=Home}/{action=Index}/{id?}");

template: "{controller:regex(^H.*)=Home}/"
  + "{action:regex(^Index$|^About$)=Index}/{id?}");
```

Combining Constraints

```
template: "{controller=Home}/{action=Index}"
    + "/{id:alpha:minlength(6)?}");
```

```
template: "{controller}/{action}/{id?}",
defaults: new { controller = "Home", action = "Index" },
constraints: new {  id = new CompositeRouteConstraint(
                                 new IRouteConstraint[] {
                                     new AlphaRouteConstraint(),
                                     new MinLengthRouteConstraint(6)
                                 })
                    }
```

Defining a Custom Constraint

- Implement – IRouteConstraint
- **constraints: new { id = new CustomConstraint() });**
- Defining an Inline Custom Constraint

```
services.Configure<RouteOptions>(options =>
                options.ConstraintMap.Add("customConstraint", typeof(CustomConstraint)));
```

```
template: "{controller=Home}/{action=Index}/{id:customConstraint?}"
```

Using Attribute Routing

```
[Route("myroute")]
public ViewResult Index() => View("Result",
    new Result {
        Controller = nameof(CustomerController),
        Action = nameof(Index)
    });
```

```
[Route("app/[controller]/actions/[action]/{id?}")]
public class CustomerController : Controller {
```

```
[Route("app/[controller]/actions/[action]/{id:weekday?}")]
```

ViewComponents

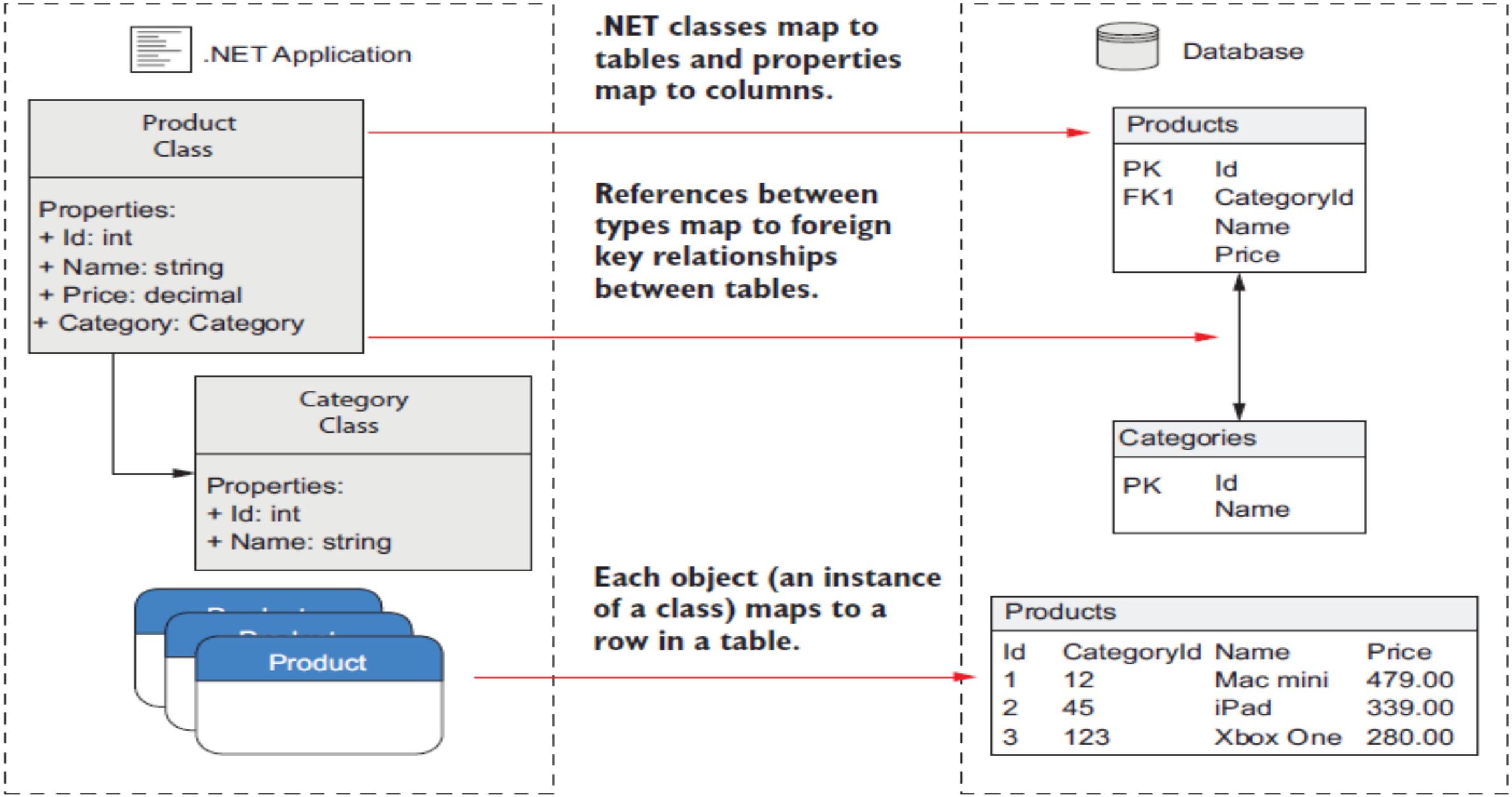
- A view component is a C# class that provides a partial view with the data that it needs, independently from the parent view and the action that renders it
- POCO ViewComponent
- Derive Form ViewComponent
- Use ViewComponent Attribute
- How to Invoke
  - @await Component.InvokeAsync("Poco")

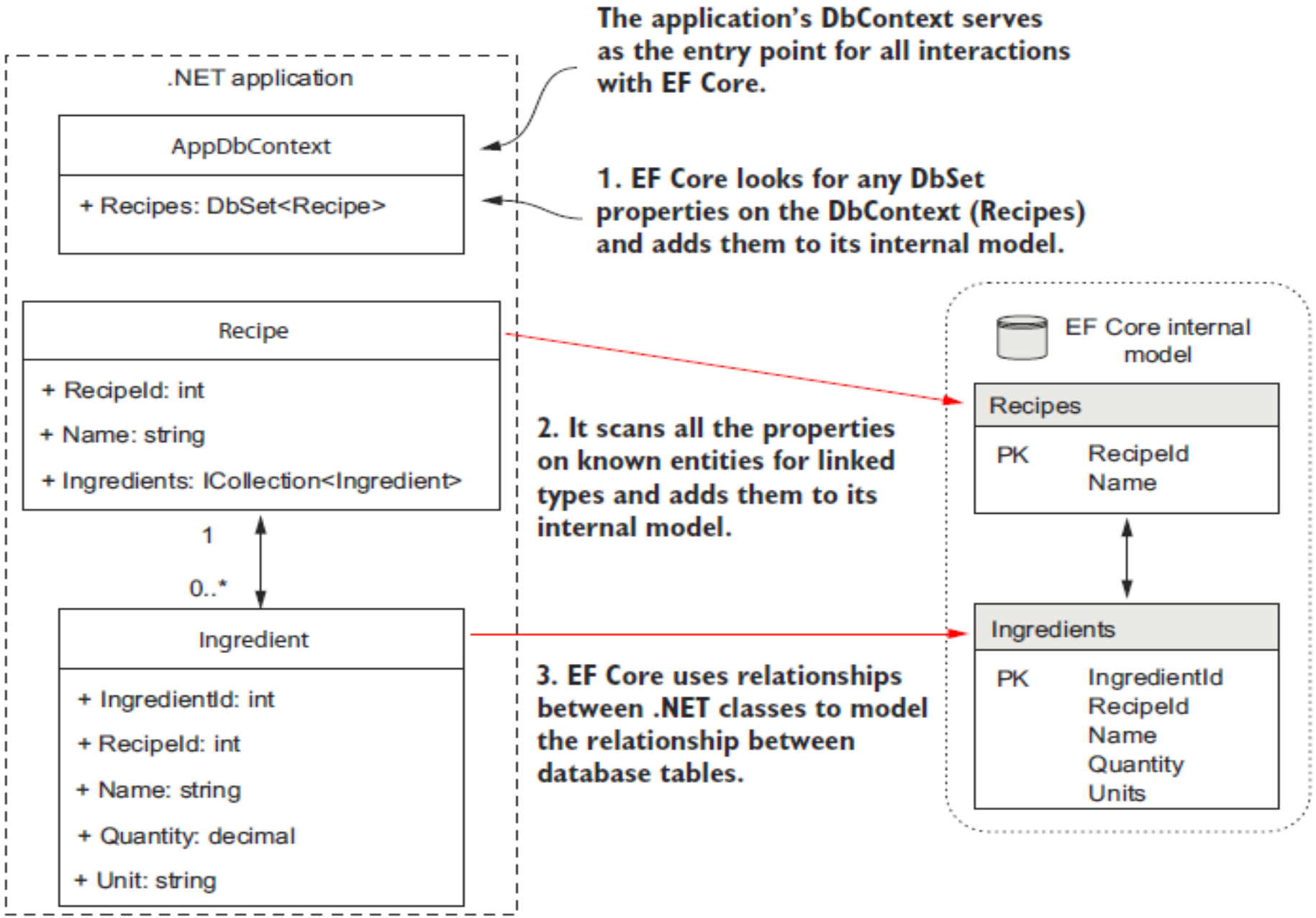
Creating ViewComponent

- ViewComponent Invoke() Return Values
  - IViewComponentResult
    - ViewViewComponentResult
    - ContentViewComponentResult
    - HtmlContentViewComponentResult

EF Core

- What Entity Framework Core is and why you should use it
- Adding Entity Framework Core to an ASP.NET Core application
- Building a data model and using it to create a database
- Querying, creating, and updating data using Entity Framework Core


.NET Application
Product
Class
Properties:
+ Id: int
+ Name: string
+ Price: decimal
+ Category: Category
Category
Class
Properties:
+ Id: int
+ Name: string
Product
.NET classes map to tables and properties map to columns.
References between types map to foreign key relationships between tables.
Each object (an instance of a class) maps to a row in a table.
Database
Products
PK Id
FK1 CategoryId
Name
Price
Categories
PK Id
Name
Products
Id CategoryId Name Price
1 12 Mac mini 479.00
2 45 iPad 339.00
3 123 Xbox One 280.00


The application's DbContext serves as the entry point for all interactions with EF Core.
.NET application
AppDbContext
+ Recipes: DbSet<Recipe>
1. EF Core looks for any DbSet properties on the DbContext (Recipes) and adds them to its internal model.
Recipe
+ RecipeId: int
+ Name: string
+ Ingredients: ICollection<Ingredient>
1
0..*
Ingredient
+ IngredientId: int
+ RecipeId: int
+ Name: string
+ Quantity: decimal
+ Unit: string
EF Core internal model
Recipes
PK RecipeId
Name
2. It scans all the properties on known entities for linked types and adds them to its internal model.
Ingredients
PK IngredientId
RecipeId
Name
Quantity
Units
3. EF Core uses relationships between .NET classes to model the relationship between database tables.

Adding EF Core to an application

- Choose a database provider; for example, MySQL, Postgres, or MS SQL Server.
- Install the EF Core NuGet packages.
- Design your app's DbContext and entities that make up your data model.
- Register your app's DbContext with the ASP.NET Core DI container.
- Use EF Core to generate a *migration* describing your data model.
- Apply the migration to the database to update the database's schema

Choosing a database provider and installing EF Core

- *PostgreSQL*—Npgsql.EntityFrameworkCore.PostgreSQL
- *Microsoft SQL Server*—Microsoft.EntityFrameworkCore.SqlServer
- *MySQL*—MySql.Data.EntityFrameworkCore
- *SQLite*—Microsoft.EntityFrameworkCore.SQLite
- More Providers
  - https://docs.microsoft.com/en-us/ ef/core/providers/.

Managing changes with migrations

