BUILDING PRODUCTION-READY ASP.NET CORE WEB APIS

A PRACTICAL GUIDE FROM FUNDAMENTALS TO ADVANCED TECHNIQUES

- SECURE
- SCALABLE
- RELIABLE
- TESTED
- DEPLOYABLE

IAN DAVIDSON

Building Production-Ready ASP.NET Core Web APIs

A Practical Guide from Fundamentals to Advanced Techniques

Steve Publications

This book is available at
https://leanpub.com/buildingproduction-readyaspnetcorewebapis

This version was published on 2026-08-08

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

A Practical Guide from Fundamentals to Advanced Techniques

This book teaches you how to design, build, secure, and deploy production-grade REST APIs using modern ASP.NET Core. Through a realistic running example project that evolves chapter by chapter, you will learn every essential skill: request handling, middleware pipelines, dependency injection, JWT authentication, rate limiting, Entity Framework Core integration, caching, resilience patterns, testing, and containerized deployment. Every concept is explained with complete, runnable C# code, clear diagrams in text form, API request examples, and practical guidance drawn from real-world experience building APIs that handle millions of requests.

Introduction

A few years ago, a team at a mid-sized SaaS company launched their new billing API. It worked perfectly in staging. Within three days of production deployment, they were getting paged at 2 AM: the API was timing out under load, returning inconsistent errors, leaking memory on file uploads, and exposing internal exception details to anyone who knew how to read a stack trace.

The problem was not bad code. The developers were skilled. They had built functional endpoints that passed their unit tests. What they lacked was a systematic understanding of what makes an API production-ready.

This book exists to close that gap.

What This Book Is About

Building a Web API is straightforward. Building one that survives in production is harder. A production-ready API must be:

- Reliable: it handles errors gracefully, recovers from failures, and does not crash under unexpected load.
- Secure: it authenticates users properly, authorizes access correctly, protects against common attacks, and never leaks sensitive information.
- Performant: it responds quickly, uses resources efficiently, and scales when demand increases.
- Maintainable: its code is organized logically, documented clearly, and tested thoroughly so future changes are safe and predictable.

This book covers all four dimensions through the lens of ASP.NET Core, Microsoft's modern cross-platform web framework. We focus exclusively on building RESTful Web APIs. There are no tangents into Blazor, SignalR, or MVC views unless they directly relate to API development.

The target version throughout this book is ASP.NET Core 8, which is a Long Term Support release with three years of support and updates. Where relevant, I note differences in .NET 9 and later versions so the material remains useful as you upgrade.

How This Book Is Structured

The chapters follow a deliberate progression from fundamentals to advanced topics. Early chapters establish the foundation: how ASP.NET Core processes requests, how middleware works, how routing connects URLs to handlers, and how dependency injection keeps code organized. Middle chapters cover critical production concerns: authentication, authorization, security hardening, error handling, database integration with Entity Framework Core, and caching strategies. Later chapters address advanced patterns: resilience when calling external services, API versioning, health checks, testing strategies, and deployment to containerized environments.

Each chapter builds on previous ones. A running example project - a Task Management API - evolves throughout the book. You will see how authentication is added, how database entities are introduced, how caching improves performance, and how security layers protect the endpoints. This continuity helps you understand how individual pieces fit into a cohesive system rather than treating each topic in isolation.

What You Should Know Before Reading

You need:

- Basic familiarity with C#: classes, interfaces, async/await, LINQ queries, and dependency injection concepts.
- Comfort using the command line to run dotnet CLI commands.
- A general understanding of HTTP: what requests and responses are, common status codes and JSON as a data format.

You do not need prior experience with ASP.NET Core. Chapter 1 starts from scratch with project creation and walks you through every step explicitly.

How to Use the Code Examples

Every code example in this book is designed to be complete and runnable. When I show a file's contents, that is the full file - copy it directly into your

project. Commands are shown exactly as they should be typed. API requests use curl syntax so you can test endpoints immediately.

Example responses show what you will see when calling those endpoints. If something looks different, check the troubleshooting notes at the end of each chapter before assuming you made a mistake - most issues trace back to configuration details or version differences.

The Running Example Project

Throughout the book we build a Task Management API that supports:

- Creating, reading, updating, and deleting tasks
- User authentication via JWT tokens
- Role-based access control (administrators can manage all tasks; regular users manage only their own)
- Database persistence with Entity Framework Core
- Pagination, filtering, and sorting of task lists
- File attachments for tasks
- Integration with an external notification service

This is not a trivial todo list. It is intentionally complex enough to surface real-world problems while remaining focused enough to follow without getting lost in domain logic. By the end of the book, you will have a complete, production-ready API built from the ground up using modern ASP.NET Core practices.

What This Book Does Not Cover

To keep the scope manageable and relevant:

- No exercises or quizzes at the end of chapters. The practical examples serve as hands-on practice.
- Minimal coverage of .NET topics unrelated to Web APIs (such as Windows Forms, WPF, MAUI, or advanced C# language features).
- No deep dive into microservice architecture patterns beyond what is needed for API development.
- No exhaustive DevOps tutorial - deployment chapters focus on getting your API running in production environments rather than teaching Kubernetes or Terraform from scratch.

