



# Getting Started with Visual Studio 2022

Learning and Implementing New Features

—  
*Third Edition*

—  
Dirk Strauss

Apress®

Dirk Strauss

# **Getting Started with Visual Studio 2022**

## **Learning and Implementing New Features**

Third Edition

Apress®

Dirk Strauss  
Uitenhage, South Africa

ISBN 979-8-8688-2113-4      e-ISBN 979-8-8688-2114-1  
<https://doi.org/10.1007/979-8-8688-2114-1>

Previously Published by Apress - Springer Nature Co

© Dirk Strauss 2020, 2023, 2025

This work is subject to copyright. All rights are solely and exclusively licensed by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Apress imprint is published by the registered company APress Media, LLC, part of Springer Nature.

The registered company address is: 1 New York Plaza, New York, NY  
10004, U.S.A.



*For my mom, Sheila Margaret Strauss - forever loved and dearly missed.*

# Introduction

Visual Studio remains one of the most comprehensive and powerful integrated development environments (IDEs) available to developers today. With each new version, it brings refinements, productivity features, and deeper integration with the tools modern developers rely on. This book is designed to give you a practical, hands-on understanding of Visual Studio, whether you are new to the IDE or an experienced developer looking to sharpen your workflow.

The focus here is not on abstract theory, but on concrete skills that improve how you use Visual Studio day to day. Throughout the book, you will explore features that make you more productive, learn how to manage your projects and codebases more effectively, and gain insight into modern development practices supported by the IDE.

---

## Who the Book Is For

This book is written for developers who spend their time in Visual Studio and want to get the most out of it. Whether you are a professional building enterprise-grade applications, a student just starting out, or a hobbyist experimenting with .NET, the examples and workflows in these pages will help you improve your efficiency and confidence in using the IDE.

Some prior knowledge of C# and .NET is assumed, but the material is accessible enough that developers from other backgrounds will also benefit. The content ranges from basic setup to advanced topics like unit testing and source control integration, making it useful both as a learning resource and a desk reference.

The book is organized into five chapters, each building on the previous one:

### **Chapter 1 – Getting to Know Visual Studio 2022**

We start with the essentials, installation, workloads, orientation in the IDE, and everyday productivity features, then dive straight into AI. You'll install and enable GitHub Copilot, work with Chat and Agent modes, upload screenshots to give Copilot context, and see how Agent-driven tools automate real tasks (like running the .NET Upgrade tool end-to-end). You'll also learn how model selection works in Visual Studio (including the GPT-4.1 completions model and "bring your own model" options) and how MCP (Model Context Protocol) servers extend Agent with task-specific tools.

### **Chapter 2 – Working with Visual Studio 2022**

This chapter rounds out the day-to-day tooling: choosing the right project types, working with .NET Aspire, managing NuGet (and npm where applicable), creating reusable code snippets, navigating quickly with bookmarks/shortcuts, exploring data and services via Server Explorer, and getting comfortable with key windows and menus. It also covers the new, simpler .slnx solution format and the CLI/IDE paths for migrating and opening solutions cleanly. In Aspire, you'll see how the built-in dashboard surfaces logs, traces, and metrics, and how Copilot can sit alongside that telemetry to answer questions in context.

### **Chapter 3 – Debugging Your Code**

You'll move from basic breakpoints to conditional and labeled breakpoints, DataTips and visualizers, the Immediate and Watch windows, attaching to processes, and remote debugging. Then you'll see how Copilot becomes "debugger-aware," analyzing variables from the Autos/DataTips views and bringing exception context into a chat workflow so you can move from symptom to fix faster.

### **Chapter 4 – Unit Testing**

We cover creating and running tests (MSTest), setup/teardown patterns, data-driven tests, and Live Unit Testing in Visual Studio Enterprise. You'll measure and interpret code coverage and, importantly, use Copilot to generate tests from inline chat or the Copilot window. The chapter also shows how to wire up an Obsidian MCP server so Copilot Agent can document your tests into a knowledge base automatically.

### **Chapter 5 – Source Control**

You'll go end-to-end with Git and GitHub inside Visual Studio, including account setup, cloning, committing, branching, and pull requests, then step into advanced flows such as multi-repo work, stashes, comparing branches, checking out a specific commit, and staging by line. Copilot helps write consistent commit messages (with team-defined custom instructions), performs local code reviews before you push, and even supports quick sharing via Git permalinks from the editor.

---

## **What You Will Gain**

By the end of this book, you will not only understand the core features of Visual Studio but also be able to integrate them into your daily development practices. You will learn how to navigate the IDE more effectively, debug and test your code with confidence, and collaborate smoothly using GitHub. In short, this book equips you with both the foundational knowledge and the practical skills needed to work productively in Visual Studio.

Copilot is an accelerator, not an autopilot. You'll see where it shines (boilerplate, refactors, test scaffolding, reviews) and where you should slow down and verify.

If you are looking for a focused guide that deals exclusively with Visual Studio, its features, its workflows, and how to make the most of them, this book was written for you. Keep it within reach on your desk, and let it serve as both a learning tool and a reference as you continue your development journey.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub (<https://github.com/Apress>). For more detailed information, please visit <https://www.apress.com/gp/services/source-code>.

# Acknowledgments

I would like to express my sincere gratitude to Apress for their support and guidance in bringing this book to publication. Their commitment to technical excellence and to helping authors share knowledge with the world has been invaluable.

To the developer community, thank you for your passion, curiosity, and constant innovation. Your willingness to share ideas and solutions continues to inspire me and drives the spirit of collaboration that makes our industry thrive.

Most importantly, I wish to thank my family. To my wife, Adele, and my children, Irénée and Tristan, your love, encouragement, and patience make everything possible. This book is as much yours as it is mine.

# Table of Contents

[Chapter 1: Getting to Know Visual Studio 2022](#)

[Installing Visual Studio](#)

[Visual Studio 2022 System Requirements](#)

[Visual Studio Is 64-Bit](#)

[Full .NET 9.0 Support](#)

[Using Workloads](#)

[Exploring the IDE](#)

[The Solution Explorer](#)

[Toolbox](#)

[The Code Editor](#)

[Hot Reload](#)

[Navigating Code](#)

[Navigate Forward and Backward Commands](#)

[Navigation Bar](#)

[Find All References](#)

[Find Files Faster](#)

[Reference Highlighting](#)

[Go To Commands](#)

[Go To Definition](#)

[Peek Definition](#)

[Subword Navigation](#)

[Features and Productivity Tips](#)

[Track Active Item in Solution Explorer](#)

[Hidden Editor Context Menu](#)



[Open in File Explorer](#)

[Finding Keyboard Shortcut Mappings](#)

[Clipboard History](#)

[Go To Window](#)

[Navigate to Last Edit Location](#)

[Multi-caret Editing](#)

[Sync Namespaces to Match Your Folder Structure](#)

[Paste JSON As Classes](#)

[Enable Code Cleanup on Save](#)

[Add Missing Using on Paste](#)

[Customize File Encoding](#)

[Indent Word Wrap](#)

[The Horizontal Scrollbar Reimagined](#)

[Automatic Documentation Comment Generation](#)

[HTTP File Updates](#)

[Features in Visual Studio 2022](#)

[Visual Studio Search](#)

[Solution Filters](#)

[Visual Studio IntelliCode](#)

[Visual Studio Live Share](#)

[Copy Files Across Visual Studio Instances](#)

[Adding and Using GitHub Copilot](#)

[GitHub Settings](#)

[GPT-4.1 Code Completion Model](#)

[Code Completions](#)

[Inline Chat](#)

[Slash Commands](#)

[Copilot Agent Mode](#)

[Agent Mode Tools and Using MCP Servers](#)

[Using Images in GitHub Copilot](#)

[Custom Instructions](#)

[Upgrade with Copilot](#)

[Summary](#)

[Chapter 2: Working with Visual Studio 2022](#)

[Visual Studio Project Types](#)

[Various Project Templates](#)

[Creating a .NET Aspire Project](#)

[Prerequisites and Installation](#)

[Using Visual Studio 2022](#)

[The Solution Structure](#)

[Viewing the .NET Aspire Dashboard](#)

[Dashboard Overview](#)

[Resources Page](#)

[Viewing Logs and Telemetry](#)

[Standalone Dashboard Mode](#)

[Adding .NET Aspire to an Existing Application](#)

[Adding .NET Aspire Packages](#)

[Adding SQL Server](#)

[Installing the SQL Server Integration Packages](#)

[Add a SQL Server Resource in AppHost](#)

[A Word on NuGet Packages](#)

[Hosting Your Own NuGet Feeds](#)

[Using npm in .NET Projects](#)

[Anatomy of the package.json](#)

[Managing npm Packages](#)

[Security Considerations](#)

[Creating and Using Code Snippets](#)

[Creating Code Snippets](#)

[Using Bookmarks and Code Shortcuts](#)

[Bookmarks](#)

[Code Shortcuts](#)

[Adding Custom Tokens](#)

[The Server Explorer](#)

[Running SQL Queries](#)

[Visual Studio Windows](#)

[C# Interactive](#)

[Code Metrics Results](#)

[Send Feedback](#)

[Personalizing Visual Studio](#)

[Adjust Line Spacing](#)

[Document Management Customizations](#)

[Visual Studio Themes](#)

[The New .SLNX Solution Format](#)

[What Is an .SLNX File](#)

[Why Introduce .SLNX Files](#)

[Creating and Using .SLNX Solution Files](#)

[Final Considerations](#)

[Summary.](#)

## [Chapter 3: Debugging Your Code](#)

### [Debugging Toolbar Location](#)

### [Working with Breakpoints](#)

### [Setting a Breakpoint](#)

### [Visualizers](#)

### [Conditional Breakpoints and Actions](#)

### [Temporary Breakpoints](#)

### [Dependent Breakpoints](#)

### [Dragging Breakpoints](#)

### [Manage Breakpoints with Labels](#)

### [Exporting Breakpoints](#)

### [Viewing Inline Return Values](#)

### [Using DataTips](#)

### [Using the Watch Window](#)

### [The DebuggerDisplay Attribute](#)

### [Evaluate Functions Without Side Effects](#)

### [Format Specifiers](#)

### [Error Copying Improvements](#)

### [Debugging with GitHub Copilot](#)

### [Variable Analysis](#)

### [Debugging an Exception](#)

### [Get Conditional Breakpoint Suggestions](#)

### [Diagnostic Tools](#)

### [CPU Usage](#)

### [Memory Usage](#)

### [Immediate Window](#)

[Attaching to a Running Process](#)

[Attach to a Remote Process](#)

[Reattaching to a Process](#)

[Remote Debugging](#)

[System Requirements](#)

[Download and Install Remote Tools](#)

[Running Remote Tools](#)

[Start Remote Debugging](#)

[Summary.](#)

[Chapter 4: Unit Testing](#)

[Creating and Running Unit Tests](#)

[Choosing a Test Framework](#)

[Writing Your First Test](#)

[Running Your Tests](#)

[Create and Run a Test Playlist](#)

[Unit Test Attributes](#)

[Using Live Unit Tests](#)

[Using Copilot to Generate Unit Tests](#)

[Document Unit Tests with Obsidian MCP Server](#)

[How to Measure Code Coverage in Visual Studio](#)

[Summary.](#)

[Chapter 5: Source Control](#)

[Create a GitHub Account](#)

[Improve GitHub Productivity with Multi-account Support](#)

[Create and Clone a Repository](#)

[Cloning a Repository](#)

[Create a Branch from Your Code](#)

[Creating Pull Requests](#)

[Handling Pull Requests](#)

[Working with Stashes](#)

[Multi-repo Support](#)

[Compare Branches](#)

[Check Out Commit](#)

[Line Staging](#)

[Copy Git Permalink in Visual Studio](#)

[Commit with GitHub Copilot](#)

[Stop Issues Before Committing to Git](#)

[Summary](#)

[Index](#)

## About the Author

### **Dirk Strauss**

is a seasoned software developer specializing in C# and Visual Studio. Over the course of his career, he has contributed to a range of companies and gained valuable insights from working alongside leading professionals in the industry. He is the author of several technical books covering topics such as Visual Studio, C#, and ASP.NET Core. With a strong passion for software engineering, Dirk is dedicated to continuous learning and to sharing his expertise with the broader developer community.



# About the Technical Reviewer

## Kapil Bansal

is a PhD Scholar and Lead DevOps engineer at S&P Global Market Intelligence, India. He has more than 16 years of experience in the IT industry, having worked on Azure cloud computing (PaaS, IaaS, and SaaS), Azure Stack, DevSecOps, Kubernetes, Terraform, Office 365, SharePoint, release management, application lifecycle management (ALM), Information Technology Infrastructure Library (ITIL), and Six Sigma. He completed certification programs in Advanced Program in Strategy for Leaders from IIM Lucknow and Cyber Security and Cyber Defense from IIT Kanpur. He has worked with companies such as IBM India Pvt Ltd, HCL Technologies, NIIT Technologies, Encore Capital Group, and Xavient Software Solutions, Noida, and has served multiple clients based in the United States, the UK, and Africa, such as T-Mobile, World Bank Group, H&M, WBMI, Encore Capital, and Bharti Airtel (India and Africa). Kapil also reviewed *Hands-On Kubernetes on Azure: Run your applications securely and at scale on the most widely adopted orchestration platform* and *Azure Networking Cookbook: Practical recipes to manage network traffic in Azure, optimize performance, and secure Azure resources* and many other books published by Packt. He also reviewed *Practical Microsoft Azure IaaS: Migrating and Building Scalable and Secure Cloud Solutions* and *Beginning SharePoint Communication Sites* and many other books published by Apress.





# 1. Getting to Know Visual Studio 2022

Dirk Strauss<sup>1</sup> 

(1) Uitenhage, South Africa

---

Visual Studio is an amazing bit of software. If you have been using Visual Studio for a number of years, you will certainly agree that the IDE offers developers a host of tools and features to make them more productive. You will also be aware that it has grown a lot during the past couple of years and is an absolute powerhouse when it comes to providing tools to develop world-class software.

Initially released as Visual Studio 97 in February 1997, this was the first attempt at using a single development environment for multiple languages. The evolution of Visual Studio is detailed in Table [1-1](#).

**Table 1-1** The Evolution of Visual Studio

Release	Ver	.NET framework	.NET (Core)	Release date
Visual Studio 2022	17.0	3.5, 4.6–4.8	2.1–3.1, 5.0–9.0	Feb 15, 2022
Visual Studio 2019	16.0	3.5–4.8	2.1–3.1, 5.0	Apr 2, 2019
Visual Studio 2017	15.0	3.5–4.7	1.0, 1.1, 2.0, 2.1	Mar 7, 2017
Visual Studio 2015	14.0	2.0–4.6	1.0	July 20, 2015
Visual Studio 2013	12.0	2.0–4.5.2		Oct 17, 2013
Visual Studio 2012	11.0	2.0–4.5.2		Sept 12, 2012
Visual Studio 2010	10.0	2.0–4.0		Apr 12, 2010
Visual Studio 2008	9.0	2.0, 3.0, 3.5		Nov 19, 2007
Visual Studio 2005	8.0	2.0, 3.0		Nov 7, 2005
Visual Studio .NET 2003	7.1	1.1		Apr 24, 2003
Visual Studio .NET 2002	7.0	1.0		Feb 13, 2002

Release	Ver	.NET framework	.NET (Core)	Release date
Visual Studio 6.0	6.0	N/A		June 1998
Visual Studio 97	5.0	N/A		Feb 1997

There is so much to see and learn when it comes to Visual Studio. Therefore, in this chapter, we will start by having a look at the following:

- Installing Visual Studio
- What workloads are
- Exploring the IDE
- Existing and new features available in Visual Studio 2022
- Productivity tips

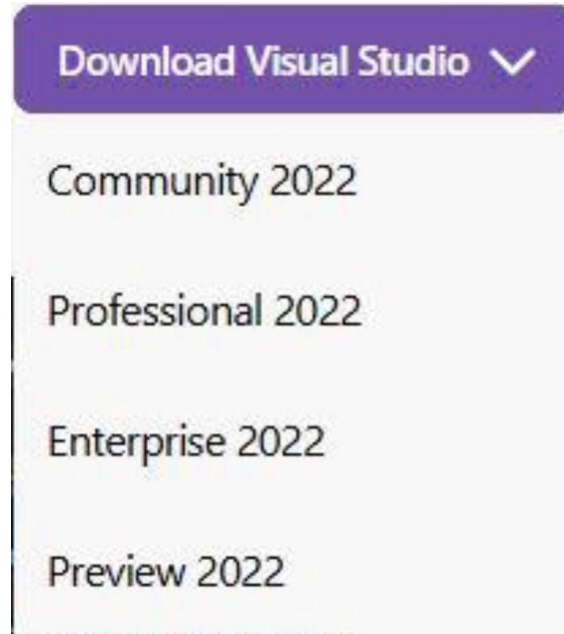
Let us see where to find the Visual Studio Installer and get going.

---

## Installing Visual Studio

At the time of this writing, Visual Studio 2022 is available for Windows machines. Unfortunately, Visual Studio for Mac has been retired as of August 31, 2024, in accordance with Microsoft's Modern Lifetime Policy. Alternatives such as the C# Dev Kit for Visual Studio Code allow developers to continue where Visual Studio for Mac left off. You can download Visual Studio 2022 for Windows from <https://visualstudio.microsoft.com/vs/>, and if you are on macOS, you will need to head on over to <https://code.visualstudio.com/docs/csharp/get-started/> to get started with C# in VS Code.

Clicking the Download Visual Studio button, you will see a list drop-down with the options as displayed in Figure [1-1](#).



**Figure 1-1** Versions of Visual Studio Available

If you would like to compare the Visual Studio 2022 editions, you can have a look at <https://visualstudio.microsoft.com/vs/compare/> for a detailed comparison. The bottom line is that if you want Visual Studio 2022 for free, download Visual Studio Community 2022.

Visual Studio Community 2022 is aimed at students, open source, and individual developers. The paid tiers include Visual Studio Professional 2022, which is aimed at small teams and Visual Studio Enterprise 2022, aimed at large development teams.

Microsoft specifies enterprise organizations as those having more than 250 PCs or more than 1 million US dollars in annual revenue.

Let us have a brief look at the recommended system requirements for installing Visual Studio on your machine. For a comprehensive list, browse to <https://docs.microsoft.com/en-us/visualstudio/releases/2022/system-requirements> and have a read through that.

## **Visual Studio 2022 System Requirements**

The system requirements for installing Visual Studio 2022 might differ from those of previous versions of Visual Studio. Refer to the documentation on <https://docs.microsoft.com> to review the system requirements for previous versions of Visual Studio.

Visual Studio Enterprise 2022, Visual Studio Professional 2022, and Visual Studio Community 2022 all support the following minimum system requirements.

## ***Operating Systems***

The following 64-bit Windows operating systems are supported:

- Windows 11 Home, Pro, Pro Education, Pro for Workstations, Enterprise, and Education
- Windows 10 Home, Professional, Education, and Enterprise
- Windows Server 2022 – Standard and Datacenter
- Windows Server 2019 – Standard and Datacenter
- Windows Server 2016 – Standard and Datacenter

## ***Hardware***

There is a line here that developers generally don't like to cross when it comes to the minimum hardware specs. Many developers I know will geek out on system RAM and favor SSDs over HDDs. Nevertheless, here are the minimum recommended requirements:

- ARM64 or x64 processor (Quad-core or better) recommended. ARM 32 processors are not supported.
- Minimum of 4 GB of RAM (16 GB of RAM recommended).
- Windows 365: Minimum 2 vCPU and 8 GB RAM. 4 vCPU and 16 GB of RAM is recommended.
- Minimum of 850 MB and up to 210 GB of available hard disk space (depending on installed features, 20–50 GB of free space is typically required).
- For improved performance, install Windows and Visual Studio on an SSD.
- Minimum display resolution of WXGA (1366 by 768) but works best at 1920 by 1080 or higher.

## ***Supported Languages***

Visual Studio and the Visual Studio Installer are available in 14 languages as follows:

- English
- Chinese (Simplified)
- Chinese (Traditional)
- Czech
- French
- German
- Italian
- Japanese
- Korean
- Polish
- Portuguese (Brazil)
- Russian
- Spanish
- Turkish

### ***Additional Notes***

There are several additional requirements to take note of that I will briefly list here. There are, however, other requirements that might be of importance to your unique development environment. For a full list, refer to the system requirements at the following link:

<https://docs.microsoft.com/en-us/visualstudio/releases/2022/system-requirements>.

- Administrator rights are required to install Visual Studio.
- .NET Framework 4.7.2 or above is required to run the Visual Studio Installer and install Visual Studio (.NET Framework 4.5.2 or above is required to install Visual Studio version 17.8 and earlier).
- Visual Studio requires .NET Framework 4.8 and will be installed during setup.

### **Visual Studio Is 64-Bit**

Visual Studio 2022 on Windows will now run as a 64-bit application, which means that you can open, modify, and debug really large solutions without running out of memory. Being a 64-bit application,

Visual Studio 2022 is no longer limited to just 4 GB of memory in the main devenv.exe process.

Even though Visual Studio 2022 is 64-bit, it doesn't change the bitness of the apps you build. You can still create and deploy 32-bit applications.

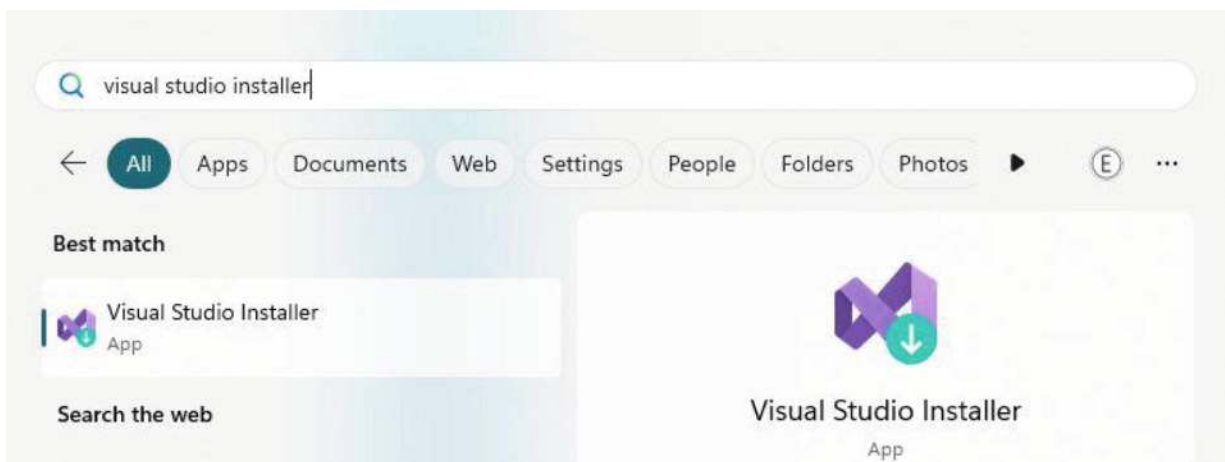
## Full .NET 9.0 Support

As of version 17.12, Visual Studio 2022 has full support for .NET 9. Visual Studio 2022 provides an excellent developer experience for working with .NET 9 projects, and new features such as AI productivity enhancements, GitHub Copilot, Git tooling, and IDE enhancements reduce developer toil significantly.

## Using Workloads

After Visual Studio has been installed, you can customize the installation by selecting feature sets, also known as workloads. Think of workloads as a collection of individual features that belong together. This allows you to easily modify Visual Studio to include only what you need.

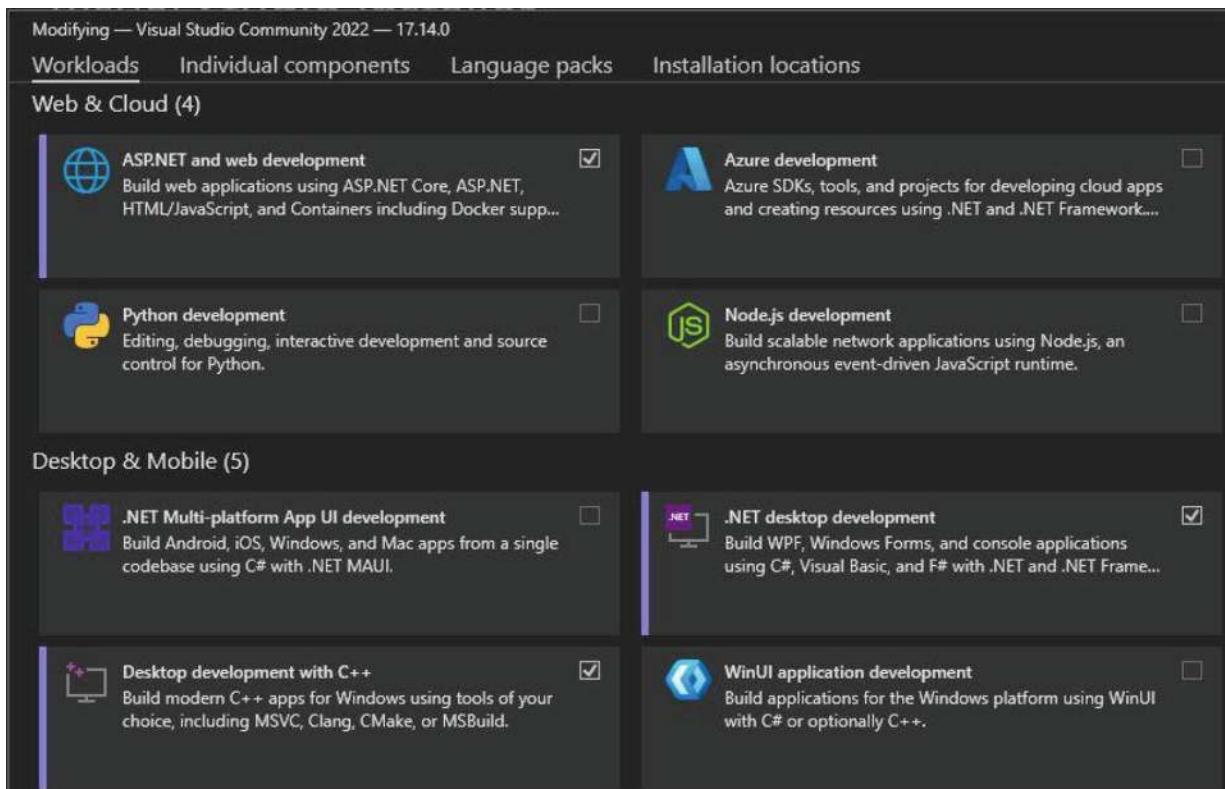
To launch the workloads screen, find the Visual Studio Installer from the Start menu (this is on Windows 11), shown in Figure [1-2](#).



**Figure 1-2** Find the Visual Studio Installer

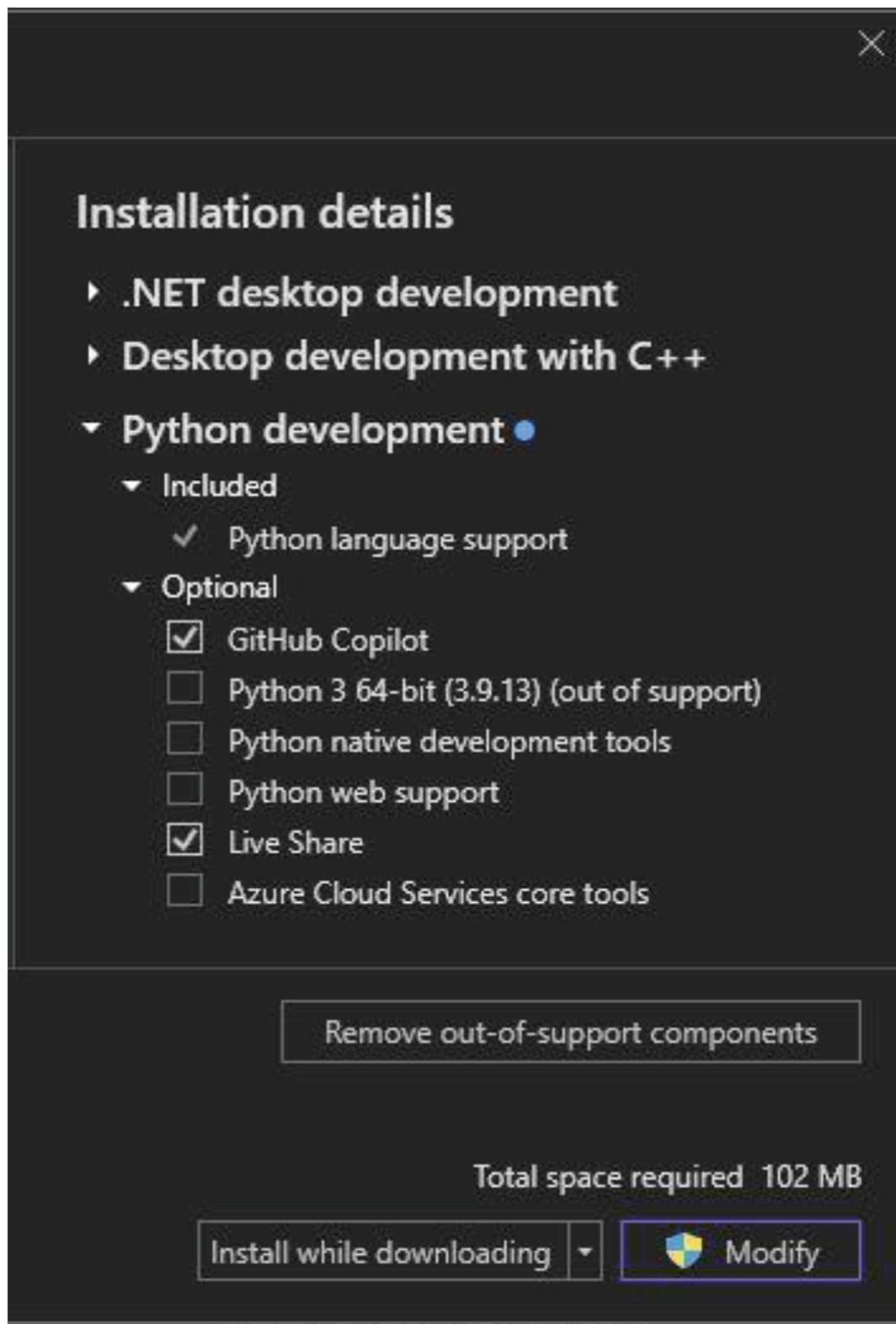
Clicking the Visual Studio Installer will launch the installer from where you can click on the Modify button next to your installed

instance of Visual Studio to modify your installation, as seen in Figure 1-3.



**Figure 1-3** Installing Additional Workloads

If you want to start doing Python development, you can check the *Python development* workload and install that. As can be seen in Figure 1-4, this will update the installation details section and show you exactly what is being installed and how much additional space you will need to install the selected workload.

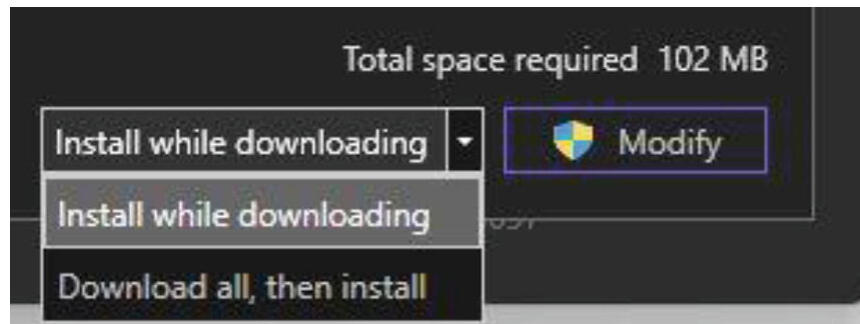


**Figure 1-4** Workload Installation Details



The workloads also contain tabs, namely, Workloads, Individual components, Language packs, and Installation locations. If you need to install an additional component such as Service Fabric Tools, you can do so by selecting the component on the Individual components tab.

When you have selected all the workloads and individual components you would like to install, you can choose to do the installation while downloading or to download everything before installing, as shown in Figure [1-5](#).



**Figure 1-5** Installation Options

This will modify your existing installation of Visual Studio 2022 and apply the changes you selected.

---

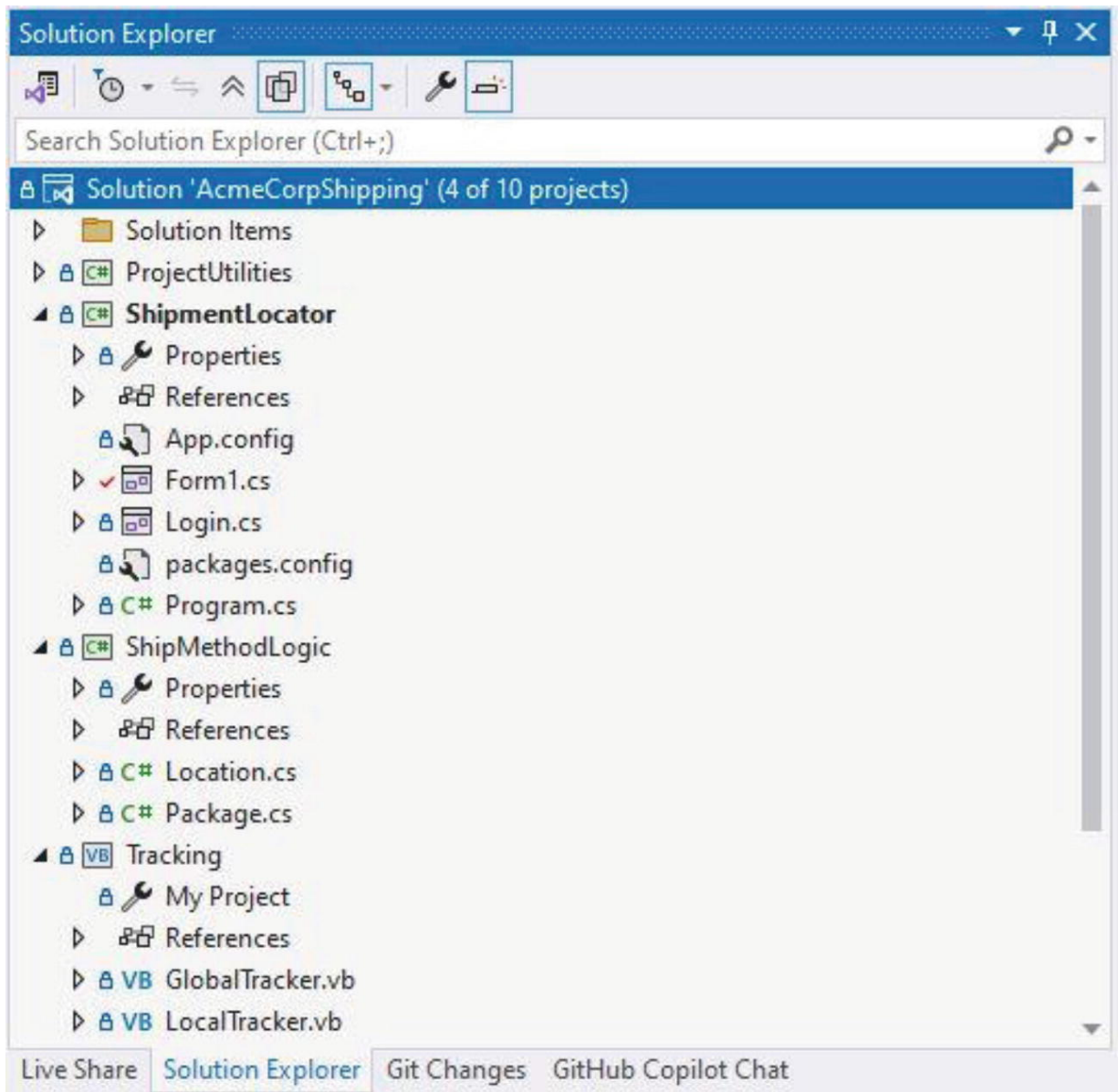
## Exploring the IDE

The Visual Studio IDE is full of features and tools that help developers do what they need to do, efficiently and productively. Developers start off creating one or more projects that contain the logic for their code. These projects are contained in what we call a solution. Let's have a look at the Solution Explorer first.

### The Solution Explorer

In Visual Studio, the notion of solutions and projects is used. A solution contains one or more projects. Each project contains code that runs the logic you need for your application to do what it does.

Consider the example of a Shipment Locator application as can be seen in Figure [1-6](#).



**Figure 1-6** The Shipment Locator Solution

It is with this solution that you will add all the required projects to create your application. From the example in Figure 1-6, we can see that the solution contains three projects. The projects are as follows:

- ShipmentLocator – WinForms application – C#
- ShipMethodLogic – Class Library – C#
- Tracking – Class Library – VB.NET

Of particular interest, you will notice that you can have a solution that contains a mix of C# projects and VB.NET projects. The various

project templates are discussed in Chapter 2 of this book, but if you would like to read more on creating a new project in Visual Studio, refer to the documentation at the following link:

<https://docs.microsoft.com/en-us/visualstudio/ide/create-new-project?view=vs-2022>.

Take note that you do not need the AcmeCorpShipping source code for this chapter. The project is illustrative and is used to explain the concepts of Visual Studio solutions and projects. If, however, you would like to view the code, it is available from the GitHub repository for this book at the following URL:

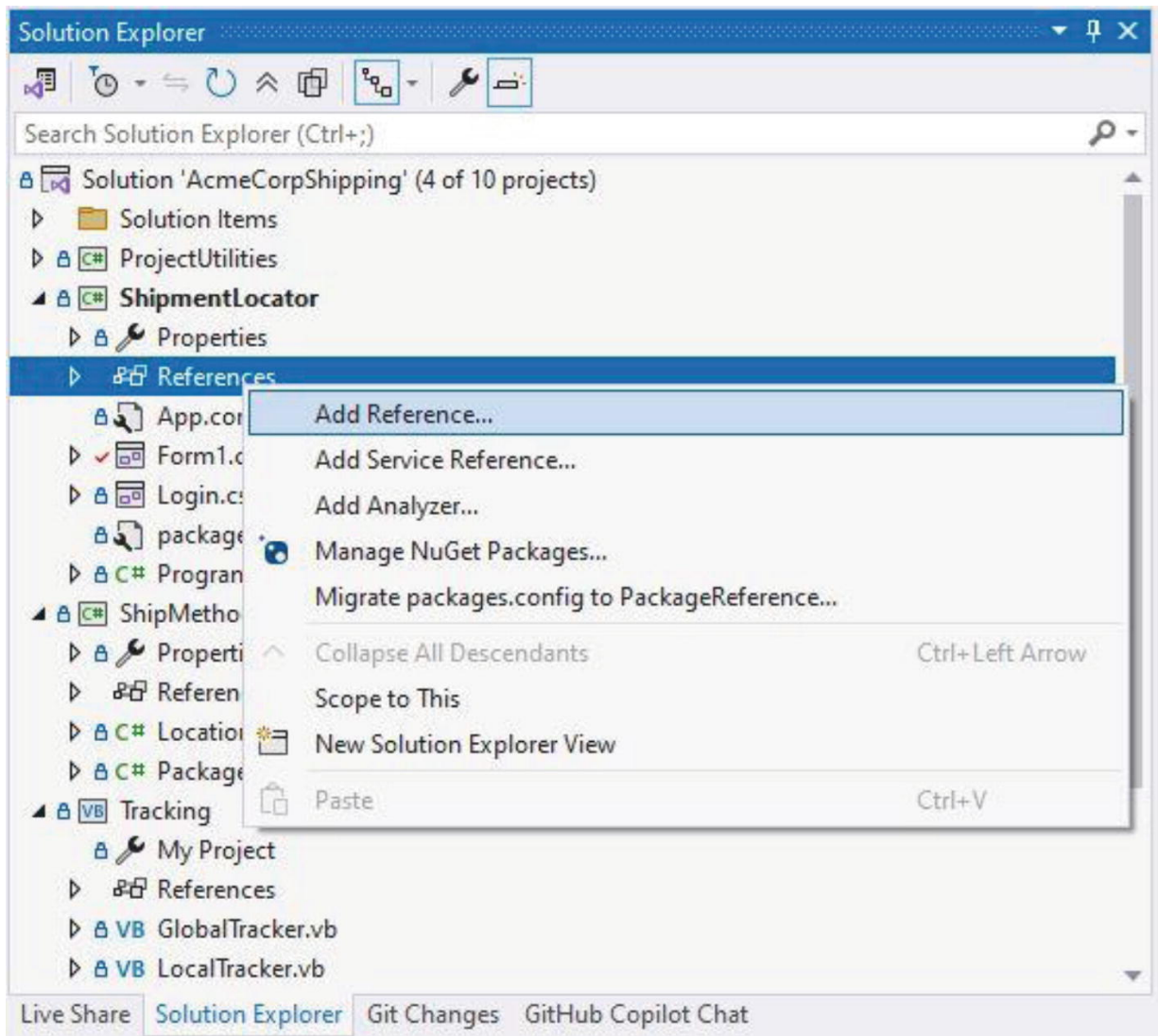
[github.com/apress/getting-started-vs2022](https://github.com/apress/getting-started-vs2022).

You are therefore not limited by a particular language and can create applications containing a mix of .NET languages.

The reason that we can mix .NET languages in the same solution is due to something we call IL (Intermediate Language). IL is used by the .NET Framework to create machine-independent code from the source code used in your projects.

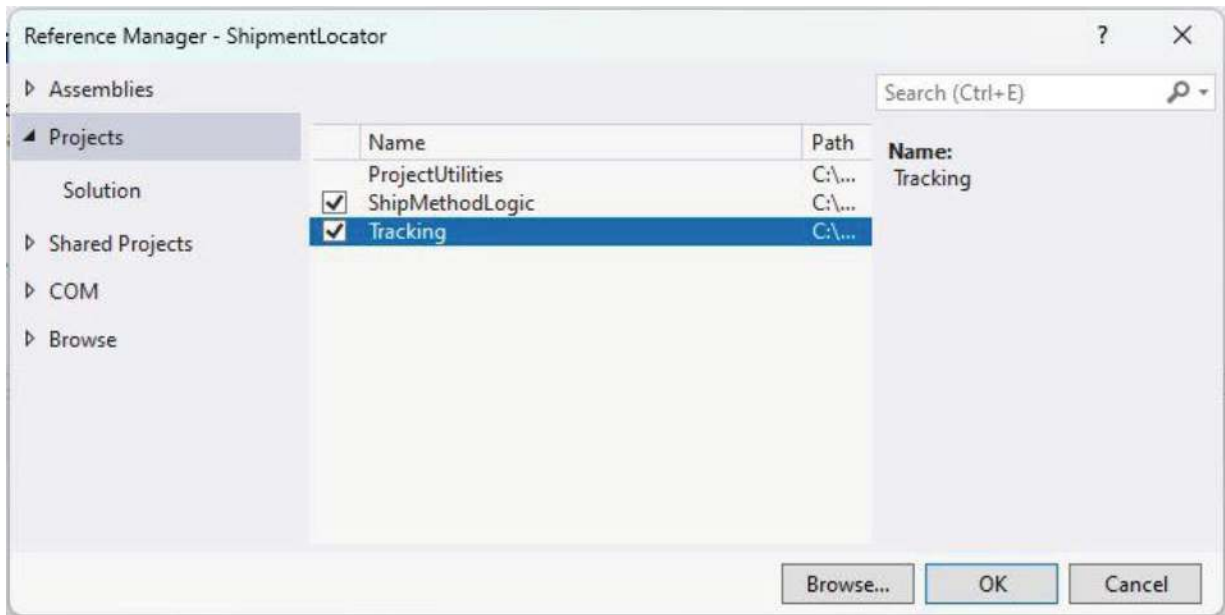
The WinForms application will contain the UI needed to track and trace shipments. For the WinForms application to be able to use the logic contained in the other two class libraries, we need to add what is called a reference to the other projects.

This is done by right-clicking the project that you want to add the reference to and selecting *Add Reference* from the context menu (Figure [1-7](#)).



**Figure 1-7** Adding a Project Reference

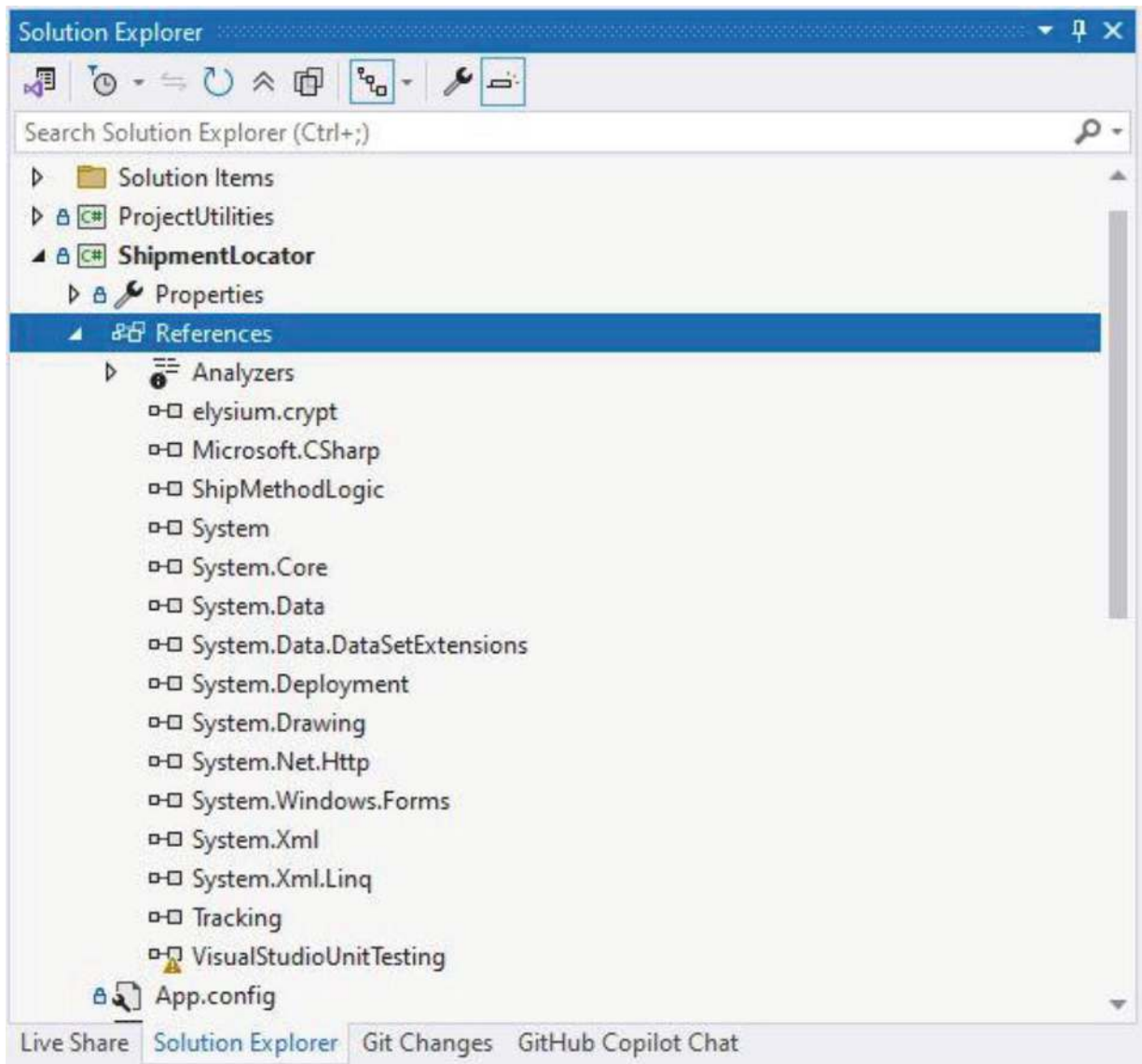
When you click the Add Reference menu, you will be presented with the Reference Manager screen shown in [Figure 1-8](#).



**Figure 1-8** The Reference Manager Screen

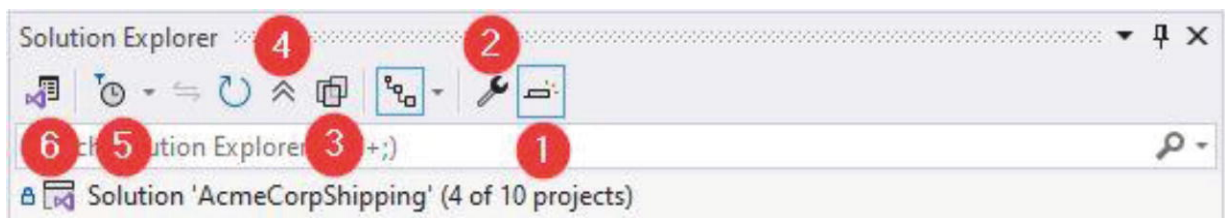
Under the Projects tab, you will find the other two Class Library projects in your solution. By checking each one and clicking the OK button, you will add a reference to the code in these projects.

If you had to expand the References section under the ShipmentLocator project, you will see that there are two references to our Class Library projects ShipMethodLogic and Tracking as can be seen in Figure [1-9](#).



**Figure 1-9** Added References

This will now make all the code you write in the ShipMethodLogic and Tracking projects available to the ShipmentLocator project. Having a look at the toolbar on the Solution Explorer (Figure [1-10](#)), you will notice that it contains several buttons.





**Figure 1-10** The Solution Explorer Toolbar

The buttons contained here are displayed as needed. The View Code button, for example, will only show up in the toolbar when a file containing code is selected in the Solution Explorer. These buttons give you quick access to the following features, as outlined in the following:

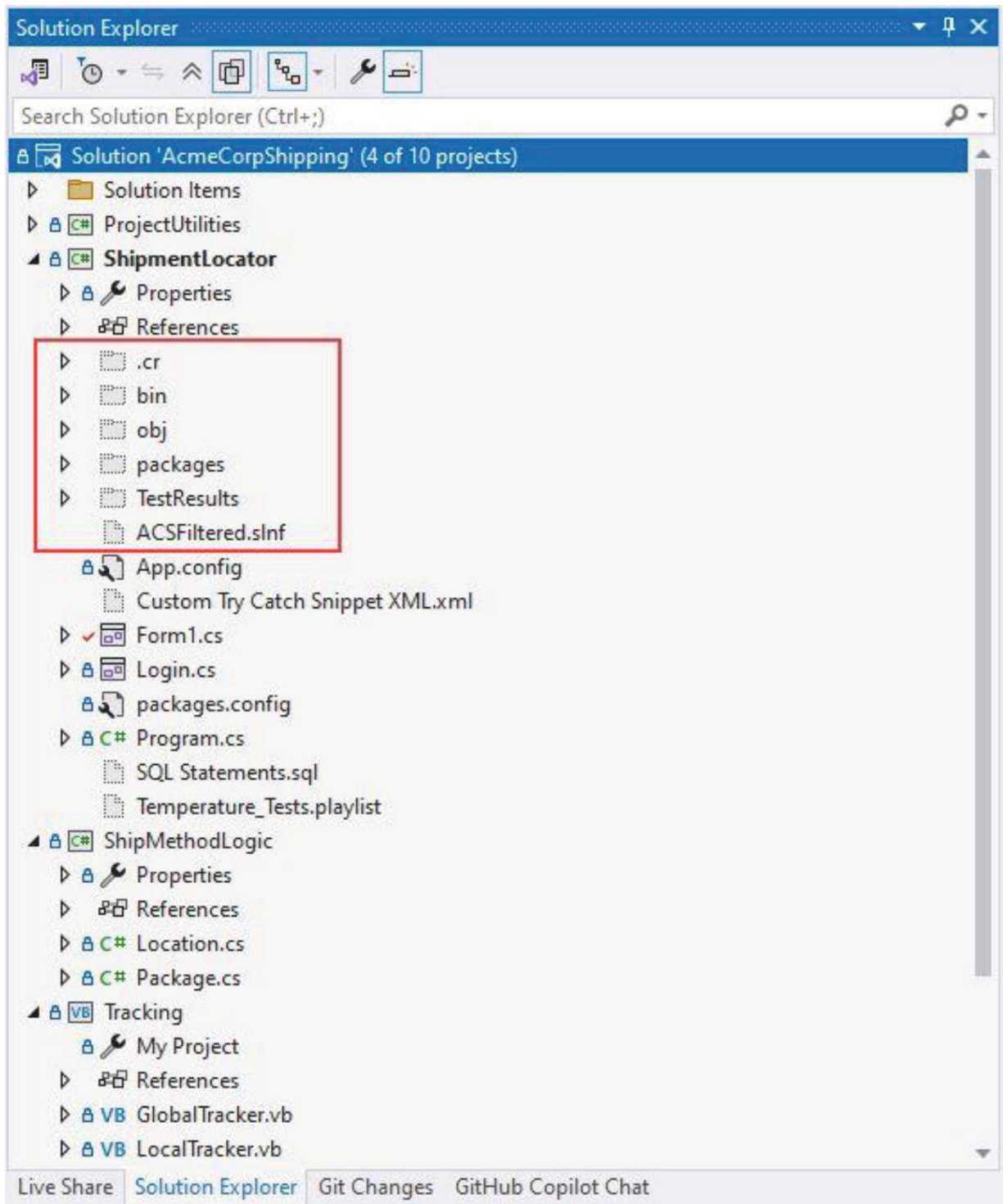
1. Preview Selected Items
2. Properties
3. Show All Files
4. Collapse All
5. Pending Changes Filter
6. Toggle between Solution and Folder views

I will not go through each one in detail, but of particular interest, you will notice that the Show All Files will display unnecessary files and folders such as the bin folder in your Solution Explorer. Go ahead and click the Show All Files button, and look at the Solution Explorer again.

By looking at Figure [1-11](#), you can see that it now displays the bin folder and the obj folder. These folders are not necessary for your code but are important to your solution.

The obj folder contains bits of files that will be combined to produce the final executable. The bin folder contains the binary files that are the executable code for the application you are writing.

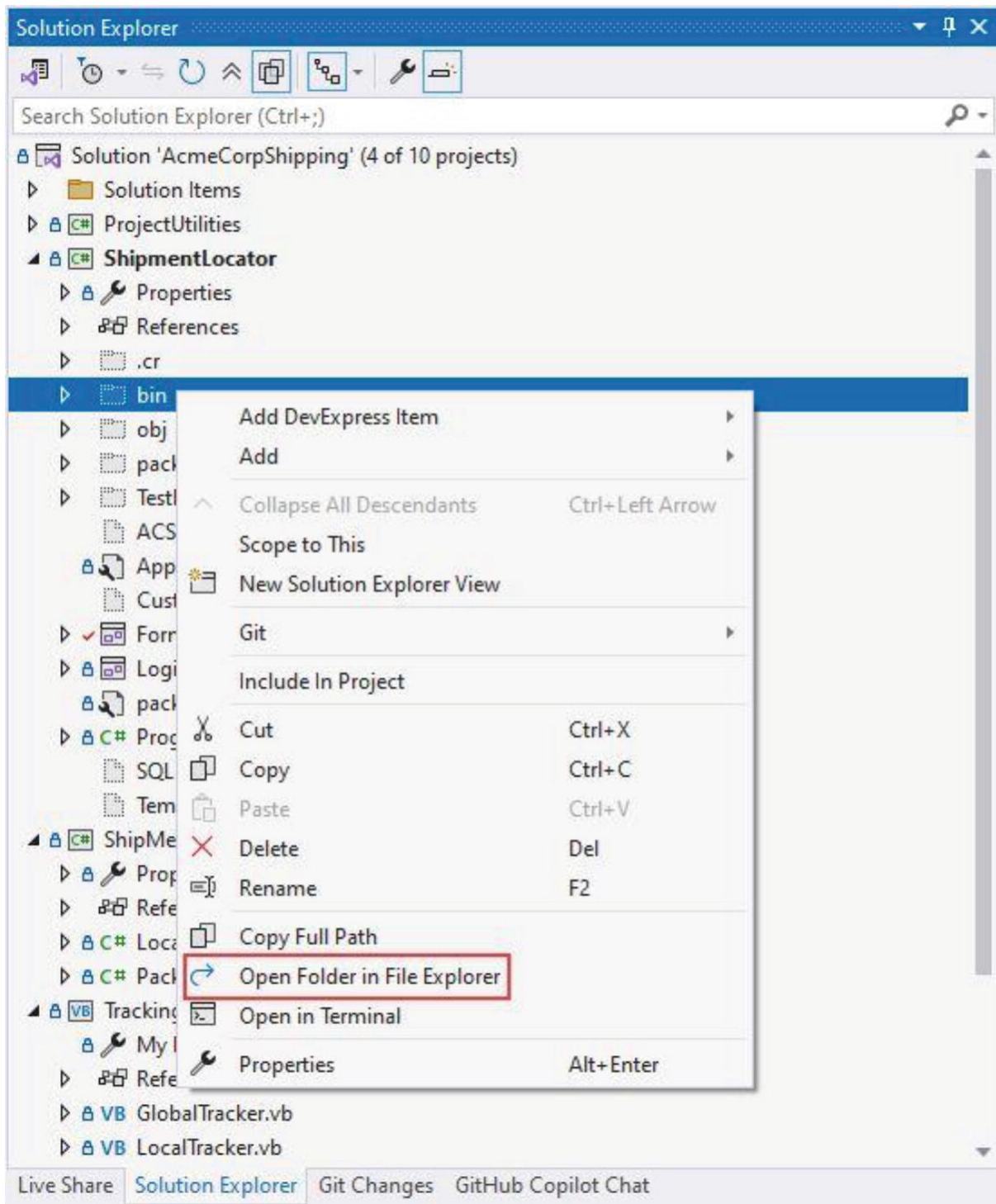
Each obj and bin folder will contain a Debug and Release folder that simply matches the currently selected build configuration of your project.



**Figure 1-11** Solution Explorer Displaying All Files

You can now right-click the bin folder, as seen in Figure [1-12](#), and click the Open Folder in File Explorer menu to quickly have a look at the contents of the folder.

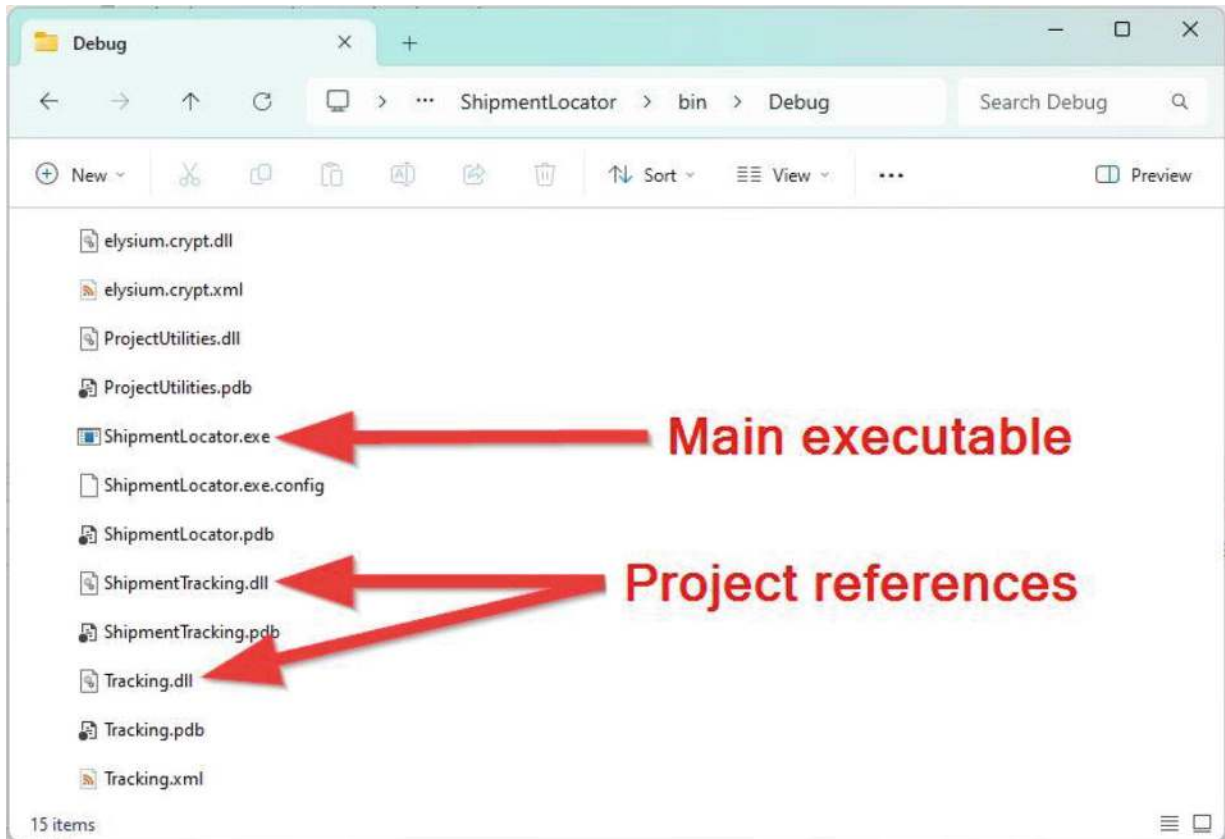




**Figure 1-12** Open Folder in File Explorer

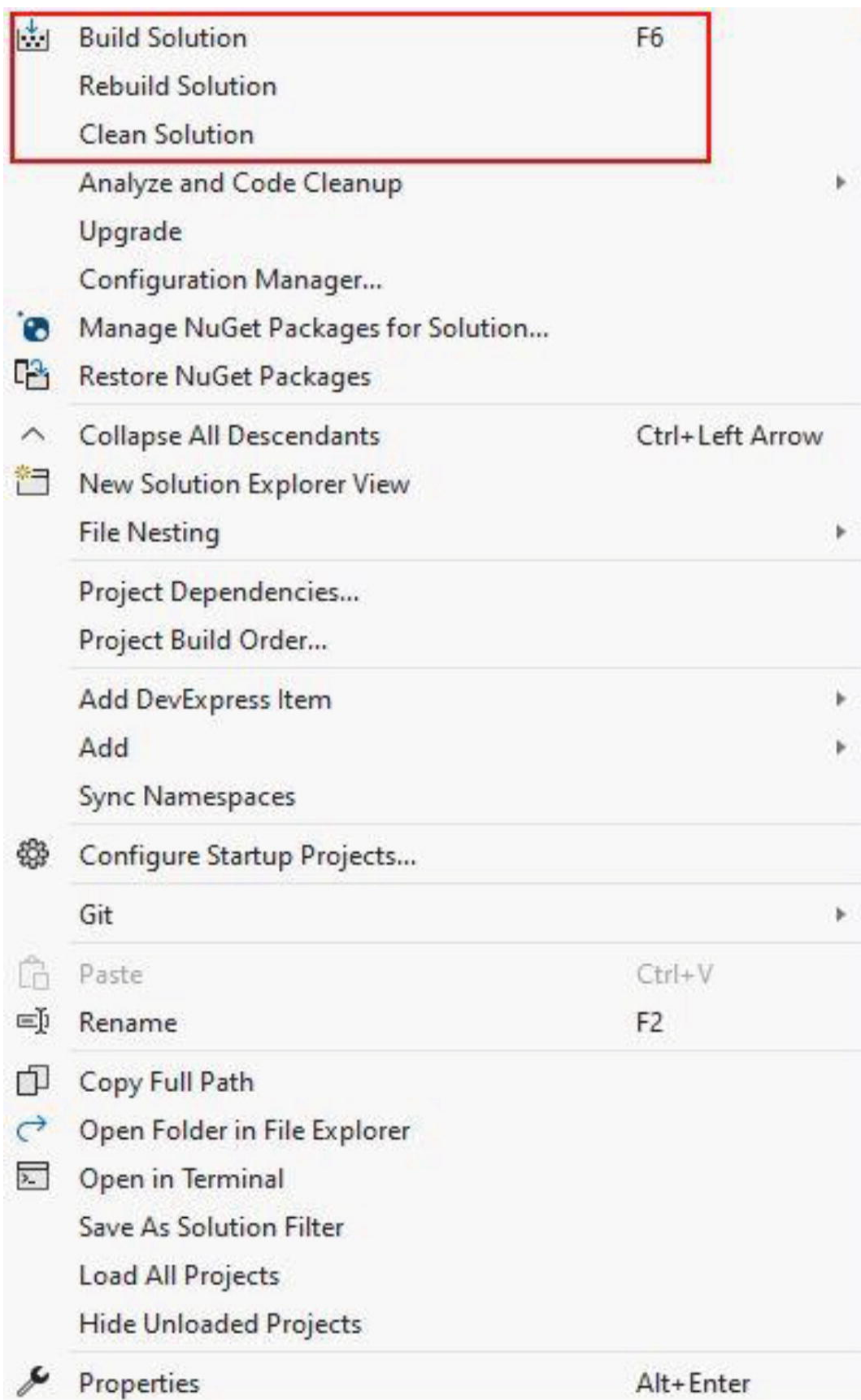
This is a nice shortcut for anyone needing to navigate to the location of the Visual Studio files in the solution.

If you open the bin folder and click the Debug folder contained in the bin folder, you will see the main exe as well as any referenced dll files in the project (Figure 1-13).



**Figure 1-13** The Contents of the Debug Folder

These files will be updated each time you build or run your project. If this folder is blank, perform a build of your solution by pressing F6 or by right-clicking the solution and clicking Build Solution from the context menu, as seen in Figure 1-14.



---

**Figure 1-14** Right-Click Solution Options

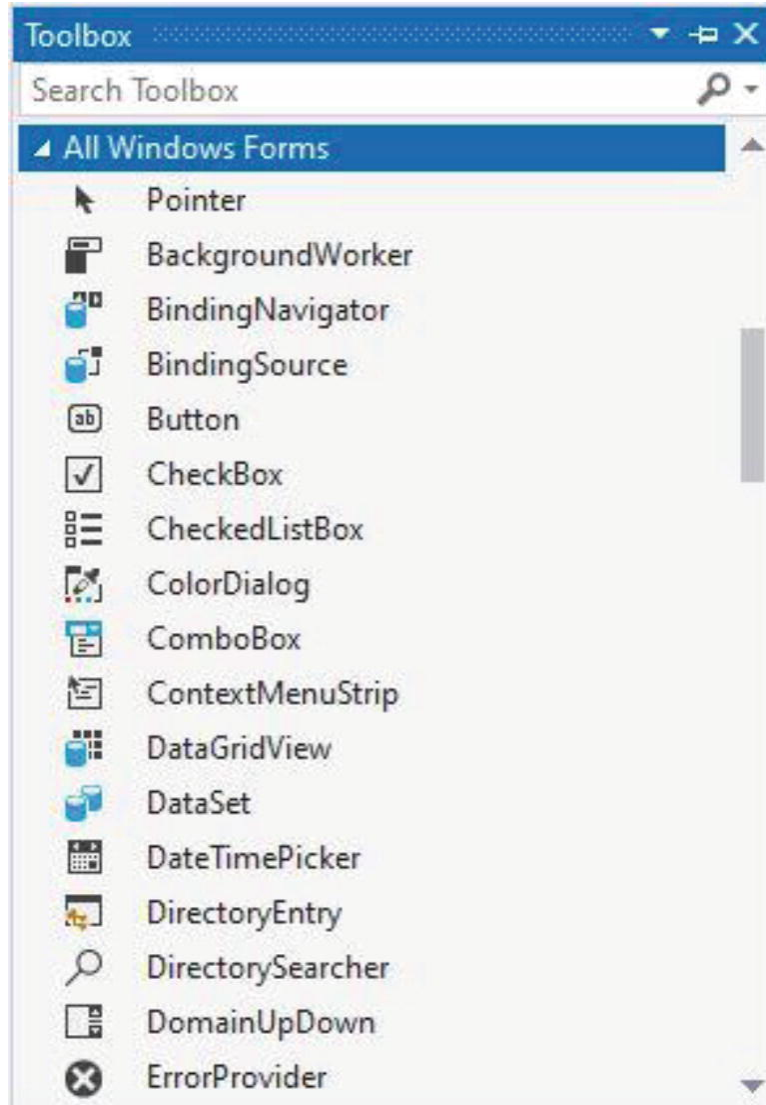
You might be wondering what the difference is between Build Solution, Rebuild Solution, and Clean Solution. The differences are as follows:

- Build Solution will do an incremental build of the solution for anything that has changed since the last build.
- Rebuild Solution will clean the solution and then rebuild it from scratch.
- Clean Solution will only clean the solution by removing any build artifacts left over by the previous builds.

If you are encountering unexpected build errors that do not seem to be errors in your code editor, try cleaning your solution and then building it again.

## **Toolbox**

When dealing with a UI file such as a web application or a WinForms application, you will notice that you have a Toolbox at your disposal (Figure [1-15](#)).

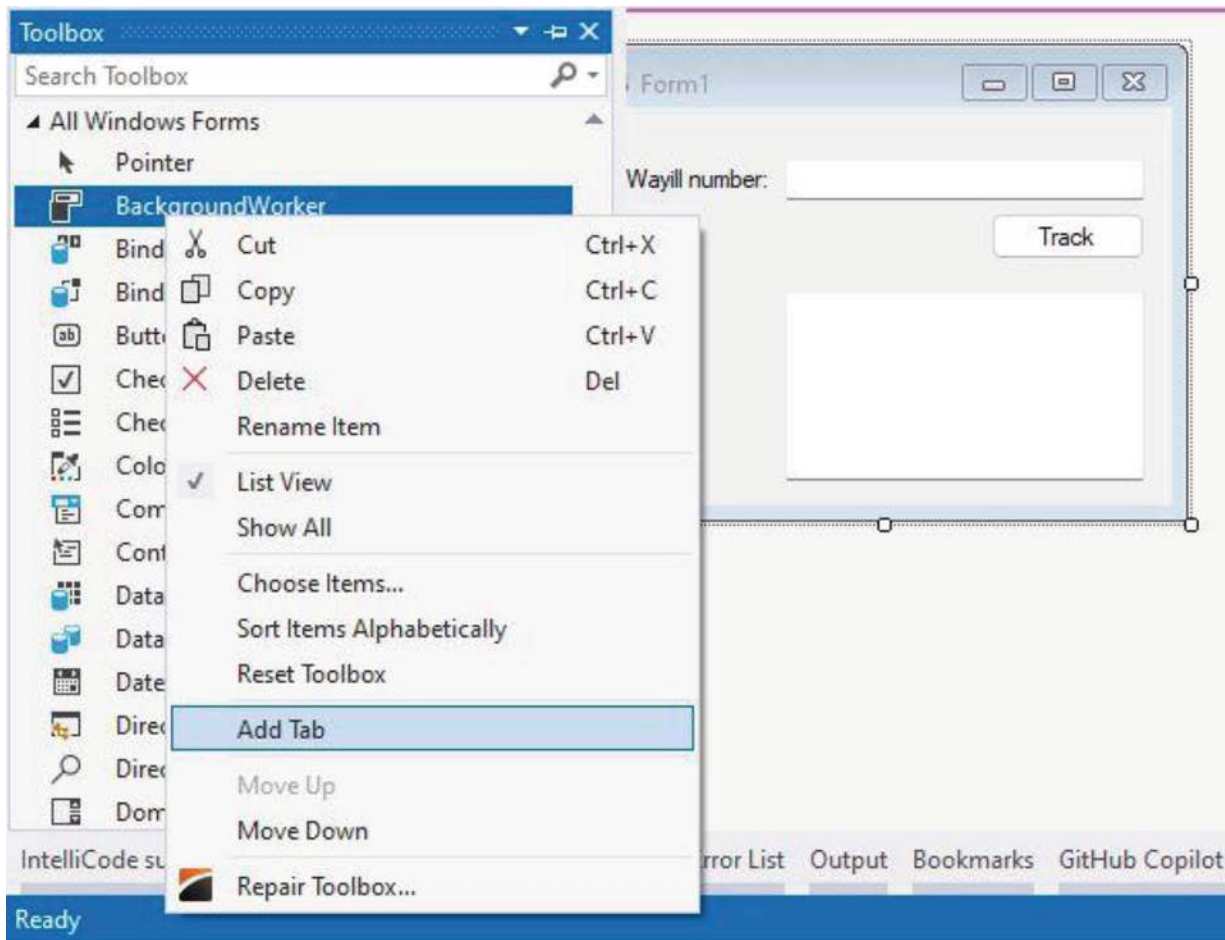


**Figure 1-15** The WinForms Toolbox in Visual Studio

The Toolbox allows you to add controls to your application such as text boxes, buttons, drop-down lists, and so on. This allows developers to design the UI of the application by dragging and dropping the relevant controls on the design surface.

You can also open the Toolbox by clicking the View menu and selecting the Toolbox menu item. It is worth noting that for some project types, you will not see any items in the Toolbox.

If you do not like the default layout of the Toolbox, you can right-click the tab or an individual item in the Toolbox and perform one of several actions from the context menu, as seen in [Figure 1-16](#).



**Figure 1-16** Toolbox Context Menu

The context menu allows you to do the following:

- Rename an item
- Choose additional items
- Remove items
- Move items up and down
- Sort items
- Add a new tab

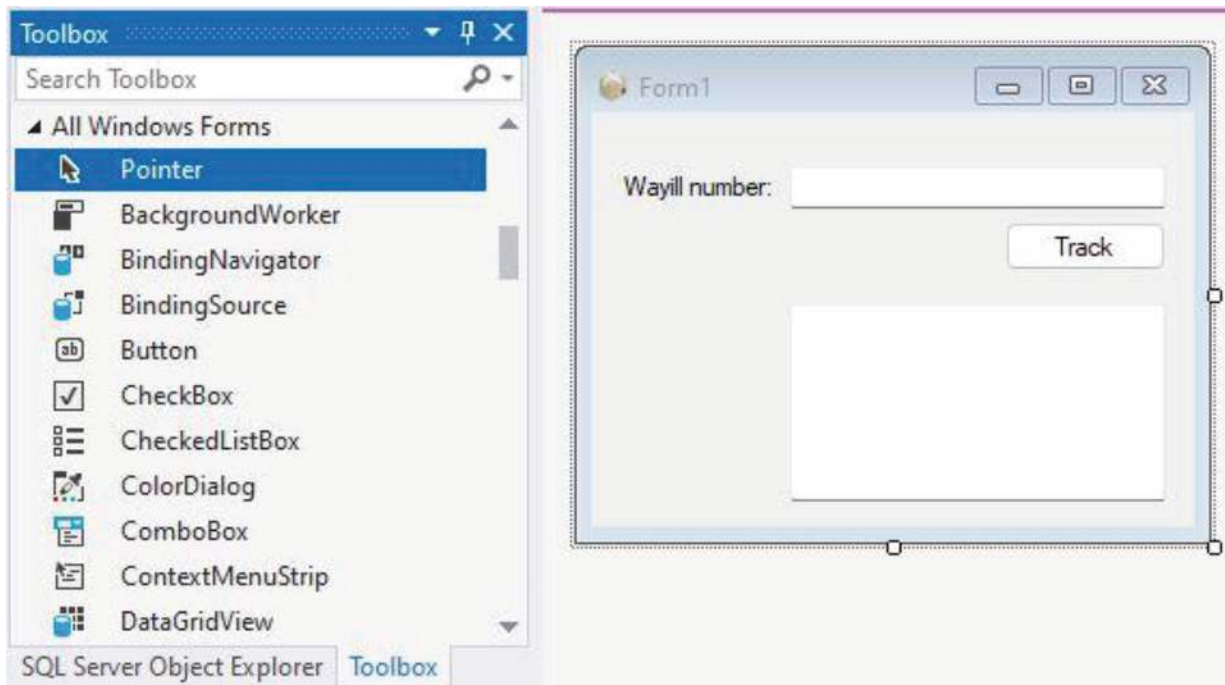
If you have third-party controls installed such as DevExpress or Telerik, you will find the controls specific to the installed components under their own tab in the Toolbox.

## The Code Editor

Let's add some basic UI components to our WinForms application (Figure [1-17](#)). To this code-behind, we will add some code to our



project, just to get the ball rolling. All that this application will do is take a given waybill number and return some location data for it.



**Figure 1-17** The Application Design

After adding the UI elements to the designer, open the code window for the main WinForms application called Form1.cs. Add the code in Listing [1-1](#) to the code-behind.

You will notice after adding the code that Visual Studio starts to underline some of the added code, as shown in Figure [1-18](#). This is because Visual Studio is making suggestions to improve the quality of your code.

```
private int WBPartB(int min, int max)
{
    var rngCrypto = new
    RNGCryptoServiceProvider();
    var bf = new byte[4];

    rngCrypto.GetBytes(bf);
    var result = BitConverter.ToInt32(bf, 0);
```

```

    return new Random(result).Next(min, max);
}

```

**Listing 1-1** The Code-Behind Form1.cs

```

1 reference
private int WbPartB(int min, int max)
{
    var rngCrypto = new RNGCryptoServiceProvider();
    var bf = new byte[4];

    rngCrypto.GetBytes(bf);
    var result = BitConverter.ToInt32(bf, 0);

    return new Random(result).Next(min, max);
}

```

**Figure 1-18** Visual Studio Code Improvement Suggestions

The underlined code is code that Visual Studio is making suggestions for improvement on.

To view the details of the suggestion, hover your mouse over one of the underlined lines of code. Visual Studio will now display the details of the suggested change as seen in Figure 1-19.



**Figure 1-19** Code Change Suggestion



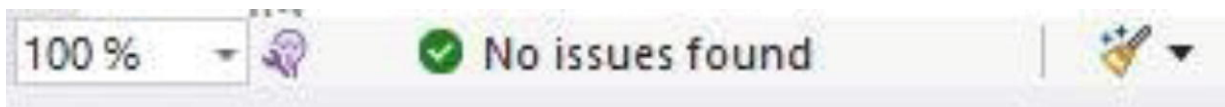
Here, we can see that Visual Studio is warning you that the use of the `RNGCryptoServiceProvider` is considered obsolete. At the bottom of the code editor, you will also see that Visual Studio displays the count of errors and warnings as seen in Figure 1-20.



**Figure 1-20** Errors and Warnings

You are able to navigate between the warnings and errors by clicking the up and down arrows. You can also perform a code cleanup by clicking the little brush icon or by holding down Ctrl+K, Ctrl+E.

After cleaning up the code and adding the code suggestions, Visual Studio displays a clean bill of health (Figure 1-21).



**Figure 1-21** Code Suggestions Applied

With these types of code improvements, you can be sure that you are writing code that is performant and generally free of basic errors.

---

## Hot Reload

Starting in Visual Studio 2022, the Hot Reload feature works for both managed .NET and native C++ apps. Hot Reload saves a developer from having to stop the debug process between edits. This means less rebuilding, restarting, and re-navigating to the specific location in the application you were debugging.

For a list of supported .NET app frameworks and scenarios, view the document at the following link:

<https://docs.microsoft.com/en-us/visualstudio/debugger/hot-reload?view=vs-2022#supported-net-app-frameworks-and-scenarios>.

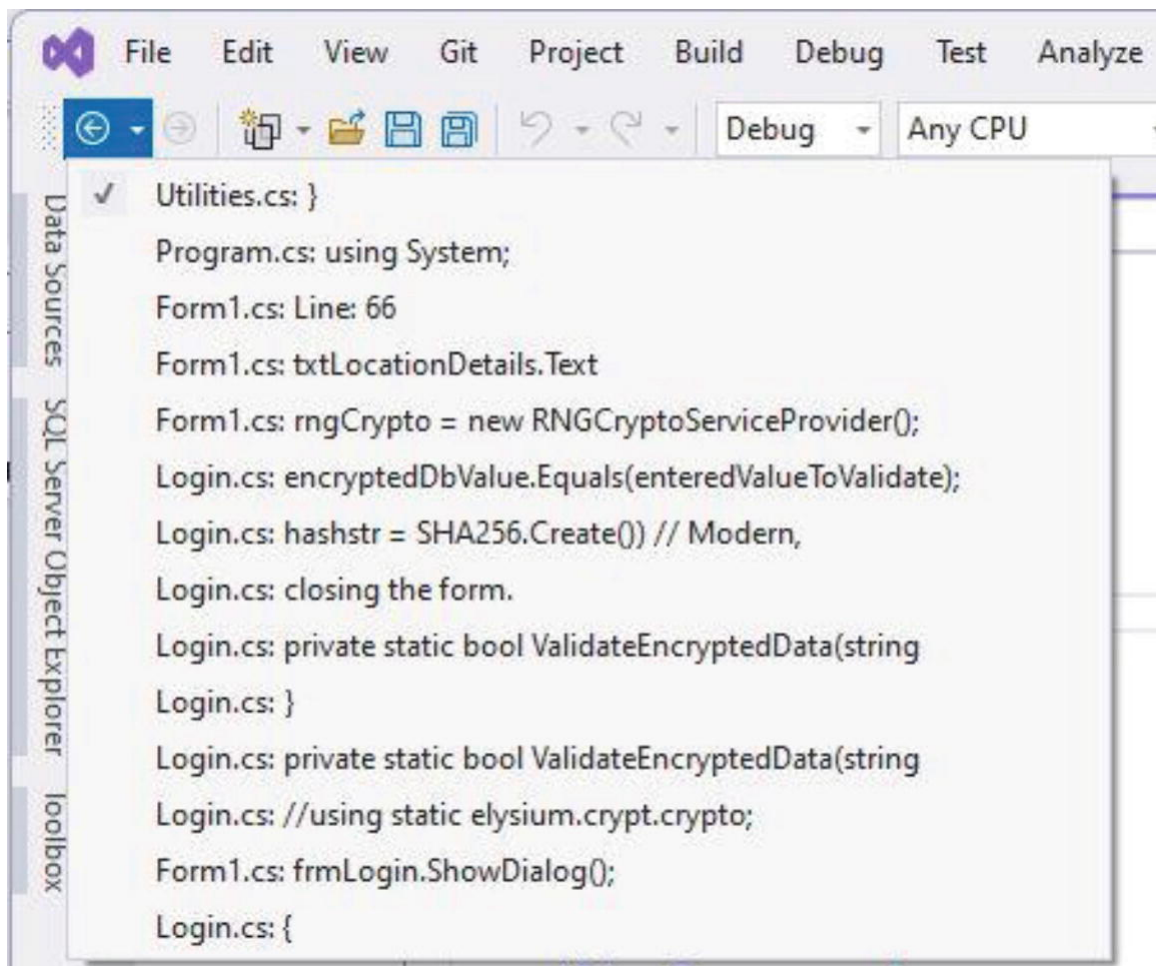
---

## Navigating Code

Visual Studio provides several features allowing developers to navigate code throughout the solution. Knowing how to use these navigation features will save you a lot of time.

### Navigate Forward and Backward Commands

If you look at the toolbar in Visual Studio, you will see the Navigate Forward (Ctrl+Shift+;) and Navigate Backward (Ctrl+;) buttons. These allow developers to return to the last 20 locations that the developer was at, as displayed in Figure [1-22](#).

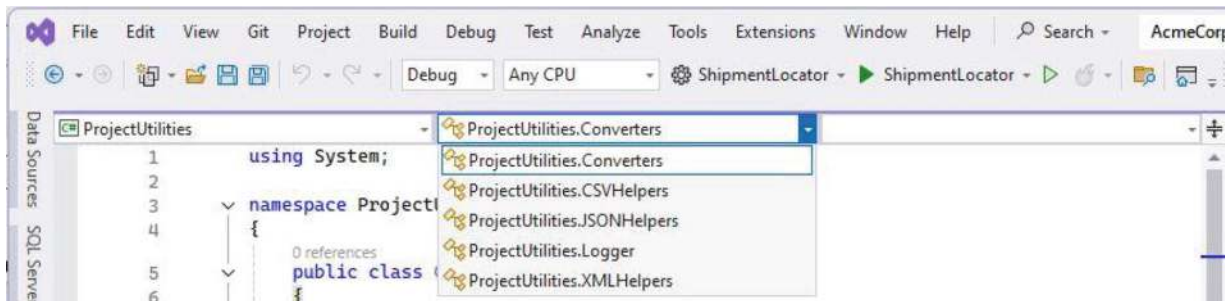


**Figure 1-22** Navigate Forward and Backward

You can also find these commands from the View menu under Navigate Backward and Navigate Forward.

## Navigation Bar

The navigation bar in Visual Studio (Figure 1-23) provides drop-down boxes that allow you to navigate the code in the code base. You can choose a type or member to jump directly to it in the code editor.



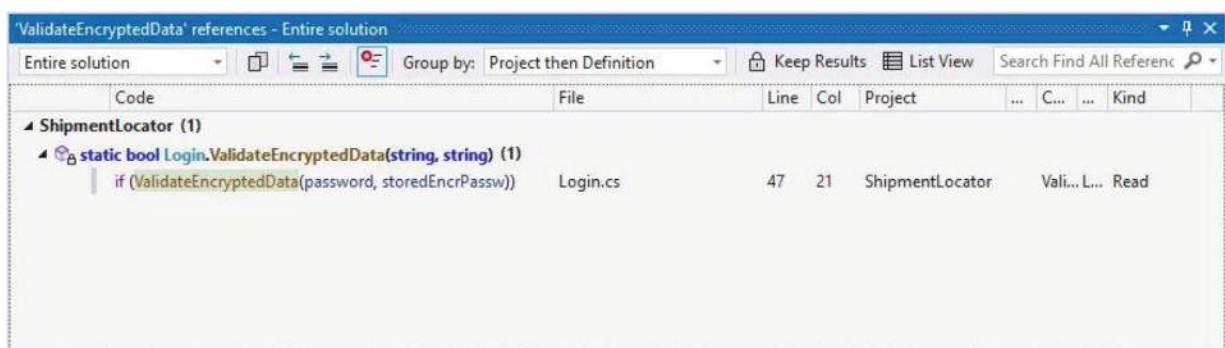
**Figure 1-23** Visual Studio Navigation Bar

It is useful to take note that members defined outside the current code file will be displayed but will be disabled and appear gray. You can cycle through the drop-down boxes in the navigation bar by pressing the tab key.

Each drop-down also has its own individual function. The left drop-down will allow you to navigate to another project that the current file belongs to. To change the focus to another class or type, use the middle drop-down to select it. To navigate to a specific procedure or another member in a particular class, select it from the right drop-down.

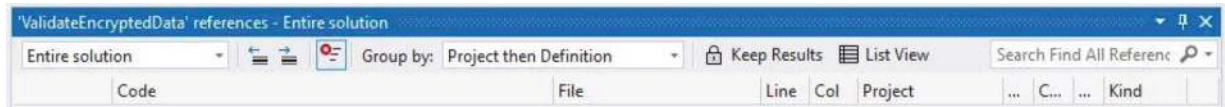
## Find All References

Visual Studio allows you to find all the references for a particular element in your code editor. You can do this by selecting the code element and pressing *Shift+F12* or by right-clicking and selecting *Find All References* from the context menu.



**Figure 1-24** Find All References Results

The find results are displayed in a tool window as seen in Figure [1-24](#). The toolbar for the find results tool window, as seen in Figure [1-25](#), is also really helpful.



**Figure 1-25** References Window Toolbar

From here, you can do the following:

- Change the search scope
- Copy the selected referenced item
- Navigate forward or backward in the list
- Clear any applicable search filters (filters are added by hovering over a column in the results window and clicking the filter icon that is displayed)
- Change the grouping of the returned results
- Keep the search results (new searches are opened in a new tool window)
- Search the returned results by entering text in the Search Find All References text box

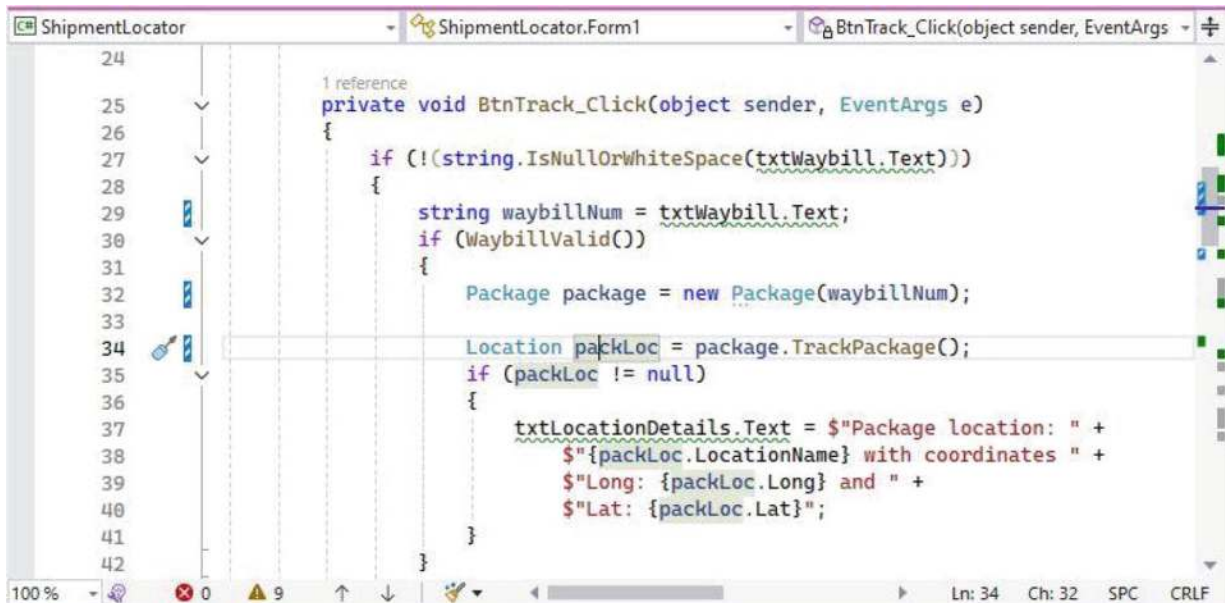
Hovering your mouse on a returned search result will pop up a preview screen of the code. To navigate to a search result, press the Enter key on a reference or double-click it.

## Find Files Faster

When compared to Visual Studio 2019, the Find in Files feature in Visual Studio 2022 is more than 2x faster for 95% of searches. Starting in Visual Studio 2022 17.1 Preview 3, Microsoft introduced indexed Find in Files. This means that according to benchmarks provided by Microsoft, all matches to a search query are returned in just over one second, making the search experience feel instantaneous.

## Reference Highlighting

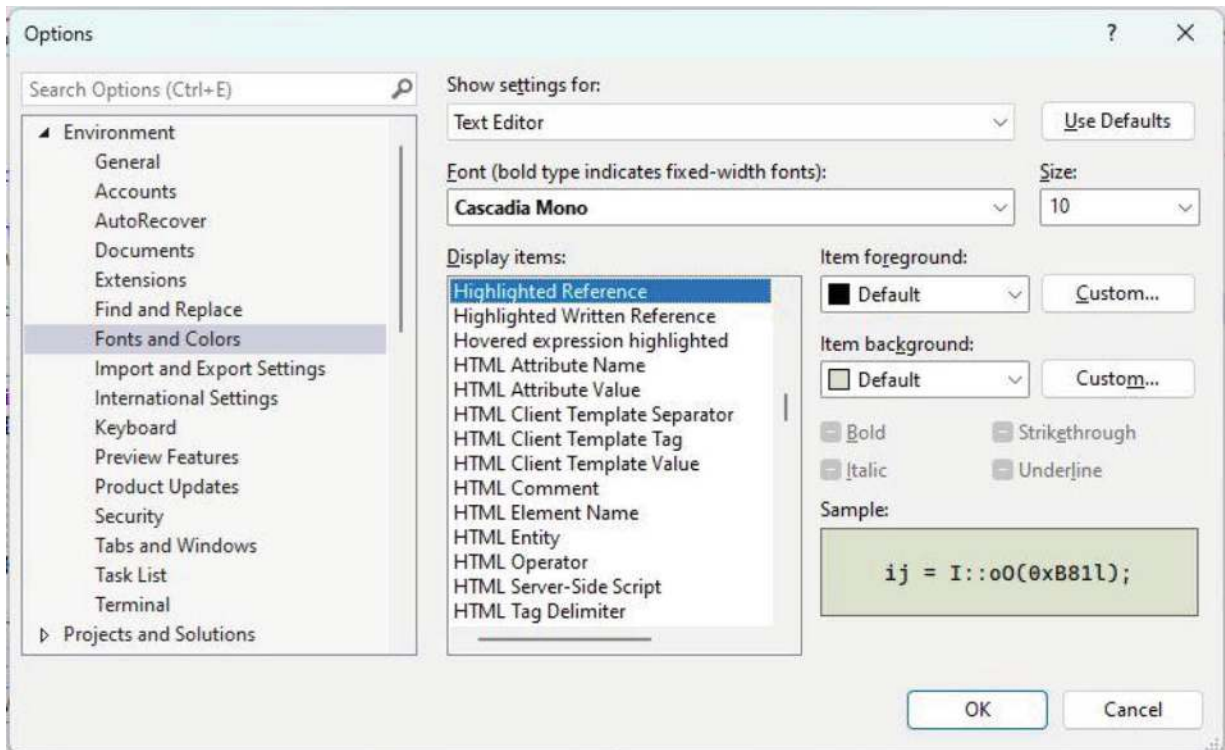
Visual Studio makes it easy to see selected items in the code editor. If you click a variable, for example, you will see all the occurrences of that variable highlighted in the code editor, as seen in Figure [1-26](#).



**Figure 1-26** Default Reference Highlighting

But did you know that you can change the color of the highlight from the Options in Visual Studio? Go to *Tools ► Options ► Environment ► Fonts and Colors ► Highlighted Reference* as seen in Figure [1-27](#).





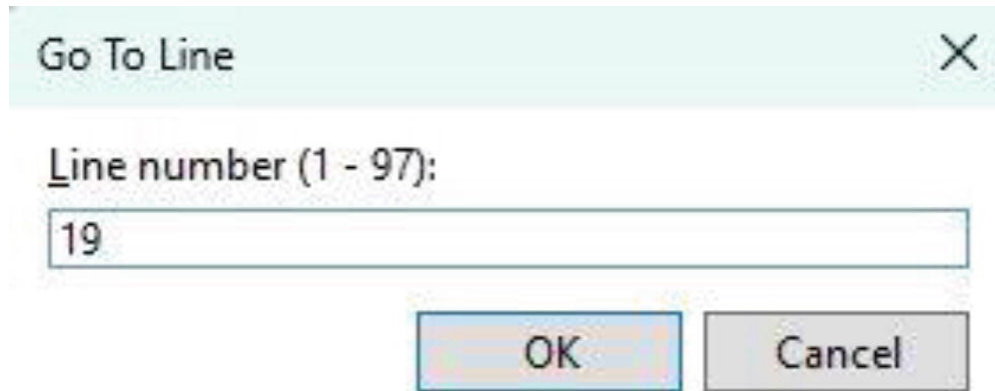
**Figure 1-27** Change the Highlighted Reference Color

Change the color to yellow and click OK. All the references to the variable you just selected will now be highlighted in yellow.

## Go To Commands

I'll admit that these are probably the commands that I use the least in Visual Studio. All with the exception of Ctrl+G. Go ahead and open Visual Studio, and press Ctrl+G while in the code editor.

As shown in Figure [1-28](#), a window appears that lets you jump directly to a specific line. This is especially useful when working with large code files.



**Figure 1-28** Go To Line

The list of Go To commands is as follows:

- Ctrl+G – Go To Line allows you to move to the specified line number in the currently active document.
- Ctrl+T or Ctrl+, – Go To All allows you to move to the specified line, type, file, member, or symbol.
- Ctrl+1, Ctrl+F – Go To File allows you to move to a specified file in the solution.
- Ctrl+1, Ctrl+R – Go To Recent File allows you to move to a recently visited file in the solution.
- Ctrl+1, Ctrl+T – Go To Type allows you to move to a specific type in the solution.
- Ctrl+1, Ctrl+M – Go To Member allows you to move to the specific member in the solution.
- Ctrl+1, Ctrl+S – Go To Symbol allows you to move to the specific symbol in the solution.
- Alt+PgDn – Go To Next Issue in File.
- Alt+PgUp – Go To Previous Issue in File.
- Ctrl+Shift+Backspace – Go To Last Edit Location.

While pressing Ctrl+1 might seem slightly finicky, you soon get used to it, and the commands start to feel more natural. Ctrl+Shift+Backspace is another command that I find very useful.

## **Go To Definition**

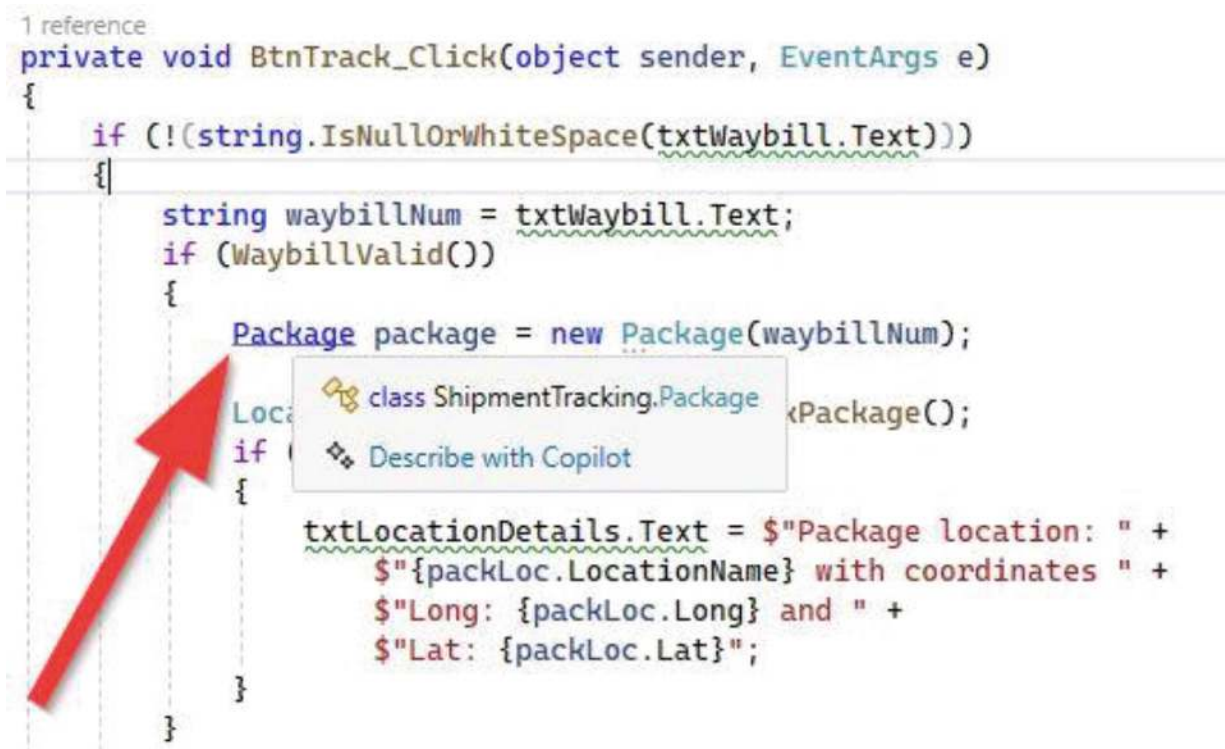
Go To Definition allows you to jump to the definition of the selected element. Go to the example project for this chapter, and find the click

event for the Track button.

Just a reminder that the code for this book can be downloaded from GitHub at the following URL:  
[github.com/apress/getting-started-vs2022](https://github.com/apress/getting-started-vs2022).

In there, you will see that we are working with a class called `Package` that creates a new package we would like to track.

Place your cursor on `Package`, and hit F12 to jump to the class definition as seen in Figure 1-29. You can also hold down the Ctrl button and hover over the class name. You will notice that the `Package` becomes a link you can click. Lastly, if you have your feet up and you only have your mouse to navigate with (the other hand is holding a cup of coffee), you can right-click and select *Go To Definition* from the context menu.

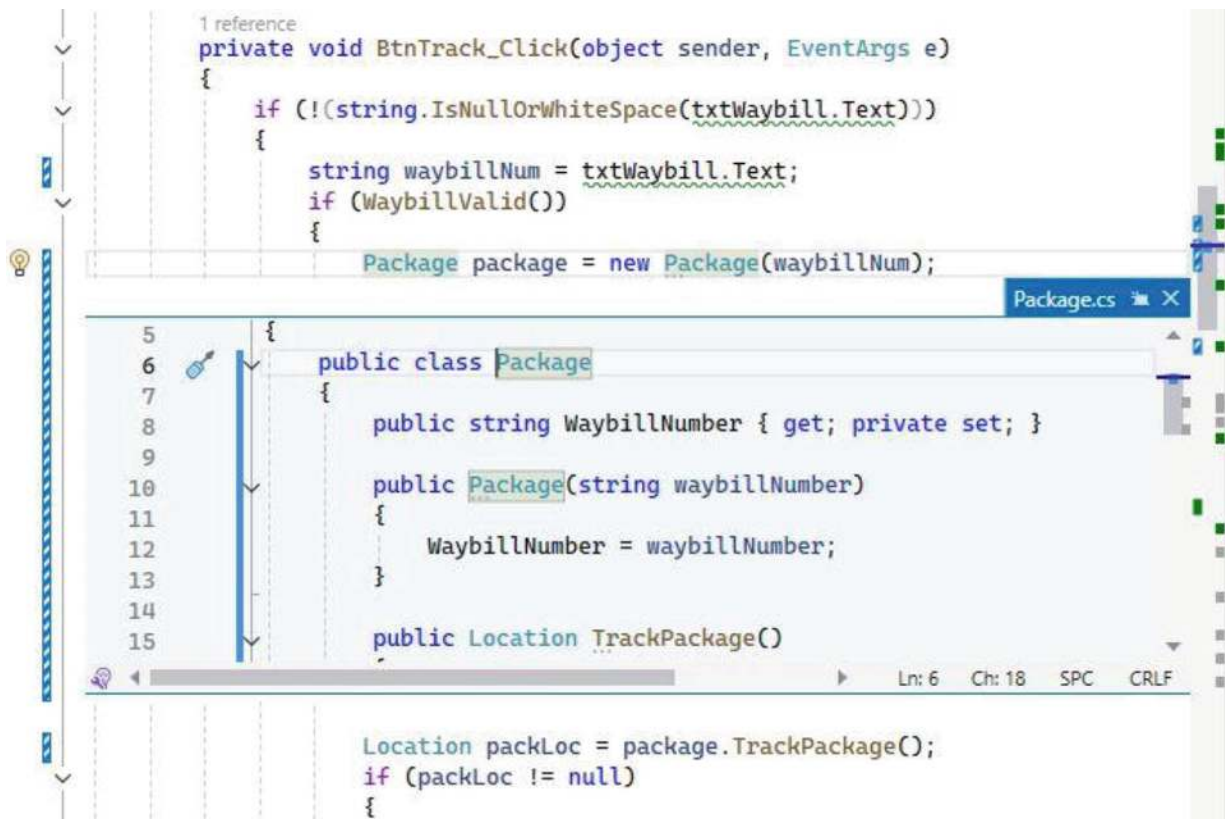


**Figure 1-29** Go To Definition

## Peek Definition



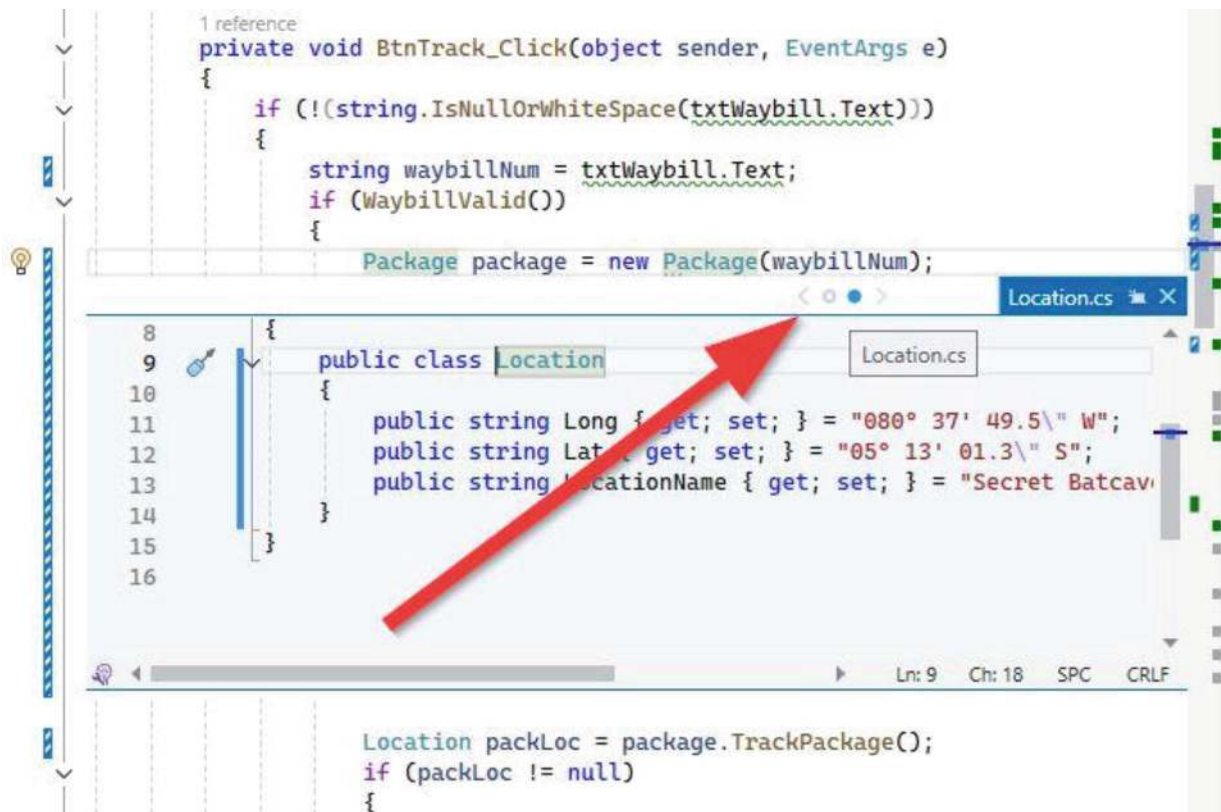
Where Go To Definition navigates to the particular definition in question, Peek Definition simply displays the definition of the selected element in a pop-up. Place your cursor on `Package` and right-click. From the context menu, select Peek Definition.



**Figure 1-30** Peek Definition Pop-up

As can be seen in Figure [1-30](#), the pop-up window displays the code for the `Package` class. You can navigate through the code displayed in this pop-up as you would any other code window. You can even use Peek Definition or Go To Definition inside this pop-up.

In the pop-up window, right-click `Location`, and select Peek Definition from the context menu. The second Peek Definition will start a breadcrumb path as seen in Figure [1-31](#).



**Figure 1-31** Breadcrumb Path

You can now navigate using the circles and arrows that appear above the Peek Definition pop-up window. The arrows only appear when you hover your mouse over the circles, but this makes it much easier to move between the code windows.

## Subword Navigation

Subword navigation is a very nice feature in Visual Studio 2022. Suppose you have a method name called `DetermineValueOfFoo` that consists of four subwords, namely, `Determine`, `Value`, `Of`, and `Foo`. Subword navigation allows you to move the caret to the next or previous subword in the string by holding down `Ctrl+Alt+Left` or `Ctrl+Alt+Right`. To select the previous or next subword, hold down `Ctrl+Alt+Shift+Left` or `Ctrl+Alt+Shift+Right`. This will select the next subword in the string. You can also turn on `Select subword on double click` by going to `Tools > Options > Text Editor > General` and check the `Select subword on double click` option. Now, when you double-click

a string, it will select the current subword you clicked instead of the entire string.

---

## Features and Productivity Tips

Visual Studio is full of existing productivity tips that have been around for years and that some developers do not know about. In this section, we will be looking at some of those.

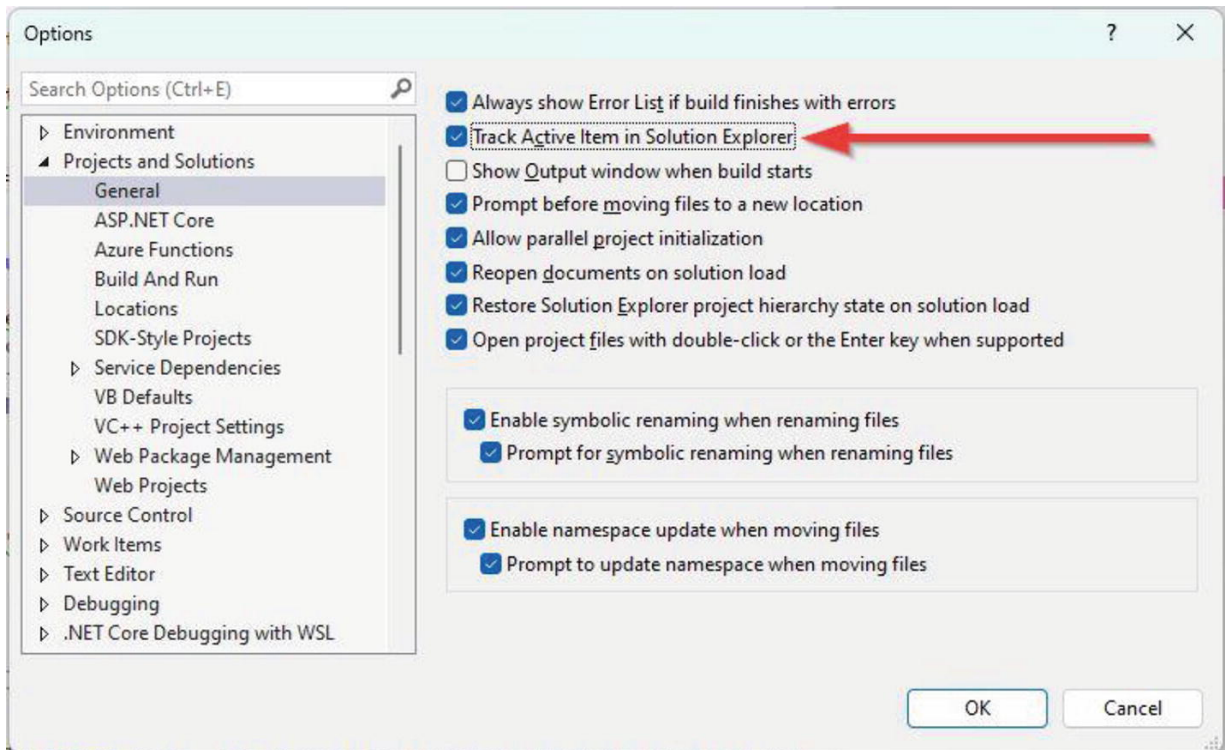
### Track Active Item in Solution Explorer

In Visual Studio, this option is not on by default. As you change the code file you are working in, the file isn't highlighted in the Solution Explorer. You will know this is the case when you see the following arrows in the toolbar of the Solution Explorer, as seen in Figure [1-32](#).



**Figure 1-32** Track Active Item in Solution Explorer

Clicking these arrows will highlight the file that you are currently editing in the Solution Explorer. I find this extremely useful, so I let Visual Studio permanently track the current file. To set this, click Tools ► Options ► Projects and Solutions ► General, and check Track Active Item in Solution Explorer (Figure [1-33](#)).











**Figure 1-33** Track Active Item Setting

With this setting enabled, the arrows are not displayed in the toolbar of the Solution Explorer.

## Hidden Editor Context Menu

When you are in a code file, you can access a variety of menu items by right-clicking and selecting the menu items in the context menu. But did you know that you can hold down **Alt+`** to bring up a special context menu (that is different from right-click)? It has more editor commands in it, as seen in [Figure 1-34](#).

	Peek Definition	Alt+F12
	Go To Definition	F12
	Go To Base	
	Go To Implementation	Ctrl+F12
	Find All References	Ctrl+K, R
	View Call Hierarchy	Ctrl+K, Ctrl+T
	Track Value Source	
	Locate in Solution Explorer	Ctrl+[, S
	Navigate Forward	Ctrl+Shift+-
	Navigate Backward	Ctrl+-
	Go To Last Edit Location	Ctrl+Shift+Bkspce
	Go To Next Issue in File	Alt+PgDn
	Go To Previous Issue in File	Alt+PgUp
	Go To Containing Block	Shift+Alt+[
	Next Method	
	Previous Method	
	Next Bookmark	Ctrl+B, N
	Previous Bookmark	Ctrl+B, P

**Figure 1-34** Special Context Menu

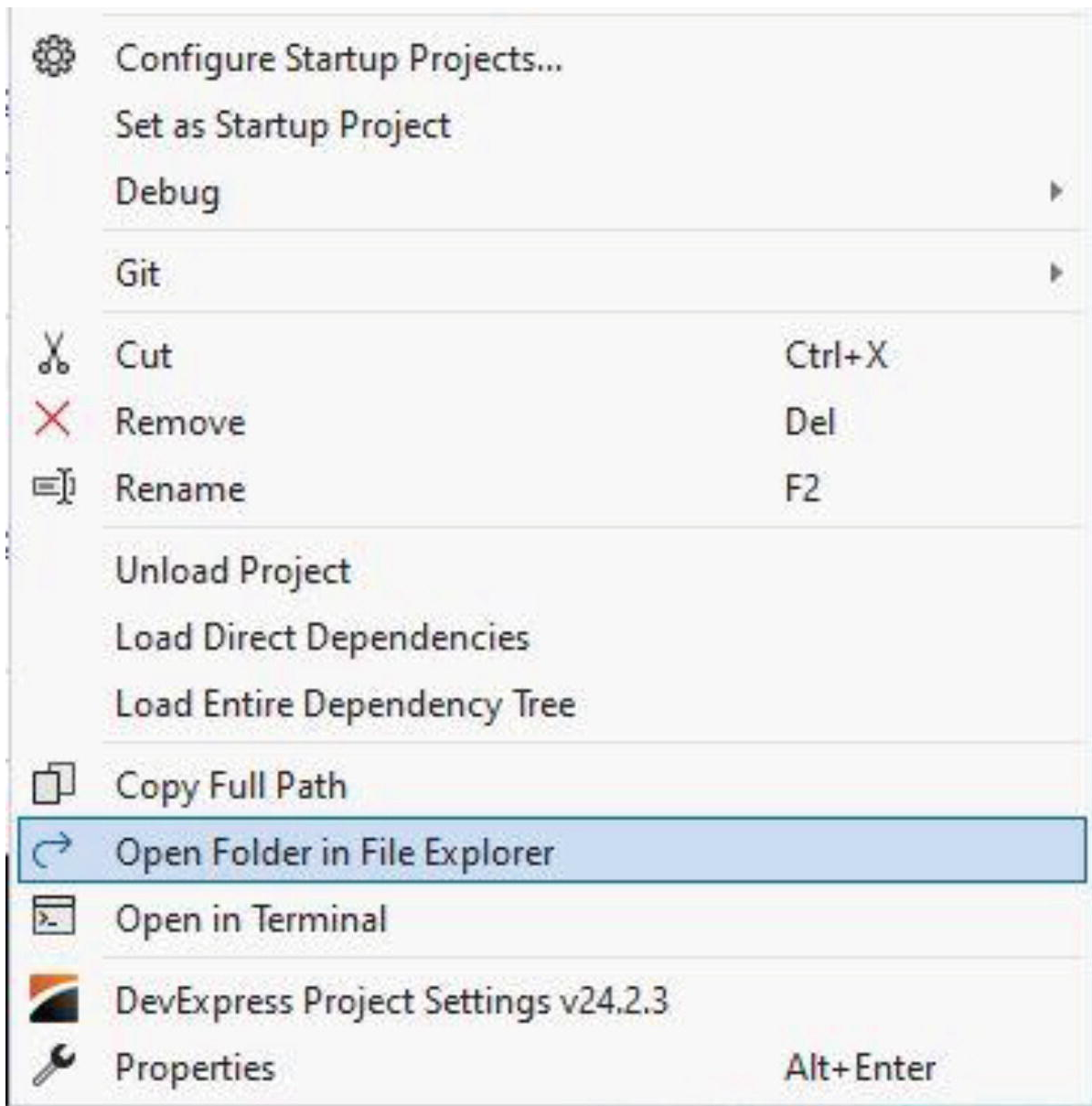
This gives you a little more control over navigating through errors, methods, etc., in your current code file.

## Open in File Explorer

Sometimes, you need to quickly get to the actual Visual Studio files of your solution. This might be to go and copy a file from the bin folder or to open a file in another text editor. To do this, you don't even need to



leave Visual Studio. As seen in Figure 1-35, you can right-click the solution and click Open Folder in File Explorer.



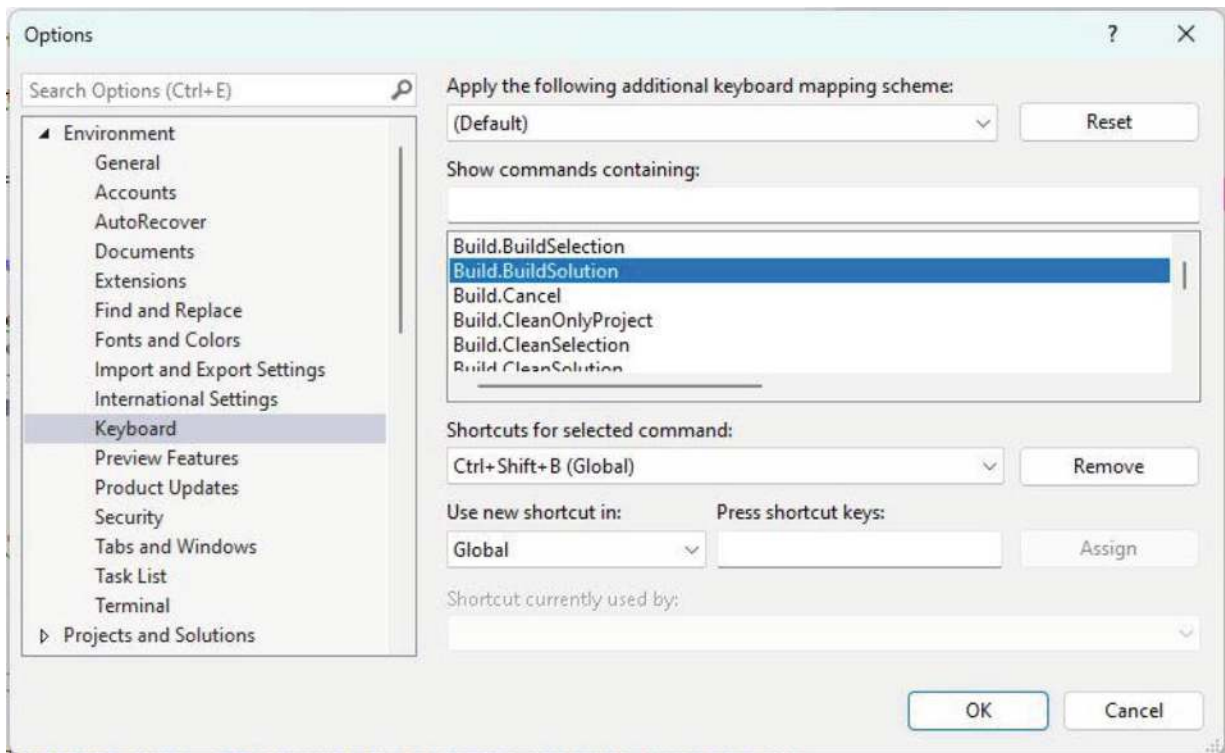
**Figure 1-35** Open Folder in File Explorer

This will open a new File Explorer window where your Visual Studio solution is located.

## Finding Keyboard Shortcut Mappings

Sometimes, when you use a keyboard shortcut and nothing happens, you might be using it in the wrong context. To see what keyboard

shortcuts are mapped to, head on over to Tools ► Options ► Environment ► Keyboard, as seen in Figure [1-36](#).



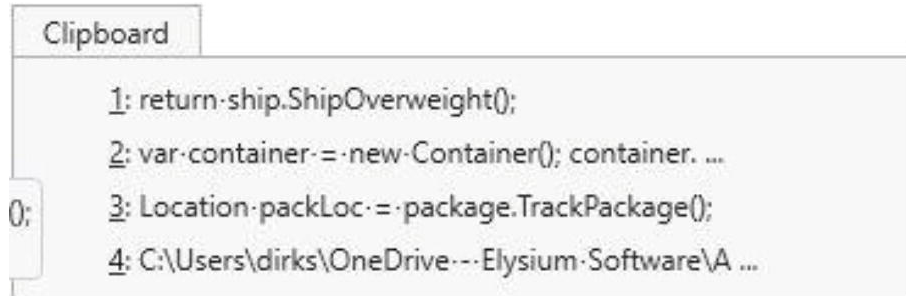
**Figure 1-36** Find Keyboard Shortcut Mappings

Press the shortcut keys, and Visual Studio will show you what the shortcut is currently used for. This is also very useful for assigning new keyboard shortcuts to check that the keyboard shortcut you have in mind is not already bound to another command.

## Clipboard History

Visual Studio allows you to access your clipboard history. This is very useful if you have to copy and paste several items repeatedly.

Instead of going back and forth between copy and paste, simply hold down Ctrl+Shift+V to bring up the clipboard history as seen in Figure [1-37](#).



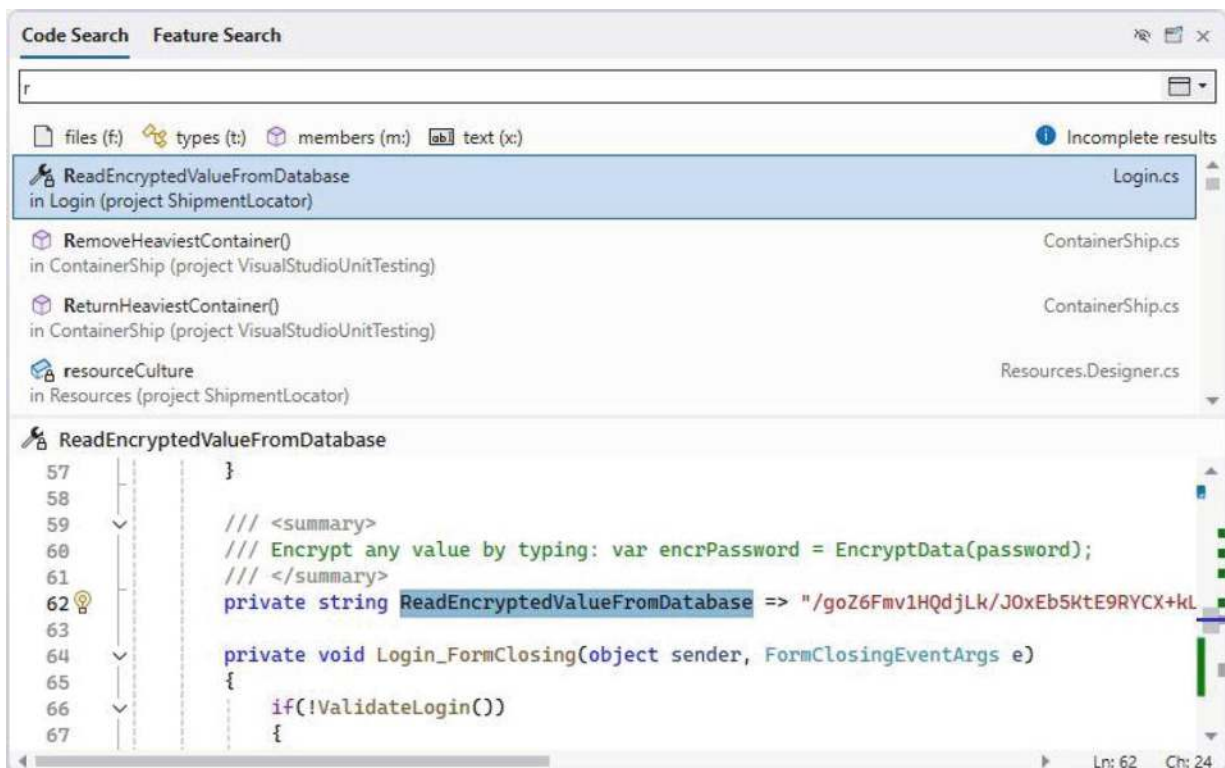
**Figure 1-37** Clipboard History

Now you can just select the copied text that you want to paste and carry on with editing your code. The copied item also remains in the clipboard history after pasting.

## Go To Window

So this could actually have gone under the “Navigating Code” section, but I wanted to add it here because it made more sense to discuss it as a productivity tip.

Hold down Ctrl+T and you will see the Search window pop up as seen in Figure [1-38](#).





**Figure 1-38** Go To window

You can view the recent files by typing in an *r*. Also nice to note is the ability to jump to a specific line of code. You will remember earlier in the chapter that we discussed the Go To commands and Ctrl+G in particular. Here, you can do the same thing by typing in : followed by the line number.

## Navigate to Last Edit Location

Earlier in this chapter, we discussed the Navigate Backward and Navigate Forward commands. This is great, but if you want to navigate to the last place you made an edit in the code file, hold down Ctrl+Shift+Backspace. This will jump to the last place that you made an edit in one of your code files.

## Multi-caret Editing

This is a feature that I love using. Consider the SQL create table statement in Listing [1-2](#).

```
CREATE TABLE [dbo].[menu] (  
    [itemName] [varchar](50) NOT NULL,  
    [category] [varchar](50) NOT NULL,  
    [description] [varchar](50) NOT NULL,  
    CONSTRAINT [PK_menu] PRIMARY KEY CLUSTERED
```

**Listing 1-2** Create SQL Table Statement

This is a rather small table, but sometimes we have very large tables that we need to work with. I want to create a simple C# class for this table and need to create some C# properties. Why type out everything when you can copy, paste, and edit all at once?

Paste the column names into a C# class file, then hold down Ctrl+Alt, and click in front of the first square bracket of each column as can be seen in Figure [1-39](#).

```
public string itemName][varchar](50) NOT NULL,  
public string category] [varchar] (50) NOT NULL,  
public string description] [varchar] (50) NOT NULL,
```

**Multi-caret**

**Figure 1-39** Multi-caret Selection

The cursor is placed at each line at the position you placed it. Now start typing the property definition. All the lines are edited. After typing `public string`, hit delete to remove the first square bracket.

I now want to add the `{ get; set; }` portion of my property. I can either do the same multi-caret selection or select one or more characters and hold down `Shift+Alt+.` to select matching selections as seen in [Figure 1-40](#).

```
public string itemName] [varchar](50) NOT NULL,  
public string category] [varchar] (50) NOT NULL,  
public string description] [varchar] (50) NOT NULL,
```

**Select matching**

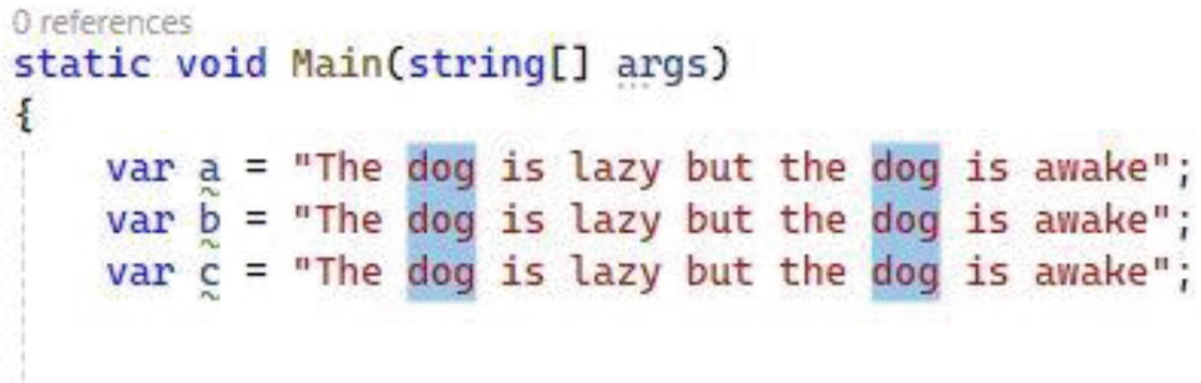
**Figure 1-40** Selecting Matching Selections

This now allows me to easily select exactly all the lines I want to edit at the same time and allows me to paste the `{ get; set; }` needed for my properties. I now end up with the completed code as seen in [Figure 1-41](#).

```
public string itemName { get; set; } //[varchar](50) NOT NULL,  
public string category { get; set; } //[varchar] (50) NOT NULL,  
public string description { get; set; } //[varchar] (50) NOT NULL,
```

**Figure 1-41** Completed Code Properties

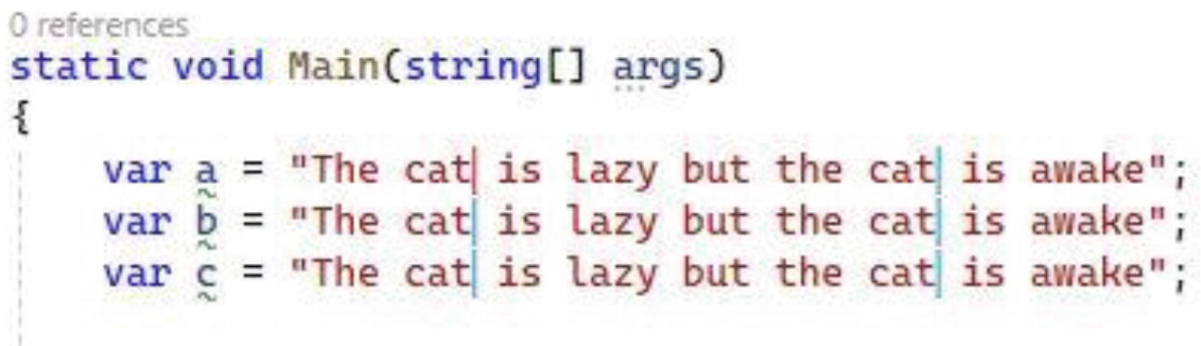
Being able to easily select code or place a caret in several places on the same line or across lines allows developers to be flexible when editing code. Speaking about placing the caret in several places on the same line, it is, therefore, possible to do the one shown in Figure [1-42](#).



A screenshot of a code editor showing a C# `Main` method. The first three lines of the method body are: `var a = "The dog is lazy but the dog is awake";`, `var b = "The dog is lazy but the dog is awake";`, and `var c = "The dog is lazy but the dog is awake";`. In each of these three lines, the word `dog` is selected with a blue highlight. Above the first line, the text "0 references" is visible.

**Figure 1-42** Multi-caret Selection on the Same Line

We can now edit everything at once (even if we have selected multiple places on the same line), as seen in Figure [1-43](#).



A screenshot of a code editor showing the same C# `Main` method as in Figure 1-42. The first three lines are: `var a = "The cat is lazy but the cat is awake";`, `var b = "The cat is lazy but the cat is awake";`, and `var c = "The cat is lazy but the cat is awake";`. In each of these three lines, the word `cat` has a blue vertical caret positioned at the end of the word. Above the first line, the text "0 references" is visible.

**Figure 1-43** Multi-caret Editing on the Same Line

Holding down `Ctrl+Z` will also work to undo everything at once. If you want to insert carets at all matching selections, you can select some text and hold down `Shift+Alt+;` to select everything that matches your current selection, as seen in Figure [1-44](#).

```

0 references
public string itemName { get; set; } //varchar] (50) NOT NULL,
0 references
public string category { get; set; } //varchar] (50) NOT NULL,
0 references
public string description { get; set; } //varchar] (50) NOT NULL,

0 references
static void Main(string[] args)
{
    var a = "The cat is lazy but the cat is awake";
    var b = "The cat is lazy but the cat is awake";
    var c = "The cat is lazy but the cat is awake";

    try
    {
    }
    catch (Exception ex)
    {
        Logger.Log(ex.Message);
    }
}

```

**Figure 1-44** Insert Carets at All Matching Selections

I selected the text “cat” and held down Shift+Alt+; and Visual Studio selected everything that matches. As you can see, it also selected the category property, which I don’t want to be selected. In this instance, Shift+Alt+; will allow me to be more specific in my selection.

If you find yourself forgetting the keyboard shortcuts, you can find them under the Edit menu. Click Edit ► Multiple Carets to see the keyboard shortcuts.

## Sync Namespaces to Match Your Folder Structure

Another great feature is the ability to keep your namespaces in sync with your folder structure. This is useful when you need to restructure your solution by moving files around to new folders. What you want to do is ensure that the namespace in the file stays in sync with the new folder structure. To do this, place your cursor on the namespace name

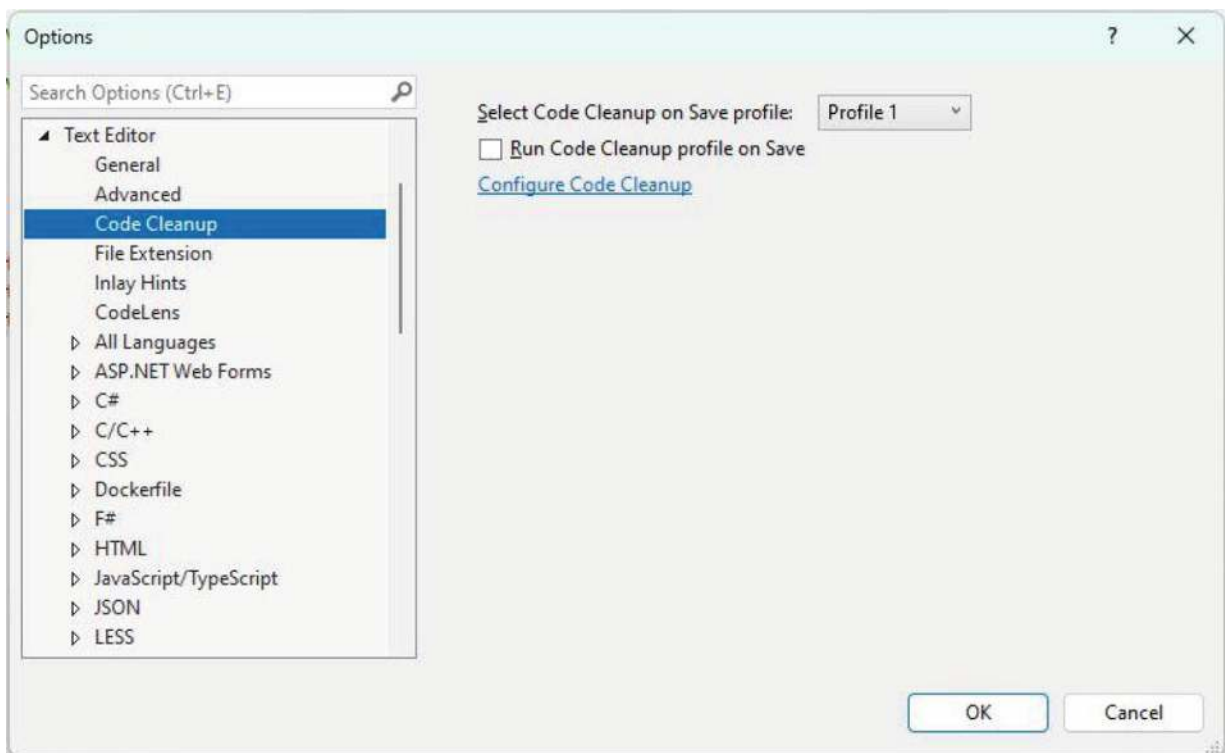
and hold down Ctrl+. to bring up the Quick Actions and Refactoring menu. Select the option to change the namespace to your folder name.

## Paste JSON As Classes

If you work with JSON often, the following feature is another gem. If you have copied some JSON and need an object to serialize and deserialize the JSON into (using `System.Text.Json`, for example), you can select Edit from the Visual Studio menu, and select the Paste Special option. From there, you will see an option to Paste JSON as Classes. You can do the same for XML. This will quickly generate the correct class structure to represent the JSON copied by yourself. This is a fantastic time saver.

## Enable Code Cleanup on Save

Another great feature in Visual Studio 2022 is the ability to perform a code cleanup on save. From the Tools menu, select Options ► Text Editor ► Code Cleanup as seen in Figure 1-45.



**Figure 1-45** Code Cleanup on Save



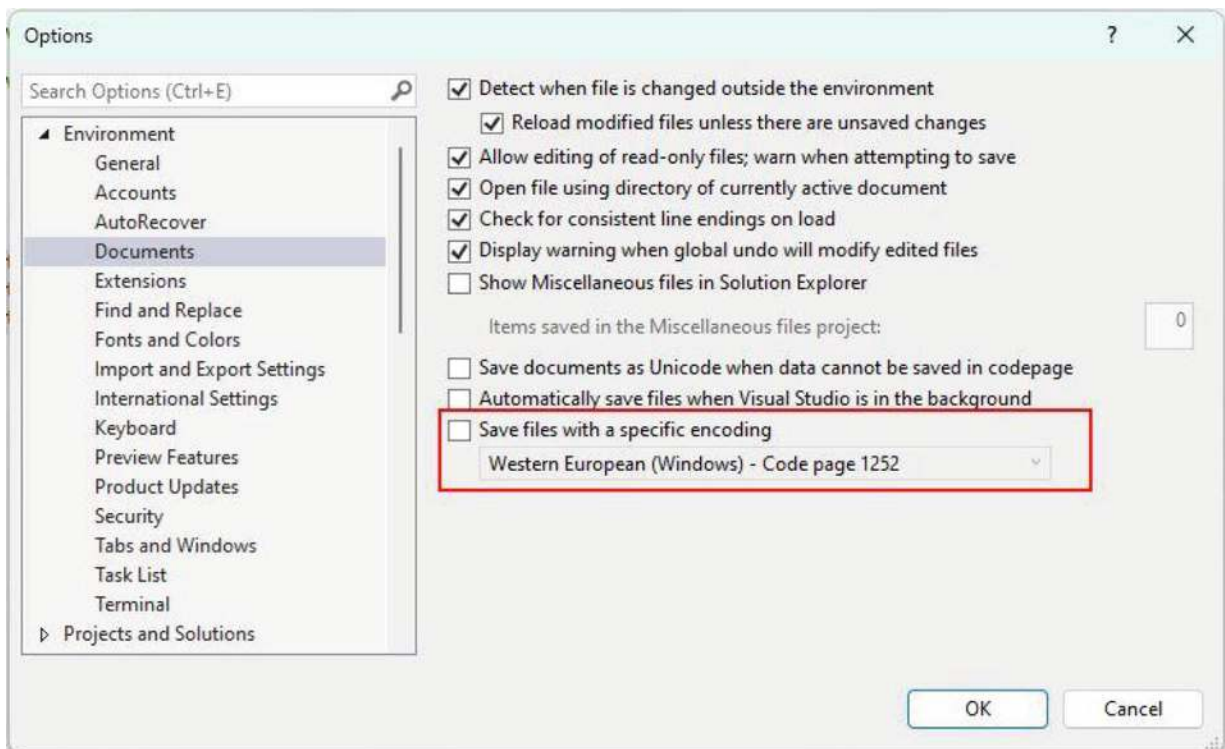
Here, you will see that you can configure the code cleanup options by clicking the link and selecting the available fixes to apply. Once you have configured a code cleanup profile, you can check the option to run the code cleanup profile when saving a file.

## Add Missing Using on Paste

Developers often copy and paste code found in another section of the project or from another online resource. Doing this usually requires developers to add the missing using statements. Now you can enable this feature to automatically add the missing using statements when pasting copied code. Go to Tools ► Options ► Text Editor ► C# ► Advanced and scroll down a bit, and you will see an option to Add missing using directives on paste.

## Customize File Encoding

For developers that work in cross-platform environments, there is often a need to save files with specific encodings. Visual Studio now allows developers to set the default file encoding when saving files.

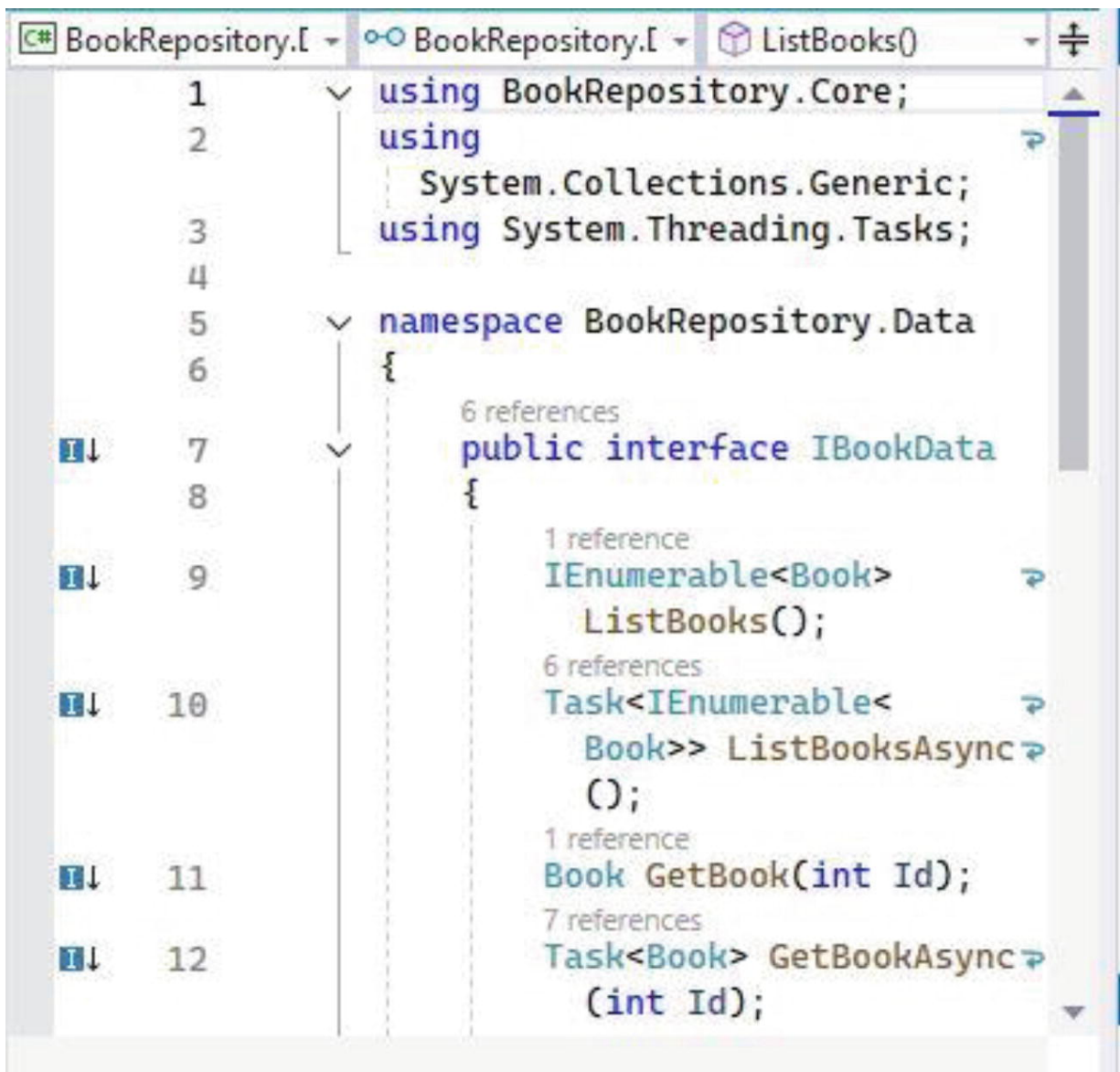


**Figure 1-46** Customize File Encoding

To enable this, navigate to Tools ► Options ► Environment ► Documents and look for an option titled **Save Files with a specific encoding**. Check this option to specify the encoding in the adjacent combo box.

## Indent Word Wrap

Another exciting feature is that Visual Studio now allows you to specify whether wrapped lines are indented in the editor or not.



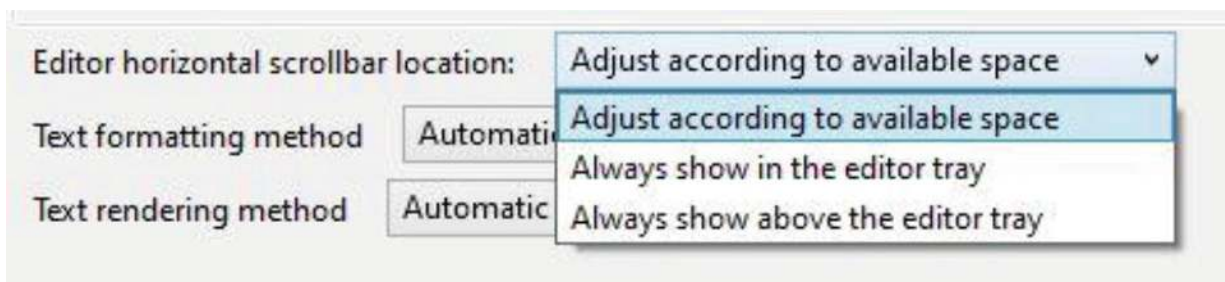
**Figure 1-47** Indent Word Wrap

This option is enabled by default, but if you want to change it, go to Tools ► Options ► Text Editor ► General and look for the option labeled **Automatically indent when wrap is enabled**. Setting this feature on can help improve readability for developers with little screen real estate.

## The Horizontal Scrollbar Reimagined

Visual Studio's editor tray provides developers with a wealth of information. From controlling the zoom to seeing what line you're on. All this information can sometimes crowd out the horizontal scrollbar, making it challenging when trying to scroll through your window. This can be especially noticeable when viewing code side-by-side.

Visual Studio allows you to control what happens when the scrollbar drops below a usable width.



**Figure 1-48** Horizontal Scrollbar Location

Go to Tools ► Options ► Text Editor ► Advanced and change the setting under **Editor horizontal scrollbar location**. Doing this will allow you to choose whether the scrollbar changes position according to the space available, stays in the editor tray, or always displays above the editor tray.

## Automatic Documentation Comment Generation

As you have probably already heard, Copilot is automatically integrated into Visual Studio's editor and helps you generate doc comments for functions. We will discuss Copilot a little later on in this chapter, but just know that you will need a GitHub Copilot subscription to access this feature. You can sign up for GitHub Copilot for free from

<https://github.com/settings/copilot>.



If you have enabled doc comment generation in your settings, you can simply type `///` (or whatever triggers your doc comment format) and Copilot suggestions will allow you fill out the function description automatically as seen in Figure [1-49](#).