```
EFDemo>dotnet ef --help
```

```
<ItemGroup>
  <DotNetCliToolReference
   Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools" Version="2.0.0" />
<DotNetCliToolReference
    Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.0" />
</ItemGroup>
```

```
EFDemo>dotnet ef migrations add InitialSchema
```

Migration file—A file with the Timestamp_MigrationName.cs format. This describes the actions to take on the database, such as Create table or Add column.

Migration designer.cs *file*—This file describes EF Core's internal model of your data model at the point in time the migration was generated

AppDbContextModelSnapshot.cs—This describes EF Core's *current* internal model. This will be updated when you add another migration, so it should always be the same as the current, latest migration

How to Apply Migration

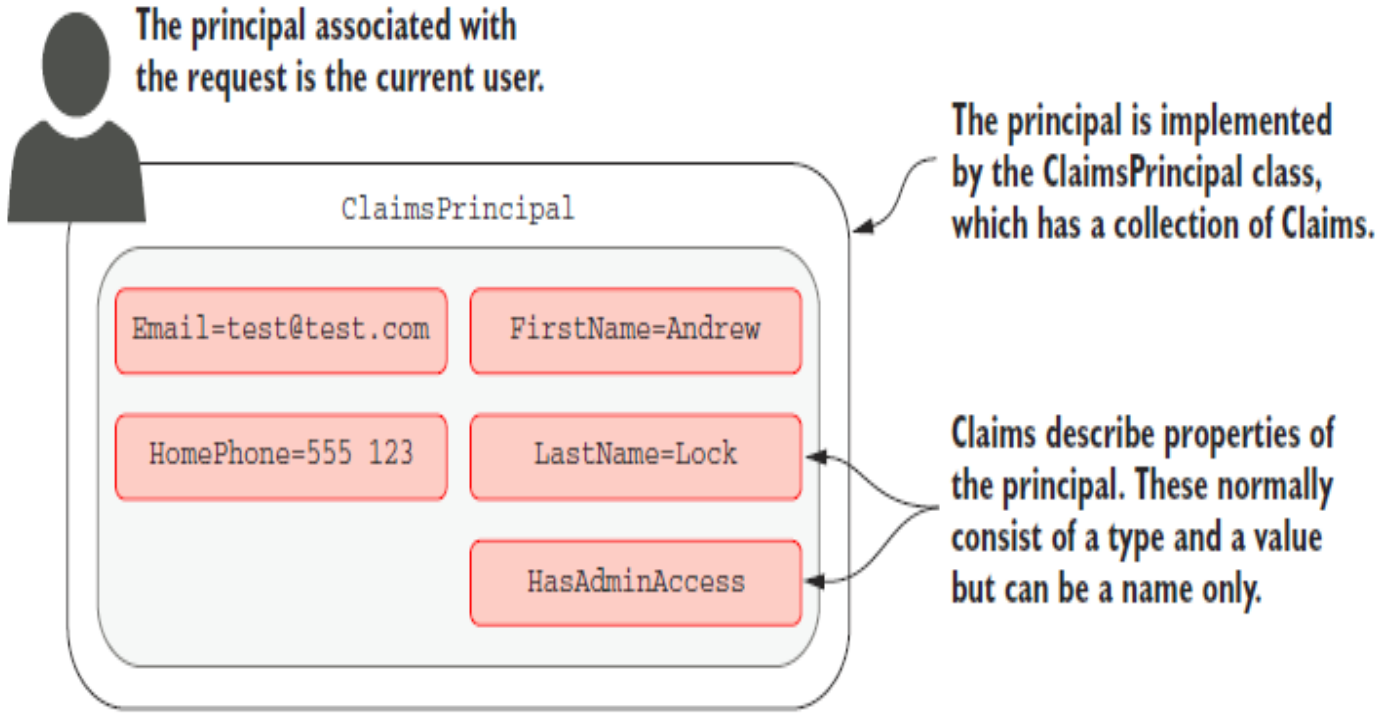
- Using the .NET CLI
  - **dotnet ef database update**
- Using the Visual Studio PowerShell cmdlets
- In code, by obtaining an instance of your AppDbContext and calling contextInstance.Database.Migrate().

Asp.Net Identity Management

