Module 3

Configure Host, Middleware and Services in ASP.NET Core

Module Overview

- WebApplication
- Configuring Middleware
- Configuring Services

Lesson 1: WebApplication

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

- **WebApplicationBuilder** is responsible for 4 main things:
  - Adding Configuration using builder.**Configuration**.
  - Adding Services using builder.**Services**
  - Configure Logging using builder.**Logging**
  - General **IHostBuilder** and **IWebHostBuilder** configuration

- `var app = builder.Build();`

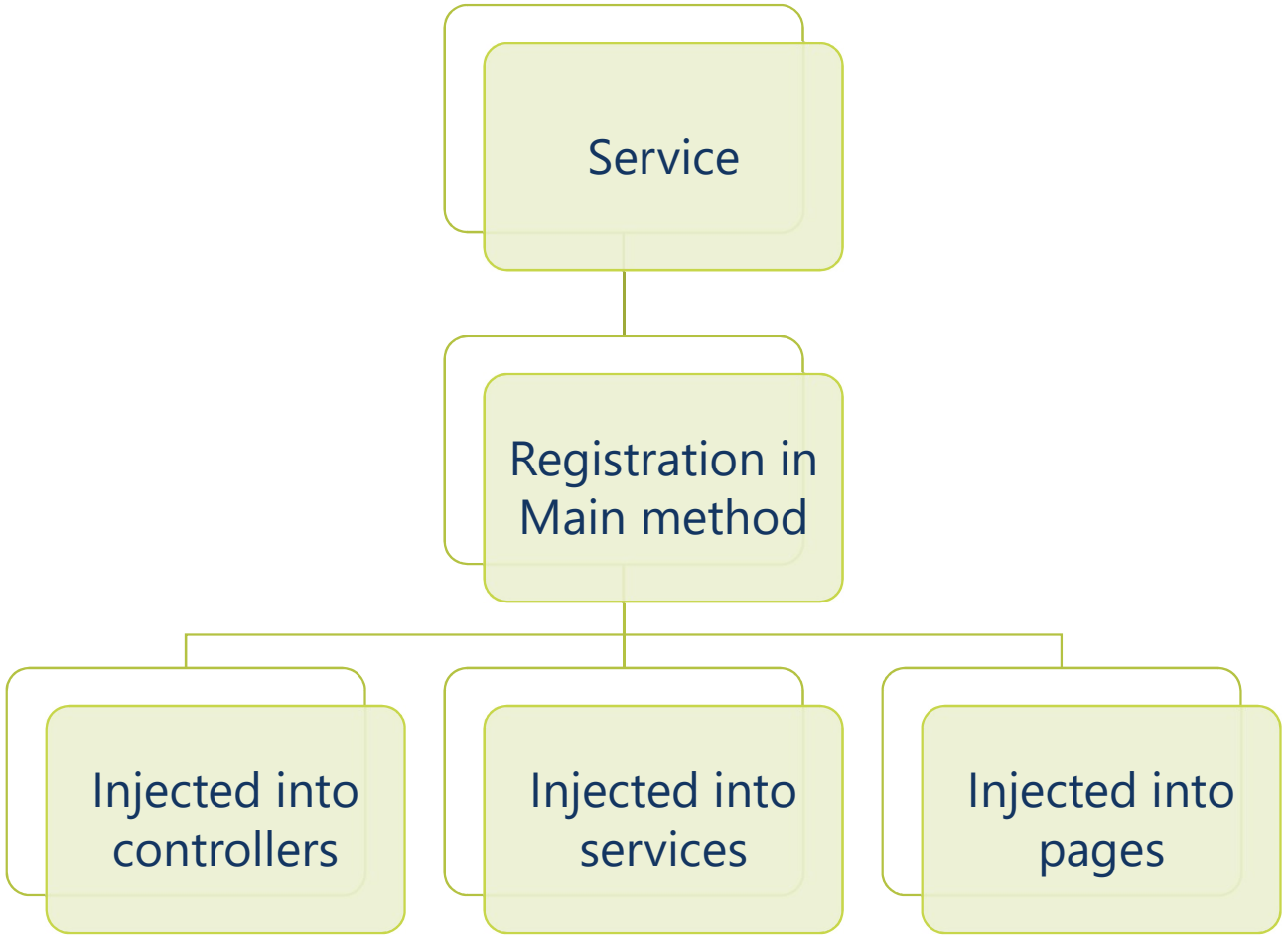
WebApplication implements multiple different interfaces:

- **IHost** - used to start and stop the host
- **IApplicationBuilder** - used to build the middleware pipeline
- **IEndpointRouteBuilder** - used to add endpoints

Lesson 2: Configuring Services

- Introduction to Dependency Injection
- Using builder to Configure Services
- Demonstration: How to Use Dependency Injection
- Inject Services to Pages
- Service Lifetime

Introduction to Dependency Injection



Using the builder to Configure Services

- Any class can act as a service
- By using the **Services** property of the **builder** object, you can register any service you would like to use in the application
- By using Dependency Injection you can utilize the services inside any other class

Inject Services to Pages

- A PageModel is used to handle requests from the client
- PageModel support constructor dependency injection
- If the internal behavior of a service or its dependencies change, you will not need to update the PageModel
- You cannot have more then one constructor in the PageModel, as the default dependency injection container cannot handle it

Service Lifetime

- AddSingleton – Instantiates once in the application's lifetime
- AddScoped – Instantiates once per request made to the server
- AddTransient – Instantiates every single time the service is injected

```
builder.Services.AddSingleton<IFirstService, FirstService>();
builder.Services.AddScoped<ISecondService, SecondService>();
builder.Services.AddTransient<IThirdService, ThirdService>();
```

Service Configuration and Injection

```
builder.Services.AddSingleton<IService, Service>();

public class MyRazorPage: PageModel {
	public MyRazorPage(IService service){
		service.DoSomething();
	}
}
```

Demonstration: How to Use Dependency Injection

In this demonstration, you will learn how to:

- Create a service in ASP.NET Core
- Register a service
- Inject a service by using Dependency Injection

Lesson 2: Configuring Middleware

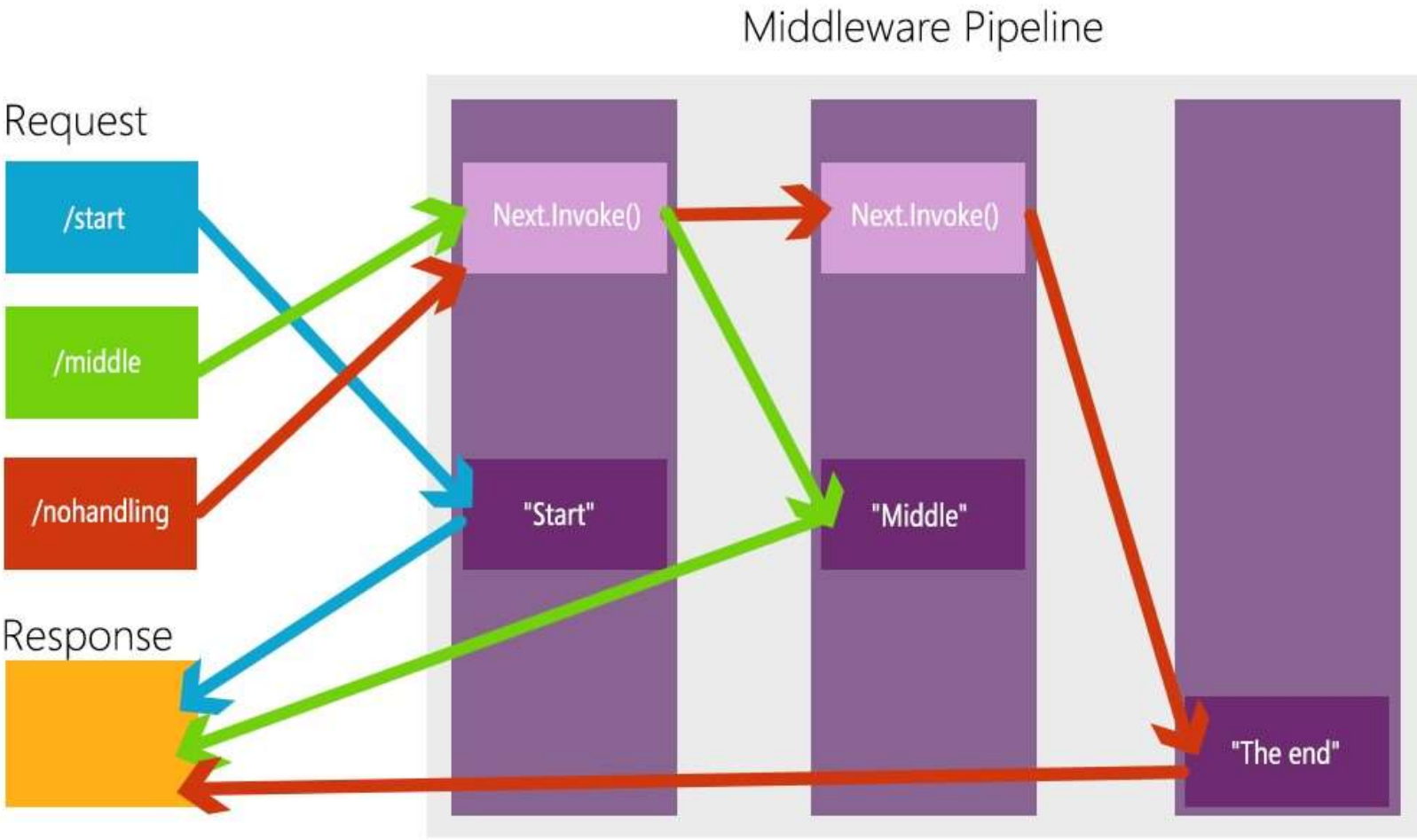
- Middleware Fundamentals
- Demonstration: How to Create Custom Middleware
- Working with Static Files
- Demonstration: How to Work with Static Files

