

.NET Basics

Session #01

dotnet training

Agenda

.NET Basics

Async programming

ASP .NET Core

Dependency Injection

Minimal API

Async programming

TAP: Task Async Programmin Model

Async Programming

- .NET uses the TAP approach
- The code is written as a sequence of statements but the compiler perform the async model based on specific keyword
- Mainly usage of `async` and `await` keyword
- `async` marks a method that can be executed asynchronously
- `await` is the way you execute that method asynchronously
- By convention, async method name finished with `Async` keyword

Two types of async

Async Programming

- For I/O-bound code, you await an operation that returns a Task or Task<T> inside an async method
- For CPU-bound code, you await an operation that is started on a background thread with Task.Run method
- I/O-bound should be use when you need to “wait” for some data (database, external service, ...)
- CPU-bound should be use when you need to compute a complex algorithm that can be run ion parallel

I/O-bound async

Async Programming

- You have specific objects and methods to use when need to ask for some data
- For HTTP requests, use HttpClient object
- For files, JsonSerializer, StreamReader, StreamWriter
- For databases, most of all ORM libraries have async methods, like we'll see later on EntityFramework

Async method

Async Programming

```
public async Task<int> GetUrlContentLengthAsync()  
{  
    var client = new HttpClient();  
  
    Task<string> getStringTask =  
        client.GetStringAsync("https://learn.microsoft.com/dotnet");  
  
    DoIndependentWork();  
  
    string contents = await getStringTask;  
  
    return contents.Length;  
}  
  
void DoIndependentWork()  
{  
    Console.WriteLine("Working...");  
}
```

ASP .NET Core

ASP .NET Core

Overview

- Open-source and community-focused
- Ability to develop and run on Windows, macOS, and Linux
- Building web UI and web APIs
- Razor Pages is used for WebUI
- Blazor lets you use C# in the browser alongside JavaScript. Share server-side and client-side app logic all written with .NET

ASP .NET Core vs ASP .NET

ASP.NET Core	ASP.NET 4.x
Build for Windows, macOS, or Linux	Build for Windows
Razor Pages is the recommended approach to create a Web UI as of ASP.NET Core 2.x. See also MVC , Web API , and SignalR .	Use Web Forms , SignalR , MVC , Web API , WebHooks , or Web Pages
Multiple versions per machine	One version per machine
Develop with Visual Studio [↗] , Visual Studio for Mac [↗] , or Visual Studio Code [↗] using C# or F#	Develop with Visual Studio [↗] using C#, VB, or F#
Higher performance than ASP.NET 4.x	Good performance
Use .NET Core runtime	Use .NET Framework runtime

ASP .NET Core Projects

Razor component	razorcomponent	[C#]	Web/ASP.NET	3.0
Razor page	page	[C#]	Web/ASP.NET	2.0
MVC ViewImports	viewimports	[C#]	Web/ASP.NET	2.0
MVC ViewStart	viewstart	[C#]	Web/ASP.NET	2.0
Blazor server app	blazorserver	[C#]	Web/Blazor	3.0
Blazor WebAssembly app	blazorwasm	[C#]	Web/Blazor/WebAssembly	3.1.300
ASP.NET Core empty	web	[C#], F#	Web/Empty	1.0
ASP.NET Core web app (Model-View-Controller)	mvc	[C#], F#	Web/MVC	1.0
ASP.NET Core web app	webapp, razor	[C#]	Web/MVC/Razor Pages	2.2, 2.0
ASP.NET Core with Angular	angular	[C#]	Web/MVC/SPA	2.0
ASP.NET Core with React.js	react	[C#]	Web/MVC/SPA	2.0
ASP.NET Core with React.js and Redux	reactredux	[C#]	Web/MVC/SPA	2.0
Razor class library	razorclasslib	[C#]	Web/Razor/Library/Razor Class Library	2.1
ASP.NET Core web API	webapi	[C#], F#	Web/WebAPI	1.0
ASP.NET Core gRPC service	grpc	[C#]	Web/gRPC	3.0

App Start Up

ASP .NET Core

- Can use Controller-based or minimal API (more about this later)
- Dotnet CLI templates creates a Program.cs with basic code
- Use a `WebApplicationBuilder` that is initialised with default configuration
- Can be used to add additional configuration and/or services
- Then, create a `WebApplication` object that are the execution runner