- ASP.NET Core Identity is an API for managing users and storing user data in repositories such as relational databases through Entity Framework Core
- Identity is used through services and middleware added to the Startupclass and through classes that act as bridges between the application and the Identity functionality.
- Packages
  - Microsoft.EntityFrameworkCore.Tools.DotNet

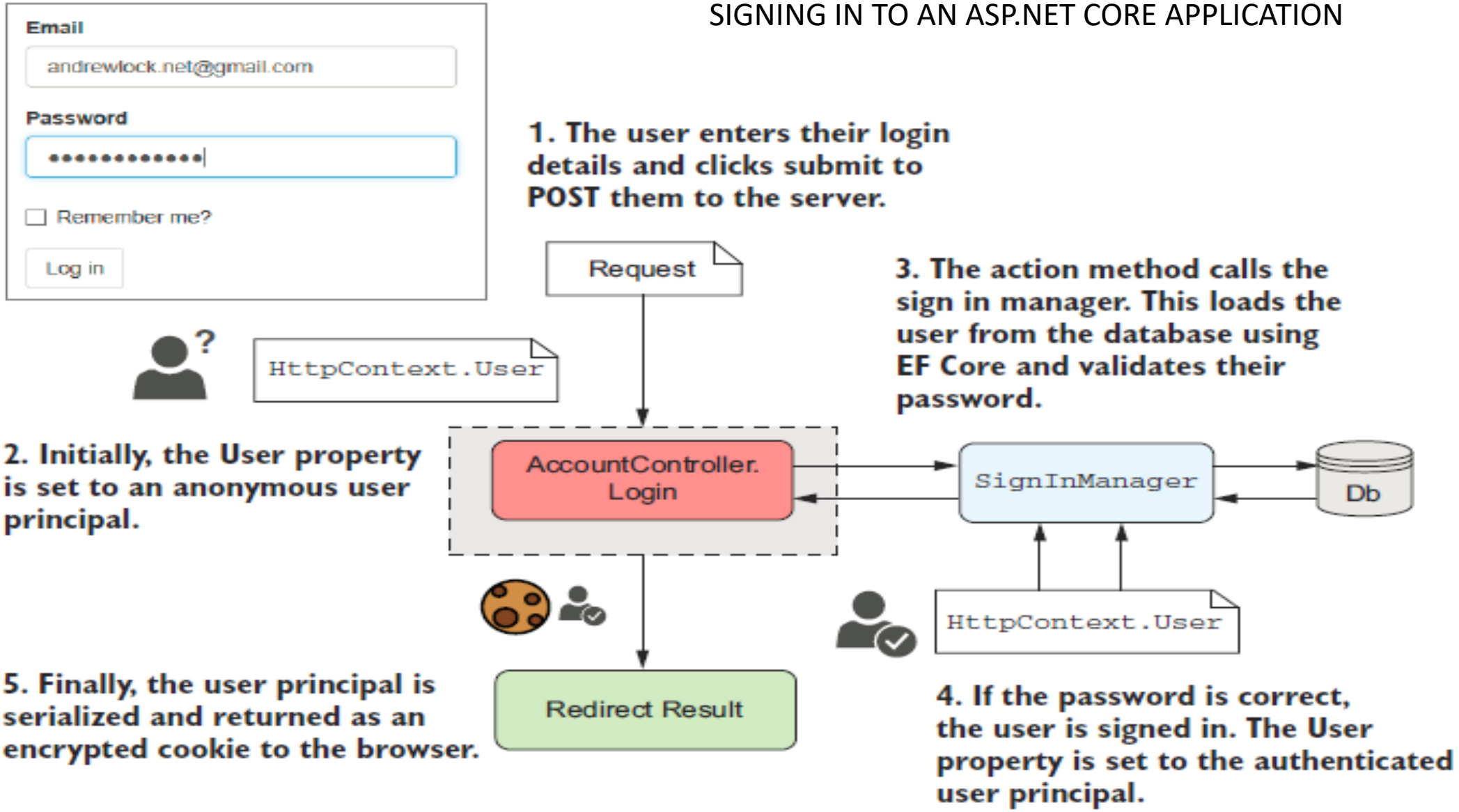
- *Claims*
  - A claim is a single piece of information about a principal, which consists of a *claim type* and an optional *value*

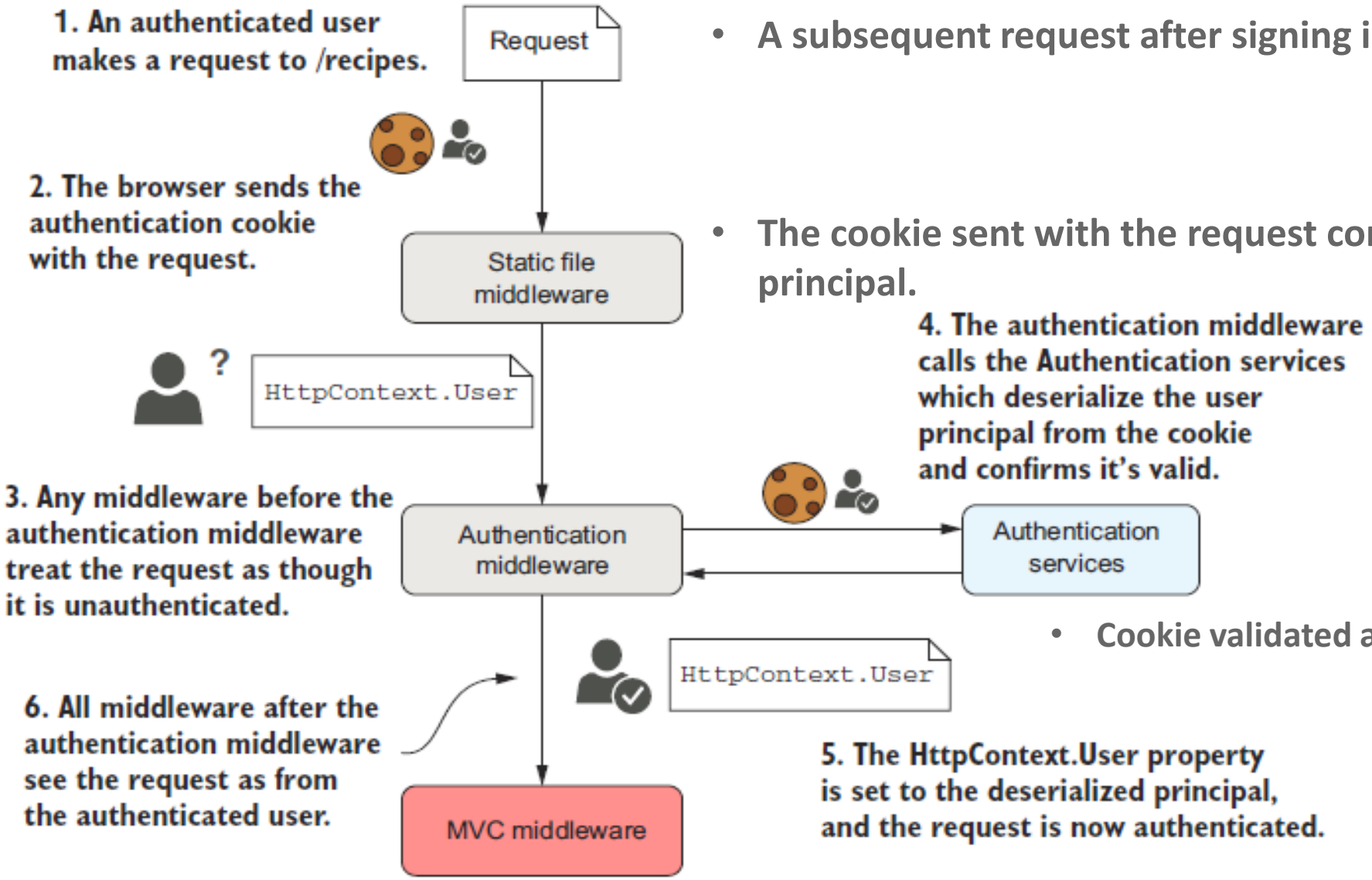
Identity Building Blocks



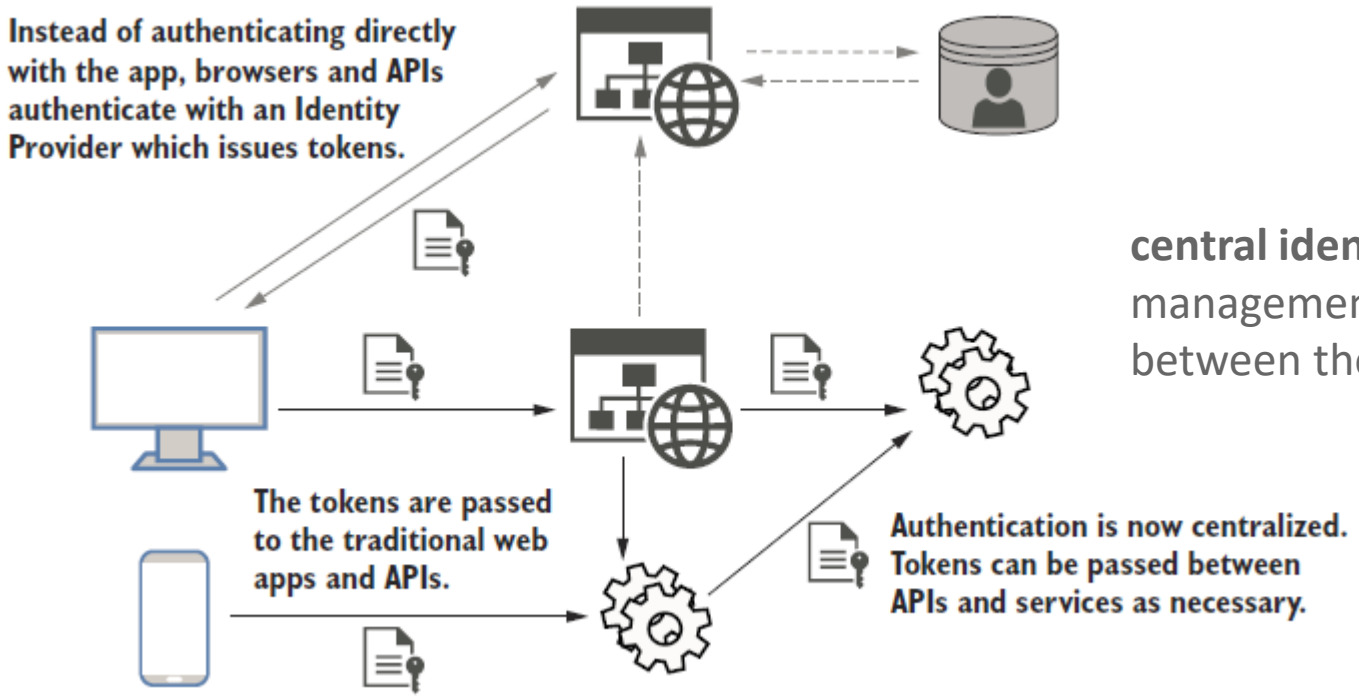
- ClaimsPrincipals
  - Current User – HttpContext.Use
  - Associated With collection of Cl…

SIGNING IN TO AN ASP.NET CORE APPLICATION

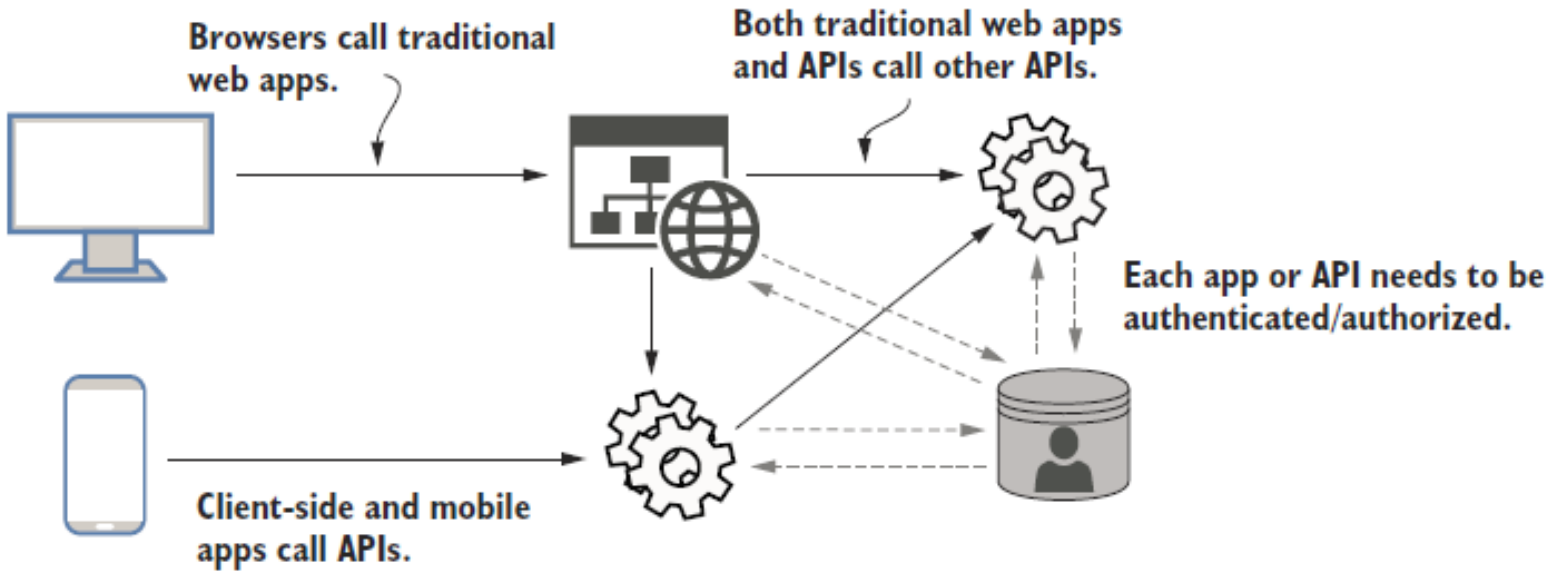




- A subsequent request after signing in to an application.
- The cookie sent with the request contains the user principal.
- Cookie validated and used to authenticate the request.



central identity provider to handle all the authentication and user management for the system. Tokens are passed back and forth between the identity provider, apps, and APIs.



What is ASP.NET Core Identity?

Managed by ASP.NET Core Identity	Implemented by the developer
Database schema for storing users and claims.	UI for logging in, creating, and managing users (controller, actions, view models). This is included in the default templates.
Creating a user in the database.	Sending emails and SMS messages.
Password validation and rules.	Customizing claims for users (adding new claims).
Handling user account lockout (to prevent brute-force attacks).	Configuring third-party identity providers.
Managing and generating 2FA codes.	
Generating password-reset tokens.	
Saving additional claims to the database.	
Managing third-party identity providers (for example Facebook, Google, Twitter).	

Adding ASP.NET Core Identity services to ConfigureServices

ASP.NET Core Identity uses EF Core, so it includes the standard EF Core configuration.

Adds the Identity system, and configures the user and role types

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    services.AddTransient<IEmailSender, AuthMessageSender>();
    services.AddTransient<ISmsSender, AuthMessageSender>();

    services.AddMvc();
}
```

