

# ASP.NET, Azure, GitHub Actions, Terraform Training





## The world's largest provider of classroom and online training courses

- ✓ World Class Training Solutions
- ✓ Subject Matter Experts
- ✓ Highest Quality Training Material
- ✓ Accelerated Learning Techniques
- ✓ Project, Programme, and Change Management, ITIL® Consultancy
- ✓ Bespoke Tailor Made Training Solutions
- ✓ PRINCE2®, MSP®, ITIL®, Soft Skills, and More

# Course Syllabus

<b>Module 1:</b> Introduction to ASP.NET Core	5
<b>Module 2:</b> Implementing ASP.NET Core with React.js for Front-End Development	28
<b>Module 3:</b> Working with Web APIs in ASP.NET Core	71
<b>Module 4:</b> Authentication and Authorization in ASP.NET Core	94
<b>Module 5:</b> Entity Framework Core and Data Access	114
<b>Module 6:</b> Debugging and Troubleshooting in ASP.NET Core	123

# Course Syllabus

<b>Module 7:</b> Testing ASP.NET Core Applications	146
<b>Module 8:</b> Performance Optimisation in ASP.NET Core	174
<b>Module 9:</b> Containerised Deployment with Azure Container Apps	212
<b>Module 10:</b> ASP.NET Core Security Fundamentals	225
<b>Module 11:</b> Asynchronous Programming in ASP.NET Core	247
<b>Module 12:</b> Dependency Injection and Middleware in ASP.NET Core	268
<b>Module 13:</b> Building Microservices with ASP.NET Core	291

# Module 1: Introduction to ASP.NET Core

- Overview and Benefits of ASP.NET Core
- ASP.NET Core Application Lifecycle
- Setting Up Development Environment



# Overview and Benefits of ASP.NET Core

- ✓ ASP.NET Core is a modern, open-source, and cross-platform framework developed by Microsoft for building high-performance, scalable, and cloud-optimised web applications.
- ✓ It is designed to unify the development experience across various platforms, enabling developers to create web APIs, dynamic websites, and microservices efficiently.
- ✓ ASP.NET Core supports modular architecture, dependency injection, and improved middleware pipelines, making it highly flexible and customisable.
- ✓ The framework embraces a lightweight design that allows for faster startup times and reduced memory footprint.
- ✓ Additionally, it offers seamless integration with modern front-end frameworks and tools, enhancing full-stack development capabilities.

# Overview and Benefits of ASP.NET Core

(Continued)

- ✓ *The following are the key benefits of ASP.NET core:*

1. *Cross-Platform Compatibility*

2. *Performance and Scalability*

3. *Unified Development Model*

4. *Cloud-Ready Architecture*

5. *Open Source and Community Driven*

6. *Improved Security Features*

7. *Dependency Injection by Default*

# Overview and Benefits of ASP.NET Core

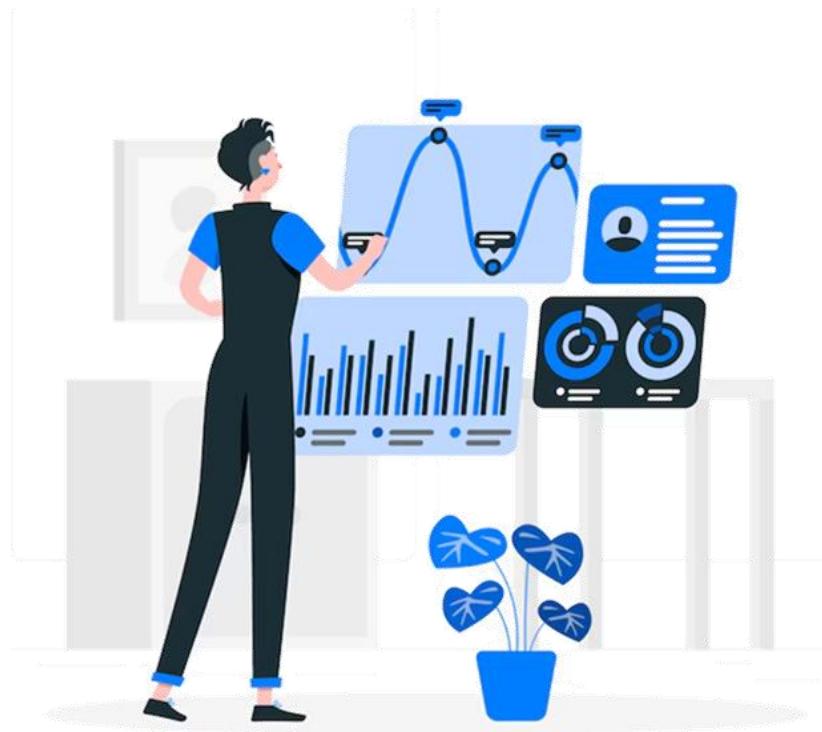
## 1. Cross-Platform Compatibility:

- Runs seamlessly on Windows, Linux, and macOS, allowing broader deployment options and cost-efficient infrastructure choices.
- This flexibility enables organisations to leverage existing infrastructure or adopt new environments without being locked into a single operating system.
- It also facilitates easier collaboration among development teams using diverse platforms.



# Overview and Benefits of ASP.NET Core

## 2. Performance and Scalability:



- ASP.NET Core delivers significant performance improvements compared to its predecessors due to lightweight and modular components, optimised runtime, and asynchronous programming support.
- Its efficient request processing pipeline and support for Kestrel web server allow applications to handle high loads with minimal resource consumption.
- This makes it ideal for developing scalable, enterprise-grade applications and microservices.

# Overview and Benefits of ASP.NET Core

## 3. Unified Development Model:

- Supports MVC (Model-View-Controller), Razor Pages, and API development under a single framework, simplifying the developer experience and reducing learning curves.
- This integration streamlines development workflows and encourages code reuse, helping teams deliver robust web applications faster while maintaining clean separation of concerns.

## 4. Cloud-Ready Architecture:

- Designed for seamless integration with cloud services such as Microsoft Azure, facilitating containerisation, microservices, and DevOps practices.
- Its built-in support for configuration, logging, and health monitoring simplifies deployment and management in cloud environments.
- The framework's compatibility with Docker and Kubernetes further accelerates continuous delivery and scaling.

# Overview and Benefits of ASP.NET Core

## 5. Open Source and Community Driven:

- Backed by a vibrant developer community and transparent development on GitHub, ensuring rapid innovation and extensive support.
- Open source contributions accelerate feature releases and bug fixes, while comprehensive documentation and community-driven extensions provide valuable resources for developers at all levels.

## 6. Improved Security Features:

- Includes built-in security mechanisms like data protection, authentication, authorisation, and HTTPS enforcement out of the box.
- The framework supports multiple authentication protocols such as OAuth, OpenID Connect, and JWT, enabling developers to implement secure, standards-compliant solutions. Regular updates address emerging threats, maintaining a secure application environment.

# Overview and Benefits of ASP.NET Core

## 7. Dependency Injection by Default:

- Built-in support for dependency injection encourages loosely coupled, testable, and maintainable code.
- This design pattern facilitates better separation of concerns, easier unit testing, and improved code flexibility.
- It enables developers to manage service lifetimes and dependencies consistently across the application without relying on third-party libraries.



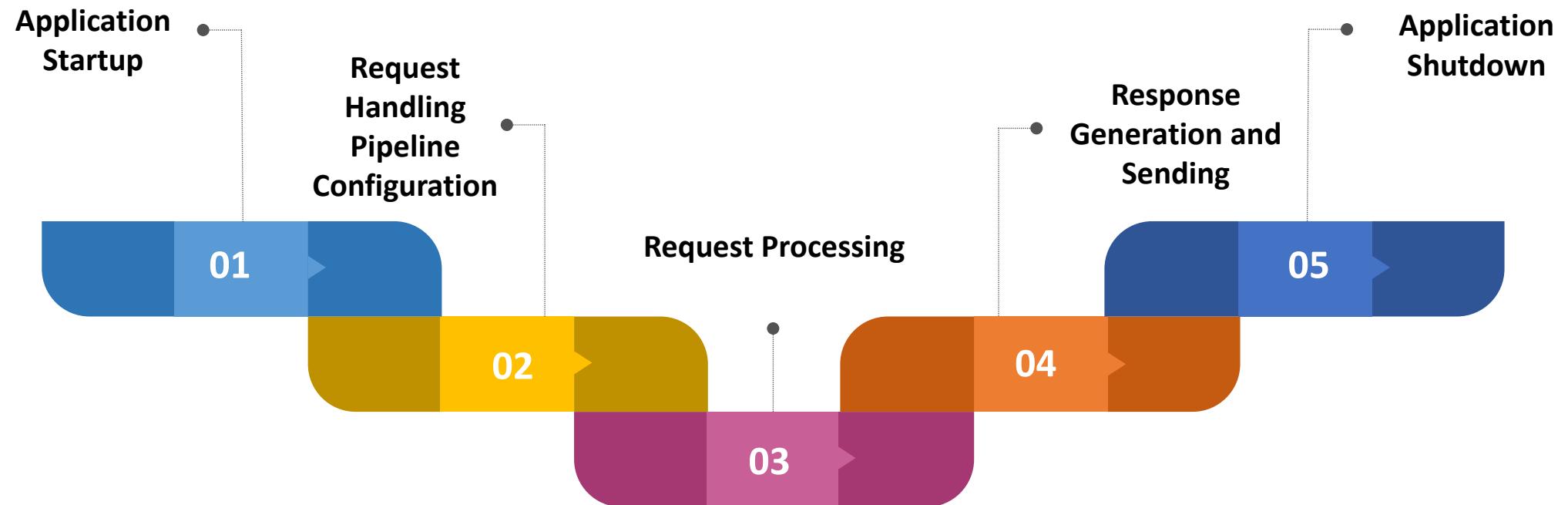
# ASP.NET Core Application Lifecycle

- ✓ The ASP.NET Core application lifecycle represents the sequence of stages an application undergoes from startup to shutdown.
- ✓ Understanding this lifecycle is crucial for developers to effectively manage application configuration, middleware setup, request processing, and graceful resource disposal.
- ✓ ASP.NET Core's pipeline-based model provides a flexible and modular approach to handle HTTP requests and responses through middleware components.
- ✓ This architecture promotes high performance and scalability by allowing precise control over how requests are handled.
- ✓ Additionally, it enables developers to easily extend and customise the application behaviour to meet diverse requirements.

# ASP.NET Core Application Lifecycle

(Continued)

- ✓ *The following are the key stages of the lifecycle:*



# ASP.NET Core Application Lifecycle

## 1. Application Startup:

- When the application starts, the runtime initialises the host, configures services, and builds the middleware pipeline.
- The Program.cs file typically contains the host builder that sets up the web server (Kestrel) and configures essential services like logging, configuration, and dependency injection.
- The Startup class (or equivalent) defines middleware components and service registrations.

## 2. Request Handling Pipeline Configuration:

- The middleware pipeline processes incoming HTTP requests sequentially. Each middleware component can inspect, modify, or terminate the request, as well as invoke the next component in the chain.
- This design allows for concerns like authentication, routing, exception handling, and response compression to be modularised and managed efficiently.

# ASP.NET Core Application Lifecycle

## 3. Request Processing:

- When a request arrives, it passes through the middleware pipeline to reach the endpoint (such as a controller action or Razor page).
- Middleware components perform tasks such as authentication checks, routing decisions, and model binding. The endpoint executes business logic and generates the response.

## 4. Response Generation and Sending:

- After the endpoint processes the request, the response travels back through the middleware pipeline, allowing components to modify headers, compress content, or perform logging before the response is sent to the client.

# ASP.NET Core Application Lifecycle

## 5. Application Shutdown:

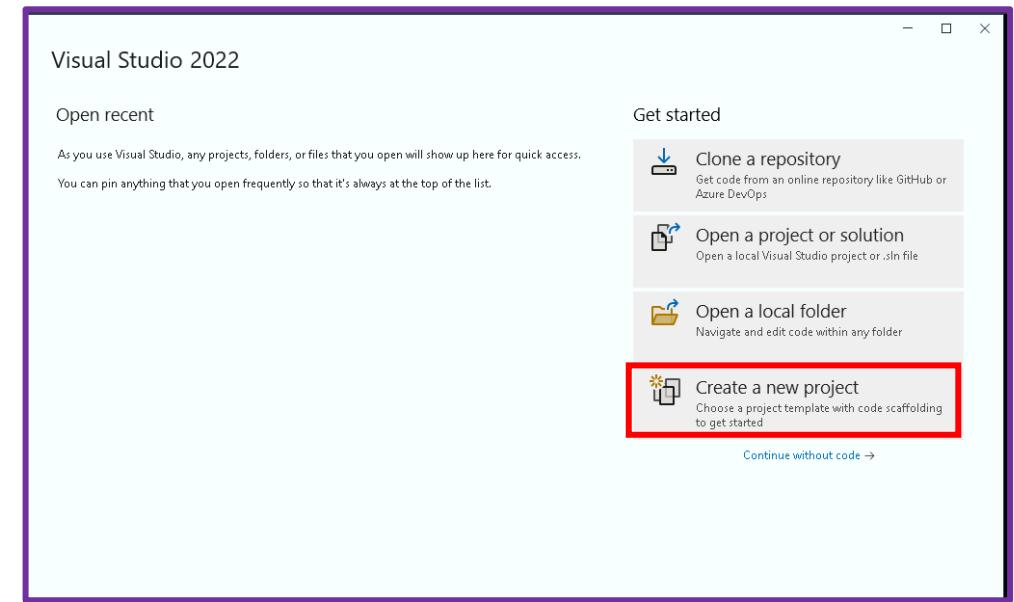
- During shutdown, the application disposes of resources and executes any registered cleanup tasks.
- Graceful shutdown ensures that ongoing requests are completed and resources like database connections and file handles are properly released.
- It also triggers events or callbacks that allow developers to run custom logic, such as logging shutdown activities or notifying external services.
- Proper shutdown handling helps maintain application stability and prevents data corruption or resource leaks during restarts or deployments.



# Setting Up Development Environment

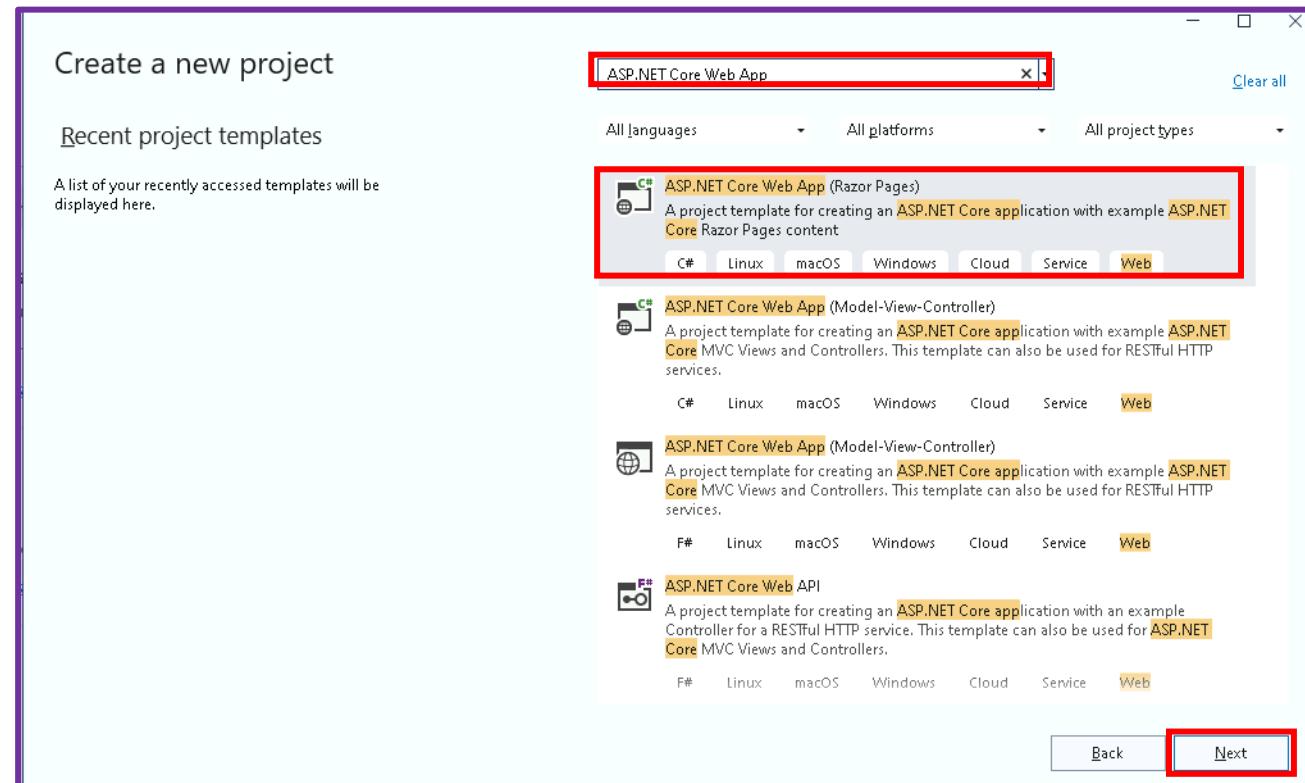
- ✓ Setting up the development environment is a crucial first step to ensure a smooth and efficient coding experience.
- ✓ It involves installing and configuring the necessary tools, frameworks, and dependencies required for building and running your applications.
- ✓ ***The following are the steps for setting up the development environment:***

**Step 1:** Open Visual Studio, then click on **Create a new project**



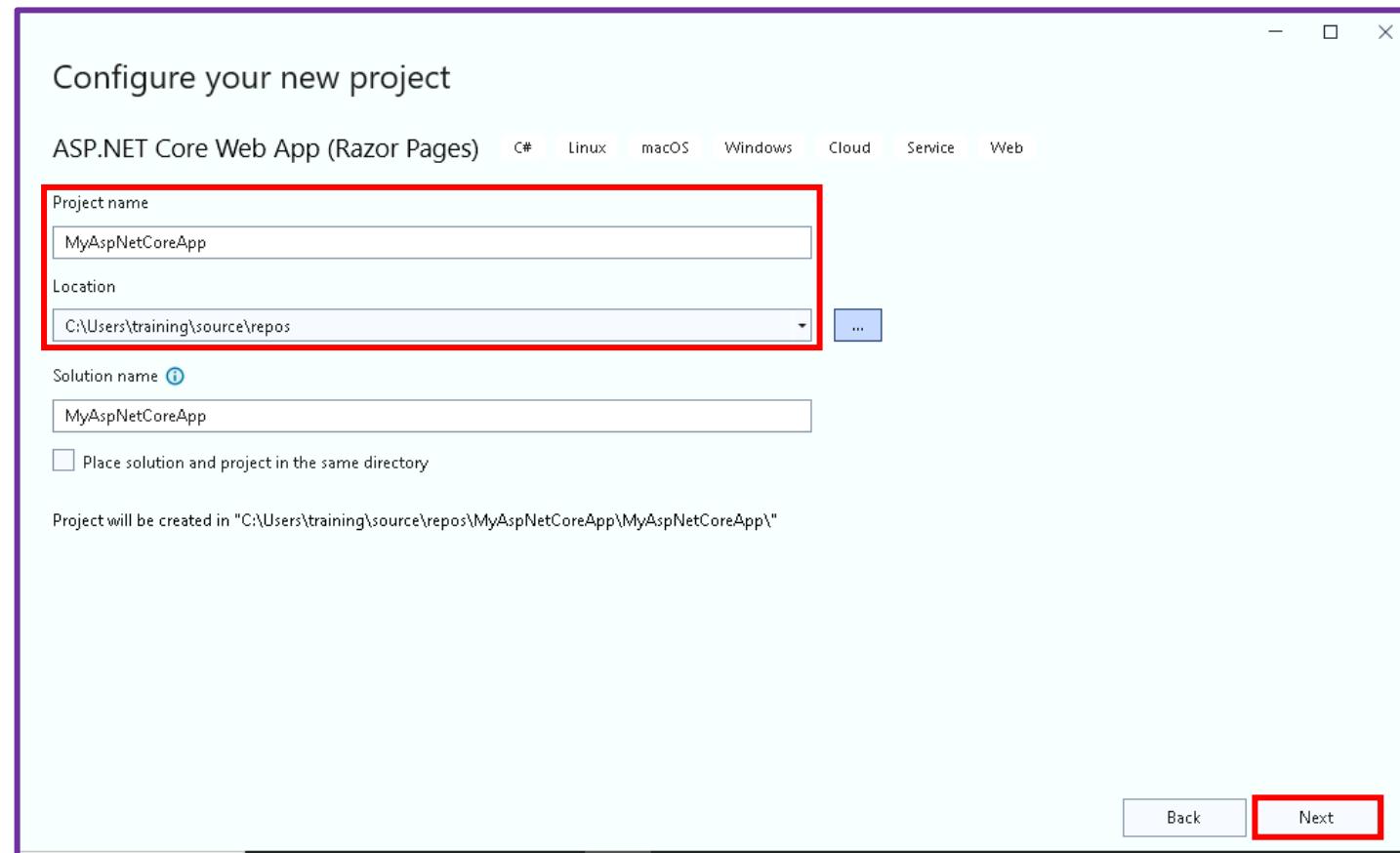
# Setting Up Development Environment

**Step 2:** Next, search for ASP.NET Core Web App (Razor Pages) in the search bar, then click **Next**



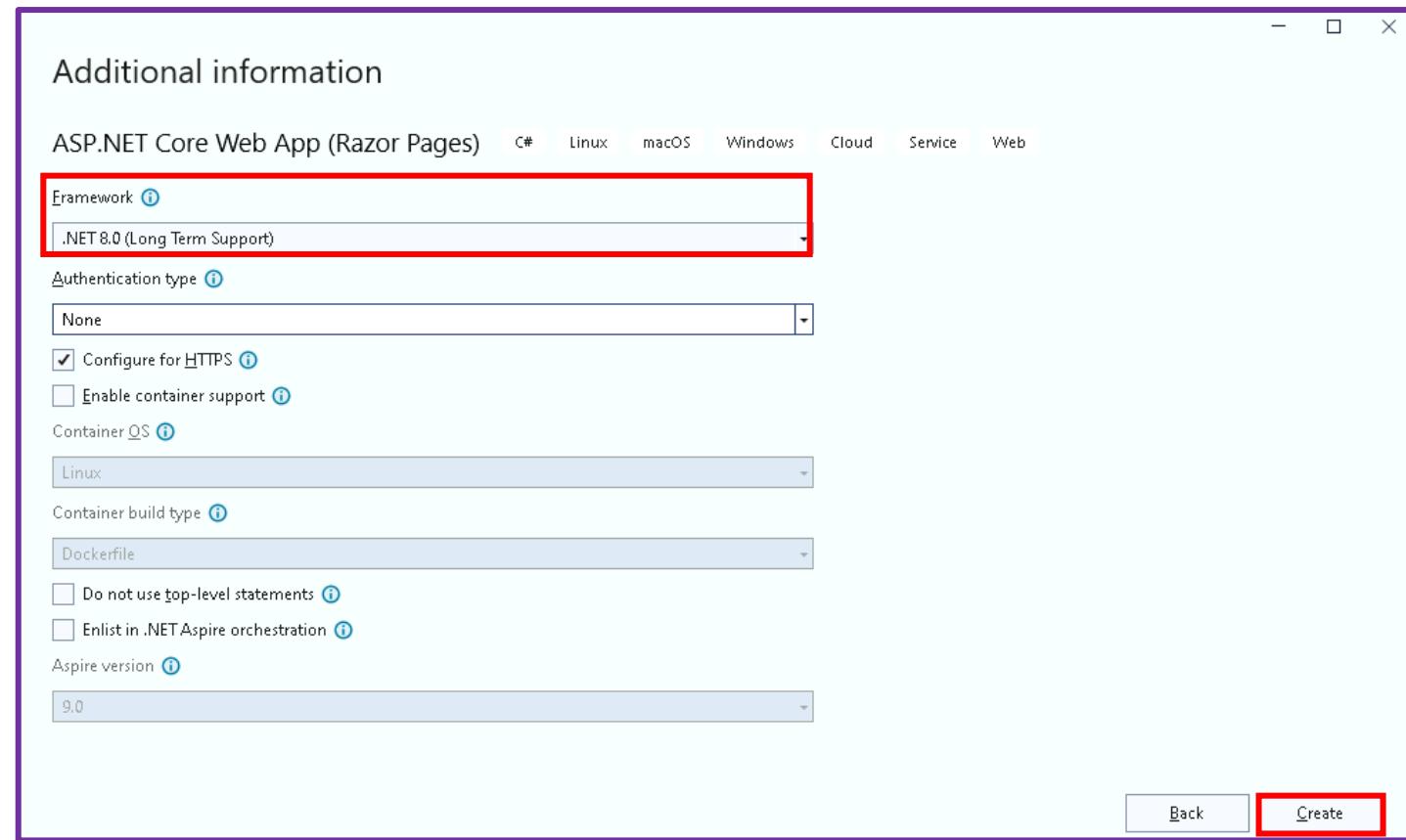
# Setting Up Development Environment

**Step 3:** Enter the **Project name**, add the location, and then click **Next**



# Setting Up Development Environment

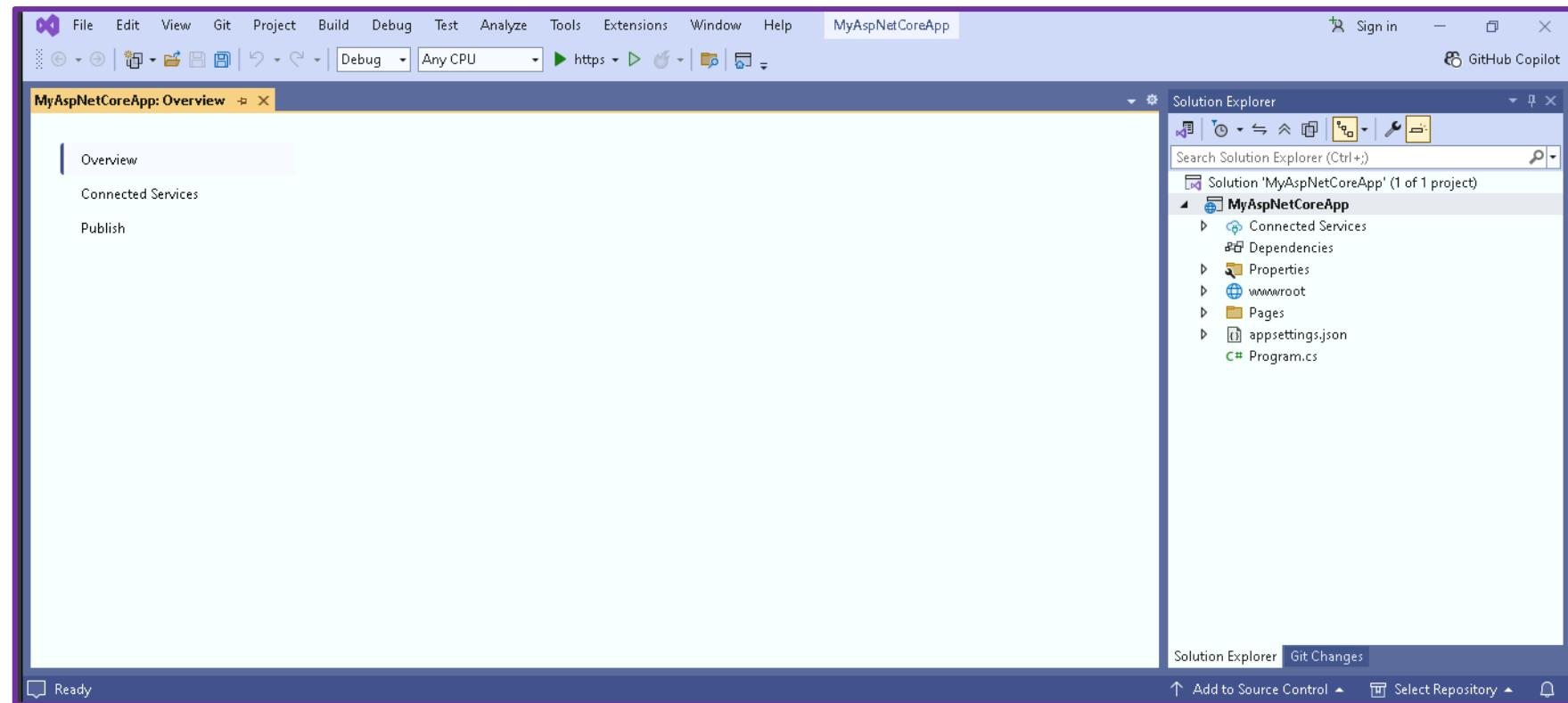
**Step 4:** Select the target **Framework**, then click **Create**



# Setting Up Development Environment

(Continued)

- ✓ You have successfully created and opened your **ASP.NET Core Web App** project in Visual Studio.



# Quiz



**Question 1:** What is one of the key benefits of ASP.NET Core?

- A) Only runs on Windows operating systems
- B) Supports a unified development model with MVC, Razor Pages, and APIs
- C) Does not support dependency injection
- D) Has a large memory footprint and slow startup times



# Quiz



**Question 2:** Which web server does ASP.NET Core use to handle requests efficiently?

- A) IIS
- B) Apache
- C) Kestrel
- D) Nginx



# Quiz



**Question 3:** During the application shutdown phase in ASP.NET Core, what happens?

- A) Resources are not released until a manual restart
- B) Ongoing requests are terminated immediately
- C) Registered cleanup tasks are executed to release resources gracefully
- D) Middleware pipeline is bypassed



# Q&A Session

**Question 1:** What does the modular architecture of ASP.NET Core enable?



**Question 2:** Why is ASP.NET Core considered cross-platform?

# Q&A Session

**Question 3:** How does the middleware pipeline function in ASP.NET Core's application lifecycle?



**Question 4:** What are the essential steps to set up a new ASP.NET Core Web App project in Visual Studio?

# Module 2: Implementing ASP.NET Core with React.js for Front-End Development

- Introduction to React.js with ASP.NET Core
- Setting Up React.js in an ASP.NET Core Project
- Best Practices for Performance Optimisation in React.js with ASP.NET Core



# Module 2: Implementing ASP.NET Core with React.js for Front-End Development

- Managing State and Routing in React with ASP.NET Core
- Server-Side Rendering (SSR) and Client-Side Rendering (CSR)
- Handling API Calls and Performance Considerations



# Introduction to React.js with ASP.NET Core

- ✓ React.js is a powerful JavaScript library used to build dynamic and interactive user interfaces through reusable components.
- ✓ When paired with ASP.NET Core, React handles the front-end development, while ASP.NET Core manages the back-end logic and data access.
- ✓ This combination allows developers to create modern, scalable web applications by separating UI concerns from server processes.
- ✓ React's virtual DOM enhances performance by efficiently updating the user interface.
- ✓ Together, they offer a flexible and efficient full-stack development approach for building robust web solutions.



# Introduction to React.js with ASP.NET Core

(Continued)

## ***Key Advantages of Using React.Js with ASP.NET Core***

### **1. Separation of concerns:**

- Front-end UI and back-end services are developed independently but communicate smoothly via APIs.
- This separation enables teams to work in parallel and maintain the codebase more easily as projects grow in complexity.

### **2. Component-based architecture:**

- React's reusable components make UI development modular and maintainable.



# Introduction to React.js with ASP.NET Core

(Continued)

- Components encapsulate their own logic and presentation, allowing for better code organisation and reusability across different parts of the application.

## **3. Improved performance:**

- React's virtual DOM optimises rendering, enhancing user experience. Instead of updating the entire page, React calculates the minimal set of changes required, making UI updates faster and reducing browser workload.

## **4. Rich ecosystem:**

- Large community support with libraries for routing, state management, and testing.

# Introduction to React.js with ASP.NET Core

(Continued)

- Tools like React Router for navigation and Redux for state management help build complex applications with ease and consistency.

## 5. Full-stack development:

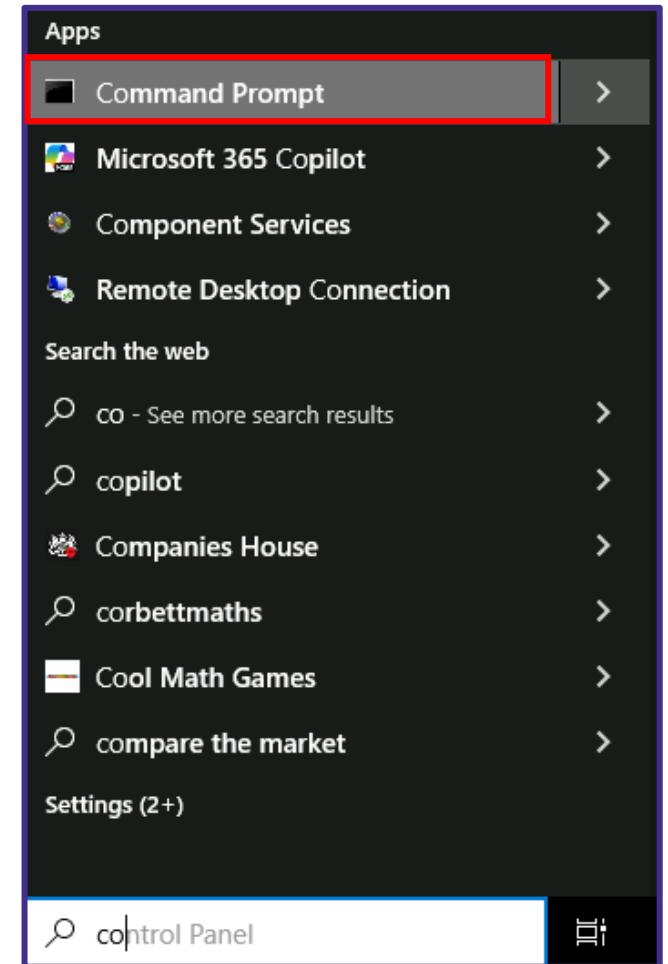
- Using React with ASP.NET Core enables building scalable, modern web applications combining the power of .NET backend with modern front-end technologies.
- This setup supports secure API-driven communication and seamless integration with cloud services.



# Setting Up React.js in an ASP.NET Core Project

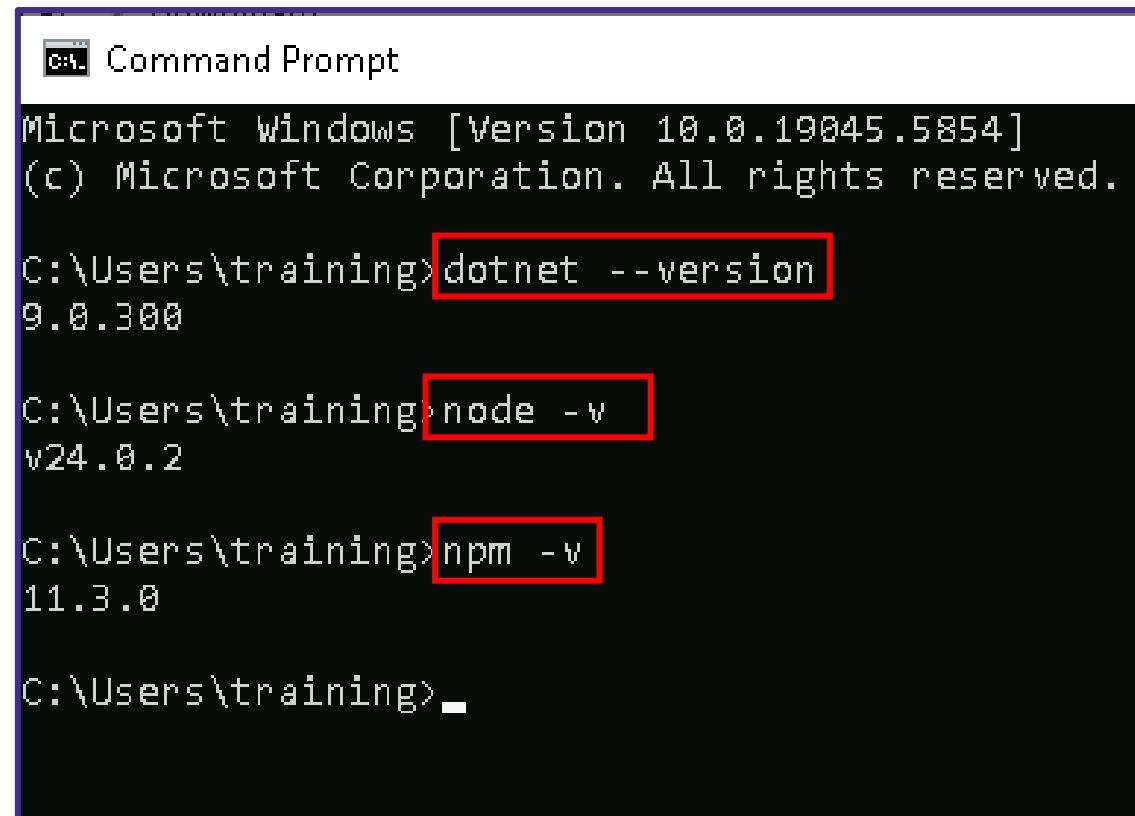
- ✓ Setting up React.js within an ASP.NET Core project enables developers to build modern, dynamic front-end interfaces integrated seamlessly with robust backend services.
- ✓ The process involves configuring the project environment to support both frameworks effectively.
- ✓ ***The following are the steps for setting up react.js in an ASP.NET core project:***

**Step 1:** Open **Command Prompt** by typing it into the Windows search bar



# Setting Up React.js in an ASP.NET Core Project

**Step 2:** Run the following command to confirm that the software is installed



```
Microsoft Windows [Version 10.0.19045.5854]
(c) Microsoft Corporation. All rights reserved.

C:\Users\training>dotnet --version
9.0.300

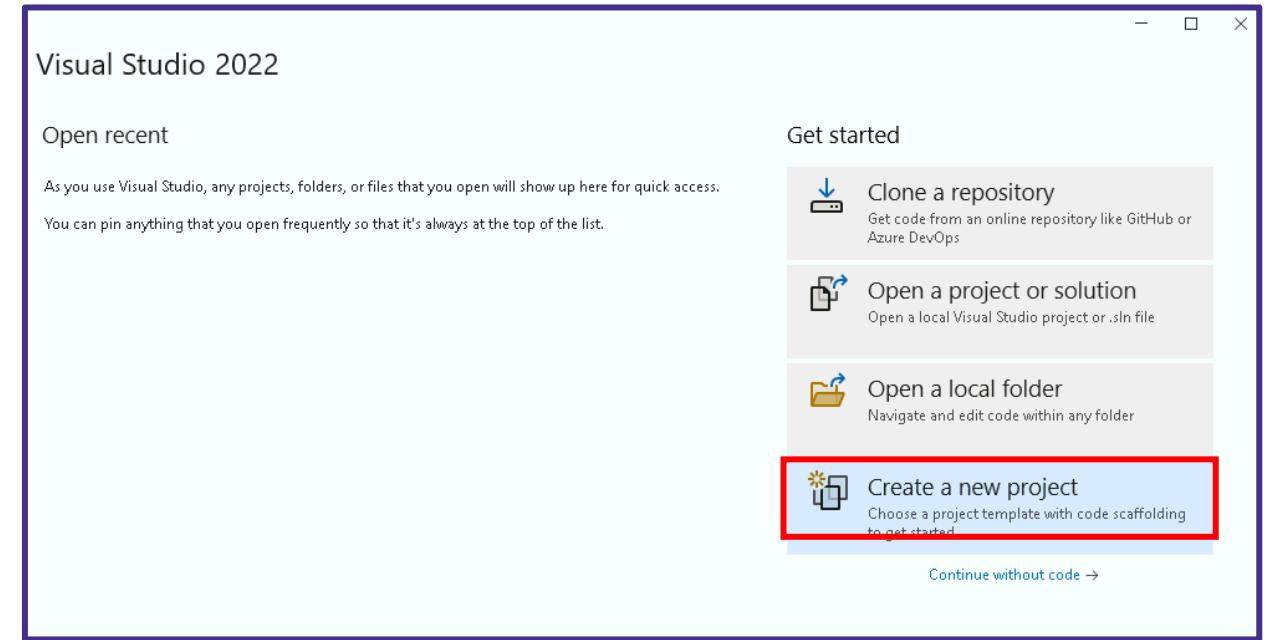
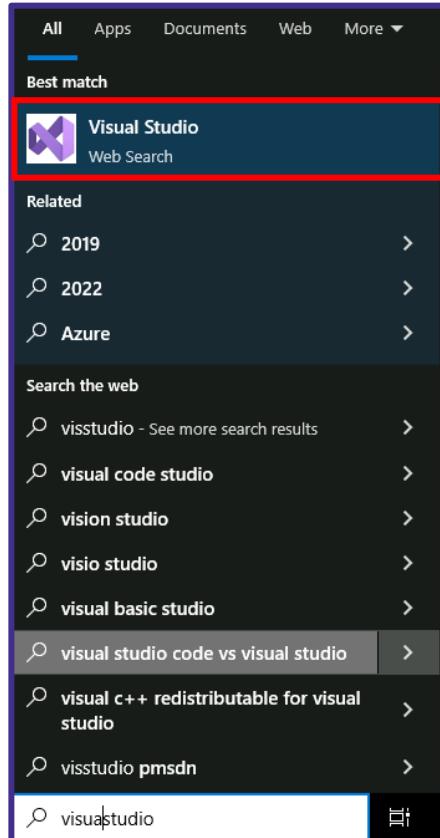
C:\Users\training>node -v
v24.0.2

C:\Users\training>npm -v
11.3.0

C:\Users\training>
```

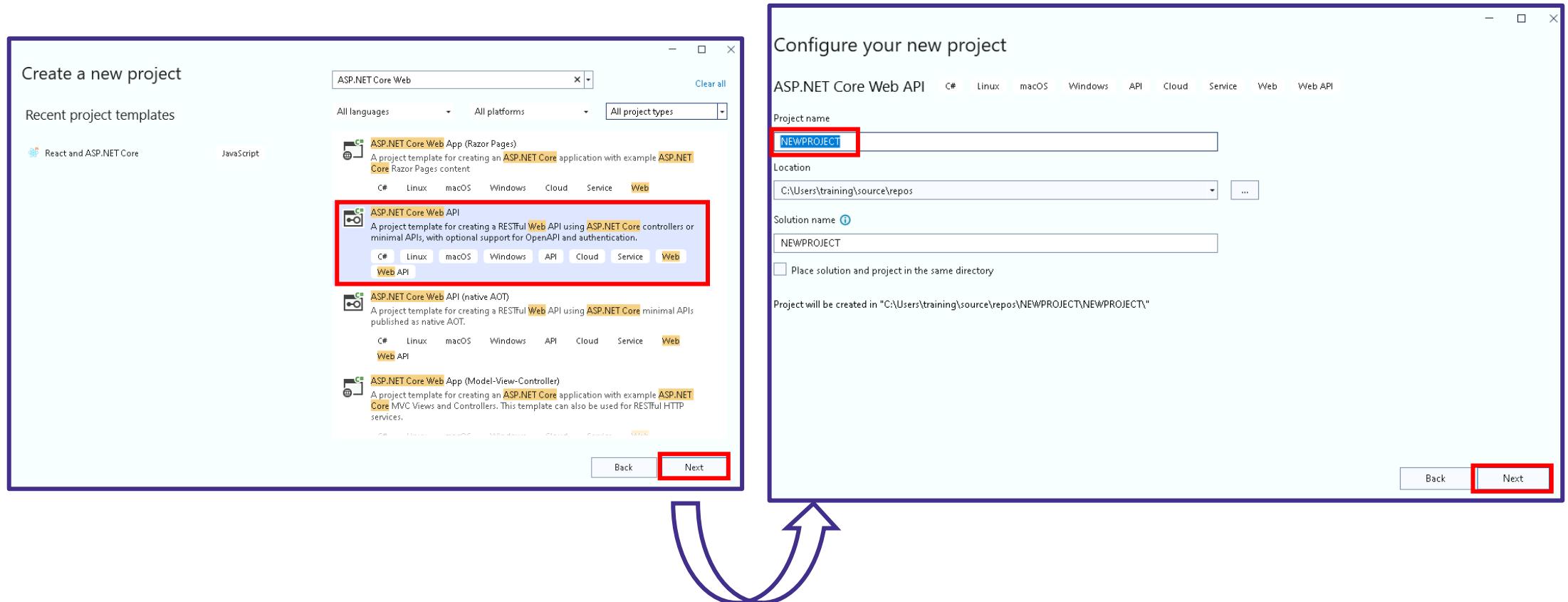
# Setting Up React.js in an ASP.NET Core Project