A Note on Best Practices

The term “best practice” often masks opinions dressed up as rules. I use it sparingly and always explain the reasoning behind recommendations. When there are legitimate trade-offs, I lay them out honestly so you can choose what fits your context. Some decisions depend on team size, deployment environment, regulatory requirements, or performance constraints - there is rarely a single correct answer.

What matters most is consistency. A mediocre pattern applied consistently across an entire codebase is easier to work with than a collection of clever solutions that contradict each other.

Let us get started.

Chapter 1: Getting Started with ASP.NET Core Web API

Every production API starts with the same few steps: creating a project, understanding its structure, and running your first endpoint. This chapter walks through those fundamentals so you have a solid foundation for everything that follows.

Prerequisites and Development Setup

Before creating your first project, ensure you have the required tools installed.

Required software:

- .NET 8 SDK (or later). Download from https://dotnet.microsoft.com/download/dotnet
- A code editor or IDE: Visual Studio 2022 (free Community edition), Visual Studio Code with the C# Dev Kit extension, or Rider by JetBrains
- Git for version control (optional but recommended)

Verify your installation:

Open a terminal and run:

```
dotnet --version
```

You should see an output like `8.0.xxx`. If not, install or update the .NET 8 SDK before proceeding.

Creating Your First Web API Project

The .NET CLI provides templates that generate a ready-to-run project structure. For this book, we will use the webapi template.

Create the project:

```
dotnet new webapi -n TaskManagerApi -f net8.0
cd TaskManagerApi
```

This command creates a minimal API project named `TaskManagerApi` targeting .NET 8. The `-f net8.0` flag explicitly specifies the framework version, though it is usually optional if .NET 8 is your only installed SDK.

Examine the generated structure:

```
ls -la
```

You will see:

- `TaskManagerApi.csproj`: project file defining dependencies and build settings
- `Program.cs`: application entry point with all configuration
- `appsettings.json`: default configuration values
- `appsettings.Development.json`: development-specific overrides
- `Properties/launchSettings.json`: profiles for running the app locally

Project file contents:

The generated `.csproj` file looks like this:

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

  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.OpenApi" Version="8.0.x" />
    <PackageReference Include="Swashbuckle.AspNetCore" Version="6.5.x" />
  </ItemGroup>

</Project>
```

Key points:

- `TargetFramework` specifies .NET 8.

- `Nullable enable` enables nullable reference type warnings, helping catch null-related bugs.
- `ImplicitUsings enable` automatically imports common namespaces so you do not need explicit using statements for types like `System.Console` or `Microsoft.AspNetCore.Builder`.
- The OpenAPI and Swagger packages are included by default in the webapi template starting with .NET 8.

Anatomy of the Program File

ASP.NET Core 6+ uses a simplified hosting model where all configuration lives in `Program.cs`. There is no separate Startup class unless you choose to add one for organizational purposes.

Here is the generated Program.cs:

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

// Add services to the container.
builder.Services.AddControllers();

// Learn more about configuring OpenAPI at https://aka.ms/aspnet/openapi
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

Let us break this down line by line.

Step 1: Create the builder

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

`WebApplication.CreateBuilder` initializes a `WebApplicationBuilder` that reads configuration from multiple sources in order of precedence:

1. appsettings.json
2. appsettings.{Environment}.json (e.g., appsettings.Development.json)
3. Environment variables
4. Command-line arguments

Higher-precedence sources override lower ones. This means you can keep defaults in JSON files and override sensitive values like connection strings via environment variables in production without changing code.

Step 2: Register services

```
builder.Services.AddControllers();
```

This registers all the services needed to support controller-based endpoints. Under the hood it adds:

- Model binding services (to extract data from requests)
- JSON serialization services (to format responses)
- Validation services (to check model state)
- Action filter infrastructure
- Routing services for controllers

We will explore dependency injection in depth in Chapter 5. For now, understand that `builder.Services` is a collection where you register everything your application needs.

Step 3: Add OpenAPI/Swagger support

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

These two calls enable API documentation:

- `AddEndpointsApiExplorer` discovers all endpoints (both controllers and minimal APIs) and makes metadata available.
- `AddSwaggerGen` generates an OpenAPI document describing your API's structure - endpoints, parameters, request bodies, response types.

Step 4: Build the application

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

This creates a `WebApplication` instance with all registered services and configuration. After this point you configure the middleware pipeline that processes HTTP requests.

Step 5: Configure the middleware pipeline

The middleware pipeline determines how requests are handled. Each middleware component can inspect the request, modify it, generate a response, or pass control to the next component.

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

In development only, Swagger middleware is enabled:

- `UseSwagger` serves the OpenAPI document as JSON at `/swagger/v1/swagger.json`
- `UseSwaggerUI` provides an interactive HTML interface at `/swagger` where you can browse and test endpoints

We will customize Swagger extensively in Chapter 15.