Configure Method

WebApplication is used to set up middleware for our application

```
app.Run(async (context) =>
{
    await context.Response.WriteAsync("Hello World!");
});
```

Middleware Fundamentals



Run Middleware

- Always terminates the middleware pipeline
  - Middleware after it will never be run
- Should always provide a response for handling the final case

```
app.Run(async (context) =>
{
  await context.Response.WriteAsync("Inside run middleware");
});
```

Use Middleware

- Can call the next middleware in the chain by using the next parameter
- Can short circuit the pipeline chain by not calling for the next middleware
- The middleware most frequently used in the pipeline

```
app.Use(async (context, next) =>
{
    await context.Response.WriteAsync("Inside use middleware");
    await next.Invoke();
});
```

Map Middleware

- Allows us to create alternative behavior for specific paths
- Does not continue the main path
- Can be nested
- Does not do anything when used on its own
- Not frequently used

```
app.Map("/Map", (map) => {
 map.Run(async (context) => {
  await context.Response.WriteAsync("Run inside of map middleware");
  });
});
```

Demonstration: How to Create Custom Middleware

In this demonstration, you will learn:

- How to add custom middleware to the application
- How to use the middleware's context parameter
- How to use the Invoke method on the "next" parameter
- How the order of middleware affects their execution

Working with Static Files

- Static files are files that do not change at run time
- In ASP.NET Core applications static files can be served to clients by using the **UseStaticFiles** middleware

```
app.UseStaticFiles();
```

Common types of static files

Common types of static files:

- HTML files
- CSS stylesheets
- JavaScript files
- Images and other assets
- JSON or XML files

Demonstration: How to Work with Static Files

In this demonstration, you will learn:

- How to add HTML and image files as static files in the **wwwroot** folder
- How to serve static files by using the **UseStaticFiles** middleware
- What is the result of attempting to access a file that does not exist

Lab: Configuring Middleware and Services in ASP.NET Core

- Exercise 1: Working with Static Files
- Exercise 2: Creating Custom Middleware
- Exercise 3: Using Dependency Injection
- Exercise 4: Injecting a Service to a PageModel

Estimated Time: 75 Minutes

Lab Scenario

In this lab, we're going to create the initial structure of the PhotoSharing application.

Since we want all our parts to be testable and losely coupled, we're going to use the CLEAN architecture, making sure that each page can be dependent on abstractions and that each concern is neatly separated.

Lab Review

- What is the difference between app.Use and app.Run?
- What will change when you update the service from AddSingleton to AddScoped, or to AddTransient?
- What happens to the UseStaticFiles middleware when the browser is directed to a path where no static file is found?

Module Review and Takeaways

- Review Question
- Best Practice
- Common Issues and Troubleshooting Tips