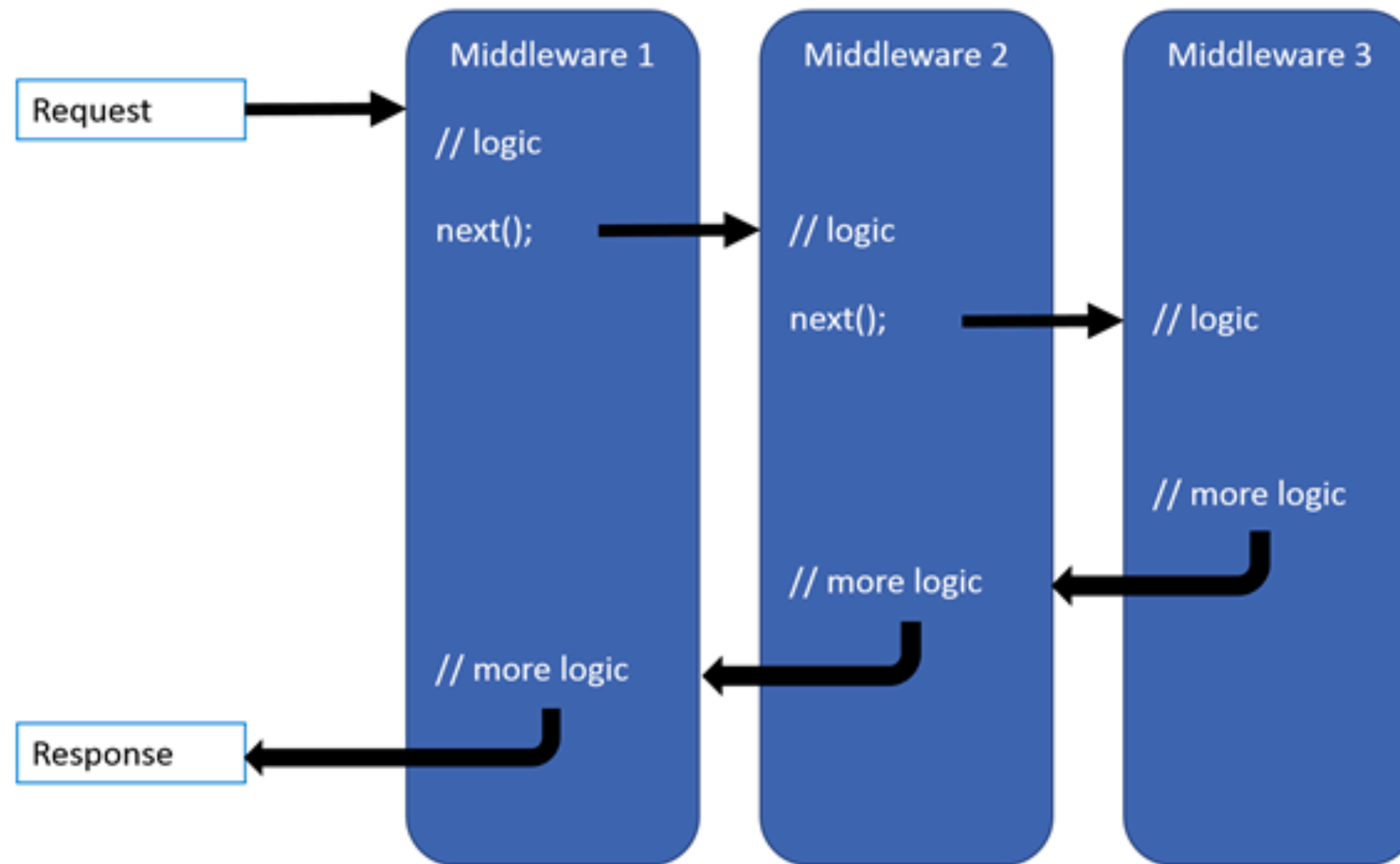
App Configuration Precedence

ASP .NET Core

- Command-line arguments
- Non-prefixed environment variables
- appsettings.{Environment}.json using the JSON configuration provider
- appsettings.json using the JSON configuration provider
- A fallback to the host configuration described in the next section (ASPNET_ prefixed environment variables)