The screenshot shows a code editor with a C# method `SaveAsync`. Above the method signature, Copilot has generated XML documentation comments. A small 'Tab: accept' button is visible above the first line of the comments. The comments include a summary, remarks, type parameters, a parameter, and a return value. The method signature is `public async Task<int> SaveAsync<T>(T entity) where T : IEntity`. The method body contains logic to add the entity to the database, save changes, and return the entity's ID.

```
/// Tab: accept
/// Asynchronously saves the specified entity to the database and returns its identifier.
/// </summary>
/// <remarks>This method adds the entity to the database context and attempts to persist changes
/// asynchronously. If the save operation is successful, the method retrieves the value of the "Id" property
/// from the entity.</remarks>
/// <typeparam name="T">The type of the entity to save. Must implement <see cref="IEntity"/>.</typeparam>
/// <param name="entity">The entity to save. Cannot be <see langword="null"/>.</param>
/// <returns>The identifier of the saved entity as an <see cref="int"/>. Returns -1 if the save operation fails.</returns>
3 references
public async Task<int> SaveAsync<T>(T entity) where T : IEntity
{
    var addedEntity = _database.Add(entity);
    var entityId = -1;

    if (await _database.SaveChangesAsync() > -1)
    {
        entityId = Convert.ToInt32(addedEntity.Property("Id").CurrentValue);
    }

    return entityId;
}
```

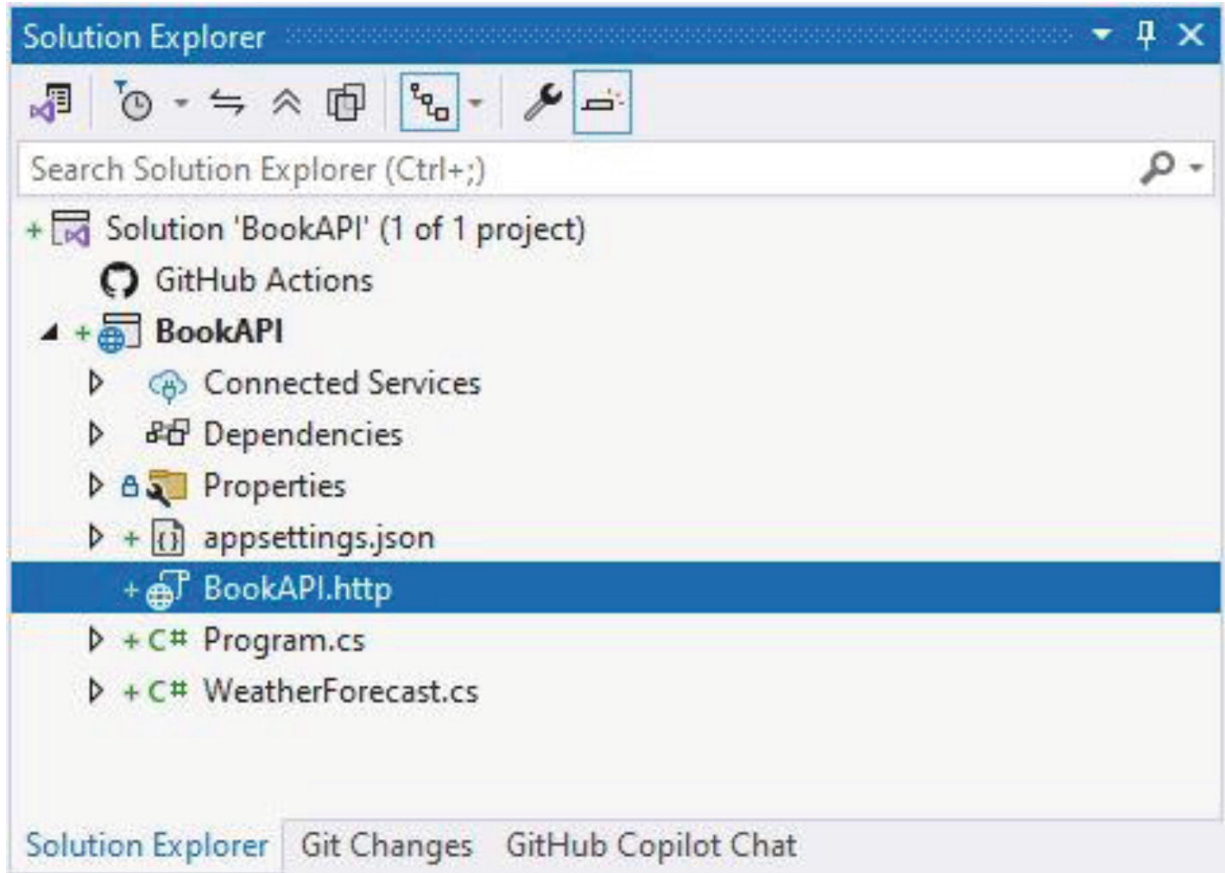
**Figure 1-49** Automatic Documentation Comment Generation

To switch on doc comments in your settings (for C#) you need to go to Tools ► Options ► Text Editor ► C# ► Advanced and check the options under the Comments section.

## HTTP File Updates

If you develop API projects, you would, in all likelihood, have used `.http` files in Visual Studio. For those that haven't worked with `.http` files in Visual Studio, check out this link [learn.microsoft.com/en-us/aspnet/core/test/http-files](https://learn.microsoft.com/en-us/aspnet/core/test/http-files).

Visual Studio now supports using request variables in `.http` files.



**Figure 1-50** The .http File in a Minimal API Project

If you create a new ASP.NET Core API project in Visual Studio, you will notice that an .http file is included in the template. If you have not used .http files before, you might be wondering what the big deal is. It comes down to this: in order to test my API, I no longer need to use Postman or Insomnia, for example. I can invoke every endpoint directly inside Visual Studio, edit the request, and see the results instantly. This tightens my inner dev loop (that edit-build-run-validate cycle), providing less friction while developing API solutions.

## ***Request Variables***

If you have worked with APIs, you will know that a common thing to do is to read a value from an endpoint and then use that value in a subsequent request. In Visual Studio, .http files now support request variables.

Let's see this implemented in a working example. The code is in Listing [1-3](#).

```

@base_address = https://reqres.in/api

###
# 1) Define your login credentials as variables
@username = eve.holt@reqres.in
@password = cityslicka

###
# 2) Login request - name it "login" to capture
the response
# @name login
POST {{base_address}}/login
Content-Type: application/json
x-api-key: reqres-free-v1

{
  "email": "{{username}}",
  "password": "{{password}}"
}

###

# 3) Protected call - reuse the extracted token
GET {{base_address}}/users/1
Authorization: Bearer
{{login.response.body.$.token}}
x-api-key: reqres-free-v1

```

**Listing 1-3** The .http file

I have used REQRES, which is a handy public API that lets me demonstrate authentication flows without spinning up my own backend. The URL is <https://reqres.in>, just in case you wanted to have a look at what is available. Let's break the .http file up and look at the parts.

```
@base_address = https://reqres.in/api

###
# 1) Define your login credentials as variables
@username = eve.holt@reqres.in
@password = cityslicka
```

**Figure 1-51** Top-level Assignments

The top-level assignments as seen in Figure [1-51](#) keep sensitive bits out of the request body and make the file easy to tweak later.

```
###
# 2) Login request - name it "login" to capture the response
# @name login
Send request | Debug
POST {{base_address}}/login
Content-Type: application/json
x-api-key: reqres-free-v1

{
  "email": "{{username}}",
  "password": "{{password}}"
}

###
```

**Figure 1-52** Login and Capture Response

The line `# @name login`, as seen in Figure [1-52](#), tells Visual Studio to store both the request and its response under the name `login`. After the call completes, everything in that exchange is at my fingertips.

```
# 3) Protected call - reuse the extracted token
Send request | Debug
GET {{base_address}}/users/1
Authorization: Bearer {{login.response.body.$.token}}
x-api-key: reqres-free-v1
```

**Figure 1-53** Reuse the Token

In fact, I can drill into the JSON payload returned by the login request and pluck out the token field by writing `login.response.body.$.token`. No more copy-and-paste or messy edits, stops, and starts. The moment the first call succeeds, the second request has an up-to-date bearer token.

## ***\$shared Environment***

Using the `$shared` environment allows developers to avoid copy-and-paste overload. Most developers juggle at least two API endpoints. This is usually a local and a remote endpoint, and occasionally, they need to juggle more. Visual Studio's `http-client.env.json` file allows you to switch between them with a drop-down, but repeating the same values in every environment gets old fast. This is where the `$shared` section comes in. Have a look at Listing [1-4](#).

```
{
  "$shared": {
    "message": "Shared environment message",
    "username": "httpfile-user",
    "hosturl": "http://example.com/api/sample"
  },
  "dev": {
    "hosturl": "http://localhost:5000/api/sample"
  },
  "prod": {
    "message": "Production environment message"
  }
}
```

#### **Listing 1-4** The Environment File

Here I have specified a `$shared` section. These variables will be used everywhere unless an environment overrides them. In other words, if a key appears in both `$shared` and a specific environment, the environment-specific value wins.



**Figure 1-54** Selecting the dev Environment

Looking at Figure [1-54](#), you will notice that the environment selected in the top right corner is the dev environment. The response from this request is listed in Listing [1-5](#).

```
{
  "headers": {
    "Host": "httpbin.org",
    "Traceparent": "00-
1da30b3a47f83246b5526d9a8a45cff4-09cbdcf68670250e-
00",
    "X-Amzn-Trace-Id": "Root=1-682aca37-
2257e3d42d17487e328b007c",
    "X-Hosturl":
"http://localhost:5000/api/sample",
    "X-Message": "Shared environment message",
    "X-User": "httpfile-user"
  }
}
```

**Listing 1-5** The dev Response

Changing the environment in the top right corner to prod will result in the following response, as seen in Listing [1-6](#).

```
{
  "headers": {
    "Host": "httpbin.org",
    "Traceparent": "00-
b2825110f3399368fd557ba0eb00d799-bc223e0fa85f9d9f-
00",
    "X-Amzn-Trace-Id": "Root=1-682acaef-
086783095e9bcd30a87cfc3",
    "X-Hosturl": "http://example.com/api/sample",
    "X-Message": "Production environment message",
    "X-User": "httpfile-user"
  }
}
```

**Listing 1-6** The prod Response

Notice that the message differs based on the environment, where the prod environment's message variable wins, even though it is defined in `$shared`. The key takeaway here is that the `$shared` block turns environment management from a chore into a one-liner.

---

## Features in Visual Studio 2022

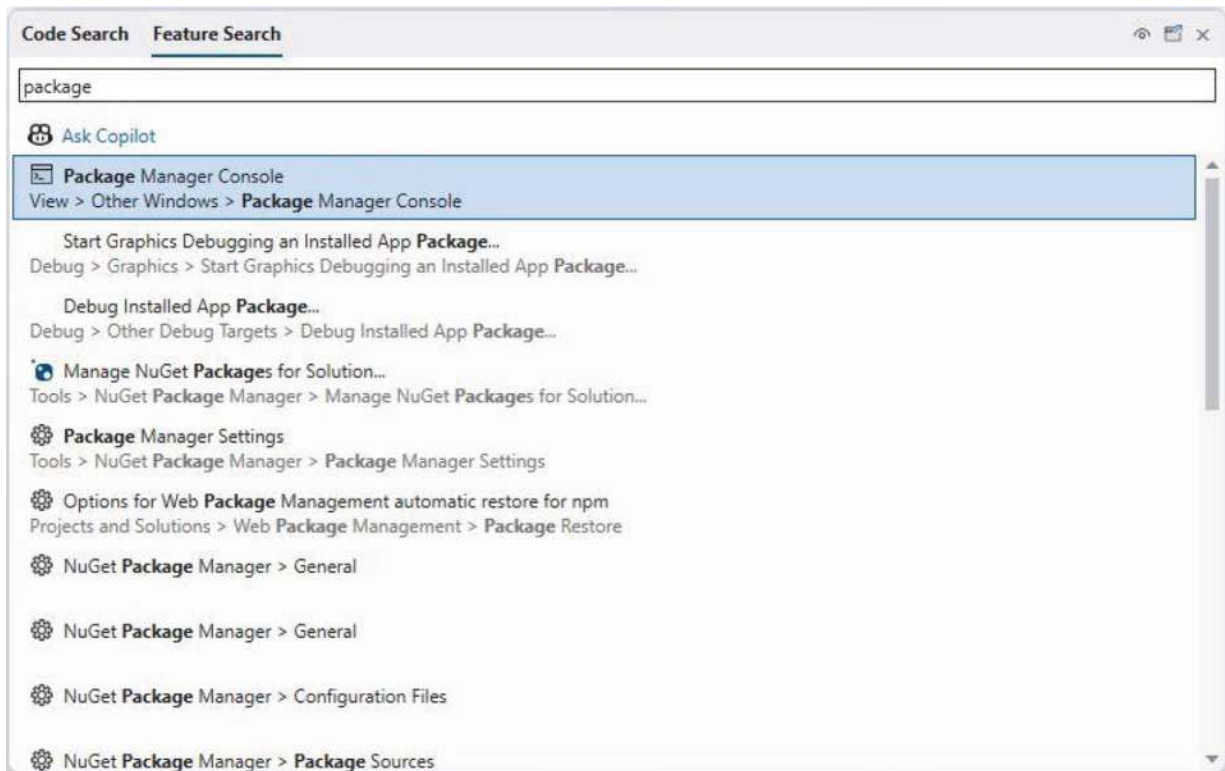
Visual Studio 2022 comes packed with a few very nice productivity features. A lot of thought has been put into making Visual Studio easy to navigate and to find things in Visual Studio 2022. The first feature I want to have a look at is Visual Studio Search.

### Visual Studio Search

I think that we can all agree that more speed equals improved productivity. The faster I can access a menu item, and the less time I have to spend looking for something, the more my productivity increases. This is where Visual Studio Search comes in.

If you hold down `Ctrl+Q`, you will jump to the search bar, where you can immediately start typing, as seen in Figure [1-55](#).





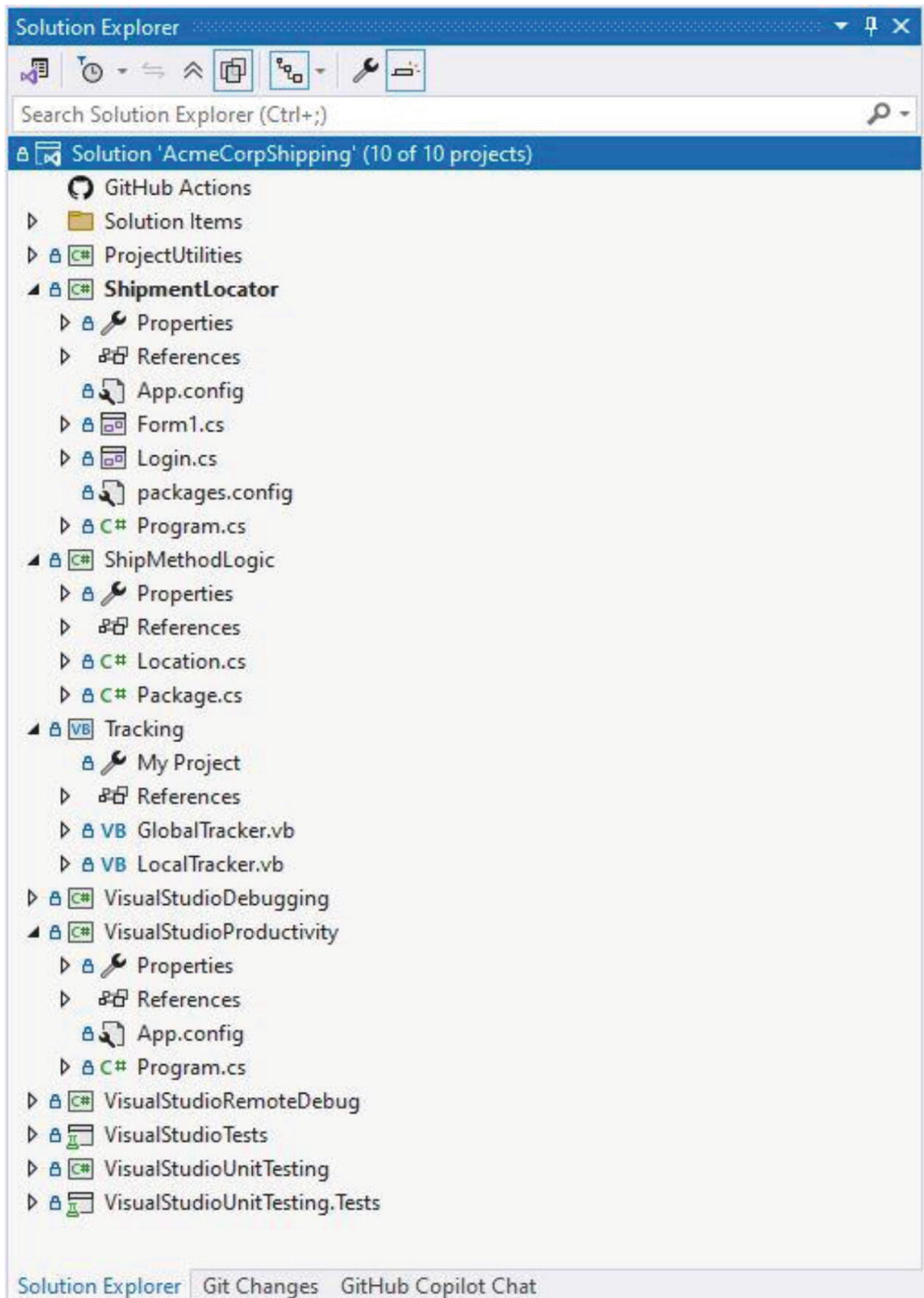
**Figure 1-55** Visual Studio Search

Visual Studio performs a fuzzy search, which means that even if you misspell a word, chances are that it will know what you intended to type and return the correct results to you. You can even click *Ask Copilot*, displayed as a link just under the search box. This will send your query to Copilot Chat, resulting in AI-powered search.

## Solution Filters

Sometimes, we have to work on Solutions that contain a lot of projects. More often than not, you as a developer will not be working on every project in that specific solution. This is where Solution Filters come in. They allow you to only load the projects that you care about or are actively working on. Consider the currently loaded solution as seen in Figure [1-56](#).

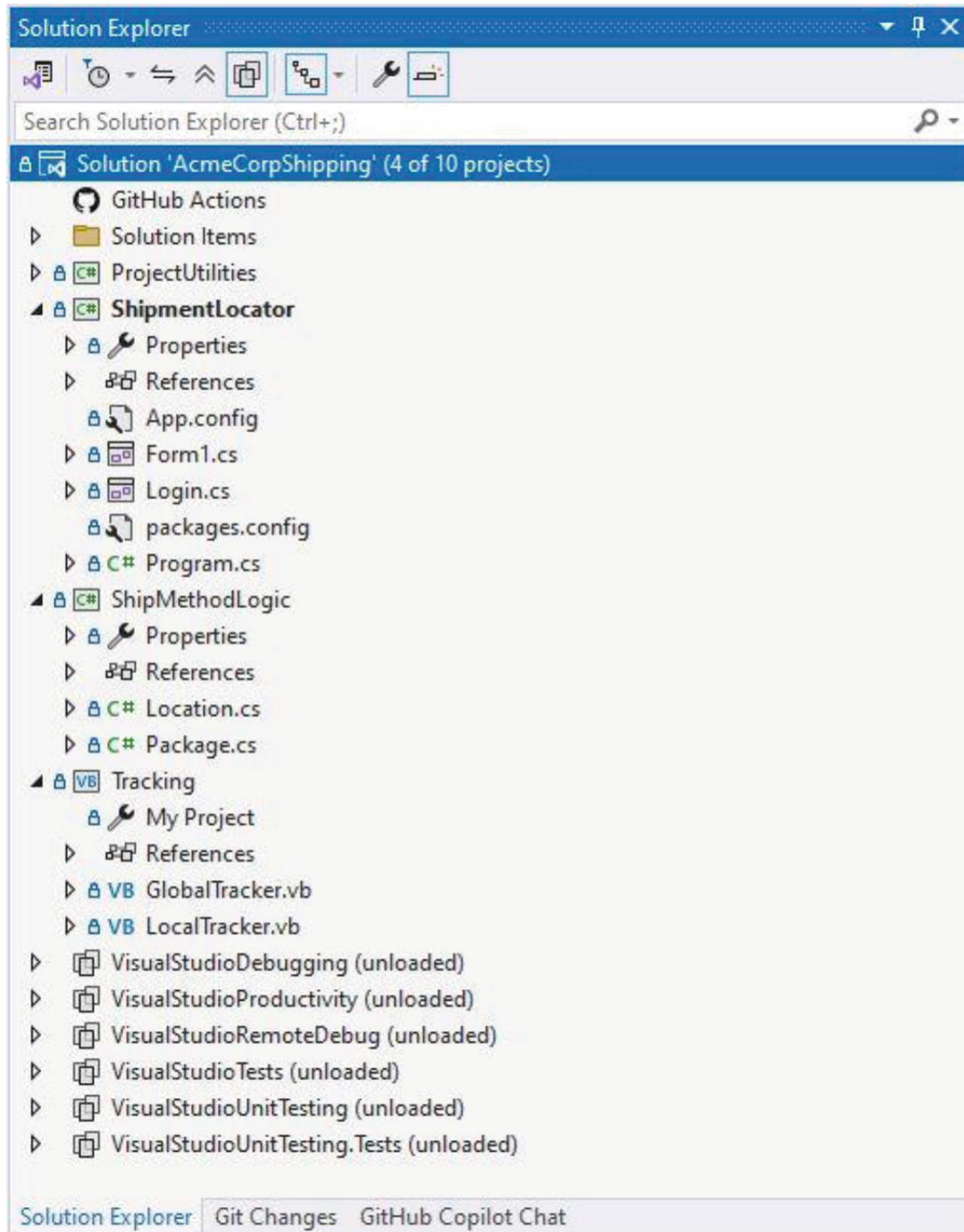




---

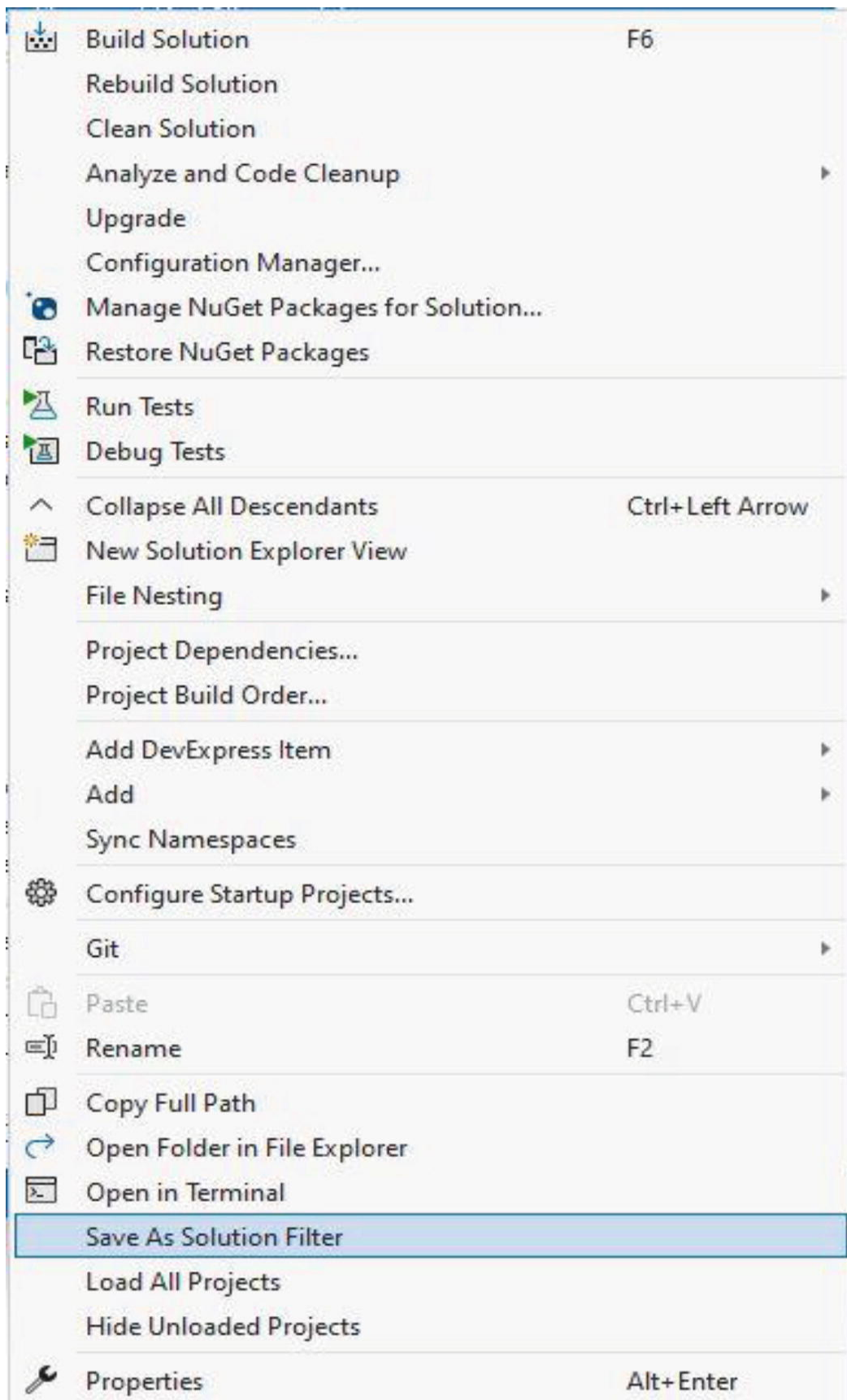
**Figure 1-56** AcmeCorpShipping Solution Unfiltered

You can see that all the projects are loaded in this solution. If we only work on the ShipmentLocator and ShipMethodLogic projects, we can create a Solution Filter to only load those projects. Right-click the projects that you don't work on, and click Unload Project from the context menu. Your solution will now look as in [Figure 1-57](#).



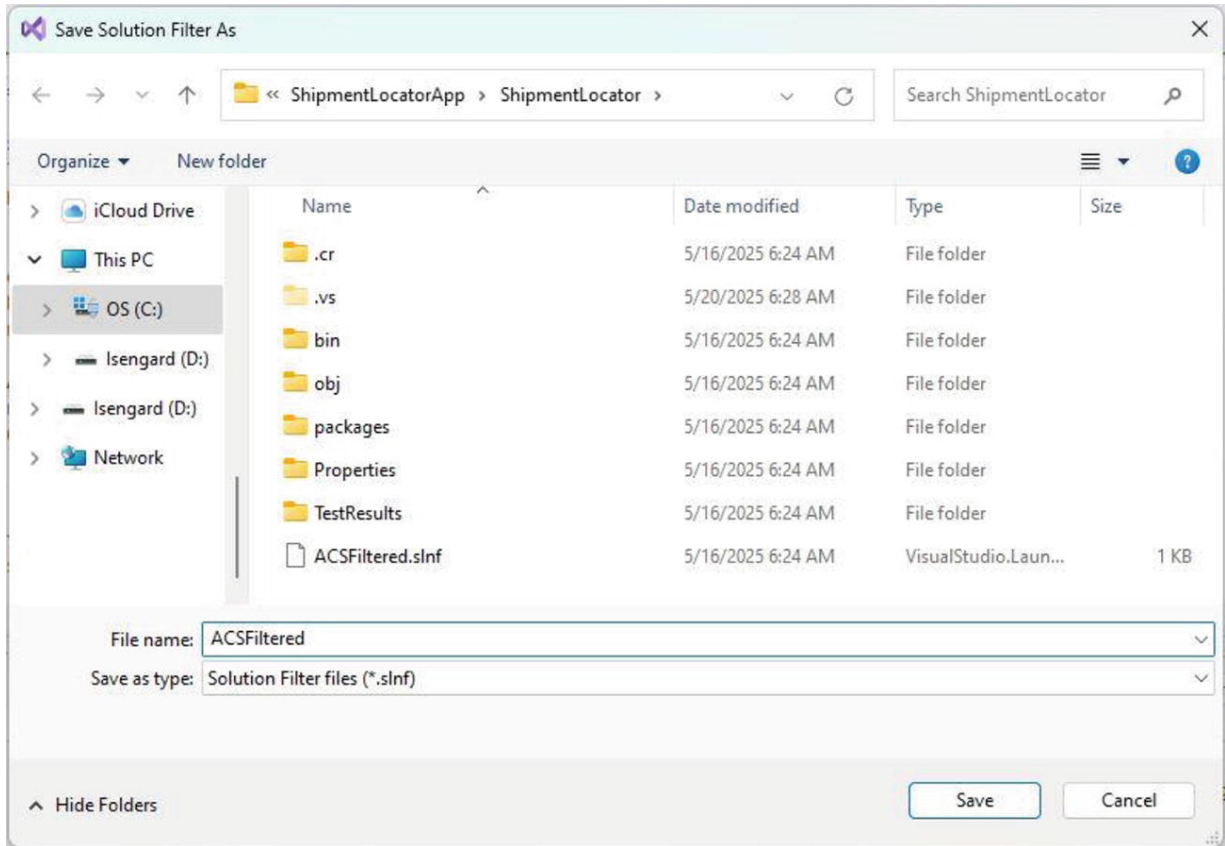
**Figure 1-57** AcmeCorpShipping Solution with Unloaded Projects

Now, with the projects unloaded that you do not work on, right-click the solution and select Save As Solution Filter, as seen in [Figure 1-58](#).



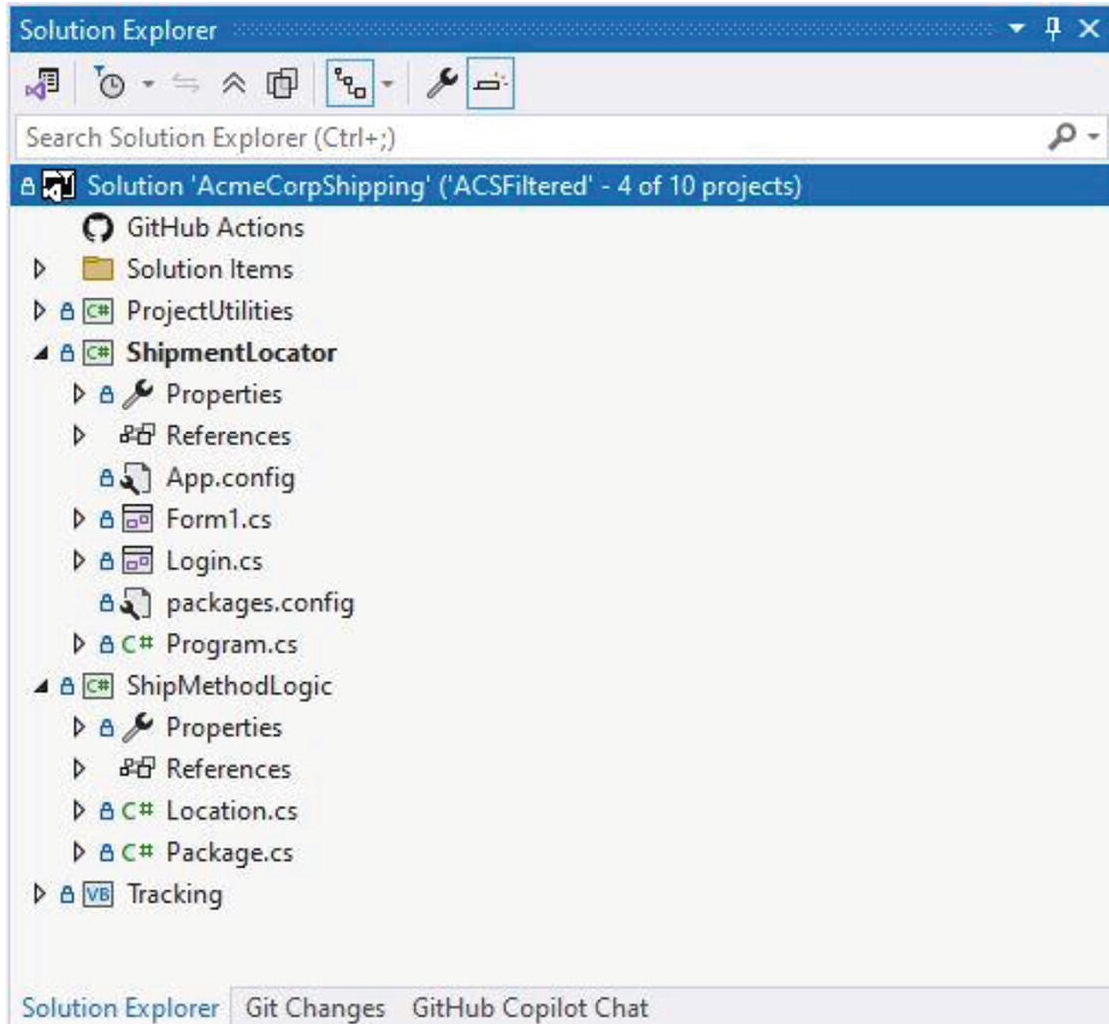
**Figure 1-58** Save As Solution Filter

Visual Studio will create a new type of solution file with an .slnf file extension as seen in Figure [1-59](#).



**Figure 1-59** Save as Solution Filter File

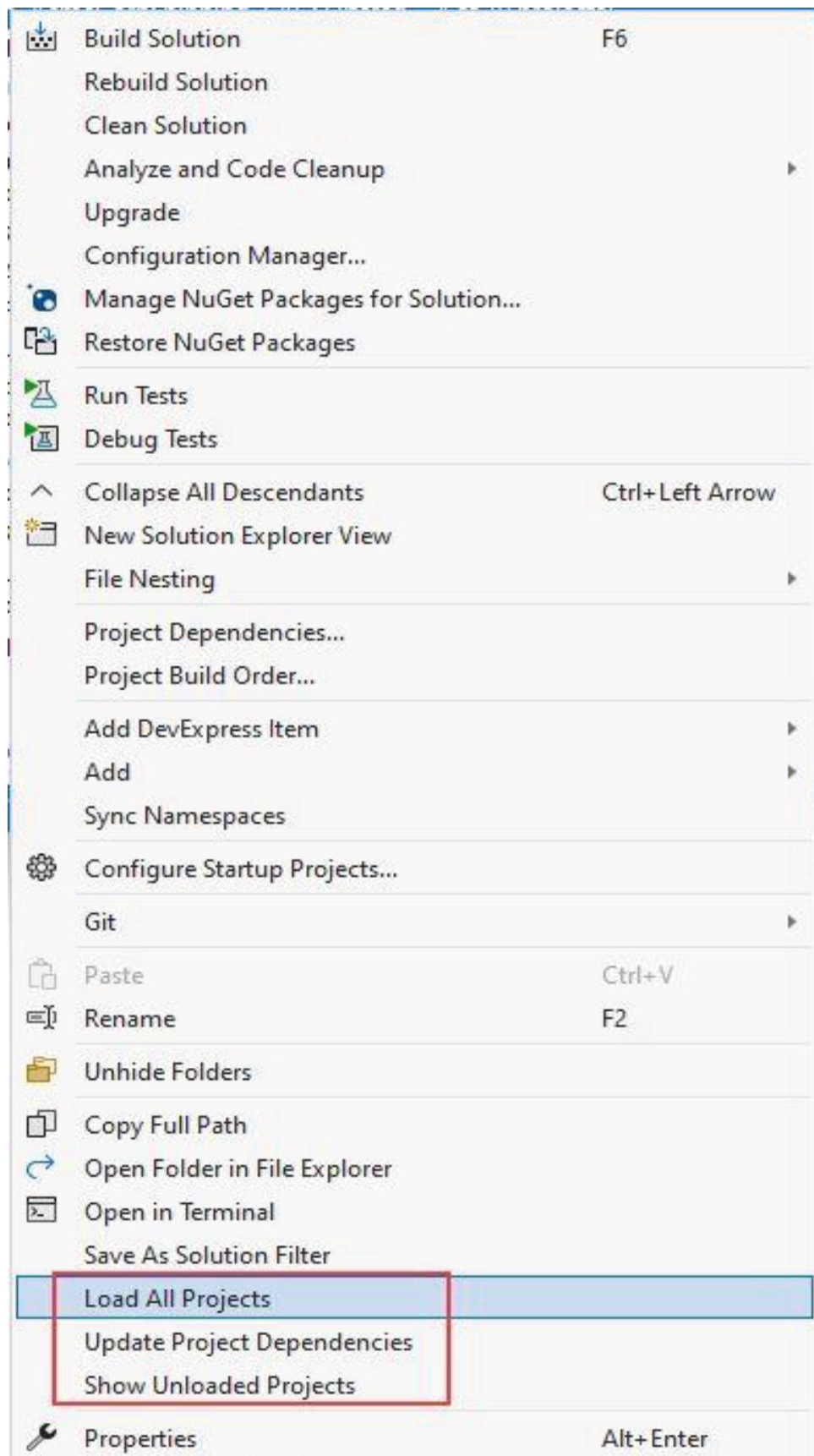
When you open your project using the Solution Filter, you will see, as in Figure [1-60](#), that only the projects you selected to keep will be loaded.



**Figure 1-60** Filtered Solution

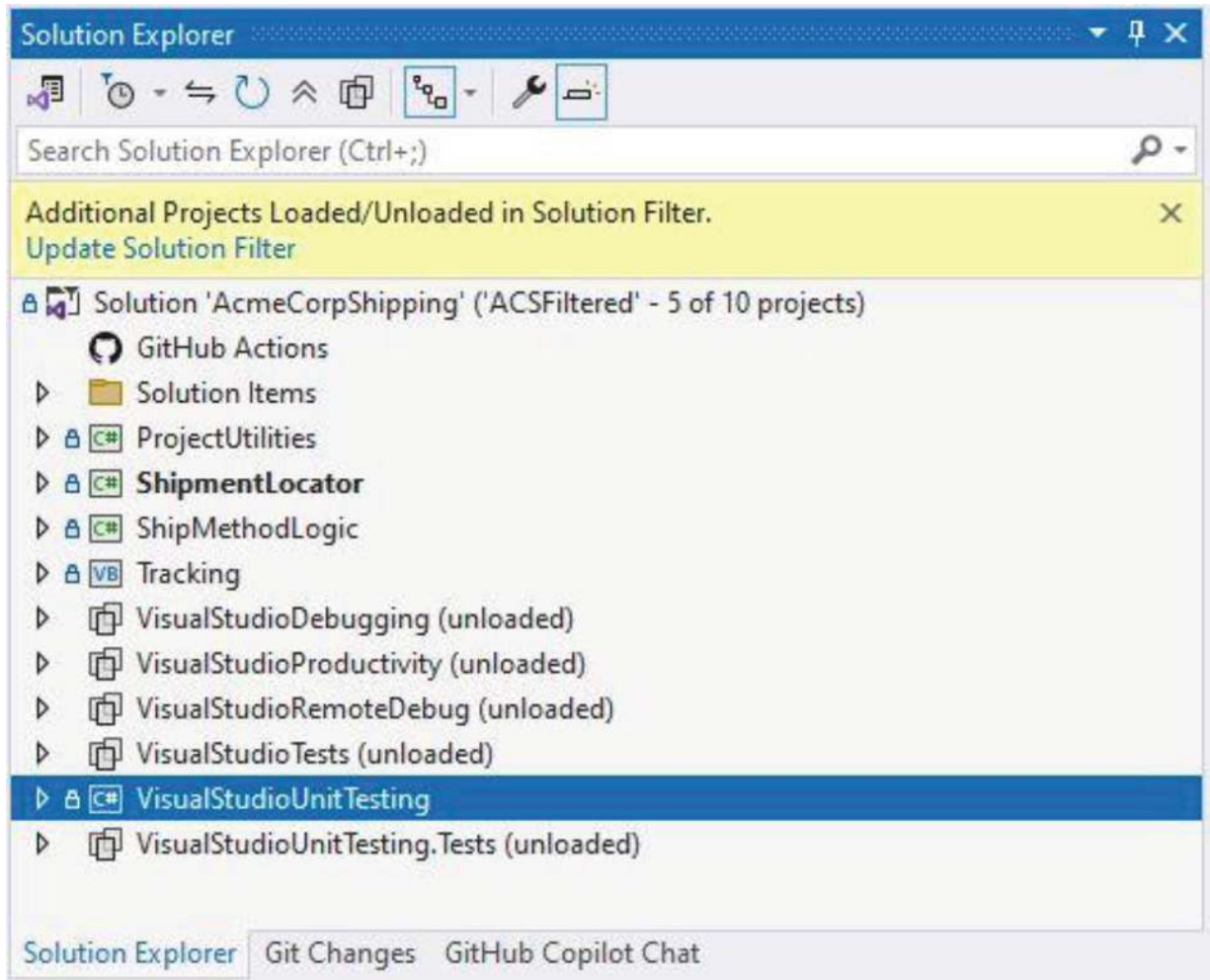
Now with the filtered solution, if you click the solution, you will see that you can Load App Projects, Show Unloaded Projects, or Load Project Dependencies from the context menu as seen in [Figure 1-61](#).





**Figure 1-61** Filtered Solution Context Menu

You still have full control of the filtered solution from the context menu and can easily load the full solution as needed.



**Figure 1-62** Updating the Solution Filter

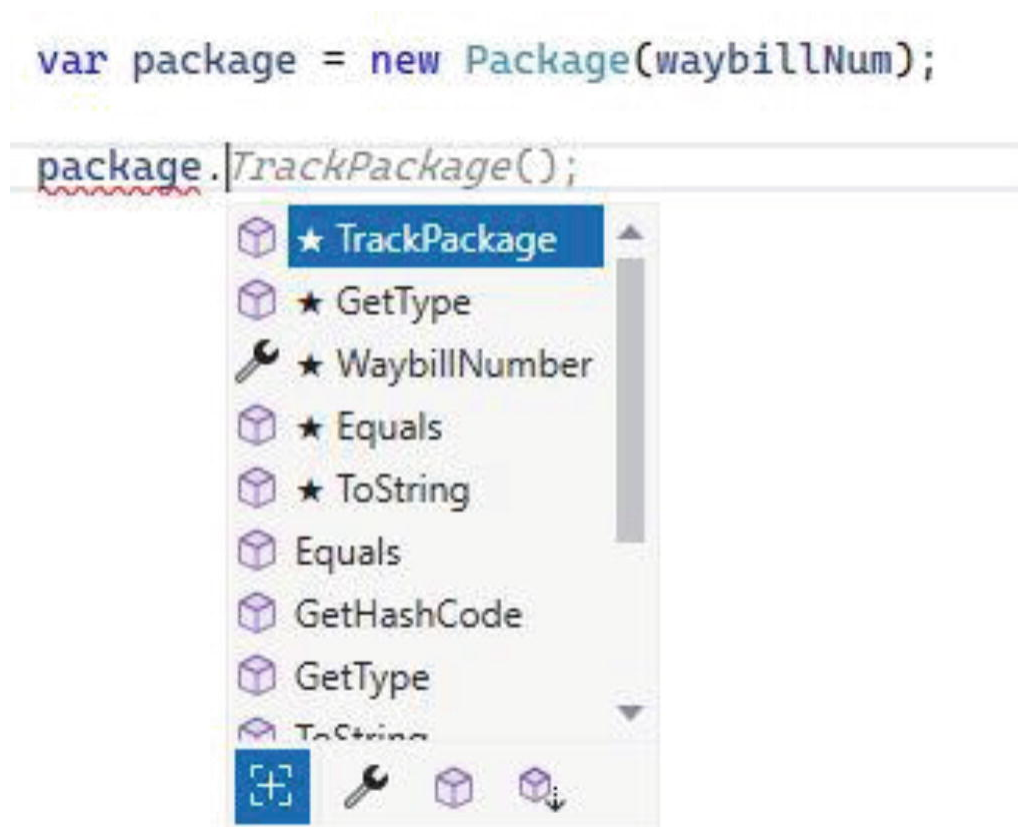
If you load additional solutions as shown in Figure [1-62](#), Visual Studio will notice this change and ask if you would like to update your solution filter. Managing solution filters is easily accomplished without having to go through the Save As Solution Filter process, as detailed in Figure [1-58](#).

## Visual Studio IntelliCode



Visual Studio IntelliCode is a really nice feature that has been added to Visual Studio. Microsoft calls it AI-assisted development because it uses machine learning to figure out what you are most likely to use next and put that suggestion at the top of your completion list. These are usually displayed as starred recommendations.

Without IntelliCode, when you dot on an object such as our `package` object in the `AcmeCorpShipping` solution, you will see the usual completion list that Visual Studio displays. With Intellicode enabled, you will see suggestions appear at the top of the completions list with a star icon next to them as shown in Figure 1-63.



**Figure 1-63** Visual Studio Completion List

Previously Visual Studio provided the ability to train a model on your code. This provided model completions for your team projects, libraries, and APIs using a simple Markov-chain model to rank methods in the IntelliSense list. As of Visual Studio 2022 (version 17.0+), this is no longer available. Instead, custom methods are ranked with the help

of a neural encoder model. This means that IntelliCode uses an on-device deep-learning network that sits on top of the static analyzer's candidates. The benefits are as follows:

- It works everywhere, even on code you've never shared or that isn't part of the public training corpus.
- Code-aware completions, tuned to your project without any manual training.
- Total privacy because everything runs locally. Your source code never leaves your machine.
- The slimmed-down model design leads to a reduced memory footprint and improved inference speed.

Behind the scenes, the model continually refines its ranking by looking at:

- Language and dependencies – it understands the syntax, frameworks, and packages you're using.
- Your recent edits – it learns from your coding patterns as you work, so suggestions are current with your style.

By embedding a continually adapting deep learning model into Visual Studio, IntelliCode turns IntelliSense into a proactive coding partner.

## Visual Studio Live Share

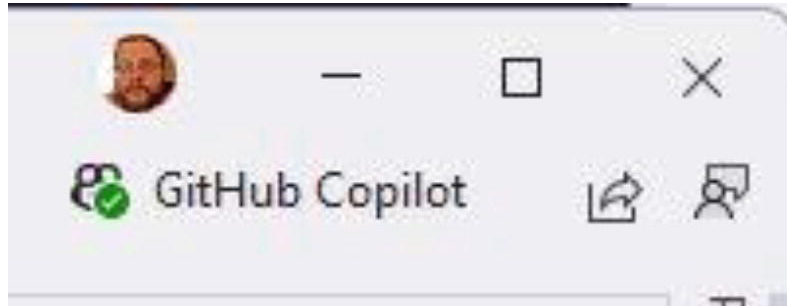
During my years of writing code, I have often needed to explain some portion of logic or a feature of the code I am working with to another developer. This usually involves them having to get a copy of the code base from source control and us having to direct each other over a Teams call, quoting line numbers to collaborate effectively. No longer with Visual Studio Live Share.

To find out more about Visual Studio Live Share, go to <https://visualstudio.microsoft.com/services/live-share/>.

Visual Studio Live Share is included by default in Visual Studio 2022. Visual Studio Live Share does not require developers to be all “set

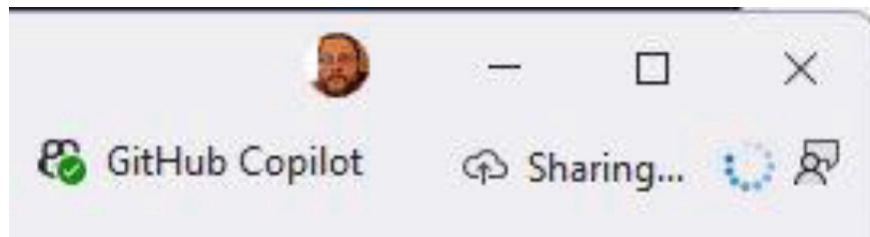
up” to assist each other or to collaborate on projects. This means that a developer running Visual Studio Code on a Linux machine can collaborate with another developer running Visual Studio 2022 on a Windows 11 machine.

To start a Visual Studio Live Share session, you need to click the Live Share icon in the top-right corner of Visual Studio 2022 next to the GitHub Copilot button as can be seen in Figure [1-64](#).



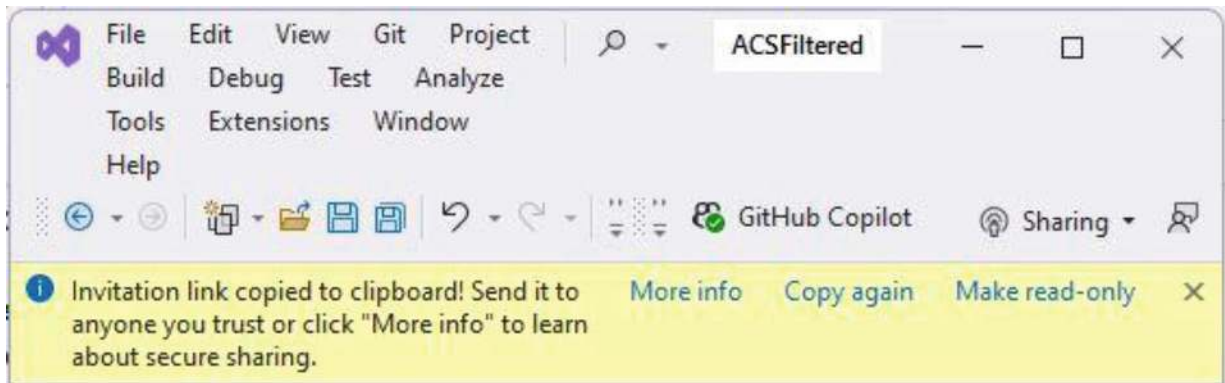
**Figure 1-64** Visual Studio Live Share Icon

When you click the icon, Visual Studio starts up Live Share, and the progress is indicated as shown in Figure [1-65](#).



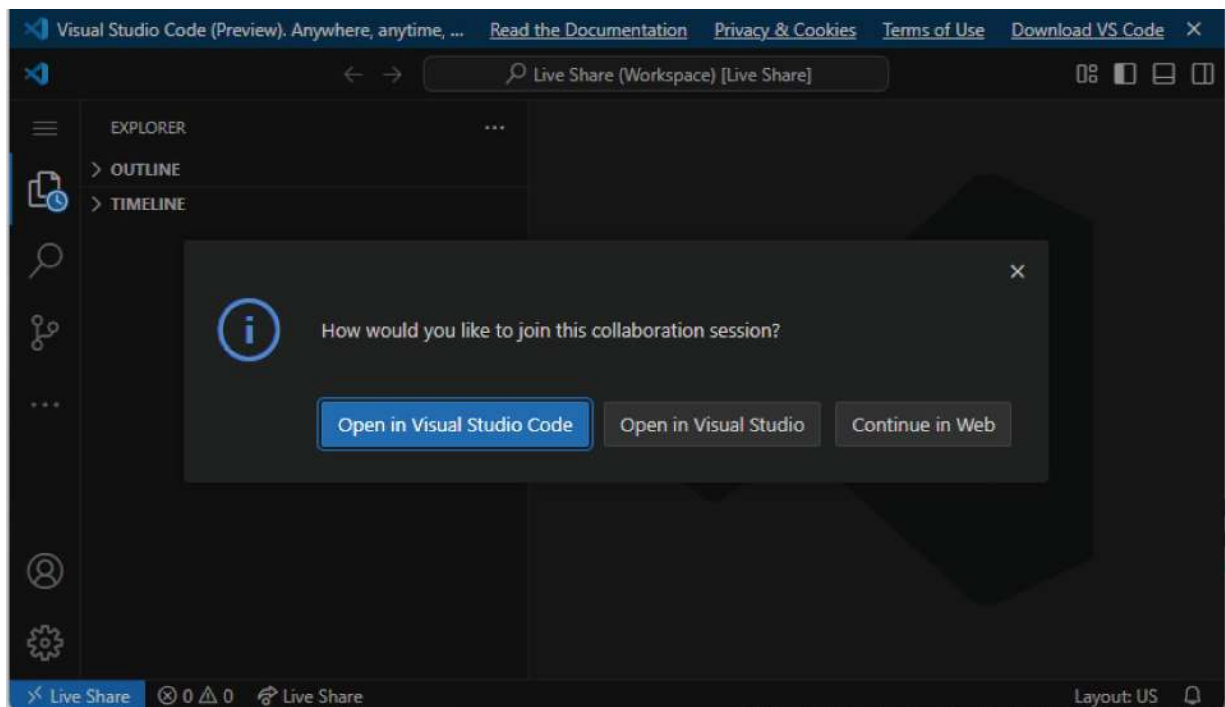
**Figure 1-65** Visual Studio Live Share in Progress

When the sharing link has been generated, Visual Studio will display a notification (Figure [1-66](#)).



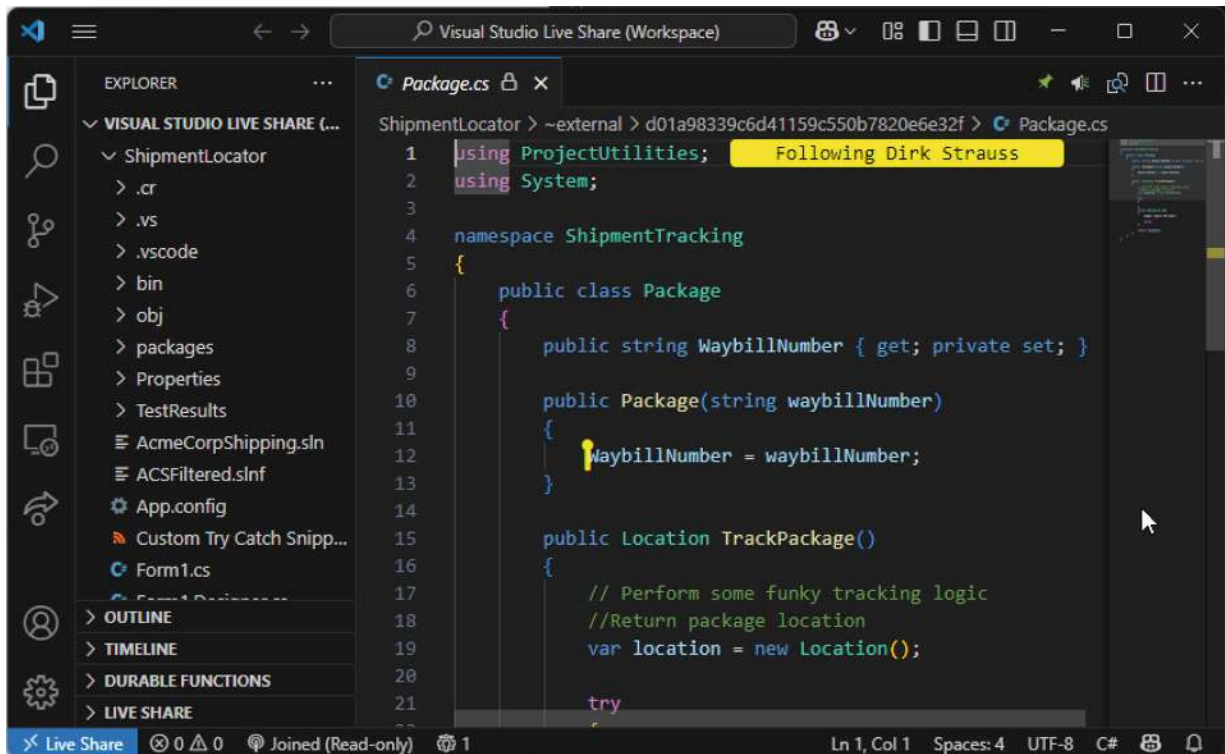
**Figure 1-66** Visual Studio Live Share Link Generated

It is copied to the clipboard by default, but you can copy it again, make it read-only, or learn more about secure sharing. When you share the link with a colleague, they simply have to paste the link into a browser to start the collaboration. Figure 1-67 shows the browser as seen by your colleague after pasting the link you shared with them. They have the option of opening Visual Studio or Visual Studio Code.



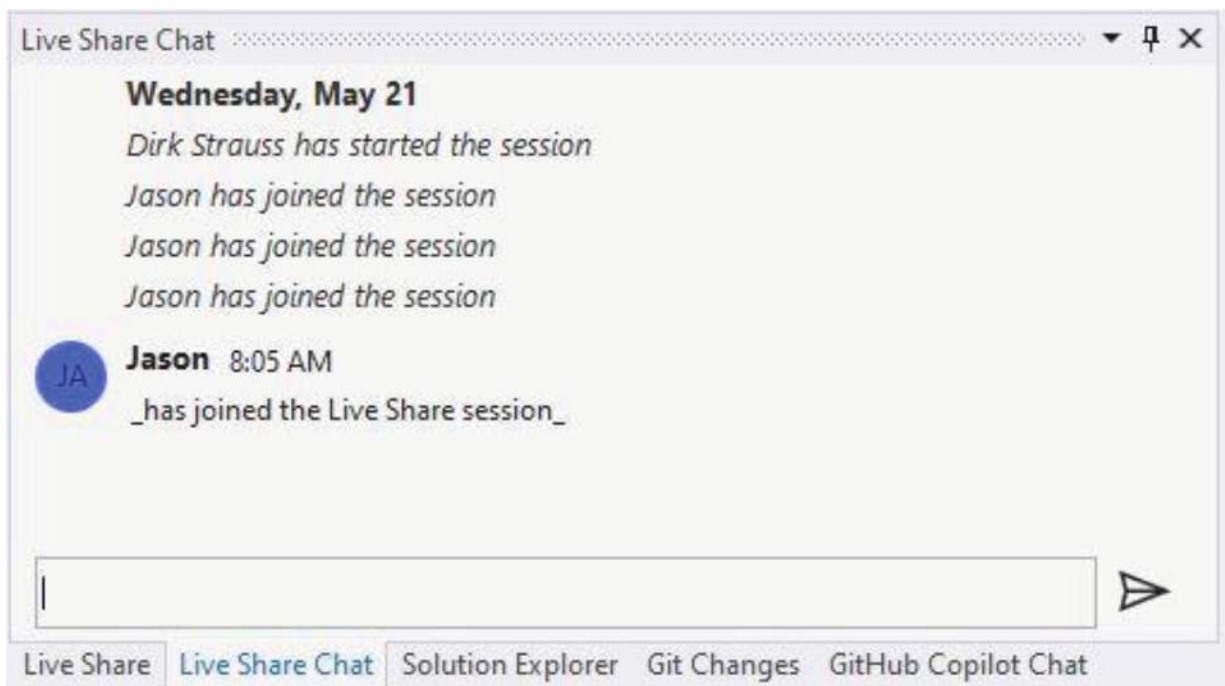
**Figure 1-67** Starting Visual Studio Live Share Session

In this example, I am sharing the link with a developer that is running Visual Studio Code on Linux Mint.



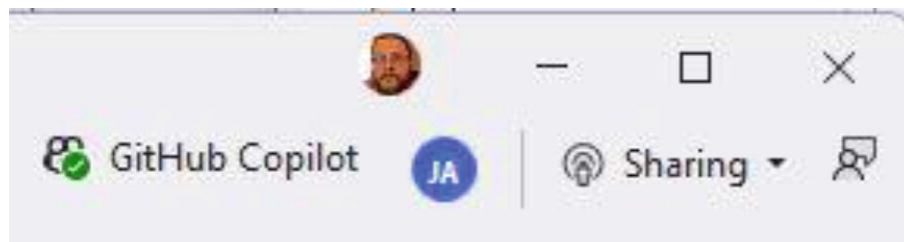
**Figure 1-68** Visual Studio Code

When Visual Studio Code is opened, my colleague will see that I am currently active in the codebase as shown in Figure [1-68](#).



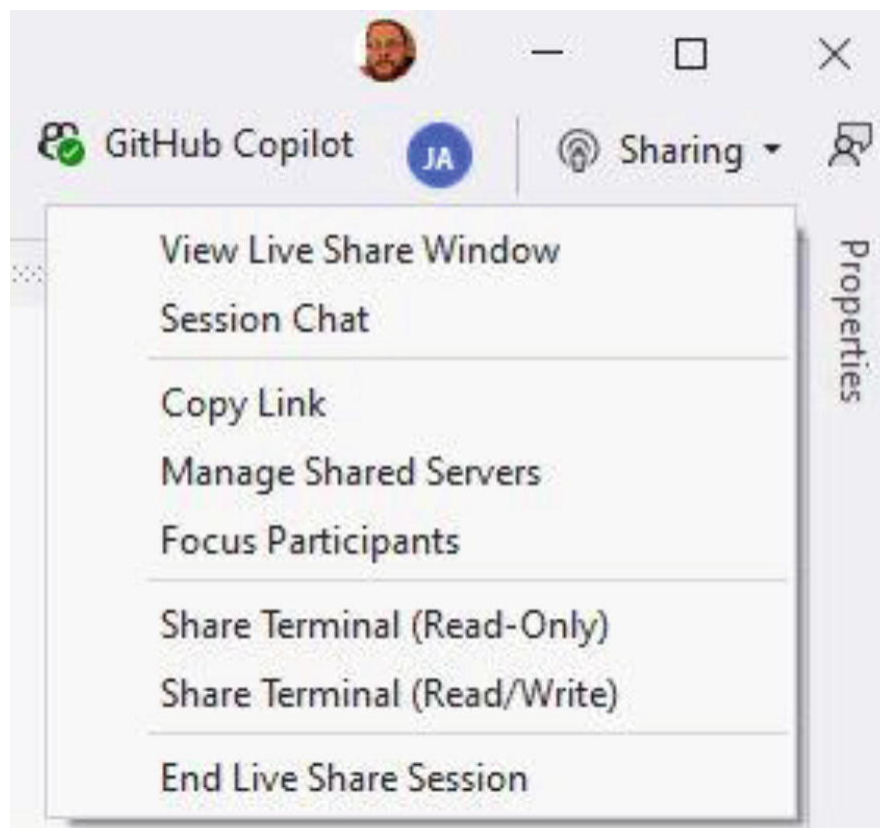
**Figure 1-69** Live Share Chat in Visual Studio

On my machine, I can see that my colleague has joined the session (Figure [1-69](#)) and I can chat with him.



**Figure 1-70** Visual Studio Sharing Indicator

The icon in Visual Studio also changes to show that I am currently in a Live Share session with someone (Figure [1-70](#)). I can click on this icon to access more options, as seen in Figure [1-71](#).



**Figure 1-71** Live Share Options



From there, I can end the Live Share session, share the terminal, manage shared servers, focus participants, or copy the sharing link again. At any time, I am in total control of what I share. It is also important to note that my code lives on my machine. It is not saved on the participant's machine.

## **Copy Files Across Visual Studio Instances**

In the past, transferring code files between Visual Studio instances has been a tedious task. Developers often had to manually relocate files and in doing so, risk introducing errors. Now that toil is a thing of the past with the new copy and paste functionality. The process is, well, as you would expect.

- Select the file or folder from the Solution Explorer of one Visual Studio instance.
- Use Ctrl+C or Ctrl+X to copy or cut the file or folder.
- Switch to another instance of Visual Studio where you want the files.
- Use Ctrl+V to paste the copied or cut file or folder.

The changes will be reflected in the second instance of Visual Studio's Solution Explorer.

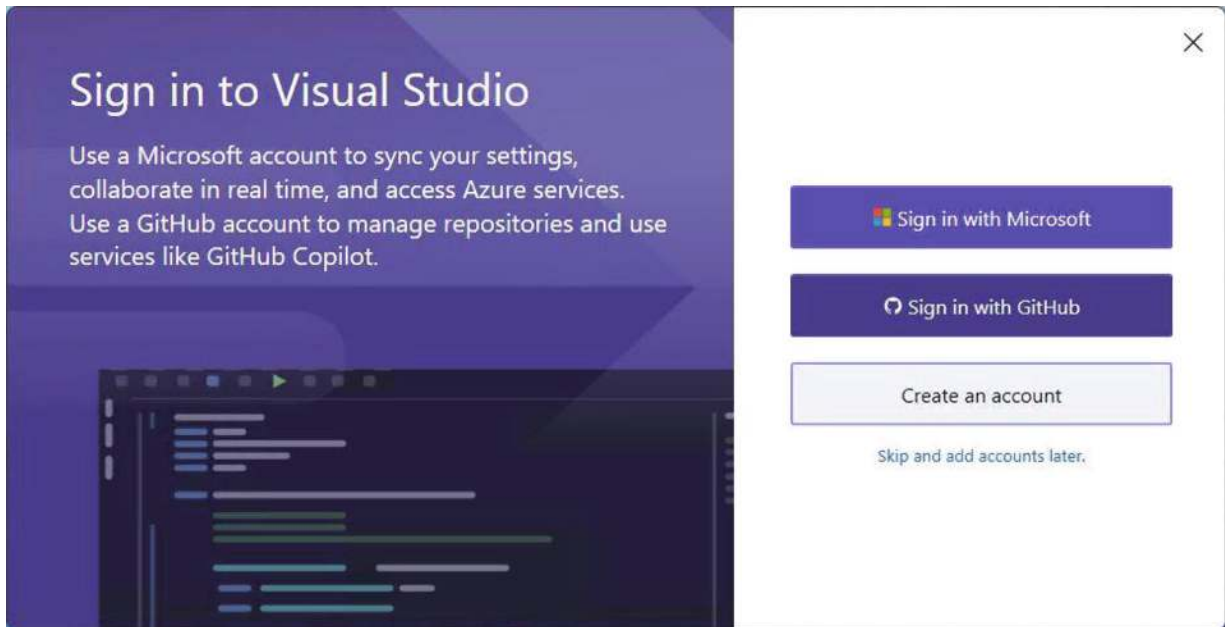
Developers can also drag and drop files and folders between instances, giving them even more flexibility in managing their projects. This feature is available in all major project types in Visual Studio.

---

## **Adding and Using GitHub Copilot**

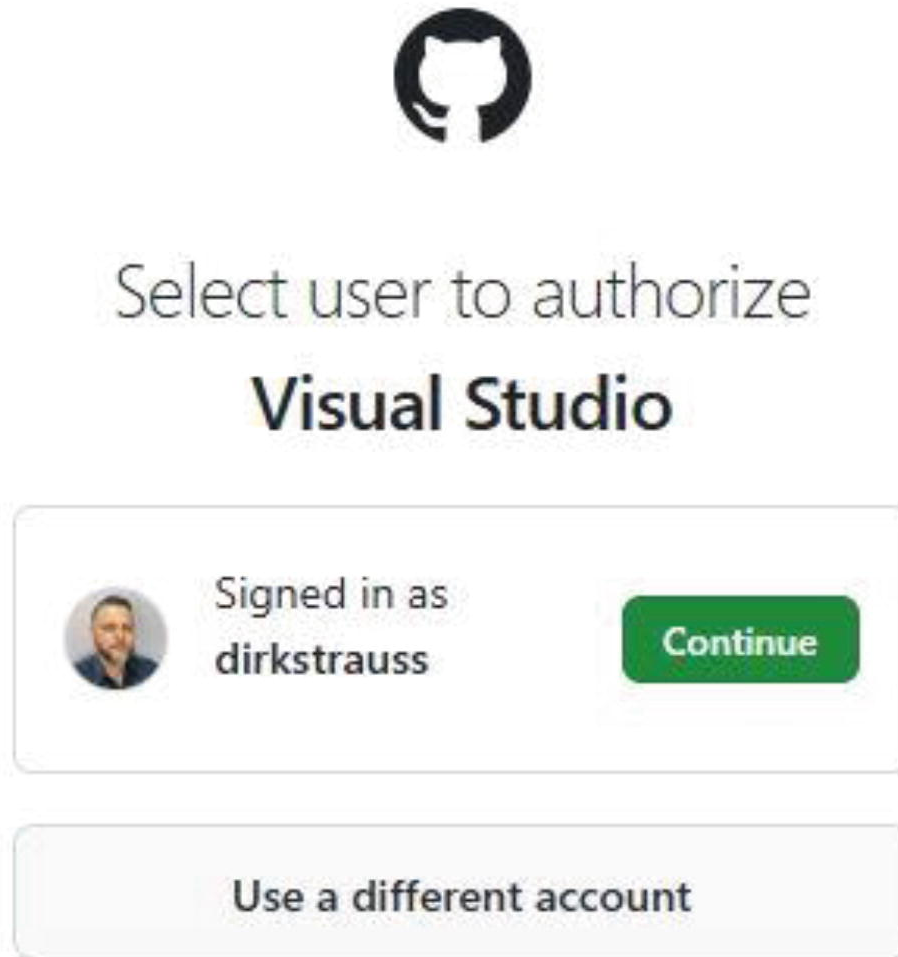
Powering your Visual Studio experience using AI is easier than you think with GitHub Copilot. To get started, make sure that you are using the latest version of Visual Studio 2022. You can then start using GitHub Copilot for free with limited access to select Copilot features. No trial or credit card is required; you only need your GitHub account.





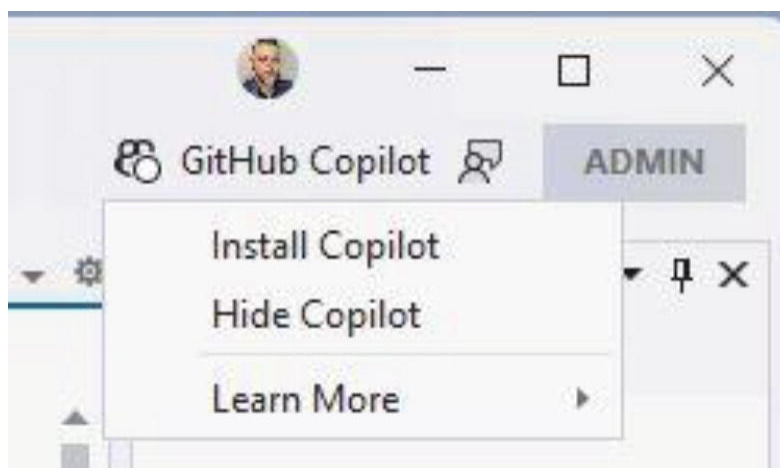
**Figure 1-72** Sign in to Visual Studio

After installation, you will see the sign-in options (Figure [1-72](#)) for Visual Studio pop up. If you have an existing installation of Visual Studio, ensure that you are on the latest version and add your GitHub account from the accounts selector (top right in Visual Studio).



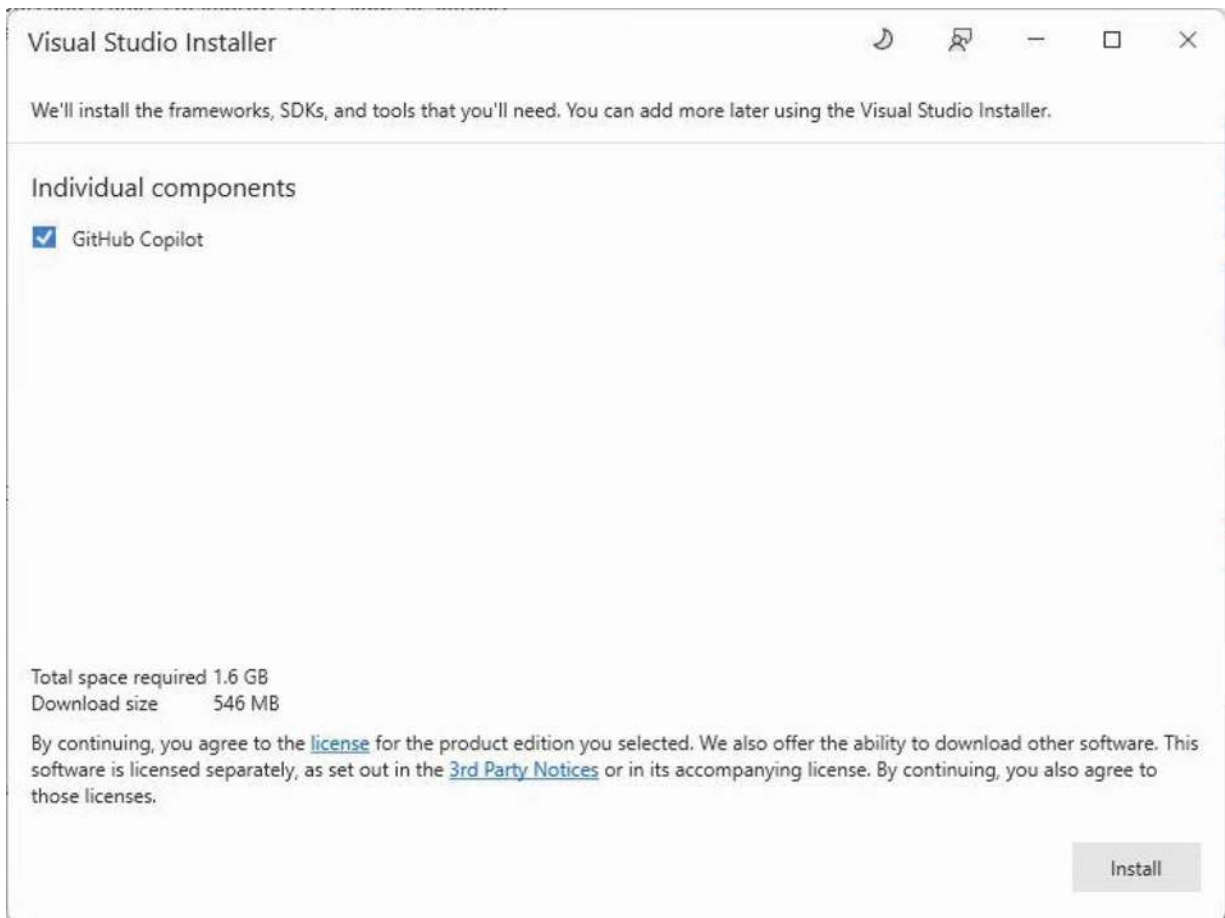
**Figure 1-73** Select User to Authorize

You will be redirected to GitHub to select a user to authorize with Visual Studio, as seen in [Figure 1-73](#).



**Figure 1-74** Install Copilot

Once you have signed in with your Visual Studio account, you can click on the GitHub Copilot button to install Copilot as seen in Figure [1-74](#).



**Figure 1-75** Visual Studio Installer

This will display the Visual Studio Installer with GitHub Copilot selected as a component to install. You can also run the Visual Studio Installer and select to install GitHub Copilot from the Individual Components tab under the AI Assistance section.

It is important to note that GitHub Copilot is merely an assistant. It makes mistakes and it is up to you as developer to verify the accuracy of the output.

After installing GitHub Copilot you will see the Copilot Chat tab as shown in Figure [1-76](#).



# Copilot Chat

I might make mistakes, so check for accuracy.



## Help me update VS

How can I find and install updates for Visual Studio?



## Explain VS functionality

What are the most common keyboard shortcuts for efficient coding in Visual Studio?



## Help me create new projects

How do I create a new project? Give an overview of the available templates in VS

Start with @ to use an extension

Type # or use 📎 to attach context

[/help what can you do?](#)



📄 Active Document ✕

Ask Copilot or use @workspace



GPT-4.1 ▾



**Figure 1-76** Copilot Chat

Right off the bat there are a few things to take note of here. In the chat window you can type @ to reference a workspace, type #, or attach a file (such as an image) by clicking on the + button to give Copilot more context on what you are asking. You can also type / to use one of the many commands at your disposal.

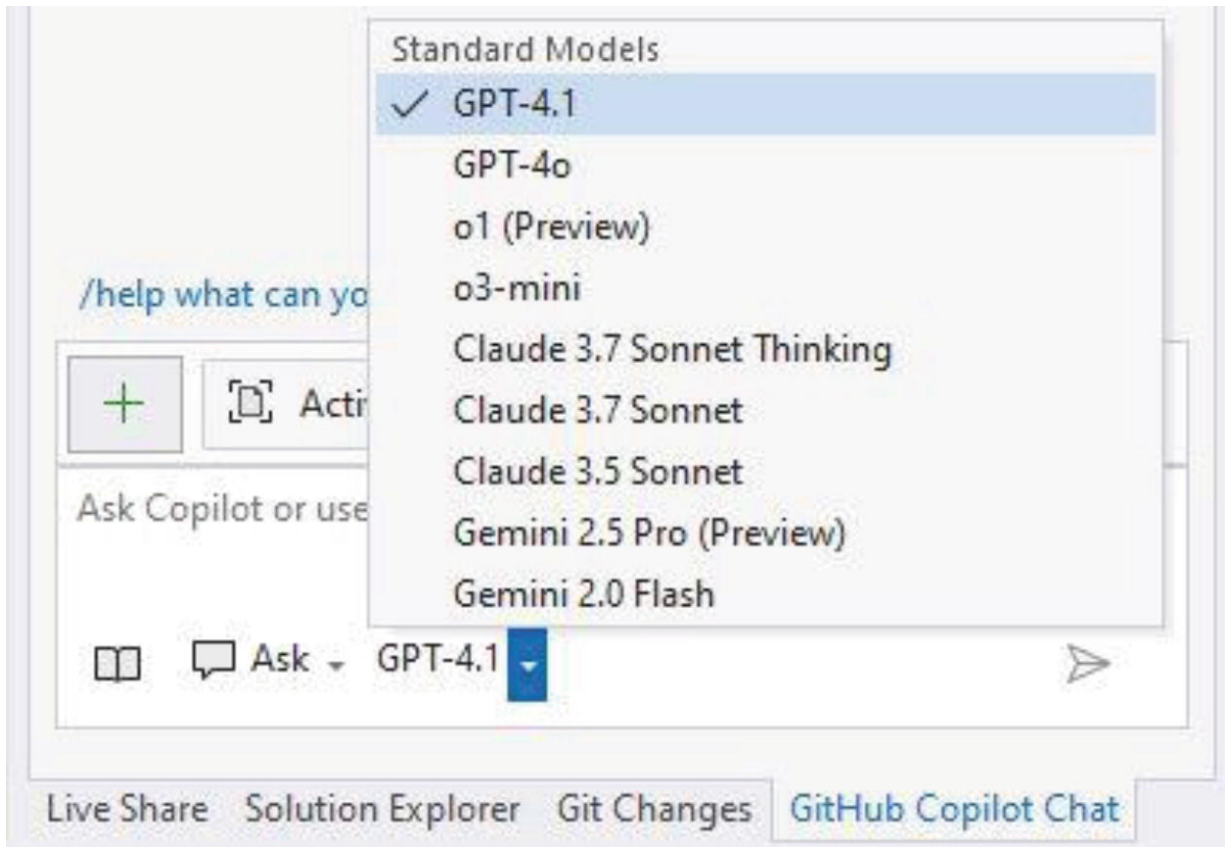
## GitHub Settings

On [github.com](https://github.com), navigate to the settings under your profile icon in the top right. Once there, under the Code, planning, and automation section, look for Copilot and click on the Features menu. Here you can find various settings associated with GitHub Copilot.



**Figure 1-77** GitHub Copilot Settings

As seen in Figure [1-77](#), for example, I have selected to enable Claude 3.5 and 3.7 Sonnet in Copilot.



**Figure 1-78** GitHub Copilot Model Selection

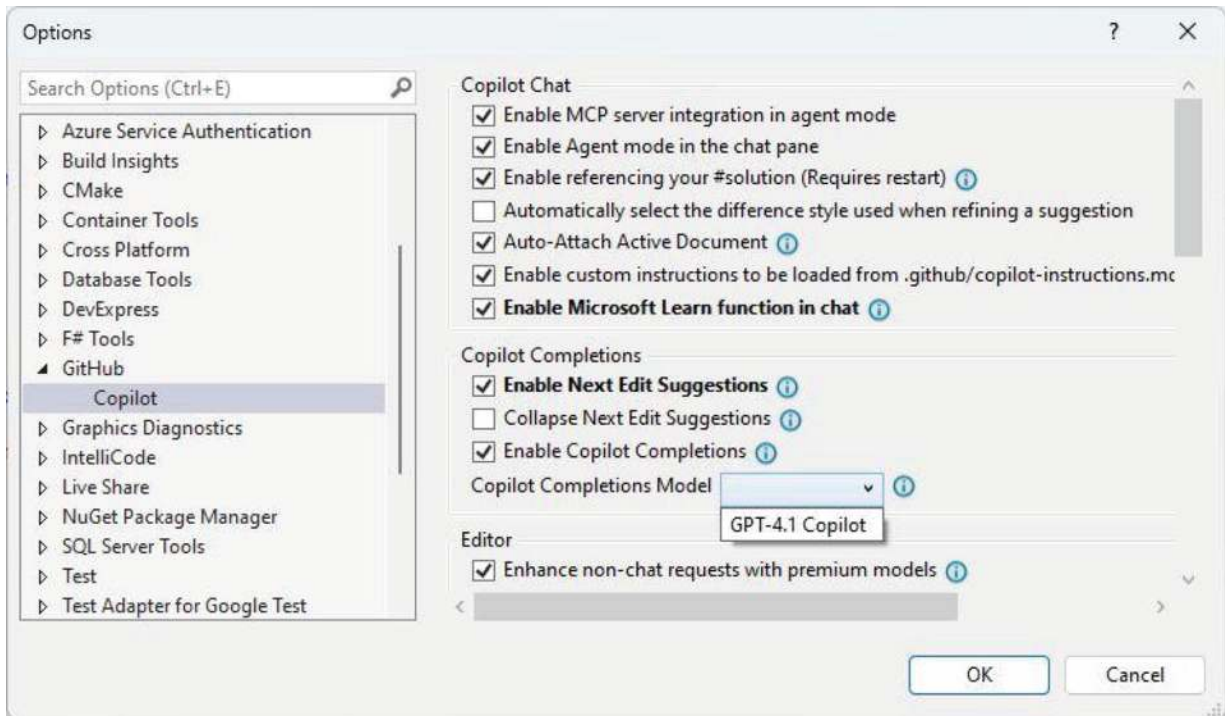
In Visual Studio on the GitHub Copilot Chat Tab (Figure [1-78](#)), you will notice that these models are available for selection.

## **GPT-4.1 Code Completion Model**

The GPT-4.1 code completion model is now available for GitHub Copilot, starting in Visual Studio 17.14. This model is based on GPT-4o mini and has undergone additional training on over 275,000 public repos across more than 30 programming languages.

This means that developers can expect more accurate code suggestions and better performance. To select the code completions model, in Visual Studio, go to Tools ► Options ► GitHub ► Copilot, as seen in Figure [1-79](#).



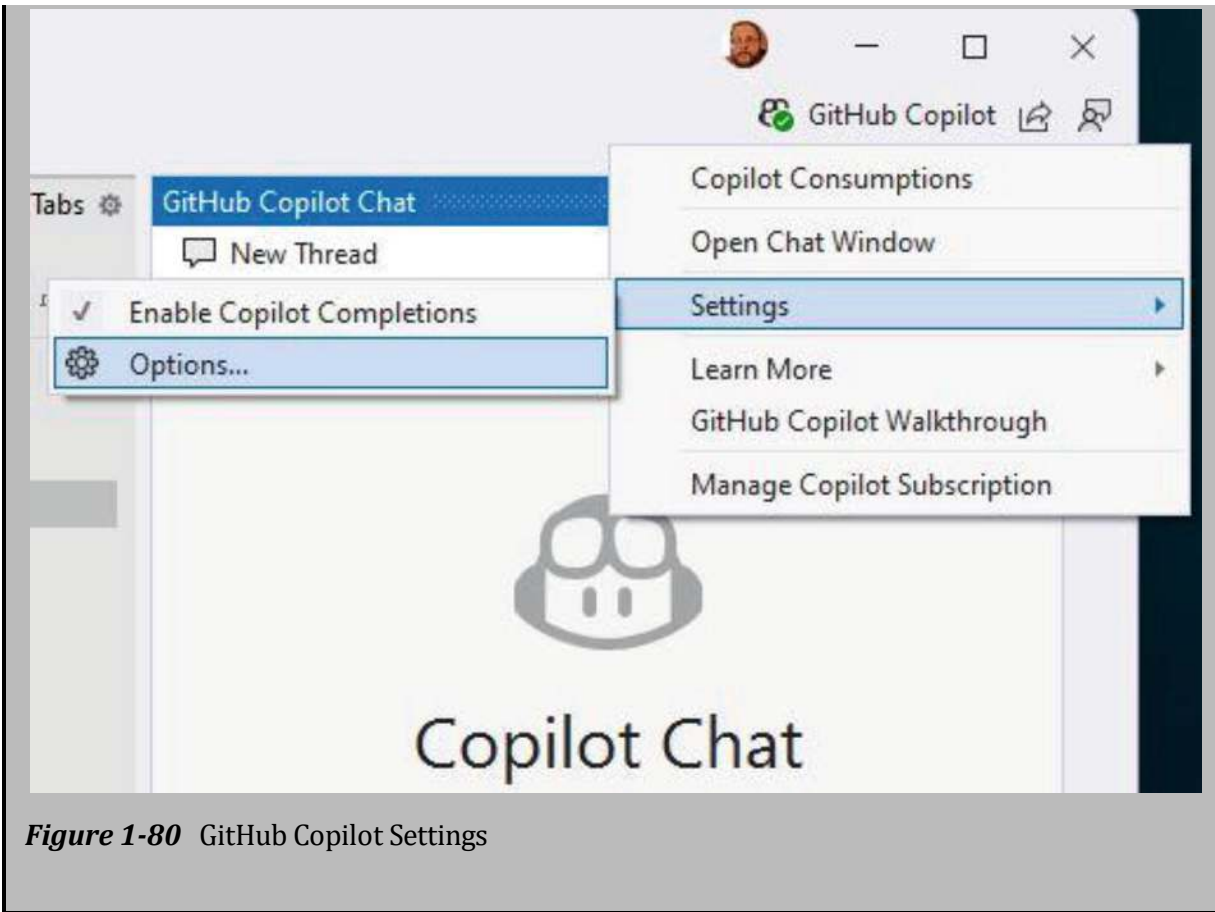


**Figure 1-79** Copilot Completions Model

Under the Copilot Completions section, check the option to enable Copilot Completions and select the Completions model from the dropdown selector.

Using this completion model will further enhance your productivity as a developer, providing higher quality code completions.

New models are often released. GPT-5 was released after the publication of this chapter. As a result, it is now available from the Model Selector in Copilot. You can now also manage your models from the Visual Studio model selector in Copilot. This displays a screen to choose a provider (OpenAI, Google, or Anthropic) and to enter your own API key. Essentially, allowing you to “Bring Your Own Model” into GitHub Copilot.



**Figure 1-80** GitHub Copilot Settings

You can also access the settings by clicking on the GitHub Copilot badge on the top right corner of Visual Studio and going to Settings ► Options, as seen in Figure [1-80](#).

## Code Completions

With Copilot installed and active, you will notice ghost text appear in your editor as you type.



**Figure 1-81** Ghost Text Code Suggestion

As seen in Figure [1-81](#), Copilot suggests what it thinks your next line of code or code completion should be. You can hot tab to accept this or click on the arrow to accept or dismiss the code suggestion.



**Figure 1-82** Code Completion from Comment

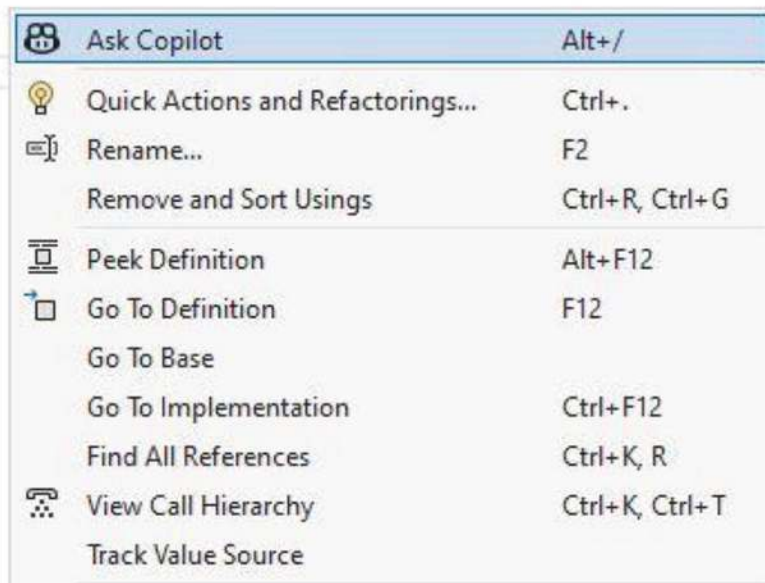
Another feature of Copilot's code completion suggestion is that when you type a comment in your code, it will use this as context to suggest the code you need to add next. As seen in Figure [1-82](#), I have added a comment that I want to add anti-forgery middleware. When I hit enter to add a new line, Copilot suggests the code to accomplish this.

## Inline Chat

GitHub Copilot allows you to chat with it in Visual Studio, displaying answers in the code editor window itself.

```
app.MapGet("/weatherforecast", (HttpContext httpContext) =>
{
    var forecast = Enumerable.Range(1, 5).Select(index =>
        new WeatherForecast
        {
            Date = DateOnly.FromDateTime(DateTime.Now.AddDays(index)),
            TemperatureC = Random.Shared.Next(-20, 55),
            Summary = summaries[Random.Shared.Next(summaries.Length)]
        })
        .ToArray();
    return forecast;
});

app.Run();
```



**Figure 1-83** Ask Copilot

If you right-click in the code editor, you will notice that Ask Copilot is available, as seen in Figure [1-83](#). You can also type Alt+/, to invoke it.



**Figure 1-84** Chatting with Copilot

After invoking Copilot, you will see a chat window displayed in your code editor, as seen in Figure [1-84](#). Here you can also select the model

to use to chat with.

```
app.MapGet("/weatherforecast", (HttpContext httpContext) =>
{
    var forecast = Enumerable.Range(1, 5).Select(index =>
        new WeatherForecast
        {
            Date = DateOnly.FromDateTime(DateTime.Now.AddDays(index)),
            TemperatureC = Random.Shared.Next(-20, 55),
            Summary = summaries[Random.Shared.Next(summaries.Length)]
        })
        .ToArray();
    return forecast;
});

app.Run();
}
```

How do I skip the anti-forgery check on my /weatherforecast endpoint?

GPT-4.1



**Figure 1-85** Asking Copilot a Question

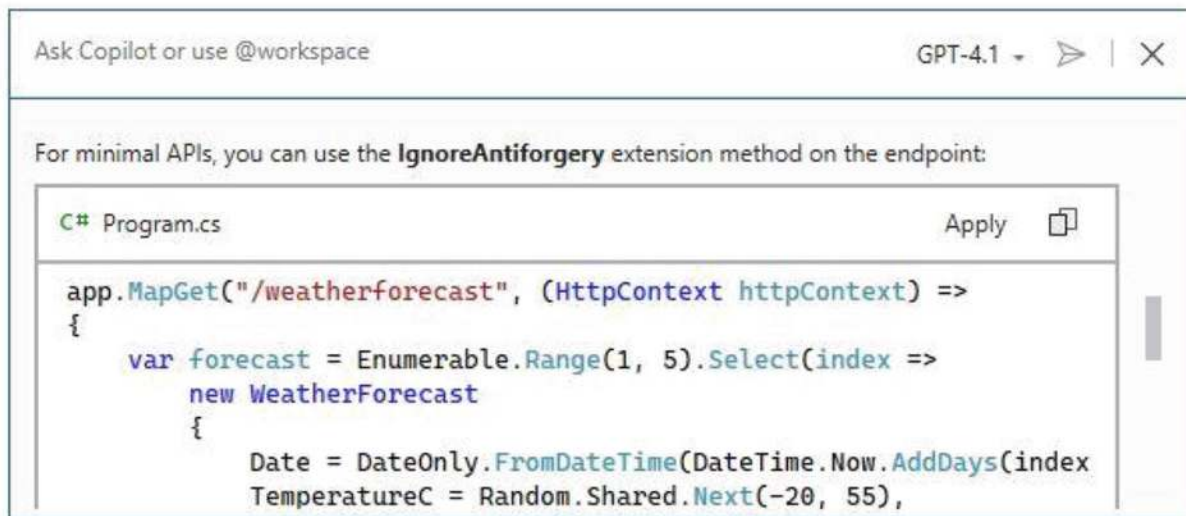
To ask Copilot a question, simply type it in the chat window and click on the send button, as seen in [Figure 1-85](#).

```

app.MapGet("/weatherforecast", (HttpContext httpContext) =>
{
    var forecast = Enumerable.Range(1, 5).Select(index =>
        new WeatherForecast
        {
            Date = DateOnly.FromDateTime(DateTime.Now.AddDays(index)),
            TemperatureC = Random.Shared.Next(-20, 55),
            Summary = summaries[Random.Shared.Next(summaries.Length)]
        })
        .ToArray();
    return forecast;
});

app.Run();
}

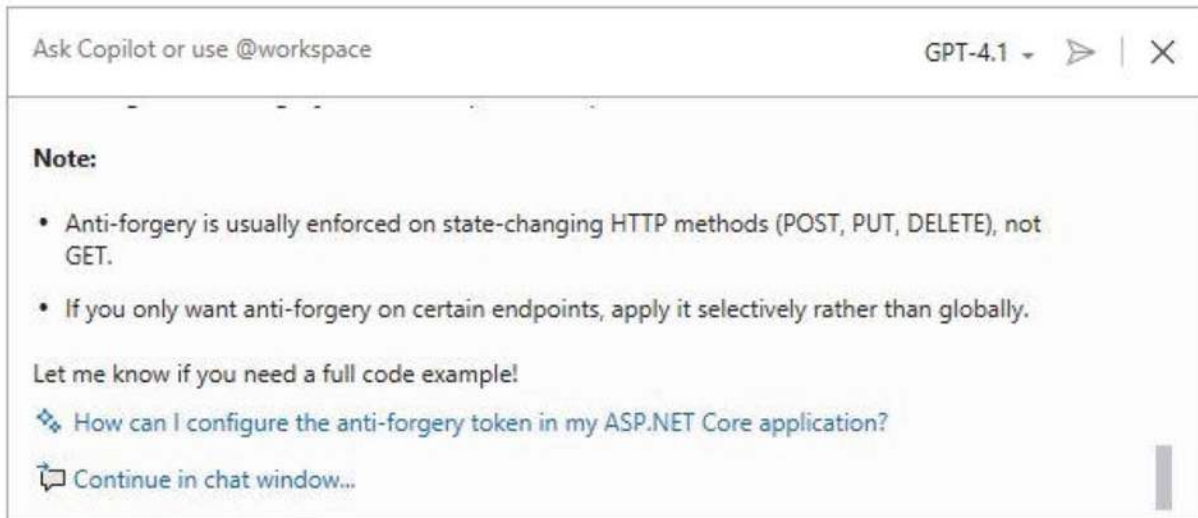
```



**Figure 1-86** Copilot Response

Copilot responds in the editor as shown in Figure [1-86](#). The response is usually detailed and you can scroll through the response.



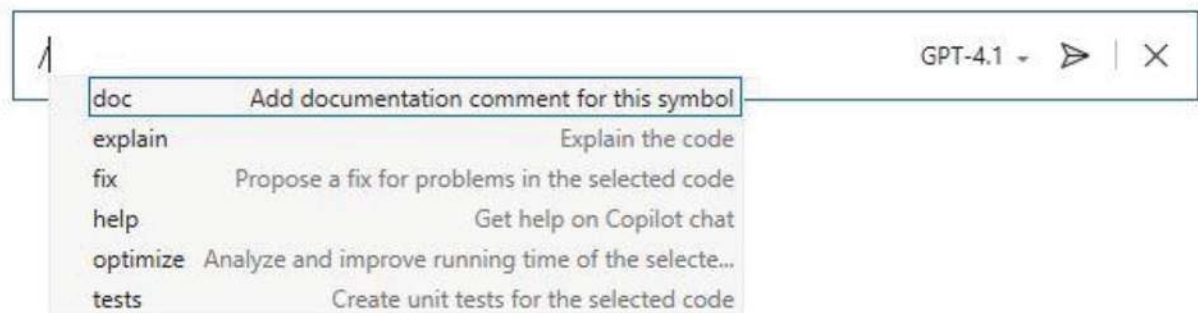


**Figure 1-87** Continue in Chat Window

At the end of the in-line chat window, you will see an option to continue in the chat window. This will send the response to the Copilot tab, and you can continue the conversation there instead of in-line in the code editor.

## Slash Commands

Another feature you can make use of is slash commands. As seen in Figure [1-88](#), you can type a / and then select the desired command.



**Figure 1-88** Available Slash Commands

Try the `/doc` command and see what happens.

## Copilot Agent Mode

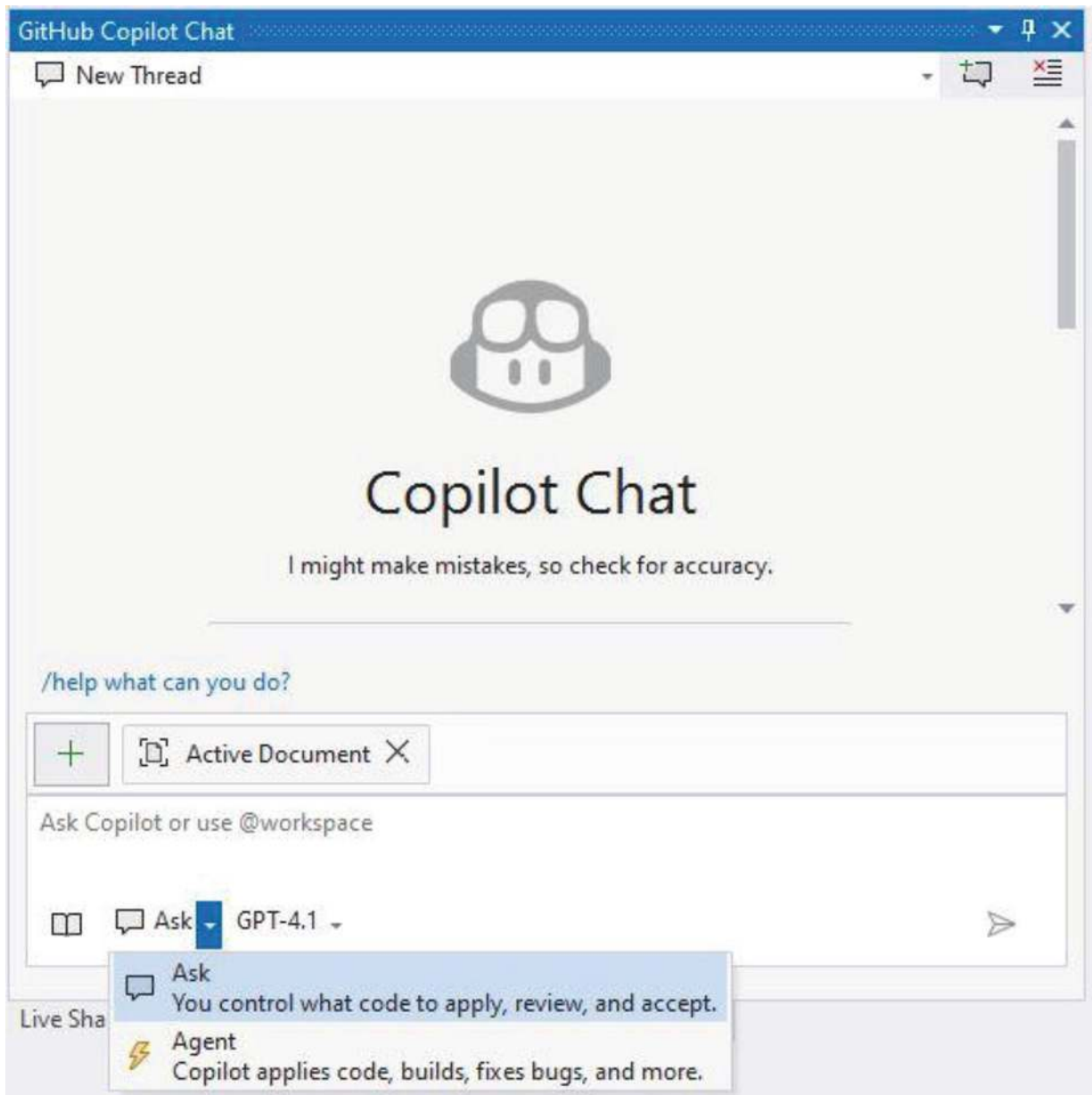


A feature of GitHub Copilot Chat is Copilot Agent (formerly known as Copilot Edits). It is designed to iterate across multiple files more efficiently. It offers the conversational flow of a chat with an inline review experience and provides the following benefits:

- Know exactly what changes are going to be made with a summary of the affected files and proposed changes.
- See the code diffs inline in your editor allowing you to press the tab key to accept or the `Alt+Del` key to reject changes.
- Allows you to use checkpoints to revisit earlier iterations of a code file or apply an alternate approach for new ideas.

The feature is a solid addition to GitHub Copilot and you might be wondering when to use Copilot Chat as opposed to Copilot Agent. To choose when to use which, think of it as follows. Copilot Chat provides developers with a general-purpose conversational interface in Visual Studio for exploring programming concepts, asking questions about the code in your solution, asking questions about C#, and generating code snippets on demand. This allows you to manually apply suggestions and rely on your normal Git version control.

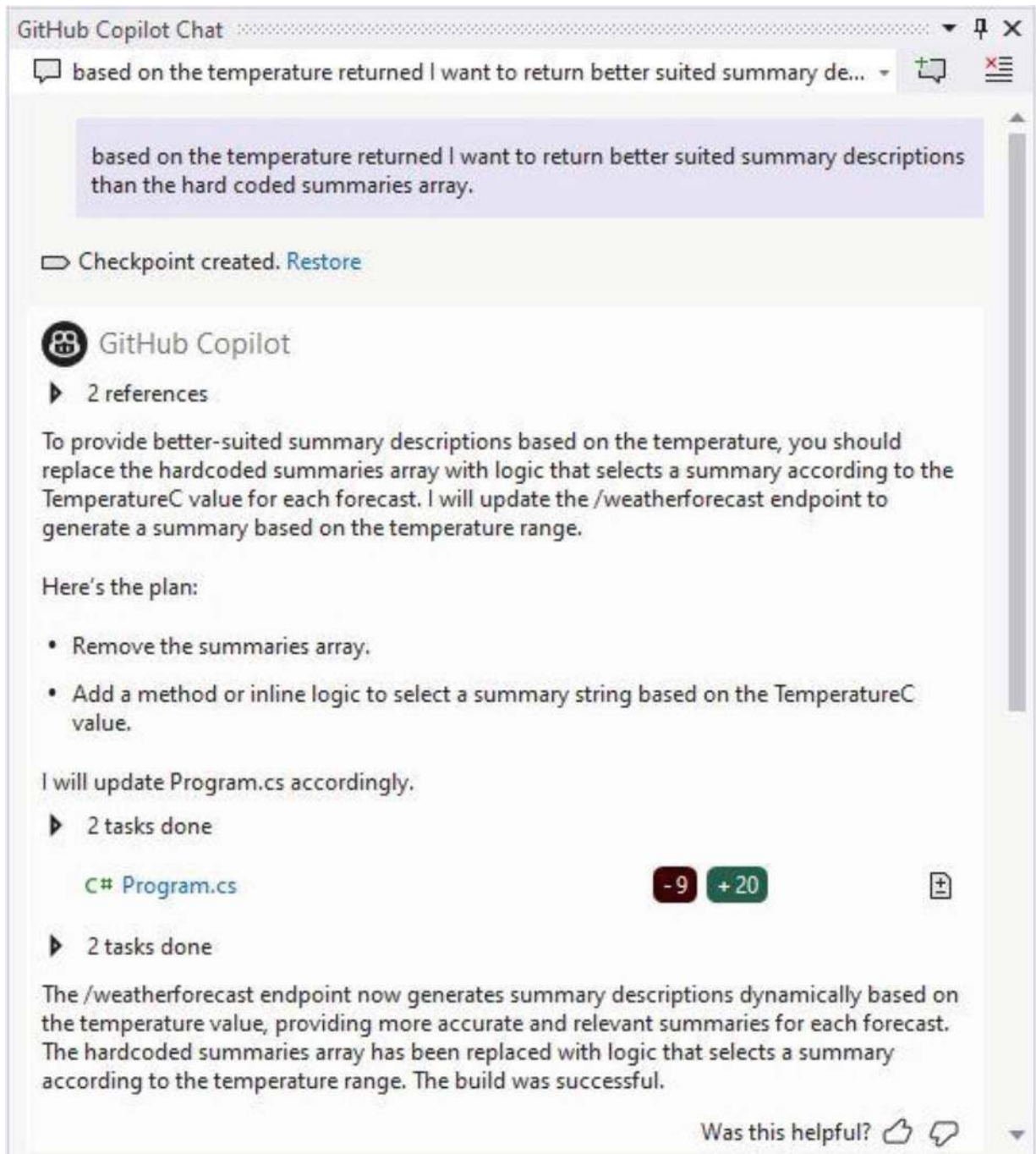
Copilot Agent, on the other hand, offers a focused multi-file editing workflow that automatically previews diffs across all affected files and allows you to accept or reject individual changes. It also maintains a built-in rollback history to reset any edits until you close Visual Studio.



**Figure 1-89** Copilot Agent

Ensure Agent mode is enabled by going to Tools ► Options ► GitHub ► Copilot and checking the option to Enable Agent mode. You might need to restart Copilot Chat or Visual Studio. To access Copilot Agent, select Agent from the dropdown menu as shown in Figure [1-89](#). Go ahead and ask it a question and review the results (Figure [1-90](#)).

To illustrate the power of Copilot Agent in screenshots in a book is to do it an injustice. You have to experience it first-hand.



**Figure 1-90** Copilot Agent in Action

Here, I have asked it to modify the standard logic in the Weather API boilerplate code that comes with the API template in Visual Studio.



Agent mode allows developers to use built-in tools or MCP tools. Model Context Protocol (MCP) is an open standard that gives AI models the ability to interact with external tools and services. Visual Studio allows developers to enhance GitHub Copilot's Agent mode by connecting to any MCP-compatible server. This means that:

- An MCP client, such as Visual Studio, can connect to MCP servers and request actions on behalf of the AI model.
- The MCP server provides tools exposing specific functionality via a well-defined interface.
- The MCP protocol will define the message format for handling communication between the client (Visual Studio) and the servers, which includes tool discovery, invocation, and response handling.

To understand how to use an MCP server, let's configure the GitHub MCP server. The steps to be taken are as follows:

1. Install Docker Desktop if your machine does not already have Docker installed.
2. Verify the Visual Studio prerequisites.
3. Generate a Personal Access Token (PAT) on GitHub.
4. Create your `.mcp.json` file.
5. Authenticate and enable tools.
6. Validate the setup.

The prerequisites for Visual Studio are that you need to be running Visual Studio 2022 version 17.14 or later and that you are signed in to Visual Studio with your GitHub account with Copilot access.

Generating a personal access token on GitHub is also straightforward. Click the GitHub account icon in the top right and select Settings. Scroll all the way to the bottom and select Developer settings. There you will be able to generate a personal access token. I suggest generating a classic token with at least the `repo` scope.

You now need to create your `.mcp.json` file. Create it in `<SOLUTIONDIR>\.mcp.json` using Visual Studio. Paste the contents in Listing [1-7](#) into the `.mcp.json` file.

```
{
  "inputs": [
    {
      "id": "github_pat",
      "description": "GitHub personal access
token",
      "type": "promptString",
      "password": true
    }
  ],
  "servers": {
    "github": {
      "type": "stdio",
      "command": "docker",
      "args": [
        "run",
        "-i",
        "--rm",
        "-e",
        "GITHUB_PERSONAL_ACCESS_TOKEN",
        "ghcr.io/github/github-mcp-server"
      ],
      "env": {
        "GITHUB_PERSONAL_ACCESS_TOKEN":
"${input:github_pat}"
      }
    }
  }
}
```

**Listing 1-7** GitHub MCP Server JSON

Visual Studio also checks for MCP configurations set up by other development environments. MCP server configurations are read from

the following directories in the following order:

1. `%USERPROFILE%\.mcp.json` – Sets a global MCP server configuration for the current user. Adding a server here makes it load for every solution in Visual Studio.
2. `<SOLUTIONDIR>\.vs\.mcp.json` – Configures MCP servers just for Visual Studio, scoped to a specific user and a specific solution.
3. `<SOLUTIONDIR>\.mcp.json` – Ideal if you want an MCP setup that you can commit to source control along with your repository.
4. `<SOLUTIONDIR>\.vscode\.mcp.json` – Tied to the repository or solution, but usually not checked into source control.
5. `<SOLUTIONDIR>\.cursor\.mcp.json` – Also tied to the repository or solution, and typically not tracked in source control.

Please note that some of these locations require `.mcp.json` while others require `mcp.json`.

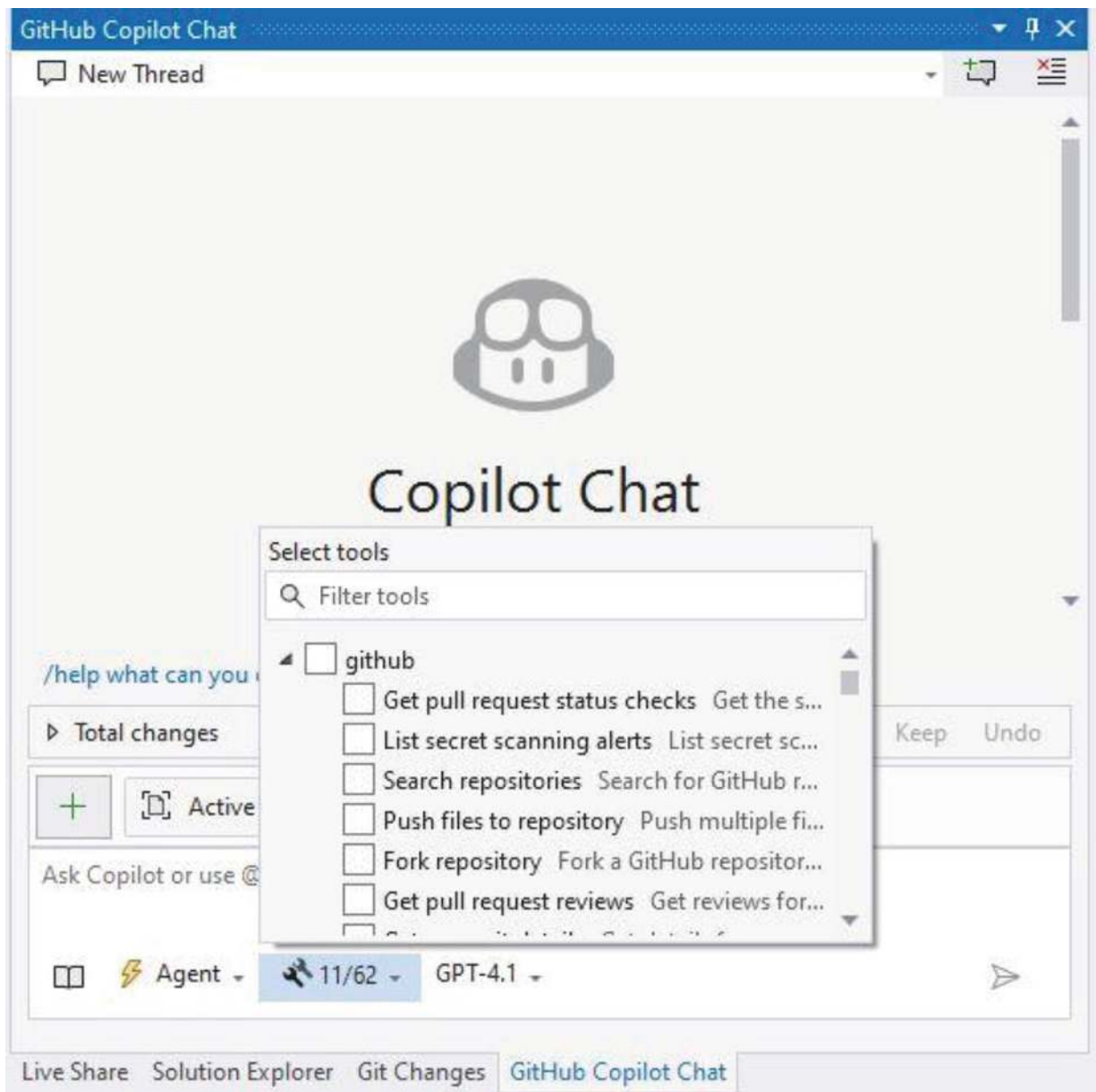
When you save this file, Visual Studio should prompt you for your PAT as seen in Figure [1-93](#).



**Figure 1-93** The Personal Access Token Prompt

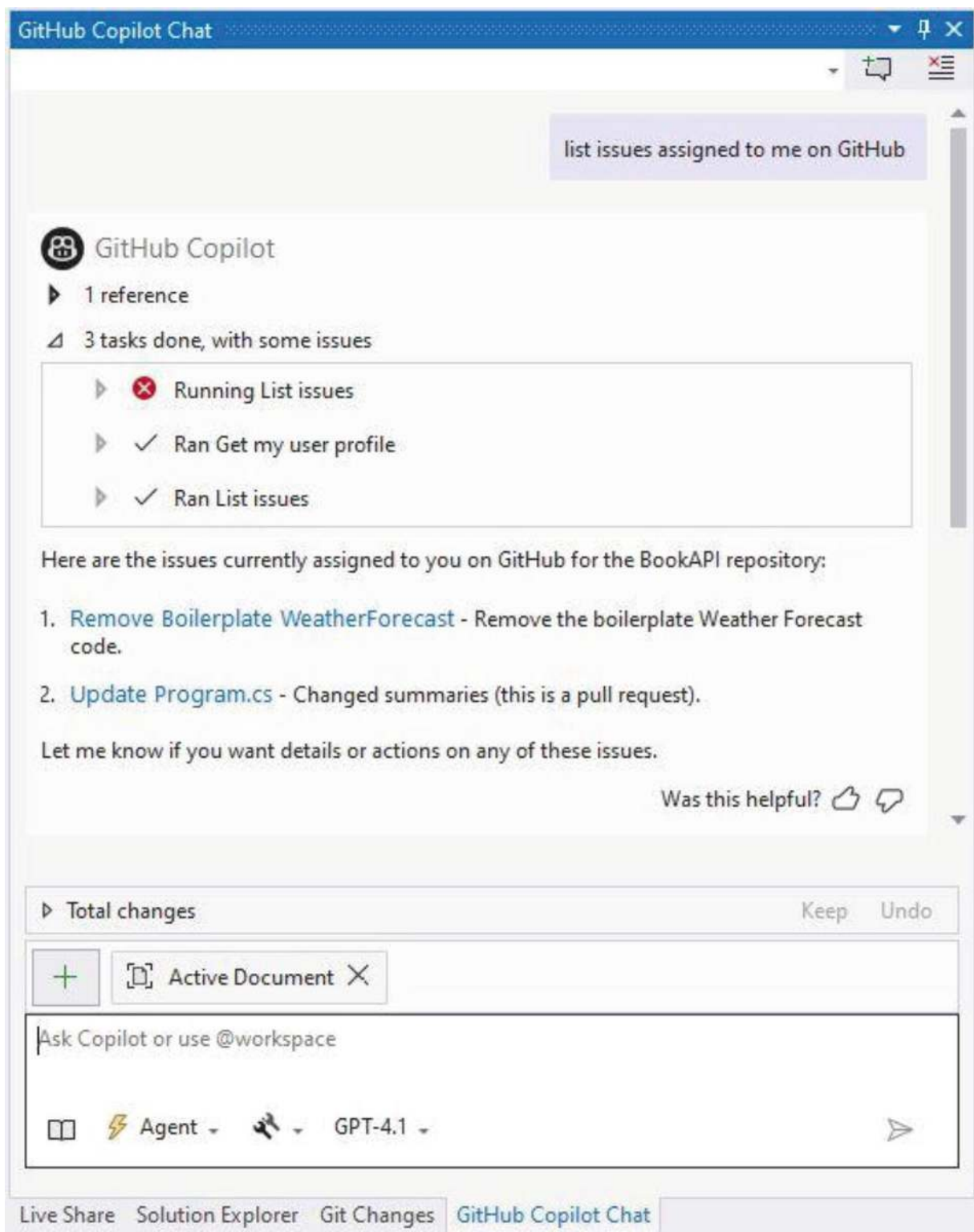
You might need to restart Visual Studio, but after adding your personal access token, in the tools dropdown in Copilot you will see the GitHub MCP server tools listed (Figure [1-94](#)).





**Figure 1-94** The Available GitHub MCP Tools

On GitHub, I went ahead and added an issue to my repository to remove the boilerplate code.



**Figure 1-95** Running a Tool

To test if my MCP server tools work, in Visual Studio, with Agent selected, I type in “list issues assigned to me on GitHub”. The Agent then starts running the required tools and returns the results to me as seen in Figure 1-95. It also returned one of the repository’s pull requests because I enabled every available tool by selecting the GitHub option, which activates all tools in the dropdown.

Visual Studio allows the MCP server to use three transport options: local standard input/output (`stdio`), server-sent events (`SSE`), and streamable HTTP (`http`). Right now, out of the three primitives—tools, prompts, and resources—servers can only deliver tools to Copilot’s agent mode. The official MCP server repository (<https://github.com/modelcontextprotocol/servers>) is a good starting point if you want examples of reference, official, and community-contributed servers that demonstrate what MCP can do. You’ll find servers handling tasks like file system operations, database work, and web services.

MCP is still fairly new, and the ecosystem is growing quickly. As adoption picks up, you’ll see more servers and tools becoming available to integrate into your projects.

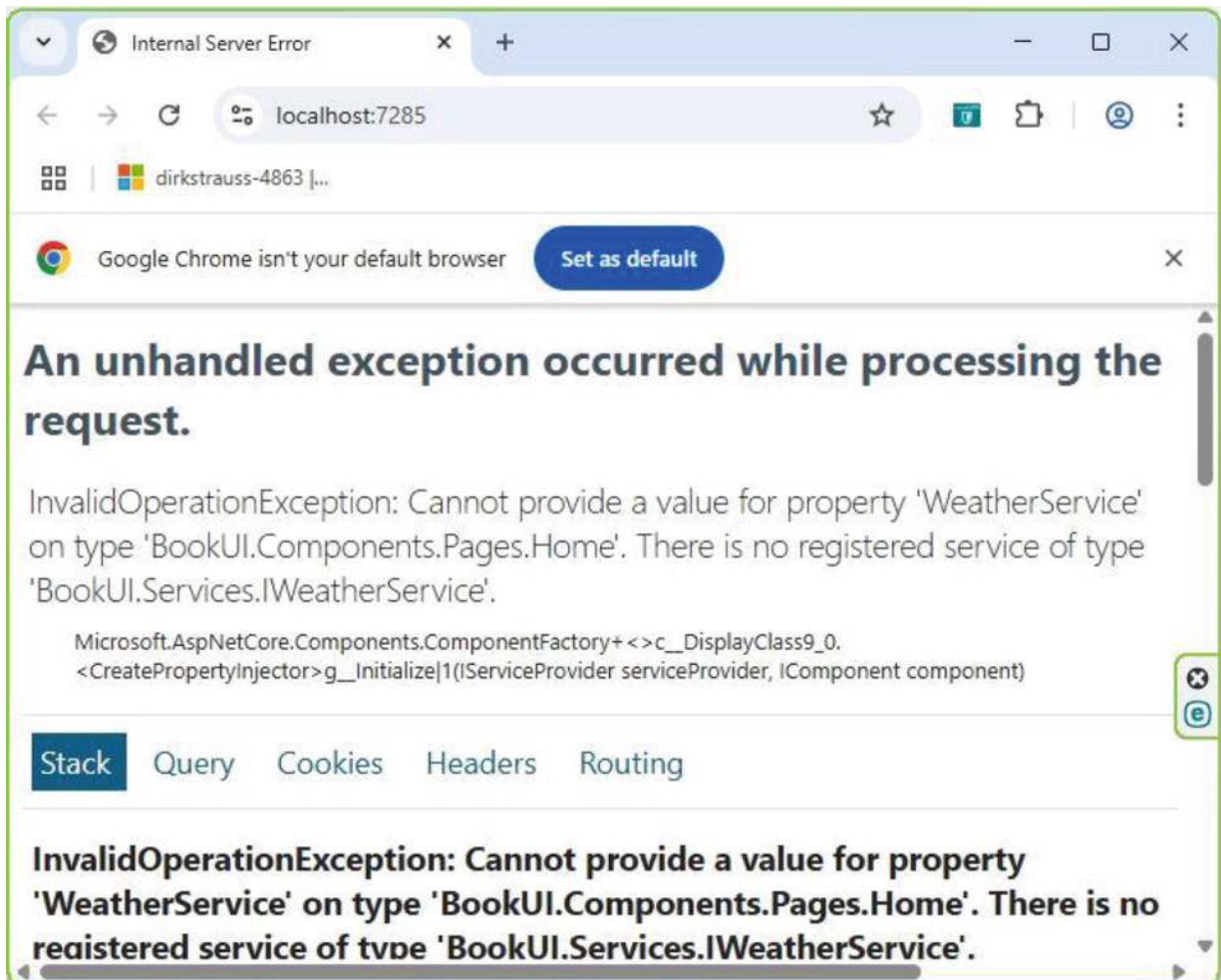
With the version 17.14.13 August release of Visual Studio 2022, MCP support became GA in Visual Studio. Released after the publication of this chapter, MCP support became a first-class experience in Visual Studio 2022. No more copying JSON snippets into `.mcp.json` files. This is another example of the rapidly evolving landscape of AI tools and Visual Studio in general.

## Using Images in GitHub Copilot

You can upload images to GitHub Copilot in Chat mode and Agent mode. This is more helpful than some folks might think. There are times when you need to “show” Copilot what you mean when describing a problem. Think of a UI component that is not being styled correctly in your Blazor application. You can explain it and ask for a solution, but to explain it and provide an image as context is so much more powerful.

To illustrate the concept of providing an image to Copilot, I have created a service in a Blazor application that is used in the home page, but that is not registered in the DI container. When I run this

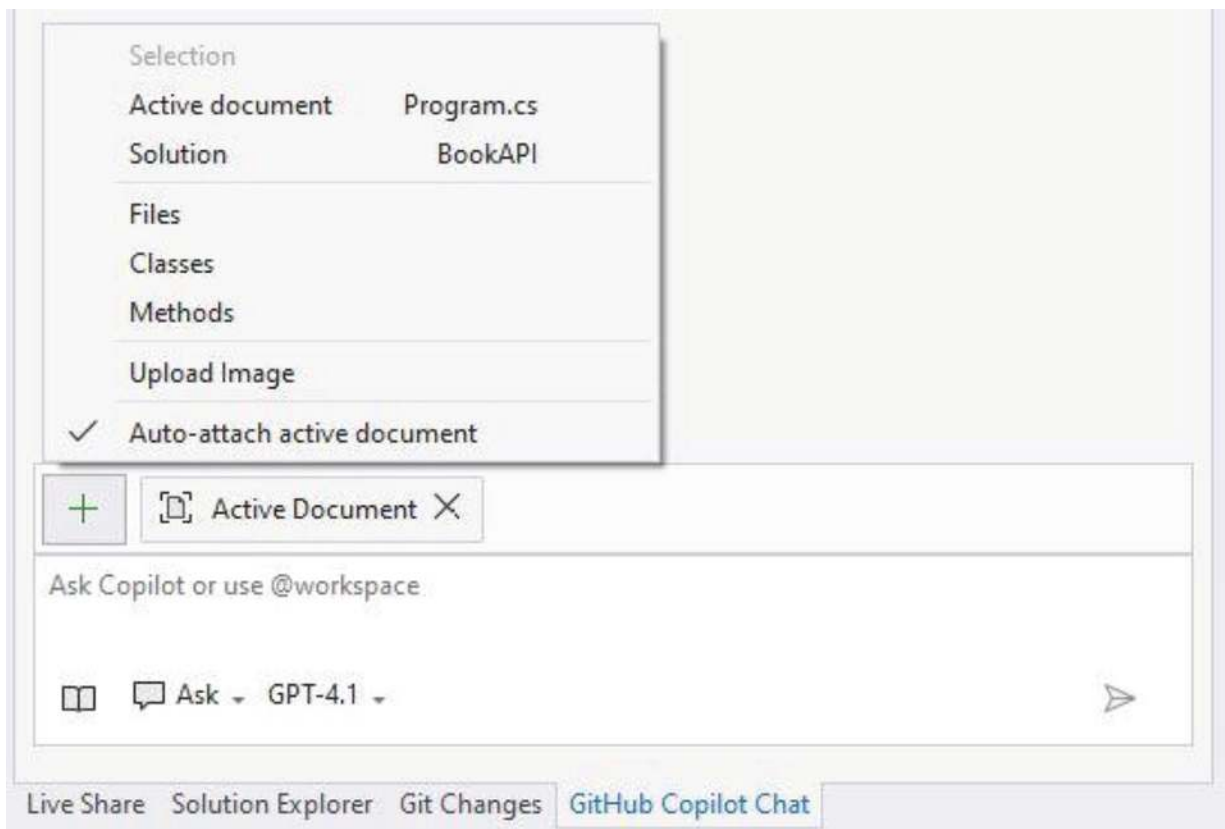
application, I will see the familiar error displayed in my browser as seen in Figure [1-96](#).



**Figure 1-96** The InvalidOperationException

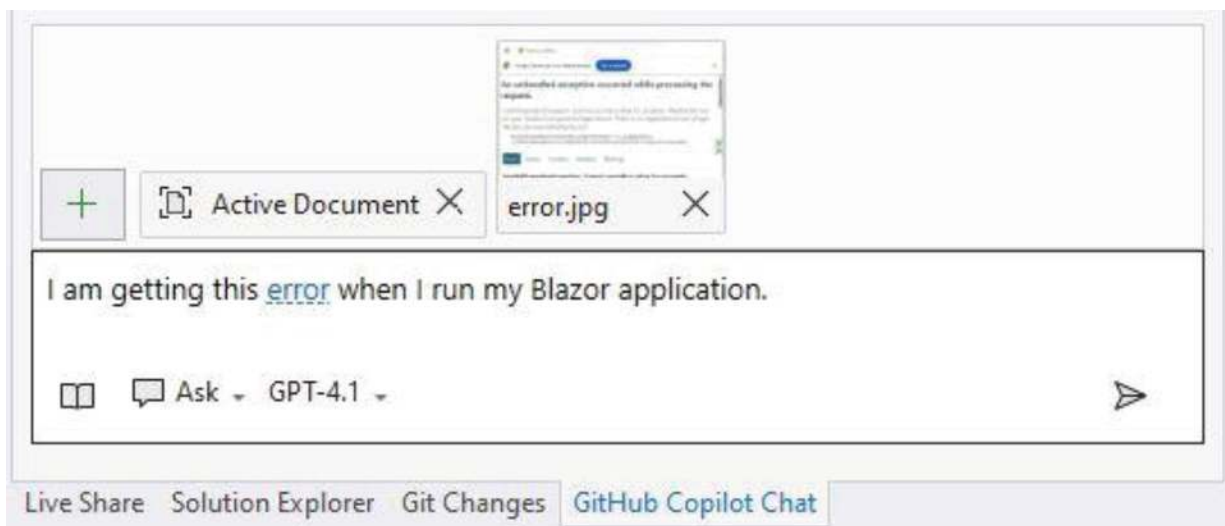
For a moment, ignore the fact that this error is very descriptive and tells you exactly what the problem is. You can take a screenshot of this error and save it to disk.

You will be using this image to provide more context to Copilot chat.



**Figure 1-97** Upload an Image

Click on the green + button and click on Upload Image to attach the screenshot you just saved (Figure [1-97](#)).



**Figure 1-98** Describe the Error

When the image is attached (Figure [1-98](#)), ask your question, but be vague. Do not give any specifics in your question.



I am getting this error when I run my Blazor application.



GitHub Copilot

1 reference

The error message indicates that your Blazor application is trying to inject a service of type `BookUI.Services.IWeatherService` into your **Home** component, but this service has not been registered with the dependency injection (DI) container.

**How to fix:** You need to register your `IWeatherService` and its implementation (likely `WeatherService`) in the DI container in your `Program.cs` file.

Add the following line before `var app = builder.Build();` in your `Program.cs`:

C# Program.cs

Apply



```
builder.Services.AddScoped<BookUI.Services.IWeatherService>
```

Found similar code in public repos. [View matches](#)

**Full context:**

C# Program.cs

Apply



```
using BookUI.Components;

namespace BookUI
{
    public class Program
    {
        public static void Main(string[] args)
        {
        }
    }
}
```

How can I test the `IWeatherService` implementation in my Blazor application?



Active Document



Ask Copilot or use @workspace



Ask



GPT-4.1

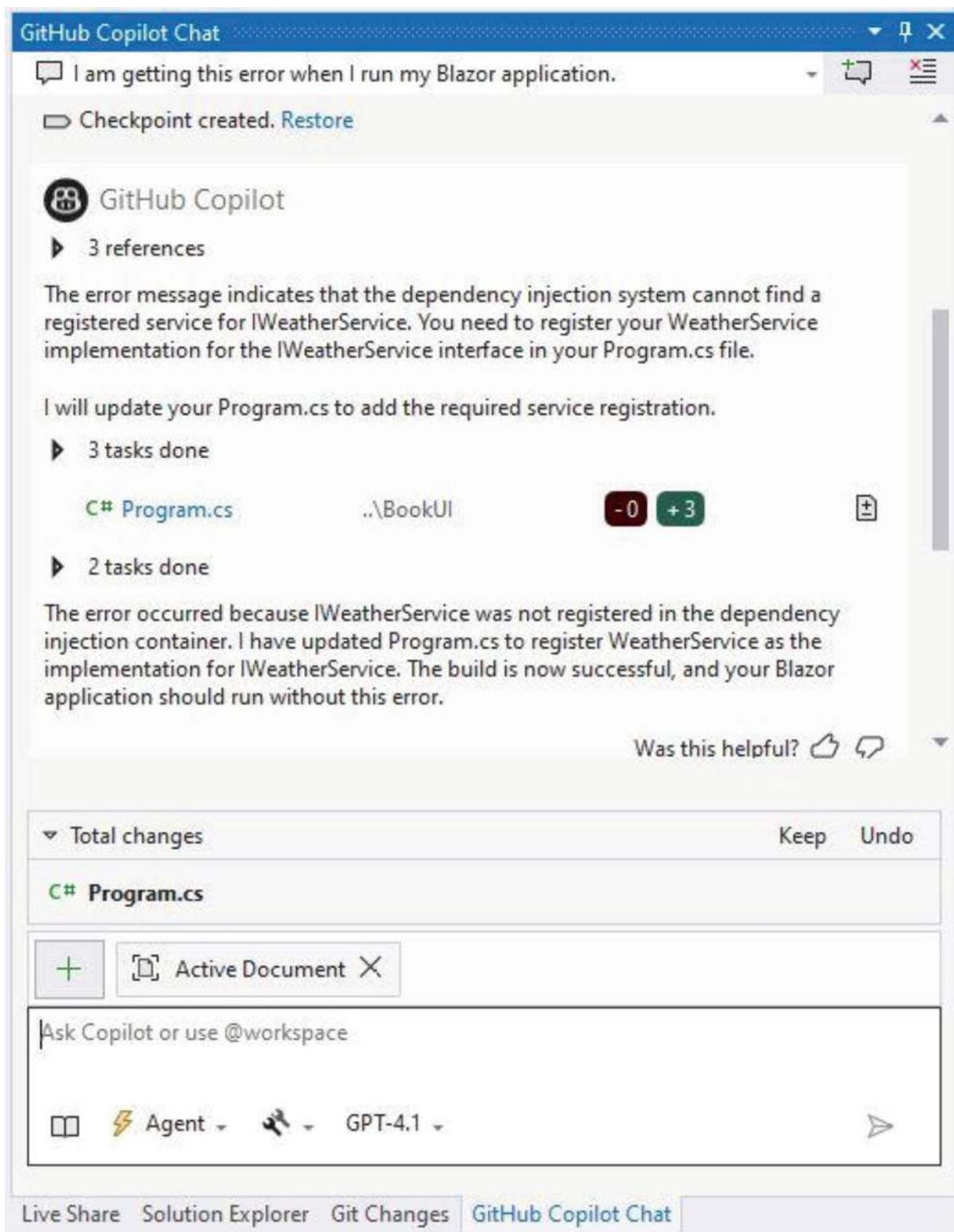




---

**Figure 1-99** Copilot Chat Providing a Solution

As seen in Figure [1-99](#), Copilot Chat provides a complete solution to your problem based solely on the uploaded image.



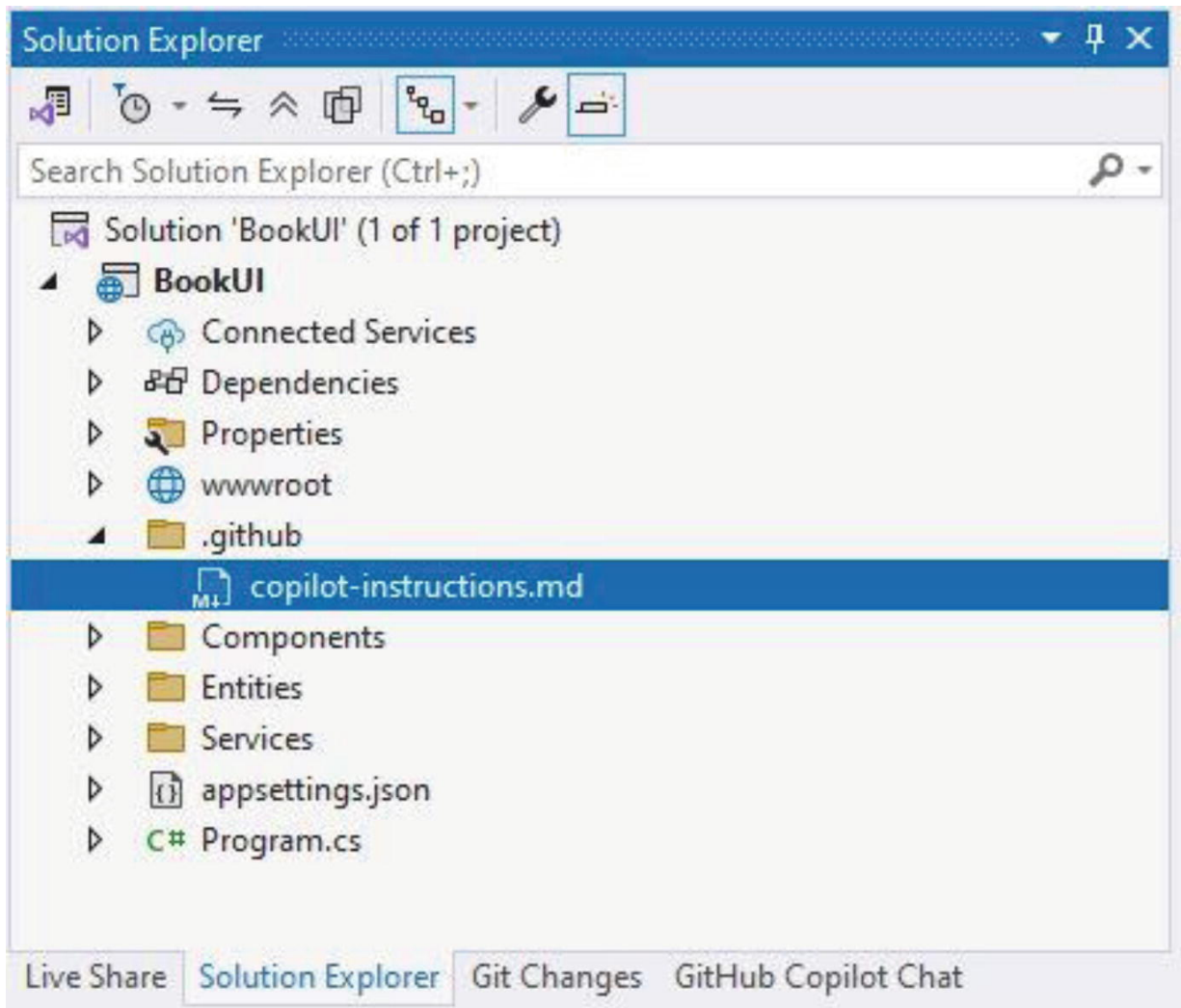
**Figure 1-100** Copilot Agent Providing a Solution

To illustrate the difference and power of Copilot Agent, upload the image again in Agent mode and ask the same question. This time, Copilot actually fixes the error for you in your code. You will see the diff window open up showing you what changes were made. You can then keep or undo these changes.

## **Custom Instructions**

Custom instructions is the best way to get Copilot up to speed on the way you code. Every dev team has some kind of knowledge base or coding standard that they follow. Be this how to get started in the repo or the frameworks that the team is using, Copilot can't easily infer this information from the codebase itself.

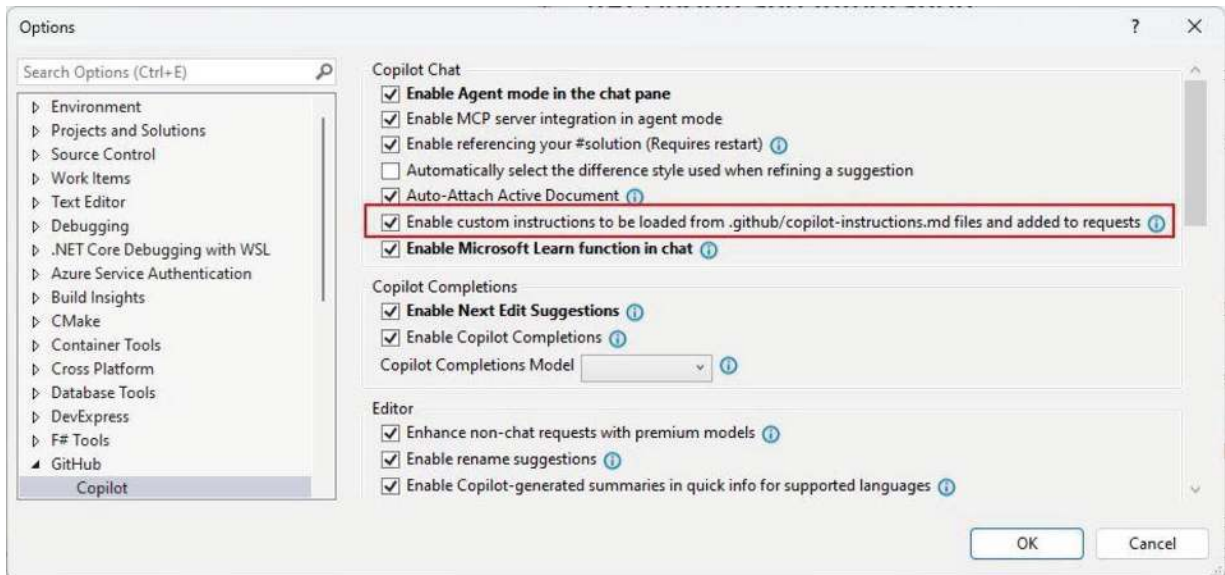
This is where custom instructions come in. Create a `.github` folder in your solution and add a file called `copilot-instructions.md` to this folder as shown in Figure [1-101](#).



**Figure 1-101** Adding the Copilot Instructions File

Here you can define the specific instructions you want Copilot to follow. This file basically acts like a system prompt, allowing me to seed information to Copilot, giving it a better idea of what my team is working on, naming conventions, and coding standards we follow, etc.

Secondly, make sure that you enable the functionality in settings, as seen in Figure [1-102](#).



**Figure 1-102** Custom Instructions Setting

Based off of the information I specify in the Copilot-instructions file, Copilot can come up with better suggestions when responding to my requests. This means that whenever Copilot suggests code, it is more likely to be in line with the standards my team follows.

For a nice example of a `copilot-instructions.md` file, have a look at

<https://gist.github.com/jamesmontemagno/ee1edce39cdedd185a789a0f69aa22b2> by James Montemagno. Just view the raw file and paste the markdown into your instructions file.

What are my team's coding standards



GitHub Copilot



▶ 1 reference

Your team's coding standards, as outlined in the `.github\copilot-instructions.md` file, are as follows:

---

## Blazor Code Style and Structure

- Write idiomatic and efficient Blazor and C# code.
- Follow .NET and Blazor conventions.
- Use Razor Components for component-based UI.
- Prefer inline functions for small components; move complex logic to code-behind or service classes.
- Use `async/await` for non-blocking UI operations.

## Naming Conventions

- PascalCase for component names, methods, and public members.
- camelCase for private fields and local variables.

What are some best practices for implementing error handling in Blazor applications?



Active Document ✕

Ask Copilot or use @workspace



Ask ▾ GPT-4.1 ▾




---

**Figure 1-103** Copilot Using Custom Instructions

Asking Copilot directly what my team's coding standards are returns the response as shown in Figure [1-103](#).





 #WeatherService create a property here to set the current city.



GitHub Copilot

▶ 3 references

C# WeatherService.cs

Apply



```
using BookUI.Entities;

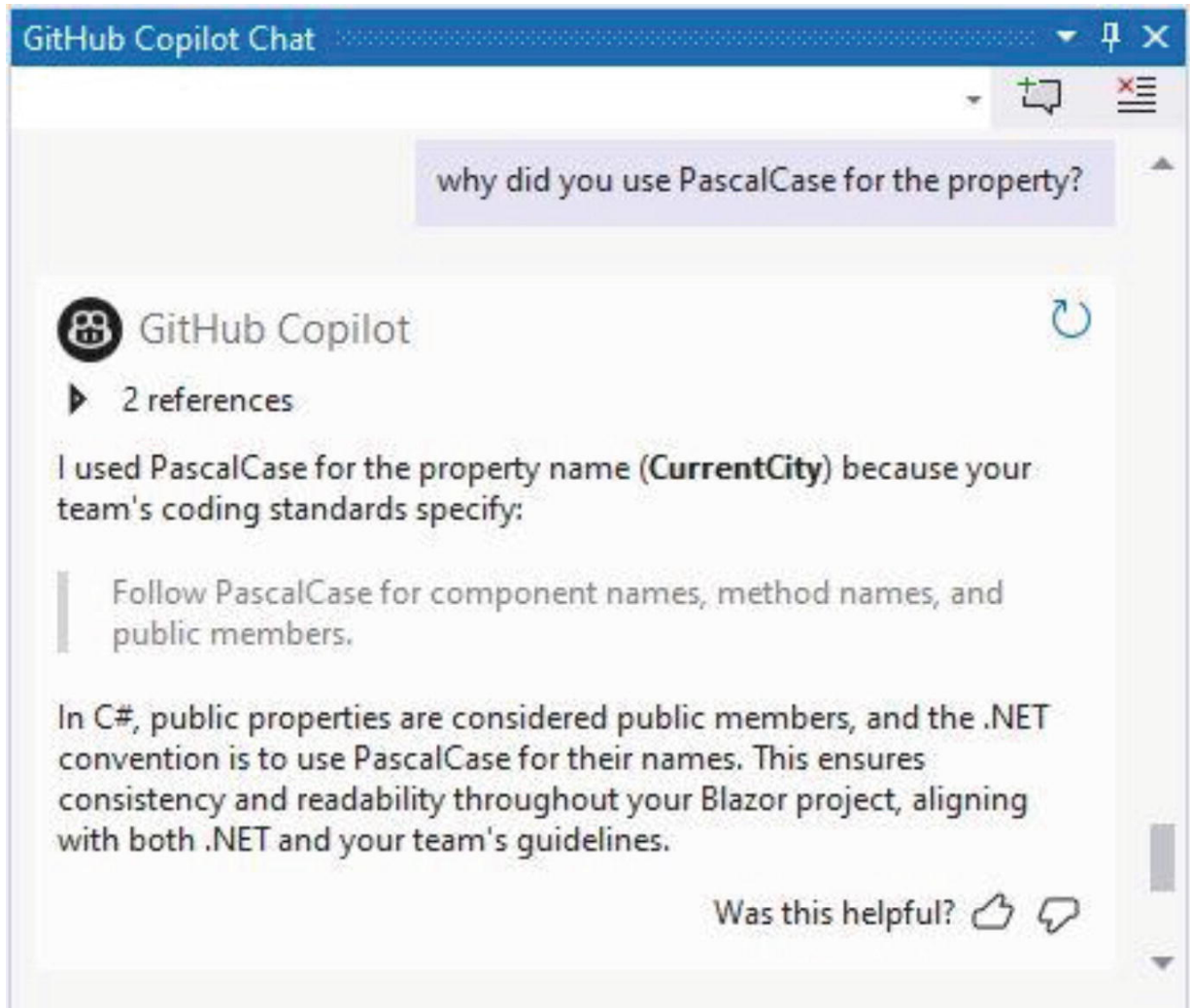
namespace BookUI.Services;

public class WeatherService : IWeatherService
{
    // Property to set or get the current city
    public string? CurrentCity { get; set; }

    public async Task<IEnumerable<WeatherForecast>
    {
        // Return hardcoded weather data
        var forecasts = new List<WeatherForecast>
        {
            new WeatherForecast { Date = DateOnly.FromDateTime(DateTime.Now.AddDays(1)), TemperatureC = 30 },
            new WeatherForecast { Date = DateOnly.FromDateTime(DateTime.Now.AddDays(2)), TemperatureC = 35 },
            new WeatherForecast { Date = DateOnly.FromDateTime(DateTime.Now.AddDays(3)), TemperatureC = 40 },
        };
        return await Task.FromResult(forecasts);
    }
}
```

**Figure 1-104** Asking Copilot to Generate Code

To test if Copilot is following the instructions when generating code, ask it to create a property for you in one of your code files. Here, as seen in Figure [1-104](#), I asked it to create a property to set the current city.