Configures Identity to store its data in EF Core

Uses the default Identity providers for generating 2FA codes

Registers the stub service for sending SMS; may be missing in some versions of Visual Studio

Registers the stub service for sending email

Setup Identity Management

- Creating the User Class
  - Derive From IdentityUser
- Create DBContext object
  - Derive From IdentityDbContext<UserClass>
- Add DBContext to Services
- Set up the services for ASP.NET Core Identity

```
...
services.AddDbContext<AppIdentityDbContext>(options =>
    options.UseSqlServer(Configuration["Data:SportStoreIdentity:ConnectionString"]));
...
```

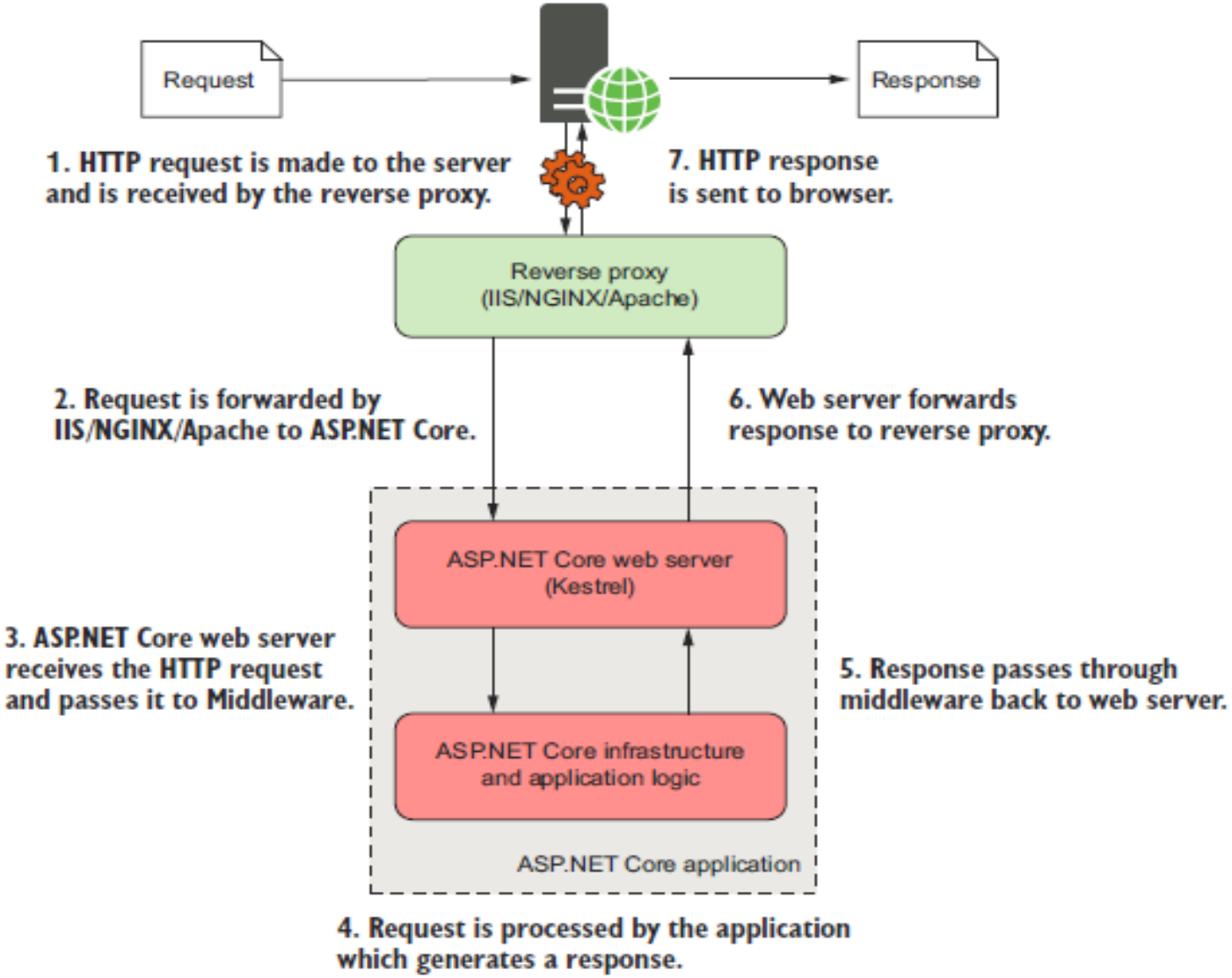
```
...
services.AddIdentity<AppUser, IdentityRole>()
    .AddEntityFrameworkStores<AppIdentityDbContext>()
    .AddDefaultTokenProviders();
...
```

```
...
app.UseAuthentication();
...
```

Identity Management Api

- UserManager<UserClassName>
- Inject to Controller
- IdentityResult

Hosting Model ASP.Net MVC Core App

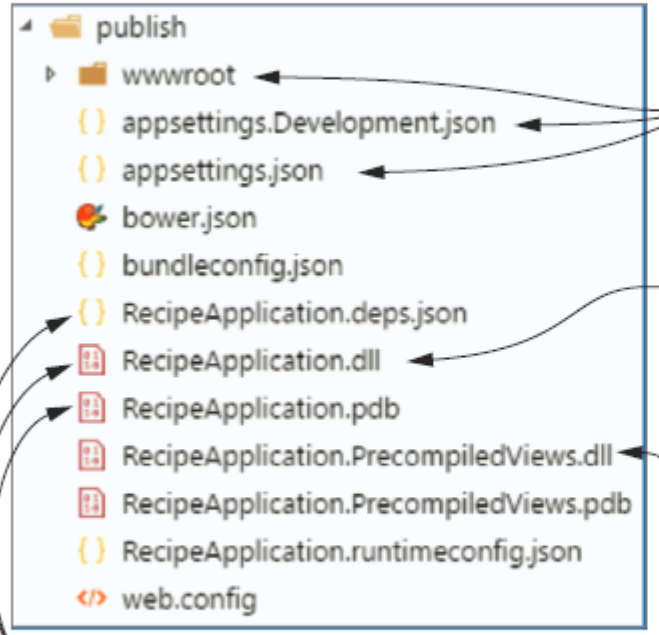


Reverse Proxy Benefits:

- *Security*—Reverse proxies are specifically designed to be exposed to malicious internet traffic, so they're typically well-tested and battle-hardened.
- *Performance*—You can configure reverse proxies to provide performance improvements by aggressively caching responses to requests.
- *Process management*—An unfortunate reality is that apps sometimes crash. Some reverse proxies can act as a monitor/scheduler to ensure that if an app crashes, the proxy can automatically restart it.
- *Support for multiple apps*—It's common to have multiple apps running on a single server. Using a reverse proxy makes it easier to support this scenario by using the host name of a request to decide which app should receive the request.

Publishing an ASP.NET Core app

The published output includes many additional files compared to the bin folder.



The wwwroot folder and content files like appsettings.json are copied to the publish folder.

Your app code is still compiled into RecipeApplication.dll.

The app Razor views are compiled into RecipeApplication.PrecompileViews.dll when publishing the app.

The same files from the bin folder are also copied to the publish folder.

```
dotnet Application.dll
```

```
dotnet publish --output publish --configuration release
```

Deployment Types

- Framework-dependent deployments
  - Relies on the .NET Core runtime being installed on the target machine that runs your published app
- Self-contained deployments
  - The target machine doesn't need to have .NET Core installed
  - SCD contains *all* of the code required to run your app
  - Deployment pack will contain .Net Core runtime with app's code and libraries

```
<RuntimeIdentifiers>win10-x64;ubuntu.16.04-x64</RuntimeIdentifiers>
```

```
C:\>dotnet publish -c Release -r win10-x64 -o publish_win10
```

Publishing app to IIS

- Configuring IIS for ASP.NET Core
  - Install the .NET Core Windows Server Hosting Bundle
  - https://go.microsoft.com/fwlink/?linkid=848766.
    - *The .NET Core Runtime*—Runs your .NET Core application
    - *The .NET Core libraries*—Used by your .NET Core apps
    - *The IIS AspNetCore Module*—Provides the link between IIS and your app, so that IIS can act as a reverse proxy
- Configure an *application pool*
  - *Use "No Managed Code* pool"

The ASP.NET Core logging abstractions

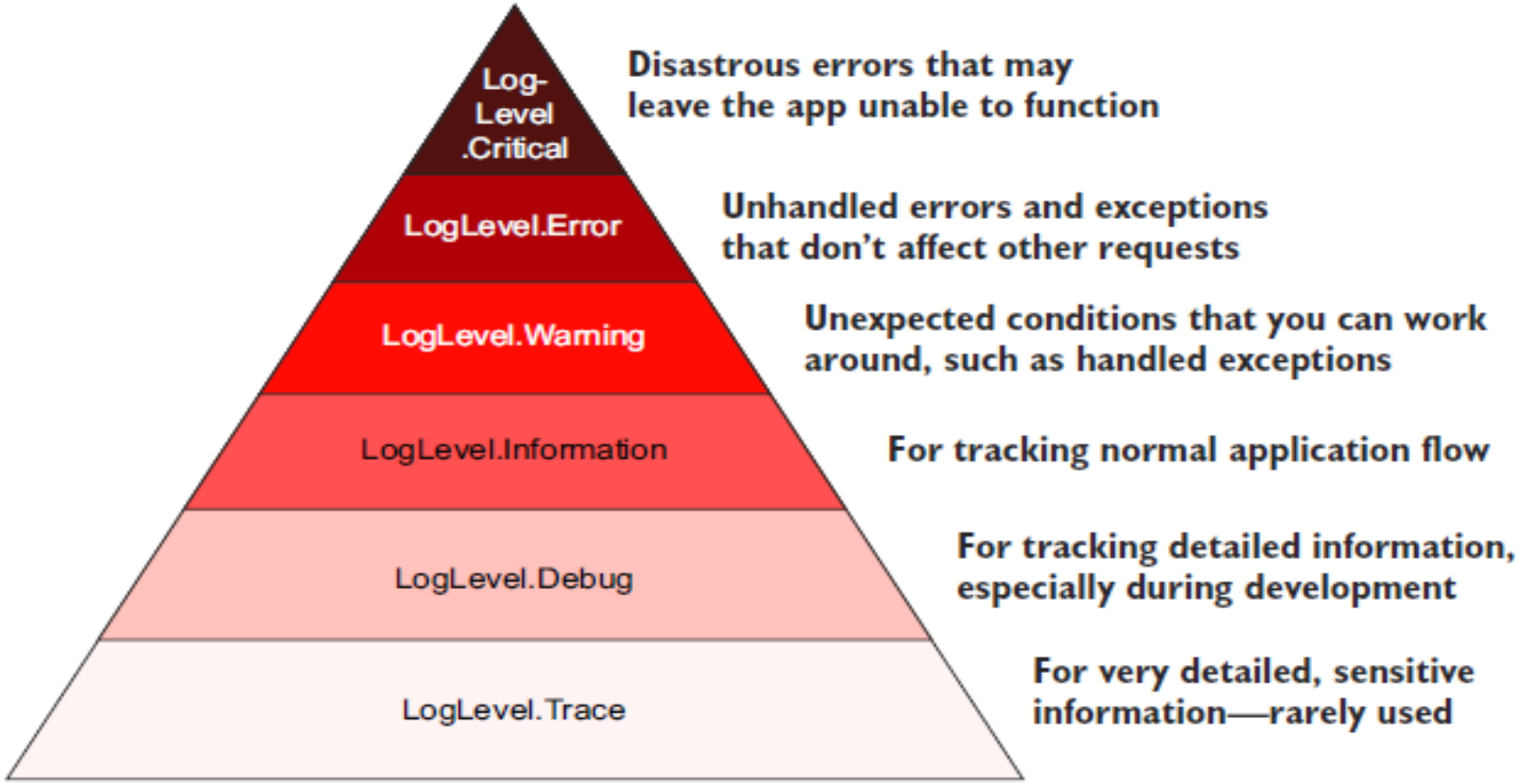
- ILogger
  - This is the interface you'll interact with in your code. It has a Log() method, which is used to write a log message.
- ILoggerProvider—This is used to create a custom instance of an I Logger, depending on the provider
- ILoggerFactory—The glue between the ILoggerProvider instances and the ILogger you use in your code.

Example

```
public static IWebHost BuildWebHost(string[] args) =>
    new WebHostBuilder()
        .UseStartup<Startup>()
        .ConfigureLogging(builder =>builder.AddConsole())
        .Build();
}
```

Add new providers with the ConfigureLogging extension method on WebHostBuilder.