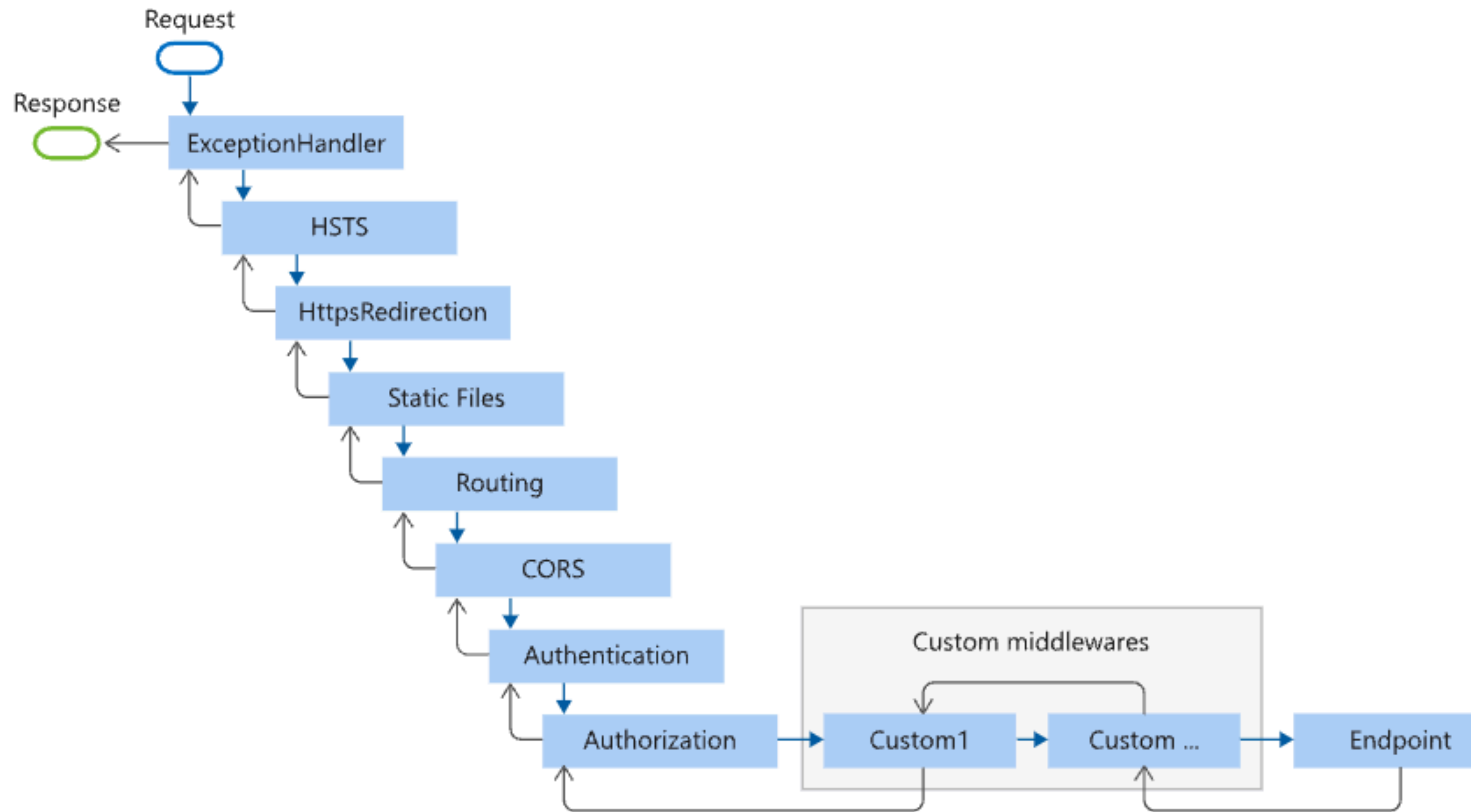
Middleware

ASP .NET Core



Middleware

ASP .NET Core



Dependency Injection

Overview

Dependency Injection

- Software design pattern, which is a technique for achieving Inversion of Control (IoC) between classes and their dependencies
- To handle dependencies directly, you may face several issues:
 - To replace dependency with a different implementation, several classes may be modified
 - If the dependency has dependencies, they must also be configured by the every class
 - This implementation is difficult to unit test