```
app.UseHttpsRedirection();
```

Redirects HTTP requests to HTTPS. This is essential for production security but disabled during local development by default through launch settings.

```
app.UseAuthorization();
```

Checks authorization policies on each request. If an endpoint requires authentication or specific permissions and the user does not qualify, the pipeline short-circuits with a 401 or 403 response. We cover this in Chapter 9.

```
app.MapControllers();
```

Maps incoming requests to controller action methods based on routing rules. Without this line, controllers would never receive requests.

Step 6: Run the application

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

Starts the Kestrel web server and begins listening for HTTP requests. This is a blocking call that runs until the application shuts down.

Understanding appsettings.json and Configuration Basics

Configuration in ASP.NET Core is hierarchical and composable. The default `appsettings.json` looks like this:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

Key sections:

- `Logging.LogLevel`: controls which log messages are emitted. Levels from least to most severe: Trace, Debug, Information, Warning, Error, Critical. Setting Microsoft.AspNetCore to Warning reduces framework noise in development.

- `AllowedHosts`: restricts which host headers the application accepts. An asterisk means any host. In production you would set this to specific domains like `api.yourcompany.com`.

Development overrides:

The `appsettings.Development.json` file typically contains:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Debug",
      "Microsoft.AspNetCore": "Information"
    }
  }
}
```

This increases logging verbosity during development without affecting production settings. These files are excluded from source control by default via `.gitignore`.

Reading configuration:

You access configuration through `IConfiguration`, which is registered automatically and available for dependency injection:

```
var appSettings = builder.Configuration["Logging:LogLevel:Default"];
Console.WriteLine(appSettings); // Outputs: Information
```

The colon syntax navigates nested JSON properties. We will explore strongly-typed configuration using the Options pattern in Chapter 5.

Environment variables override JSON:

ASP.NET Core supports a special naming convention where environment variables override configuration keys. For example, to override `Logging:LogLevel:Default`, set an environment variable named `Logging__LogLevel__Default` (double underscores represent nested properties). This is how production deployments inject secrets without modifying files.

Running and Testing Your First Endpoint

The generated template includes a sample endpoint. Let us run it and verify everything works.

Run the application:

```
dotnet run
```

You will see output like:

```
Building...
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:7001
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5001
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
```

The application listens on both HTTPS and HTTP ports. The exact port numbers may vary if those ports are already in use - ASP.NET Core automatically picks the next available port.

Test the Swagger UI:

Open your browser and navigate to `https://localhost:7001/swagger`. You will see an interactive documentation page showing the WeatherForecast endpoint included in the template.

Click “Try it out” on the GET /weatherforecast endpoint, then click “Execute”. You should see a response like:

```
[
  {
    "date": "2024-01-15T08:00:00",
    "temperatureC": -2,
    "temperatureF": 28,
    "summary": "Freezing"
  },
  ...
]
```

Test with curl:

Open a second terminal and run:

```
curl https://localhost:7001/weatherforecast --insecure
```

The `--insecure` flag bypasses HTTPS certificate validation for the self-signed development certificate. In production with a valid certificate, you would not need this flag.

Stop the application:

Press Ctrl+C in the terminal where dotnet run is executing.

Creating Your First Real Endpoint

Let us replace the template's WeatherForecast endpoint with something more relevant to our Task Manager API. We will create a minimal in-memory endpoint first and gradually evolve it into a full database-backed API over subsequent chapters.

Create a Tasks controller:

First, add support for controllers explicitly by recreating the project with the `--use-controllers` flag, or manually add a Controllers folder and create a controller class. For clarity, let us recreate:

```
cd ..
dotnet new webapi --use-controllers -n TaskManagerApi -f net8.0
cd TaskManagerApi
```

This generates the same project but with a `Controllers/WeatherForecastController.cs` instead of minimal API endpoints in Program.cs.

Create the Tasks controller:

Create a file at `Controllers/TasksController.cs`:

```
using Microsoft.AspNetCore.Mvc;

namespace TaskManagerApi.Controllers;

[ApiController]
[Route("api/[controller]")]
public class TasksController : ControllerBase
{
    private static readonly List<TaskItem> _tasks = new()
    {
        new TaskItem { Id = 1, Title = "Set up project", Description =
        ↪  "Initialize repository and configure build", IsCompleted = true },
        new TaskItem { Id = 2, Title = "Design API", Description = "Define
        ↪  endpoints and data models", IsCompleted = false }
    };

    private static int _nextId = 3;

    [HttpGet]
    public ActionResult<IEnumerable<TaskItem>> GetAll()
    {
        return Ok(_tasks);
    }

    [HttpGet("{id:int}")]
    public ActionResult<TaskItem> GetById(int id)
    {
        var task = _tasks.FirstOrDefault(t => t.Id == id);
        if (task == null)
            return NotFound();

        return Ok(task);
    }

    [HttpPost]
    public ActionResult<TaskItem> Create([FromBody] CreateTaskRequest request)
    {
        var task = new TaskItem
        {
            Id = _nextId++,
            Title = request.Title,
            Description = request.Description,
            IsCompleted = false
```

```
        };

        _tasks.Add(task);
        return CreatedAtAction(nameof(GetById), new { id = task.Id }, task);
    }

    [HttpPut("{id:int}")]
    public IActionResult Update(int id, [FromBody] UpdateTaskRequest request)
    {
        var task = _tasks.FirstOrDefault(t => t.Id == id);
        if (task == null)
            return NotFound();

        task.Title = request.Title;
        task.Description = request.Description ?? task.Description;
        task.IsCompleted = request.IsCompleted ?? task.IsCompleted;

        return NoContent();
    }

    [HttpDelete("{id:int}")]
    public IActionResult Delete(int id)
    {
        var task = _tasks.FirstOrDefault(t => t.Id == id);
        if (task == null)
            return NotFound();

        _tasks.Remove(task);
        return NoContent();
    }
}

public record TaskItem(int Id, string Title, string Description, bool
↪  IsCompleted);
public record CreateTaskRequest(string Title, string? Description);
public record UpdateTaskRequest(string Title, string? Description, bool?
↪  IsCompleted);
```

This controller implements full CRUD (Create, Read, Update, Delete) operations:

- `GET /api/tasks`: returns all tasks
- `GET /api/tasks/{id}`: returns a single task by ID
- `POST /api/tasks`: creates a new task from JSON body
- `PUT /api/tasks/{id}`: updates an existing task
- `DELETE /api/tasks/{id}`: deletes a task

Key attributes explained:

- `[ApiController]`: enables API-specific behaviors like automatic model validation and binding source inference. We cover this in detail in Chapter 4.
- `[Route("api/[controller]")]`: sets the base route pattern. `[controller]` is replaced with the controller name without the "Controller" suffix, resulting in `api/tasks`.
- HTTP verb attributes (`[HttpGet]`, `[HttpPost]`, etc.) map methods to specific HTTP operations.

Run and test:

```
dotnet run
```

Test each endpoint:

```
# Get all tasks
curl https://localhost:7001/api/tasks --insecure

# Get task by ID
curl https://localhost:7001/api/tasks/1 --insecure

# Create a new task
curl -X POST https://localhost:7001/api/tasks \
  --insecure \
  -H "Content-Type: application/json" \
  -d '{"title":"Write tests","description":"Add unit and integration tests"}'

# Update a task
curl -X PUT https://localhost:7001/api/tasks/1 \
  --insecure \
  -H "Content-Type: application/json" \
  -d '{"title":"Set up project (done)","isCompleted":true}'

# Delete a task
curl -X DELETE https://localhost:7001/api/tasks/2 --insecure
```

Expected responses:

GET /api/tasks returns 200 OK with JSON array:

```
[
  {
    "id": 1,
    "title": "Set up project",
    "description": "Initialize repository and configure build",
    "isCompleted": true
  },
  {
    "id": 2,
    "title": "Design API",
    "description": "Define endpoints and data models",
    "isCompleted": false
  }
]
```

POST /api/tasks returns 201 Created with a Location header pointing to the new resource:

```
{
  "id": 3,
  "title": "Write tests",
  "description": "Add unit and integration tests",
  "isCompleted": false
}
```

GET /api/tasks/999 (non-existent) returns 404 Not Found with an empty body.

Common Mistakes and Troubleshooting

Port already in use:

If you see `System.IO.IOException: Failed to bind to address https://localhost:7001`, another process is using that port. ASP.NET Core will usually auto-increment, but you can specify ports explicitly:

```
dotnet run --urls "https://localhost:5000;http://localhost:5001"
```

SSL certificate warnings:

The development HTTPS certificate is self-signed and may trigger browser warnings. In Chrome or Edge, navigate to the URL anyway after clicking "Advanced". For curl, use `--insecure` during development only. To trust the dev certificate system-wide on Linux:

```
dotnet dev-certs https --trust
```

Swagger not loading:

If `/swagger` returns a blank page or error:

1. Verify that `UseSwagger()` and `UseSwaggerUI()` are called before `MapControllers()`.
2. Check that the calls are inside the `IsDevelopment()` block if you only want them in development.
3. Ensure no middleware above Swagger is short-circuiting requests (e.g., authentication middleware placed too early).

JSON serialization issues:

If your response JSON looks unexpected:

- ASP.NET Core 8 uses System.Text.Json by default, which serializes property names as camelCase for web responses.
- Null properties are omitted from responses by default in web scenarios.
- If you need different behavior, configure options via `AddJsonOptions`. We cover this in Chapter 7.

Summary

In this chapter you:

- Installed and verified the .NET 8 SDK
- Created a Web API project using dotnet new webapi
- Understood the structure of Program.cs and the middleware pipeline
- Learned how configuration works with appsettings.json and environment variables
- Built your first CRUD controller with proper HTTP methods and status codes
- Tested endpoints using curl and Swagger UI

The TasksController we built is functional but has significant limitations: data lives only in memory, there is no validation on input, errors return generic responses, and anyone can access all endpoints. The next chapters address these gaps systematically.