**Step 3:** In the Windows search bar, type and open **Visual Studio**, then click on **Create a new project**



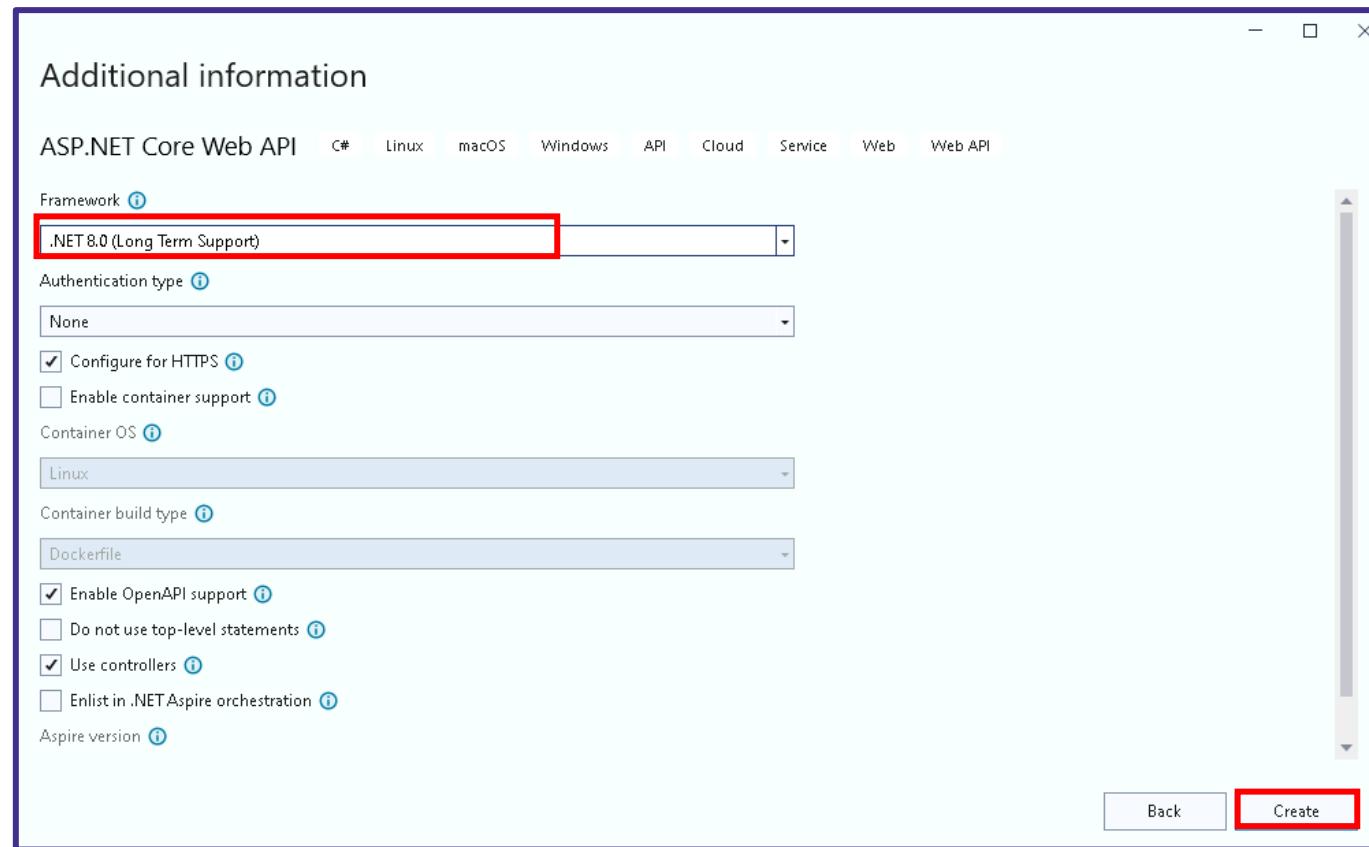
# Setting Up React.js in an ASP.NET Core Project

**Step 4:** Choose **ASP.NET Core Web API**, then click **Next**. Enter the project name, select the location, and click **Next** again



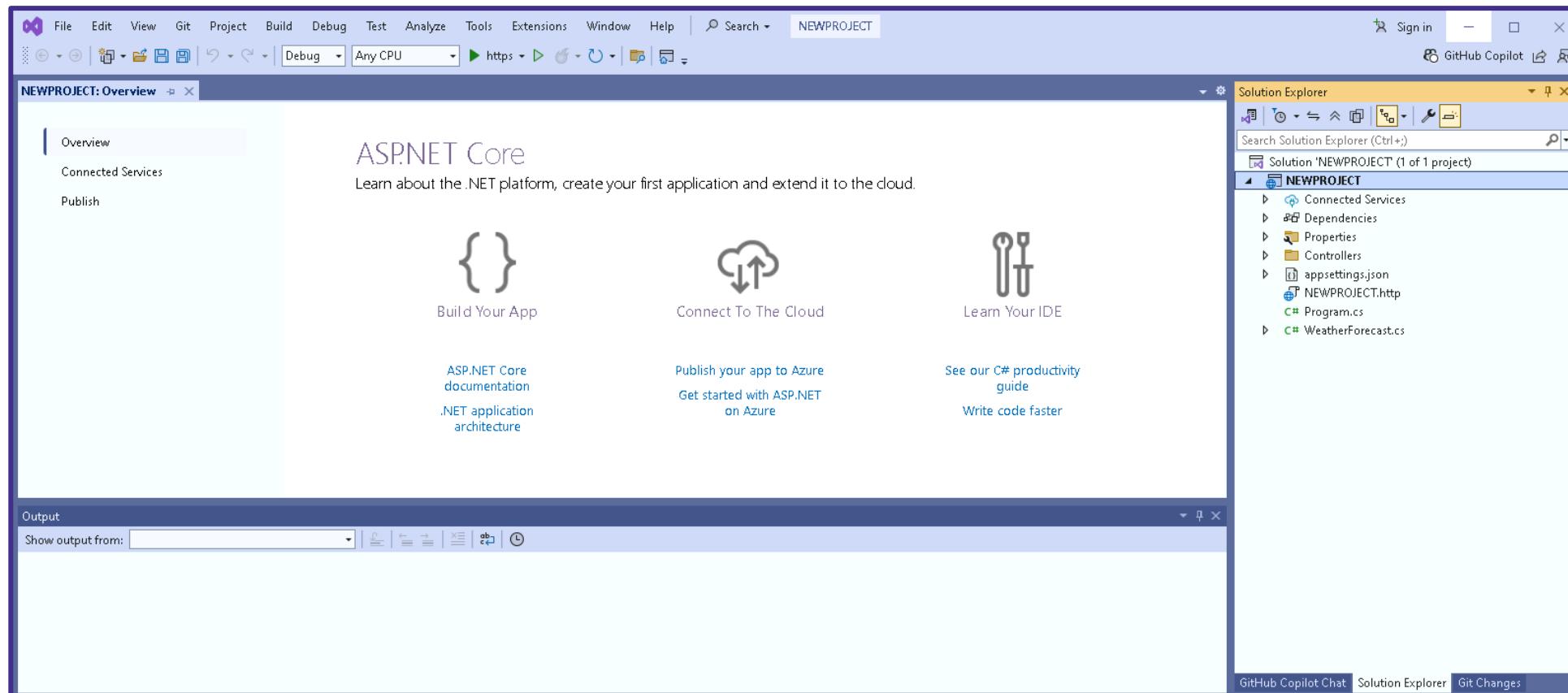
# Setting Up React.js in an ASP.NET Core Project

**Step 5:** Choose .NET 8.0 as the framework and then click **Create**



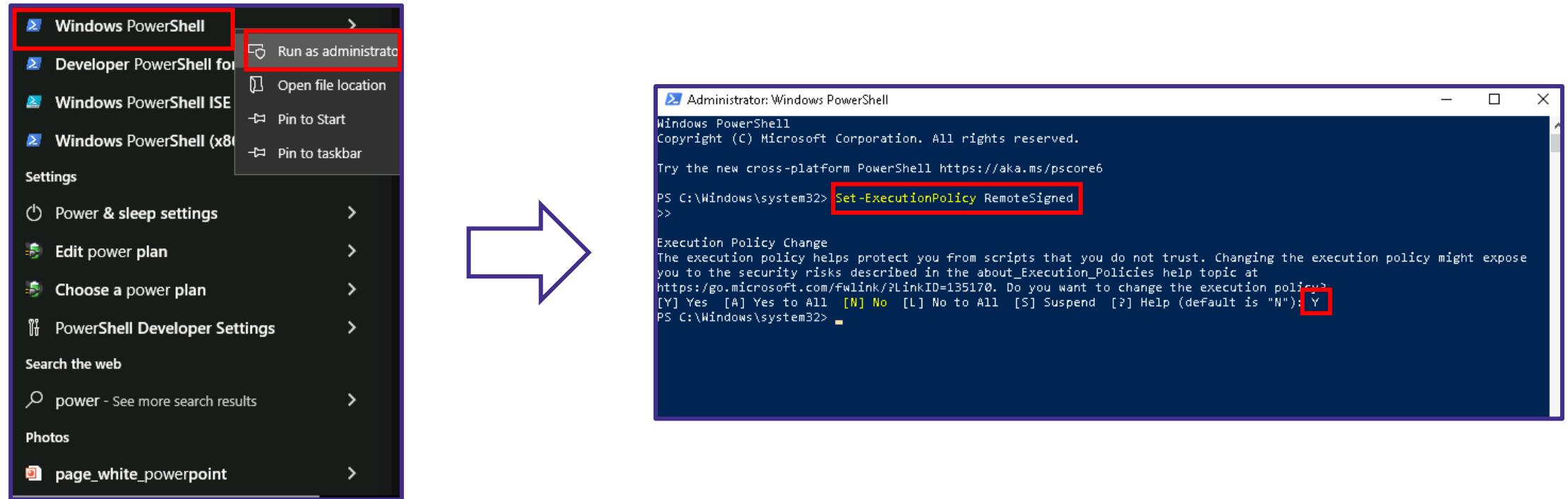
# Setting Up React.js in an ASP.NET Core Project

**Step 6:** The new project will be created as shown



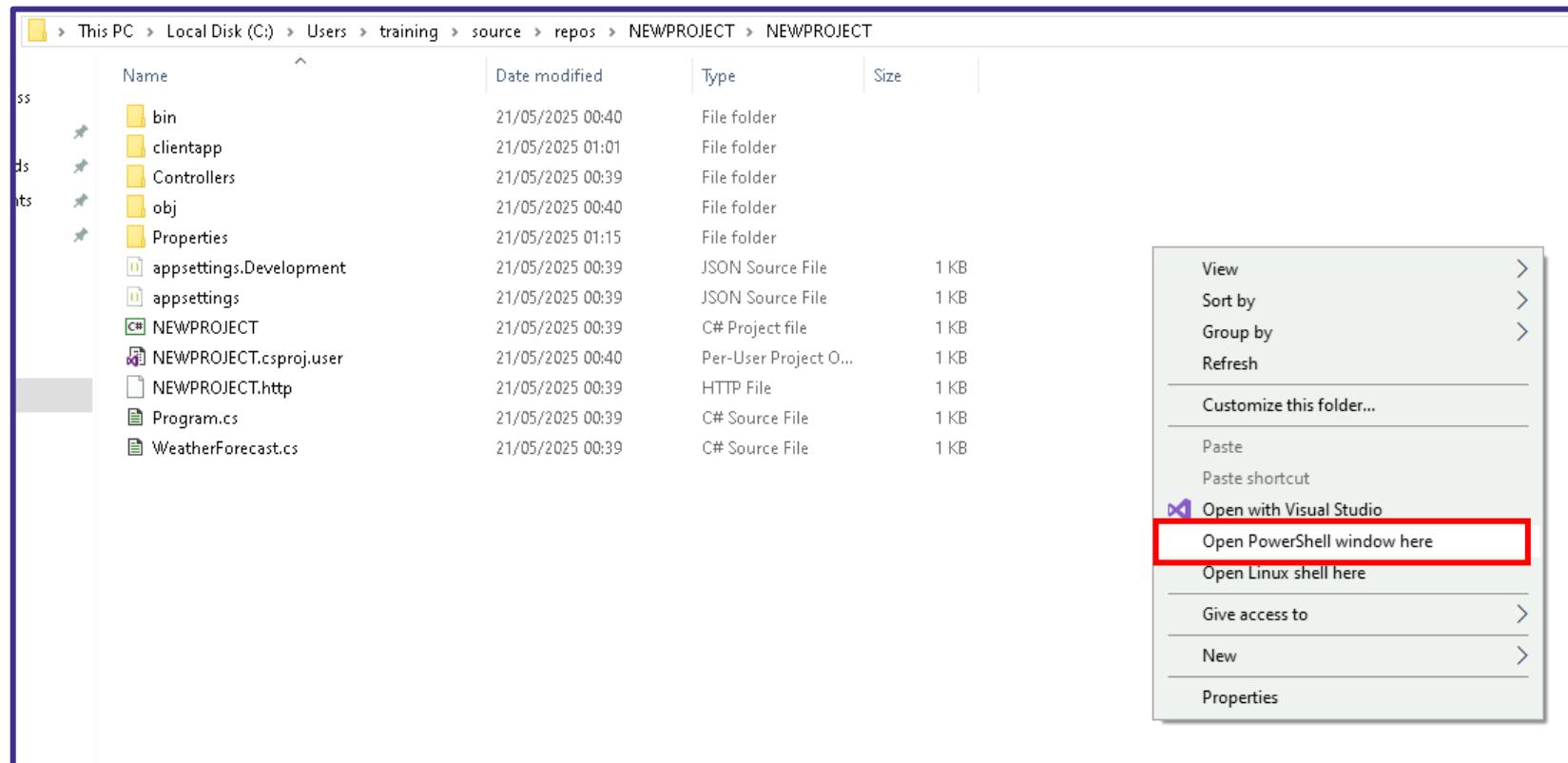
# Setting Up React.js in an ASP.NET Core Project

Step 7: Open PowerShell with administrative privileges, paste the provided query, and then press Y to proceed



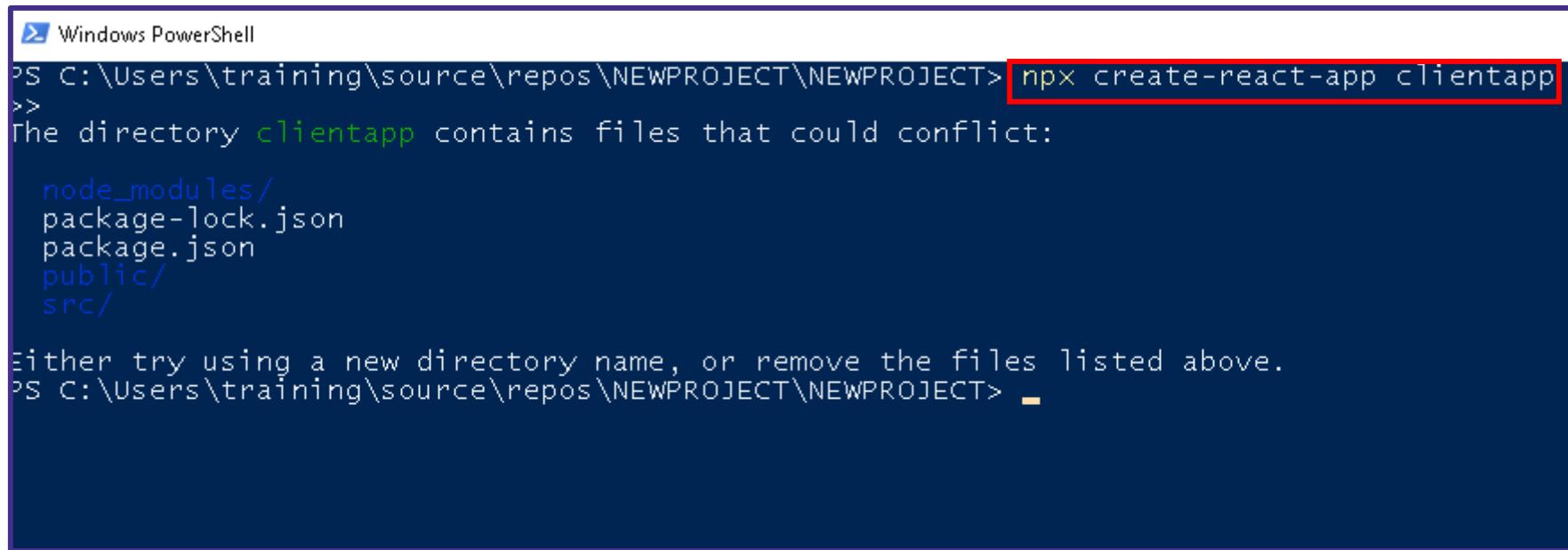
# Setting Up React.js in an ASP.NET Core Project

**Step 8:** Navigate to the project location shown, then hold Shift, right-click, and select **Open PowerShell window here**



# Setting Up React.js in an ASP.NET Core Project

**Step 9:** Now, paste the given query into the PowerShell interface to create the **ClientApp** folder

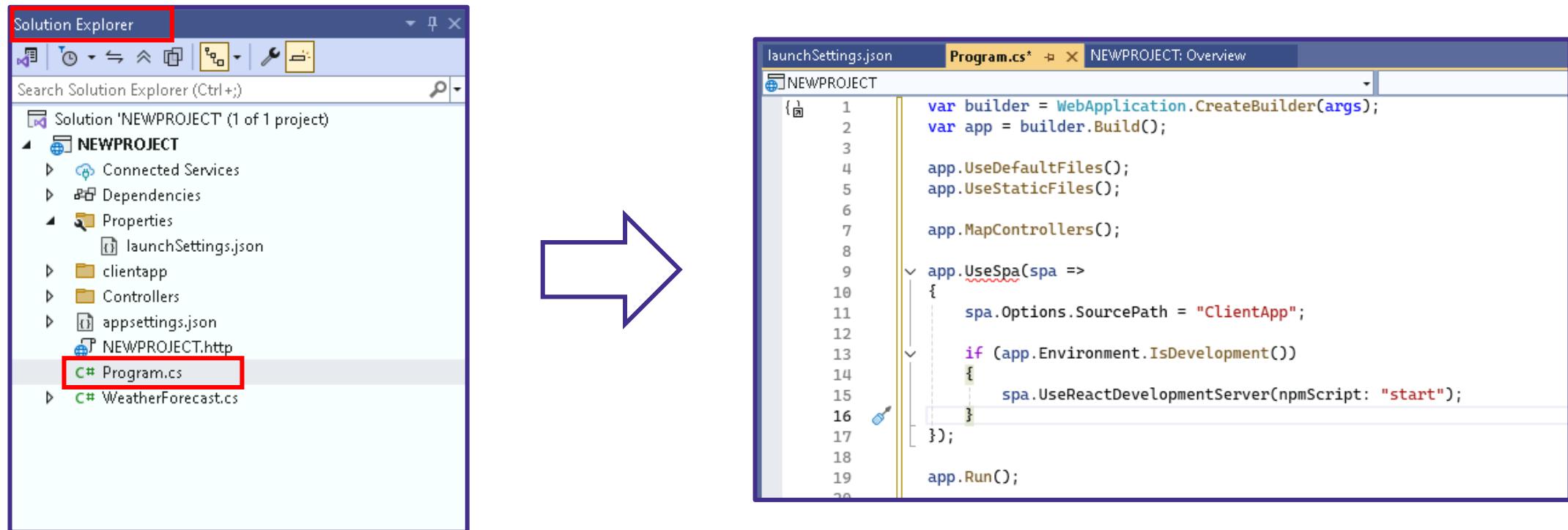


```
Windows PowerShell
PS C:\Users\training\source\repos\NEWPROJECT\NEWPROJECT> npx create-react-app clientapp
>>
The directory clientapp contains files that could conflict:
node_modules/
package-lock.json
package.json
public/
src/

Either try using a new directory name, or remove the files listed above.
PS C:\Users\training\source\repos\NEWPROJECT\NEWPROJECT> -
```

# Setting Up React.js in an ASP.NET Core Project

**Step 10:** Now, navigate to the **Solution Explorer** on the right side, click on **Program.cs**, and replace the existing code with the provided code



# Setting Up React.js in an ASP.NET Core Project

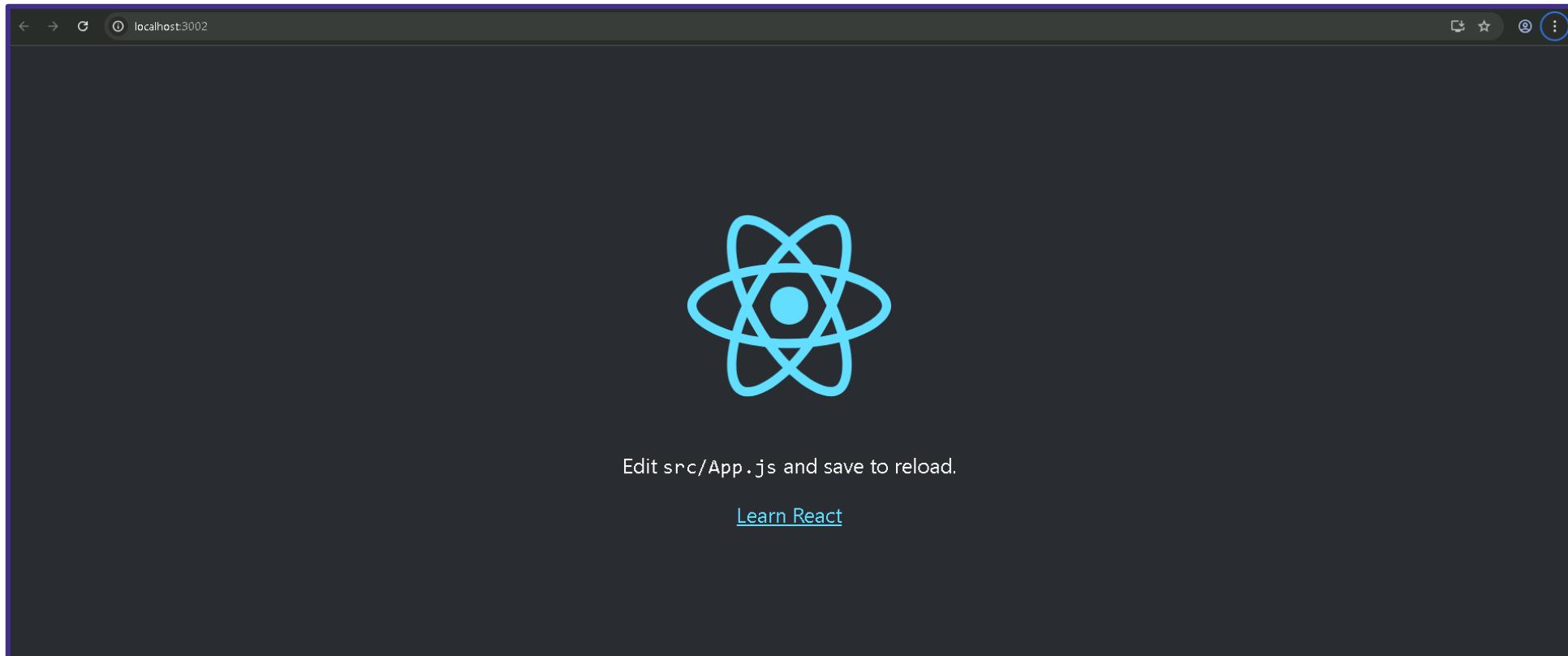
**Step 11:** Now, enter the following command in PowerShell to start the React development server

```
Windows PowerShell
PS C:\Users\training\source\repos\NEWPROJECT\NEWPROJECT> npx create-react-app clientapp
>>
The directory clientapp contains files that could conflict:
  node_modules/
  package-lock.json
  package.json
  public/
  src/
Either try using a new directory name, or remove the files listed above.
PS C:\Users\training\source\repos\NEWPROJECT\NEWPROJECT> cd ClientApp
>> npm start
>>
> clientapp@0.1.0 start
> react-scripts start
(node:2304) [DEP0176] DeprecationWarning: fs.F_OK is deprecated, use fs.constants.F_OK instead
(Use `node --trace-deprecation ...` to show where the warning was created)
/ Something is already running on port 3000.

Would you like to run the app on another port instead? ... yes
(node:2304) [DEP_WEBPACK_DEV_SERVER_ON_AFTER_SETUP_MIDDLEWARE] DeprecationWarning: 'onAfterSetupMiddleware' option is de
precated. Please use the 'setupMiddlewares' option.
```

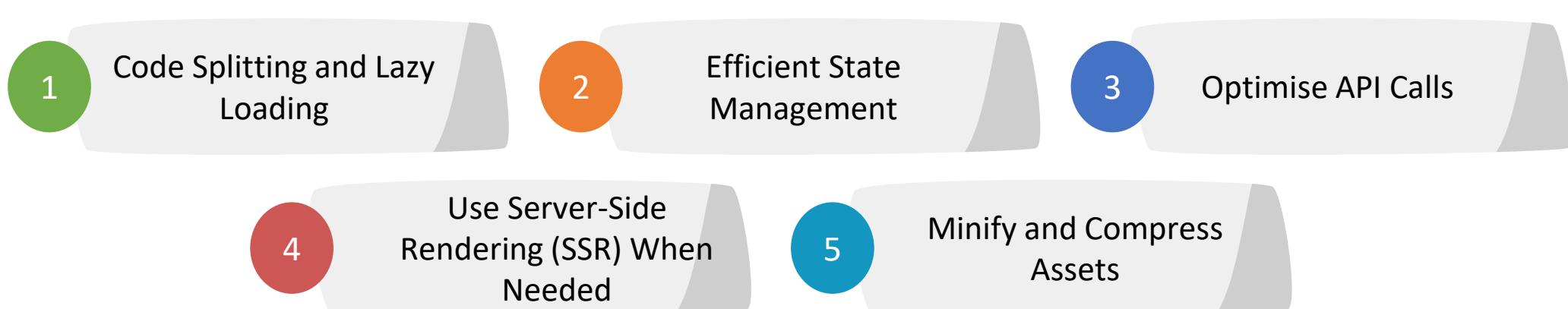
# Setting Up React.js in an ASP.NET Core Project

**Step 12:** The React app is running at <http://localhost:3002> as shown



# Best Practices for Performance Optimisation in React.js with ASP.NET Core

- ✓ Performance optimisation is crucial when building React.js front-end applications integrated with ASP.NET Core to ensure fast load times and a smooth user experience.
- ✓ By applying best practices such as efficient state management, code splitting, and server-side rendering, developers can significantly improve application responsiveness and reduce server load.
- ✓ *The following best practices for performance optimisation in React.js with ASP.NET Core:*



# Managing State and Routing in React with ASP.NET Core

## Managing State in React

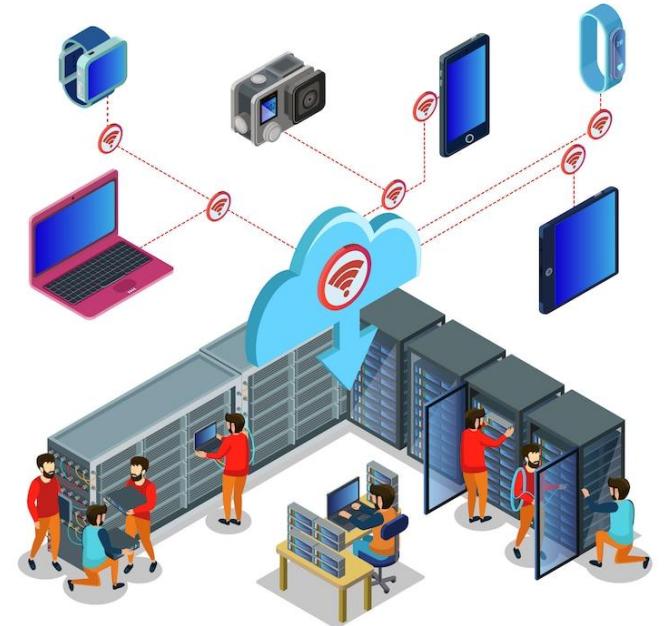
- ✓ State management in React is crucial for controlling how data flows and changes within the application. React components can hold local state using the useState hook or class component state.
- ✓ For larger applications, managing global state becomes essential to share data across components effectively.
- ✓ This can be done using tools such as **React Context API** or third-party libraries like **Redux**.
- ✓ Integrating state management properly ensures smooth UI updates and predictable data flow, especially when React is combined with ASP.NET Core backend APIs.



# Managing State and Routing in React with ASP.NET Core

## Routing in React

- ✓ Routing allows navigation between different views or pages without refreshing the entire web application.
- ✓ React Router is the standard library used for routing in React apps. It manages the browser history, enabling dynamic rendering of components based on the URL path.
- ✓ When combined with ASP.NET Core, React Router handles client-side routes while ASP.NET Core manages API routes on the server.
- ✓ This separation of concerns allows a seamless single-page application (SPA) experience.

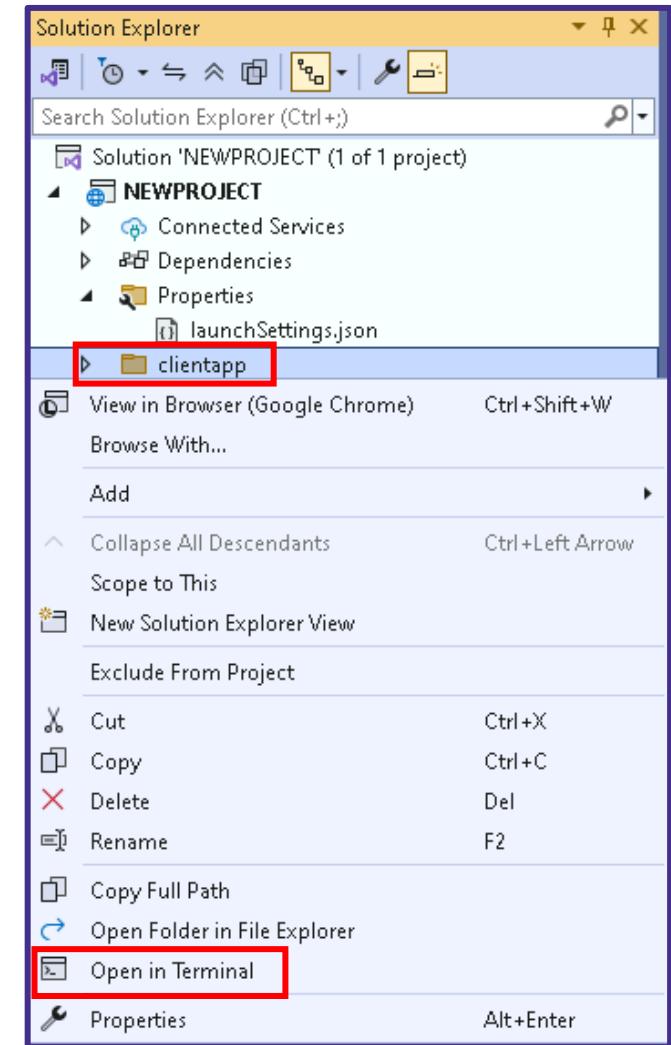


# Managing State and Routing in React with ASP.NET Core

(Continued)

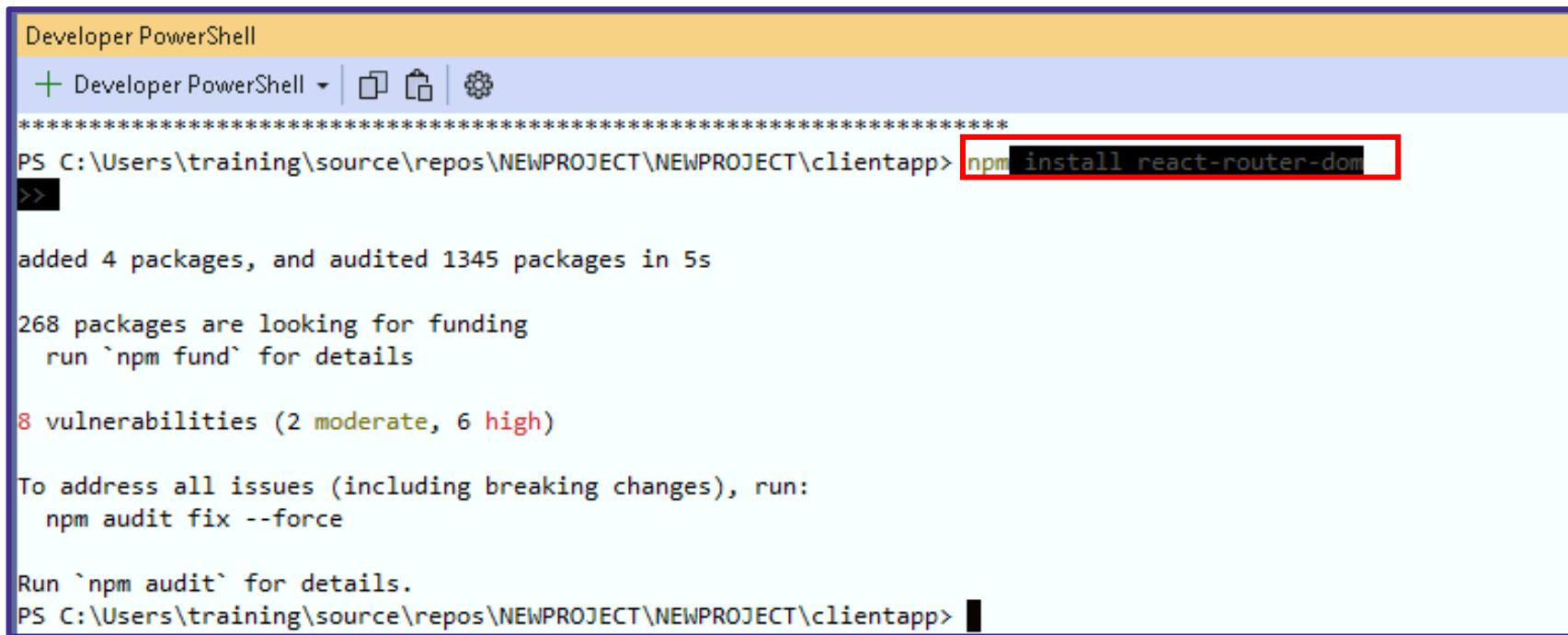
- ***The following are the steps to manage state and routing in react with ASP.NET core:***

**Step 1:** Right-click on **clientapp** and choose **Open in Terminal**



# Managing State and Routing in React with ASP.NET Core

**Step 2:** Paste the given query



Developer PowerShell

+ Developer PowerShell - | ⌂ ⌂ | ⚙

```
*****
PS C:\Users\training\source\repos\NEWPROJECT\NEWPROJECT\clientapp> npm install react-router-dom
>>
```

added 4 packages, and audited 1345 packages in 5s

268 packages are looking for funding  
run `npm fund` for details

8 vulnerabilities (2 moderate, 6 high)

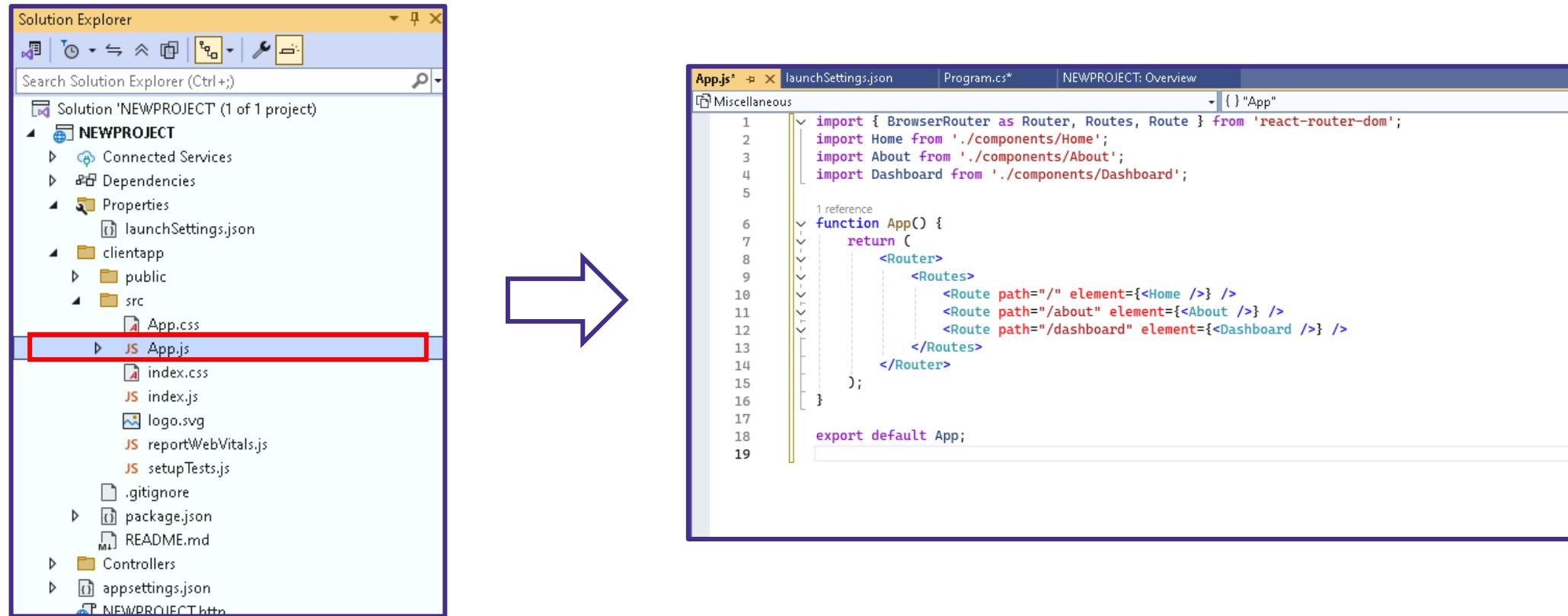
To address all issues (including breaking changes), run:  
npm audit fix --force

Run `npm audit` for details.

```
PS C:\Users\training\source\repos\NEWPROJECT\NEWPROJECT\clientapp>
```

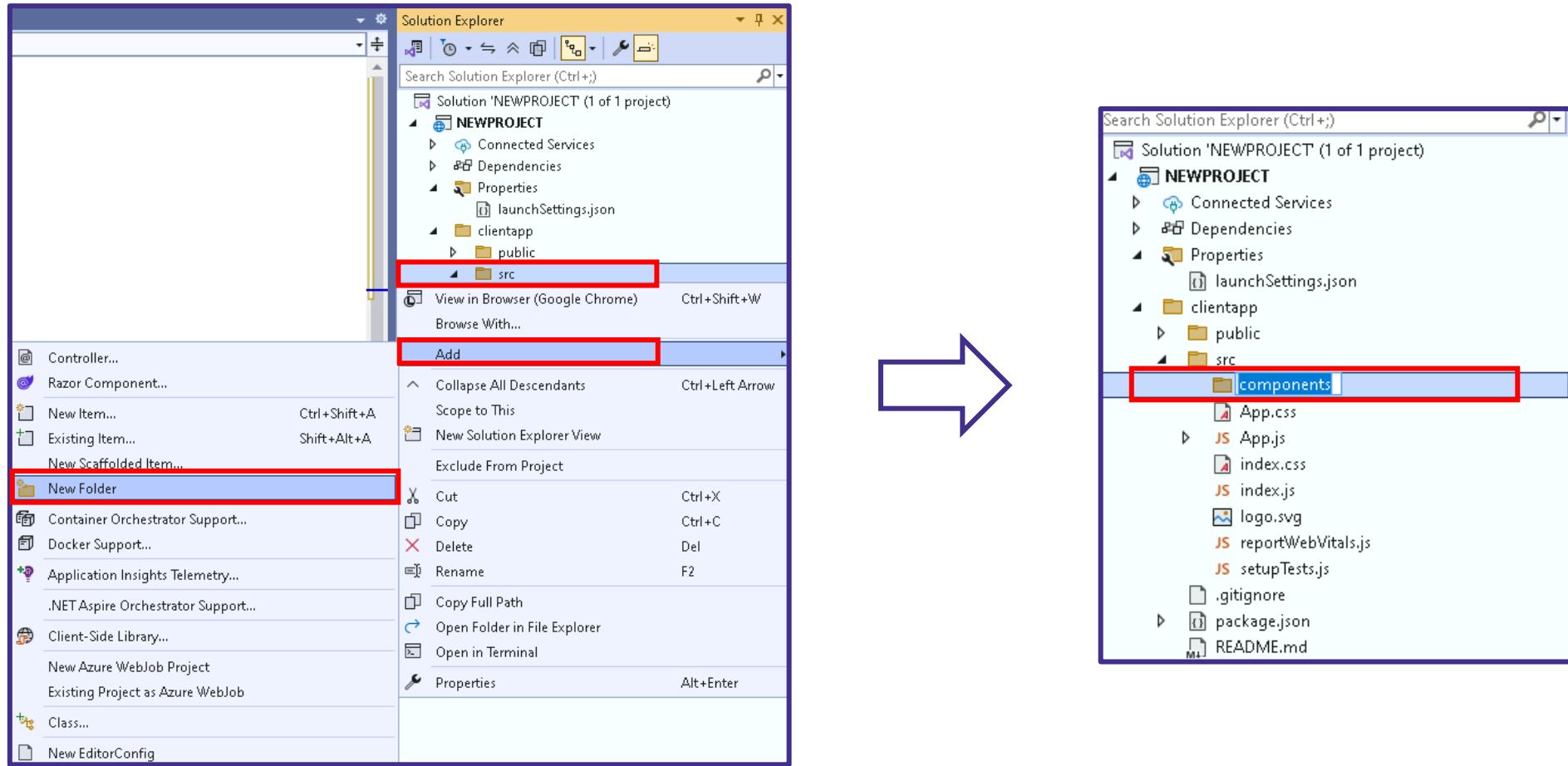
# Managing State and Routing in React with ASP.NET Core

Step 3: Click on App.js to open it and paste the provided code



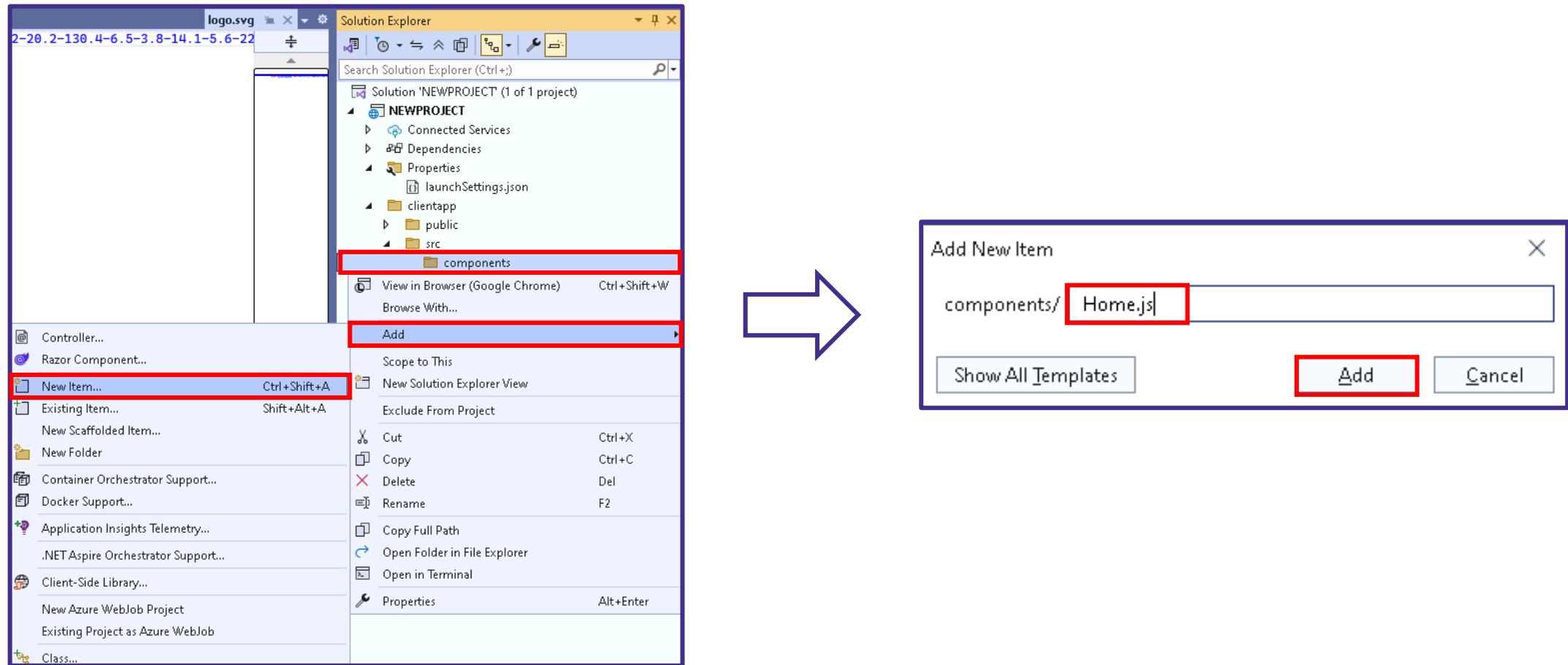
# Managing State and Routing in React with ASP.NET Core

Step 4: Right-click on the **src** folder inside clientapp, select **Add > New Folder**, and name it **components**



# Managing State and Routing in React with ASP.NET Core

**Step 5:** Right-click on the newly created **components** folder, select **Add > New Item**, name it **Home.js**, and then click **Add**



# Managing State and Routing in React with ASP.NET Core

**Step 6:** Now paste the given code in Home.js file

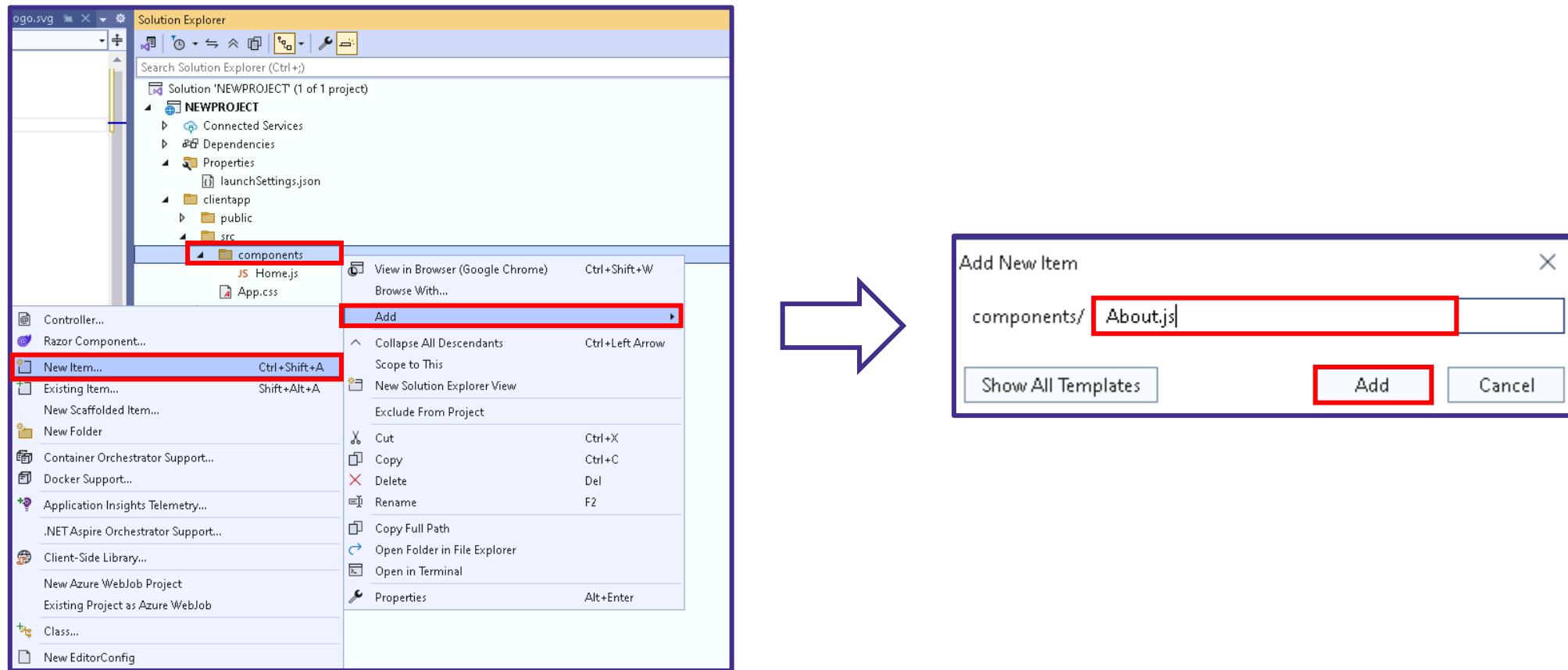
The screenshot shows the Visual Studio IDE interface. On the left, the code editor displays the `Home.js` file with the following content:

```
1  using static System.Runtime.InteropServices.JavaScript.JSType;
2
3  import React from 'react';
4
5  function Home()
6  {
7      return (
8          <div>
9              <h1> Home Page </h1>
10         </div>
11     );
12
13     export default Home;
14
15
16
17
18
19
```

On the right, the Solution Explorer pane shows the project structure for "NEWPROJECT". The `clientapp` folder contains the `src` folder, which in turn contains the `components` folder and the `JS Home.js` file. Other files visible in the `clientapp/src` folder include `App.css`, `index.css`, `index.js`, `logo.svg`, `reportWebVitals.js`, and `setupTests.js`. The `Properties` folder contains the `launchSettings.json` file. The `Controllers` and `WeatherForecast.cs` files are also listed under the `clientapp` folder.