**Figure 1-105** Copilot Following Instructions

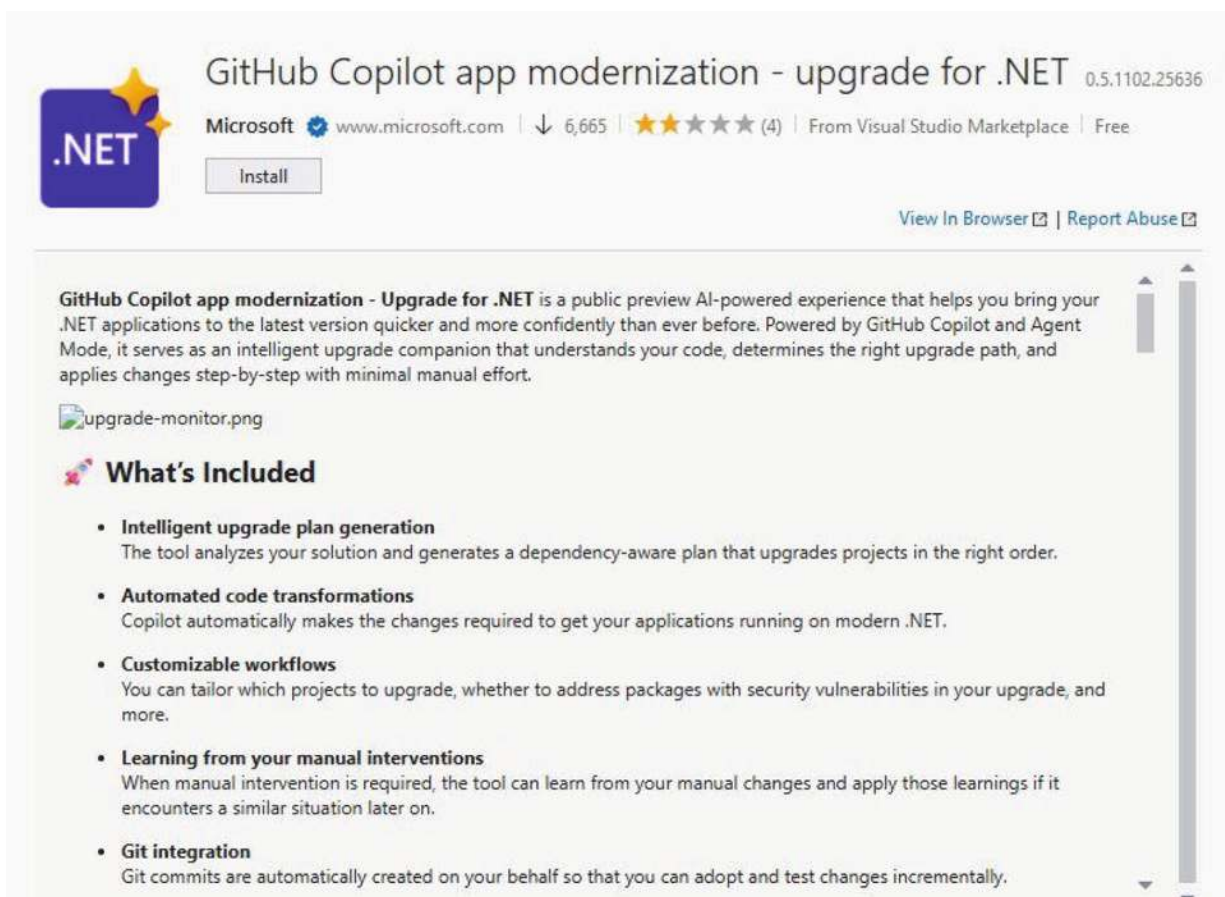
When I asked Copilot why it used PascalCase for the property, it told me that it did so because this was defined in the coding standards in the `copilot-instructions.md` file (Figure [1-105](#)).

The power of this file should not be underestimated. You can commit this instruction file to your repo, allowing all team members to

benefit from a single cohesive set of standards throughout your code base.

## Upgrade with Copilot

The process of upgrading old projects is probably one of the least favourite tasks a developer has to perform. It can be time consuming, risky, and full of manual toil. Well, Copilot is here to help. GitHub Copilot app modernization – upgrade for .NET is an AI-powered experience that helps you upgrade your .NET applications to the latest version more conveniently than before.

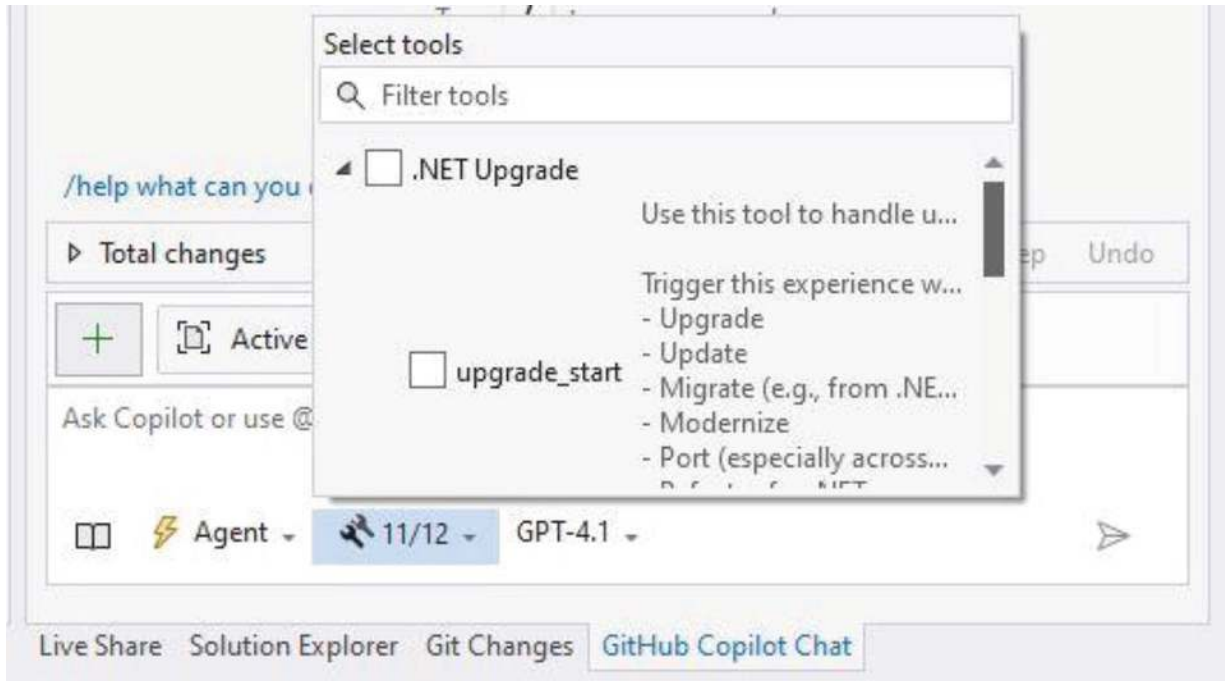


**Figure 1-106** GitHub Copilot App Modernization

To get started, from the toolbar in Visual Studio select Extensions ► Manage Extensions. Search for GitHub Copilot app modernization. You will see the extension listed in the search results, as seen in Figure [1-106](#).

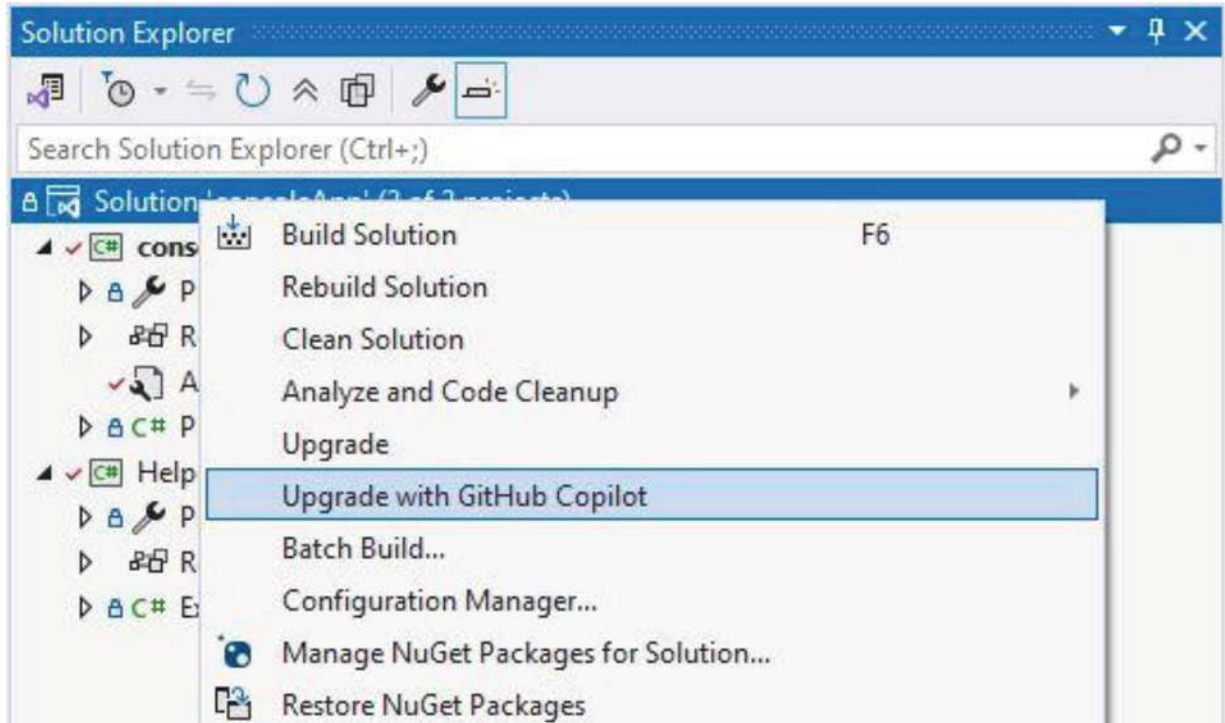
Click on the Install button. You will now be prompted to close Visual Studio, after which the extension will be installed. Once the installation completes, start Visual Studio again and open your solution that you want to upgrade.

To unlock the full power of AI-assisted upgrades, make sure that you are running Visual Studio 17.14 and that you have enabled Agent mode from the GitHub Copilot settings.



**Figure 1-107** Select the .NET Upgrade Tool

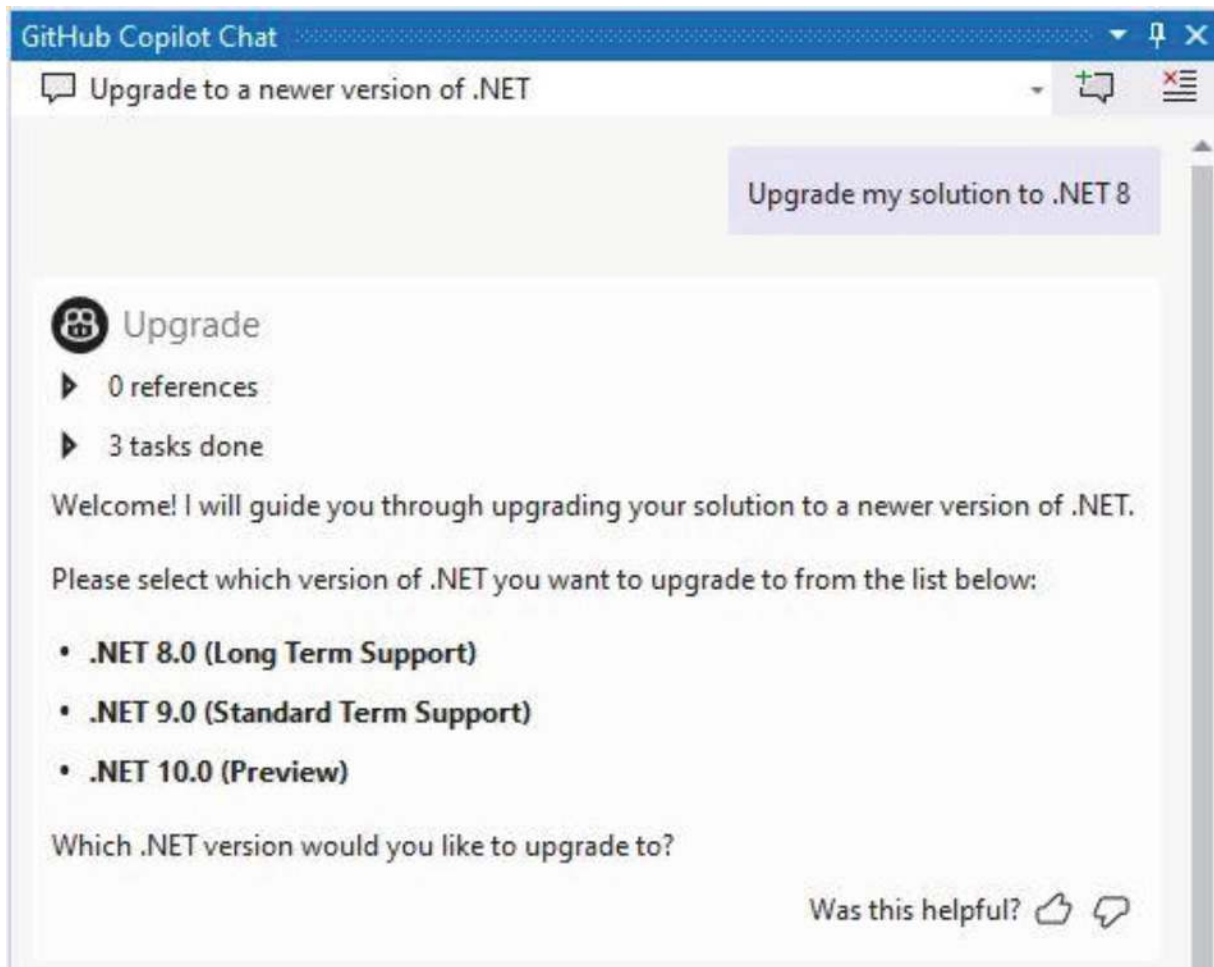
As seen in Figure [1-107](#), select Agent mode and check the .NET Upgrade tool from the tools dropdown.



**Figure 1-108** Upgrade via Right-Click

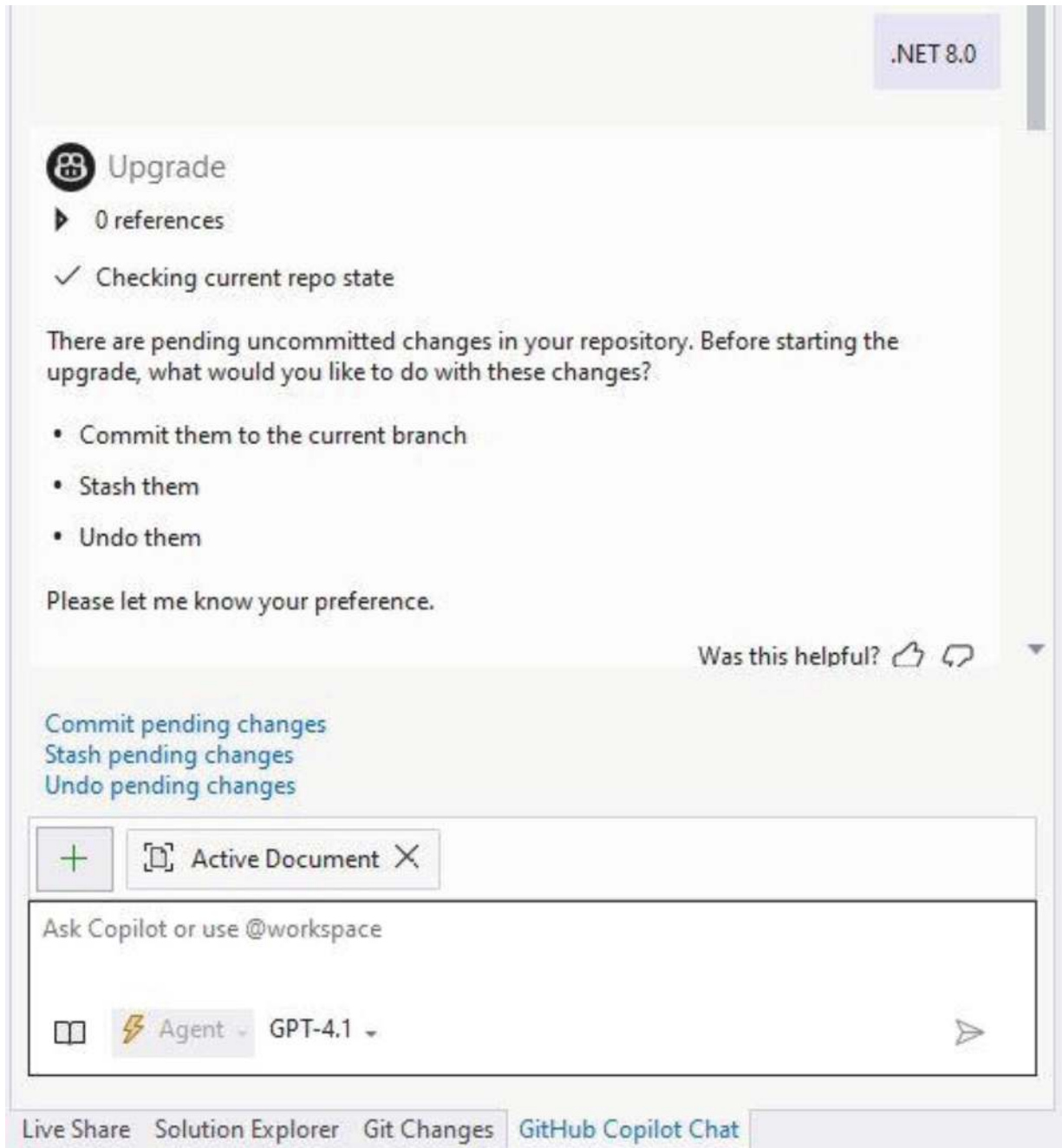
To start upgrading your solution, you can right-click your solution and select Upgrade with GitHub Copilot, as seen in Figure [1-108](#).





**Figure 1-109** Starting the Upgrade Process

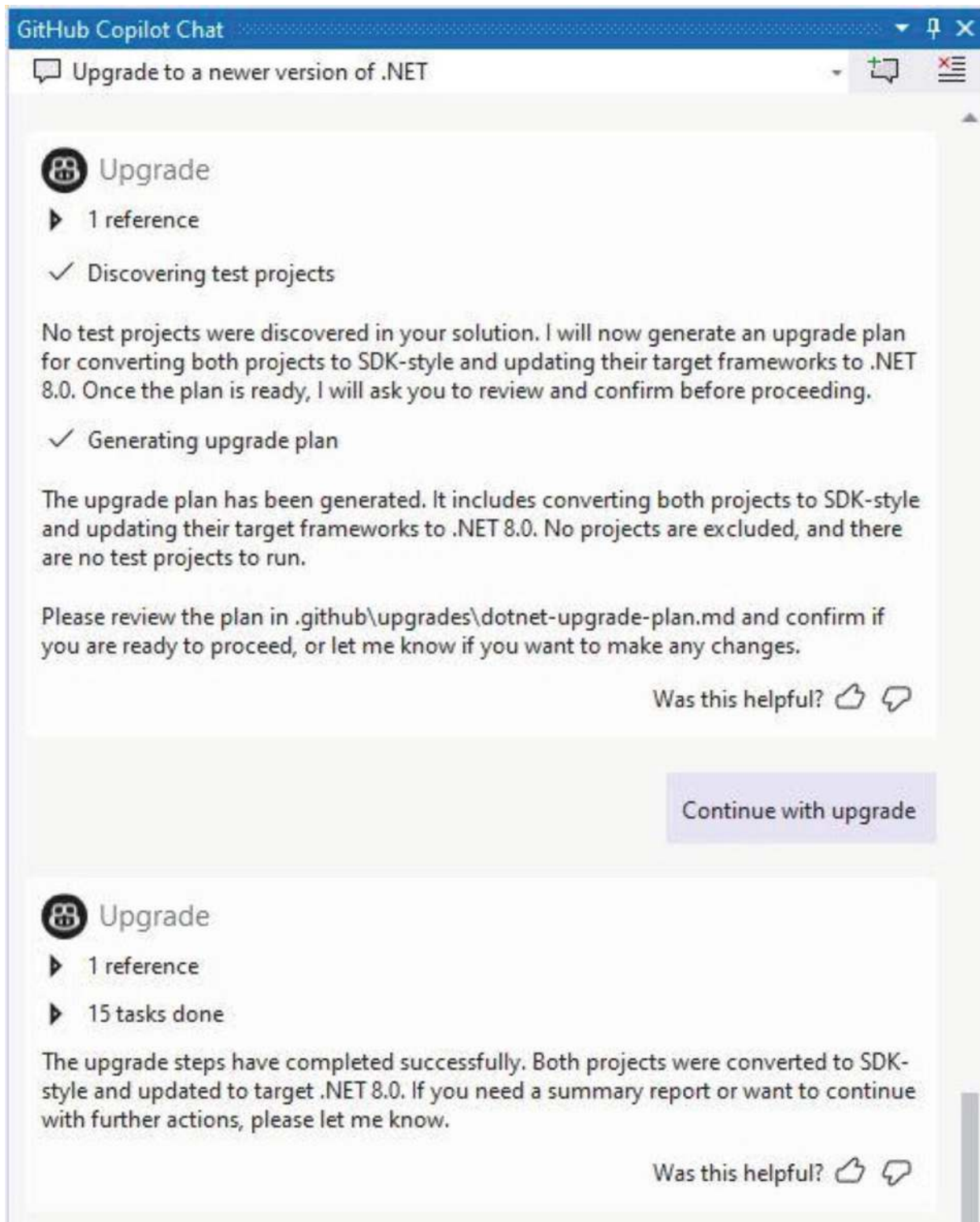
Alternatively, you can just tell Copilot what you want to do. As seen in Figure [1-109](#), I asked Copilot to upgrade my solution to .NET 8. Something slightly strange that responds with a choice of upgrade options after I specified .NET 8, but hey, this tool is saving me time, so I'm not complaining.



**Figure 1-110** Performing Additional Actions

Copilot Agent will now start iterating through a series of steps and ask you for confirmation along the way. Here, as seen in [Figure 1-110](#), Copilot has noticed that I have some pending changes to my solution. It now asks me what I want to do. Commit the changes, stash them, or undo the changes. Conveniently, I can click on the option and send that as a response to Copilot.





**Figure 1-111** Generating an Upgrade Plan

Copilot then might go through an additional series of steps, asking you for confirmation along the way where necessary. It then generates

an upgrade plan for you that you can review and ultimately approve.

Copilot will then start to perform the upgrade, and depending on your application size and complexity, this could take some time. Right throughout the process, Copilot will keep you informed of the status of the upgrade tasks. Once it has completed the upgrade you will see the confirmation that your project has been upgraded (Figure [1-111](#)).

Give this tool a try. With the power of AI, mundane tasks are streamlined and completed with minimal developer interaction (apart from answering a few Agent questions).

---

## Summary

In this chapter, we explored what Visual Studio 2022 brings to the table for modern development. We covered how to install Visual Studio and choose the right workloads for your projects. You learned how to navigate the IDE efficiently, from using Solution Explorer and the Toolbox to managing references and viewing build artifacts. I showed you how to use core navigation features, keyboard shortcuts, and the powerful search to stay productive. We also touched on features like code cleanup, IntelliCode, Live Share, and GitHub Copilot to help you write better code faster and collaborate seamlessly. This foundation sets you up to take full advantage of what Visual Studio 2022 offers as we dive deeper into the chapters ahead.

## 2. Working with Visual Studio 2022

Dirk Strauss<sup>1</sup> 

(1) Uitenhage, South Africa

---

If you have worked with previous versions of Visual Studio, you may find that Visual Studio 2022 does not break the mold. Visual Studio 2022 feels much the same as previous versions, and that's a good thing.

While there are new features and enhancements in Visual Studio 2022, developers find it easy to work with it from the start. If you are new to Visual Studio, a few topics deserve a closer look. In this chapter, we are going to look at the following:

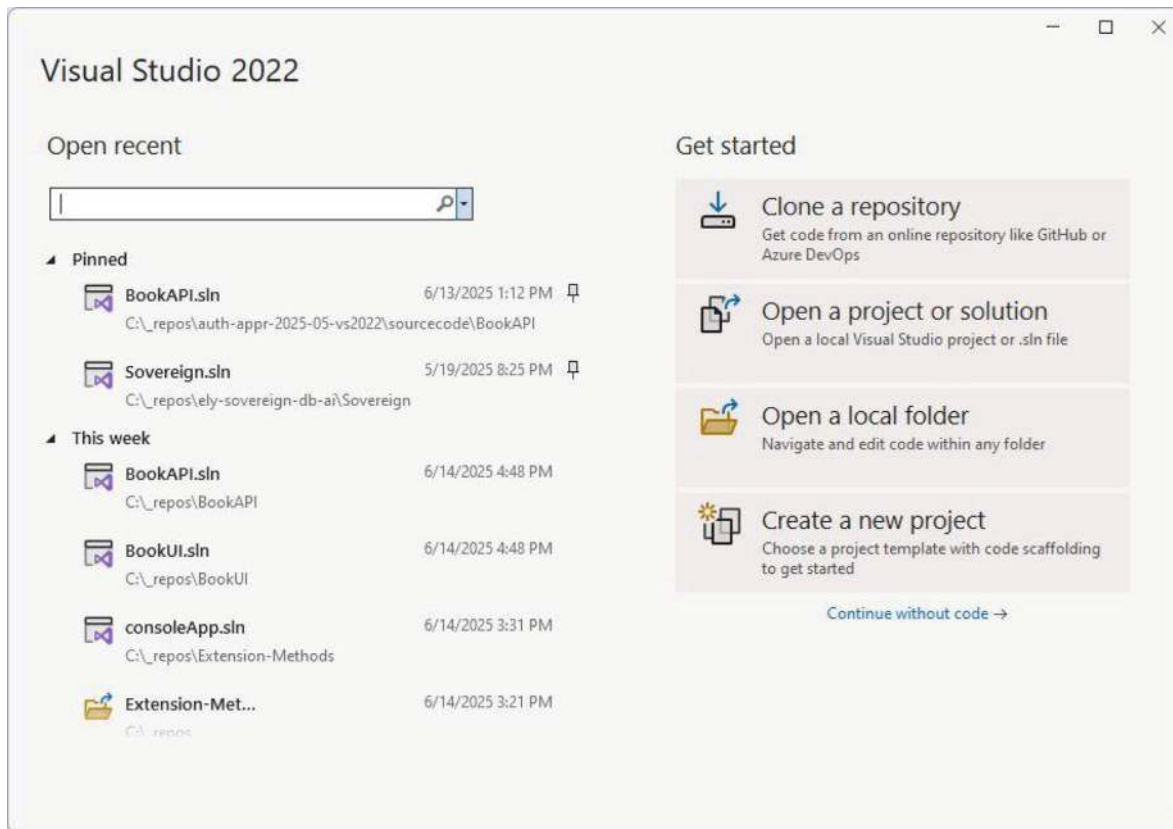
- Visual Studio project types and when to use them
- .NET Aspire
- Managing NuGet packages
- Creating project templates
- Creating and using code snippets
- Using bookmarks and code shortcuts
- The Server Explorer window
- Visual Studio Windows

This chapter is an extension of Chapter [1](#) in many respects. Things that didn't make it into Chapter [1](#) are discussed in this chapter. I do believe, however, that these are essential to working with Visual Studio and benefit developers in their day-to-day coding.

---

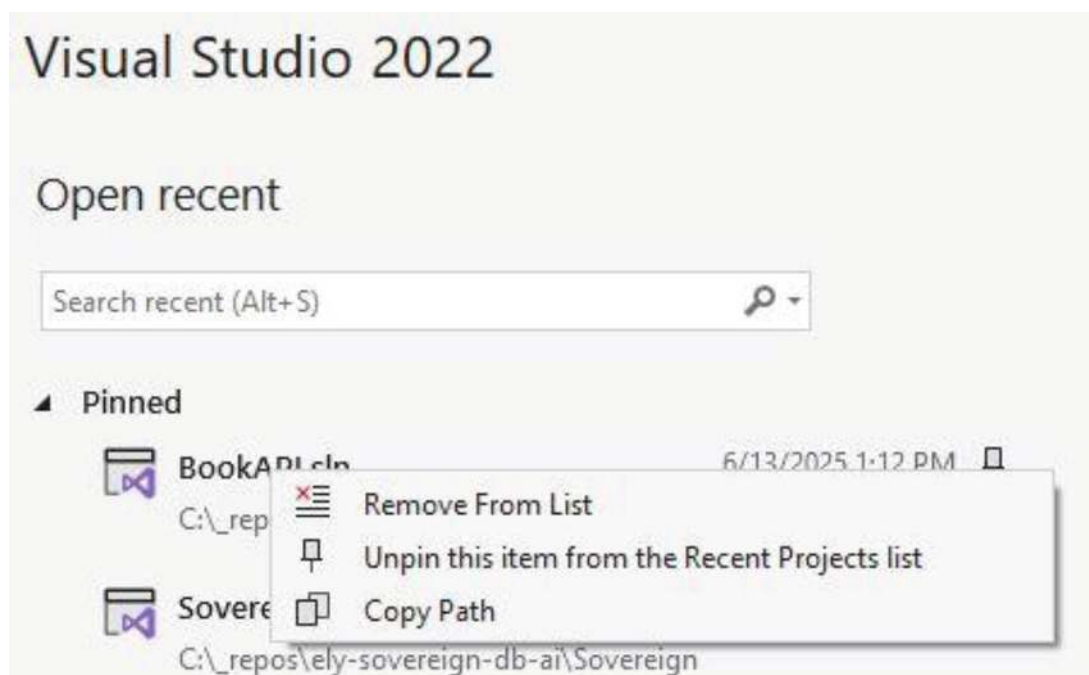
### Visual Studio Project Types

Visual Studio 2022 allows developers to create a new project in several ways. The most obvious is when you start Visual Studio 2022. You are presented with the Start screen displayed in Figure [2-1](#).



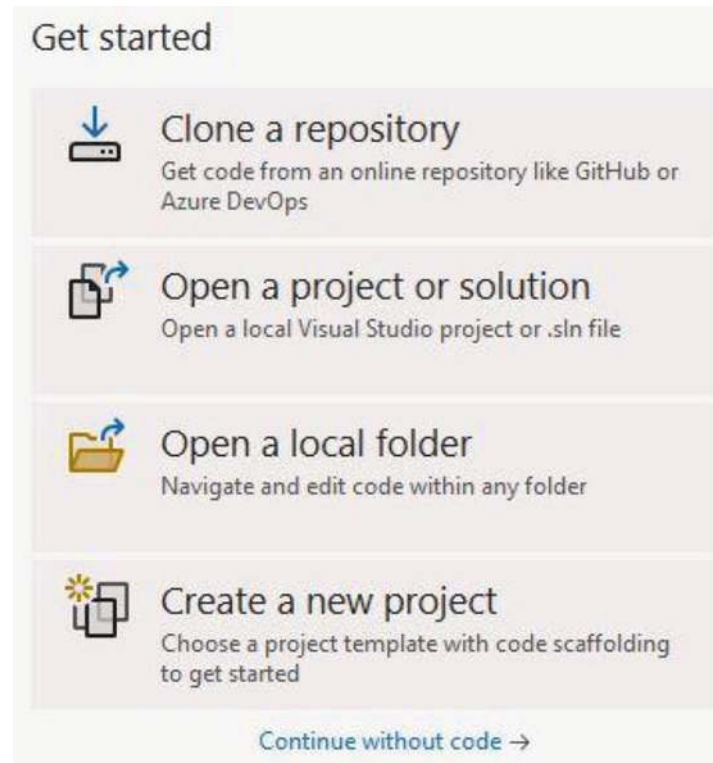
**Figure 2-1** Visual Studio 2022 Start Screen

Here, you can see recent projects that you can pin to the Start screen and always keep available. If you right-click any recent projects, you can see a context menu pop up, as shown in Figure 2-2.



**Figure 2-2** Context Menu Options on Recent Projects

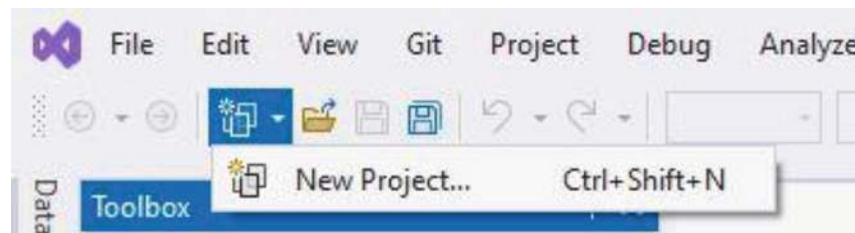
You can remove the project from the list, pin or unpin it, or copy the path to the project. Visual Studio 2022 has also made it relatively easy to get where you need to go when working on projects. As seen in Figure 2-3, developers have a few options available when they are ready to start working with code.



**Figure 2-3** Get Started Section

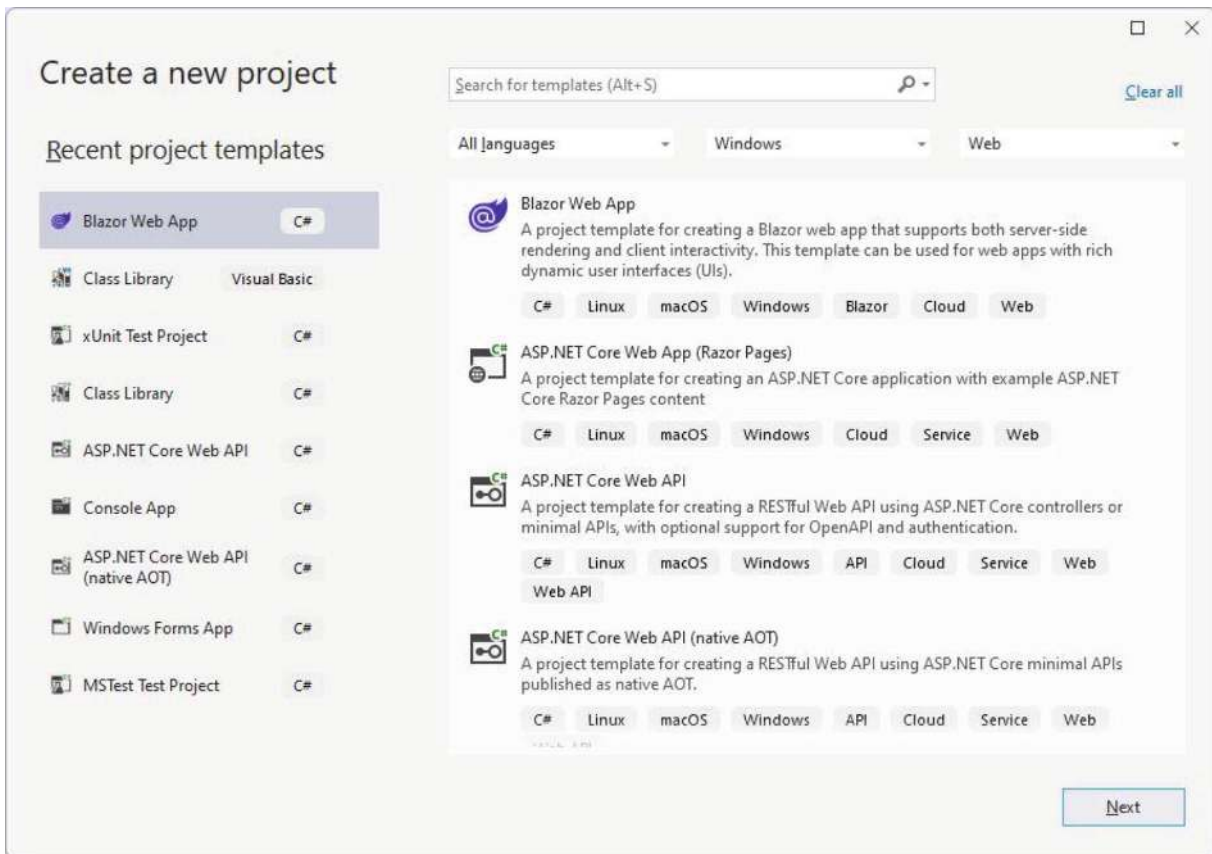
You can start by grabbing code from GitHub or Azure DevOps, opening a local Visual Studio project, opening a local folder to edit code files, creating a new project, or continuing without code.

If Visual Studio is already open, you can create a new project from the menu bar by clicking the New Project button, shown in Figure 2-4. You can also hold down Ctrl+Shift+N.



**Figure 2-4** New Project Toolbar Button

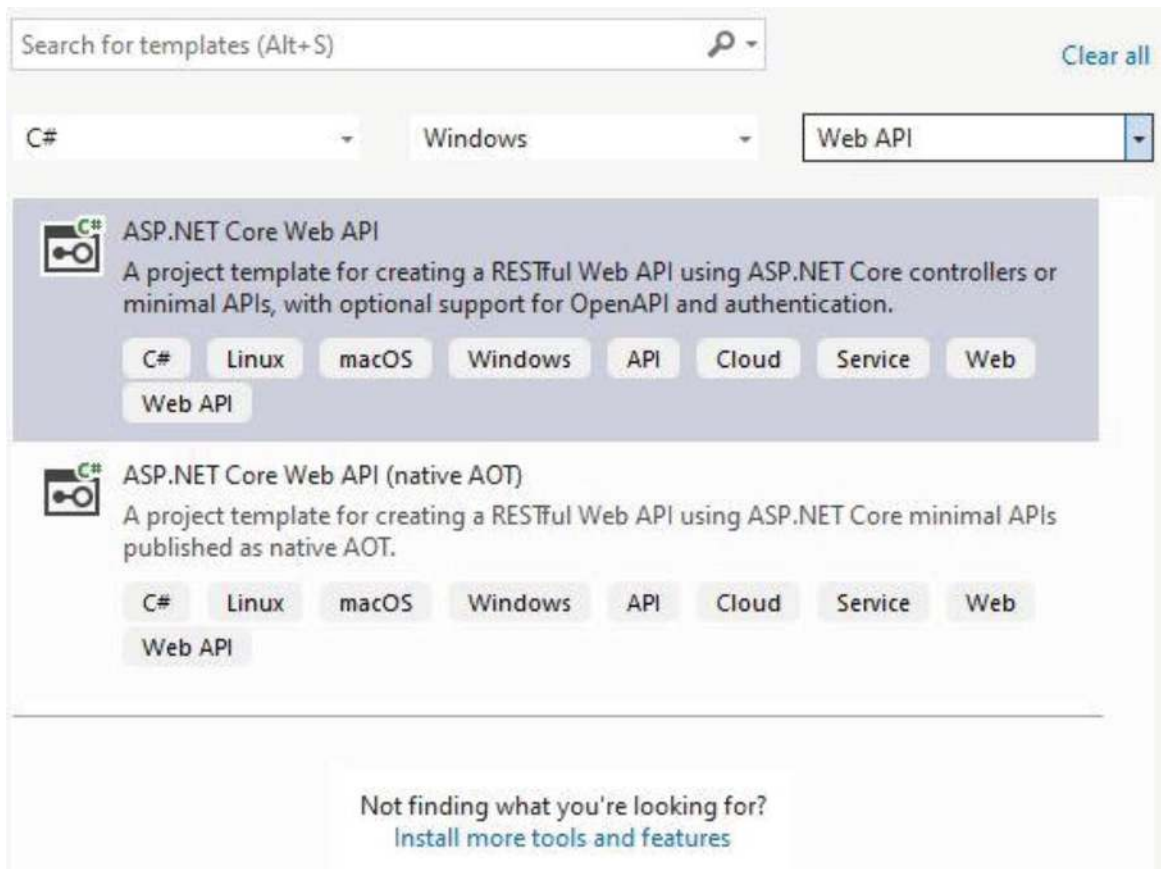
The Create a new project screen is displayed (Figure 2-5), and you have a whole new experience here, too, when it comes to finding the project type you want to create.



**Figure 2-5** Create a New Project

You can see recent project templates displayed, which is excellent, should you need to get up and running with similar projects as what you have created before. You can also search for and filter project templates by language, platform, or project type, shown in [Figure 2-6](#).





**Figure 2-6** Filter Project Templates

You are, therefore, able to quickly find the most suitable project template.

Take note, though, that you may need to install a workload if you do not find the project template you want. To do this, click [Install more tools and features](#). Refer to [Chapter 1](#) to see how to use workloads in Visual Studio.

Of the several project templates available, let's see which ones there are and what project is suitable for specific situations.

## Various Project Templates

You can choose from a host of project templates in Visual Studio 2022. I would even go as far as to say that it's now even easier to find the template you need to use due to the filters in the Create a new project window. Let's have a look at a few of these project templates next.

### Console Applications

I remember the first time I wrote a single line of code. I used a Console Application, a great template to use when you don't need a UI for your application. The Console Application project template running on the .NET Framework is displayed in [Figure 2-7](#).





**Figure 2-7** Console App (.NET Framework)

You will notice that this application is suited for running on Windows machines. But what if you need to run the Console Application across platforms such as Windows, Linux, and macOS? Here is where the second Console App template can be used as displayed in Figure [2-8](#).



**Figure 2-8** The Console App Template for Linux, macOS, and Windows

A good modern example of when a Console Application shines is for building lightweight automation tools. Think of a cross-platform deployment script that backs up configuration files, applies database migrations, and restarts services. Because a Console Application has no UI overhead and runs identically on Windows, Linux, or macOS, it's a perfect fit for scripting tasks and DevOps workflows where reliability and portability matter more than user interaction.

### ***Windows Forms Application***

In contrast to the Console Application, the Windows Forms application template is used when you need to create an app that has a UI and runs on Windows. The project template (like the Console Application) can run on the .NET Framework or the latest version of .NET as shown in Figure [2-9](#).



**Figure 2-9** Windows Forms App (.NET and .NET Framework)

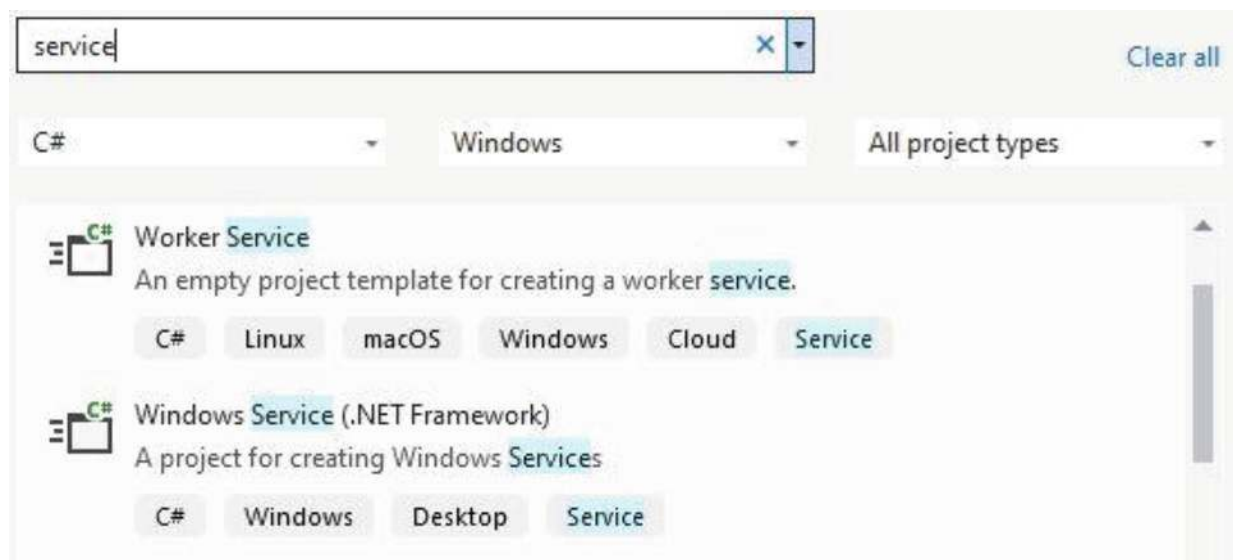
It is important to note that before .NET, there were two separate products: the .NET Framework and .NET Core. If you're part of the Old Guard, you will have worked extensively with the .NET Framework. With .NET Core, however, a process was set in motion that would change

the face of .NET development forever. Being open source, .NET Core was written from scratch and is designed to work across all platforms, such as Windows, Linux, and Mac.

With .NET 6.0 going forward, developers had just a single version of .NET (and it's just called .NET). Think of .NET 6 as the beginning of a unified development platform that allows developers to create desktop, cloud, web, mobile, gaming, IoT, and AI applications. As of this writing, Visual Studio 2022 allows you to choose .NET 6 and .NET 7, which are out of support, .NET 8, which has Long Term Support, and .NET 9, which has Standard Term Support. August 2025 will see the release of .NET 10.

## Services

If you ever need to create an application that continually runs in the background, performing some specific task, your best choice would be to use a Windows Service or a Worker Service template as seen in Figure 2-10.



**Figure 2-10** Windows Service (.NET Framework) and Worker Service

### Windows Service (.NET Framework)

A Windows Service running on the .NET Framework is an older project type that allows developers to build long-running background processes that run specifically on Windows.

You will choose this project when you're maintaining or extending legacy .NET Framework systems seeing as this will only run on Windows. In other words, a Windows Service is your go-to for older, Windows-bound projects.

### Worker Service

This is a modern, cross-platform template for building background services using .NET. It uses a generic host that is designed to run a Windows Service (when deployed with proper registration), as a Linux daemon, a Console app for local dev testing, or in a container such as Docker.

You will choose this template when you are building new services that might run on Windows, Linux, in containers, or in the cloud. You might also choose this template if you want to use dependency injection, configuration, logging, and lifecycle management. Finally, if the solution you want to provide needs to run cross-platform (Windows, Linux, macOS, containers, and cloud hosts), you will not go wrong with choosing the Worker Service template.

The screenshot shows a dialog box titled "Additional information" with a close button in the top right corner. Below the title is a tabbed interface with tabs for "Worker Service", "C#", "Linux", "macOS", "Windows", "Cloud", and "Service". The "Worker Service" tab is selected. The dialog contains the following configuration options:

- Framework**: A dropdown menu showing ".NET 8.0 (Long Term Support)".
- Enable container support**: A checked checkbox.
- Container OS**: A dropdown menu showing "Linux".
- Container build type**: A dropdown menu showing "Dockerfile".
- Do not use top-level statements**: A checked checkbox.
- Enable native AOT publish**: An unchecked checkbox.
- Enlist in .NET Aspire orchestration**: An unchecked checkbox.
- Aspire version**: A dropdown menu showing "9.0".

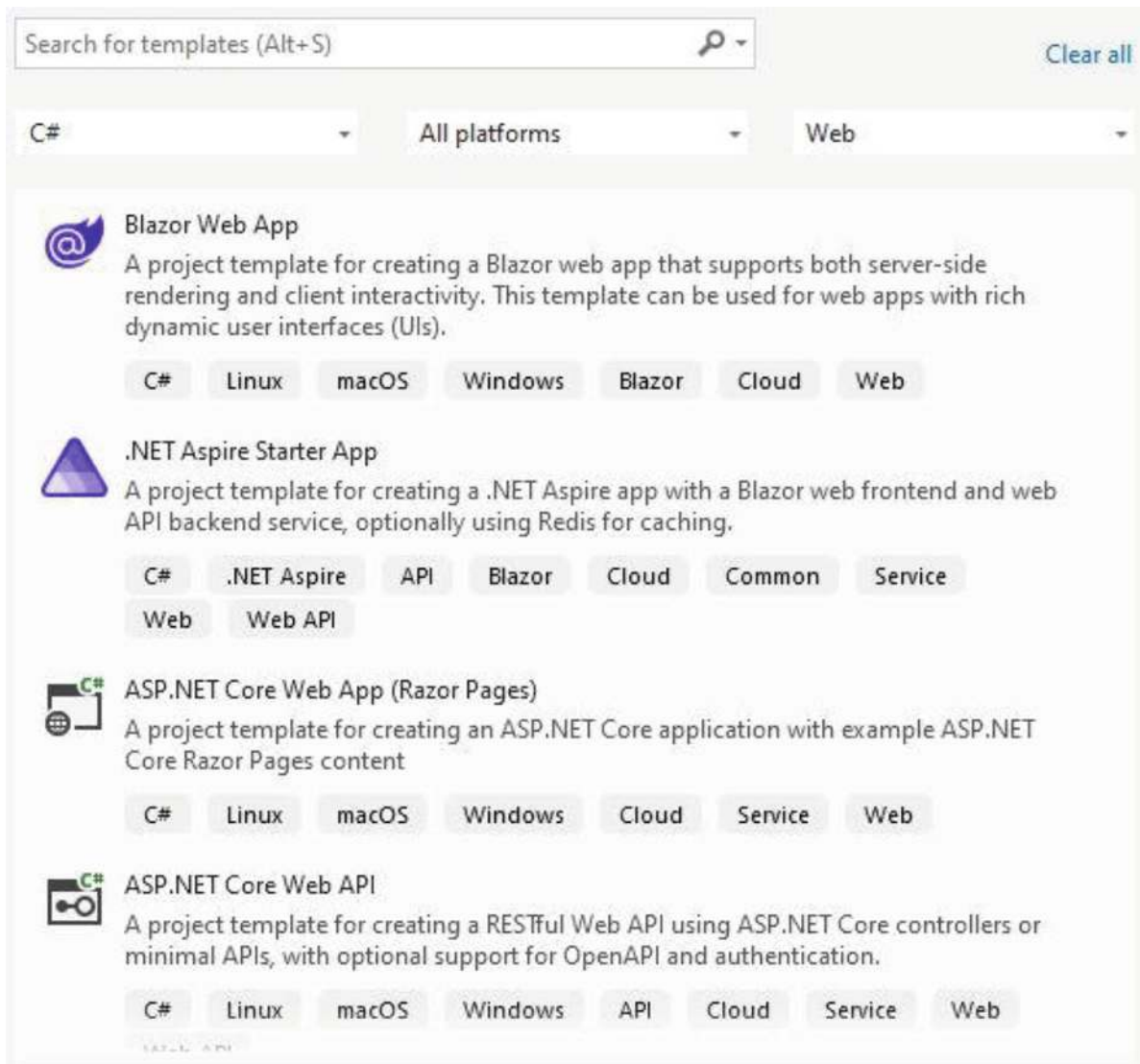
At the bottom right of the dialog are two buttons: "Back" and "Create".

**Figure 2-11** Creating a Worker Service

In short, a Worker Service is the modern, portable approach for any new background workload.

### ***Web Applications***

If you need to create web-based applications, you will see that you have various options to choose from (Figure 2-12). I will not cover .NET Aspire in this section. I will cover it in its own section as I feel that it warrants a focused discussion given its specific design goals. Looking at the choices, however, you might be wondering when to choose which project template.



**Figure 2-12** The Various Web Application Templates

Let's break this down a bit more and see when each project type might make more sense over the other.

### **Blazor Web App**

If you want to build interactive, client-side web applications using C# instead of JavaScript, the Blazor project is your template of choice. Blazor can run on the server (Blazor Server) or entirely in the browser via WebAssembly (Blazor WebAssembly). You will want to choose Blazor if you want to share logic and models between your front-end and back-end, and you prefer working fully in .NET rather than mixing in a JS framework like React or Angular.

### **ASP.NET Core Web App (Razor Pages)**

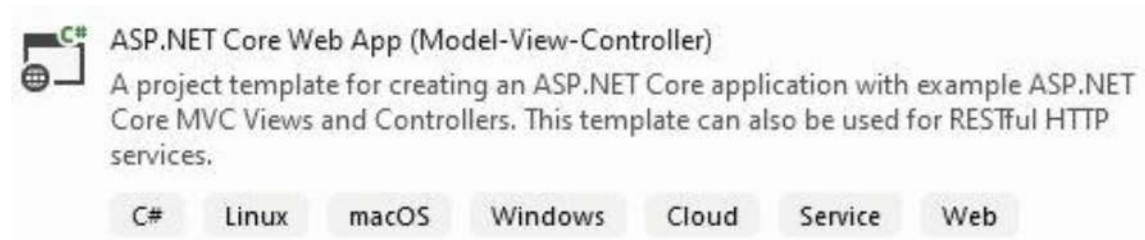
This is ideal for traditional, page-focused websites where each page handles its own logic. Razor pages keep page markup and associated C# logic tightly coupled and simple to maintain.

### **ASP.NET Core Web API**

If your primary goal is to expose RESTful endpoints for use by client applications, mobile apps, third-party consumers, or other services, then this is your template of choice. The Web API template is lean and purpose-built for creating HTTP APIs with JSON or other negotiation, without any front-end UI included.

### ***ASP.NET Core Web App (Model-View-Controller)***

If you want to use a Model-View-Controller approach to create your web application, you can select the ASP.NET Core Web App (Model-View-Controller) template, as shown in Figure [2-13](#).

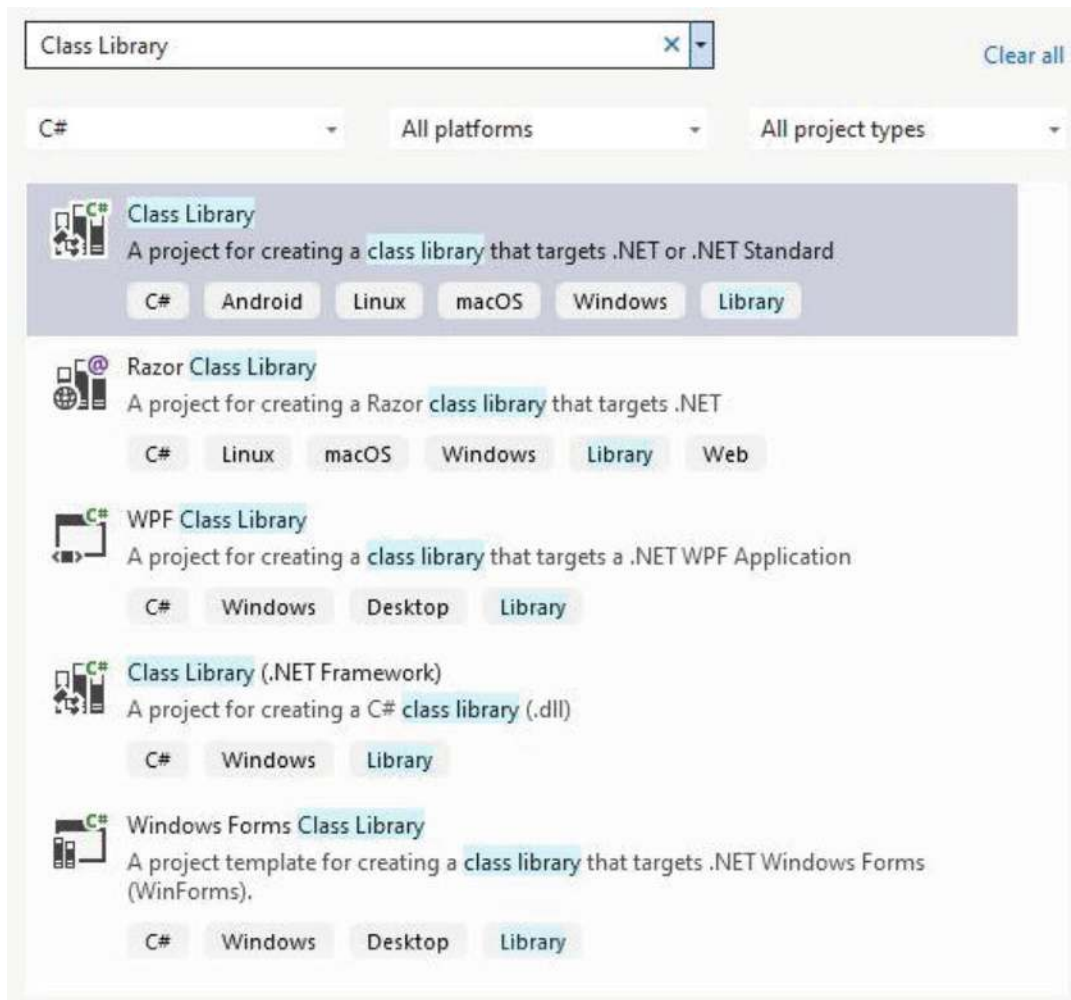


**Figure 2-13** ASP.NET Core MVC Web App Template

As seen in the template description, this template allows you to create a web application using Views and Controllers.

### ***Class Library***

The following project template we will look at is the Class Library. From Figure [2-14](#), you can see, the Class Library can be based on .NET, .NET Framework, or .NET Standard.



**Figure 2-14** Class Library Projects

Let's discuss the differences between the various class library projects.

### ***Class Library (.NET or .NET Standard)***

This is the go-to project type for general-purpose, reusable code that you want to share across different .NET workloads (e.g., ASP.NET Core, Blazor, Console Apps). To ensure the broadest compatibility with older projects, .NET Standard is a solid choice. For newer applications, however, the recommended choice here is .NET and is ideal for applications built on .NET 5 and beyond. This project type is generally used for code logic, utility functions, data access layers, or any shared code that is UI-agnostic.

### ***Razor Class Library***

If you need to bundle reusable Razor components, partial views, or pages that can be embedded in multiple web projects, you should pick this template. In fact, if you plan to distribute a UI component library, this is your template of choice. For more info on RCL browse to <https://learn.microsoft.com/en-us/aspnet/core/razor-pages/ui-class>.

### ***WPF Class Library***

If you need to write code specifically for a WPF (Windows Presentation Foundation) application, this template ensures that your library references the correct WPF assemblies. Choose this when



writing custom controls, converters, or behaviors in a WPF-only solution.

### ***Class Library (.NET Framework)***

This template exists for backwards compatibility, and you should only use it when you need to create a library that must run specifically on the .NET Framework such as 4.8 or older. Your use case here will be to support legacy WinForms, WPF, or ASP.NET apps that have not been migrated to .NET Core or .NET 5 or newer.

### ***Windows Forms Class Library***

The detail is in the name. This is scoped for Windows Forms apps, and it sets up the right project references and ensures seamless integration with WinForm-specific types. If you need to create custom WinForm controls or UI helpers for reuse across multiple WinForm apps, this is your safest pick.

### ***.NET Aspire***

The last project template we will look at is probably the most exciting, and we will see how to create a .NET Aspire application in the next section. The idea behind Aspire is to provide tools, templates, and packages to assist developers in building observable, production-ready apps. The promise of .NET Aspire is to simplify common challenges in modern application development, especially when these applications rely on multiple services such as databases, messaging, caching, etc.

To create a .NET Aspire application, search for and select the .NET Aspire Starter App as seen in Figure [2-15](#).



**Figure 2-15** The .NET Aspire Template

In the next section, I will go through creating a .NET Aspire project, viewing the .NET Aspire dashboard, adding .NET Aspire to existing applications, adding .NET Aspire Packages, and adding SQL Server to your .NET Aspire project.

---

## **Creating a .NET Aspire Project**

Creating a new project is the first step in experiencing the capabilities of .NET Aspire. In this section, we will discuss the prerequisites, using Visual Studio or CLI templates, and take a look at the structure of the created .NET Aspire solution. By the end of the Aspire section of this chapter, you will have a running Aspire app with front-end, back-end, and integrated services.

### **Prerequisites and Installation**

Before you can create a .NET Aspire project, you need to ensure that your development environment is set up with the required tools. These are:

- .NET SDK – Ensure that you have .NET 8.0 or .NET 9.0 installed.



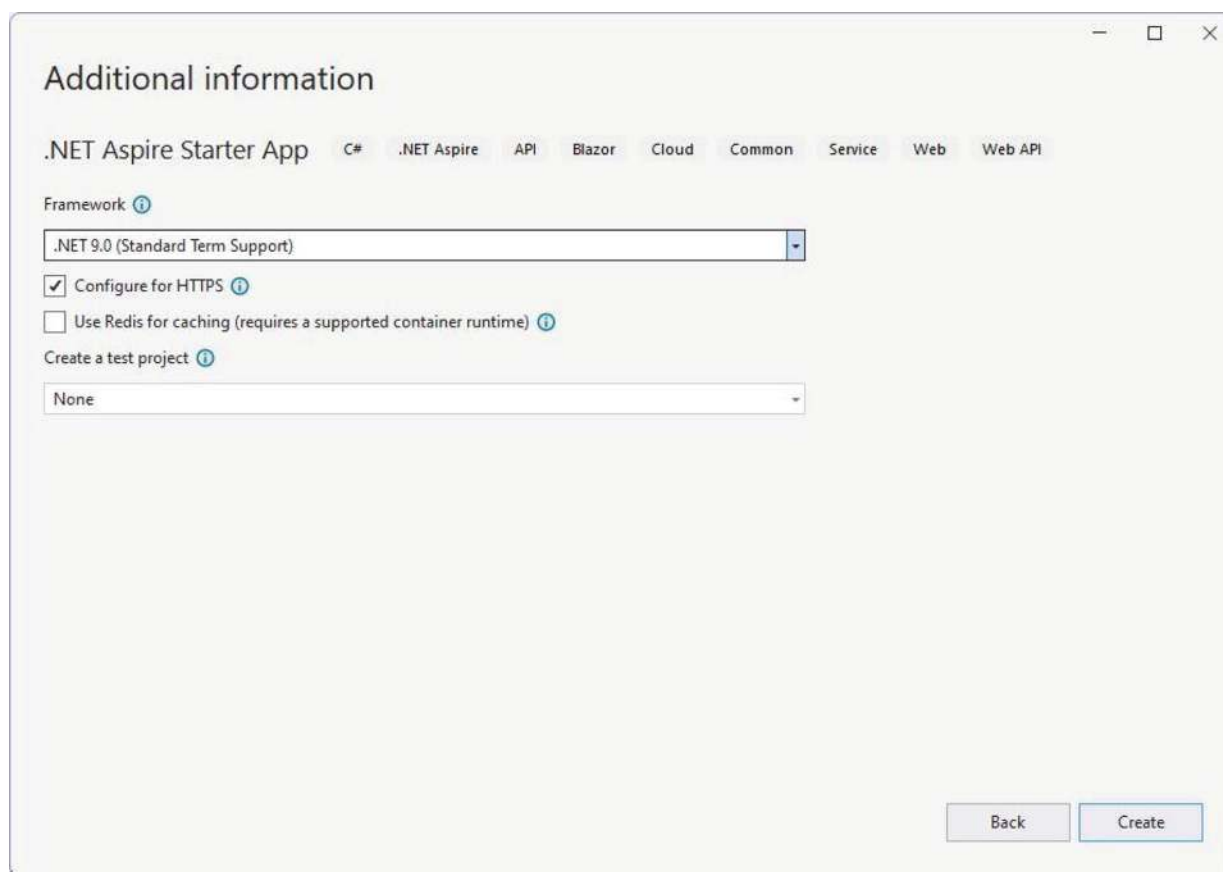
- Container Runtime – Install an OCI-compliant container runtime such as Docker or Podman.
- IDE or Editor – You can use Visual Studio 2022 (v17.9+) or Visual Studio Code (with the C# Dev Kit extension installed) or JetBrains Rider (with the Aspire Plugin). Note that Visual Studio 2022 version 17.10 and up already includes the .NET Aspire SDK component as part of the ASP.NET and web development workload.
- .NET Aspire Workload – If you prefer to use the CLI, you can install the Aspire workload by running `dotnet workload update` and then `dotnet workload install aspire`.

## Using Visual Studio 2022

After ensuring that your development environment is set up with the required tools, you can follow the following steps to create a new Aspire app using the New Project template. From the Visual Studio 2022 launch screen, create a new project. If Visual Studio is already open, go to File ► New ► Project. Search for Aspire and select the .NET Aspire Starter App template and click Next. On the next screen, give your project a name (I called mine HelloAspire) and select a location for your project, and click Next.

From the Additional Information screen, as seen in Figure 2-16, choose the target framework .NET 9.0 (STS). You will also see options to include certain integrations:

- Use Redis for caching – Check this option to include a Redis cache integration. This will require a supported container runtime such as Docker or Podman.
- Create a test project – You can optionally include a test project.

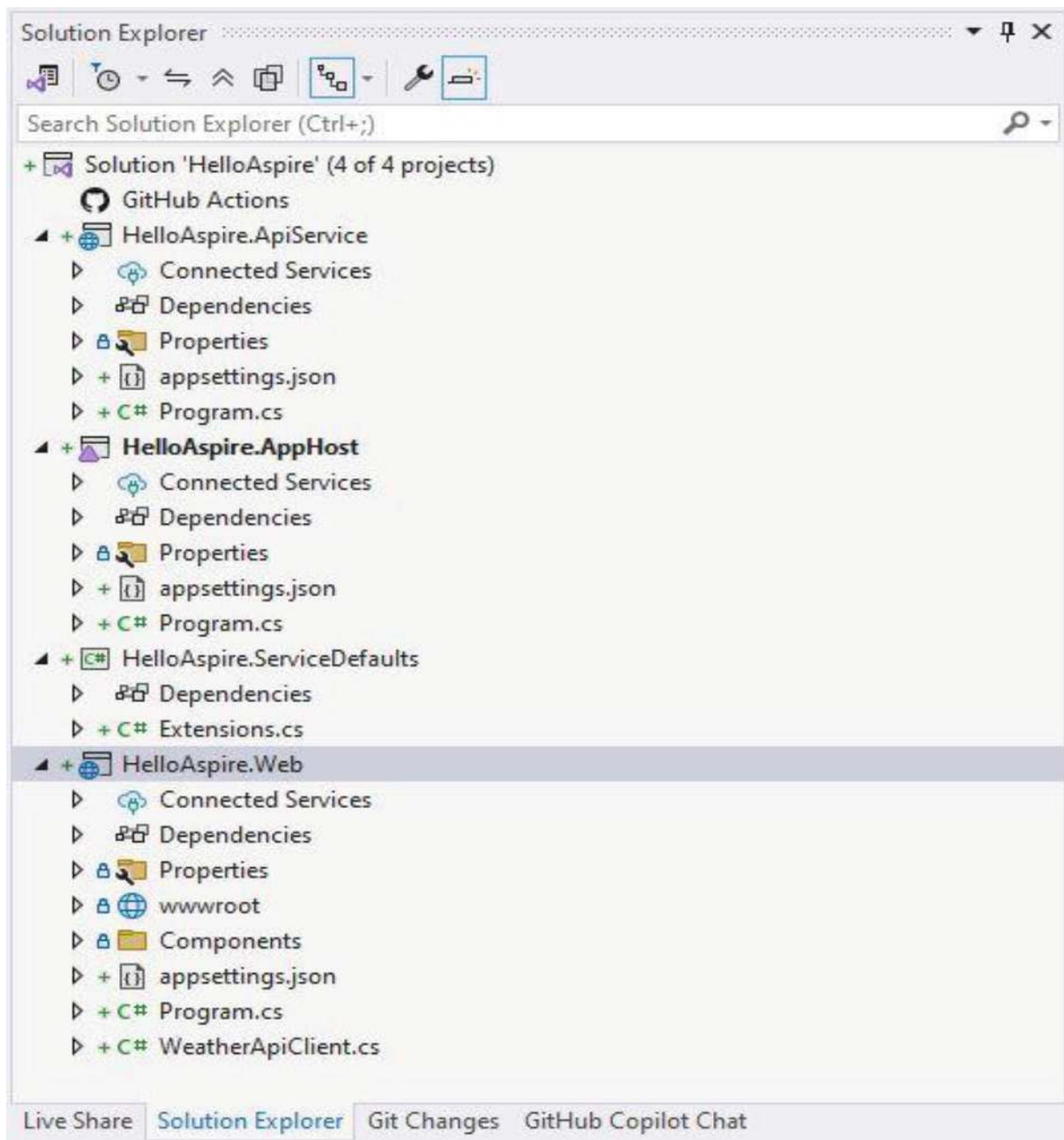


**Figure 2-16** The Additional Information Screen

A word on supported containers; as of this writing, .NET Aspire can run containers using several OCI-compatible runtimes. This includes Docker Desktop and Podman. There have been reports of developers successfully using Rancher Desktop, especially when configured to use the Docker CLI, but this is not an officially supported scenario.

When you have configured the Additional Information, click Create to finish. Visual Studio will create a new solution pre-configured for .NET Aspire, complete with the required projects and NuGet packages.

Once completed, the solution typically contains a few projects as seen in Figure 2-17.



**Figure 2-17** The Aspire Solution

The Aspire solution we created includes:

- `HelloAspire.AppHost` – This is the orchestration project that will start and wire up all the components.
- `HelloAspire.ServiceDefaults` – This is a shared project for common configuration such as resilience, telemetry, etc. across services.
- `HelloAspire.Web` and `HelloAspire.ApiService` – These are simply just a Blazor project and minimal API project.

The projects in this solution are structured using Aspire conventions. We'll explore their unique roles shortly, but it is interesting to note that Visual Studio automatically sets the `AppHost` as the startup project. This means that the solution will launch the orchestrator by default.

## The Solution Structure

When you create an Aspire solution using the .NET Aspire Starter App template, you will notice that the solution is organized into multiple projects, each with specific roles. Understanding these roles will help you navigate the code.

### *AppHost Project*

This is the heart of Aspire's orchestration. It contains a `Program.cs` file that makes use of the `DistributedApplication` builder APIs to define the "app model" (in other words, which projects and resources your application uses), and then runs them.

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

var cache = builder.AddRedis("cache");
var apiService = builder.AddProject<Projects.HelloAspire_ApiService>
("apiservice")
    .WithReference(cache).WaitFor(cache);

builder.AddProject<Projects.HelloAspire_Web>("webfrontend")
    .WithExternalHttpEndpoints()
    .WithReference(cache).WaitFor(cache)
    .WithReference(apiService).WaitFor(apiService);

builder.Build().Run();
```

**Listing 2-1** The `Program.cs` File

The code, as seen in Listing [2-1](#), is doing a few important things. These are:

- `DistributedApplication` - which is an implementation of the `Microsoft.Extensions.Hosting.IHost` interface that orchestrates a .NET Aspire application.
- `AddRedis("cache")` – this includes a Redis cache with the name "cache" that will spin up a Redis container when your app runs.
- `AddProject<...>("apiservice")` – This adds the existing API project into the Aspire app model and names it "apiservice".
- `AddProject<...>("webfrontend")` – This adds the Blazor project named "webfrontend" and chains `.WithReference(cache)` and `.WithReference(apiService)` along with `.WaitFor` on each one. The `.WithReference` method is used to link a resource or service to the project, ensuring that

necessary information (like connection strings or service discovery URLs) are injected for those projects. The `.WaitFor` method ensures that those referenced resources (the API service and Redis cache) are up and running before the Web UI starts.

- `Builder.Build().Run()` – The final step in the process, starts the orchestrator and launches the projects and containers.

The `AppHost` project essentially defines how all the parts of our distributed app come together in development. In our project, for Listing [2-1](#), it orchestrates a Redis container, an API service, and a Web application, handling their configurations and startup order automatically.

### ***ServiceDefaults Project***

This shared class library contains common configuration for resilience and observability. Looking at the project file (right-click the project and click on Edit Project File from the context menu), you will notice that it is marked with

`<IsAspireSharedProject>true</IsAspireSharedProject>`. This class typically contains packages for OpenTelemetry, logging, Service Discovery, and Resilience libraries.

The running `AppHost` ensures that each app project references the `ServiceDefaults` and in doing so, ensuring consistent settings across all. An example of this would be enabling structured logging, distributed tracing, and health checks by default.

As seen in Listing [2-2](#), the `ServiceDefaults` adds an extension method called `AddServiceDefaults()` to apply all the defaults it defines to an app builder.

```
public static TBuilder AddServiceDefaults<TBuilder>(this TBuilder
builder) where TBuilder : IHostApplicationBuilder
{
    builder.ConfigureOpenTelemetry();
    builder.AddDefaultHealthChecks();
    builder.Services.AddServiceDiscovery();

    builder.Services.ConfigureHttpClientDefaults(http =>
    {
        // Turn on resilience by default
        http.AddStandardResilienceHandler();

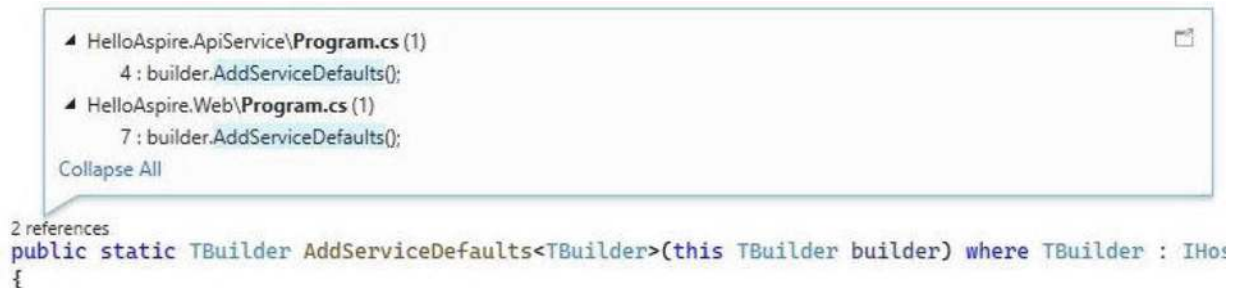
        // Turn on service discovery by default
        http.AddServiceDiscovery();
    });

    // Uncomment the following to restrict the allowed schemes for
    service discovery.
    // builder.Services.Configure<ServiceDiscoveryOptions>(options
=>
    // {
    //     options.AllowedSchemes = ["https"];
    // });

    return builder;
}
```

**Listing 2-2** The `AddServiceDefaults` Extension Method

This means that in practice, each app (the API and Web app in our solution) will call `builder.AddServiceDefaults()` in its Program file before adding its own services. You can see this in Figure 2-18, where clicking on the references will show you the applications referencing the `AddServiceDefaults()` extension method.



**Figure 2-18** The `AddServiceDefaults` References

This means that cross-cutting concerns are uniformly set up throughout your applications.

## Application Projects

These projects are the Blazor application (HelloAspire.Web) and the minimal API project (HelloAspire.ApiService), as previously seen in Figure 2-17.

### HelloAspire.Web

The web frontend is a Blazor server project. This is the UI of your solution. It uses an `HttpClient` to communicate with the API project (HelloAspire.ApiService). Looking at the `Program.cs` file, you will notice that it calls the `builder.AddServiceDefaults()` and then the Redis integration for Output caching with `builder.AddRedisOutputCache("cache")` since we included Redis when we created our Aspire project.

The `Program.cs` also configures `HttpClient.BaseAddress` to enable communication with the API via service discovery. Interestingly, note the URL used here, which is `https+http://apiservice`. This is a special scheme supported by .NET Aspire's service discovery. It means:

- Prefer HTTPS – The client will try to connect to `https://apiservice` first.
- Fallback to HTTP – If HTTPS is not available, it will try `http://apiservice`.

This approach is useful in Aspire environments where services may be available over both HTTP and HTTPS, and you need to ensure that secure connections are preferred but will allow fallback for development or local scenarios.

### HelloAspire.ApiService

This is a minimal API project that provides data access (in our solution, the boilerplate weather forecast service). It too calls the `builder.AddServiceDefaults()` as well as the Redis cache. Because the AppHost linked the Redis resource to this API project, the API can find the correct connection string automatically by referencing the name "cache".

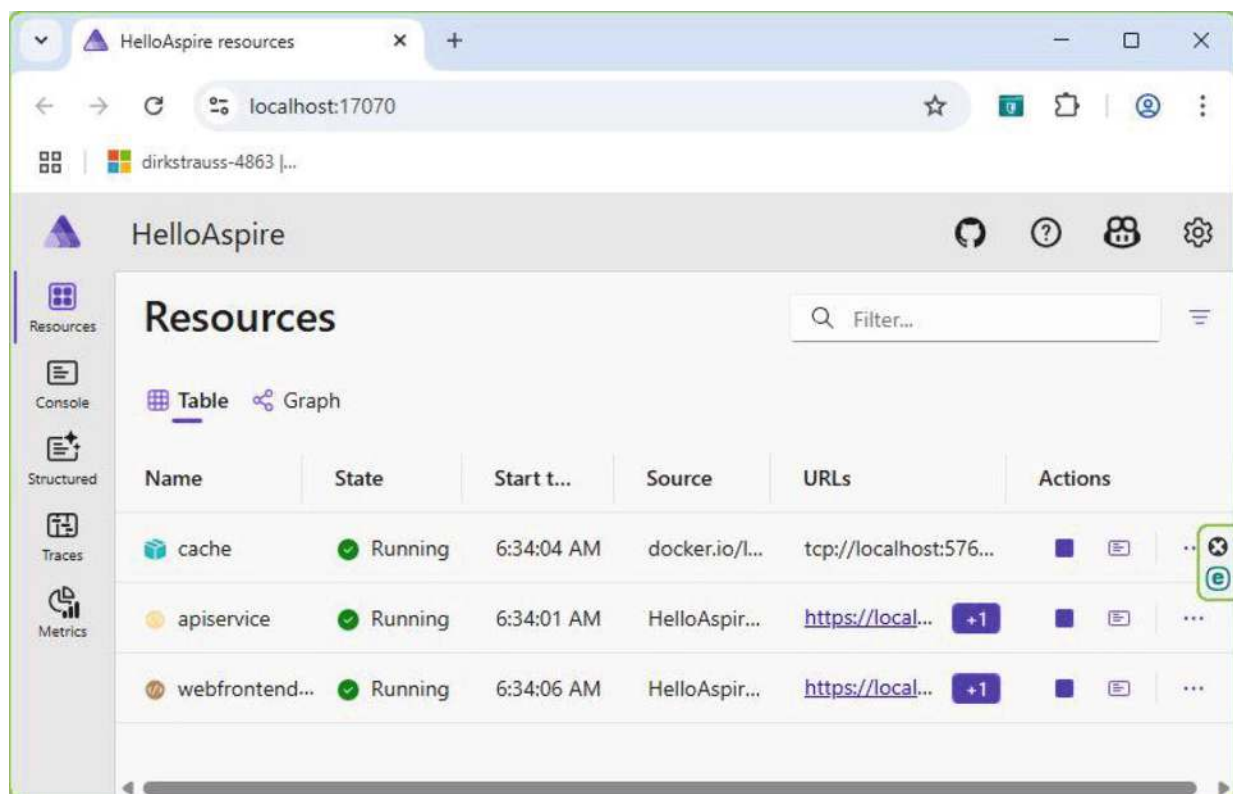
Both API and Web projects will be launched by the AppHost instead of running on their own. Both projects are enriched with environment variables or configurations by Aspire. The web project, for example, doesn't need the hardcoded API URL. It simply needs to call `http://apiservice`, and Aspire's service discovery takes care of the rest.

## Application Projects in Summary

The .NET Aspire Starter application creates a multi-project solution where the AppHost orchestrates a couple of services and a cache (in our example). Doing this sets the stage for the rapid development of complex, distributed applications on your local machine without manually configuring each one. As the .NET team describes, Aspire “simplifies common challenges in modern app development” by handling service wiring, connection strings, and reducing dev-time toil by setting up infrastructure for you.

## Running and Testing the Aspire Project

After you have created your project, run it locally to see if everything works. In Visual Studio, ensure that your `<YourApp>.AppHost` project (`HelloAspire.AppHost` in our example) is set as the startup project (which by default, it should be). Then press F5 to run with the debugger (or Ctrl+F5 without the debugger). Visual Studio will build all projects and start the AppHost. This will spin up any container resources (Note that Docker must be running).



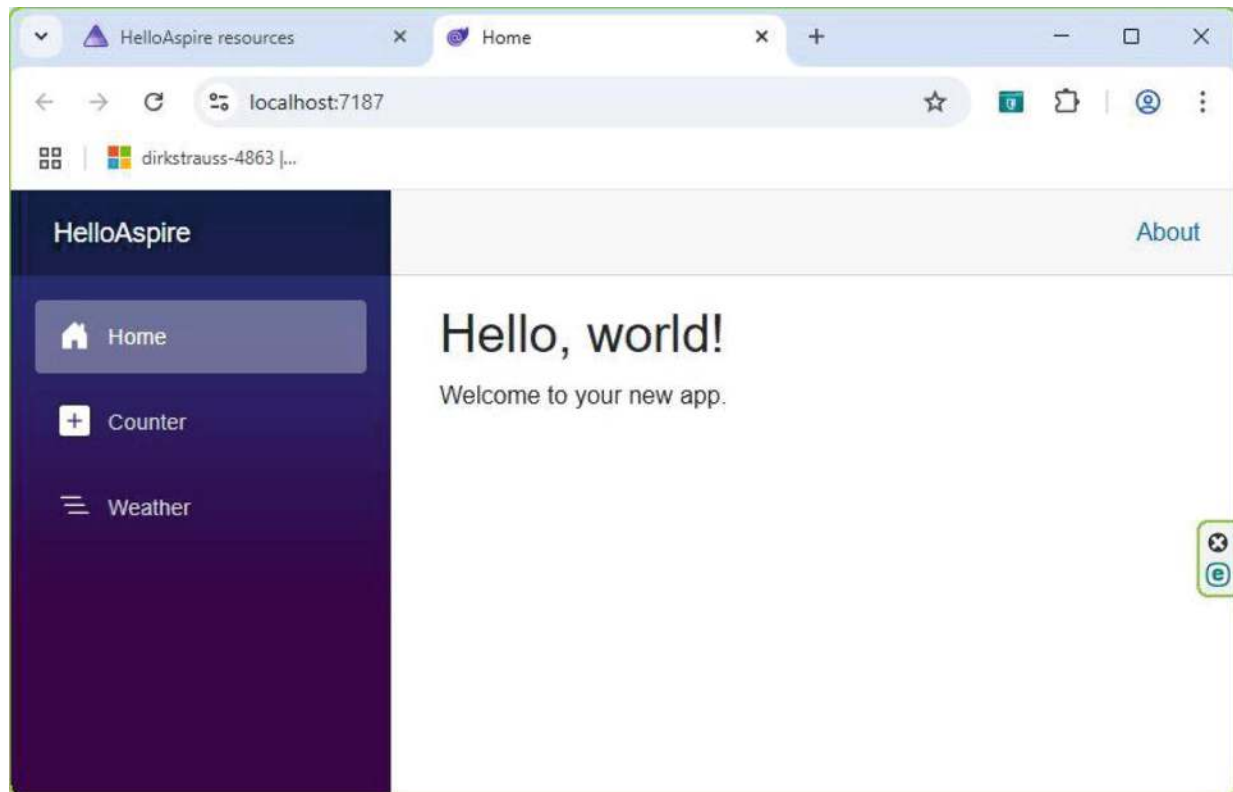
**Figure 2-19** The Aspire Dashboard

In our example, we included Redis in our project, so you’ll see Docker pull and start a Redis container. Once this is done, the dashboard will automatically open in your browser shown in Figure 2-19. The dashboard provides a UI to allow you to monitor your application. In the dashboard’s Resources list (seen in Figure 2-19) you will see entries for “apiservice” and “webfrontend” which are the two Application projects. You will also see “cache” which is the Redis container. Each resource shows its state (Running) and its available URLs.

In the Aspire dashboard, click on the URL for the `webfrontend` project. This will open the front-end app in a new tab in your browser (Figure 2-20).

You can navigate around the default web application to get a feel for how it works. The point here is that it works like this out of the box. Clicking on the Weather tab will call the API wired

up to the Blazor app by Aspire. Again, this just works without you having to wire anything up manually.



**Figure 2-20** The Running Blazor Application

You can also click on your API URL and add `/weatherforecast` to the end of the URL to see the returned JSON from the API service, as seen in Listing 2-3.

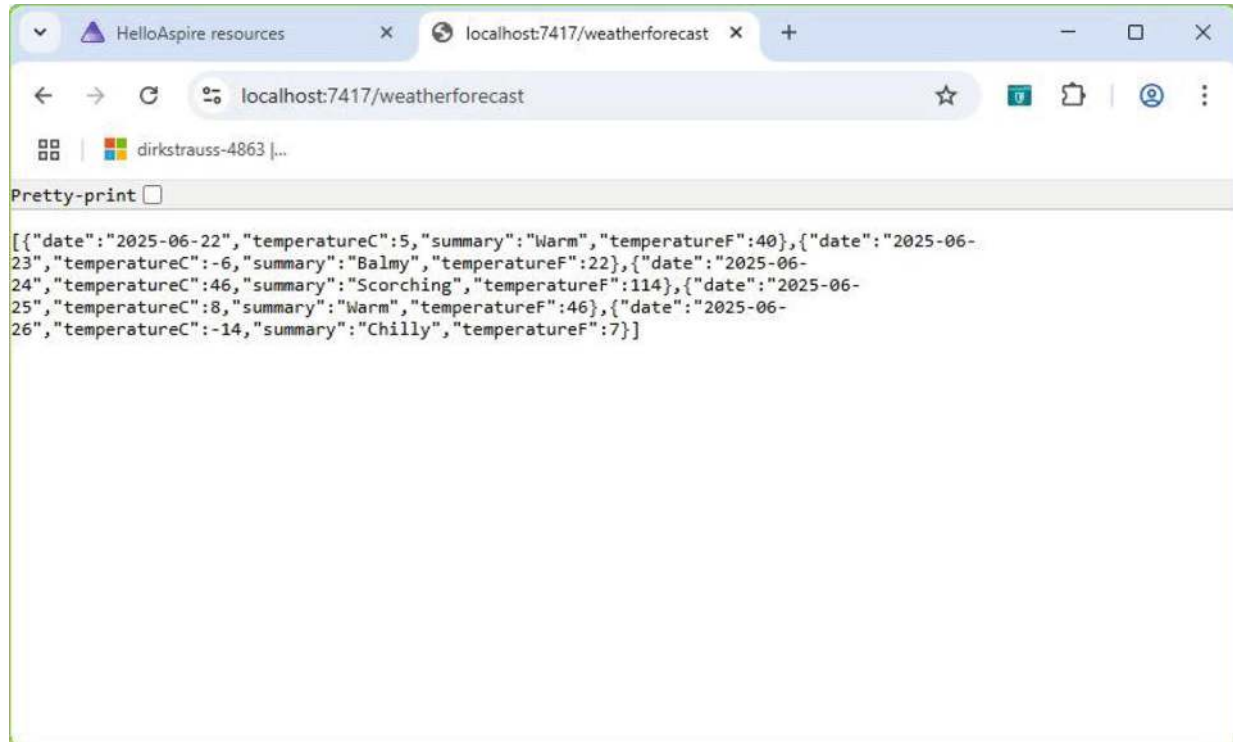
```
[
  {
    "date": "2025-06-22",
    "temperatureC": 5,
    "summary": "Warm",
    "temperatureF": 40
  },
  {
    "date": "2025-06-23",
    "temperatureC": -6,
    "summary": "Balmy",
    "temperatureF": 22
  },
  {
    "date": "2025-06-26",
    "temperatureC": -14,
    "summary": "Chilly",
    "temperatureF": 7
  }
]
```



]

**Listing 2-3** Example of Returned Weatherforecast JSON

In your browser, the JSON will simply be output as shown in Figure [2-21](#).



**Figure 2-21** The Weatherforecast API

What this shows us is that your API and Web Application are functioning correctly. You have successfully created and run your first .NET Aspire solution. You now have a feel for how Aspire automates local orchestration of multiple services.

Next, let's have a look at the Aspire dashboard in a bit more detail.

## Viewing the .NET Aspire Dashboard

One of the standout features of .NET Aspire is its dashboard. If you refer back to Figure [2-19](#), you will see the running Aspire dashboard. This dashboard provides a real-time monitoring UI for your application. It provides deep insights into all the running projects and resources. In fact, if you look at the toolbar as seen in Figure [2-22](#) you will notice that it has Copilot baked in.



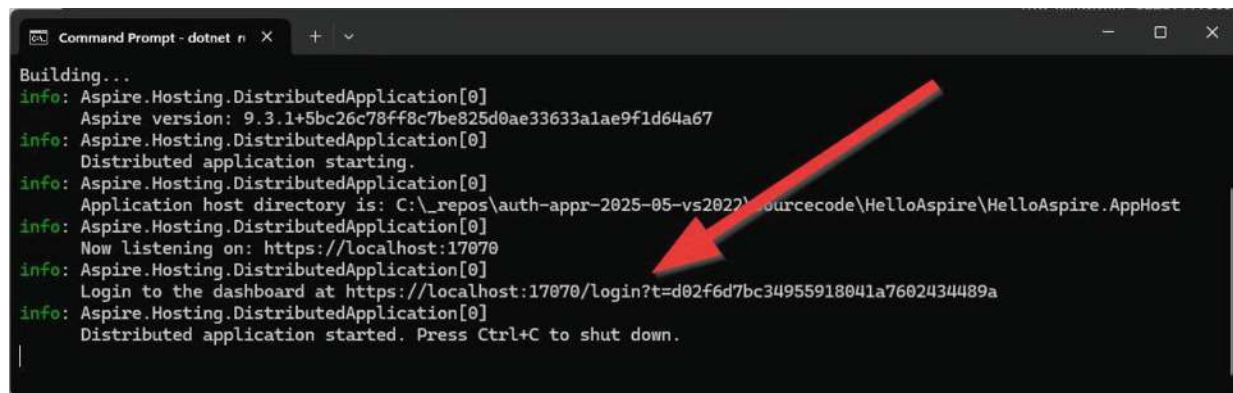
**Figure 2-22** Copilot in the Aspire Dashboard

The dashboard gives you all the tools you need to view settings, logs, traces, metrics, start or stop services, and use Copilot to help you diagnose issues.

## Dashboard Overview

The Aspire dashboard is essentially just a web application that the AppHost launches on startup. It runs locally and requires an authentication token for security. Starting the AppHost from Visual Studio allows the tooling to automatically pass the auth token to the dashboard page and log you in. This means that the dashboard will just appear without any login prompts for the typical F5 developer scenario.

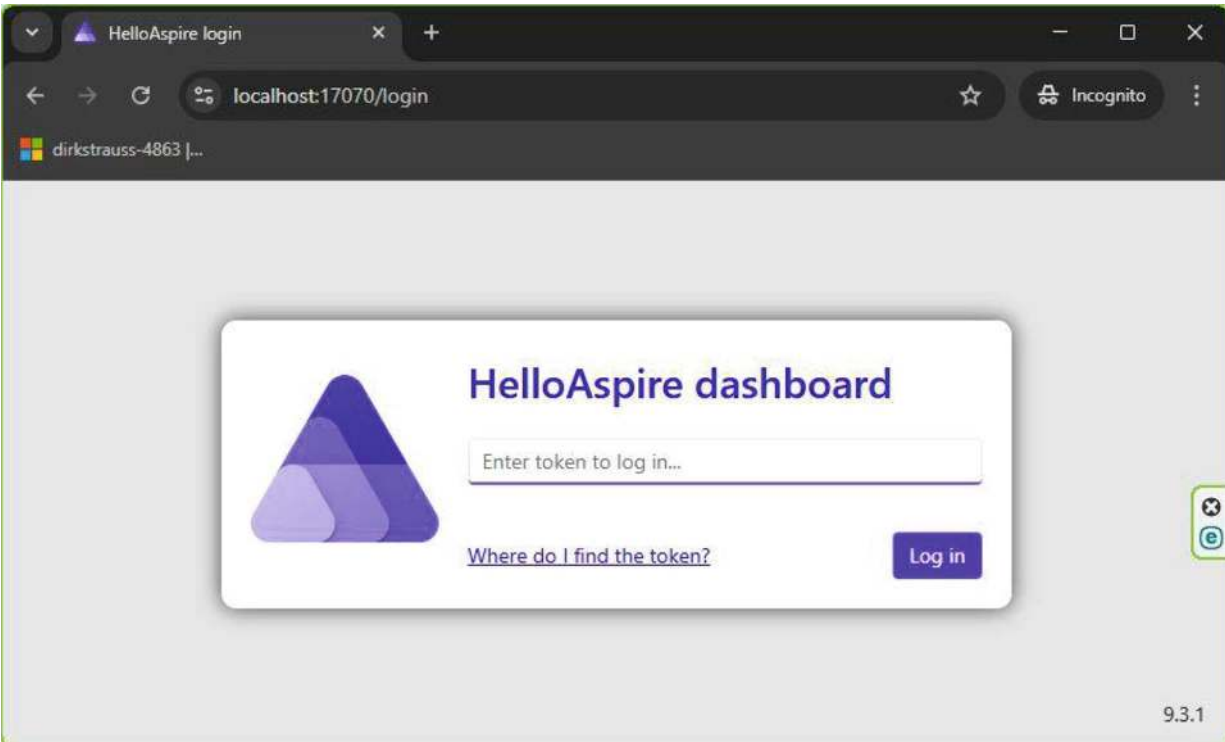
If, however, you open a terminal and navigate to the AppHost directory and type in the `dotnet run` command, you will see the application launch and an authentication link with a token in the console (Figure 2-23). Incidentally, you see the same in the console that launches when you debug via Visual Studio F5. In Figure 2-23 below, you will notice that my login page is at `https://localhost:17070/login`.



```
Building...
info: Aspire.Hosting.DistributedApplication[0]
      Aspire version: 9.3.1+5bc26c78ff8c7be825d0ae33633alae9f1d64a67
info: Aspire.Hosting.DistributedApplication[0]
      Distributed application starting.
info: Aspire.Hosting.DistributedApplication[0]
      Application host directory is: C:\_repos\auth-app-2025-05-vs2022\Sourcecode\HelloAspire\HelloAspire.AppHost
info: Aspire.Hosting.DistributedApplication[0]
      Now listening on: https://localhost:17070
info: Aspire.Hosting.DistributedApplication[0]
      Login to the dashboard at https://localhost:17070/login?t=d02f6d7bc34955918041a7602434489a
info: Aspire.Hosting.DistributedApplication[0]
      Distributed application started. Press Ctrl+C to shut down.
```

**Figure 2-23** The Console Window

Copy the login URL without the token query parameter and paste it into a new browser window (Figure 2-24).



**Figure 2-24** The Aspire Login Page

The Aspire login page will be displayed. Copy the token from the login URL and paste it into the login page. This will allow you to log in to the Aspire dashboard. Once the token is used to log in to the dashboard, a persistent login cookie is set. This means that you won't need to re-enter the token for a few days. This is an important security measure because the dashboard displays sensitive information such as connection strings and secrets in environment variables. Only authorized local users (you the developer) are allowed to see it.

So to sum things up, to access the dashboard, you simply have to run your .NET Aspire app. The dashboard opens automatically. If it does not open up automatically (if, for example, you're using `dotnet run`), look for the login URL in the console and login to Aspire like that. Once logged in, you will see an interface with a left navigation menu to Resources, Console, Structured (which are logs), Traces, and Metrics. Let's have a look at each in turn.

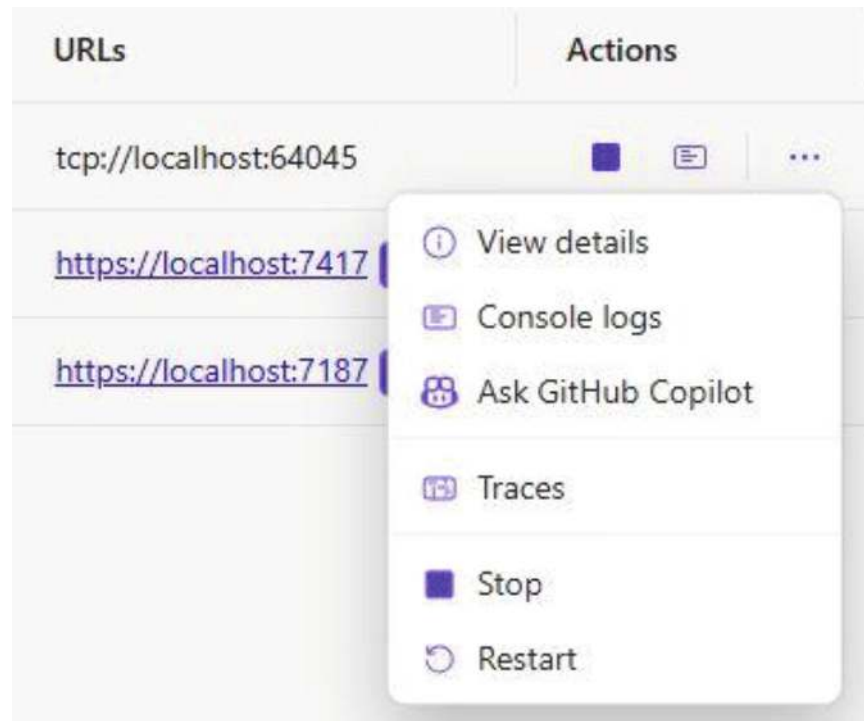
## Resources Page

This page gives you a bird's eye view of all the components that make up your .NET Aspire application (Figure 2-19). Each resource, be that a project or a container, is listed in a table. This table shows:

- **Name** – The name identifier of the resource as defined in your code. In our example, "apiservice", for example, corresponds to what you passed to `AddProject` in the `AppHost`.
- **State** – This indicates if the resource is running, stopped, or in an error state. If there are any errors, you will see a red icon with the error count. You can click on that to be taken directly to the error logs for that resource.
- **Start time** – This is the date and time when the resource started.
- **Source** – This is the path or origin of the resource on disk. For projects, you will see the path to the folder, and for containers, you will see the image info.

- URLs – If your project is running, this will list the URL where that project is listening. You can see multiple URLs here if the project has multiple endpoints. Clicking on the endpoint will open the service in the browser.
- Actions – Here you will see buttons to stop or start a resource. You can also access the Console Logs from here as well as click on the ellipses ( . . . ) that display the full Actions menu, allowing you to View Details, Console logs, Ask GitHub Copilot, access Traces, Stop and Restart the service.

The Stop button will obviously only show if the resource is in a running state. You can see this menu in Figure [2-25](#).

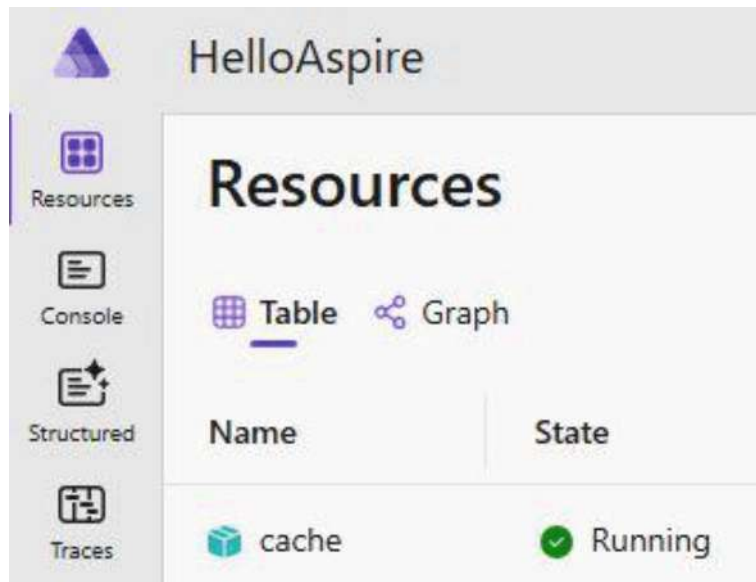


**Figure 2-25** The Actions Menu

Referring back to Figure [2-19](#), you will see that we have two project resources and a container resource, all of which are running. If you wanted to simulate a failure, you could stop the Redis container and your web application/API will start logging the errors until you restart the container again.

### **Graph View**

Easily overlooked, the Graph view is accessed by clicking on the Graph menu located next to the Table menu (or tab), as seen in Figure [2-26](#).



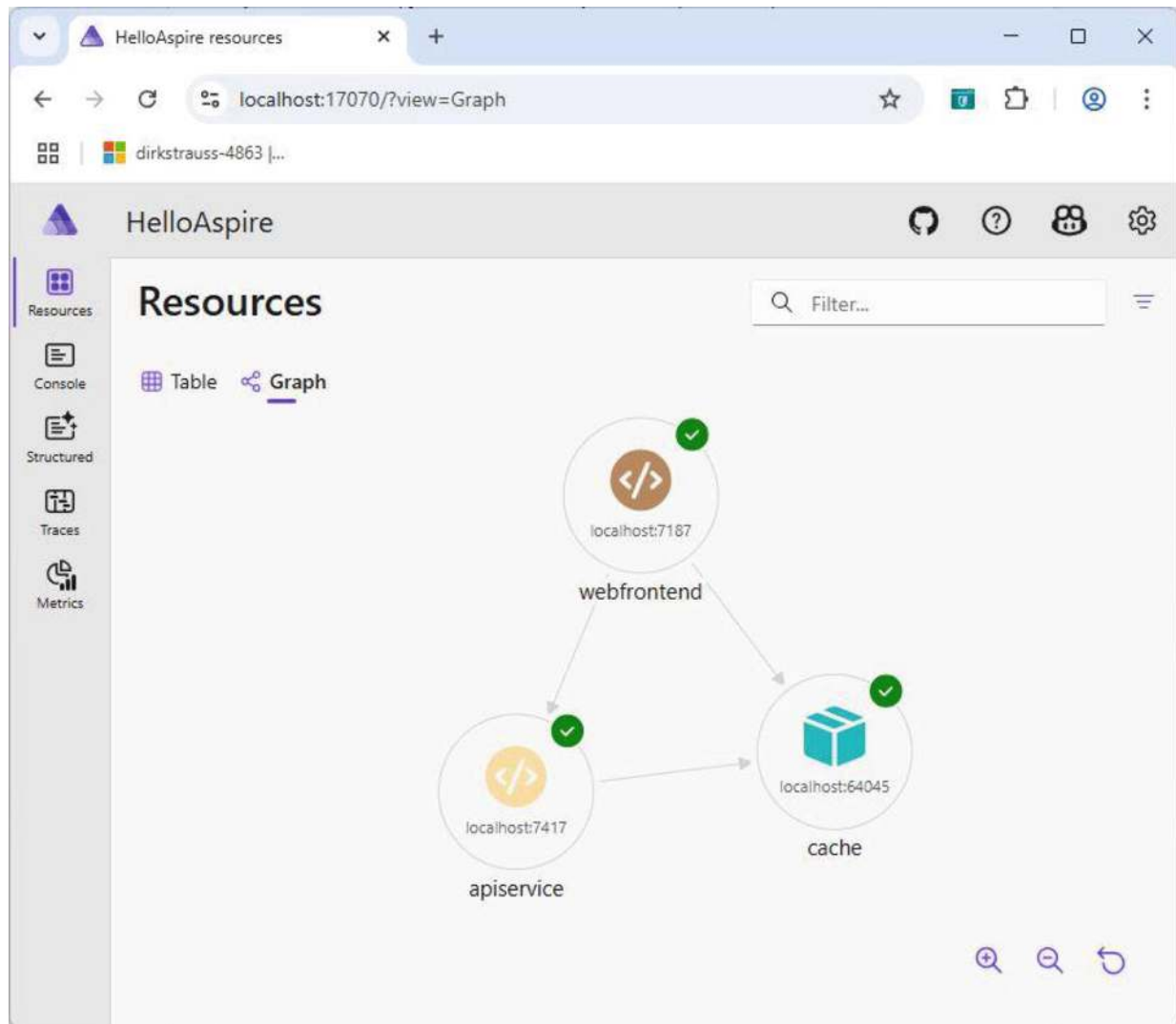
**Figure 2-26** The Graph Menu

The graph gives you a view of your application’s architecture. As seen in [Figure 2-27](#), in graph mode, you will see boxes or nodes for each project and resource.

The lines between the boxes and nodes indicate their dependencies. In our example, the arrows from the `webfrontend` resource to the `apiservice` and `cache` show that it is dependent on them.

This visualization helps us understand how the pieces connect. You can also click on a node in the graph to highlight it and see additional details.

You can also zoom or pan around the graph as required. The graph in our example might not look all too impressive, but it becomes especially useful when your application grows and becomes more complex. Seeing all the resources and services in one diagram comes to you for free, out of the box, with .NET Aspire.



**Figure 2-27** The Graph View

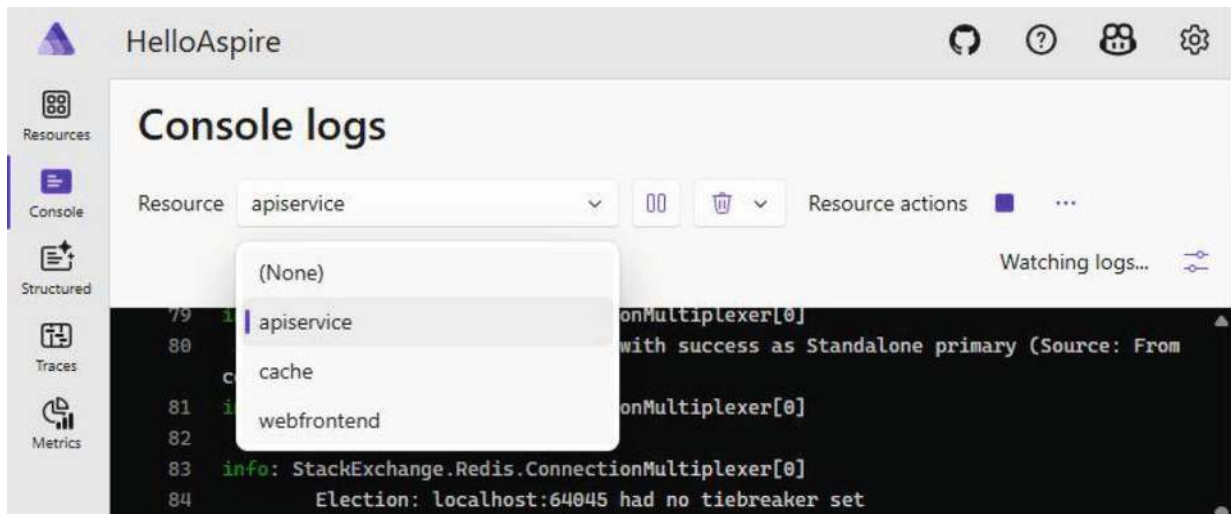
But .NET Aspire doesn't stop there. You have rich telemetry and logs at your disposal. Let's look at these next.

## Viewing Logs and Telemetry

One major advantage of the Aspire dashboard is the consolidated observability. From one single dashboard, you have access to a rich set of data about the resources in your solution. The rest of the menus in the left navigation allow us to view logs and telemetry.

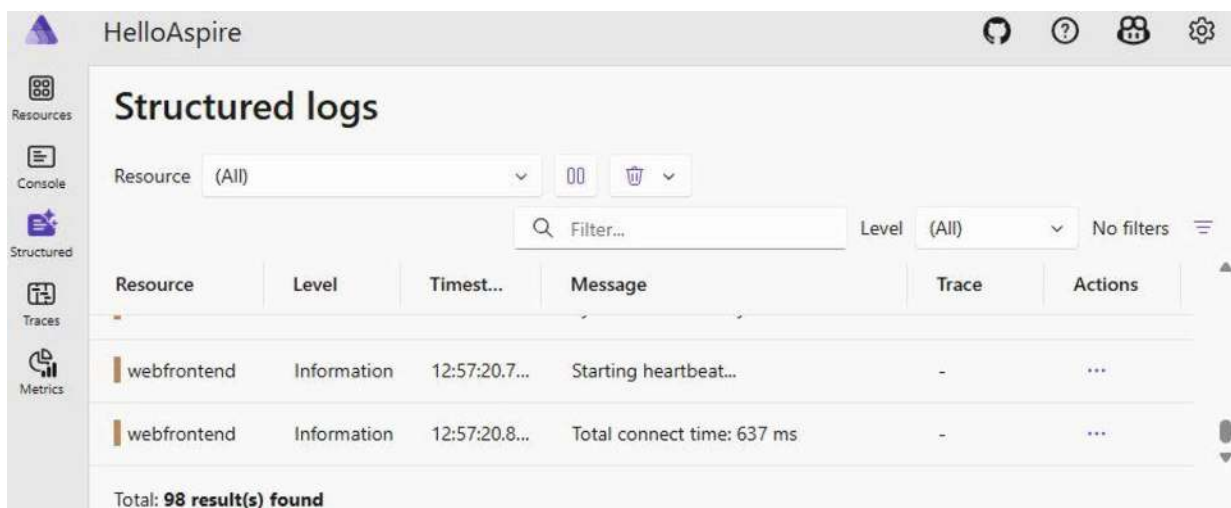
### Console Page

This page shows the console output for the projects in your solution. It's like having access to all your terminal outputs in a single view. The resource can be selected from the Resource dropdown as seen in [Figure 2-28](#).



**Figure 2-28** The Console Logs Resource Selector

This allows you to laser focus on a specific application’s logs, helping you see what’s going on with your running applications.



**Figure 2-29** Structured Logs

## Structured Logs Page (Structured)

Aspire collects structured logs from your applications via the built-in logging and OpenTelemetry integration, which is then presented in a table format shown in [Figure 2-29](#).

This page allows you to filter by log level and resource. You can also search for text in the log messages. You can also view more details under the Actions Menu.

If a resource has errors, clicking on the error icon will bring you to this view and filter to that resource and error level.

## Traces

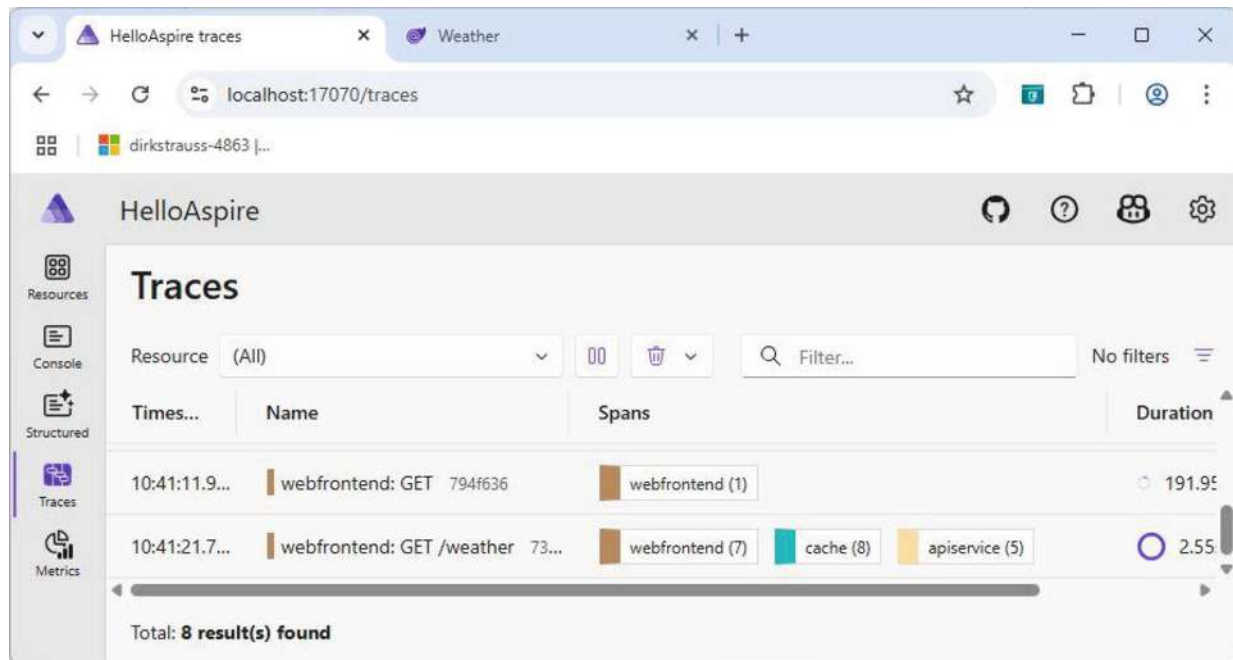
Distributed tracing visualization is probably one of the most powerful features. Aspire automatically instruments common operations such as HTTP requests, database calls, etc., with OpenTelemetry tracing.



The Traces page allows developers to see end-to-end traces of requests across services. For example, if the web application makes an HTTP call to the API, and the API hits the Redis cache. You could see a trace that spans `webfrontend` - `apiservice` - `cache` as seen in Figure 2-30.

Clicking on a trace displays a timeline view of the operations and how long each one took. Clicking on the `/weatherforecast` request, for example, will display the time it took for the Web UI to handle the request, which in turn included a call to the API that took X milliseconds, that included a Redis cache check of Y milliseconds.

This allows developers to pinpoint performance issues or errors across distributed components.



**Figure 2-30** The Traces Page

The dashboard's Trace page essentially replaces the need for external tools such as Jaeger or Zipkin in the dev environment, giving you immediate insights into the flow of requests.

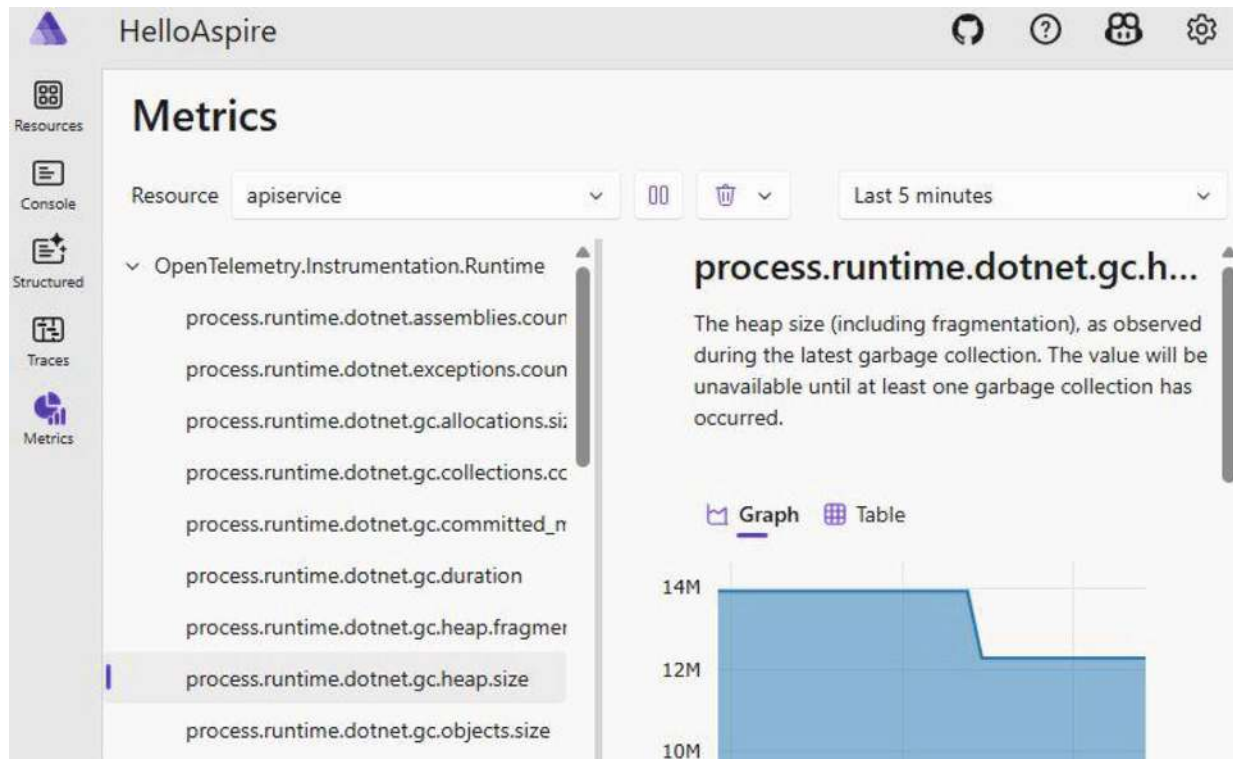
## Metrics

The Metrics page displays live metrics captured from your application (Figure 2-31). Aspire integrations automatically collect many metrics such as CPU, memory, HTTP requests, database connections, etc., via OpenTelemetry meters.

The Metrics page lists all available metric instruments (counters, gauges) and their values or dimensions.

You might see metrics like requests per second or active server requests or TLS handshakes, or even custom app metrics if you have defined any.

These metrics are not as fully featured as, say, Azure Monitor or Prometheus & Grafana, but it does give you a quick glance at important stats while you are developing your application.



**Figure 2-31** The Metrics Page

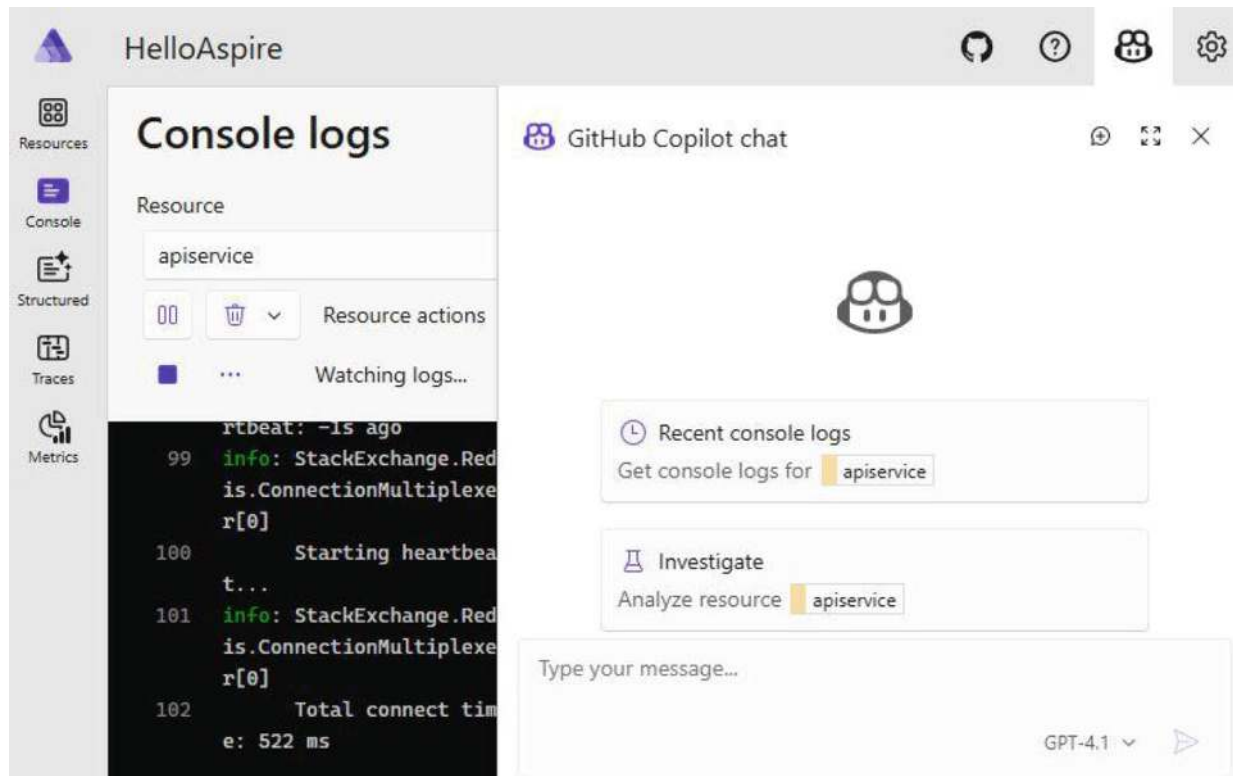
Again, you need to keep in mind that all this comes to you for free, out of the box, without you having to do anything extra. Metrics such as these can often help you catch issues such as memory leaks or too many retries, early, simply by watching the output on this page.

## Copilot

A notable addition to the Aspire dashboard is the integration of GitHub Copilot chat on the side.

Refer back to [Chapter 1](#) on Getting Started for free with GitHub Copilot, and how to use its features in your daily development.

As seen in [Figure 2-32](#), GitHub Copilot can assist you by answering questions about your logs or telemetry.



**Figure 2-32** GitHub Copilot Integration

You can see from Figure [2-32](#) that I am using Copilot on the Console logs page. The prompt suggestions will also change as the selected page changes. Here, Copilot is suggesting to analyze the `apiservice` resource. Changing to a different page, Structured for example, will provide different prompt suggestions. Copilot, therefore, has context of which page you are viewing on the Aspire dashboard, making it a solid debugging tool for finding issues.

## Standalone Dashboard Mode

We have seen how the .NET Aspire dashboard is tied to a running AppHost, but you can also run the Aspire dashboard in standalone mode. This means that you can monitor any application that emits OpenTelemetry data without the rest of .NET Aspire.

Microsoft provides a Docker image `mcr.microsoft.com/dotnet/aspire-dashboard:9.0` that you can run independently. This is essentially exactly the same UI, but you need to feed it telemetry from any source.

To launch the dashboard container on its own, run the command as seen in Listing [2-4](#).

```
docker run --rm -it -d \
  -p 18888:18888 \
  -p 4317:18889 \
  --name aspire-dashboard \
  mcr.microsoft.com/dotnet/aspire-dashboard:latest
```

**Listing 2-4** The Docker Command

This Docker command does the following:

- Starts a container from the `mcr.microsoft.com/dotnet/aspire-dashboard:9.0` image.
- Exposes two ports by
  - Mapping the dashboard's OTLP port 18889 to the host's port 4317 that receives OpenTelemetry data from apps.
  - Mapping the dashboard's port 18888 to the host's port 18888. Port 18888 has the dashboard UI, and if you navigate to `http://localhost:18888` in the browser, you will see the dashboard.

You can now point any OpenTelemetry-enabled app to send data to `localhost:4317` and be able to view it on the dashboard. This is a fantastic way to use Aspire's dashboard for scenarios that fall outside a full .NET Aspire project.

---

## Adding .NET Aspire to an Existing Application

If you have existing applications that could benefit from .NET Aspire, adding this to your existing apps is not only possible, but quite easy. Before you continue, ensure that you have the following installed locally:

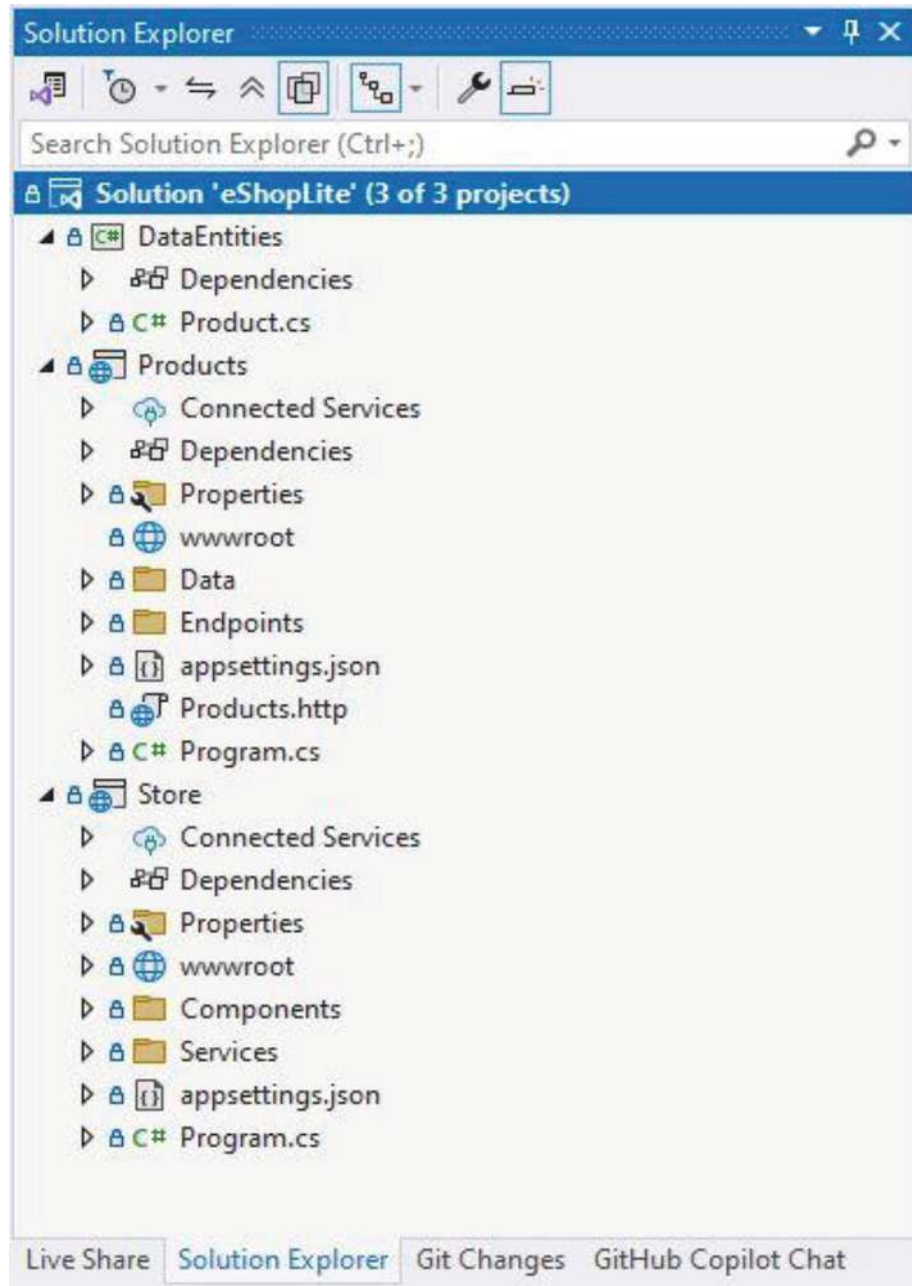
- .NET 8.0 or .NET 9.0
- An OCI-compliant container runtime such as Docker or Podman
- Visual Studio 2022 version 17.9 or higher

If you want an example project to follow along with, you can clone the eShopLite project by Microsoft.

To clone the project, use the following `git clone` command:

```
git clone https://github.com/MicrosoftDocs/mslearn-dotnet-cloudnative-devops.git eShopLite
```

This will clone a .NET 9 example application to your machine and place it in a new local folder called eShopLite.

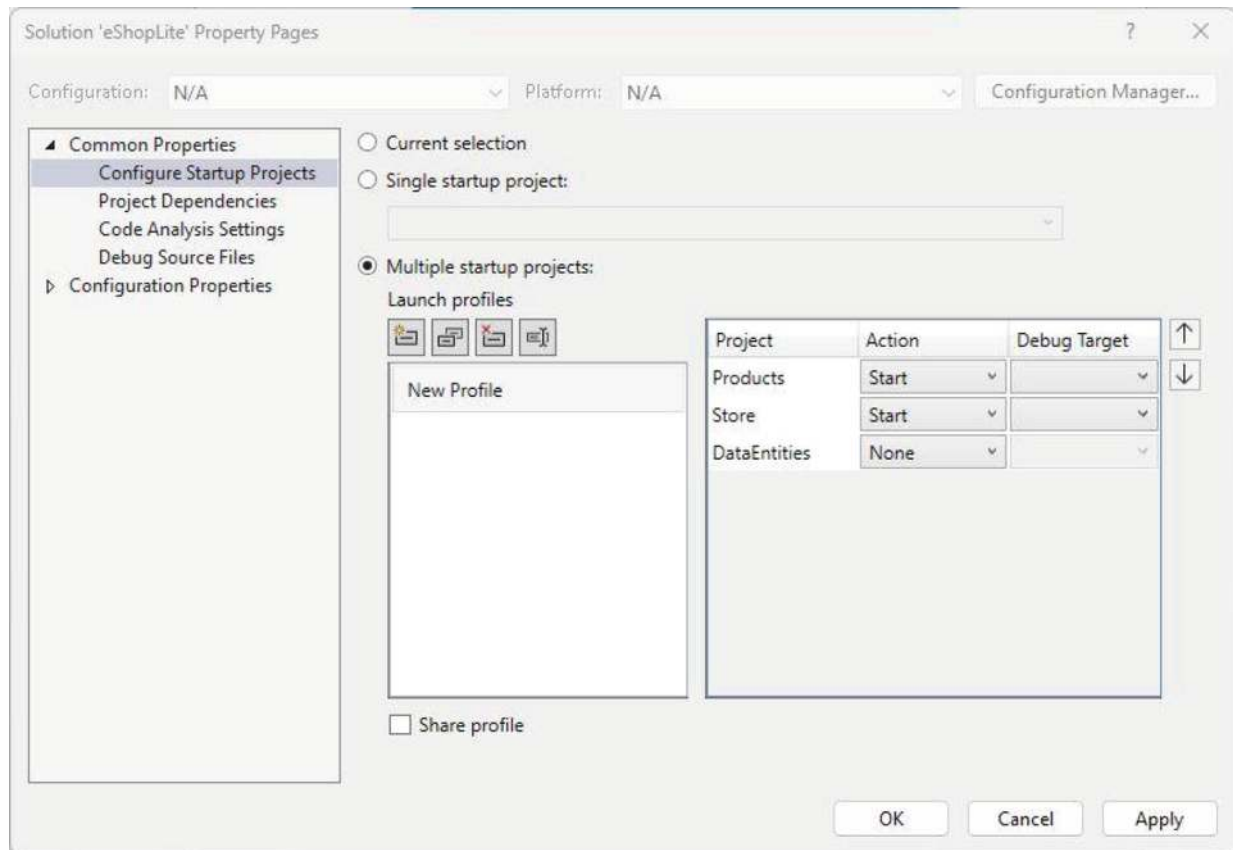


**Figure 2-33** The eShopLite Solution

You can see the solution in Figure [2-33](#). The solution contains three projects:

- DataEntities – This is a class library project that defines the `Product` class used in the Blazor application and the API.
- Products – This is a simple API project that returns a list of products in the database.
- Store – This is the Blazor application that pulls the product information from the API and displays it to the user.

Next we need to configure the startup projects. To do this, right-click the eShopLite solution in the Solution Explorer and select the context menu option to Configure Startup Projects.

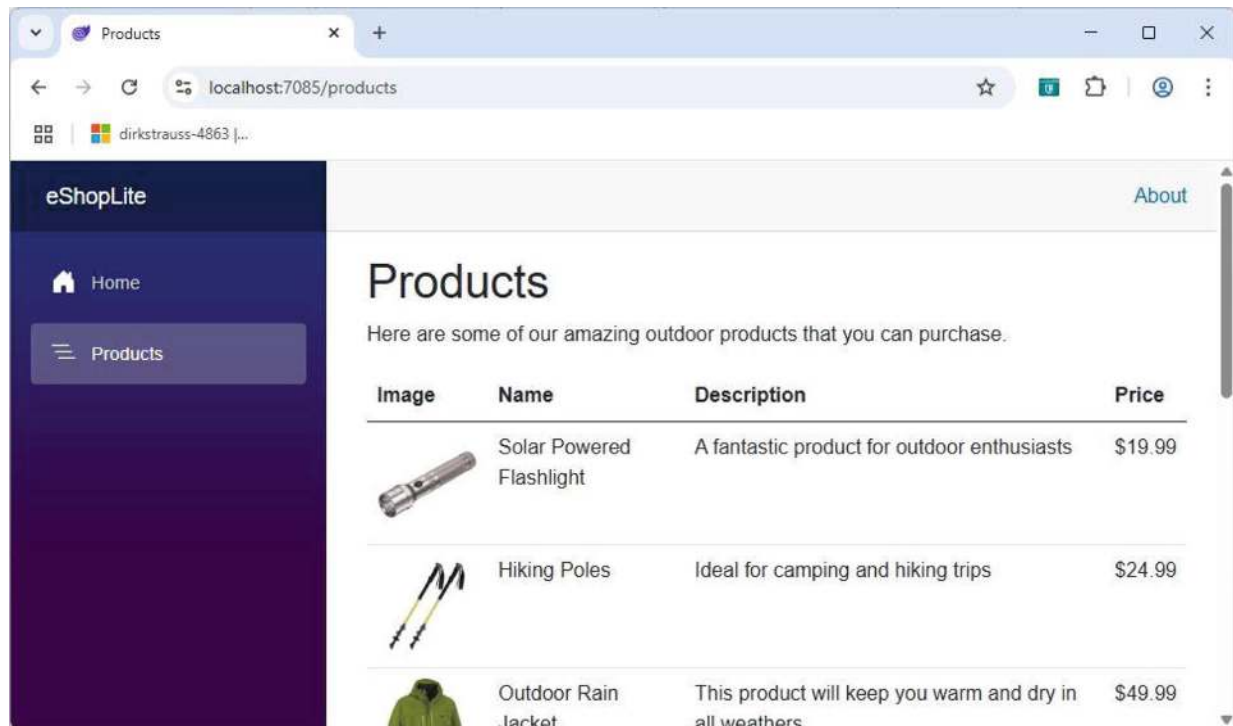


**Figure 2-34** Configure Multiple Startup Projects

As seen in Figure [2-34](#), select Multiple Startup Projects.

In the Action column, set Start for both the Products and Store projects, then click on OK to save.

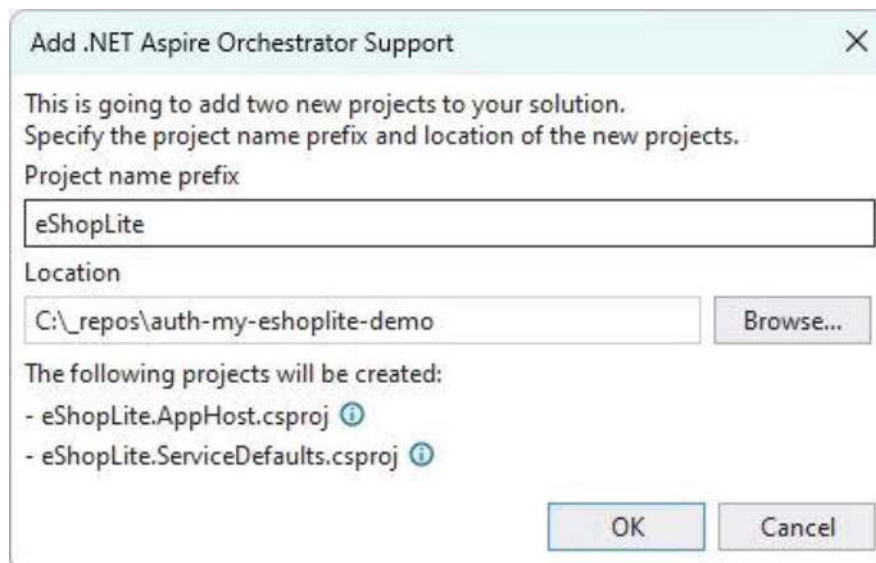
Now you can press F5 or select Start to start debugging your application. You will see two pages open in the browser. One browser page displays the Product output from the API in JSON. The other browser page displays the Blazor application's home page. To see the products, click on the Products menu shown in Figure [2-35](#).



**Figure 2-35** The Products Page

You now have a running application to test with. So let's enrol the Store project in .NET Aspire orchestration. To do this, right-click the Store project and select Add ► .NET Aspire Orchestrator Support from the menu options.

The Add .NET Aspire Orchestrator Support dialog will open as seen in Figure 2-36.

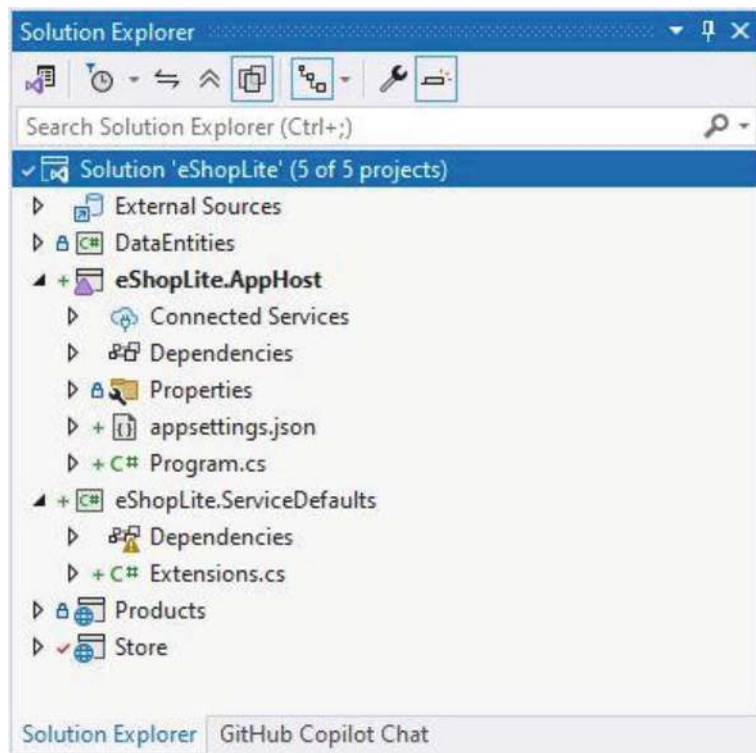


**Figure 2-36** Add .NET Aspire Orchestrator Support

Here you can select a prefix for the project name and the location. You can leave these set to default.

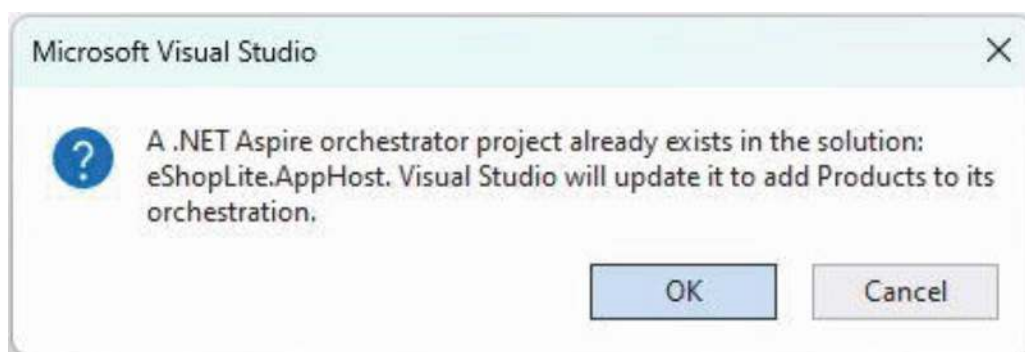


It also informs you that two projects will be created. These are the AppHost and the ServiceDefaults projects (Figure 2-37).



**Figure 2-37** The Added Aspire Projects

Clicking OK then creates these projects in your solution. Next we also want to add the Products project to .NET Aspire. Follow the same steps as previously followed, the only difference will be that this time you will right-click on the Products project and select Add ►.NET Aspire Orchestrator Support.



**Figure 2-38** Adding .NET Aspire to Products

As seen in Figure 2-38, we receive a message that we have already added a .NET Aspire Orchestrator project to the solution, and that it will only be updated to include Products to the orchestration. Looking at the eShopLite.AppHost project's Program.cs file, you will see the registrations of Store and Products (Listing 2-5).

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

```
builder.AddProject<Projects.Store>("store");

builder.AddProject<Projects.Products>("products");

builder.Build().Run();
```

**Listing 2-5** The AppHost Program.cs File

We now have a little bit of code work to complete the addition of Aspire Orchestration to our solution. Both our projects are now part of .NET Aspire Orchestration, but we need to enable service discovery to allow the Store project to rely on the Products backend address.

To do this, modify the AppHost's Program.cs file as seen in Listing [2-6](#).

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

var prodstore = builder.AddProject<Projects.Products>("products");

builder.AddProject<Projects.Store>("store")
    .WithExternalHttpEndpoints()
    .WithReference(prodstore)
    .WaitFor(prodstore);

builder.Build().Run();
```

**Listing 2-6** The Modified AppHost Program.cs File

Here we are telling .NET Aspire that the Store project relies on the Products project. You will also notice that the Store project is configured to use external HTTP endpoints.

This is necessary because in a distributed or cloud-native application such as those built on .NET Aspire and Blazor, resources such as services or APIs often define endpoints which by default, may only be accessible internally (e.g., within a container).

The `WithExternalHttpEndpoints()` extension method marks all HTTP and HTTPS endpoints on a resource as external. Setting them as external signals to the orchestrator that these endpoints should be exposed outside the internal network, making them accessible from your host machine, other services, or the public Internet.

You can see the `WithExternalHttpEndpoints()` extension method code in Listing [2-7](#).

Lastly, we call the `WaitFor` API that ensures that the store app doesn't start until the products API is fully started and ready to serve requests.

```
public static IResourceBuilder<T> WithExternalHttpEndpoints<T>(this
IResourceBuilder<T> builder) where T : IResourceWithEndpoints
{
    if
(!builder.Resource.TryGetAnnotationsOfType<EndpointAnnotation>(out
var endpoints))
    {
        return builder;
    }

    foreach (var endpoint in endpoints)
    {
        if (endpoint.UriScheme == "http" || endpoint.UriScheme ==
"https")
```

```

        {
            endpoint.IsExternal = true;
        }
    }

    return builder;
}

```

**Listing 2-7** WithExternalHttpEndpoints Extension Method

We also need to change the `appsettings.json` file in the Store project (Listing 2-8).

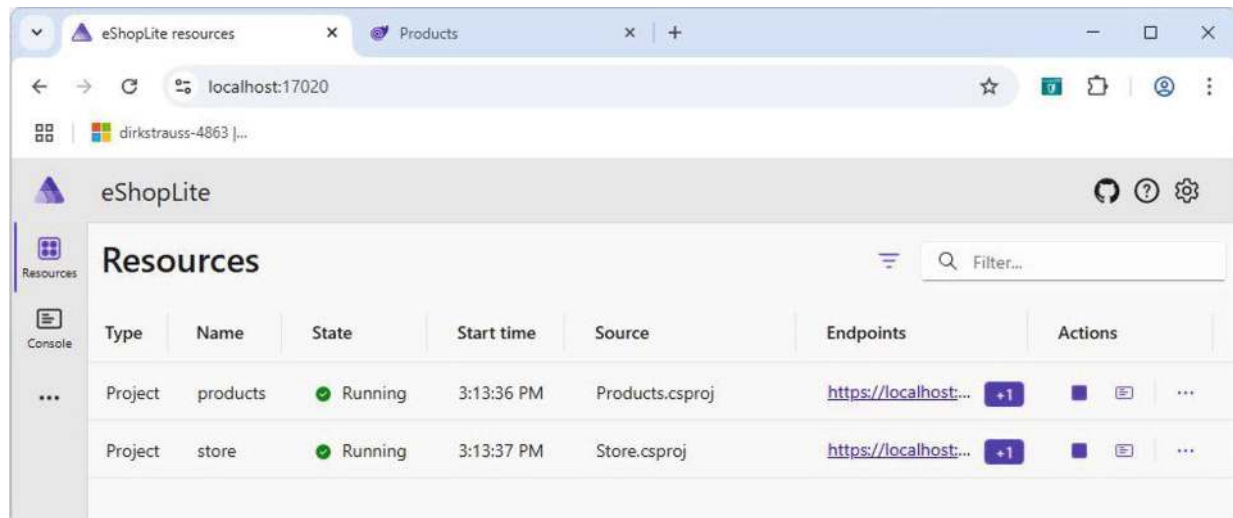
```

{
  "DetailedErrors": true,
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ProductEndpoint": "http://products",
  "ProductEndpointHttps": "https://products"
}

```

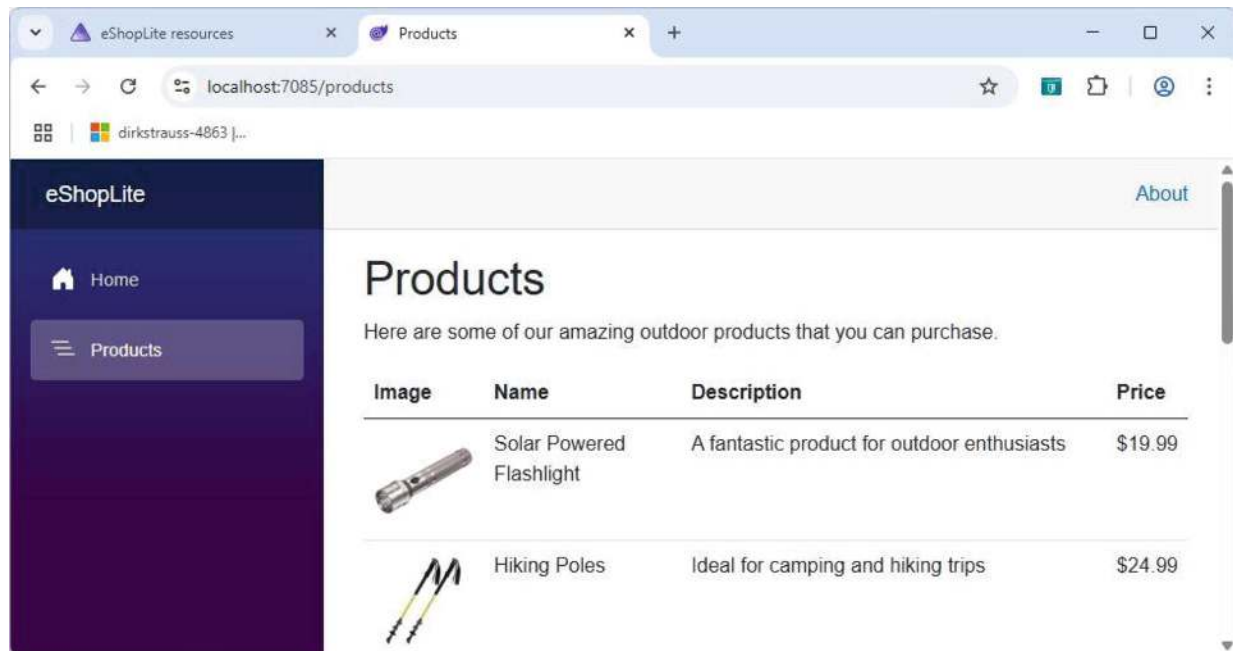
**Listing 2-8** The Store `appsettings.json` File

Here we are telling Aspire that the endpoints now use the "products" name that the orchestrator added to the AppHost.



**Figure 2-39** The Aspire Dashboard

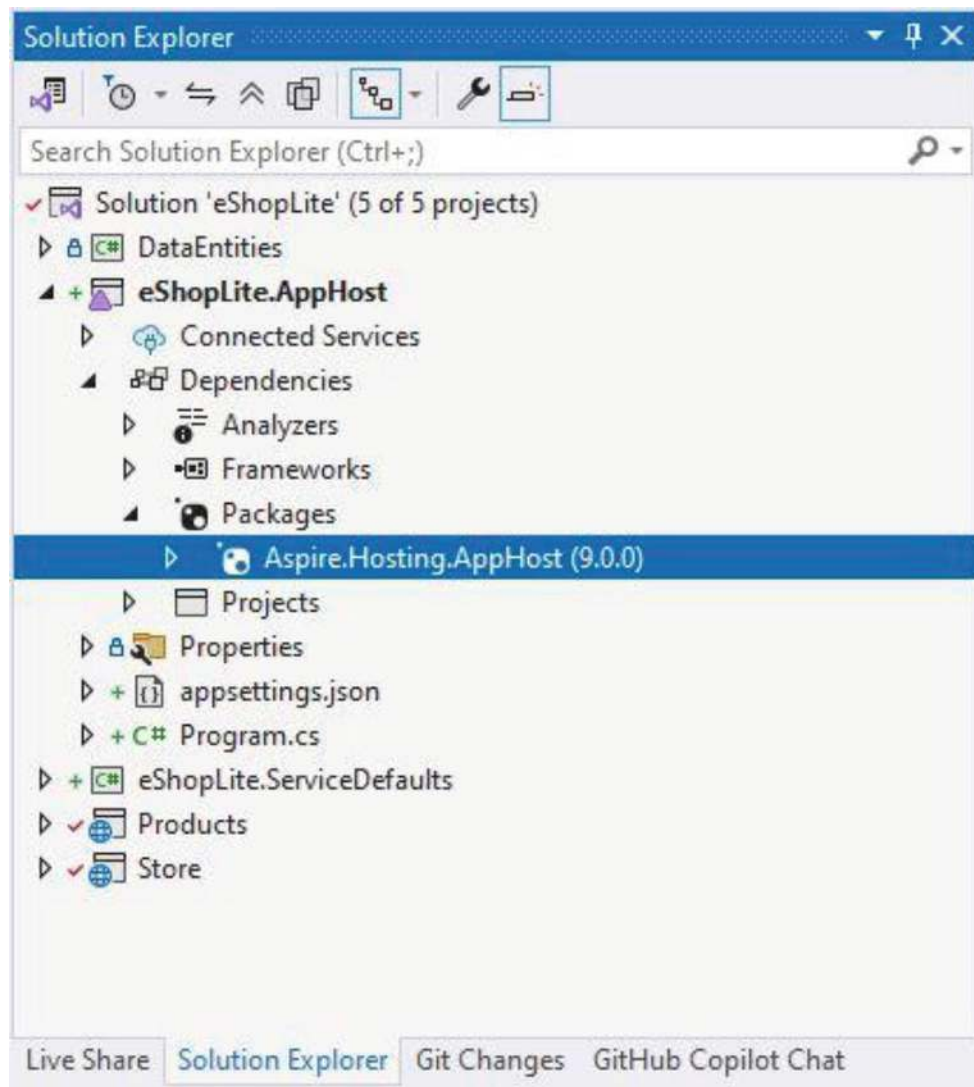
Running the project will launch the dashboard as seen in Figure 2-39.