Chapter 2 explores the middleware pipeline in depth - how requests flow through your application and how you can insert custom processing at any stage. This understanding is essential because almost every ASP.NET Core feature (authentication, logging, CORS, rate limiting) is implemented as middleware.

Chapter 2: The Request Pipeline and Middleware

The middleware pipeline is the backbone of every ASP.NET Core application. It determines how HTTP requests are processed, transformed, authenticated, authorized, routed, and ultimately responded to. Understanding middleware is not optional - it is the key to controlling your API's behavior at a fundamental level.

How the Request Pipeline Works

A middleware pipeline is a chain of components, each responsible for a specific piece of work. When a request arrives, it passes through each component in registration order. Each component can:

- Inspect or modify the incoming request
- Generate a response directly and stop further processing (short-circuiting)
- Pass control to the next middleware in the chain
- Perform work after the downstream middleware has responded (on the way back out)

Think of it as an onion. The request travels inward through each layer, reaches the center (your endpoint handler), and then the response travels outward through the same layers in reverse order.

Visual representation:

```
Request → [Middleware 1] → [Middleware 2] → [Middleware 3] → Endpoint → Response
          ←————————————————————— on the way back out —————————————————————←
```

If Middleware 2 generates a response directly, Middleware 3 and the endpoint never execute. This short-circuiting behavior is how error handling

middleware catches exceptions, how static file middleware serves files without hitting your business logic, and how authentication middleware rejects unauthenticated requests before they reach protected endpoints.

The technical mechanism:

Under the hood, ASP.NET Core builds the pipeline using a pattern called reverse folding. When you call `app.Build()`, the framework takes all registered middleware and composes them into a single `RequestDelegate`. The last middleware wraps the endpoint handler, each previous middleware wraps the one after it:

```
pipeline = MW1(context => MW2(context => MW3(context => endpointHandler)))
```

This composition is highly optimized - ASP.NET Core generates the pipeline at build time using source generation, resulting in minimal overhead compared to dynamic invocation patterns used in earlier frameworks.

Built-in Middleware Components

ASP.NET Core ships with many middleware components. Here are the most important ones for API development:

UseExceptionHandler: Catches unhandled exceptions and returns a custom error response instead of a raw stack trace. Must be placed first in the pipeline to wrap everything else. We cover this extensively in Chapter 8.

```
app.UseExceptionHandler("/error");
```

UseDeveloperExceptionPage: Shows detailed exception information including stack traces, request data, and environment details. Use only in development - never in production, as it leaks sensitive information.

```
if (app.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}
```

UseHttpsRedirection: Redirects HTTP requests to HTTPS with a 301 status code. Essential for ensuring all traffic is encrypted.

```
app.UseHttpsRedirection();
```

UseStaticFiles: Serves static files (images, CSS, JavaScript) from the wwwroot folder. Not typically needed for pure APIs but useful if your API serves documentation or health check pages.

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

UseRouting: Examines the request and determines which endpoint should handle it based on routing rules. Must be placed before UseAuthorization and MapControllers/MapEndpoints because authorization policies are evaluated per-endpoint.

```
app.UseRouting();
```

UseAuthentication: Validates authentication tokens (JWT, cookies, etc.) and populates `HttpContext.User` with the authenticated principal. Must come after UseRouting so that authentication can be skipped for public endpoints.

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

UseAuthorization: Checks whether the authenticated user is authorized to access the matched endpoint based on policies, roles, or requirements. Returns 401 (Unauthorized) if not authenticated or 403 (Forbidden) if authenticated but lacking permissions.

```
app.UseAuthorization();
```

UseCors: Handles Cross-Origin Resource Sharing headers for browser-based clients accessing your API from a different domain. Must be placed after UseRouting and before UseAuthorization so that CORS preflight requests can succeed before authentication is enforced.

```
app.UseCors("MyPolicy");
```

UseRateLimiter: Enforces rate limiting policies to prevent abuse and protect resources. Placed early in the pipeline to reject excessive requests before they consume application resources.

```
app.UseRateLimiter();
```

MapControllers / MapEndpoints: The terminal middleware that dispatches requests to controller actions or minimal API handlers based on routing rules. Everything above this point prepares the request; this is where your business logic executes.

```
app.MapControllers();
// or for minimal APIs:
app.MapGet("/tasks", () => Results.Ok(new[] { "Task 1", "Task 2" }));
```

Ordering Matters: Middleware Sequence

Middleware order is one of the most common sources of subtle bugs in ASP.NET Core applications. Getting it wrong can cause authentication to fail, CORS errors in browsers, exceptions going unhandled, or performance issues from unnecessary processing.