# Managing State and Routing in React with ASP.NET Core

Step 7: Right-click the Components folder, select Add > New Item, name it **About.js**, and then click **Add**



# Managing State and Routing in React with ASP.NET Core

**Step 8:** Now paste the code in the About.js as displayed

The screenshot shows the Visual Studio IDE interface. On the left, the code editor displays the `About.js` file with the following content:

```
1  using static System.Runtime.InteropServices.JavaScript.JSType;
2
3  import React from 'react';
4
5  function About()
6  {
7      return (
8          <div>
9              <h1> About Page </h1>
10             </div>
11     );
12 }
13
14 export default About;
```

The `About.js` file is highlighted in the editor. On the right, the Solution Explorer pane shows the project structure for "NEWPROJECT". The `clientapp` folder contains `public` and `src` subfolders. The `src` folder contains a `components` folder which contains the `About.js` file, which is also highlighted with a red border.

# Managing State and Routing in React with ASP.NET Core

Step 9: Create a new item named **Dashboard**, similar to the previous one, and paste the provided code into it



```
Dashboard.js + X About.js Home.js App.js launchSettings.json Program.cs NEWPROJECT: Overview
Miscellaneous ↴ Dashboard
1 import React, { useState } from 'react';
2
3 function Dashboard() {
4   const [count, setCount] = useState(0);
5
6   return (
7     <div>
8       <h2>Dashboard</h2>
9       <p>Count: {count}</p>
10      <button onClick={() => setCount(count + 1)}>Increment</button>
11    </div>
12  );
13}
14
15 export default Dashboard;
```

# Managing State and Routing in React with ASP.NET Core

Step 10: Now, paste the provided code into the **Program.cs** file as shown

The screenshot shows the Visual Studio IDE interface. The main window displays the **Program.cs** file content:

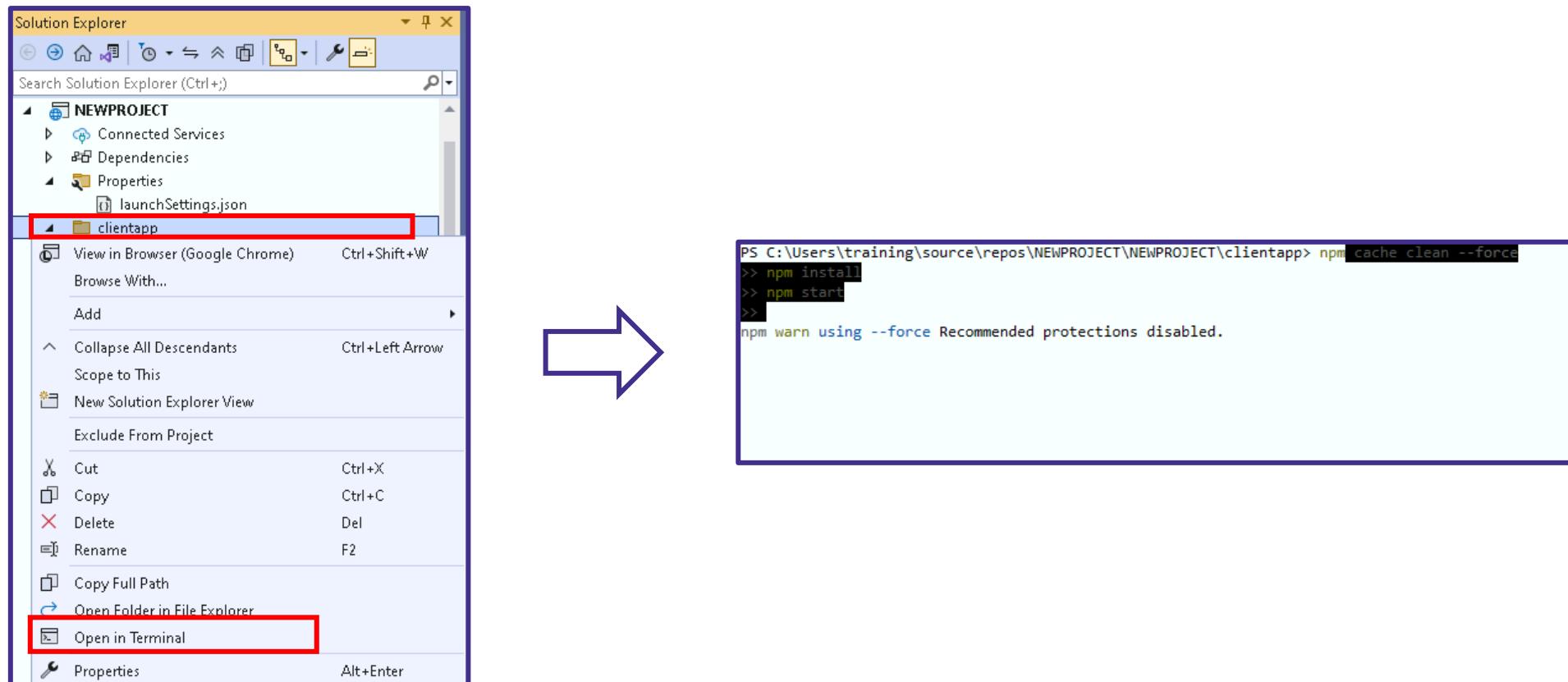
```
1 var builder = WebApplication.CreateBuilder(args);
2 var app = builder.Build();
3 
4 app.UseDefaultFiles();
5 app.UseStaticFiles();
6 
7 app.MapControllers();
8 
9 // If you want to serve React app static files in production:
10 app.MapFallbackToFile("index.html");
11 
12 app.Run();
13 
```

The Solution Explorer on the right side shows the project structure for "NEWPROJECT". The **Program.cs** file is highlighted with a red border.

Developer PowerShell window at the bottom shows the command `dotnet run` being executed in the directory `C:\Users\training\source\repos\NEWPROJECT\NEWPROJECT`.

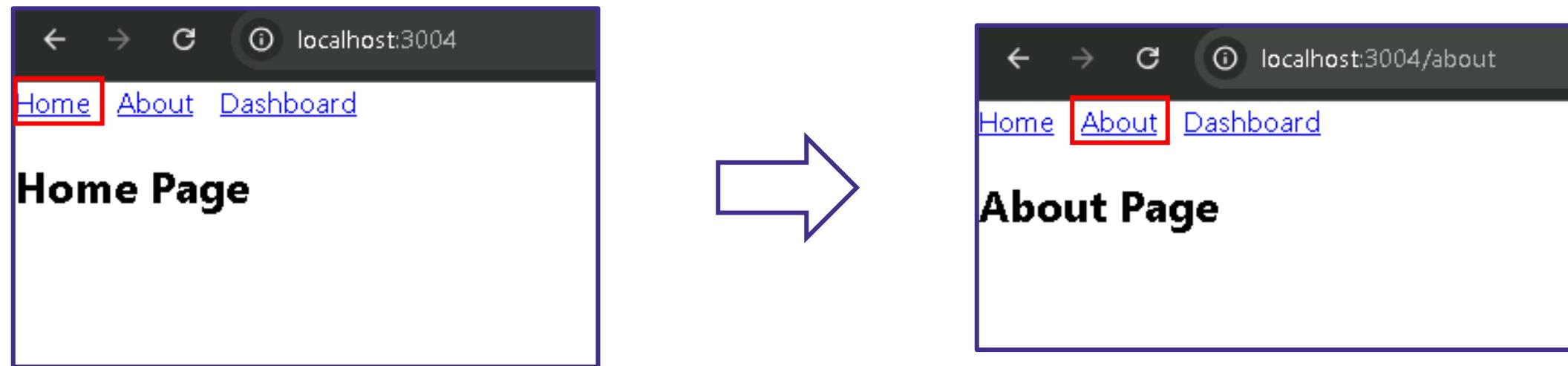
# Managing State and Routing in React with ASP.NET Core

**Step 11:** Right-click on the **clientapp** folder, choose **Open in Terminal**, then paste the provided command to run the output in the browser



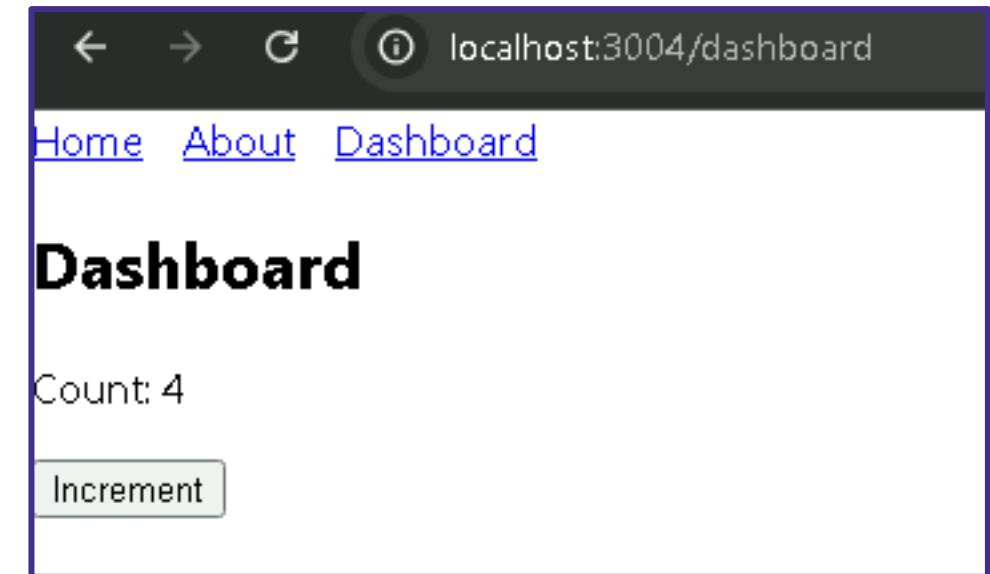
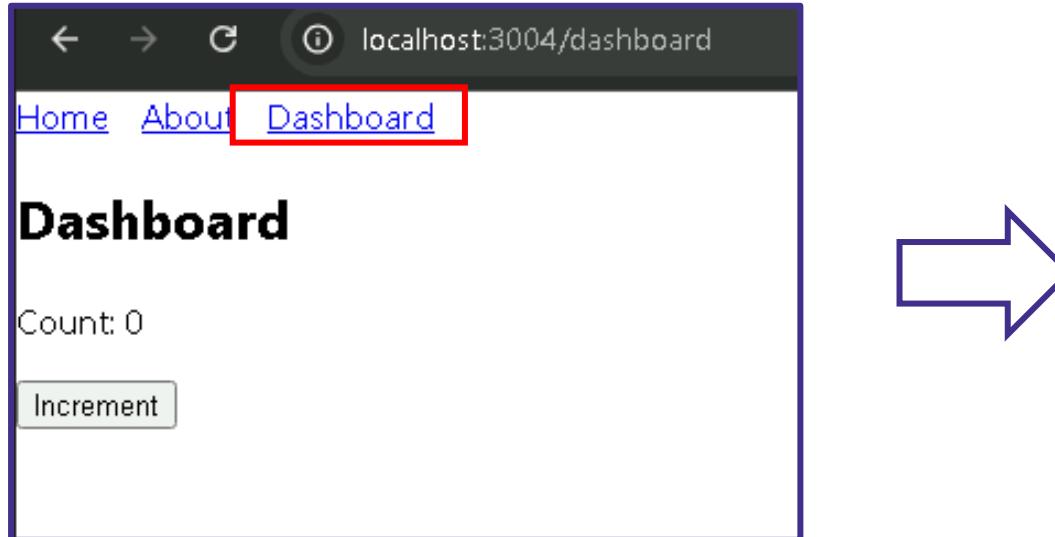
# Managing State and Routing in React with ASP.NET Core

**Step 12:** The output will be shown in the browser at localhost. Simply click the **Home** link to open the Home page, and click the **About** link to view the About page as displayed



# Managing State and Routing in React with ASP.NET Core

**Step 13:** Click on **Dashboard**, and it will appear as shown



# Server-Side Rendering (SSR) and Client-Side Rendering (CSR)

## Server-Side Rendering (SSR)

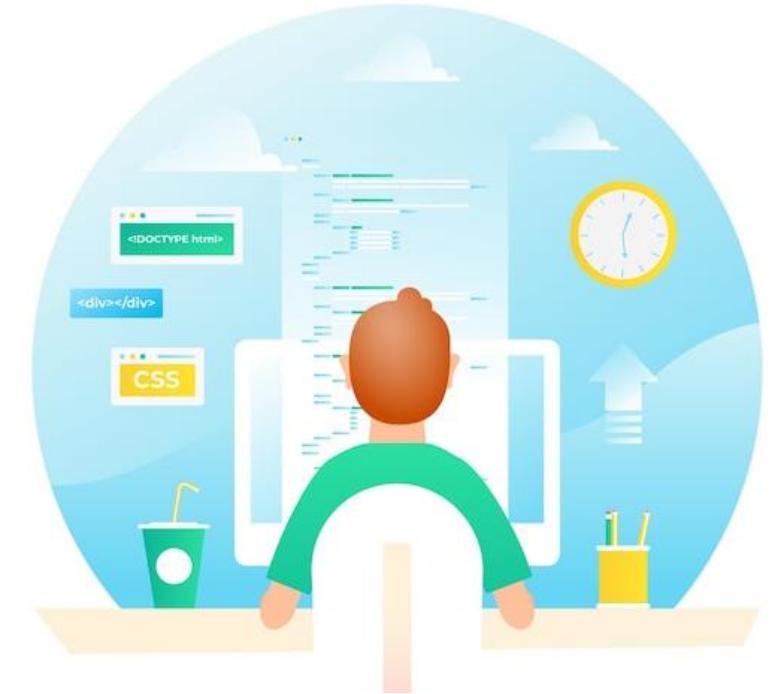
- ✓ SSR is a technique where the React components are rendered on the server into fully formed HTML pages before being sent to the client's browser.
- ✓ This approach improves initial page load speed and SEO since the content is immediately available to search engines and users.
- ✓ ASP.NET Core can serve these rendered pages, enhancing performance especially for content-heavy or SEO-sensitive applications.
- ✓ SSR requires more server resources as the server generates the UI on each request.



# Server-Side Rendering (SSR) and Client-Side Rendering (CSR)

## Client-Side Rendering (CSR)

- ✓ CSR involves sending a minimal HTML shell to the client, with JavaScript code that runs in the browser to dynamically build and update the user interface.
- ✓ React components load and render entirely on the client side after the initial HTML page is received.
- ✓ This approach allows for rich interactivity and smooth user experiences but may have slower initial load times and can be less SEO-friendly unless handled properly.



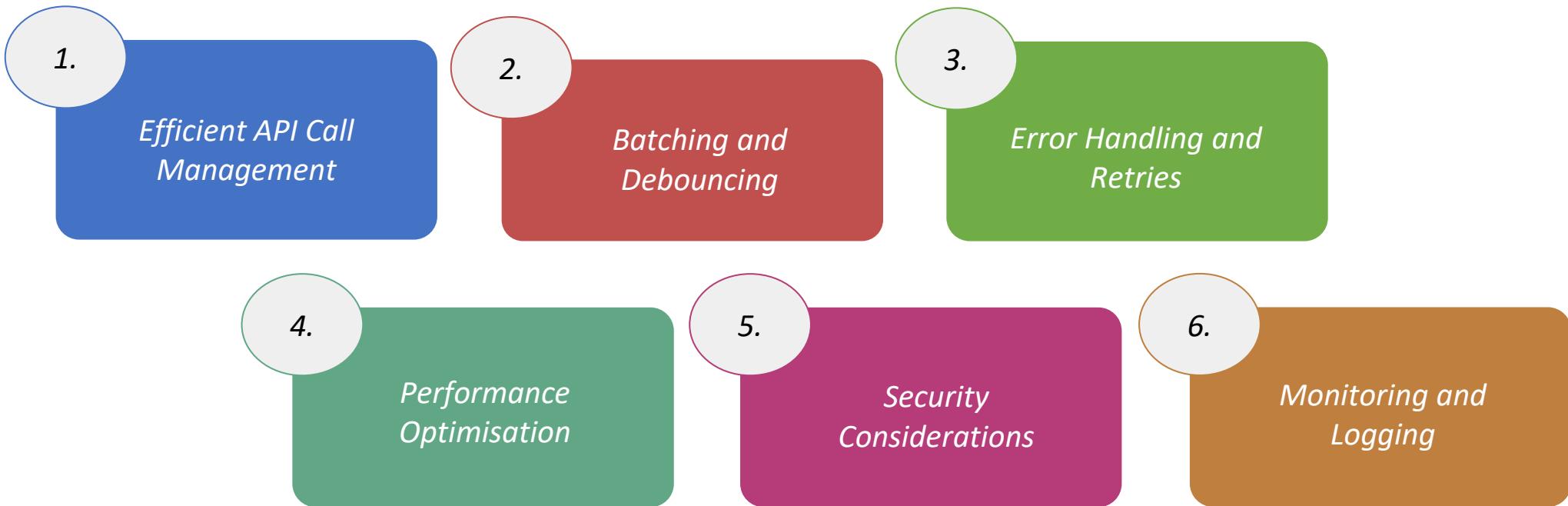
# Server-Side Rendering (SSR) and Client-Side Rendering (CSR)

## Key Difference

Aspect	Server-Side Rendering (SSR)	Client-Side Rendering (CSR)
<b>Rendering Location</b>	Rendered on the server before sending HTML to client	Rendered in the browser using JavaScript after page load
<b>Initial Load Time</b>	Faster initial page load since full HTML is sent	Slower initial load due to JavaScript execution
<b>SEO Friendliness</b>	Better SEO because content is available in HTML	Poorer SEO unless additional steps like prerendering
<b>User Experience</b>	Initial interaction might be slower, but content visible immediately	More dynamic and interactive after load
<b>Use Case</b>	Suitable for content-heavy, SEO-critical pages	Suitable for highly interactive applications
<b>Examples in ASP.NET Core</b>	Using ReactJS.NET or Next.js with ASP.NET Core backend	Using Create React App or React SPA hosted on ASP.NET Core

# Handling API Calls and Performance Considerations

- ✓ In modern web applications built with ASP.NET Core and React.js, efficient handling of API calls is crucial for a responsive user experience and scalable architecture. Below are key considerations:



# Quiz



**Question 1: What is one key advantage of using React.js with ASP.NET Core?**

- A) Combining front-end and back-end code into a single file
- B) React's virtual DOM optimizes rendering for better performance
- C) React eliminates the need for server-side code
- D) ASP.NET Core replaces React for UI rendering



# Quiz



**Question 2: Which React feature helps in navigating between different views without refreshing the entire web application?**

- A) React Router
- B) Redux
- C) Context API
- D) useState Hook



# Quiz



**Question 3: What is the main difference between Server-Side Rendering (SSR) and Client-Side Rendering (CSR) in React applications?**

- A) SSR renders UI in the browser, CSR renders UI on the server
- B) SSR sends fully rendered HTML from server; CSR builds UI in browser with JavaScript
- C) CSR improves SEO more than SSR
- D) SSR eliminates the need for JavaScript



# Q&A Session

**Question 1:** Why is state management important in React when used with ASP.NET Core?



**Question 2:** What are the benefits of separating front-end UI (React) and back-end services (ASP.NET Core) in development?

# Q&A Session

**Question 3:** What role does React's component-based architecture play in application development?



**Question 4:** How does Server-Side Rendering (SSR) improve SEO for React applications?

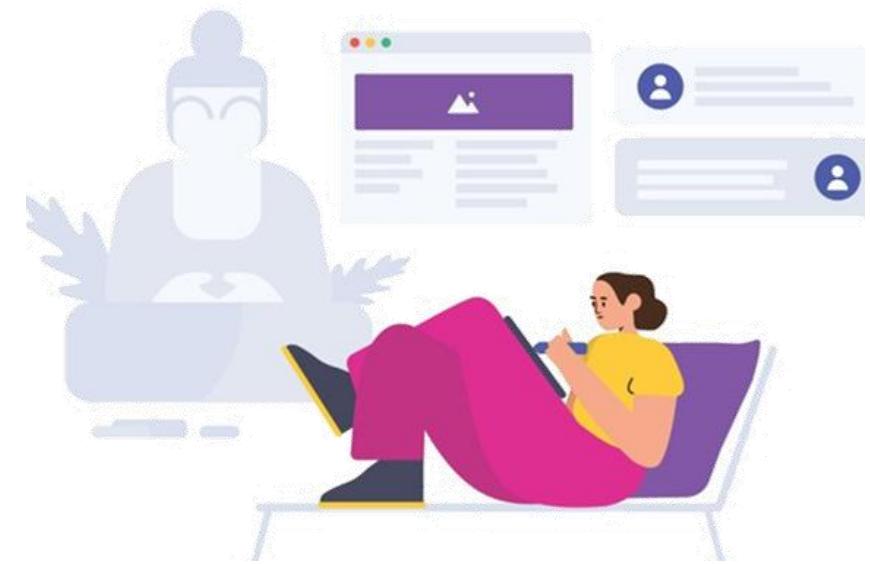
# Module 3: Working with Web APIs in ASP.NET Core

- Introduction to RESTful Web APIs
- Handling HTTP Methods and Status Codes
- Structuring Web APIs in ASP.NET Core
- Securing APIs



# Introduction to RESTful Web APIs

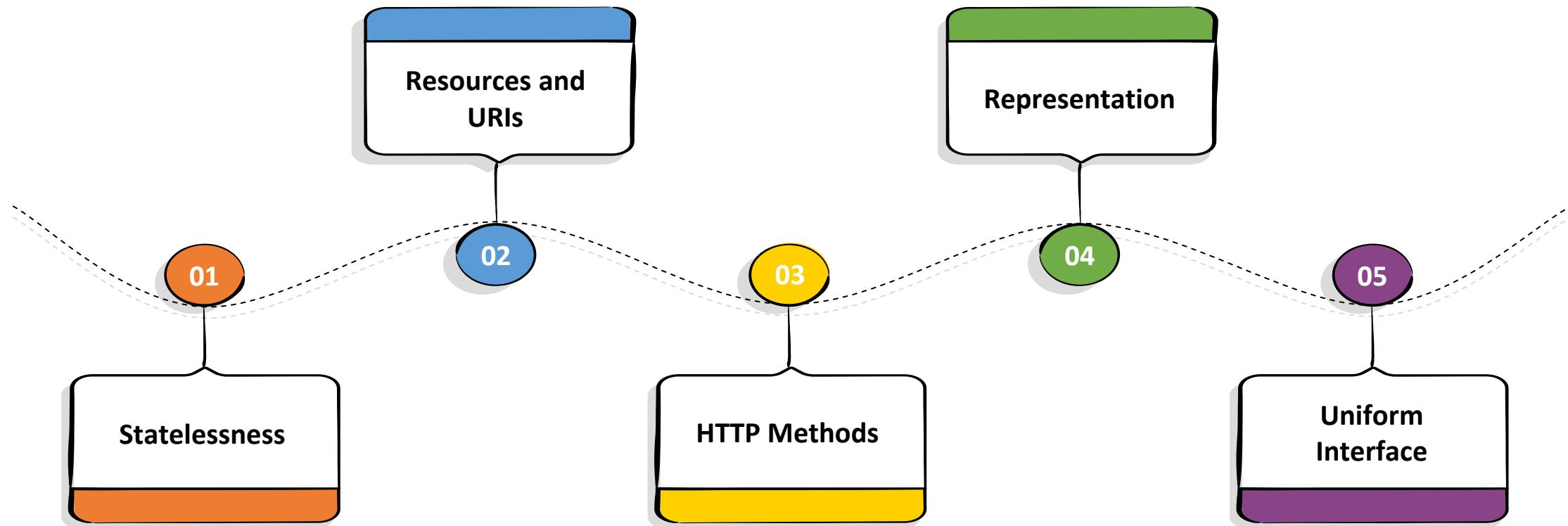
- ✓ RESTful Web APIs are a standardised approach to building web services that allow different software applications to communicate over the internet using HTTP protocols.
- ✓ REST stands for Representational State Transfer, a design philosophy that emphasises stateless communication, resource-based URLs, and the use of standard HTTP methods such as GET, POST, PUT, DELETE, and PATCH.
- ✓ This architectural style promotes scalability, simplicity, and loose coupling between client and server.
- ✓ It enables developers to create flexible and interoperable services that can be easily consumed by various types of clients.



# Introduction to RESTful Web APIs

(Continued)

- ✓ *The following are the key concepts of RESTful APIs:*



# Introduction to RESTful Web APIs

1. **Statelessness:** Each API request contains all the information needed to process it, and the server does not store client context between requests.
2. **Resources and URIs:** Every resource (e.g., users, products) is identified by a unique URI (Uniform Resource Identifier).
3. **HTTP Methods:** RESTful APIs use standard HTTP methods to perform CRUD operations — Create (POST), Read (GET), Update (PUT/PATCH), and Delete (DELETE).
4. **Representation:** Resources can be represented in various formats, most commonly JSON or XML, allowing flexibility for clients.
5. **Uniform Interface:** REST enforces a uniform and consistent interface, simplifying interaction and decoupling client and server.

# Introduction to RESTful Web APIs

## ***ASP.NET Core and RESTful APIs***

- ✓ ASP.NET Core provides a powerful framework to build RESTful APIs with features such as routing, model binding, validation, authentication, and middleware support.
- ✓ It enables developers to create scalable, maintainable, and secure APIs that can serve various client applications including web, mobile, and IoT devices.
- ✓ Its modular architecture allows easy integration of third-party libraries and tools to enhance functionality.
- ✓ Additionally, ASP.NET Core supports asynchronous programming, improving performance and responsiveness in high-load scenarios.



# Handling HTTP Methods and Status Codes

- ✓ HTTP methods define the type of operation a client wants to perform on a resource in a RESTful API.
- ✓ They specify the intended action, allowing servers to understand how to process the incoming request appropriately.
- ✓ Each HTTP method has a distinct semantic meaning, guiding how data should be handled, whether it's retrieving, creating, updating, or deleting resources.
- ✓ Proper use of these methods helps maintain clarity, consistency, and adherence to REST principles, ensuring predictable and standardised communication between clients and servers.



# Handling HTTP Methods and Status Codes

## HTTP Methods

1. **GET:** Retrieve data from the server.
  2. **POST:** Create a new resource.
  3. **PUT:** Update an existing resource completely.
  4. **PATCH:** Partially update an existing resource.
  5. **DELETE:** Remove a resource.
- ASP.NET Core allows developers to handle these HTTP methods explicitly using routing and controller actions decorated with method-specific attributes such as [HttpGet], [HttpPost], [HttpPut], [HttpPatch], and [HttpDelete].

# Handling HTTP Methods and Status Codes

## HTTP Status Codes

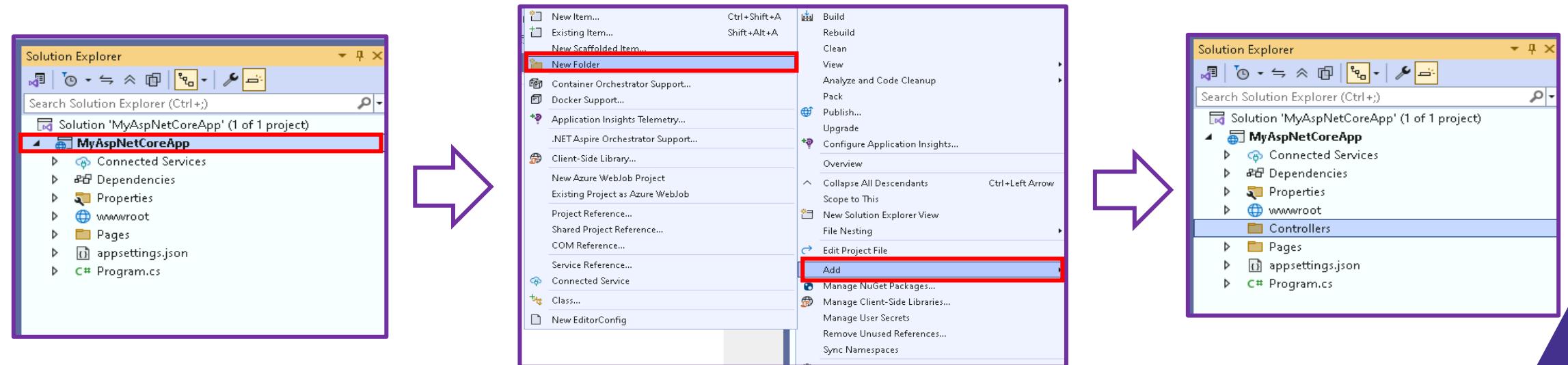
- ✓ Status codes communicate the result of the HTTP request from the server to the client. They provide standardised information about whether a request was successful, encountered an error, or requires further action.
  - ✓ Proper use of status codes enhances client-server communication by enabling clients to understand and handle responses effectively.
1. **1xx (Informational)**: Request received, continuing process.2xx (Success): Successful request (e.g., 200 OK, 201 Created).
  2. **3xx (Redirection)**: Further action needed to complete the request.
  3. **4xx (Client Errors)**: Request has errors (e.g., 400 Bad Request, 404 Not Found).
  4. **5xx (Server Errors)**: Server failed to fulfill a valid request (e.g., 500 Internal Server Error)

# Structuring Web APIs in ASP.NET Core

- ✓ Structuring Web APIs in ASP.NET Core involves organising your code and resources to build scalable and maintainable services. A well-structured API ensures clear separation of concerns, easy navigation, and efficient management of endpoints and data flow.
- ✓ *The following are the steps for structuring Web APIs in ASP.NET Core:*

**Step 1:** In the **Solution Explorer** pane on the right, right-click on the project name

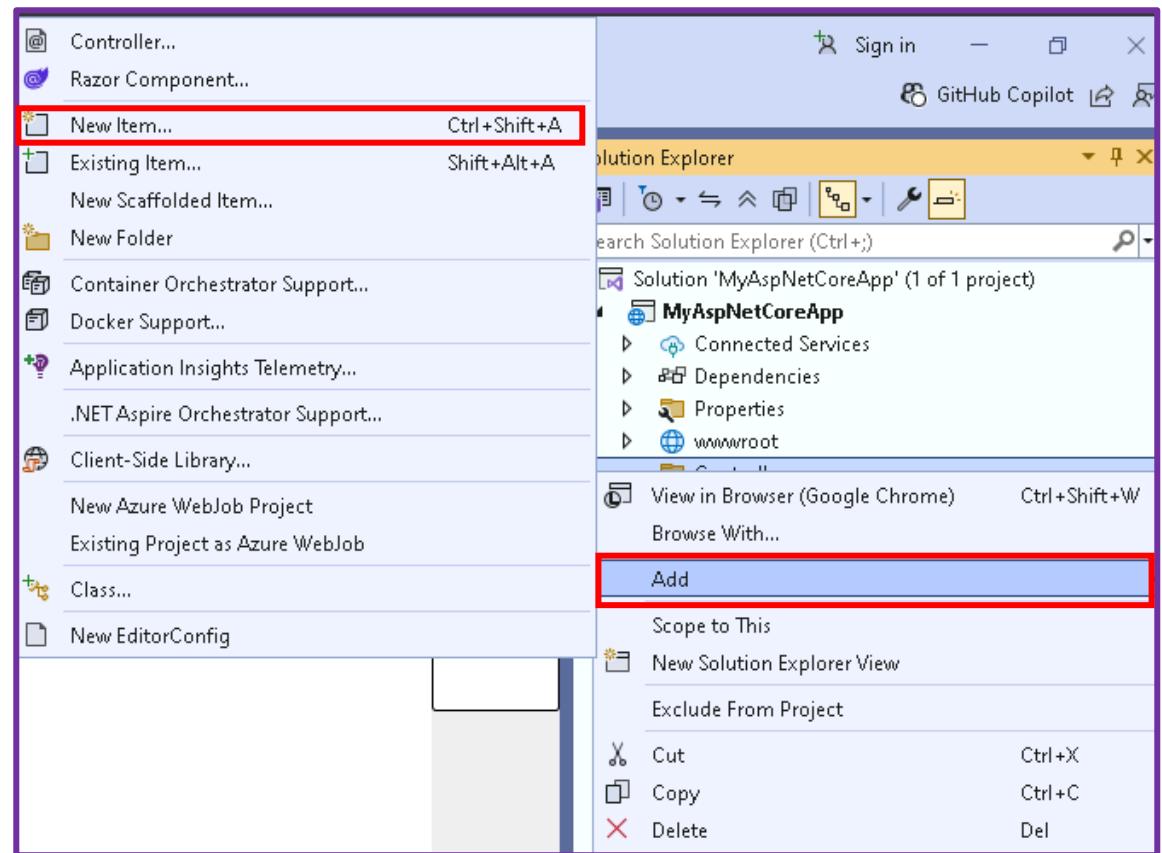
**Step 2:** Select **Add > New Folder**. Name it Controllers



# Structuring Web APIs in ASP.NET Core

**Step 3:** Right-click on the newly created Controllers folder

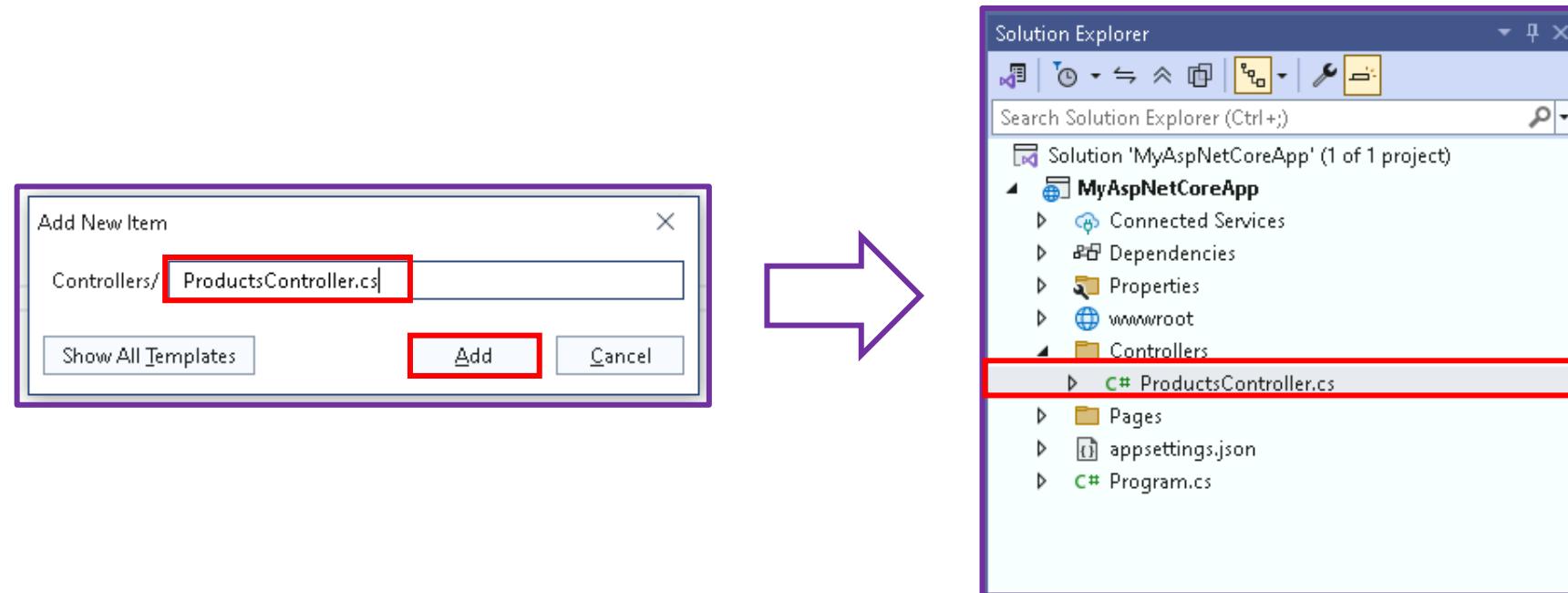
**Step 4:** Select Add > New Item...



# Structuring Web APIs in ASP.NET Core

**Step 5:** Enter a filename

**Step 6:** Click the Add button



# Structuring Web APIs in ASP.NET Core

**Step 7:** Add the following code template to create a basic API controller

**Step 8:** Save the file by pressing **Ctrl + S**

The screenshot shows the Visual Studio IDE interface. On the left is the code editor window titled "ProductsController.cs" under "MyAspNetCoreApp". The code defines a basic API controller:

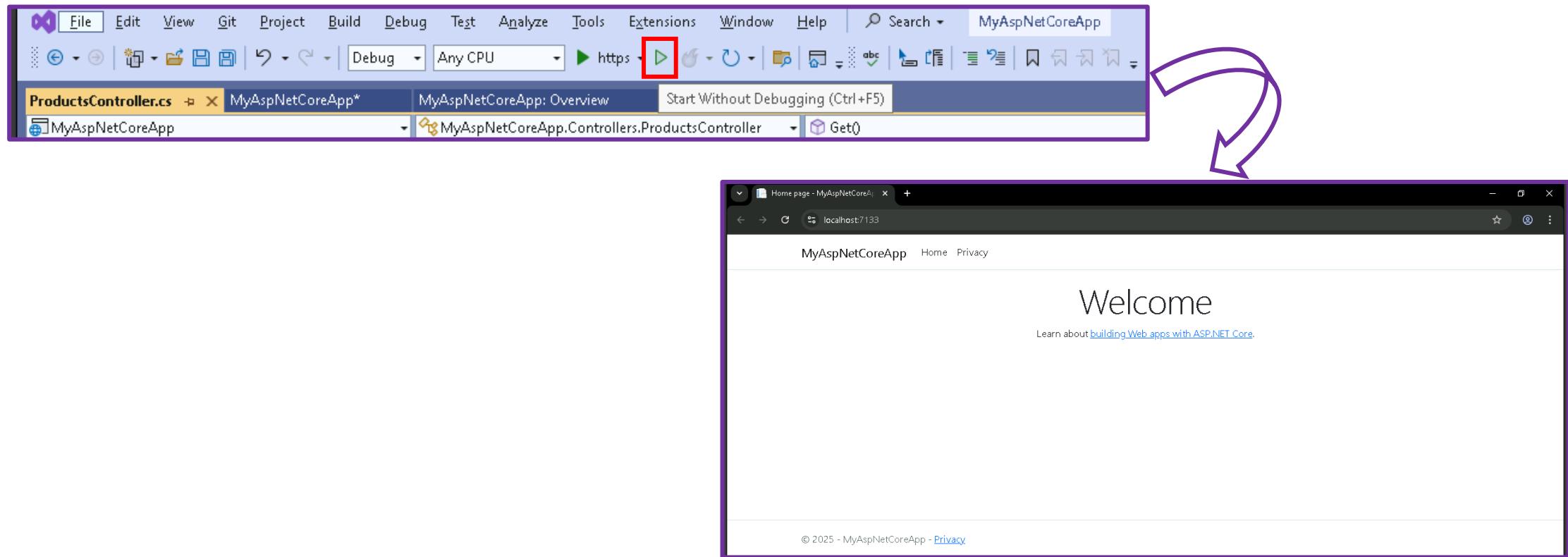
```
1  using Microsoft.AspNetCore.Mvc;
2  using System.Collections.Generic;
3
4  namespace MyAspNetCoreApp.Controllers
5  {
6      [ApiController]
7      [Route("api/[controller]")]
8      public class ProductsController : ControllerBase
9      {
10         [HttpGet]
11         public IEnumerable<string> Get()
12         {
13             return new string[] { "Product1", "Product2", "Product3" };
14         }
15     }
16 }
17
```

The Solution Explorer window on the right shows the project structure for "MyAspNetCoreApp". It includes a "Controllers" folder containing the "ProductsController.cs" file, along with "Pages", "appsettings.json", and "Program.cs".

# Structuring Web APIs in ASP.NET Core

**Step 9:** Click the green **Play** button at the top toolbar to start debugging

**Step 10:** Open your browser to see the API response



# Securing APIs

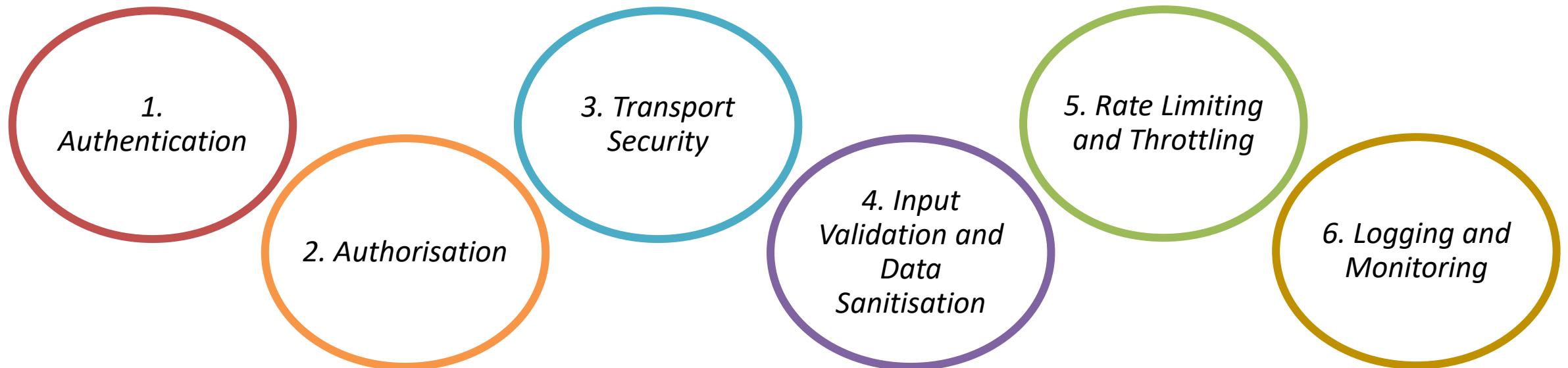
- ✓ Securing APIs is critical to protect sensitive data, prevent unauthorised access, and ensure that only authenticated and authorised clients can consume the services.
- ✓ APIs, especially those exposed over the internet, face various security threats including data breaches, injection attacks, and denial of service.
- ✓ Therefore, implementing robust security mechanisms is fundamental in modern API development.
- ✓ Effective security not only safeguards data integrity but also builds trust with users and stakeholders.
- ✓ Additionally, it helps organisations comply with regulatory requirements and industry standards.



# Securing APIs

(Continued)

- ✓ *The following are the key security concepts of APIs:*



# Securing APIs

## 1. Authentication:

- The process of verifying the identity of a client or user accessing the API. Common methods include API keys, OAuth 2.0, OpenID Connect, JWT (JSON Web Tokens), and Basic Authentication.
- Authentication ensures that only legitimate users or services can access the API endpoints.

## 2. Authorisation:

- After authentication, authorisation determines whether the authenticated client has permission to perform specific actions or access certain resources.
- This can be role-based (RBAC), policy-based, or claim-based, controlling fine-grained access within the API.

# Securing APIs

## 3. Transport Security:

- Securing the communication channel between the client and API server using protocols like HTTPS to encrypt data in transit and prevent eavesdropping or tampering.

## 4. Input Validation and Data Sanitisation:

- Ensuring that incoming data conforms to expected formats and rejecting malicious input to prevent injection attacks such as SQL injection or cross-site scripting (XSS).

## 5. Rate Limiting and Throttling:

- Controlling the number of requests a client can make in a given timeframe to protect APIs from abuse and denial-of-service (DoS) attacks.

# Securing APIs

## 6. Logging and Monitoring:

- Keeping track of access and usage patterns to detect suspicious activities, audit usage, and support incident response.
- Effective logging helps identify performance bottlenecks and system errors early, enabling proactive maintenance.
- Continuous monitoring ensures the API's health and security posture, facilitating timely alerts and automated responses to potential threats.



# Quiz



**Question 1:** What is the main principle of RESTful APIs?

- A) Maintaining client state on the server
- B) Stateless communication and resource-based URLs
- C) Using SOAP for message formatting
- D) Only supporting XML data format



# Quiz



**Question 2:** Which HTTP method is used to create a new resource in a RESTful API?

- A) GET
- B) POST
- C) PUT
- D) DELETE



# Quiz



**Question 3:** What security practice involves controlling the number of API requests a client can make within a certain time frame?