**Figure 2-40** The Running Store App

Clicking on the `store` resource's URL in the dashboard will launch the Store application (Figure 2-40).

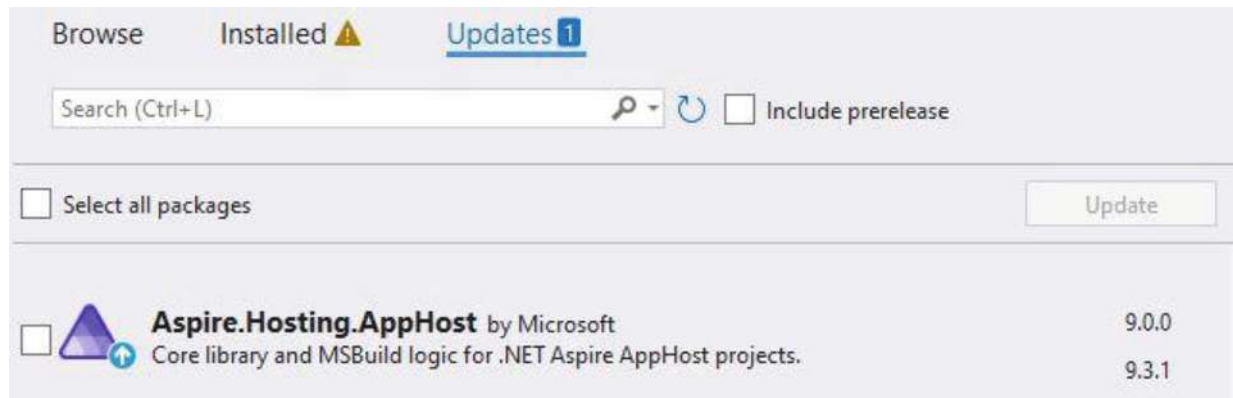
But something is missing. In the dashboard (Figure 2-39), there is no GitHub Copilot integration.



**Figure 2-41** The Aspire NuGet Package Version

This is because the .NET Aspire NuGet package that was added is most likely (as of this writing) version 9.0.0, as seen in [Figure 2-41](#).

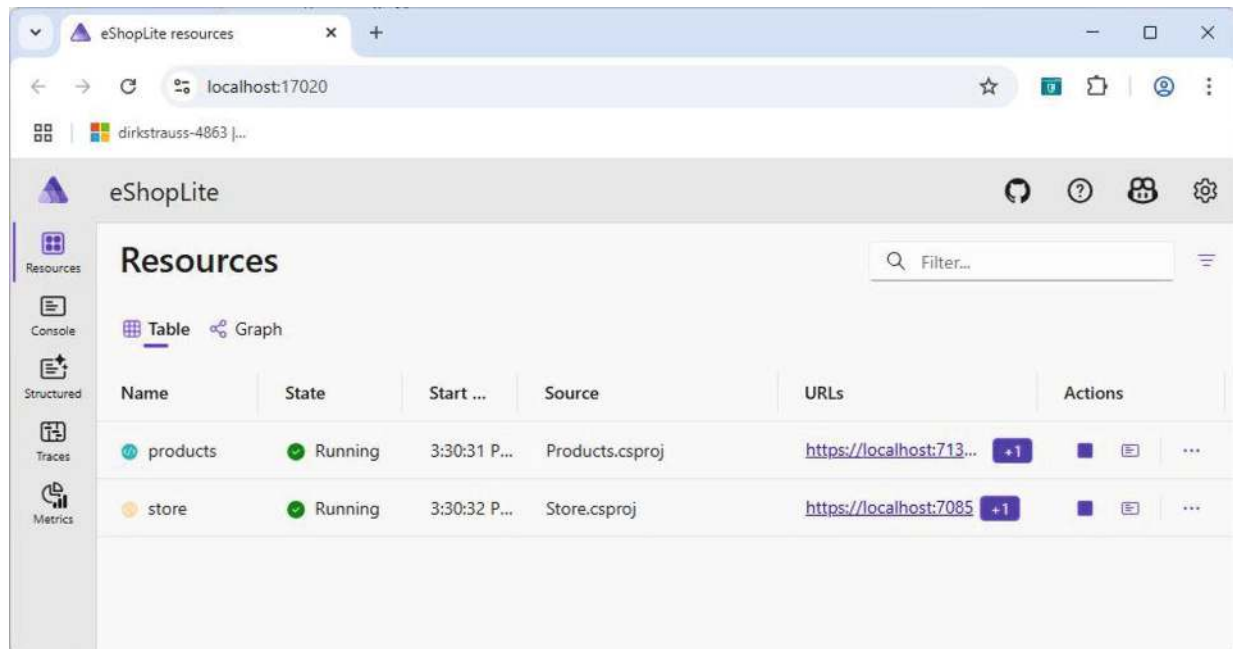
To fix this, right-click on the AppHost project and select Manage NuGet packages from the context window.



**Figure 2-42** Update AppHost NuGet Package

Under the Updates tab, you will see that the NuGet package for Aspire.Hosting.AppHost might be outdated. Update this to the latest version available to you (for me, it was version 9.3.1 as seen in Figure 2-42).

Rebuild your solution and run the Aspire project again.



**Figure 2-43** The Aspire Dashboard with Copilot

This time around, you will see GitHub Copilot integrated into your Aspire dashboard.

Although the eShopLite application is a small solution, it clearly demonstrates how to add .NET Aspire to an existing project. This allowed us to benefit from:

- Unified Startup.
- Service discovery, avoiding the need for hardcoded URLs.
- Telemetry and Health out of the box.
- Dev-time Containers - Aspire set the stage to easily add other services, which we'll see in the next section.

Aspire is capable of scaffolding away the complexities of managing multiple projects in a solution and, in doing so, reducing developer toil, allowing you to be more productive and effective (reducing your time to F5).

---

## Adding .NET Aspire Packages

One of the core benefits of .NET Aspire is the ability to use pre-built connectors (or packages) for popular services. These integrations come in two flavors:

- Hosting integrations – These provision or connect to external resources like databases, caches, message brokers, etc.
- Client integrations – These configure your application to use those resources via DI, health checks, etc.

Adding Aspire packages to your solution allows you to easily incorporate things like Redis, SQL Server, Azure Services, AWS services, and more into your application using a consistent pattern.

The .NET Aspire team provides a curated suite of NuGet packages under the `Aspire.*` naming. These are categorized by their service type and are designed to be cloud-agnostic or cloud-specific.

Cloud-agnostic packages are for technologies such as Redis, PostgreSQL, SQL Server, RabbitMQ, Kafka, MongoDB, etc.

Azure-specific packages can be found for Azure services such as Azure Service Bus, Azure Key Vault, Azure Storage, Azure Cosmos DB, Azure Functions, etc. The hosting integrations can be found in the `Aspire.Hosting.Azure.*` NuGet packages with their client integrations living in the `Aspire.*` NuGet packages.

The .NET Aspire Community Toolkit is a collection of integrations for .NET Aspire that has been created by the community, but that is not officially supported by the .NET Aspire team. For more information on what the Community Toolkit provides, browse to <https://learn.microsoft.com/en-us/dotnet/aspire/community-toolkit/overview>.

The key takeaway here is that adding an integration is as simple as installing a NuGet package and calling the corresponding extension method. You need to remember, though, that .NET Aspire is evolving, and you will want to keep your packages updated. Use the previous section as an example where we had to update the AppHost package (Figure 2-42). This update added GitHub Copilot integration into our Aspire dashboard.

A good rule of thumb is to use the latest stable Aspire packages and review the release notes regularly.

---

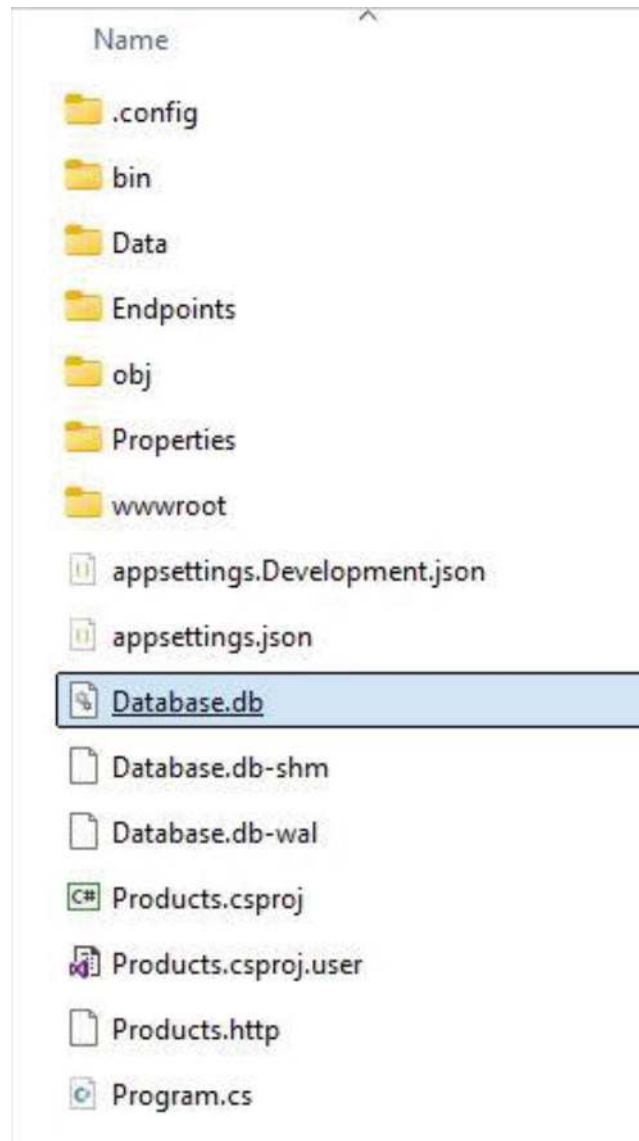
## Adding SQL Server

We had a look at adding .NET Aspire to an existing application called eShopLite earlier. This application is configured to use SQLite by default. This means that it uses a local SQLite database named `Database.db` located in the application's working directory (Figure 2-44).

You can see this by going to the Products project in your Solution Explorer, right-clicking on it and selecting Open Folder in File Explorer.

While this is great for debugging locally, we want to add SQL Server to our eShopLite application. Luckily, .NET Aspire makes it easy to add a SQL Server database.



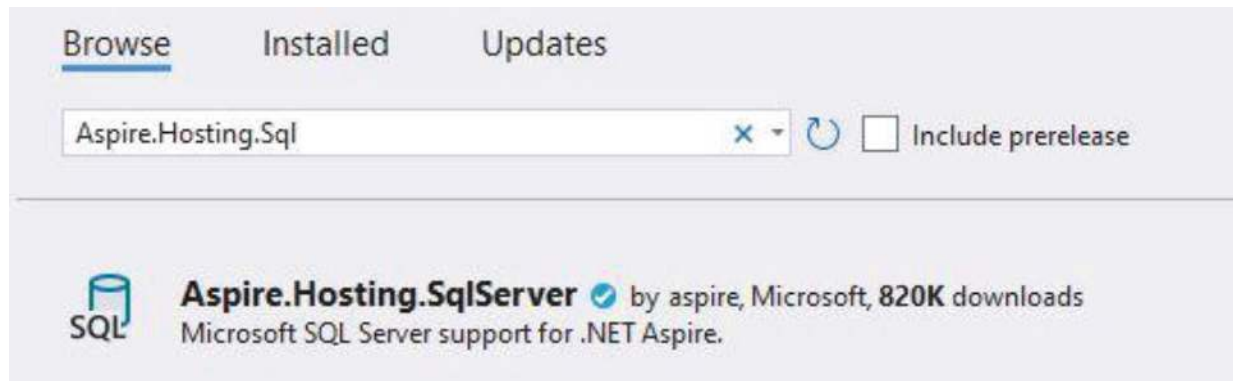


**Figure 2-44** The Local SQLite Database

Let's see how to do this by adding a local SQL Server container and connecting our app to it via Entity Framework Core.

### Installing the SQL Server Integration Packages

Let us identify which packages we will need. To enable running SQL Server in a container or handling an external connection, we will need to install `Aspire.Hosting.SqlServer` as seen in [Figure 2-45](#).

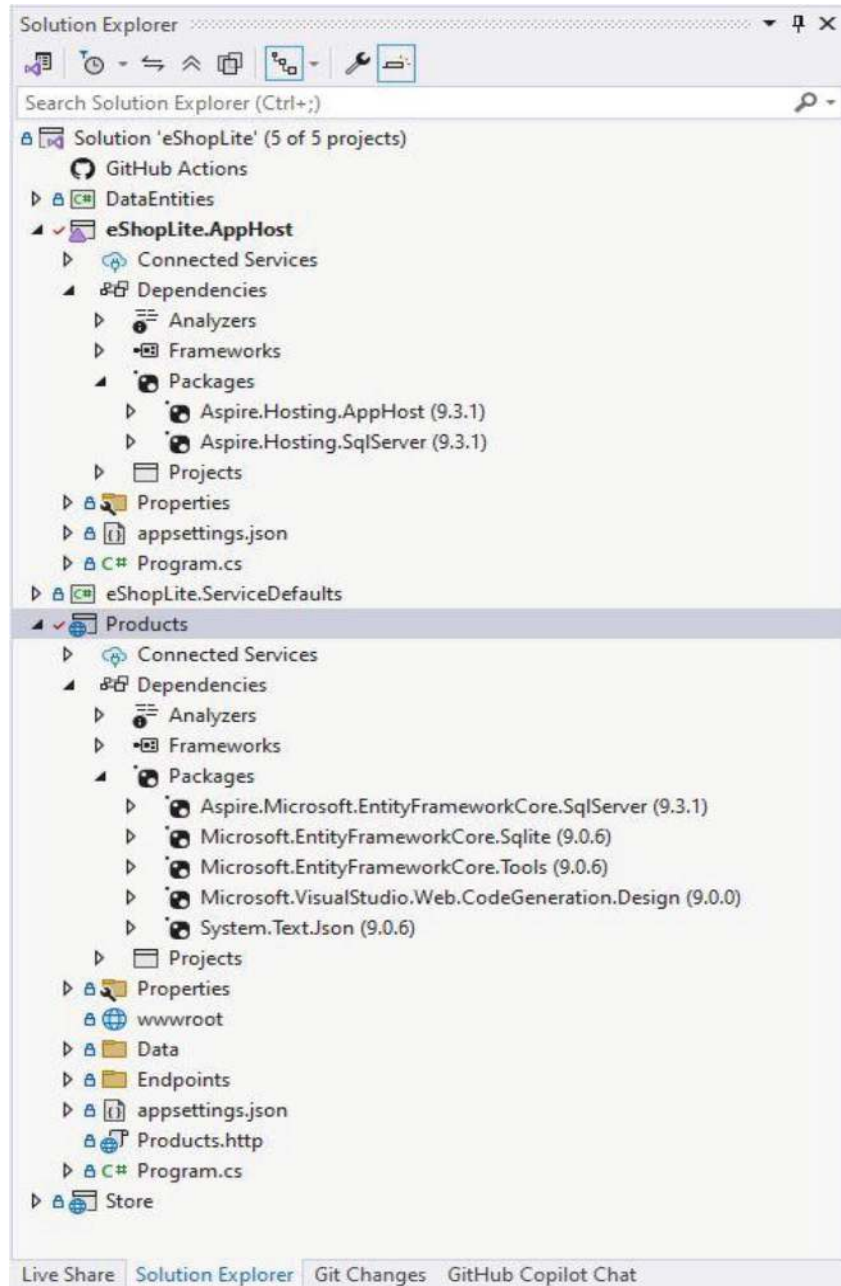


**Figure 2-45** The Aspire.Hosting.SqlServer Package

We also need a client integration for our data access approach. If we use EF Core, we will want the `Aspire.Microsoft.EntityFrameworkCore.SqlServer` package. If we want to use raw ADO.NET or Dapper, we will need `Aspire.Microsoft.Data.SqlClient`. Because EF Core usage is common, we'll be using that. To get started, add the following:

- `eShopLite.AppHost` project – add the `Aspire.Hosting.SqlServer` NuGet package, matching the version of Aspire you're using. You can do this by right-clicking the `AppHost` project and selecting `Add -> .NET Aspire Package` from the context menu. This will take you to the available Aspire NuGet packages by applying the filter `owner:Aspire tags:integration+hosting`.
- `Products` project – to the project that needs database access, add `Aspire.Microsoft.EntityFrameworkCore.SqlServer`.

These packages will bring the required dependencies into your `eShopLite` solution. When you have done this, the solution dependencies for the two projects should look as in [Figure 2-46](#).



**Figure 2-46** The Added NuGet Packages

We now have a few code changes to make next. We need to tell the AppHost that we want to use SQL server.

### Add a SQL Server Resource in AppHost

To configure the AppHost to include a SQL Server instance, we need to open the `Program.cs` file and make some code changes as shown in Listing 2-9.

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

var sql = builder.AddSqlServer("sql")
```

```

        .WithLifetime(ContainerLifetime.Persistent);

var db = sql.AddDatabase("eShopLite");

var prodstore = builder.AddProject<Projects.Products>("products")
    .WithReference(db)
    .WaitFor(db);

builder.AddProject<Projects.Store>("store")
    .WithExternalHttpEndpoints()
    .WithReference(prodstore)
    .WaitFor(prodstore);

builder.Build().Run();

```

**Listing 2-9** The AppHost Program.cs File

What we have done is added a SQL server resource to the AppHost by adding the line `builder.AddSqlServer("sql")`.

### ***AddSqlServer***

This will pull and run the official Docker image when you start the application. The string `"sql"` is the resource name. You use this to reference this resource in connection strings.

### ***WithLifetime***

We have also chained `.WithLifetime(ContainerLifetime.Persistent)` to specify that the SQL container should use a persisted lifetime. SQL Server can take some time to start up, so using this is recommended. Aspire will therefore try to keep the container running across multiple debug sessions of your app, instead of tearing it down each time. In reality, Aspire will stop it when the AppHost stops, but mark it to reuse the container with its data volume on the next run. This ensures that subsequent debug sessions are faster and your database persists. For dev scenarios, persistence is a good idea.

### ***AddDatabase***

To add a database resource to SQL Server, we used `var db = sql.AddDatabase("eShopLite")`. Our database name is specified as `"eShopLite"`. In fact, this is the resource name. Under the hood, Aspire will use the resource name as the database name if not provided. You could specify the resource name and database name separately by changing that line of code to `var db = sql.AddDatabase("db", "eShopLite")`. Here, the resource name is `"db"` and the database name is `"eShopLite"`.

### ***WithReference and WaitFor***

Next, we use our resource in the project that needs the database, which is (in our `eShopLite` solution) the `Products` project. Here we add `WithReference` and `WaitFor`.

### ***WithReference***

This injects the connection information for the database into our `Products` project's configuration. In fact, it uses the resource name as the connection string name, unless it is overridden.

### ***WaitFor***

As mentioned previously, `WaitFor` ensures that our Products API doesn't start until the database is ready.

### First Run

At this point, when you run your AppHost, you will notice that all your resources are either starting or waiting (Figure 2-47).

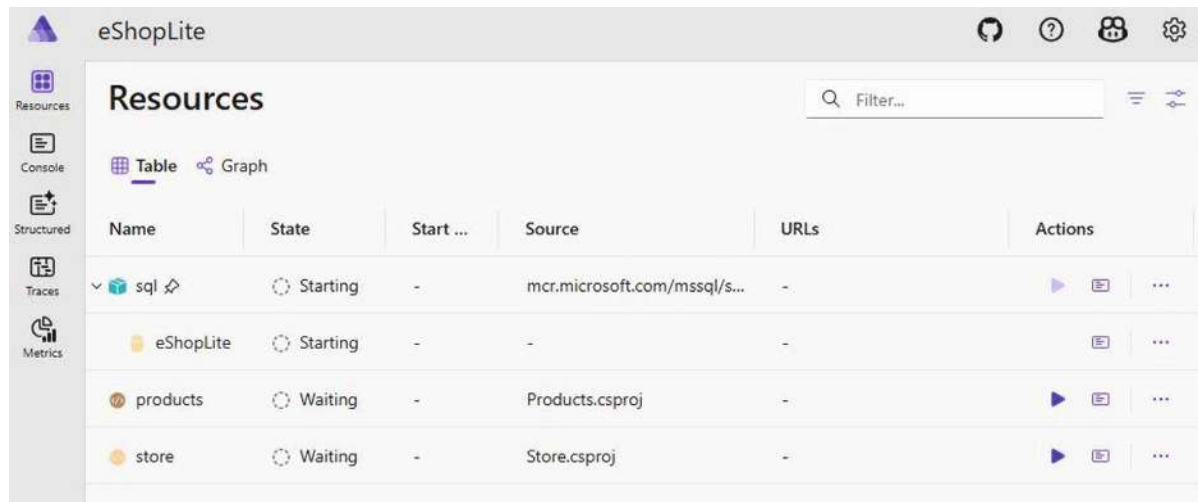
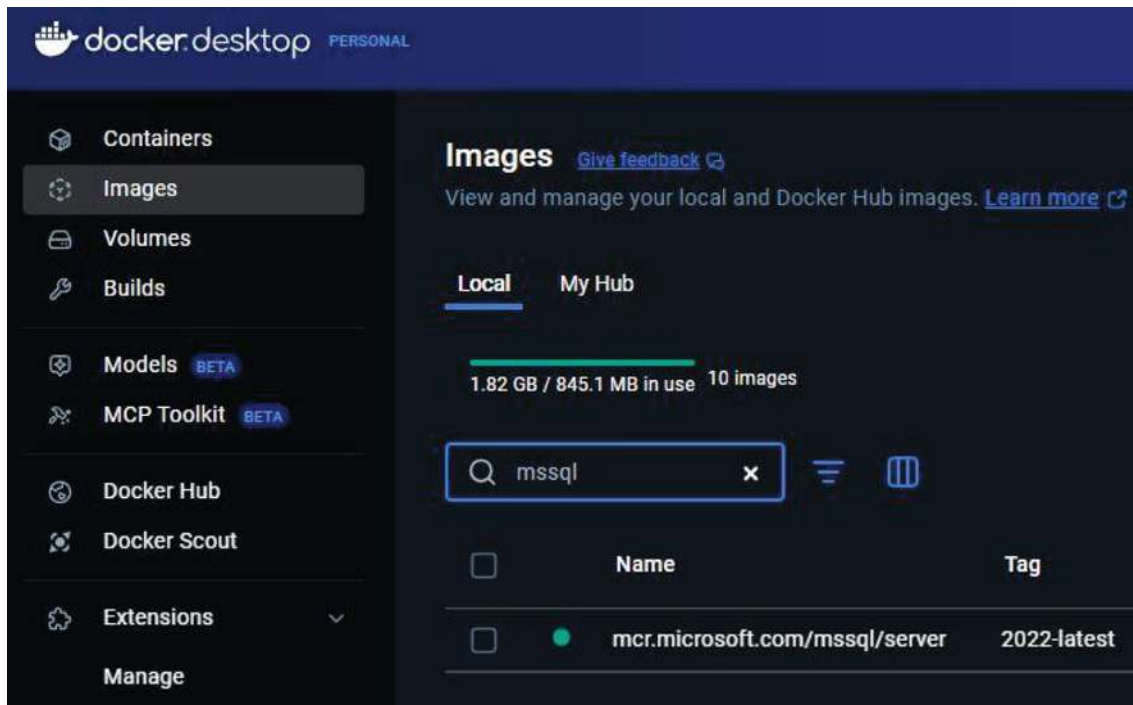


Figure 2-47 Aspire Dashboard

As specified in our AppHost, the Products project needs to `WaitFor` the database to be ready.

Docker then pulls the `mcr.microsoft.com/mssql/server` image if it hasn't been pulled previously (Figure 2-48). Aspire then starts the container and waits until the SQL server is responding. Once the database eShopLite has been created (on first run), Aspire sets the environment variable or config for the connection string. Once all that is in place, the API will start up.



**Figure 2-48** Docker Desktop

The next thing we need to do is add the client integration. Stop your Aspire project and let's set that up next.

### ***Connecting the Application via EF Core***

Our Products project is the API project for our application. You will remember (refer back to Figure 2-46) that we added `Aspire.Microsoft.EntityFrameworkCore.SqlServer` to this project. This means that we have an extension to register our `DbContext`.

We can modify our code to use EF Core instead of SQLite by adding `builder.AddSqlServerDbContext<ProductDataContext>(connectionName: "eShopLite")` to the `Program.cs` file of the Products project.

You can see the complete code in Listing 2-10.

```
var builder = WebApplication.CreateBuilder(args);
builder.AddServiceDefaults();
builder.AddSqlServerDbContext<ProductDataContext>(connectionName:
    "eShopLite");
// Add services to the container.
var app = builder.Build();
app.MapDefaultEndpoints();
// Configure the HTTP request pipeline.
app.MapProductEndpoints();
app.UseStaticFiles();
await app.CreateDbIfNotExistsAsync();
await app.RunAsync();
```

**Listing 2-10** The Products `Program.cs` File

The last code change we need to make is to modify the `ProductDataContext` class.

```

public static async Task CreateDbIfNotExistsAsync(this IHost host)
{
    using var scope = host.Services.CreateScope();

    var services = scope.ServiceProvider;
    var context = services.GetRequiredService<ProductDataContext>();
    // Remove EnsureCreatedAsync
    //await context.Database.EnsureCreatedAsync();
    // Add MigrateAsync
    await context.Database.MigrateAsync();

    await DbInitializer.InitializeAsync(context);
}

```

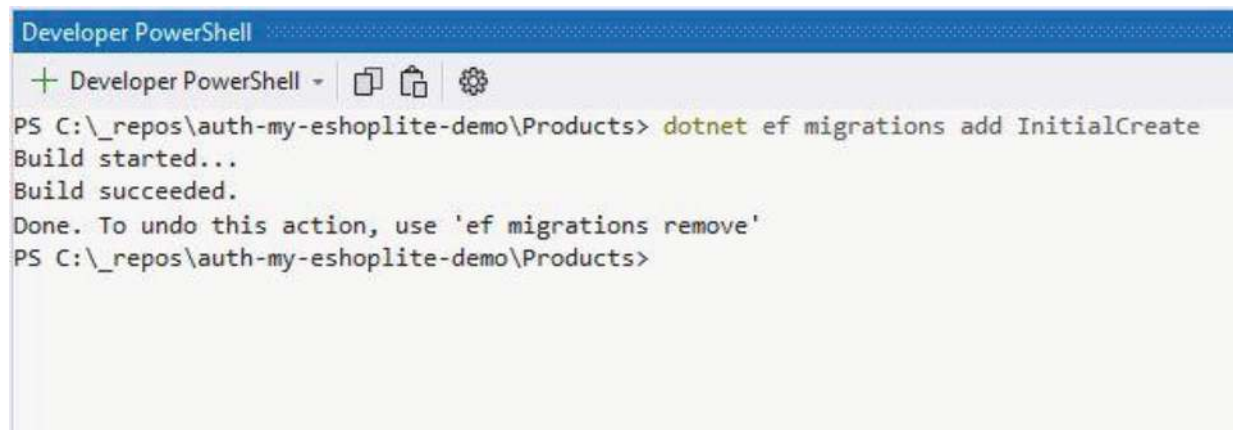
**Listing 2-11** Modifying the ProductDataContext Class

You will see that we have commented out the `EnsureCreatedAsync()` method and replaced it with `MigrateAsync()` method. This will asynchronously apply any pending migrations for the context to the database, and it will also create the database if it does not already exist.

Now that we have that figured out, we need to get the tables created on our database. Aspire has only created the database for us; it hasn't created any tables. To do this, we will be using migrations.

## Adding and Running Migrations

To get the process going, right-click on the `Products` project and select "Open in Terminal" from the context menu. Once the terminal is open, run the command `dotnet ef migrations add InitialCreate`. You can see this in Figure [2-49](#).



```

Developer PowerShell
+ Developer PowerShell
PS C:\_repos\auth-my-eshoplite-demo\Products> dotnet ef migrations add InitialCreate
Build started...
Build succeeded.
Done. To undo this action, use 'ef migrations remove'
PS C:\_repos\auth-my-eshoplite-demo\Products>

```

**Figure 2-49** Adding Migrations

This will add a Migrations folder to your `Products` project as seen in Figure [2-50](#).

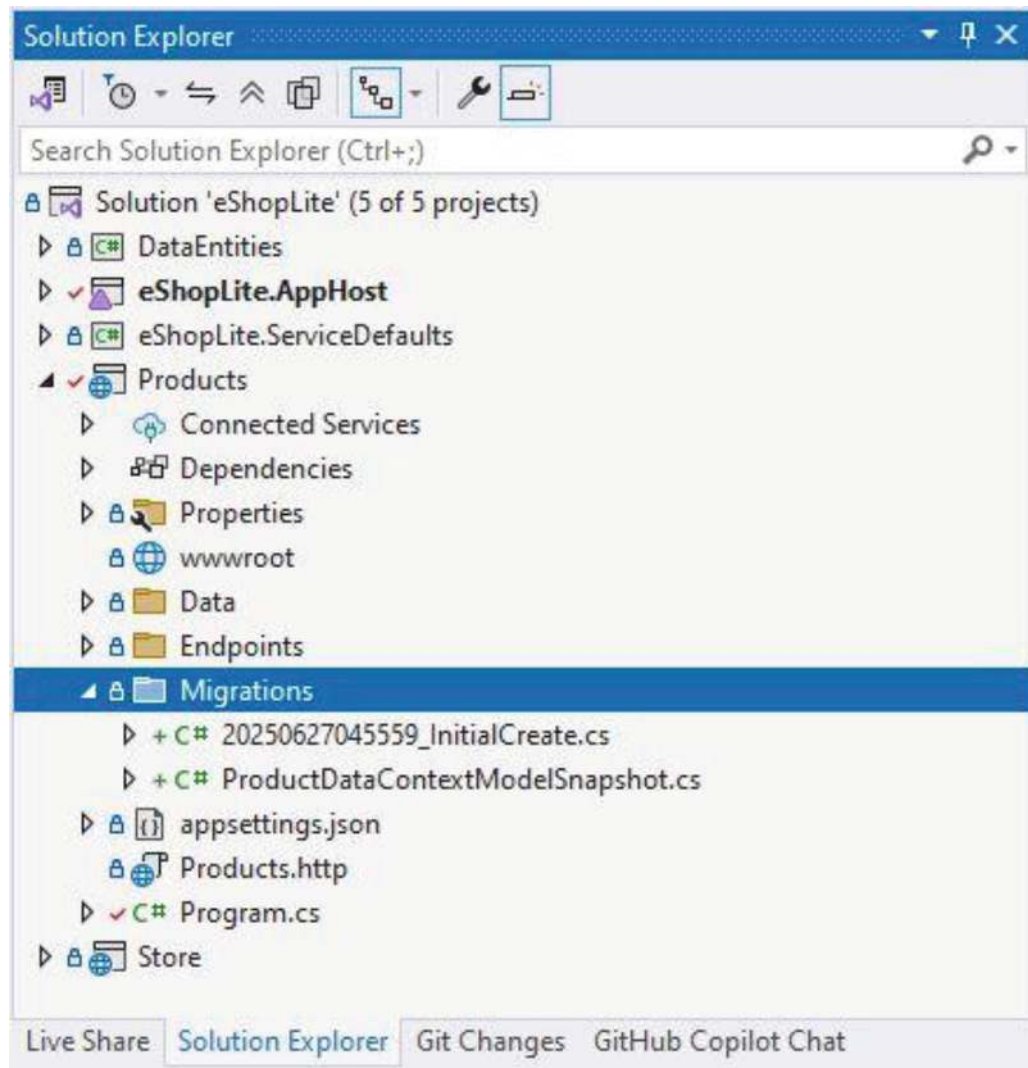
Although we manually ran `dotnet ef migrations add` to generate the initial migration, we do not actually need to apply the migration using `dotnet ef database update`.

When our Aspire application starts up, it will automatically apply all pending migrations at runtime thanks to the `context.Database.MigrateAsync()` call in our



`CreateDbIfNotExistsAsync` method. This allows us to maintain a clean deployment pipeline, where schema creation and evolution are handled within the application itself, no separate migration steps required.

Later on in development, if you added a new entity (e.g., `Category` or `Order`), you will only be required to generate a new migration to reflect that change. This is done by running the `dotnet ef migrations add` command, followed by the name that describes the purpose of the migration. For example, `dotnet ef migrations add AddCategoryEntity`. In this example, `AddCategoryEntity` is not a command or keyword; it's simply the name of the migration.



**Figure 2-50** The Migrations Folder

This command scans your updated `DbContext` and generates the appropriate `Up()` and `Down()` methods to apply or remove the schema change. Once this migration exists in your project, you do not need to manually apply it. Aspire will run the app as usual, and because we have hooked up `context.Database.MigrateAsync()`, any pending migrations, including ones for newly added entities, will automatically be applied to the database at runtime.

## Running Your Aspire App

We can now run our Aspire application to kick off the process of applying the migrations we added.

If you get a 'Globalization Invariant Mode is not supported' error, stop your application and edit the .csproj file of the Products project. Look for and remove the tag `<InvariantGlobalization>true</InvariantGlobalization>` from the `<PropertyGroup>`. Then run your Aspire solution again.

Once all your resources are marked as running in your Aspire dashboard, open up a Console and run the command `docker ps`. This will list your running Docker containers as seen in Figure 2-52. Look for the PORTS column in the output for the container spun up by Aspire for your SQL Server as seen in Figure 2-51.

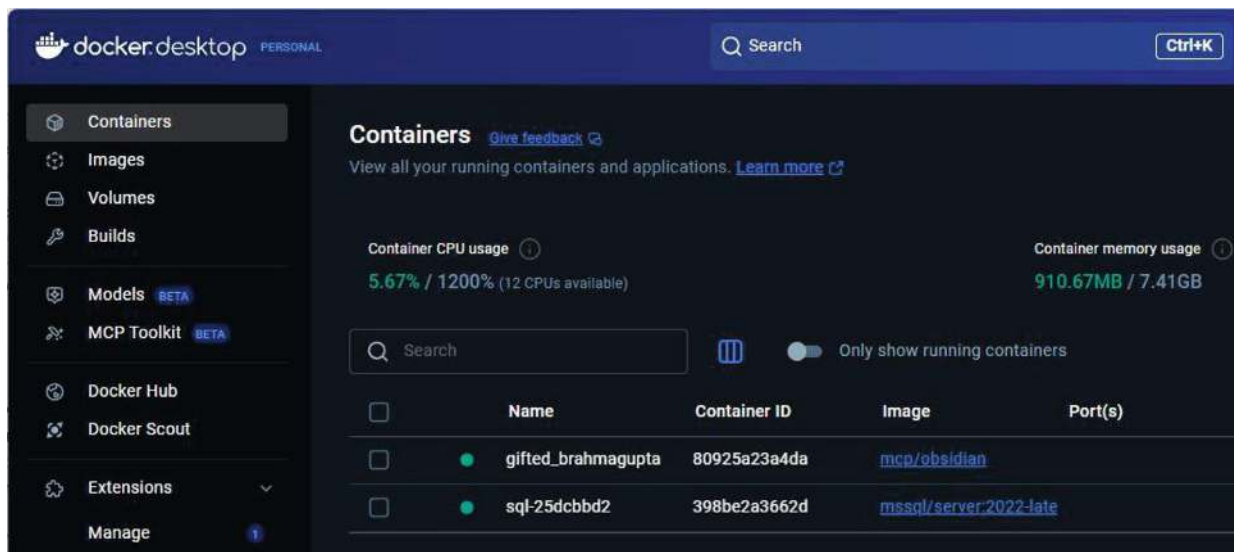


Figure 2-51 The Running Containers

From the Console, you will see that our container is listening on port 1433 internally (SQL Server’s default port). Docker is exposing it to my local machine on port 63377. This means that I can connect to my database via SQL Server Management Studio using that IP and Port.

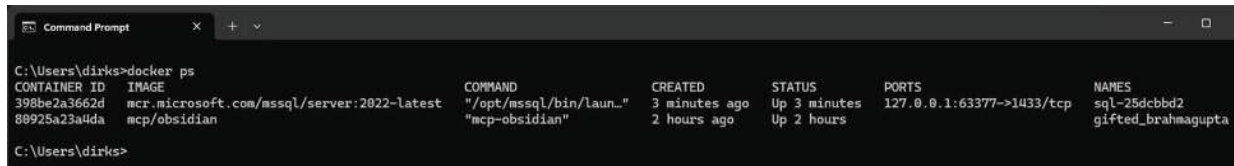
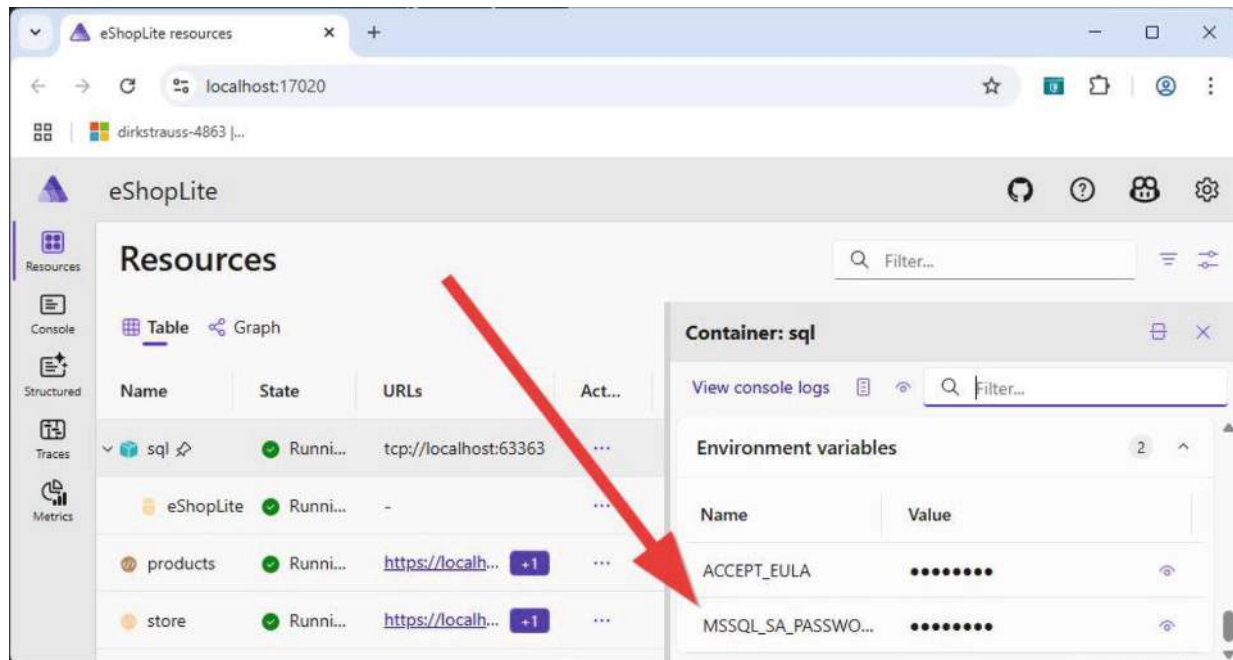


Figure 2-52 The Running Containers in the Console

Open SQL Server Management Studio and paste `127.0.0.1, 63377` into your Server Name. Take note of the comma between the IP address and the Port. To find the password of the `sa` SQL user, in the running Aspire dashboard, click on View Details under the Actions column on the SQL resource (Figure 2-53).



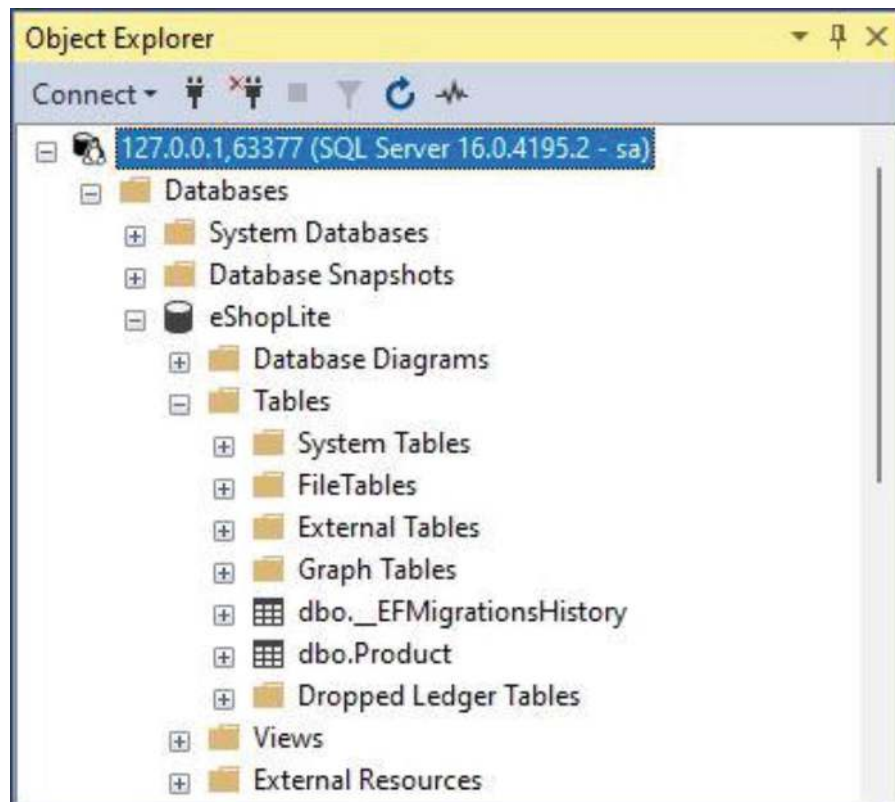
**Figure 2-53** Finding the SQL Server Password

With the password copied paste it into the Connect to Server dialog (Figure 2-54).



**Figure 2-54** Connect to Server Dialog

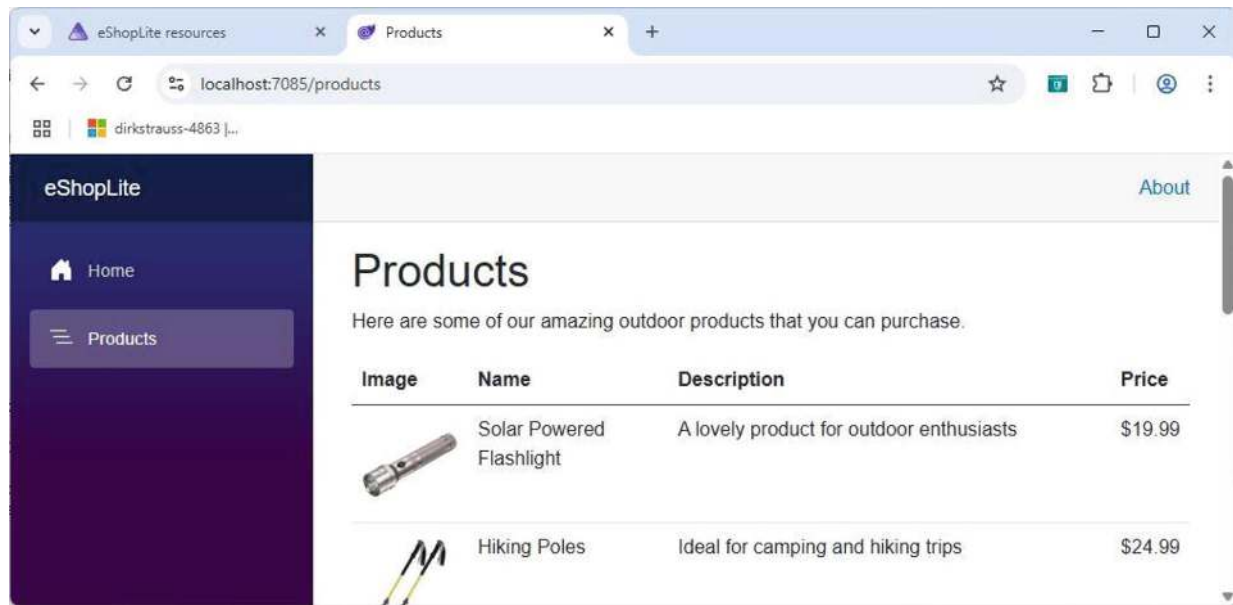
Once you click on the Connect button, you will connect to the database in the container that Aspire spun up for you, as seen in Figure 2-55. The database name is the same name displayed in our Aspire dashboard and is the same as we had defined in the AppHost's `Program.cs` file.



**Figure 2-55** The eShopLite Database

If you select the Product table, you will see the data that your Products API project seeded when it started. The seeding data can be seen in the `ProductDataContext` class in the Products API.

Running the web application from your Aspire dashboard and browsing to the Products menu, you will see the product data seeded in your Docker container's SQL database.



**Figure 2-56** The Running Product Store

Congratulations, you have successfully added a SQL database to your .NET Aspire application. By adding SQL Server through .NET Aspire, you have essentially containerized your database dependency without the usual hassle. Many developers will find this extremely useful for ensuring that their dev/test environment is as “production-like” as possible. Not having to maintain separate local connection strings or keep a local SQL Express running reduces complexity and improves developer productivity.

## A Word on NuGet Packages

As a software developer, being able to reuse code is essential to any modern development effort. Being able to share code is the cornerstone of a healthy development community. Many developers create handy code libraries that can add functionality to your particular application. We saw how NuGet packages allowed us to add functionality to our Aspire application.

This is where NuGet becomes an essential tool for developers to create, share, and consume helpful code. As a developer, you can package a DLL and other required content needed for the DLL to function correctly into a NuGet package.

NuGet is just a ZIP file with a .nupkg extension containing the DLLs you have created for distribution. Included inside this package is a manifest file that contains additional information, such as the version number of the NuGet package.

Packages uploaded to nuget.org are public and available to all developers that use NuGet in their projects. However, developers can create NuGet packages that are exclusive to a particular organization and not publicly available. Let’s look at hosting your own NuGet feeds.

## Hosting Your Own NuGet Feeds

Sometimes you need to share NuGet packages with a limited audience, typically internal developers, without publishing them publicly on nuget.org. Whether due to security policies or proprietary code, NuGet supports several options for private hosting:

- Local feed – Simply share a folder over the network and point NuGet to it. This is ideal for quick internal distribution.

- Azure Artifacts – A managed and scalable solution for hosting private NuGet feeds as part of Azure DevOps.
- Third-party solutions:
  - MyGet – <https://myget.org/>
  - ProGet – <https://inedo.com/proget>
  - TeamCity – [www.jetbrains.com/teamcity/](http://www.jetbrains.com/teamcity/)

For a complete list of NuGet hosting products and for more information on creating your NuGet feeds, have a look at the following link on Microsoft Learn:

<https://learn.microsoft.com/en-us/nuget/hosting-packages/overview>.

---

## Using npm in .NET Projects

Experienced .NET developers are used to managing libraries with NuGet, but modern web development often relies on npm (Node Package Manager) for client-side packages and build tools. In ASP.NET Core web apps, npm is commonly used to pull in JavaScript frameworks, CSS libraries, and tools such as Webpack or Gulp.

Visual Studio makes integrating npm into your workflow easy by providing native support for npm. One of the advantages of using npm is that it hosts virtually any front-end library or tool you might need, all in one place.

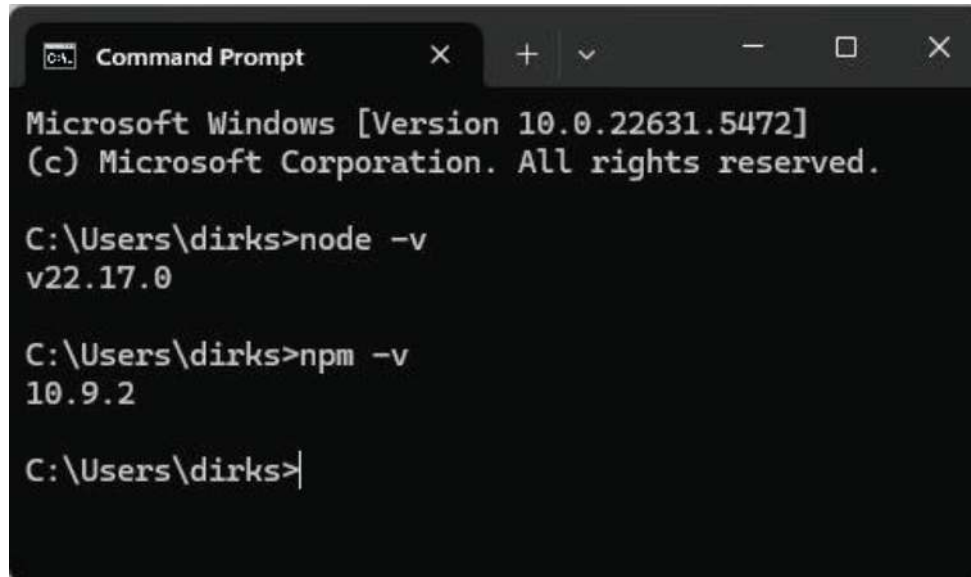
Unlike NuGet, which can manage .NET and some JS packages, npm is specifically used for Node.js and front-end ecosystems. Packages are managed by project, and are installed into a local `node_modules` folder. By contrast, NuGet uses a global cache and will allow multiple versions of a library in different projects. With npm, each project maintains its own dependencies.

This means that only one version of a specific package is used per project, which is stored under that project's `node_modules` folder. The result is that this isolation from other projects means that there could potentially be a duplication of packages across projects.

If you want to use npm, you need to have Node.js (which includes npm) installed on your system. You can use Visual Studio's Node.js development workload to install Node.js for you, but you can also install Node.js from the official website at <https://nodejs.org/>.

You can verify if Node.js is installed by running `node -v` and `npm -v` as seen in Figure 2-57. Project templates such as React or Angular include a `package.json` by default, and Visual Studio will automatically trigger an `npm install` on creation of such a project to retrieve all the client-side dependencies.





```
Microsoft Windows [Version 10.0.22631.5472]
(c) Microsoft Corporation. All rights reserved.

C:\Users\dirks>node -v
v22.17.0

C:\Users\dirks>npm -v
10.9.2

C:\Users\dirks>
```

**Figure 2-57** Check if Node.js is Installed

In other words, when using Visual Studio 2022 to create an Angular or React app using the built-in templates, the corresponding CLI tooling is invoked under the hood. This is why the `package.json` is automatically created and the `npm install` command is executed to pull the required client-side dependencies.

## Anatomy of the `package.json`

The heart of your project's client-side setup is the `package.json` file when working with npm. An example of this file can be seen in Listing 2-12. This JSON file lives at the root of your project and describes your project to npm. This includes information such as the project's name as well as metadata like what versions of packages your project has dependencies on.

```
{
  "name": "reactdemo.client",
  "private": true,
  "version": "0.0.0",
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "lint": "eslint .",
    "preview": "vite preview"
  },
  "dependencies": {
    "react": "^19.1.0",
    "react-dom": "^19.1.0"
  },
  "devDependencies": {
    "@cap-js/sqlite": "^2.0.1",
    "@eslint/js": "^9.29.0",
    "@types/react": "^19.1.8",
    "@types/react-dom": "^19.1.6",
```



```

    "@vitejs/plugin-react": "^4.5.2",
    "eslint": "^9.29.0",
    "eslint-plugin-react-hooks": "^5.2.0",
    "eslint-plugin-react-refresh": "^0.4.20",
    "globals": "^16.2.0",
    "vite": "^7.0.0"
  }
}

```

**Listing 2-12** The package.json File

Essentially, the `package.json` file tells npm what needs to be installed. The key fields in the file include:

- **name** – This is the name of your application, usually all lowercase.
- **version** – This is the current version of your project, which needs to be updated as your project evolves.
- **scripts** – An object listing command scripts you can run via npm.
- **dependencies** – These are the runtime dependencies for your app.
- **devDependencies** – As the name suggests, these are the development-time dependencies required during development.

In the example `package.json` file in Listing 2-12, you will notice version numbers next to the dependencies. For package versions, npm uses semantic versioning that allows you to be flexible with regard to version ranges. A version number is defined as `major.minor.patch` and the way you specify the version in this file has significance.

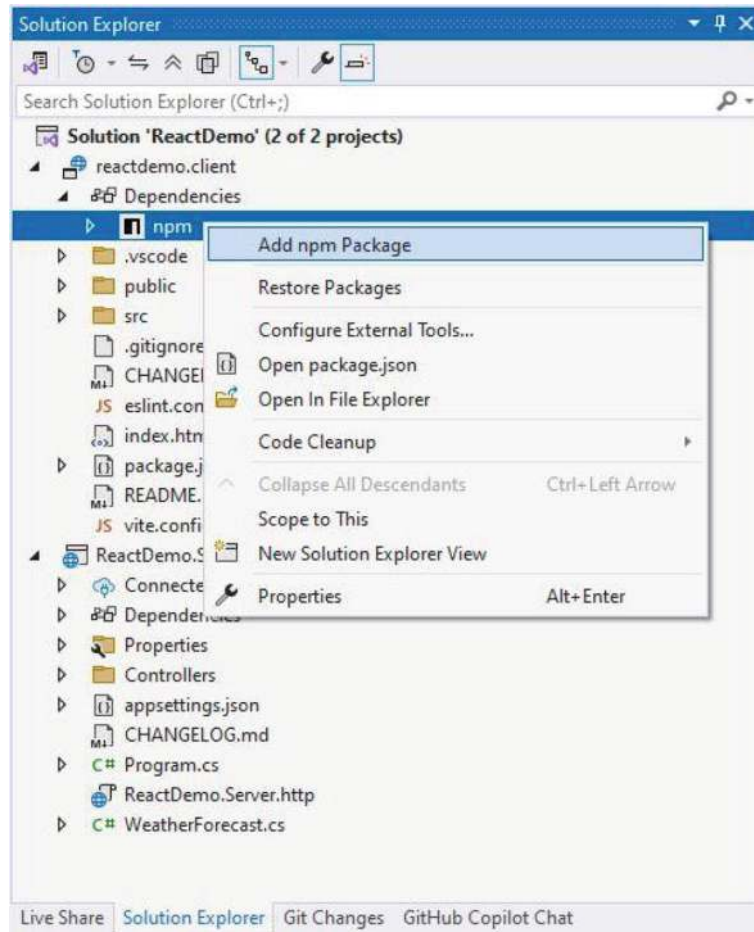
- **Caret (^)**: For example, `"react": "^19.1.0"` allows updates to any version 19.x.x newer than 19.1.0 but not 20.x. This means you will get the latest minor/patch updates but no major updates that might break compatibility.
- **Tilde (~)**: For example, `"react": "~19.1.0"` allows updates to any version 19.1.x (patch versions), but will lock the minor version. This means 19.1.0 can update to 19.1.1 but not 19.2.0.
- **No prefix (exact version)**: For example, `"react": "19.1.0"` locks the version to exactly what is defined. This means npm will not update it unless you manually change the version.

Using these notations allows you to control the aggressiveness of your updates. In summary, the `package.json` file is your go-to place for configuring npm packages. Visual Studio and npm work in unison to keep this file and your `node_modules` in sync.

## Managing npm Packages

In a similar vein to NuGet, npm is the tool for managing client-side JavaScript packages. It's widely used in ASP.NET Core projects, specifically those utilizing modern front-end frameworks like Angular, React, or Vue.

Visual Studio 2022 integrates well with npm through the `package.json`, allowing you to install packages, run build scripts, and manage front-end dependencies directly from the IDE. The package manager can be opened by right-clicking the npm node in the Solution Explorer as seen in Figure 2-58.



**Figure 2-58** Installing New npm Packages

You can now search for npm packages and then select one to install, as seen in [Figure 2-59](#). Clicking the Install Package button installs the package.

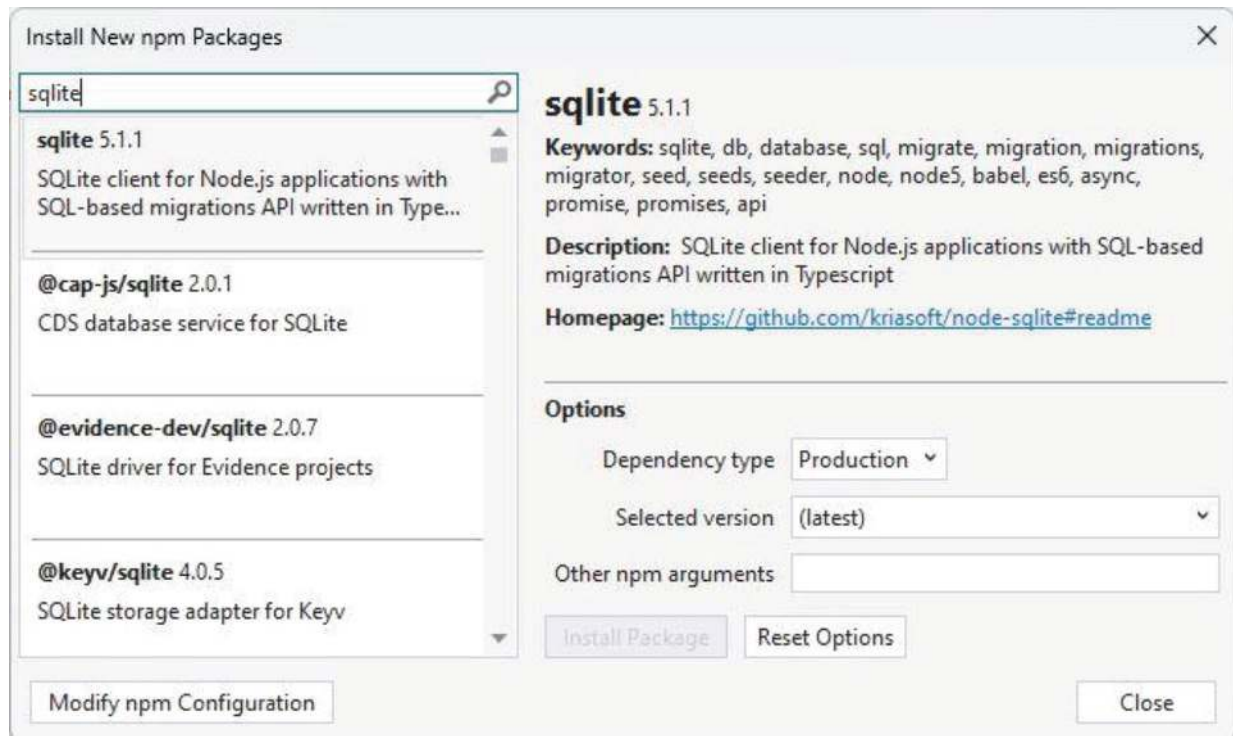


Figure 2-59 Searching for npm Packages

To learn more, visit the official documentation at <https://docs.npmjs.com/>.

## Using the CLI

You can use npm from the command line in Visual Studio. Sometimes it's faster to run npm commands directly. If, for example, you want to add a package using the CLI, you can open a terminal in Visual Studio and type `npm install <package-name> --save-dev` to install it under `devDependencies` or `npm install <package-name> --save` to install it under `dependencies` in the `package.json`. You can see an example of this in Figure 2-60.

```
Developer PowerShell
+ Developer PowerShell
*****
** Visual Studio 2022 Developer PowerShell v17.14.5
** Copyright (c) 2025 Microsoft Corporation
*****
PS C:\_temp\a\react> npm install gulp-scss-lint --save-dev
added 128 packages in 12s

12 packages are looking for funding
  run `npm fund` for details
PS C:\_temp\a\react> █
```

Figure 2-60 Installing gulp-scss-lint

Looking at the `package.json` file in Listing 2-13, you can see the added package `gulp-scss-lint` in `devDependencies`.

```

{
  "name": "reactdemo.client",
  "private": true,
  "version": "0.0.0",
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "lint": "eslint .",
    "preview": "vite preview"
  },
  "dependencies": {
    "react": "^19.1.0",
    "react-dom": "^19.1.0"
  },
  "devDependencies": {
    "@cap-js/sqlite": "^2.0.1",
    "@eslint/js": "^9.29.0",
    "@types/react": "^19.1.8",
    "@types/react-dom": "^19.1.6",
    "@vitejs/plugin-react": "^4.5.2",
    "eslint": "^9.29.0",
    "eslint-plugin-react-hooks": "^5.2.0",
    "eslint-plugin-react-refresh": "^0.4.20",
    "globals": "^16.2.0",
    "gulp-scss-lint": "^1.0.0",
    "vite": "^7.0.0"
  }
}

```

**Listing 2-13** The Updated package.json File

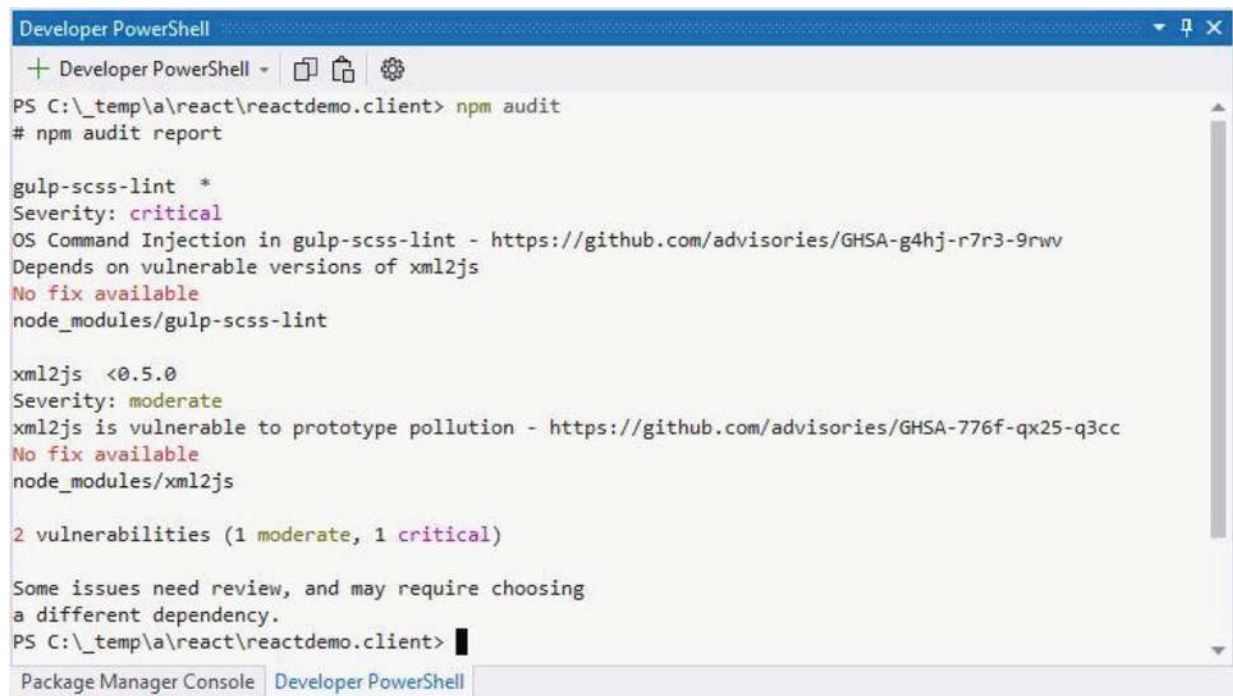
Visual Studio allows you to manage npm packages using either the built-in user interface or the integrated terminal, making it extremely convenient to handle npm packages directly from within the IDE.

## Security Considerations

Using npm packages often means pulling in third-party code. This could introduce potential security vulnerabilities if the packages you add or their dependencies have known issues. Luckily, we can rely on npm to provide tools to help us with maintaining security.

The npm CLI has a built-in security audit feature that checks your project dependencies against a database of known vulnerabilities. You can run this tool at any time by executing `npm audit` as seen in Figure [2-61](#).

This will analyze the `package.json` and send dependency info (package names and versions) to the npm registry's audit service. If any vulnerabilities are found, these will be reported along with their severity and if a fix is available or not.



```
Developer PowerShell
PS C:\_temp\a\react\reactdemo.client> npm audit
# npm audit report

gulp-scss-lint *
Severity: critical
OS Command Injection in gulp-scss-lint - https://github.com/advisories/GHSA-g4hj-r7r3-9rvv
Depends on vulnerable versions of xml2js
No fix available
node_modules/gulp-scss-lint

xml2js <0.5.0
Severity: moderate
xml2js is vulnerable to prototype pollution - https://github.com/advisories/GHSA-776f-qx25-q3cc
No fix available
node_modules/xml2js

2 vulnerabilities (1 moderate, 1 critical)

Some issues need review, and may require choosing
a different dependency.
PS C:\_temp\a\react\reactdemo.client>
```

**Figure 2-61** Running npm Audit

If there are any fixes available, you can run `npm audit fix` to automatically apply recommended updates to your `package.json` file and install the new versions. Sometimes `npm audit` will report a vulnerability for which there is no available fix yet (as seen in [Figure 2-61](#)). For these issues, you have a few options:

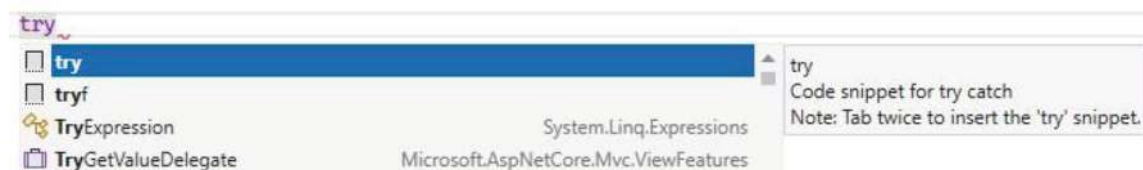
- Check if the package has an open issue or patch in progress. The audit report might even include a URL for you to review.
- Consider if you really need the package, or if you could replace it with an alternative package that is better maintained.

As a rule of thumb, make sure that you regularly run `npm outdated` to see which of your direct dependencies have newer versions.

## Creating and Using Code Snippets

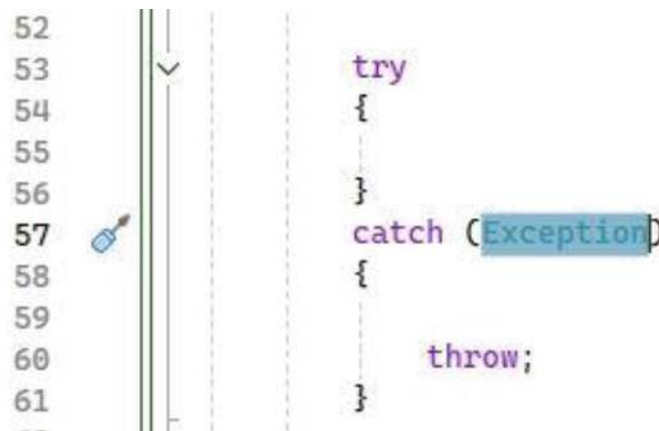
Code snippets in Visual Studio are small blocks of reusable code that you can insert into your code file by typing a shortcut and tabbing twice or using the right-click menu.

As an example ([Figure 2-62](#)), open a C# code file in Visual Studio and type the word `try` and hit the tab key twice.



**Figure 2-62** Inserting a Try Code Snippet

This inserts a try-catch into your code file and allows you to enter the specific exception type being handled, as seen highlighted in Figure 2-63.

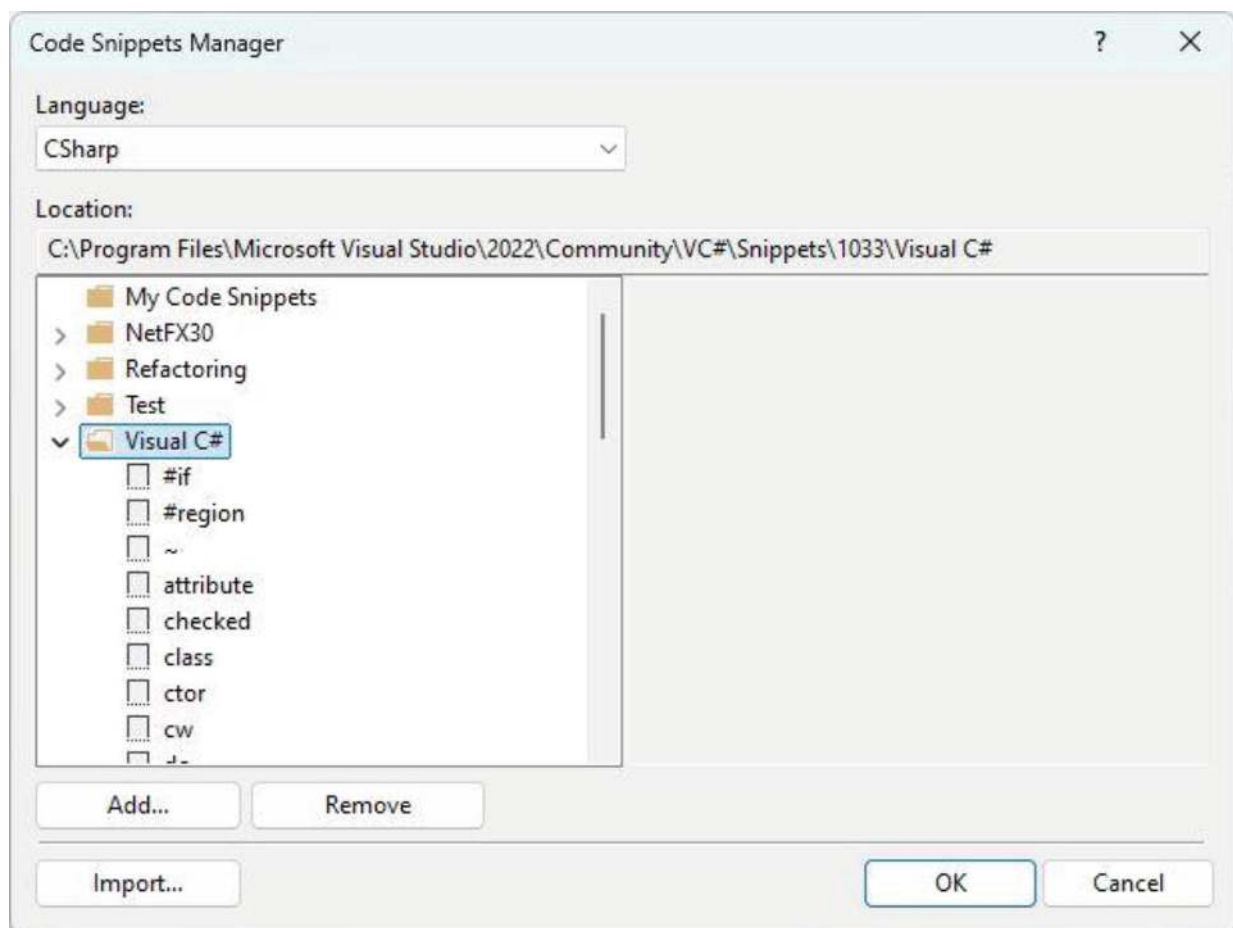


```
52  
53  
54  
55  
56  
57  
58  
59  
60  
61
```

```
try  
{  
}  
catch (Exception)  
{  
    throw;  
}
```

**Figure 2-63** The Inserted try-catch Block

If you want to see all the available code snippets, you can open the Code Snippets Manager (Figure 2-64) by going to the Tools menu and clicking Code Snippets Manager.



**Figure 2-64** Code Snippets Manager



You can also hold down Ctrl+K, Ctrl+B to open the Code Snippets Manager window. Clicking each code snippet displays the description, shortcut, snippet type (expansion or surrounds with), and author. While some shortcuts are obvious (do, else, enum, for, and so on), others are not and might take some getting used to remembering to enter the shortcut and tabbing twice to insert the snippet.

If you can't remember the shortcut, you can invoke the snippets by hitting Ctrl+K, Ctrl+X (as seen in Figure 2-65) while inside the code file you are editing. This will display a menu in place that will allow you to search for and select the specific code snippet you want to use.



**Figure 2-65** Ctrl+K, Ctrl+X to Invoke a Code Snippet

You can also right-click and select Snippets and Insert Snippet from the context menu. The last way to insert a code snippet is via the menu bar by going to Edit, IntelliSense, and clicking Insert Snippet. Visual Studio also allows developers to create their own code snippets. Let's have a look at that process next.

## Creating Code Snippets

One feature I've always felt Visual Studio is missing is a user-friendly interface for creating code snippets. Maybe that will come in a future release. For now, we'll work with what's available by defining snippets manually in XML. The basic snippet template XML is shown in Listing 2-14.

```
<?xml version="1.0" encoding="utf-8"?>
<CodeSnippets
  xmlns="http://schemas.microsoft.com/VisualStudio/2005/CodeSnippet">
  <CodeSnippet Format="1.0.0">
    <Header>
      <Title></Title>
    </Header>
    <Snippet>
      <Code Language="">
        <![CDATA[]]>
      </Code>
    </Snippet>
  </CodeSnippet>
</CodeSnippets>
```

**Listing 2-14** Basic Snippet Template

Let's assume we have created a Custom project template that includes dependency injection for a logger and a service interface. This is useful if you are creating services or controllers that require logging and other injected dependencies. You will save time and reduce boilerplate in every new project by standardizing this pattern.



But you can make this even more convenient by creating a custom code snippet that inserts the constructor and includes the injected dependencies and null checks. The snippet shown in Listing 2-15 does exactly that. It defines the fields, generates the constructor, and sets up the assignment with guard clauses.

```
<CodeSnippets
xmlns="http://schemas.microsoft.com/VisualStudio/2005/CodeSnippet">
  <CodeSnippet Format="1.0.0">
    <Header>
      <Title>Constructor Injection</Title>
      <Shortcut>ctori</Shortcut>
      <Description>Creates a constructor with injected dependencies
and null checks.</Description>
      <Author>Dirk Strauss</Author>
      <SnippetTypes>
        <SnippetType>Expansion</SnippetType>
      </SnippetTypes>
    </Header>
    <Snippet>
      <Declarations>
        <Literal>
          <ID>ClassName</ID>
          <ToolTip>The class name for ILogger<T></ToolTip>
          <Default>MyClass</Default>
        </Literal>
        <Literal>
          <ID>ServiceName</ID>
          <ToolTip>The injected service type</ToolTip>
          <Default>IMyService</Default>
        </Literal>
        <Literal>
          <ID>FieldName</ID>
          <ToolTip>The field name for the injected service</ToolTip>
          <Default>myService</Default>
        </Literal>
      </Declarations>
      <Code Language="csharp"><![CDATA[
private readonly $ServiceName$ _$FieldName$;
private readonly ILogger<$ClassName$> _logger;

public $ClassName$($ServiceName$ $FieldName$, ILogger<$ClassName$>
logger)
{
    _$FieldName$ = $FieldName$ ?? throw new
ArgumentNullException(nameof($FieldName$));
    _logger = logger ?? throw new
ArgumentNullException(nameof(logger));
}
$end$
]]></Code>
```

```

    </Snippet>
</CodeSnippet>
</CodeSnippets>

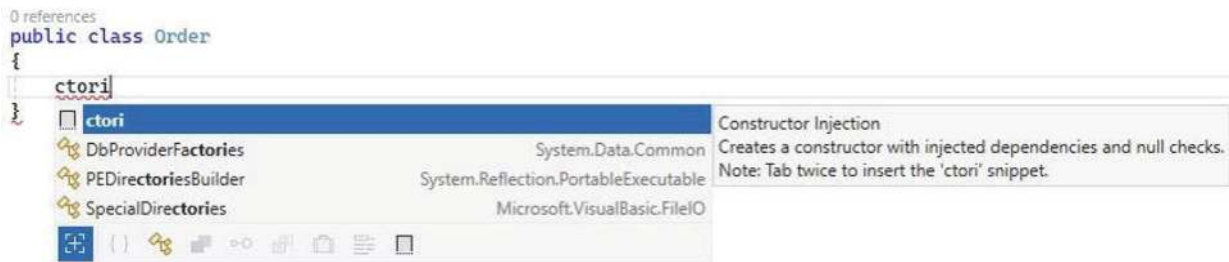
```

**Listing 2-15** Basic Logging Class

Create a file with the XML in Listing [2-15](#) and save it with a `.snippet` extension, for example `ctori.snippet`. To import this snippet into Visual Studio:

- Press `Ctrl+K, Ctrl+B` or go to **Tools ► Code Snippet Manager**.
- In the dropdown, select Visual C# (or relevant language for your snippet).
- In the Code Snippet Manager, click **Import**.
- Browse to the `.snippet` file you saved earlier and click **Open**.
- Select the folder where you want it to live, for example, “My Code Snippets” and click **Finish**.

Back in your project, create a new class and use the `ctori` shortcut as seen in Figure [2-66](#).



**Figure 2-66** Using the New Code Snippet

Hitting the tab key twice will insert the snippet into your code file, shown in Figure [2-67](#).

```

5 references
public class Order
{
    private readonly IMyService _myService;
    private readonly ILogger<MyClass> _logger;

    0 references
    public MyClass(IMyService myService, ILogger<MyClass> logger)
    {
        _myService = myService ?? throw new ArgumentNullException(nameof(myService));
        _logger = logger ?? throw new ArgumentNullException(nameof(logger));
    }
}

```

**Figure 2-67** The Inserted Code Snippet

You can now fill in the values for the placeholders, tabbing between each to go to the next. When you are done, your class should look similar to the code in Listing [2-16](#), with only the class name and value for `IMyService` differing.

```
public class Order
```

```

{
    private readonly IOrderRepository _orderRepo;
    private readonly ILogger<Order> _logger;

    public Order(IOrderRepository orderRepo, ILogger<Order> logger)
    {
        _orderRepo = orderRepo ?? throw new
ArgumentNullException(nameof(orderRepo));
        _logger = logger ?? throw new
ArgumentNullException(nameof(logger));
    }
}

```

**Listing 2-16** An Example of the Inserted Snippet

This new code snippet is now available in all your future projects. Your C# code snippets live in the Documents folder in \Visual Studio 2022\Code Snippets\Visual C#\My Code Snippets.

The code snippet schema reference is available on Microsoft Docs at the following link:  
<https://learn.microsoft.com/en-us/visualstudio/ide/code-snippets-schema-reference>.

More often than not, you will create your own code snippets based on an existing code snippet. This allows you to reuse functionality you know is working in the existing snippet and include it in your own.

Code snippets are a very powerful productivity feature in Visual Studio, but you might be wondering if code snippets still make sense with the advent of GitHub Copilot. The short answer is, yes. Their role, however, have shifted from being the go-to productivity tool for quick, reusable code snippets to being a supportive tool alongside GitHub Copilot.

## Using Bookmarks and Code Shortcuts

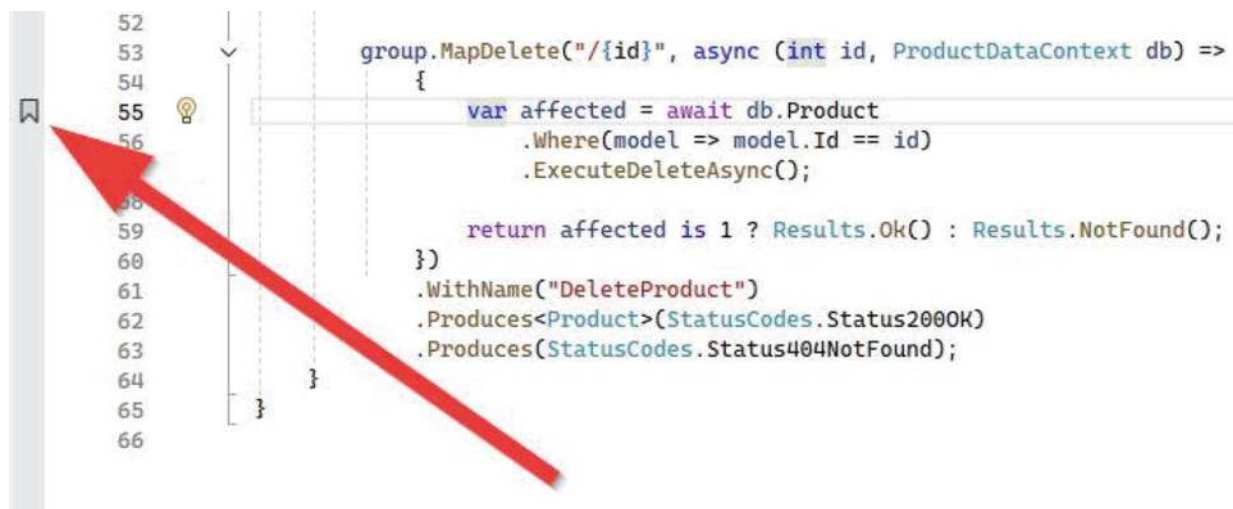
You will likely be working on an extensive code base at some point in your career. Do this for a while, and you will get bogged down with remembering where a specific bit of code is or where you need to go to get to a specific portion of logic.

Visual Studio can assist developers in bookmarking certain sections of code and add shortcuts to other areas of code. Let's have a look at what bookmarks and shortcuts are and when to use them.

### Bookmarks

The Bookmarks feature in Visual Studio has been around since the early versions of the IDE. It was designed to solve a common problem for developers, and that is keeping track of important locations in your code. It does not matter if you have a very large codebase or if you are working across multiple files and classes; bookmarks (as their real-world counterpart) help you avoid losing your place. They are very handy when debugging, trying to figure out legacy code, or reviewing related methods. You no longer need to rely on memory or scattered comments; bookmarks provide a structured, built-in method to mark and revisit key pieces of code at a later stage.

To see how bookmarks work, place your cursor at the line of code you want to return to, and hold down Ctrl+K, Ctrl+K, and Visual Studio will add a bookmark, as seen in Figure [2-68](#).

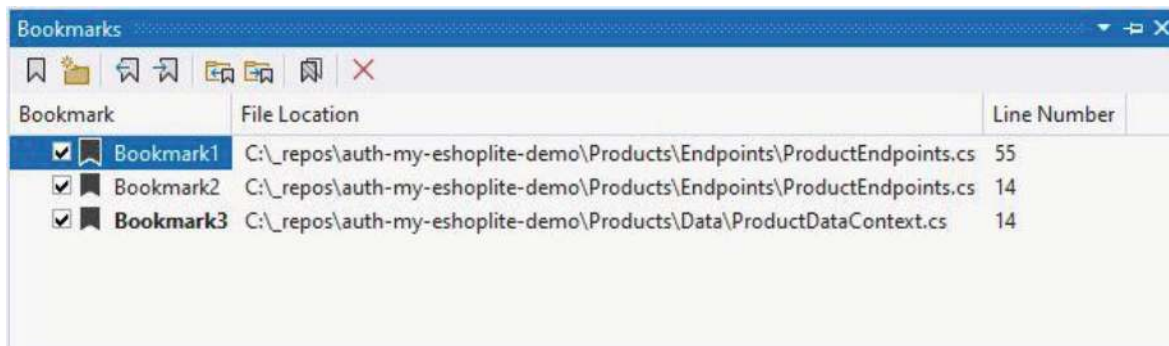


**Figure 2-68** Bookmark in Visual Studio

The bookmark is added to the side of the code editor and is indicated by a single grey bookmark icon.

To see all the bookmarks in your project, you can hold down Ctrl+K, Ctrl+W, or go to the View menu item, select Other Windows, and then click Bookmark window.

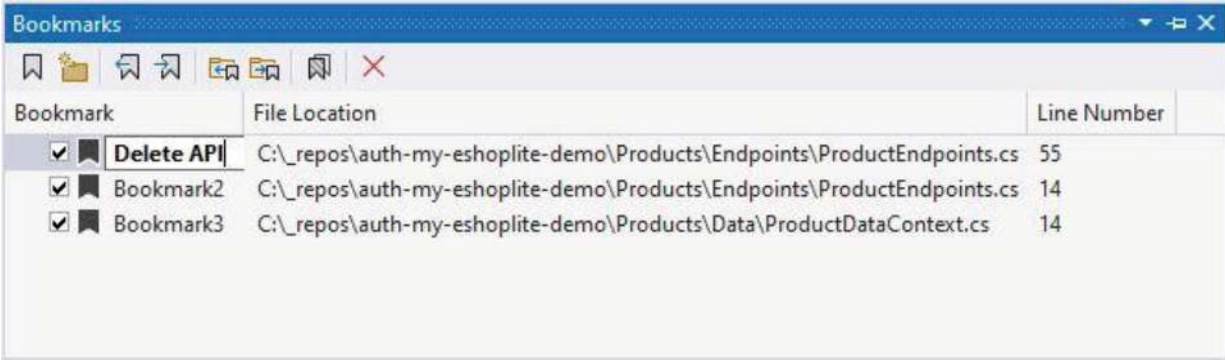
The Bookmarks window is displayed as shown in Figure 2-69. From the toolbar in the Bookmarks window, you can group bookmarks in folders, navigate between bookmarks, navigate between bookmarks in the current folder, toggle a bookmark on the currently selected line in code, disable all bookmarks, and delete bookmarks.



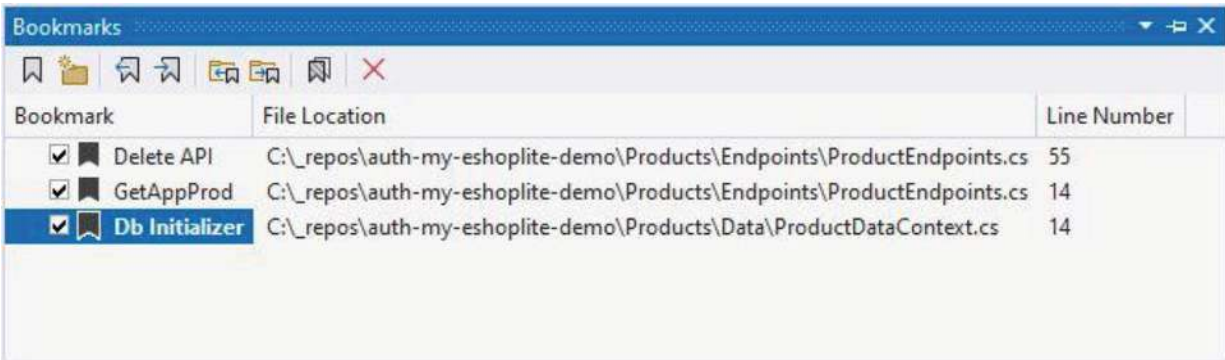
**Figure 2-69** Bookmarks Window

One less obvious feature of the Bookmarks window is the ability to rename bookmarks. To rename a bookmark, click a selected bookmark, and you will see that the name (in this case, "Bookmark1") becomes editable.

Now you can rename your bookmark to something more relevant to what you need to remember, as shown in Figure 2-70. Compare Figure 2-70 with the bookmark windows as seen in Figure 2-71. Renaming your bookmarks to more suitable names gives your bookmarks window more meaning.



**Figure 2-70** Renamed Bookmark



**Figure 2-71** Bookmarks Collection

Now, click the delete button on the Bookmarks toolbar. The bookmark is deleted without any confirmation from the user.

This is something I can almost understand. It might become cumbersome to have to confirm every bookmark deletion, especially when you want to remove only a subsection of bookmarks from your collection.

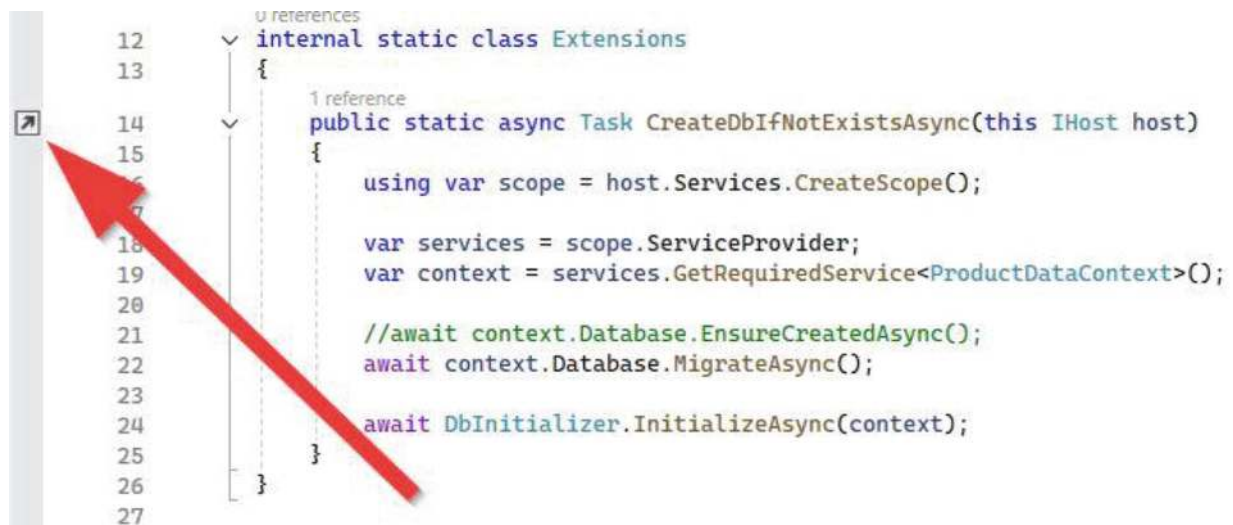
For this reason, I only use bookmarks as a short-term solution to remind me to go and perform some action in code or refactor something I think needs refactoring. The ease at which they can be deleted, however, does make me wary of using them to mark important lines of code that I could potentially want to revisit often. It is for this reason that I treat a bookmark as something I will come back to within the next day or so. Something I don't want to put off doing. It is, therefore, a temporary placeholder to something I need to revisit.

But what if I wanted to go and add a more permanent pointer to some logic in the code? This is where code shortcuts come in. Let's have a look at this next.

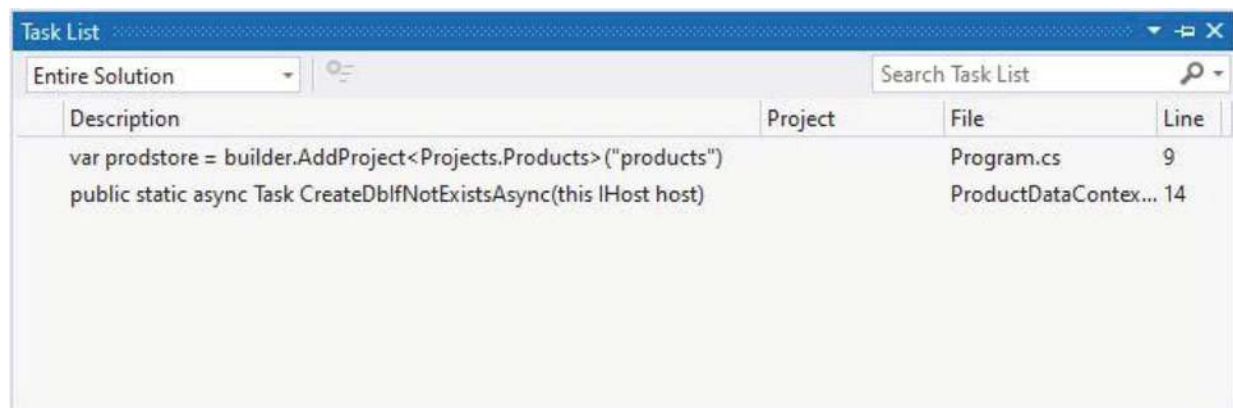
## Code Shortcuts

The ability to add code shortcuts in Visual Studio is more helpful when you need to jump to a certain section of code regularly. To add a shortcut to a specific section of code in Visual Studio, you need to place your cursor on the line of code you need to revisit and press Ctrl+K, Ctrl+H.

Visual Studio will then add the shortcut, as seen in [Figure 2-72](#). To view all the shortcuts added to your project, hold down Ctrl+\, Ctrl+T, or go to the View menu, and select Task List to open the Task List window, shown in [Figure 2-73](#).



**Figure 2-72** Code Shortcut Added Indicator



**Figure 2-73** The Task List

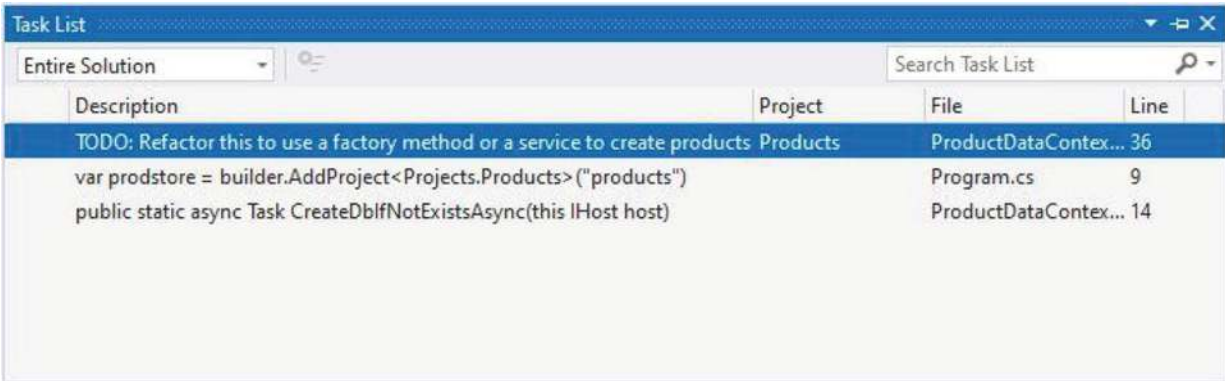
In some ways, I prefer the Task List over bookmarks since I can quickly add reminders by inserting `//TODO:` in my code. With your Task List open, go to any place in your code, and add the comment in Listing 2-17.

```
// TODO: Remember to do something here
```

**Listing 2-17** TODO Comment

Now, have a look at your Task List. You will notice that the `TODO` comment has been added to your Task List. I wanted to remember to refactor some code. Adding that `TODO` as shown in Figure 2-74 puts the reminder front and center so that I will not forget.





**Figure 2-74** TODO Comments in the Task List

This is a nice and quick method for adding reminders to your code so that you can easily refer, and navigate to them by double-clicking the item in the Task List. You can, therefore, use the Task List to take you directly to the relevant location in the code.

In Visual Studio, `TODO` is what we call a predefined token. Therefore, a comment in your code that uses a predefined token will appear in your Task List. The tokenized comment is made up of the following:

- The comment marker, which is `//`
- The predefined token (`TODO` in our example)
- The rest of the comment

The code in Listing [2-17](#) is a valid comment using a token and will appear in the Task List. Visual Studio includes the following default tokens:

- HACK
- TODO
- UNDONE
- UnresolvedMergeConflict

These are by no means case sensitive and will appear in your Task List if following the form in Listing [2-17](#). You can also add your own custom tokens. Let's see how to do that next.

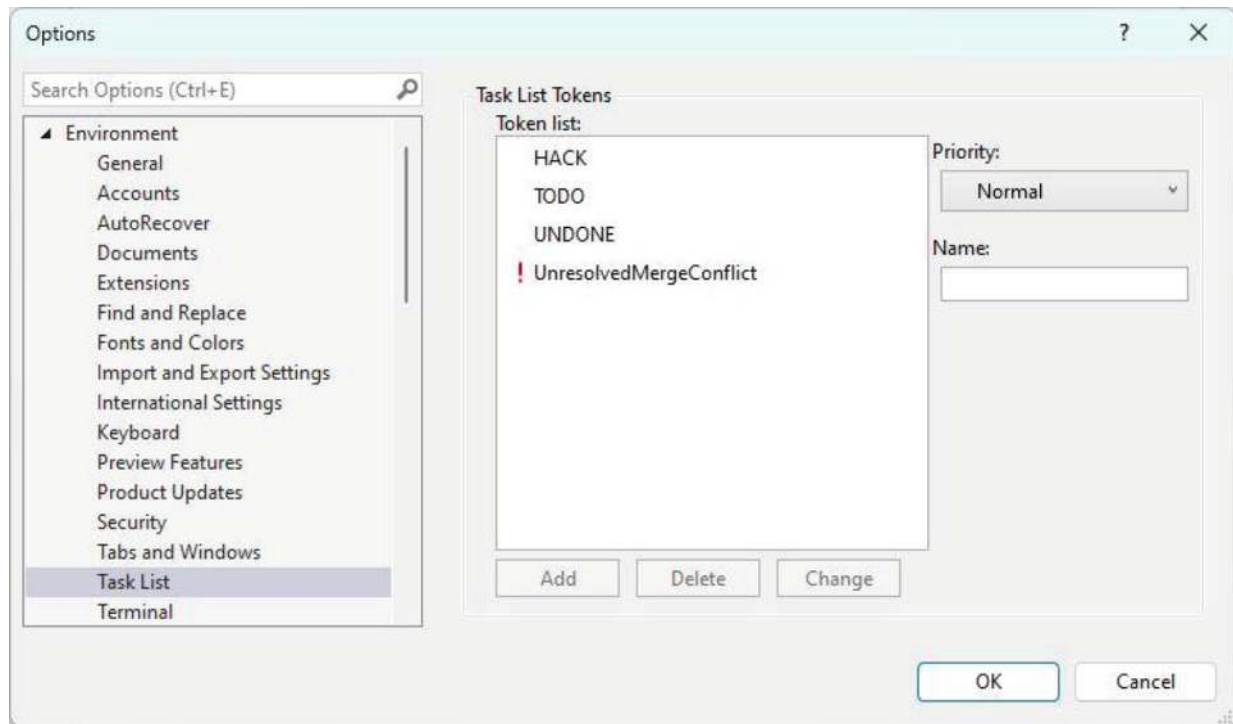
## Adding Custom Tokens

I like the idea of `TODO` to add items to my Task List, but I would also like to add a custom token to add an entry in my Task List, which is a nice-to-have feature. Something that is less restrictive than a `TODO`, because that implies that this action must be completed.

I do not want to have a bunch of `TODO` entries for items that are simply nice-to-have features. For this reason, I want to add a custom token called `NOTE`, simply as a reminder to look at something if and when I have the time.

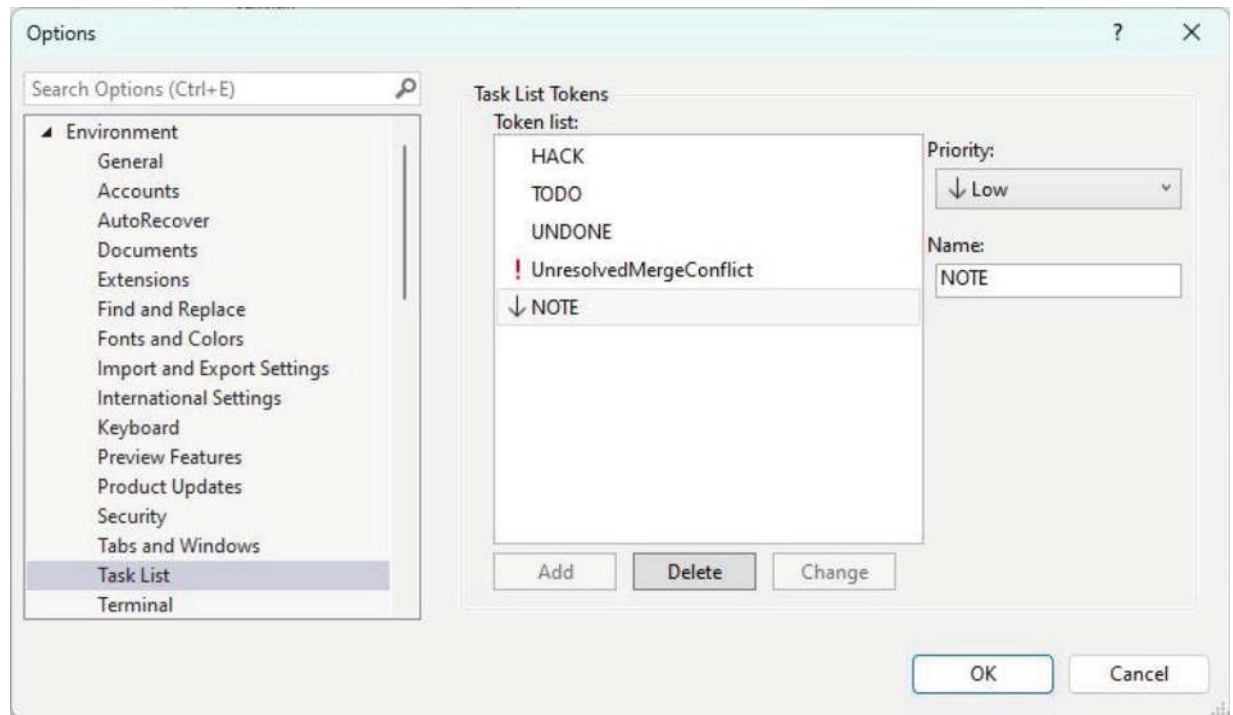
To add the custom token, go to the Tools menu and click Options. Under Environment, select Task List as seen in Figure [2-75](#).





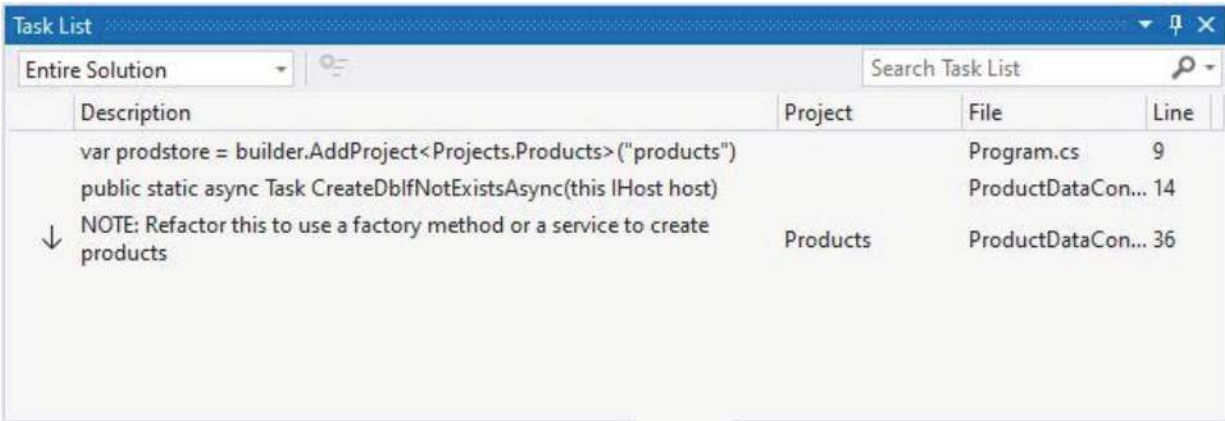
**Figure 2-75** Add Custom Tokens

In the Name text box, add the word NOTE and set the priority to Low. Then click the Add button. The custom token NOTE is added, as shown in Figure [2-76](#).



**Figure 2-76** The Custom Token Added

Adding a NOTE to your code pops up in your Task List as a low-priority task (Figure [2-77](#)).



**Figure 2-77** Adding a NOTE Token Comment

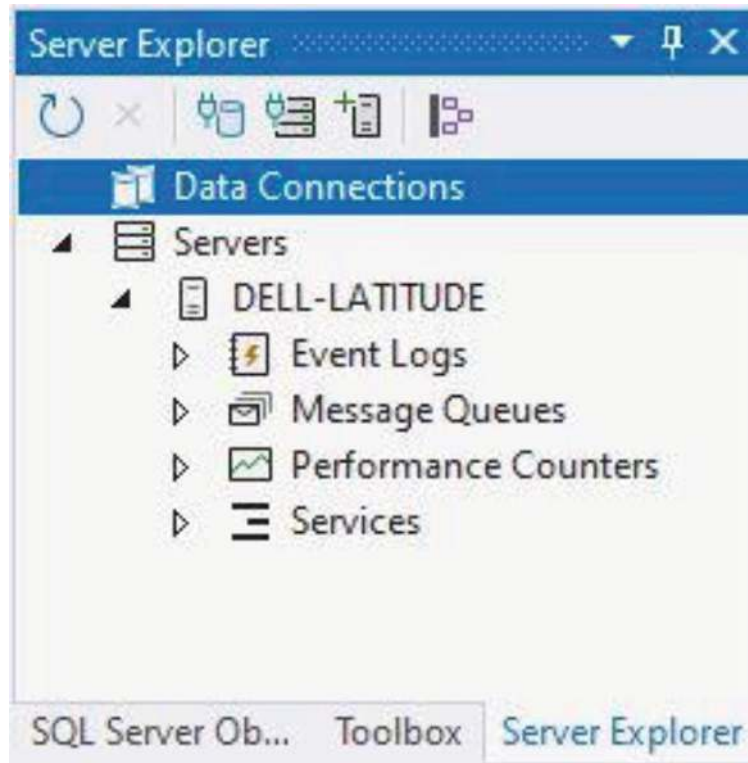
Being able to add custom tokens in Visual Studio, as well as applying a priority to each, allows you to be very specific with comments that contain tokens. This way, you can significantly increase the ease and efficiency of navigating a large codebase.

## The Server Explorer

As the name suggests, the Server Explorer provides a quick and easy way of accessing servers. You can use it to test connections and view SQL Server databases or any databases that have the ADO.NET provider installed.

You can access the Server Explorer by holding down Ctrl+Alt+S or by going to the View menu and clicking Server Explorer.

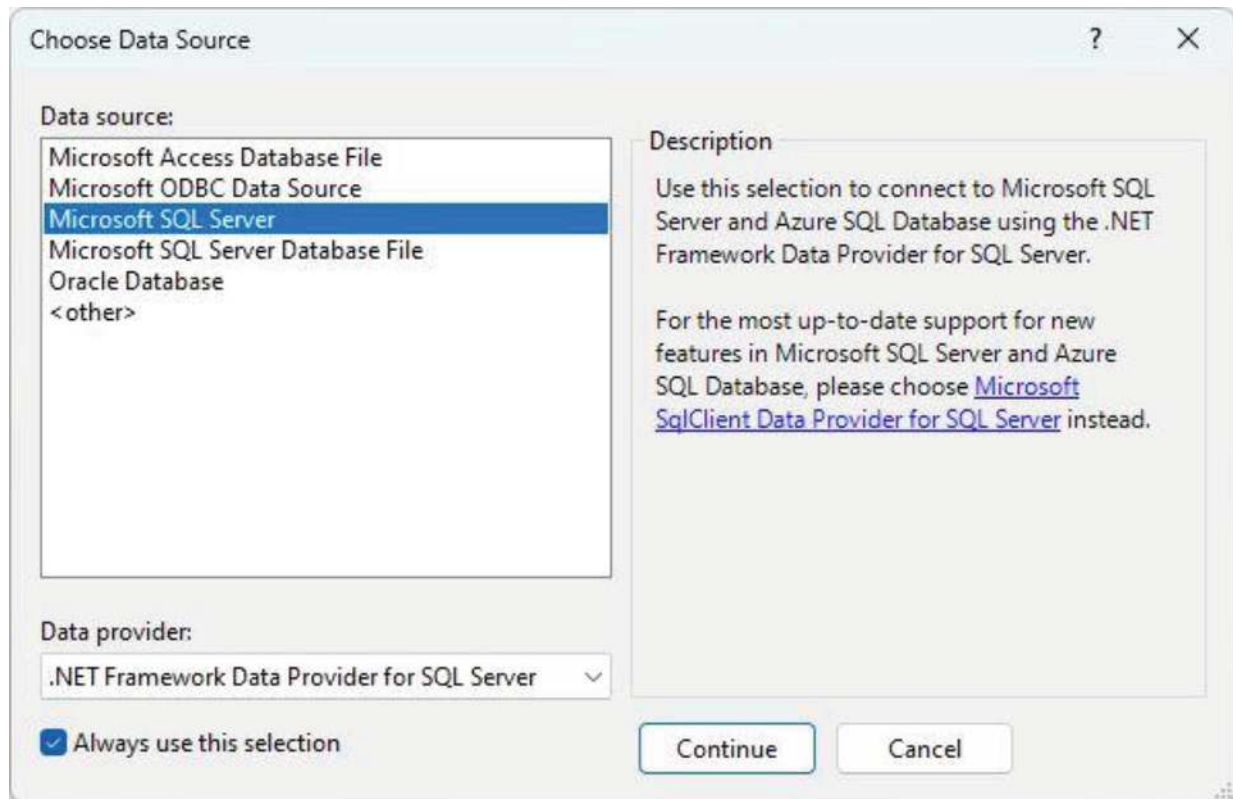
Figure [2-78](#) shows that the Server Explorer offers access to Event Logs, Message Queues, Performance Counters, and Services on my local machine (DELL-LATITUDE).



**Figure 2-78** Server Explorer

I have a local instance of SQL Server installed, so now I can connect to this instance directly from within Visual Studio by clicking Connect to Database.

This displays a window allowing you to choose a data source, as seen in [Figure 2-79](#). You can connect to various data sources, but we are only interested in Microsoft SQL Server for now. Select that from the list and click Continue.



**Figure 2-79** Choose Data Source

The following window (Figure 2-80) allows you to define your connection to the database. Here, you need to specify the server name and the authentication type, and if SQL Server Authentication is selected, provide the username and password.

The 'Add Connection' dialog box is shown with the following settings:

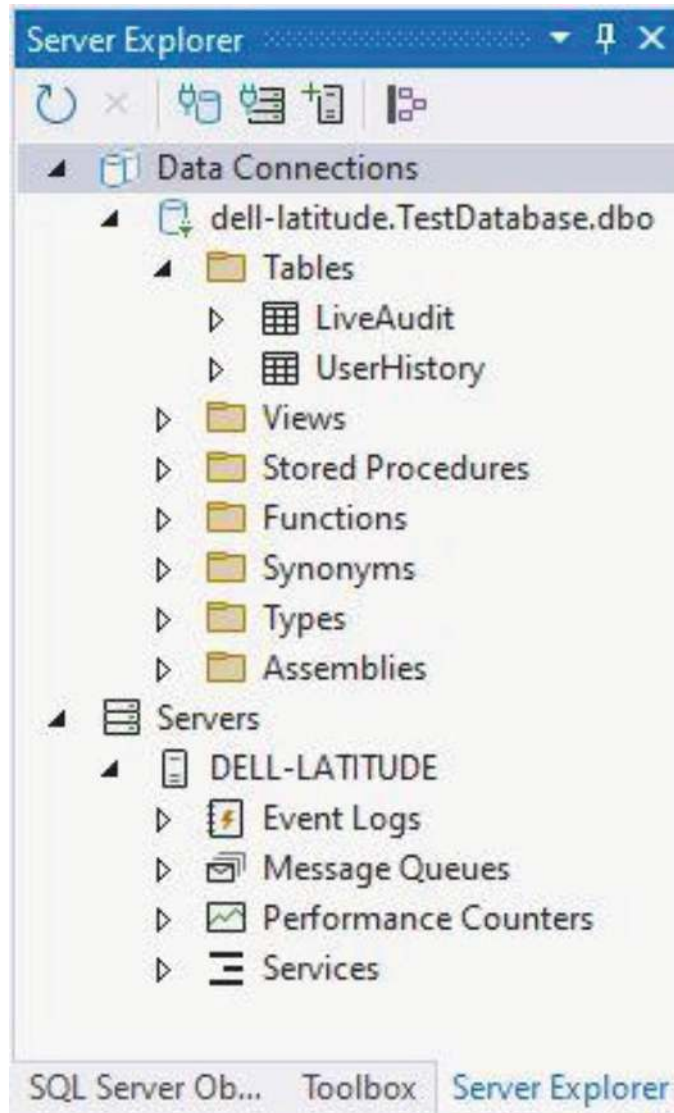
- Data source:** Microsoft SQL Server (SqlClient)
- Server name:** DELL-LATITUDE
- Log on to the server:**
  - Authentication:** SQL Server Authentication
  - User name:** sa
  - Password:** (masked with dots)
  - Encrypt:** Mandatory (True)
  - ☒ Trust Server Certificate
  - ☐ Save my password
- Connect to a database:**
  - ☒ Select or enter a database name: TestDatabase
  - ☐ Attach a database file: (empty field)

Buttons at the bottom: Test Connection, OK, Cancel, and Advanced...

**Figure 2-80** Add Connection

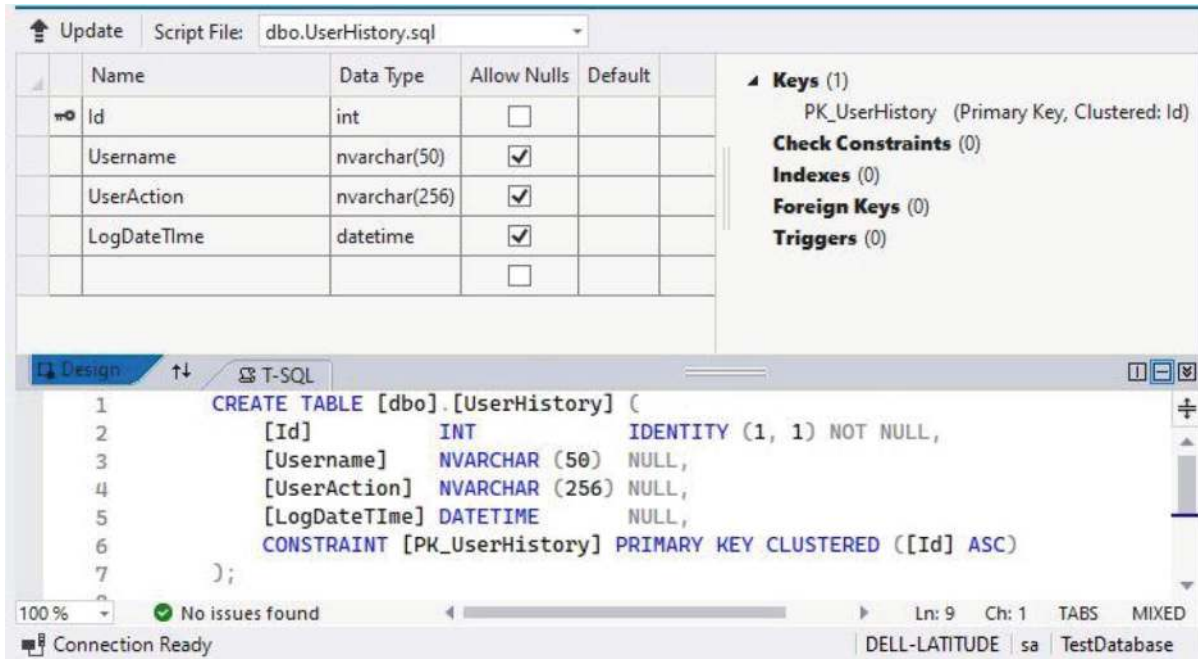
You can now select a database from the list to connect to. Click on the Test Connection button to check if the connection settings are correct.

After adding the database to your Server Explorer, you will see the instance added to your list from where you can expand the various nodes to view Tables, Views, and Stored Procedures, as shown in [Figure 2-81](#).



**Figure 2-81** Database Added to Server Explorer

By double-clicking a table, Visual Studio displays the table designer for you and a create table SQL statement (Figure [2-82](#)).



**Figure 2-82** Table Designer

From this window, you can easily update the table. The create table statement in [Figure 2-82](#) is listed in [Listing 2-18](#).

```

CREATE TABLE [dbo].[UserHistory] (
    [Id] INT IDENTITY (1, 1) NOT NULL,
    [Username] NVARCHAR (50) NULL,
    [UserAction] NVARCHAR (256) NULL,
    [LogDateTime] DATETIME NULL,
    CONSTRAINT [PK_UserHistory] PRIMARY KEY CLUSTERED ([Id] ASC)
);
  
```

**Listing 2-18** Create table statement.

We can now modify the UserHistory table by altering the SQL statement as shown in [Listing 2-19](#).


```

CREATE TABLE [dbo].[UserHistory] (
    [Id] INT IDENTITY (1, 1) NOT NULL,
    [Username] VARCHAR (50) NOT NULL,
    [UserAction] VARCHAR (256) NOT NULL,
    [LogDateTime] DATETIME NULL,
    [LogCategory] VARCHAR (5) NULL,
    CONSTRAINT [PK_UserHistory] PRIMARY KEY CLUSTERED ([Id] ASC)
);
  
```

**Listing 2-19** Modified Create Table Statement

I want to add a log category field to the table. When I modify the create table statement, I see the changes reflected in the table designer, as seen in [Figure 2-83](#).

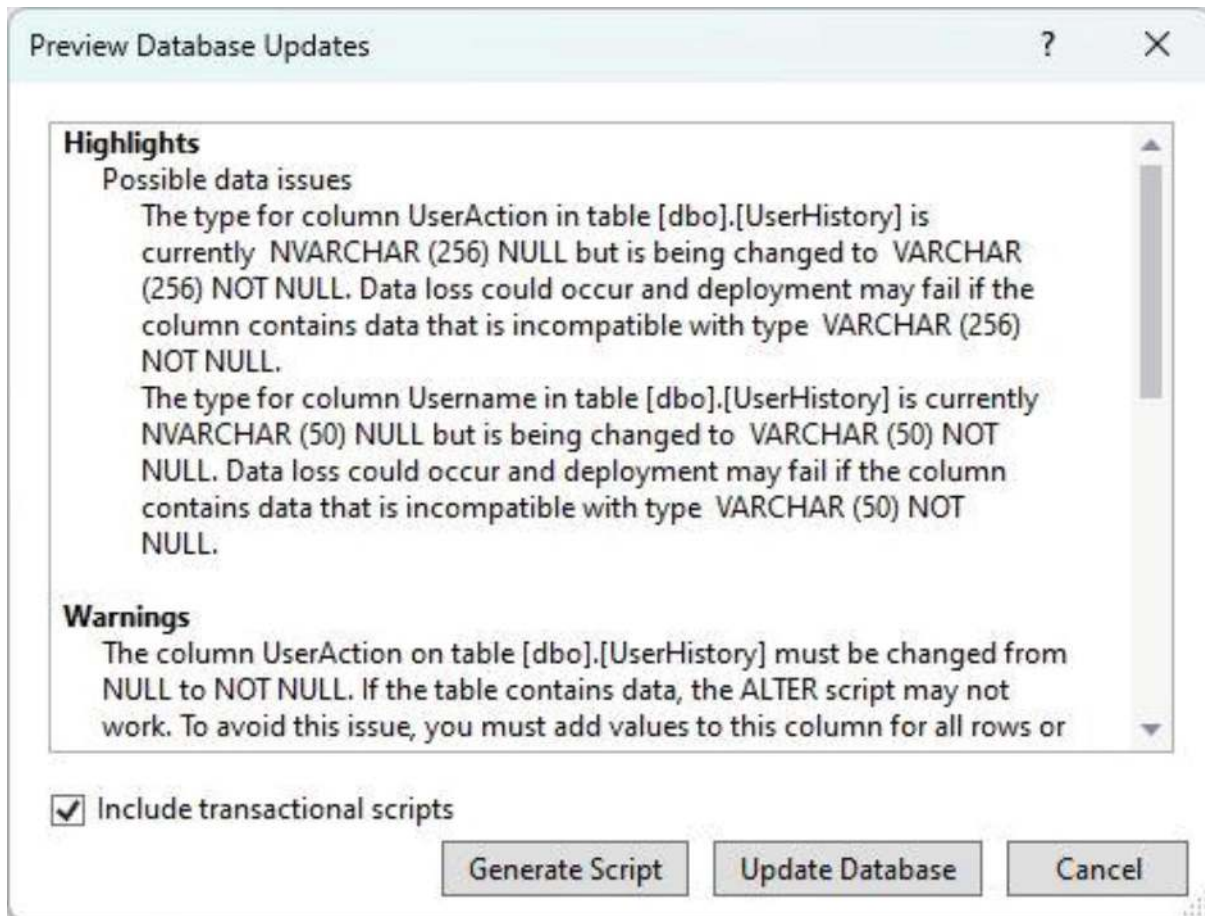


Update		Script File: <input type="text" value="dbo.UserHistory.sql*"/>			
	Name	Data Type	Allow Nulls	Default	
	Id	int	<input type="checkbox"/>		
	Username	varchar(50)	<input type="checkbox"/>		
	UserAction	varchar(256)	<input type="checkbox"/>		
	LogDateTime	datetime	<input checked="" type="checkbox"/>		
	LogCategory	varchar(5)	<input checked="" type="checkbox"/>		
			<input type="checkbox"/>		

**Figure 2-83** Table Design Updated

The changes have not been applied to my table yet. To update the table, I need to click the Update button.

This allows me to preview the database updates, as seen in Figure [2-84](#). If you do not want to let Visual Studio update the table, you can have it generate the script by clicking the Generate Script button. Alternatively, you can go ahead and click the Update Database button after reviewing any warnings and possible data issues in the Preview Database Updates window.

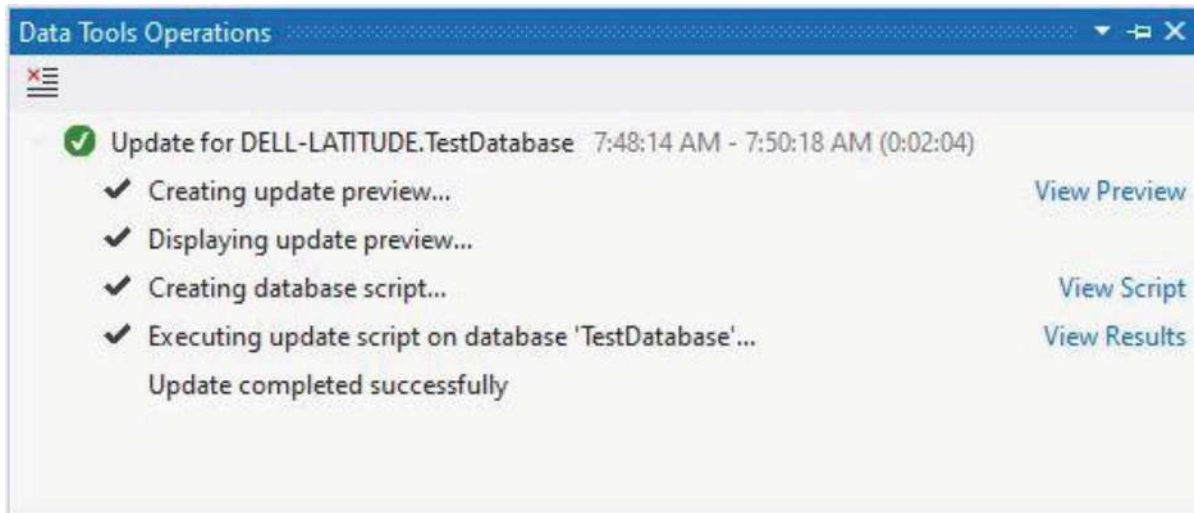


**Figure 2-84** Preview Database Updates

This will then start the process of updating the database table with the changes you made.

I am working with an empty table. I can, therefore, ignore the possible data issues and warnings in the Preview Database Updates window.

After the update, you can see the results in the Data Tools Operations window (Figure [2-85](#)). From here, you can view the script and the results by clicking on the links on the right of the Data Tools Operations window.



**Figure 2-85** Data Tools Operations

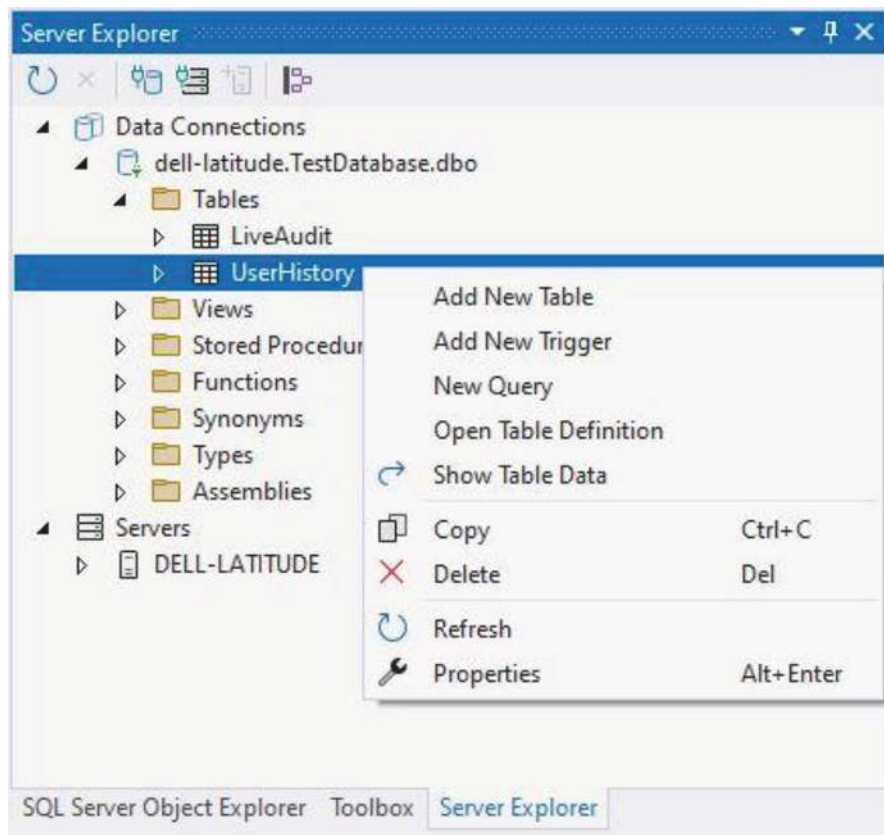
## Running SQL Queries

Visual Studio's Server Explorer is more than just a view into your connected databases and database objects. It enables direct interaction with your connected databases through an integrated SQL query editor.

The functionality is made possible by Visual Studio's underlying data provider architecture, which uses ADO.NET to connect to various database engines such as SQL Server, Azure SQL, etc. You can communicate with the database using standard T-SQL over a managed connection.

This gives developers a lightweight environment to run queries, inspect the output, and make changes. All this can be done without leaving the IDE. Tight integration like this is great for quick data exploration and testing of small queries while coding.

Go ahead, right-click a table (Figure [2-86](#)), and click New Query from the context menu.



**Figure 2-86** Run a SQL Query

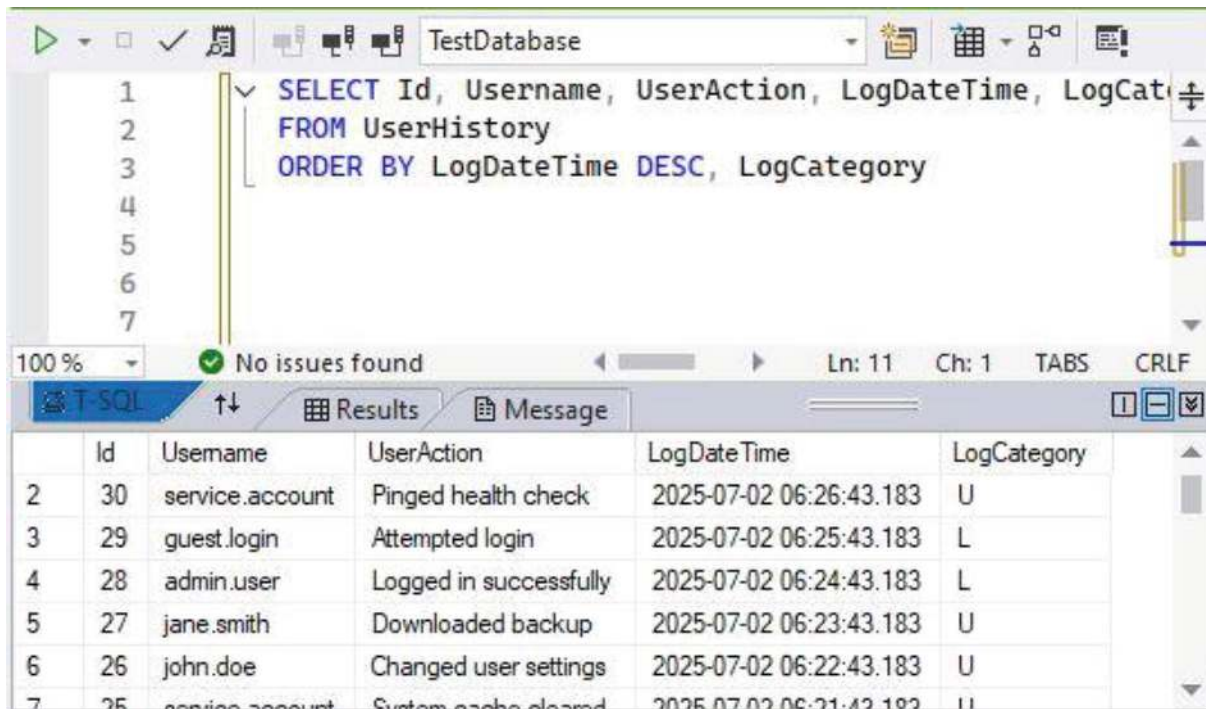
**Note** That the context menu changes depending on what item you have right-clicked in the Server Explorer. When right-clicking a table, you see items related to a SQL table. When right-clicking a View, you see items specific to the View, such as Show Results and Open View Definition. The context menu will display the Execute command when right-clicking a Stored Procedure.

Copy the SQL query in Listing [2-20](#). You will have had to create the table using the CREATE statement in Listing [2-19](#).

```
SELECT Id, Username, UserAction, LogDateTime, LogCategory
FROM UserHistory
ORDER BY LogDateTime DESC, LogCategory
```

**Listing 2-20** SQL Select Statement

When you have pasted the SQL statement (Figure [2-87](#)), execute it by clicking the run button, holding down Ctrl+Shift+E, or executing it with the debugger Alt+F5.



**Figure 2-87** Running a Select Statement

If you're used to pressing F5 in SQL Server Management Studio, you might find yourself accidentally starting the Visual Studio debugger instead of running the query. I find clicking the run button helps me avoid this muscle memory faux pas.

Adding additional items to the table is quickly done by running `INSERT` statements as illustrated in Listing 2-21.

```
INSERT INTO [dbo].[UserHistory]
(
    [Username],
    [UserAction],
    [LogDateTime],
    [LogCategory]
)
VALUES
(
    'TestUser',
    'Logged into the system',
    GETDATE(),
    'INFO'
)
```

**Listing 2-21** Insert Statement

If we rerun the `SELECT` statement, you will see that the entry has been added to the table, as seen in Figure 2-88.

TestDatabase

```

1 SELECT Id, Username, UserAction, LogDateTime, LogCategory
2 FROM UserHistory
3 ORDER BY LogDateTime DESC, LogCategory
4
5
6
7

```

100 % No issues found Ln: 11 Ch: 1 TABS CRLF

	Id	Username	UserAction	LogDateTime	LogCategory
1	50	TestUser	Logged into the system	2025-07-02 06:55:40.963	INFO
2	30	service.account	Pinged health check	2025-07-02 06:26:43.183	U
3	29	guest.login	Attempted login	2025-07-02 06:25:43.183	L
4	28	admin.user	Logged in successfully	2025-07-02 06:24:43.183	L
5	27	jane.smith	Downloaded backup	2025-07-02 06:23:43.183	U
6	26	john.doe	Changed user settings	2025-07-02 06:22:43.183	U

**Figure 2-88** New User History Entry Inserted

From the results in Figure 2-88, you will notice an invalid value for LogCategory. Let's change that by running the SQL statement in Listing 2-22.

```

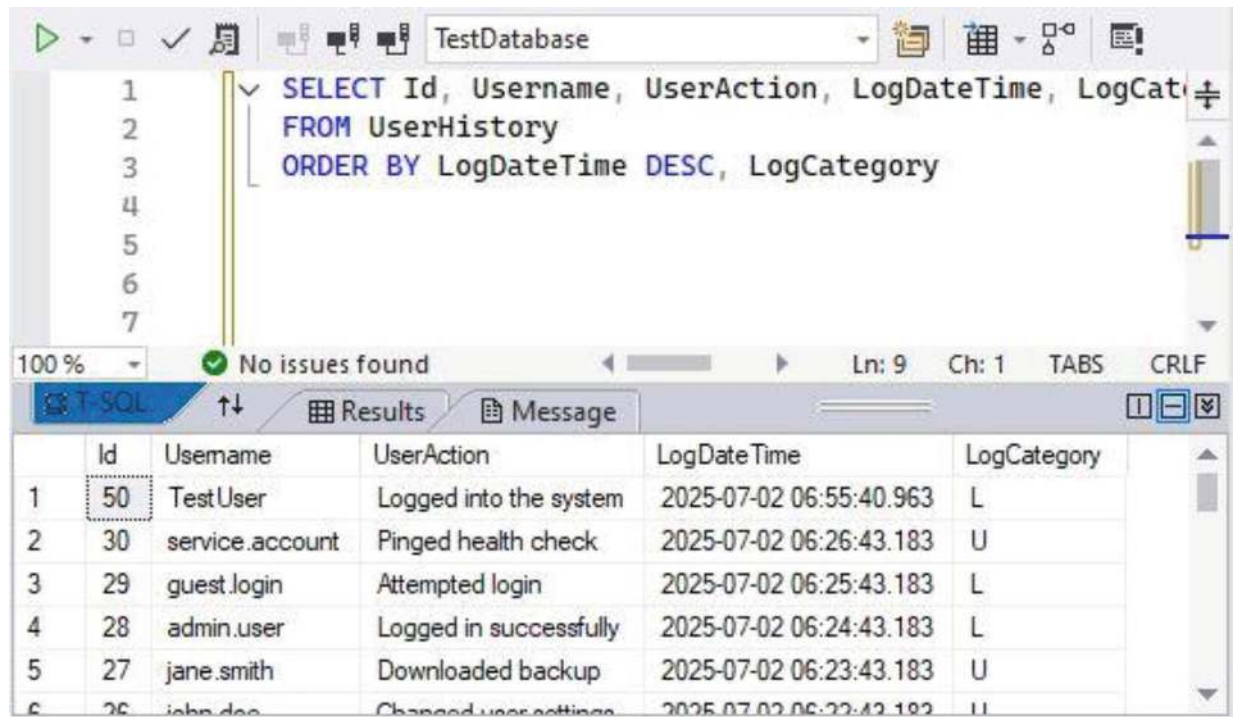
UPDATE [dbo].[UserHistory]
SET
    [LogCategory] = 'L'
WHERE [Id] = 50

```

**Listing 2-22** SQL Update Statement

When we look at the table data after the UPDATE statement (Figure 2-89), you will see that the table has been updated to display the correct LogCategory value for the TestUser in the table.





**Figure 2-89** TestUser Entry Updated

While running SQL statements directly from Visual Studio might not seem revolutionary, the convenience it provides can't be overstated. Being able to interact with your database, run queries, and view results from the queries without leaving the IDE streamlines your workflow and reduces context switching.

Context switching shifts your focus between different tasks or mental models. This incurs real cognitive and time costs. When developers have to break focus, it is estimated that it takes 20 to 25 minutes to completely regain concentration on the original task. Frequent interruptions such as pings, meetings, or build waits can consume hours of focused work per week. Understand what impact this can have in your daily workflow, and find ways to mitigate the effect of context switching on your productivity.

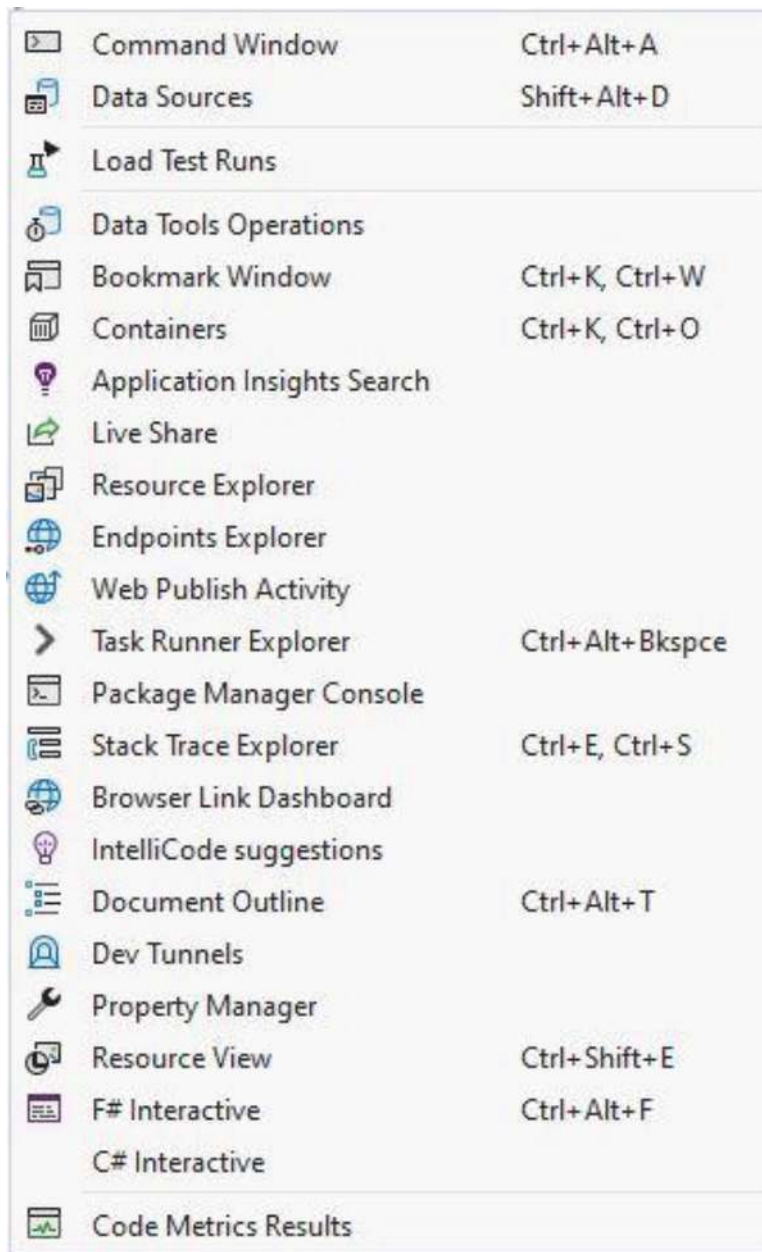
The Server Explorer provides much more functionality than what I have covered here, including being able to manage connections, exploring database assets, and even generating scripts. I encourage you to spend some time exploring these features further. You might be surprised at how much it can enhance your productivity.

## Visual Studio Windows

Many developers fall into a comfortable routine when working in Visual Studio, often relying on familiar workflows and overlooking powerful tools that could significantly improve their productivity.

While there is nothing inherently wrong with sticking to what you know and to what works for you, it often comes at the cost of missing out on lesser-known but highly effective features provided by Visual Studio. In this section, we will take a closer look at two utilities found under the View ► Other Windows menu, as shown in Figure [2-90](#).





**Figure 2-90** Other Windows in Visual Studio

There are too many windows to discuss in this chapter, but I will touch on two that I find very useful.

## C# Interactive

How often have you wanted to test a tiny bit of code to see if it works correctly? With C# Interactive, you can do just that without having to debug your entire solution. Found toward the bottom of the View ► Other Windows menu, C# Interactive is almost hidden. But gems usually are, and you'll love using it if you don't already.

Click C# Interactive and paste the following code shown in Listing [2-23](#). After pasting the code into C# Interactive, hit the Enter key to run the code.

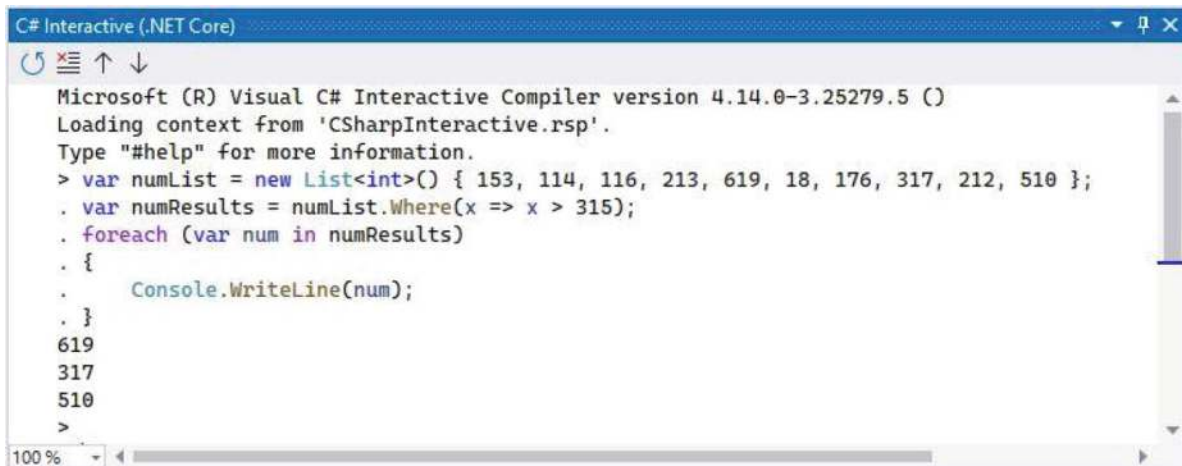
```

var numList = new List<int>() { 153, 114, 116, 213, 619, 18, 176,
317, 212, 510 };
var numResults = numList.Where(x => x > 315);
foreach(var num in numResults)
{
    Console.WriteLine(num);
}

```

**Listing 2-23** Running a LINQ Query

The results are displayed below the code you pasted. Your C# Interactive window should now appear as illustrated in Figure [2-91](#).

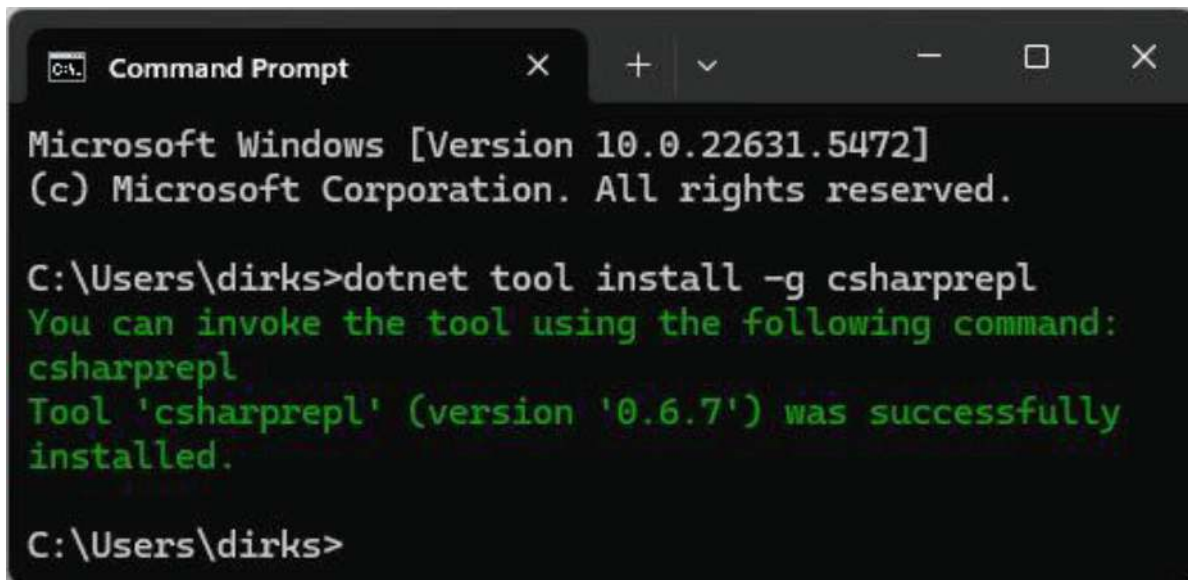


**Figure 2-91** C# Interactive Code Results

C# Interactive is what we refer to as a REPL (Read-Eval-Print Loop). Being able to input expressions that are evaluated and having the results returned makes on-the-spot debugging possible in Visual Studio.

C# Interactive supports IntelliSense, so you get the same kind of editor experience as in Visual Studio. For a list of available keyboard shortcuts, REPL commands, and Script directives that C# Interactive supports, just type in `#help` and press the Enter key.

Incidentally, you can run a C# REPL from the Command Prompt too. Install it using the `dotnet tool install -g csharp REPL` command as seen in Figure [2-92](#).



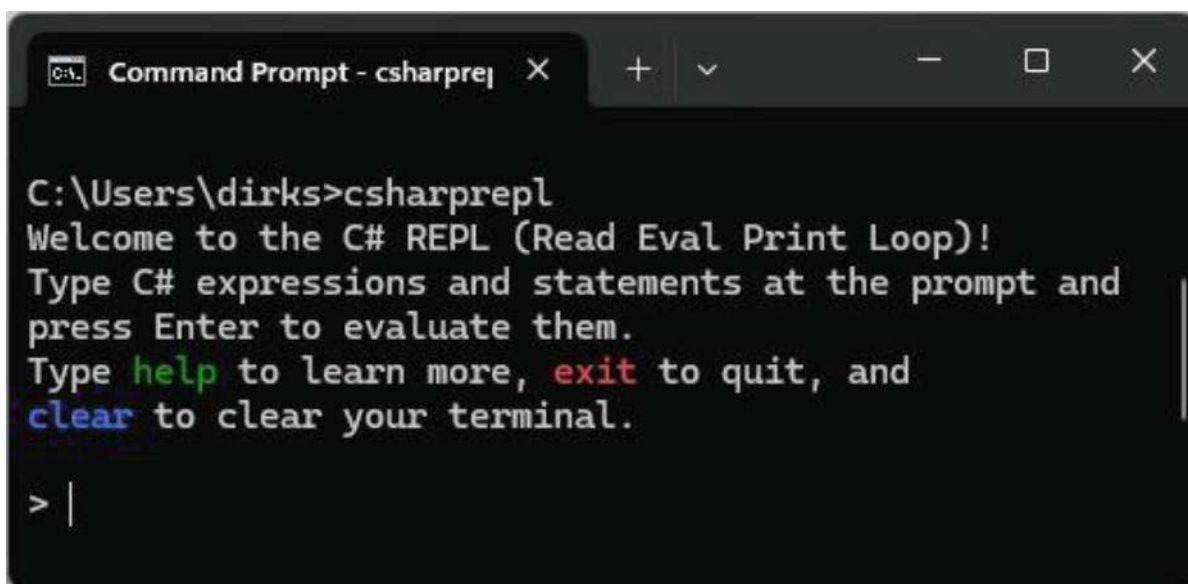
```
Microsoft Windows [Version 10.0.22631.5472]
(c) Microsoft Corporation. All rights reserved.