The recommended production pipeline order:

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

// 1. Exception handling  - must be first to catch all downstream errors
app.UseExceptionHandler("/error");

// 2. HTTPS redirection  - redirect before any other processing
app.UseHttpsRedirection();

// 3. Static files (if needed)  - serve without hitting routing/auth pipeline
app.UseStaticFiles();

// 4. Routing  - determine which endpoint handles this request
app.UseRouting();

// 5. Rate limiting  - reject excessive requests early
app.UseRateLimiter();

// 6. CORS  - handle cross-origin headers before auth (preflight needs to
↪  succeed)
app.UseCors("DefaultPolicy");

// 7. Authentication  - validate tokens and establish identity
app.UseAuthentication();

// 8. Authorization  - check permissions for the matched endpoint
app.UseAuthorization();

// 9. Endpoints  - dispatch to controllers or minimal API handlers
app.MapControllers();

app.Run();
```

Why this order matters:

- Exception handling first: If it were placed after routing, exceptions thrown during routing would not be caught and would result in raw error pages.
- Routing before authentication: You need to know which endpoint is being requested before deciding whether to authenticate. Public endpoints (health checks, documentation) should not require authentication.
- CORS before authorization: Browser preflight requests use the OPTIONS method and do not include authentication headers. If authorization runs first, the preflight fails with 401/403 and the actual request never happens.
- Authentication before authorization: You cannot authorize a user whose identity has not been established.

Common ordering mistakes:

1. Placing UseAuthorization before UseAuthentication: Results in all requests being rejected because no identity exists to check.
2. Placing UseCors after UseAuthorization: Causes CORS preflight failures for authenticated endpoints because the OPTIONS request lacks credentials.
3. Placing UseExceptionHandler after UseRouting: Exceptions during routing are not handled, producing unprofessional error responses.
4. Placing static file middleware after authorization: Every static file request goes through authentication, which is unnecessary overhead and may block legitimate access to public assets.

Writing Custom Middleware

Built-in middleware covers most scenarios, but sometimes you need custom behavior - logging request details, adding security headers, enforcing tenant isolation, or modifying responses globally. There are two ways to create custom middleware: inline using `Run`, `Map`, and `Use` extension methods, or as a dedicated class implementing `IMiddleware`.

Inline middleware with Use:

The simplest approach uses the `Use` extension method with a lambda:

```
app.Use(async (context, next) =>
{
    // Code that runs BEFORE the next middleware
    Console.WriteLine($"Request to {context.Request.Path} at
    ↪ {DateTime.UtcNow:O}");
    var startTime = DateTime.UtcNow;

    await next();

    // Code that runs AFTER downstream middleware completes
    var duration = DateTime.UtcNow - startTime;
    Console.WriteLine($"Response {context.Response.StatusCode} in
    ↪ {duration.TotalMilliseconds}ms");
});
```

This pattern gives you access to both the request (before calling `next()`) and the response status code (after calling `next()`). You can inspect headers, modify cookies, measure timing, or perform any cross-cutting logic.

Short-circuiting middleware:

If a middleware component decides not to call `next()`, it short-circuits the pipeline:

```
app.Use((context, next) =>
{
    // Block all requests to /internal from external sources
    if (context.Request.Path.StartsWithSegments("/internal") &&
        !IsInternalRequest(context))
    {
        context.Response.StatusCode = StatusCodes.Status403Forbidden;
        return context.Response.WriteAsync("Access denied");
    }

    return next();
});

bool IsInternalRequest(HttpContext context)
{
    // Check if request originates from internal network
    var clientIp = context.Connection.RemoteIpAddress?.ToString();
    return clientIp?.StartsWith("10.") == true ||
           clientIp?.StartsWith("192.168.") == true;
}
```

Branching with Map:

The `Map` extension creates a branch in the pipeline for specific path prefixes:

```
app.Map("/admin", adminApp =>
{
    // Middleware that applies only to /admin/* paths
    adminApp.Use(async (context, next) =>
    {
        Console.WriteLine("Admin area accessed");
        await next();
    });

    adminApp.MapGet("/", () => "Admin dashboard");
});

// This middleware runs for all requests except those mapped above
app.MapGet("/", () => "Public homepage");
```

Dedicated IMiddleware class:

For complex or reusable middleware, create a dedicated class:

```
public class RequestLoggingMiddleware : IMiddleware
{
    private readonly ILogger<RequestLoggingMiddleware> _logger;

    public RequestLoggingMiddleware(ILogger<RequestLoggingMiddleware> logger)
    {
        _logger = logger;
    }

    public async Task InvokeAsync(HttpContext context, RequestDelegate next)
    {
        var requestId = Guid.NewGuid().ToString("N")[..8];
        context.Items["RequestId"] = requestId;

        _logger.LogInformation(
            "{RequestId} {Method} {Path} started",
            requestId,
            context.Request.Method,
            context.Request.Path);

        var stopwatch = System.Diagnostics.Stopwatch.StartNew();
        await next(context);
        stopwatch.Stop();