- A) Authentication
- B) Input Validation
- C) Rate Limiting and Throttling
- D) Logging and Monitoring



# Q&A Session

**Question 1:** What does it mean that RESTful APIs are stateless?



**Question 2:** How does ASP.NET Core help in handling different HTTP methods for Web APIs?

# Q&A Session

**Question 3:** Why is transport security important for securing APIs?



**Question 4:** What are some of the key steps in structuring Web APIs in ASP.NET Core?

# Module 4: Authentication and Authorization in ASP.NET Core

- Understanding Authentication and Authorization
- Implementing JWT Authentication
- Role-Based Authorization
- Securing APIs and Web Apps



# Understanding Authentication and Authorization

## Authentication

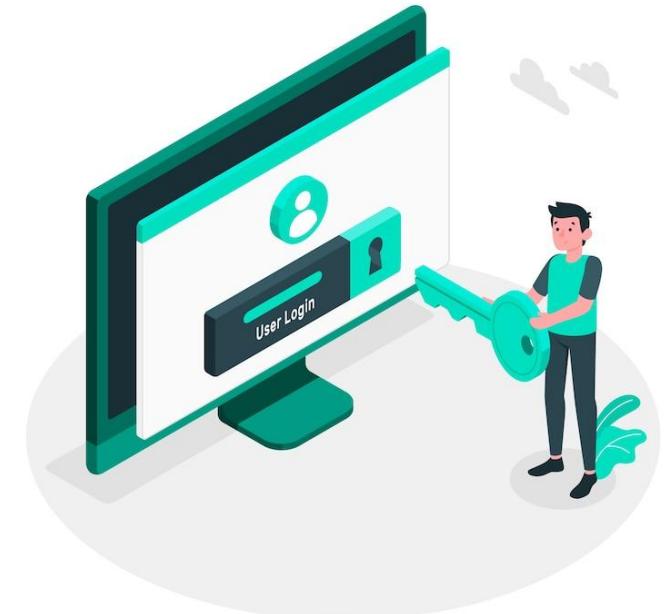
- ✓ Authentication is the process of verifying the identity of a user or system trying to access an application.
- ✓ It answers the question, “Who are you?” This involves checking credentials such as usernames and passwords, biometric data, or security tokens to confirm that the user is who they claim to be.
- ✓ Successful authentication creates a security context that represents the user within the application.
- ✓ Without authentication, an application cannot distinguish between different users or protect sensitive resources.



# Understanding Authentication and Authorization

## Authorization

- ✓ Authorization is the process of determining what an authenticated user or system is allowed to do within an application.
- ✓ It answers the question, “What can you access?” Once a user’s identity is confirmed, authorization enforces rules that restrict or grant access to resources, features, or data.
- ✓ This can be based on user roles (e.g., Admin, Editor) or policies that check specific claims or conditions.
- ✓ Proper authorization ensures users only perform actions permitted for their access level.



# Understanding Authentication and Authorization

(Continued)

## ***Key Points***

1. Authentication verifies user identity; authorization controls access rights.
2. Authentication establishes the user's security context; authorization uses this context to enforce permissions.
3. Common authentication methods include passwords, tokens (like JWT), and multi-factor authentication.
4. Authorization can be role-based or policy-based, allowing flexible access control.
5. ASP.NET Core provides built-in middleware to handle both authentication and authorization.
6. Separating authentication from authorization improves security and application design.

# Implementing JWT Authentication

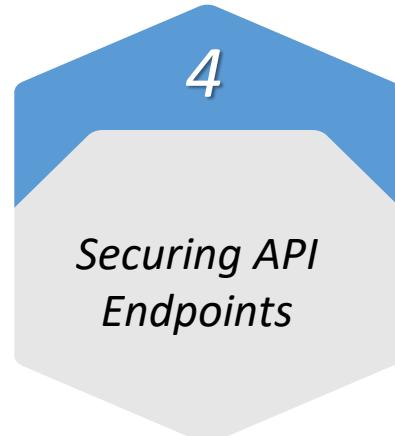
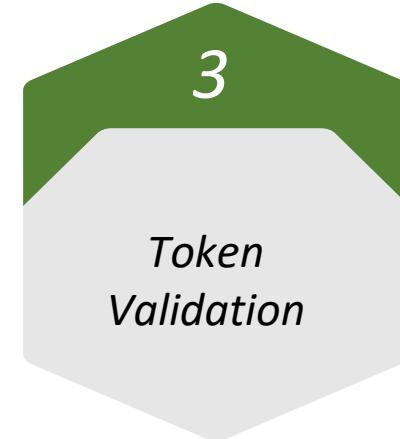
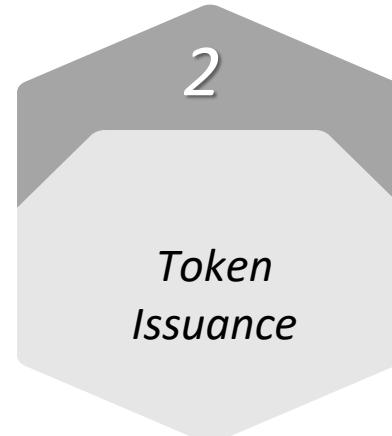
- ✓ JWT (JSON Web Token) Authentication is a widely used method to secure APIs and web applications by transmitting encoded JSON objects that verify the identity of clients.
- ✓ It provides a stateless and compact way to authenticate users without maintaining session state on the server.
- ✓ In ASP.NET Core, implementing JWT authentication involves configuring the authentication middleware to validate incoming JWT tokens in the request headers.
- ✓ This validation ensures that only clients with valid tokens issued by trusted authorities can access protected resources.



# Implementing JWT Authentication

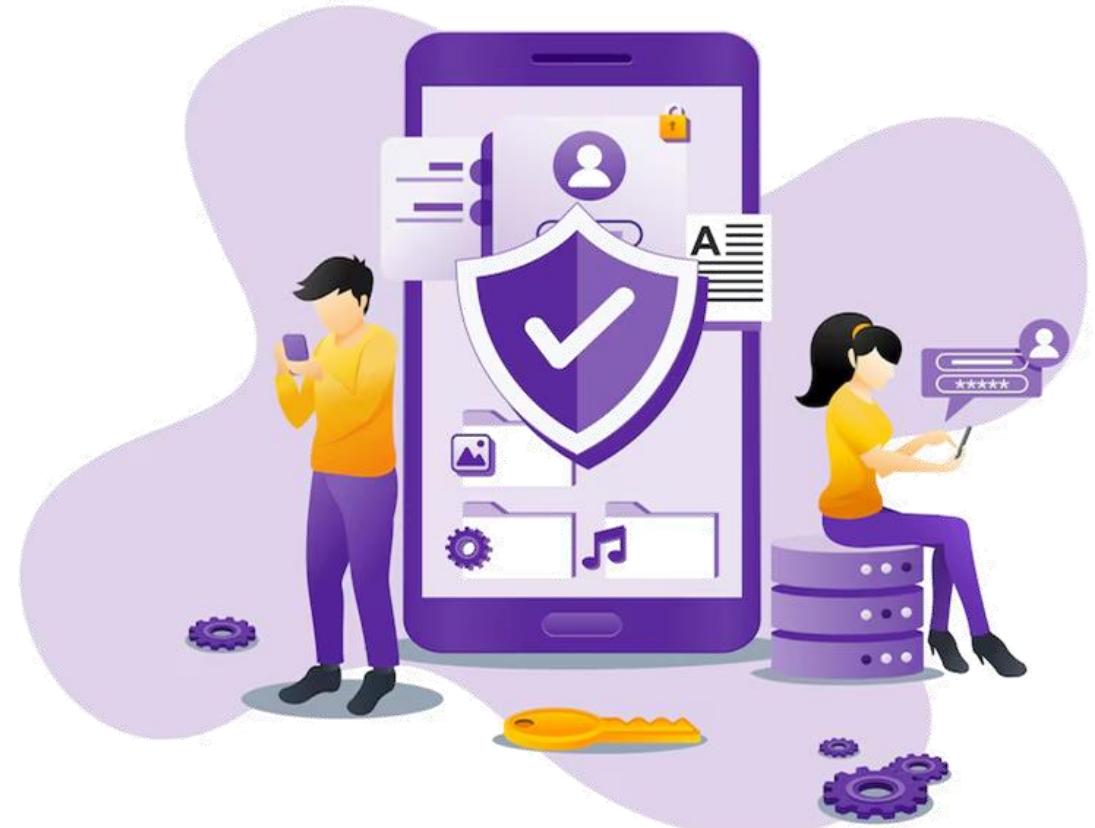
(Continued)

- ✓ *The following are the key concepts for implementing JWT authentication:*



# Implementing JWT Authentication

- Configuring JWT Bearer Authentication:** Setting up the middleware to read and validate JWT tokens in HTTP Authorization headers.
- Token Issuance:** Typically handled by an identity provider or authentication server that generates tokens after user login.
- Token Validation:** Verifying token signature, issuer, audience, and expiry before allowing access.
- Securing API Endpoints:** Applying authentication and authorization policies to protect APIs and restrict access based on user roles or claims.



# Role-Based Authorization

- ✓ Role-Based Authorization (RBA) is a widely used access control mechanism that restricts system access to users based on their assigned roles.
- ✓ In ASP.NET Core, RBA enables developers to define roles within an application and control access to resources, actions, or areas of the application based on these roles.
- ✓ This approach simplifies permission management by grouping users into roles with predefined permissions, rather than assigning rights to individual users.
- ✓ It promotes a clear separation of concerns by decoupling authentication from authorization decisions. Additionally, it supports dynamic and flexible access control, which can adapt as user roles evolve over time.



# Role-Based Authorization

(Continued)

- ✓ *The following are the key concepts of role-based authorization:*



# Role-Based Authorization

- 1. Roles:** Logical groupings that represent a set of permissions, such as “Admin,” “Manager,” or “User.” Each role defines what actions or resources its members can access.
- 2. Claims:** Data associated with a user’s identity that can include roles as part of the claims-based identity system in ASP.NET Core.
- 3. Authorization Policies:** Policies define access rules, often based on roles, and are used by the framework to enforce authorization.
- 4. Attributes:** ASP.NET Core provides the [Authorize] attribute, which can be configured to restrict access to users in specific roles using syntax like [Authorize(Roles = "Admin,Manager")].



# Role-Based Authorization

## ***How It Works in ASP.NET Core***

- ✓ When a user attempts to access a protected resource, ASP.NET Core checks the user's roles against the roles specified in the authorization attributes or policies.
- ✓ If the user is a member of the required role(s), access is granted; otherwise, it is denied. This process integrates seamlessly with ASP.NET Core's authentication mechanisms, enabling secure and maintainable access control.

## ***Benefits***

1. Simplifies access management by assigning permissions to roles rather than individuals.
2. Enhances security by ensuring users can only perform actions allowed by their role.
3. Facilitates scalability and maintainability in applications with multiple user types.
4. Integrates smoothly with other ASP.NET Core security features like claims-based identities and policies.

# Securing APIs and Web Apps

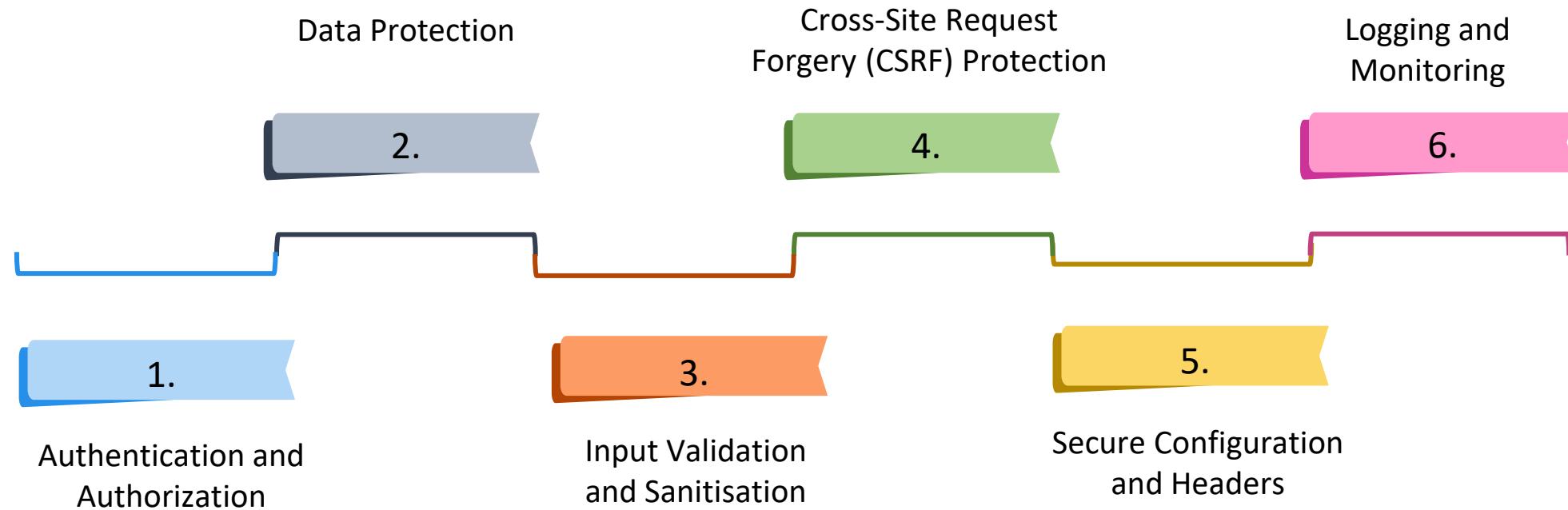
- ✓ Securing APIs and web applications is essential to protect sensitive data, maintain user privacy, and ensure the integrity and availability of services.
- ✓ Both APIs and web apps face a range of security threats such as unauthorised access, data breaches, injection attacks, cross-site scripting (XSS), and cross-site request forgery (CSRF).
- ✓ Implementing comprehensive security strategies is crucial to mitigate these risks.
- ✓ Adopting a layered security approach helps in defending against sophisticated attacks and minimising vulnerabilities.
- ✓ Continuous security assessments and updates are necessary to stay ahead of evolving threats.



# Securing APIs and Web Apps

(Continued)

- ✓ ***The following are the key security measures:***



# Securing APIs and Web Apps

## 1. Authentication and Authorization:

- Implement strong authentication mechanisms to verify user identities and use authorization to control access levels. Techniques include OAuth 2.0, OpenID Connect, JWT tokens, and role-based access control (RBAC).

## 2. Data Protection:

- Use encryption (e.g., HTTPS/TLS) to secure data in transit and at rest. Sensitive data such as passwords should be hashed and salted to prevent compromise.

## 3. Input Validation and Sanitization:

- Validate all incoming data to prevent injection attacks and sanitise outputs to avoid XSS vulnerabilities.

# Securing APIs and Web Apps

## 4. Cross-Site Request Forgery (CSRF) Protection:

- Implement anti-CSRF tokens to ensure that requests made to the web app are intentional and not forged.

## 5. Secure Configuration and Headers:

- Use security headers like Content Security Policy (CSP), Strict-Transport-Security (HSTS), and X-Content-Type-Options to enhance security posture.

## 6. Logging and Monitoring:

- Continuously monitor access and usage patterns to detect anomalies and respond to incidents promptly.



# Quiz



**Question 1:** What is the main purpose of authentication in ASP.NET Core?

- A) Determining what resources a user can access
- B) Verifying the identity of a user or system
- C) Encrypting data during transmission
- D) Logging user activity



# Quiz



**Question 2:** What does JWT stand for and what is its primary use in ASP.NET Core?

- A) Java Web Token, used for encrypting data
- B) JSON Wide Transfer, used for large data transfers
- C) JavaScript Web Transport, used for real-time communication
- D) JSON Web Token, used for stateless authentication



# Quiz



**Question 3:** How does Role-Based Authorization control access in ASP.NET Core?

- A) By checking the user's IP address
- B) By verifying user's assigned roles against required permissions
- C) By encrypting user data
- D) By logging all user actions



# Q&A Session

**Question 1:** What is the difference between authentication and authorization?



**Question 2:** How does JWT Authentication work in ASP.NET Core?

# Q&A Session

**Question 3:** What are the benefits of Role-Based Authorization in ASP.NET Core?



**Question 4:** What are some key security measures to protect APIs and web applications?

# Module 5: Entity Framework Core and Data Access

- Overview of Entity Framework Core



# Overview of Entity Framework Core

- ✓ Entity Framework Core is a modern, lightweight, and extensible Object-Relational Mapper for .NET applications. It enables developers to work with databases using .NET objects, eliminating most of the data-access code typically required.
- ✓ EF Core supports LINQ queries, change tracking, updates, and schema migrations, providing a streamlined way to manage data access. The following are the key features and advantages of Entity Framework Core:



# Overview of Entity Framework Core

## 1. Cross-Platform Support:

- EF Core works seamlessly on Windows, Linux, and macOS, making it suitable for a wide range of application environments including cloud and containerised deployments.

## 2. Code-First and Database-First Approaches:

- Developers can either create the database schema from their code models (Code-First) or generate code models from an existing database (Database-First), offering flexibility in development workflows.

## 3. LINQ Integration:

- EF Core allows querying the database using LINQ syntax, enabling strongly typed queries with compile-time checking and IntelliSense support, enhancing productivity and reducing errors.

# Overview of Entity Framework Core

## 4. Change Tracking:

- EF Core automatically tracks changes made to entities during the application's runtime, simplifying the process of persisting updates back to the database.

## 5. Migrations Support:

- The framework provides tools to incrementally update the database schema while preserving existing data, supporting agile development and continuous integration.

## 6. Extensibility and Performance:

- EF Core supports extension points for custom conventions, logging, and interceptors, and is optimised for performance with features like batching and compiled queries.

# Quiz



## Question 1: What is one key advantage of using Entity Framework Core (EF Core)?

- A) It requires manual SQL query writing for all database operations
- B) It does not support schema migrations
- C) It only supports Windows operating systems
- D) It enables developers to work with databases using .NET objects



# Quiz



**Question 2:** Which approach allows developers to create the database schema from their code models in EF Core?

- A) Database-First
- B) Code-First
- C) Model-First
- D) Schema-First



# Quiz



**Question 3: What feature in EF Core helps to keep track of changes made to entities during application runtime?**

- A) LINQ Queries
- B) Schema Migrations
- C) Change Tracking
- D) Query Optimization



# Q&A Session

**Question 1:** What platforms does Entity Framework Core support?



**Question 2:** How does EF Core support database schema evolution during development?

# Q&A Session

**Question 3:** How does EF Core enhance productivity when querying databases?



**Question 4:** What performance optimizations does EF Core offer?

# Module 6: Debugging and Troubleshooting in ASP.NET Core

- Debugging Techniques in ASP.NET Core
- Using Logging for Application Monitoring
- Resolving Common Errors
- Exception Handling Best Practices



# Debugging Techniques in ASP.NET Core

- ✓ Effective debugging is essential for identifying and resolving issues in ASP.NET Core applications to ensure reliability and optimal performance.
- ✓ It involves a systematic approach to inspect code behaviour, trace errors, and analyse application flow during development and after deployment. The following are the key debugging techniques used in ASP.NET Core development:

01

*Using Visual Studio Debugger*

02

*Logging with ILogger Interface*

03

*Remote Debugging*

04

*Exception Handling and Middleware*

05

*Diagnostic Tools and Profilers*

06

*Use of Developer Exception Page*

# Debugging Techniques in ASP.NET Core

## 1. Using Visual Studio Debugger:

- Leverage the powerful Visual Studio debugger to set breakpoints, step through code, inspect variables, and evaluate expressions, allowing precise control over the debugging process.

## 2. Logging with ILogger Interface:

- Implement structured logging using the built-in ILogger interface to capture detailed runtime information and error messages, facilitating easier diagnosis without halting the application.

## 3. Remote Debugging:

- Attach the debugger to an ASP.NET Core application running on a remote server or container, enabling troubleshooting in production-like environments without local replication.

# Debugging Techniques in ASP.NET Core

## 4. Exception Handling and Middleware:

- Use global exception handling middleware to catch and log unhandled exceptions consistently, providing centralised error management and detailed error responses.

## 5. Diagnostic Tools and Profilers:

- Utilise diagnostic tools like dotnet-trace, dotnet-counters, and performance profilers to monitor application performance metrics and trace runtime behaviour for deeper analysis.

## 6. Use of Developer Exception Page:

- Enable the Developer Exception Page in development environments to display detailed exception information and stack traces directly in the browser, aiding quick issue identification.

# Using Logging for Application Monitoring

- ✓ Effective logging is a cornerstone of application monitoring, providing insight into the behaviour and health of ASP.NET Core applications. By capturing structured and meaningful logs, teams can proactively respond to errors and improve overall system reliability.
- ✓ Logging also facilitates auditing and compliance by maintaining detailed records of application events. The following are the key aspects and practical steps for using logging for effective monitoring:

Configure Logging Providers  
1

Use Built-in ILogger Interface  
2

Structured Logging for Better Insights  
3

Integrate Logging with Azure Monitor  
4

Implement Log Filtering and Configuration  
5

Centralised Log Storage and Analysis  
6

# Using Logging for Application Monitoring

## 1. Configure Logging Providers:

- ASP.NET Core supports multiple logging providers such as Console, Debug, EventSource, and third-party systems like Serilog and NLog.

## 2. Use Built-in ILogger Interface:

- The ILogger<T> interface provides a flexible and consistent way to write log messages across your application. It supports different log levels like Trace, Debug, Information, Warning, Error, and Critical, allowing you to classify the severity of events.

## 3. Structured Logging for Better Insights:

- Implement structured logging by including contextual data with log messages. This practice enhances log searchability and enables advanced analytics in log management systems such as Azure Monitor or ELK stack.

# Using Logging for Application Monitoring

## 4. Integrate Logging with Azure Monitor:

- For applications deployed on Azure, integrate your logging with Azure Monitor and Application Insights. This integration offers real-time monitoring, automatic performance tracking, and intelligent alerts based on log patterns.

## 5. Implement Log Filtering and Configuration:

- Control the verbosity and sources of logs via configuration files (appsettings.json) or environment-specific settings. This helps in reducing noise and focusing on critical events during troubleshooting.

## 6. Centralised Log Storage and Analysis:

- Use centralised log storage solutions, either cloud-based or on-premises, to aggregate logs from multiple instances of your application. Centralised storage supports easier correlation of events, trend analysis, and faster root cause identification.

# Resolving Common Errors

- ✓ Encountering errors is inevitable, but understanding how to quickly identify and resolve them is crucial for maintaining application stability and developer productivity.
- ✓ Common errors range from configuration mistakes to runtime exceptions, and systematic troubleshooting helps minimise downtime. Mastery of these techniques ensures smoother development cycles and higher-quality applications. The following are the key approaches to resolving common errors:

01

Analysing Exception Messages  
and Stack Traces

02

Using the Built-in Debugger

03

Checking Configuration and  
Dependency Injection Issues

04

Inspecting Middleware and Routing  
Problems

05

Monitoring Logs for Detailed  
Diagnostics

06

Validating External Dependencies  
and APIs

# Resolving Common Errors

## 1. Analysing Exception Messages and Stack Traces:

- Carefully read the exception details and stack traces displayed in the terminal or output window to identify the source of errors. These messages often include the exact file and line number causing the issue, enabling targeted debugging.

## 2. Using the Built-in Debugger:

- Leverage VSCode's debugging capabilities by setting breakpoints, stepping through code, and inspecting variables to understand the application's flow and detect where it deviates from expected behaviour.

## 3. Checking Configuration and Dependency Injection Issues:

- Validate appsettings.json, environment variables, and the Startup.cs configuration for misconfigurations or missing service registrations, which commonly cause errors during application startup or runtime.

# Resolving Common Errors

## 4. Inspecting Middleware and Routing Problems:

- Errors in middleware setup or route definitions can lead to unexpected behaviours or 404 errors. Review the pipeline configuration in Startup.cs and verify route templates to ensure proper request handling.

## 5. Monitoring Logs for Detailed Diagnostics:

- Enable and examine application logs through ILogger implementations to track warnings, errors, and critical failures that might not surface immediately but affect app performance or stability.

## 6. Validating External Dependencies and APIs:

- Check the connectivity and response of external services, databases, or APIs integrated into the application to rule out issues beyond the codebase causing errors or timeouts.

# Exception Handling Best Practices

- ✓ Effective exception handling is crucial for building resilient ASP.NET Core applications that can gracefully manage errors and maintain a seamless user experience.
- ✓ Adopting best practices in exception handling helps developers isolate faults, provide meaningful feedback, and avoid application crashes or data corruption. Proper handling also improves maintainability, debugging efficiency, and security by preventing sensitive information leaks.
- ✓ The following are the key best practices for exception handling:



# Exception Handling Best Practices

## 1. Use Global Exception Handling Middleware:

- Implement a centralised middleware (e.g., UseExceptionHandler) to catch unhandled exceptions across the application. This ensures consistent error responses and simplifies logging and monitoring.

## 2. Avoid Catching Generic Exceptions:

- Catch specific exceptions whenever possible rather than using broad catch blocks. This allows for tailored error handling and better identification of the root cause.

## 3. Log Exceptions Effectively:

- Always log exceptions with sufficient context, including stack traces and request information. Use structured logging tools such as Serilog or Microsoft.Extensions.Logging for better traceability.

# Exception Handling Best Practices

## 4. Do Not Swallow Exceptions Silently:

- Avoid empty catch blocks or logging without rethrowing or handling the error. Silent failures can lead to hidden bugs and unpredictable application behaviour.

## 5. Return Meaningful Error Responses:

- For API applications, provide clear and standardised error messages (e.g., HTTP status codes with detailed error payloads) to clients, avoiding exposure of sensitive details.

## 6. Use Exception Filters or ProblemDetails for API:

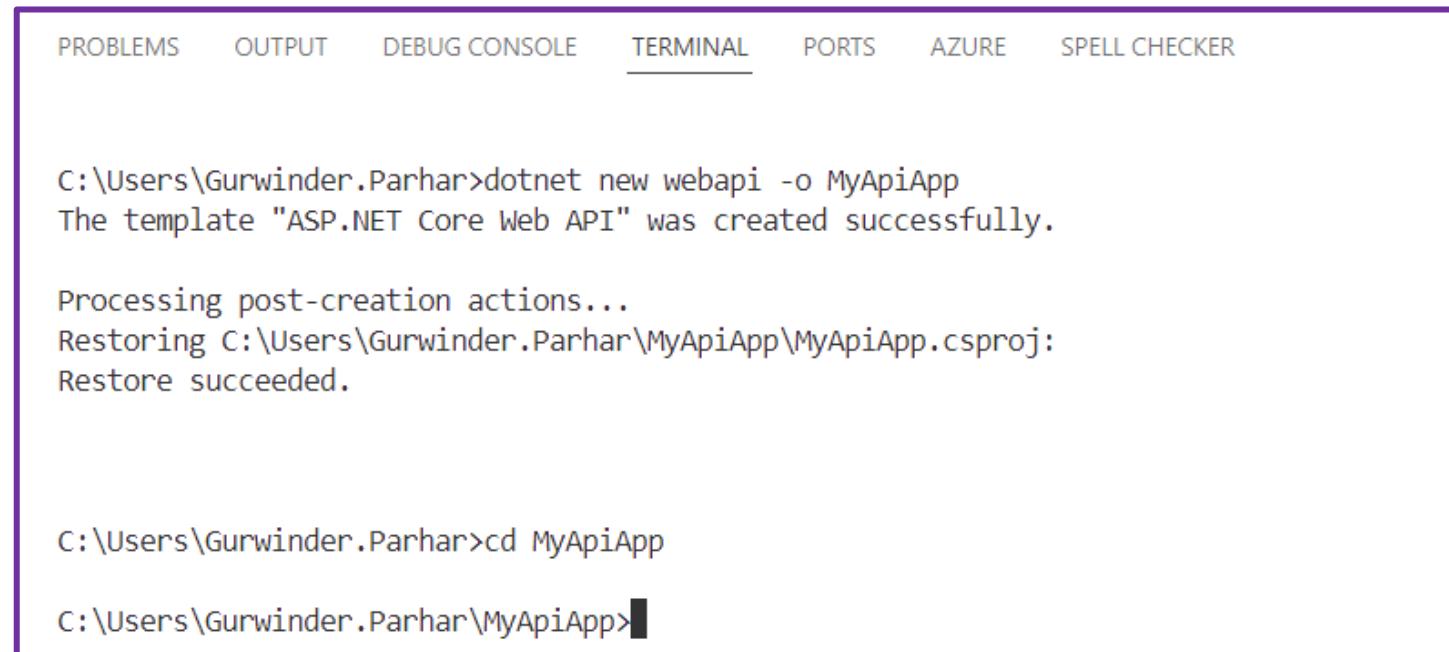
- Leverage ASP.NET Core features like Exception Filters or ProblemDetails to format error responses consistently and support client-side error handling.

# Exception Handling Best Practices

(Continued)

- ✓ The following are the steps to implement Exception Handling Best Practices:

## 1. Create a New ASP.NET Core Project:



The screenshot shows a terminal window with a purple border. At the top, there are tabs: PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL (which is underlined), PORTS, AZURE, and SPELL CHECKER. The terminal itself displays the following text:

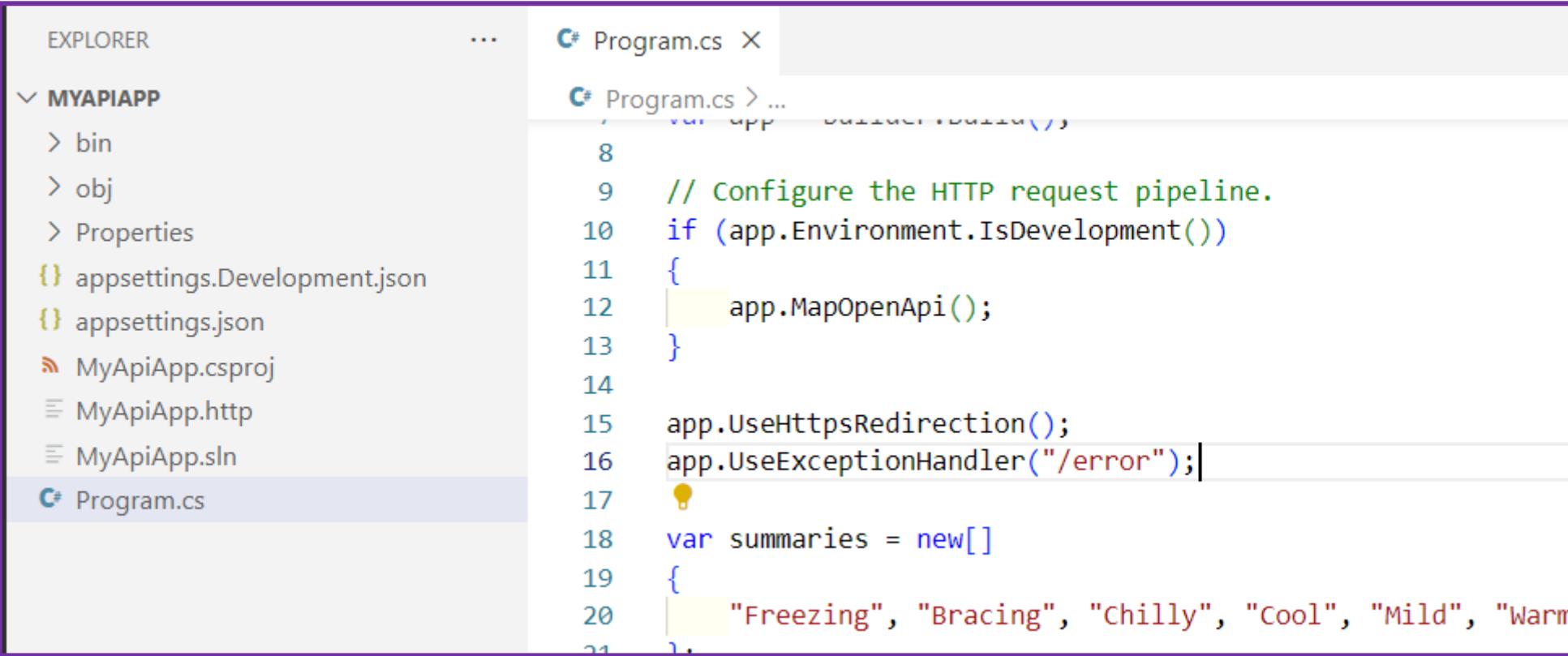
```
C:\Users\Gurwinder.Parhar>dotnet new webapi -o MyApiApp
The template "ASP.NET Core Web API" was created successfully.

Processing post-creation actions...
Restoring C:\Users\Gurwinder.Parhar\MyApiApp\MyApiApp.csproj:
Restore succeeded.

C:\Users\Gurwinder.Parhar>cd MyApiApp
C:\Users\Gurwinder.Parhar\MyApiApp>
```

# Exception Handling Best Practices

- Configure Global Exception Handling Middleware: Open `Startup.cs` or `Program.cs` (depending on project template). Add the following in the middleware pipeline:



The screenshot shows the Visual Studio IDE interface. On the left, the Explorer pane displays the project structure for "MYAPIAPP" with files like bin, obj, Properties, appsettings.Development.json, appsettings.json, MyApiApp.csproj, MyApiApp.http, and MyApiApp.sln. The "Program.cs" file is selected and highlighted in the Explorer pane. On the right, the code editor shows the "Program.cs" file with the following C# code:

```
8
9     // Configure the HTTP request pipeline.
10    if (app.Environment.IsDevelopment())
11    {
12        app.MapOpenApi();
13    }
14
15    app.UseHttpsRedirection();
16    app.UseExceptionHandler("/error");
17
18    var summaries = new[]
19    {
20        "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm"
21    }.
```

# Exception Handling Best Practices

3. **Create ErrorController.cs:** In the Controllers folder, add a new file named **ErrorController.cs** and paste the following code:

The screenshot shows the Visual Studio IDE interface. On the left, the Explorer sidebar displays the project structure for 'MYAPIAPP'. The 'Controllers' folder contains an 'ErrorController.cs' file, which is currently selected and highlighted with a light blue background. The main code editor window on the right shows the content of 'ErrorController.cs'. The code is as follows:

```
1  using Microsoft.AspNetCore.Diagnostics;
2  using Microsoft.AspNetCore.Mvc;
3  using Microsoft.Extensions.Logging;
4
5  [ApiController]
6  public class ErrorController : ControllerBase
7  {
8      private readonly ILogger<ErrorController> _logger;
9
10     public ErrorController(ILogger<ErrorController> logger)
11     {
12         _logger = logger;
13     }
14 }
```

# Exception Handling Best Practices

4. **Sample Controller to Test Exception Handling:** You can create a test controller to simulate an exception. Create **TestController.cs** in Controllers folder:

The screenshot shows the Visual Studio IDE interface. On the left, the Explorer pane displays the project structure under 'MYAPIAPP'. The 'Controllers' folder contains two files: 'ErrorController.cs' and 'TestController.cs', with 'TestController.cs' currently selected. The main code editor window shows the 'TestController.cs' file content:

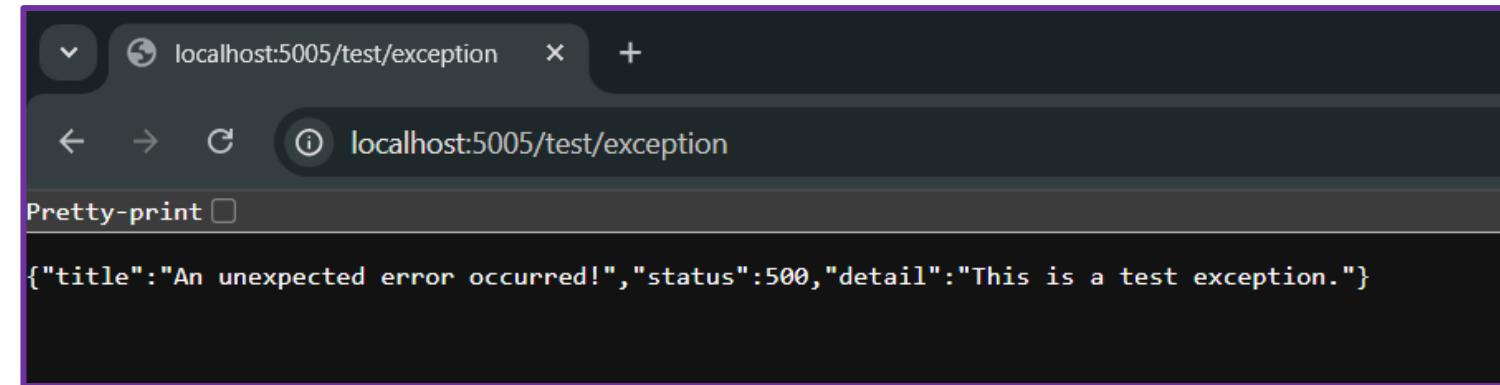
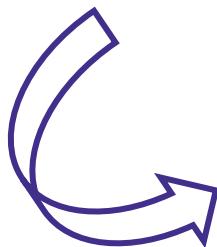
```
1  using Microsoft.AspNetCore.Mvc;
2
3  [ApiController]
4  [Route("[controller]")]
5  public class TestController : ControllerBase
6  {
7      [HttpGet("exception")]
8      public IActionResult ThrowException()
9      {
10         throw new InvalidOperationException("This is a test exception.");
11     }
12 }
```

# Exception Handling Best Practices

5. Run and Test: In the VSCode terminal (**dotnet run**):

✓ Open browser or Postman and visit: (<https://localhost:{port}/test/exception>)

```
C:\Users\Gurwinder.Parhar\MyApiApp>dotnet run
Using launch settings from C:\Users\Gurwinder.Parhar\MyApiApp\Properties\launchSettings.json...
Building...
[info]: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5005
[info]: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
[info]: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
[info]: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Users\Gurwinder.Parhar\MyApiApp
```



# Quiz



**Question 1: Which tool is commonly used to set breakpoints and step through code in ASP.NET Core debugging?**

- A) Visual Studio Debugger
- B) Azure Monitor
- C) NLog
- D) Serilog



# Quiz



**Question 2: What is the purpose of using the ILogger interface in ASP.NET Core?**

- A) To create user interfaces
- B) To capture structured runtime logs for monitoring and troubleshooting
- C) To manage database connections
- D) To write SQL queries



# Quiz



**Question 3: What is a best practice in exception handling to avoid silent failures?**

- A) Catch generic exceptions and ignore them
- B) Use empty catch blocks
- C) Always log exceptions and avoid swallowing them silently
- D) Never log exceptions



# Q&A Session

**Question 1:** How does remote debugging help in ASP.NET Core development?



**Question 2:** Why is structured logging important in ASP.NET Core applications?

# Q&A Session

**Question 3:** What are common causes of errors related to middleware and routing in ASP.NET Core?



**Question 4:** How can application logs be effectively managed for troubleshooting?

# Module 7: Testing ASP.NET Core Applications

- Introduction to Unit Testing
- Setting Up Test Projects
- Mocking Dependencies in Tests
- Best Practices for Integration Testing
- Mocking Dependencies and Running Web Server 'In Memory' for Tests



# Introduction to Unit Testing

- ✓ Unit testing is the process of verifying the smallest parts of an application, called units, to ensure they function as intended independently. It helps catch bugs early in the development cycle, reducing costly fixes later.
- ✓ By isolating components, unit testing improves code quality, maintainability, and facilitates safer refactoring. In ASP.NET Core applications, unit testing plays a critical role in validating business logic and ensuring application stability. The following are the key aspects and benefits of unit testing:

01

**Isolated Testing of Components**

02

**Automated and Repeatable**

03

**Use of Testing Frameworks**

04

**Test-Driven Development (TDD) Support**

05

**Improved Code Design**

06

**Detection of Regressions**

# Introduction to Unit Testing

## 1. Isolated Testing of Components:

- Unit tests focus on a single "unit" of code—usually a method or class—isolated from external dependencies. This isolation is often achieved using mocks or stubs to simulate interactions with databases, APIs, or other services, ensuring tests run quickly and reliably.

## 2. Automated and Repeatable:

- Unit tests are automated scripts that can be executed repeatedly throughout the development lifecycle. Automated testing supports continuous integration and continuous deployment by providing fast feedback on code changes.

## 3. Use of Testing Frameworks:

- ASP.NET Core developers typically use frameworks such as xUnit, NUnit, or MSTest to write and manage unit tests. These frameworks provide attributes, assertions, and test runners that streamline the testing process.

# Introduction to Unit Testing

## 4. Test-Driven Development (TDD) Support:

- Unit testing is integral to Test-Driven Development, where tests are written before the actual code. This approach encourages writing only the necessary code, resulting in cleaner, more reliable applications.

## 5. Improved Code Design:

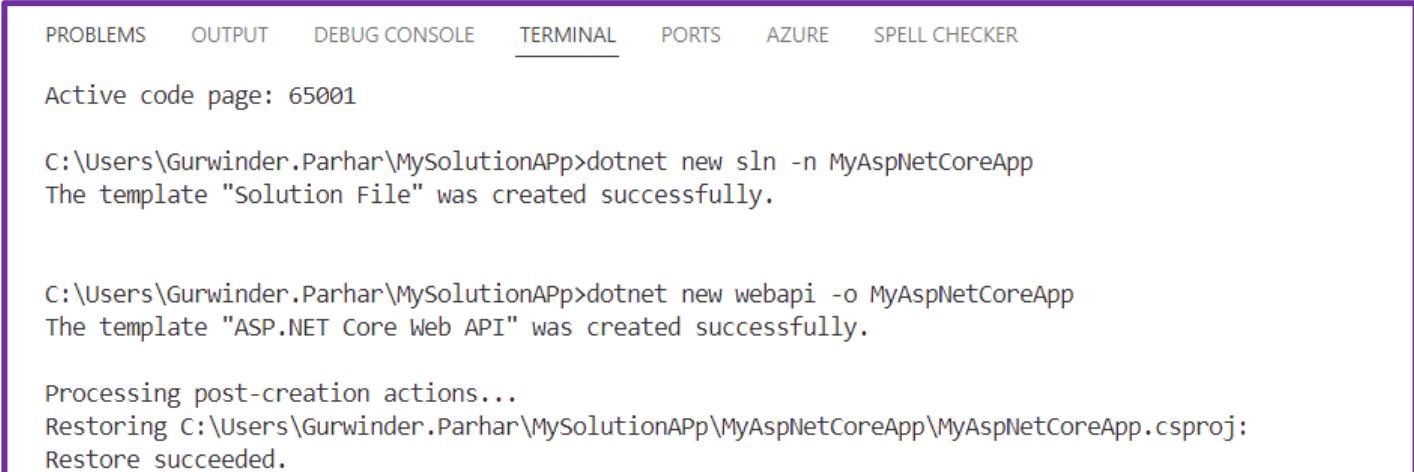
- Writing unit tests promotes better software design by encouraging modular, loosely coupled, and testable code components. This leads to easier refactoring and scalability over time.

## 6. Detection of Regressions:

- Unit tests act as a safety net that catches regressions—bugs introduced by new changes—by verifying that existing functionality remains intact after updates or enhancements.

# Setting Up Test Projects

- ✓ Setting up test projects involves creating a dedicated project within your ASP.NET Core solution specifically for writing and running automated tests.
- ✓ This separation ensures that your tests are organised, maintainable, and independent from your main application code. The following are the steps for setting up test projects:
  1. **Create a New Solution:** Run this command to create an empty solution file in your folder:
  2. **Create your main ASP.NET Core Web API or MVC project:** Choose either a Web API or MVC template. For example, to create a Web API project:



The screenshot shows the VS Code interface with the Terminal tab selected. The terminal window displays the following output:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS AZURE SPELL CHECKER

Active code page: 65001

C:\Users\Gurwinder.Parhar\MySolutionApp>dotnet new sln -n MyAspNetCoreApp
The template "Solution File" was created successfully.

C:\Users\Gurwinder.Parhar\MySolutionApp>dotnet new webapi -o MyAspNetCoreApp
The template "ASP.NET Core Web API" was created successfully.