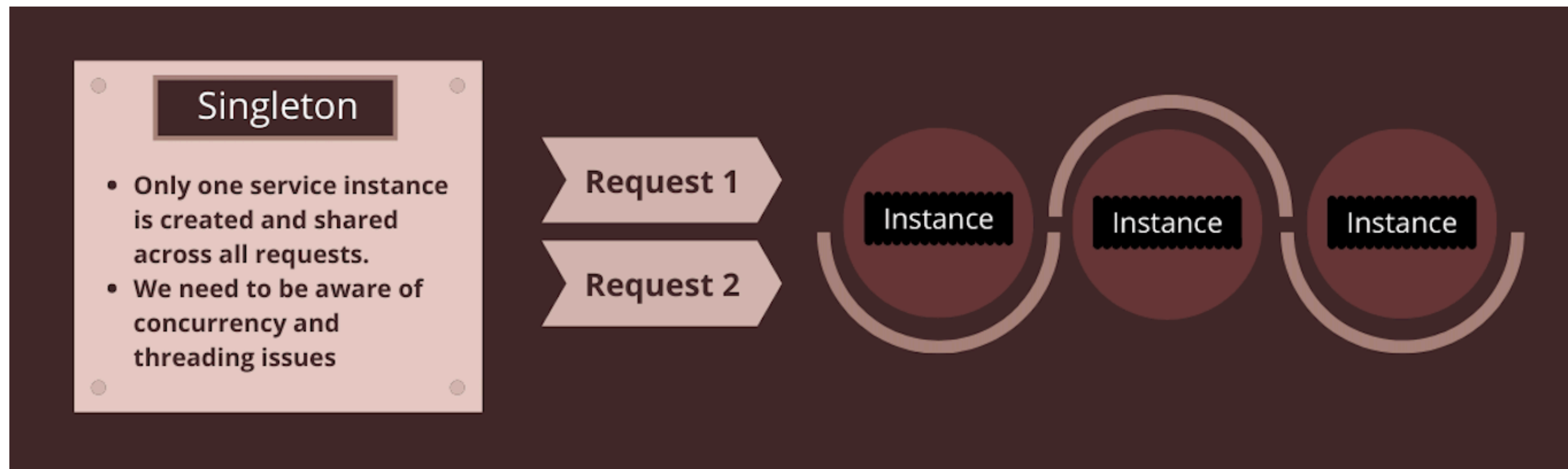
Overview

Dependency Injection

- Dependency injection addresses these problems through:
 - The use of an interface or base class to abstract the dependency implementation
 - Registration of the dependency in a service container
 - ASP.NET Core provides a built-in service container, `IServiceProvider`
 - Services are typically registered in the app's `Program.cs` file
 - Injection of the service into the constructor of the class where it's used
 - The framework takes on the responsibility of creating an instance of the dependency and disposing of it when it's no longer needed