C:\Users\dirks>dotnet tool install -g csharp REPL
You can invoke the tool using the following command:
csharp REPL
Tool 'csharp REPL' (version '0.6.7') was successfully
installed.

C:\Users\dirks>
```

**Figure 2-92** Installing the C# REPL

You can then invoke the REPL from the Command Prompt using `csharp REPL` as seen in Figure [2-93](#).



```
C:\Users\dirks>csharp REPL
Welcome to the C# REPL (Read Eval Print Loop)!
Type C# expressions and statements at the prompt and
press Enter to evaluate them.
Type help to learn more, exit to quit, and
clear to clear your terminal.

> |
```

**Figure 2-93** Invoking the C# REPL

This can be handy when you are not using Visual Studio and need to run a few lines of code.

## Code Metrics Results

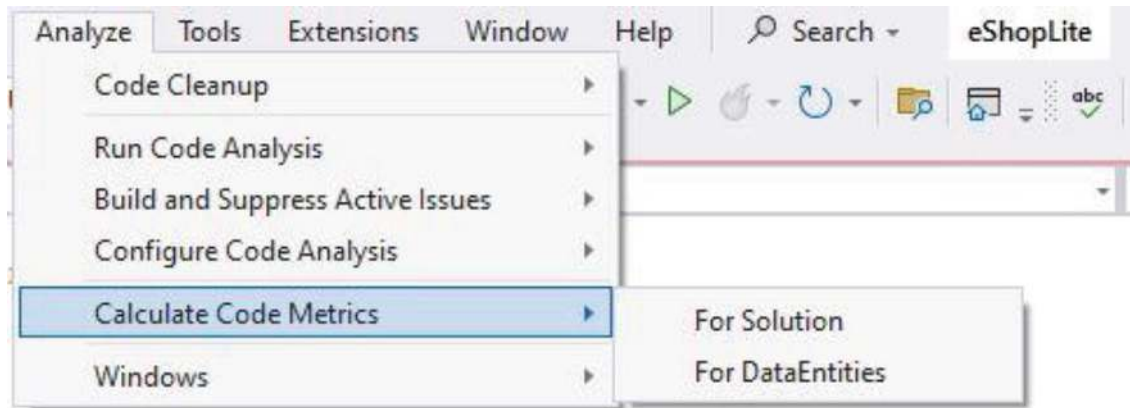
The eShopLite project we have been using is not complex. It is just used to illustrate the concepts in this book. If you take a more complex project, one of the projects you have worked on at work, this next screen might look a lot different.

Code Metrics Results (Figure [2-94](#)) is a set of measurements that allow developers to gain a better insight into the code they produce.

Hierarchy	Maintainabil...	Cyclomatic ...	Depth of Inh...	Class Coupli...	Lines of Sour...	Lines of Executa...
DataEntities (Debug)	89	63	2	45	550	114
eShopLite.AppHost (Debug)	78	9	1	20	52	18
eShopLite.ServiceDefaults (Deb)	69	7	1	41	0	27
Products (Debug)	66	17	2	71	270	79
Store (Debug)	61	23	3	64	1,018	170

**Figure 2-94** Code Metrics Results

Modern software can become quite complex, and this makes it harder to ensure reliability and maintainability. This is where Visual Studio’s Code Metrics provide a quantitative insight into the quality of your code.



**Figure 2-95** Calculate Code Metrics

If you click on Analyze ► Calculate Code Metrics as shown in Figure 2-95, Visual Studio will evaluate your code and report five key metrics. These are:

- Maintainability Index
- Cyclomatic Complexity
- Depth of Inheritance
- Class Coupling
- Lines of Code

This is baked into Visual Studio; you don’t need any third-party tools to gain valuable insights into your code. These metrics act as an early warning system for code smells, and you need to pay attention to what they tell you.

Vibe Coding is not edgy. It’s technical debt in skinny jeans. You’re not building a prototype for Burning Man, you’re shipping software. Think beyond the code.

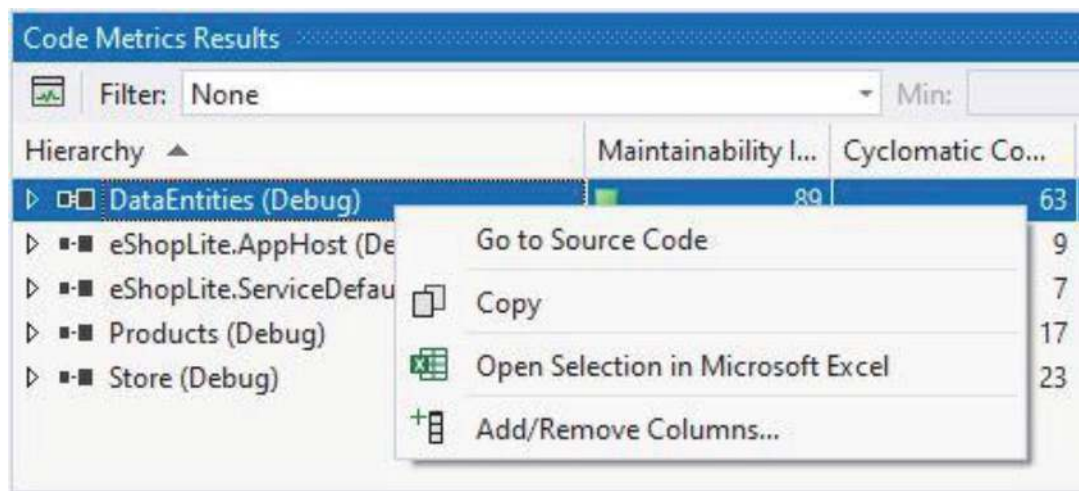
You might hear the term “vibe coding” thrown around by developers. While it sounds cool and “in the zone”, it doesn’t belong in any serious or professional context. In my opinion, it’s a red flag. A former colleague of mine used to say “think more and code less”. You need to be intentional in your design and not just flow with your feelings.

It's akin to mixing late nights, caffeine, and zero accountability into a pull request. Unfortunately, when you work on a team project, you are not solely responsible for the quality of the code. The team as a whole is responsible for maintaining a high standard of code quality.

In this section, we will look at each metric, understand how it gets calculated, and why it matters. Code metrics inform refactoring decisions, continuous integration (CI) quality gates, and code reviews. These metrics act as a guideline to improve code and are not draconian rules to be followed.

### ***Maintainability Index (Higher Is better)***

This index is a score between 0 and 100 that represents how easy the code is to maintain. The closer the code is to 100, the more maintainable it is. Visual Studio even displays this metric with colours. Green is between 20 and 100 (good maintainability), yellow for 10–19 (moderate maintainability), and red 0–9 (low maintainability).



**Figure 2-96** Go to Source Code

Right-clicking on the row of the metric allows you to navigate to the code, as seen in [Figure 2-96](#).

### ***How It Is Calculated***

The maintainability index formula combines several factors to calculate the score. These are:

- The cyclomatic complexity
- Lines of code
- Halstead volume (measure of the code's size based on operators/operands)

Originally, the maintainability Index score could exceed 100 or go negative, but to improve clarity, Microsoft later reset it to a 0–100 scale. Essentially, a large method that has many branches (high complexity) and a lot of code will have a lower Maintainability Index than a small, simple method that does one thing, and one thing only.

The formula used, as per the Microsoft documentation, to calculate the Maintainability Index can be seen in [Listing 2-24](#).

```
MI = MAX(0, (171
    - 5.2 * ln(Halstead Volume)
    - 0.23 * Cyclomatic Complexity
    - 16.2 * ln(Lines of Code)) * 100 / 171)
```



#### **Listing 2-24** The Maintainability Index Calculation

This formula will yield a number capped at 100 maximum and 0 minimum. Therefore, if you see a red flag on a low score, Visual Studio is indicating with a high degree of confidence that there is an issue with your code.

#### ***Why It Matters***

Maintainability Index is a high-level indicator of the health of your code. A low score here indicates that the code is hard to work with, excessively long, complex, or tricky to understand. There is even a Microsoft Analyzer rule (CA1505: Avoid unmaintainable code) that warns you when the Maintainability Index drops below 10.

Low Maintainability Index correlates with code that often harbors bugs or technical debt. You need to keep in mind, however, that the Maintainability Index score isn't a magic bullet. As seen from Listing 2-24, it aggregates other metrics, so it should be interpreted in context. If a function has a low Maintainability Index score due to a necessary, but complex algorithm, you might accept it, provided it is well tested.

#### ***Refactoring and Quality Gates***

When faced with low Maintainability Index scores, teams often decide to refactor. Code reviewers might ask if the code can be split or simplified. On CI/CD pipelines, teams might decide to fail the build on very low Maintainability Index scores. The .NET code analysis rule CS1505 can be enabled to break the build if the score falls below the default threshold of 10. All these actions ensure no "red code" gets merged into main without scrutiny. Again, bear in mind that the Maintainability Index score aggregates other metrics. Therefore, improving the other metric scores will naturally improve the Maintainability Index. This score is, however, an important metric for managers and code reviewers that helps gauge overall codebase maintainability at a glance. For the developers, it encourages a holistic approach to writing cleaner code.

#### ***Cyclomatic Complexity (Lower Is Better)***

Cyclomatic Complexity is a measure of the number of independent paths through a section of code. In simple terms, it counts the decision points, or "forks in the road" in your code's flow. Code that has no `if` statements or loops (straight-line code) has a complexity of 1, i.e., it only has a single path.

For every `if`, `else if`, `for`, `case`, `while` in a switch, or conditional operator (including ternary `?:`), your complexity will increase by a count of 1. In Visual Studio's Code Metrics, this score is calculated for each method and displayed as an integer.

You would obviously want this score to be lower, and a value of 2 or 5 indicates a simple method, while a count of 20 or 30 indicates an overly complex method. This makes the Cyclomatic Complexity invaluable to developers because of the quantitative nature attributed to your code. In short, the more decisions in a method, the more complex it is.

#### ***How It Is Calculated***

One of the least understood metrics, Cyclomatic Complexity, is defined by the graph of a program's control flow. The classic formula Thomas J. McCabe introduced in 1976 is  $M = E - N + 2P$ , where:

- E = number of edges (control flow transitions)
- N = number of nodes (blocks of code)
- P = number of connected components (typically functions or methods)

You can read the paper at <https://ieeexplore.ieee.org/document/1702388> if you're curious. It all boils down to this: start at 1 for a method's basic path and then add 1 for each decision point.

It is important to note that McCabe originally defined cyclomatic complexity using the formula  $v(G) = e - n + p$ . This formula is more commonly seen in a generalized form of  $M = E - N + 2P$ , which assumes each component (function/method) has a one entry and one exit point, hence the +2 instead of +p.

### ***Why It Matters***

Cyclomatic Complexity directly influences the testability and maintainability of your code. The higher the cyclomatic complexity, the more paths you need to test, which in turn raises the possibility of untested bugs. The logic of your code is also much harder to reason about. These are not just inferred from the numbers, it's backed by research. An analysis by NASA found that modules with both high complexity and large size tend to have the lowest reliability.

The SATC has found the most effective evaluation is a combination of size and (Cyclomatic) complexity. The modules with both a high complexity and a large size tend to have the lowest reliability. Modules with low size and high complexity are also a reliability risk because they tend to be very terse code, which is difficult to change or modify - Software Assurance Technology Center (SATC) at NASA.

In other words, long, decision-heavy methods are a prime risk for defects. Interestingly, a short but extremely complex method (lots of decisions in a few lines of code) is also problematic. The rule of thumb is that if you see a method with an unusually high cyclomatic complexity score in relation to its peers, you need to take a closer look.

### ***Refactor and Quality Gates***

Cyclomatic Complexity is one of the easiest metrics to act on. If code reviewers see a method with a high complexity score, they will most likely ask you to simplify that method. From a refactoring perspective, you could extract methods or rethink logic. Some teams include complexity checks into CI pipelines. Visual Studio's built-in analyzer CA1502 (Avoid Excessive Complexity) specifies a default threshold of 25, which is quite high. This is often lowered, but by enabling CA1502, you can fail a build or issue warnings if any method exceeds a set complexity threshold. Doing this creates a quality gate that developers need to pass before merging code into main.

Another approach that some teams take is to use the Cyclomatic Complexity as an indicator of methods that require extra scrutiny. If you have to introduce complex code, then you have to ensure that there are extra tests or documentation to cover those methods.

Some teams might also track the average complexity of the codebase over time. If this score tends to tick up release over release, then this would be a strong indicator of growing technical debt. At the end of the day, keeping the Cyclomatic Complexity low or in check is essential for manageability and testability.

### ***Depth of Inheritance (Lower Is Generally Better)***

As the name suggests, the depth of inheritance counts how deep a class is in the inheritance hierarchy. Since all classes ultimately inherit from `System.Object`, a class with no base class, has a depth of 1. Every additional level of inheritance increases this depth by 1.

If a class called `Parent` inherits from a class called `GrandParent`, and a class called `Child` inherits from the `Parent` class, then the `Child` class has a depth of 3. This means that because



of the chain of inheritance `Child` ► `Parent` ► `GrandParent` ► `System.Object`, the `Parent` class's depth is 2, and the `GrandParent` class's depth is 1.

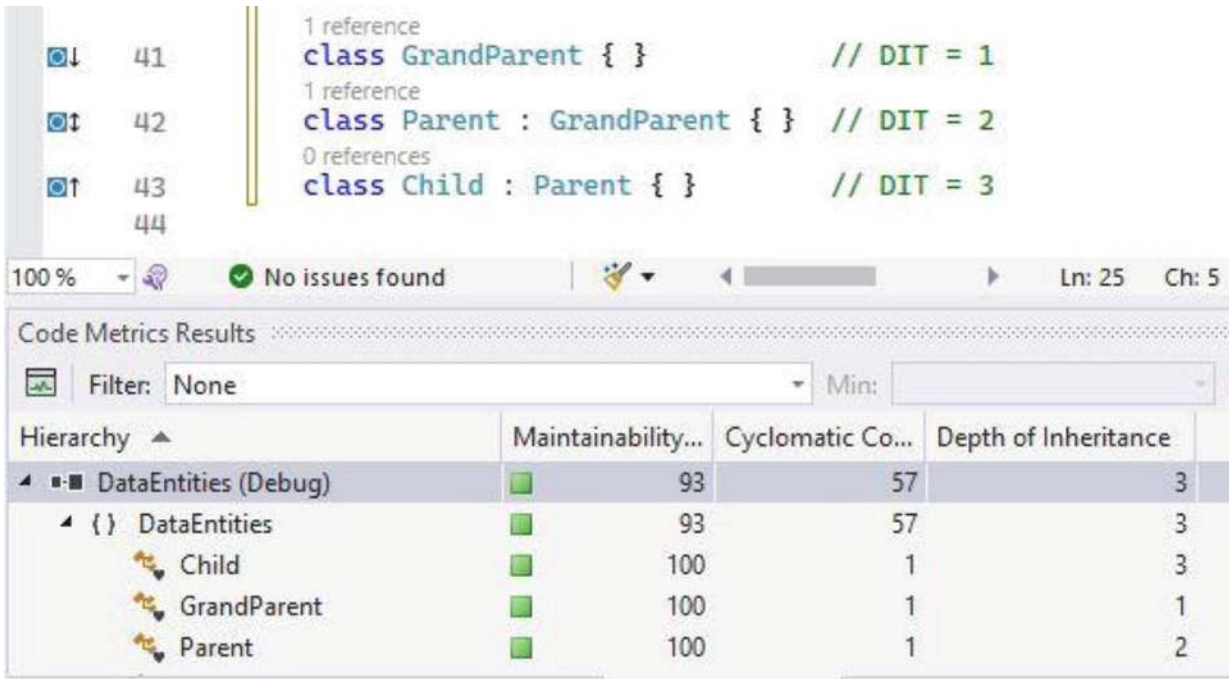


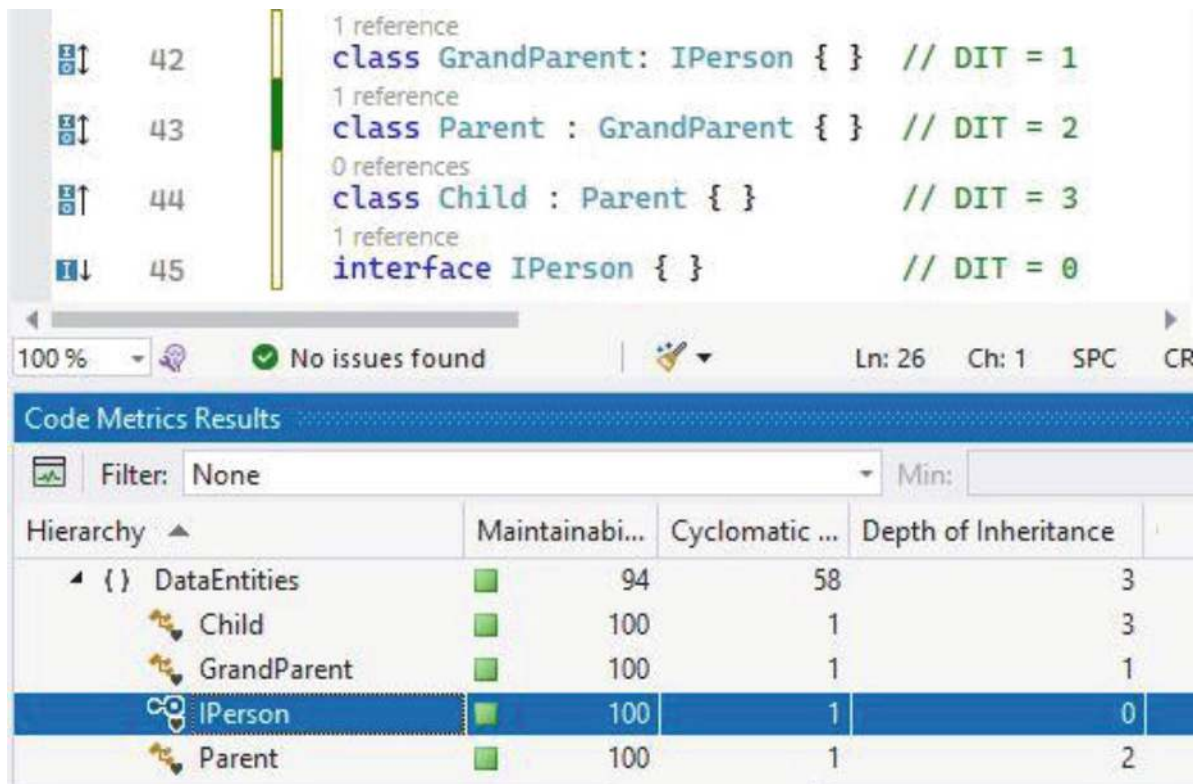
Figure 2-97 Depth of Inheritance Illustrated

You can see these results displayed in Figure 2-97 in the Code metrics results under the Depth of Inheritance column.

**How It Is Calculated**

The Depth of Inheritance is simply a count of parent classes up to the top of the hierarchy. Interfaces are excluded in this count; only base classes are counted. This is because an interface implementation does not add to the inheritance chain because it acts as a contract.

You can see this illustrated in the Code Metric results in Figure 2-98. The `GrandParent` class implements an interface called `IPerson`, and when recalculating the Depth of Inheritance, it doesn't affect the counts at all.



**Figure 2-98** The Impact of an Interface

From this, you can see that interfaces are ignored, while any class you create has a default depth of 1, and any subsequent inherited classes increase the count by 1 each time.

### Why It Matters

Deep inheritance can be construed as a red flag, but it's more like a double-edged sword. On one hand, inheritance is a fundamental aspect of reuse. On the other, a very deep chain of inherited classes (think of 8 or 9 levels deep) often indicates fragile class designs.

This is because a change in a class deep down in the inheritance stack would have a far-reaching impact on all the derived classes. If a change to the `GrandParent` class is made, it might break some or all of the classes that inherit from it.

It is for this reason that a lower Depth of Inheritance score is generally better, because it suggests a simpler class hierarchy. The code analysis rule CA1501 (Avoid excessive inheritance) will fire when the default threshold of 5 is reached. It is also important to note that this rule is not enabled by default in .NET 9. As with other analysis rules, you can modify the threshold value. A rule of thumb is to keep your inheritance level to 3 or perhaps 4 levels. Remember, for every level you add, you are increasing the cognitive load you will experience because you could potentially be opening multiple class files to try to understand what a method actually does due to overrides up the chain.

Favor object composition over class inheritance

On page 20 of chapter 1 in the highly influential 1994 book **Design Patterns: Elements of Reusable Object-Oriented Software**, it states that favouring object composition over class inheritance is the second principle of object-oriented design.

You can grab a copy of the book on Amazon by visiting this link:  
<https://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612>

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, commonly known as the “Gang of Four,” state that composition provides greater flexibility than inheritance by delegating behavior to contained objects rather than hard-coding relationships by using inheritance.

Object composition basically refers to building complex behavior by combining smaller, simpler objects as opposed to inheriting from a base class.

You can see an example of this in Listing 2-25. In this code listing, we have a class called Engine and a class called Car.

```
public class Engine
{
    public void Start() => Console.WriteLine("Engine is started");
}

public class Car
{
    private readonly Engine _engine = new Engine();

    public void Start() => _engine.Start(); // Delegation
}
```

**Listing 2-25** Object Composition

Here the Car class uses an Engine class rather than inheriting from the Engine class. This is composition in action, because Car **has** an Engine, not **is** an Engine.

```
public class Car
{
    private readonly IEngine _engine;

    public Car(IEngine engine)
    {
        _engine = engine;
    }

    public void Start() => _engine.Start();
}
```

**Listing 2-26** Dependency Injection

Interfaces are often used with composition to improve flexibility. We have all seen Dependency Injection in action. In Listing 2-26, the Car class depends on an Interface of IEngine, allowing us to easily swap out the behavior of the engine (e.g., DieselEngine vs ElectricEngine vs PetrolEngine) as long as they implement IEngine. It also reduces tight coupling and allows us to mock IEngine for use in unit tests.

### ***Refactoring and Quality Gates***

You might be inviting a discussion during a code review when you introduce a class that extends an already deep hierarchy. In fact, CI pipelines can enable the inheritance depth rule CA1501 to avoid this issue entirely.

A practical approach in large systems is to limit inheritance depth. Controllers, for example, should not inherit more than 1 level. Domain entities, on the other hand, shouldn't have a Depth of Inheritance count of more than 3. These limits are, however, up to developers to define based on what they are comfortable with.

In summary, this metric ensures that you do not create a class hierarchy that is overly complex. Try to strike a balance by using enough inheritance to leverage polymorphism, but not too much that it becomes brittle.

### ***Class Coupling (Lower Is Better)***

Not to be confused with Depth of Inheritance, Class Coupling measures how many **unique** types a given type or method is connected to. To put it simply, it is a count of the other classes that your class or method uses directly.

This includes parameters, return types, local variables, fields defined on external types, base classes or Interfaces, types used in method calls, generic type arguments, and even attribute types.

Because the emphasis is on unique types, calling one class ten times still counts as one. Using ten different classes once each, however, counts as ten. The principle of coupling in software engineering means that tightly coupled code has many interdependencies, whereas loosely coupled code has fewer.

The code metric provided by Visual Studio quantifies class coupling at a code level. When you see a low Class Coupling count, it means that your class is more focused and isolated. A high number indicates that your code is reaching out into many corners of your codebase, making it harder to change or reuse.

### ***How It Is Calculated***

Visual Studio's code metrics inspects a class (or method) and tallies all the distinct types it touches. The easiest way to understand this is to look at some code directly.

The code in Listing [2-27](#) is a good example of a tightly coupled class with a high Class Coupling count.

```
public class Car
{
    // IEngine = +1
    private readonly IEngine _engine;
    // ILogger<Car> = +1 and Car (used in generic) = +1
    private readonly ILogger<Car> _logger;
    // EngineDiagnostics = +1
    private readonly EngineDiagnostics _diagnostics;
    // List<Tire> = +1 and Tire = +1
    private readonly List<Tire> _tires;
    // NavigationSystem = +1
    private readonly NavigationSystem _navigation;
    // AudioSystem = +1
    private readonly AudioSystem _audio;
    // Driver = +1
    private readonly Driver _driver;
    // List<Passenger> = +1 and Passenger = +1
```

```

private readonly List<Passenger> _passengers;
// Garage = +1
private readonly Garage _garage;

public Car(
    IEngine engine,
    ILogger<Car> logger,
    EngineDiagnostics diagnostics,
    List<Tire> tires,
    NavigationSystem navigation,
    AudioSystem audio,
    Driver driver,
    List<Passenger> passengers,
    Garage garage)
{
    _engine = engine;
    _logger = logger;
    _diagnostics = diagnostics;
    _tires = tires;
    _navigation = navigation;
    _audio = audio;
    _driver = driver;
    _passengers = passengers;
    _garage = garage;
}

public void Start() {
    _engine.Start();
    _logger.LogInformation("Car started");
    _diagnostics.ToString();
    _navigation.ToString();
    _audio.ToString();
    _driver.ToString();
    foreach (var p in _passengers) p.ToString();
}

public bool Stop()
{
    _logger.LogInformation("Car stopped");
    return false;
}
}

```

**Listing 2-27** Class with High Class Coupling

Visual Studio reports the Class Coupling count here as 12. The code in Listing [2-27](#) highlights how Class Coupling is calculated based on the number of unique external types that the `Car` class directly depends on. Visual Studio not only considers concrete field and parameter types but also constructed generic types.

This means that `List<Tire>` and `ILogger<Car>` as well as their type arguments such as `Tire`, `Passenger`, and even `Car` itself when used as a type parameter, are used in calculating

this metric.

It is important to note that even indirect references such as a class referencing itself via a generic type can have an impact on the coupling count.

### ***Why It Matters***

The Class Coupling count is a valuable metric that lets you evaluate how interconnected and brittle a class is. Higher coupling signals the need for refactoring, especially if the class becomes difficult to change or test. This allows developers to make deliberate architectural choices by being aware of how dependencies accumulate.

High Class Coupling also indicates potential violations of the Single Responsibility Principle.

Implementing SOLID principles will result in favorable code metric results throughout your codebase.

A class might be taking on too many tasks. The ideal is that a class should be focused on a single purpose and have minimal coupling by only depending on what it truly needs. The analyzer rule CA1506 (Avoid excessive class coupling) will trigger a warning when the threshold is reached.

Again, this is not enabled by default in .NET 9 and the default threshold is 95 for types and 40 for other symbols, which is rather conservative in my opinion. This means that when enabled, the rule will not trigger unless coupling is extreme.

In practice, developers might lower the threshold to a more palatable threshold.

### ***Refactoring and Quality Gates***

In code reviews or PRs, classes with high Class Coupling counts will most likely warrant refactoring. The fix would be to apply separation of concerns, in other words, split the class so that each part deals with fewer collaborators. This metric also provides objectivity, because a reviewer can use the Class Coupling count to request a code refactor. On CI quality gates, CA1506 can be enabled to provide warnings when a class hits the threshold set by the developers. Ultimately, understanding and tracking this metric enables more targeted refactoring, resulting in code that is easier to maintain, more stable, and significantly easier to test.

### ***Lines of Code***

Lines of code refer to the size of your codebase. But there is an important distinction here. Visual Studio actually provides two related metrics, **Lines of Source Code** and **Lines of Executable Code**. Lines of Source Code include blank lines and code comments, which means that this metric counts the lines in the source file. Lines of Executable Code is an approximation of how many actual execution statements it finds.

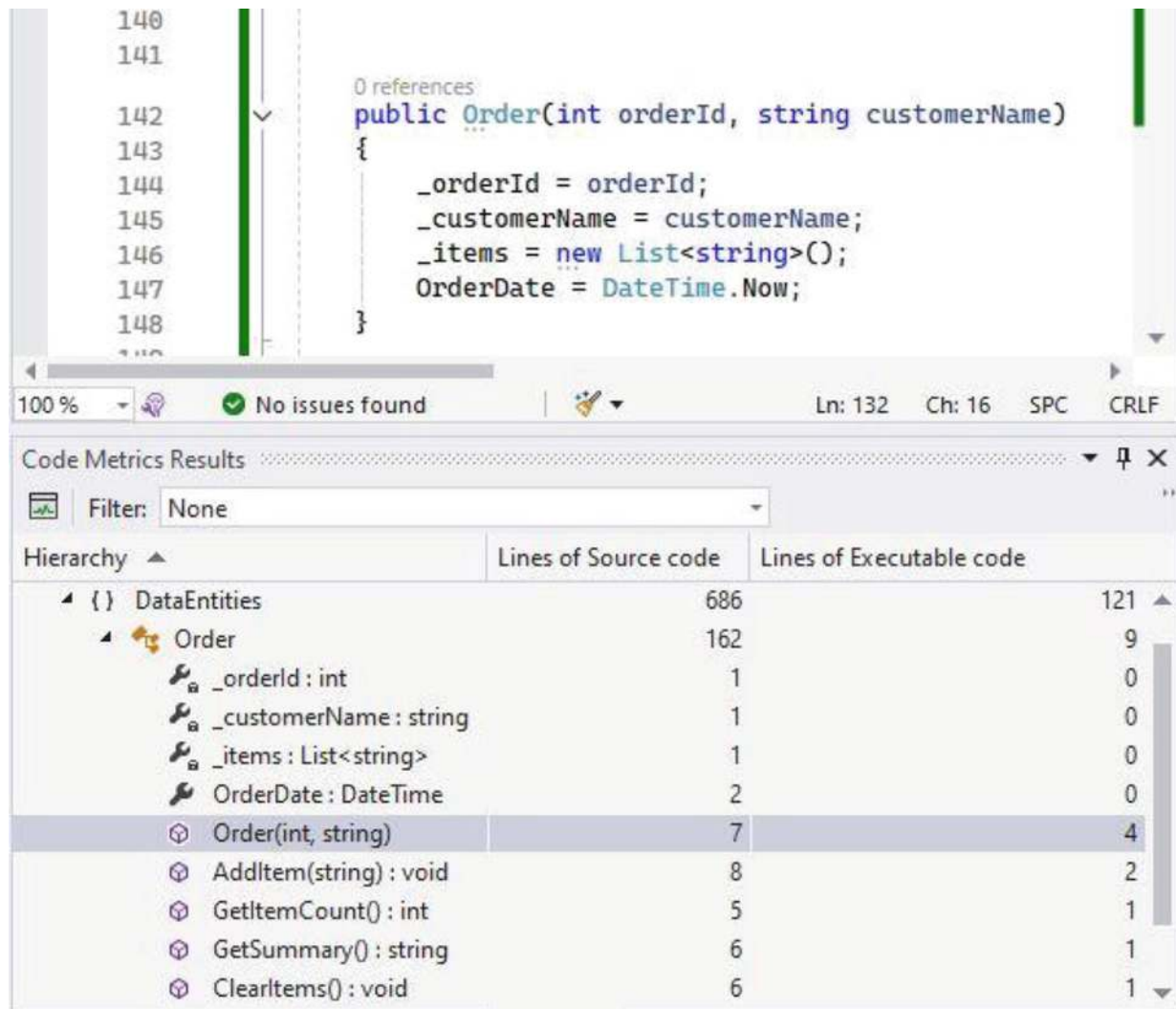
### ***How It Is Calculated***

The calculation is not complex, but as before, it's easier to illustrate using an example. If a method spans from line 142 to line 148 in the source without any blank lines or code comments, Visual Studio will count 7 lines.

When it comes to Lines of Executable Code, however, Visual Studio will only count the number of operations in the compiled code. This roughly corresponds to IL instructions. A curly brace or a code comment doesn't produce an IL instruction, so it's not included in the executable lines count.

Consider the images in [Figure 2-99](#) and [Figure 2-100](#).





**Figure 2-99** The Order Constructor Line Counts

As seen in the Code Metric Results for Lines of Source Code, the count is 7. These are:

1. `public Order(int orderId, string customerName)`
2. `{`
3. `_orderId = orderId;`
4. `_customerName = customerName;`
5. `_items = new List<string>();`
6. `OrderDate = DateTime.Now;`
7. `}`

For the Lines of Executable Code the picture looks different. The only lines that count towards this metric are:

- 1. `_orderId = orderId;`
- 2. `_customerName = customerName;`
- 3. `_items = new List<string>();`
- 4. `OrderDate = DateTime.Now;`

If we add in a comment and a blank line, the picture looks different as seen in Figure 2-100.

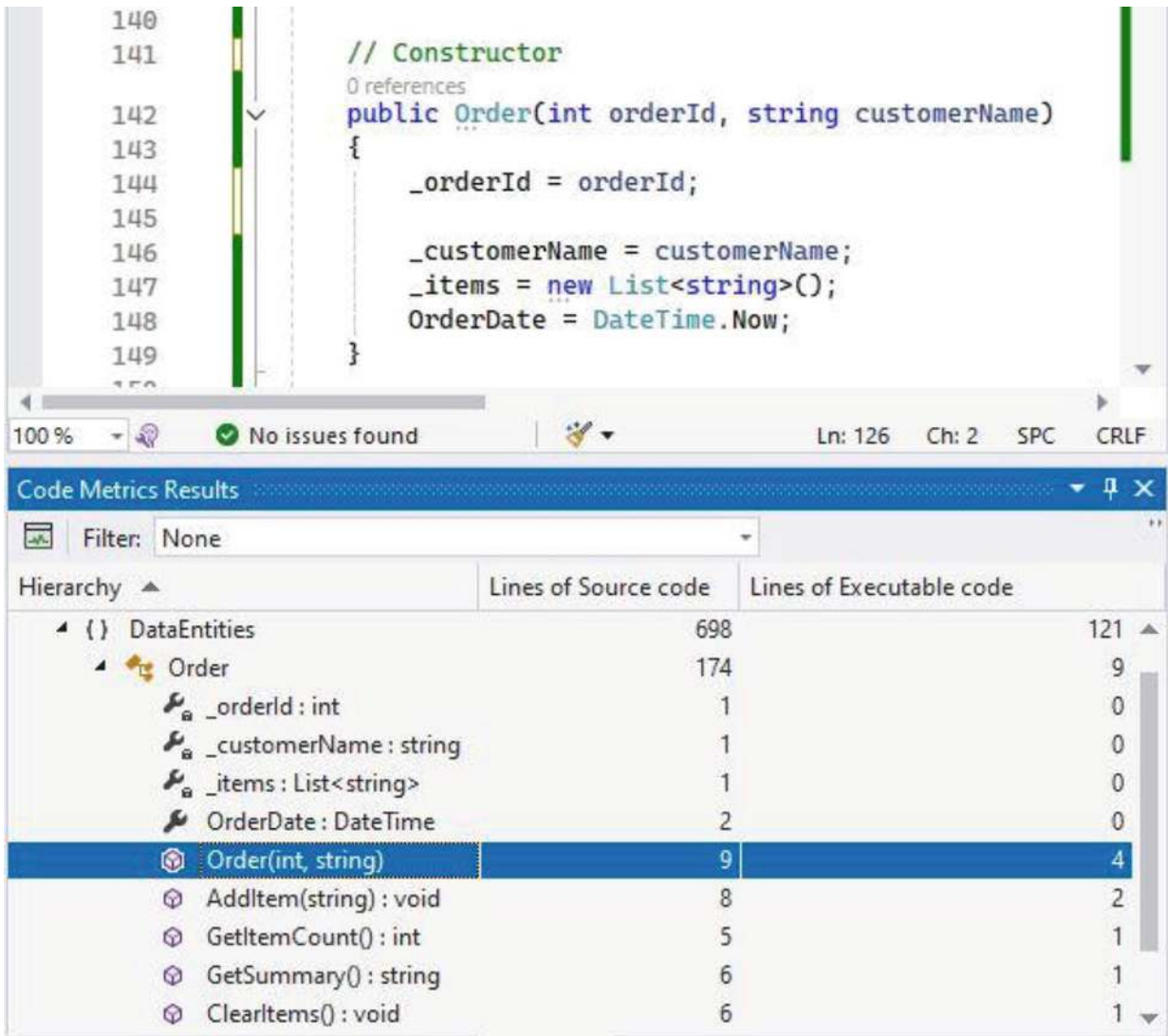


Figure 2-100 Updated Order Constructor Line Counts

Adding a single comment and a blank line increases the Lines of Source Code count by 2 while the Lines of Executable Code stay the same.

### **Why It Matters**

The simplest of all the Code Metrics provided by Visual Studio, it's deceptively meaningful. A method with 200 lines of code is usually doing more than it should, and apart from it being harder to read and follow along in logic, it's harder to maintain. As before, this could be an indicator of code that violates the Single Responsibility Principle.

Having aid that Lines of Code by itself does not tell the whole story. You need to take all the other metrics into consideration, too. For example, a method that has 300 lines and has a high Cyclomatic Complexity and Class Coupling score needs to be refactored.

Some sources online suggest that methods should aim for line counts under 40, but again, there is no hard and fast rule. My view is that if a method is long enough that you can't see it all at once without scrolling, it's probably a sign that it should be broken into smaller, more focused parts.

### **Refactoring and Quality Gates**

I would treat Lines of Code as more of an advisory metric than a strict quality gate. Some teams might have style guidelines that limit lines of code to a certain number, but one might not want to fail a CI/CD build due to the Lines of Code count.

Some developers might, however, track Lines or Code as new features are added to an existing class. Growing Lines of Code counts could hint at "God objects" being created.

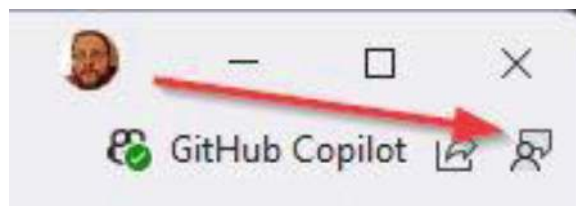
A God object is an anti-pattern where a single class knows too much, does too much, and interacts with too many other parts of the system

In summary, Lines of Code give developers a quantitative handle on code size. These reinforce an important aspect of quality, which is keeping things small and focused. Mega-methods or bloated classes are identified early on, allowing a refactor before this code becomes unmanageable.

### **Send Feedback**

The Visual Studio team takes feedback seriously. So much so that it drives much of what they do to improve Visual Studio.

If you are experiencing a problem in Visual Studio, click the feedback button, as shown in Figure [2-101](#).



**Figure 2-101** Send Feedback button

You can report a problem or suggest a feature right from inside Visual Studio 2022.

As developers, we should take the time to report issues we encounter. It can be anything from crashes to slow performance or something else unexpected.

---

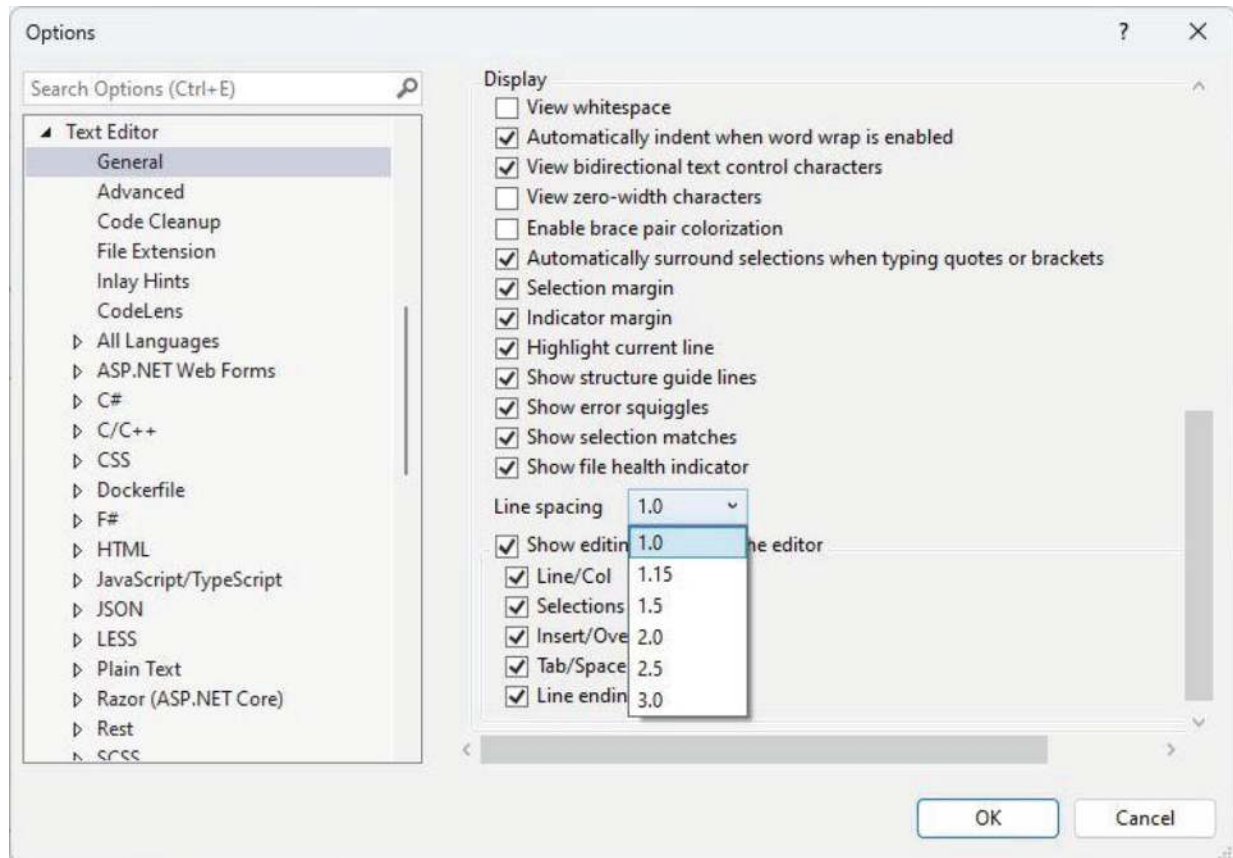
## **Personalizing Visual Studio**

Visual Studio is one of the most popular IDEs available and does a fantastic job at increasing your productivity. It also allows developers to personalize their experience based on their

individual workflow needs. Let's look at some of these improvements that help you personalize Visual Studio.

## Adjust Line Spacing

In Visual Studio 2022, developers can specify the line spacing for the text editor. To increase the line spacing, on the Tools menu, click Options and select Text Editor ► General and look under the Display group as seen in Figure 2-102.



**Figure 2-102** Adjust Line Spacing

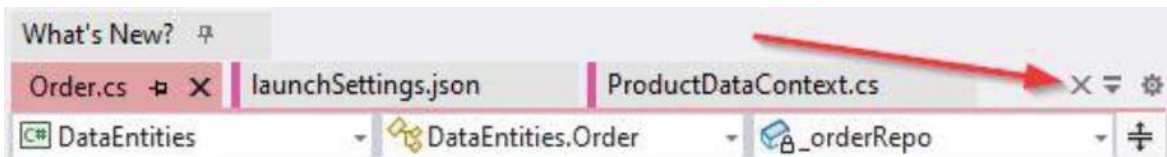
Admittedly, the options might not suit everyone, and I hope that Visual Studio allows custom values to be entered here in a future release. Nevertheless, being able to space the lines of code vertically is a quality-of-life improvement for many developers that spend most of their working day in front of Visual Studio.

## Document Management Customizations

There are several small but welcome changes to how developers manage documents. The Document Tabs group under Tools ► Options ► Environment ► Tabs and Windows has had several new options added compared to previous versions of Visual Studio. Let's have a look at some of these.

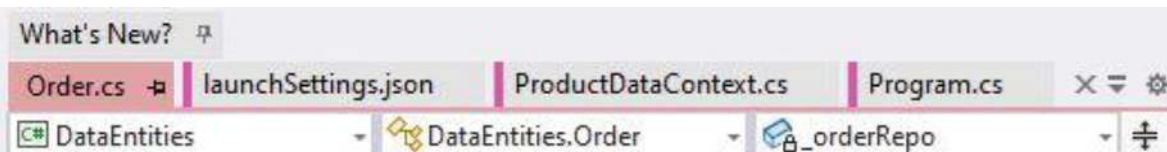
### *The Document Close Button*

Developers now have options when deciding how to show the close button for documents. You now have the option to add the close button to the document well, as seen in Figure 2-103.



**Figure 2-103** Show Close Button in Document Well

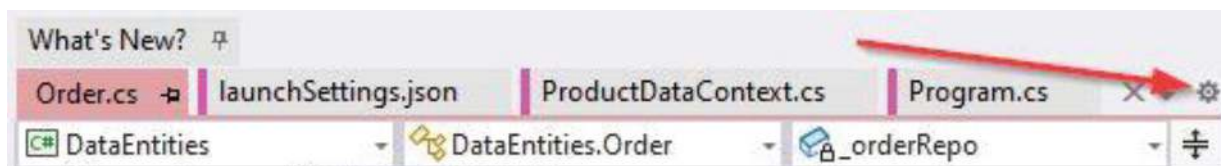
You can also remove the close button from the individual document tabs, as seen in Figure 2-104, leaving you with a single close button. Compare the Order.cs tab in Figure 2-103 with the Order.cs tab in Figure 2-104.



**Figure 2-104** Remove Close Button on Tabs

You can also opt to remove the close button from the document well as well as from the document tab. You can then right-click the document to close it.

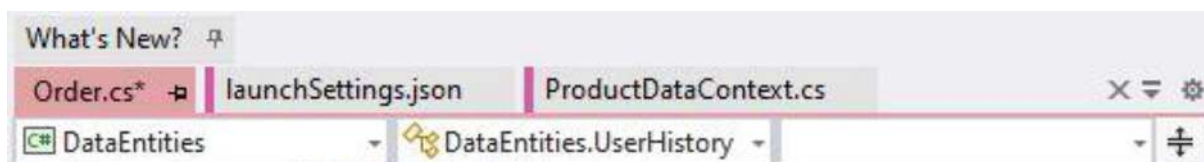
To quickly access these and other options for document management, you can click on the gear icon as shown in Figure 2-105.



**Figure 2-105** Access Options Via the Gear Icon

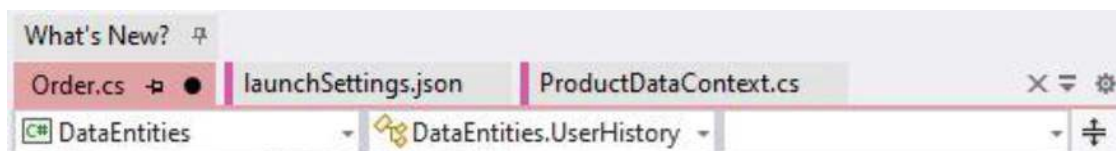
## Modify the Dirty Indicator

Developers can now choose how they would like to display dirty tabs. The default is an asterisk to the right of the file name, as seen in Figure 2-106 on the Order.cs tab.



**Figure 2-106** The Default Dirty Indicator

You can also change this indicator to be a dot, as shown in Figure 2-107.



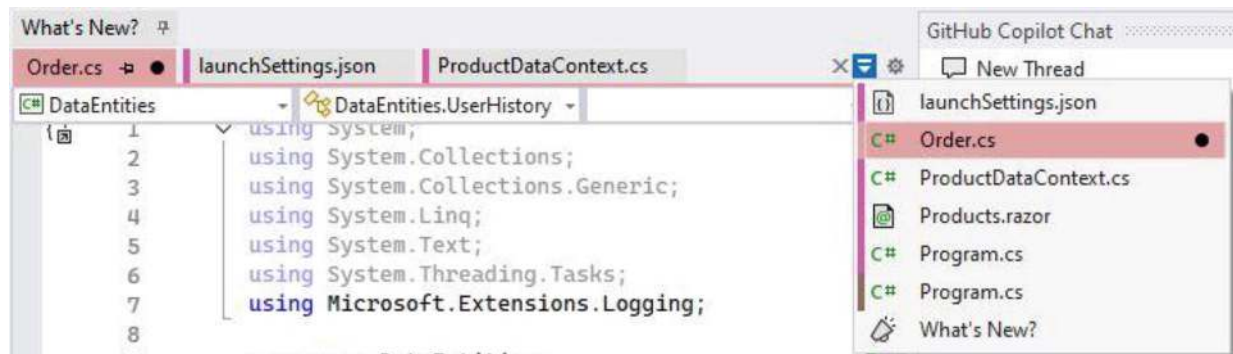
**Figure 2-107** The Dot Dirty Indicator



The dot does make the tab slightly bigger, but it is a more explicit indicator that the document has unsaved changes. What is nice, however, is that if you have set the close button to be visible on document tabs, the dot will change into a close button when you hover over it.

### **Show Invisible Tabs in Italics in the Tab Drop-Down**

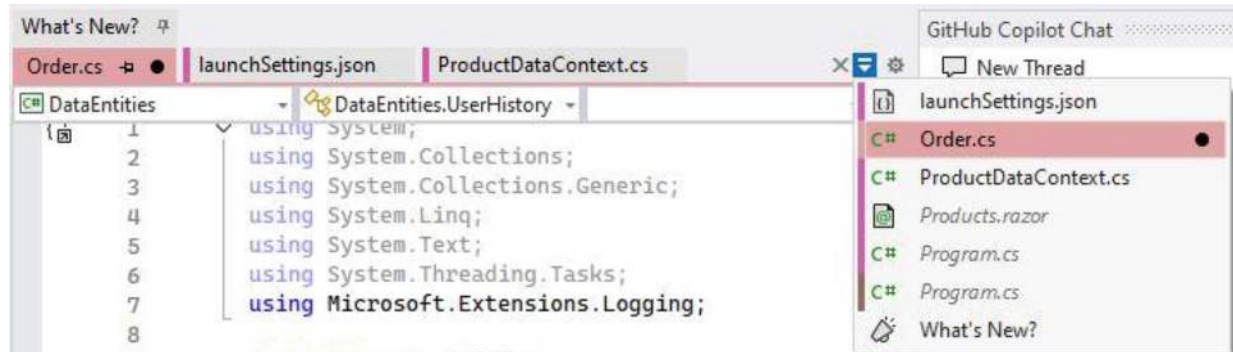
The ability to display hidden tabs in italics is something you might not know that you want until you see it in action. The tab drop-down displays all your open tabs, as seen in Figure 2-108.



**Figure 2-108** Invisible Tabs Not in Italics

The trick is knowing which documents are currently visible and which are not.

With this option set to show hidden documents in italics, the hidden tabs are immediately discernable, as seen in Figure 2-109. This is especially helpful when looking at that Program.cs file.

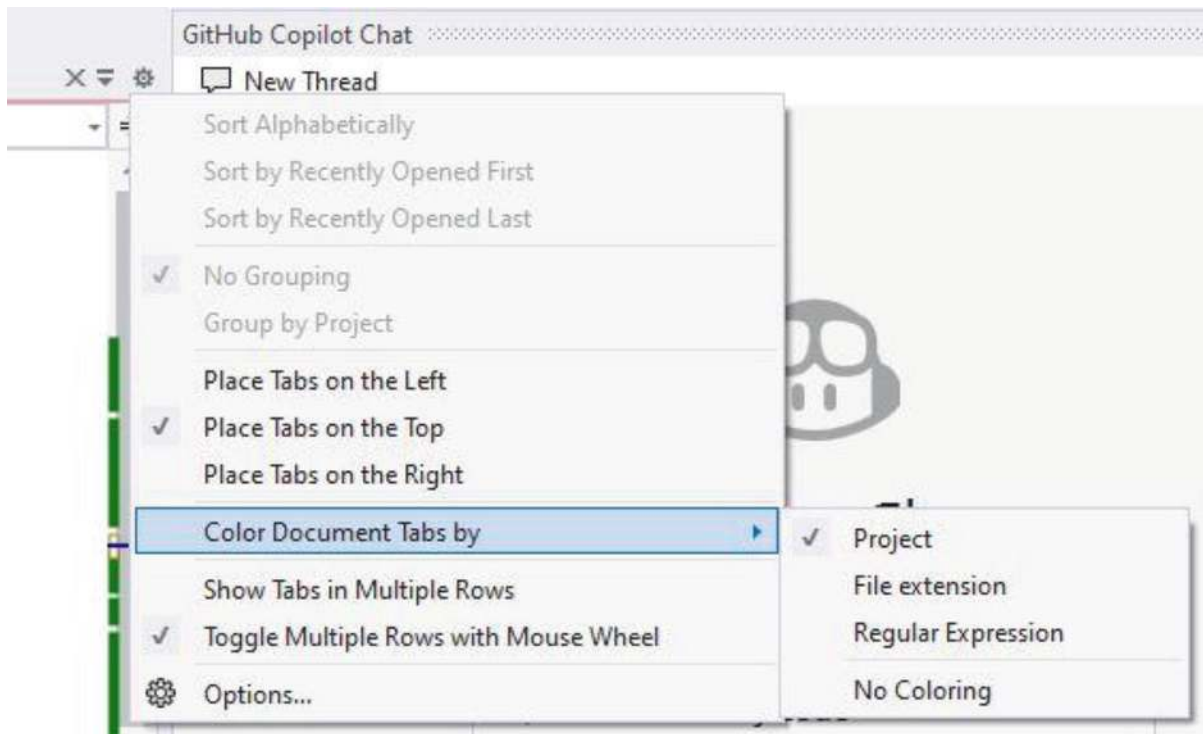


**Figure 2-109** Invisible Tabs Displayed in Italics

### **Colorize Document Tabs**

I'm sure the images displayed here might not be in color if you read the print book. Showing the color changes on tabs will, therefore, not make sense. Instead, I will show you where you can change this option.

As seen in Figure 2-110, you can colorize document tabs by Project, File Extension, Regular Expression, or choose no coloring, which is the default. Playing around with this option allows you to find a setting that is convenient for you.



**Figure 2-110** Colorizing Document Tabs

If you don't like the default color for the tab, right-click the tab and select the Set Tab Color option to choose a color more to your liking.

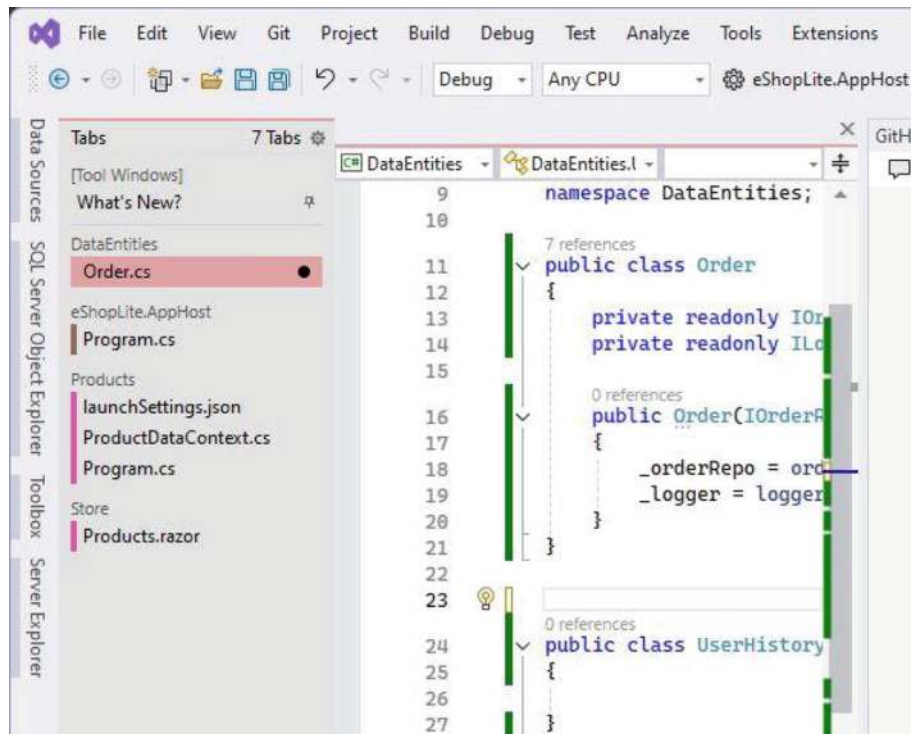
Being able to colorize document tabs helps developers cut through the noise when working in multiple tabs, especially on large projects.

### ***Tab Placement***

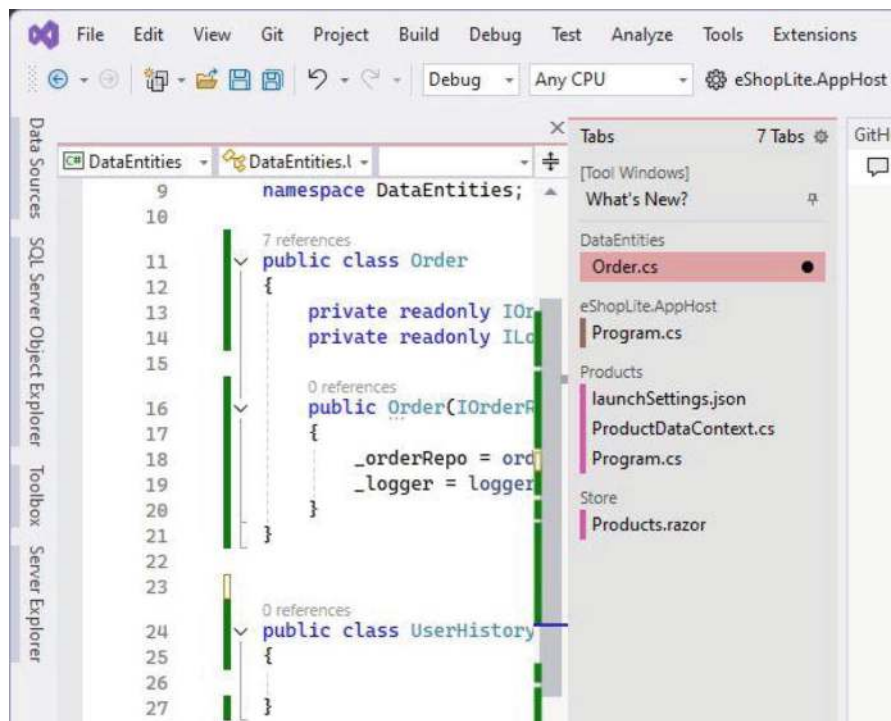
The ability to choose the tab placement is something many of you might welcome.

As seen in Figure [2-111](#), the tabs can be displayed on the left or, as in Figure [2-112](#), on the right.



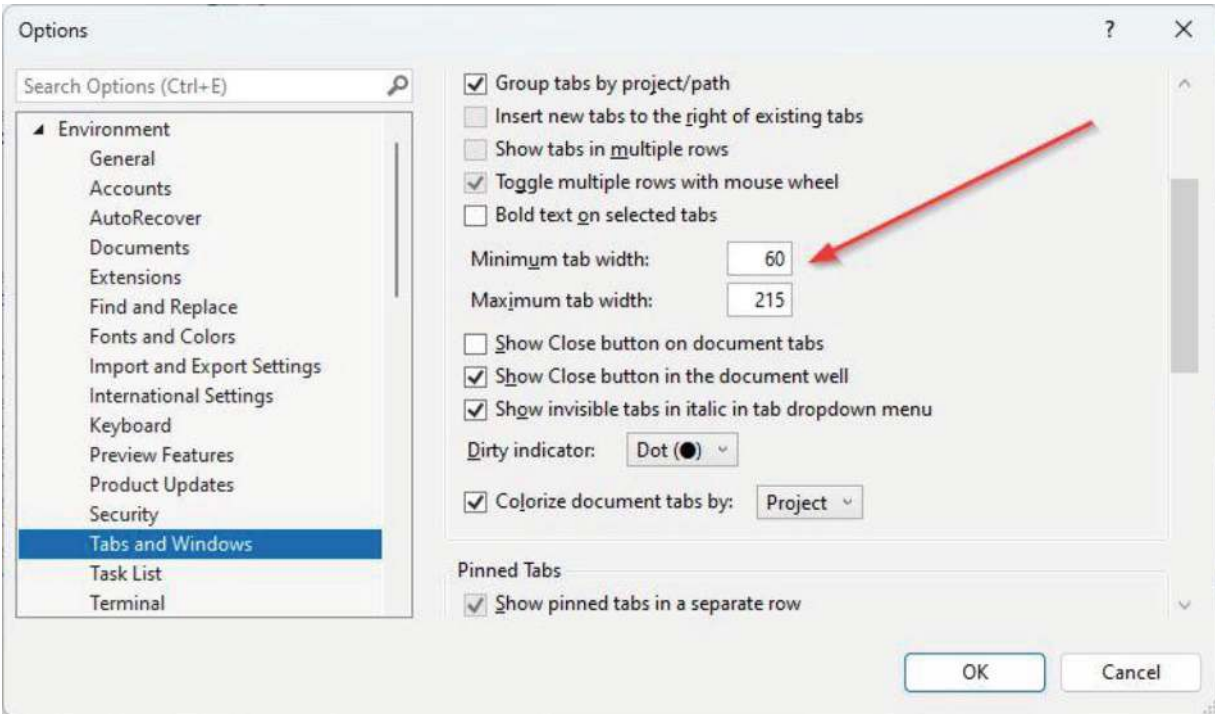


**Figure 2-111** Document Tabs on the Left



**Figure 2-112** Document Tabs on the Right

The left and right placement of tabs also highlights the color options better. More flexibility is given to developers by allowing them to choose the minimum and maximum tab width, as seen in [Figure 2-113](#).



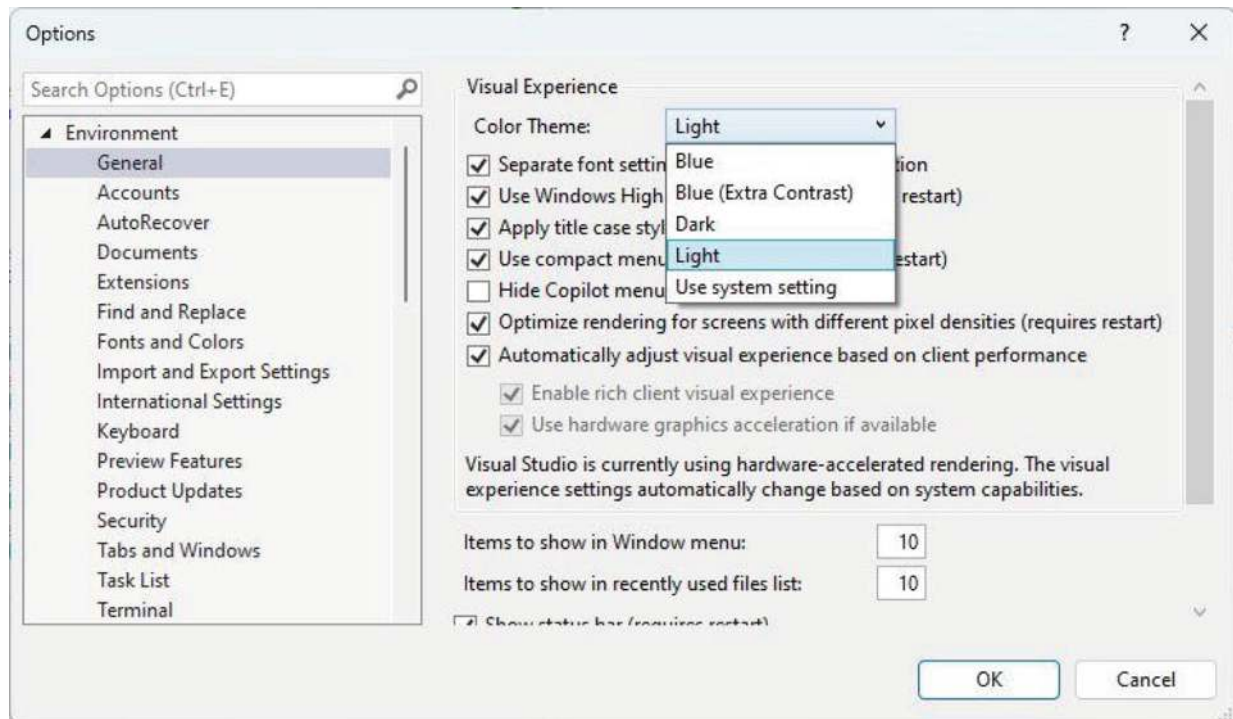
**Figure 2-113** Additional Document Tab Options

## Visual Studio Themes

Visual Studio provides a flexible theming system that allows developers to personalize the IDE's look and feel. The change can be subtle or dramatic, changing editor backgrounds and syntax colors, tool windows, and dialogs. Let's have a look at the built-in themes available in Visual Studio 2022 first, and how to switch between them.

### ***Built-in Themes***

Visual Studio comes with a few built-in color themes that cater to different preferences. As shown in [Figure 2-114](#), you can access several theme options by going to Tools ► Options ► Environment ► General and selecting the Color Theme dropdown.



**Figure 2-114** Theme Selector

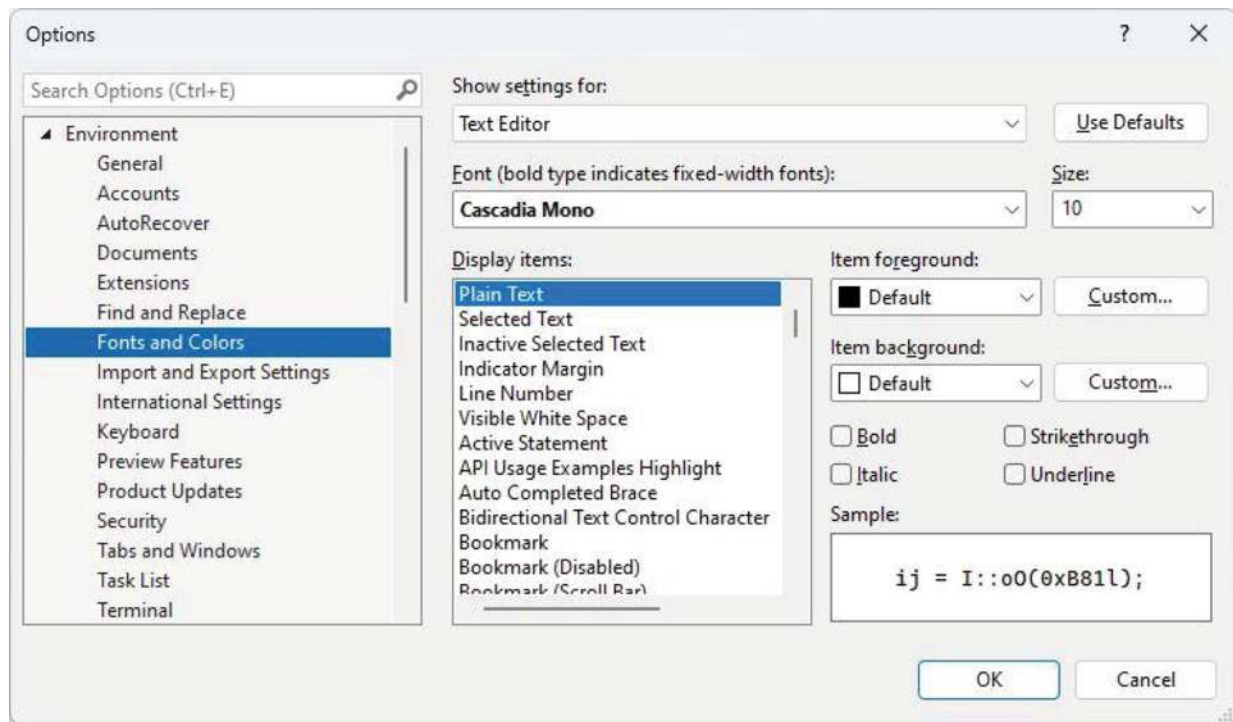
Out of the box, Visual Studio uses the familiar Blue theme, but you can change this to your liking. Here, you can choose a sleek Dark theme (popular with many developers), a bright Light theme, a Blue (Extra Contrast) theme, or an option called Use system setting, which matches Visual Studio's theme to your Windows theme preference.

Why do developers prefer dark themes? Because light attracts bugs.

After changing your theme, Visual Studio will remember your selection and use that theme the next time you open your IDE. It will also synchronize them across all Visual Studio environments, depending on your account setup. With Visual Studio version 17.12, any custom font settings you've made will be preserved when switching themes, which is a welcome quality of life enhancement.

### ***Customizing Colors and Fonts***

The built-in themes provide a complete color scheme, but you might want to make some fine-tuned changes to aspects of the editor or UI to better suit your preferences. To accommodate this, Visual Studio allows extensive customization of fonts and colors by going to Tools ► Options ► Environment ► Fonts and Colors, as shown in Figure [2-115](#).

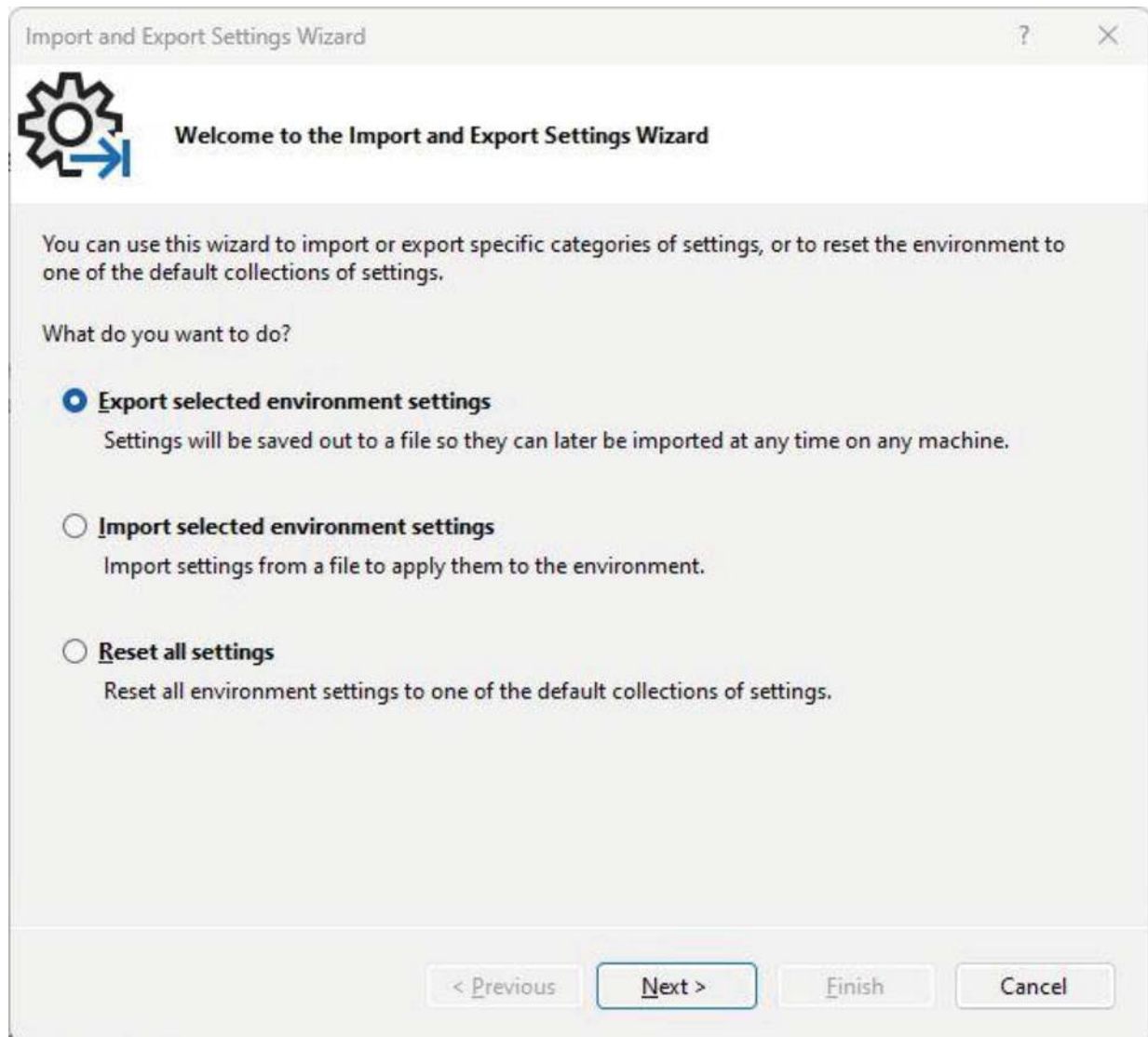


**Figure 2-115** Customizing Fonts and Colors

Here you can tweak the appearance of nearly every UI element or code element to your liking. At the top of the Options dialog for Fonts and Colors, you will see a dropdown named *Show settings for*. This allows you to choose the scope of what you are customizing.

As you can imagine, tweaking settings here might lead you down a rabbit hole and before long, you might want to undo everything you've changed. This can be accomplished by clicking on the Use Defaults button.

It is also worth mentioning that should you wish to export your settings, you can do so by clicking on Tools ► Import and Export Settings from the Visual Studio menu, as illustrated in [Figure 2-116](#).

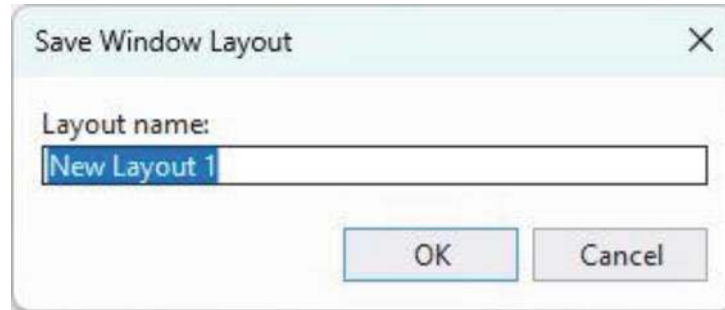


**Figure 2-116** Export Settings Wizard

This allows you to save your settings to a file for import elsewhere. While this might not be necessary in most cases, it's still a good tip for exporting and backing up a finely tuned environment.

### ***Customizing Windows Layouts***

While the theme mainly affects colors and styling, the overall feel of your IDE will depend on how your windows are laid out. Visual Studio allows developers to customize and save window layouts to suit different workflows. You might like a minimal layout when writing code, consisting of only the editor and the Solution Explorer. When debugging, however, you might want to see additional windows such as the Watch window, Autos, etc.



**Figure 2-117** Save Window Layout

You can rearrange window layouts by dragging that window around and docking it to a different side of the IDE (or you might want it undocked and floating). You can then save your layout by clicking on the menu item **Window ► Save Window Layout**.

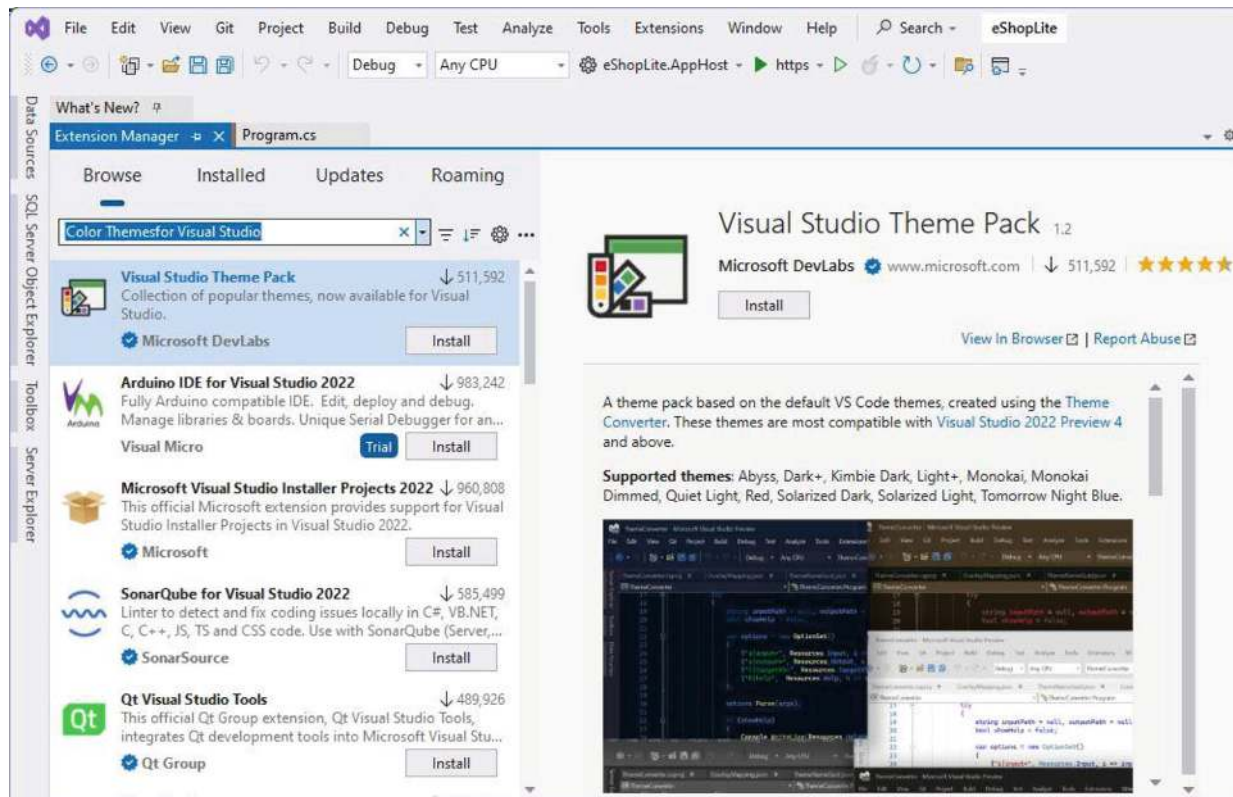
As seen in Figure 2-117, you can give your layout a name and save it. You can save several layouts and then switch between them by going to **Window ► Apply Window Layout**. You can also manage your saved layouts from the **Window** menu, as well as reset your window layout.

Window layouts are beyond the scope of Visual Studio themes, but it remains a very powerful (and underutilized) part of personalizing Visual Studio.

### ***Extensions and Additional Themes***

Built-in themes are only the beginning. Visual Studio allows you to install themes and theme packs from the Extension Manager, as seen in Figure 2-118. The Visual Studio Theme Pack by Microsoft DevLabs, for example, is a popular extension providing several themes that would suit even the most particular theme connoisseur. You can access the Extension Manager from the **Extensions** menu and then click on **Manage Extensions**.





**Figure 2-118** Visual Studio Theme Pack

For those of you that like to have more control over your Visual Studio theme, or if you need to align Visual Studio with specific company branding, you can do this by using the Visual Studio Color Theme Designer 2022. You can download it from the following URL

<https://marketplace.visualstudio.com/items?itemName=idex.colorthemedesigner2022>

This allows you to design your own theme with a live preview until you have a theme that you are happy with. The final product will be a Visual Studio extension that puts your theme alongside the default themes under Tools ► Options.

Microsoft has also teamed up with community theme authors to test a tool that converts VS Code themes to be compatible with Visual Studio 2022. You can access the Theme Converter from the following GitHub repo:

<https://github.com/microsoft/theme-converter-for-vs>.

To see which themes are available, follow this link and style Visual Studio your way:

<https://devblogs.microsoft.com/visualstudio/custom-themes/>.

## The New .SLNX Solution Format

For years, developers have relied on the traditional .sln solution file to organize projects. For those of you that have ever had to merge a solution file, know the frustration of sifting through its cryptic GUIDs. Recently, though, the Visual Studio solution team began previewing an XML-based solution file format called SLNX. As of .NET SDK 9.0.200, the `dotnet` CLI supports building and interacting with these files just like it does with traditional solution files.

### What Is an .SLNX File

In simple terms, the SLNX file is a simplified solution file format using XML instead of the old plain-text format. It serves the same purpose as the SLN file, which is defining which projects are part of your solution and how the solution is configured. Using .slnx files, however, streamlines the contents and structure, making solution files more human-readable and easier to merge under source control. The key differences between .sln and .slnx files are as follows:

### ***Format and Readability***

Looking at an .sln file, you will notice that it is a plaintext file filled with GUIDs, repeated configuration mappings, and nested sections. Compare that to a .slnx file, you will see that it is XML-based, which is a widely understood format. The structure, therefore, is much more readable. This allows developers to easily parse and even hand-edit solution files if required.

### ***Content Simplicity***

The traditional .sln file contained a lot of redundant information. Duplicate project GUIDs and repeated config mappings, with boilerplate that grows as your solution has new projects added to it. The .slnx file does away with all that. No more scrolling past lines and lines of GUIDs to see which projects are in your solution. This minimalistic approach to solution files means that even large solutions result in relatively small, clean solution files.

### ***Comparing SLN to SLNX***

Compare the .sln solution file in Listing [2-28](#) with the much simpler solution file in Listing [2-29](#).

```
Microsoft Visual Studio Solution File, Format Version 12.00
# Visual Studio Version 17
VisualStudioVersion = 17.14.36203.30 d17.14
MinimumVisualStudioVersion = 10.0.40219.1
Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") =
    "HelloAspire.AppHost",
    "HelloAspire.AppHost\HelloAspire.AppHost.csproj", "{A2009DE2-2EE5-
4C9D-9777-1BA5B818F39A}"
EndProject
Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") =
    "HelloAspire.ServiceDefaults",
    "HelloAspire.ServiceDefaults\HelloAspire.ServiceDefaults.csproj", "{
290FEB0D-8BB8-B786-4336-448974D25042}"
EndProject
Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") =
    "HelloAspire.ApiService",
    "HelloAspire.ApiService\HelloAspire.ApiService.csproj", "{779455C5-
65EE-8270-AB64-E9D4CEE5DCC8}"
EndProject
Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") =
    "HelloAspire.Web", "HelloAspire.Web\HelloAspire.Web.csproj", "{
891CCE41-52AC-0DB9-AD31-E17D318194CC}"
EndProject
Global
    GlobalSection(SolutionConfigurationPlatforms) = preSolution
        Debug|Any CPU = Debug|Any CPU
        Release|Any CPU = Release|Any CPU
    EndGlobalSection
```

```

        GlobalSection(ProjectConfigurationPlatforms) = postSolution
            {A2009DE2-2EE5-4C9D-9777-1BA5B818F39A}.Debug|Any
CPU.ActiveCfg = Debug|Any CPU
            {A2009DE2-2EE5-4C9D-9777-1BA5B818F39A}.Debug|Any CPU.Build.0
= Debug|Any CPU
            {A2009DE2-2EE5-4C9D-9777-1BA5B818F39A}.Release|Any

        EndGlobalSection
        GlobalSection(SolutionProperties) = preSolution
            HideSolutionNode = FALSE
        EndGlobalSection
        GlobalSection(ExtensibilityGlobals) = postSolution
            SolutionGuid = {EA27AE4D-36D4-4D86-B024-D0CAD2B159FD}
        EndGlobalSection
    EndGlobal

```

**Listing 2-28** An Example of an SLN File

I actually deleted a whole lot of lines from the .sln file in Listing [2-28](#). Otherwise it would have spanned a whole additional page in this book.

```

<Solution>
  <Project
Path="HelloAspire.ApiService/HelloAspire.ApiService.csproj" />
    <Project Path="HelloAspire.AppHost/HelloAspire.AppHost.csproj" />
    <Project
Path="HelloAspire.ServiceDefaults/HelloAspire.ServiceDefaults.csproj"
/>
    <Project Path="HelloAspire.Web/HelloAspire.Web.csproj" />
</Solution>

```

**Listing 2-29** The Same Solution Saved as SLNX

When I save the solution as an .slnx file, the difference is dramatic as seen in Listing [2-29](#).

## Why Introduce SLNX Files

Now that we have an idea what an .slnx file looks like, let's discuss why this new solution file format was introduced. The change is due to longstanding pain points when it comes to managing solution files.

### Frequent Merge Conflicts

When working in a team, .sln files were notorious sources for merge conflicts. If two developers on the team so much as touch the solution by adding a new project or changing some configuration, the .sln file would conflict in complex ways.

This often led to corrupted solution files due to a bad merge or time wasted untangling the mess. As evidenced in Listing [2-29](#), the .slnx file tries to (successfully) eliminate that headache by simplifying the structure to a straightforward list of projects. Adding a project now means the addition of a new line of XML, something far less likely to cause conflict headaches when merging.

### Source Control Friendliness

Apart from causing fewer conflicts, reviewing a cleaner diff during code reviews or during a merge is a boon for code reviewers. The .slnx file is easier to read and understand exactly what was changed in the solution at a glance.

This makes for a more pleasant experience when reviewing PRs involving solution changes. The .slnx file preserves comments and formatting which means that if your team likes to annotate the solution file or organize it in a specific way, these details will be saved and preserved between commits.

### ***Reducing File Bloat and Duplication***

Traditionally, the .sln file started off small but grew rapidly as more projects were added to the solution. Every project appeared with a GUID in multiple places and every configuration (Debug/Release etc.) is repeated for each project. In large solutions with many projects, the .sln becomes unwieldy.

The .slnx file resolves this problem with sensible defaults and minimal metadata. As a result of this minimalist approach, the .slnx files stays lean and clean. This minimalistic footprint also leads to faster load times.

### ***Manual Edits and Tooling***

Manually editing a .sln file that were essentially only readable by Visual Studio, was risky. The new .slnx by contrast, uses XML which is something all developers know. Using XML not only makes it much easier for developers to edit but also for other tools to parse or generate.

The Visual Studio team even provides an open-source library called `vs-solutionpersistence` that provides a unified object model and serializers for working with traditional .sln and the new .slnx formats. This allows 3rd-party tools to read, manipulate, and write solution files programmatically. You can find the GitHub repo here:

<https://github.com/microsoft/vs-solutionpersistence>

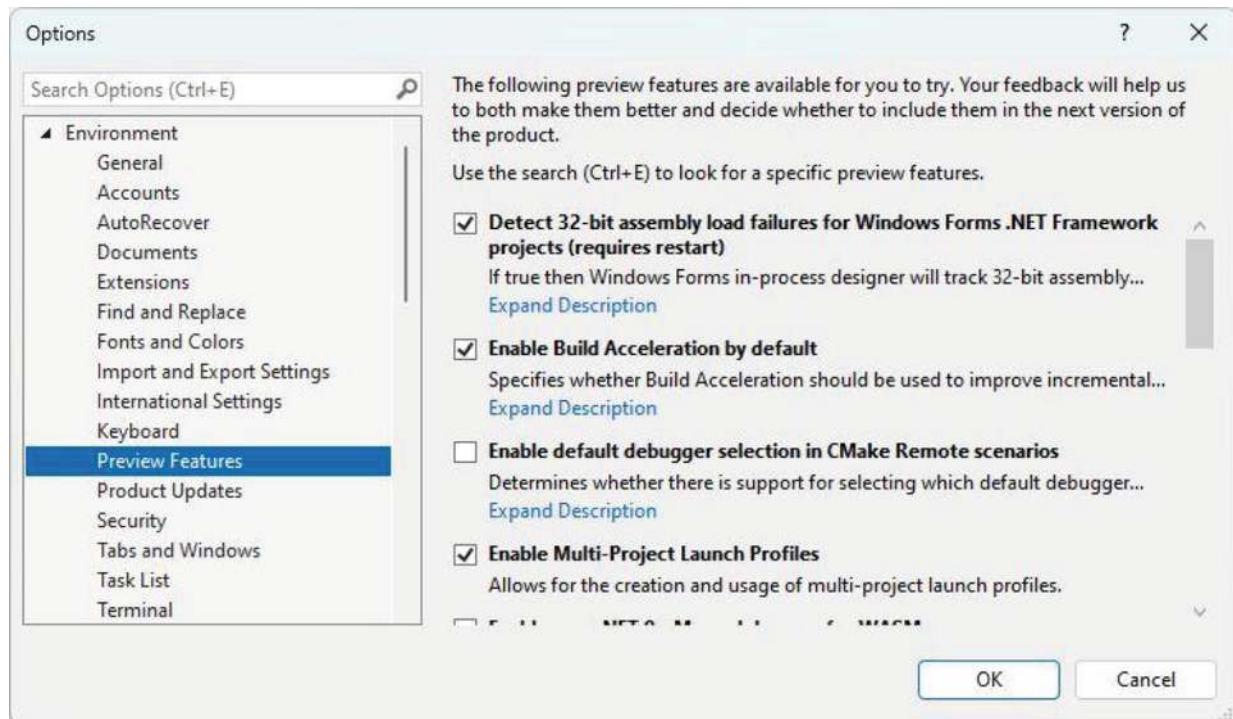
What this essentially does is open the door for better tooling and automation around solution files.

### ***Alignment with Project Files and MSBuild***

Opening up a modern .csproj file you will notice that it too, is XML. By adopting XML for solution files, aligns the solution file format with the project file format. The change is not for show, it means that the solution can integrate better with MSBuild. In fact, MSBuild was updated to natively support .slnx files allowing for seamless building and project manipulation via command-line tools.

### ***Creating and Using .SLNX Solution Files***

Let's see how developers can work with .slnx files in practice? If you are using Visual Studio 2022 17.14 or later, support for .slnx is built into Visual Studio and no longer depends on toggling an experimental flag in Preview Features (Figure 2-119). In earlier versions of Visual Studio (version 17.10 through 17.13) this feature was feature-flagged and had to be opted in to by going to Tools ► Options ► Environment ► Preview Features and checking **Use Solution File Persistence Model**.

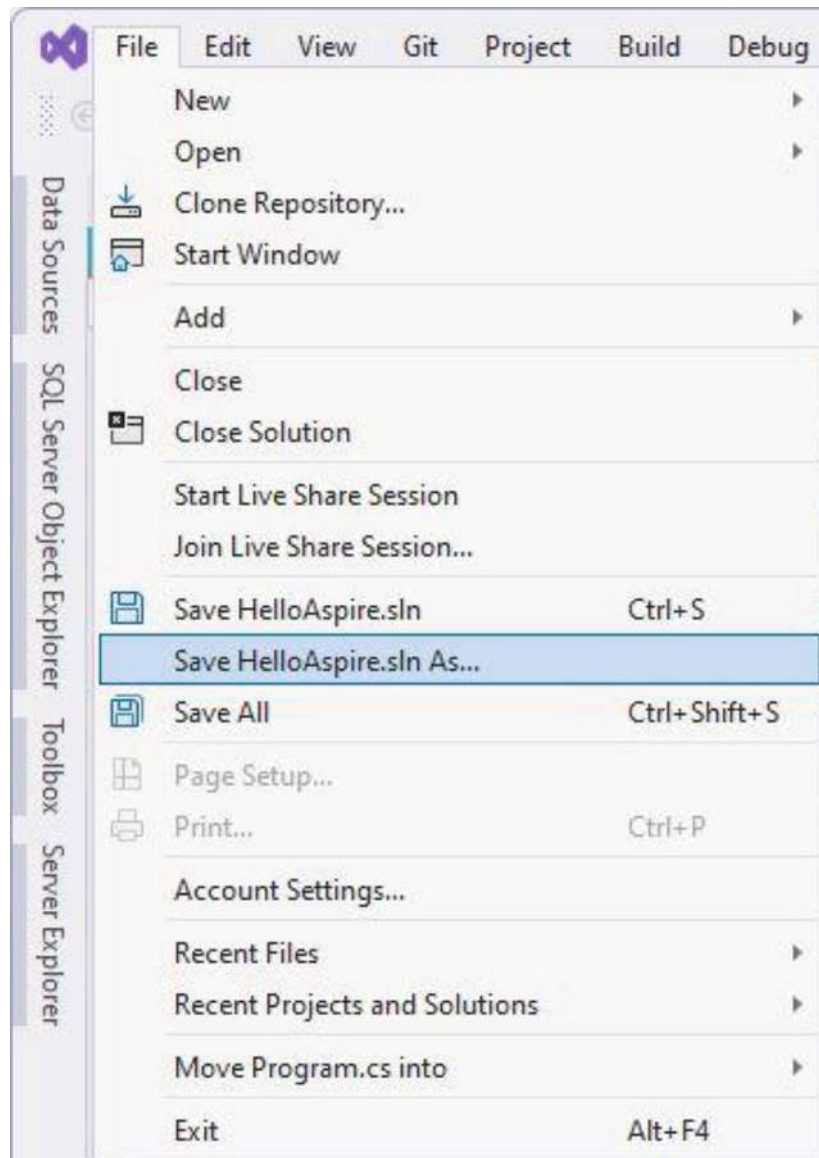


**Figure 2-119** Enabling Preview Features

Once enabled (or with newer builds), Visual Studio allows developers to create and open .slnx files. There are primarily two ways to convert an existing .sln file to a .slnx file.

### ***Using Visual Studio***

The first is via Visual Studio itself. With your solution selected in the Solution Explorer, click on **File ► Save [YourProject.sln] As**, as seen in Figure [2-120](#).



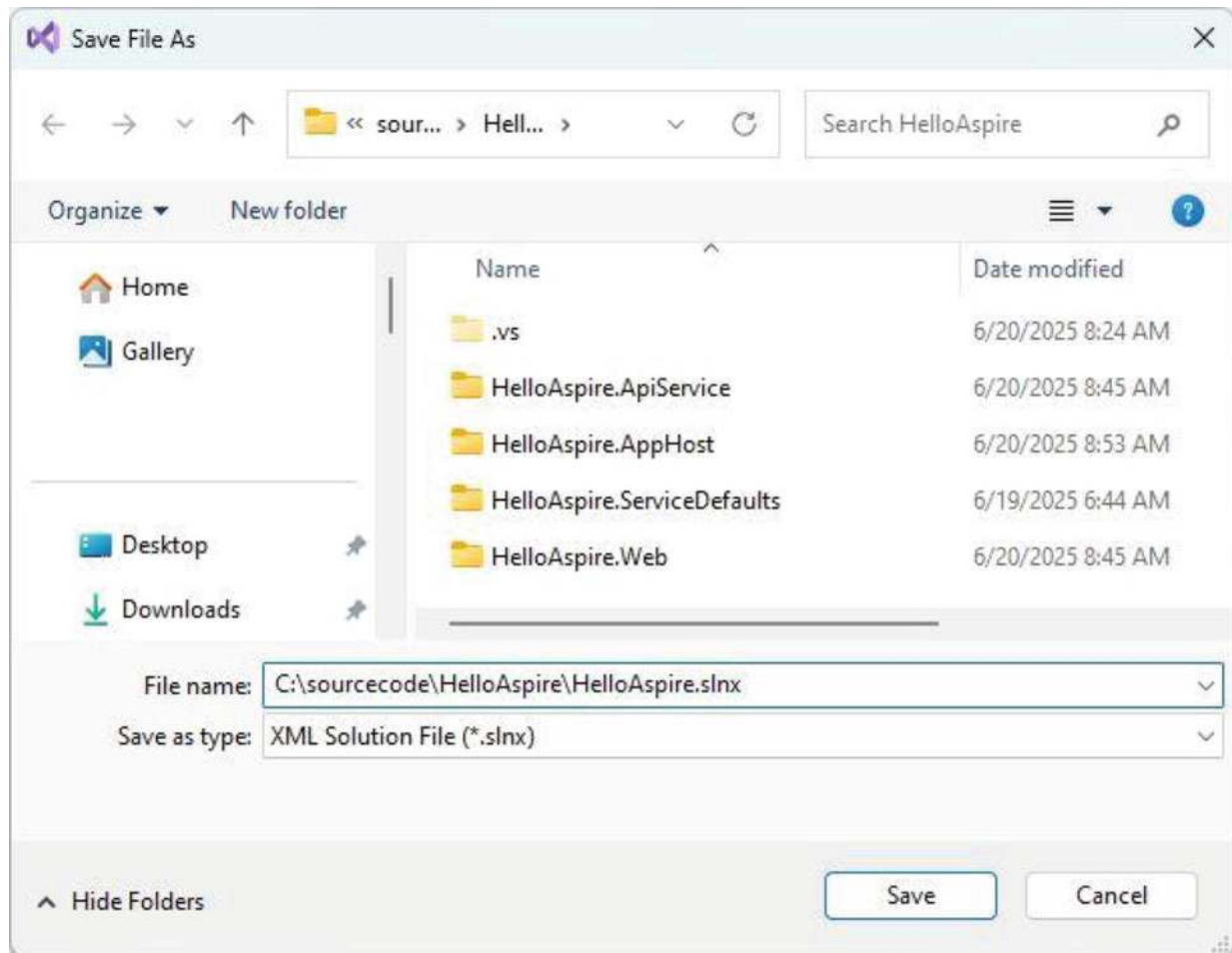
**Figure 2-120** Saving a Solution As

This will open up the familiar **Save File As** dialog, as seen in [Figure 2-121](#). Here you will be able to change the **Save as type** from .sln to .slnx. You will now have two solution files, the .sln and the new .slnx in your solution folder.

Visual Studio might not yet be registered as the default app for .slnx files in Windows. Instead, use File ► Open ► Project from within Visual Studio and select the .slnx file.

Saving your solution as a .slnx file will not remove your old .sln file.



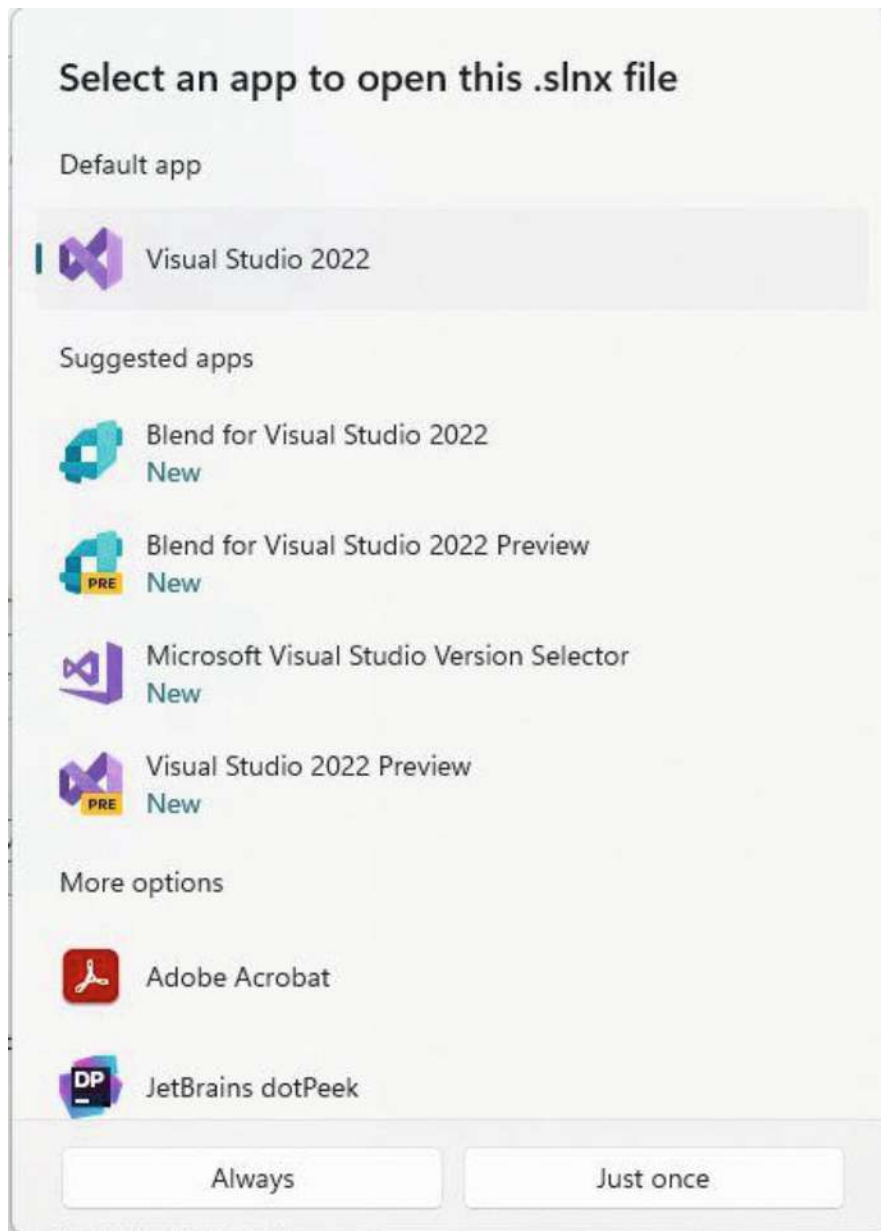


**Figure 2-121** Selecting Save As Type

### ***Via the .NET CLI***

If you prefer to use the command line, the .NET CLI has you covered. The dotnet command-line tool in SDK 9.0.200 and above includes a migrate command. Navigate to your solution folder in a terminal and run: `dotnet sln YourSolution.sln migrate`

This will generate the .slnx file alongside the original. In fact, the `dotnet sln` command allows you to add, list, remove, or migrate projects in a solution file. Type `dotnet sln -?` To see available options.



**Figure 2-122** Always Open With Visual Studio

As always, to open your .slnx in Visual Studio, right-click the file and select **Open With** from the context menu and select **Microsoft Visual Studio 2022**. You will then see the dialog displayed as seen in Figure [2-122](#). Select to always open the .slnx file in Visual Studio 2022, and you'll be all set.

You can now open solutions in Visual Studio using your .slnx file and use Visual Studio as you normally would.

## Final Considerations

It is important to know that only Visual Studio 2022 (with the appropriate updates) knows about .slnx files. If a team member is still using Visual Studio 2019 or earlier, they will not be able to open an .slnx solution.

Secondly, do not keep both the .sln and .slnx files in the repo. It might be tempting to maintain both versions of the solution file in the repo, but Microsoft recommends against it. Keeping both files around invites confusion and keeping these files in sync manually will be error-prone and defeats the purpose of simplifying the solution file in the first place. If you have to keep both files around, consider using a tool like `dotnet slnsync` to sync both the old and new solution file formats. You can find this tool here:

<https://github.com/edvilme/dotnet-sln-sync>

In real-world development scenarios, adopting the new .slnx format should relieve your source control workflow considerably. Moving from the old solution file to something as clean as the new format is a no-brainer for a file that changes often. While it isn't flashy like a new language feature, it will save teams hours of time and make collaboration smoother.

---

## Summary

In this chapter, we explored several practical aspects of working with Visual Studio 2022. We started off looking at the various project templates available, such as Console Apps, Windows Forms, Worker Services, and different web application types.

We also took a look at .NET Aspire, a modern approach to orchestrating distributed applications. We covered how to create a new Aspire application, how each project in an Aspire solution works, and how Aspire simplifies service orchestration and configuration.

Moving to productivity features, we then had a look at NuGet, npm, and creating code snippets. Next, we covered how bookmarks and code shortcuts streamline code navigation, and how Server Explorer allows developers to manage databases directly inside Visual Studio.

Finally, we discussed the Visual Studio menu system, highlighting lesser-known tools such as C# Interactive and Code Metric Results that boost your day-to-day efficiency. These topics give developers a solid foundation for working more effectively with Visual Studio 2022.

## 3. Debugging Your Code

Dirk Strauss<sup>1</sup> 

(1) Uitenhage, South Africa

---

Debugging code is probably one of the most essential tasks that a developer performs. Being able to run your application and pause the execution of code midway is a lifesaver. But there is a lot more to debugging than just setting breakpoints and viewing results.

In this chapter, we will be discussing the options available to you as a developer that needs to effectively debug their code. We will be looking at

- Using breakpoints, conditional breakpoints, breakpoint actions, and labels, and exporting breakpoints
- Using data tips
- The DebuggerDisplay attribute
- Diagnostic Tools and Immediate Window
- Attaching to a running process
- Remote debugging

Visual Studio provides all the tools you need to effectively debug problematic code. Without being able to debug your code, it will be virtually impossible to resolve any issues you might be experiencing.

Not being able to effectively debug your application (not knowing how to effectively use the tools you have) is just as bad as not having the tools to debug with in the first place.

---

### Debugging Toolbar Location

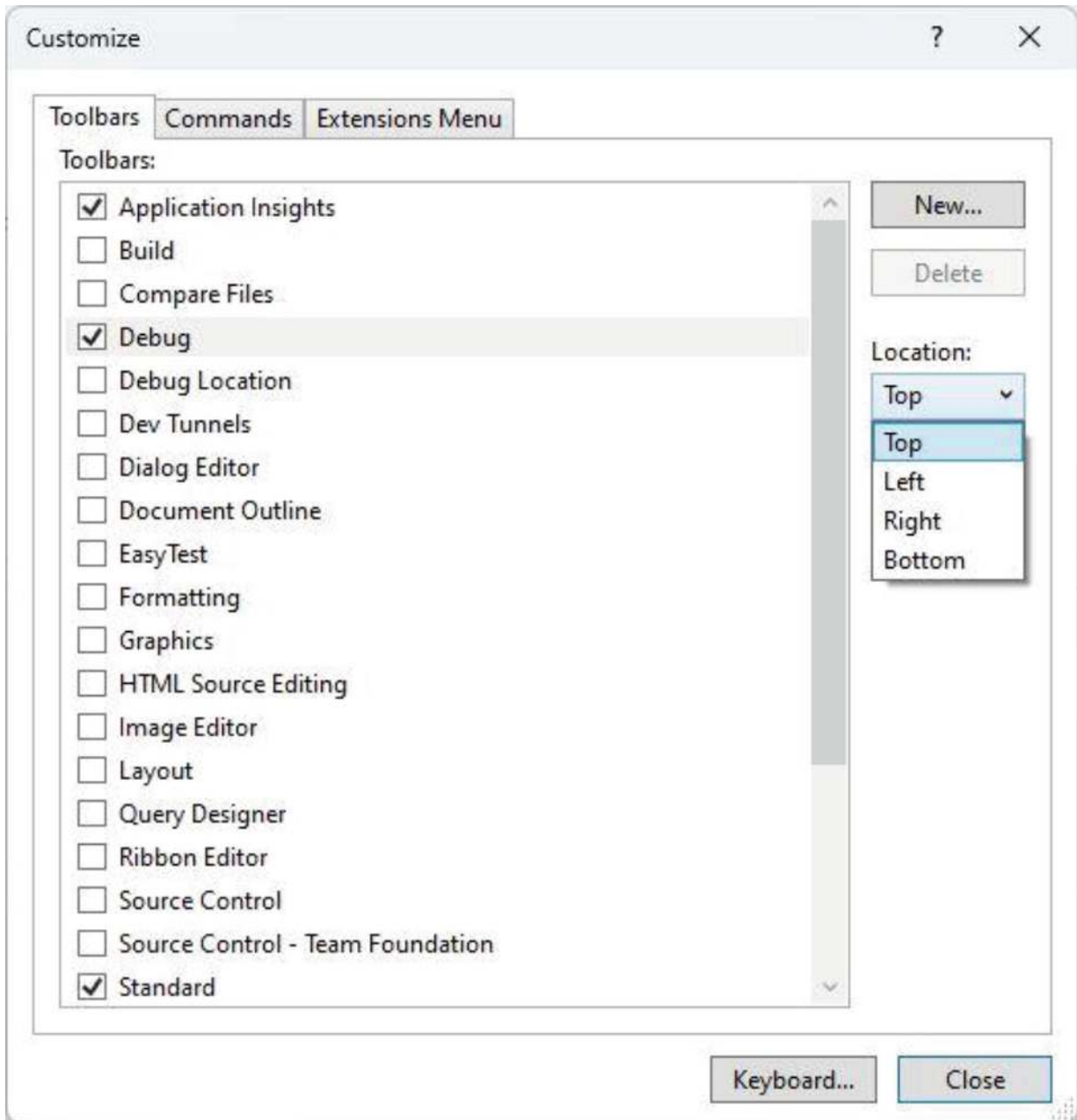
Visual Studio allows developers to set the location of any toolbar. You can, therefore, set the location of the debugging toolbar by right-clicking on any toolbar under the menu in Visual Studio and choosing Customize from the context menu (Figure [3-1](#)).

- ✓ Application Insights
  - Build
  - Compare Files
- ✓ Debug
  - Debug Location
  - Dev Tunnels
  - Dialog Editor
  - EasyTest
  - Formatting
  - Graphics
  - HTML Source Editing
  - Image Editor
  - Layout
  - Query Designer
  - Ribbon Editor
  - Source Control
  - Source Control - Team Foundation
- ✓ Standard
  - Table Designer
  - Test Adapter for Google Test
- ✓ Text Editor
  - View Designer
  - Web One Click Publish
  - XML Editor
  - XtraReports
- Customize...



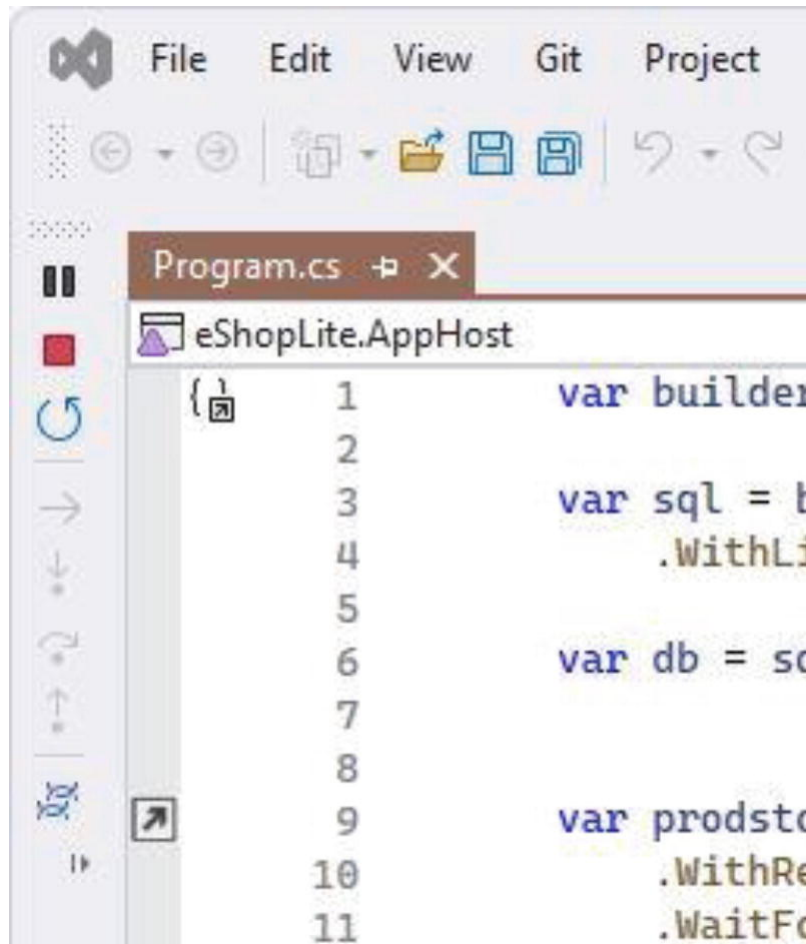
**Figure 3-1** Customize the Toolbar Location

This will display the Customize dialog window where you can select the Debug toolbar and choose the Location as seen in Figure [3-2](#).



**Figure 3-2** Choosing the Debug Toolbar Location

The next time you start a debug session, you will see the Debug toolbar pinned to the left of your Visual Studio window (Figure [3-3](#)).



**Figure 3-3** Debug Toolbar Located on the Left

Some developers might find this more suited to their workflow, as opposed to keeping the Debug Toolbar at the top.

---

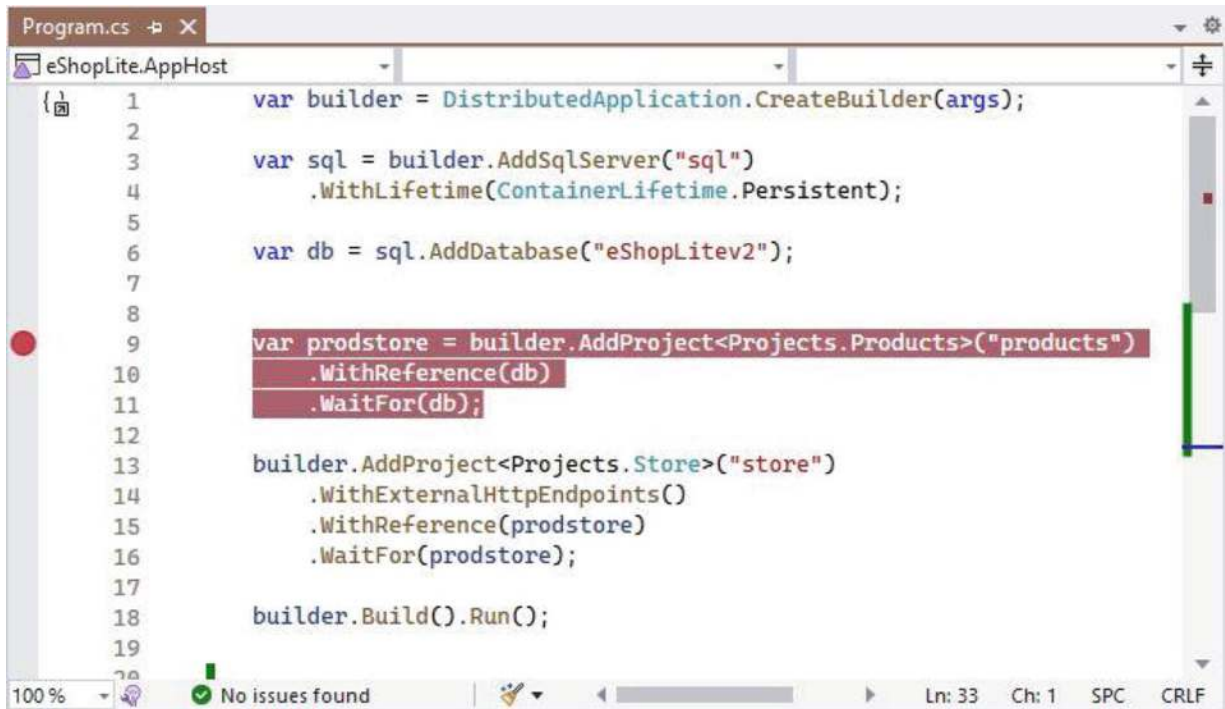
## Working with Breakpoints

If you're already familiar with debugging in Visual Studio, much of this chapter may feel routine. However, you may still discover features and techniques you haven't used before, so I invite you to read on.

If you are new to Visual Studio, debugging means running your application with the debugger attached. Debugging allows you to step through the code and view the values stored in variables. More importantly, you can see how those values change.

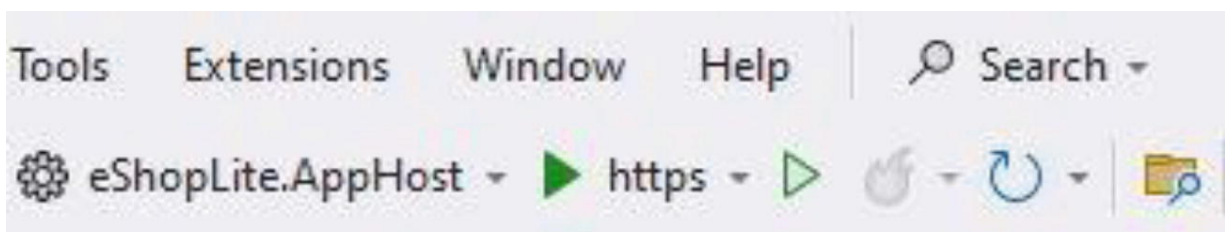
### Setting a Breakpoint

The most basic task of debugging is setting a breakpoint. Breakpoints mark the lines of code that you want Visual Studio to pause at, allowing you to take a closer look at what the code is doing at that particular point in time. To place a breakpoint in code, click the margin to the left of the line of code you want to inspect, as shown in Figure 3-4.



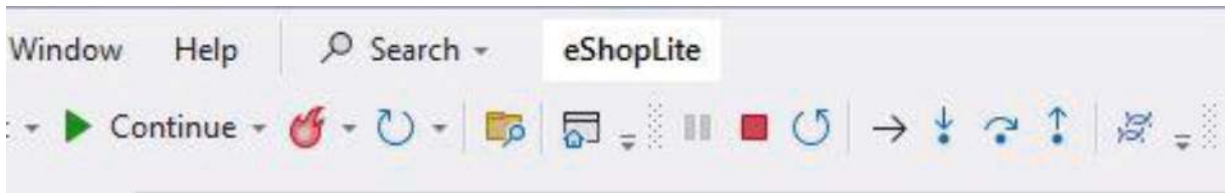
**Figure 3-4** Setting a Breakpoint

Press F5 or click Debug ► Start Debugging to run your application. You can also just click the Start button as shown in Figure 3-5.



**Figure 3-5** The Start Debug Button

After you start debugging, and a breakpoint is hit, the debug toolbar in Visual Studio changes as seen in Figure 3-6.



**Figure 3-6** Debug Toolbar When Breakpoint Hit

The *Start* button now changes and displays *Continue*. Remember, at this point, your code execution is paused in Visual Studio at the breakpoint you set earlier.

In order to step through your code, you can click the step buttons as displayed in Figure [3-7](#).



**Figure 3-7** Step Buttons

From left to right, these buttons are as follows:

- Show Next Statement (Alt+Num\*)
- Step Into (F11)
- Step Over (F10)
- Step Out (Shift+F11)

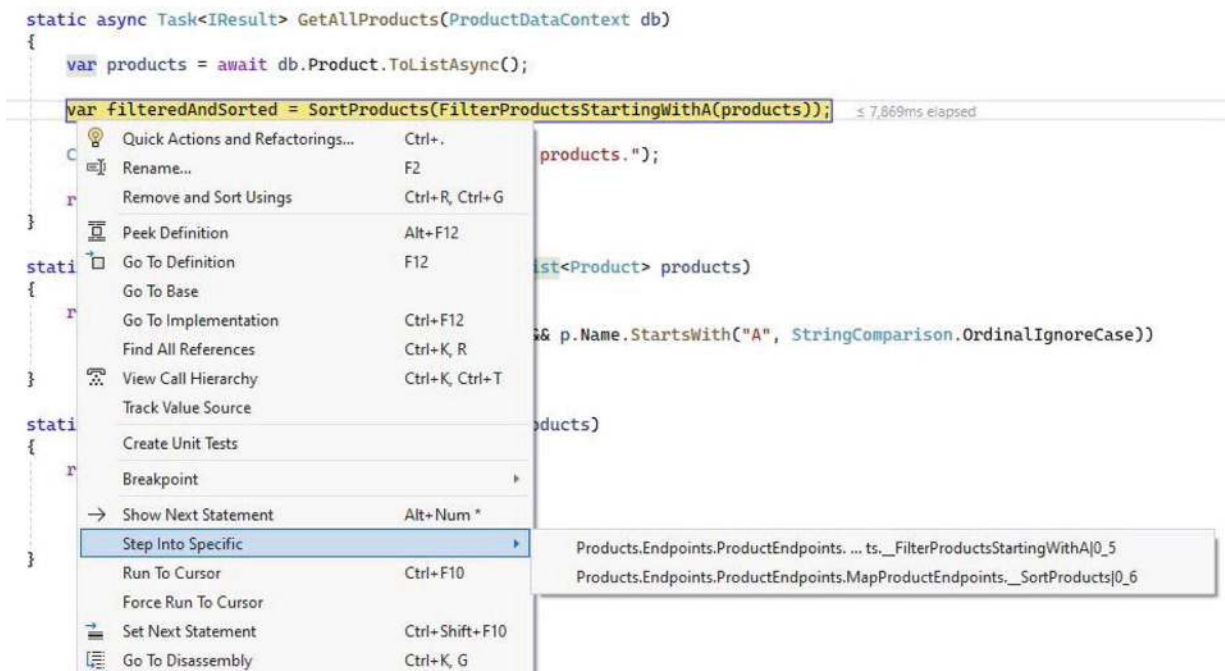
When you step into a method, you jump to the point in the editor where that method's code is. If you do not want to step into the method, you can click the Step Over button or press F10 to carry on with the next line of code. If you are inside a method and want to step out and continue debugging the calling code, click the Step Out button or press Shift+F11.

### ***Step into Specific***

When debugging your code, Step Into (F11) lands you inside a method that you did not intend to inspect. This is especially frustrating when you have to deal with nested calls. This is where Step into Specific can help. Instead of leaving you up to the mercy of the call order, this

feature gives you the flexibility to choose the exact method you want to step into from a context menu.

When you hit a breakpoint, and reach a nested call, as shown in Figure 3-8, right-click on the line of code to bring up the context menu. There, you will see an option to Step into Specific with the nested methods listed for you to choose from.



**Figure 3-8** Step into Specific

Choose the method you are interested in and Visual Studio will take you directly to that method. This is useful when dealing with large codebases and noisy method calls.

## ***Run to Click***

A quiet productivity booster that hides in plain sight is Run to Click. When you are debugging, and hover your mouse over a line of code, you will see a green glyph appear in the left margin as shown in Figure 3-9.

```

static async Task<IResult> GetAllProducts(ProductDataContext db)
{
    var products = await db.Product.ToListAsync();

    var filteredAndSorted = SortProducts(FilterProductsStartingWithA(products));

    Console.WriteLine($"Returning {products.Count} products.");

    return Results.Ok(products);
}

static List<Product> FilterProductsStartingWithA(List<Product> products)
{
    return products
        .Where(p => !string.IsNullOrEmpty(p.Name) && p.Name
            .StartsWith("A", StringComparison.OrdinalIgnoreCase))
        .ToList();
}

static List<Product> SortProducts(List<Product> products)
{
    return products
        .OrderBy(p => p.Name)
        .ThenBy(p => p.Price)
        .ToList();
}

```



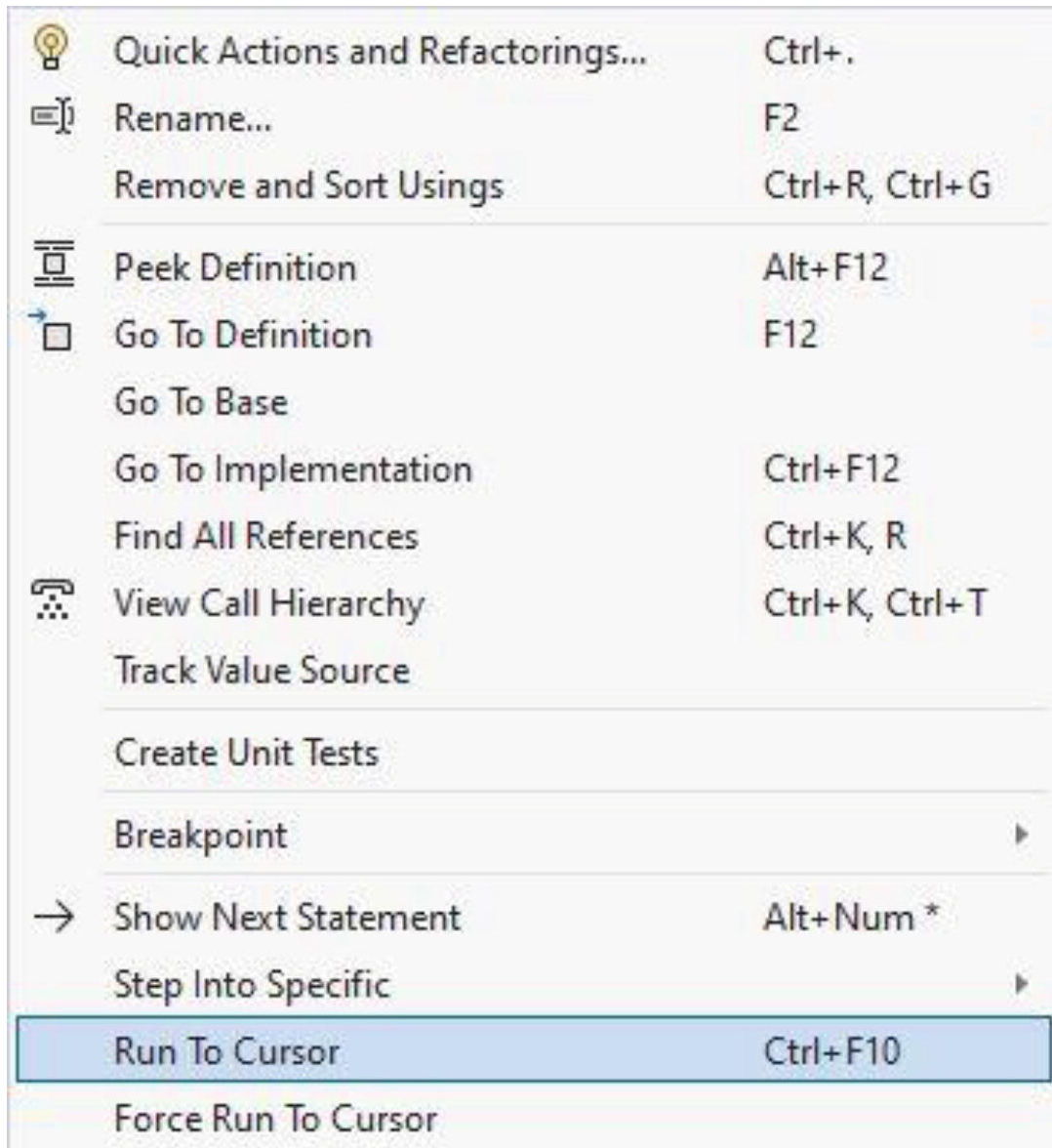
**Figure 3-9** Run to Click

Clicking on this glyph will let Visual Studio run everything up to that line, without you needing to set any additional breakpoints. Run to Click minimizes interruptions and allows developers to jump straight to the line of code they care about without having to step through setup code or lengthy iterations.

## ***Run to Cursor***

In a similar fashion to Run to Click, Run to Cursor allows you to execute code up to a specific point without setting a breakpoint. To use it, put your cursor on the line of code where you want execution to pause and right-click to bring up the context menu. Then select Run to Cursor as seen in Figure [3-10](#). You can also just hit Ctrl+F10.





**Figure 3-10** Run to Cursor

Visual Studio will run the code up to that line, providing that the code is reachable from the current execution path. While Run to Click lends itself well to mouse users, Run to Cursor is perfect for keyboard-driven workflows.

### ***Force Run to Cursor***

Where Run to Cursor is a gentle approach to debugging, Force Run to Cursor acts more like a bulldozer. Taking the Run to Cursor one step

further, Force Run to Cursor ensures that you skip over method calls and breakpoints to get you to your code.

To use it, right-click the line of code that you want to force-run to and choose Force Run to Cursor as seen in Figure 3-10 (Just below Run to Cursor). Visual Studio will now ignore breakpoints and go directly to your code. You can also hold down the Shift button, and the Run to Click glyph we saw earlier, will change to a double-arrow glyph. This also allows you to Force Run to Cursor.

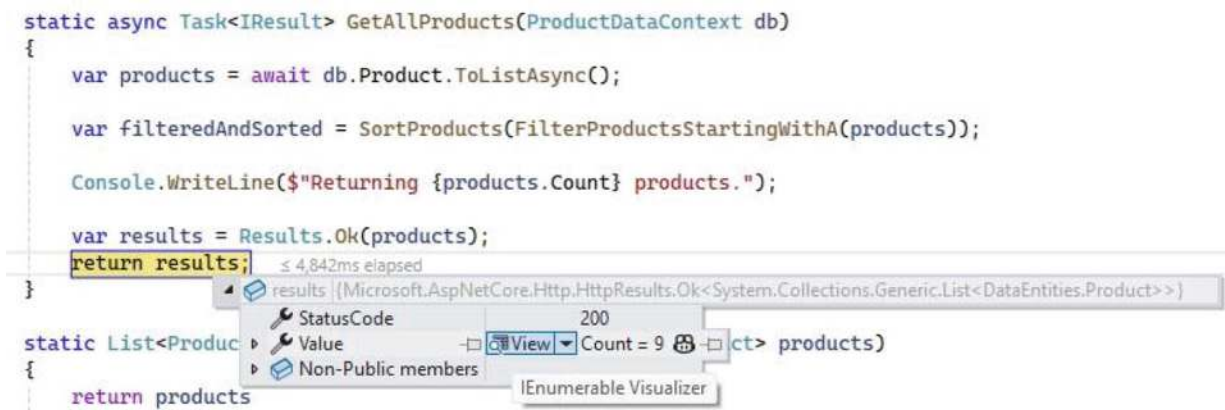
## Visualizers

Debugging your code is more than just stopping the code execution at a particular line. It's about understanding the data flowing through your application. Visual Studio provides a set of incredibly helpful tools called visualizers. These allow you to inspect complex objects such as collections, JSON, XML, and even strings in a more readable format during a debug session.

Visualizers are accessed directly from a DataTip, Watch Window, QuickWatch dialog box, or Autos window by clicking on a small magnifying glass next to a variable. In fact, a visualizer is always represented by a magnifying glass icon. This will launch the appropriate visualizer based on your selection.

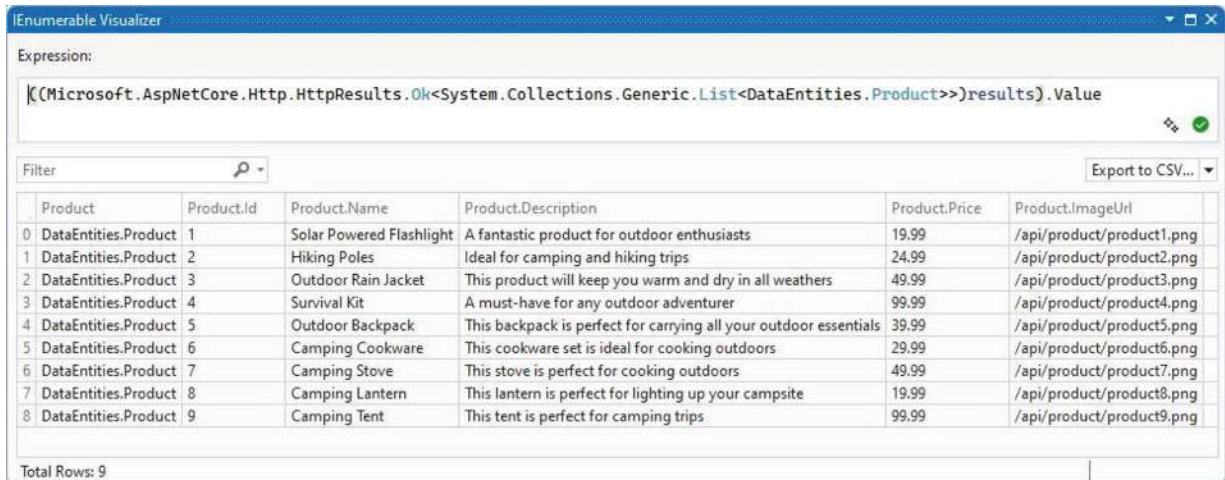
### *IEnumerable Visualizer*

The IEnumerable Visualizer (Figure 3-11) assists developers when debugging collections that might be challenging to decipher in a raw object view.



**Figure 3-11** Opening the IEnumerable Visualizer

Clicking on the View button, you will see the contents presented in a tabular grid (Figure 3-12). This allows you to scroll through the collection and inspect each one individually.



The screenshot shows the 'IEnumerable Visualizer' window. At the top, the 'Expression' field contains the code: `((Microsoft.AspNetCore.Http.HttpResults.Ok<System.Collections.Generic.List<DataEntities.Product>>)results).Value`. Below this is a 'Filter' input field and an 'Export to CSV...' button. The main area is a table with 6 columns: Product, Product.Id, Product.Name, Product.Description, Product.Price, and Product.ImageUrl. The table contains 9 rows of product data. At the bottom, it says 'Total Rows: 9'.

Product	Product.Id	Product.Name	Product.Description	Product.Price	Product.ImageUrl
0 DataEntities.Product	1	Solar Powered Flashlight	A fantastic product for outdoor enthusiasts	19.99	/api/product/product1.png
1 DataEntities.Product	2	Hiking Poles	Ideal for camping and hiking trips	24.99	/api/product/product2.png
2 DataEntities.Product	3	Outdoor Rain Jacket	This product will keep you warm and dry in all weathers	49.99	/api/product/product3.png
3 DataEntities.Product	4	Survival Kit	A must-have for any outdoor adventurer	99.99	/api/product/product4.png
4 DataEntities.Product	5	Outdoor Backpack	This backpack is perfect for carrying all your outdoor essentials	39.99	/api/product/product5.png
5 DataEntities.Product	6	Camping Cookware	This cookware set is ideal for cooking outdoors	29.99	/api/product/product6.png
6 DataEntities.Product	7	Camping Stove	This stove is perfect for cooking outdoors	49.99	/api/product/product7.png
7 DataEntities.Product	8	Camping Lantern	This lantern is perfect for lighting up your campsite	19.99	/api/product/product8.png
8 DataEntities.Product	9	Camping Tent	This tent is perfect for camping trips	99.99	/api/product/product9.png

**Figure 3-12** IEnumerable Visualizer

The IEnumerable Visualizer also allows you to filter records and export them to CSV or Excel. You can even right-click on a row and copy the data to the clipboard.

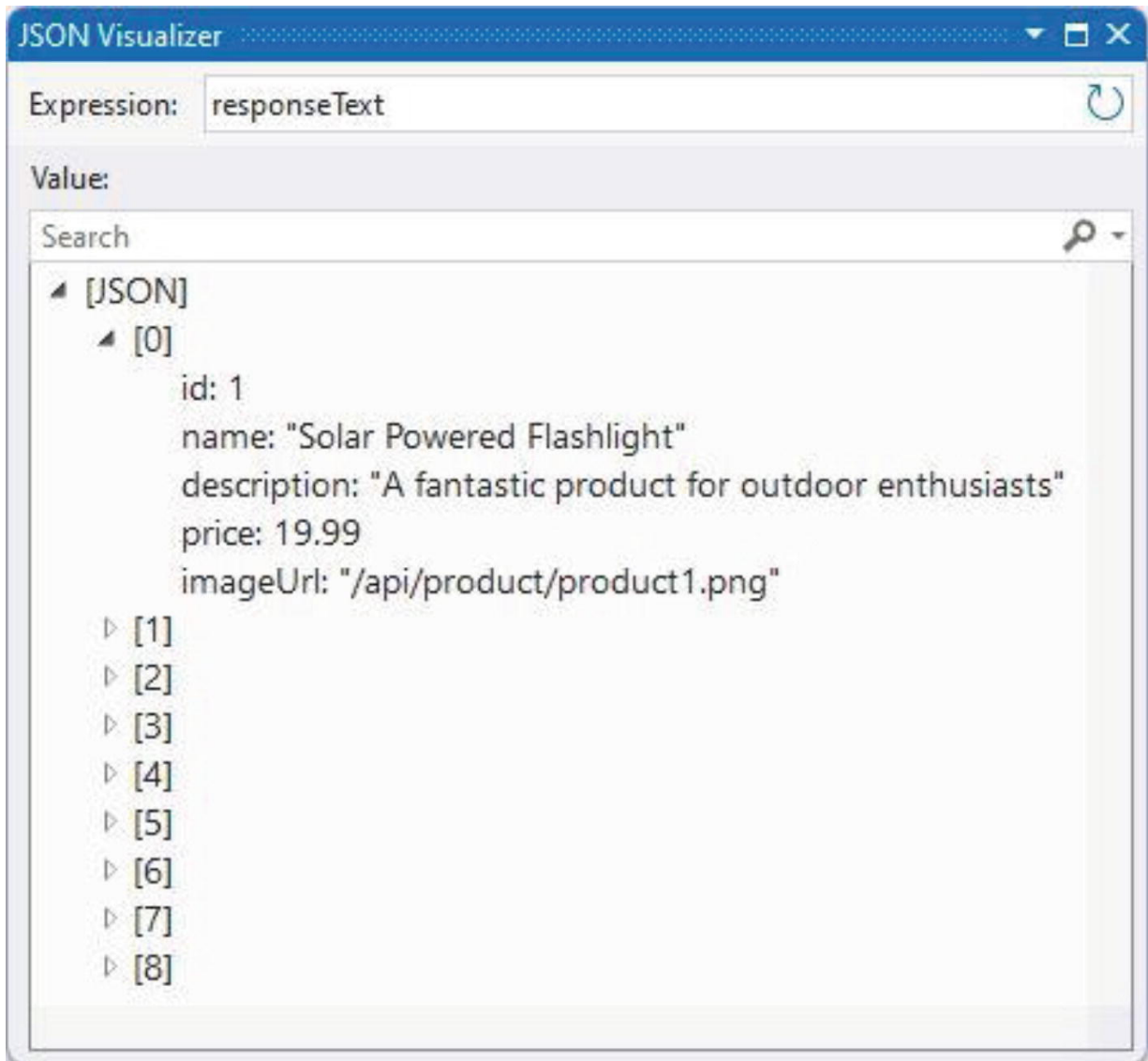
## JSON Visualizer

When dealing with APIs, for example, Visual Studio's JSON Visualizer (Figure 3-13) allows you to format the raw string into JSON.



**Figure 3-13** Opening the JSON Visualizer

The JSON Visualizer makes the wall of JSON text infinitely easier to read and understand, as shown in Figure 3-14.



**Figure 3-14** JSON Visualizer

You have similar visualizers for Text, XML, and HTML, as seen in [Figure 3-13](#)

The JSON Visualizer gives you a collapsible tree view with proper indentation.

When working with a REST API, the JSON Visualizer becomes an essential tool in your debugging toolkit.

## ***Custom Visualizers***

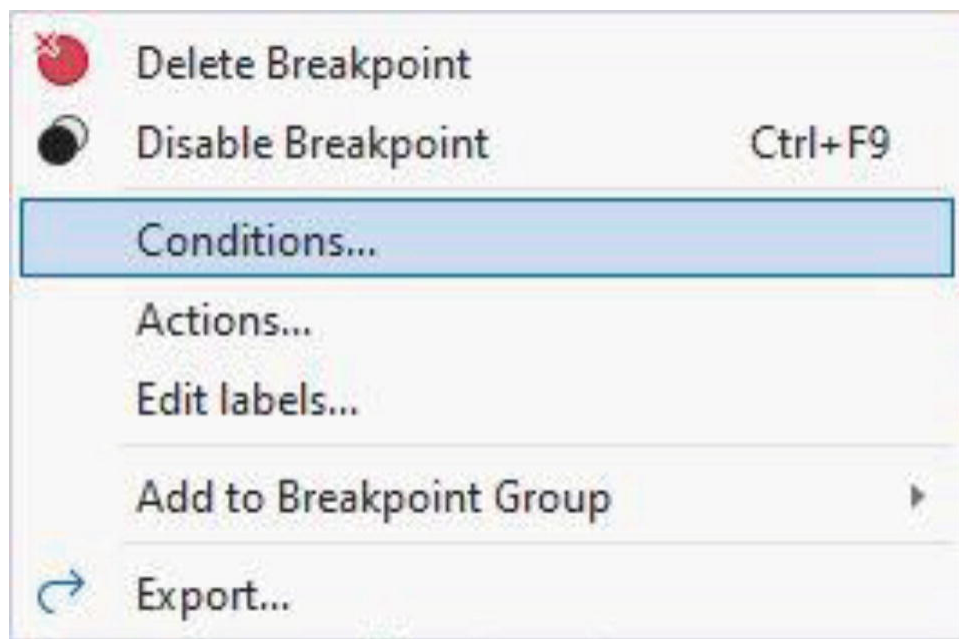
Apart from the standard visualizers that Visual Studio provides, more visualizers might be available for download from Microsoft. You can even create your own visualizer if you have a unique requirement not catered for in Visual Studio. For a walkthrough on accomplishing this, use the following URL as a starting point:

<https://learn.microsoft.com/en-us/visualstudio/debugger/create-custom-visualizers-of-data>

## Conditional Breakpoints and Actions

Sometimes, you need to use a condition to catch a bug. Let's say that you are in a for loop, and the bug seems to be data-related. The erroneous data only seems to enter the loop after several hundred iterations. If you set a regular breakpoint, you will be pressing F10 until your keyboard stops working.

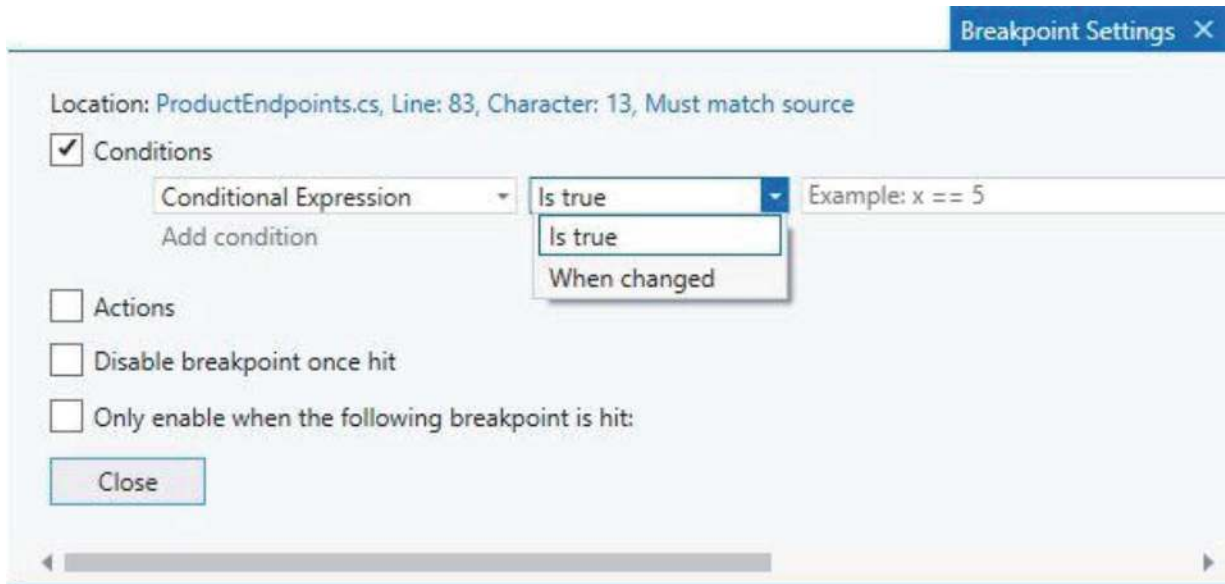
This is a perfect use case for using conditional breakpoints. You can now tell the debugger to break when a specific condition is true. To set a conditional breakpoint, right-click the breakpoint and click Conditions from the context menu, as seen in Figure 3-15.



**Figure 3-15** Breakpoint Context Menu

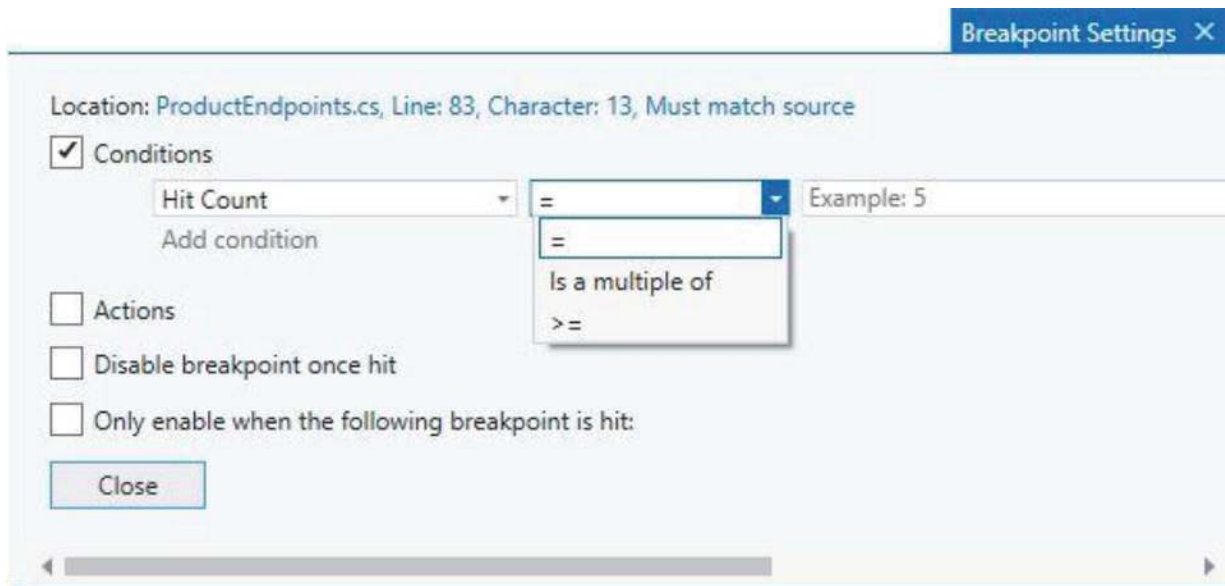


You can now select a conditional expression and select to break if this condition is true or when changed, as seen in Figure [3-16](#).



**Figure 3-16** Conditional Expression

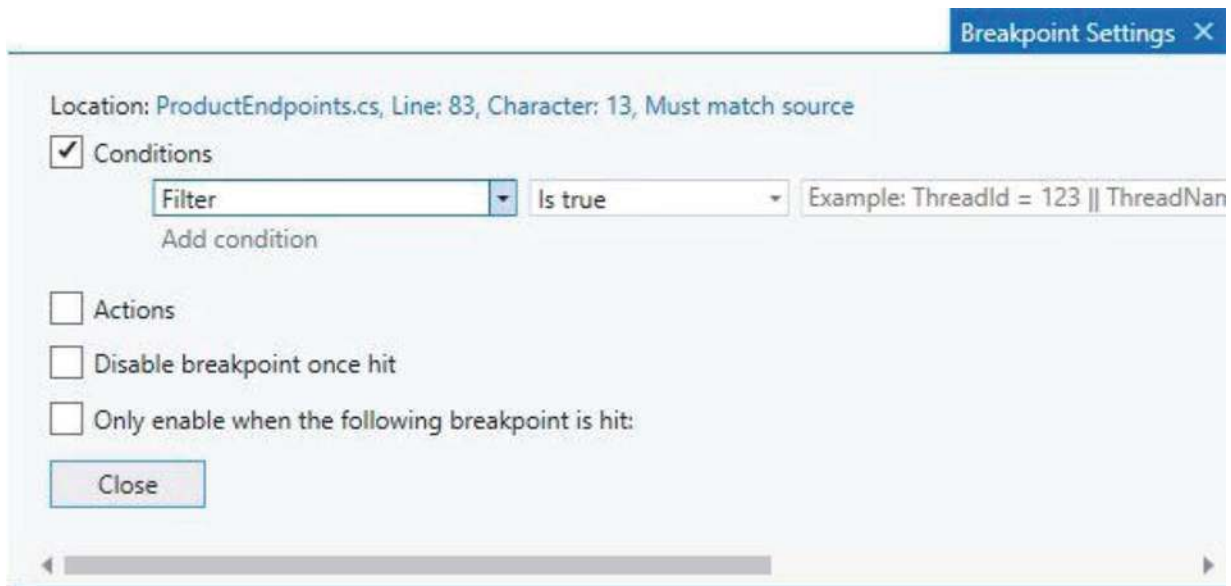
You can also select to break when the Hit Count is equal to, a multiple of, or greater than or equal to a value you set, as shown in Figure [3-17](#).



**Figure 3-17** Hit Count Condition



The last condition you can set on a conditional breakpoint is a Filter, as seen in Figure [3-18](#).



**Figure 3-18** Filter Condition

Take note of the Actions checkbox as seen in the Breakpoint Settings above (Figure [3-18](#)). This is the same Actions menu seen on the context menu in Figure [3-15](#). Here, you can add an expression to log to the Output Window using specific keywords that are accessed using the \$ symbol.

The special keywords are as follows:

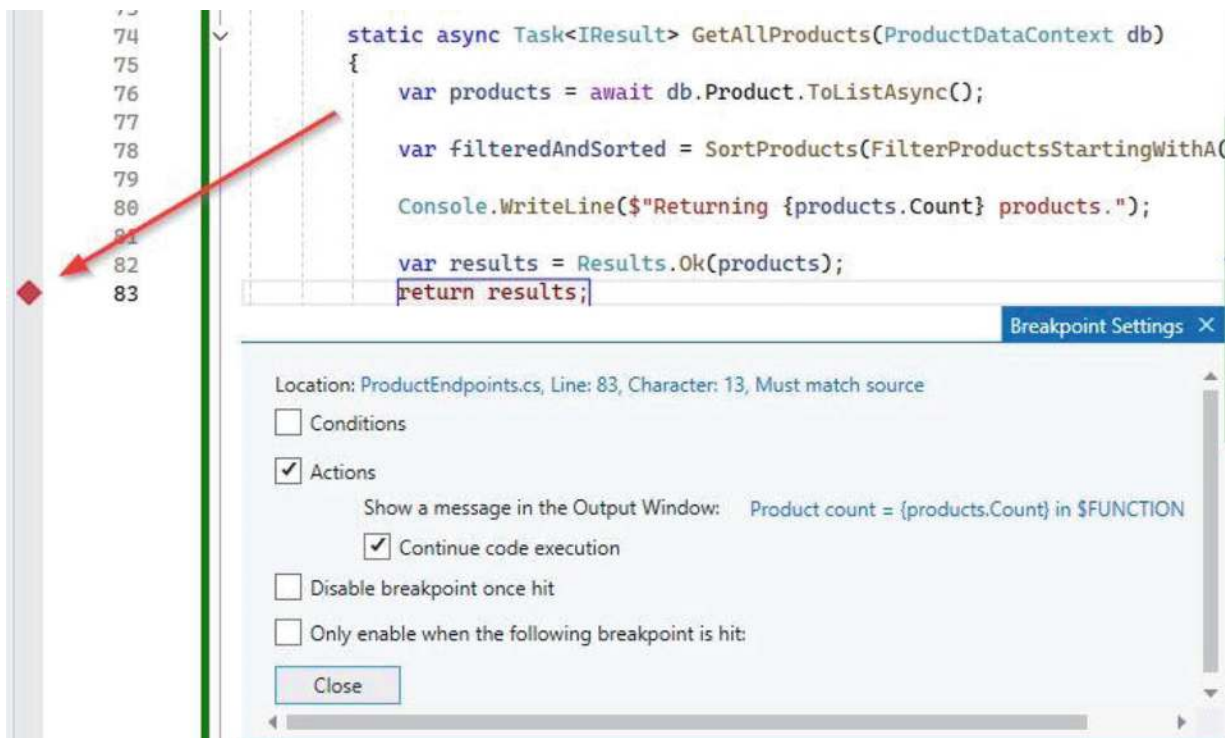
- \$ADDRESS – Current instruction
- \$CALLER – Previous function name
- \$CALLSTACK – Call stack
- \$FILEPOS – The current file and line position
- \$FUNCTION – Current function name
- \$PID – Process ID
- \$PNAME – Process name
- \$TICK – Milliseconds elapsed since the system was started, up to 49.7 days
- \$TID – Thread ID
- \$TNAME – Thread name

You can now use these special keywords to write an entry to the Output Window. You can include the value of a variable by placing it between curly braces (think of interpolated strings). Listing [3-1](#) shows an example of an expression that uses the `$FUNCTION` keyword.