Processing post-creation actions...
Restoring C:\Users\Gurwinder.Parhar\MySolutionApp\MyAspNetCoreApp\MyAspNetCoreApp.csproj:
Restore succeeded.
```

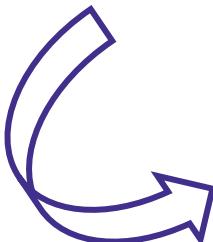
# Setting Up Test Projects

## 3. Add the Main Project to the Solution

## 4. Now Create the Test Project

```
C:\Users\Gurwinder.Parhar\MySolutionAPP>dotnet sln add MyAspNetCoreApp/MyAspNetCoreApp.csproj  
Project `MyAspNetCoreApp\MyAspNetCoreApp.csproj` added to the solution.
```

```
C:\Users\Gurwinder.Parhar\MySolutionAPP>
```



PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS    AZURE    SPELL CHECKER

```
C:\Users\Gurwinder.Parhar\MySolutionAPP>dotnet new xunit -o MyAspNetCoreApp.Tests  
The template "xUnit Test Project" was created successfully.
```

```
Processing post-creation actions...  
Restoring C:\Users\Gurwinder.Parhar\MySolutionAPP\MyAspNetCoreApp.Tests\MyAspNetCoreApp.Tests.csproj:  
Restore succeeded.
```

```
C:\Users\Gurwinder.Parhar\MySolutionAPP>
```

# Setting Up Test Projects

## 5. Add the Test Project to the Solution

## 6. Add a Project Reference to the Main Project from the Test Project

The screenshot shows a terminal window with the following interface elements:

- Top bar: PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL (underlined), PORTS, AZURE, SPELL CHECKER.
- Right side: JavaSE-21 LTS icon.

The terminal output is as follows:

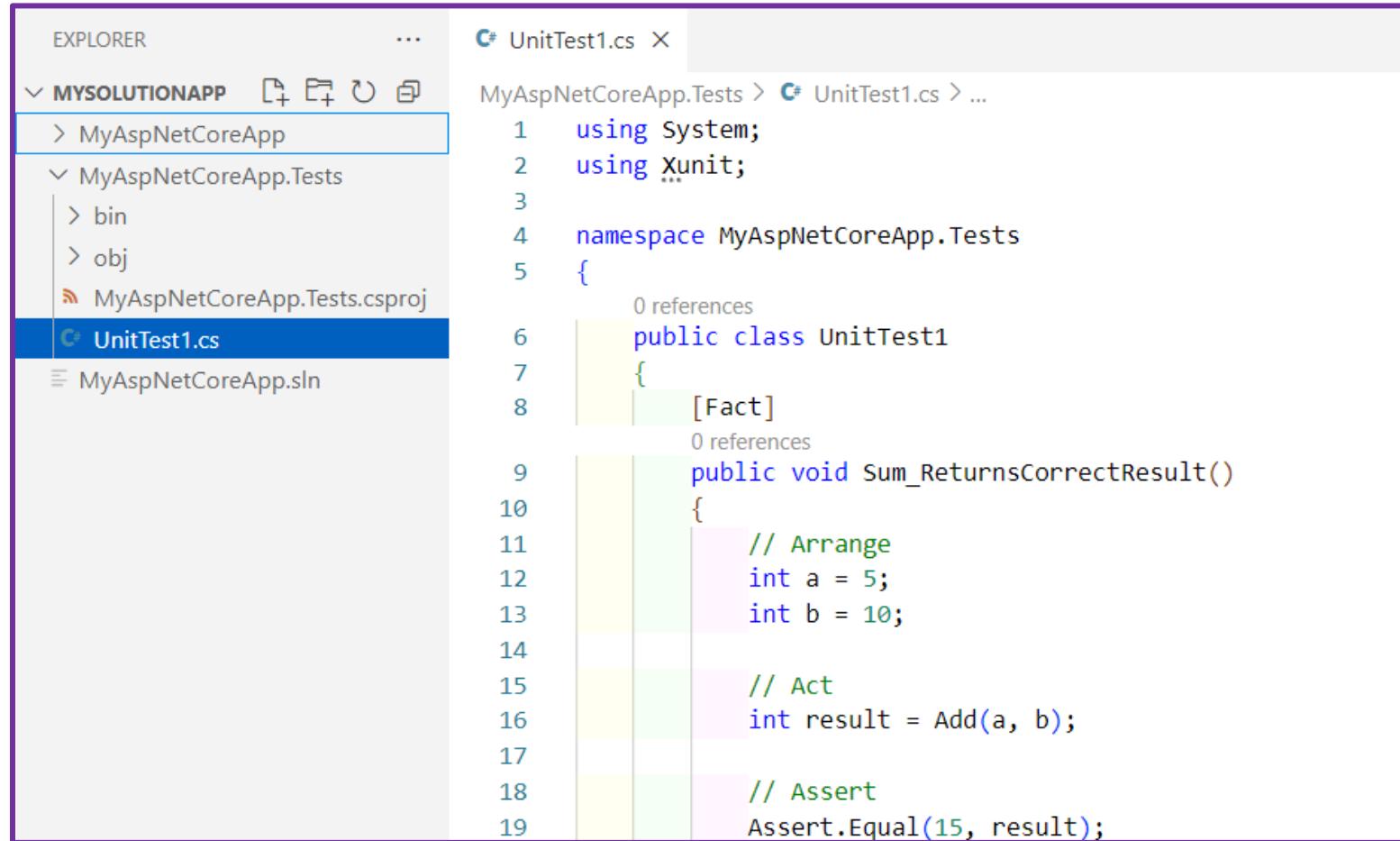
```
C:\Users\Gurwinder.Parhar\MySolutionAPP>dotnet sln add MyAspNetCoreApplication.Tests/MyAspNetCoreApplication.Tests.csproj
Project `MyAspNetCoreApplication.Tests\MyAspNetCoreApplication.Tests.csproj` added to the solution.

C:\Users\Gurwinder.Parhar\MySolutionAPP>dotnet add MyAspNetCoreApplication.Tests/MyAspNetCoreApplication.Tests.csproj reference MyAspNetCoreApplication/MyAspNetCoreApplication.csproj
Reference `..\MyAspNetCoreApplication\MyAspNetCoreApplication.csproj` added to the project.

C:\Users\Gurwinder.Parhar\MySolutionAPP>
```

# Setting Up Test Projects

## 7. Open and Start Writing Tests:



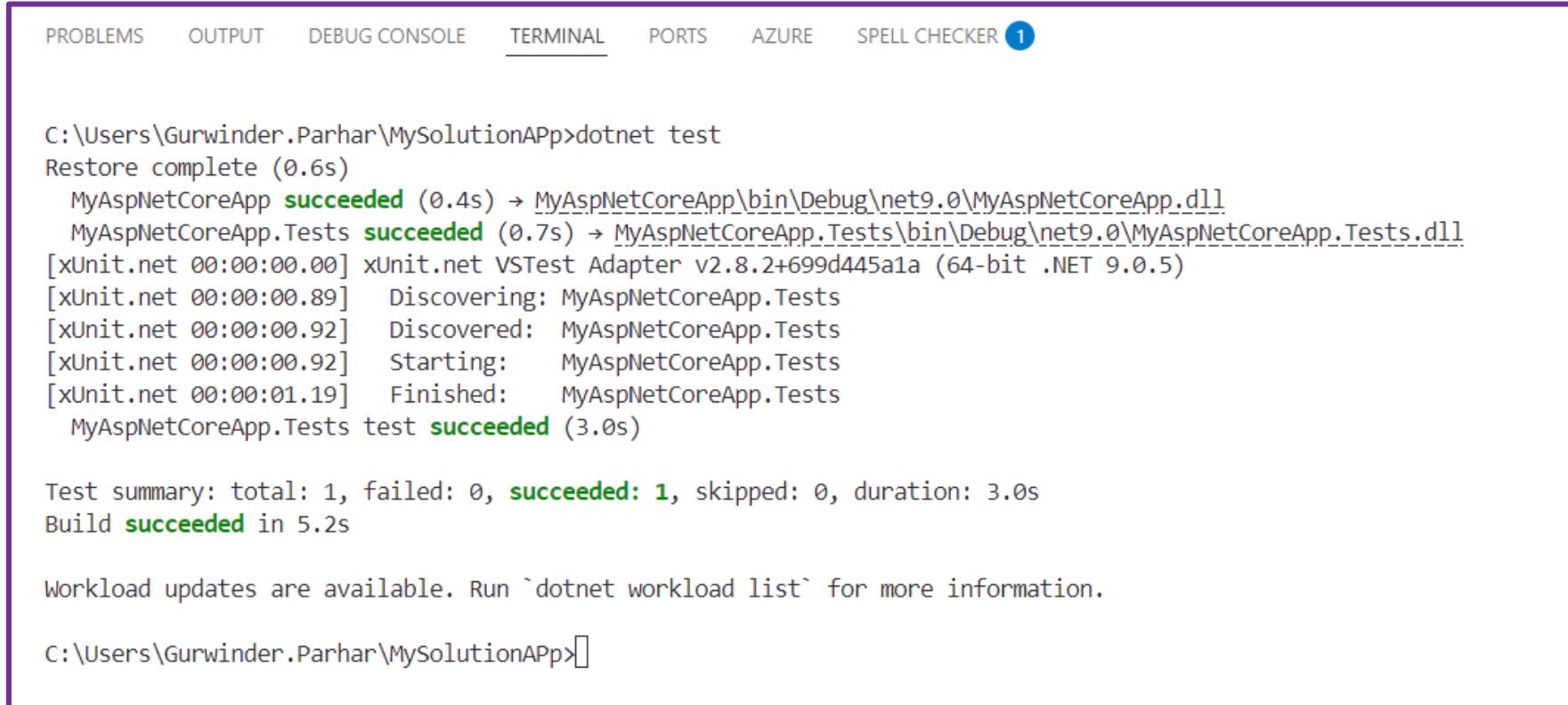
```
C# UnitTest1.cs

MyAspNetCoreApplication.Tests > C# UnitTest1.cs > ...
1  using System;
2  using Xunit;
3
4  namespace MyAspNetCoreApplication.Tests
5  {
6      0 references
7      public class UnitTest1
8      {
9          [Fact]
10         0 references
11         public void Sum_ReturnsCorrectResult()
12         {
13             // Arrange
14             int a = 5;
15             int b = 10;
16
17             // Act
18             int result = Add(a, b);
19
19             // Assert
20             Assert.Equal(15, result);
21         }
22     }
23 }
```

# Setting Up Test Projects

(Continued)

- ✓ **Run Tests Using the Command Line:** To execute tests from the terminal:



The screenshot shows a terminal window with the following output:

```
C:\Users\Gurwinder.Parhar\MySolutionAPP>dotnet test
Restore complete (0.6s)
  MyAspNetCoreApplication succeeded (0.4s) → MyAspNetCoreApplication\bin\Debug\net9.0\MyAspNetCoreApplication.dll
  MyAspNetCoreApplication.Tests succeeded (0.7s) → MyAspNetCoreApplication.Tests\bin\Debug\net9.0\MyAspNetCoreApplication.Tests.dll
[xUnit.net 00:00:00.00] xUnit.net VSTest Adapter v2.8.2+699d445a1a (64-bit .NET 9.0.5)
[xUnit.net 00:00:00.89] Discovering: MyAspNetCoreApplication.Tests
[xUnit.net 00:00:00.92] Discovered: MyAspNetCoreApplication.Tests
[xUnit.net 00:00:00.92] Starting: MyAspNetCoreApplication.Tests
[xUnit.net 00:00:01.19] Finished: MyAspNetCoreApplication.Tests
  MyAspNetCoreApplication.Tests test succeeded (3.0s)

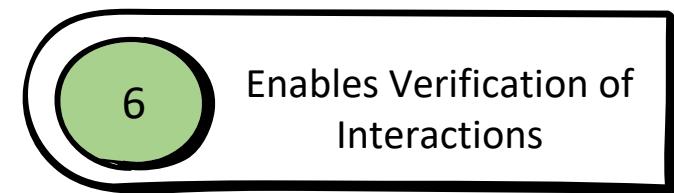
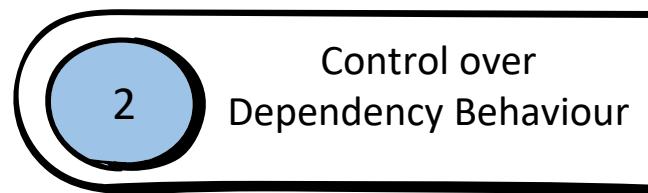
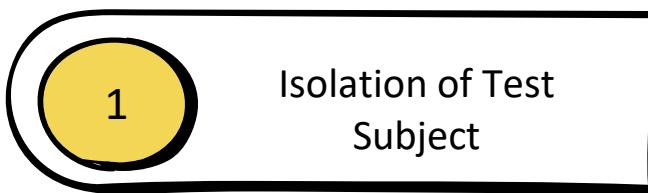
Test summary: total: 1, failed: 0, succeeded: 1, skipped: 0, duration: 3.0s
Build succeeded in 5.2s

Workload updates are available. Run `dotnet workload list` for more information.

C:\Users\Gurwinder.Parhar\MySolutionAPP>
```

# Mocking Dependencies in Tests

- ✓ Mocking dependencies is a crucial technique in unit testing ASP.NET Core applications. It allows developers to isolate the component under test by creating simulated versions of its external dependencies, such as services, databases, or APIs.
- ✓ This ensures tests focus solely on the logic of the component without interference from actual dependencies, leading to faster and more reliable tests. The following are the key aspects of mocking dependencies in tests:



# Mocking Dependencies in Tests

## 1. Isolation of Test Subject:

- Mocking helps isolate the unit being tested by replacing real dependencies with controlled mock objects, ensuring test results reflect only the unit's behaviour.

## 2. Control over Dependency Behaviour:

- Mocks allow precise control over how dependencies respond, enabling simulation of various scenarios including success, failure, or exceptions without relying on real implementations.

## 3. Improved Test Performance:

- By avoiding real dependencies like databases or external services, mocks reduce test execution time and avoid network or resource overhead.

# Mocking Dependencies in Tests

## 4. Facilitates Testing Edge Cases:

- Mocks make it easier to simulate rare or difficult-to-reproduce conditions, such as timeout errors or invalid data responses, enhancing test coverage.

## 5. Supports Dependency Injection:

- ASP.NET Core's built-in dependency injection framework simplifies injecting mocked services into components during testing, promoting clean, maintainable code.

## 6. Enables Verification of Interactions:

- Mocking frameworks provide features to verify that dependencies were called with expected parameters and correct frequency, improving test accuracy and behaviour validation.

# Best Practices for Integration Testing

- ✓ Integration testing ensures that different modules or services within an ASP.NET Core application work together correctly. It verifies the interactions between components, databases, and external APIs, helping to detect issues early in the development lifecycle.
- ✓ Effective integration tests improve application reliability and reduce costly bugs in production. The following are the best practices for integration testing in ASP.NET Core applications:

1

*Use Realistic Test Environments*

2

*Isolate Tests to Avoid Dependencies*

3

*Leverage In-Memory Databases for Speed*

4

*Automate Setup and Teardown Procedures*

5

*Focus on Critical Paths and User Scenarios*

6

*Incorporate Continuous Integration (CI) Tools*

# Best Practices for Integration Testing

## 1. Use Realistic Test Environments:

- Set up test environments that closely mirror production, including databases, external services, and configuration settings. This helps uncover environment-specific issues and ensures tests reflect real-world scenarios.

## 2. Isolate Tests to Avoid Dependencies:

- Design integration tests so that they do not depend on the execution order or share mutable state. Use techniques such as test data seeding and database transactions to maintain test isolation and reproducibility.

## 3. Leverage In-Memory Databases for Speed:

- Use in-memory databases like InMemoryProvider or SQLite in memory mode during tests to speed up execution while simulating real database behaviour. This improves test performance without compromising reliability.

# Best Practices for Integration Testing

## 4. Automate Setup and Teardown Procedures:

- Automate the initialisation and cleanup of resources such as database states and external services to ensure tests start with a consistent environment and prevent leftover data affecting subsequent tests.

## 5. Focus on Critical Paths and User Scenarios:

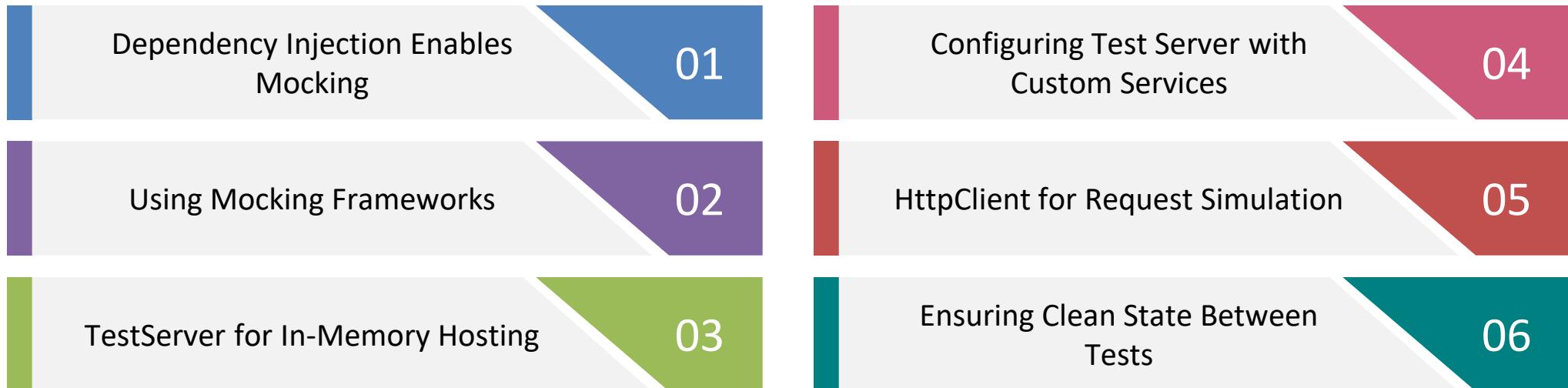
- Prioritise integration tests around essential workflows and critical business logic to maximise the value of testing efforts and ensure key application functions work end-to-end.

## 6. Incorporate Continuous Integration (CI) Tools:

- Integrate tests into CI pipelines with tools like GitHub Actions or Azure DevOps to run integration tests automatically on each code commit, providing fast feedback and maintaining code quality over time.

# Mocking Dependencies and Running Web Server 'In Memory' for Tests

- ✓ Isolating components and simulating external dependencies is crucial for reliable unit and integration tests. Mocking allows you to replace actual dependencies with controlled substitutes, while running the web server 'in memory' enables end-to-end testing without deploying to a real server.
- ✓ The following are the key concepts and practices for mocking dependencies and running an in-memory web server Core tests:



# Mocking Dependencies and Running Web Server 'In Memory' for Tests

## 1. Dependency Injection Enables Mocking:

- ASP.NET Core's built-in dependency injection allows you to substitute real services with mock implementations during testing, ensuring isolation and control over behaviour.

## 2. Using Mocking Frameworks:

- Libraries like Moq or NSubstitute help create mock objects that simulate the behaviour of real dependencies, allowing you to set expectations and verify interactions.

## 3. TestServer for In-Memory Hosting:

- The Microsoft.AspNetCore.TestHost package provides TestServer, which hosts your ASP.NET Core application entirely in memory, enabling fast and isolated integration tests.

# Mocking Dependencies and Running Web Server 'In Memory' for Tests

## 4. Configuring Test Server with Custom Services:

- You can configure TestServer to override service registrations, including replacing dependencies with mocks or stubs tailored to your test scenarios.

## 5. HttpClient for Request Simulation:

- Tests can interact with the in-memory server through an HttpClient instance, allowing you to send HTTP requests and validate responses as if communicating with a real server.

## 6. Ensuring Clean State Between Tests:

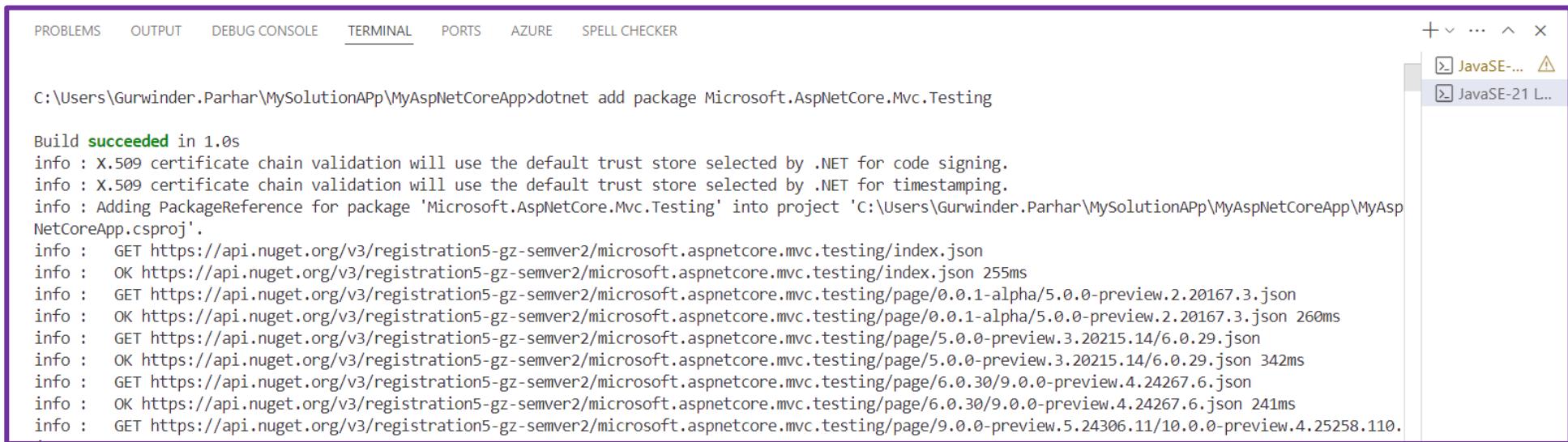
- It is important to reset or re-initialise mock states and test server instances between test runs to avoid cross-test interference and maintain test reliability.

# Mocking Dependencies and Running Web Server 'In Memory' for Tests

(Continued)

- ✓ The following are the steps to implement mocking dependencies and running web servers in memory for test:

## 1. Add Required NuGet Packages: Open terminal in VSCode and run:



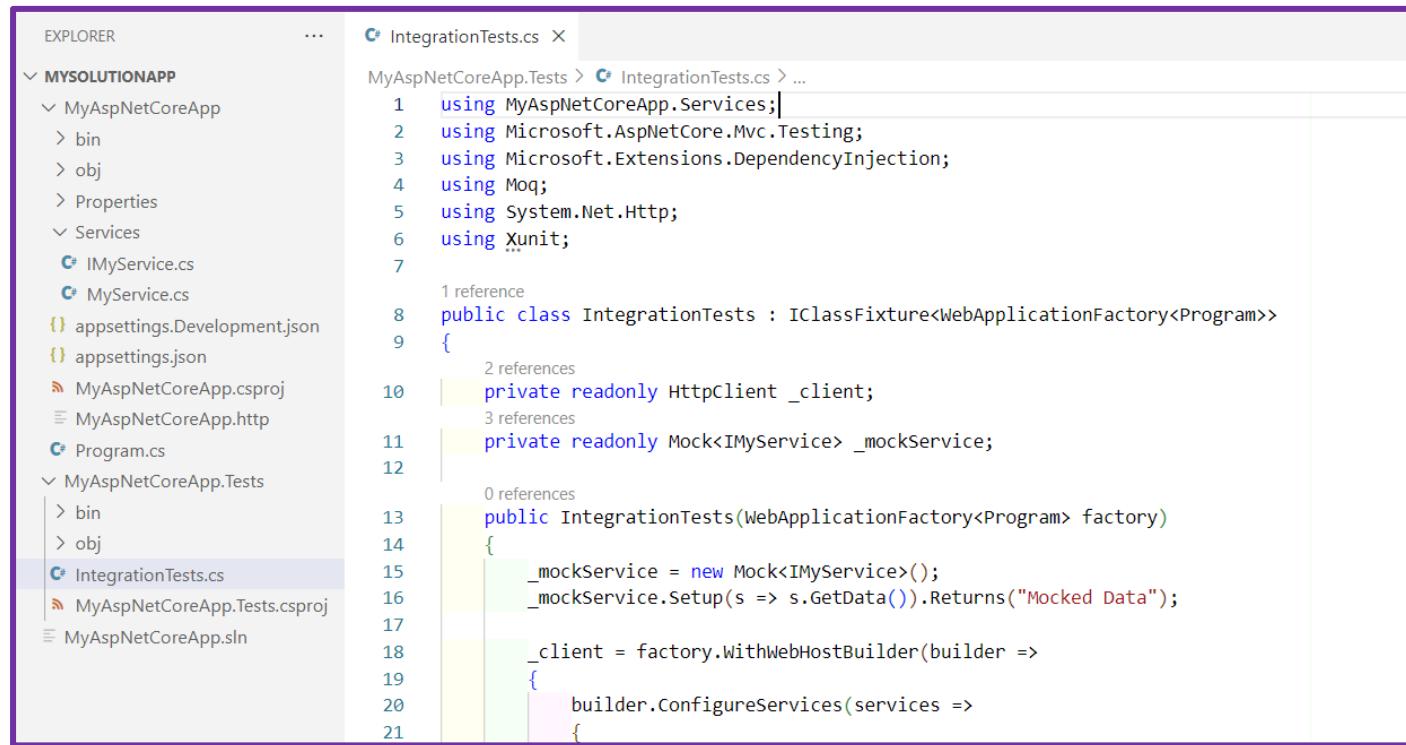
The screenshot shows the VSCode interface with the terminal tab selected. The terminal window displays the command `dotnet add package Microsoft.AspNetCore.Mvc.Testing` being run in the directory `C:\Users\Gurwinder.Parhar\MySolutionAPP\MyAspNetCoreApplication`. The output shows the build succeeded in 1.0s, and the process of downloading the package from NuGet.org.

```
C:\Users\Gurwinder.Parhar\MySolutionAPP\MyAspNetCoreApplication>dotnet add package Microsoft.AspNetCore.Mvc.Testing

Build succeeded in 1.0s
info : X.509 certificate chain validation will use the default trust store selected by .NET for code signing.
info : X.509 certificate chain validation will use the default trust store selected by .NET for timestamping.
info : Adding PackageReference for package 'Microsoft.AspNetCore.Mvc.Testing' into project 'C:\Users\Gurwinder.Parhar\MySolutionAPP\MyAspNetCoreApplication.csproj'.
info :   GET https://api.nuget.org/v3/registration5-gz-semver2/microsoft.aspnetcore.mvc.testing/index.json
info :     OK https://api.nuget.org/v3/registration5-gz-semver2/microsoft.aspnetcore.mvc.testing/index.json 255ms
info :   GET https://api.nuget.org/v3/registration5-gz-semver2/microsoft.aspnetcore.mvc.testing/page/0.0.1-alpha/5.0.0-preview.2.20167.3.json
info :     OK https://api.nuget.org/v3/registration5-gz-semver2/microsoft.aspnetcore.mvc.testing/page/0.0.1-alpha/5.0.0-preview.2.20167.3.json 260ms
info :   GET https://api.nuget.org/v3/registration5-gz-semver2/microsoft.aspnetcore.mvc.testing/page/5.0.0-preview.3.20215.14/6.0.29.json
info :     OK https://api.nuget.org/v3/registration5-gz-semver2/microsoft.aspnetcore.mvc.testing/page/5.0.0-preview.3.20215.14/6.0.29.json 342ms
info :   GET https://api.nuget.org/v3/registration5-gz-semver2/microsoft.aspnetcore.mvc.testing/page/6.0.30/9.0.0-preview.4.24267.6.json
info :     OK https://api.nuget.org/v3/registration5-gz-semver2/microsoft.aspnetcore.mvc.testing/page/6.0.30/9.0.0-preview.4.24267.6.json 241ms
info :   GET https://api.nuget.org/v3/registration5-gz-semver2/microsoft.aspnetcore.mvc.testing/page/9.0.0-preview.5.24306.11/10.0.0-preview.4.25258.110.
```

# Mocking Dependencies and Running Web Server 'In Memory' for Tests

2. Set Up TestServer with Mocked Dependencies: In a test class (e.g., `IntegrationTests.cs`), configure a `WebApplicationFactory` with overridden services:



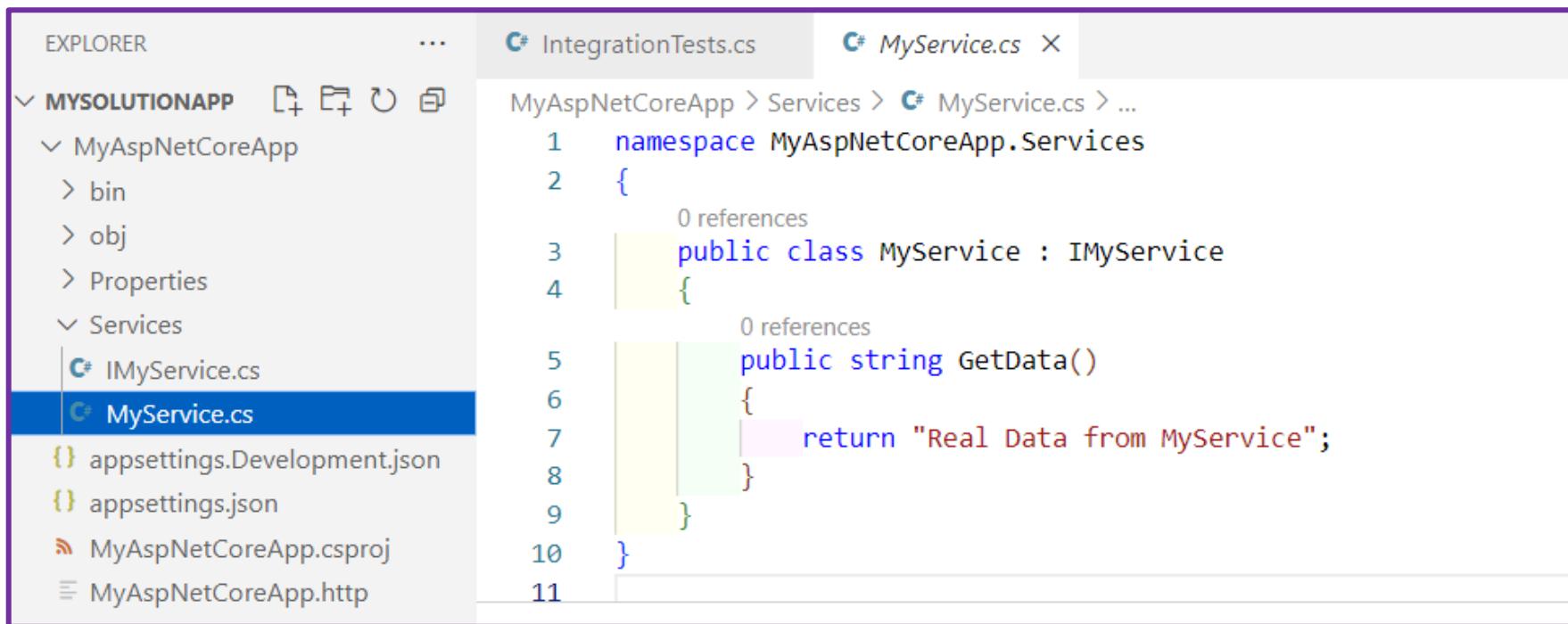
The screenshot shows the Visual Studio code editor with the `IntegrationTests.cs` file open. The code defines a test class `IntegrationTests` that inherits from `IClassFixture<WebApplicationFactory<Program>>`. It uses `HttpClient` and `Mock<IMyService>` to interact with the application's services. The `Program.cs` file is also visible in the solution explorer.

```
EXPLORER ... C# IntegrationTests.cs X
MYSOLUTIONAPP MyAspNetCoreApp
  > bin
  > obj
  > Properties
  Services
    IMyService.cs
    MyService.cs
    appsettings.Development.json
    appsettings.json
    MyAspNetCoreApp.csproj
    MyAspNetCoreApp.http
    Program.cs
MyAspNetCoreApp.Tests
  > bin
  > obj
  C# IntegrationTests.cs
  MyAspNetCoreApp.Tests.csproj
  MyAspNetCoreApp.sln
  1 using MyAspNetCoreApp.Services;
  2 using Microsoft.AspNetCore.Mvc.Testing;
  3 using Microsoft.Extensions.DependencyInjection;
  4 using Moq;
  5 using System.Net.Http;
  6 using Xunit;
  7
  8 public class IntegrationTests : IClassFixture<WebApplicationFactory<Program>>
  9 {
  10   private readonly HttpClient _client;
  11   private readonly Mock<IMyService> _mockService;
  12
  13   public IntegrationTests(WebApplicationFactory<Program> factory)
  14   {
  15     _mockService = new Mock<IMyService>();
  16     _mockService.Setup(s => s.GetData()).Returns("Mocked Data");
  17
  18     _client = factory.WithWebHostBuilder(builder =>
  19     {
  20       builder.ConfigureServices(services =>
  21       {
```

# Mocking Dependencies and Running Web Server 'In Memory' for Tests

(Continued)

- ✓ Set Up IMyService and MyService file under Services



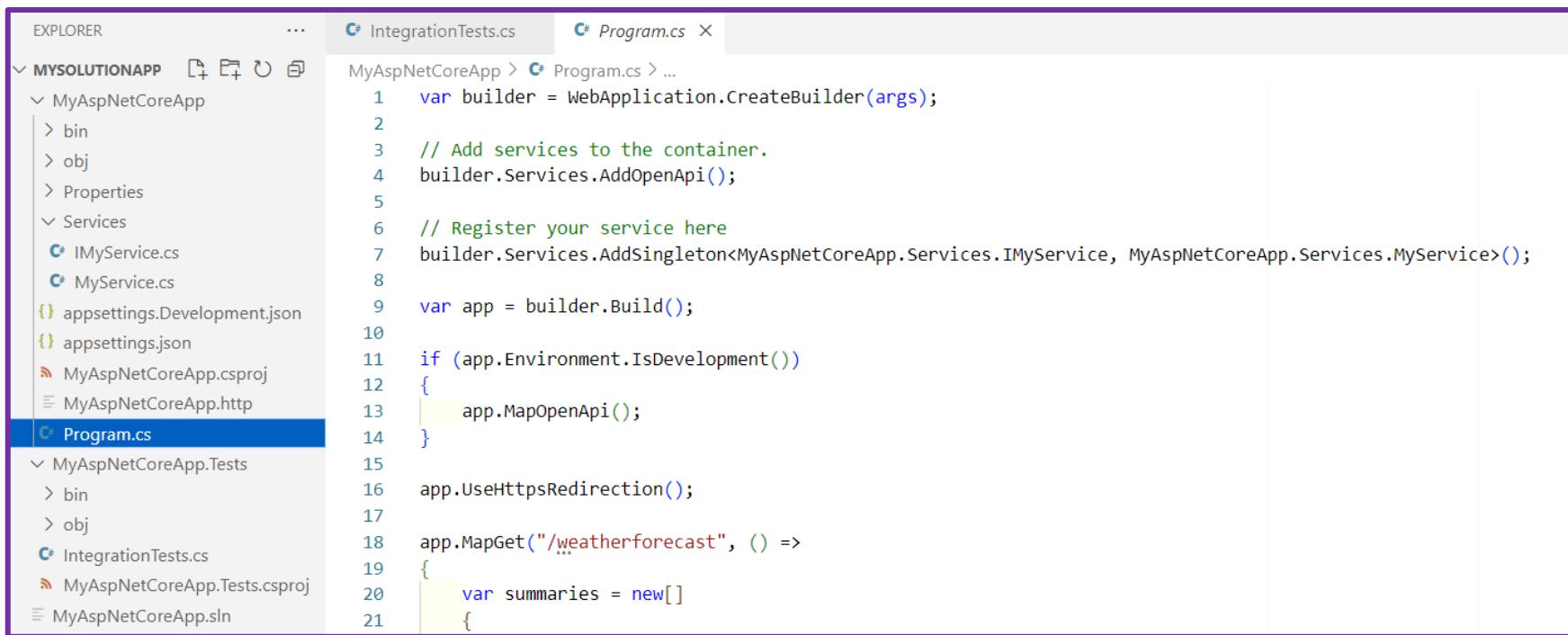
The screenshot shows the Visual Studio IDE interface. On the left, the Explorer pane displays the project structure under 'MYSOLUTIONAPP'. The 'Services' folder contains two files: 'IMyService.cs' and 'MyService.cs'. The 'MyService.cs' file is currently selected and shown in the main code editor window. The code defines a service class 'MyService' that implements the interface 'IMyService'. The implementation returns a hardcoded string.

```
namespace MyAspNetCoreApp.Services
{
    public class MyService : IMyService
    {
        public string GetData()
        {
            return "Real Data from MyService";
        }
    }
}
```

# Mocking Dependencies and Running Web Server 'In Memory' for Tests

(Continued)

- ✓ Update the **Program.cs** file

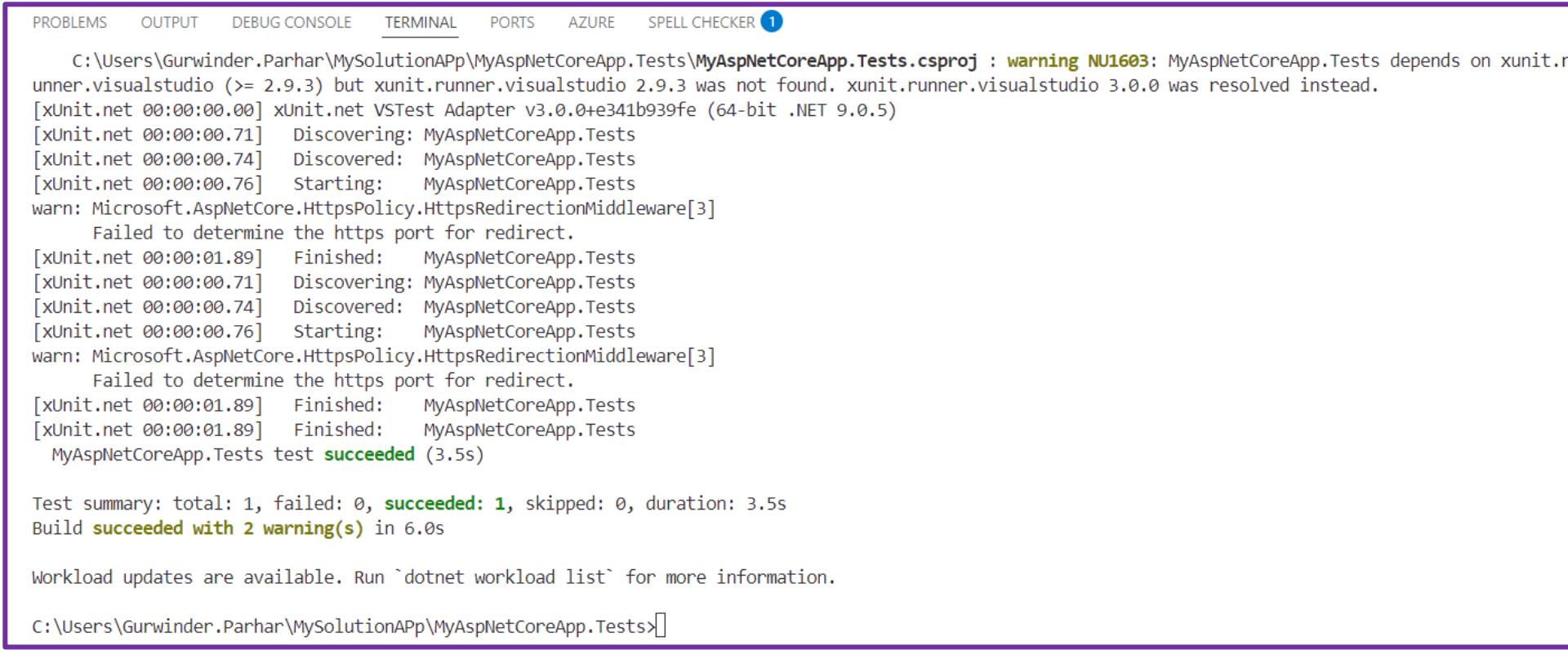


The screenshot shows the Visual Studio IDE interface. The left pane is the EXPLORER tool window, which displays the project structure of 'MYSOLUTIONAPP'. It includes the 'MyAspNetCoreApp' folder containing 'bin', 'obj', 'Properties', 'Services', 'IMyService.cs', 'MyService.cs', 'appsettings.Development.json', 'appsettings.json', 'MyAspNetCoreApp.csproj', 'MyAspNetCoreApp.http', and 'Program.cs'. Below it is the 'MyAspNetCoreApp.Tests' folder containing 'bin', 'obj', 'IntegrationTests.cs', 'MyAspNetCoreApp.Tests.csproj', and 'MyAspNetCoreApp.sln'. The 'Program.cs' file is selected in the EXPLORER window. The right pane is the code editor, showing the 'Program.cs' file content:

```
1 var builder = WebApplication.CreateBuilder(args);
2
3 // Add services to the container.
4 builder.Services.AddOpenApi();
5
6 // Register your service here
7 builder.Services.AddSingleton<MyAspNetCoreApp.Services.IMyService, MyAspNetCoreApp.Services.MyService>();
8
9 var app = builder.Build();
10
11 if (app.Environment.IsDevelopment())
12 {
13     app.MapOpenApi();
14 }
15
16 app.UseHttpsRedirection();
17
18 app.MapGet("/weatherforecast", () =>
19 {
20     var summaries = new[]
21     {
```

# Mocking Dependencies and Running Web Server 'In Memory' for Tests

### 3. Run Tests: Execute tests from VSCode terminal:



The screenshot shows the VSCode interface with the terminal tab selected. The terminal window displays the output of running unit tests for a .NET Core application. The output includes test discovery, execution, and summary information.

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS    AZURE    SPELL CHECKER 1

C:\Users\Gurwinder.Parhar\MySolutionAPp\MyAspNetCoreApplication.Tests\MyAspNetCoreApplication.Tests.csproj : warning NU1603: MyAspNetCoreApplication.Tests depends on xunit.runner.visualstudio (>= 2.9.3) but xunit.runner.visualstudio 2.9.3 was not found. xunit.runner.visualstudio 3.0.0 was resolved instead.
[xUnit.net 00:00:00.00] xUnit.net VSTest Adapter v3.0.0+e341b939fe (64-bit .NET 9.0.5)
[xUnit.net 00:00:00.71] Discovering: MyAspNetCoreApplication.Tests
[xUnit.net 00:00:00.74] Discovered: MyAspNetCoreApplication.Tests
[xUnit.net 00:00:00.76] Starting: MyAspNetCoreApplication.Tests
warn: Microsoft.AspNetCore.HttpsPolicy.HttpsRedirectionMiddleware[3]
      Failed to determine the https port for redirect.
[xUnit.net 00:00:01.89] Finished: MyAspNetCoreApplication.Tests
[xUnit.net 00:00:00.71] Discovering: MyAspNetCoreApplication.Tests
[xUnit.net 00:00:00.74] Discovered: MyAspNetCoreApplication.Tests
[xUnit.net 00:00:00.76] Starting: MyAspNetCoreApplication.Tests
warn: Microsoft.AspNetCore.HttpsPolicy.HttpsRedirectionMiddleware[3]
      Failed to determine the https port for redirect.
[xUnit.net 00:00:01.89] Finished: MyAspNetCoreApplication.Tests
[xUnit.net 00:00:01.89] Finished: MyAspNetCoreApplication.Tests
  MyAspNetCoreApplication.Tests test succeeded (3.5s)

Test summary: total: 1, failed: 0, succeeded: 1, skipped: 0, duration: 3.5s
Build succeeded with 2 warning(s) in 6.0s

Workload updates are available. Run `dotnet workload list` for more information.

C:\Users\Gurwinder.Parhar\MySolutionAPp\MyAspNetCoreApplication.Tests>
```

# Quiz



**Question 1: What is the primary focus of unit testing in ASP.NET Core applications?**

- A) Testing the entire application workflow
- B) Verifying the smallest parts of an application independently
- C) Testing user interface responsiveness
- D) Performing manual tests on APIs



# Quiz



**Question 2: Which frameworks are commonly used for writing and managing unit tests in ASP.NET Core?**

- A) React Router and Redux
- B) xUnit, NUnit, and MSTest
- C) Entity Framework and LINQ
- D) Serilog and NLog



# Quiz



## Question 3: What is mocking in the context of unit testing?

- A) Writing tests that only check UI components
- B) Testing application performance
- C) Ignoring dependencies during testing
- D) Creating simulated versions of external dependencies to isolate the unit under test



# Q&A Session

**Question 1:** Why is it important to isolate components during unit testing?



**Question 2:** What are some best practices for integration testing in ASP.NET Core?

# Q&A Session

**Question 3:** How does ASP.NET Core facilitate mocking and running tests ‘in memory’?



**Question 4:** How does setting up a separate test project improve the testing process?

# Module 8: Performance Optimisation in ASP.NET Core

- Techniques for Optimising Performance
- Caching Strategies
- Minimising Response Times for APIs
- Efficient Data Queries



# Techniques for Optimising Performance

- ✓ Performance optimisation in ASP.NET Core ensures fast, scalable, and efficient applications.
- ✓ Although the framework is built for speed, real-world projects require targeted techniques to handle growing complexity and user demand.
- ✓ Key strategies include response caching, async programming, middleware refinement, and database query tuning.
- ✓ Front-end performance is enhanced through asset minification and bundling. Continuous monitoring helps identify bottlenecks and maintain application stability.
- ✓ Optimisation is an ongoing effort that improves user experience and resource efficiency.



# Techniques for Optimising Performance

(Continued)

- The following are the effective techniques for optimising performance in ASP.NET Core applications:

1.

Response Caching

2.

Asynchronous  
Programming

3.

Response  
Compression

4.

Efficient Entity Framework  
Core Usage

5.

Middleware  
Optimisation

6.

Minification and  
Bundling of Assets

# Techniques for Optimising Performance

## 1. Response Caching:

- Caches server responses for repeated requests, reducing server processing and improving speed. Ideal for static content or rarely changing data.

## 2. Asynchronous Programming:

- Using `async` and `await` avoids blocking threads during I/O operations, allowing more requests to be processed simultaneously and improving scalability.

## 3. Response Compression:

- Reduces the size of HTTP responses using algorithms like Gzip or Brotli. This lowers bandwidth usage and speeds up client-side rendering.

# Techniques for Optimising Performance

## 4. Efficient Entity Framework Core Usage

- Applying AsNoTracking() for read-only queries, selecting only required fields, and reducing joins helps lower memory usage and database load.

## 5. Middleware Optimisation

- Trimming unused or unnecessary middleware in the request pipeline reduces overhead and improves request throughput.

## 6. Minification and Bundling of Assets

- Combining and compressing JavaScript and CSS files decreases the number and size of HTTP requests, enhancing front-end load times.

# Caching Strategies

- ✓ Caching is a critical performance enhancement technique in ASP.NET Core applications. It allows frequently accessed data or content to be stored temporarily, reducing the need for repeated data fetching or computation.
- ✓ This reduces server load and boosts response time, with ASP.NET Core offering caching strategies tailored to different application needs and scalability levels.
- ✓ By carefully choosing the right caching method, developers can ensure efficient resource usage while maintaining a responsive user experience.
- ✓ Caching can occur at various levels—application memory, response output, or distributed stores like Redis. Each approach offers trade-offs in complexity, performance, and persistence.
- ✓ Implementing caching effectively also involves handling cache expiration, invalidation, and security considerations. Overall, caching forms a core part of building fast and scalable ASP.NET Core applications.

# Caching Strategies

(Continued)

- ✓ *The following are the key caching strategies supported in ASP.NET Core:*

1.

In-Memory  
Caching

2.

Distributed  
Caching

3.

Response  
Caching

4.