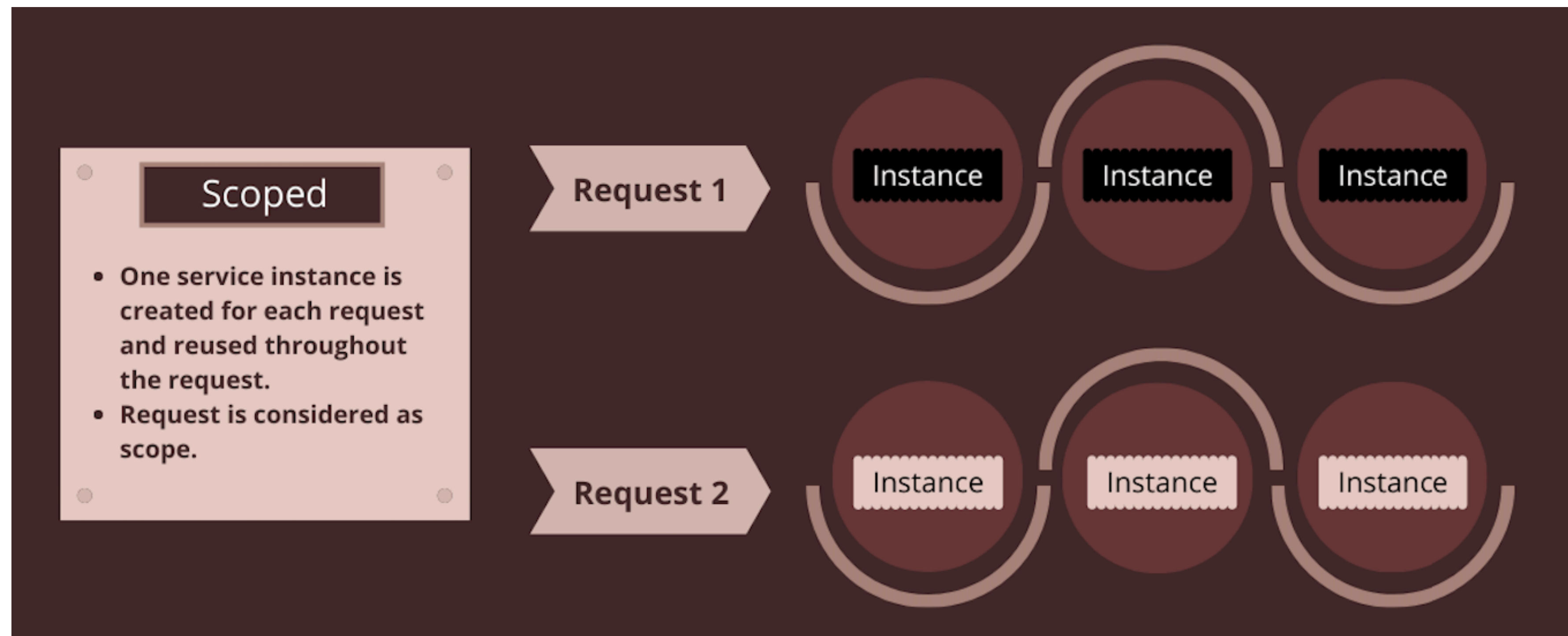
Service Lifetime: Singleton

Dependency Injection



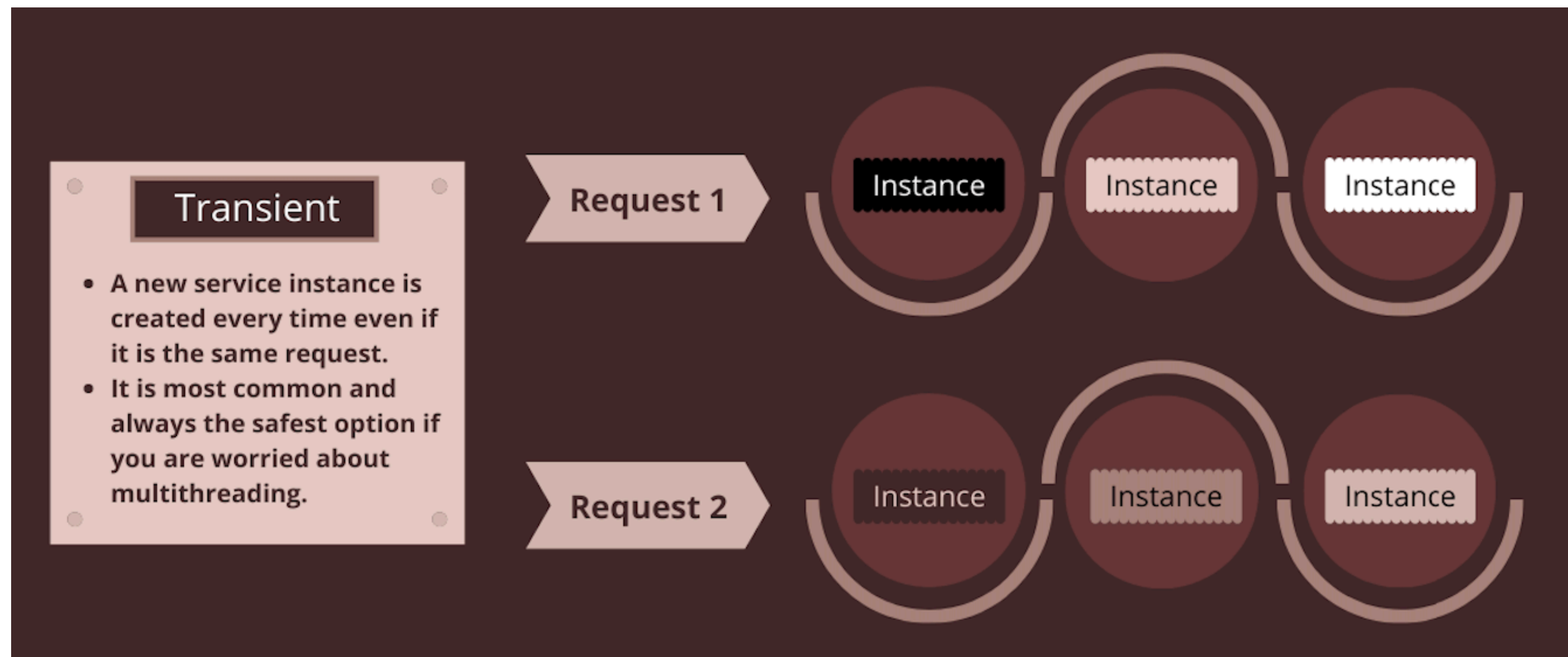
Service Lifetime: Scoped

Dependency Injection



Service Lifetime: Transient

Dependency Injection



DI and Extension Methods

Dependency Injection

- .NET brings default DI using Extension Methods
- Loggers can be created automatically using DI and Extension Methods
- Out of the box loggers
 - Console
 - Debug
 - EventSource
 - EventLog: Windows only
- Can enable automatically HTTPLogging, Swagger, etc.

Minimal API

Overview

Minimal API

- Minimal APIs are a simplified approach for building fast HTTP APIs with ASP.NET Core
- You can build fully functioning REST endpoints with minimal code and configuration
- Skip traditional scaffolding and avoid unnecessary controllers by fluently declaring API routes and actions
- Bring a more functional approach to dotnet code
- <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/minimal-apis?view=aspnetcore-7.0>

Overview

Minimal API



```
var app = WebApplication.Create(args);  
app.MapGet("/", () => "Hello World!");  
app.Run( );
```

Dotnet CLI

Minimal API

- Minimal API is not the default scaffold used by dotnet CLI
- If you want to use, you need to explicitly mention on dotnet CLI command



```
dotnet new webapi -o api-minimal -minimal
```

Route Handlers

Minimal API

- No need to define controllers
- More explicit and with additional feature instead of Controllers
- To make easier to manage, allow to define endpoint as groups or on external files/classes

Route Handlers

Minimal API



```
var builder = WebApplication.CreateBuilder(args);  
var app = builder.Build();
```

```
app.MapGet("/", () => "This is a GET");  
app.MapPost("/", () => "This is a POST");  
app.MapPut("/", () => "This is a PUT");  
app.MapDelete("/", () => "This is a DELETE");
```

```
app.Run();
```