        _logger.LogInformation(
            "{RequestId} {Method} {Path} completed with {StatusCode} in
            ↪  {Elapsed}ms",
            requestId,
            context.Request.Method,
            context.Request.Path,
            context.Response.StatusCode,
            stopwatch.ElapsedMilliseconds);
    }
}
```

Register it using the generic `UseMiddleware<T>` extension:

```
builder.Services.AddScoped<RequestLoggingMiddleware>();
app.UseMiddleware<RequestLoggingMiddleware>();
```

Benefits of the IMiddleware approach:

- Constructor injection allows dependency injection (loggers, configuration, etc.)

- Scoped lifetime means a new instance per request, which is safer for stateful middleware
- Code is organized in its own file rather than bloating Program.cs
- Reusable across multiple projects

Adding response headers with custom middleware:

A common use case is adding security or tracing headers to every response:

```
public class SecurityHeadersMiddleware : IMiddleware
{
    public async Task InvokeAsync(HttpContext context, RequestDelegate next)
    {
        // Add these headers before calling next so they appear in all responses
        context.Response.OnStarting(() =>
        {
            context.Response.Headers.Append("X-Content-Type-Options",
            ↪  "nosniff");
            context.Response.Headers.Append("X-Frame-Options", "DENY");
            context.Response.Headers.Append("X-XSS-Protection", "0");
            context.Response.Headers.Append("Referrer-Policy",
            ↪  "strict-origin-when-cross-origin");

            var requestId = context.Items["RequestId"] as string;
            if (!string.IsNullOrEmpty(requestId))
            {
                context.Response.Headers.Append("X-Request-Id", requestId);
            }

            return Task.CompletedTask;
        });

        await next(context);
    }
}
```

The `OnStarting` callback ensures headers are added after the status code is set but before the response body begins writing. This timing matters because some headers cannot be modified once the response has started.

Error Handling Middleware and Short-Circuiting

Error handling is a middleware concern because exceptions can occur anywhere in the pipeline - during authentication, routing, authorization, or end-

point execution. A centralized error handler ensures consistent error responses regardless of where things go wrong.

UseExceptionHandler basics:

```
app.UseExceptionHandler(options =>
{
    options.Path = "/error";
    options.MapExceptionHandler(appBuilder =>
    {
        appBuilder.Run(async context =>
        {
            var feature = context.Features.Get<IExceptionHandlerFeature>();
            var exception = feature?.Error;

            context.Response.StatusCode =
            ↪  StatusCodes.Status500InternalServerError;
            context.Response.ContentType = "application/json";

            var errorResponse = new
            {
                error = "An unexpected error occurred",
                message = context.HostingEnvironment.IsDevelopment()
                    ? exception?.Message
                    : null
            };

            await context.Response.WriteAsJsonAsync(errorResponse);
        });
    });
});
```

This configuration:

- Routes all unhandled exceptions to `/error`
- Returns a generic error message in production
- Includes the actual exception message only in development for debugging
- Uses `WriteAsJsonAsync` for clean JSON serialization

The IExceptionHandler interface (.NET 8+):

ASP.NET Core 8 introduced `IExceptionHandler`, a cleaner alternative to UseExceptionHandler that supports multiple handlers and better integration with ProblemDetails:

```
builder.Services.AddExceptionHandler<GlobalExceptionHandler>();
builder.Services.AddProblemDetails();
app.UseExceptionHandler();
```

Where GlobalExceptionHandler is:

```
public class GlobalExceptionHandler : IExceptionHandler
{
    private readonly ILogger<GlobalExceptionHandler> _logger;

    public GlobalExceptionHandler(ILogger<GlobalExceptionHandler> logger)
    {
        _logger = logger;
    }

    public async ValueTask<bool> TryHandleAsync(
        HttpContext httpContext,
        Exception exception,
        CancellationToken cancellationToken)
    {
        _logger.LogError(exception, "Unhandled exception occurred");

        var problemDetails = new ProblemDetails
        {
            Status = StatusCodes.Status500InternalServerError,
            Title = "Server Error",
            Type = "https://tools.ietf.org/html/rfc7231#section-6.6.1",
            Instance = httpContext.Request.Path
        };

        if (httpContext.HostingEnvironment.IsDevelopment())
        {
            problemDetails.Detail = exception.ToString();
        }

        httpContext.Response.StatusCode =
        ↪  StatusCodes.Status500InternalServerError;
        await httpContext.Response.WriteAsJsonAsync(problemDetails,
        ↪  cancellationToken);