Pyramid of Log Levels



Logging Providers

- *Console provider*—Writes messages to the console.
- *Debug provider*—Writes messages to the debug window when you're debugging
- an app in Visual Studio or Visual Studio Code.
- *EventLog provider*—Writes messages to the Windows Event Log. Only available when targeting .NET Framework, not .NET Core, as it requires Windowsspecific APIs.
- *EventSource provider*—Writes messages using Event Tracing for Windows (ETW). Available in .NET Core, but only creates logs when running on Windows.
- *Azure App Service provider*—Writes messages to text files or blob storage if you're running your app in Microsoft's Azure cloud service
- Third Party Loggers
  - Nlog
  - SeriLog
  - Loggr
  - NetEscapades.Extensions.Logging (File Logger)

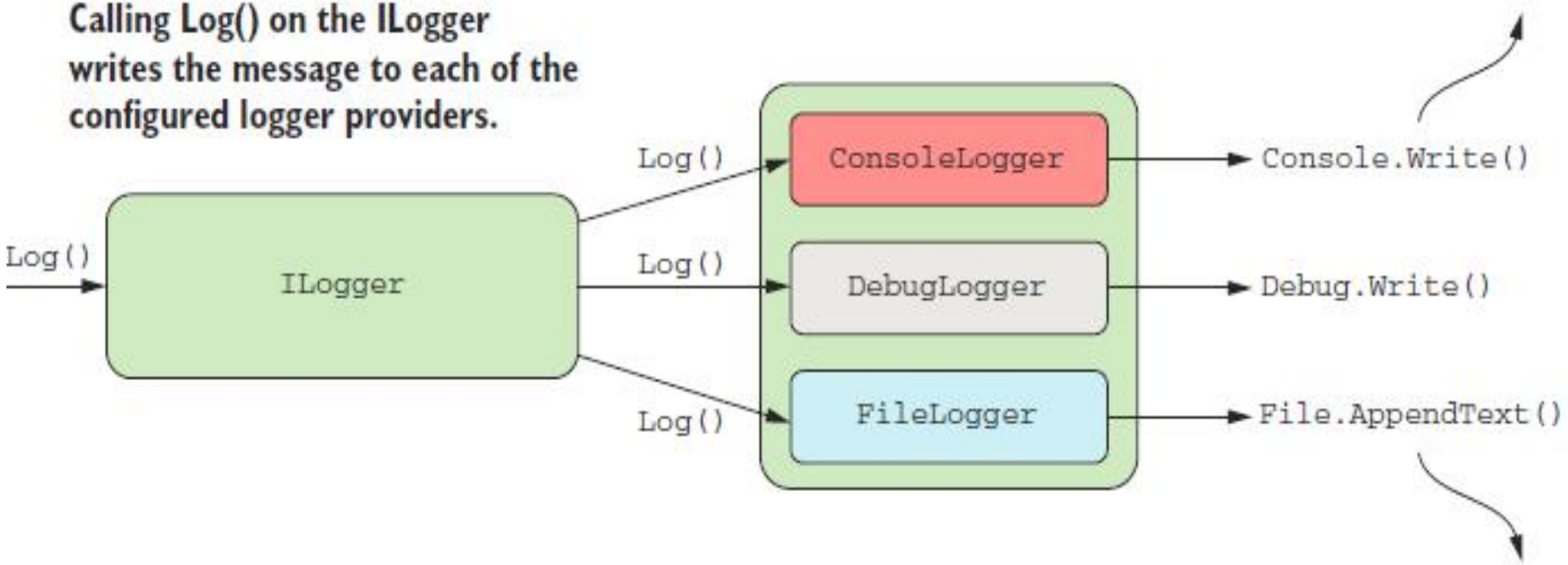
Adding a third-party logging provider to **WebHostBuilder**

```
public static void Main(string[] args)
{
    BuildWebHost(args).Run();
}

public static IWebHost BuildWebHost(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .ConfigureLogging(builder => builder.AddFile())
        .UseStartup<Startup>()
        .Build();
```

The CreateDefaultBuilder method configures the console and debug providers as normal.

Adds the new file logging provider to the logger factory

Calling Log() on the ILogger writes the message to each of the configured logger providers.



logs-20170904.txt - Notepad

File Edit Format View Help

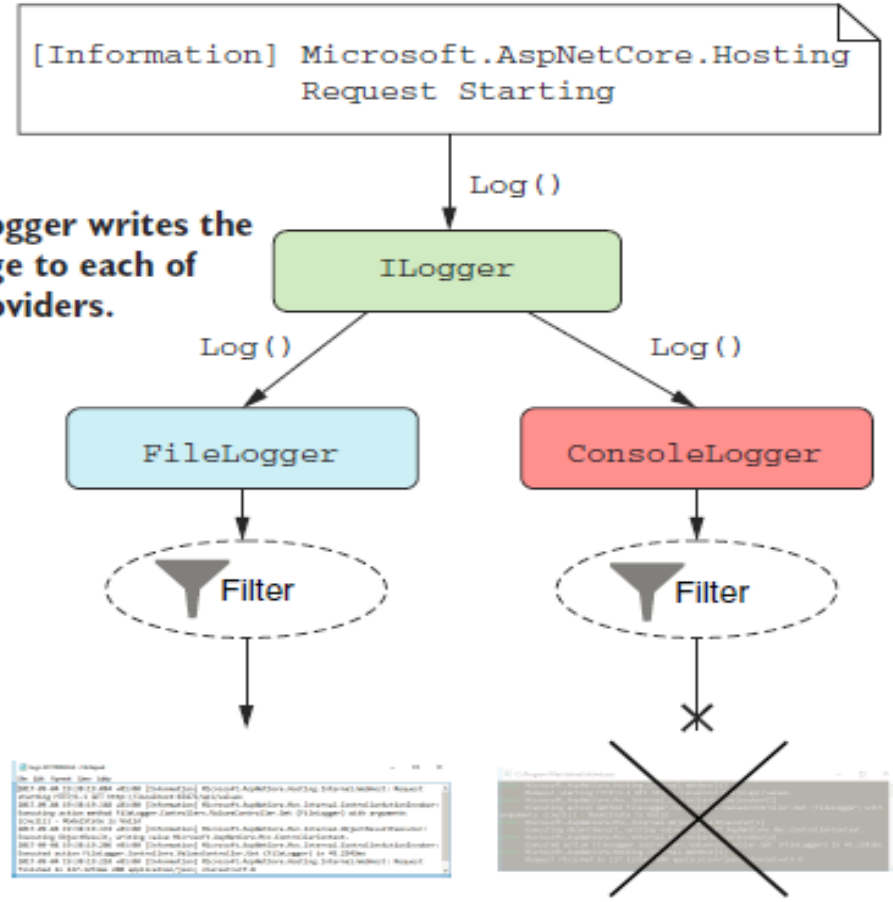
```
2017-09-04 19:38:19.084 +01:00 [Information] Microsoft.AspNetCore.Hosting.Internal.WebHost: Request
starting HTTP/1.1 GET http://localhost:65472/api/values
2017-09-04 19:38:19.168 +01:00 [Information] Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker:
Executing action method FileLogger.Controllers.ValuesController.Get (FileLogger) with arguments
((null)) - ModelState is Valid
2017-09-04 19:38:19.174 +01:00 [Information] Microsoft.AspNetCore.Mvc.Internal.ObjectResultExecutor:
Executing ObjectResult, writing value Microsoft.AspNetCore.Mvc.ControllerContext.
2017-09-04 19:38:19.206 +01:00 [Information] Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker:
Executed action FileLogger.Controllers.ValuesController.Get (FileLogger) in 45.2242ms
2017-09-04 19:38:19.210 +01:00 [Information] Microsoft.AspNetCore.Hosting.Internal.WebHost: Request
finished in 127.1231ms 200 application/json; charset=utf-8
```

Changing log verbosity with filtering

The application writes a log message to the ILogger with a category, level, and message.