Output  
Caching

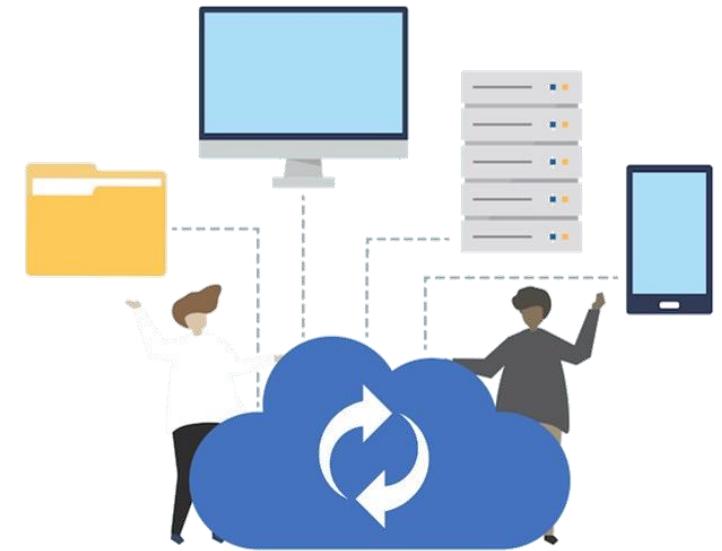
5.

Cache Tag  
Helpers

# Caching Strategies

## 1. In-Memory Caching:

- In-memory caching stores data directly within the application's memory space using the IMemoryCache interface.
- It is ideal for single-server applications where performance is critical, and the cached data does not need to persist across restarts.
- This strategy is simple to implement and provides very fast access, making it suitable for configuration values, static lookups, and reference data.
- It is not shared across servers, making it unsuitable for distributed or cloud environments.



# Caching Strategies

## 2. Distributed Caching:



- Distributed caching allows cache storage to be shared across multiple application instances or servers.
- ASP.NET Core supports providers like Redis and SQL Server for this purpose through the `IDistributedCache` interface.
- This strategy is essential for applications deployed in load-balanced or cloud environments, ensuring consistency and availability of cached data.
- It also provides durability across app restarts, but may introduce slight network latency compared to in-memory caching.

# Caching Strategies

## 3. Response Caching:

- Response caching stores the final HTTP response returned from a controller or Razor page, enabling faster responses for subsequent identical requests.
- It uses middleware and can be controlled via headers or attributes like [ResponseCache].
- This is particularly useful for caching publicly accessible, rarely-changing content such as landing pages, blog posts, or product listings.
- It's not suitable for dynamic or user-specific content unless properly configured.



# Caching Strategies

## 4. Output Caching:



- Output caching captures the rendered output of an action or Razor Page and stores it in memory or external stores.
- It offers more flexibility than response caching, including the ability to vary cache entries based on route parameters, headers, or tags.
- This approach reduces server processing time for frequently requested dynamic content while still allowing fine-grained control over cache duration and invalidation.
- It's ideal for complex UIs with partially dynamic content, improving performance and load handling.

# Caching Strategies

## 5. Cache Tag Helpers:

- Cache tag helpers in Razor views allow developers to cache specific HTML fragments or partial views.
- They are useful for sections of the page that do not change often, such as menus, headers, or sidebars.
- This form of fragment caching helps improve rendering performance by reducing the need to re-process Razor code for common UI elements.
- Tag helpers offer options for setting expiration policies directly in the markup, keeping the implementation clean and maintainable.



# Minimising Response Times for APIs

- ✓ Minimising response times for APIs is critical to delivering a responsive and scalable application experience.
- ✓ ASP.NET Core provides several built-in mechanisms to reduce latency, optimise resource use, and handle concurrent requests more efficiently.
- ✓ Response time is influenced by both application code and server-level factors, so optimisation should take a holistic approach.
- ✓ Caching, asynchronous processing, and efficient routing all contribute to faster API responses.
- ✓ Proper use of middleware and dependency injection can further streamline request handling and resource allocation.



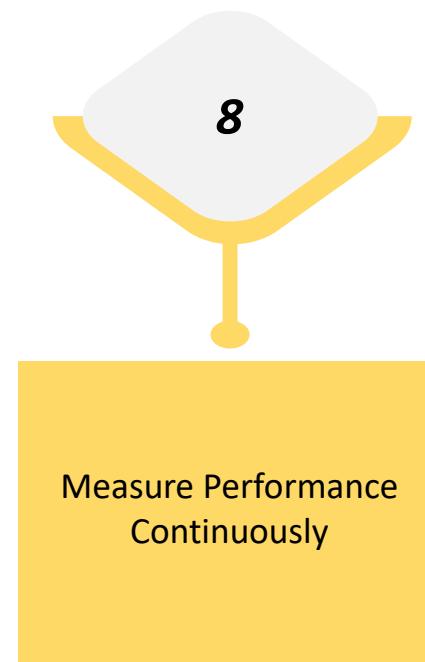
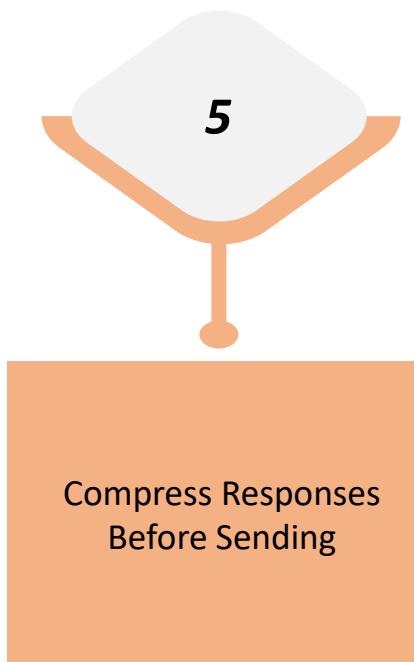
# Minimising Response Times for APIs

## *Key Techniques to Minimise API Response Times*

- 
- 1 Use Non-Blocking Operations
  - 2 Introduce Caching for Repeated Requests
  - 3 Optimise How Data Is Retrieved
  - 4 Streamline the Request Handling Steps

# Minimising Response Times for APIs

## *Key Techniques to Minimise API Response Times*



# Minimising Response Times for APIs

## 1. Use Non-Blocking Operations:

- Instead of making the system wait for each task to finish, non-blocking operations allow it to handle other tasks in the meantime, improving responsiveness when many users are active.

## 2. Introduce Caching for Repeated Requests:

- Caching allows frequently requested data to be stored temporarily. When the same information is requested again, it is delivered instantly without reprocessing.

## 3. Optimise How Data Is Retrieved:

- Reduce the amount of data pulled from databases by retrieving only what is necessary. Also, avoid including unnecessary background details during data access.

# Minimising Response Times for APIs

## 4. Streamline the Request Handling Steps:

- Remove unnecessary processing layers and steps. Keep only the essential actions required to respond to a request. This reduces the time taken to reach a response.

## 5. Compress Responses Before Sending:

- Compressing information before it is sent to users can drastically reduce the delivery time, especially when dealing with large sets of data.

## 6. Limit Unnecessary Data in Responses:

- Design responses to include only relevant and important information. This avoids delays caused by transferring large or overly detailed responses.

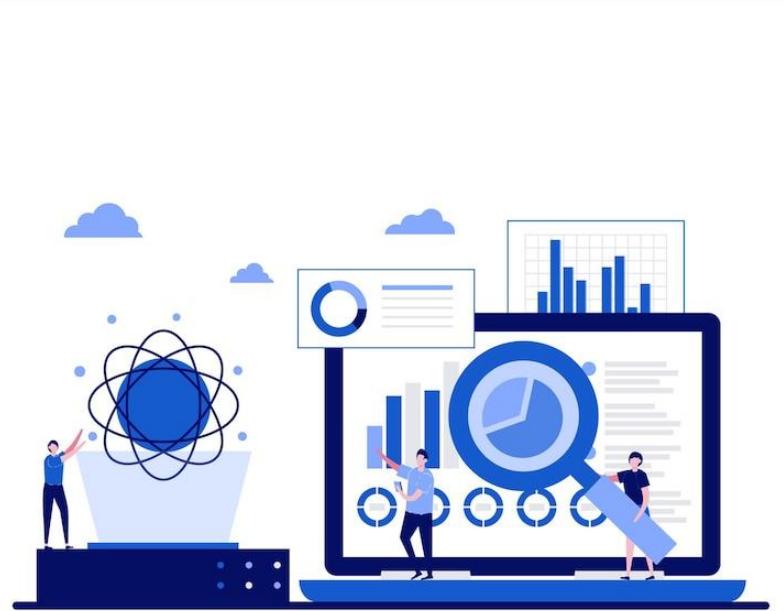
# Minimising Response Times for APIs

## 7. Use More Efficient Protocols and Settings:

- Use advanced communication settings that allow multiple requests to be handled on one connection and avoid repeated negotiation between the server and the client.

## 8. Measure Performance Continuously:

- Regularly monitor how long different actions take within the system.
- Use insights from this analysis to identify delays and improve those specific areas.



# Efficient Data Queries

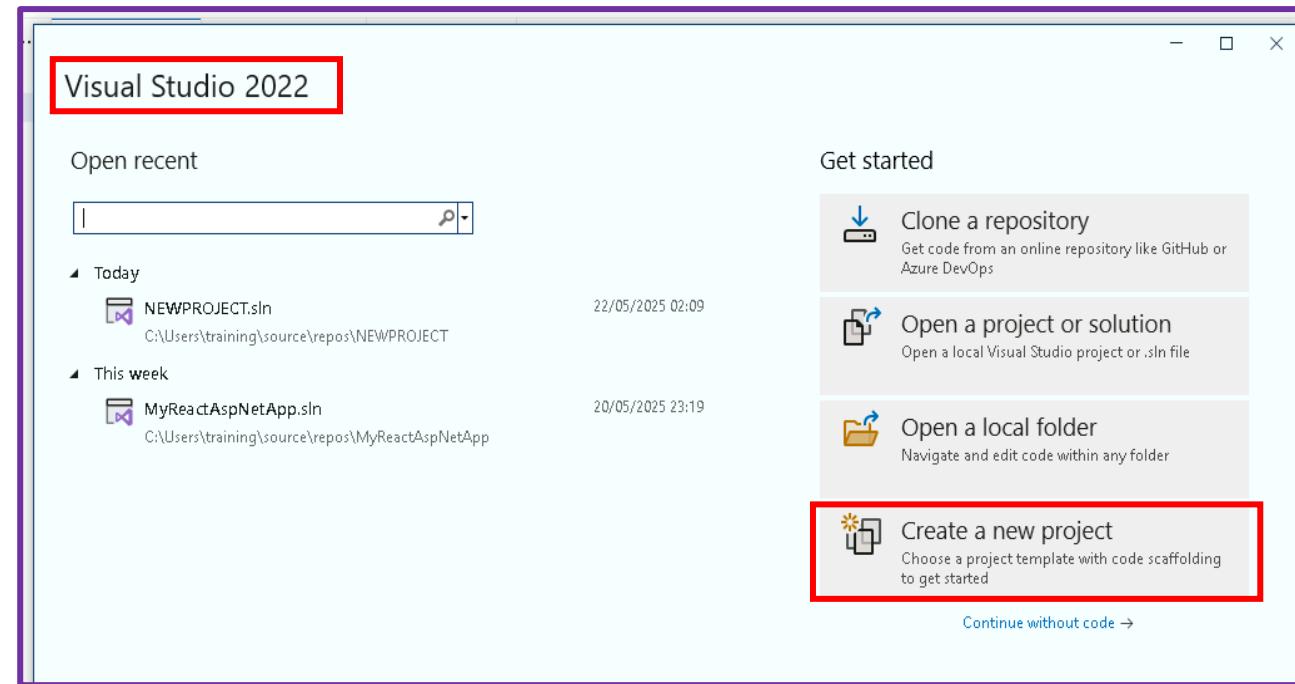
- ✓ Efficient data querying boosts ASP.NET Core performance by preventing slow responses, high memory use, and bottlenecks.
- ✓ By optimising how data is retrieved from the database, developers can reduce latency and improve scalability.
- ✓ Techniques such as using `AsNoTracking()` for read-only data, projecting only necessary columns with `Select()`, and implementing pagination can greatly reduce overhead.
- ✓ Leveraging `FirstOrDefault()` or `SingleOrDefault()` instead of retrieving large collections also enhances efficiency. In high-load scenarios, compiled queries can further reduce query execution time.
- ✓ Efficient data access isn't just about speed — it directly impacts user experience and resource utilisation.



# Efficient Data Queries

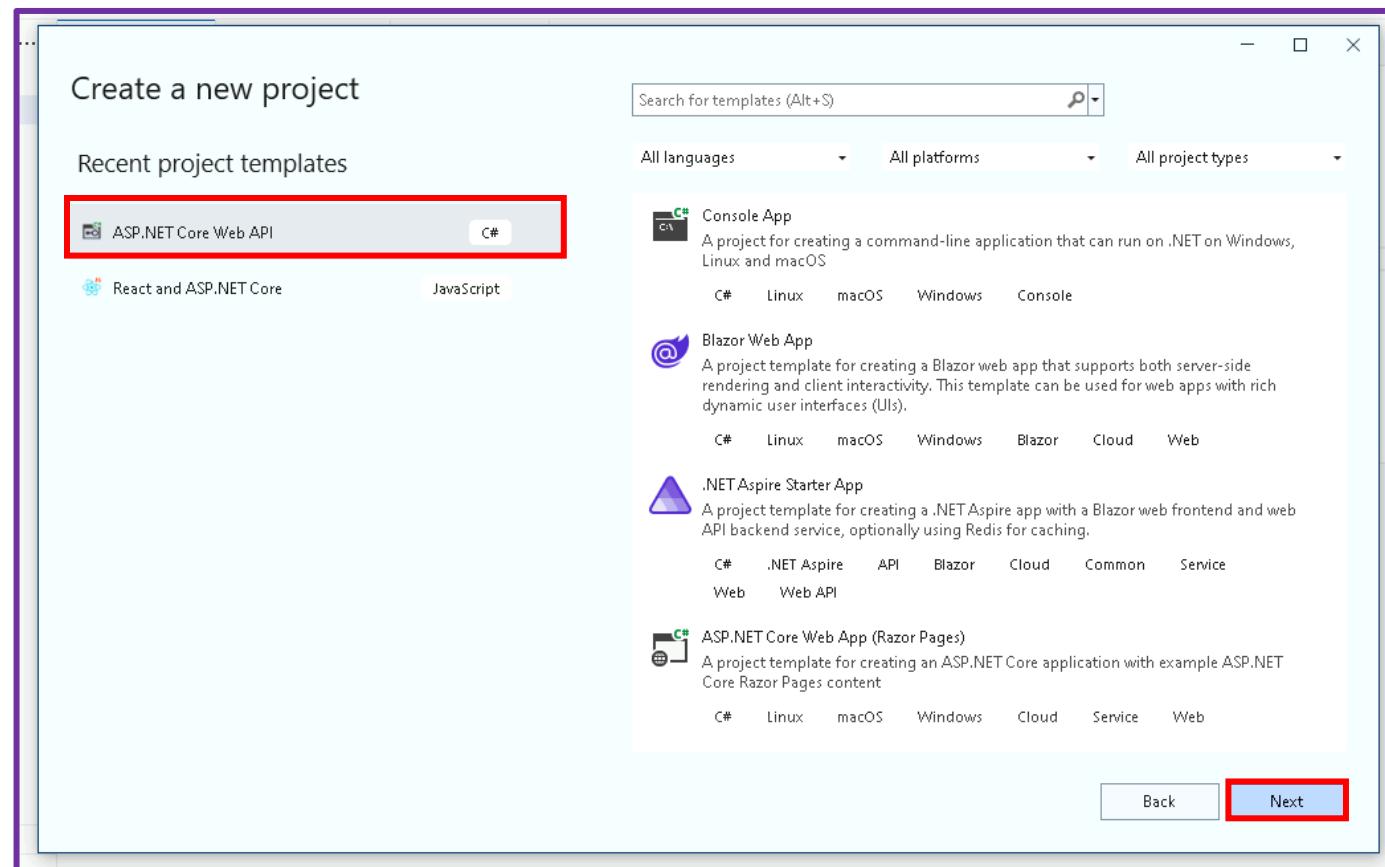
## *Steps for Implementing Efficient Data Queries*

**Step 1:** Open **Visual Studio 2022** and select **Create a new project**



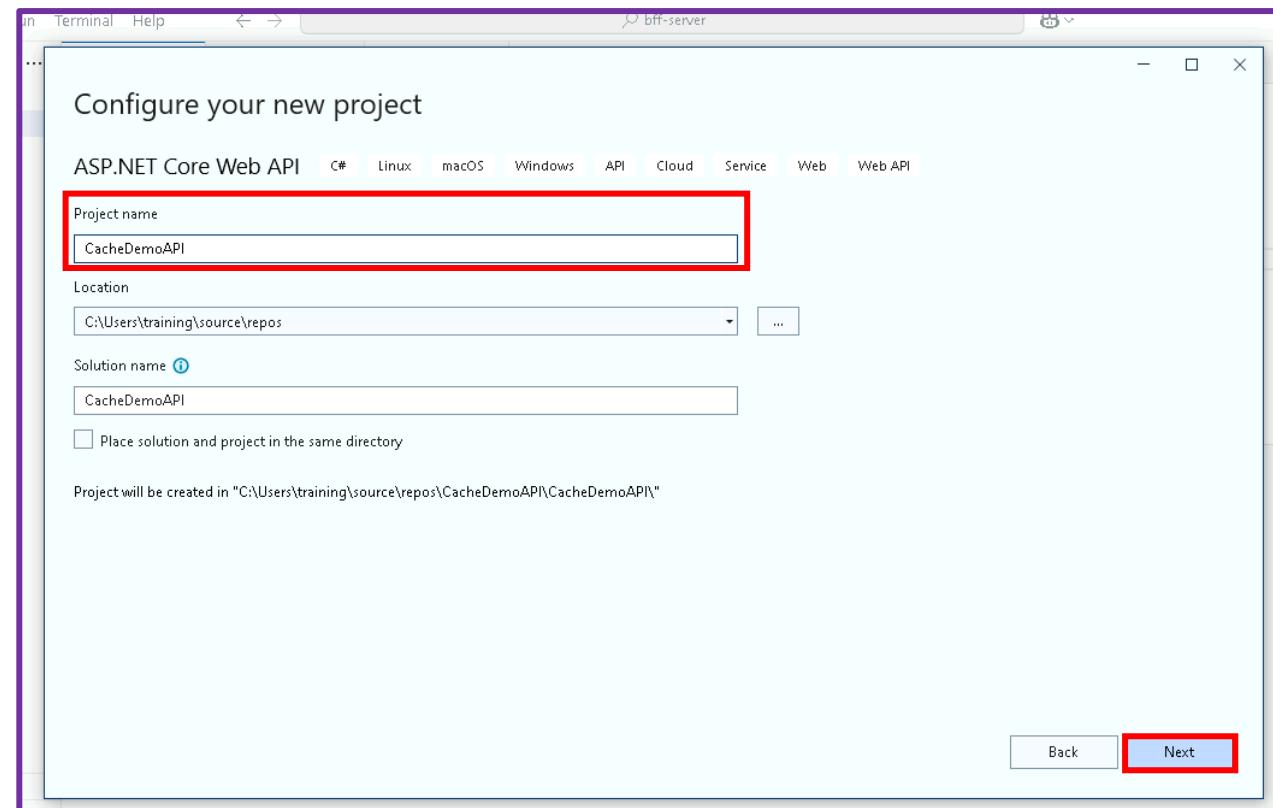
# Efficient Data Queries

Step 2: Choose **ASP.NET Core Web API** and click on **Next**



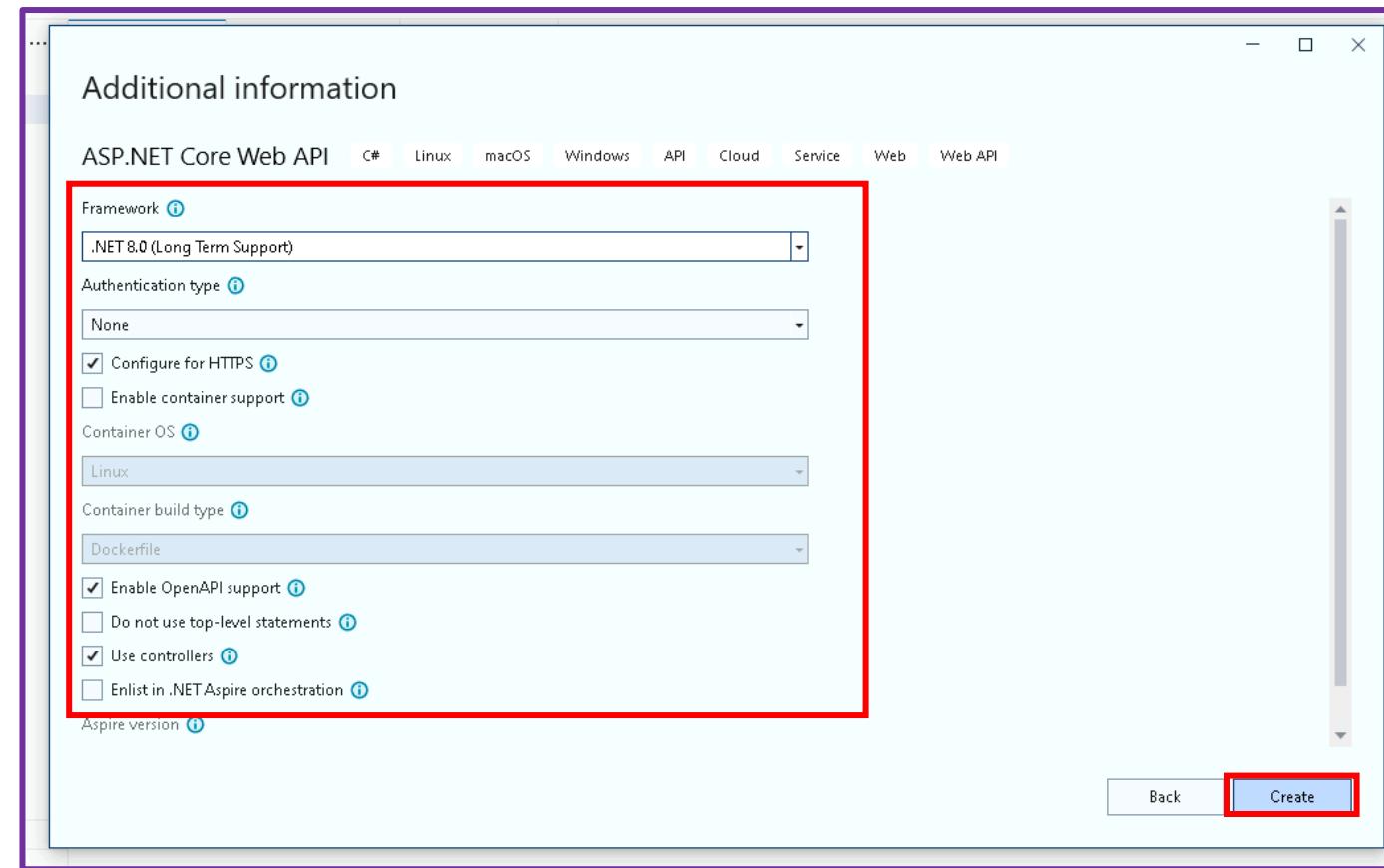
# Efficient Data Queries

**Step 3:** Enter the **Project name** as **CacheDemoAPI** and click **Next**



# Efficient Data Queries

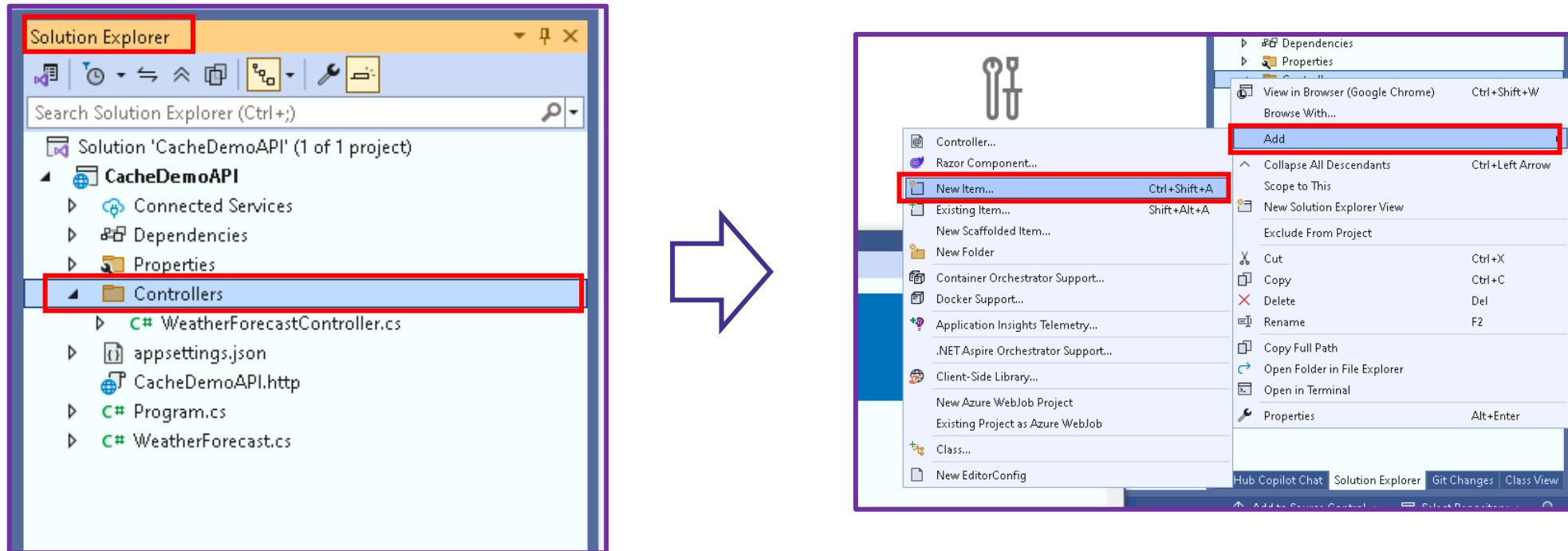
**Step 4:** Enter the required details and click on **Create**



# Efficient Data Queries

Step 5: In Solution Explorer, right-click the **Controllers** folder

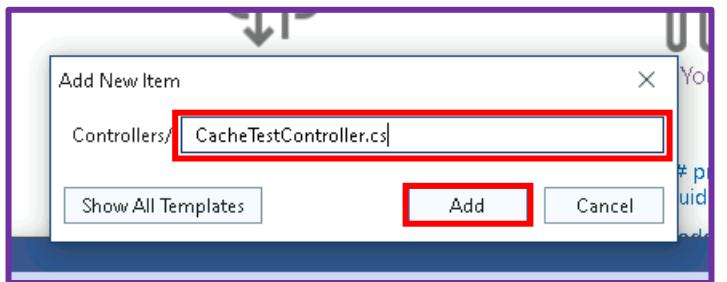
Step 6: Click on **Add > New Item...**



# Efficient Data Queries

**Step 7:** Enter the file name **CacheTestController.cs**, then click on **Add** to create the controller

- ✓ A new file has been successfully added to the Controllers folder.



The screenshot shows the Visual Studio IDE with the 'CacheTestController.cs' file open in the code editor. The code is as follows:

```
1  namespace CacheDemoAPI.Controllers
2  {
3      public class CacheTestController
4      {
5      }
6  }
```

Below the code editor is the Solution Explorer window, which lists the project structure. The 'Controllers' folder is expanded, and the 'CacheTestController.cs' file is visible and highlighted with a red box.

# Efficient Data Queries

## Step 8: Replace the Existing Code with the Updated Version

The screenshot shows the Visual Studio IDE interface. The main window displays the `CacheTestController.cs` file under the `CacheDemoAPI` project. A red box highlights the code block from line 18 to line 30, which contains the logic for retrieving and setting data in the cache. The Solution Explorer on the right shows the project structure with files like `CacheTestController.cs`, `WeatherForecastController.cs`, and `appsettings.json`. The Developer PowerShell window at the bottom shows the command prompt environment.

```
1  using Microsoft.AspNetCore.Mvc;
2  using Microsoft.Extensions.Caching.Memory;
3
4  namespace CacheDemoAPI.Controllers
5  {
6      [Route("api/[controller]")]
7      [ApiController]
8      public class CacheTestController : ControllerBase
9      {
10          private readonly IMemoryCache _cache;
11
12          public CacheTestController(IMemoryCache memoryCache)
13          {
14              _cache = memoryCache;
15          }
16
17          [HttpGet]
18          public IActionResult Get()
19          {
20              if (!_cache.TryGetValue("Time", out string cachedResult))
21              {
22                  cachedResult = $"Generated at: {DateTime.Now}";
23
24                  var options = new MemoryCacheEntryOptions()
25                      .SetAbsoluteExpiration(TimeSpan.FromSeconds(60));
26
27                  _cache.Set("Time", cachedResult, options);
28              }
29
30              return Ok(cachedResult);
31          }
32      }
33  }
```

# Efficient Data Queries

Step 9: Open the **Program.cs** file and add the following line: **builder.Services.AddMemoryCache();**

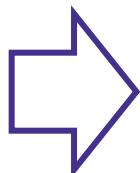
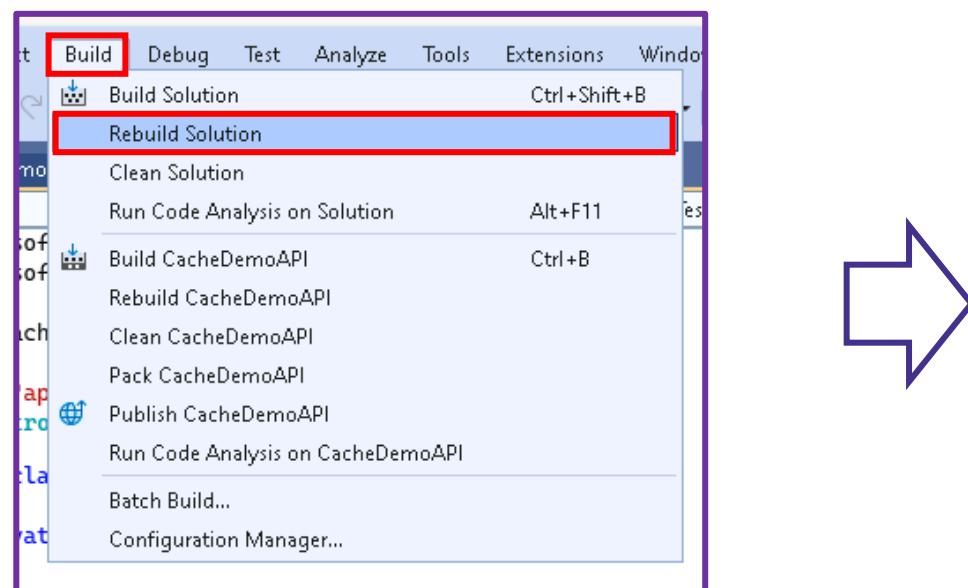
The screenshot shows the Visual Studio IDE interface. The left pane displays the code editor with the **Program.cs** file open. A red box highlights the line **builder.Services.AddMemoryCache();**. The right pane shows the **Solution Explorer** with the project **CacheDemoAPI** selected. A red box highlights the **Program.cs** file in the list. The bottom right pane shows the **Output** window with the following text:

```
Show output from: Build  
Build started at 03:04...  
===== Build: 0 succeeded, 0 failed, 1 up-to-date, 0 skipped ======
```

# Efficient Data Queries

**Step 10:** Save the file, then go to the **Build** tab and select **Rebuild Solution**

- ✓ Rebuilding has been completed successfully.

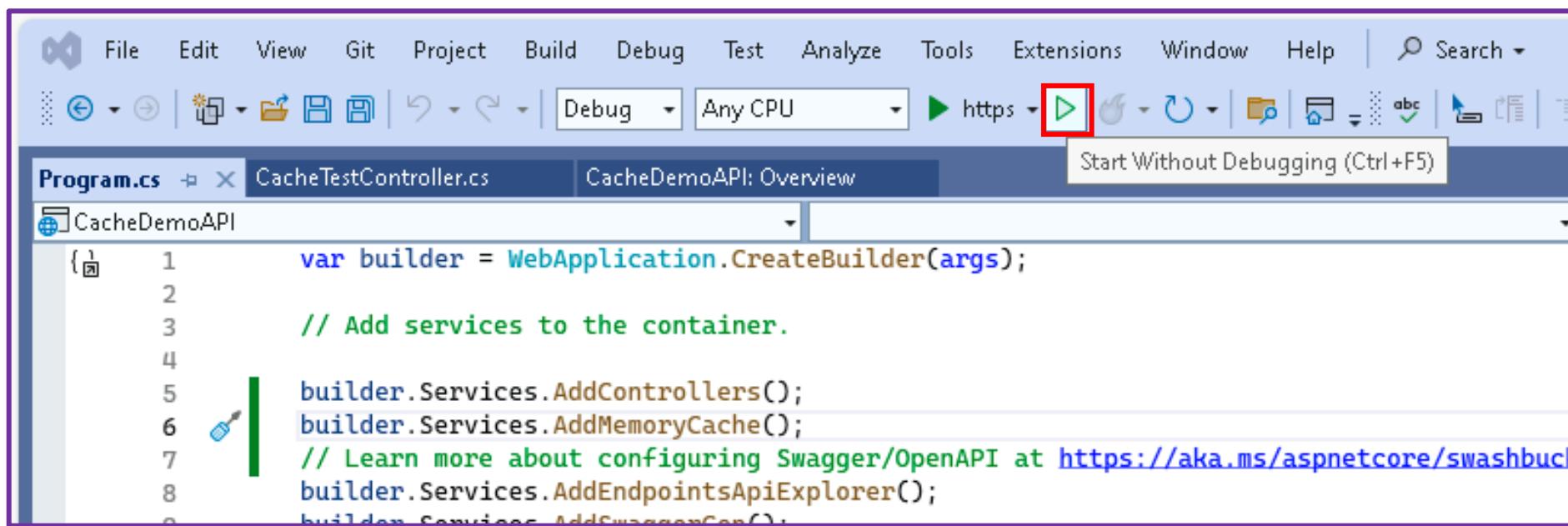


A screenshot of the Visual Studio Output window. The window title is 'Output'. It shows the command 'Rebuild started at 03:14...'. The log details the rebuild process: 'Rebuild All started: Project: CacheDemoAPI, Configuration: Debug Any', followed by a list of files being built, and concludes with 'Done building project "CacheDemoAPI.csproj". ===== Rebuild All: 1 succeeded, 0 failed, 0 skipped ===== Rebuild completed at 03:15 and took 08.123 seconds ====='. The bottom of the window shows tabs for 'Developer...', 'Developer...', 'Developer...', 'Developer...', 'Developer...', 'Developer...', 'Developer...', and 'Output'.

```
Rebuild started at 03:14...
Restored C:\Users\training\source\repos\CacheDemoAPI\CacheDemoAPI\CacheDemoAPI.csproj
1>----- Rebuild All started: Project: CacheDemoAPI, Configuration: Debug Any
1>C:\Users\training\source\repos\CacheDemoAPI\CacheDemoAPI\Controllers\CacheController.cs
1>CacheDemoAPI -> C:\Users\training\source\repos\CacheDemoAPI\CacheDemoAPI\bin\Debug\cacheDemoAPI.dll
1>Done building project "CacheDemoAPI.csproj".
=====
===== Rebuild All: 1 succeeded, 0 failed, 0 skipped =====
=====
===== Rebuild completed at 03:15 and took 08.123 seconds =====
```

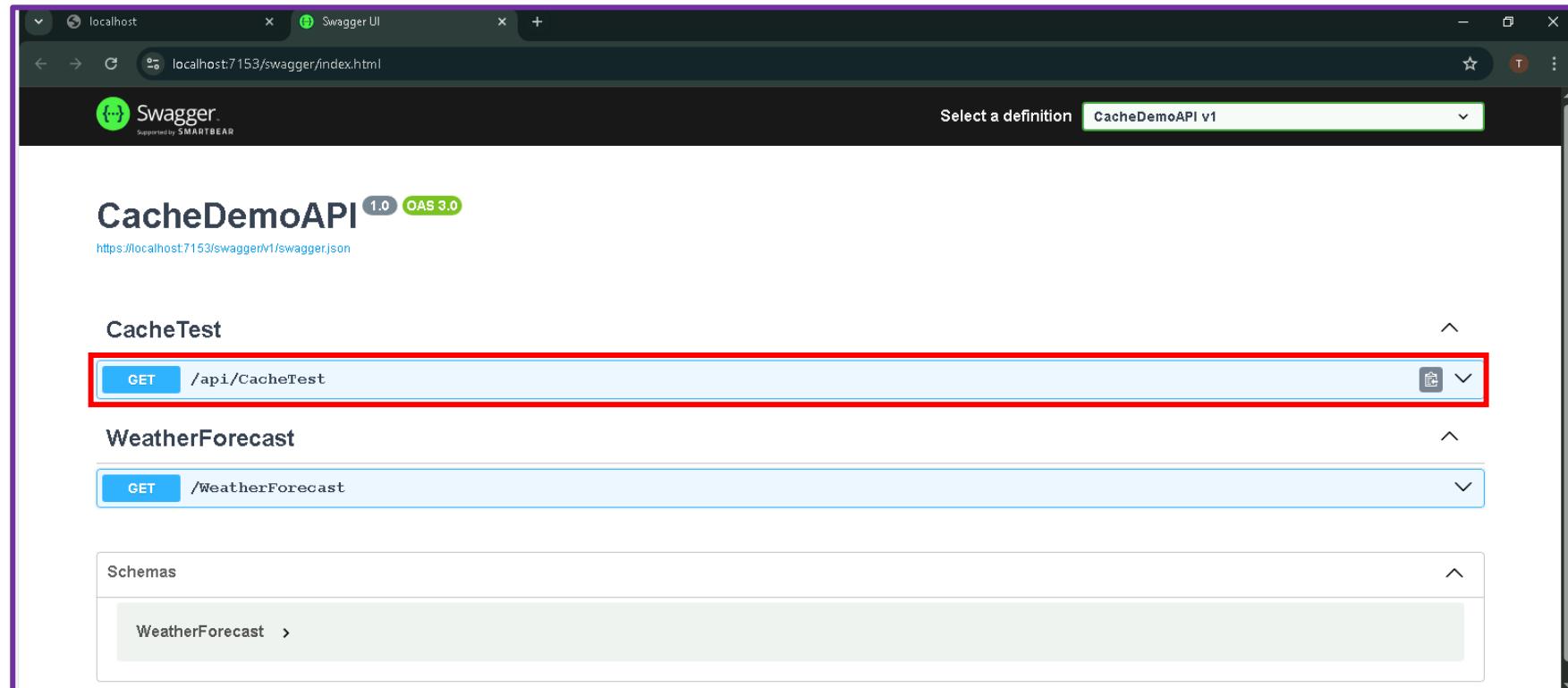
# Efficient Data Queries

**Step 11:** Click on the **Start Without Debugging** button to run the application



# Efficient Data Queries

Step 12: Click on **GET /api/CacheTest**



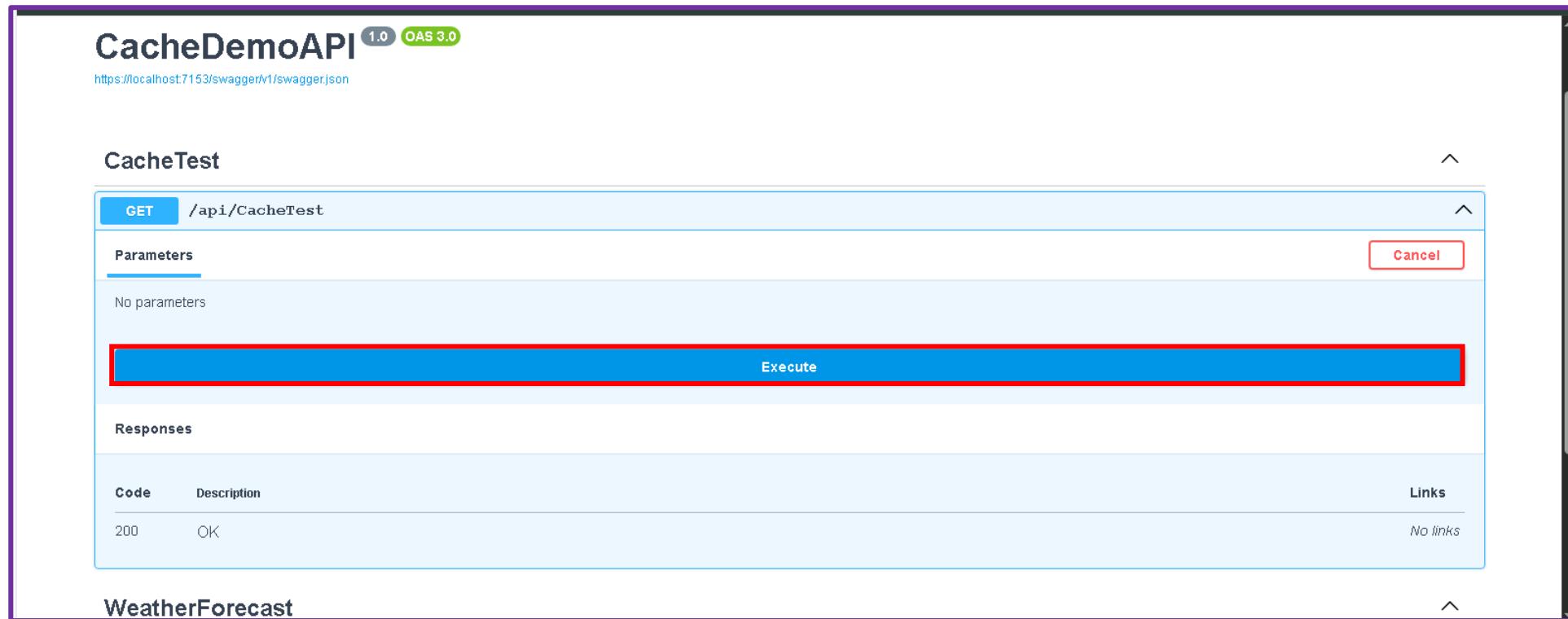
# Efficient Data Queries

**Step 13:** Click on Try it out

The screenshot shows a Swagger UI interface with a purple border. At the top left, it says "CacheTest". Below that is a "GET /api/CacheTest" button. To its right is a "Try it out" button, which is highlighted with a red box. Underneath the button are sections for "Parameters" (which says "No parameters") and "Responses". In the "Responses" section, there's a table with columns "Code" and "Description". It has one row with "200" and "OK". To the right of the table is a "Links" section that says "No links". At the bottom left, it says "WeatherForecast". Below that is a "GET /WeatherForecast" button. To its right is a "▼" button.

# Efficient Data Queries

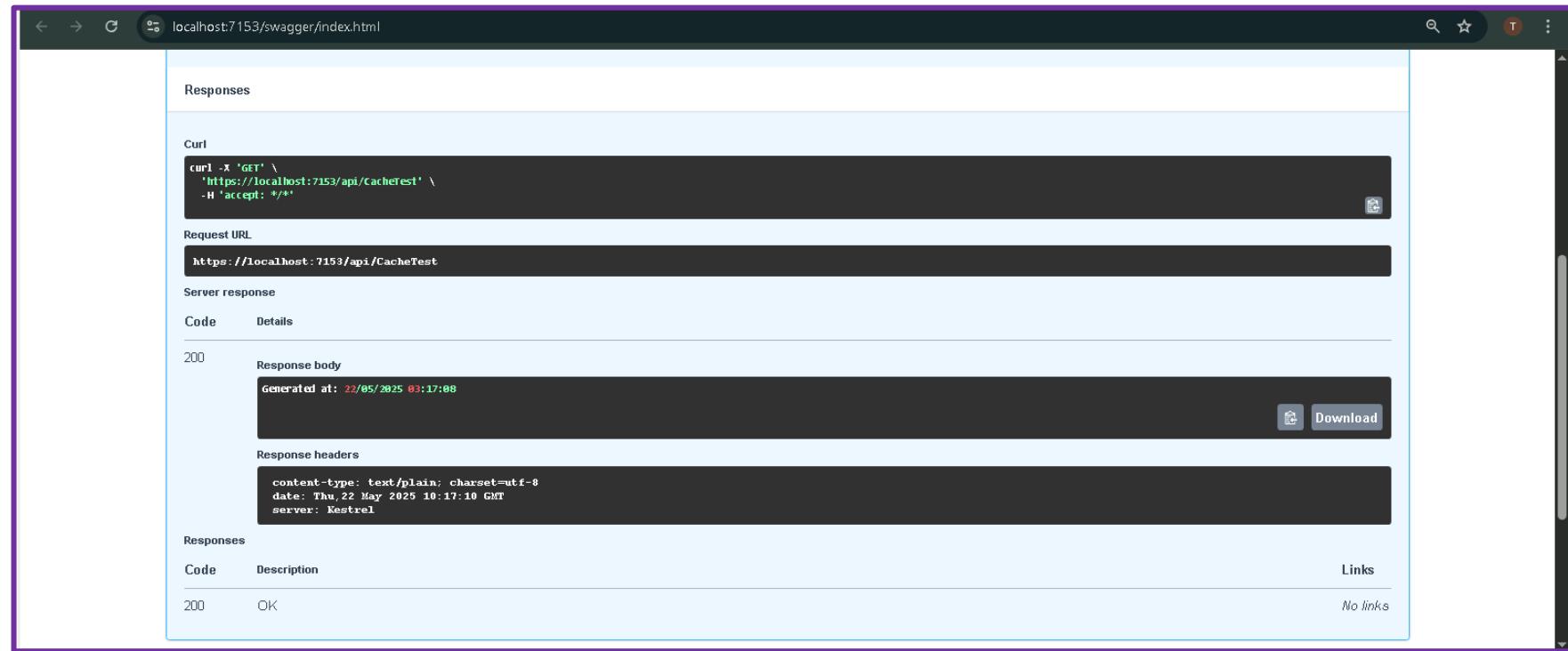
Step 14: Click on Execute



# Efficient Data Queries

(Continued)

- ✓ The efficient data queries and in-memory caching setup in ASP.NET core has been successfully implemented and tested using swagger UI.