```
Product count = {products.Count} in $FUNCTION
```

**Listing 3-1** Action Expression

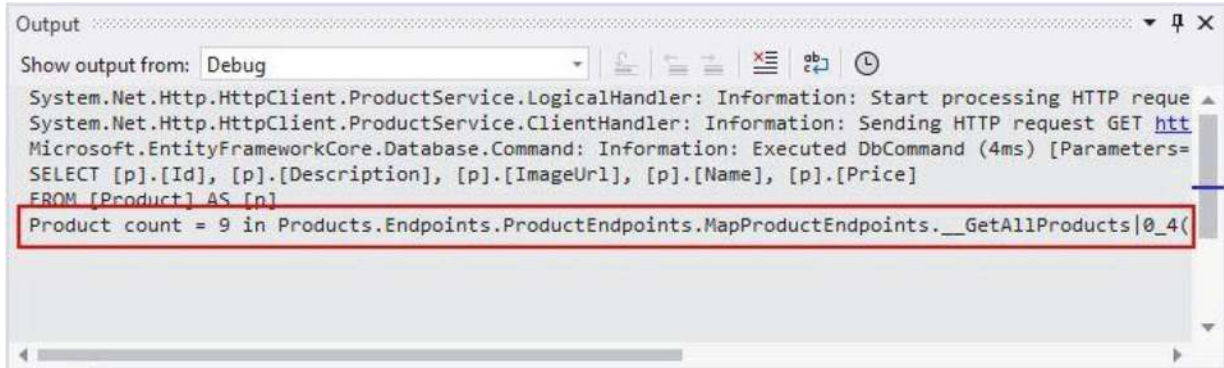
Placing this breakpoint action in your code will cause Visual Studio to change the breakpoint indicator from a circle to a diamond, as shown in Figure [3-19](#).



**Figure 3-19** Breakpoint Action

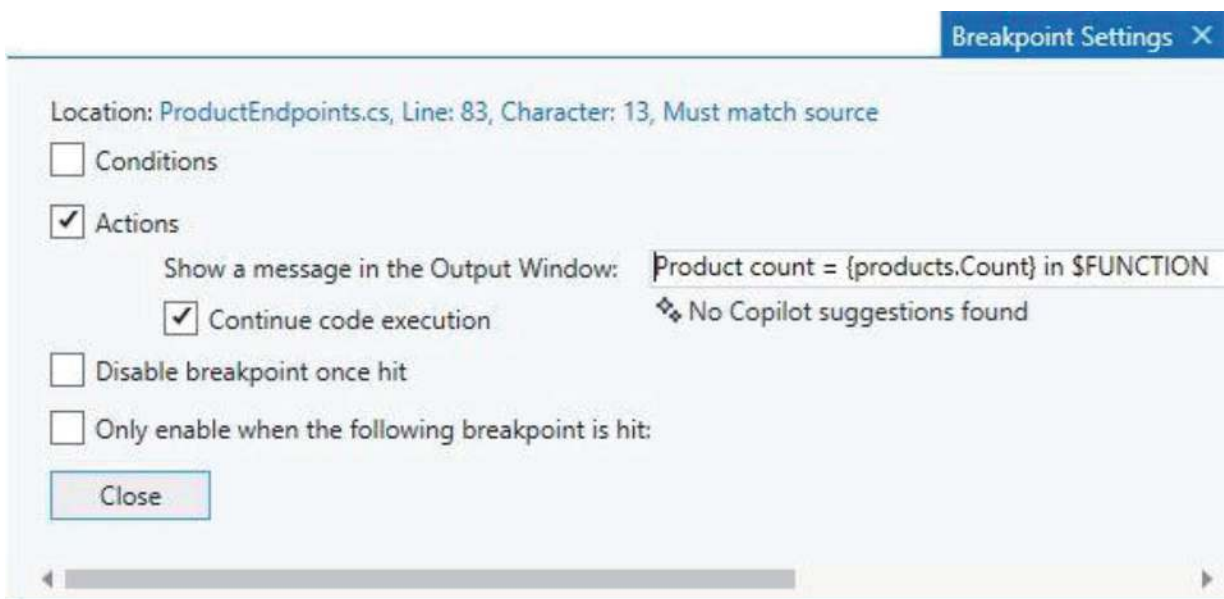
This is also known as a Tracepoint

When you run your application, you will see the expression output in the Output Window displayed in Figure [3-20](#).



**Figure 3-20** Action Expression in Output Window

This is great for debugging because if you don't select a condition, the Action will be displayed in the Output Window without hitting the breakpoint and pausing the code. The breakpoint action can also be edited by clicking on the expression, as seen in Figure 3-21.



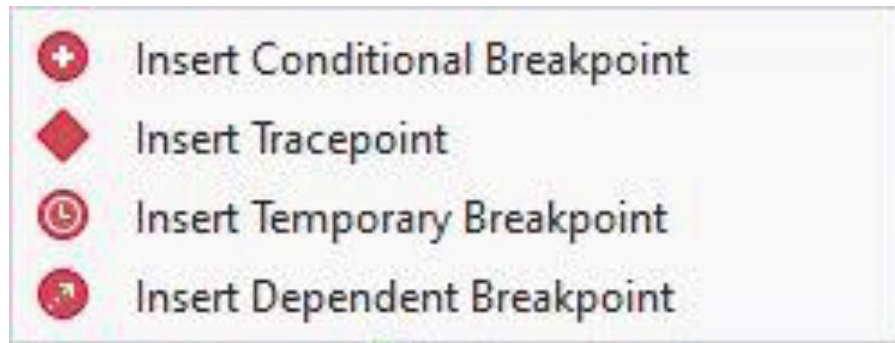
**Figure 3-21** Edit the Breakpoint Action

If you want to pause the code execution, then you need to uncheck the *Continue code execution* checkbox.

## Temporary Breakpoints

There might be times when you only want a breakpoint hit once and never again. A scenario could exist where you need to check the change of a variable in several places, but once this is confirmed to be working,

you do not need to check the variable again. Instead of having to add several breakpoints and having to remove them again afterward, Visual Studio 2022 allows you to set a temporary breakpoint. Instead of clicking in the margin as you normally would do to place a breakpoint, right-click in the margin to see the breakpoint menu. Then select Insert Temporary Breakpoint as seen in Figure [3-22](#).



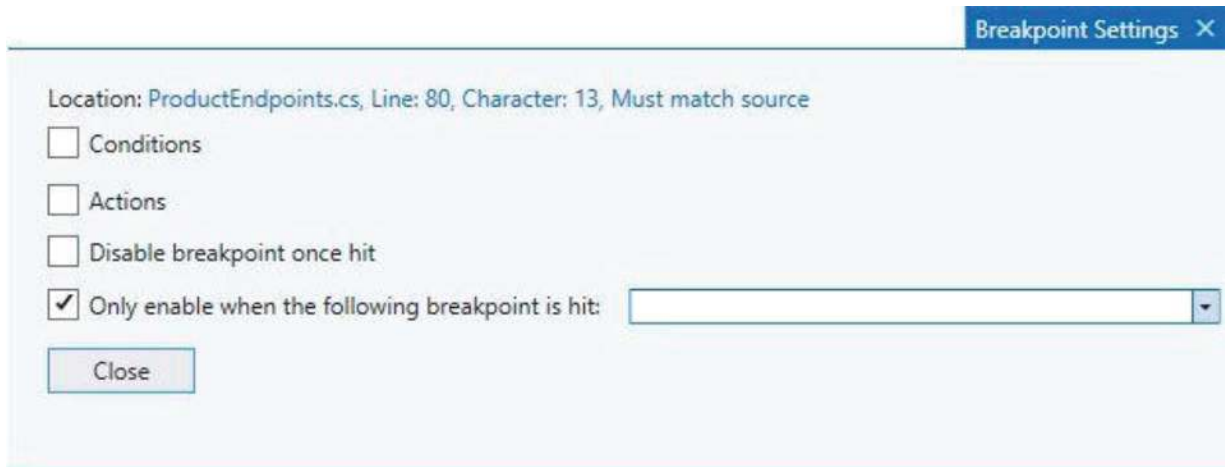
**Figure 3-22** Insert Temporary Breakpoint

You can also hold down Shift+Alt+F9, T to do the same thing (good luck with that muscle memory).

## Dependent Breakpoints

If you look at Figure [3-22](#) again, you will notice an option to insert a dependent breakpoint. A dependent breakpoint is a fantastic addition to Visual Studio because it is a breakpoint that will only pause the debugger when another breakpoint is hit on which it has been marked as a dependent.

As seen in Figure [3-23](#), you can click the drop-down menu, which will show you a list of other breakpoints to choose from. After selecting a dependent breakpoint, the debugger will only pause when the breakpoint you selected from the drop-down is hit.



**Figure 3-23** Insert a Dependent Breakpoint

## Dragging Breakpoints

You can also drag breakpoints to a different line of code. To do this, click and hold on the breakpoint and start dragging your mouse. You can now move it to another line.

## Manage Breakpoints with Labels

As you continue debugging your application, you will be setting many breakpoints throughout the code. Different developers have different ways of debugging. Personally, I add and remove breakpoints as needed, but some developers might end up with several breakpoints (or breakpoints across multiple files) as seen in [Figure 3-24](#).





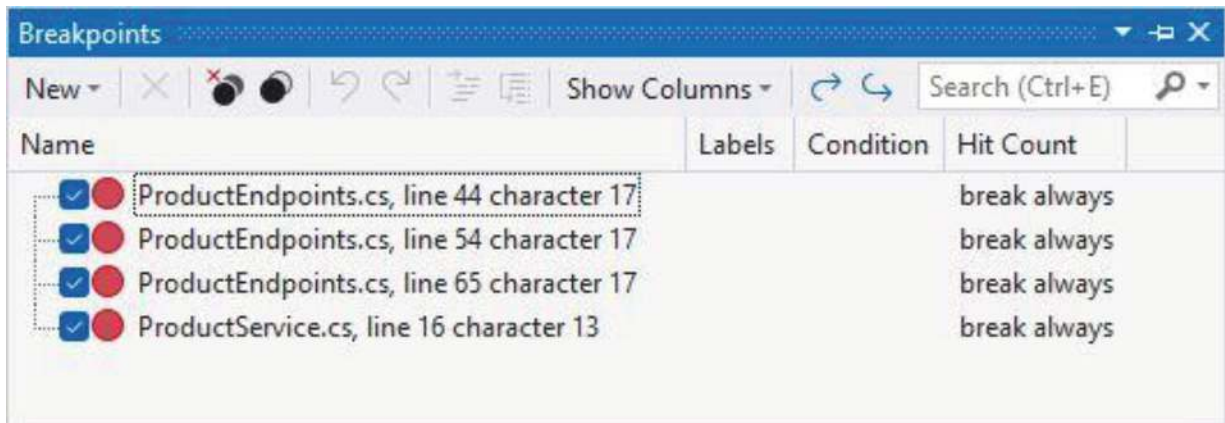
**Figure 3-24** Multiple Breakpoints Set

This is where the Breakpoints window comes in handy. Think of it as mission control for managing complex debugging sessions. This is especially helpful in large solutions where you might have many breakpoints set in various code files throughout your solution.

The Breakpoints window allows developers to manage the breakpoints that they have set by allowing them to search, sort, filter, enable, disable, and delete breakpoints. The Breakpoints window also allows developers to specify conditional breakpoints and actions.

To open the Breakpoints window, click the Debug menu, Windows, and then Breakpoints. You can also press Ctrl+D, Ctrl+B. The Breakpoints window will now be displayed as seen in Figure [3-25](#).



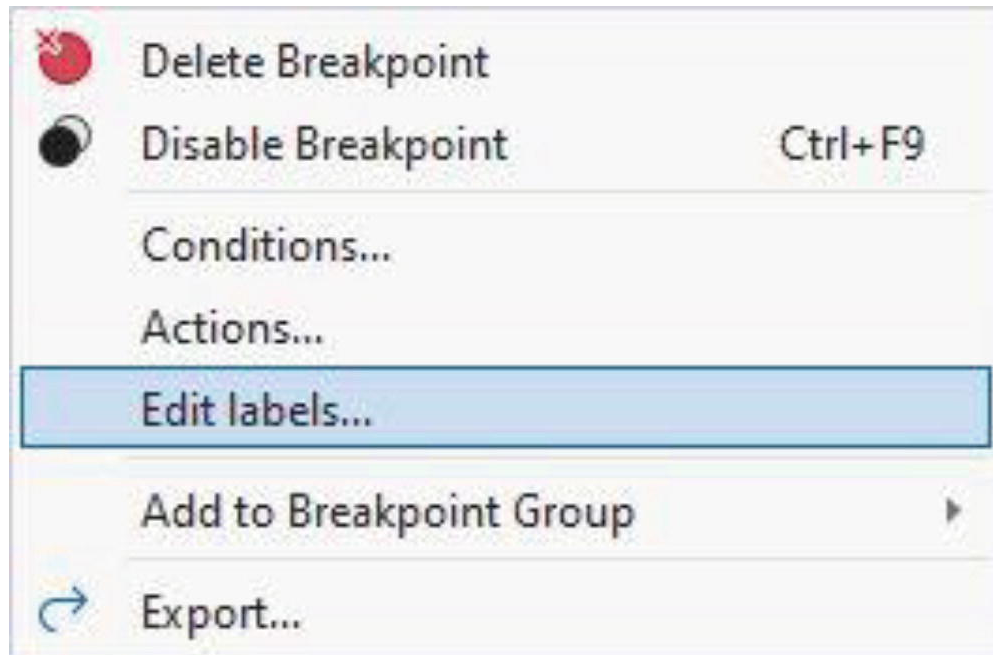


**Figure 3-25** Breakpoints Window

Compare the line numbers of the breakpoints listed in Figure [3-25](#) with the breakpoints displayed in Figure [3-24](#). You will see that this accurately reflects the breakpoints displayed in the Breakpoints window.

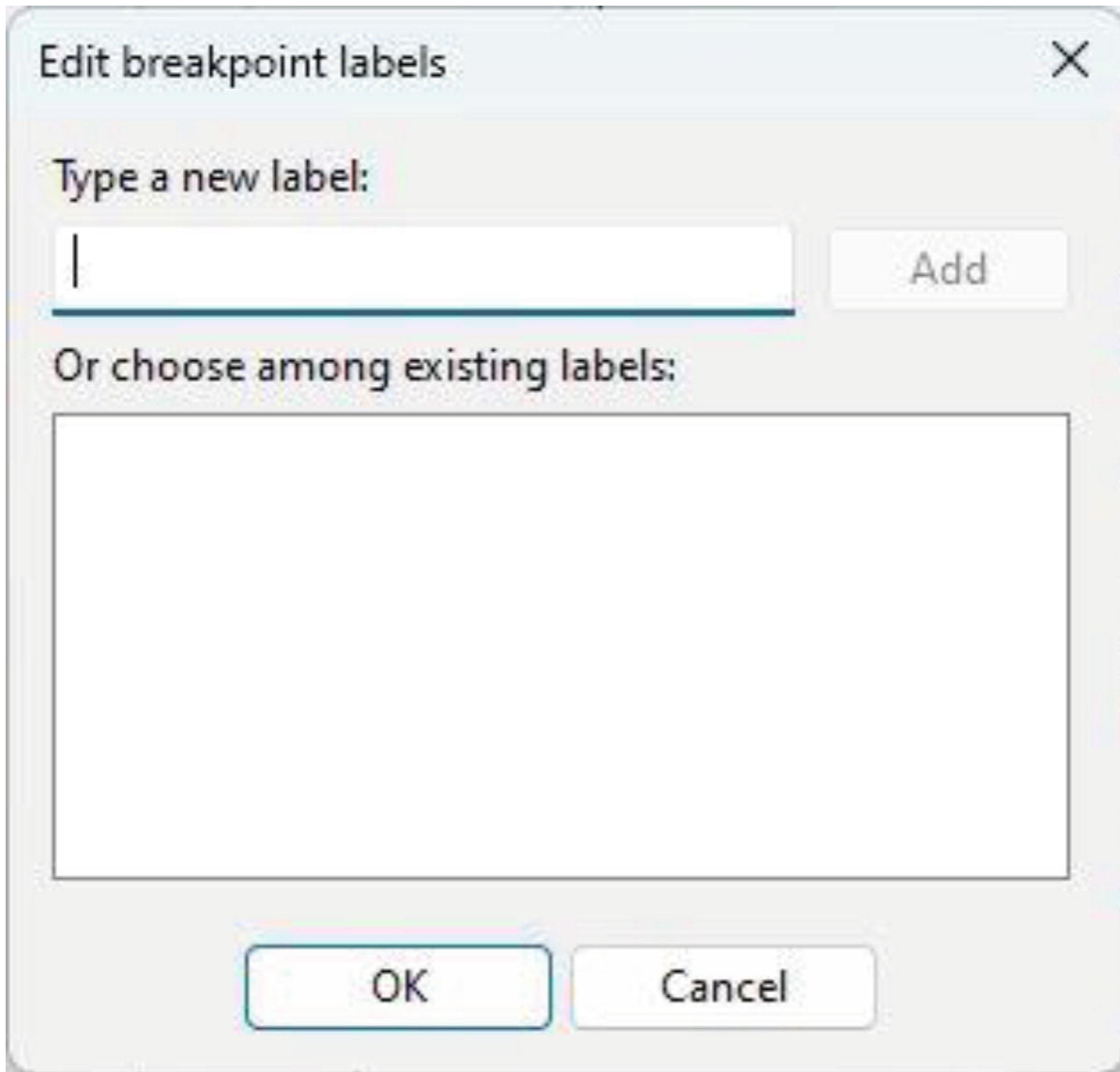
The only problem with this window is that it doesn't help you much in the way of managing your breakpoints. At the moment, the only information displayed in the Breakpoints window is the class name and the line number.

This is where breakpoint labels are very beneficial. To set a breakpoint label, right-click a breakpoint and click Edit labels from the context menu, as shown in Figure [3-26](#).



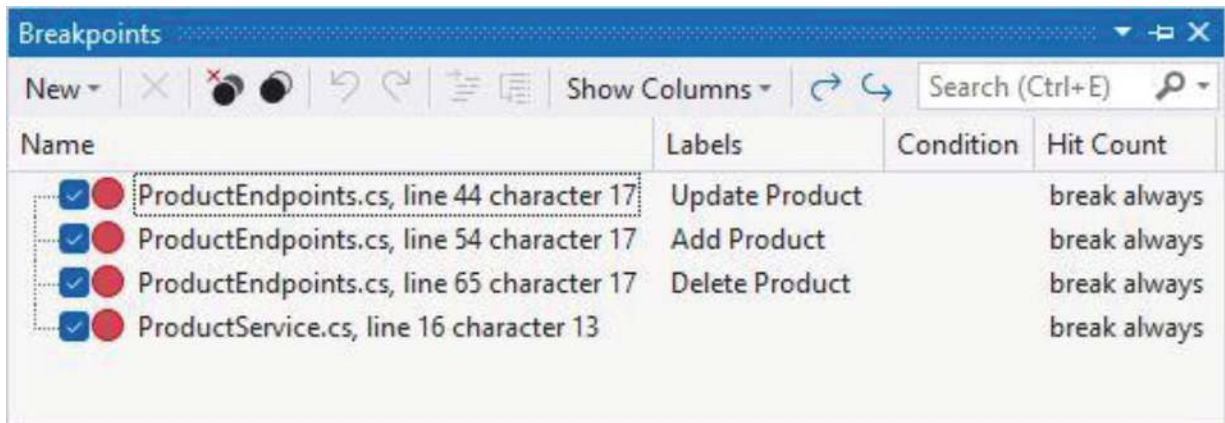
**Figure 3-26** Edit Breakpoint Labels

The Edit breakpoint labels window is then displayed ([Figure 3-27](#)).



**Figure 3-27** Add a New Breakpoint Label

You can type in a new label or choose from any of the existing labels available. If you switch back to the Breakpoints window, you will see that the labels you entered in the Edit Breakpoint Labels window, are displayed (Figure [3-28](#)), making the identification and management of your breakpoints much easier.



**Figure 3-28** Breakpoints Window with Labels Set

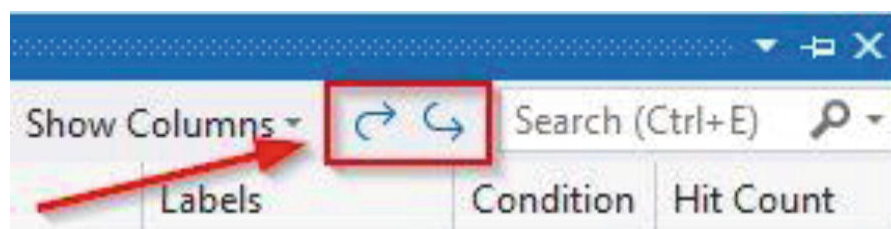
You are in a better position with the breakpoint labels set to manage your breakpoints more effectively. This is because you can immediately identify the purpose of a breakpoint based on the label you've chosen.

## Exporting Breakpoints

If you would like to save the current state and location of the breakpoints you have set, Visual Studio allows you to export and import breakpoints. This will create an XML file with the exported breakpoints that you can then share with a colleague.

I think that the use of Visual Studio Live Share negates the need to share breakpoints with a colleague just for the sake of aiding in debugging an application. There are, however, other situations I can see exporting breakpoints as being beneficial.

To export your breakpoints, you can right-click a breakpoint and click Export from the context menu, or you can click the export button in the Breakpoints window. You can also import breakpoints from the Breakpoints window by clicking the export or import button as highlighted in Figure [3-29](#).



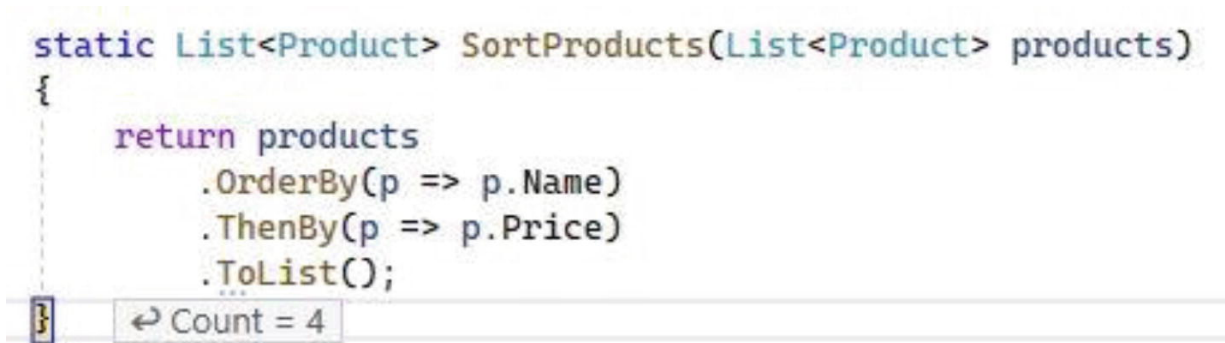
**Figure 3-29** Import or Export Breakpoints

I'm not too convinced that the icons used on the import and export buttons are indicative of importing and exporting something, but that is just my personal opinion.

---

## Viewing Inline Return Values

Visual Studio has introduced Inline Return Values. You no longer need to create temporary variables to inspect return values from functions. As shown in Figure [3-30](#), a return value is displayed.



**Figure 3-30** Inline Return Value

This quality of life improvement allows developers to remain focused on coding without having to go through extra steps of adding temporary variables and removing them afterwards, just to inspect a return value.

As you hit a breakpoint and step through your code, Visual Studio automatically displays the return value next to the closing brace of the method. This immediate feedback makes spotting issues easier, allowing you to verify that your functions are returning the expected results.

---

## Using DataTips

DataTips in Visual Studio allow developers to view information about variables during a debug session. You can only view DataTips in break

mode, and DataTips only work with variables that are currently in scope.

This means that before you can see a DataTip, you are going to have to debug your code. Place a breakpoint somewhere in your code and start debugging. When you hit the breakpoint that you have set, you can hover your mouse cursor over a variable. The DataTip now appears, showing the name of the variable and the value it currently holds. You can also pin this DataTip as shown in Figure [3-31](#).



**Figure 3-31** Debugger DataTip

When you pin a DataTip, a pin will appear in the gutter next to the line number. You can now move this DataTip around to another position on the screen. If you look below the pin icon on the DataTip, you will see a “double down arrow” icon. If you click this, you can add a comment to your DataTip as illustrated in Figure [3-32](#).



**Figure 3-32** DataTip Comment

DataTips also allow you to edit the value of the variable, as long as the value isn’t read-only. To do this, simply select the value in the



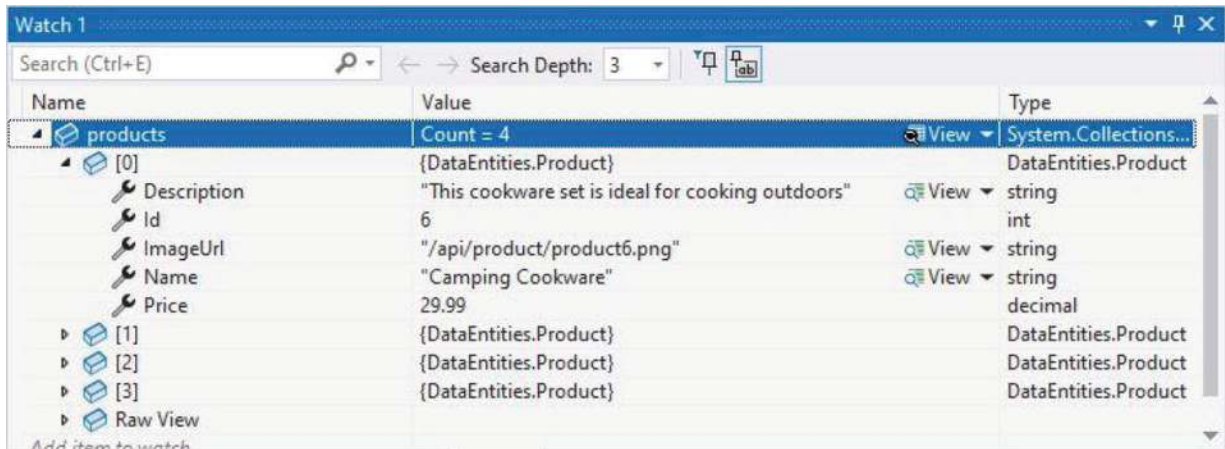
DataTip and enter a new value. Then press the Enter key to save the new value.

---

## Using the Watch Window

The Watch window allows us to keep track of the value of one or more variables and also allows us to see how these variable values change as one steps through the code.

You can easily add a variable to the Watch window by right-clicking the variable and selecting Add Watch from the context menu. Doing this with the `products` variable in the previous section will add it to the Watch 1 window as illustrated in Figure [3-33](#).



**Figure 3-33** The Watch 1 Window

Here, you can open the visualizer by clicking the magnifying glass icon or expanding the `products` variable to view the other properties of the object. I use the Watch window often as it is a convenient way to keep track of several variables at once.

---

## The DebuggerDisplay Attribute

In the previous section, we discussed how to add a variable to the Watch window in Visual Studio. We saw that we can view the value of a variable or variables easily from this single window. Looking back at the `products` variable in Figure [3-33](#), each item in the list displays as

`DataEntities.Product`. You need to expand each list item to view the property values.

This quickly becomes untenable when dealing with a list of hundreds of items and you are looking for a specific value.

This is where the `DebuggerDisplay` attribute comes into play. Because the `products` variable is a `List<Product>` we can modify the `Product` class as seen in Listing [3-2](#).

Ensure that you add the statement using `System.Diagnostics` to your code file.

By adding the `DebuggerDisplay` attribute you can control how a class or field is displayed in the Watch window.

```
[DebuggerDisplay("Product {Id} - {Name}")]
public sealed class Product
{
    [JsonPropertyName("id")]
    public int Id { get; set; }

    [JsonPropertyName("name")]
    public string? Name { get; set; }

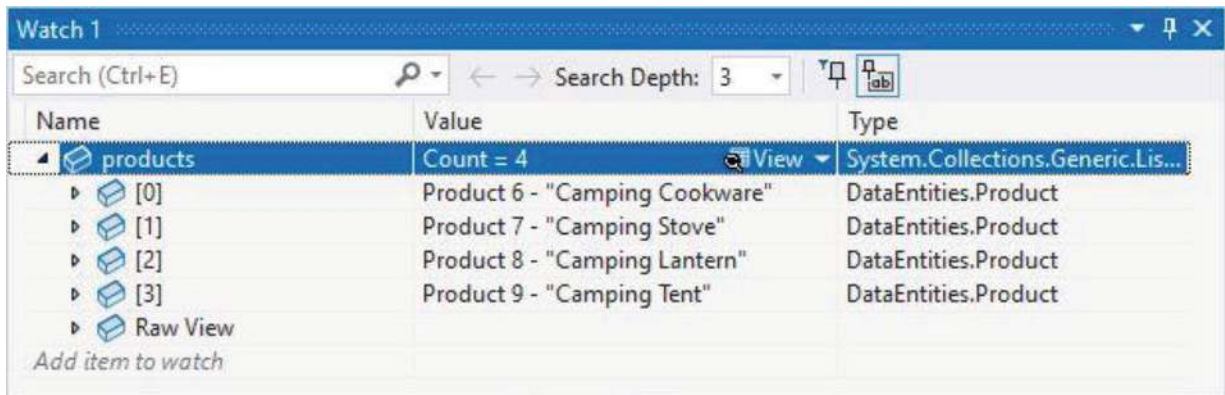
    [JsonPropertyName("description")]
    public string? Description { get; set; }

    [JsonPropertyName("price")]
    public decimal Price { get; set; }

    [JsonPropertyName("imageUrl")]
    public string? ImageUrl { get; set; }
}
```

**Listing 3-2** Modified Subject Class

Start debugging your code again and have a look at your Watch window after adding the `DebuggerDisplay` attribute. Your item values are more readable and easily identifiable, as seen in Figure [3-34](#).



**Figure 3-34** The `IstSubjects` Variable Values with `DebuggerDisplay`

You can also go a bit further by adding a format specifier to the attribute. The use of “nq” in the `DebuggerDisplay` attribute will remove the quotes when the final value is displayed. The “nq” means “no quotes.” To do this, modify the attribute as seen in [Figure 3-35](#).

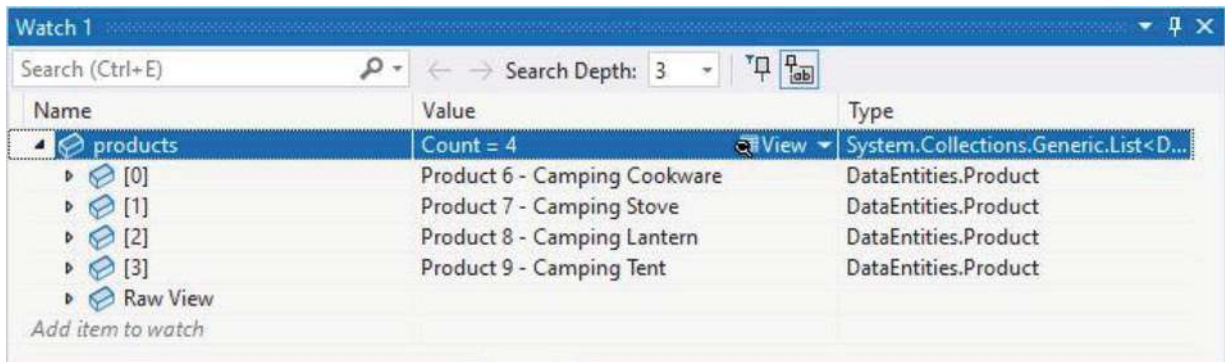
```
using System.Text.Json.Serialization;
using System.Diagnostics;

namespace DataEntities;

[DebuggerDisplay("Product {Id} - {Name,nq}")]
76 references
public sealed class Product
{
```

**Figure 3-35** Specifying no Quotes

Running your code again and viewing the output in your Watch window, you will notice the change to the `Name` property.



**Figure 3-36** The Name Property Without Quotes

This provides a cleaner output making the Watch window less noisy.

## Evaluate Functions Without Side Effects

While debugging an application, we probably do not want the state of the application to change because of an expression we are evaluating. It is, unfortunately, a fact that evaluating some expressions might cause side effects.

To illustrate this, have a look at the `Counter` class in Listing [3-3](#).

```
public class Counter
{
    public int Value { get; private set; } = 0;

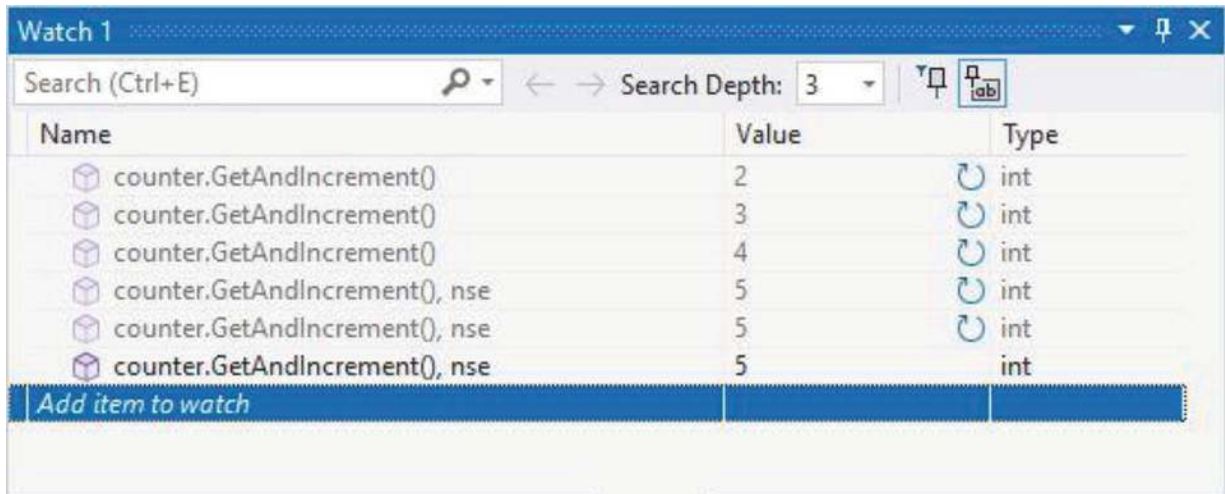
    public int GetAndIncrement()
    {
        Value++; // side effect: changes internal
state
        return Value;
    }
}
```

**Listing 3-3** The Counter Class

Here we see that the class contains a method called `GetAndIncrement()` that will cause changes to the internal state of the class.

When we place a breakpoint somewhere in our code after creating an instance of the `Counter` class and adding `counter.GetAndIncrement()` to the Watch window, you will notice that the value is incremented each time (as expected).

If, however, you add `counter.GetAndIncrement(), nse` to the Watch window, you will see that all future evaluations are suppressed (Figure 3-37).



**Figure 3-37** Suppress Future Evaluations with `nse`

When adding a side-effecting method call to the Watch window with, `nse` the method is still executed once, which is why you see the Value increment the first time you call `counter.GetAndIncrement(), nse`.

Side effects can cause all sorts of issues further down the debugging path, and sometimes the change might be so subtle that you don't even notice it. You could end up chasing "bugs" that never really were bugs to begin with.

As you have probably guessed by now, the `nse` added after the expression stands for "No Side Effects."

## Format Specifiers

Format specifiers allow you to control the format in which a value is displayed in the Watch window. Format specifiers can also be used in the Immediate and Command window. Using a format specifier is as easy as entering the variable expression and typing a comma followed

by the format specifier you want to use. The following are the C# format specifiers for the Visual Studio debugger.

***ac***

Force evaluation of an expression decimal integer

***d***

Decimal integer

***dynamic***

Displays the specified object using a Dynamic View

***h***

Hexadecimal integer

***nq***

String with no quotes

***nse***

Evaluates expressions without side effects where “nse” means “No Side Effects”

***hidden***

Displays all public and nonpublic members

***raw***

Displays item as it appears in the raw node. Valid on proxy objects only

***results***

Used with a variable that implements `IEnumerable` or `IEnumerable<T>`. Displays only members that contain the query result.

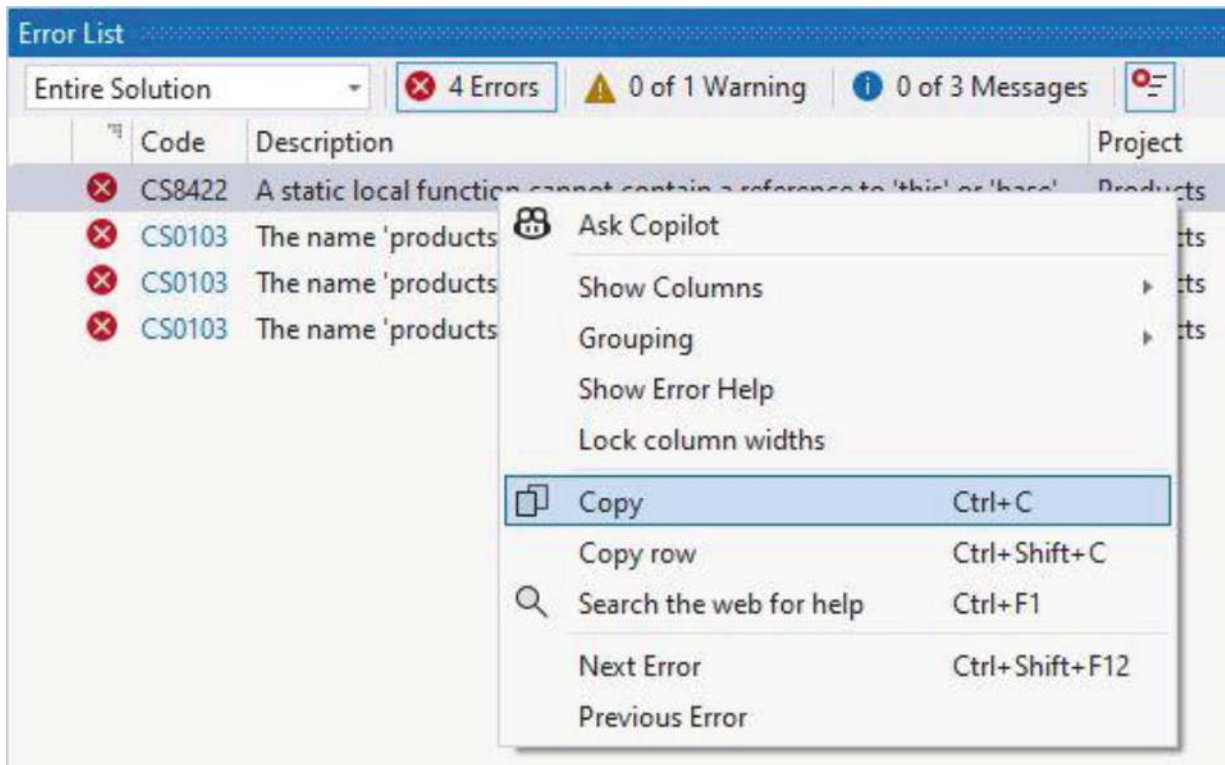
You will recall that we used the “nq” format specifier with the `DebuggerDisplay` attribute discussed in a previous section.



---

## Error Copying Improvements

Improvements have been made to copying errors from the Error List as displayed in Figure [3-38](#).



**Figure 3-38** Copy Error Description Only

When you right-click on an error, you can choose to copy just the error description or the entire row. Before this change, you would have copied detailed data that might not have been necessary. Now, you can copy just the description making it much easier when searching for solutions to the error online.

---

## Debugging with GitHub Copilot

GitHub Copilot is available to developers throughout the Visual Studio IDE. It isn't just for autocompleting code via code suggestions, it's your debugging wingman too. It understands runtime state, call stacks, and also suggests fixes targeting the error you're experiencing. This means

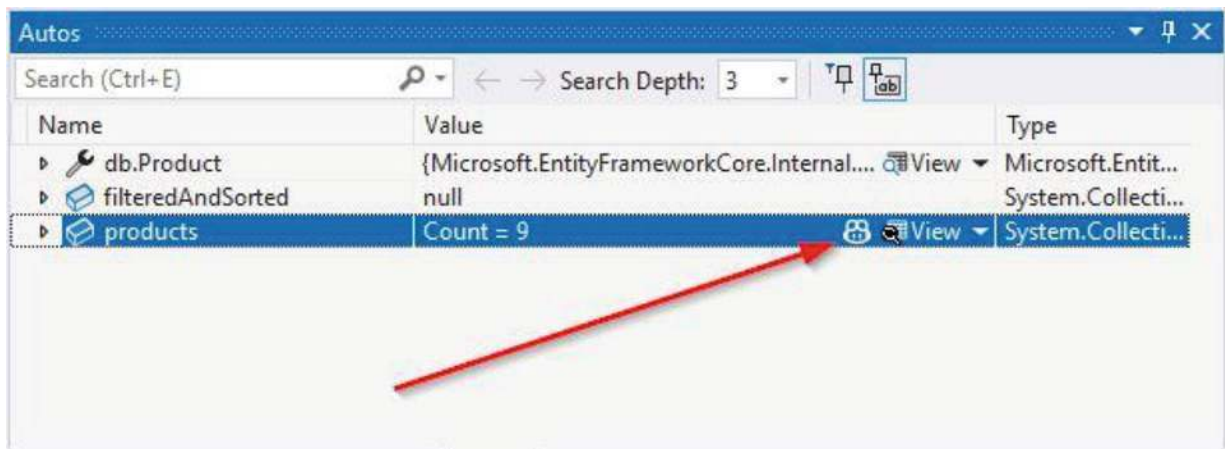
that when you interact with GitHub Copilot, you're interacting with a debugger-aware AI.

To get started with GitHub Copilot, see Chapter [1](#) of this book.

To find GitHub Copilot, just look for the Ask Copilot button.

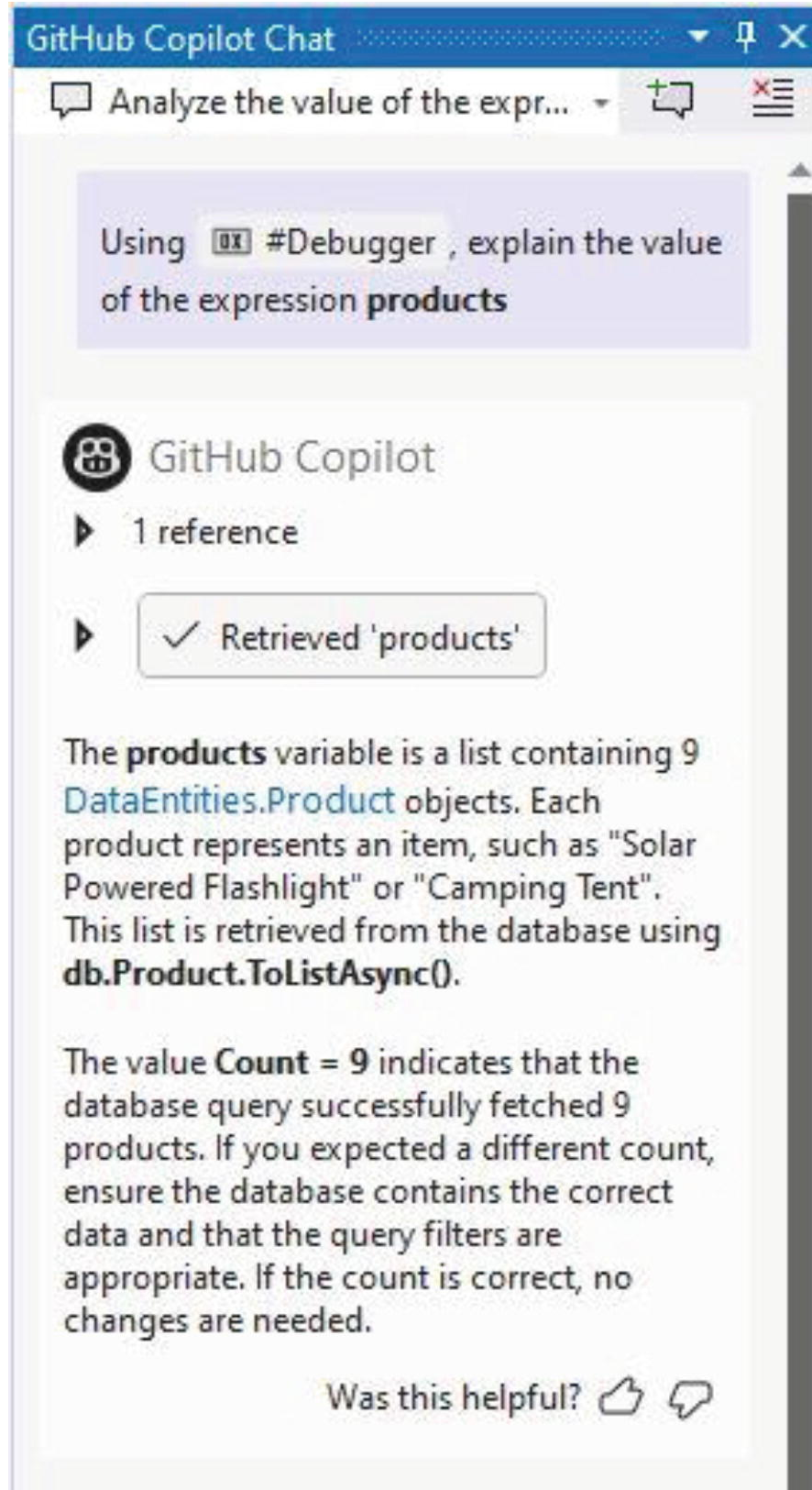
## Variable Analysis

When you hit a breakpoint in your code, you can use Copilot for variable analysis.



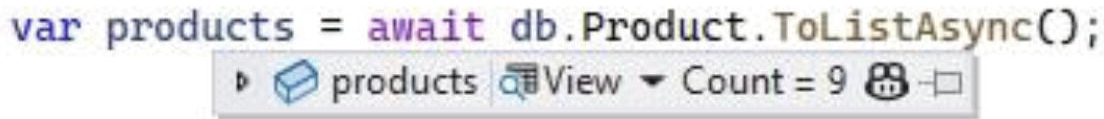
**Figure 3-39** Copilot in the Autos Window

When you view the Autos Window (Figure [3-39](#)), for example, and hover over a variable, you will see the Copilot button appear. Clicking on the button will invoke GitHub Copilot chat, and it will analyze the variable for you as shown in Figure [3-40](#).



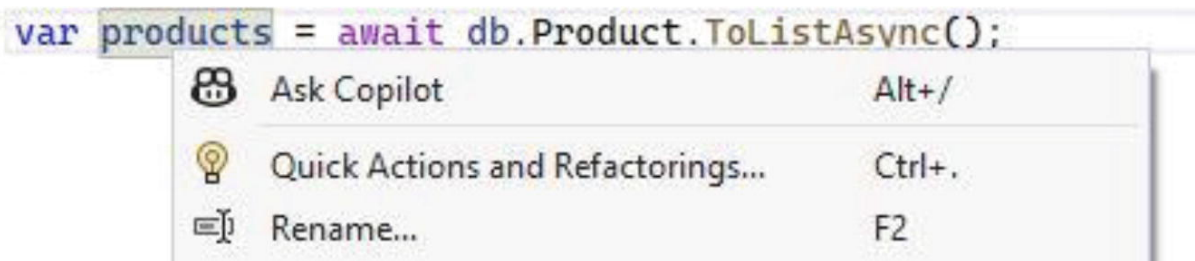
**Figure 3-40** Copilot Variable Analysis

You can also do the same thing in code. You can hover over a variable in your code, and on the DataTip that is displayed, you will see the GitHub Copilot button appear (Figure [3-41](#)).



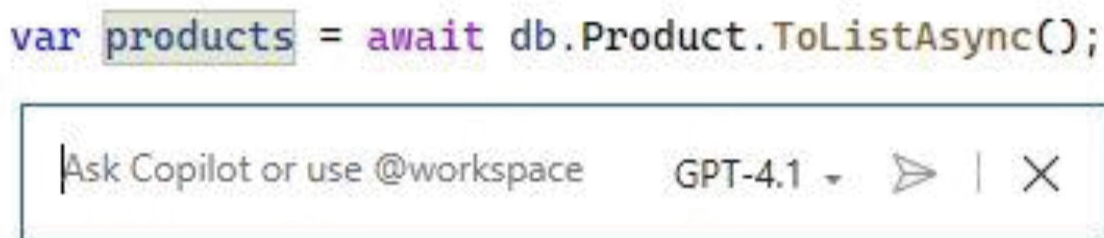
**Figure 3-41** Copilot When Hovering Over a Variable

Alternatively, you can right-click a variable in your code to bring up the context menu. The *Ask Copilot* button will be available to you as shown in Figure [3-42](#).



**Figure 3-42** Right-Click on a Variable

When you click on the Ask Copilot button, you will see that you can start an in-line chat with Copilot as shown in Figure [3-43](#).

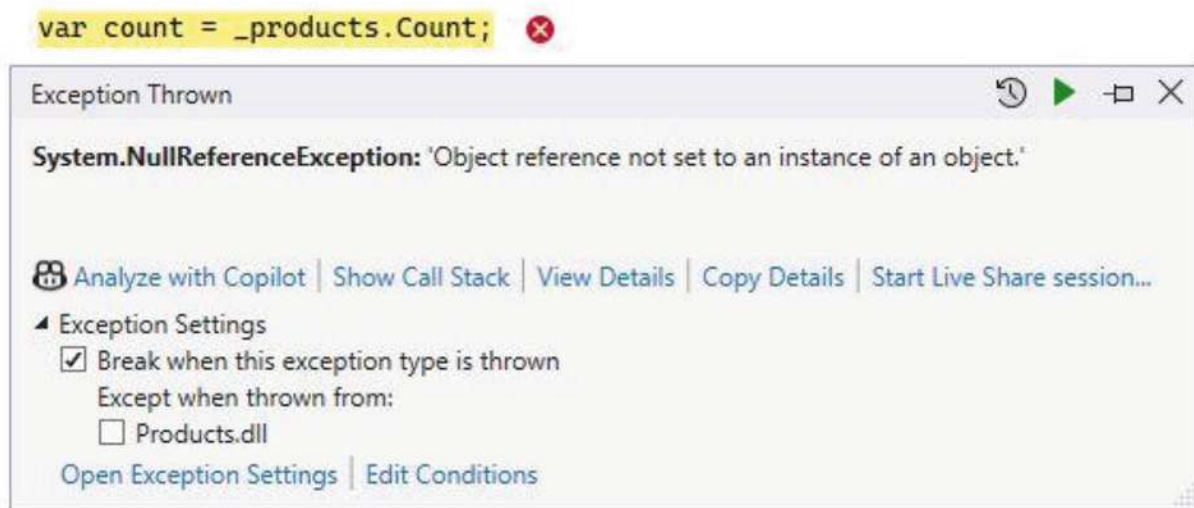


**Figure 3-43** Copilot Inline Chat

You can now ask Copilot for suggestions and fixes as you normally would when using the GitHub Copilot Chat window. This flexibility makes GitHub Copilot available to developers with less friction, allowing you to remain focused on the task at hand.

## Debugging an Exception

With GitHub Copilot, exception handling is enhanced through the use of context-aware, intelligent suggestions that help identify root causes and accelerate resolutions. As seen in Figure 3-44, we have a `NullReferenceException`, and the Copilot button is displayed in the Exception Thrown dialog as one of the first options to follow (next to Show Call Stack, View Details, Copy Details, and Start Live Share Session).



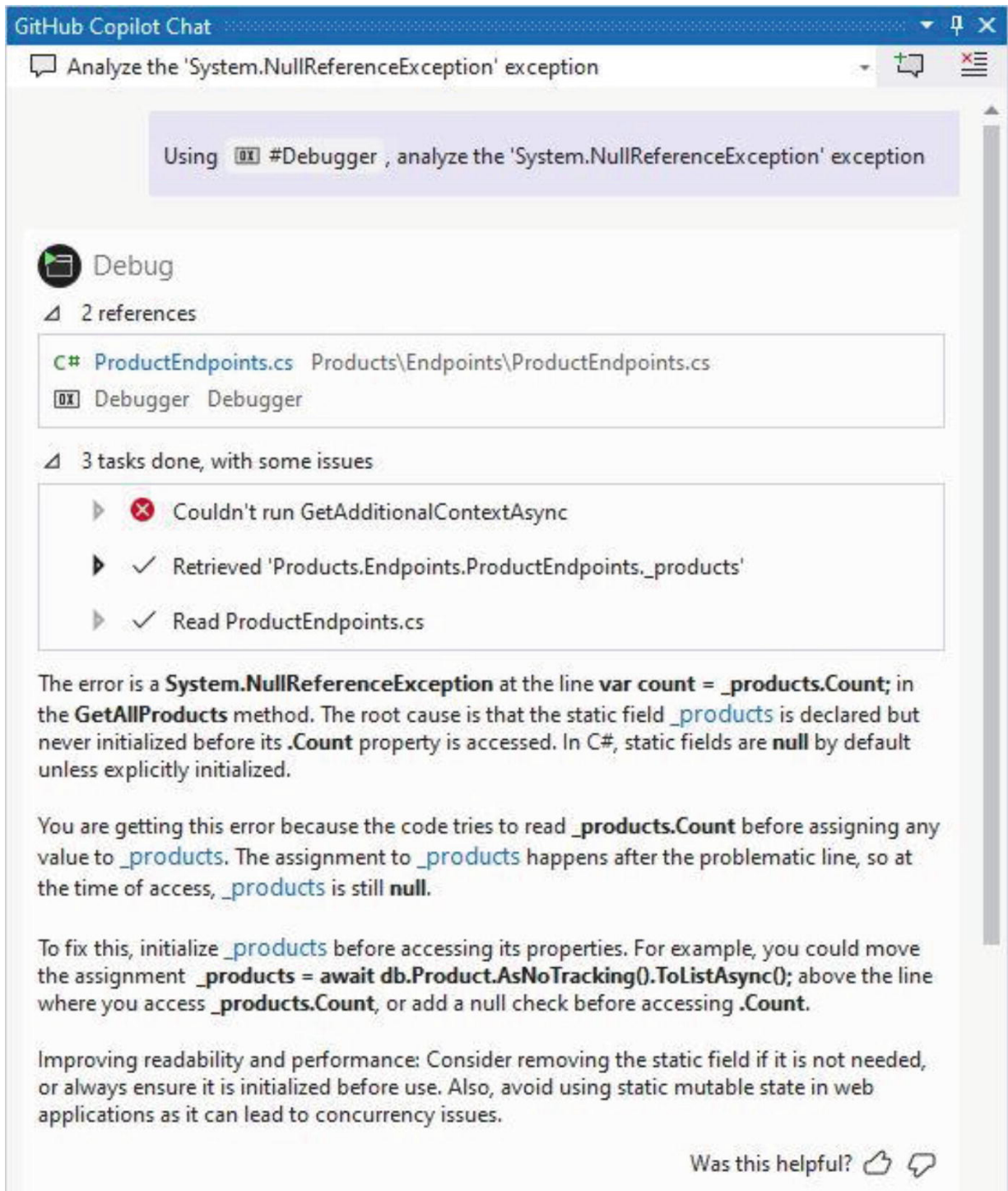
**Figure 3-44** `NullReferenceException` with Copilot

Clicking on the Copilot button will invoke a GitHub Copilot Chat session, with Copilot having context of the exception you are dealing with.

As seen in Figure 3-45, Copilot uses the debugger as a reference and then correlates the exception to the code context. Copilot doesn't just parrot the error message, but digs down to the root cause. It correctly identified that `_products` is `null` because it has not been initialized before being used.

What is interesting is that Copilot walks the call path backwards and detects that `_products` is a static field, explaining the implications of this in C#. It correctly states that static fields are `null` by default unless explicitly set. Copilot then suggests some remediation steps to resolve the error.





**Figure 3-45** Copilot Debugging the Error

Copilot understands the execution order of the code, the state of the code, and C# memory behavior. It also goes one step further and flags

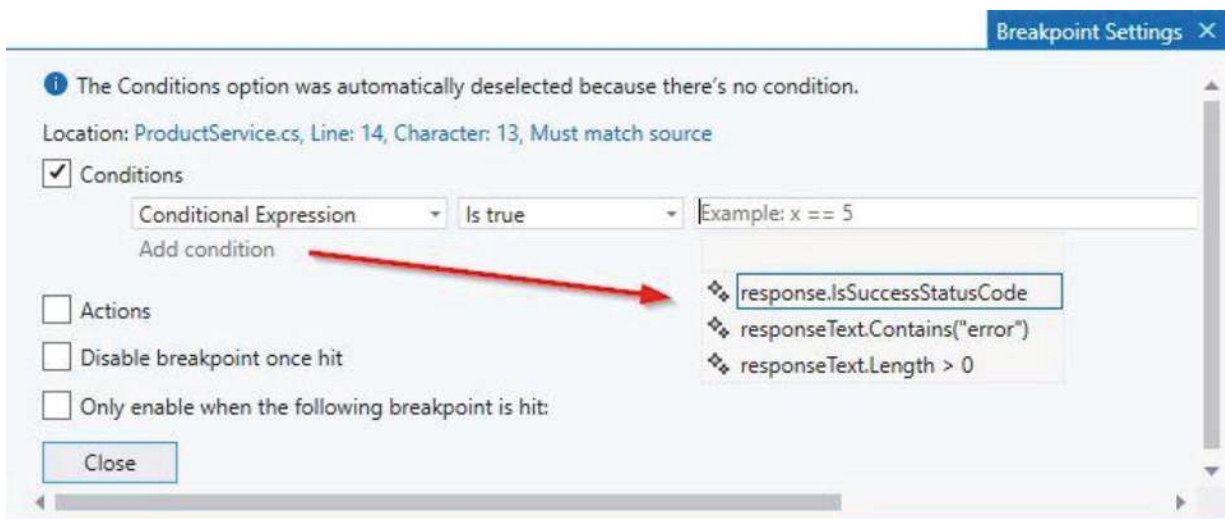


architectural concerns like the issues static mutable state causes in web applications.

While this is a contrived example, this is the kind of targeted response you can expect from Copilot. It is rooted in the context of your exception, saves time debugging and reduces cognitive load, especially when dealing with complex or obscure exceptions.

## Get Conditional Breakpoint Suggestions

Thinking back to when we discussed conditional breakpoints earlier in this chapter, you might have noticed Copilot's subtle yet powerful influence.



**Figure 3-46** Copilot and Conditional Breakpoints

Shown in Figure [3-46](#), when you click in the expression field, Copilot starts working by parsing the scope, inspecting nearby variables, and surfacing relevant suggestions for expressions that would make sense to use in your conditional breakpoint.

As you can see from Figure [3-46](#), the suggestions Copilot provided are for `response.IsSuccessStatusCode`, `responseText.Contains("error")`, and `responseText.Length > 0`. It did not suggest this randomly, it drew from symbols in context and aligned it with debugging patterns developers commonly use.

The suggestions Copilot provides become even more useful when you are dealing with nullable state, LINQ queries, Enum comparisons,

or Boolean patterns.

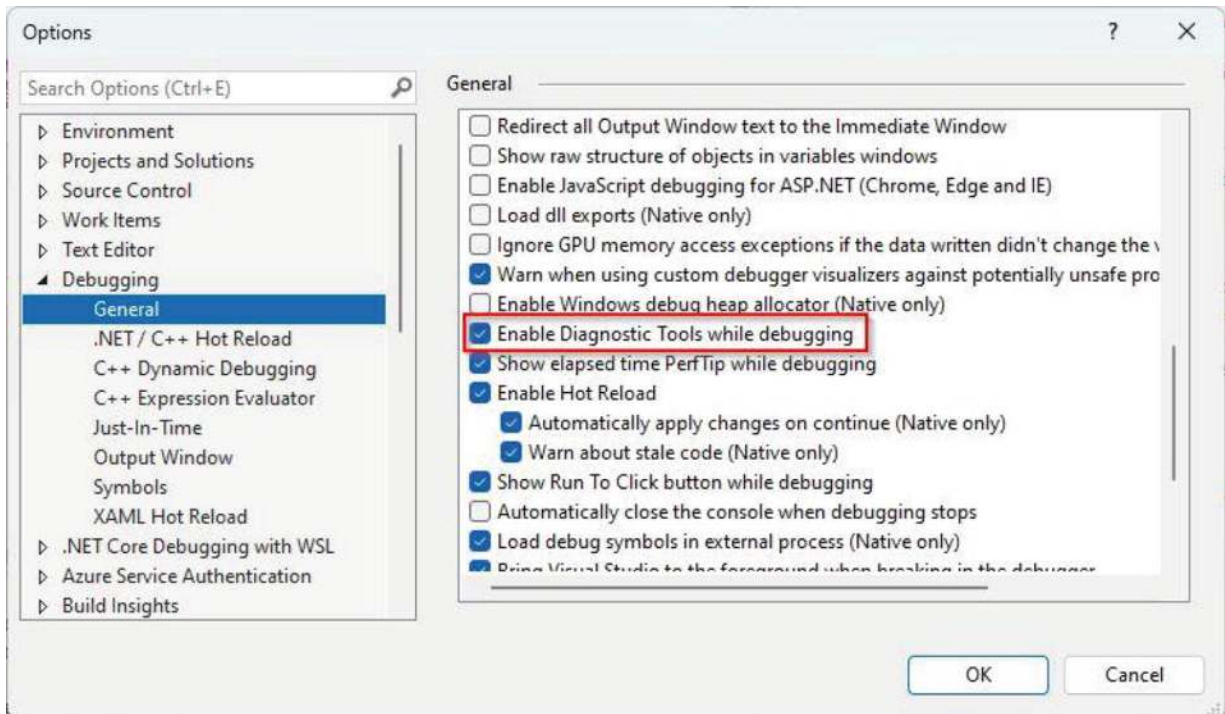
Accepting a suggestion with a simple selection, or using it as a baseline to refine further, helps keep you in that inner dev loop. As with most AI-generated suggestions (for now), these might not be perfect, but they offer a solid starting point, especially when dealing with non-trivial condition logic.

---

## Diagnostic Tools

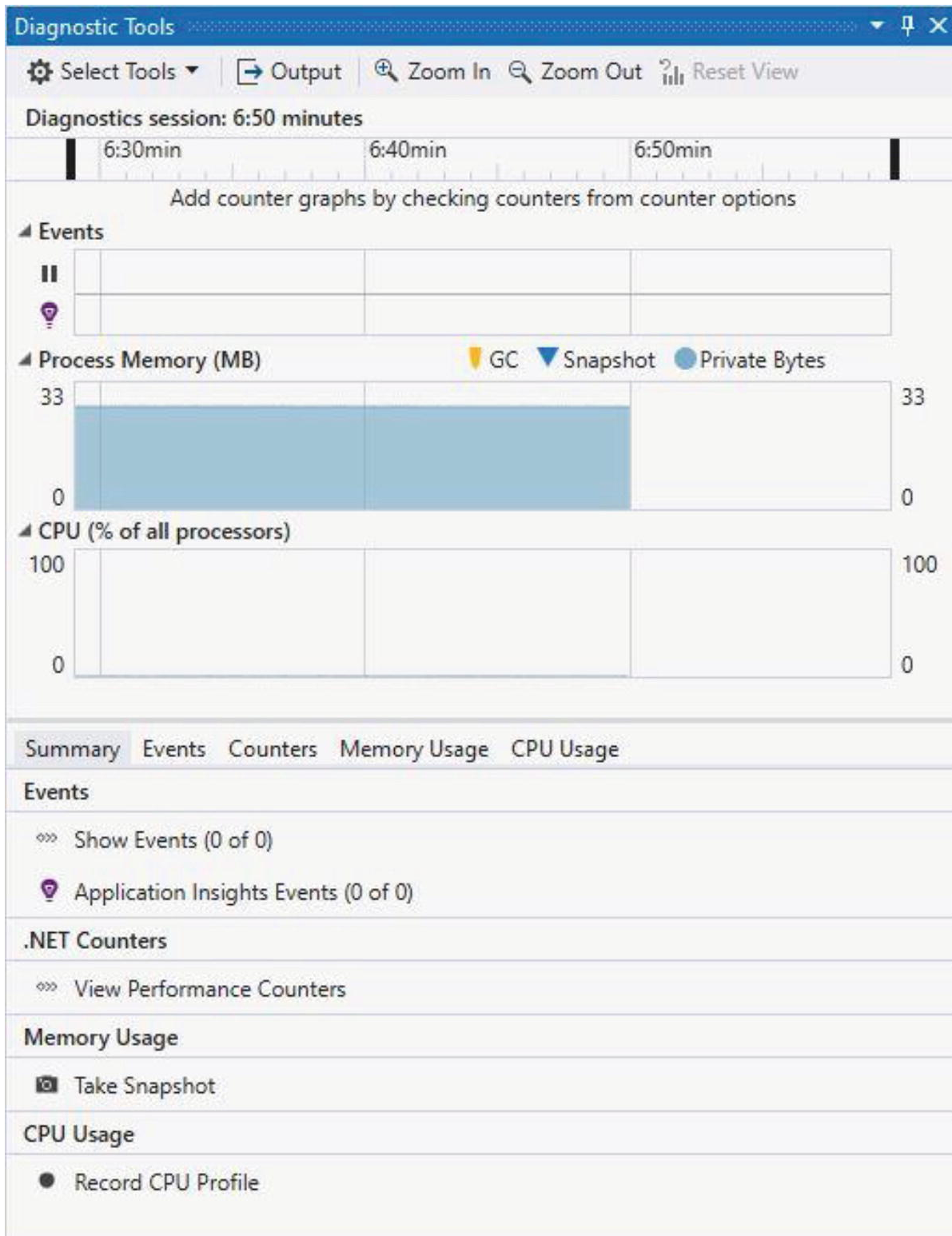
Performance profiling is no longer something that only concerns developers post-deployment. It is a first-class aspect of modern application development, and Visual Studio provides Diagnostic Tools to aid developers in analyzing application behavior during a debug session. Whether you are chasing a CPU spike, investigating memory pressure, or validating baseline performance, the Diagnostic Tools bring clarity to your application's runtime behavior directly inside the IDE.

Visual Studio Diagnostic Tools might be enabled by default. If not, enable Diagnostic Tools by going to the Tools menu and clicking Options ► Debugging, and then General. Ensure that Enable Diagnostic Tools while debugging is checked, as shown in Figure [3-47](#).



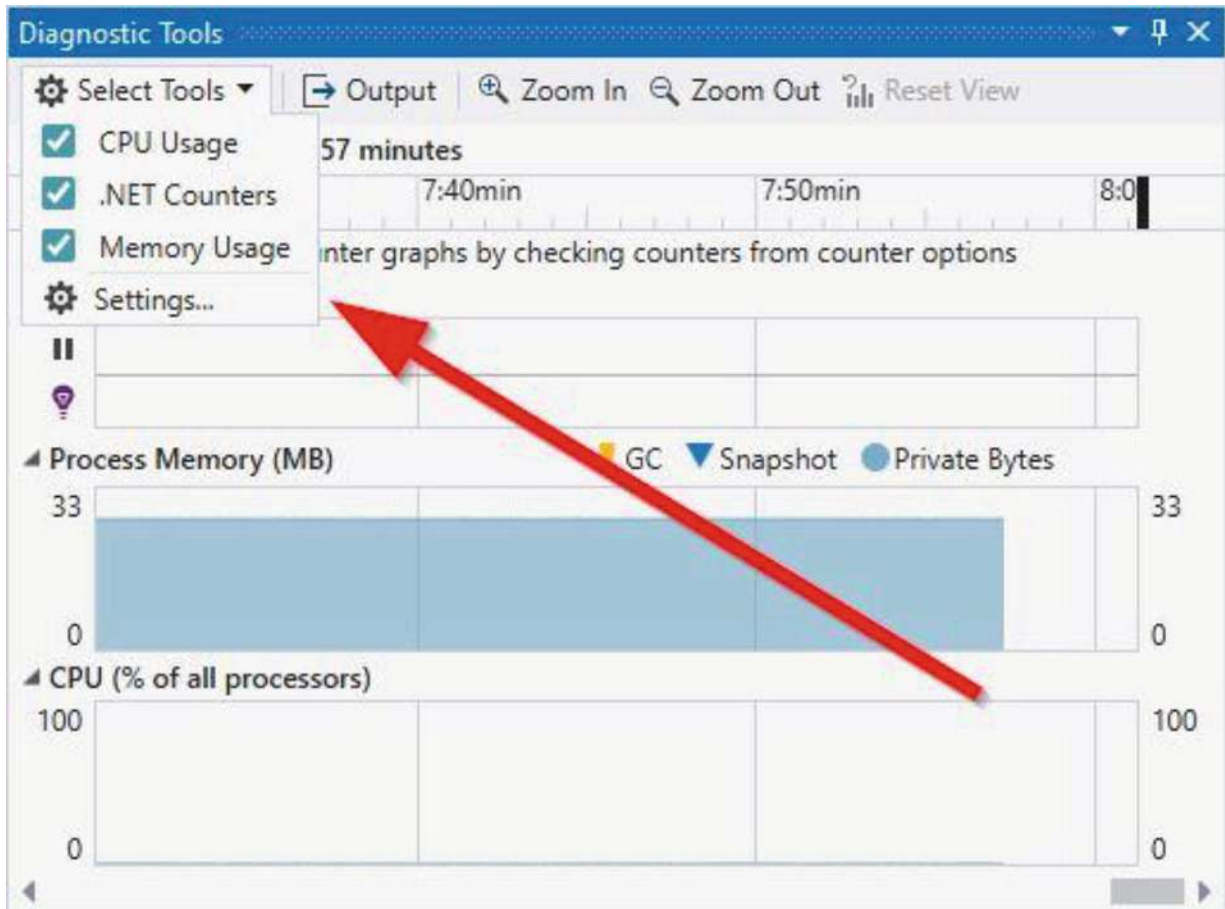
**Figure 3-47** Enable Diagnostic Tools

Once enabled, the Diagnostic Tools window (Figure [3-48](#)) will appear as a docked pane, typically beside the code editor. The window comprises multiple tabs, each surfacing different performance metrics. These range from event timelines and memory snapshots to CPU profiling and .NET runtime counters. Together, these tools make them ideal for detecting performance issues or regressions.



**Figure 3-48** Diagnostic Tools

At the top of the Diagnostic Tools window (Figure 3-49), you can select the performance aspect to analyze by clicking on the *Select Tools* dropdown. This allows you to toggle various counters such as CPU Usage, .NET Counters, and Memory Usage. These selections will feed into the graphical area and visualize trends over time, providing real-time feedback on the performance characteristics of your application as you interact with it.



**Figure 3-49** Select What to Analyze

Diagnostic Tools' superpower is its ability to correlate performance events with your code execution timeline. For example:

- Trends in memory usage can be aligned with garbage collection events or allocation spikes.
- Specific method calls or threads can be correlated against specific CPU utilization.

- .NET Counters can provide insights such as exception rates, Garbage Collection activity, and thread pool usage.

This correlation takes the guesswork out of what your app was doing when it hit 100% CPU or started aggressively allocating memory. Let's see in the upcoming sections how each of these tabs provides deeper granularity and understanding of your application's performance.

## CPU Usage

The CPU usage tab in the Diagnostic Tools window gives developers a real-time view into how much processor time their application is consuming. In a world where performance bottlenecks often hide behind otherwise functional code, being able to understand the CPU consumption of your application is not a luxury; it's essential in pinpointing where that time is being spent.

With Diagnostic Tools enabled, when you start a debug session, you can switch to the CPU Usage tab to begin recording. As your application runs, Visual Studio will sample stack traces to analyze where the CPU is spending time.

From the sample code used for this demonstration, you will clearly see that `SlowPrimeCheck(int)` is a hot path (Figure [3-50](#)). Without looking at the actual source code, the CPU usage tells us that this function dominates execution time. Top functions are ranked by Total CPU, and for each method, you get a breakdown of how much time was spent directly in that method. You can then double-click the function in the list. This will jump you to the definition in the source code where you can start optimizing your code. But Visual Studio goes a step further.



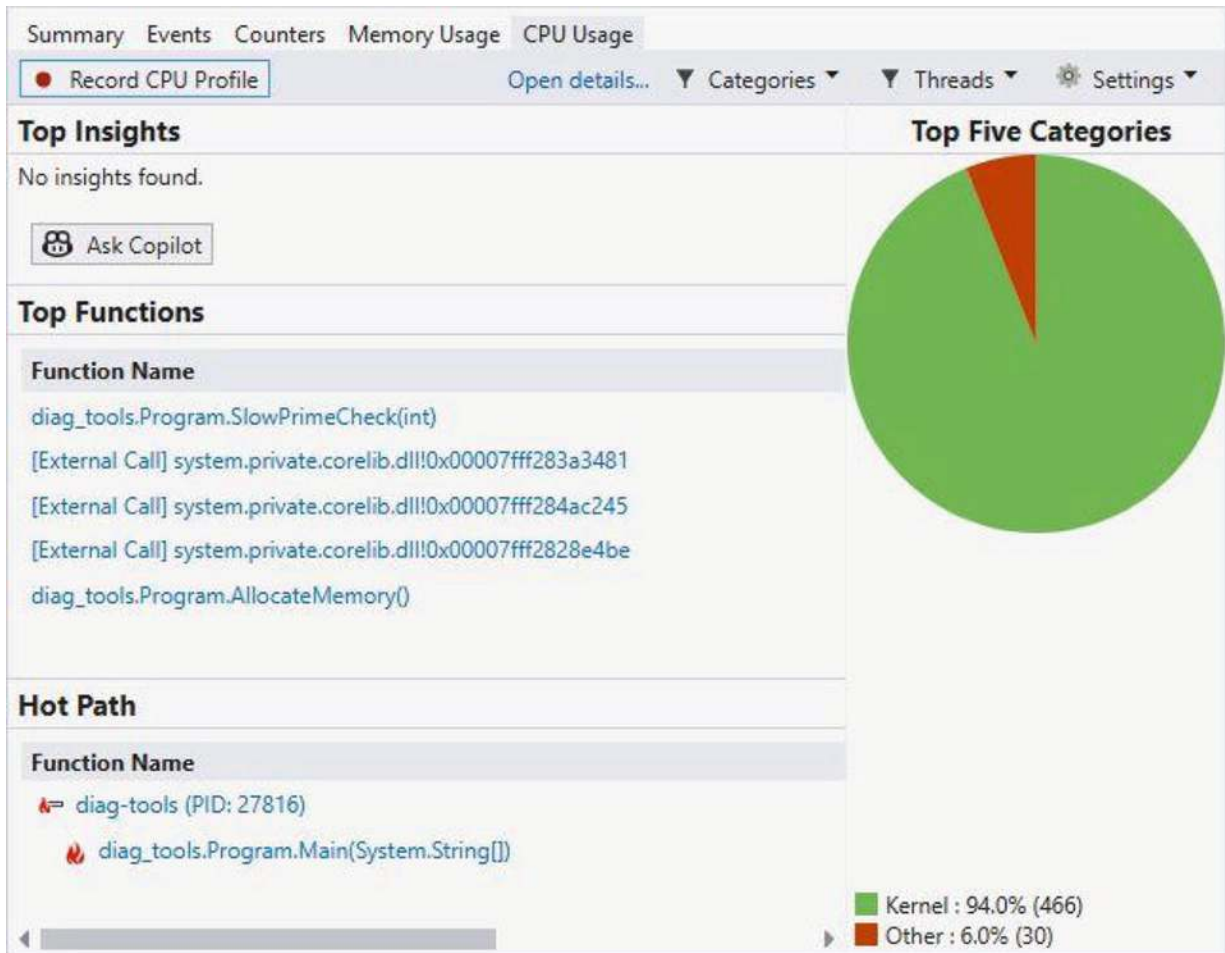


Figure 3-50 Viewing CPU Usage

## Get AI Assistance

When you analyze the CPU usage of your application, Visual Studio may surface AI-driven insights in the Top Insights section.

Copilot assistance is indicated in Figure [3-51](#) with arrows.

These insights are generated by analyzing the data collected and matching this data against known performance anti-patterns. If Visual Studio does not find any issues outright, you can invoke the AI help by clicking on the **Ask Copilot** button, as shown in Figure [3-51](#).

You will also see a button to **Analyze method with Copilot** in the source editor. This will let Copilot analyze the method and suggest optimizations based on the current profiling session.

In Figure 3-51, the methods `AllocateMemory()` and `SlowPrimeCheck()` are both excellent candidates for improvement.

The sample code in this chapter is deliberately engineered to be inefficient by design. It exists to trigger CPU load, memory usage, and runtime events so that the Diagnostic Tools in Visual Studio have something meaningful to report. You are encouraged to apply the diagnostic techniques illustrated here in your own production code and not to mistake the code illustrated here as production-ready.

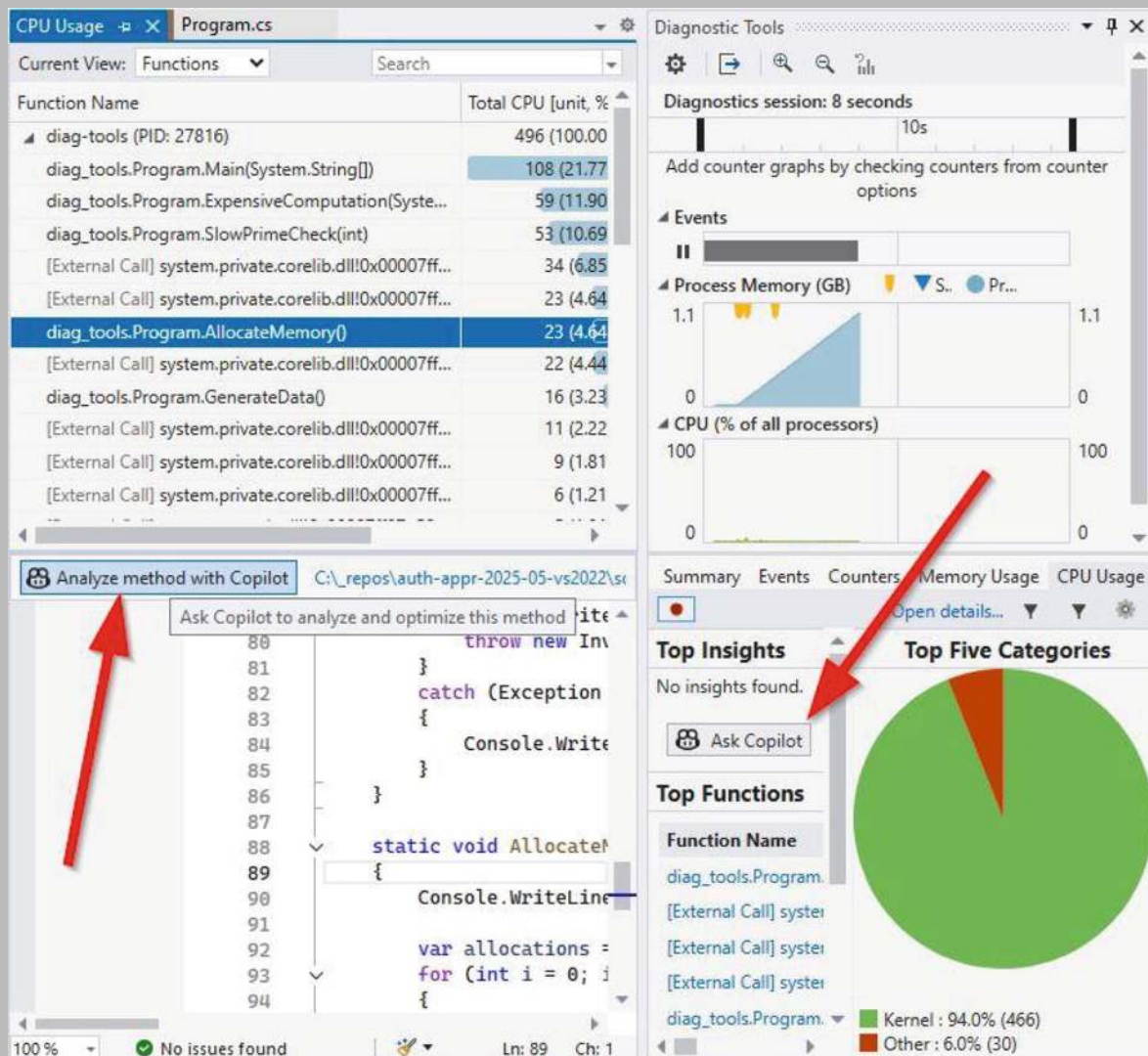


Figure 3-51 AI Assistance in CPU Insights

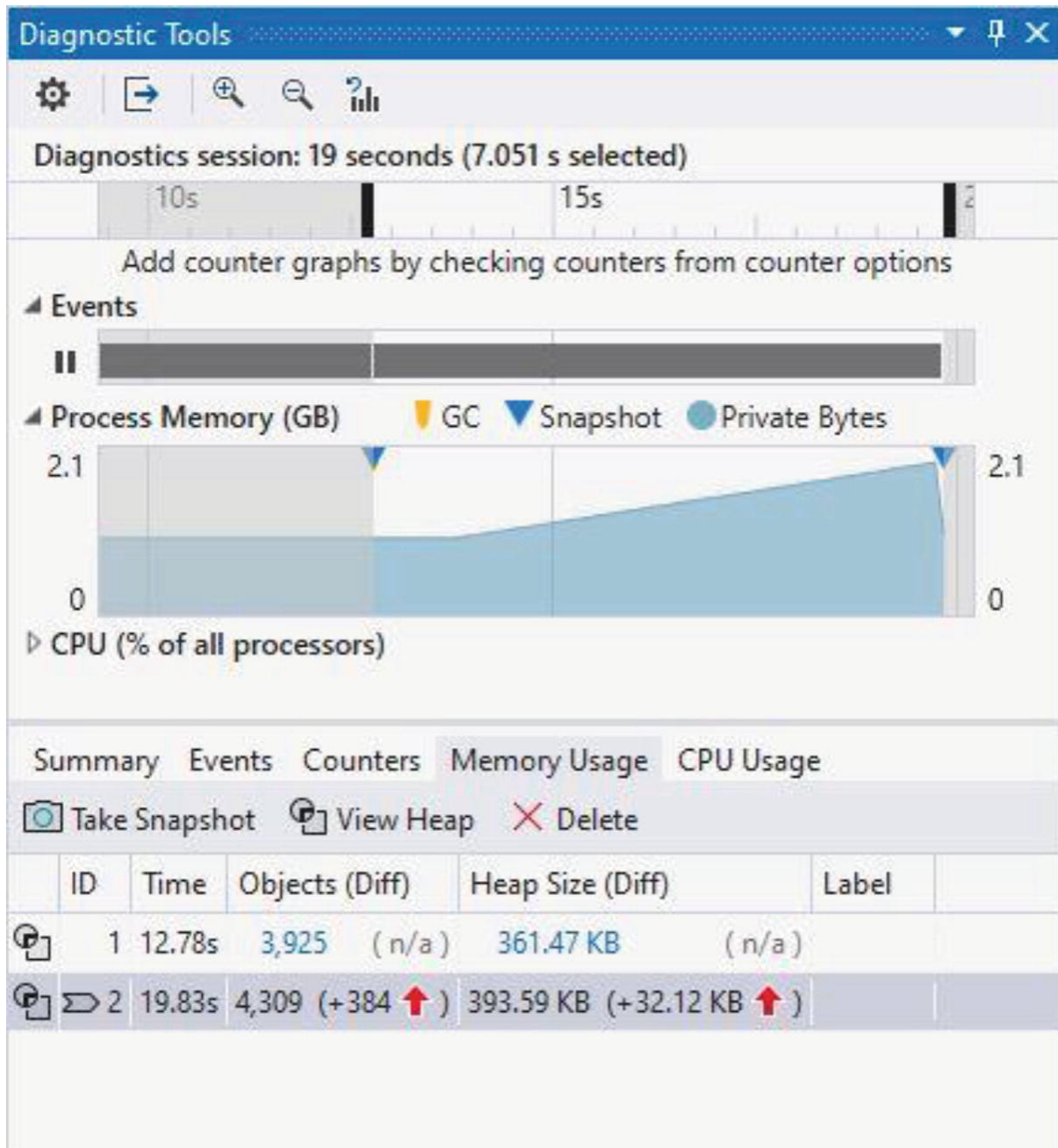
The level of detail that this assistive profiling offers is especially useful for large legacy codebases where trying to understand the intent behind every method is impractical. Copilot insights and suggestions here allow you to significantly shorten the diagnosis-to-fix cycle.

The CPU usage tool is not just a live view into your application's performance, it's a window into the runtime behaviour of your code. Combined with Copilot assistance, it becomes a powerful force for performance tuning and cleaning up technical debt.

## **Memory Usage**

Visual Studio Diagnostic Tools allows developers to see what the change in memory usage is. This is done by taking snapshots. When you start debugging, place a breakpoint on a method you suspect is causing a memory issue. Then you step over the method and place another breakpoint. An increase is indicated with a red up arrow as shown in Figure [3-52](#).

This is often the best way to analyze memory issues. Two snapshots will give you a nice diff and allow you to see exactly what has changed.



**Figure 3-52** Memory Usage Snapshots

This view is essential for identifying memory leaks, bloated object graphs, or inefficient allocations that creep in during development. I have taken two snapshots as illustrated in Figure 3-52. One snapshot was taken at the start of the workload and the other was taken after several memory-intensive operations. You will see that the number of objects has grown as well as the overall heap size.

## Managed Memory Analysis

Drilling into a snapshot will take you to one of Visual Studio's most powerful debugging tools. The Managed Memory View displays a table of object types, their instance counts, sizes etc.

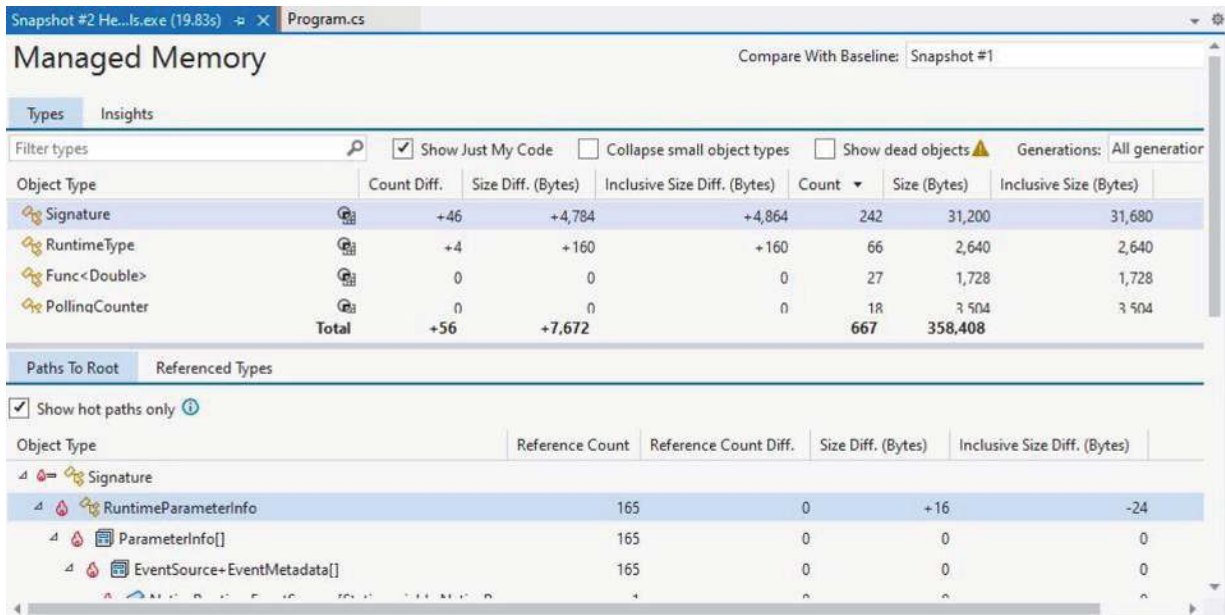


Figure 3-53 Managed Memory Analysis

As shown in Figure 3-53, when you drill into a memory snapshot to begin analyzing object types, you will often see internal .NET types such as:

- Signature
- RuntimeType
- PollingCounter

At first glance these would not resemble anything you have written in your code. They are, however, very real and relevant. Signature, for example, is an internal CLR type used by the runtime to represent method signatures, especially when reflection or dynamic calls are involved. RuntimeType represents a .NET type at runtime. You'll often see this when your code (or a class) is using `GetType()` or `typeof()`.

In short, these are normal object to see in your Managed Memory Analysis view. What matters is not that they exist, but whether they are



growing unexpectedly or staying around longer than they should be.

To leverage these insights, use the Paths To Root pane to check why these objects are still alive. Opening this pane for a selected object type, Visual Studio will display the chain of references keeping that object alive in memory. Think of this as a breadcrumb trail back to the source.

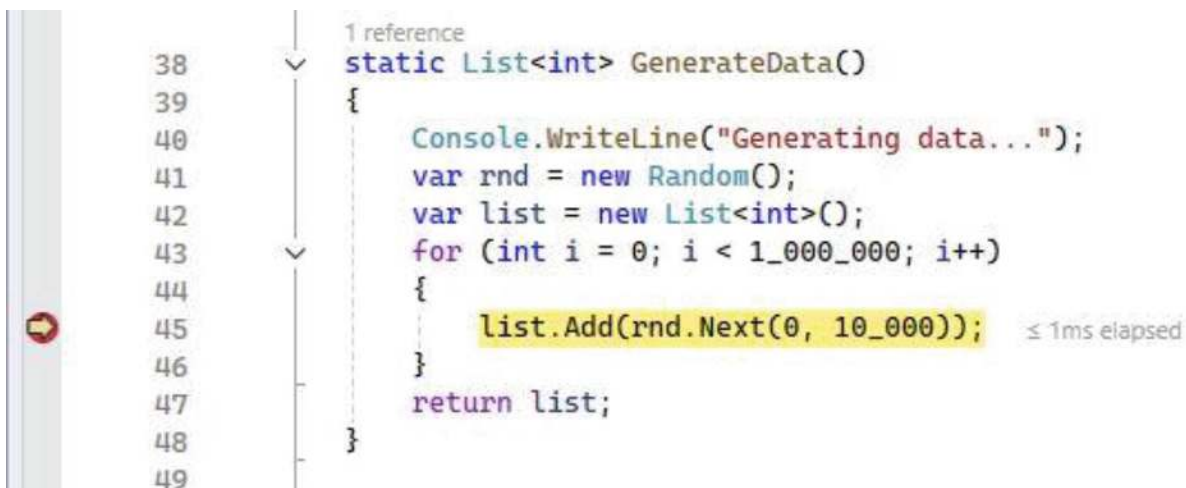
By following these root paths, you are able to gain an understanding of how memory is being used and retained in your application. You can easily spot unintended references, long-lived objects, or potential leaks. You gain insights that are often difficult to surface through code analysis alone. By creating a few snapshots and exploring a little, you can essentially turn raw memory data into actionable understanding. This is the power of the Managed Memory Analysis view.

---

## Immediate Window

The Immediate Window in Visual Studio allows you to debug and evaluate expressions, execute statements, and print the values of variables. If you don't see the Immediate Window, go to the Debug menu, select Windows, and click Immediate or hold down Ctrl+D, Ctrl+I.

Let's place a breakpoint in one of our `for` loops, as seen in Figure [3-54](#).

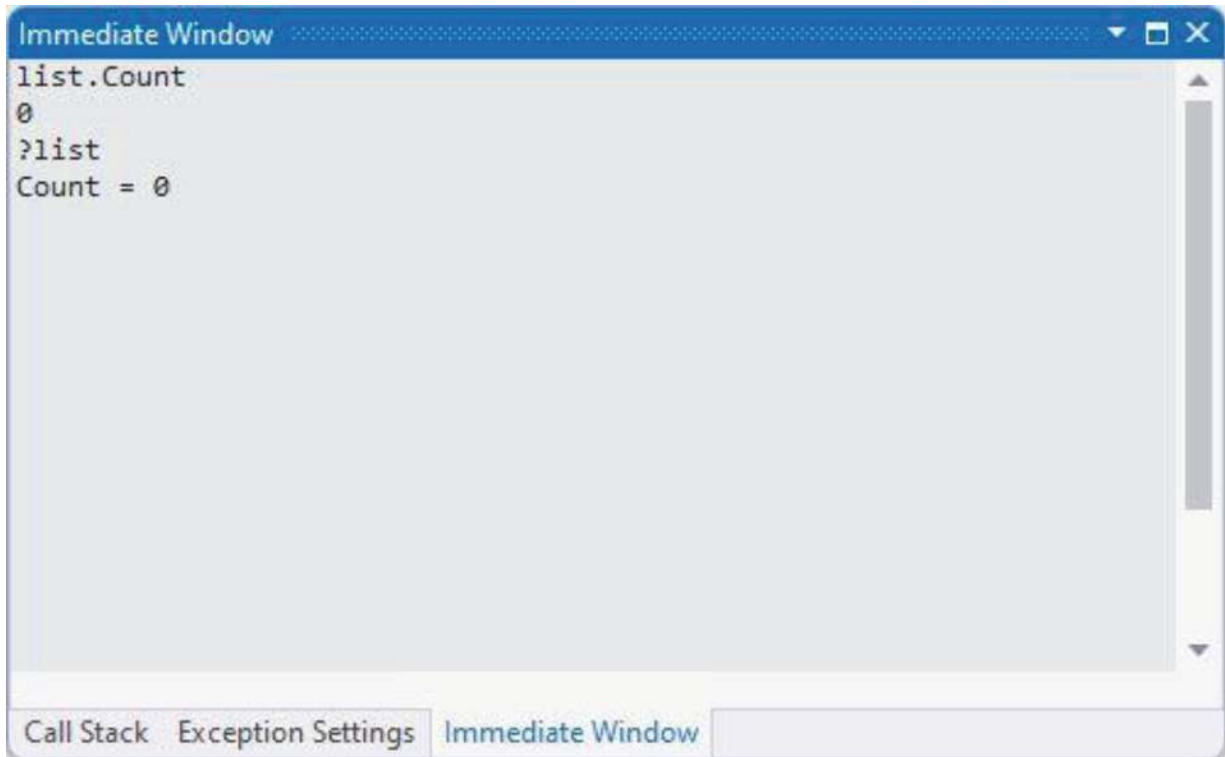


**Figure 3-54** Breakpoint Hit



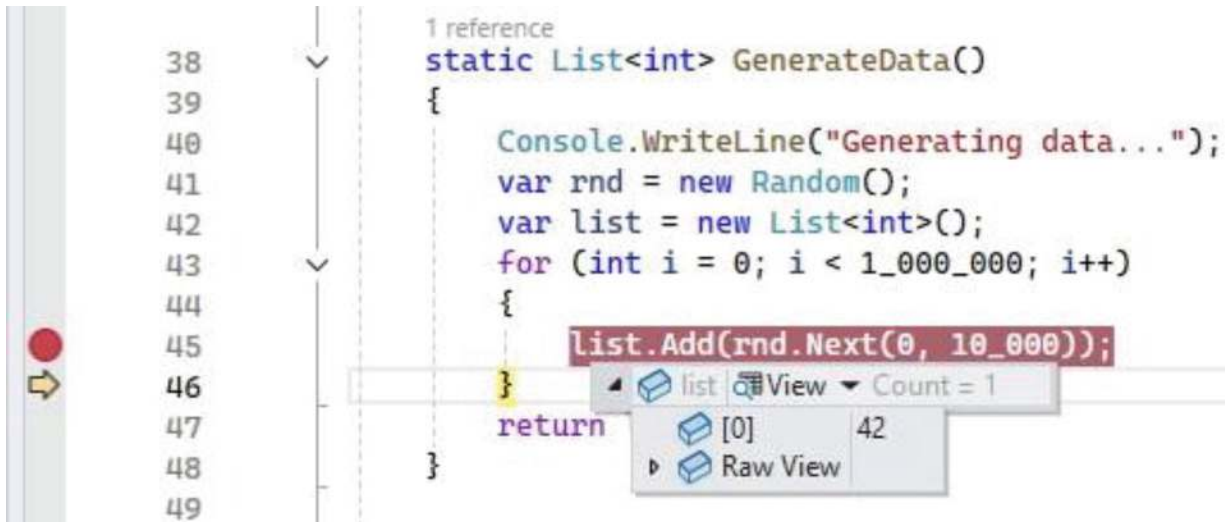
Opening up the Immediate Window and typing in `list.Count` will display its value as seen in Figure [3-55](#).

You can also use `?list` to view the value of the list.



**Figure 3-55** Immediate Window

If you step over the breakpoint and type `list[0] = 42` in the immediate window, you would be updating the value from whatever random value was assigned to the first index of the list, to the number 42 as seen in Figure [3-56](#).



**Figure 3-56** Variable Value Changed

You can also execute a function at design time (i.e., while not debugging) using the Immediate Window. The code in Listing [3-4](#) just outputs a simple string.

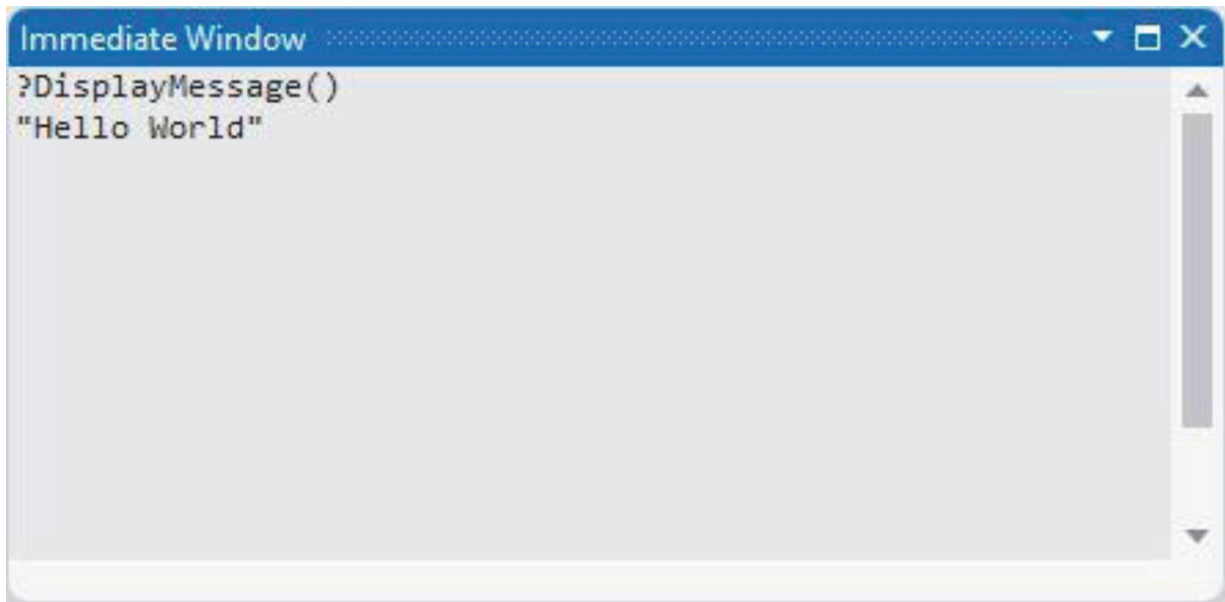
```

static string DisplayMessage()
{
    return "Hello World";
}

```

**Listing 3-4** DisplayMessage Function

In the Immediate Window, type `?DisplayMessage()` and press Enter. The Immediate Window will run the function and return the result seen in Figure [3-57](#).



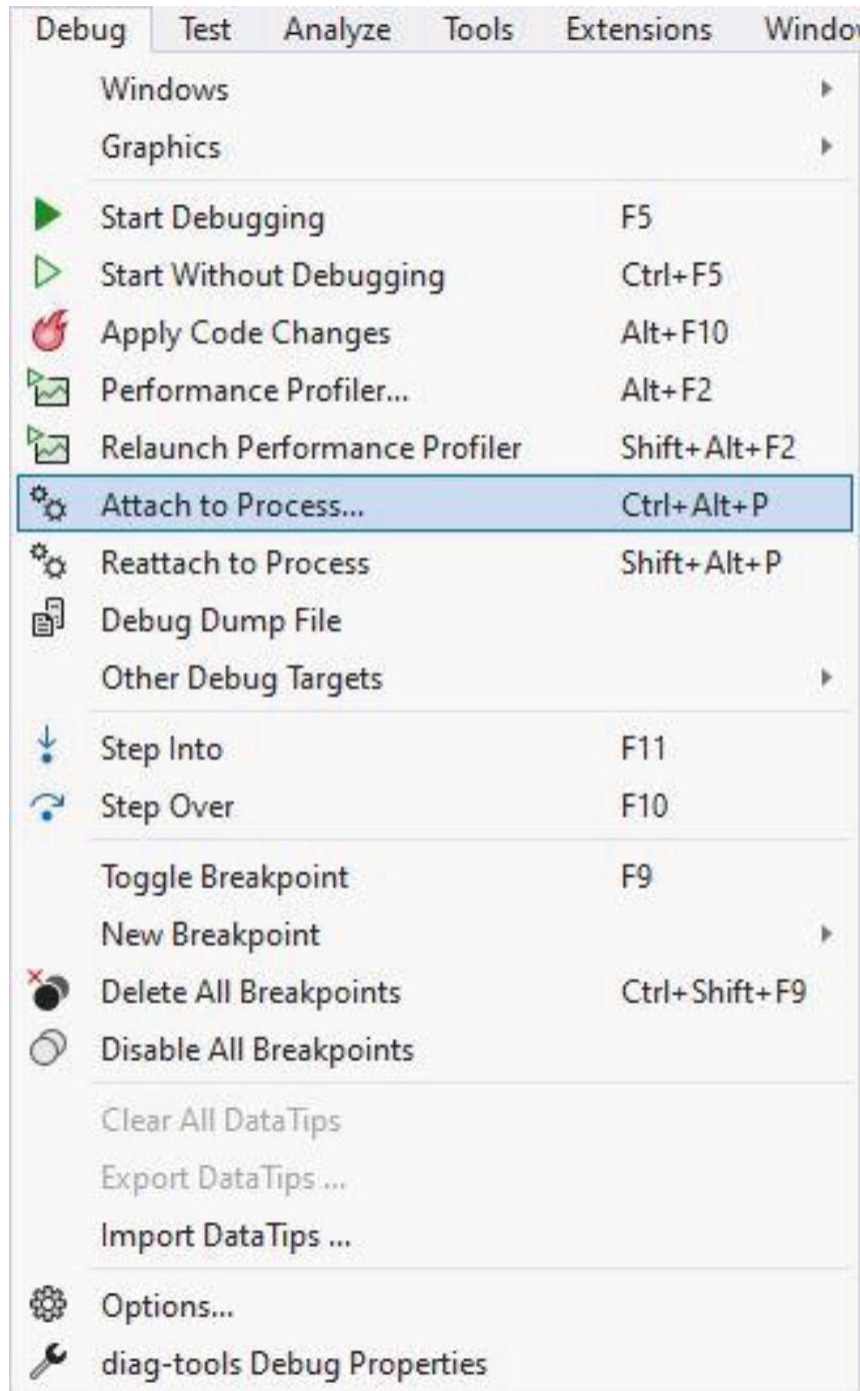
**Figure 3-57** Execute the DisplayMessage function

Any breakpoints contained in the function will break the execution at the breakpoint. Use the debugger to examine the program state.

---

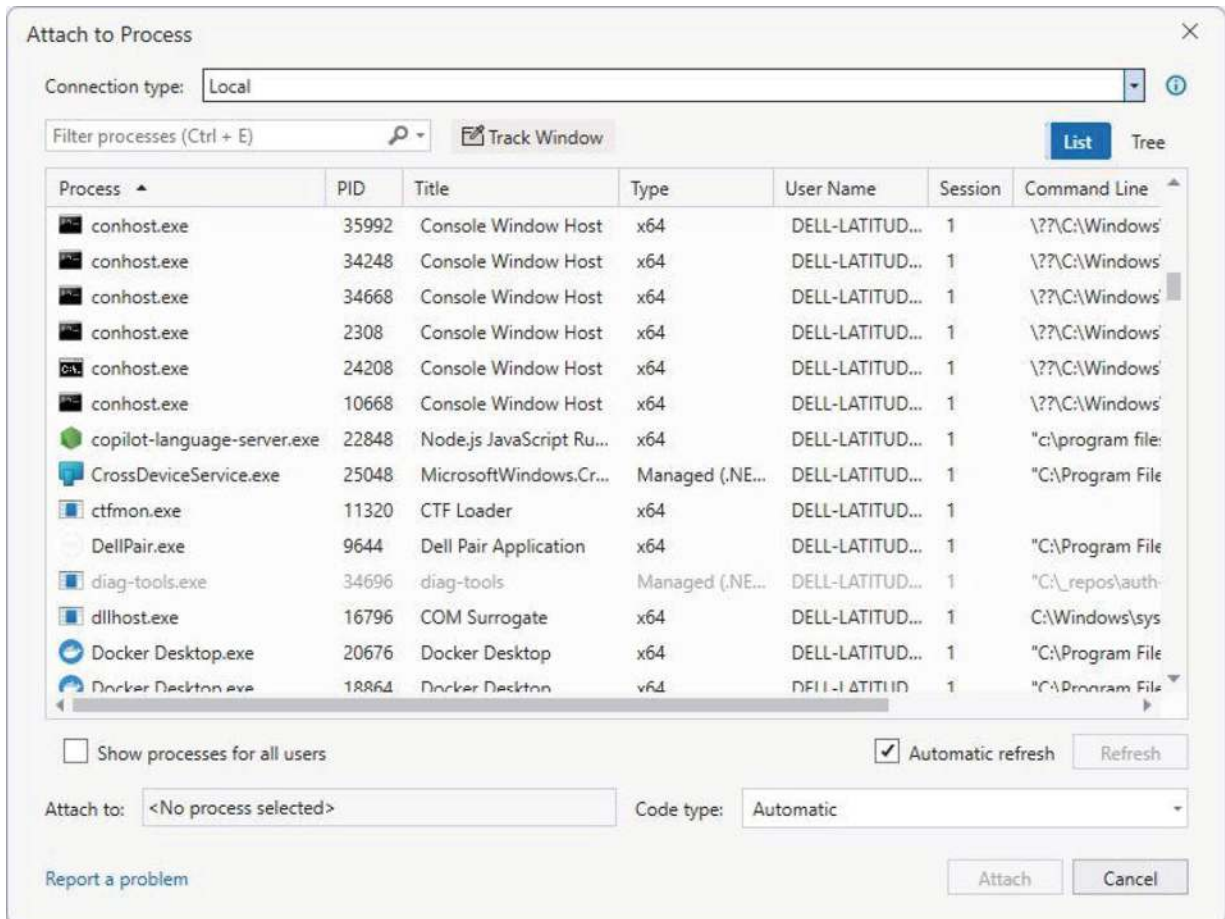
## Attaching to a Running Process

Attaching to a process allows the Visual Studio Debugger to attach to a running process on the local machine or a remote computer. With the process you want to debug already running, select Debug and click Attach to Process as seen in Figure [3-58](#). You can also hold down Ctrl+Alt+P to open the Attach to Process window.



**Figure 3-58** Attach to Process

The Attach to Process window is then displayed (Figure [3-59](#)). The connection type must be set to Local.



**Figure 3-59** Attach to Process

The available processes list allows you to select the process you want to attach to. You can quickly find the process you want by typing the name in the filter process text box.

You can, for example, attach to the `w3wp.exe` process to debug a web application running on IIS. To debug a C#, VB.NET, or C++ application on the local machine, you can use the Attach to Process by selecting the `<appname>.exe` from the available processes list (where `<appname>` is the name of your application).

## Attach to a Remote Process

To debug a process running on a remote computer, select Debug and click the Attach to Process menu, or hold down Ctrl+Alt+P to open the Attach to Process window. This time, change the Connection type to Remote and select the remote computer name in the Connection target

by selecting it from the drop-down list or typing the name in the connection target text box and pressing Enter.

If you are unable to connect to the remote computer using the computer name, use the IP and port address.

### ***Remote Debugger Port Assignments***

The port assignments for the Visual Studio Remote Debugger are as follows:

- Visual Studio 2022: 4026
- Visual Studio 2019: 4024
- Visual Studio 2017: 4022
- Visual Studio 2015: 4020
- Visual Studio 2013: 4018
- Visual Studio 2012: 4016

The port assigned to the Remote Debugger is incremented by two for each release of Visual Studio.

### **Reattaching to a Process**

You can quickly reattach to a process you previously attached to. To do this, you can click the Debug menu and select Reattach to Process or hold down Shift+Alt+P. The debugger will try to attach to the last process you attached to by matching the previous process ID to the list of running processes. If that fails, it tries to attach to a process by matching the name. If neither is successful, the Attach to Process window is displayed and lets you select the correct process. The option to reattach to a process you previously attached will only be available if you had previously attached to it.

---

## **Remote Debugging**

Sometimes, you need to debug an application that has already been deployed to a different computer. Visual Studio allows you to do this via remote debugging. To start, you need to download and install remote tools for Visual Studio 2022 on the remote computer.

Remote tools for Visual Studio 2022 enables app deployment, remote debugging, testing, profiling, and unit testing on computers that



don't have Visual Studio 2022 installed.

## **System Requirements**

The supported operating systems for the remote computer must be one of the following:

- Windows 11
- Windows 10 (not phone)
- Windows 8 or 8.1 (not phone)
- Windows 7 SP 1
- Windows Server 2016
- Windows Server 2012 or Windows Server 2012 R2
- Windows Server 2008 SP 2, Windows Server 2008 R2 Service Pack 1

The supported hardware configurations to enable remote debugging are detailed in the following list:

- 1.6 GHz or faster processor
- 1 GB of RAM (1.5 GB if running on a VM)
- 1 GB of available hard disk space
- 5400 RPM hard drive
- DirectX 9-capable video card running at 1024 x 768 or higher display resolution

The remote computer and your local machine (the machine containing Visual Studio) must both be connected over a network, workgroup, or homegroup. The two machines can also be connected directly via an Ethernet cable.

Take note that trying to debug two computers connected through a proxy is not supported.

It is also not recommended to debug via a dial-up connection (do those still exist?) or over the Internet across geographical locations. The high latency or low bandwidth will make debugging unacceptably slow.

## **Download and Install Remote Tools**

Connect to the remote machine and download and install the correct version of the remote tools required for your version of Visual Studio. The link to download the remote tools compatible with all versions of Visual Studio 2022 is

<https://visualstudio.microsoft.com/downloads#remote-tools-for-visual-studio-2022>.

If you are using Visual Studio 2017, for example, download the latest update of remote tools for Visual Studio 2017.

Also, be sure to download the remote tools with the same architecture as the remote computer. This means that even if your app is a 32-bit application, and your remote computer is running a 64-bit operating system, download the 64-bit version of the remote tools.

Install the remote tools and click Install after agreeing to the license terms and conditions (Figure 3-60).

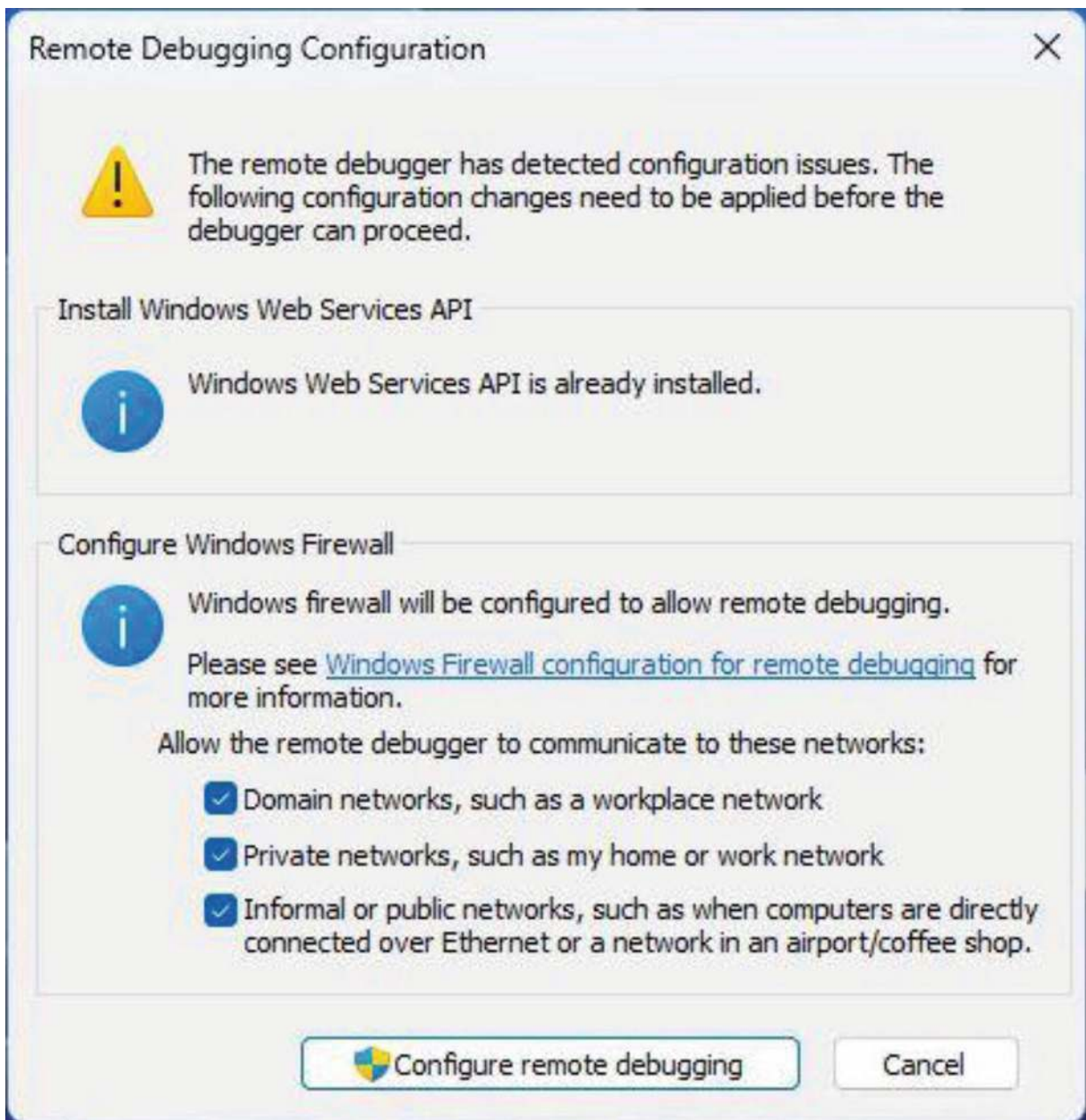


**Figure 3-60** Remote Tools for Visual Studio 2022

## Running Remote Tools

After the installation has been completed on the remote machine, run the Remote Debugger application as Administrator if you can. To do this, right-click the Remote Debugger app and click Run as Administrator.

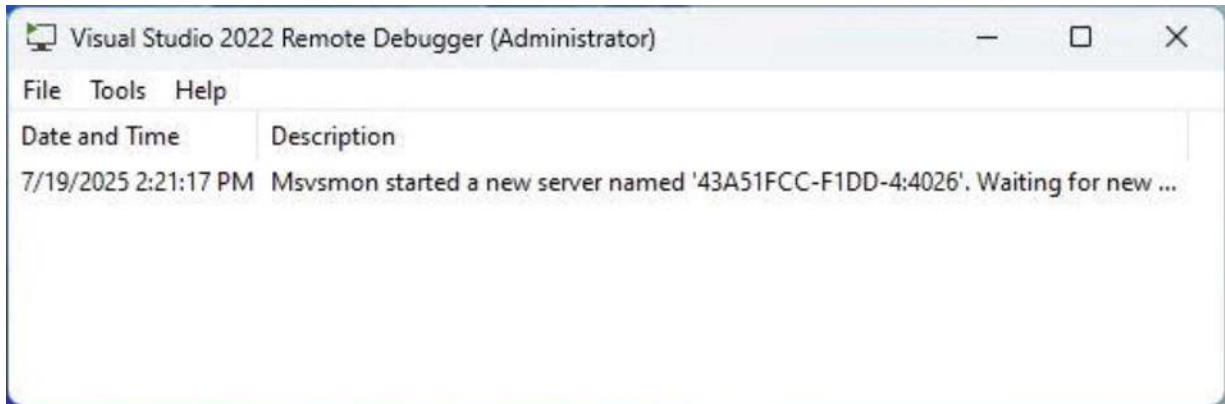
At this point, you might be presented with a Remote Debugging Configuration dialog box as seen in Figure 3-61.



**Figure 3-61** Remote Debugging Configuration

If you encounter this window, it possibly means that there is a configuration issue that you need to resolve. The Remote Debugging Configuration dialog box will prompt you to correct configuration errors it picks up. Do this by clicking the Configure remote debugging button.

You will then see the Remote Debugger window shown in Figure [3-62](#).



**Figure 3-62** Visual Studio 2022 Remote Debugger

You are now ready to start remote debugging your application.

## Start Remote Debugging

The great thing about the Remote Debugger on the remote computer is that it tells you the server name to connect to. In Figure [3-62](#), you can see that the server is named `43A51FCC-F1DD-4:4026` where `4026` is the port assignment for Visual Studio 2022. Make a note of this server name and port number.

In your application, set a breakpoint somewhere in the code. Now right-click the project in the Solution Explorer and click Properties. The project properties page opens as seen in Figure [3-63](#).

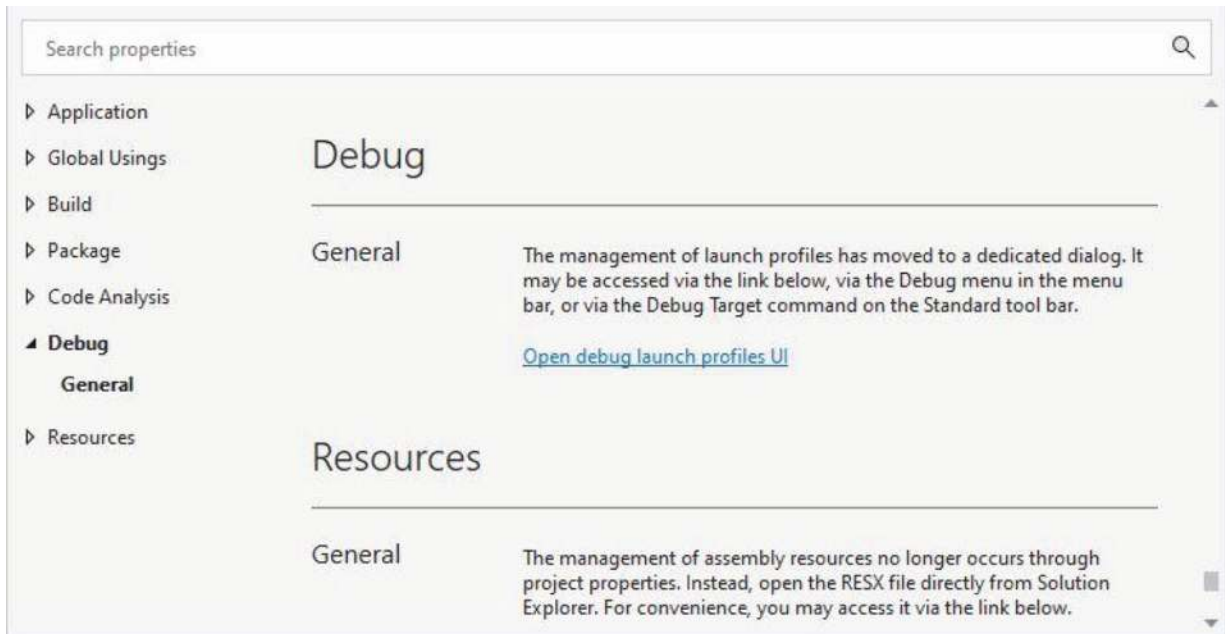


Figure 3-63 Project Properties Page

Next, you need to click on the **Open debug launch profiles UI** link as the launch profiles have moved to a dedicated screen, as seen in Figure [3-64](#).

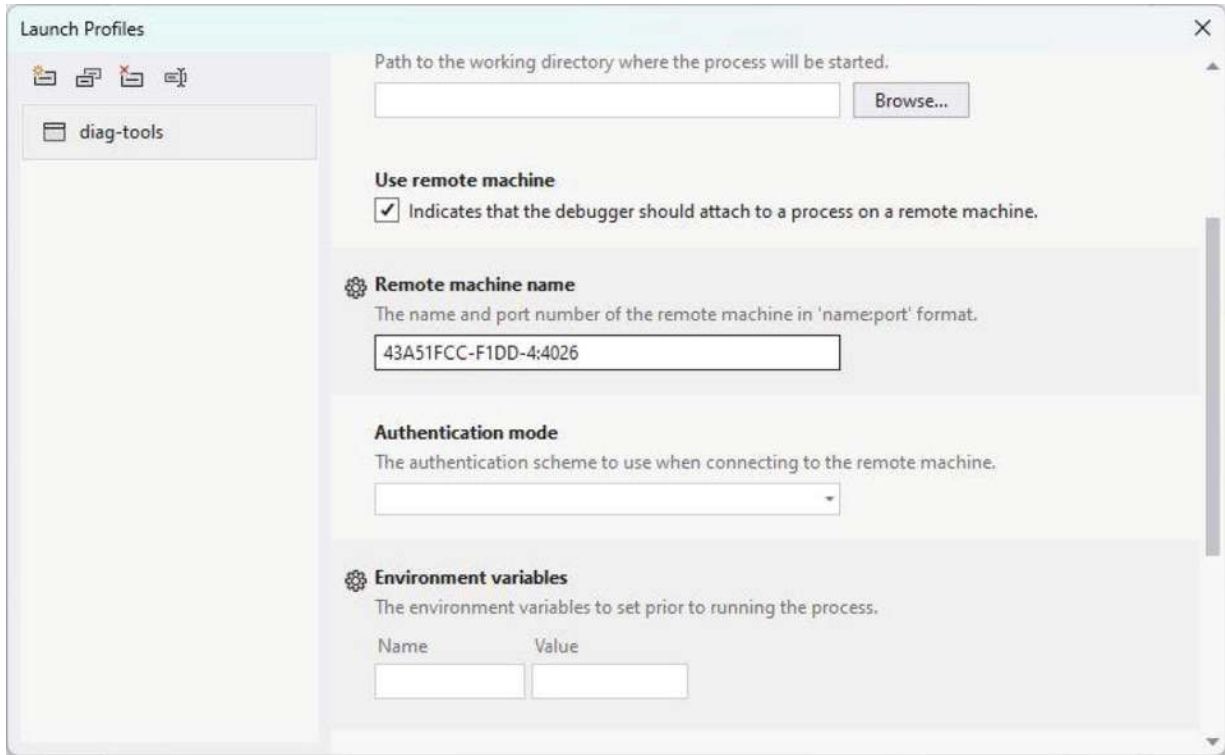


Figure 3-64 Launch Profiles Dialog

Now, perform the following steps to remotely debug your application:

1. Check the Use remote machine checkbox, and enter the remote machine name and port noted earlier. In our example, this is 43A51FCC-F1DD-4:4026.
2. Make sure that you leave the Working directory text box empty and do not check Enable native code debugging.
3. When all this is done, close the properties and build your project.
4. You now need to create a folder on the remote computer that is the same path as the Debug folder on your local machine (the Visual Studio machine). For example, the path to the project Debug folder on my local machine is `<source path>DiagnosticToolsDemo\diag-tools\bin\Debug`. Create this same path on the remote machine.
5. Copy the executable that was just created by the build you performed in step 3 to the newly created Debug folder on the remote computer.

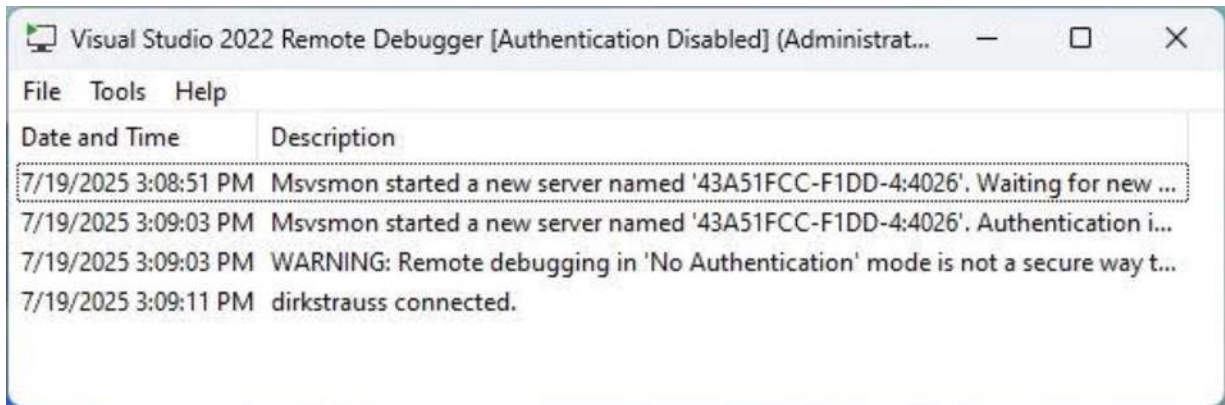
Be aware that any changes to your code or rebuilds to your project will require you to repeat step 5.

6. Ensure that the Remote Debugger is running on the remote computer. The description (Figure [3-62](#)) should state that it is waiting for new connections.
7. On your local machine, start debugging your application, and if prompted, enter the credentials for the remote machine to log on. Once logged on, you will see that the Remote Debugger on the remote computer displays that the remote debug session is now active (Figure [3-65](#)). A point to note here is that if you trust the network that you are debugging across, and you are having problems logging on, you can specify that no authentication is done. In the project properties, change the Authentication mode



from Windows Authentication to No Authentication (Figure 3-64). Then, on the remote machine, click the Remote Debugger and click

the Tools menu, and select Options. Here, you can specify that no authentication is done and that any user can debug.



**Figure 3-65** Remote Debug Session Connected

8. After a few seconds, you will see your application's main window displayed on the remote machine.
9. On the remote machine, take whatever action is needed to hit any breakpoints you might have set. When you hit the breakpoint, it will be hit on your local machine (Visual Studio machine).

If you need any project resources to debug your application, you will have to include these in your project. The easiest way is to create a project folder in Visual Studio and then add the files to that folder. For each resource you add to the folder, ensure that you set the Copy to Output Directory property to Copy always.

---

## Summary

This chapter explored Visual Studio's powerful debugging capabilities. The emphasis here is that effective debugging is vital to professional software development. We discussed the fundamentals such as setting

and managing breakpoints, including conditional, temporary, dependent, and labeled breakpoints.

We had a look at debugging tools such as DataTips, Visualizers, the Watch Window and the Immediate Window that provide developers with various ways to inspect the state of their application at runtime.

GitHub Copilot also made an appearance as a debugging assistant, offering suggestions, variable analysis, and insights into exception handling. The Diagnostic Tools in Visual Studio were then introduced as a performance profiler, allowing developers to gain valuable insights into the CPU usage, memory snapshots, and managed heap behavior in real time, enhanced with AI.

Finally, the chapter concluded with a practical guide on attaching to running processes and remote debugging.

## 4. Unit Testing

Dirk Strauss<sup>1</sup> 

(1) Uitenhage, South Africa

---

Experienced developers know that in order to maintain the quality of their code is a relentless pursuit. To this end, Unit Testing is one of the most effective tools we have to catch bugs early, refactor code safely, and build software that is as robust as possible.

In this chapter, we will have a look at

- Creating and running unit tests
- Using live unit tests
- Using GitHub Copilot to generate unit tests
- How to measure Code Coverage in Visual Studio

Unit tests allow you to maintain the health of your code and find errors quickly, before shipping your application to your customers. To introduce you to unit testing, we will start off with a very basic example of creating a unit test.

---

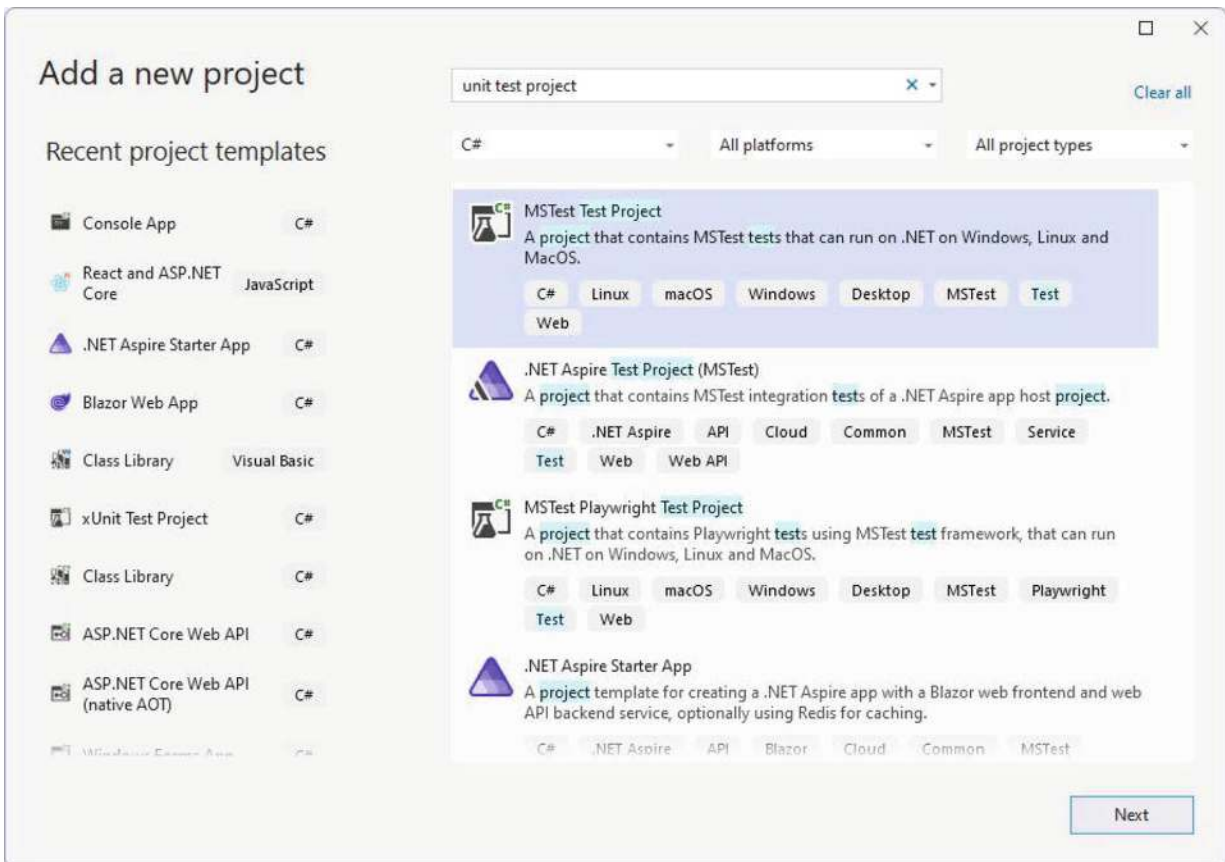
### Creating and Running Unit Tests

The purpose of unit tests is to ensure that units of code (often individual functions and classes) work as intended when they were originally written. Writing unit tests acts as a safety net, alerting you when changes in code break expected behavior. Visual Studio provides a Test Explorer to make it easy to create, organize, and run unit tests.

### Choosing a Test Framework

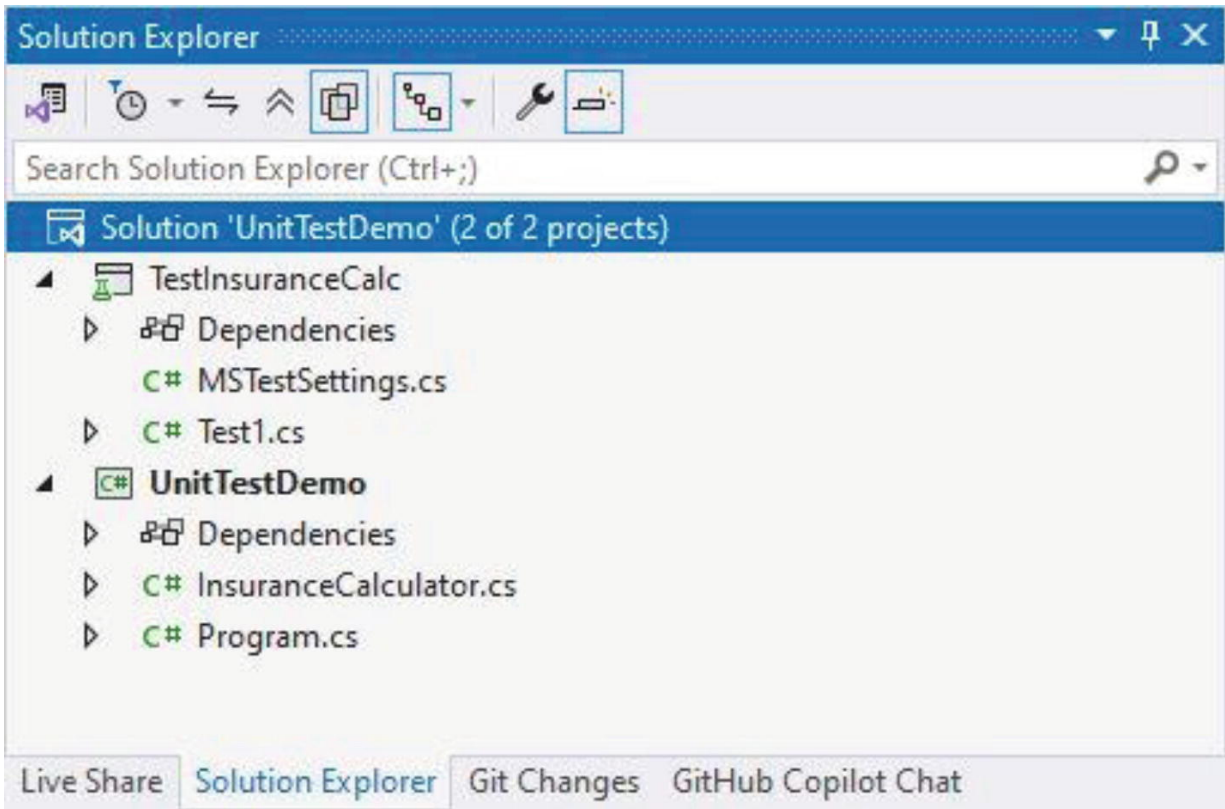
Out of the box, Visual Studio supports multiple test frameworks such as MSTest, NUnit, and xUnit. You can then decide to use the framework that is best suited for your team's habits. Adding a new test project to your solution, you will see that Visual Studio provides templates for all three frameworks.

In our example, we will be using MSTest. Adding a unit test project is done by adding a new project to your existing solution. Search for *Unit Test* in the Add new Project dialog and select *Unit Test Project* as shown in Figure [4-1](#).



**Figure 4-1** Add a New Unit Test Project

Visual Studio adds the test project to your solution with the distinct flask icon shown in Figure [4-2](#).



**Figure 4-2** The Added Unit Test Project

It is to this project that you will add your unit tests as required. To do this, start by adding a reference to the project containing the code that you want to test. Your test code can now instantiate and call methods in your class or project under test.

## Writing Your First Test

To illustrate creating unit tests, I have a small Console Application that calculates insurance premiums based on the age and driving history of the driver.

You will see this entire class in Listing [4-1](#). The static method `CalculatePremium` implements a few business rules and we want to ensure that these rules remain consistently applied.

```
public static class InsuranceCalculator
{
    public static double CalculatePremium(int age, int
accidents)
    {
        if (age < 18)
```

```

        throw new ArgumentException("Age must be 18
or older.", nameof(age));
        if (accidents < 0)
            throw new ArgumentException("Accident count
cannot be negative.", nameof(accidents));

        double basePremium = 500;
        if (age < 25)
            basePremium *= 2;
        else if (age > 70)
            basePremium *= 1.5;
        else if (age >= 50 && age <= 70)
            basePremium *= 0.8;

        if (accidents == 0)
            basePremium -= 200;
        else
            basePremium += 100;

        // Minimum premium enforcement
        if (basePremium < 300)
            basePremium = 300;

        return basePremium;
    }
}

```

**Listing 4-1** The Code to Test

Next, we need to create a test class called `InsuranceCalculatorTests` in the test project (Listing [4-2](#)). A common structure for a unit test method is Arrange-Act-Assert. We set up the inputs (Arrange), then we execute the code that we want to test (Act), and finally we verify the results (Assert).