The log message is compared to the filtering rules based on the provider, category, and log level.

	Provider	Category	Min level
1.	File		Information
2.		System	Warning
3.	Console	Microsoft	Warning
4.	Console		Debug
5.		Microsoft	Warning
6.			Debug



The ILogger writes the message to each of the providers.

The provider matches rule 1, and the log level exceeds the minimum log level, so the log is written.

The provider and category match rule 3, but the log level is less than the minimum log level, so the log is discarded.

Loading logging configuration in ConfigureLogging

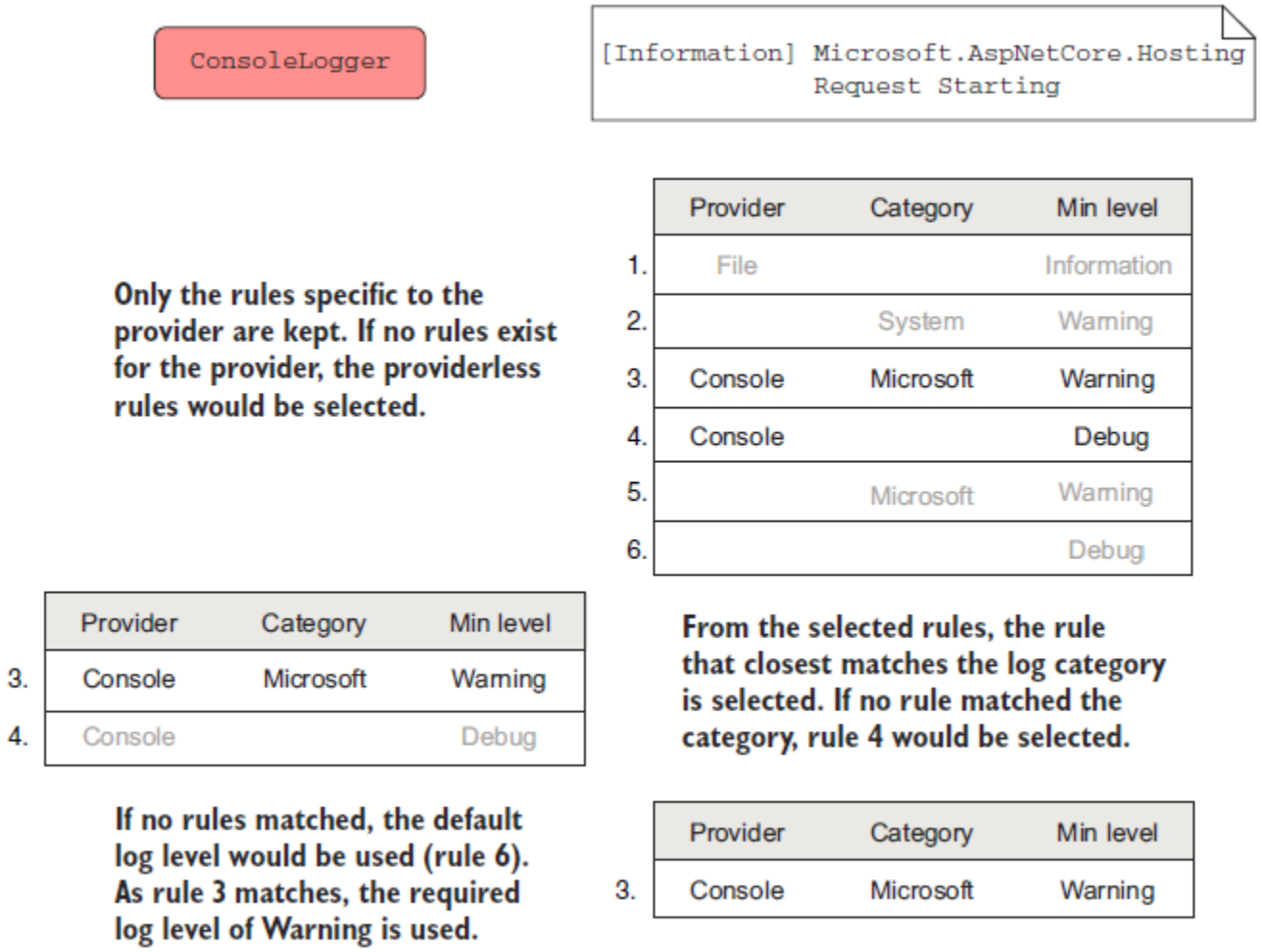
```
public class Program
{
    public static void Main(string[] args)
    {
        BuildWebHost(args).Run();
    }

    public static IWebHost BuildWebHost(string[] args) =>
        new WebHostBuilder()
            .UseKestrel()
            .UseContentRoot(Directory.GetCurrentDirectory())
            .ConfigureAppConfiguration(config =>
                config.AddJsonFile("appsettings.json"))
            .ConfigureLogging((ctx, builder) =>
            {
                builder.AddConfiguration(
                    ctx.Configuration.GetSection("Logging"));
                builder.AddConsole();
            })
            .UseStartup<Startup>()
            .Build();
}
```

Loads configuration values from appsettings.json

Loads the log filtering configuration from the Logging section and adds to ILoggingBuilder

Adds a console provider to the app

Logging Rules Evaluation



	Provider	Category	Min level
1.	File		Information
2.		System	Warning
3.	Console	Microsoft	Warning
4.	Console		Debug
5.		Microsoft	Warning
6.			Debug

	Provider	Category	Min level
3.	Console	Microsoft	Warning
4.	Console		Debug

	Provider	Category	Min level
3.	Console	Microsoft	Warning

Adding Dynamic Content - View

Inline code	Use for small, self-contained pieces of view logic, such as if and foreach statements
Tag helpers	Used to generate attributes on HTML elements
Sections	Use for creating sections of content that will be inserted into layout at specific locations
Partial views	Use for sharing subsections of view markup between views. Partial views can contain inline code, HTML helper methods, and references to other partial views. Partial views do not invoke an action method, so they cannot be used to perform business logic.
View components	Use for creating reusable UI controls or widgets that need to contain business logic

View Components

- View components are classes that provide application logic to support partial views or to inject small fragments of HTML or JSON data into a parent view.
- Derived from the ViewComponent
- Rendered in a parent view using the @await Component.InvokeAsync expression
- Specialized Action used only to provide a partial view with data, independently from the parent view and the action that renders it
- ViewComponent cannot receive HTTP requests
- View component classes must be public, non-nested, and non-abstract classes.

Creating a View Component

- POCO View Components
- Deriving from the ViewComponent Base Class
- Creating Hybrid Controller/View Component Classes

IViewComponentResult

ViewViewComponentResult	This class is used to specify a Razor view, with optional view model data
ContentViewComponentResult	This class is used to specify a text result that will be safely encoded for inclusion in an HTML document
HtmlContentViewComponentResult	This class is used to specify a fragment of HTML that will be included in the HTML document without further encoding

Tag Helper

- Html Transformers

```
...
<button type="submit" bs-button-color="danger">Add</button>
...
```

will be transformed into this:

```
...
<button type="submit" class="btn btn-danger">Add</button>
...
```