# Quiz



**Question 1:** What is the purpose of response caching in ASP.NET Core?

- A) To store user credentials securely
- B) To cache server responses for repeated requests and improve speed
- C) To compress server responses
- D) To optimize database queries



# Quiz



**Question 2:** Which caching strategy is suitable for applications running on multiple servers or cloud environments?

- A) In-Memory Caching
- B) Response Caching
- C) Distributed Caching
- D) Output Caching



# Quiz



**Question 3:** What technique helps reduce bandwidth usage by decreasing the size of HTTP responses?

- A) Middleware Optimization
- B) Response Compression
- C) Entity Framework Optimization
- D) Asynchronous Programming



# Q&A Session

**Question 1:** What are some effective techniques for optimising performance in ASP.NET Core applications?



**Question 2:** How does distributed caching differ from in-memory caching?

# Q&A Session

**Question 3:** Why is minimizing API response times important, and what are some ways to achieve it?



**Question 4:** What role does efficient data querying play in performance optimisation?

# Module 9: Containerised Deployment with Azure Container Apps

- Overview of Azure Container Apps and Their Benefits
- Best Practices for Containerised Deployment

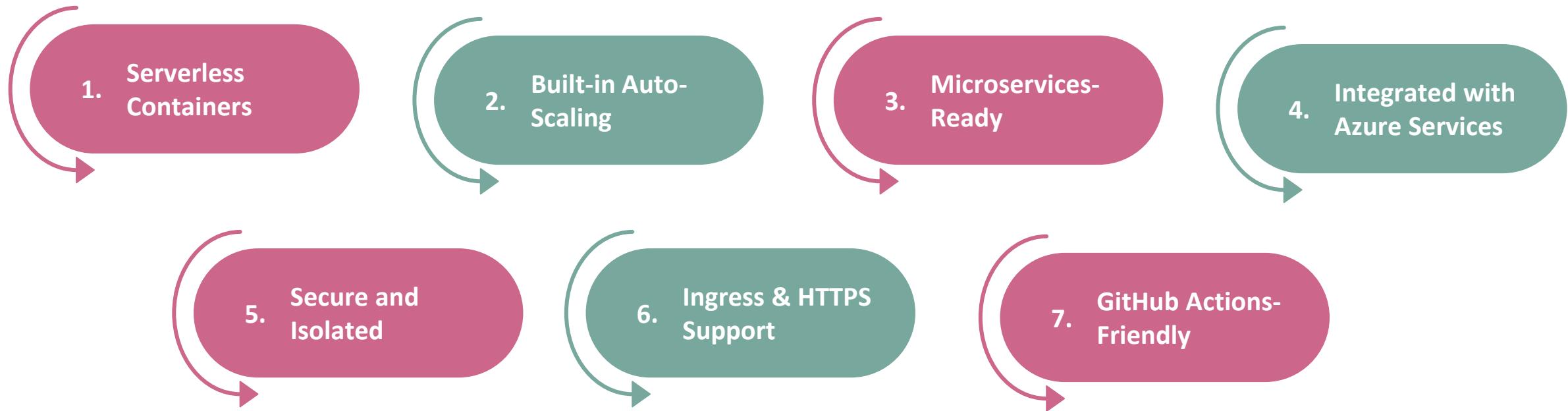


# Overview of Azure Container Apps and Their Benefits

- ✓ Azure Container Apps is a serverless platform designed for deploying containerised applications and microservices with ease and flexibility.
- ✓ It abstracts away the complexity of managing underlying infrastructure, making it ideal for developers who want to focus on code rather than clusters or servers.
- ✓ Container Apps supports both HTTP-based applications and background services, with built-in support for automatic scaling based on HTTP traffic, CPU, memory, or event-driven triggers.
- ✓ It supports stateless and stateful microservices with Dapr integration, allowing container deployment from any registry via Azure Portal, CLI, or GitHub Actions.
- ✓ With support for environment variables, secure secrets via Azure Key Vault, and seamless integration with Azure Monitor, observability is straightforward.

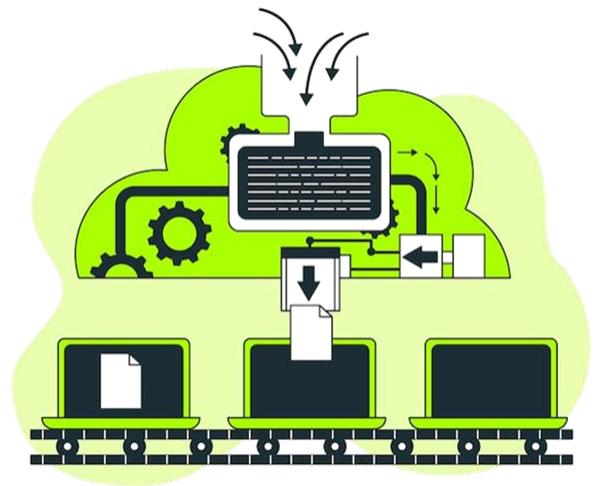
# Overview of Azure Container Apps and Their Benefits

## *Key Benefits of Azure Container Apps*



# Best Practices for Containerised Deployment

- ✓ Deploying applications using containers has become a standard approach for achieving consistency across development, testing, and production environments.
- ✓ When paired with Azure Container Apps, it enables a serverless container experience with built-in scaling, secure networking, and easy integration with Azure services.
- ✓ Simply containerising an app doesn't guarantee efficiency or reliability. Ensuring applications are secure, lightweight, portable, and optimised for cloud-native environments is essential.
- ✓ This includes considerations from how images are built to how services scale and recover. These strategies help teams avoid common deployment pitfalls.
- ✓ Ultimately, they form the foundation for resilient and maintainable containerised solutions on Azure.



# Best Practices for Containerised Deployment

## *Key Best Practices for Containerised Deployment*

Use Lightweight, Purpose-Built Images

01

Build Once, Run Anywhere

02

Leverage Multi-Stage Builds

03

Externalise Configuration with Environment Variables

04

Health Probes and Liveness Checks

05

# Best Practices for Containerised Deployment

## *Key Best Practices for Containerised Deployment*

- 
- 06 Tag Images Strategically
  - 07 Secure Container Registry Access
  - 08 Enable Auto-Scaling
  - 09 Implement Logging and Monitoring
  - 10 Use Immutable Infrastructure Principles

# Best Practices for Containerised Deployment

- 1. Use Lightweight, Purpose-Built Images:** Start with a minimal base image (e.g. Alpine, Distroless) to reduce attack surface and build times. Include only what's essential to run your application to keep images small and secure.
- 2. Build Once, Run Anywhere:** Stick to a consistent CI/CD pipeline where containers are built once and moved across environments (dev, test, prod) without rebuilds. This avoids discrepancies across environments.
- 3. Leverage Multi-Stage Builds:** Use multi-stage Docker builds to separate build tools from the final image. This reduces image size and excludes sensitive build artefacts from production deployments.
- 4. Externalise Configuration with Environment Variables:** Keep environment-specific values (e.g. connection strings, API keys) out of the image. Use Azure App Settings or Azure Key Vault to inject them at runtime via environment variables.
- 5. Health Probes and Liveness Checks:** Implement liveness and readiness probes in Azure Container Apps to ensure your app is started correctly and can receive traffic. This improves availability and enables graceful restarts.

# Best Practices for Containerised Deployment

6. **Tag Images Strategically:** Always use specific version tags (e.g. v1.2.3) instead of latest. This makes deployments reproducible and easier to roll back if needed.
7. **Secure Container Registry Access:** Store container images in Azure Container Registry (ACR) and link it securely with Azure Container Apps using managed identities, avoiding embedded credentials.
8. **Enable Auto-Scaling:** Take advantage of Azure Container Apps' built-in support for scale-in/out based on HTTP traffic or custom metrics (e.g., queue length). This optimises cost and performance.
9. **Implement Logging and Monitoring:** Integrate with Azure Monitor or Log Analytics to capture container logs and metrics. Enable distributed tracing where needed to troubleshoot performance bottlenecks.
10. **Use Immutable Infrastructure Principles:** Avoid patching running containers. Instead, build a new container image for every code or dependency update, test it, and deploy it as a new version.

# Quiz



**Question 1:** What is a key benefit of using Azure Container Apps?

- A) Requires manual infrastructure management
- B) Only supports Windows containers
- C) Provides a serverless platform with built-in automatic scaling
- D) Requires complex cluster configuration



# Quiz



**Question 2:** Why should you use lightweight, purpose-built container images in deployment?

- A) To increase image size and complexity
- B) To embed all environment variables in the image
- C) To avoid multi-stage builds
- D) To reduce attack surface and improve build times



# Quiz



**Question 3:** What is the recommended approach to manage environment-specific configuration in containerised apps?

- A) Embed them directly in the container image
- B) Use environment variables injected at runtime via Azure App Settings or Azure Key Vault
- C) Hard-code values in application code
- D) Store credentials inside container registry images



# Q&A Session

**Question 1:** What does Azure Container Apps abstract for developers, and why is this beneficial?



**Question 2:** How do multi-stage Docker builds help in container image creation?

# Q&A Session

**Question 3:** What role do health probes and liveness checks play in Azure Container Apps?



**Question 4:** Why is it important to tag container images with specific versions instead of “latest”?

# Module 10: ASP.NET Core Security Fundamentals

- Security Best Practices
- Preventing XSS and CSRF
- Using HTTPS for Security
- Data Encryption Techniques



# Security Best Practices

- ✓ Security is a foundational element of any web application, and ASP.NET Core is designed with security in mind.
- ✓ While the framework provides many built-in safeguards, developers must apply best practices to ensure complete protection.
- ✓ These practices help prevent common vulnerabilities such as unauthorised access, data breaches, and injection attacks.
- ✓ Secure coding, proper configuration, and layered defences are essential in minimising risk. Ignoring security considerations can lead to severe legal, financial, and reputational damage.
- ✓ By following well-established principles, developers can build resilient, trustworthy applications that stand strong against evolving threats.



# Security Best Practices

(Continued)

- ✓ ***The following are the essential best practices that help enhance the security of applications built with ASP.NET Core:***

**1.** Enforce Secure Communication

**2.** Manage Secrets Safely

**3.** Control Access Strictly

**4.** Protect Against Cross-Site Request Forgery

**5.** Validate All Inputs

**6.** Structure Middleware Responsibly

**7.** Limit Error Exposure

**8.** Guard Against Abuse

# Preventing XSS and CSRF

- ✓ Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF) are common web vulnerabilities that can compromise user data and application security.
- ✓ Preventing these attacks is essential to safeguard web applications from malicious scripts and unauthorised actions.
- ✓ ***The following are the key practices and implementation steps to prevent XSS and CSRF attacks:***

01

*Understanding XSS and CSRF*

02

*Input Validation and Output Encoding*

03

*Use ASP.NET Core's AntiForgeryToken*

04

*Content Security Policy (CSP)*

05

*HttpOnly and Secure Cookies*

06

*Regular Security Updates and Libraries*

# Preventing XSS and CSRF

## 1. Understanding XSS and CSRF:

- XSS occurs when an attacker injects malicious scripts into trusted websites, while CSRF tricks a user's browser into performing unwanted actions on a web application where they are authenticated.

## 2. Input Validation and Output Encoding:

- Always validate user inputs and encode outputs to prevent injection of malicious scripts that could execute in the browser context.

## 3. Use ASP.NET Core's AntiForgeryToken:

- ASP.NET Core provides built-in CSRF protection via anti-forgery tokens that must be included and validated with each state-changing request.

# Preventing XSS and CSRF

## 4. Content Security Policy (CSP):

- Implement CSP headers to restrict the sources from which scripts can be loaded, reducing the risk of executing malicious code.

## 5. HttpOnly and Secure Cookies:

- Configure cookies with HttpOnly and Secure flags to prevent access from client-side scripts and ensure transmission over HTTPS only.

## 6. Regular Security Updates and Libraries:

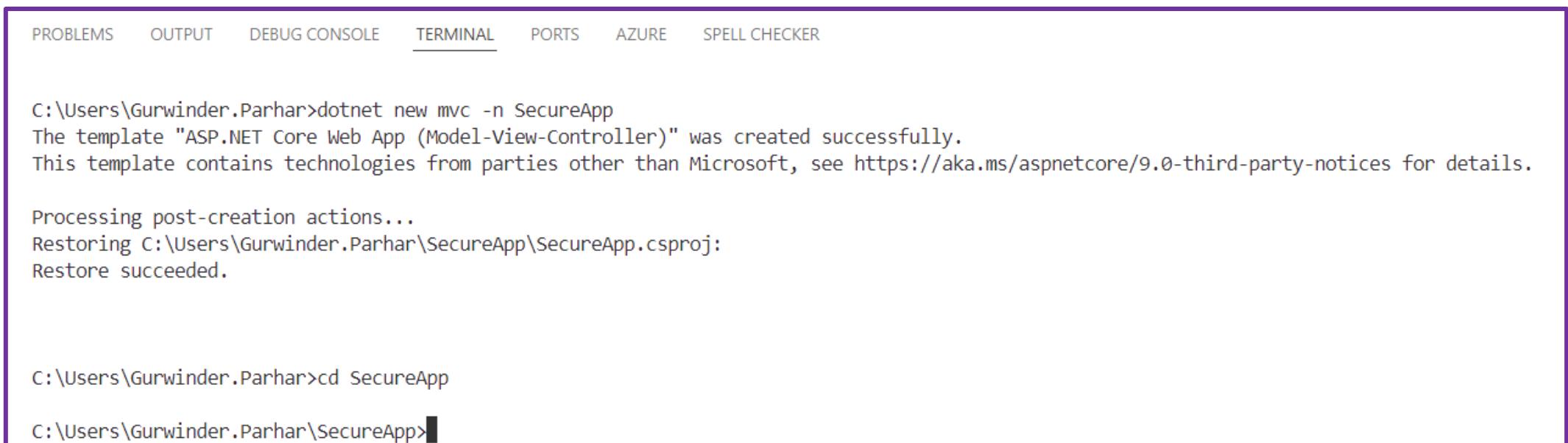
- Keep your frameworks and libraries up to date to benefit from the latest security patches and improvements.

# Preventing XSS and CSRF

(Continued)

- ✓ ***The following are the steps for preventing XSS and CSRF:***

1. **Create a new ASP.NET Core MVC project:** Open your terminal in VSCode or system terminal and run:



A screenshot of the VS Code interface showing the Terminal tab selected. The terminal window displays the command 'dotnet new mvc -n SecureApp' being run, followed by output indicating the template was created successfully and contains third-party technologies. It then shows the process of restoring the project and a successful restore. Finally, the user navigates to the directory 'SecureApp'.

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS AZURE SPELL CHECKER

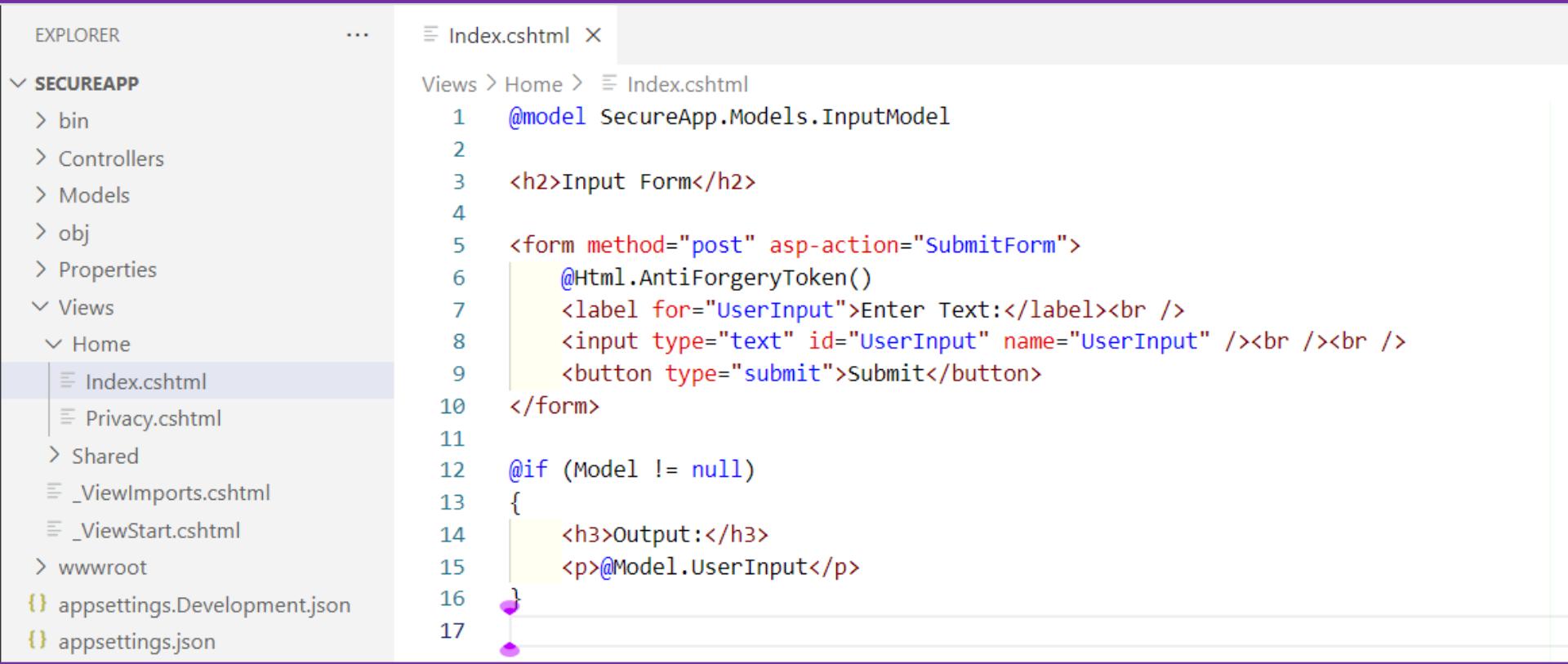
C:\Users\Gurwinder.Parhar>dotnet new mvc -n SecureApp
The template "ASP.NET Core Web App (Model-View-Controller)" was created successfully.
This template contains technologies from parties other than Microsoft, see https://aka.ms/aspnetcore/9.0-third-party-notices for details.

Processing post-creation actions...
Restoring C:\Users\Gurwinder.Parhar\SecureApp\SecureApp.csproj:
Restore succeeded.

C:\Users\Gurwinder.Parhar>cd SecureApp
C:\Users\Gurwinder.Parhar\SecureApp>
```

# Preventing XSS and CSRF

## 2. Add a simple form to test CSRF protection and XSS prevention: Open Views/Home/Index.cshtml

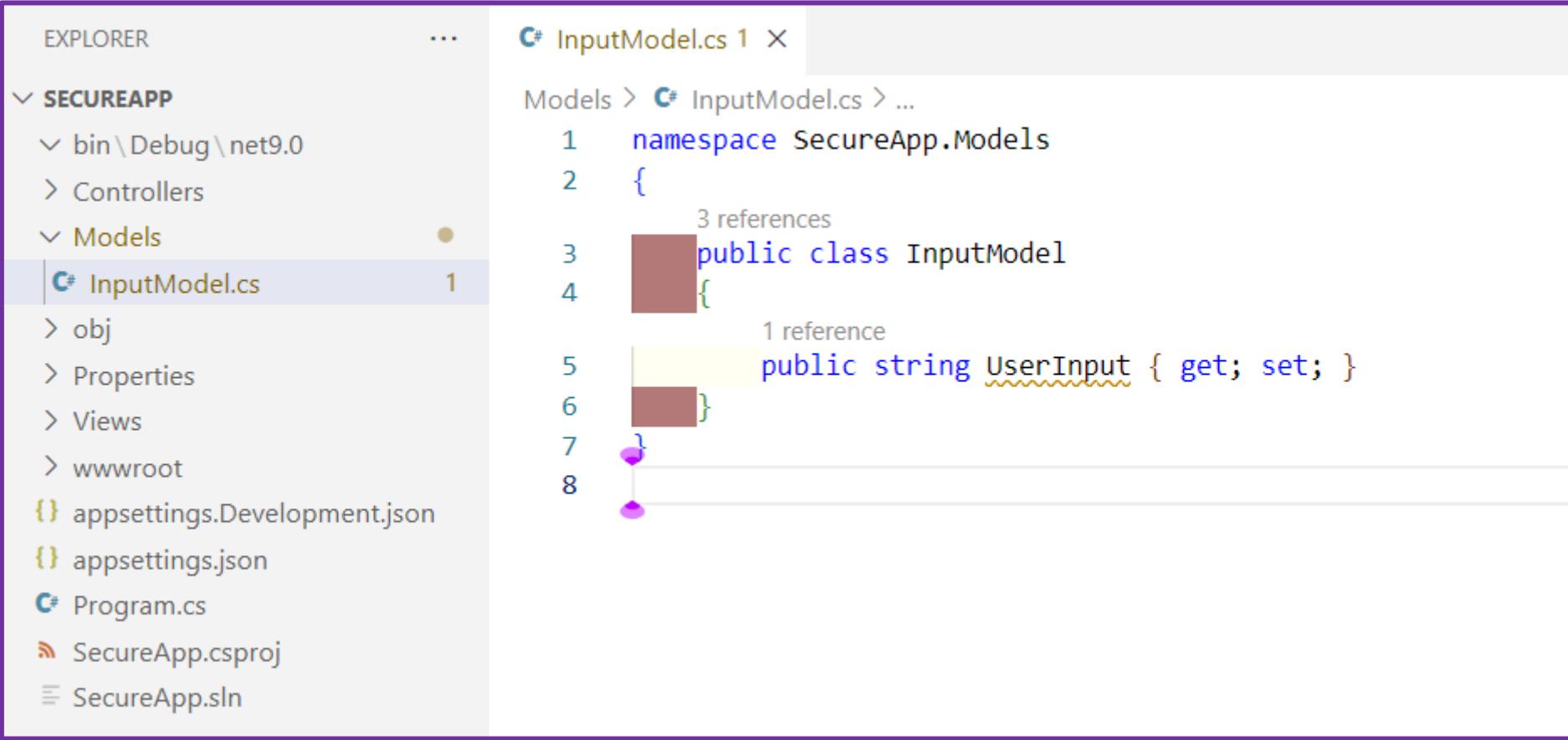


The screenshot shows the Visual Studio code editor with the file `Views/Home/Index.cshtml` open. The code implements a simple input form with CSRF protection and XSS prevention.

```
1  @model SecureApp.Models.InputModel
2
3  <h2>Input Form</h2>
4
5  <form method="post" asp-action="SubmitForm">
6      @Html.AntiForgeryToken()
7      <label for="UserInput">Enter Text:</label><br />
8      <input type="text" id="UserInput" name="UserInput" /><br /><br />
9      <button type="submit">Submit</button>
10     </form>
11
12     @if (Model != null)
13     {
14         <h3>Output:</h3>
15         <p>@Model.UserInput</p>
16     }
17
```

# Preventing XSS and CSRF

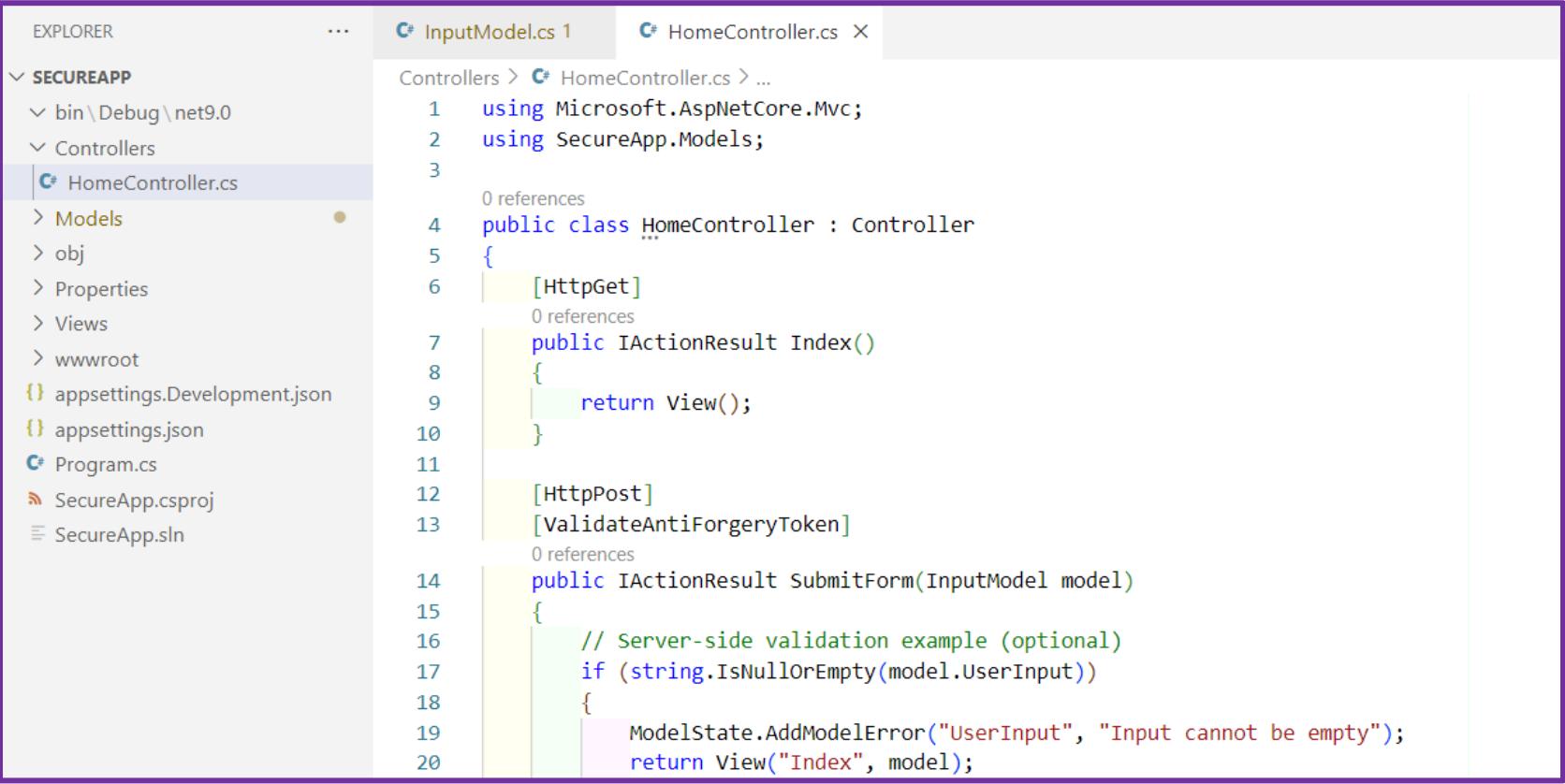
3. Create a model for the input: Create a folder Models. Add a new file **InputModel.cs** in Models folder with:



```
namespace SecureApp.Models
{
    public class InputModel
    {
        public string UserInput { get; set; }
    }
}
```

# Preventing XSS and CSRF

4. Update HomeController to handle form submission: Open Controllers/HomeController.cs and add the following action method:



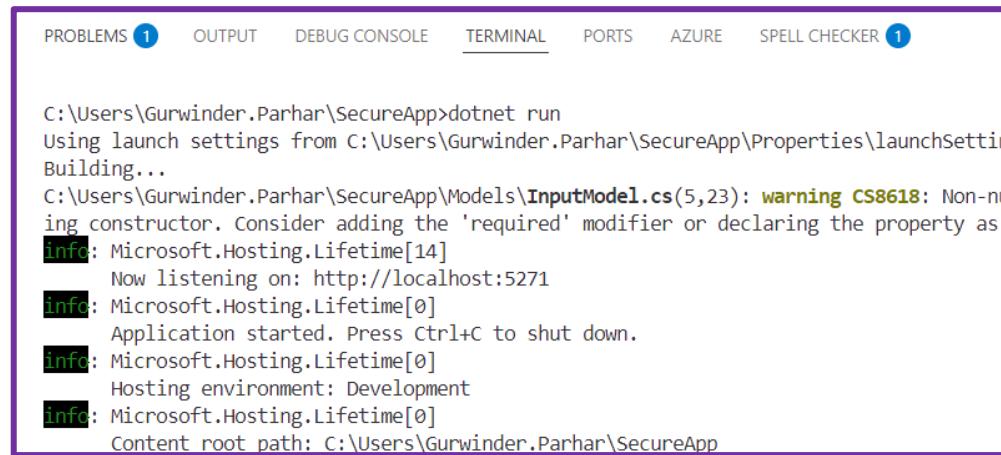
The screenshot shows the Visual Studio IDE interface. On the left, the Project Explorer displays the project structure for 'SECUREAPP' with files like 'HomeController.cs', 'Models', 'obj', 'Properties', 'Views', 'wwwroot', and configuration files 'appsettings.Development.json', 'appsettings.json', 'Program.cs', 'SecureApp.csproj', and 'SecureApp.sln'. The 'HomeController.cs' file is selected in the Explorer and is also the active tab in the main code editor window. The code editor contains the following C# code:

```
1  using Microsoft.AspNetCore.Mvc;
2  using SecureApp.Models;
3
4  public class HomeController : Controller
5  {
6      [HttpGet]
7      public IActionResult Index()
8      {
9          return View();
10     }
11
12     [HttpPost]
13     [ValidateAntiForgeryToken]
14     public IActionResult SubmitForm(InputModel model)
15     {
16         // Server-side validation example (optional)
17         if (string.IsNullOrEmpty(model.UserInput))
18         {
19             ModelState.AddModelError("UserInput", "Input cannot be empty");
20             return View("Index", model);
21         }
22     }
23 }
```

# Preventing XSS and CSRF

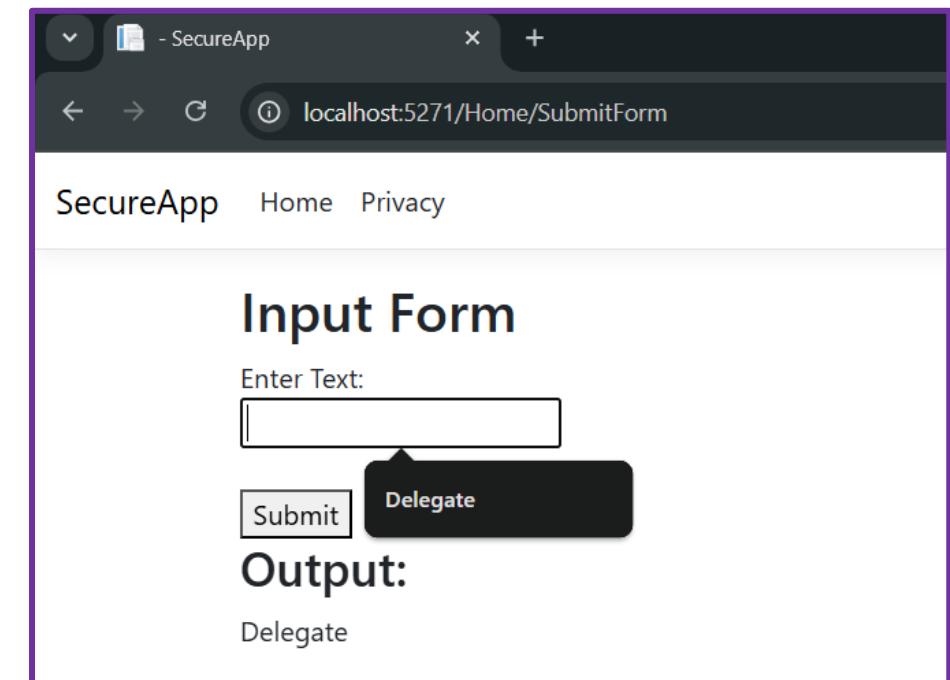
(Continued)

- ✓ **Run the application:** In terminal, run (**dotnet run**).
- ✓ Navigate to the link (<http://localhost:5271>). Enter some **Text** in the input box, then click on **Submit**



```
PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS AZURE SPELL CHECKER 1

C:\Users\Gurwinder.Parhar\SecureApp>dotnet run
Using launch settings from C:\Users\Gurwinder.Parhar\SecureApp\Properties\launchsettings.json...
Building...
C:\Users\Gurwinder.Parhar\SecureApp\Models\InputModel.cs(5,23): warning CS8618: Non-nullable property 'Text' must have a non-null value. Consider adding the 'required' modifier or declaring the property as nullable.
  info: Microsoft.Hosting.Lifetime[14]
    Now listening on: http://localhost:5271
  info: Microsoft.Hosting.Lifetime[0]
    Application started. Press Ctrl+C to shut down.
  info: Microsoft.Hosting.Lifetime[0]
    Hosting environment: Development
  info: Microsoft.Hosting.Lifetime[0]
    Content root path: C:\Users\Gurwinder.Parhar\SecureApp
```



# Using HTTPS for Security

- ✓ HTTPS (Hypertext Transfer Protocol Secure) ensures encrypted communication between the client and server, protecting sensitive data such as login credentials, tokens, and financial information.
- ✓ In ASP.NET Core applications, enforcing HTTPS is essential for maintaining confidentiality, integrity, and trust in web communications. The following are the key practices and considerations when using HTTPS for security:

1 Enforce HTTPS in Application Startup

2 Configure HSTS (HTTP Strict Transport Security)

3 Use Secure Cookies

4 Update launchSettings.json for HTTPS

5 Obtain and Use Valid SSL Certificates

6 Redirect HTTP to HTTPS in Production Environment Only

# Using HTTPS for Security

## 1. Enforce HTTPS in Application Startup:

- Use `app.UseHttpsRedirection()` in the `Startup.cs` to automatically redirect HTTP requests to HTTPS, ensuring all traffic is encrypted.

## 2. Configure HSTS (HTTP Strict Transport Security):

- Enable HSTS using `app.UseHsts()` to instruct browsers to only communicate over HTTPS, preventing protocol downgrade attacks and cookie hijacking.

## 3. Use Secure Cookies:

- Set the `Secure` flag on cookies so they are only transmitted over HTTPS, reducing the risk of session theft or exposure via unencrypted channels.

# Using HTTPS for Security

## 4. Update launchSettings.json for HTTPS:

- Ensure HTTPS profiles are configured correctly in the launchSettings.json file to support local development with secure connections.

## 5. Obtain and Use Valid SSL Certificates:

- Use certificates from trusted Certificate Authorities (CAs) in production environments. ASP.NET Core supports certificates through Kestrel and IIS bindings.

## 6. Redirect HTTP to HTTPS in Production Environment Only:

- Conditionally enable HTTPS redirection and HSTS for production environments to avoid issues with development and local testing setups.

# Data Encryption Techniques

- ✓ Data encryption is a critical security measure that protects sensitive information by converting it into unreadable ciphertext, ensuring confidentiality and preventing unauthorised access.
- ✓ In ASP.NET Core applications, encryption safeguards data both at rest and in transit. The following are the key data encryption techniques used in securing ASP.NET Core applications:

1

Symmetric Encryption

3

Hashing

5

Data Protection API (DPAPI)

2

Asymmetric Encryption

4

Key Derivation Functions (KDFs)

6

Transport Layer Security (TLS)

# Data Encryption Techniques

## 1. Symmetric Encryption:

- This technique uses a single secret key for both encryption and decryption, offering fast performance. However, secure key management is essential to prevent unauthorised access.

## 2. Asymmetric Encryption:

- Utilises a pair of keys—public and private—for encryption and decryption respectively. It enables secure key exchange and is widely used in SSL/TLS protocols to protect data transmission.

## 3. Hashing:

- Converts data into a fixed-length string of characters, which cannot be reversed to the original data. Hashing is commonly used for storing passwords securely using algorithms like SHA-256 and bcrypt.

# Data Encryption Techniques

## 4. Key Derivation Functions (KDFs):

- KDFs generate cryptographic keys from passwords or other sources, adding computational difficulty to deter brute-force attacks. Examples include PBKDF2 and Argon2.

## 5. Data Protection API (DPAPI):

- ASP.NET Core's built-in Data Protection API provides a framework for encrypting and decrypting data, managing keys automatically, and integrating with application services.

## 6. Transport Layer Security (TLS):

- Encrypts data transmitted over networks using protocols such as HTTPS, ensuring data integrity and confidentiality between client and server communications.

# Quiz



**Question 1:** What is the primary purpose of ASP.NET Core's AntiForgeryToken?

- A) To encrypt sensitive data
- B) To prevent Cross-Site Request Forgery (CSRF) attacks
- C) To improve API response times
- D) To authenticate users



# Quiz



**Question 2:** Which method is recommended to enforce secure HTTPS connections in an ASP.NET Core app?

- A) Use `app.UseHttpRedirection()`
- B) Use `app.UseHttpsRedirection()`
- C) Disable HTTPS for development
- D) Use `app.UseHttpCompression()`



# Quiz



**Question 3:** Which of the following is an example of symmetric encryption?

- A) Public and private key pair
- B) Hashing passwords with bcrypt
- C) Using a single secret key for both encryption and decryption
- D) Using TLS for network encryption



# Q&A Session

**Question 1:** What are XSS and CSRF attacks, and how does ASP.NET Core help prevent them?



**Question 2:** How does HTTPS improve security for ASP.NET Core applications?

# Q&A Session

**Question 3:** What is the difference between symmetric and asymmetric encryption techniques?



**Question 4:** What role does the Data Protection API (DPAPI) play in ASP.NET Core security?

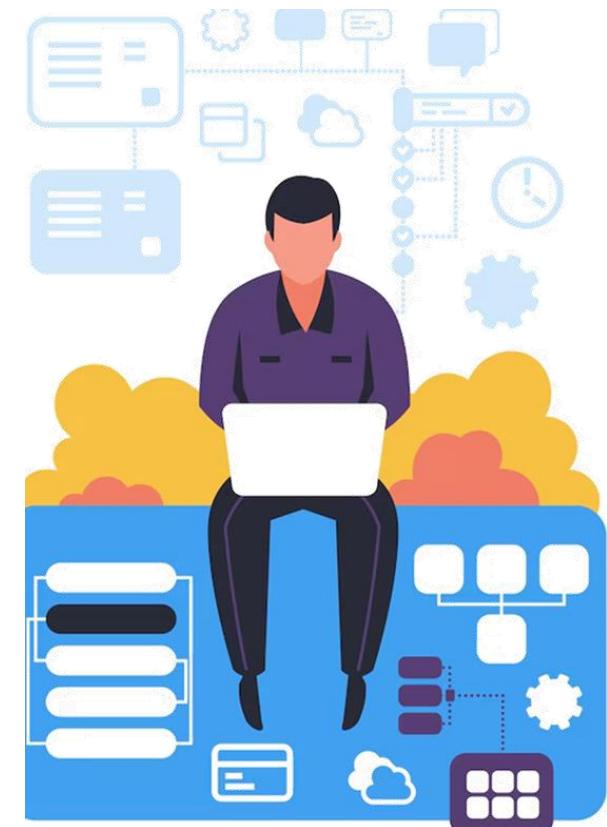
# Module 11: Asynchronous Programming in ASP.NET Core

- Async and Await Basics
- Handling Long-Running Tasks
- Performance Benefits of Asynchronous Programming
- Task-Based Programming for Web APIs



# Async and Await Basics

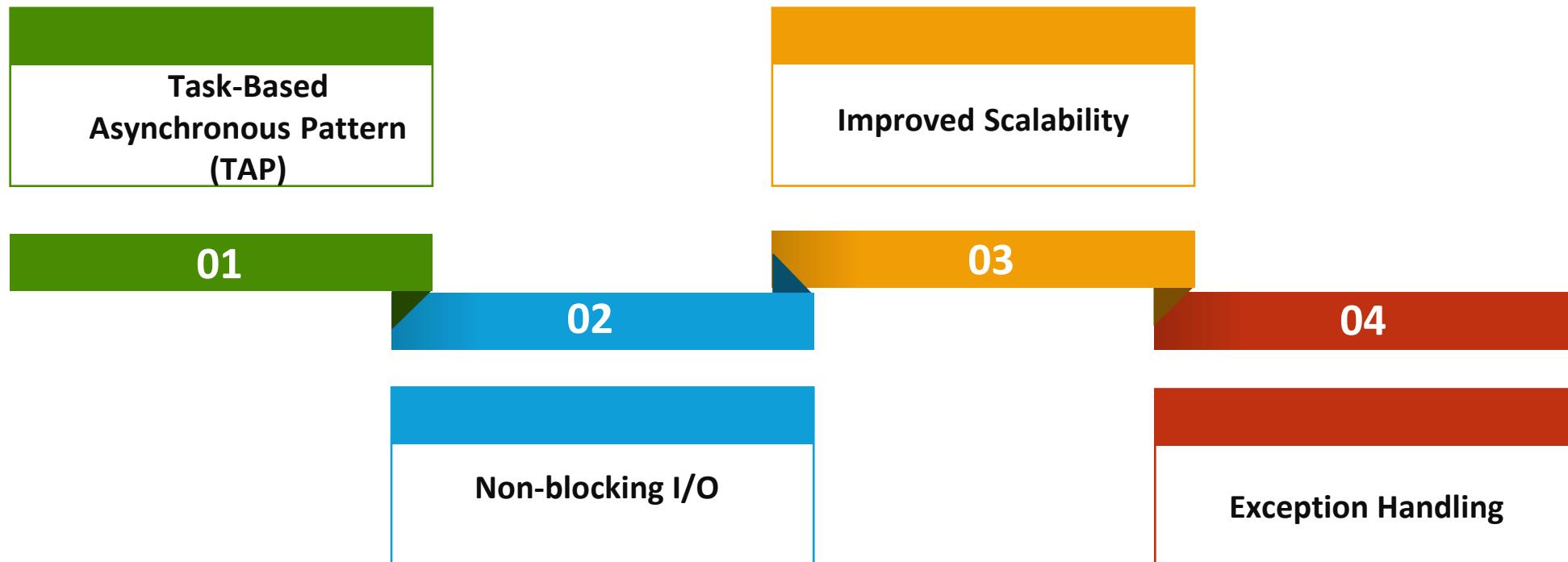
- ✓ Asynchronous programming is a technique that allows a program to perform multiple operations concurrently, improving responsiveness and scalability, especially in web applications.
- ✓ In ASP.NET Core, asynchronous methods enable non-blocking operations, such as I/O-bound tasks like database access, web service calls, or file handling, allowing the server to handle more requests efficiently.
- ✓ This approach helps prevent thread blocking, which can degrade application performance under high load.
- ✓ Additionally, it enhances the user experience by reducing latency and enabling faster response times.



# Async and Await Basics

(Continued)

- ✓ *The following are the key concepts of async and await basics:*



# Async and Await Basics

## 1. Task-Based Asynchronous Pattern (TAP):

- ASP.NET Core leverages the TAP model using `async` and `await` keywords to simplify writing asynchronous code.
- This pattern improves code readability and maintainability by allowing developers to write asynchronous logic in a linear, easy-to-understand style.

## 2. Non-blocking I/O:

- Asynchronous operations free up server threads while waiting for external resources, increasing throughput and reducing thread starvation.
- This allows the server to allocate resources dynamically and respond promptly to incoming requests without unnecessary delays.

# Async and Await Basics

## 3. Improved Scalability:

- By avoiding thread blocking, asynchronous programming enhances the ability of the web server to process numerous simultaneous requests.
- This efficiency is particularly important in high-traffic applications where resource management directly impacts performance.

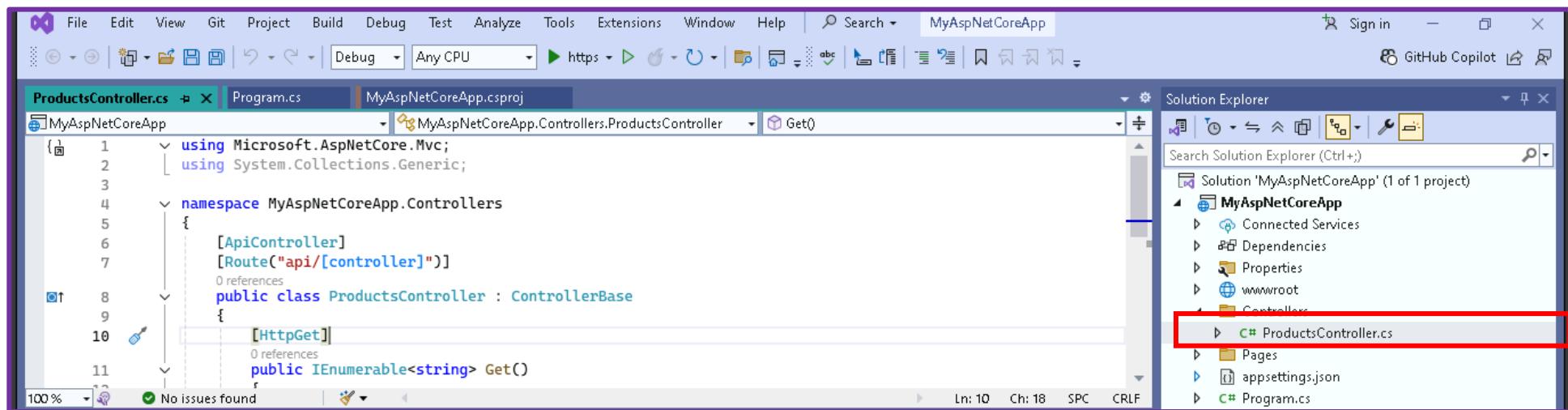
## 4. Exception Handling:

- Asynchronous methods require careful exception handling, often using try-catch blocks with async methods.
- Developers must ensure that exceptions are properly awaited and handled to avoid unobserved exceptions and maintain application stability.

# Handling Long-Running Tasks

- ✓ Handling long-running tasks is essential to maintain application responsiveness and avoid blocking critical threads. In ASP.NET Core, asynchronous programming techniques help manage these tasks efficiently without degrading user experience or server performance.

## Step 1: Open your controller file



# Handling Long-Running Tasks

**Step 2:** Enter the code to replace the old method signature

The screenshot shows the Visual Studio IDE interface. The main window displays the `ProductsController.cs` file under the `MyAspNetCoreApp` project. The code implements an `ApiController` with a `[HttpGet]` action named `Get()`. Inside this action, there is a simulated asynchronous task delay of 2000 milliseconds, followed by a return statement providing a list of three product names. The Solution Explorer on the right shows the project structure, including the `Controllers` folder containing the `ProductsController.cs` file.