```

[TestClass]
public sealed class InsuranceCalculatorTests
{
    [TestMethod]
    public void
PremiumCalc_YoungDriverWithAccident_ReturnsHigherPremium
()

```



```

{
    // Arrange
    int age = 22;
    int accidents = 1;
    // Expected: base $500 * 2 for young driver +
    $100 for one accident = $1100
    double expected = 1100;

    // Act
    double actual =
InsuranceCalculator.CalculatePremium(age, accidents);

    // Assert
    Assert.AreEqual(expected, actual, 0.001,
        "Premium for young driver with an accident
should be high");
}
}

```

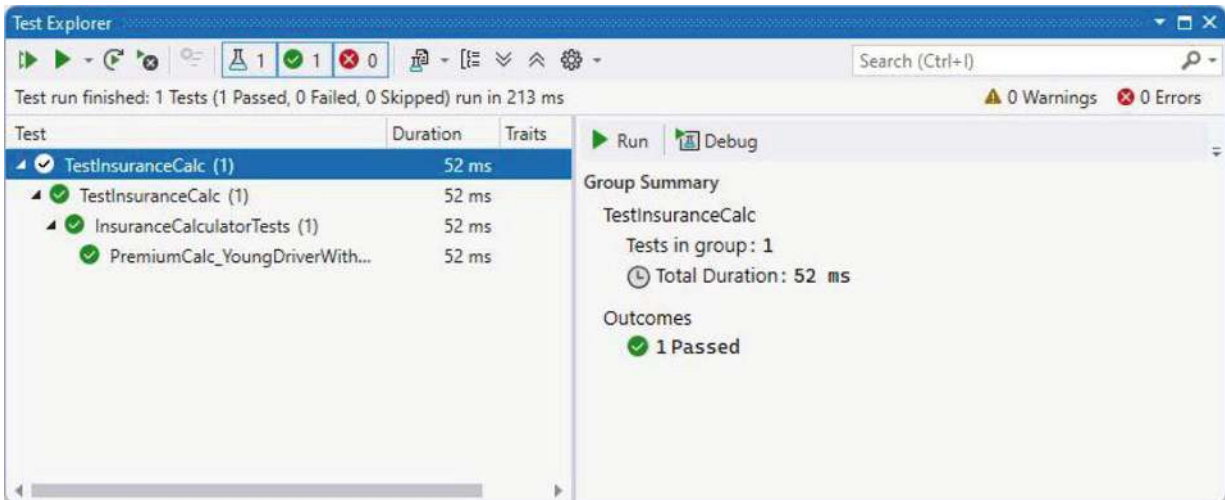
**Listing 4-2** The Test Class

Using MSTest, you will notice that we mark the class with `[TestClass]` and every method with `[TestMethod]`. What this method is testing is that a 22-year-old driver that has previously been involved in an accident is charged the correct higher premium.

The message in the `Assert.AreEqual` clarifies the assumption being checked and this message will appear if the assertion fails. The additional tolerance of `0.001` is also good practice when dealing with double calculations because floating-point results might not match exactly.

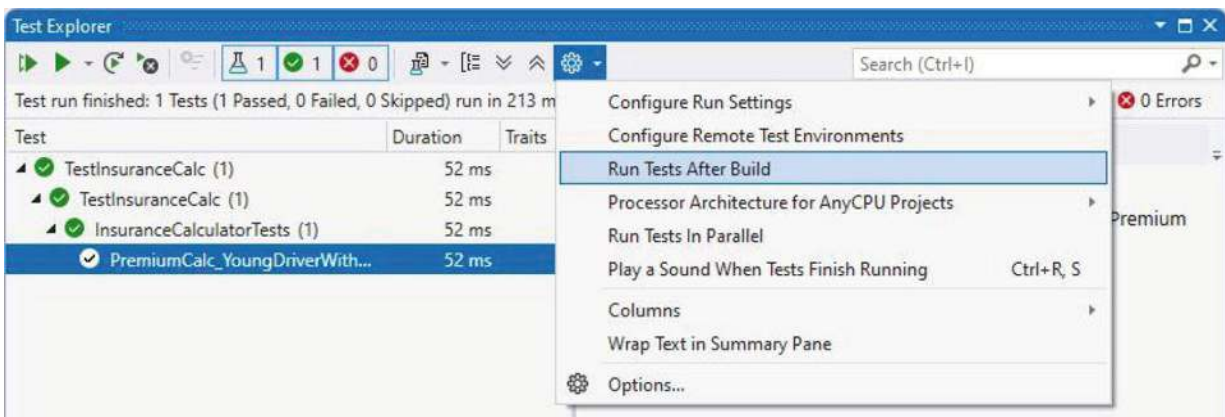
## Running Your Tests

When it's time to run your tests, Visual Studio's Test Explorer (Figure [4-3](#)) serves as the central hub for running and managing them. Invoked by typing `Ctrl+E, T`, be sure to build your solution. Visual Studio will discover and list the tests you wrote in real time (Tools ► Options ► Test ► Test Discovery).



**Figure 4-3** Test Explorer Window

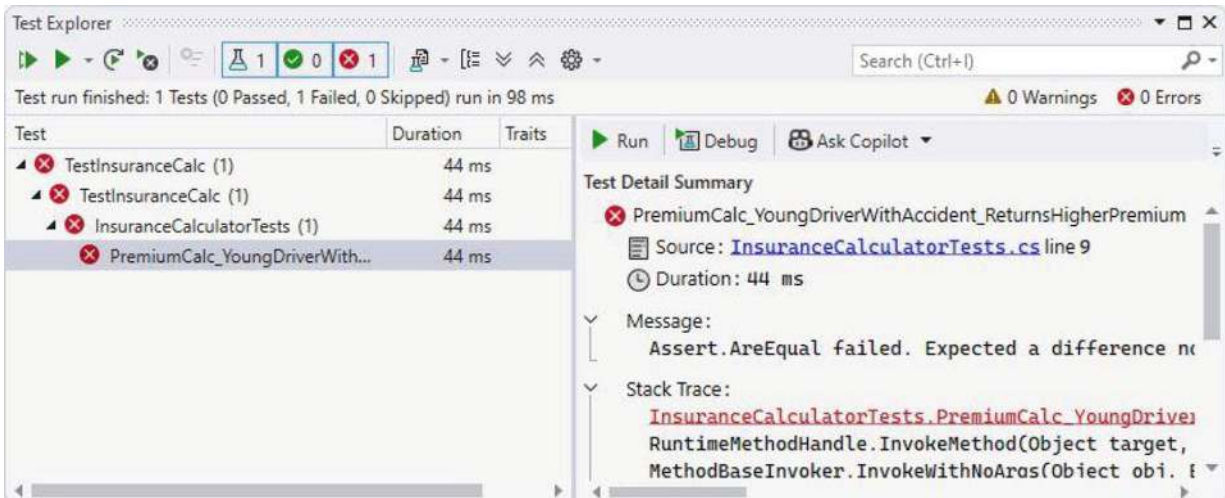
To run all your tests, click the Run All button indicated by a double green arrow on the Test Explorer toolbar.



**Figure 4-4** Test Explorer Settings

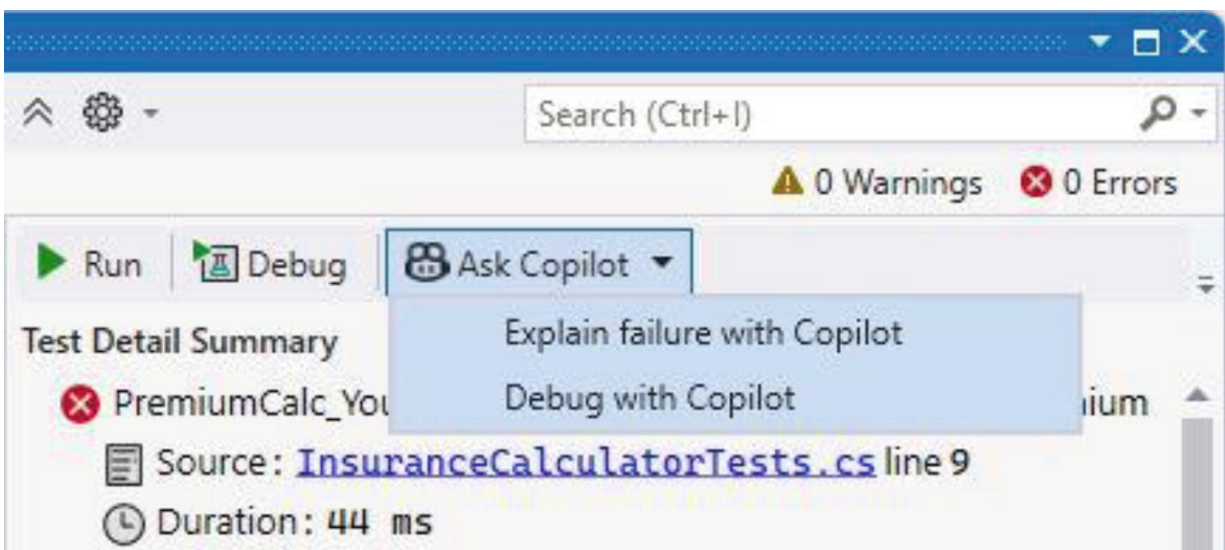
The Test Explorer Options, as seen in Figure 4-4, allow you to configure additional behavior such as running tests in parallel, running tests after a build, selecting the processor architecture, and more.

Once the tests have been run, Test Explorer will display the test results as passed or failed.



**Figure 4-5** Failed Tests

Failed tests are indicated with a red error indicator as seen in Figure 4-5. The test detail summary shows you the assertion message with the expected versus actual values.



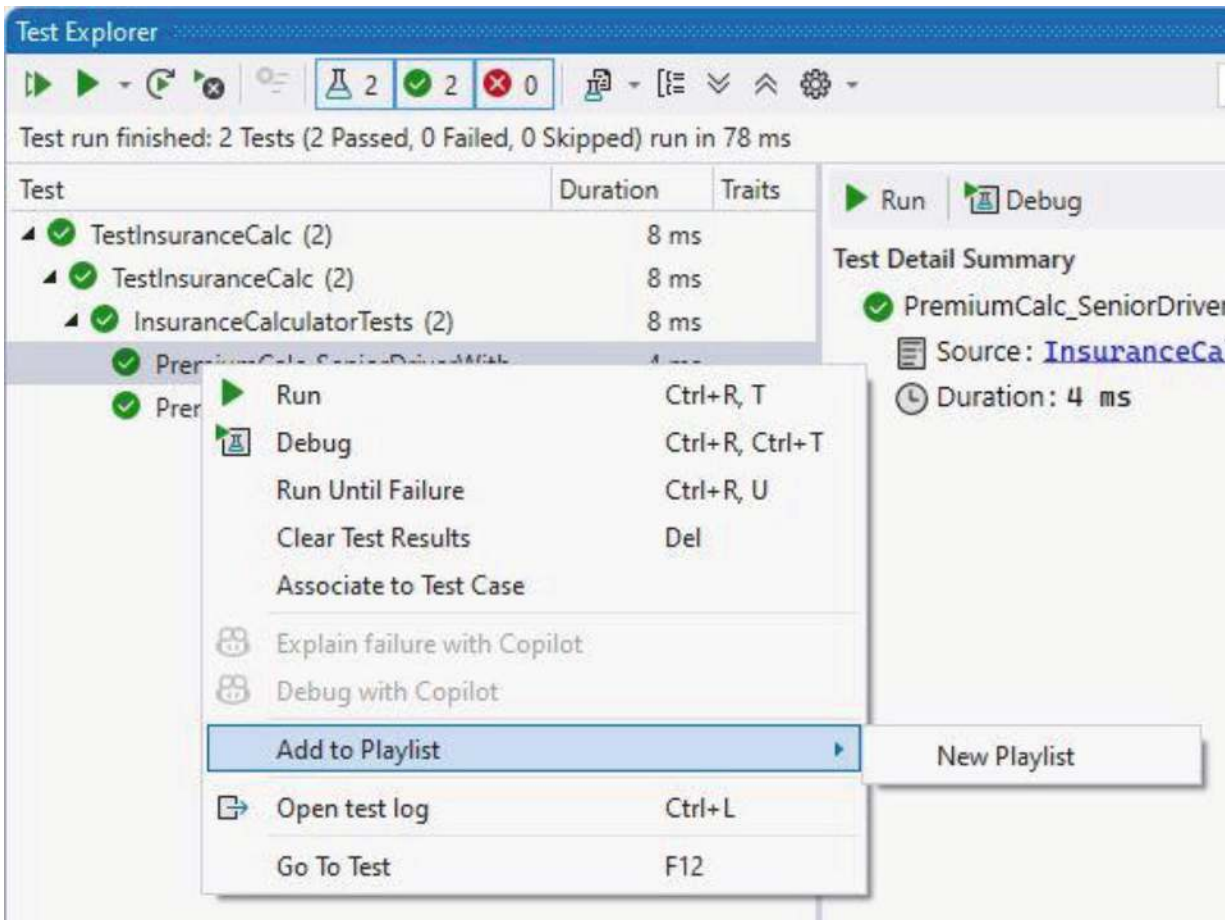
**Figure 4-6** Ask Copilot Option

For any failed tests, Copilot is also available, as seen in Figure 4-6. You can get Copilot assistance to explain the failure or to debug the issue for you.

## Create and Run a Test Playlist

If your project contains many tests, and you want to run those tests as a group, you can create a playlist. To create a playlist, select the tests that you want to

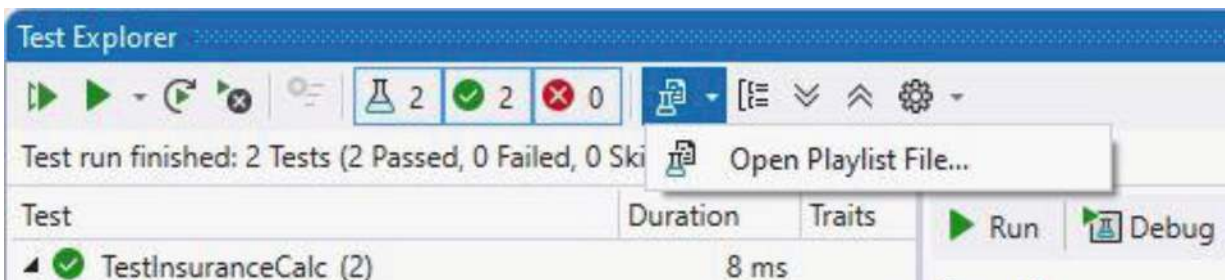
group from the Test Explorer, and right-click them. From the context menu that pops up, select Add to Playlist ► New Playlist as seen in Figure 4-7.



**Figure 4-7** Create a Playlist

This will open a new Test Explorer window where you can run the tests and save the tests you selected under a new playlist name. This will create a .playlist file for you.

I created a new playlist called `SeniorJunior.playlist` from two of the tests that test senior drivers and young drivers. The playlist file it creates is simply an XML file that you can view and open from your Solution Explorer.



**Figure 4-8** Open a Playlist File

To open and run a playlist again, click the Open Playlist File button shown in Figure [4-8](#) and select the playlist file you want to run.

## Unit Test Attributes

In MSTest, you need to mark each class with `[TestClass]` and each method with `[TestMethod]`. Using the `[TestClass]` attribute specifies that a class contains unit tests along with optional setup and teardown methods. The `[TestMethod]` attribute specifies an individual test method that needs to be run by the test runner. You will see this if you refer back to the code in Listing [4-2](#).

## Setting Up and Tearing Down

MSTest also allows you to define setup and teardown logic using two special attributes. These are:

- `[TestInitialize]` – marks a method to run before each test.
- `[TestCleanup]` – marks a method to run after each test.

These are instance methods and can optionally be async. They run around every test, setting up and tearing down logic. You can also define class-level setup and teardown logic with `[ClassInitialize]` and `[ClassCleanup]`, which will be called once per class. These need to be public static methods, and the class initialization method takes a `TestContext` parameter.

Similarly, assembly-level equivalents `[AssemblyInitialize]` and `[AssemblyCleanup]` will run once per test assembly. These are also public static methods.

You can see an example of this structure in Listing [4-3](#). Being able to specify setup and teardown logic in this manner allows developers to be specific about the creation and destruction of their unit tests.

```
[TestClass]
public sealed class InsuranceCalculatorTests
{
    [AssemblyInitialize]
    public static void AssemblyInitialize(TestContext
context)
    {
        // Any assembly-level setup code if needed
    }
}
```

```

    }

    [ClassInitialize]
    public static void ClassInitialize(TestContext
context)
    {
        // Any class-level setup code if needed
    }

    [TestInitialize]
    public void TestInitialize()
    {
        // Any test-level setup code if needed
    }

    [TestMethod]
    public void Test1() { }

    [TestMethod]
    public void Test2() { }

    [TestCleanup]
    public void TestCleanup()
    {
        // Runs after each test method
    }

    [ClassCleanup]
    public static void ClassCleanup()
    {
        // Runs once after all tests in this class
    }

    [AssemblyCleanup]
    public static void AssemblyCleanup()
    {
        // Any assembly-level cleanup code if needed
    }
}

```

**Listing 4-3** Test Class Structure



## ***Data Driven Tests***

MSTest also supports tests that are data-driven or parameterized. To do this, you need to add the `[DataRow]` attribute that will provide the data for each test method. In other words, every `[DataRow]` attribute added to the method will cause the method to run once with those specified parameters. You also indicate a data-driven test by using the `[DataTestMethod]` attribute. An example can be seen in Listing [4-4](#).

```
[DataTestMethod]
[DataRow(22, 1, 1100)]
[DataRow(30, 0, 300)]
[DataRow(72, 0, 550)]
public void TestVariableCalculations(int age, int
accidents, double expected)
{
    // Act
    double actual =
InsuranceCalculator.CalculatePremium(age, accidents);
    // Assert
    Assert.AreEqual(expected, actual, 0.0001,
        $"Premium for driver age {age} with {accidents}
accidents should be {expected}");
}
```

**Listing 4-4** A Data-Driven Test

In the code example in Listing [4-4](#), the test will be called three times with the provided arguments. Once for each `[DataRow]` attribute set. The number and types of the parameters in the `[DataRow]` must match the method signature.

## ***Dynamic Data Driven Tests***

We saw that we can use `[DataRow]` with hardcoded data to run test methods, but we can also supply data to a test method via a property or method using `[DynamicData]`.

Looking at the code in Listing [4-5](#), you will see that I have added a read only property called `AdditionalData` that simply returns `IEnumerable<object[]>` of the test cases we want to execute.

On the test method `TestDynamicDataCalucaltions` I then use the `[DynamicData(nameof(AdditionalData))]` attribute.

```

public static IEnumerable<object[]> AdditionData
{
    get
    {
        yield return new object[] { 22, 1, 1100 };
        yield return new object[] { 30, 0, 300 };
        yield return new object[] { 72, 0, 550 };
    }
}

[TestMethod]
[DynamicData(nameof(AdditionData))]
public void TestDynamicDataCalculations(int age, int
accidents, double expected)
{
    // Act
    double actual =
InsuranceCalculator.CalculatePremium(age, accidents);
    // Assert
    Assert.AreEqual(expected, actual, 0.0001,
        $"Premium for driver age {age} with {accidents}
accidents should be {expected}");
}

```

**Listing 4-5** Using Dynamic Data

This will tell MSTest to fetch each row of that data and feed it to the test method. You can also optionally specify a custom display name for each data row via `DynamicDataDisplayName` to improve the readability of the test results.

In Listing [4-6](#) I have created a method called `GetCustomDynamicDataDisplayName` that will enrich the test results summary. The attribute on the test method is also changed from `[DynamicData(nameof(AdditionData))]` to `[DynamicData(nameof(AdditionData), DynamicDataDisplayName = nameof(GetCustomDynamicDataDisplayName))]`.

```

public static IEnumerable<object[]> AdditionData
{
    get

```

```

        {
            yield return new object[] { 22, 1, 1100 };
            yield return new object[] { 30, 0, 300 };
            yield return new object[] { 72, 0, 550 };
        }
    }

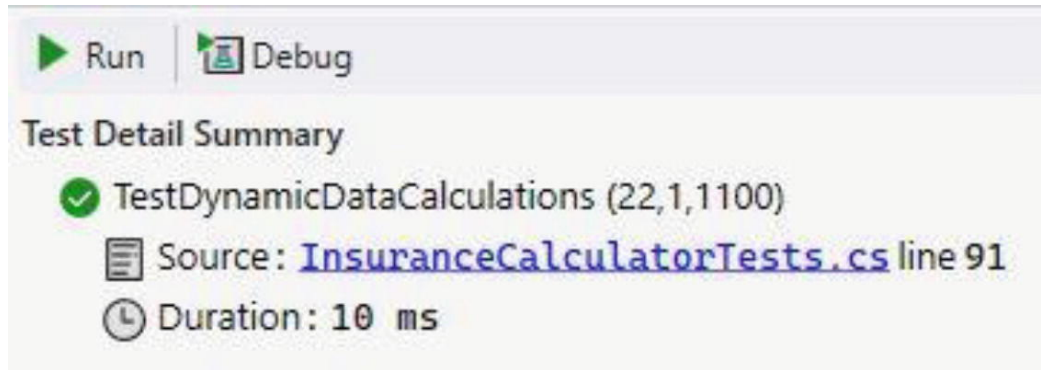
    public static string
    GetCustomDynamicDataDisplayName(MethodInfo methodInfo,
    object[] data)
    {
        return string.Format("DynamicDataTestMethod {0}
with {1} parameters", methodInfo.Name, data.Length);
    }

    [TestMethod]
    [DynamicData(nameof(AdditionData),
DynamicDataDisplayName =
nameof(GetCustomDynamicDataDisplayName))]
    public void TestDynamicDataCalculations(int age, int
accidents, double expected)
    {
        // Act
        double actual =
InsuranceCalculator.CalculatePremium(age, accidents);
        // Assert
        Assert.AreEqual(expected, actual, 0.0001,
            $"Premium for driver age {age} with {accidents}
accidents should be {expected}");
    }
}

```

**Listing 4-6** Using DynamicDataDisplayName

This will result in your test summary changing from what we see in [Figure 4-9](#) to what you specified in the `GetCustomDynamicDataDisplayName` method and output in the test summary as seen in [Figure 4-10](#).



**Figure 4-9** Without Using GetCustomDynamicDataDisplayName

While this example is rather contrived, it illustrates the flexibility you have over what can be displayed in the test Detail Summary.



**Figure 4-10** Using GetCustomDynamicDataDisplayName

The flexibility provided to developers does not stop there. You also have access to execution control attributes, which we will be looking at next.

## Testing Timeouts

The speed of your code is also very important. Using the MSTest framework, you can set a timeout attribute to specify a timeout after which a test should fail. This is convenient because as you write code for a specific method, you can immediately identify if the code you are adding to a method is causing a potential bottleneck. Consider the `TestTimeoutOfMethod` test in Listing [4-7](#).

Here we are specifying that if the test runs for longer than 2 seconds (2000 milliseconds), then it should be aborted and marked as failed. To force a failure, I have added a delay in the code of 2100 milliseconds.

```
[TestMethod, Timeout(2000)]
public void TestTimeoutOfMethod()
{
```

```

// Arrange
int age = 22;
int accidents = 1;
// Expected: base $500 * 2 for young driver + $100
for one accident = $1100
double expected = 1100;

Task.Delay(2100).Wait(); // Simulate some delay

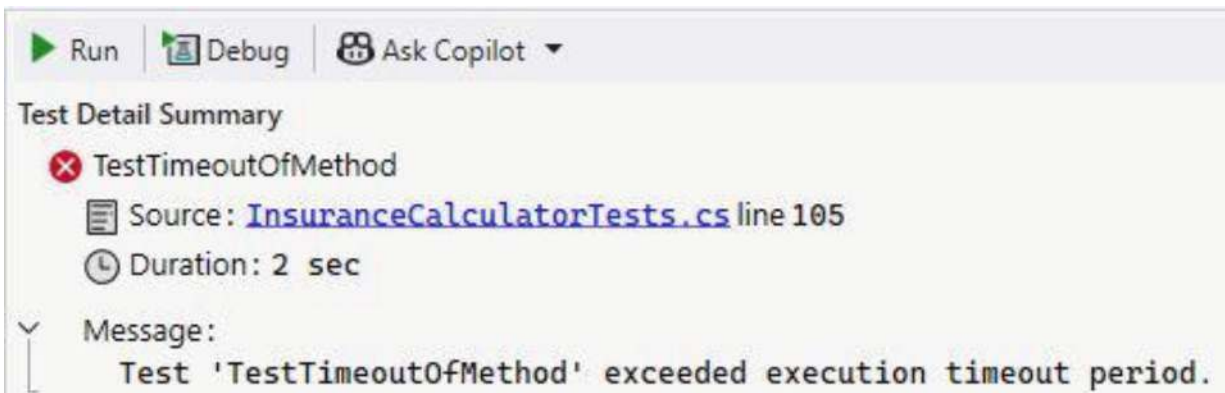
// Act
double actual =
InsuranceCalculator.CalculatePremium(age, accidents);

// Assert
Assert.AreEqual(expected, actual, 0.0001,
    "Premium for young driver with an accident
should be high");
}

```

**Listing 4-7** Adding a Timeout Attribute

Running this test you will notice that it will fail because it has exceeded the specified timeout value set by the `Timeout` attribute (Figure 4-11).



**Figure 4-11** Test Timeout Exceeded

Identifying critical methods in your code and setting a specific timeout on that method will allow developers to catch issues early on when tests start exceeding the timeout set. You can then go back and immediately refactor the code that was recently changed to improve the execution time.

## **Retry**

You can specify that MSTest automatically retries a failed test. As of MSTest v3.8, the `Retry` attribute allows you to specify how many times a failing test should be retried.

```
private static int _retryAttempt = 0;

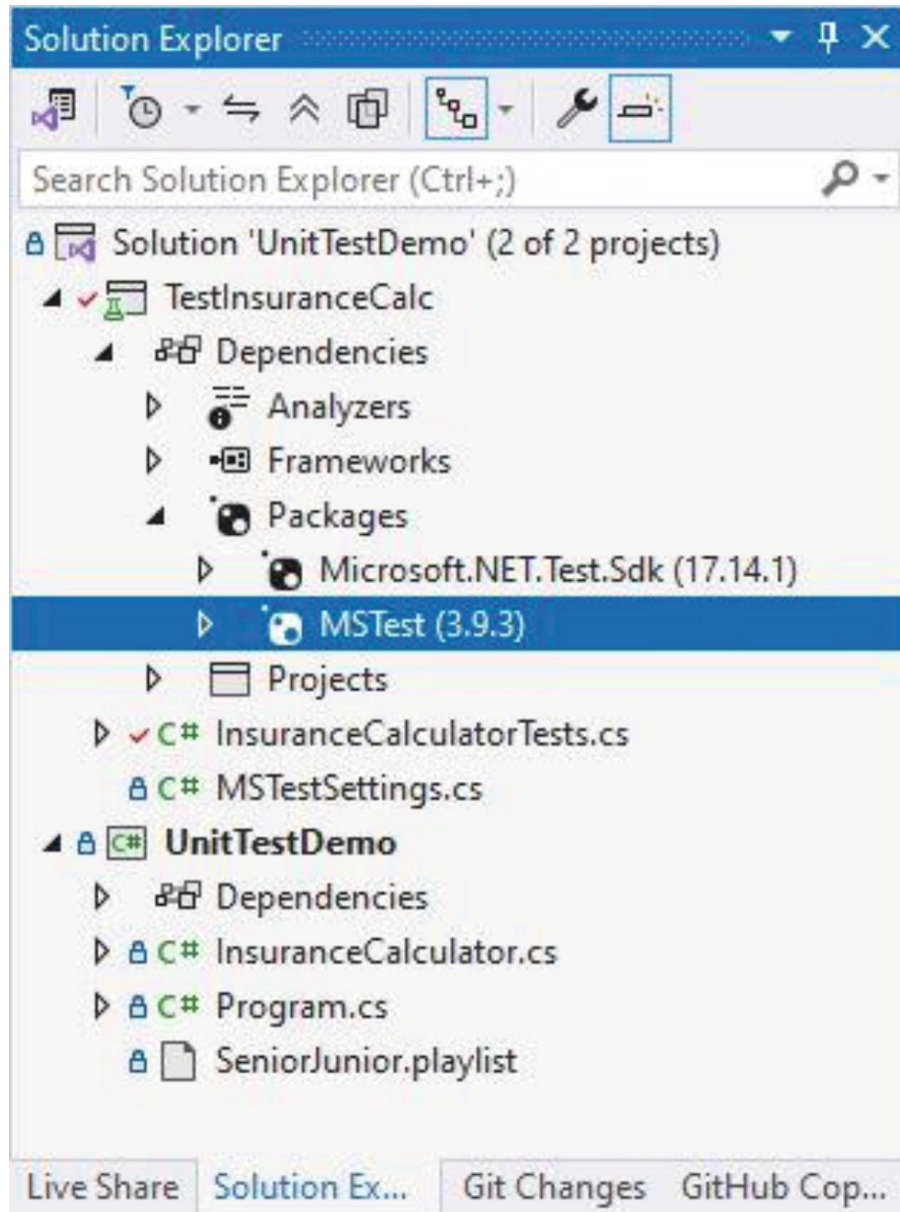
[TestMethod, Retry(2)]
public void TestRetryAttempt()
{
    _retryAttempt++;
    if (_retryAttempt == 1)
    {
        Assert.Fail("Intentional failure on first
attempt.");
    }

    // Succeeds on retry
    Assert.IsTrue(true, "Test succeeded on retry.");
}
```

**Listing 4-8** Using `Retry` on a Unit Test

From the contrived example in Listing [4-8](#) you will see that I have specified that a test needs to be retried twice. The code then forces a failure. To see this in action, place a breakpoint on the `Assert.Fail` and debug your unit test. After you step over the failure, the unit test will be run again, succeeding the second time.





**Figure 4-12** MSTest Version

Ensure that you have at least MSTest 3.8 or later installed, as seen in Figure [4-12](#).

### ***Metadata Attributes***

There are several attributes that exist within MSTest that add metadata or alter the way tests appear in Test Explorer. While they do not change test execution, it helps organize your tests.

```
[TestMethod]
```

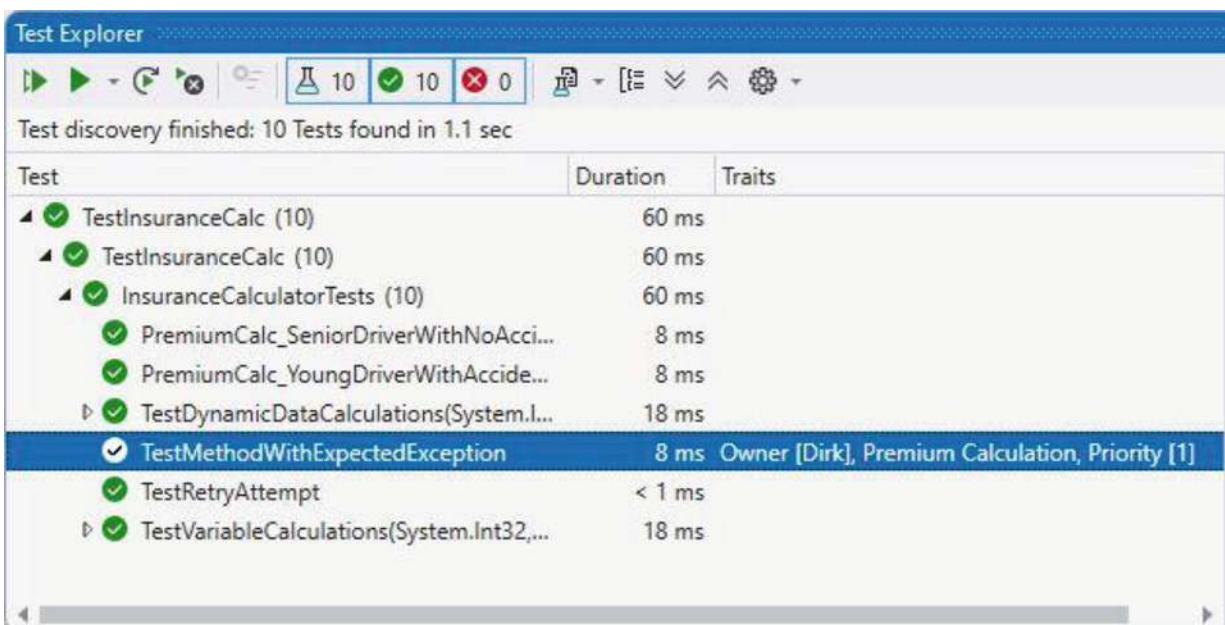
```

[Description("Throws ArgumentException when age is less
than 18.")]
[TestCategory("Premium Calculation")]
[Priority(1)]
[Owner("Dirk")]
[WorkItem(1234)]
public void TestMethodWithExpectedException()
{
    // Arrange
    int age = 16; // Invalid age
    int accidents = 0;
    // Act & Assert
    Assert.ThrowsException<ArgumentException>(() =>
InsuranceCalculator.CalculatePremium(age, accidents),
    "Expected ArgumentException for age less than
18.");
}

```

**Listing 4-9** Metadata Attributes

If you look at the Test Explorer after running your test, you will see the metadata for Owner, TestCategory, and Priority displayed in the Traits column (Figure 4-13).



**Figure 4-13** Test Explorer Displaying Metadata

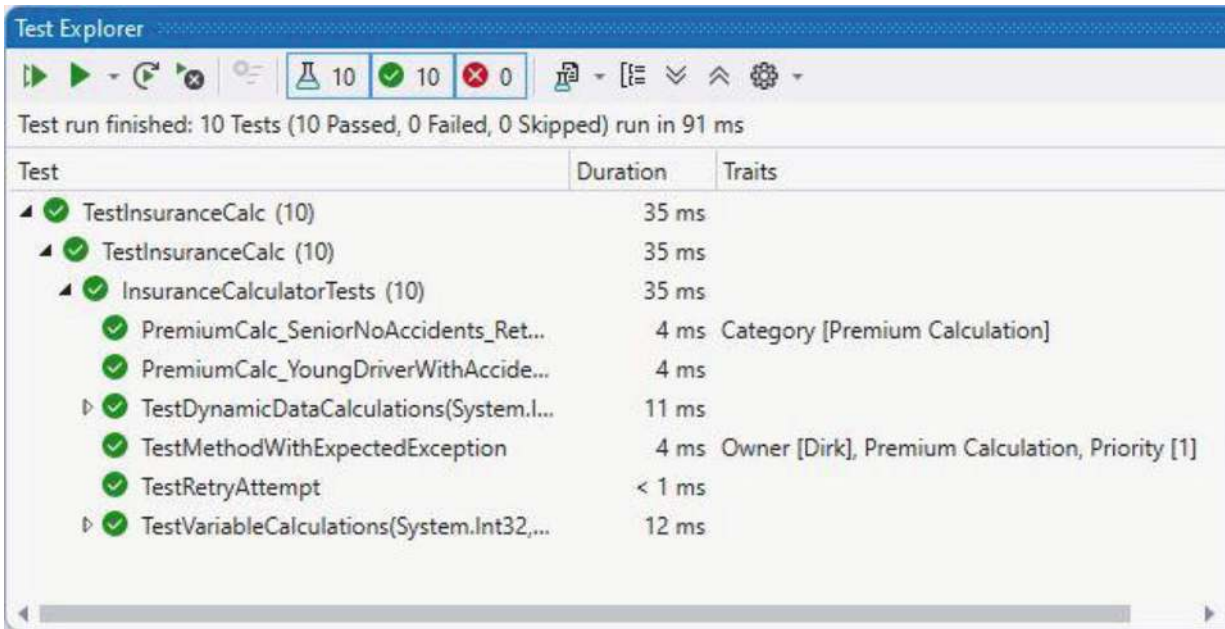
As for `Description` and `WorkItem`, these attributes are applied but not displayed in the Traits column, potentially for you to enrich the test method with additional information about its purpose.

If you need to add an arbitrary key/value property to your test, you can use the `TestProperty` attribute as seen in Listing [4-10](#).

```
[TestMethod]
[TestProperty("Category", "Premium Calculation")]
public void
PremiumCalc_SeniorNoAccidents_ReturnsDiscountedPremium()
{
    // Arrange
    int age = 72;
    int accidents = 0;
    // Expected: base $500 * 1.5 for senior driver -
    $200 for no accidents = $550
    double expected = 550;
    // Act
    double actual =
InsuranceCalculator.CalculatePremium(age, accidents);
    // Assert
    Assert.AreEqual(expected, actual, 0.0001,
        "Premium for senior driver with no accidents
should be discounted to minimum");
}
```

**Listing 4-10** Adding the `TestProperty` Attribute

Once added you will see this displayed in the Traits column as displayed in Figure [4-14](#).



**Figure 4-14** Viewing the TestProperty Attribute

As you can see these attributes do not affect test execution, but help organize your tests.

## Ignore

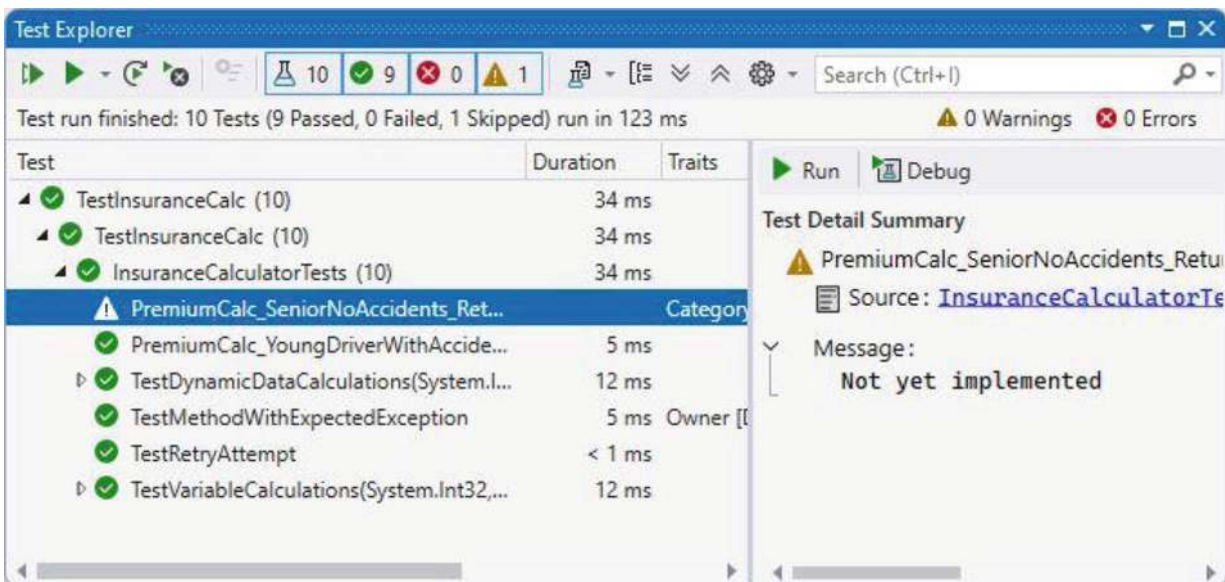
You can also choose to exclude tests from execution. You can optionally provide a reason message, but marking a test with `Ignore` will report it as skipped in the test results.

```
[TestMethod]
[TestProperty("Category", "Premium Calculation")]
[Ignore("Not yet implemented")]
public void
PremiumCalc_SeniorNoAccidents_ReturnsDiscountedPremium()
{
    // Arrange
    int age = 72;
    int accidents = 0;
    // Expected: base $500 * 1.5 for senior driver -
    $200 for no accidents = $550
    double expected = 550;
    // Act
    double actual =
InsuranceCalculator.CalculatePremium(age, accidents);
```

```
// Assert
Assert.AreEqual(expected, actual, 0.0001,
    "Premium for senior driver with no accidents
should be discounted to minimum");
}
```

**Listing 4-11** Ignoring a Test

Using the code in Listing [4-11](#), you will notice that the `Ignore` attribute has been added to the test method.



**Figure 4-15** Skipping a Test with Ignore

When running your tests, the test will be marked as skipped and the message you provided (if any) will be displayed in the summary.

You might skip a test if you're still working on it and need to commit your code, or if the test has become obsolete but you don't want to remove it yet. In any event, having this kind of control over test execution is very convenient.

## Using Live Unit Tests

First introduced in Visual Studio 2017, Live Unit Testing runs your unit tests automatically as you make changes to your code. You can then see the results of your unit tests in real time.

Live Unit Testing is only available in Visual Studio Enterprise edition for C# and Visual Basic projects targeting the .NET Framework or .NET 5 and later. For a full comparison between the editions of Visual Studio, refer to



the following link:

<https://visualstudio.microsoft.com/vs/compare/>.

The benefits of Live Unit Testing are as follows:

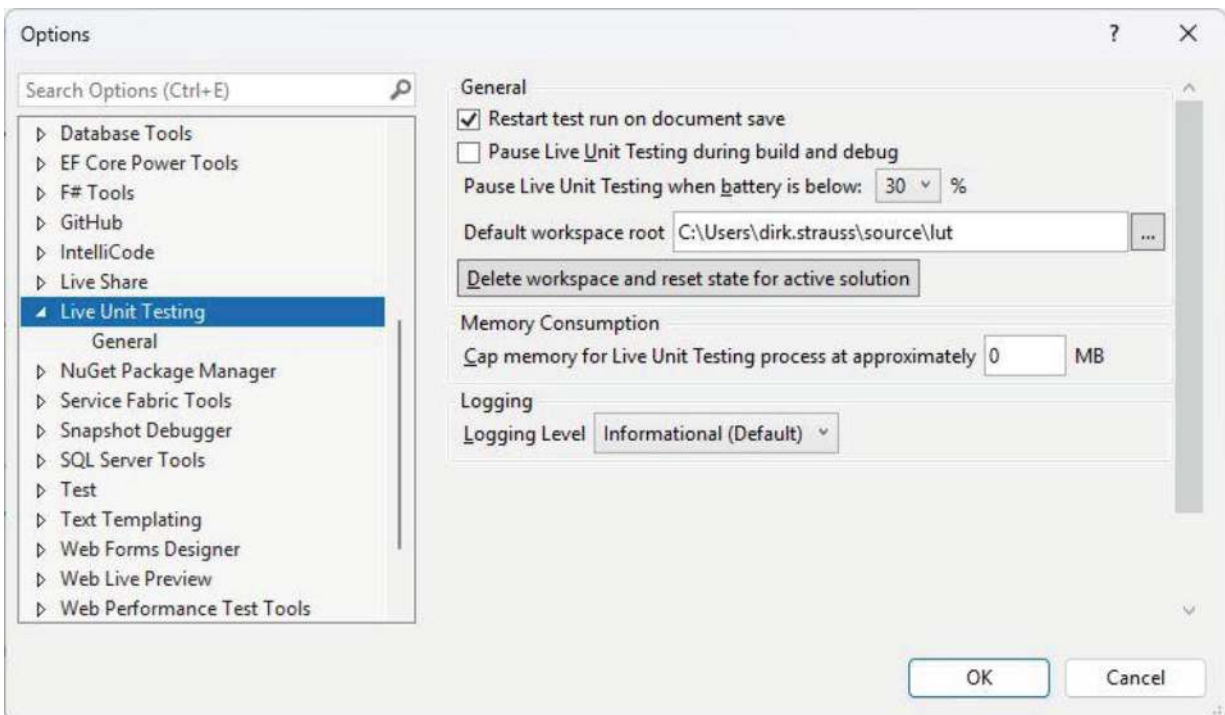
- You will immediately see failing tests, allowing you to easily identify breaking code changes.
- It indicates Code Coverage, allowing you to see what code is not covered by any unit tests.

Live Unit Testing persists the data of the status of the tests it ran. It then uses the persisted data to dynamically run your tests as your code changes. Live Unit Testing supports the following test frameworks:

- xUnit.net – Minimum version xunit 1.9.2
- NUnit – Minimum version NUnit version 3.5.0
- MSTest – Minimum version MSTest.TestFramework 1.0.5-preview

To view the Live Unit Testing frequently asked questions, browse to <https://learn.microsoft.com/en-us/visualstudio/test/live-unit-testing-faq>.

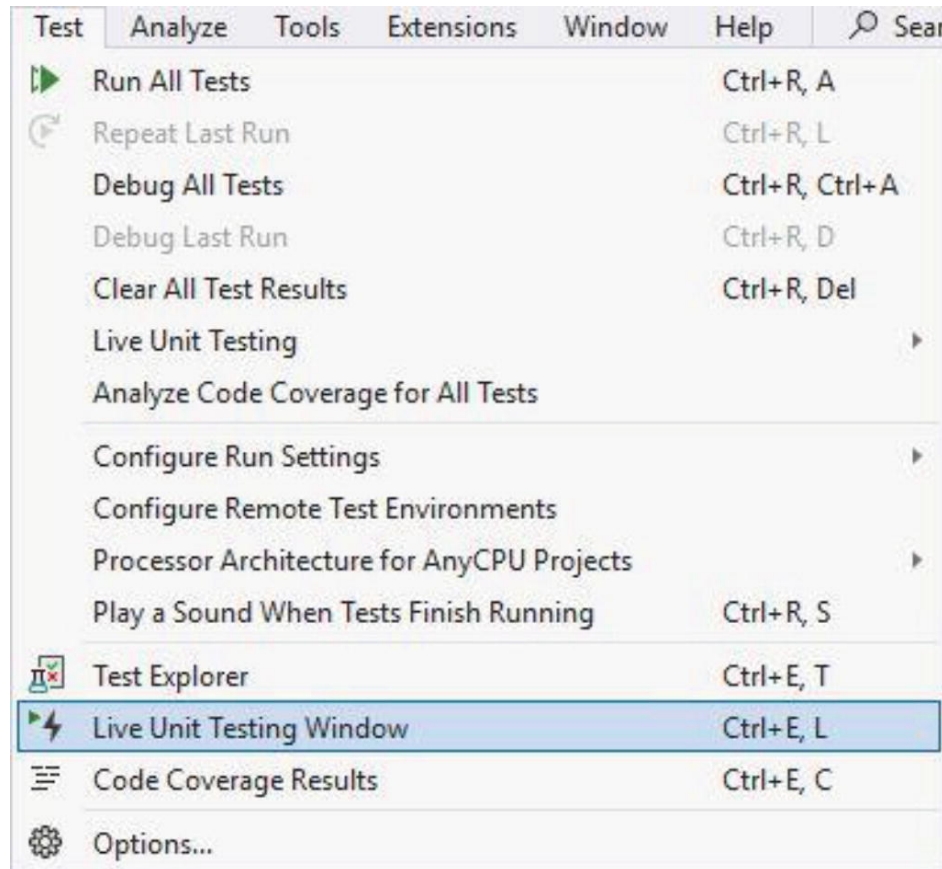
Before you can start using Live Unit Testing, you need to configure it by going to Tools ► Options and selecting Live Unit Testing in the left pane (Figure 4-9).





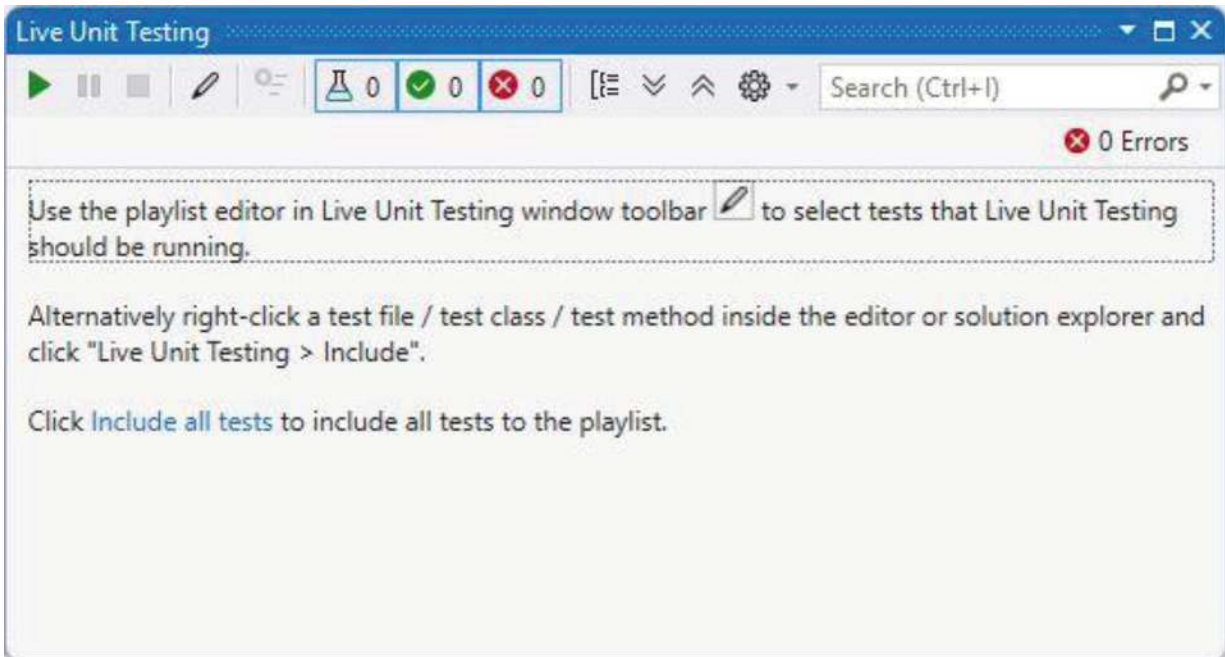
**Figure 4-16** Configure Live Unit Testing

Once you have configured the Live Unit Testing options, you can enable it from Test ► Live Unit Testing Window (Figure [4-17](#)) or by holding down Ctrl+E, L.



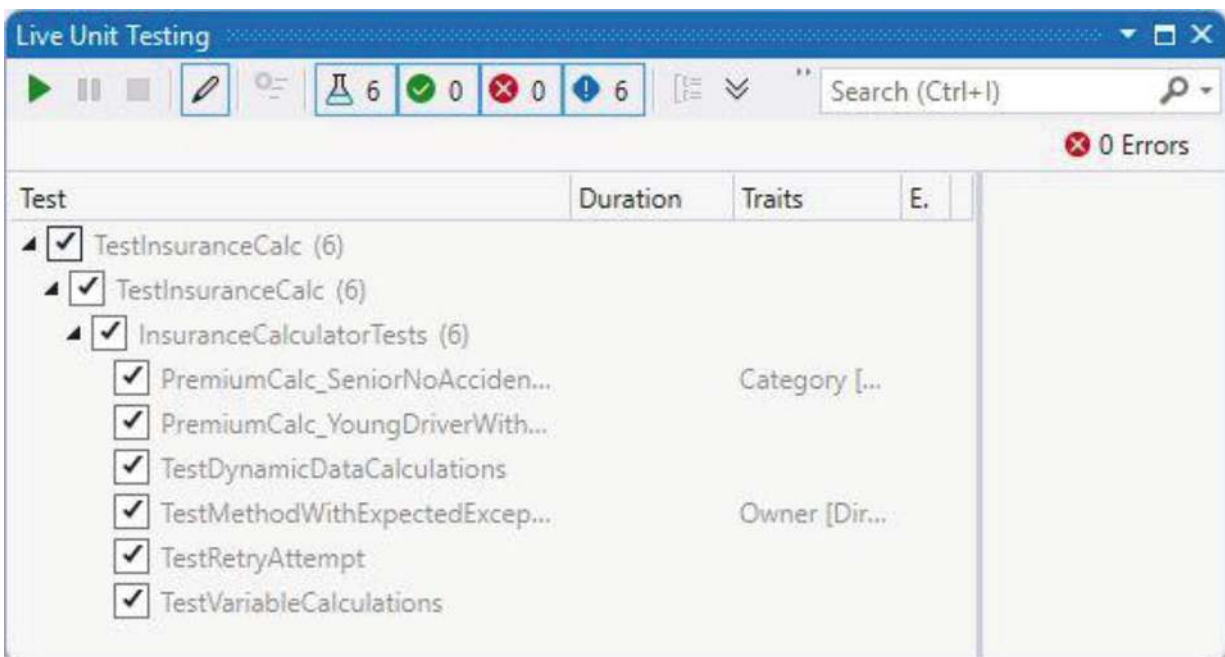
**Figure 4-17** Open the Live Unit Testing Window

The Live Unit Testing window is displayed as seen in Figure [4-18](#).



**Figure 4-18** Live Unit Testing window

Click on the playlist editor in the toolbar to select the tests that Live Unit Testing should run, shown in Figure 4-19. When selected, click the pencil icon again and then Start.



**Figure 4-19** Selecting Tests to be Run

You will now see the Configure Live Unit Testing screen displayed (Figure 4-20).

Configure Live Unit Testing

Repository root ⓘ  
C:\temp\\_UnitTestDemo\UnitTestDemo ...

Workspace folder ⓘ  
C:\Users\dirk.trauss\source\lut\UnitTestDemo ...

Exclude these files when syncing between repository root and workspace folder  
Common artifacts eg. bin/, obj/, \*.dll, \*.obj

**Build options**  
☒ When building, use multiple MSBuild processes

**Test options**  
Test case timeout (s) ⓘ  
180  
☒ When running tests, use multiple test processes

[How do I configure Live Unit Testing?](#)

Back Next Finish Cancel

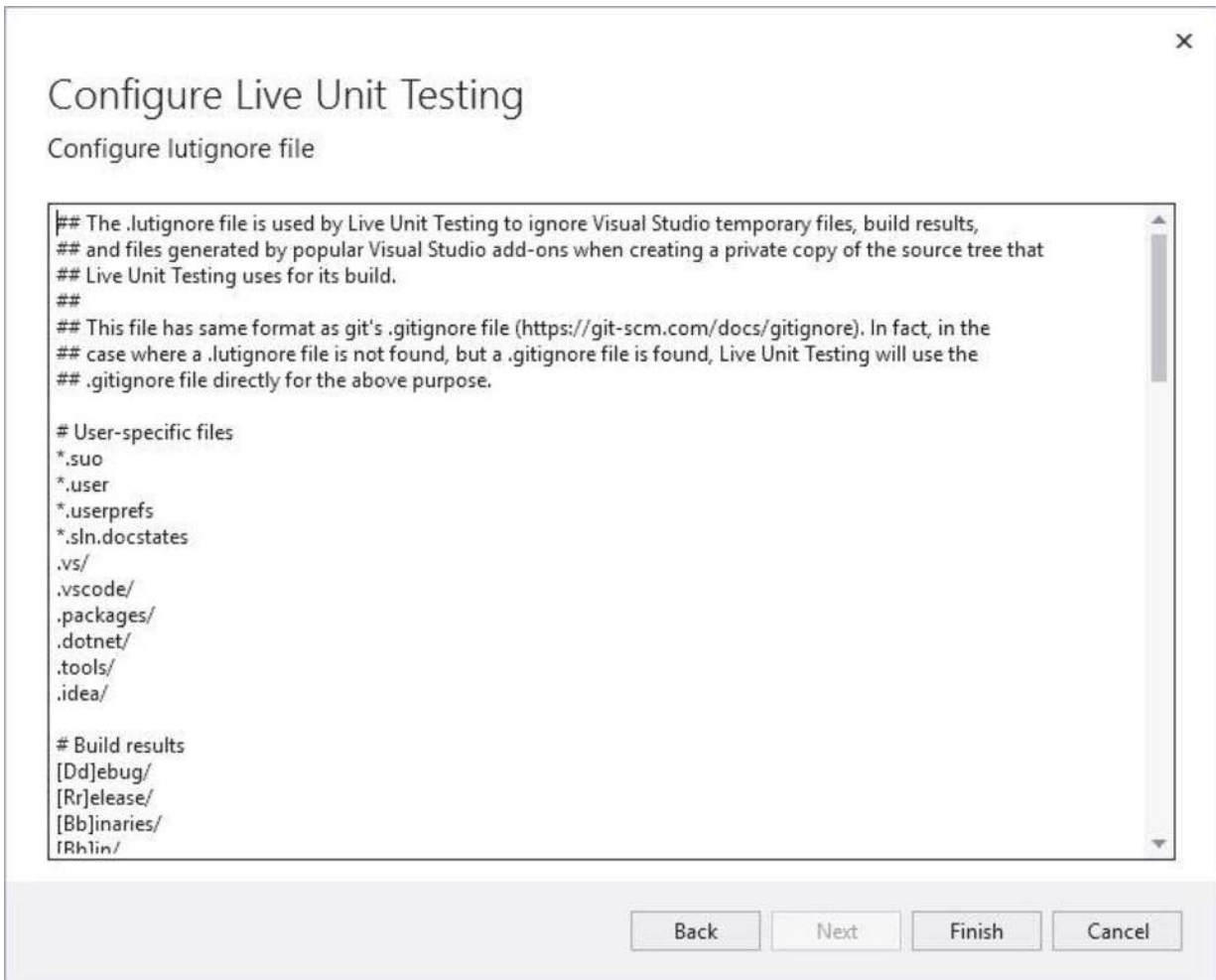
**Figure 4-20** Configure Live Unit Testing

This screen allows developers to tailor how Live Unit Testing operates on their solution. You will see this screen when initializing Live Unit Testing (LUT) for the first time. Here you specify a *Repository* root and *Workspace* folder.

Live Unit Testing requires the repository root to be the parent folder that contains all projects referenced in the solution. If a solution references sibling project folders using relative paths, the repository root must be set above all those folders to resolve correctly.

The Workspace folder isolates the working copy and avoids polluting the source code repo with LUT artifacts. You can also filter common build outputs

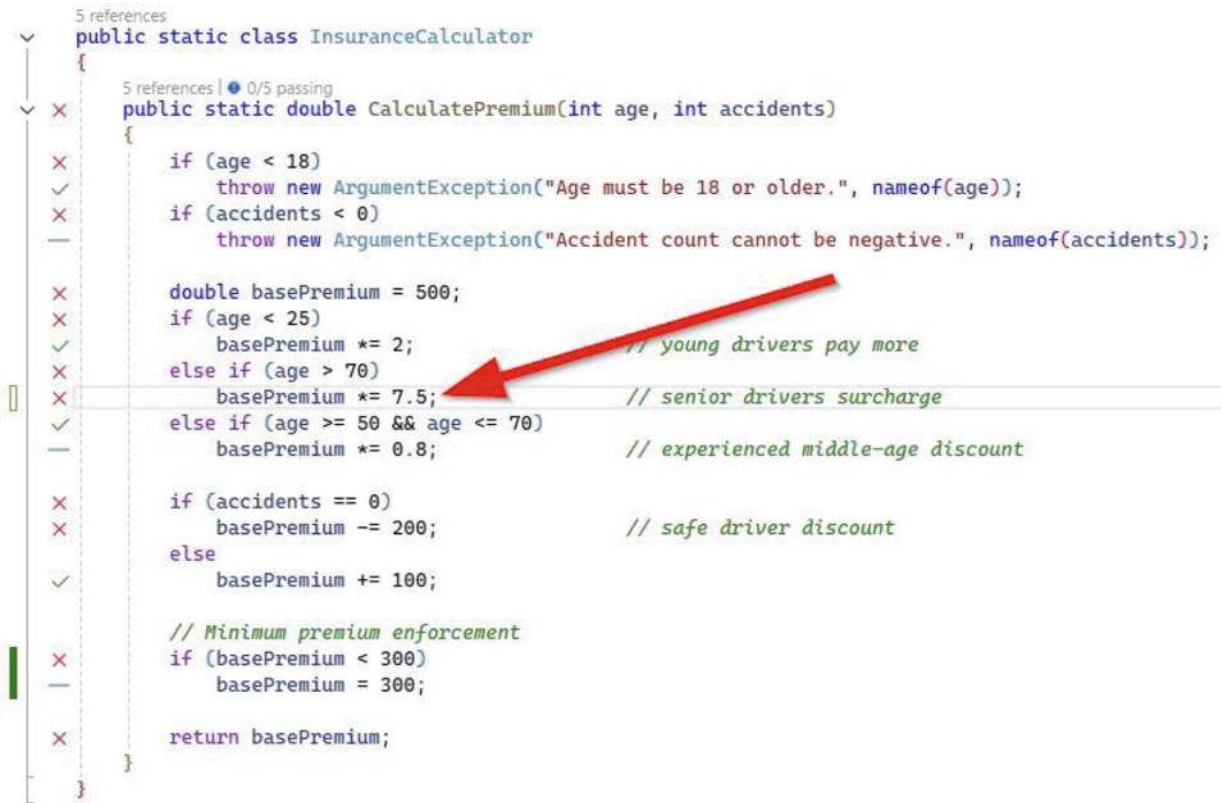
such as `bin/`, `obj/`, `*.dll`, and `*.obj` to optimize sync performance. Enabling multiple MSBuild processes under *Build options* can significantly reduce build time in large solutions. For the *Test options* section you can configure a timeout (in seconds) and parallelize test execution using multiple test processes, which is ideal for improving responsiveness and test throughput. Configuring your Live Unit Testing ensures seamless integration with real-world dev workflows, minimizing overhead while maintaining immediate feedback on code changes.



**Figure 4-21** Configure Lutignore File

Clicking Next will display the *Configure lutignore file* screen. This allows you to define which files and directories to exclude when creating its workspace. The `.lutignore` file follows the same syntax as `.gitignore`, filtering out user-specific files, build outputs, and temporary artifacts. If a `.lutignore` file is not present, Live Unit Testing will fall back to the `.gitignore` file, ensuring clean and efficient test runs.

Referring back to Listing [4-1](#), make a breaking change to your CalculatePremium method as illustrated in Figure [4-22](#). You will notice that I changed the basePremium multiplier from 1.5 to 7.5.



```
5 references
public static class InsuranceCalculator
{
    5 references | 0/5 passing
    public static double CalculatePremium(int age, int accidents)
    {
        if (age < 18)
            throw new ArgumentException("Age must be 18 or older.", nameof(age));
        if (accidents < 0)
            throw new ArgumentException("Accident count cannot be negative.", nameof(accidents));

        double basePremium = 500;
        if (age < 25)
            basePremium *= 2; // young drivers pay more
        else if (age > 70)
            basePremium *= 7.5; // senior drivers surcharge
        else if (age >= 50 && age <= 70)
            basePremium *= 0.8; // experienced middle-age discount

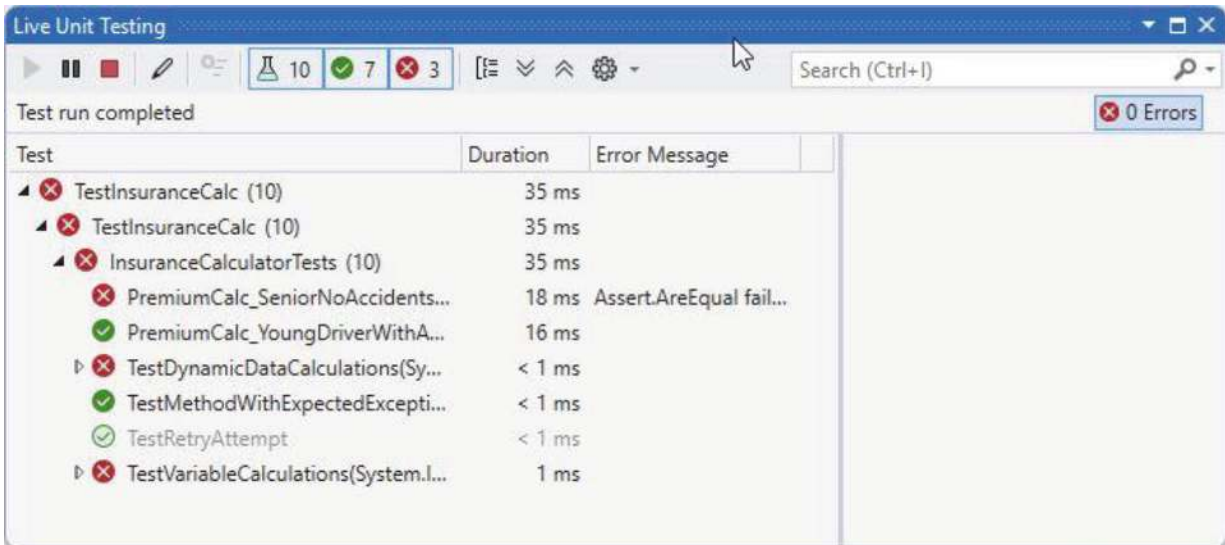
        if (accidents == 0)
            basePremium -= 200; // safe driver discount
        else
            basePremium += 100;

        // Minimum premium enforcement
        if (basePremium < 300)
            basePremium = 300;

        return basePremium;
    }
}
```

**Figure 4-22** Making a Breaking Change

A breaking change will immediately be visualized in the code editor as seen in Figure [4-22](#). With Live Unit Testing, areas of code indicated by a dash are not covered by any tests. A green tick indicates that the code is covered by a passing test. A red X indicates that the code is covered by a failing test. You will also see that the Live Unit Testing window is updated to display the failing tests as shown in Figure [4-23](#).



**Figure 4-23** Live Unit Testing Results Updated

Live Unit Testing provides real-time insight into the stability of your code as you write it.

## Using Copilot to Generate Unit Tests

GitHub Copilot can assist developers when creating Unit Tests for their source code. In the `InsuranceCalculator` class I have a method that determines the risk category of an individual based on age and the number of previous accidents they have been involved in. You can see this code in Listing [4-12](#).

```
public static string GetRiskCategory(int age, int
accidents)
{
    if (age < 18)
        throw new ArgumentException("Age must be 18 or
older.", nameof(age));
    if (accidents < 0)
        throw new ArgumentException("Accident count
cannot be negative.", nameof(accidents));

    if (accidents == 0 && age >= 25 && age <= 50)
        return "Low";
    if (accidents <= 1 && age >= 18 && age < 25)
        return "Medium";
    if (accidents >= 2 || age > 70)
```



```

        return "High";
    return "Medium";
}

```

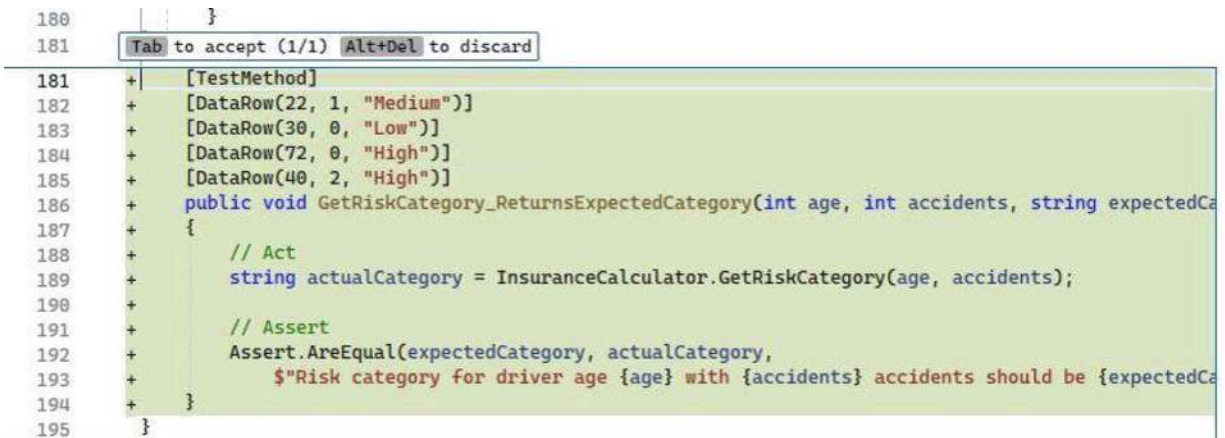
**Listing 4-12** The Risk Category Method

The code is straightforward, but I want to be sure that I have it sufficiently covered with a Unit Test. Open the `InsuranceCalculatorTests` class and position your cursor where a new test method should go. Invoke Copilot's inline chat by pressing `Alt+/` or right-clicking and selecting *Ask Copilot* from the context menu, as seen in Figure 4-24.



**Figure 4-24** Invoking Copilot Inline Chat

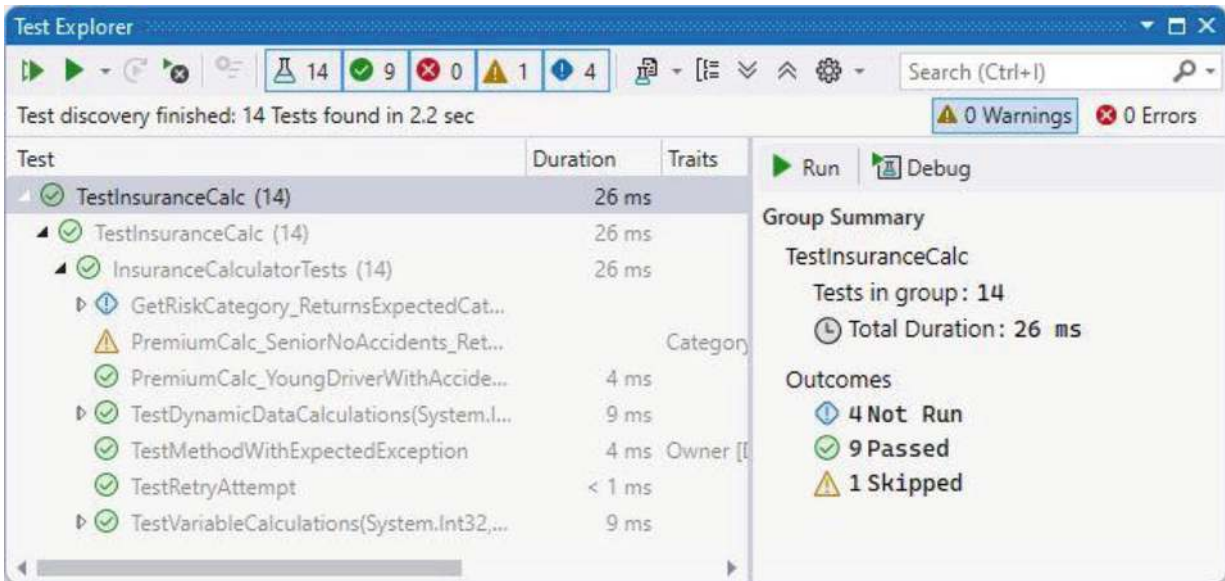
You can now prompt Copilot to create a suitable unit test for your `GetRiskCategory` method. The inline chat allows you to change the LLM, but as seen in Figure 4-24, I am just using the default (at the time of this writing) GPT-4.1 with its large context window.



**Figure 4-25** Copilot Generated Unit Test

Copilot will generate a Unit Test for you and display that as seen in Figure 4-25.

You can press (or click on) the `Tab` button to accept the change or `Alt+Del` to discard the suggestion from Copilot. You will see the newly added test displayed in the Test Explorer and indicated with a blue exclamation icon, as seen in Figure 4-26.



**Figure 4-26** Updated Test Explorer

As before, Test Explorer provides a centralized view to monitor test execution. You can run the newly added test to verify that it passes.

The example above illustrates using Copilot Inline chat to create a Unit Test, but you can do the same thing using the GitHub Copilot Chat window, as seen in [Figure 4-27](#).

By invoking the slash command, you can ask Copilot to create a unit test for the selected code in your editor window.



**Figure 4-27** Using Copilot Chat

This enables you to perform the same action directly within the Copilot Chat window. You may be wondering whether Copilot can assist with specific use cases, such as generating documentation.

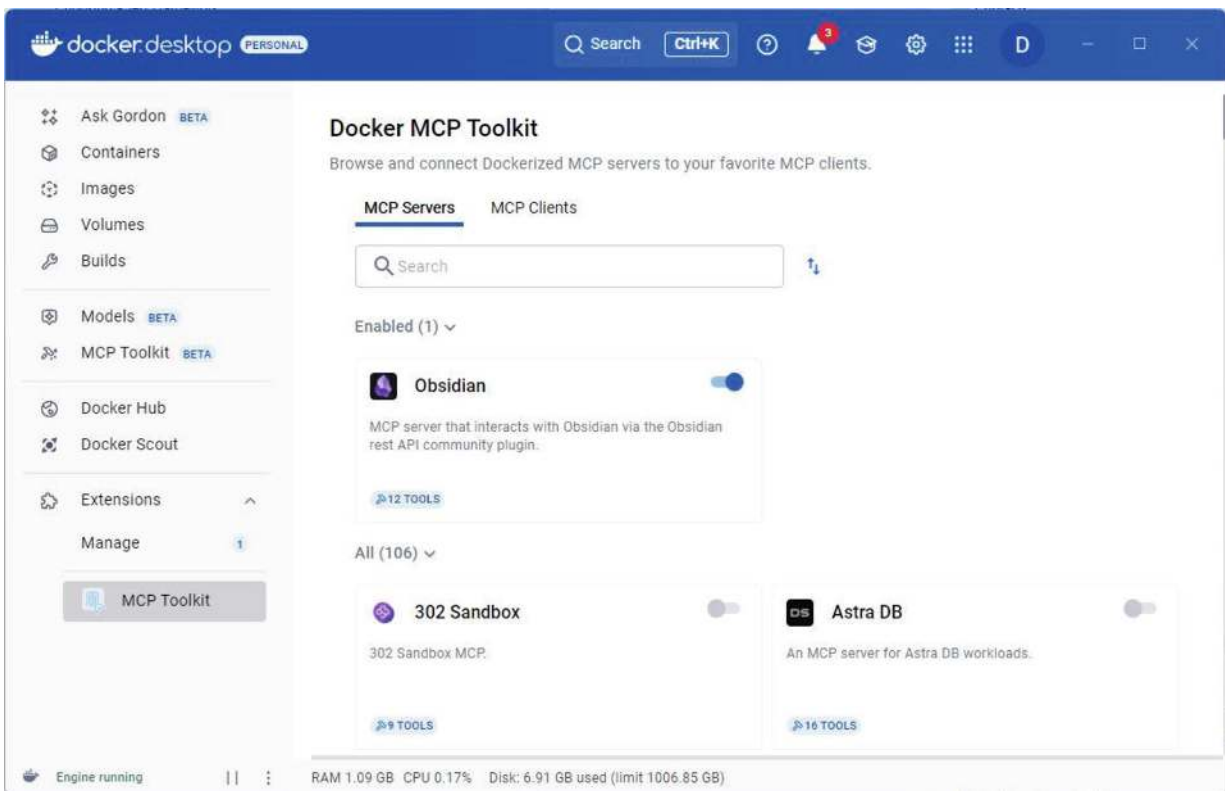
Documenting unit tests is a common task for developers, though it is rarely a favorite. In the next section, we will explore how to leverage AI to help

document your unit tests, or any code file, efficiently and accurately.

## Document Unit Tests with Obsidian MCP Server

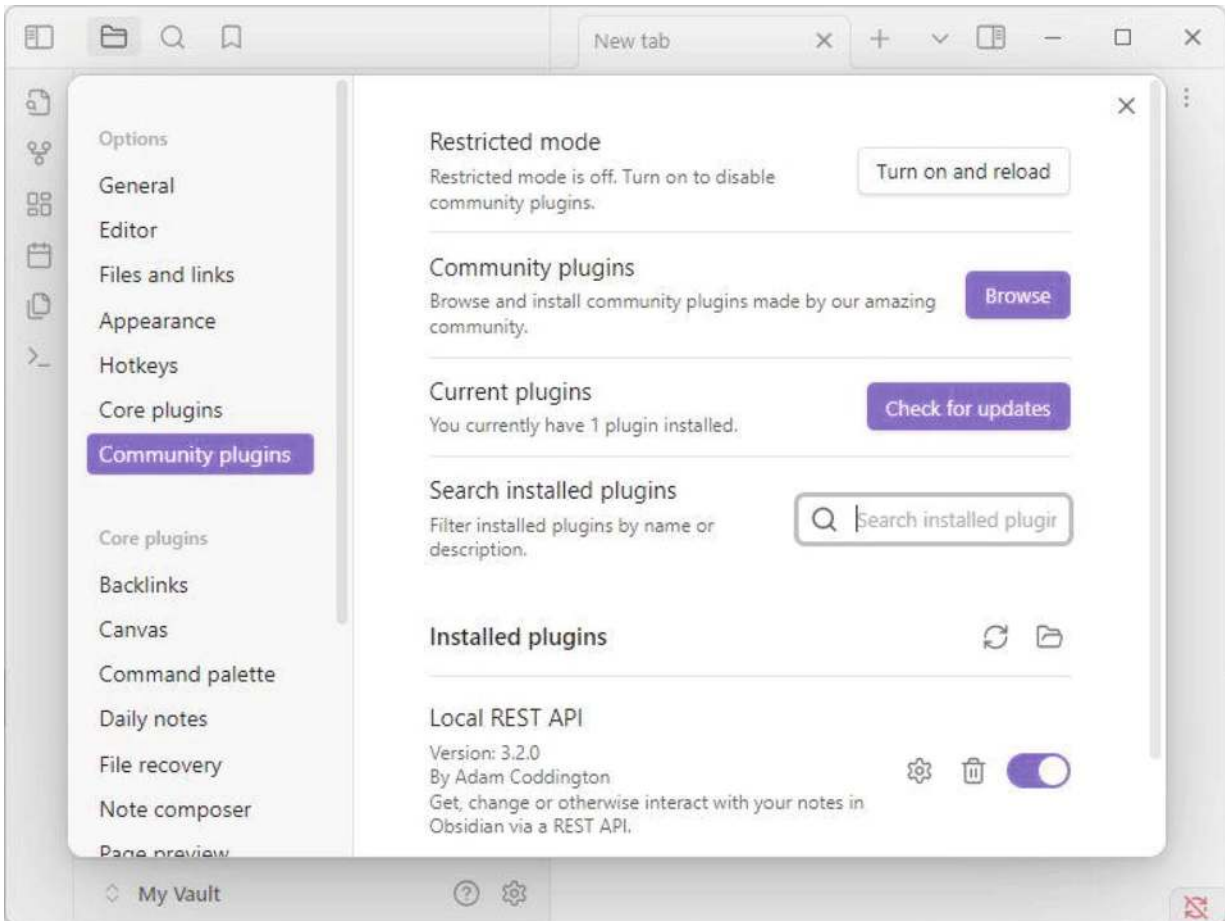
You are able to integrate Obsidian with Visual Studio through the use of an MCP (Model Context Protocol) Server. This allows you to leverage the power of GitHub Copilot Agent to document your unit tests (or any code for that matter) directly into your Obsidian knowledge base. Using this approach, you not only centralize your documentation but you also enable AI-assisted updates as your unit tests evolve over time.

The process begins by ensuring that you have Docker installed and ready to run containers. In Docker, install the Docker MCP Toolkit (Figure [4-28](#)).



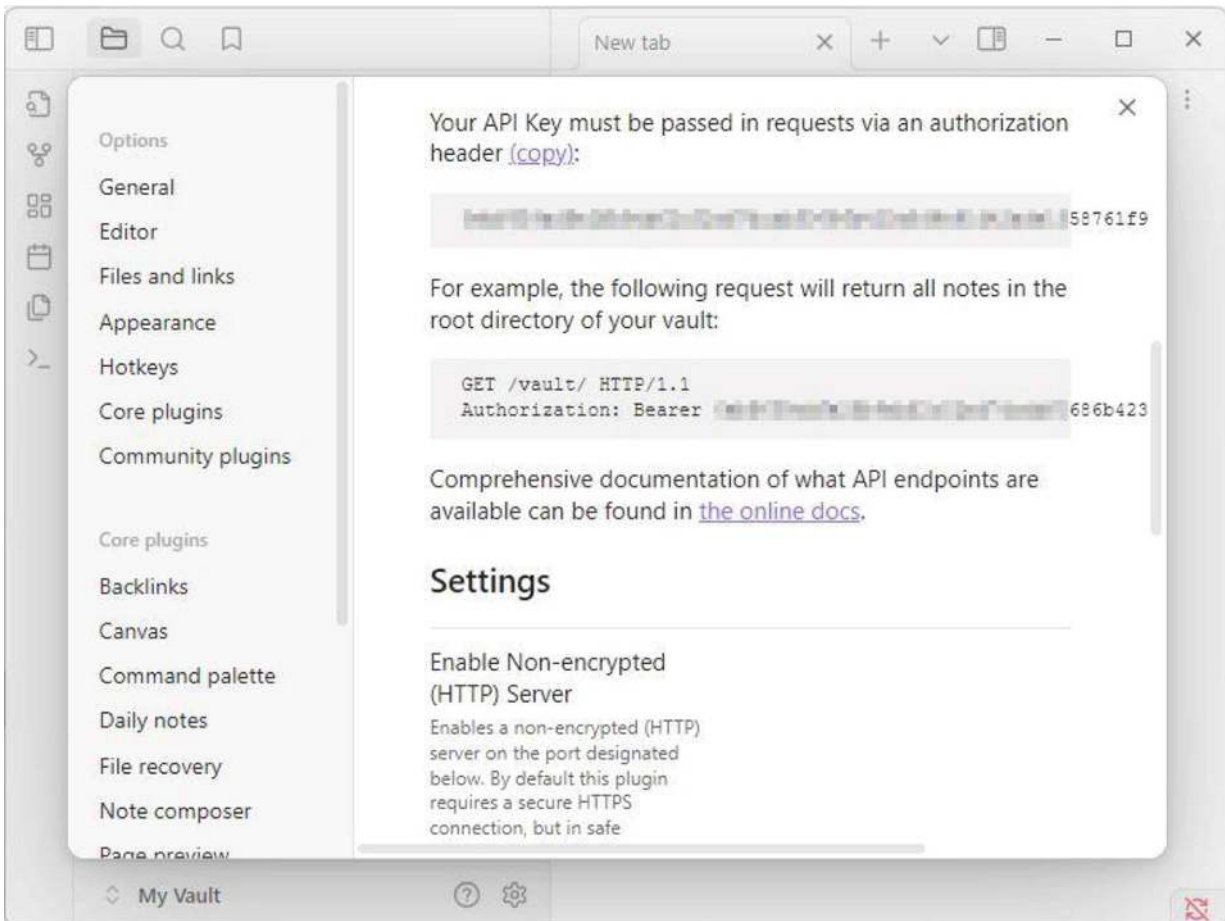
**Figure 4-28** MCP Toolkit in Docker

This toolkit provides the infrastructure required to host and run MCP servers, including the one we will need for Obsidian. With the toolkit installed, you can enable the Obsidian MCP Server. Since the server relies on Obsidian, you must have Obsidian installed locally. In Obsidian, open the Community Plugins section and install the Local REST API Plugin (Figure [4-29](#)). This plugin exposes an HTTP API for interacting with your vault, which the MCP server will utilize to create and update notes.



**Figure 4-29** Obsidian Local REST API Plugin

After installation, navigate to the plugin's settings and get your API key as seen in [Figure 4-30](#). This key is critical as it authenticates requests from the MCP server to your Obsidian vault.

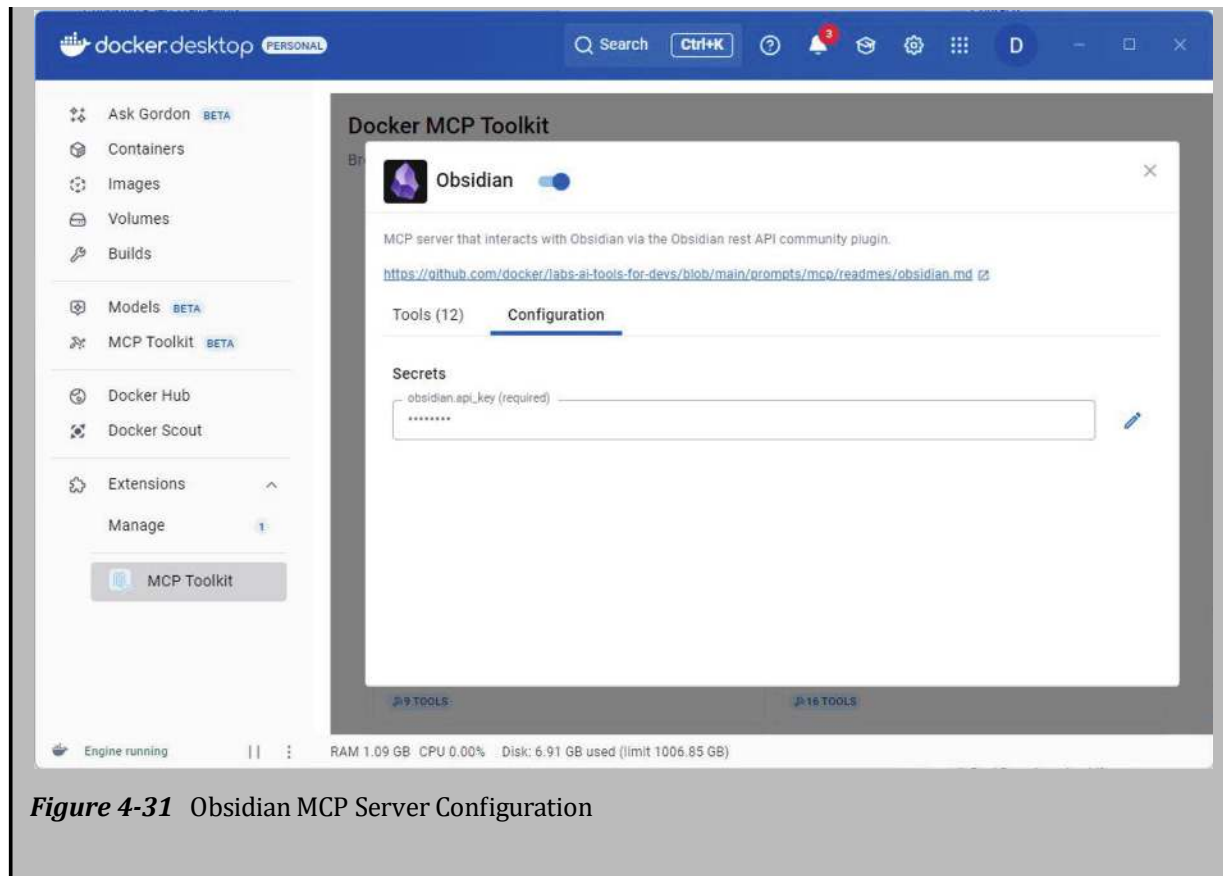


**Figure 4-30** Obsidian API Key

Back in Docker, add the API key to the Obsidian MCP server configuration shown in Figure 4-31.

With the version 17.14.13 August release of Visual Studio 2022, MCP support became GA in Visual Studio. Released after the publication of this chapter, MCP support became a first-class experience in Visual Studio 2022. No more copying JSON snippets into .mcp.json files. This is another example of the rapidly evolving landscape of AI tools and Visual Studio in general.





**Figure 4-31** Obsidian MCP Server Configuration

Above the *Configuration* tab in Docker, click the URL to open the Obsidian.md page on GitHub and scroll down until you locate the JSON configuration for the MCP server. Copy this JSON, which resembles the code in Listing 4-13.

```
{
  "mcpServers": {
    "obsidian": {
      "command": "docker",
      "args": [
        "run",
        "-i",
        "--rm",
        "-e",
        "OBSIDIAN_HOST",
        "-e",
        "OBSIDIAN_API_KEY",
        "mcp/obsidian"
      ],
    },
  },
}
```