        return true; // Indicates the exception was handled
    }
}
```

Returning `true` tells ASP.NET Core that the exception has been handled and no further handlers should run. Returning `false` passes it to the next registered handler or the default behavior.

We will expand significantly on error handling in Chapter 8, including RFC 7807 ProblemDetails compliance and custom exception types.

Short-circuiting for performance:

Middleware that can determine early whether a request should proceed is valuable for performance. Examples:

- Health check middleware responds immediately without hitting the database
- Static file middleware serves cached assets without routing overhead
- Rate limiting middleware rejects excess requests before they consume resources

Design your custom middleware with short-circuiting in mind when it makes sense. Avoid expensive operations on every request unless necessary.

Summary

In this chapter you:

- Understood how the middleware pipeline processes requests as a chain of components
- Learned the purpose and proper placement of key built-in middleware components
- Recognized why middleware order matters and identified common ordering mistakes
- Created custom middleware using inline lambdas, Map branching and IMiddleware classes
- Implemented error handling middleware with UseExceptionHandler and IExceptionHandler

The middleware pipeline is where infrastructure concerns live - logging, security headers, rate limiting, authentication checks. By mastering it, you gain precise control over how every request flows through your application.

Chapter 3 builds on this foundation by exploring routing in detail - how URLs are mapped to handlers, route parameters and constraints, RESTful HTTP methods, and choosing the right status codes for different scenarios.

Chapter 3: Routing, HTTP Methods, and Status Codes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Conventional vs Attribute Routing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Route Templates and Parameters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

RESTful HTTP Methods (GET, POST, PUT, PATCH, DELETE)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Choosing the Right Status Code

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Response Formats and Content Negotiation Basics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Chapter 4: Controllers and Minimal APIs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Controller-Based Endpoints

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Minimal API Endpoints

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Comparing Controllers vs Minimal APIs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Mixing Both Approaches in One Project

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Organizing Large Numbers of Endpoints

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Chapter 5: Dependency Injection and Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

The Service Container and Lifetimes (Transient, Scoped, Singleton)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Registering Services and Resolving Dependencies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Configuration Sources and Hierarchy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Strongly-Typed Configuration with Options Pattern

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Common DI Mistakes and How to Avoid Them

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Chapter 6: Model Binding, Validation, and DTOs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Model Binding from Request Data

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Data Annotations and Built-in Validation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Custom Validation with FluentValidation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Designing Effective DTOs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Mapping Between Entities and DTOs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Chapter 7: JSON Serialization and Response Formatting

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

System.Text.Json Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Naming Policies and Custom Converters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Handling Null Values and Reference Loops

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Versioning Your JSON Schema

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Performance Considerations for Serialization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Chapter 8: Exception Handling and Logging

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Global Exception Handling Strategy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Creating Standardized Error Responses

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

ProblemDetails for RFC 7807 Compliance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Implementing IExceptionHandler (.NET 8+)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Structured Logging with ILogger

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Log Levels, Enrichment, and Production Logging

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Chapter 9: Authentication and Authorization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Authentication vs Authorization Fundamentals

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Implementing JWT Bearer Token Authentication

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Creating and Validating Tokens

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Claims-Based Identity

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Role-Based Authorization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Policy-Based Authorization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Refresh Tokens and Token Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Chapter 10: API Security Essentials

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Configuring CORS Correctly

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Enforcing HTTPS in Production

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Rate Limiting and Throttling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Protecting Against Common Attacks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Secrets Management and Sensitive Data

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Chapter 11: Database Integration with Entity Framework Core

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Setting Up Entity Framework Core

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Designing Your Data Model

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Migrations and Database Schema Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

CRUD Operations with DbContext

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Query Optimization and N+1 Problems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Repository Pattern Considerations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Chapter 12: Advanced API Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Pagination Strategies (Offset, Cursor-Based)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Filtering and Sorting Request Parameters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

File Uploads and Downloads

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Streaming Large Responses

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Batch Operations and Bulk Updates

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Chapter 13: Caching and Performance Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

In-Memory Caching with IMemoryCache

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Distributed Caching with Redis

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Response Caching Middleware

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

HTTP Caching Headers (ETag, Cache-Control)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Database Query Performance Tips

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Chapter 14: Asynchronous Programming and HttpClient

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Async/Await Best Practices in ASP.NET Core

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Avoiding Deadlocks and Common Pitfalls

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Using IHttpClientFactory for External Calls

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Resilience Patterns (Retries, Circuit Breakers)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Timeout and Cancellation Handling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Chapter 15: API Documentation and Versioning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Setting Up Swagger/OpenAPI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Customizing the Swagger UI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Documenting Authentication in Swagger

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

API Versioning Strategies (URL, Header, Query String)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Managing Breaking Changes Gracefully

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Chapter 16: Health Checks and Production Readiness

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Built-in Health Check Endpoints

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Custom Health Checks for Dependencies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Startup and Liveness Probes for Containers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Metrics and Monitoring Integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Graceful Shutdown and Draining

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Chapter 17: Testing ASP.NET Core Web APIs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Unit Testing Controllers and Services

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Integration Testing with WebApplicationFactory

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Testing Authentication and Authorization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Test Containers for Database Testing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Mocking External Dependencies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Chapter 18: Deployment and Production Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Publishing Your Web API

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Docker Containerization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Deploying to Azure App Service

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Environment-Specific Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

CI/CD Pipeline Considerations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Conclusion: Building APIs That Last

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Key Principles to Remember

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

A Production Readiness Checklist

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

Continuing Your Journey

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingproduction-readyaspnetcorewebapis.