```
File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help | Search | MyAspNetCoreApp | Sign in | GitHub Copilot |
```

```
ProductsController.cs | Program.cs | MyAspNetCoreApp.csproj
```

```
MyAspNetCoreApp | MyAspNetCoreApp.Controllers.ProductsController | Get()
```

```
1 using Microsoft.AspNetCore.Mvc;
2 using System.Collections.Generic;
3 using System.Threading.Tasks;
4
5 namespace MyAspNetCoreApp.Controllers
6 {
7     [ApiController]
8     [Route("api/{controller}")]
9     public class ProductsController : ControllerBase
10    {
11        [HttpGet]
12        public async Task<IEnumerable<string>> Get()
13        {
14            // Your async code here
15            await Task.Delay(2000); // Simulated async work
16
17            return new string[] { "Product1", "Product2", "Product3" };
18        }
19    }
20}
21
```

```
100% | No issues found | Line: 21 Ch: 1 SPC CRLF
```

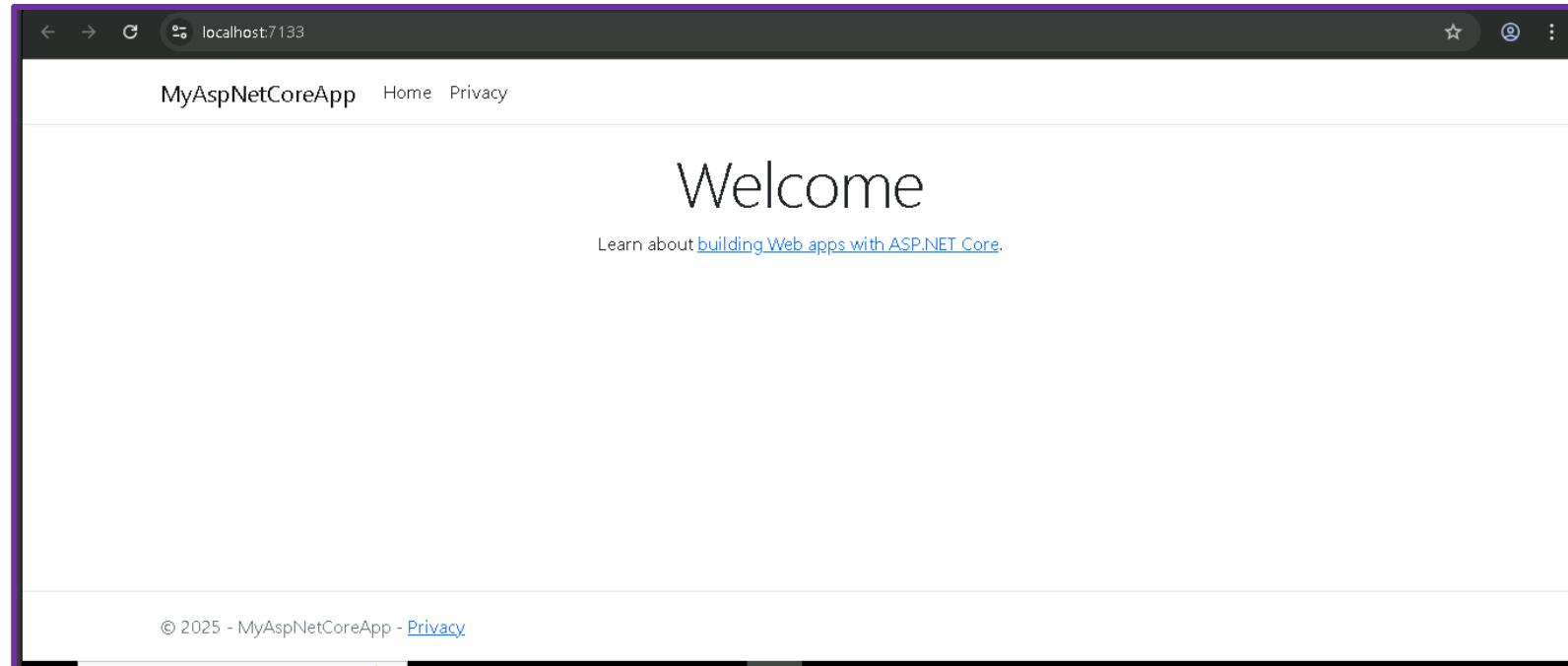
```
Solution Explorer | Search Solution Explorer (Ctrl+.)
```

```
Solution 'MyAspNetCoreApp' (1 of 1 project)
MyAspNetCoreApp
    Connected Services
    Dependencies
    Properties
    wwwroot
    Controllers
        ProductsController.cs
    Pages
    appsettings.json
    Program.cs
```

# Handling Long-Running Tasks

(Continued)

- ✓ It remove extra Get methods or duplicate declarations outside the class



# Performance Benefits of Asynchronous Programming

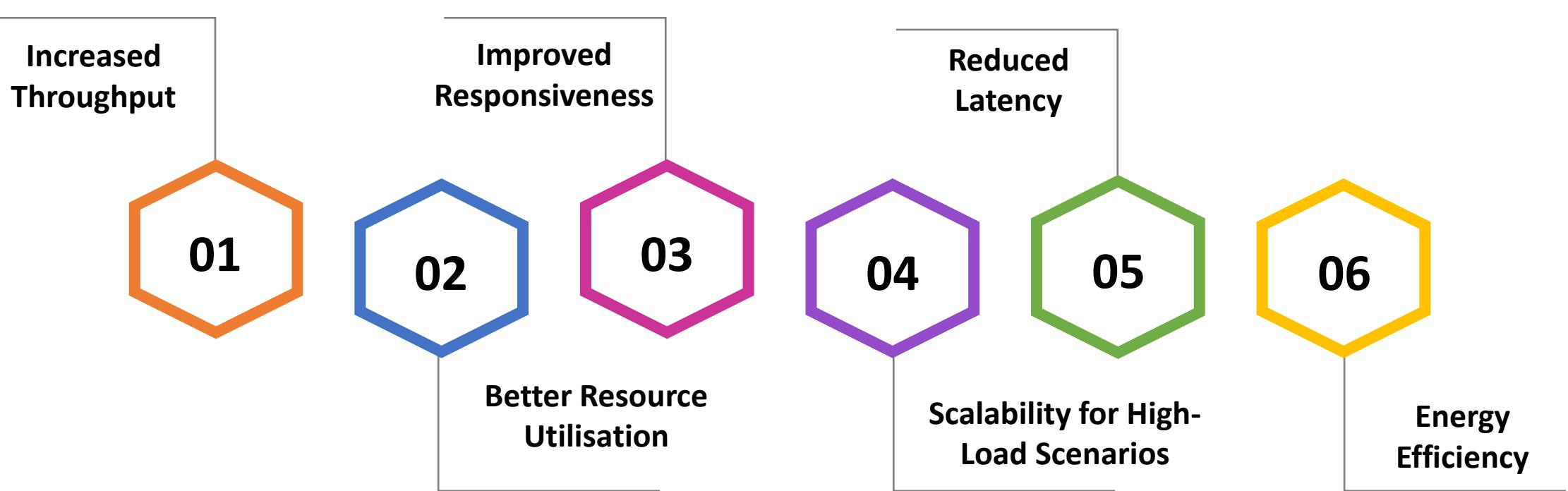
- ✓ Asynchronous programming offers significant performance advantages, particularly in web applications where handling numerous simultaneous requests efficiently is critical.
- ✓ By allowing operations to run without blocking threads, asynchronous programming maximises resource utilisation and enhances overall application responsiveness.
- ✓ This approach helps avoid bottlenecks caused by thread contention and improves the ability to scale applications under heavy load.
- ✓ Additionally, it contributes to better user experiences by minimising wait times and enabling faster processing of client requests.



# Performance Benefits of Asynchronous Programming

(Continued)

- ✓ *The following are the key performance benefits of asynchronous programming:*



# Performance Benefits of Asynchronous Programming

## 1. Increased Throughput:

- In synchronous programming, threads are blocked while waiting for I/O operations (such as database queries, file access, or network calls) to complete.
- This blocking limits the number of requests the server can process concurrently, as each blocked thread cannot handle other work.
- Asynchronous programming releases threads during these waits, allowing the server to handle more requests simultaneously and improving throughput.

## 2. Better Resource Utilisation:

- Threads consume system resources such as memory and CPU cycles. By freeing threads during I/O waits, asynchronous programming reduces the number of active threads required to serve requests.
- This leads to lower memory consumption and reduces context-switching overhead, which improves overall system efficiency.

# Performance Benefits of Asynchronous Programming

## 3. Improved Responsiveness:

- Applications using asynchronous programming can remain responsive under heavy load because they are not stalled by long-running operations.
- This is especially important for user-facing web applications where delays translate to poor user experience.

## 4. Scalability for High-Load Scenarios:

- Asynchronous code scales more effectively in environments with high concurrency demands. It prevents thread pool starvation, where too many threads are blocked, which can cause request queuing and increased latency.
- By enabling efficient thread reuse, asynchronous programming supports thousands of concurrent requests without proportional increases in hardware resources.

# Performance Benefits of Asynchronous Programming

## 5. Reduced Latency:

- Since asynchronous operations can run concurrently, they can reduce the overall latency of complex workflows involving multiple I/O-bound tasks.
- This concurrency allows faster completion times compared to sequential synchronous execution.

## 6. Energy Efficiency:

- Efficient use of CPU and memory resources through asynchronous programming can lead to reduced power consumption on servers, which is beneficial for data center cost savings and environmental impact.

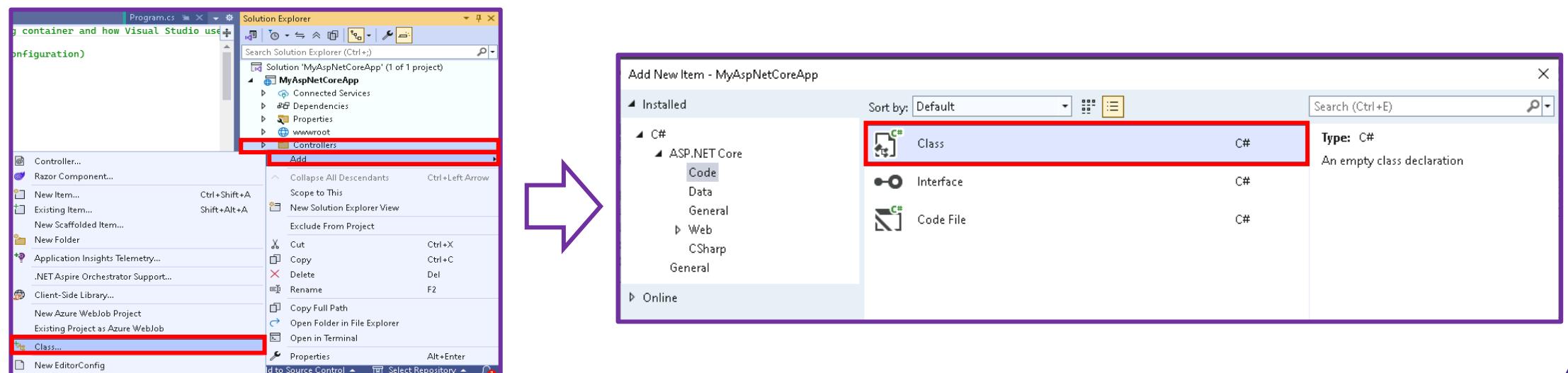


# Task-Based Programming for Web APIs

- ✓ Task-Based Programming is a modern approach to handling asynchronous operations in Web APIs, enabling efficient and scalable processing of client requests. By leveraging tasks, developers can write non-blocking code that improves application responsiveness and resource utilisation.
- ✓ *The following are the steps for test-based programming for web APIs:*

**Step 1:** Right click on **Controllers** folder, click on **Add > Class..**

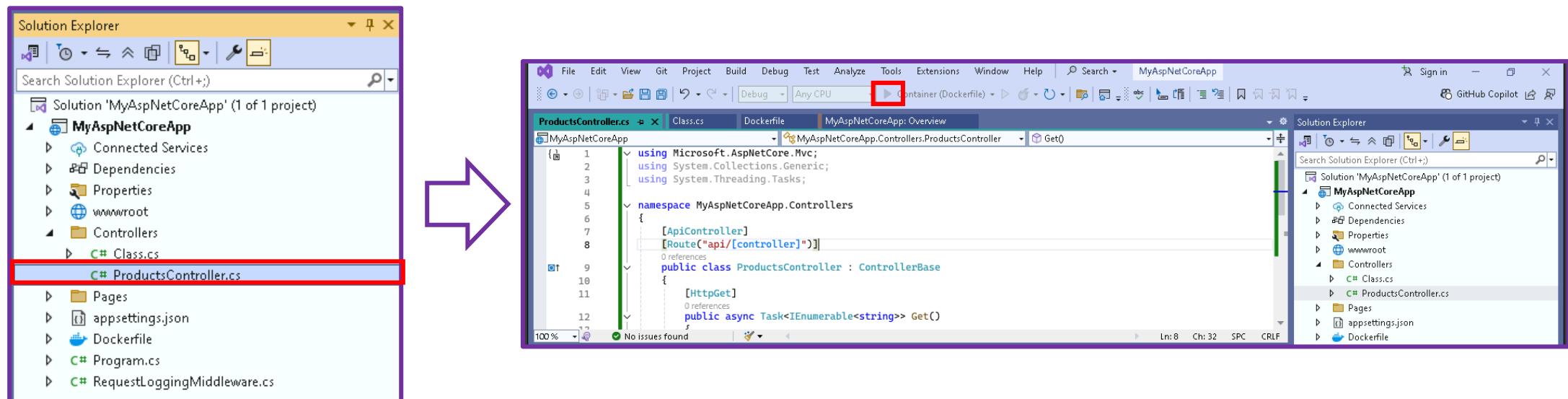
**Step 2:** Double click on Class



# Task-Based Programming for Web APIs

**Step 3:** Next, name it **ProductsController.cs**

**Step 4:** Enter the code and then click on play button



# Task-Based Programming for Web APIs

(Continued)

- ✓ This confirms that your asynchronous Get method in ProductsController is working properly and the API is responding as intended.

The screenshot shows the Visual Studio IDE interface. The main area displays the `ProductsController.cs` file, which contains the following code:

```
1  using Microsoft.AspNetCore.Mvc;
2  using System.Collections.Generic;
3  using System.Threading.Tasks;
4
5  namespace MyAspNetCoreApp.Controllers
6  {
7      [ApiController]
8      [Route("api/[controller]")]
9      public class ProductsController : ControllerBase
10     {
11         [HttpGet]
12         public async Task<IEnumerable<string>> Get()
13         {
14             return new List<string> { "value1", "value2" };
15         }
16     }
17 }
```

The `Output` window at the bottom shows the build logs:

```
Build started at 21:04...
1>----- Build started: Project: MyAspNetCoreApp, Configuration: Debug Any CPU -----
1> Skipping analyzers to speed up the build. You can execute 'Build' or 'Rebuild' command to run analyzers.
1>MyAspNetCoreApp -> C:\Users\training\source\repos\MyAspNetCoreApp\MyAspNetCoreApp\bin\Debug\net8.0\MyAspNetCoreApp.dll
1>"C:\Program Files\Docker\resources\bin\docker.exe" build -f "C:\Users\training\source\repos\MyAspNetCoreApp\MyAspNetCoreApp\Dock
```

# Quiz



**Question 1:** What is the primary benefit of asynchronous programming in ASP.NET Core?

- A) Simplifies synchronous code execution
- B) Improves responsiveness and scalability by avoiding thread blocking
- C) Increases memory usage by blocking threads
- D) Eliminates the need for exception handling



# Quiz



**Question 2:** Which programming pattern does ASP.NET Core leverage for asynchronous methods?

- A) Event-Driven Programming
- B) Task-Based Asynchronous Pattern (TAP)
- C) Callback Pattern
- D) Polling Pattern



# Quiz



**Question 3:** Why is exception handling important in asynchronous methods?

- A) Because async methods don't throw exceptions
- B) To avoid unobserved exceptions that can cause application instability
- C) Because synchronous methods handle exceptions better
- D) To prevent deadlocks in synchronous code



# Q&A Session

**Question 1:** How does asynchronous programming improve scalability in ASP.NET Core applications?



**Question 2:** What is non-blocking I/O and why is it important?

# Q&A Session

**Question 3:** What are some key concepts behind the Task-Based Asynchronous Pattern (TAP) used in ASP.NET Core?



**Question 4:** How should long-running tasks be handled in ASP.NET Core controllers?

# Module 12: Dependency Injection and Middleware in ASP.NET Core

- Introduction to Dependency Injection
- Using Middleware in ASP.NET Core
- Service Lifetimes and Best Practices
- Creating Custom Middleware



# Introduction to Dependency Injection

- ✓ Dependency Injection (DI) is a design pattern used to achieve Inversion of Control (IoC) between classes and their dependencies.
- ✓ Instead of a class creating its own dependencies, they are provided to the class externally, promoting loose coupling and enhancing testability, maintainability, and flexibility of applications.
- ✓ In ASP.NET Core, DI is a first-class citizen built into the framework. It allows developers to register services and their implementations in a central container during application startup.
- ✓ These services can then be injected into constructors of classes (such as controllers, middleware, or other services) where they are needed.
- ✓ This approach simplifies managing dependencies and promotes modular architecture by clearly defining the relationships between components without hardcoding them.

# Introduction to Dependency Injection

(Continued)

- ✓ ***ASP.NET Core provides three main lifetimes for service registration:***
  1. Transient: Services are created each time they are requested.
  2. Scoped: Services are created once per request.
  3. Singleton: Services are created once and shared throughout the application lifetime.
- ✓ Dependency Injection (DI) is a key principle in modern software development that helps decouple the creation of dependent objects from their usage.
- ✓ This leads to more maintainable, testable, and flexible applications.

# Introduction to Dependency Injection

## *Why Use Dependency Injection?*

1. **Loose Coupling:** DI reduces tight coupling between classes by injecting dependencies, making it easier to swap implementations without changing dependent code.
2. **Improved Testability:** With DI, dependencies can be mocked or stubbed during testing, facilitating unit testing and increasing code coverage.
3. **Centralised Configuration:** ASP.NET Core provides a built-in DI container that centralises service registration, making management of dependencies easier and consistent.
4. **Lifecycle Management:** DI allows control over the lifecycle of services through service lifetimes (Transient, Scoped, Singleton), which optimises resource usage.

# Introduction to Dependency Injection

## ***How DI Works in ASP.NET Core***

### **1. Service Registration:**

- Services and their implementations are registered in the Startup.
- ConfigureServices method or Program.cs in newer templates. This defines how and when instances of these services are created.

### **2. Service Resolution:**

- When a class requires a service, it declares the dependency in its constructor (Constructor Injection).
- The framework's DI container automatically injects the registered service implementation at runtime.

# Introduction to Dependency Injection

## 3. Service Lifetimes:

- **Transient:** A new instance is provided every time the service is requested. Suitable for lightweight, stateless services.
- **Scoped:** A single instance is used throughout a single client request. Ideal for database contexts or request-specific data.
- **Singleton:** A single instance is created and shared across the entire application's lifetime. Use with caution for services maintaining state.



# Introduction to Dependency Injection

## *Common DI Patterns in ASP.NET Core*

- 
- 1. Constructor Injection:** Most common, dependencies are passed through the constructor.
  - 2. Method Injection:** Dependencies are passed as method parameters, useful in some cases.
  - 3. Property Injection:** Dependencies are assigned to public properties (less common in ASP.NET Core).

## *Built-in Services with DI*

- ASP.NET Core itself leverages DI extensively. Many framework services like logging (ILogger), configuration ( IConfiguration), and database contexts (DbContext) are registered and injected using DI.

# Using Middleware in ASP.NET Core

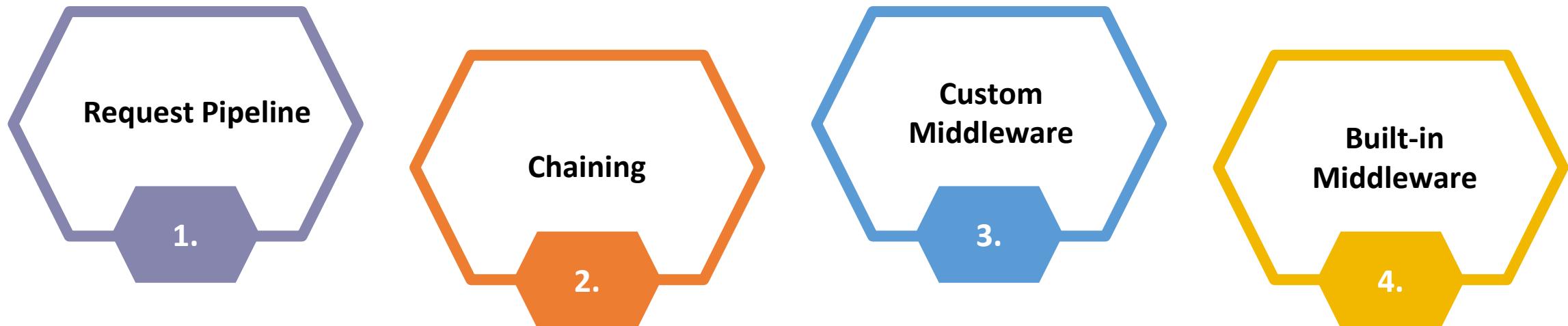
- ✓ Middleware in ASP.NET Core is software that is assembled into an application pipeline to handle requests and responses.
- ✓ Each component in the middleware pipeline can inspect, modify, or short-circuit HTTP requests and responses as they pass through the pipeline.
- ✓ Middleware components are executed in the order in which they are registered in the Program.cs (or Startup.cs in older templates) file.
- ✓ ASP.NET Core embraces a modular and lightweight architecture, and middleware plays a critical role in configuring how the application behaves during HTTP request processing.
- ✓ Middleware can perform actions such as authentication, logging, routing, error handling, compression, and more.



# Using Middleware in ASP.NET Core

(Continued)

- ✓ *The following are the key concepts of Middleware:*



# Using Middleware in ASP.NET Core

- 1. Request Pipeline:** Middleware components are invoked sequentially to process incoming HTTP requests.
- 2. Chaining:** Each middleware can pass control to the next component in the pipeline using `await _next(context);`.
- 3. Custom Middleware:** Developers can create their own middleware to encapsulate reusable logic for handling cross-cutting concerns.
- 4. Built-in Middleware:** ASP.NET Core includes many ready-to-use middleware like `UseRouting`, `UseAuthentication`, `UseAuthorisation`, `UseEndpoints`, etc.



# Using Middleware in ASP.NET Core

## ***Common Use Cases***

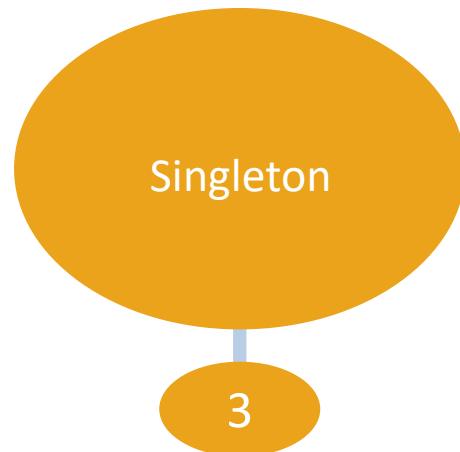
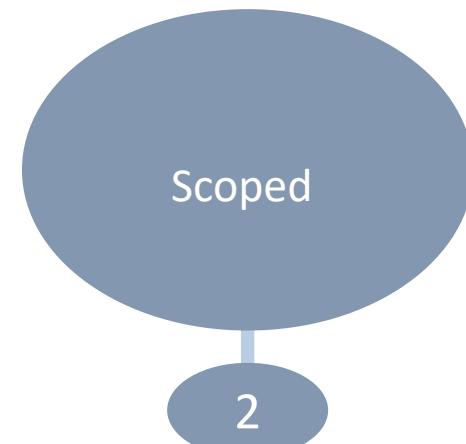
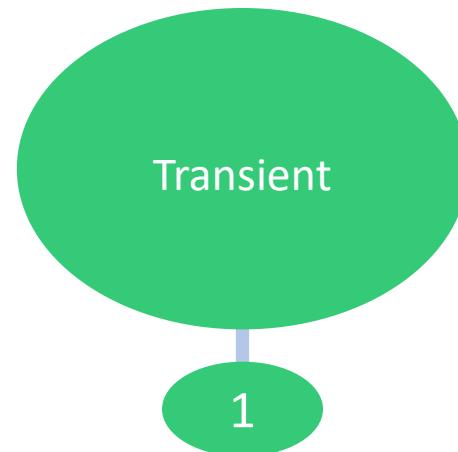
1. Logging and diagnostics
2. Error handling
3. Authentication and Authorization
4. Response compression
5. CORS (Cross-Origin Resource Sharing)
6. Session and cookie management



# Service Lifetimes and Best Practices

## *Service Lifetimes in ASP.NET Core*

- In ASP.NET Core, service lifetimes determine the duration that a service instance is retained and reused within the application. Understanding these lifetimes is essential for designing efficient, scalable, and thread-safe applications.



# Service Lifetimes and Best Practices

## 1. Transient

- A new instance of the service is created every time it is requested.
- Use for lightweight, stateless services that don't maintain any shared data.
- Example: formatting services, utility classes.

## 2. Scoped

- A new instance is created once per client request (HTTP request). Use for services that should be consistent within a request but isolated across requests.
- Commonly used for database contexts (e.g., Entity Framework DbContext).
- Prevents shared state between concurrent requests, avoiding thread safety issues.

# Service Lifetimes and Best Practices

## 3. Singleton

- A single instance is created once for the lifetime of the application and shared across all requests.
- Use for services that maintain global state or expensive resources, such as caching or configuration providers.
- Be cautious: Singleton services are shared across threads, so they must be thread-safe.
- Prevents shared state between concurrent requests, avoiding thread safety issues



# Service Lifetimes and Best Practices

## ***Best Practices***

### **1. Choose Appropriate Service Lifetimes:**

- Select the correct lifetime (Transient, Scoped, or Singleton) based on the service's responsibility and usage to optimise performance and resource management.

### **2. Avoid Injecting Scoped Services into Singletons:**

- Injecting scoped services into singleton services can cause unintended shared state and potential memory leaks; instead, use factory patterns or refactor lifetimes appropriately.

### **3. Register Services via Interfaces:**

- Always register services using interfaces rather than concrete implementations to promote loose coupling and improve testability.

# Service Lifetimes and Best Practices

## 4. Dispose of IDisposable Services Properly:

- Ensure that services implementing IDisposable are registered and managed correctly by the DI container to prevent resource leaks.

## 5. Minimise Use of Singleton Services with Mutable State:

- If singletons hold mutable state, implement thread-safety measures to avoid concurrency issues in multi-threaded environments.

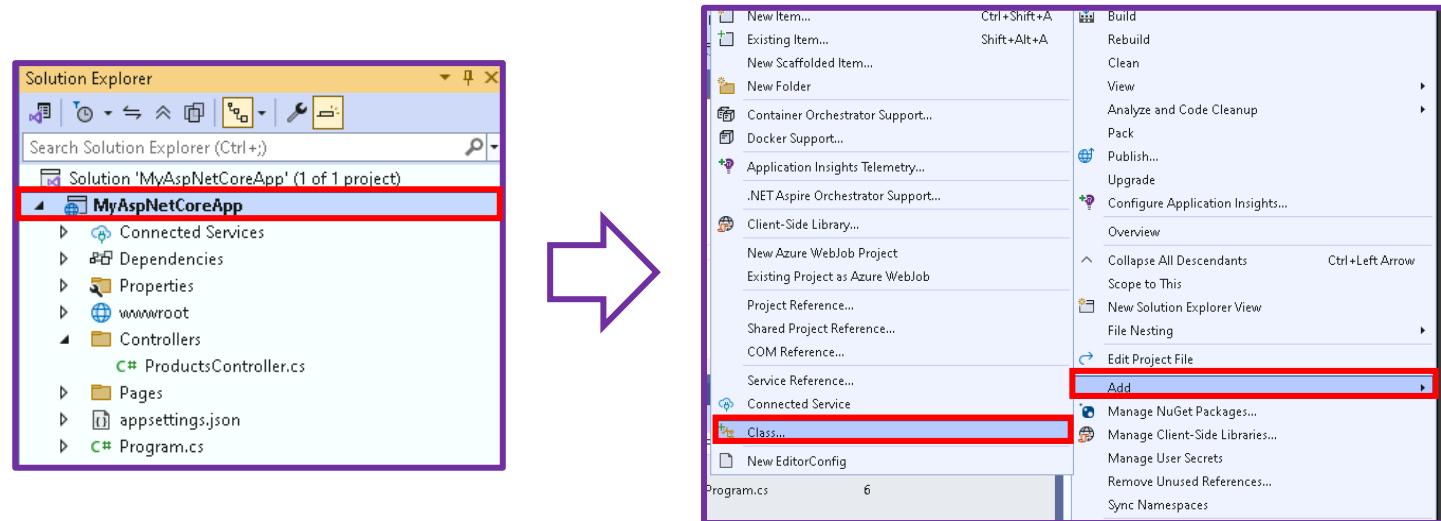


# Creating Custom Middleware

- ✓ Creating custom middleware in ASP.NET Core allows developers to insert their own processing logic into the HTTP request pipeline. This enables handling tasks such as logging, authentication, or modifying requests and responses to meet specific application needs.
- ✓ ***The following are the steps to create custom middleware:***

**Step 1:** Right-click on your **project** in the **Solution Explorer**

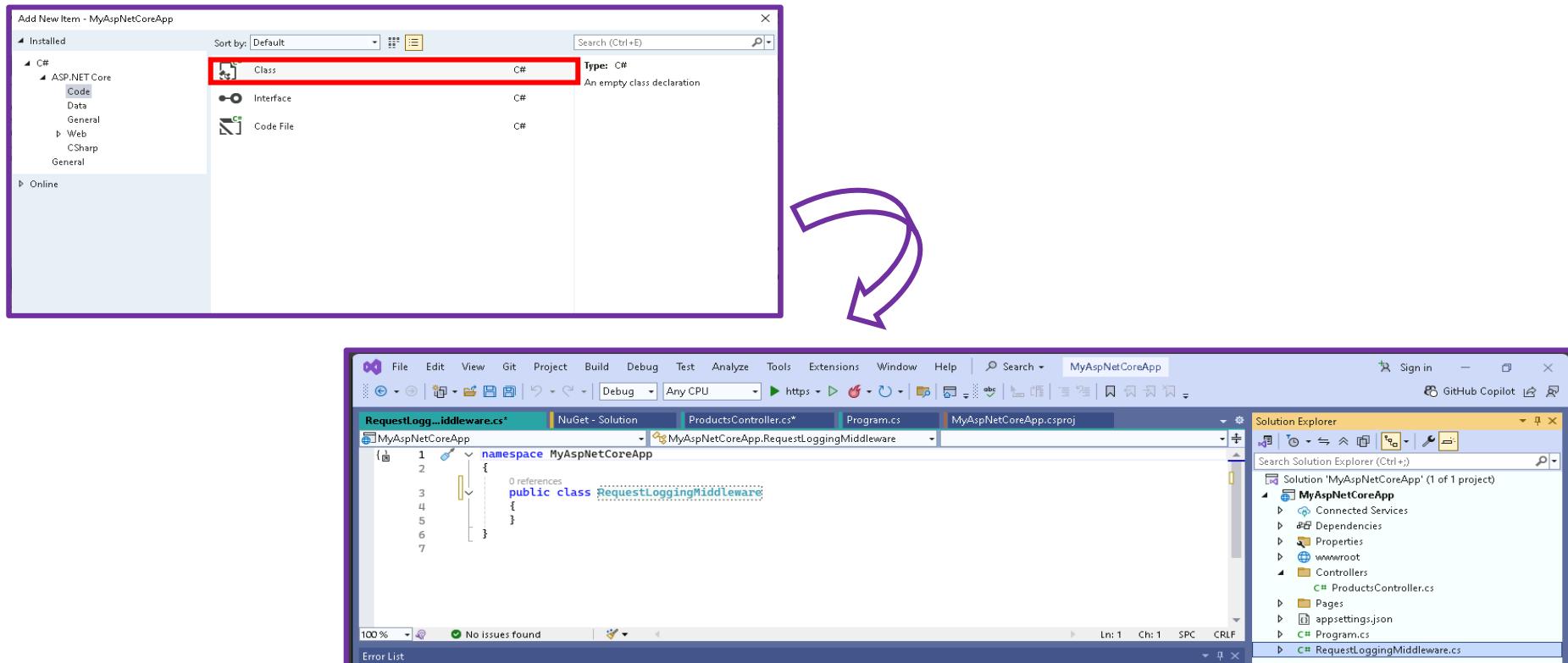
**Step 2:** Select **Add → Class...**



# Creating Custom Middleware

## Step 3: Double-click on Class

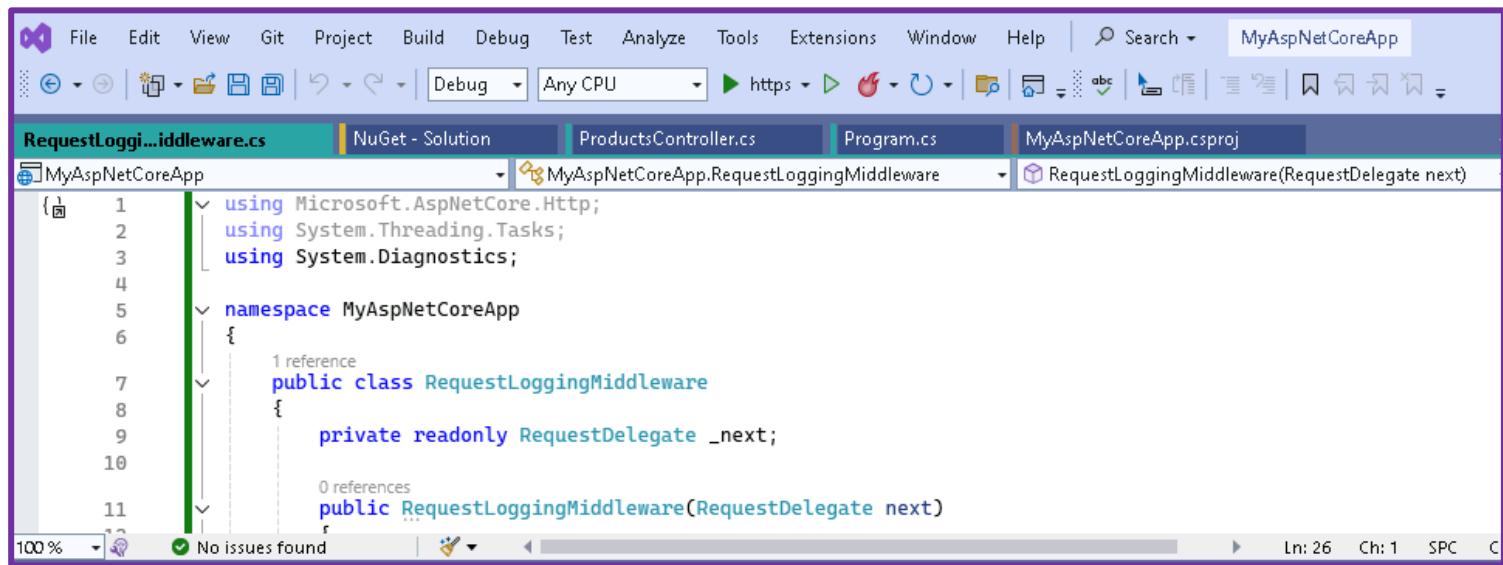
- ✓ The class named **RequestLoggingMiddleware** has been created successfully and matches the required middleware name.



# Creating Custom Middleware

**Step 4:** Next enter the code to implement the middleware logic inside this class

- ✓ The middleware has been created successfully.



The screenshot shows the Visual Studio IDE interface with the following details:

- Title Bar:** MyAspNetCoreApp
- Solution Explorer:** Shows files like RequestLoggingMiddleware.cs, ProductsController.cs, Program.cs, and MyAspNetCoreApp.csproj.
- Toolbars:** Standard Visual Studio toolbars for file operations, search, and navigation.
- Code Editor:** The RequestLoggingMiddleware.cs file is open, displaying the following C# code:

```
1  using Microsoft.AspNetCore.Http;
2  using System.Threading.Tasks;
3  using System.Diagnostics;
4
5  namespace MyAspNetCoreApp
6  {
7      public class RequestLoggingMiddleware
8      {
9          private readonly RequestDelegate _next;
10
11          public RequestLoggingMiddleware(RequestDelegate next)
```

The code implements a middleware class named RequestLoggingMiddleware that takes a RequestDelegate as a parameter in its constructor. It also includes using statements for Microsoft.AspNetCore.Http, System.Threading.Tasks, and System.Diagnostics.

# Quiz



**Question 1:** What is the main purpose of Dependency Injection (DI) in ASP.NET Core?

- A) To tightly couple classes and their dependencies
- B) To achieve Inversion of Control by providing dependencies externally
- C) To increase memory usage by creating multiple instances
- D) To replace Middleware components



# Quiz



**Question 2:** Which service lifetime in ASP.NET Core creates a new instance once per HTTP request?

- A) Singleton
- B) Transient
- C) Scoped
- D) Static



# Quiz



**Question 3:** In ASP.NET Core, what is the role of middleware?

- A) To execute code before and after HTTP requests
- B) To compile the application code
- C) To replace the DI container
- D) To store data permanently



# Q&A Session

**Question 1:** How does Dependency Injection improve application maintainability?



**Question 2:** What are the differences between Transient, Scoped, and Singleton service lifetimes?

# Q&A Session

**Question 3:** How does middleware in ASP.NET Core work in the request pipeline?



**Question 4:** What are the basic steps to create custom middleware in ASP.NET Core?

# Module 13: Building Microservices with ASP.NET Core

- Microservices Architecture Overview
- API Communication Between Microservices
- Service Discovery and Data Consistency



# Microservices Architecture Overview

- ✓ Microservices architecture is an approach to software design where an application is composed of small, independent services that communicate over well-defined APIs.
- ✓ This style enables greater scalability, flexibility, and faster deployment cycles compared to monolithic architectures. The following are the key characteristics and benefits of microservices architecture:

1

Decentralised and  
Independent Services

3

Scalability and Resilience

5

Improved Fault Isolation

2

Technology Diversity

4

Continuous Deployment

6

Organisational Alignment

# Microservices Architecture Overview

## 1. Decentralised and Independent Services:

- Each microservice focuses on a specific business capability and can be developed, deployed, and scaled independently, reducing dependencies and increasing agility.

## 2. Technology Diversity:

- Teams can choose different technologies, frameworks, or databases best suited for each microservice without impacting the entire system, enabling optimal performance and innovation.

## 3. Scalability and Resilience:

- Microservices can be scaled individually based on demand, improving resource utilisation. Failures in one service typically do not bring down the entire application, enhancing system reliability.

# Microservices Architecture Overview

## 4. Continuous Deployment:

- The modular nature facilitates faster and more frequent deployments, allowing teams to deliver new features and fixes rapidly with minimal risk.

## 5. Improved Fault Isolation:

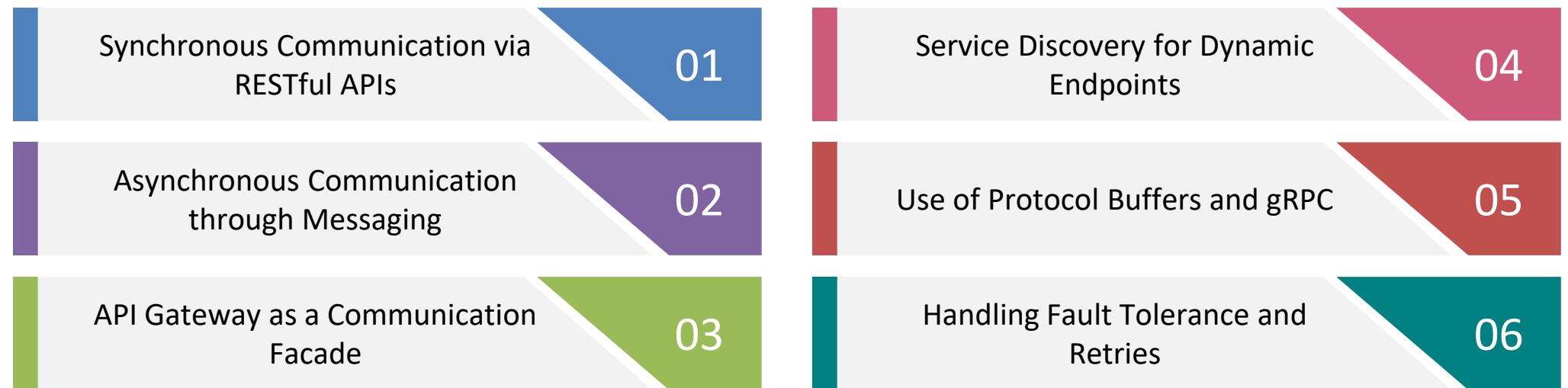
- Issues in one microservice are isolated from others, simplifying troubleshooting and reducing system-wide outages.

## 6. Organisational Alignment:

- Microservices enable teams to own complete service lifecycles, fostering cross-functional collaboration aligned with business domains and accelerating development velocity.

# API Communication Between Microservices

- ✓ API communication is the backbone of microservices architecture, enabling independent services to interact and exchange data seamlessly.
- ✓ Effective communication strategies ensure scalability, reliability, and maintainability in distributed systems. The following are the key aspects of API communication between microservices:



# API Communication Between Microservices

## 1. Synchronous Communication via RESTful APIs:

- Microservices often communicate synchronously using HTTP-based REST APIs, allowing real-time request-response interactions.

## 2. Asynchronous Communication through Messaging:

- Using message brokers like RabbitMQ or Azure Service Bus enables asynchronous communication, improving system resilience and decoupling. Services send messages or events, which consumers process independently, allowing better scalability and fault tolerance.

## 3. API Gateway as a Communication Facade:

- An API Gateway manages and routes external client requests to the appropriate microservices, handling concerns like authentication, rate limiting, and load balancing.

# API Communication Between Microservices

## 4. Service Discovery for Dynamic Endpoints:

- Microservices register themselves with a service registry, enabling other services to discover their endpoints dynamically. This facilitates load balancing, failover, and dynamic scaling without hardcoded URLs.

## 5. Use of Protocol Buffers and gRPC:

- For performance-critical communication, microservices can use gRPC and Protocol Buffers, which offer efficient binary serialisation and support for bi-directional streaming, reducing latency compared to traditional JSON-based REST APIs.

## 6. Handling Fault Tolerance and Retries:

- Communication between microservices must incorporate fault tolerance patterns like retries, circuit breakers, and timeouts to handle transient failures gracefully, ensuring robustness in distributed environments.

# Service Discovery and Data Consistency

- ✓ In microservices architecture, Service Discovery enables services to dynamically locate each other without hard-coded network locations, ensuring seamless communication.
- ✓ Data Consistency addresses the challenge of maintaining reliable and synchronised data across distributed services, critical for system correctness and user experience. The following are the key aspects of Service Discovery and Data Consistency:



# Service Discovery and Data Consistency

## 1. Dynamic Location Resolution:

- Service Discovery allows microservices to find the network location of other services at runtime, supporting scalability and fault tolerance without manual configuration.

## 2. Load Balancing and Fault Tolerance:

- By tracking healthy service instances, service discovery enables load balancing across multiple instances and reroutes requests when failures occur, improving resilience.

## 3. Consistency Models:

- Data consistency can be achieved through various models such as strong consistency, eventual consistency, or causal consistency, depending on application requirements and performance trade-offs.

# Service Discovery and Data Consistency

## 4. Data Replication and Synchronisation:

- Ensuring consistent data across distributed services often involves replication mechanisms and synchronisation protocols to handle updates and prevent conflicts.

## 5. Challenges of Distributed Transactions:

- Maintaining ACID properties across microservices is complex; thus, patterns like Saga and event-driven architectures are used to manage data consistency in distributed environments.

## 6. Impact on System Design:

- Both service discovery and data consistency influence microservices design decisions, affecting scalability, availability, latency, and complexity.

# Service Discovery and Data Consistency

(Continued)

- ✓ The following are the key differences between Service Discovery and Data Consistency:

Aspect	Service Discovery	Data Consistency
Purpose	Enables services to dynamically locate each other	Ensures data remains accurate and synchronised across services
Focus Area	Network and communication layer	Data storage and transactional integrity
Mechanism	Uses registries (e.g., Consul, Eureka) or DNS-based solutions	Uses consistency models, replication, and transactional patterns
Challenges Addressed	Handling dynamic scaling, failures, and instance availability	Handling concurrent updates, distributed transactions, and eventual data convergence

# Quiz



**Question 1:** What is a key advantage of microservices architecture?

- A) Single large application deployment
- B) Tight coupling of services
- C) Independent scalability and deployment of services
- D) Using only one technology stack



# Quiz



**Question 2:** Which communication method allows microservices to interact asynchronously?

- A) RESTful HTTP requests
- B) Messaging with brokers like RabbitMQ or Azure Service Bus
- C) Direct database access
- D) Monolithic function calls



# Quiz



**Question 3:** What is the role of service discovery in microservices?

- A) Hardcoding service endpoints
- B) Dynamically locating service endpoints for scalability and fault tolerance
- C) Encrypting API requests
- D) Logging user activity



# Q&A Session

**Question 1:** Why is technology diversity beneficial in a microservices architecture?



**Question 2:** How does an API Gateway assist in microservices communication?

# Q&A Session

**Question 3:** What challenges are associated with data consistency in microservices, and how are they addressed?



**Question 4:** How does fault tolerance improve reliability in microservices communication?



## The World's Largest Global Training Provider

✉ theknowledgeacademy.com

🌐 info@theknowledgeacademy.com

 /The.Knowledge.Academy.Ltd

 /TKA\_Training

 /the-knowledge-academy

 /TheKnowledgeAcademy