Parameter Binding

Minimal API

- Process of converting request data into strongly typed parameters that are expressed by route handlers.
- A binding source determines where parameters are bound from
- Binding sources can be explicit or inferred based on HTTP method and parameter type
- Available bindings:
 - Route values
 - Query string
 - Header
 - Body (as JSON)
 - Services provided by dependency injection

Parameter Binding

Minimal API



```
app.MapGet("/{id}", ([FromRoute] int id,  
    [FromQuery(Name = "p")] int page,  
    [FromServices] Service service,  
    [FromHeader(Name = "Content-Type")] string contentType)  
    => {});
```

Open API

Minimal API

- Programming language-agnostic standard for documenting HTTP APIs
- This standard is supported in minimal APIs through a combination of built-in APIs and open-source libraries
- There are three key aspects to OpenAPI integration in an application:
 - Generating information about the endpoints in the app
 - Gathering the information into a format that matches the OpenAPI schema
 - Exposing the generated OpenAPI schema via a visual UI or a serialized file

Open API

Minimal API

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();

app.MapGet("/hello/{name}", (string name) =>
{
    return Results.Text($"Hello, {name}!");
})
.Produces<string>(StatusCodes.Status200OK)
.WithName("GetHello")
.WithOpenApi()
.WithTags("HelloGroup");

app.MapGet("/goodbye", () => "Goodbye, World!")
.ExcludeFromDescription();

app.Run();
```


Q&A

Build your first API

Lab #03

dotnet training

Build your first app

Lab #02

- Learning Objectives
 - Create and WebAPI
 - Use async programming
 - Use dependency injection
 - Learn about minimal API
 - Validation libraries to make your API robust
- MD Link: <https://github.com/tasb/dotnet-training/blob/main/labs/lab03.md>
- HTML Link: <https://tasb.github.io/dotnet-training/labs/lab03.html>