```

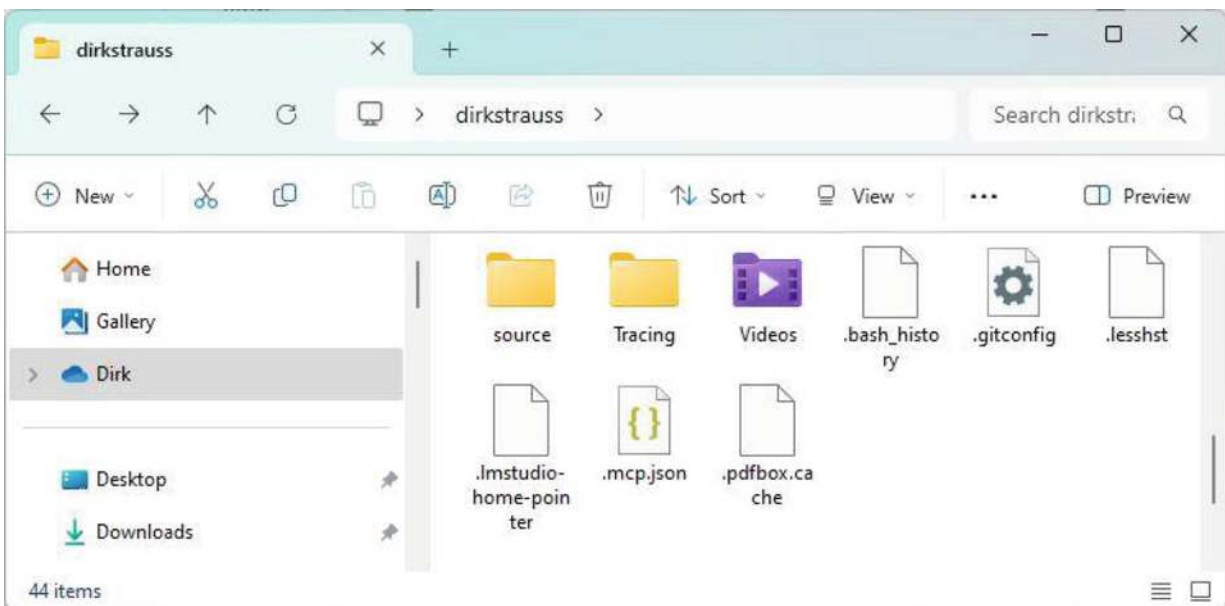
    "env": {
      "OBSIDIAN_HOST": "host.docker.internal",
      "OBSIDIAN_API_KEY": "YOUR_OBSIDIAN_API_KEY"
    }
  }
}

```

**Listing 4-13** MCP Server Config JSON

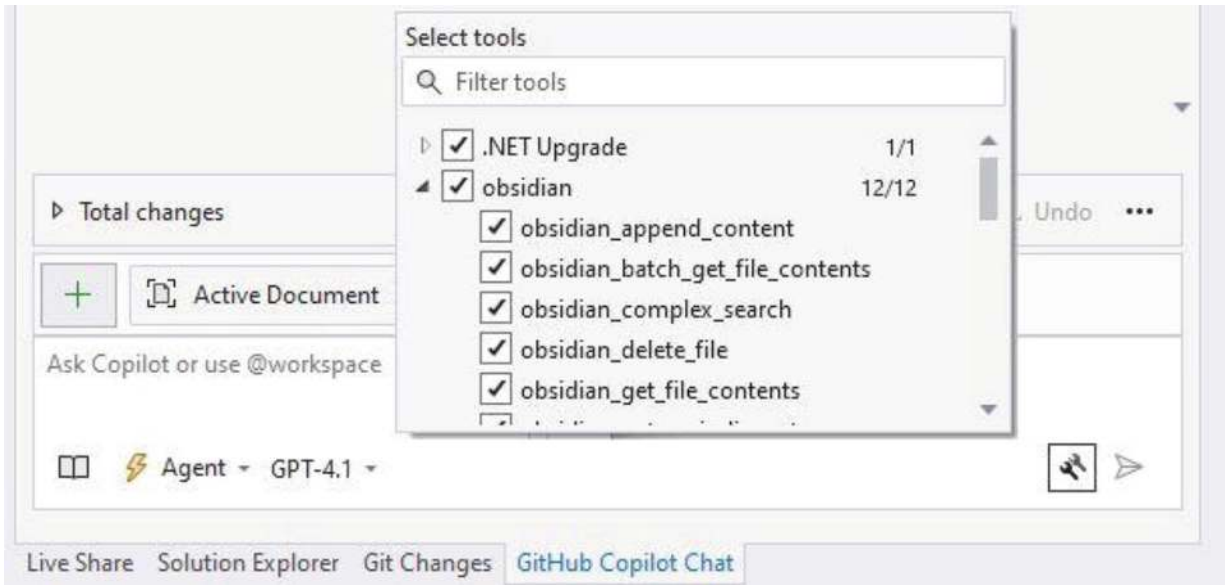
This JSON defines the MCP server for Obsidian, passing in the host address and the unique API key (copied earlier) via environment variables.

Once you have this JSON, open Windows Explorer and navigate to %USERPROFILE%. In this directory, create a new file named `.mcp.json` (note the leading period) as shown in Figure 4-32. Paste the JSON from Listing 4-13 into this file.



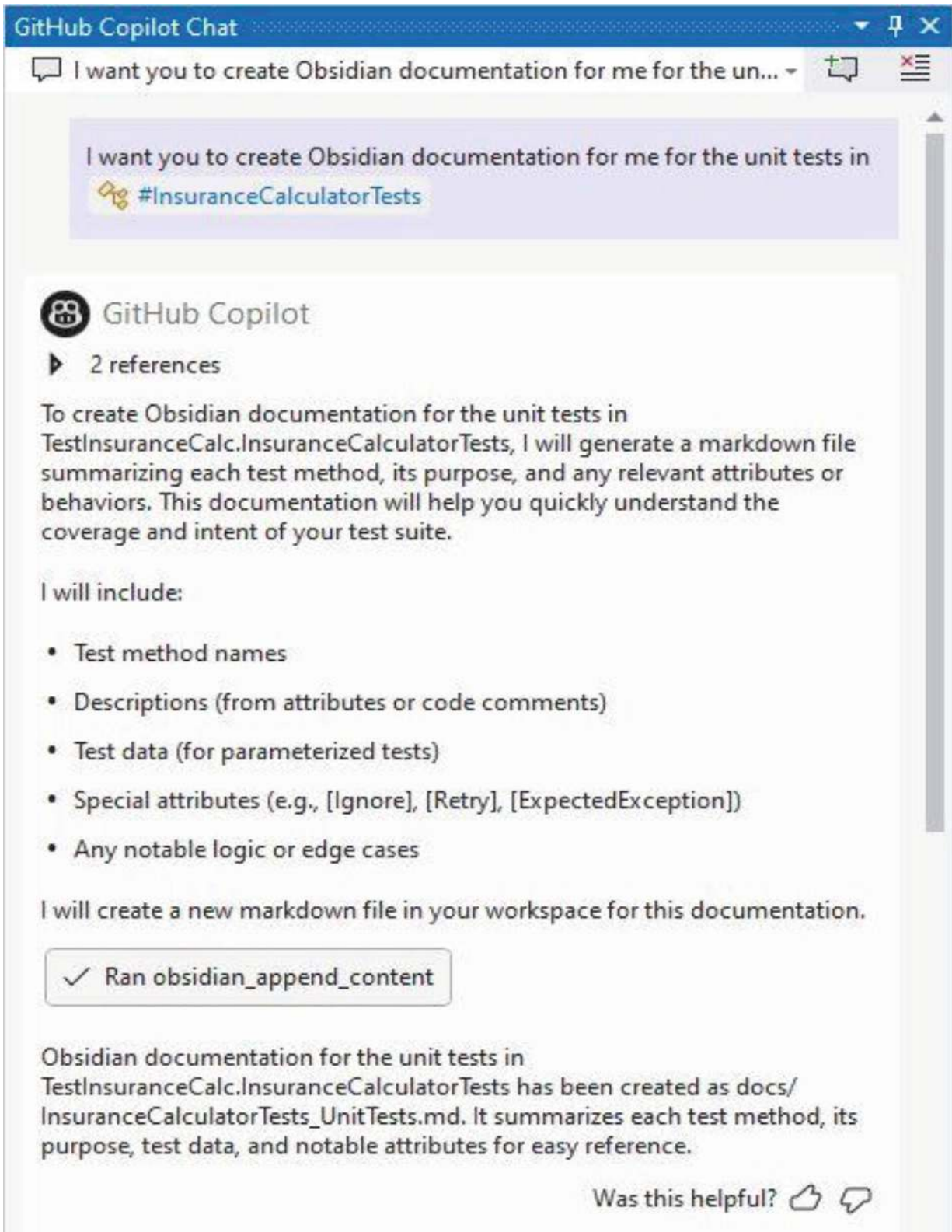
**Figure 4-32** The .mcp.json File

If you already have other MCP servers configured, append this JSON to the existing `mcpServers` section (you can define multiple MCP servers in this file). Replace `YOUR_OBSIDIAN_API_KEY` with the key you copied from your Obsidian settings earlier.



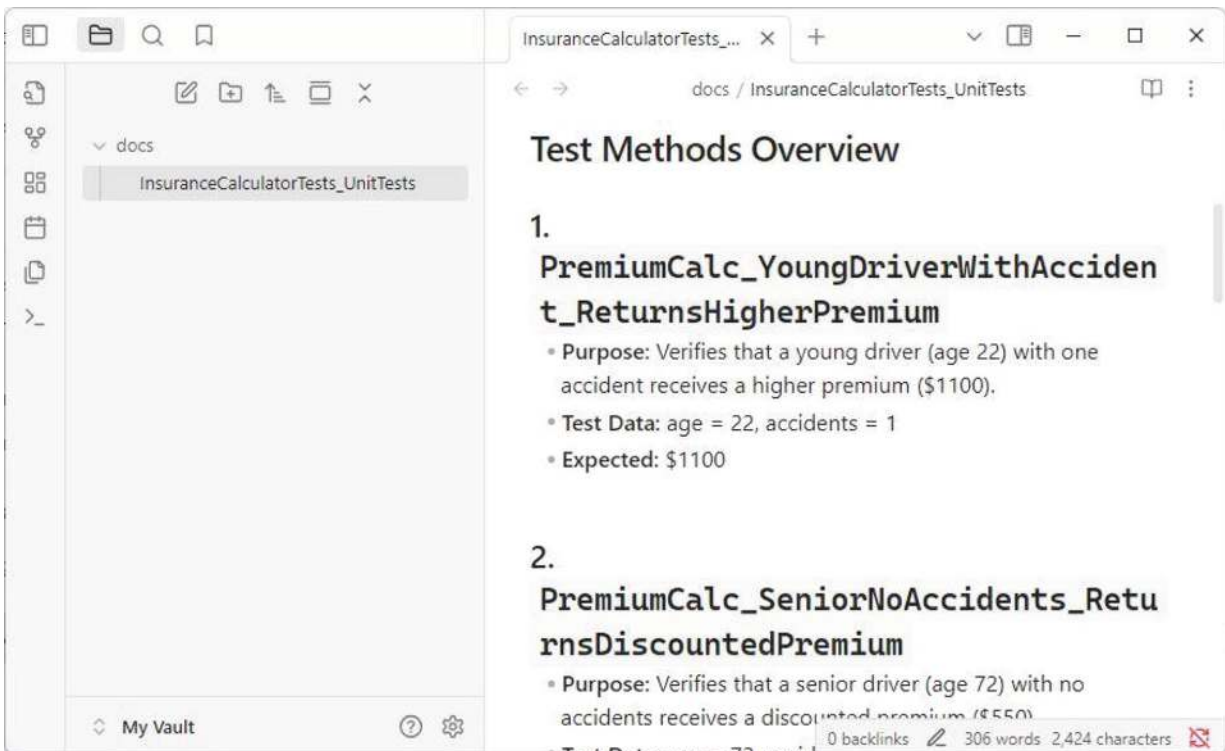
**Figure 4-33** Obsidian Tool Selection

After saving the file, restart Visual Studio to load the updated MCP configuration. Switch to GitHub Copilot Agent mode as shown in Figure [4-33](#). Clicking on the tools icon, you will see the Obsidian tools that are now available to you. This allows you to select the specific Obsidian tools Copilot can use.



**Figure 4-34** Prompting Copilot

From this point, you can prompt GitHub Copilot to document your unit tests directly into Obsidian, as shown in Figure 4-34. Copilot will send structured documentation to your Obsidian vault, creating or updating notes accordingly.



**Figure 4-35** Generated Unit Test Documentation

Once the process has completed, open Obsidian to review the generated unit test documentation (Figure 4-35). The advantage of this setup is the iterative nature that you can implement. You can quickly regenerate or update documentation with a single Copilot prompt, ensuring that your test documentation remains synchronized with the unit tests you wrote. No more manual copy-and-paste updates.

---

## How to Measure Code Coverage in Visual Studio

Code Coverage indicates what portion of your code is covered by Unit Tests. To guard against bugs, it becomes obvious that the more code is covered by Unit Tests, the better tested it is.

This feature is only available in Visual Studio Enterprise edition.

The Code Coverage feature in Visual Studio will give you a good idea of your current Code Coverage percentage. To run the Code Coverage analysis, open Test Explorer, and click the drop-down next to the play button (Figure 4-36).

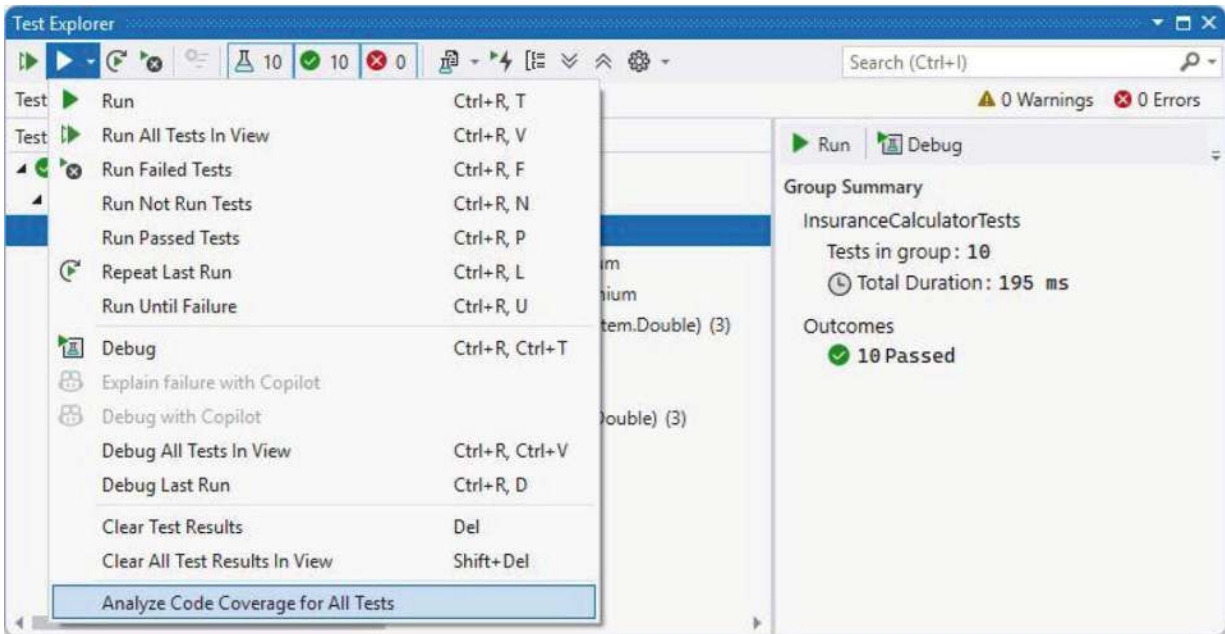


Figure 4-36 Analyze Code Coverage

Click *Analyze Code Coverage for All Tests* in the menu. The Code Coverage Results are then displayed in a new window (Figure 4-37). You can access this window from the **Test** menu and then select **Code Coverage Results** or hold down Ctrl+E, C on the keyboard.

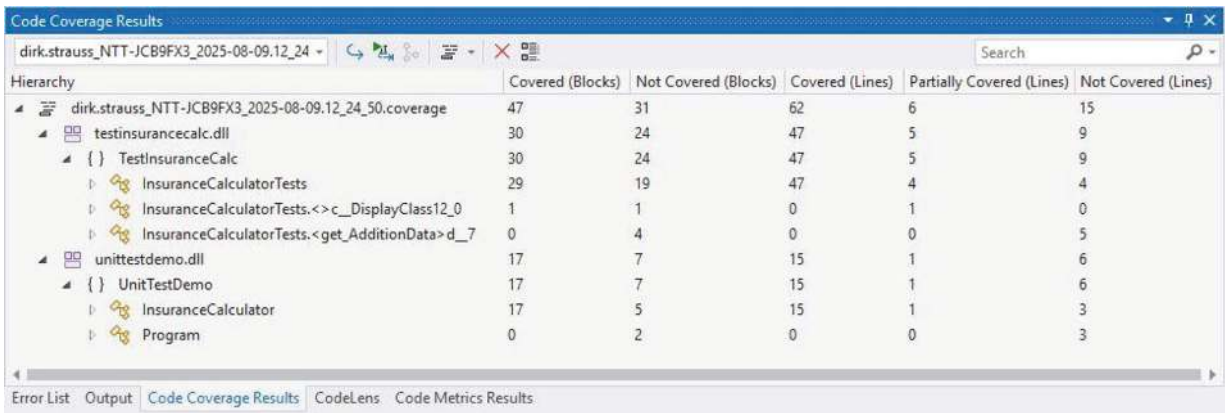
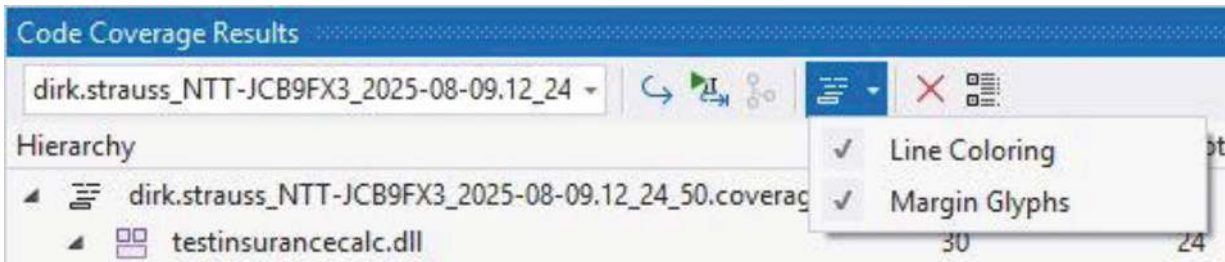


Figure 4-37 Code Coverage Results

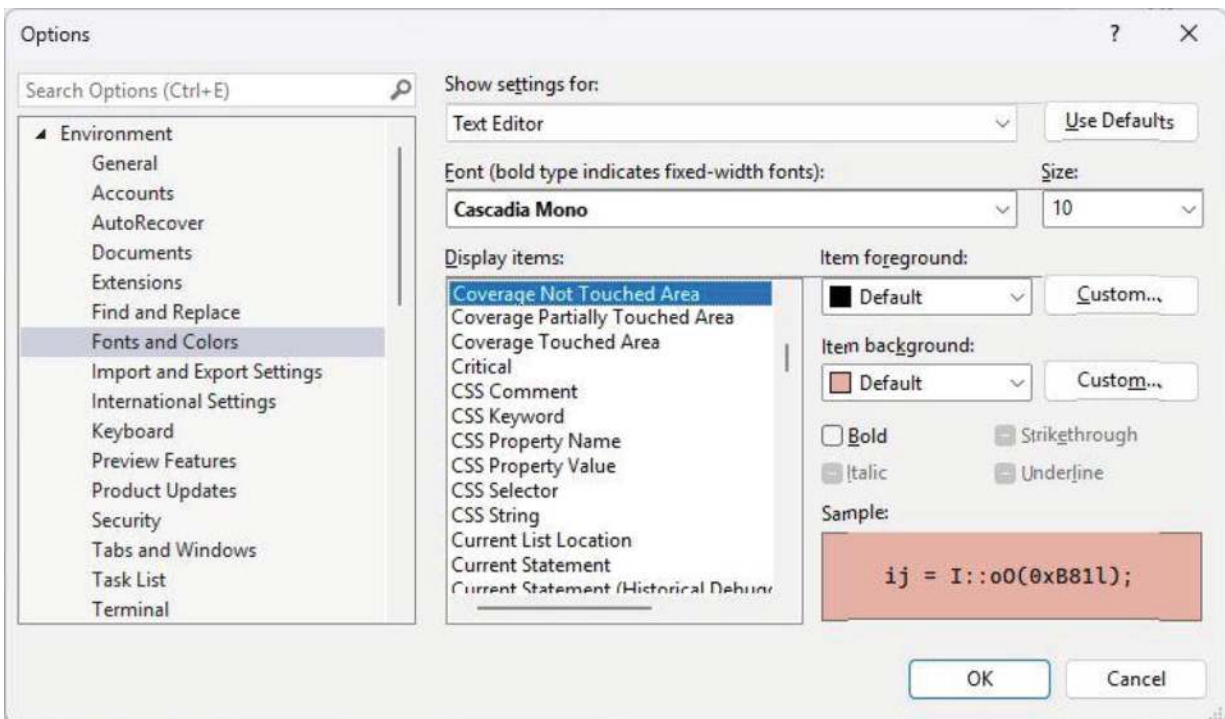
In the Code Coverage Results window, you can Export the results, Import Results, Show Code Coverage Color (Figure 4-38), or Remove the results.





**Figure 4-38** Toggle Code Coverage Coloring

This will toggle colors in your code editor to highlight areas touched, partially touched, and not touched at all by tests. The colors used to highlight the code can also be changed. To do this, head on over to Tools ► Options ► Environment ► Fonts and Colors (Figure 4-39).



**Figure 4-39** Change Fonts and Colors

This should give you a good understanding of how much code is covered by unit tests. Developers should typically aim for at least 80% Code Coverage. If the Code Coverage is low, then modify your code to include more tests. Once you are done modifying your code, run the Code Coverage tool again as the results are not automatically updated as you modify your code.

Code Coverage is typically measured in blocks. A block of code is a section of code that has exactly one entry point and one exit point. If you prefer to see

the Code Coverage in terms of lines covered, you can change the results by choosing Add/Remove Columns in the results table header (Figure 4-40).

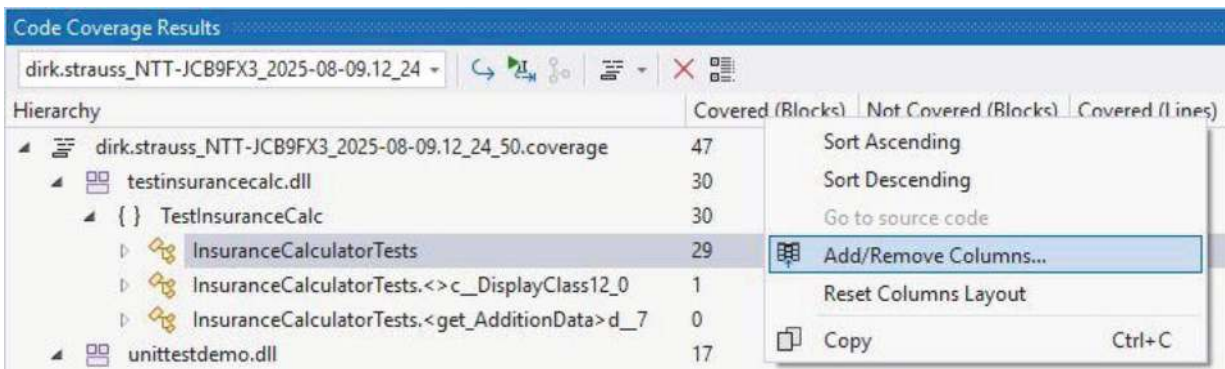


Figure 4-40 Code Coverage Expressed in Lines

Code Coverage is a great tool to allow you to check if your code is sufficiently covered by unit tests. If you aim for 80% Code Coverage, you should be able to produce well-tested code. The 80% Code Coverage is not always attainable. This is especially true if the code base you’re working on has a lot of generated code. In instances such as these, a lower percentage of code cover is acceptable.

## Summary

This chapter explored how unit testing safeguards code quality, catches errors early, and supports safe refactoring. We began by creating and running tests in Visual Studio and running them in Test Explorer. We had a look at the fundamentals of MSTest such as `[TestClass]` and `[TestMethod]` attributes. We discussed setup and teardown methods at various scopes as well as data-driven tests using `[DataRow]` and `[DynamicData]`. Execution control techniques were also discussed such as timeouts, retrying failed tests, and skipping tests with `[Ignore]`.

The chapter then examined advanced capabilities such as

- Live Unit Testing for real-time feedback and coverage indicators
- GitHub Copilot integration to create unit tests via inline chat or chat prompts
- Obsidian MCP Server integration to document unit tests into an Obsidian knowledge base

It concluded with measuring code coverage in Visual Studio Enterprise, interpreting results, and customizing coverage highlighting. Together, these

tools and techniques provide a robust framework for ensuring that developers ship quality code with confidence.

In the next chapter, we will be looking at a feature that all developers should be very familiar with. Source control management is essential to any project. We will explore this and some new features of Visual Studio 2022 such as multi-repo support, comparing branches, Check out Commit, and line staging, but to name a few. Let's look at source control next.

## 5. Source Control

Dirk Strauss<sup>1</sup> 

(1) Uitenhage, South Africa

---

If you have worked on projects in a team environment, or if you need a place to keep your own code safe, then you'll agree that using a source control solution is essential. It doesn't matter if it's a large enterprise solution or a small personal project, Visual Studio makes it extremely easy for developers to use Git and GitHub.

Git is a distributed version control system that developers install locally to track changes to their code. GitHub is a cloud-based hosting service for Git repositories, enabling developers to store their code online, collaborate with others, and manage projects using the Git tool.

In 2018, Microsoft acquired GitHub for \$7.5 billion in Microsoft stock. This acquisition of GitHub brought about changes to their pricing tiers. Previously, developers could only create public repos on the free tier. In January 2019, however, GitHub announced that developers can now create unlimited private repositories on the free tier.

This is great, especially if you are working on a side project that you do not want to share with anyone just yet. In this chapter, we will be looking at using Git and GitHub inside Visual Studio 2022. We will see how to

- Create a GitHub account
- Create and clone a repository
- Commit changes to a repository
- Create a branch from your code

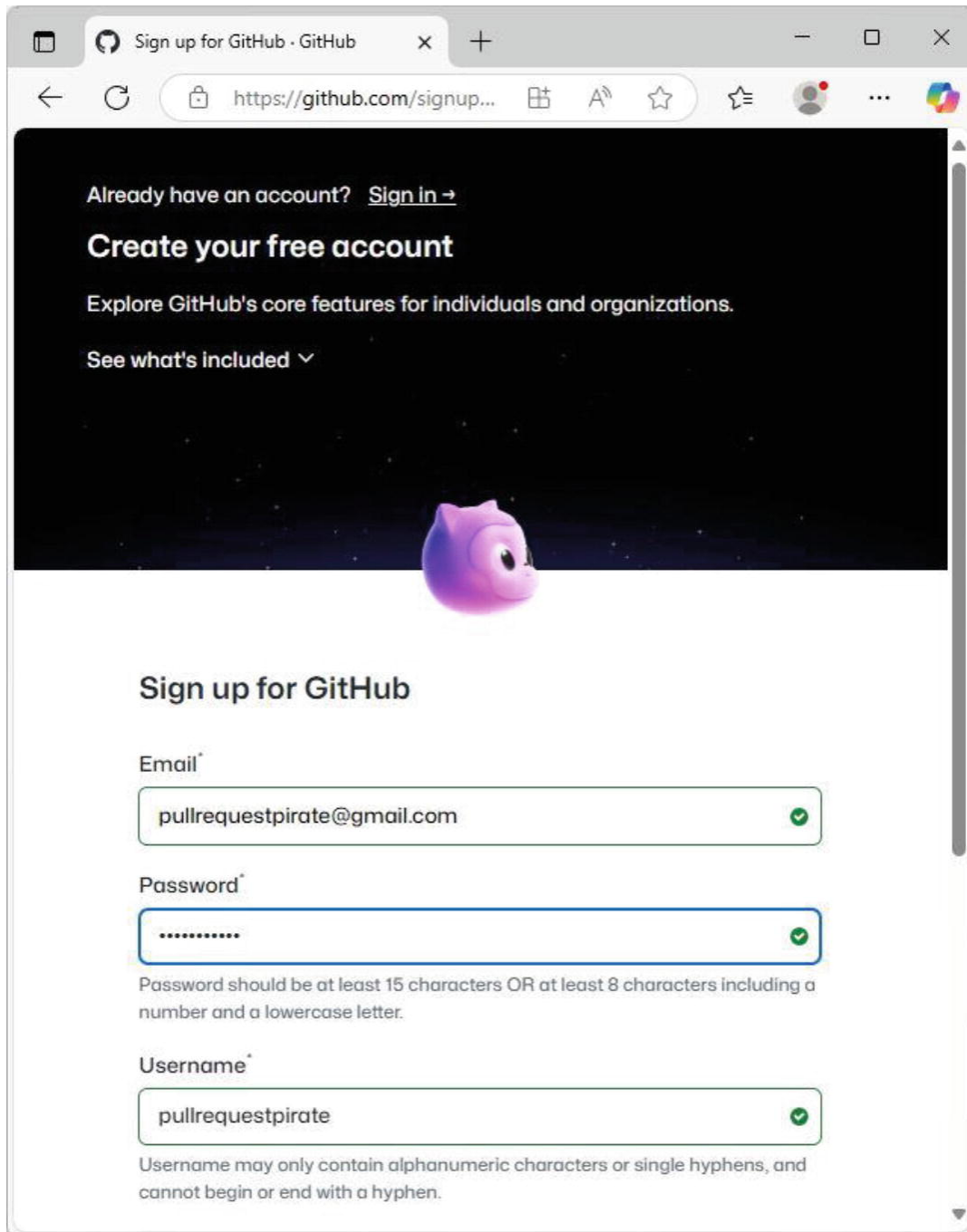
- Create and handle pull requests

These are all things that developers will do on a daily basis when working with Git and GitHub. While the process might change slightly if you use a different source control strategy, the concepts remain the same.

---

## Create a GitHub Account

Let's start with creating a GitHub account. Point your browser to [www.github.com](https://www.github.com), as shown in Figure [5-1](#), and create an account by clicking the signup button.

A screenshot of a web browser showing the GitHub sign-up page. The browser's address bar displays 'https://github.com/signup...'. The page has a dark header with the text 'Already have an account? [Sign in →](#)' and 'Create your free account'. Below this, it says 'Explore GitHub's core features for individuals and organizations.' and 'See what's included ▾'. A small purple cat-like character is visible in the background. The main form is titled 'Sign up for GitHub' and contains three input fields: 'Email\*' with the value 'pullrequestpirate@gmail.com', 'Password\*' with masked characters, and 'Username\*' with the value 'pullrequestpirate'. Each field has a green checkmark icon to its right. Below the password field, there is a note: 'Password should be at least 15 characters OR at least 8 characters including a number and a lowercase letter.' Below the username field, there is a note: 'Username may only contain alphanumeric characters or single hyphens, and cannot begin or end with a hyphen.'

**Figure 5-1** Sign up for GitHub

Enter a username (Figure 5-1), email address, and password. GitHub then checks your password to verify that it does not appear on a list of known compromised passwords. If all checks out, a confirmation code is sent to your email address. Once you verify your email address,



GitHub signs you in to your account. From there, you can modify your profile and complete your account setup.

You can continue using the free account, but there is also the option to upgrade to a Team or Enterprise account. The free subscription offers the basics suitable for most developers, while the Team subscription offers several more features aimed at advanced collaboration for individuals and organizations. The free subscription is very generous and will appeal to most developers. The free subscription includes the following, among others:

- Unlimited public and private repositories
- Unlimited collaborators for public and private repositories
- Issues – Track bugs, enhancements, and other requests
- Project, Milestones, and Team discussions
- Pages and wikis for public repos
- GitHub Actions (2000 minutes/month) or free for public repos

The Team and Enterprise subscriptions, on the other hand, offer more, which includes

- Unlimited public and private repositories
- Unlimited collaborators for public repositories, while private repositories incur the subscription pricing per user per month
- Issues
- Project, Milestones, and Team discussions
- Insights into security overview
- GitHub Actions (3000 minutes/month for Team and 50,000 minutes/month for Enterprise) or free for public repos

GitHub Team and Enterprise allow a team of developers to collaborate on projects, and GitHub bills for GitHub Team and Enterprise on a per-user basis.

For more info on all GitHub's pricing, browse to the following URL: <https://github.com/pricing#compare-features>

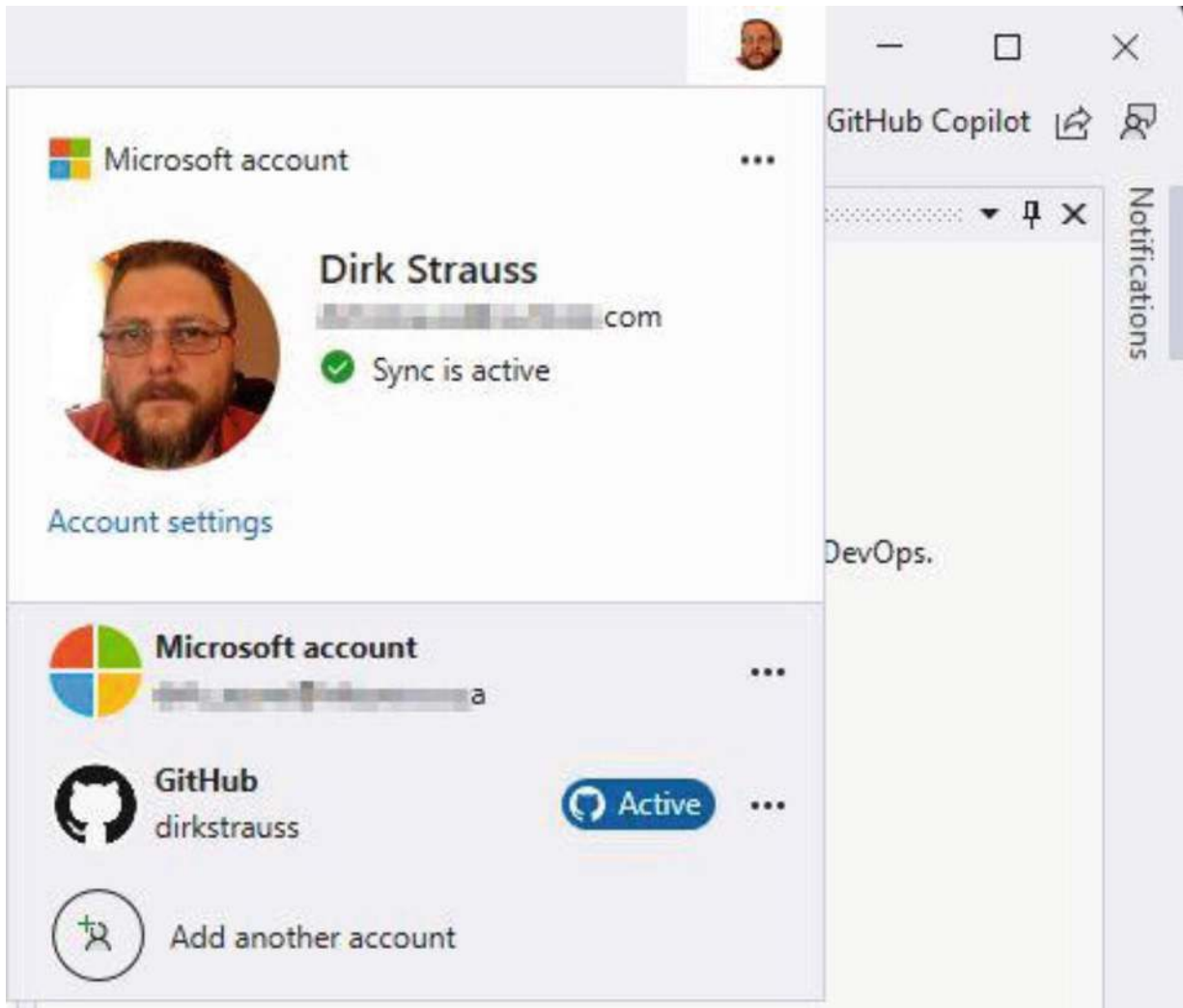
To follow along in the book, the free GitHub account will be sufficient.

---

## Improve GitHub Productivity with Multi-account Support

GitHub account management has undergone a significant overhaul in Visual Studio, making it much easier to work with multiple accounts across different projects and organizations. This improvement is particularly beneficial for developers who need to switch between personal and work-related GitHub profiles, or who contribute to various repositories under different credentials. You can now add, authenticate, and manage multiple GitHub accounts directly within Visual Studio without the constant need to sign out and back in again. Once your accounts are connected, Visual Studio remembers your authentication details and applies them seamlessly to operations such as cloning repositories, pushing changes, or creating pull requests.

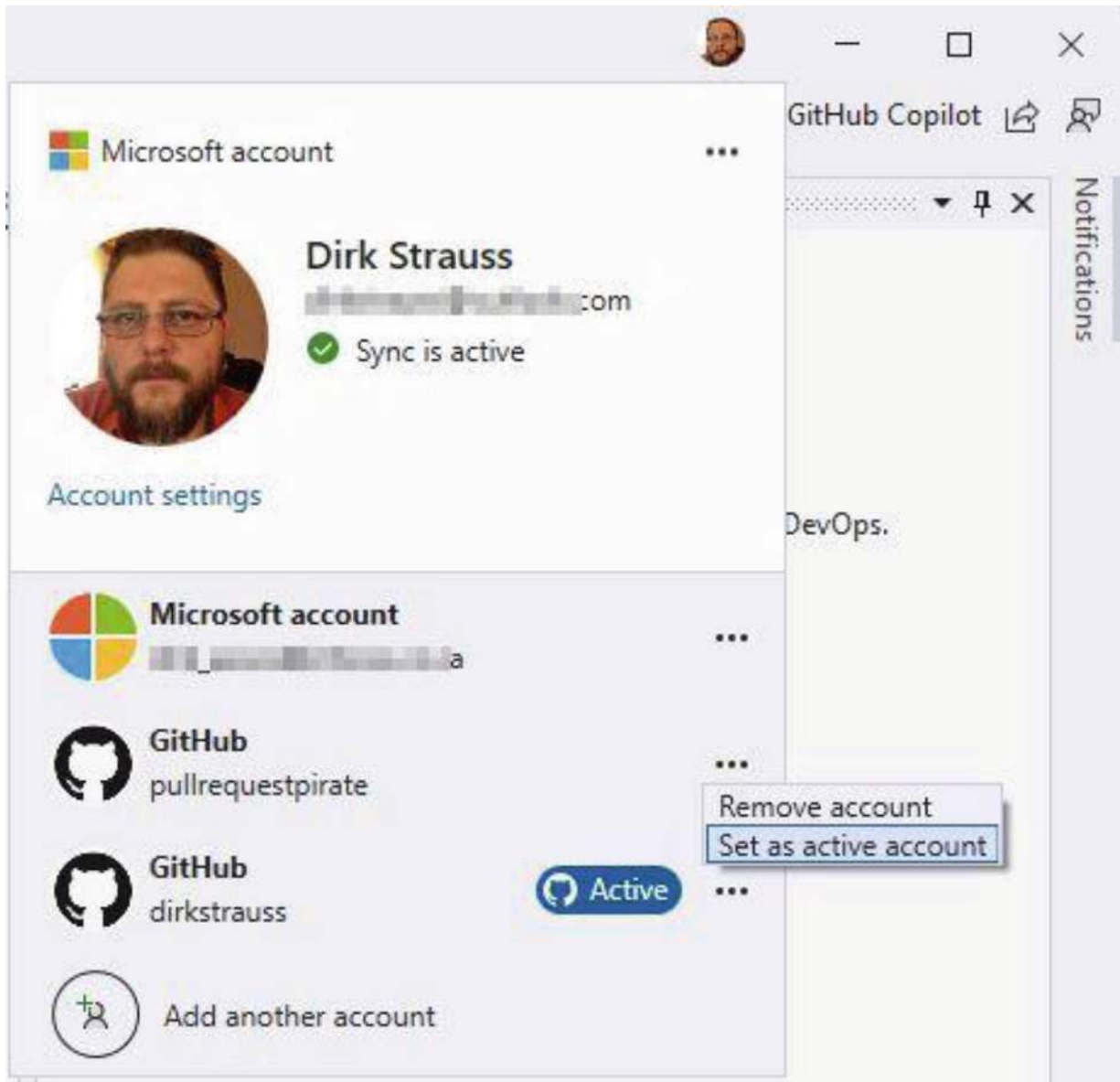
The integration is designed to be intuitive; simply click on your profile icon in the top-right corner of the Visual Studio window (as shown in Figure [5-2](#)) to view and manage all linked accounts. From here, you can quickly switch between accounts, check the connected organization, or Account settings. This streamlined approach not only saves time but also reduces friction when collaborating across multiple teams, ensuring that you're always working under the correct GitHub identity.



**Figure 5-2** Managing Multiple GitHub Accounts

At the bottom of this profile card, you can click on *Add another account*, sign in to your second GitHub account, and repeat as required.

The sign-in process is straightforward and once your account is added, you can switch between the accounts effortlessly by selecting *Set as active account* from the ellipses in the profile card.



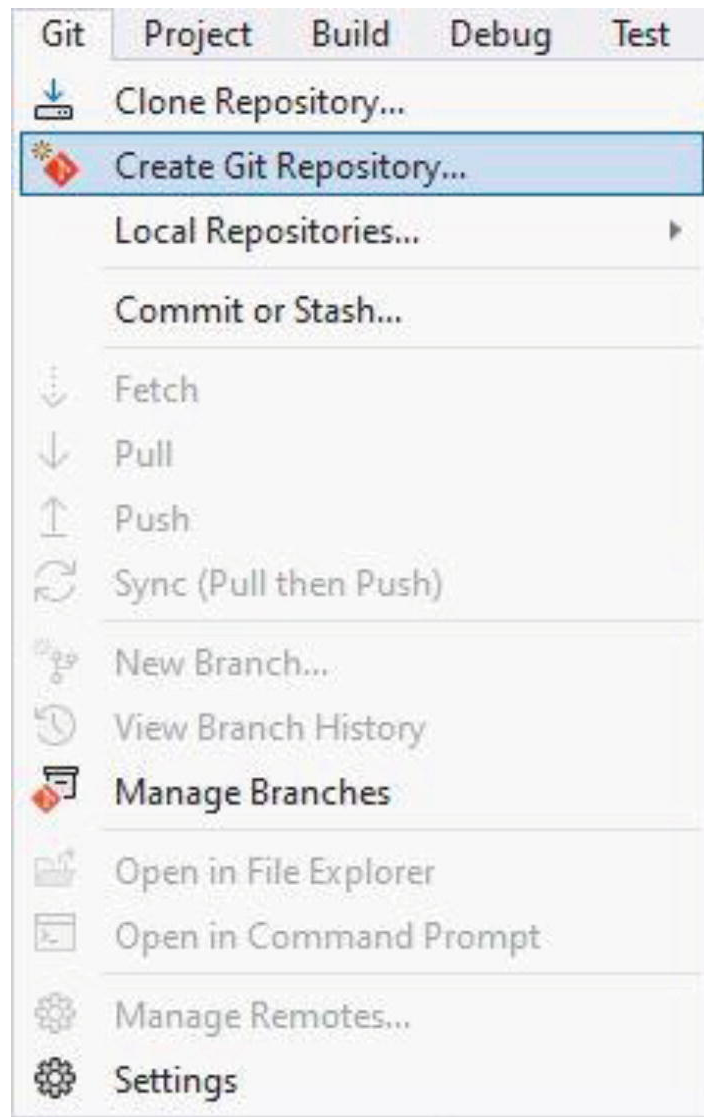
**Figure 5-3** Multiple GitHub Accounts

Just remember that the active GitHub account drives the activation of GitHub Copilot. Depending on your GitHub Copilot subscription, those features will automatically be activated as you switch accounts.

## Create and Clone a Repository

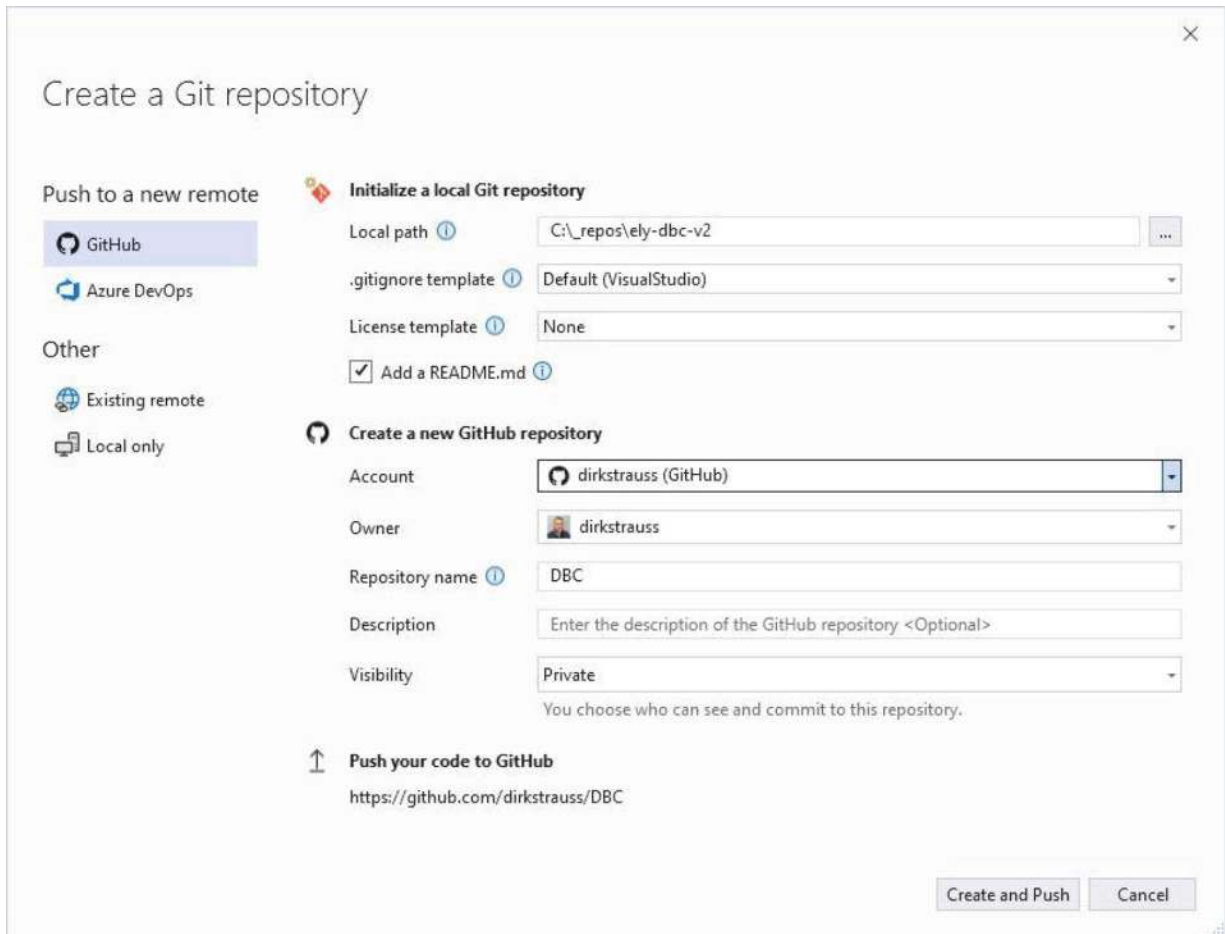
Visual Studio 2022 makes it extremely easy to create a repository on GitHub. Before I start writing actual code, I want to set up my GitHub

repo. From the menu in Visual Studio, select *Git* and click *Create Git Repository* as seen in Figure 5-4.



**Figure 5-4** Create Git Repository

You will then be presented with the Create a Git repository window as seen in Figure 5-5. The window will default the local path to the path that your project is currently saved in. I always add a README file to my repos and keep the rest set to the default options regarding the license template and .gitignore template.



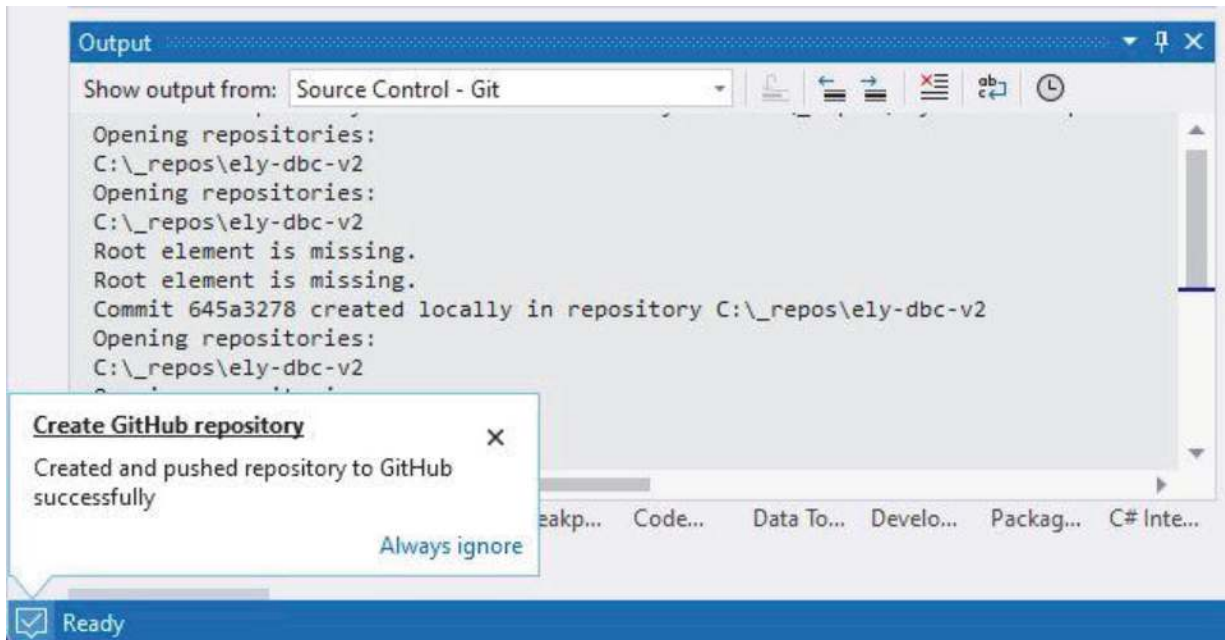
**Figure 5-5** Create a Git Repository

Because I already have a GitHub account, the Account drop-down will default to that account. You can also add a GitHub account from the drop-down or select another GitHub account if you have multiple accounts, as shown earlier in [Figure 5-3](#).

You can now click the Create and Push button that will create the repo on GitHub for you. This will also create a new local Git repository for your solution.

Open up the Output Window (Ctrl+Alt+O) from the View menu and click Output. Here, you will see that a new local Git repository has been created ([Figure 5-6](#)).



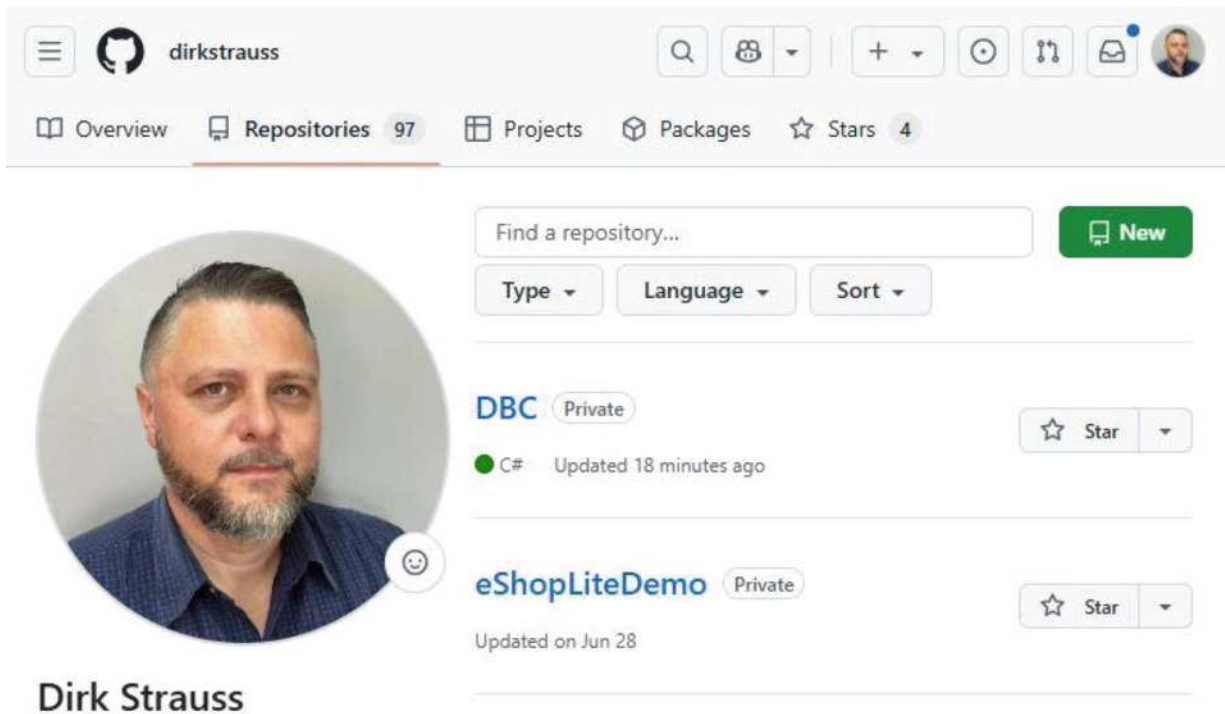


**Figure 5-6** New Git Repository Created

It is important to remember that this project is now under source control using Git. Remember that we mentioned earlier that Git is the source control plumbing, the tool that developers install locally on their machines.

If you never want a backup of your code in the cloud or never want to collaborate with other developers, you can just use Git. This is, however, quite an unlikely scenario, especially now that GitHub allows you free private repositories.

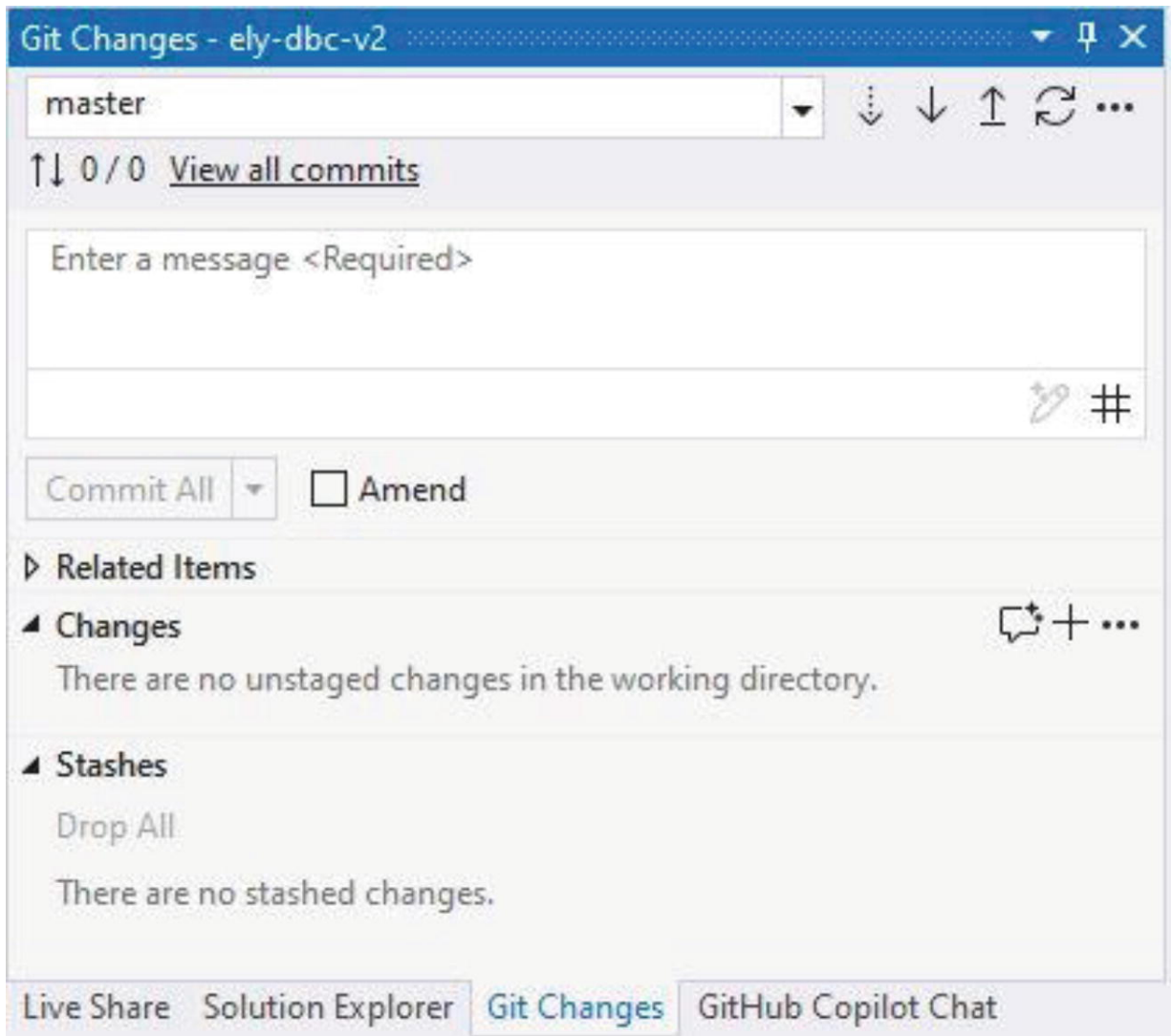
Therefore, you pushed your code to a GitHub repository after connecting your account. In my case, I can see that the repository has been created for me by going to my GitHub account, as seen in [Figure 5-7](#).



**Figure 5-7** The Created Repository on GitHub

The first repo listed will be the one I just pushed from Visual Studio, and you will find your repo here, too, under your GitHub profile.

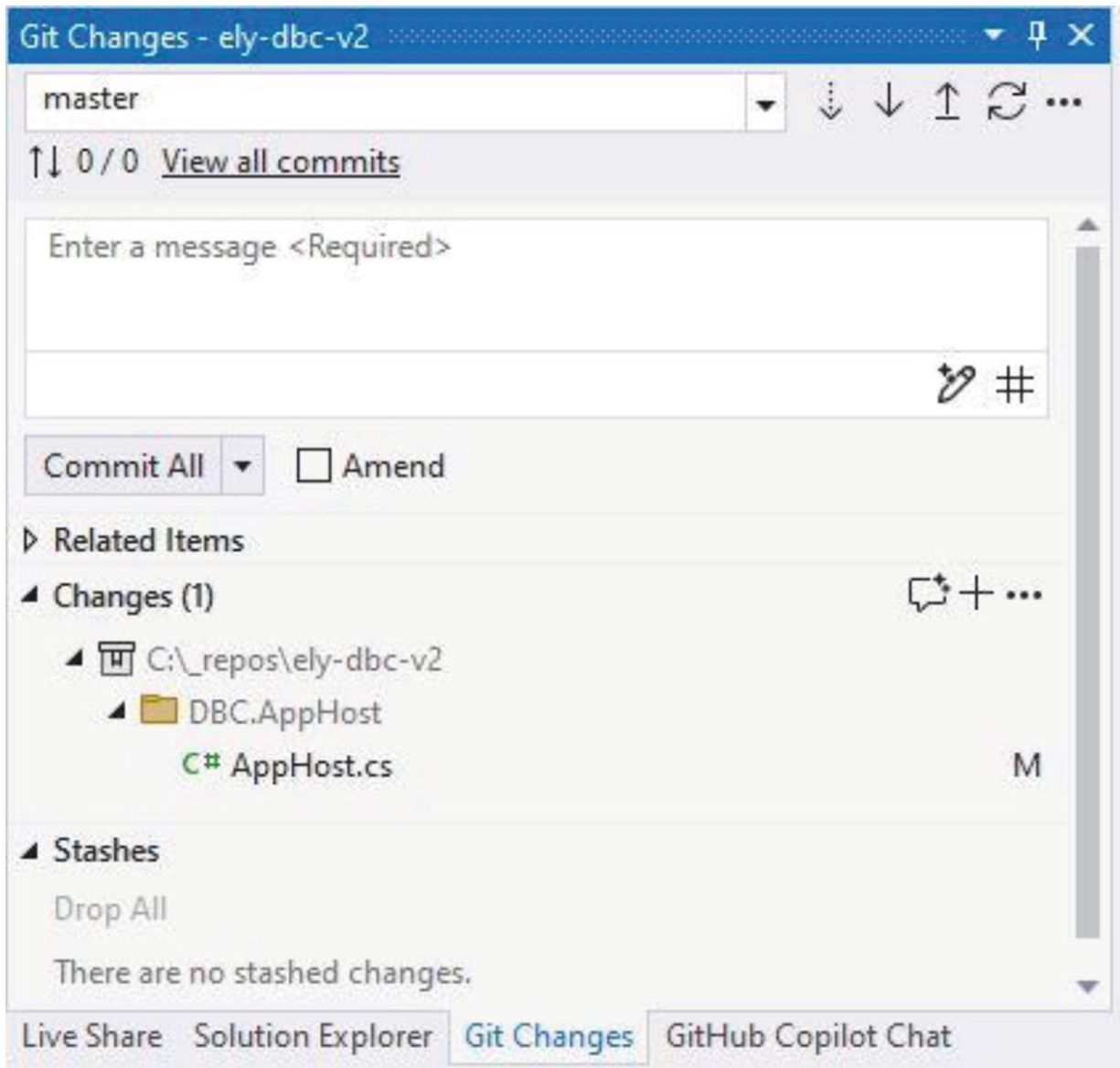
Next to the Solution Explorer (unless you have created a different window layout or docked the Git Changes somewhere else), you will see the Git Changes tab as displayed in Figure 5-8. Currently, there are no changes in our project to commit.



**Figure 5-8** The Git Changes

Make some UI or code changes to your project, and you will immediately notice that the Git Changes window (Figure 5-9) in Visual Studio reflects these updates in real time. Any file that has been modified, added, or deleted since your last commit will be listed here, giving you a clear and organized view of pending changes.

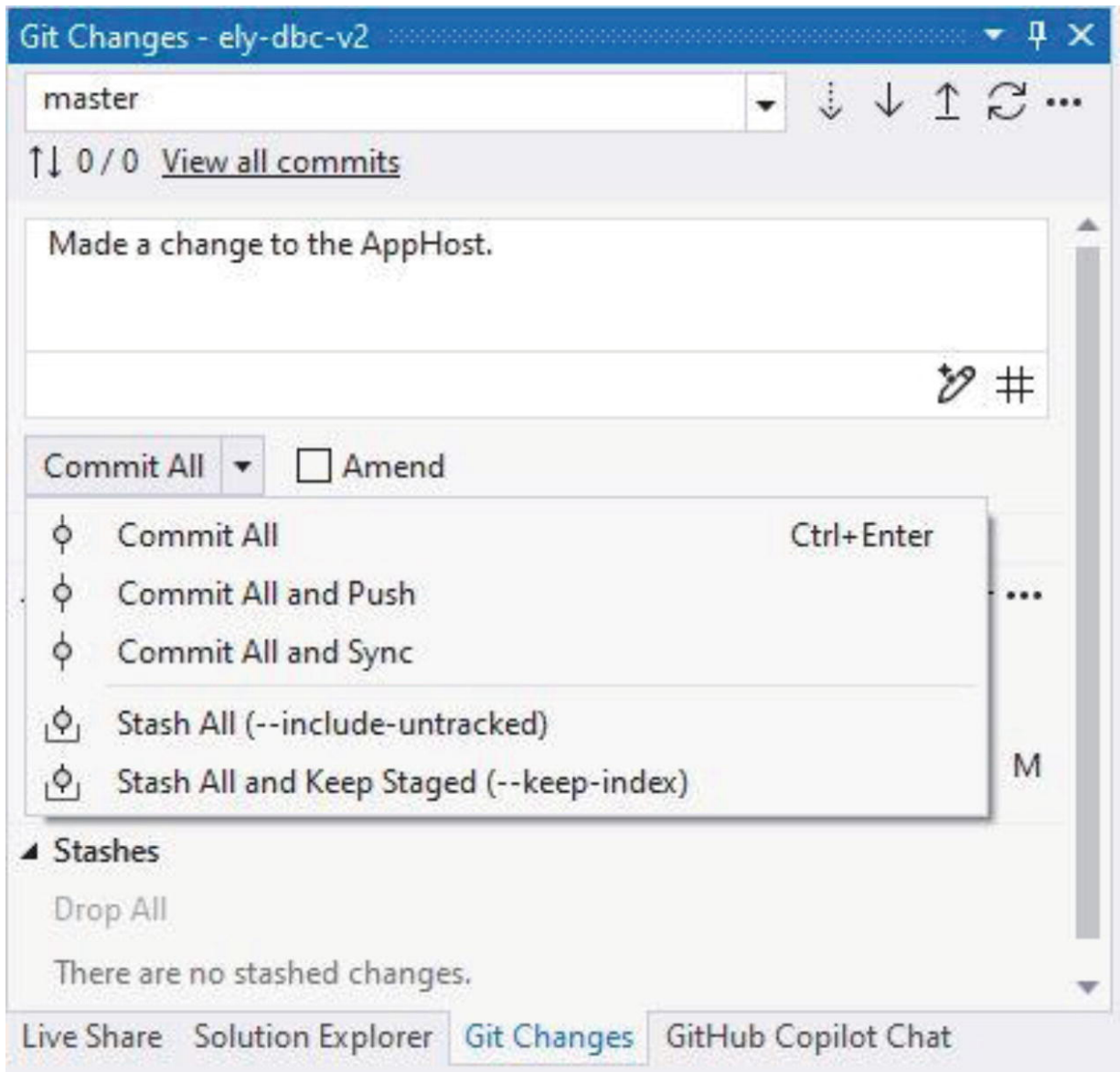
This view helps you keep track of exactly what is being altered in your working directory, ensuring that nothing is accidentally committed without your knowledge.



**Figure 5-9** Git Changes Showing Changes Files

Enter a commit message and then click the drop-down arrow next to the Commit All button.

Here, you can see the options for committing your code as seen in Figure 5-10. You can simply commit all your changes, commit and push to the remote repo, commit to the remote repo and do a sync, or stash your code.

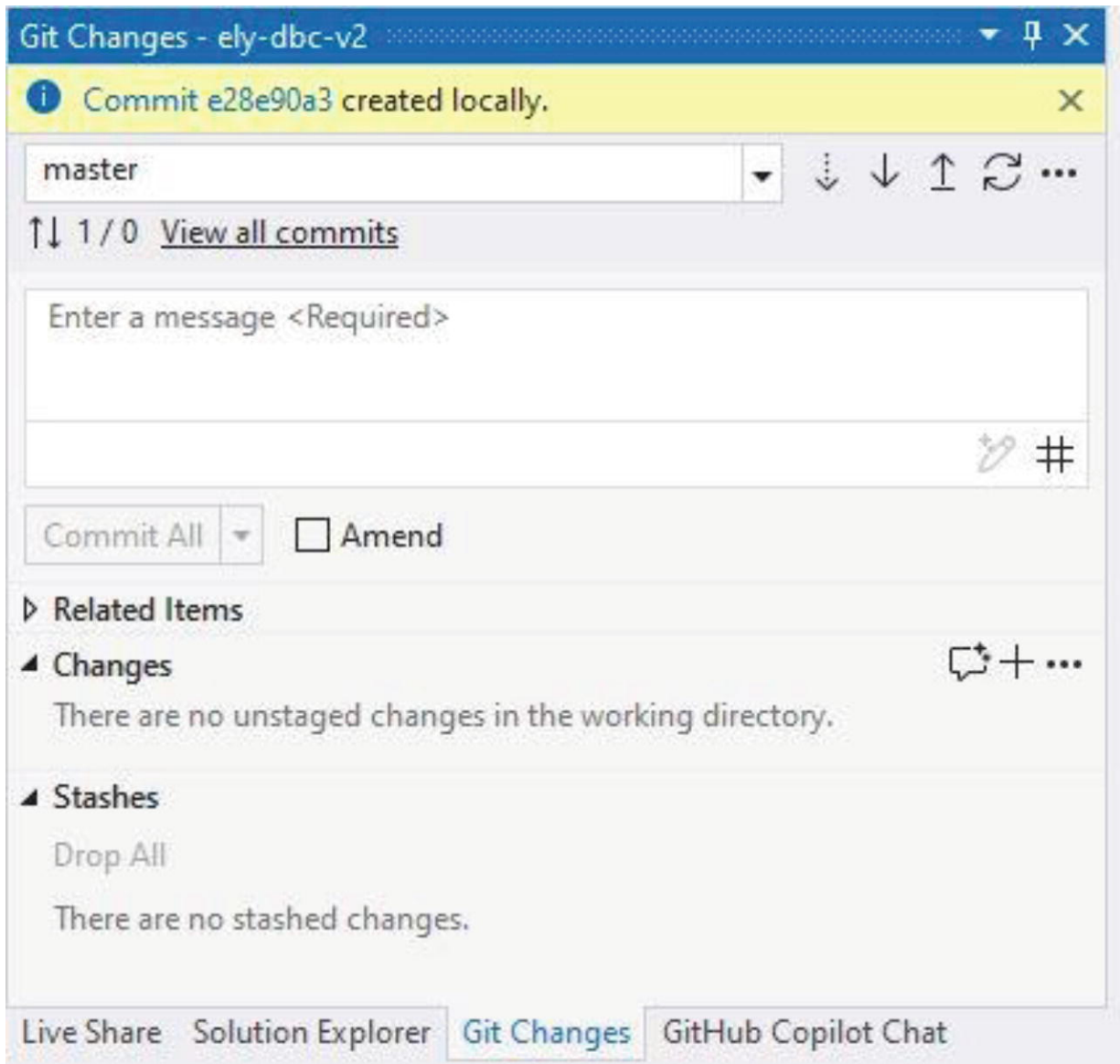


**Figure 5-10** Commit Options

We are just going to commit the changes without pushing to the remote repo.

Remember, the Commit All will create the changes locally. Nothing will be created on the remote GitHub repo yet.

After clicking the Commit All option, the Git Changes tab will tell you that the changes were committed locally and display the commit ID (in this case, e28e90a3) as seen in [Figure 5-11](#).



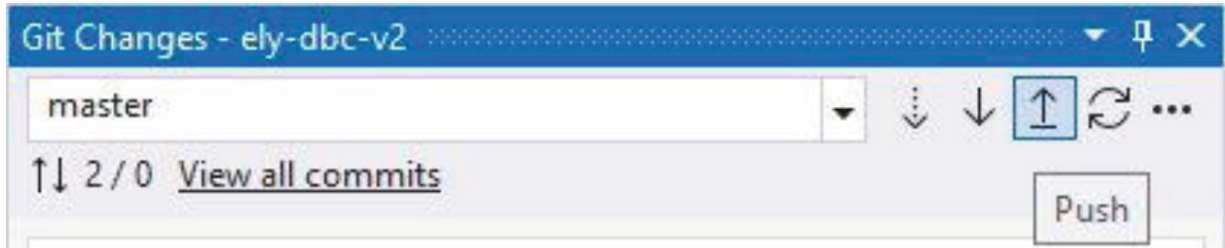
**Figure 5-11** Commit the Changes Locally

You might want to commit locally while working on code, and you are not quite ready to push the changes to the remote. This is a nice workflow to follow. Make some changes, commit locally with a specific commit message, make some more changes, commit again with another commit message, and so on.

**Note** Each commit message and ID will also be visible on GitHub along with the files that have changed.

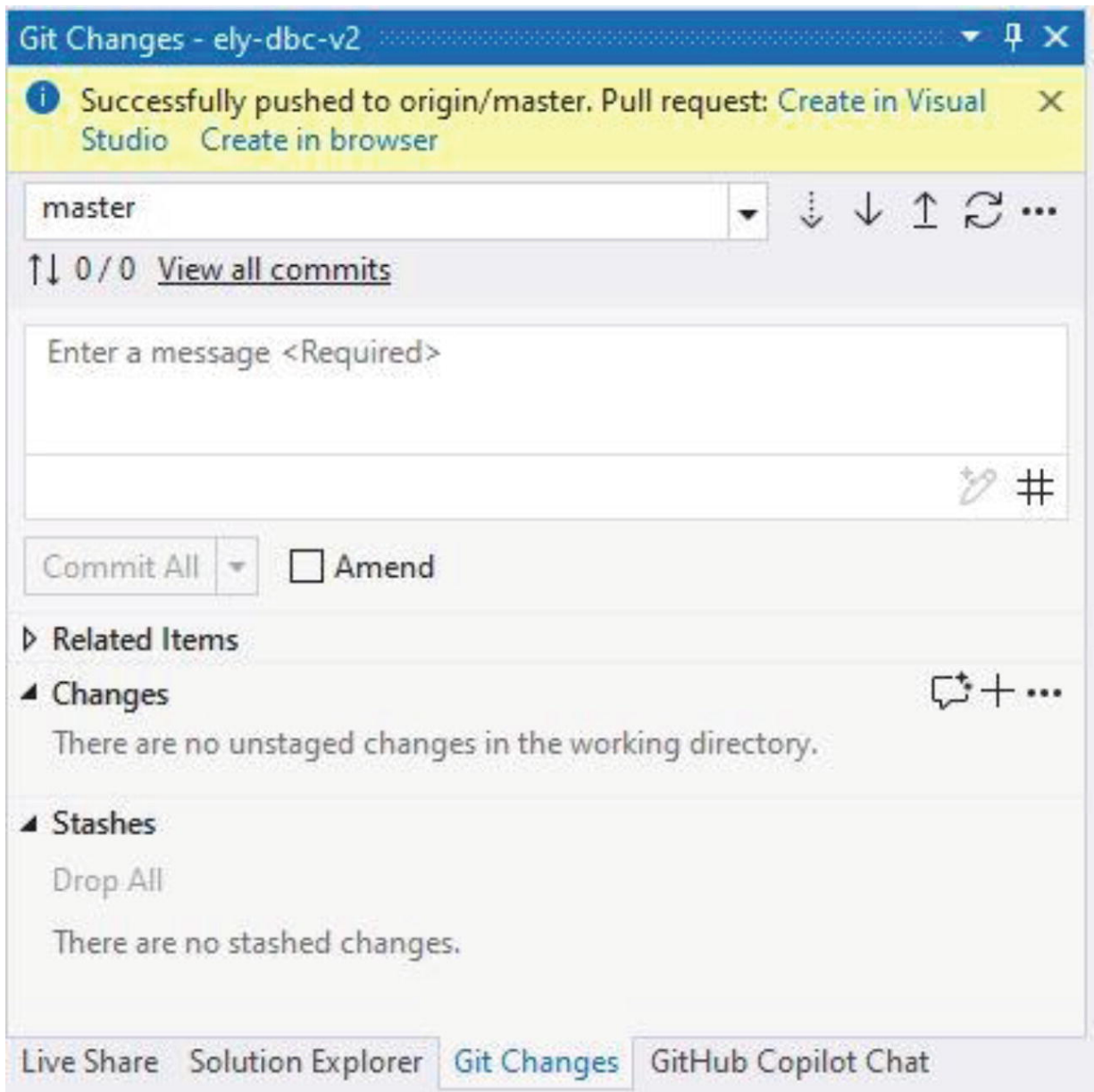


When you have completed the changes and committed everything locally, you can push these changes to the remote repo by clicking the Push button, as seen in Figure [5-12](#).



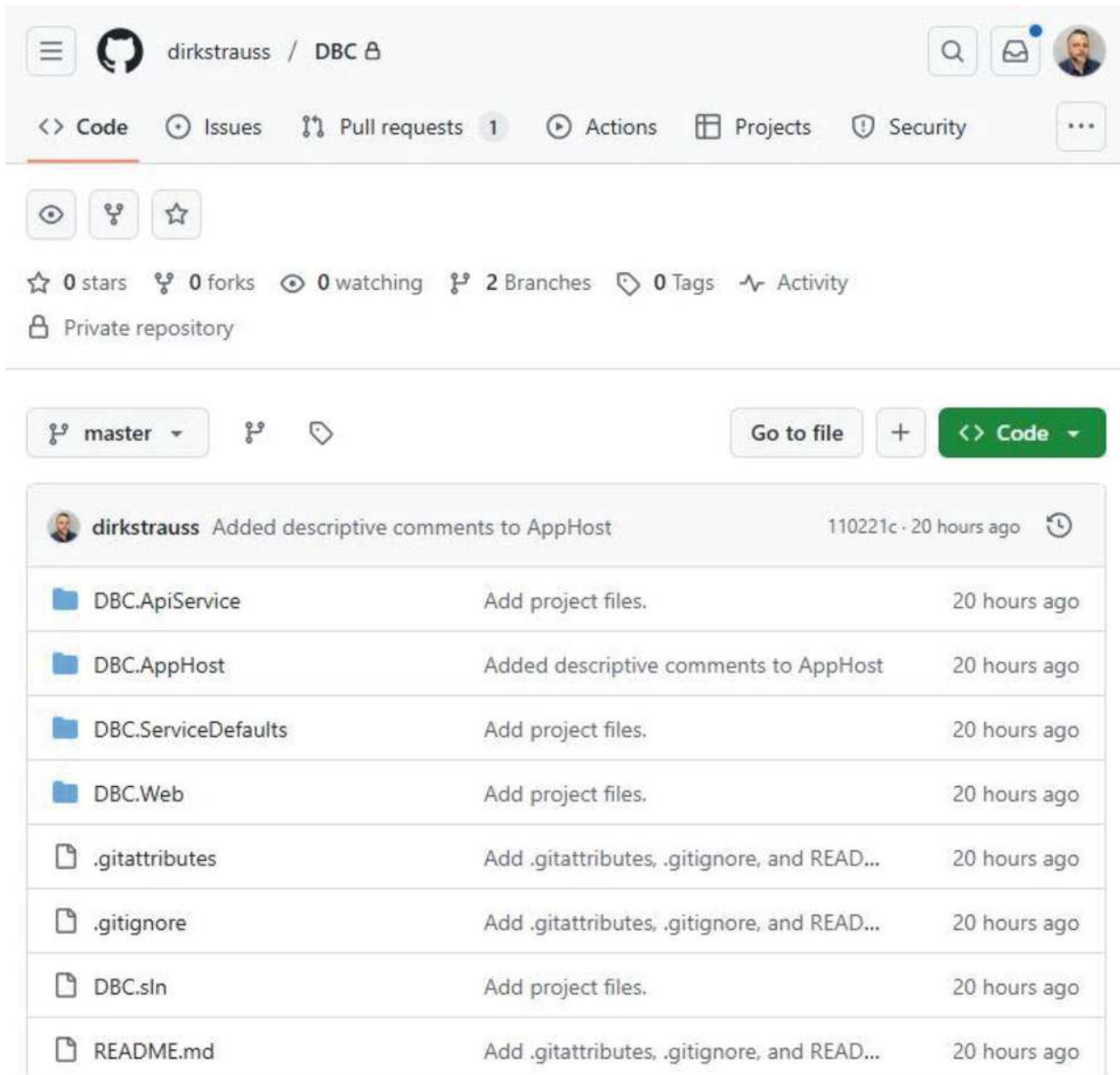
**Figure 5-12** Push Changes to the Remote Repo

Doing this will push the changes to GitHub, and you will be notified in the Git Changes tab that your changes have been pushed to the remote repo on GitHub, as seen in Figure [5-13](#).



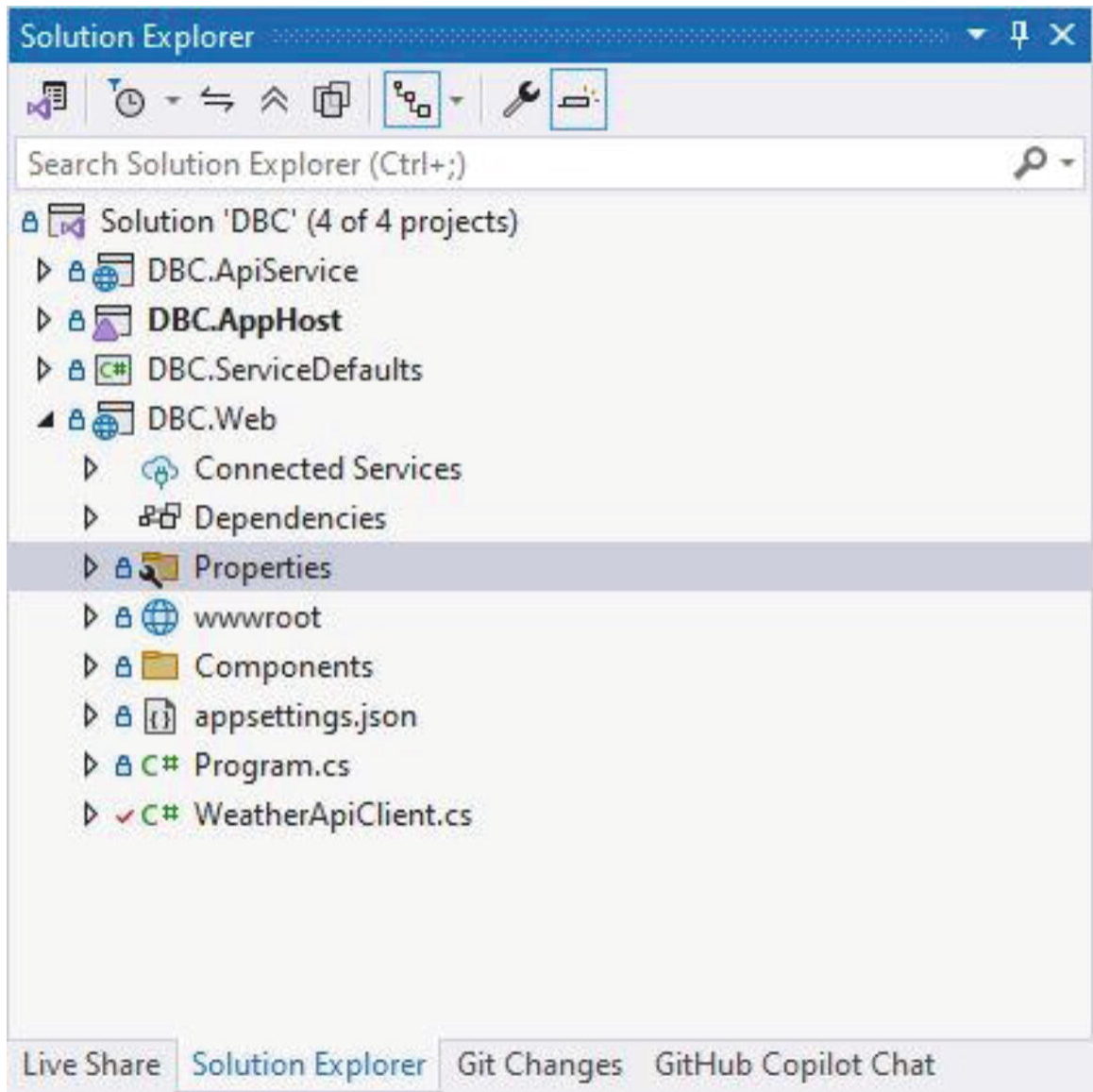
**Figure 5-13** Successful Push to the Remote Repo

If you go to your GitHub repository, you will see that the code you pushed is displayed in your repo (Figure [5-14](#)) along with the commit messages entered.



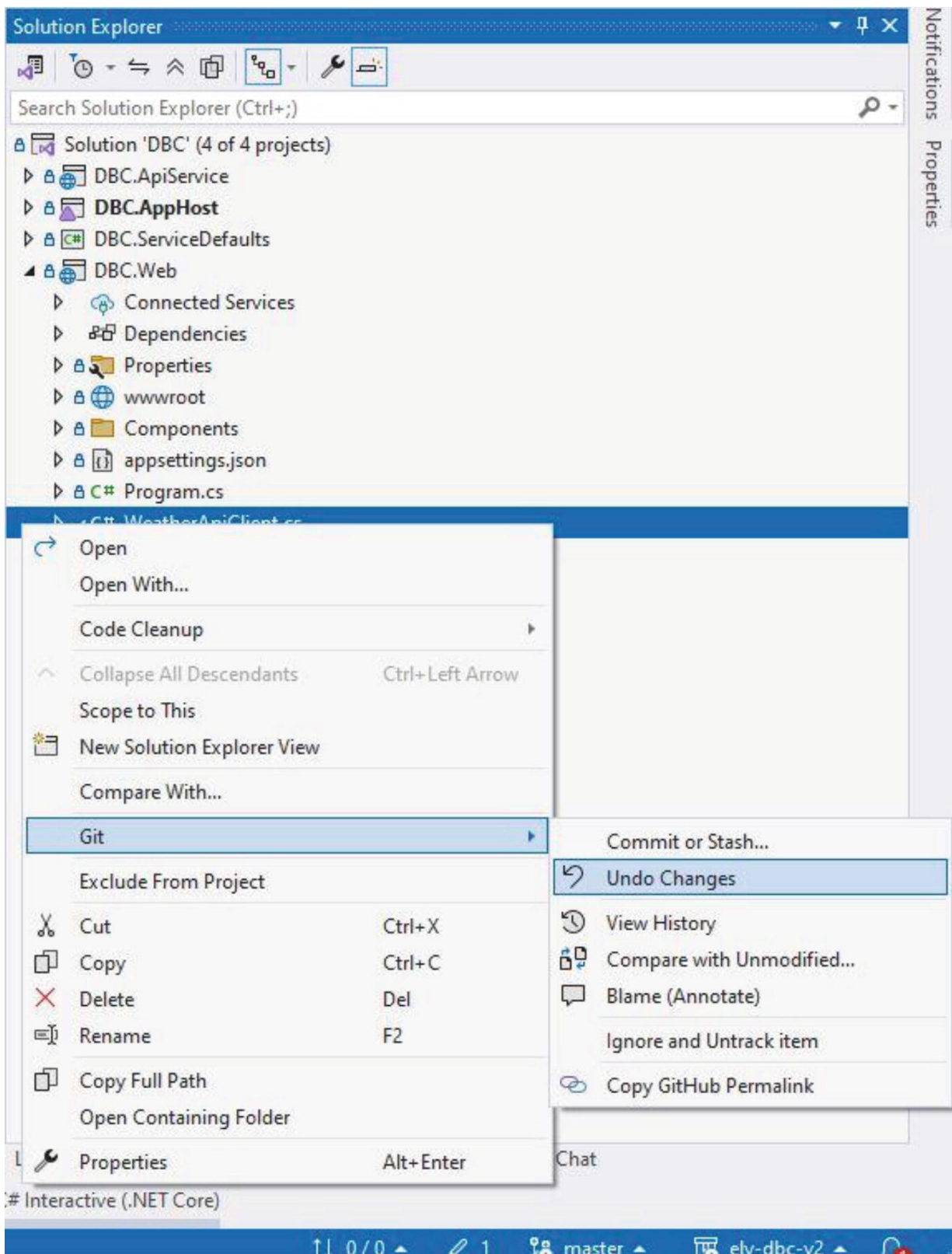
**Figure 5-14** The Pushed Changes in GitHub

It is also nice to remember that if you have made changes to a file that you have not committed yet and want to change the file back to the state it was since the last commit, you can undo these changes easily. As seen in [Figure 5-15](#), I have made some changes to the `WeatherApiClient.cs` file that I no longer want.



**Figure 5-15** Changed Files You Want to Undo

By right-clicking the file as seen in Figure [5-16](#) and selecting Git ➤ Undo Changes from the context menu, you can revert the file back to the state it was at the last commit.



**Figure 5-16** The Undo Changes Option

The ability to quickly revert files becomes especially valuable when working with temporary or experimental changes, for example, debug-specific modifications to config files that you do not intend to keep in the final codebase. Rather than editing the file manually back to its previous state, this feature guarantees an exact restoration to the last committed version.

## **Cloning a Repository**

If you use multiple computers, you can easily clone repositories to each, allowing you to work on the same codebase (e.g., at home and at the office). To do this in Visual Studio is simple and quick. We are just going to be cloning the same repository we created in the previous section.

Start Visual Studio, and then click the Clone a repository option under the *Get started* section of the Visual Studio start screen (Figure [5-17](#)).



## Get started



### Clone a repository

Get code from an online repository like GitHub or Azure DevOps



### Open a project or solution

Open a local Visual Studio project or .sln file



### Open a local folder

Navigate and edit code within any folder



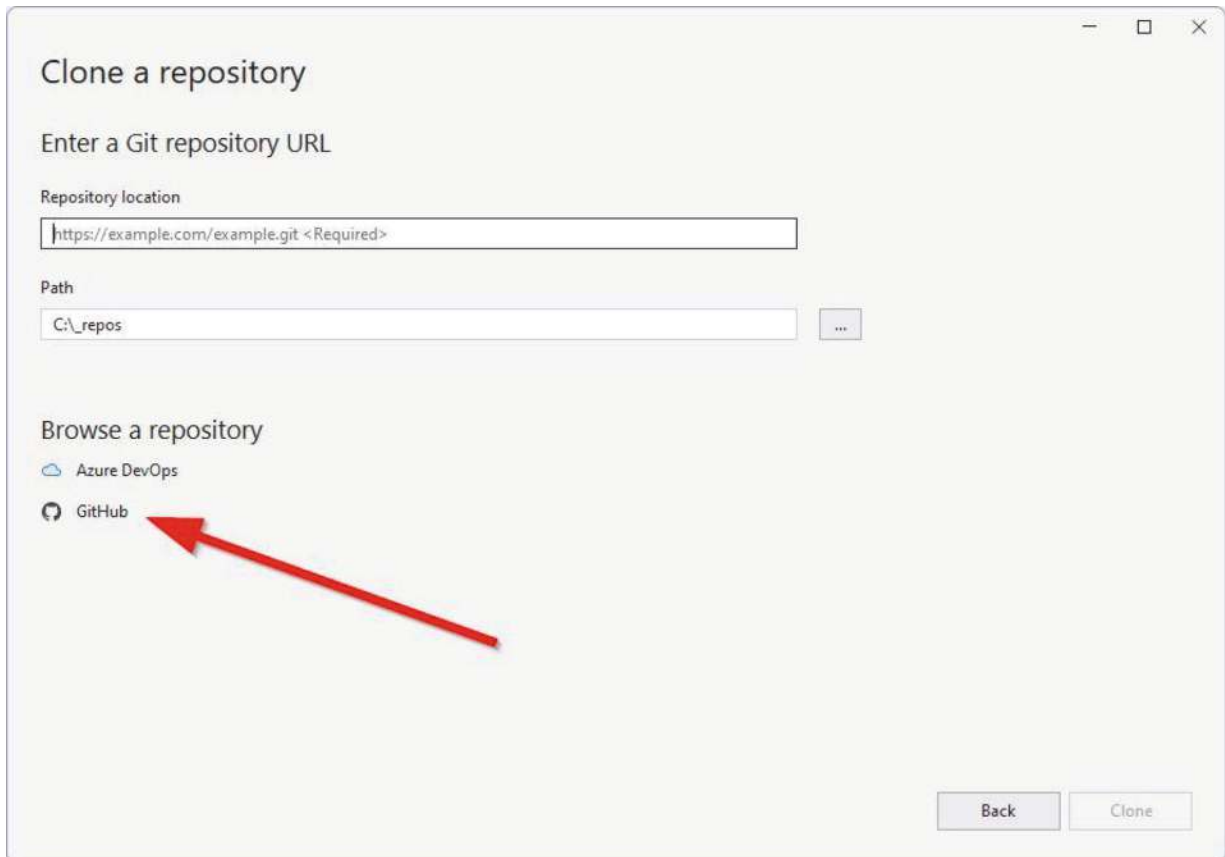
### Create a new project

Choose a project template with code scaffolding to get started

[Continue without code](#) →

**Figure 5-17** Clone or Check Out Code

This will take you to the Clone a repository screen as seen in [Figure 5-18](#).

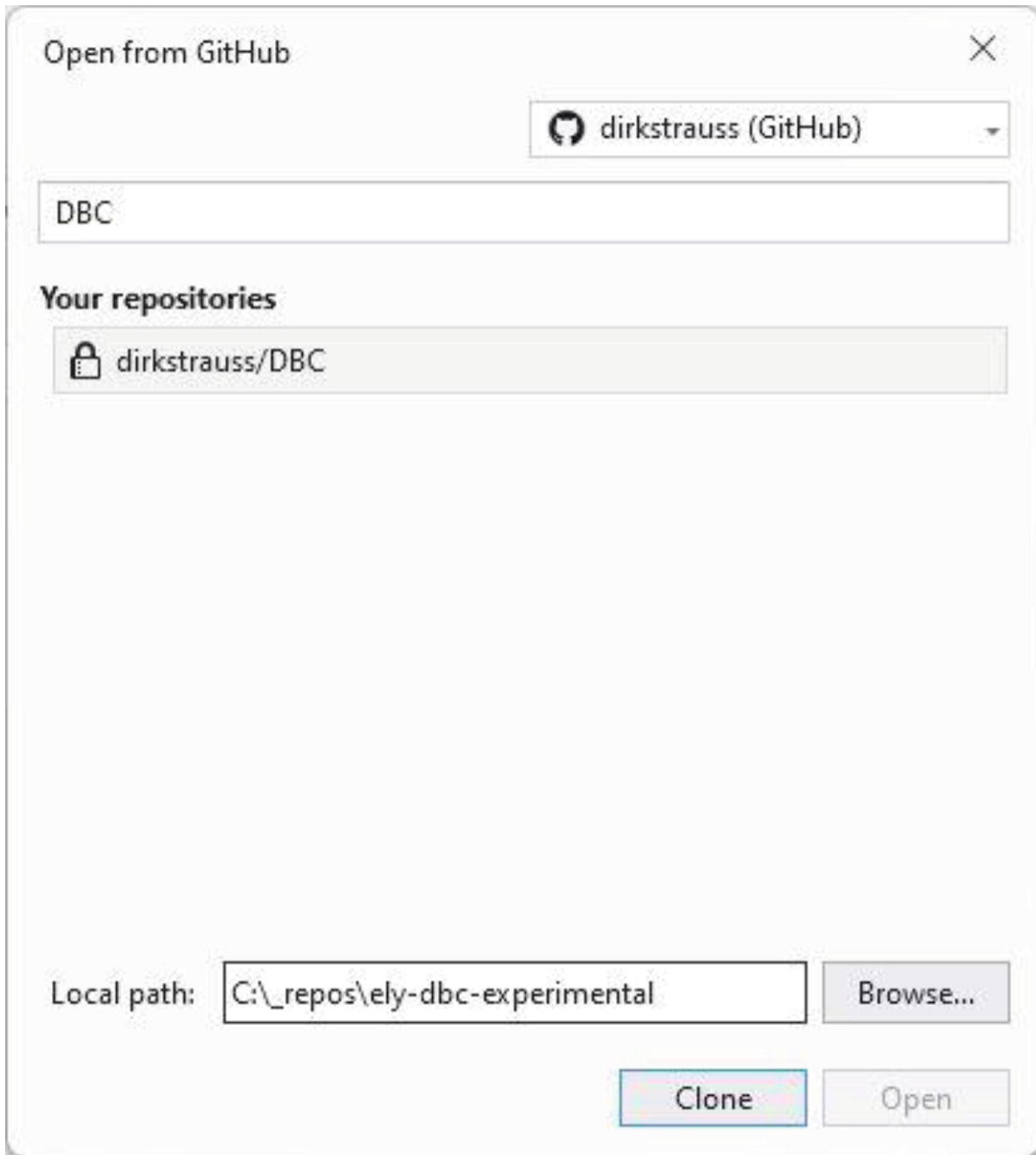


**Figure 5-18** Clone from GitHub

From this screen, you can enter the repository location to get the code from, but because we are using GitHub, we can simply click the GitHub option (Figure [5-18](#)).

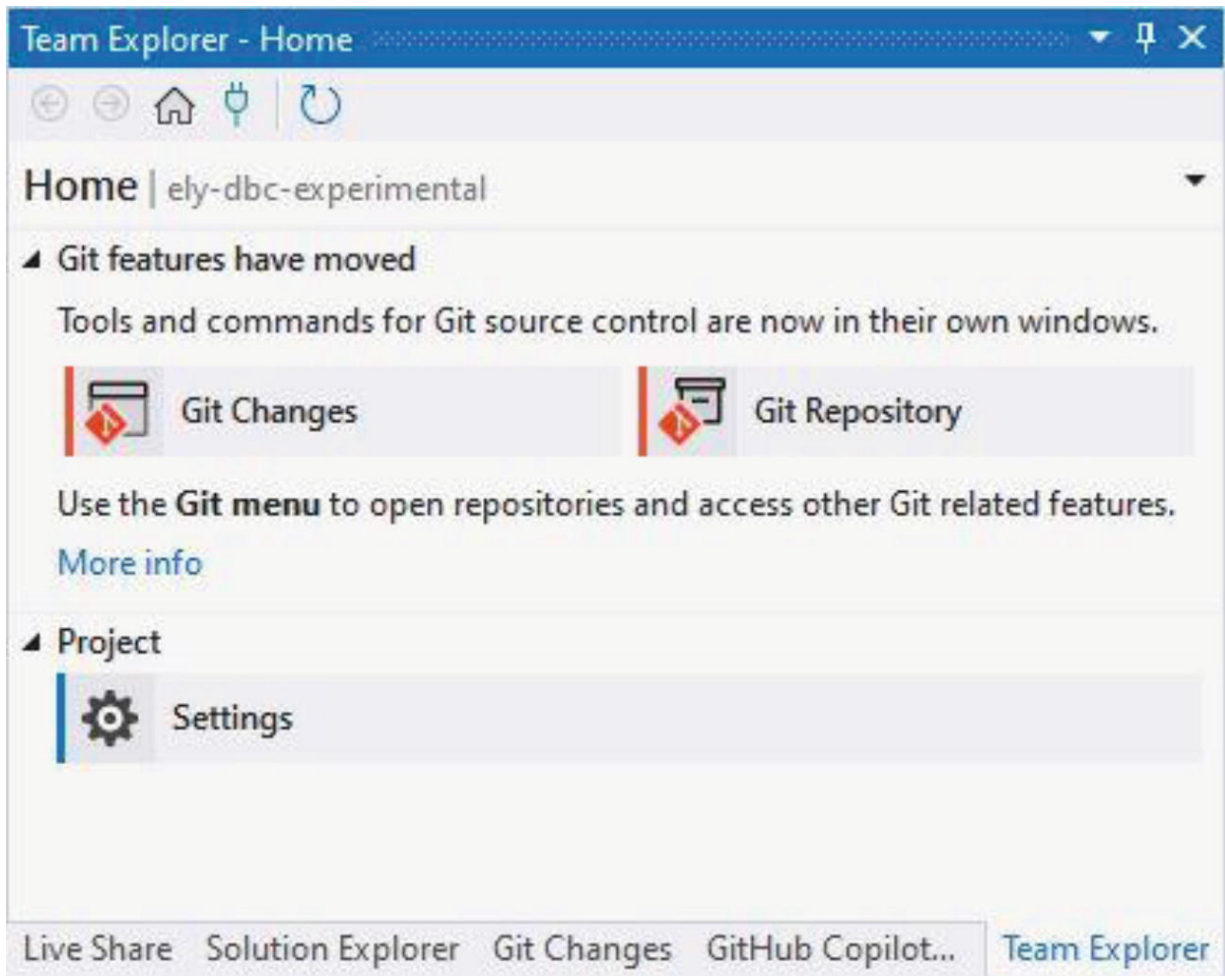
You can also clone repos from Azure DevOps if you are using that as your source control solution.

This will display the Open from GitHub screen (Figure [5-19](#)).



**Figure 5-19** Open from GitHub

Search for and select the project, ensure that the local path is correct, and click the Clone button. The Visual Studio project is then cloned to my machine. In previous versions of Visual Studio, you would see the repo in the Team Explorer window. Opening Team Explorer in Visual Studio 2022, you will see a message stating that the Git features have moved to their own window ([Figure 5-20](#)).



**Figure 5-20** The Team Explorer Window

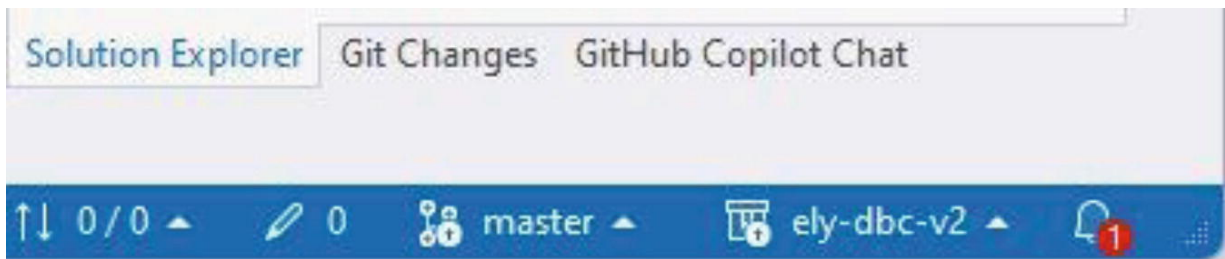
The new Git tool window in Visual Studio 2022 streamlines Git-related actions such as cloning repos, working with stashes, creating and comparing branches, line staging, etc. We will have a look at these Git-related actions in the next sections.

---

## Create a Branch from Your Code

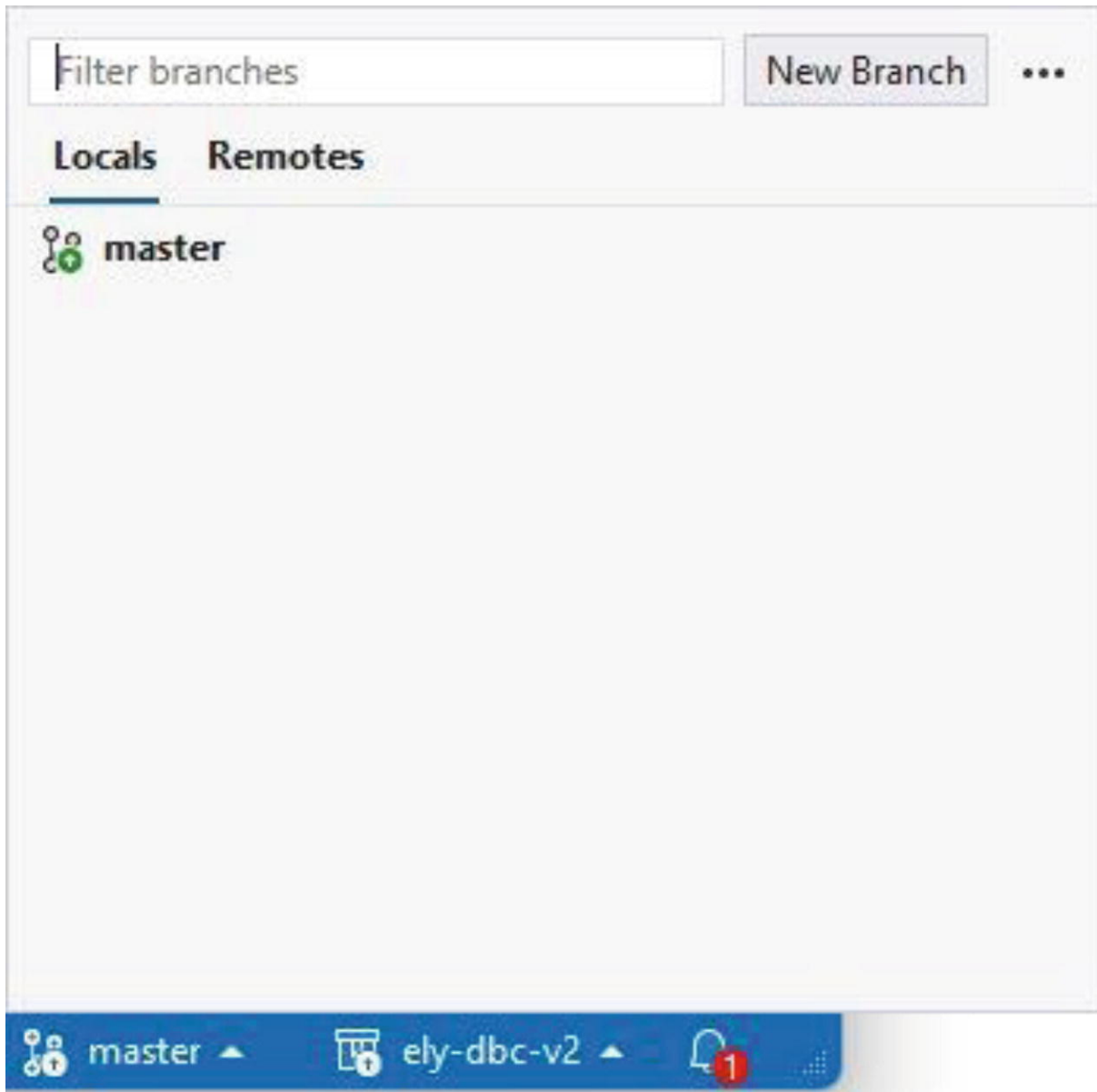
When implementing a new feature or making significant changes to an existing codebase, best practice dictates that it's best to work in isolation to avoid disrupting the stability of the main branch. This is often referred to as master or main. Creating a dedicated branch in Git provides this isolation, allowing you to develop, test, and refine your changes without impacting production-ready code. In Visual Studio, your current branch is displayed in the status bar at the bottom right of

the IDE, making it easy to confirm your working context before you begin. (Figure [5-21](#)).



**Figure 5-21** Working in the Master Branch

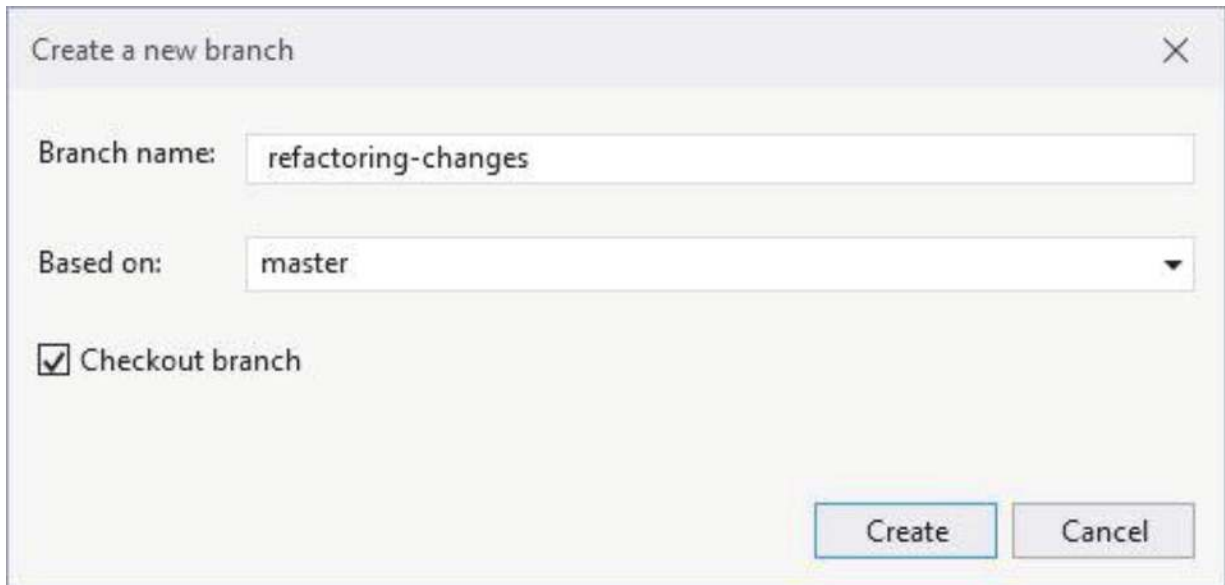
To create a new branch, click the current branch to open the Branches view (Figure [5-22](#)).



**Figure 5-22** Branches View

I will now create a local branch in Visual Studio. To do this, click the New Branch button in the Branches view.





Create a new branch

Branch name: refactoring-changes

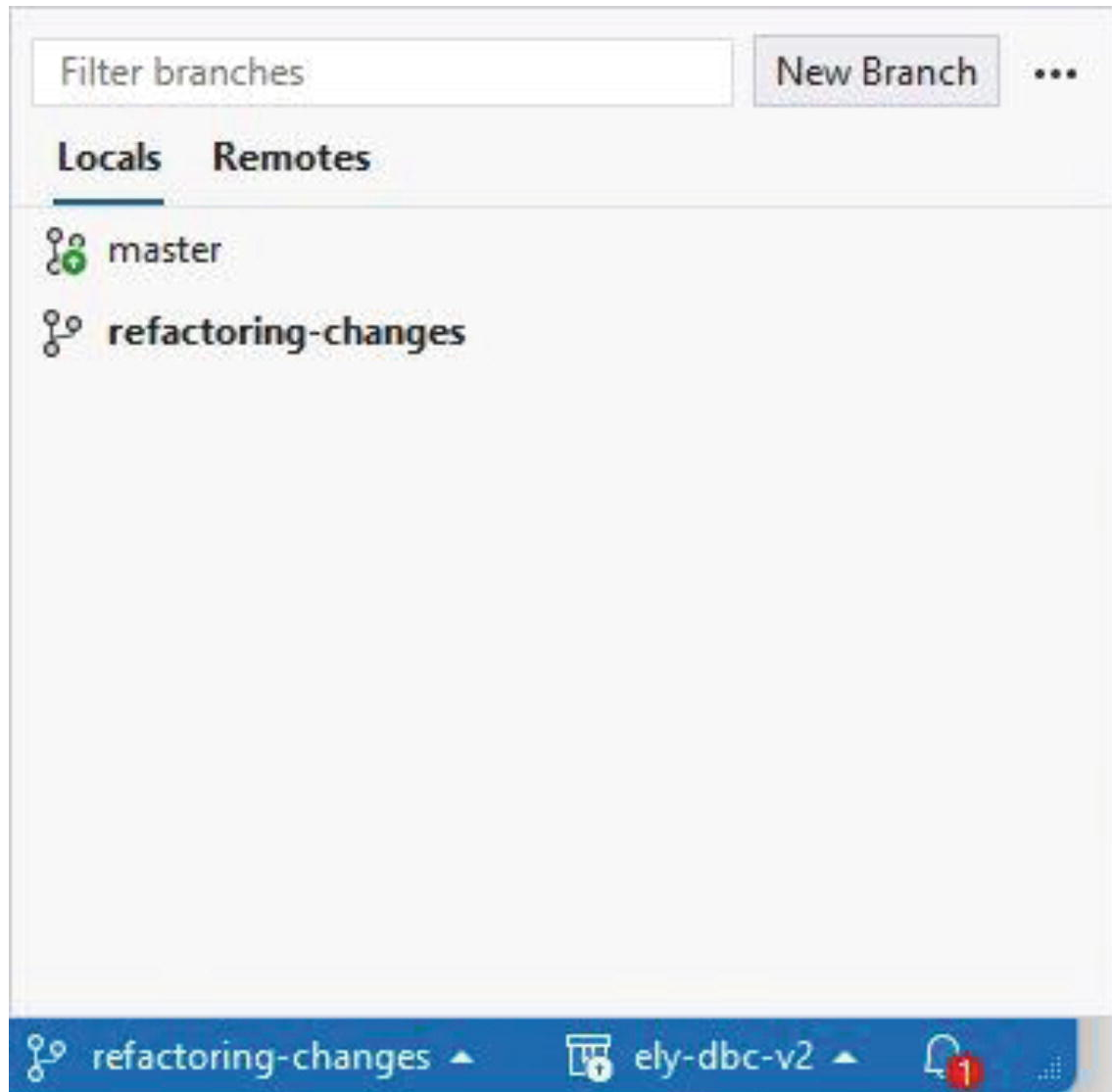
Based on: master

☒ Checkout branch

Create Cancel

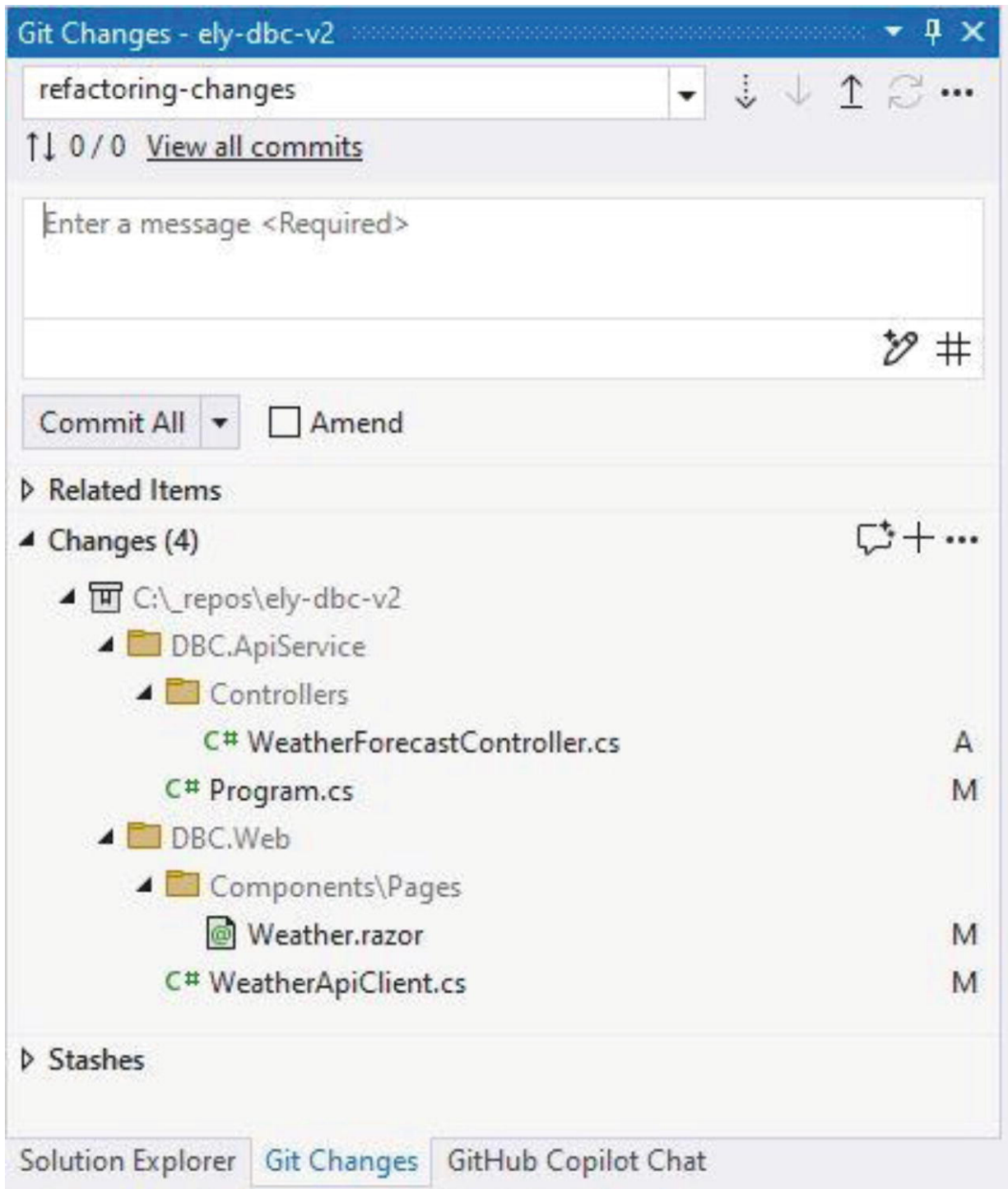
**Figure 5-23** Create a New Branch

I can now give my new branch a suitable name (Figure [5-23](#)) and tell it to create the branch from the master branch. I keep the Checkout branch selected to check out my new branch and click the Create button.



**Figure 5-24** Feature Branch Created

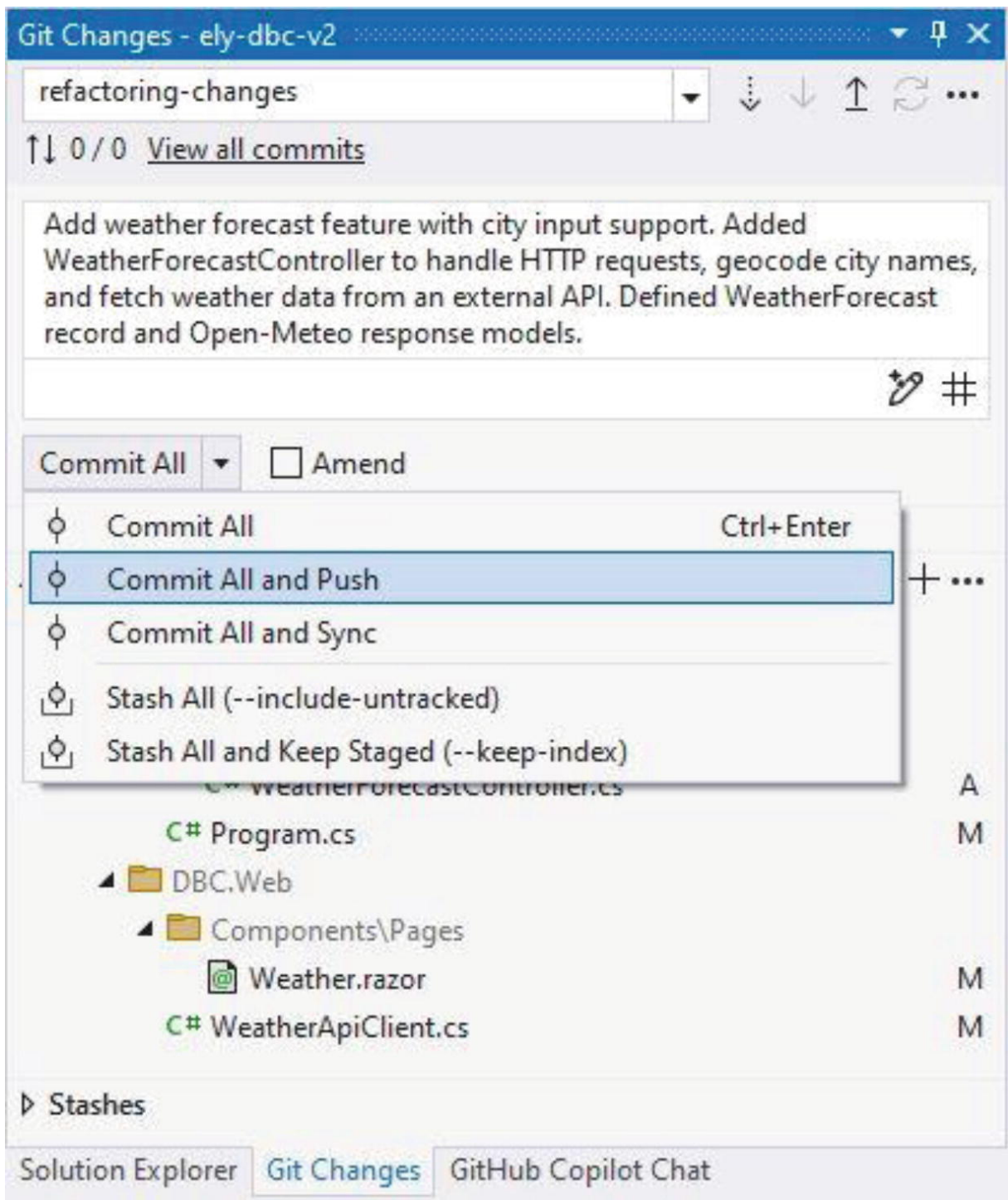
As seen in Figure [5-24](#), my new local branch is created and checked out. This means that from now on, all changes made to the code will stay in this branch. Let's add some new code to the project.



**Figure 5-25** New Feature Code Added

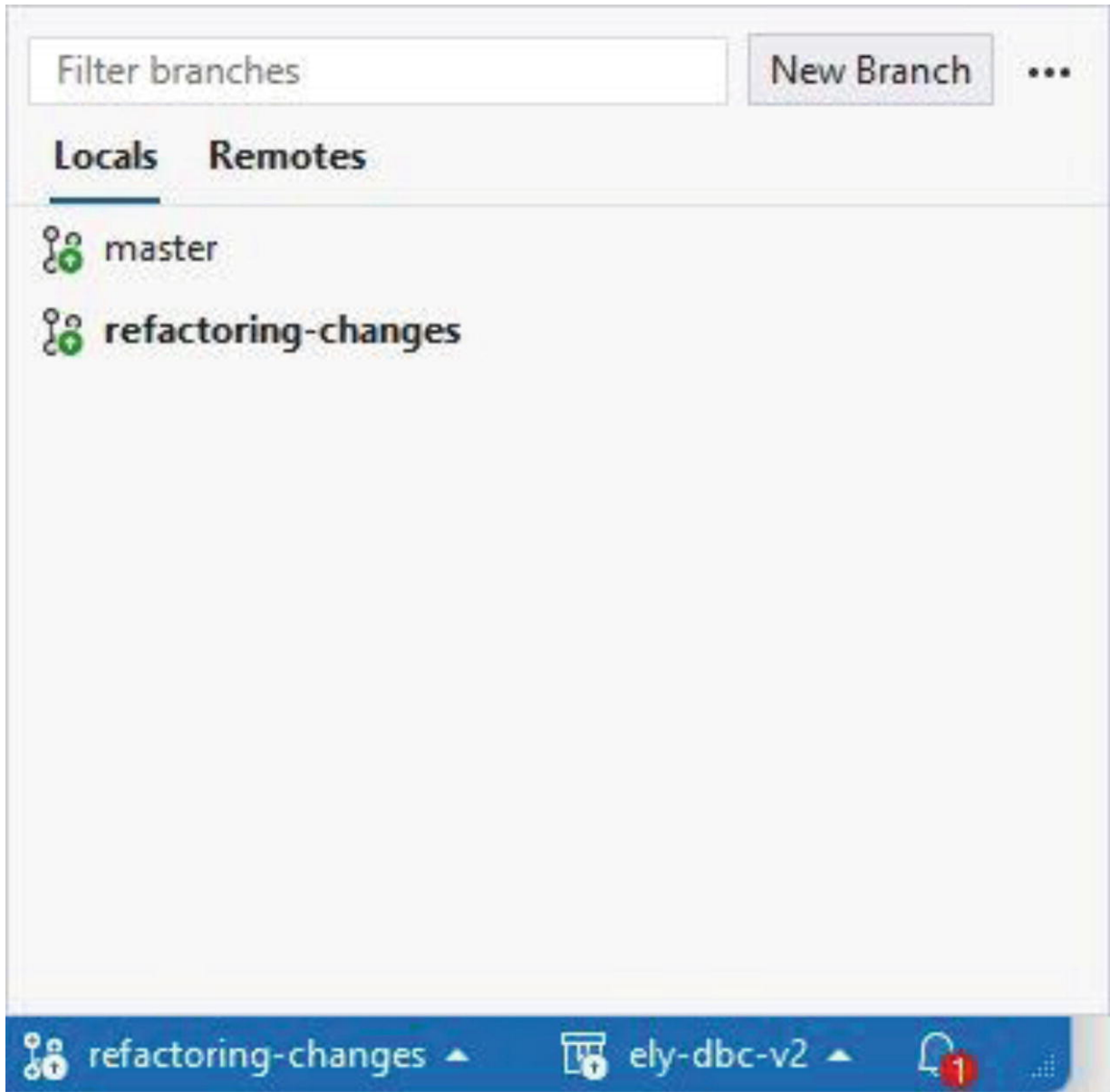
As seen in Figure [5-25](#), I have made several changes to my code. I now need to commit the changes to my branch. These changes include both new and modified files. Before committing code, I need to add a

suitable commit message. Then I can click the drop-down next to the Commit All button and select Commit All and Push (Figure 5-26).



**Figure 5-26** Commit All and Push

This action commits the changes to the local repository and immediately pushes them to the remote repository. By selecting the branch indicator and switching to the Remotes tab, you can confirm that your feature branch has been pushed to GitHub (Figure 5-27).



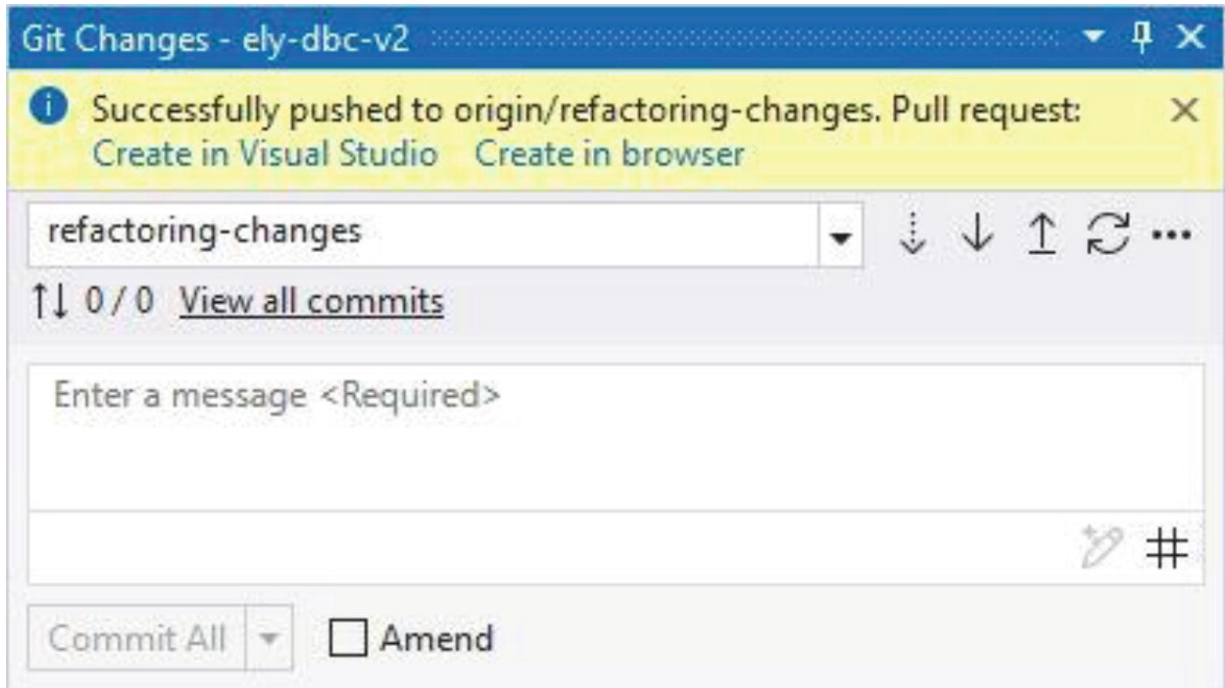
**Figure 5-27** Branch Pushed to the Server

With the branch now available in the remote repository, the next step is to merge these changes into the master branch. To achieve this, a pull request will be created.

---

## Creating Pull Requests

With our changes committed and pushed to the feature branch, we can easily create a pull request without leaving Visual Studio. After pushing your branch, Visual Studio will display a notification banner in the Git Changes window shown in Figure 5-28. Here you have the option to create a Pull Request in Visual Studio or in the Browser.



**Figure 5-28** Notification to Create a Pull Request

If you do not see this banner, you can manually open this dialog by using the menu. Go to Git ► GitHub ► New Pull Request. If you're using Azure DevOps, you will see the menu Git ► Azure DevOps ► New Pull Request.

We will be opting to do this inside Visual Studio, and clicking the *Create in Visual Studio* link, opens the New Pull Request window.

Clicking on the Link or using the GitHub or Azure DevOps menu will all lead to the same Pull Request creation interface within Visual Studio.



From the New Pull Request screen shown in Figure 5-29, you can select the branches you want to create the PR for and enter the pull request details. Generally, you would verify the source branch (Visual Studio pre-selects this based on the branch that contains your most recent commits) and the branch that you want to merge into. In our example, we want to merge our changes into the `master` branch from our `refactoring-changes` branch.

You would then enter your Pull Request title and provide the details of the changes you are creating a pull request for in the Details section. The description you enter supports Markdown, allowing you to include rich formatting such as bullet lists and code blocks. You can also click on the pencil icon to have AI generate your Pull Request details for you.

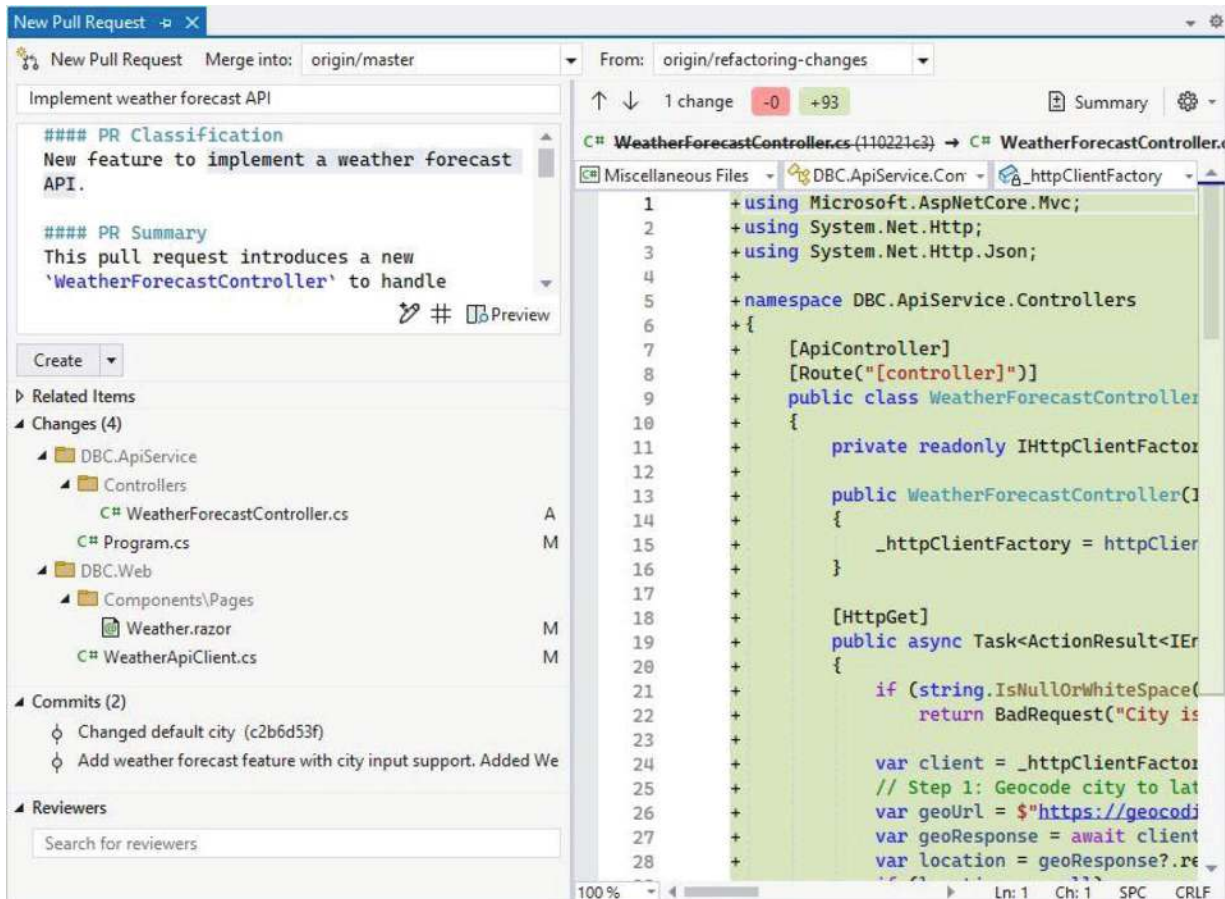


Figure 5-29 New Pull Request Window

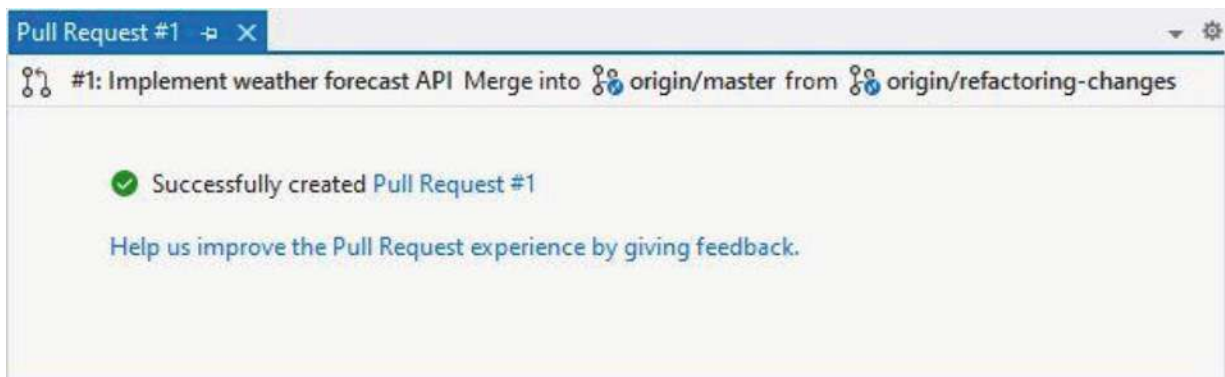
You can preview the entered details by clicking on the Preview button. This will open a Markdown viewer above your diff window. You

can also link work items or issues by clicking on the # next to the Preview button. This will trigger an issue search, allowing you to easily reference the relevant issue number or Azure work item.

A diff window will be displayed showing you exactly what you will be merging. This is nice as it gives you a final look at exactly what is being included in this pull request.

The New Pull Request window in Visual Studio also allows you to add reviewers. This ensures that the correct people are notified to review the code that you want to merge. Once you are happy with the details you entered, click the Create button. This will submit the PR directly from Visual Studio to the server and notify any reviewers you selected that a PR is assigned to them. The Create button also allows you to create a draft by clicking on the dropdown next to the button and clicking on *Create as Draft*. Doing this will create your PR on the server in a draft state.

This is useful for sharing changes for early feedback without marking it as ready to merge into the main or master branch. Whichever option you choose, after creating the PR, Visual Studio will provide feedback that the Pull Request creation was successful, as shown in Figure 5-30. Here it shows that *Pull Request #1* was created.



**Figure 5-30** Pull Request Successful

You now have an active pull request against your branch and your team can begin reviewing the changes and start providing feedback to you.

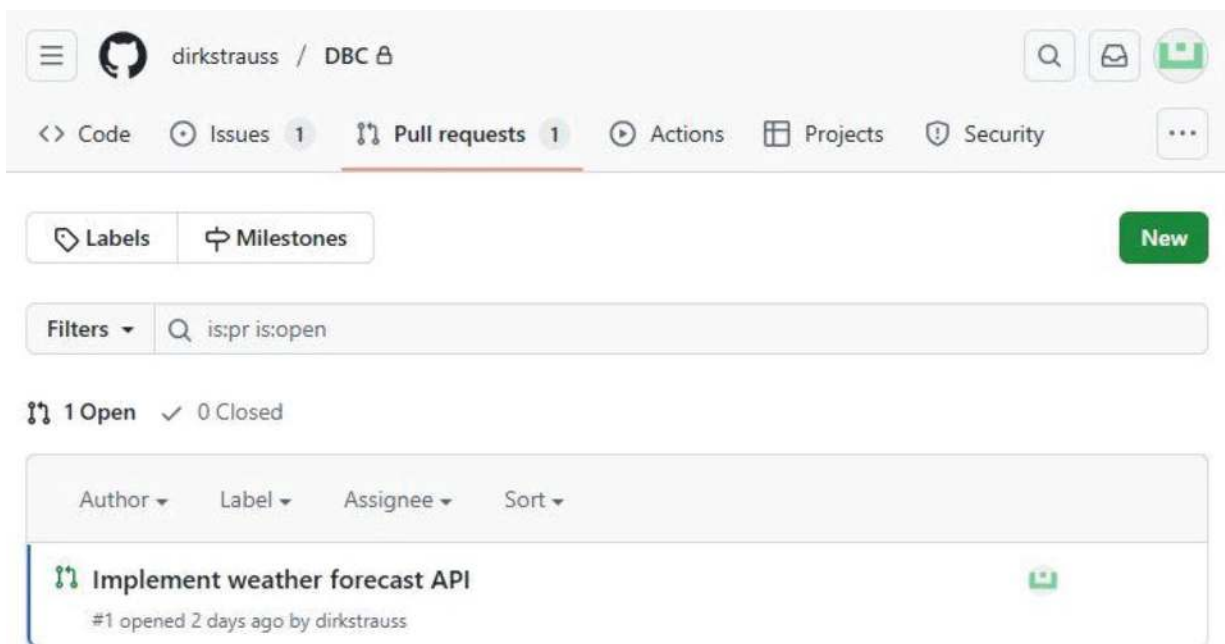
You can browse to your GitHub repo, as shown in Figure 5-31, to view your active pull request.

---

# Handling Pull Requests

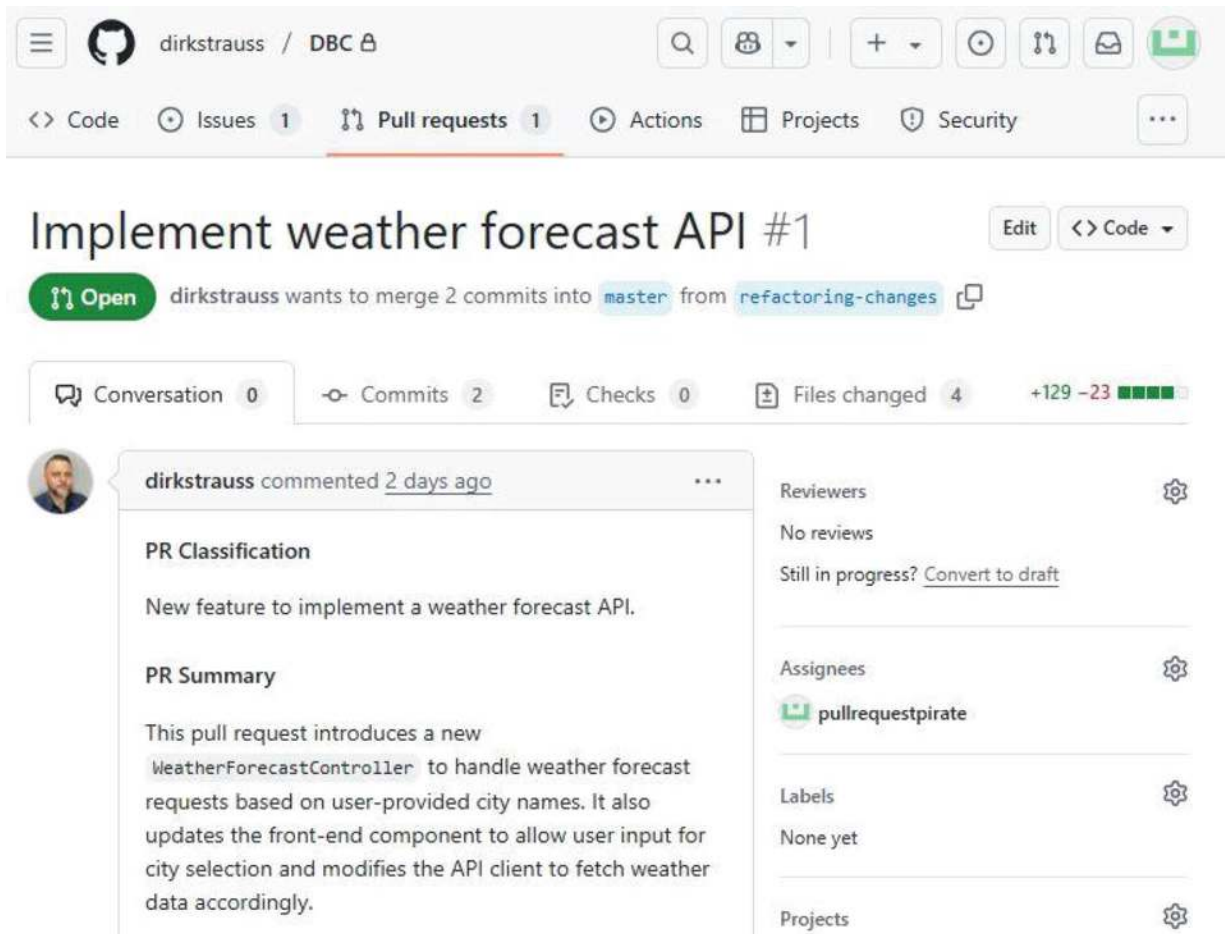
Pull requests (PRs) are the backbone of modern collaborative development. Teams can review, discuss, and validate any changes before they are merged into the main branch. The integration Visual Studio provides with GitHub (or Azure DevOps) streamlines this workflow, allowing you to manage PRs without leaving the IDE.

Viewing your open pull requests can be done by navigating to the *Git* menu and selecting *GitHub* (or *Azure DevOps* if you are working in that environment), and choosing *View Pull Requests*.



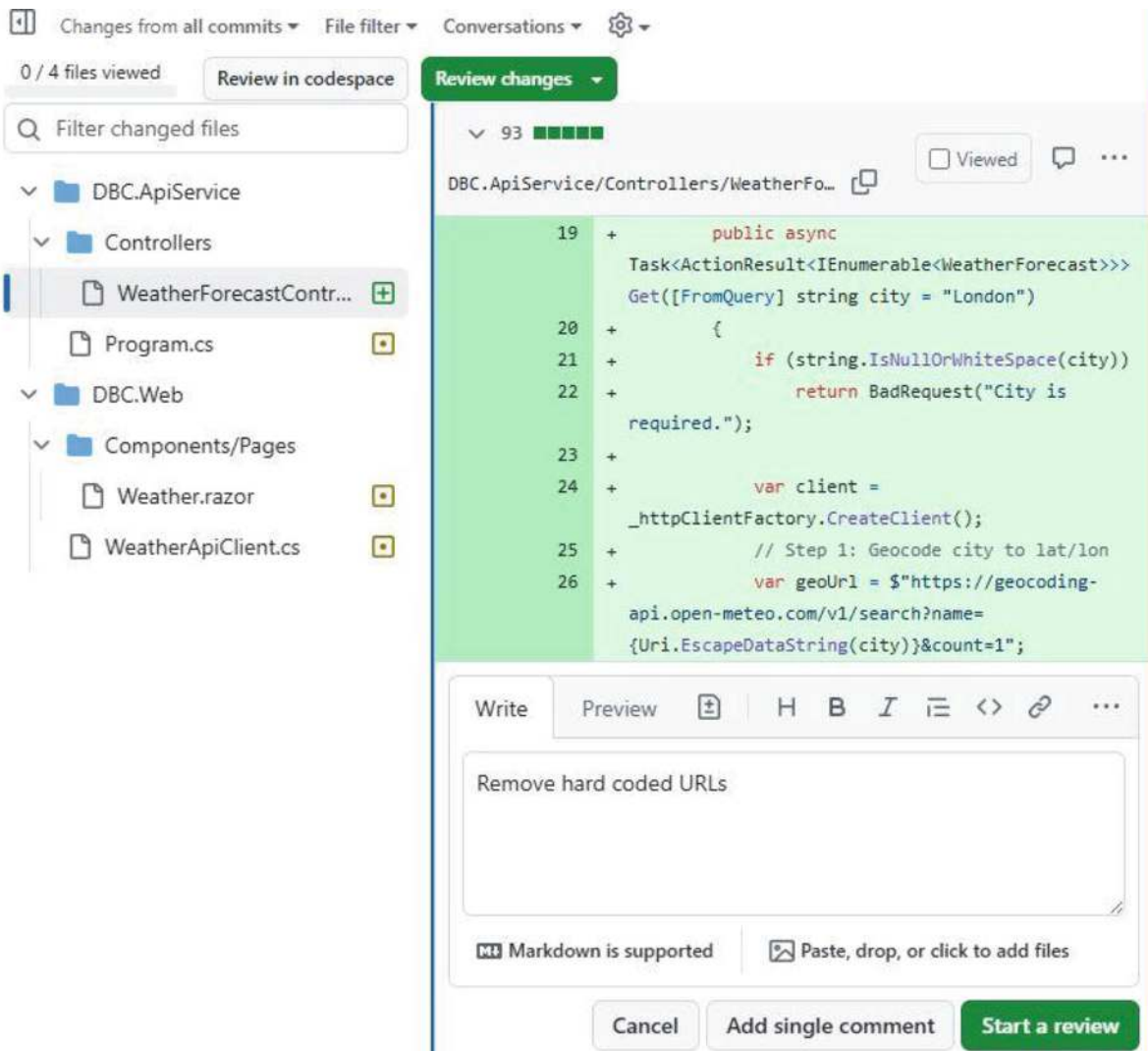
**Figure 5-31** The Active PR on GitHub

This will take you to all the active pull requests on GitHub for the repository you're connected to (Figure [5-31](#)). From this screen on GitHub, you can quickly see which PRs have been assigned to you, the current status of those PRs, and any PRs that require your attention.



**Figure 5-32** Viewing the Pull Request

Acting as the reviewer, you can open any PR to see the description, related discussions, and the files that are changed. The *Files changed* tab shown in Figure 5-32 will display a detailed breakdown of all the modifications to a particular file, as well as any added or removed files.

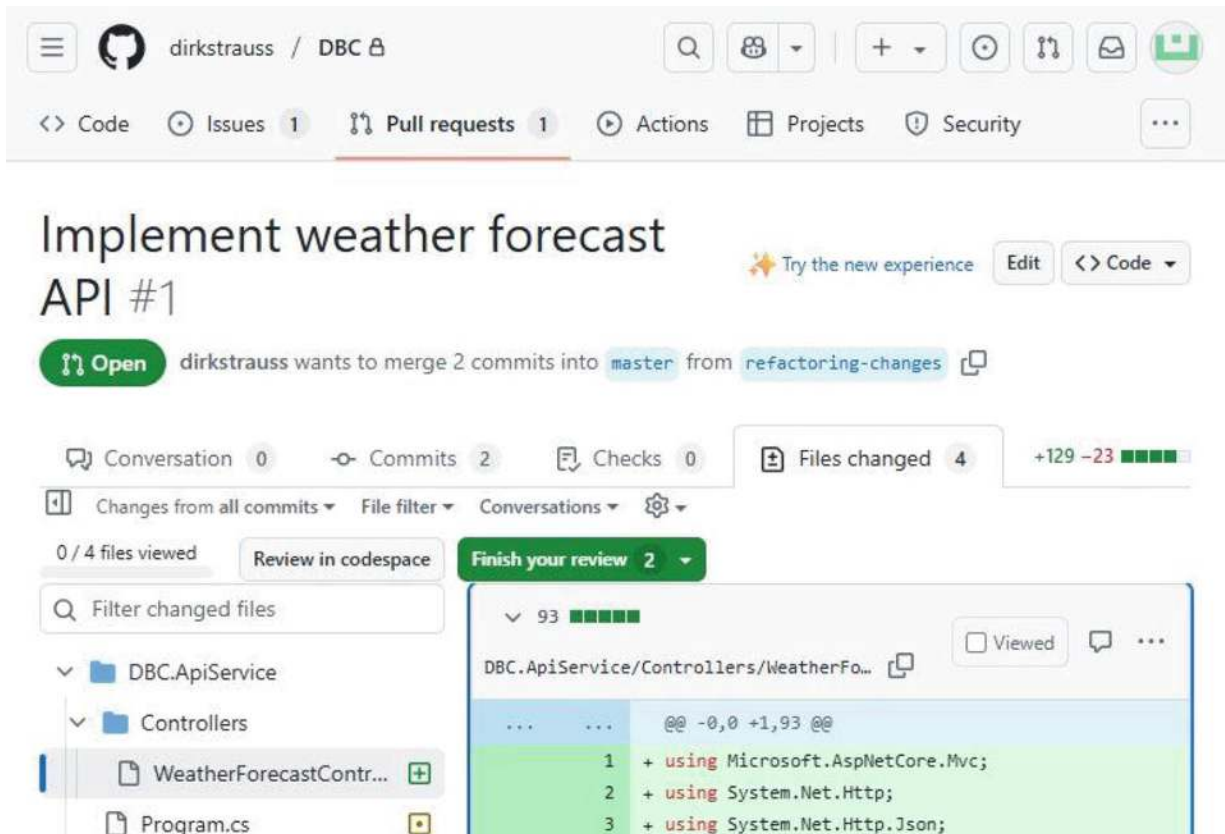


**Figure 5-33** Start a Review

As shown in Figure [5-33](#), the reviewer is reviewing a new file, so no diff is available. In the left tree view, files marked with a green square with a plus icon are new files, while a red square with a dot indicates a modified file.

Clicking any file in the tree view opens its contents for inspection. If you spot an issue, hover over the relevant line number and click the plus sign to add a comment. This displays an editor from where you can either leave a single, isolated comment or start a full review. Reviews are useful when you have multiple feedback items and want to submit them all at once for consideration.





**Figure 5-34** Finish the Code Review

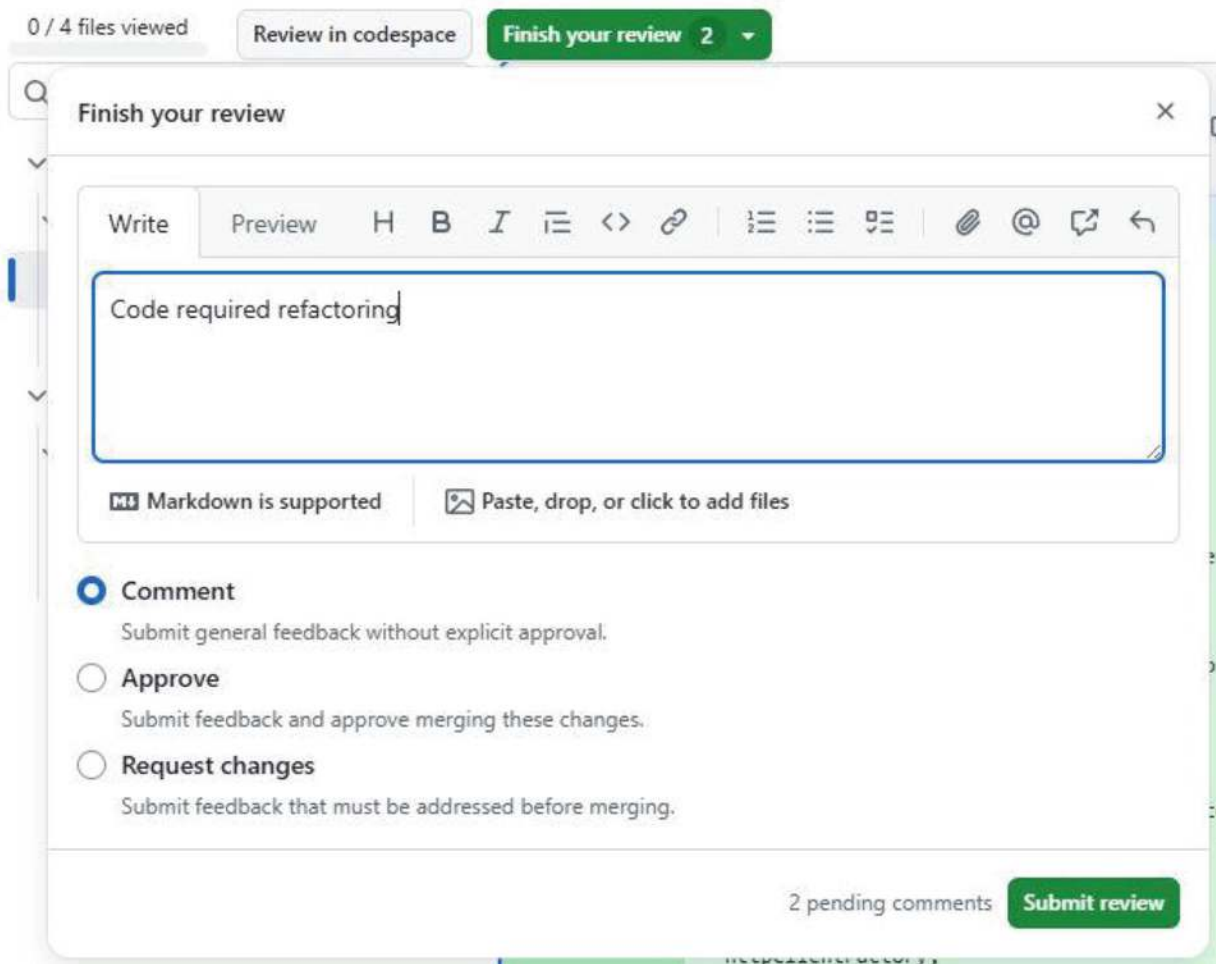
Once all the feedback on the code changes has been added, you can complete that code review by clicking the *Finish your review* button as shown in Figure 5-34.

This will prompt you to choose an action as shown in Figure 5-35:

- Comment
- Approve
- Request changes

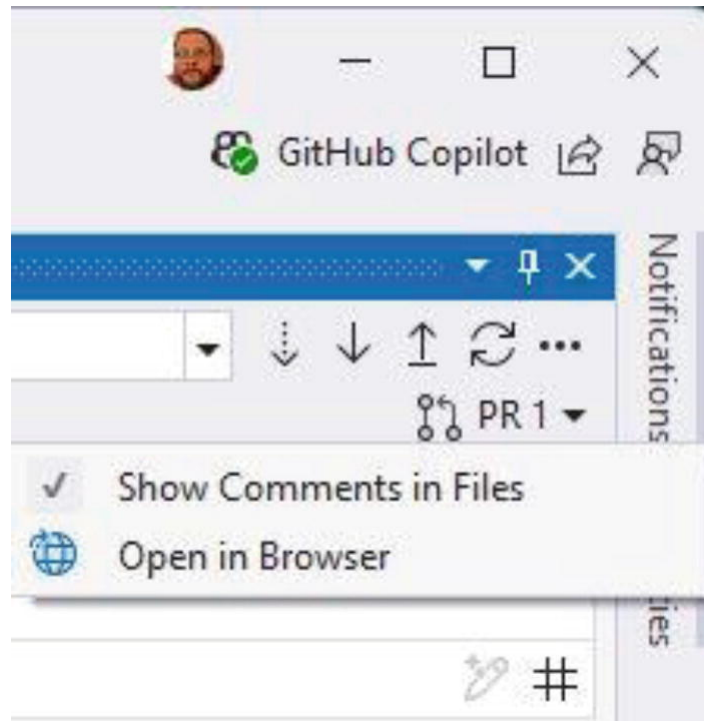
On submission, GitHub will notify the PR author by email that a code review has been done and that there are some items that need resolving before the changes will be merged.





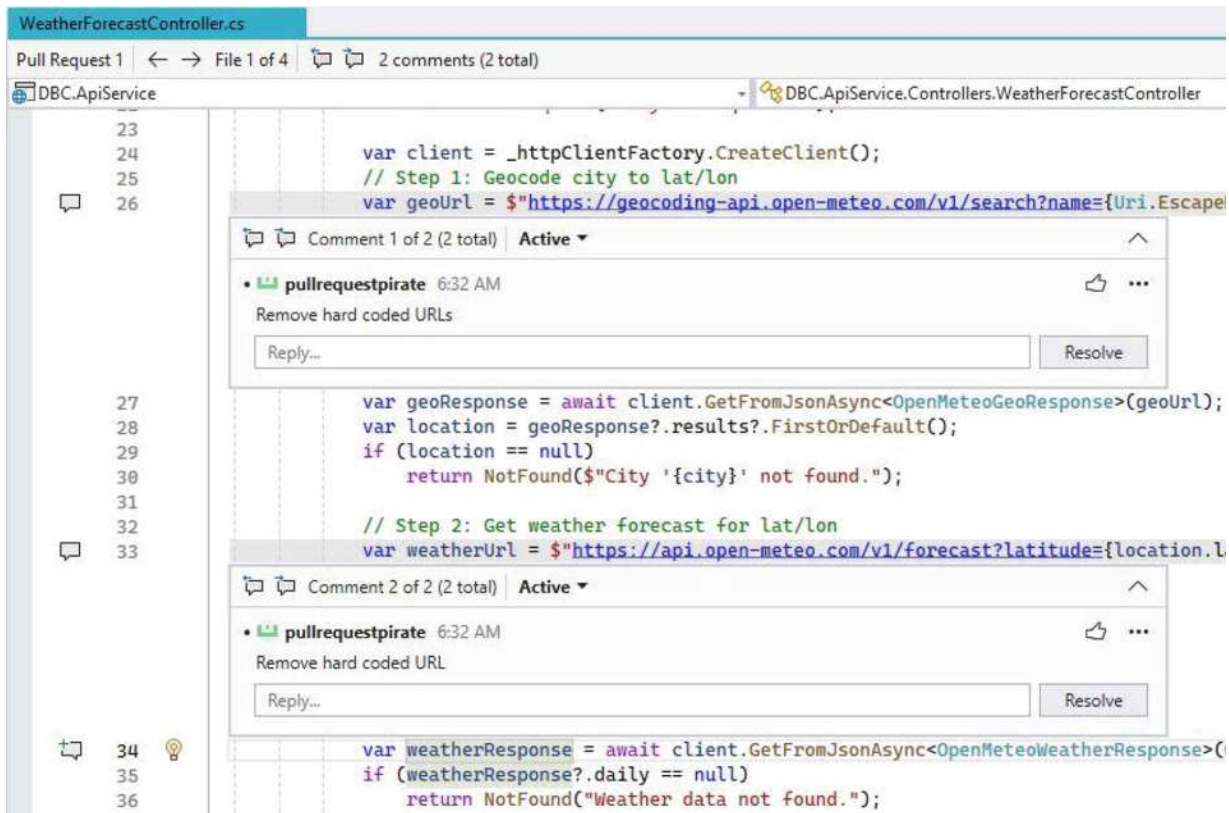
**Figure 5-35** Submit the Code Review

While you can always view review feedback on GitHub, Visual Studio allows you to bring those comments directly into the IDE. To do this, click on the PR dropdown as shown in Figure [5-36](#) and check the *Show Comments in Files* option or go to the menu **Git** ➤ **GitHub** ➤ **Show Comments In Files** to enable the comments to be shown in your code file.



**Figure 5-36** Show Comments in Files

After a short sync, comments will appear in line with your code as shown in Figure [5-37](#).



**Figure 5-37** Showing PR Comments

This is particularly useful when making changes in response to the feedback you receive, as you can see the reviewer's remarks exactly where they apply.

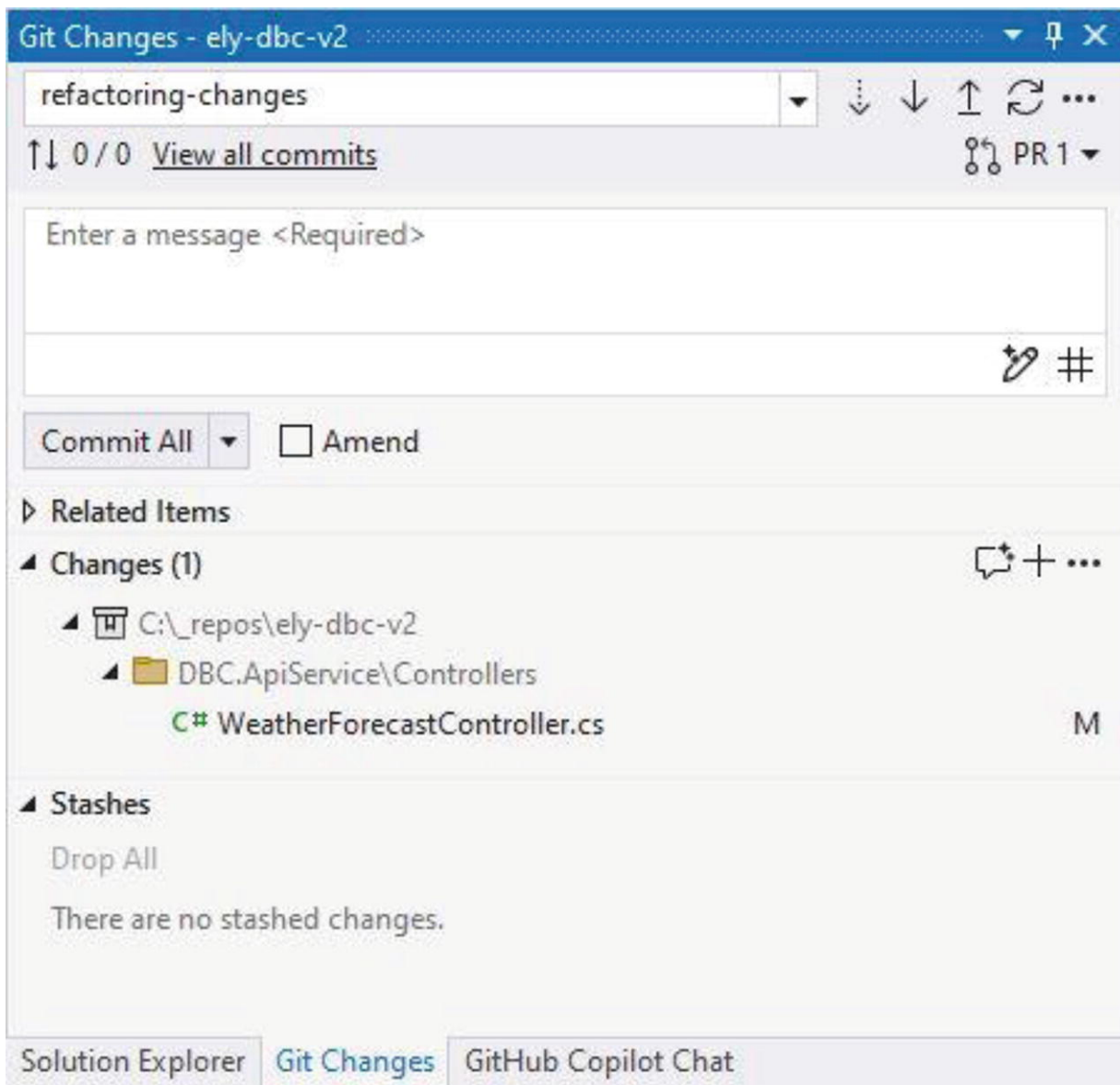
At this stage, your team's workflow dictates the next steps. You can action the comments received immediately by making the changes or discussing the changes with the reviewer to clarify intent. Either way, once you've resolved an issue, reply to the comment and mark it as resolved. You would then need to commit and push these changes to your remote branch. This action will automatically update the pull request with the new commits, allowing the reviewer to re-check and, if satisfied, approve the merge.

By leveraging Visual Studio's GitHub integration, you reduce context-switching, keep reviews closer to the code, and maintain a clear audit trail of decisions.

---

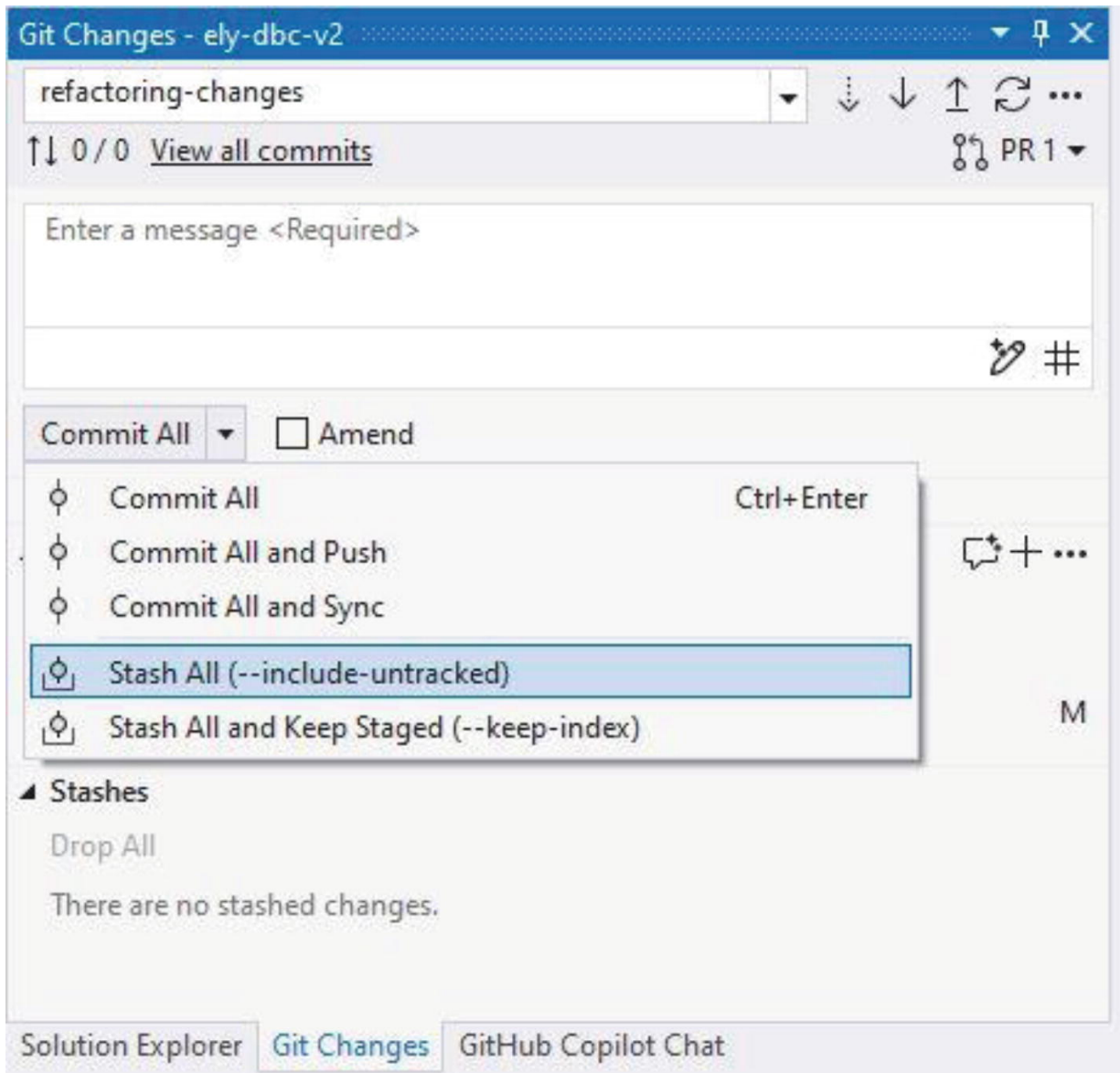
## Working with Stashes

It isn't uncommon to realize midway through a task that you have been working on the wrong branch.



**Figure 5-38** Changes in the Wrong Branch

In this example shown in Figure 5-38, I have made several edits to the controller, but the changes were intended for a new branch. This kind of oversight can happen easily when working with multiple branches. It is, however, easily remedied by using stashes.



**Figure 5-39** Stashing Changes

From the Git Changes window (Figure 5-39), clicking on the dropdown next to the *Commit* button will allow you to stash your changes. A stash stores your uncommitted changes locally and reverts your working directory to match the last commit made to the branch.

This makes it possible to switch to a new or different branch without losing any of your work. As shown in Figure 5-39, you have two options. These are to *Stash All* and *Stash All and Keep Staged*. The difference between them is as follows:

1. Stash All (--include-untracked)

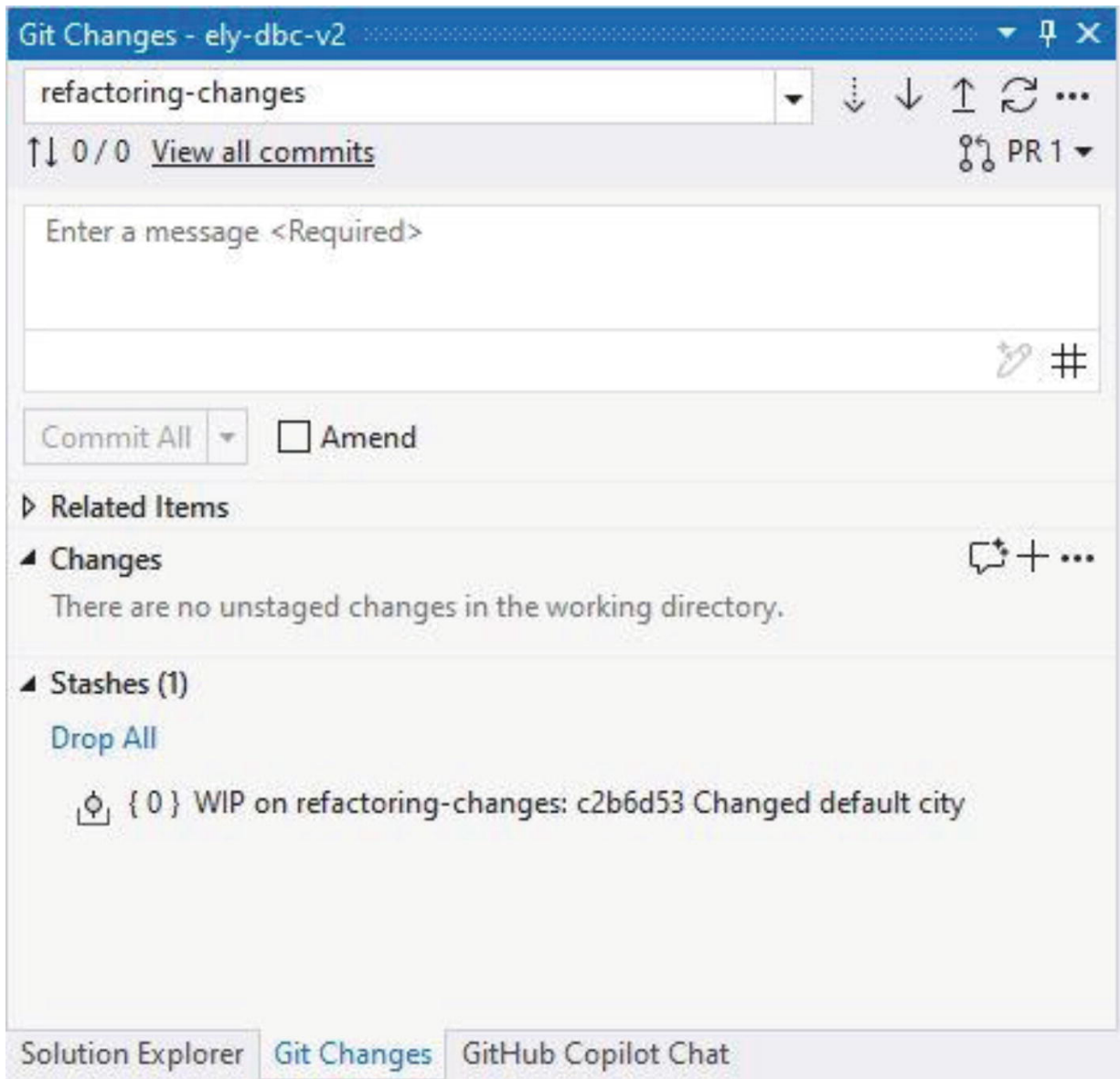
- What it does:  
Saves all changes in your working directory to a stash, including tracked files you have modified as well as new files that Git isn't tracking yet.
- What happens after:  
Your working directory is cleaned and matches the last commit. All untracked files are removed from the working directory (but they are safely stored in your stash).
- When to use:  
You will choose this option when you want to clear your workspace completely. This will include new files that have not been committed. You can then switch to another branch or work on something else without losing any work.

2.

#### Stash All and Keep Staged (`--keep-index`)

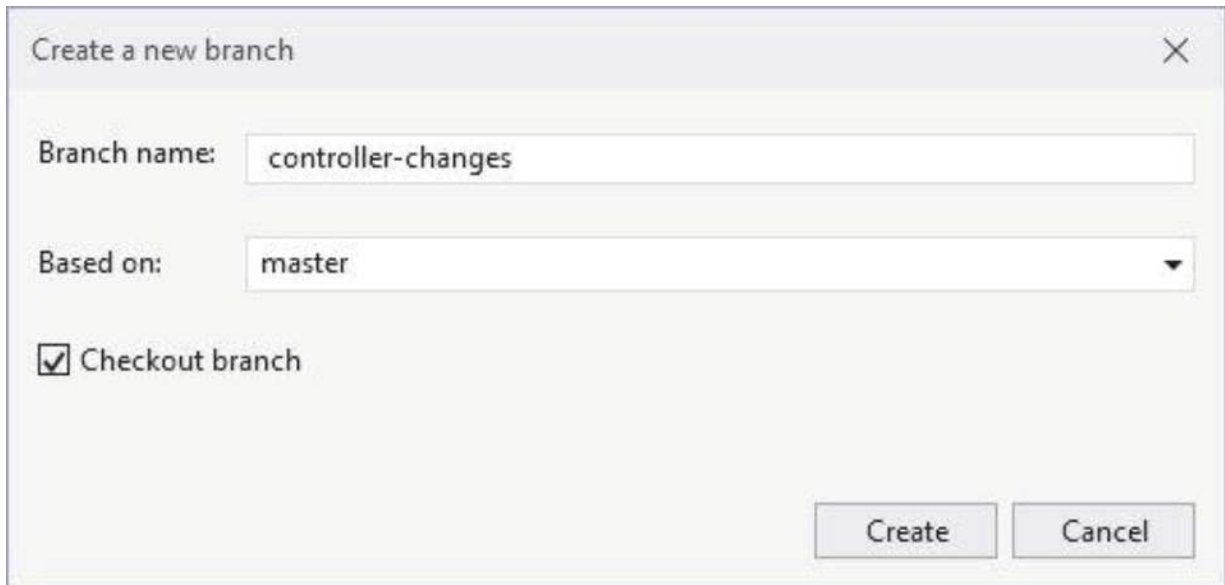
- What it does:  
Saves all changes in your working directory except the changes that you have already staged. Staged changes will remain staged and remain in the working directory. Only unstaged changes are stashed.
- What happens after:  
Your index (staging area) is left untouched. This means that you can continue with an in-progress commit or partial work while excluding the rest.
- When to use:  
When you have some work that you want to commit but also have changes that you want to exclude temporarily.





**Figure 5-40** Stashed Changes

With your changes stashed, the Git Changes window will display the stash under the Stashes section (Figure [5-40](#)). You are then free to switch branches as required.

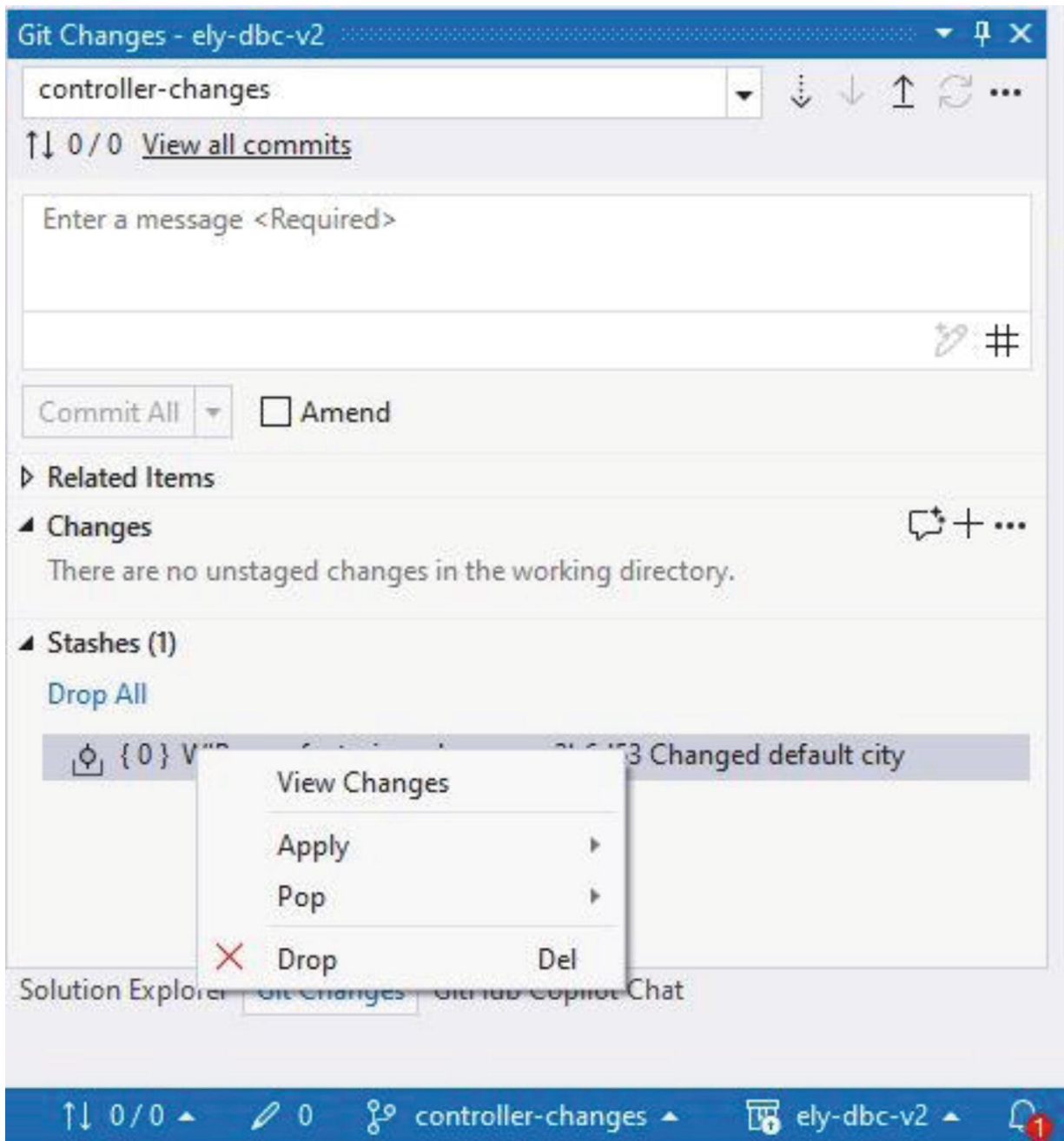


**Figure 5-41** Creating a New Branch

In this example (Figure [5-41](#)), I am creating a totally new branch for my changes.

Refer back to the section on creating branches earlier in this chapter if you need a refresher on the process to follow.

I am basing my new branch on the master branch and checking the branch out after creation.



**Figure 5-42** Applying or Popping a Stash

After switching to the correct branch, you can decide how to reapply the stashed changes as shown in Figure 5-42. You have three options:

- Apply – Merge the stashed changes into the current branch while keeping the stash.
- Pop – Merge the stashed changes into the current branch and remove the stash.

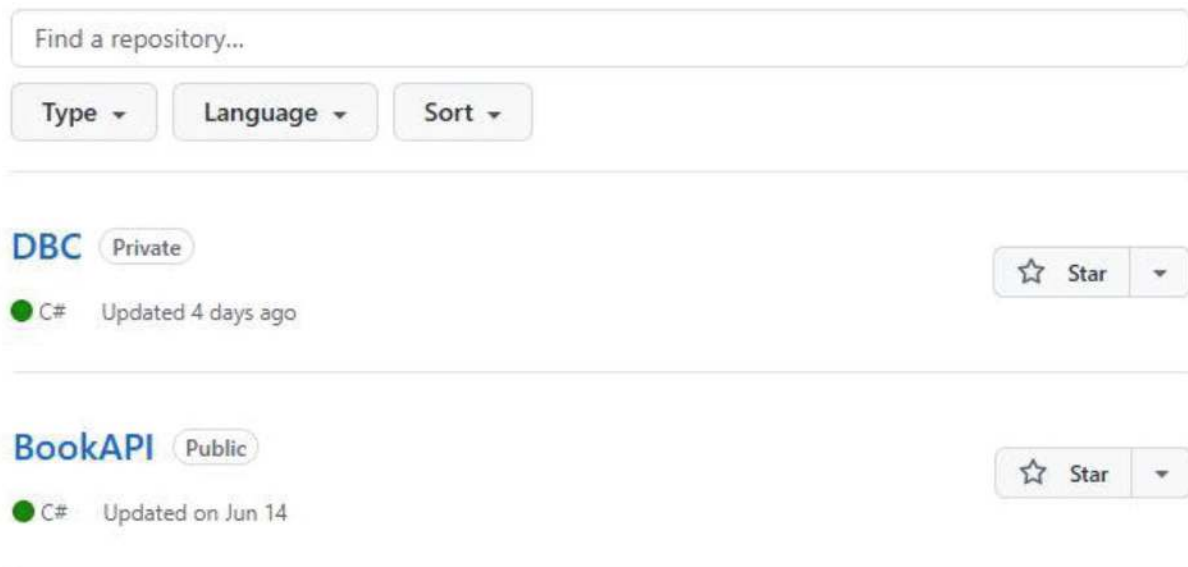
- Drop – Permanently delete the stash without applying it to the current branch.

Stashing is useful if you need to pause your current work to deal with a high-priority task, for example, a production bug fix. You can stash your changes, switch branches, fix and push code fixes, and then return to the current task at hand by applying the stash. The workflow followed here helps keep your branches clean and prevents code from slipping into branches they weren't intended for.

---

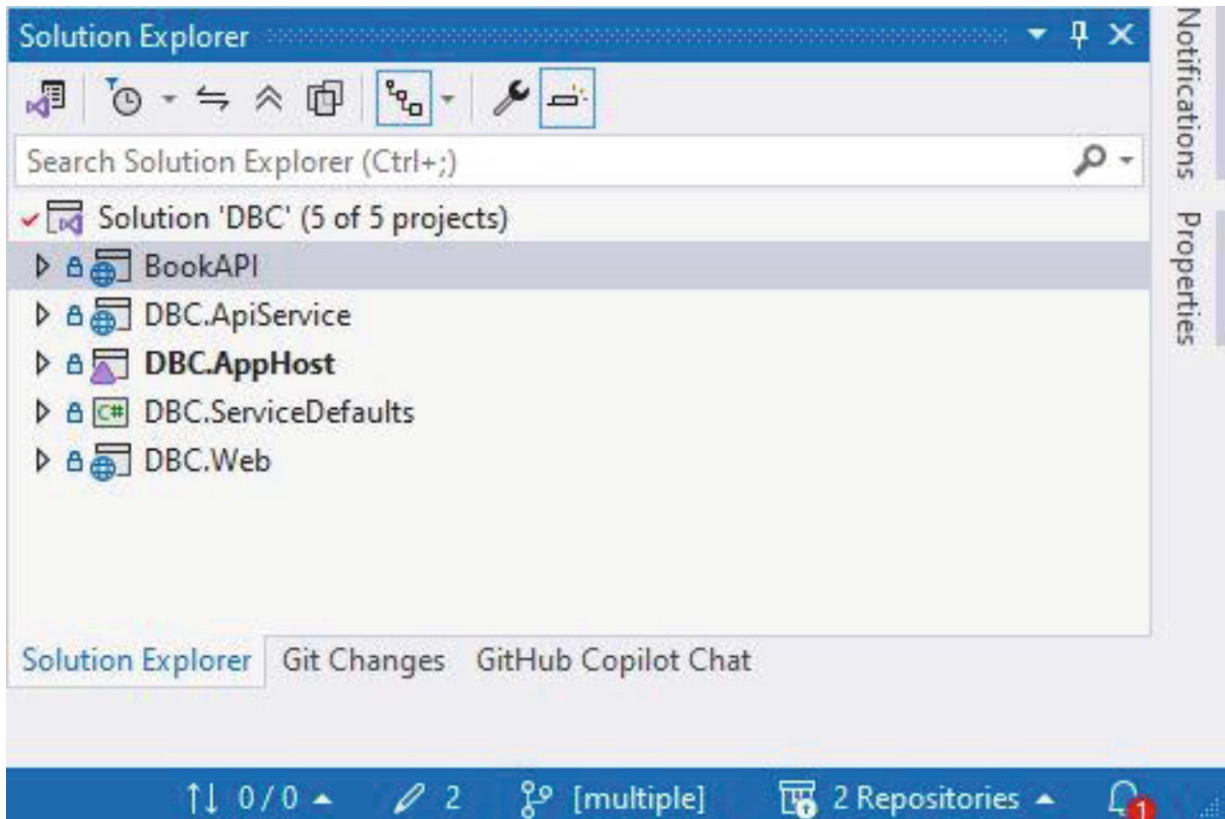
## Multi-repo Support

A nice feature in Visual Studio 2022 is multi-repo support. You can now work in a single solution, with projects hosted in different Git repositories.



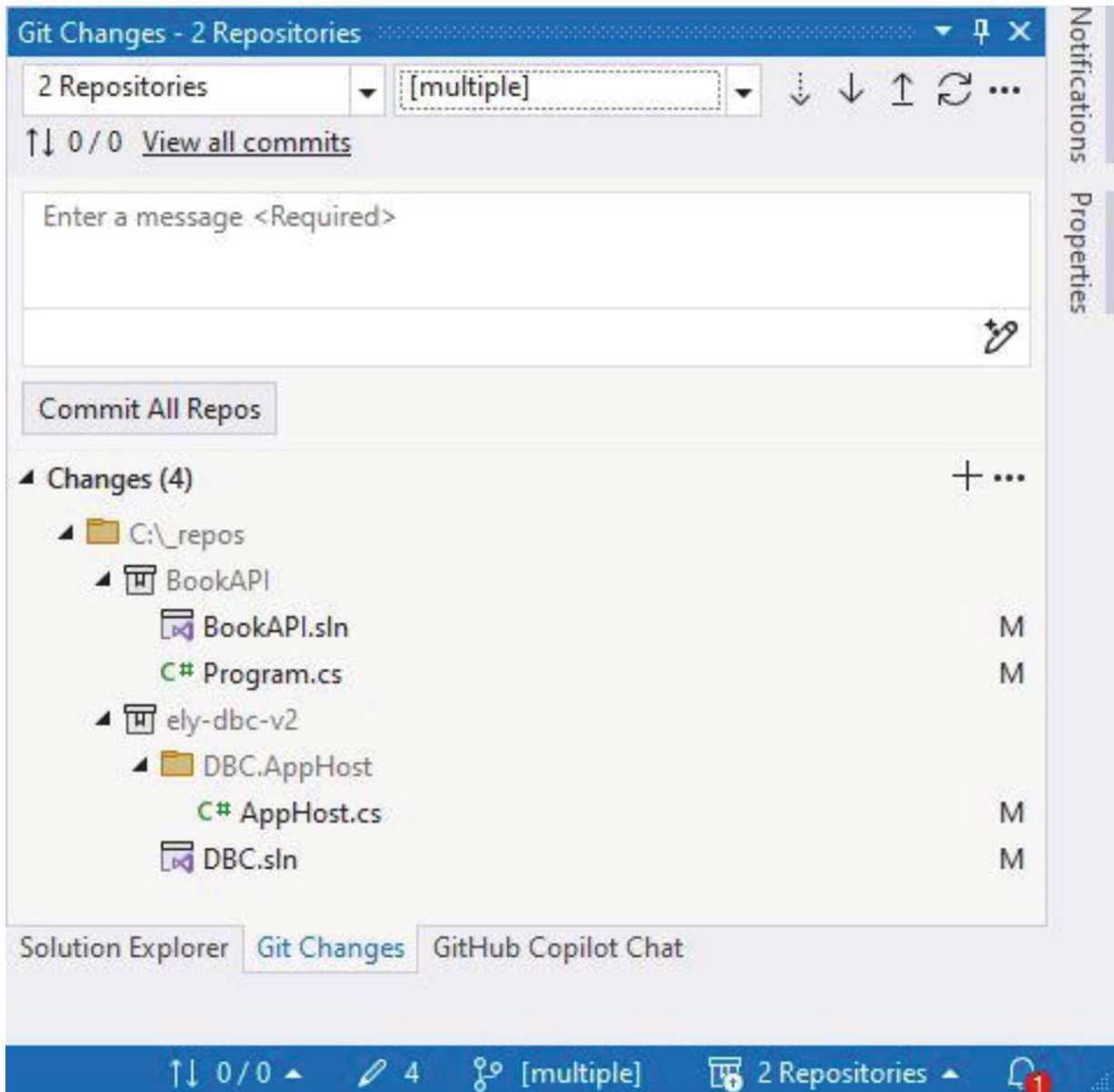
**Figure 5-43** Multiple GitHub Repos

I have two repositories in GitHub: an API project and the DBC project shown in Figure [5-43](#). With multi-repo support, I can now open one project in Visual Studio 2022 and then add an existing project by right-clicking the solution and selecting *Add ► Existing Project* and select the second project.



**Figure 5-44** Multi-repos Open in Visual Studio

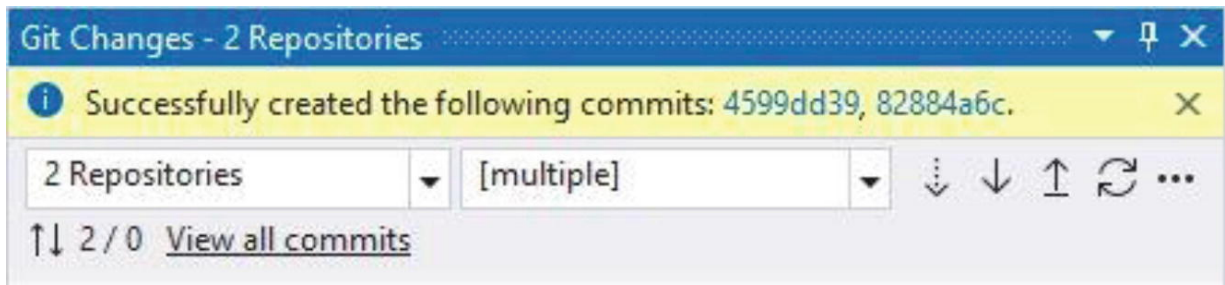
As seen in Figure [5-44](#), both my projects across both repositories are loaded into Visual Studio, and the repo in the status bar shows that I am working with two repositories.



**Figure 5-45** Changes in Both Repositories

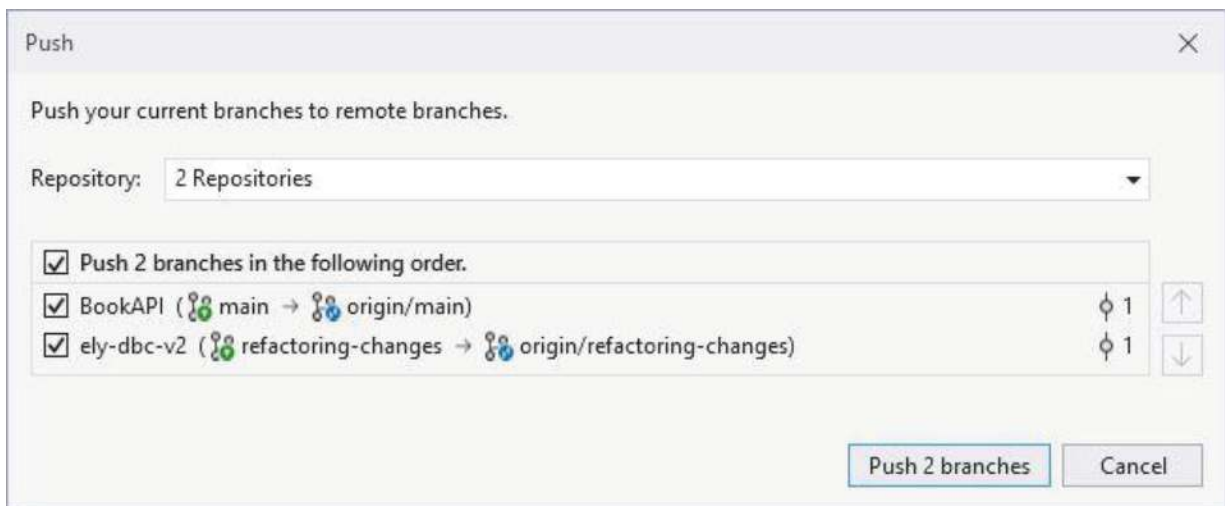
As seen in Figure 5-45, I can make changes in both repositories and commit all these changes locally by clicking the *Commit All Repos* button.





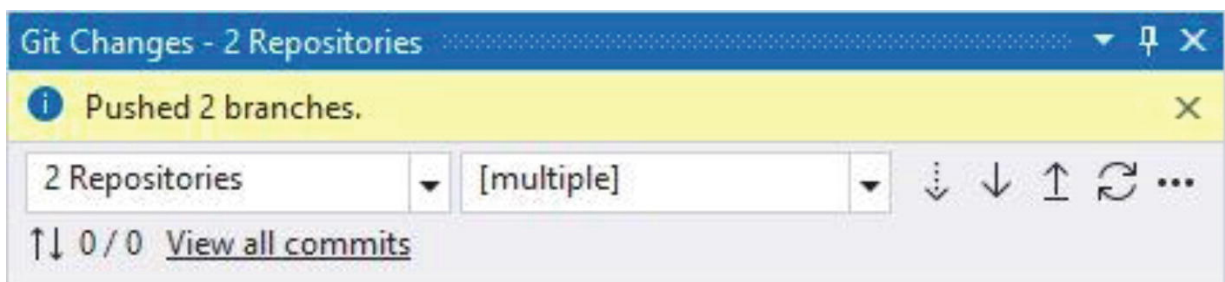
**Figure 5-46** Changes Committed Locally

Once these changes are committed locally, I can click on the Push button (the up arrow seen in Figure 5-46). This will display a Push dialog shown in Figure 5-47.



**Figure 5-47** Select the Branch to Push

You can select both branches or choose to push them one by one. In Figure 5-47, I have selected to push both branches in both repositories. The commit notification in Figure 5-48 shows that both branches were pushed successfully.



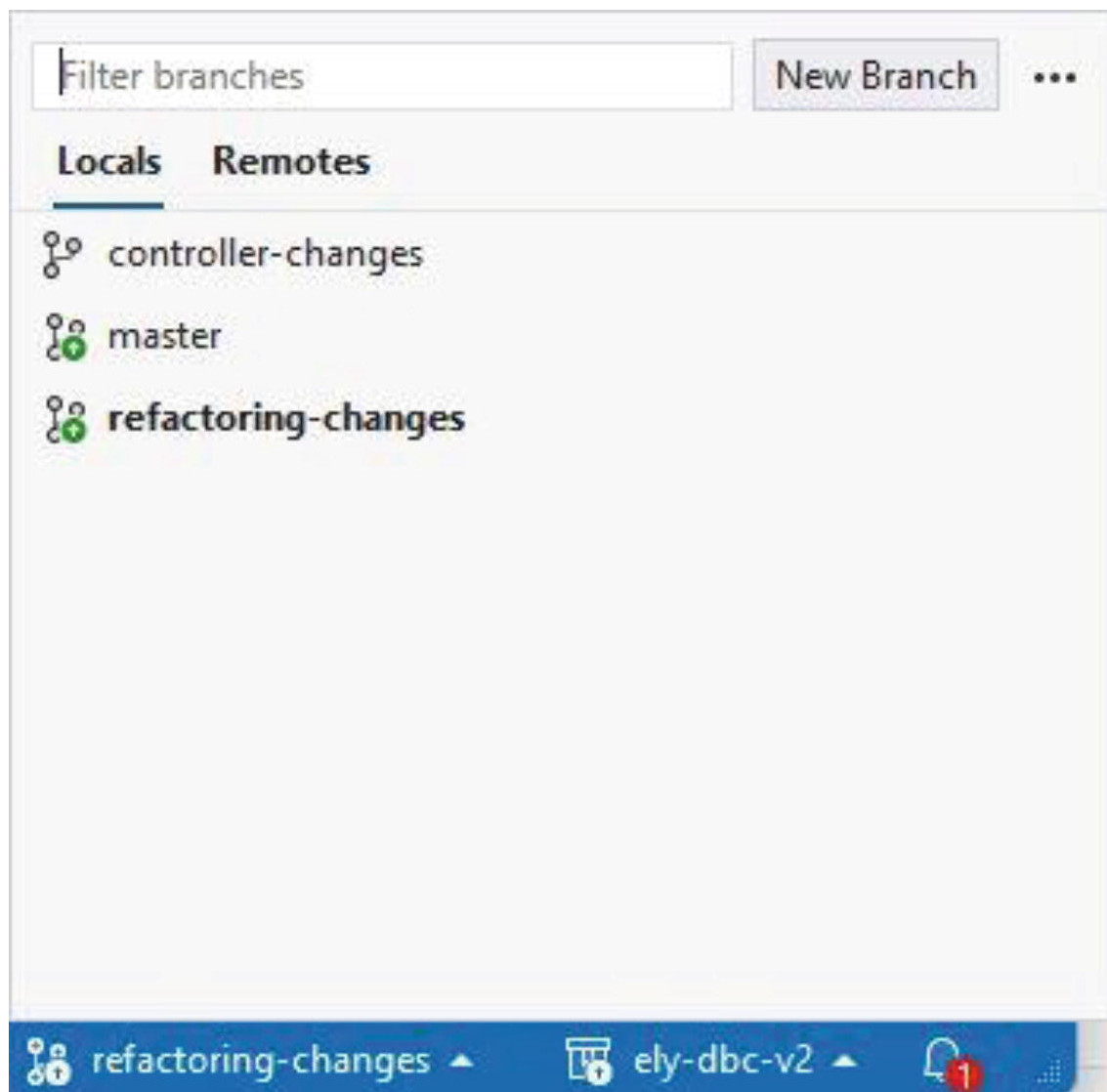
**Figure 5-48** Both Branches Pushed

Having the ability to work with multiple repositories in Visual Studio is convenient because I no longer need to open a separate instance of Visual Studio to work on a different repository.

---

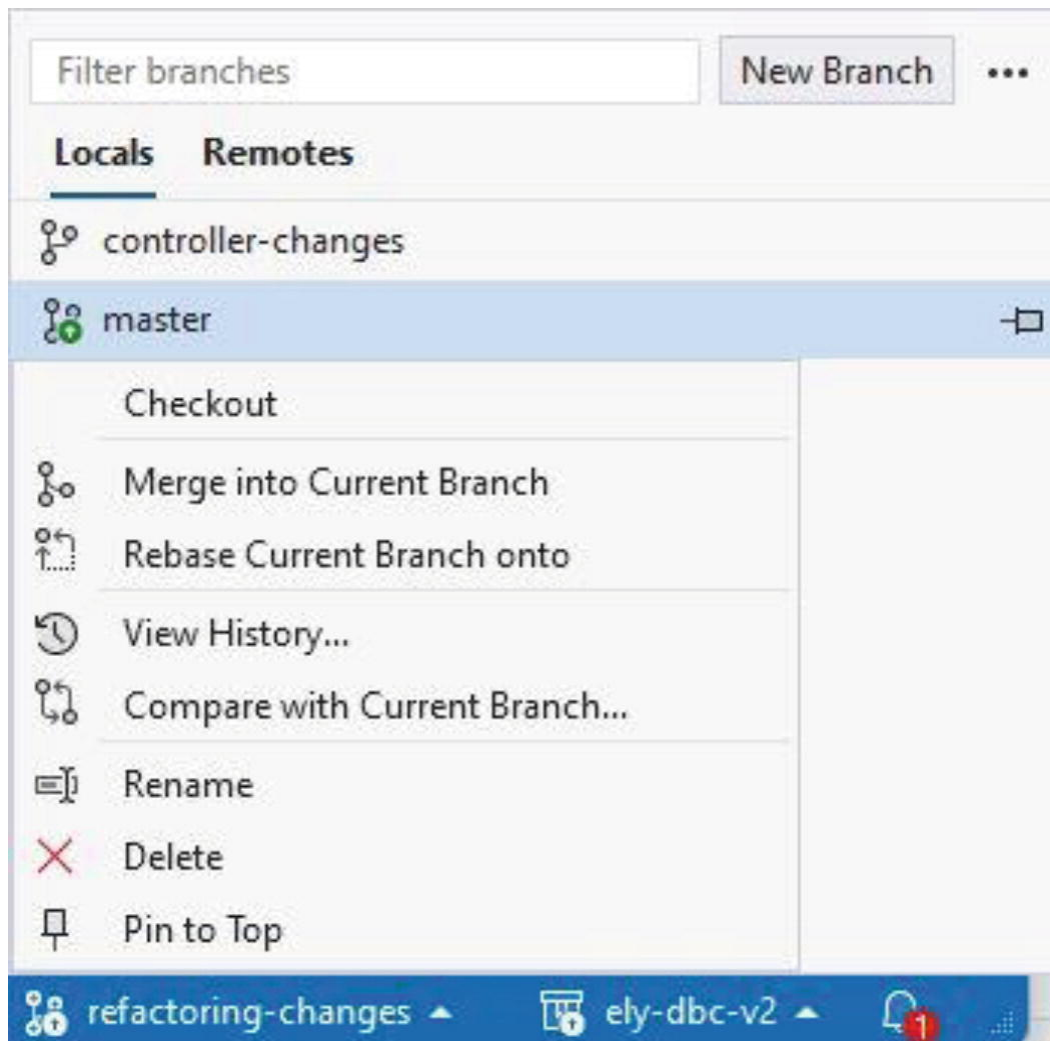
## Compare Branches

Visual Studio 2022 now allows you to compare branches. This provides a convenient way to see the differences between the two branches you are comparing and will be helpful before creating a pull request, before merging, or even when choosing to delete a branch.



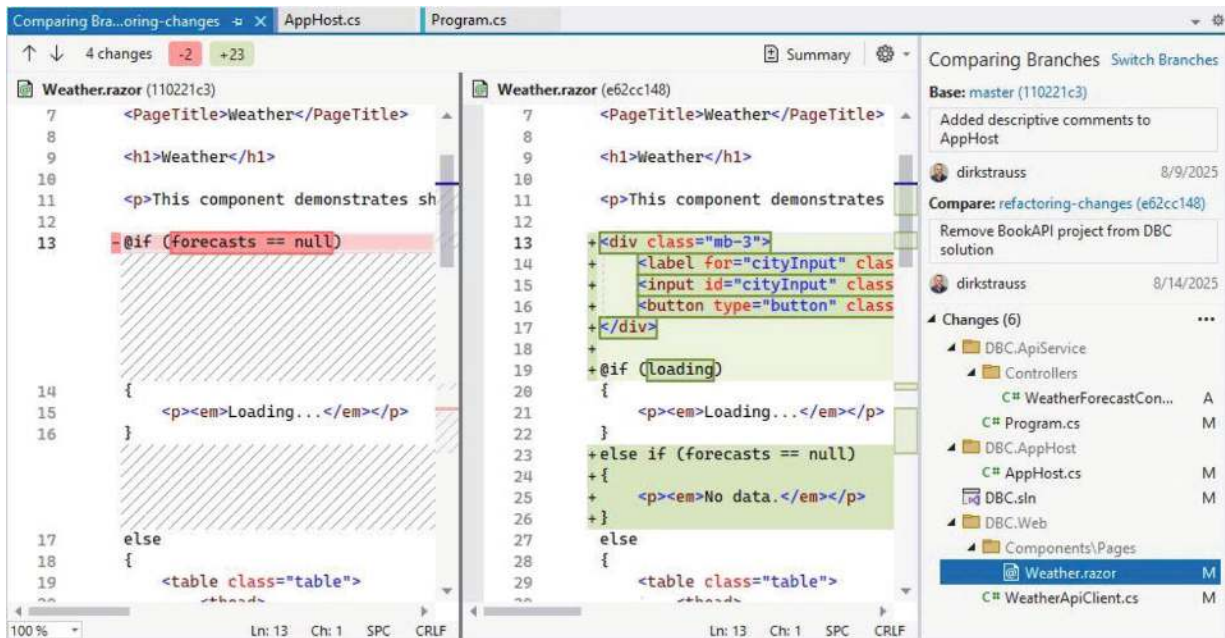
**Figure 5-49** View the Branches to Compare

As seen in Figure [5-49](#), when I click the currently checked out branch, I can see all the other branches for my project, too. To compare the current branch with the master branch, for example, right-click the master branch and select Compare with Current Branch from the context menu, as seen in Figure [5-50](#).



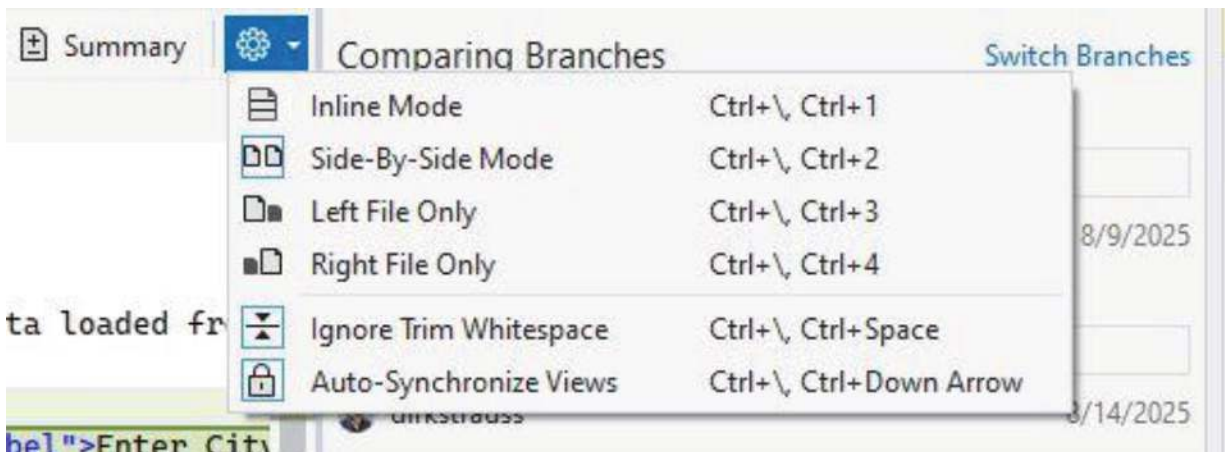
**Figure 5-50** Compare to Master Branch

Visual Studio will then display a diff between the two branches you selected to compare, as seen in Figure [5-51](#).



**Figure 5-51** The Branch Diff

I am now able to see exactly what is different between the two branches without leaving Visual Studio.

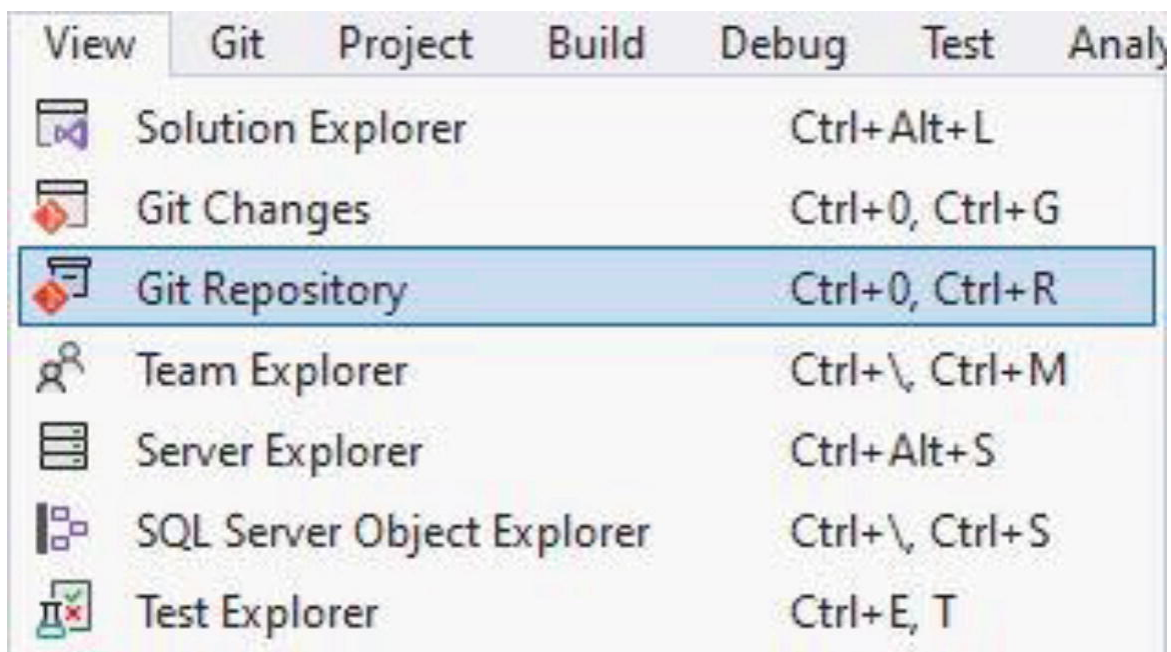


**Figure 5-52** Changing the Preferred Diff Layout

I can also use the diff configuration options gear in the top-right corner to switch to an inline diff view, as shown in [Figure 5-52](#).

## Check Out Commit

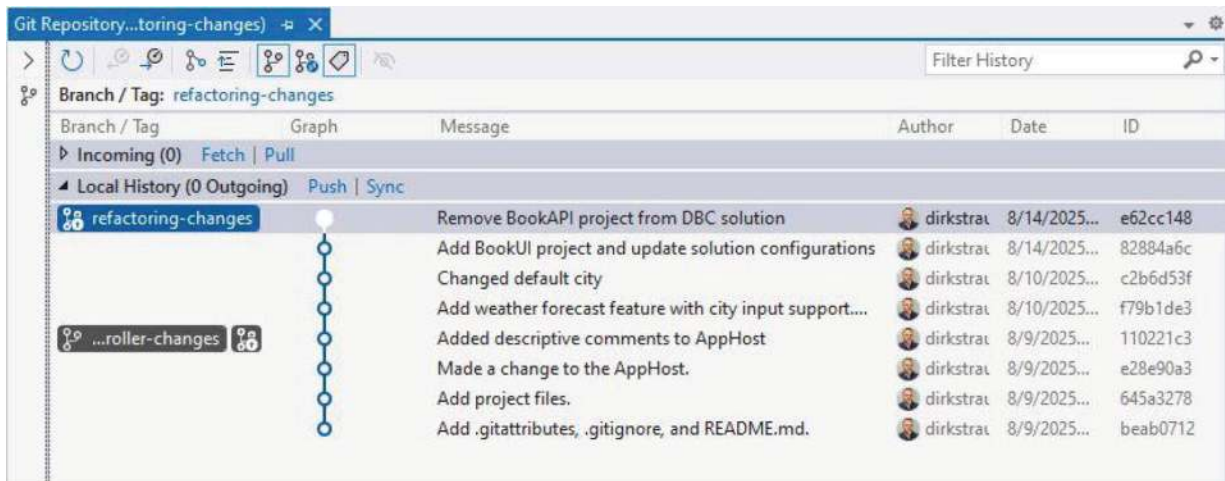
When working with Git in Visual Studio, you are not limited to the latest state of a branch. You can check out any commit in your repository's history and load the project exactly as it existed at that point in time. You are effectively traveling back in time to a previous state of your codebase. This can be invaluable when you need to investigate a bug, validate a fix, or just review how a specific feature was originally implemented. Visual Studio makes this process seamless, allowing you to freely move between different points in your project history. To do this, open the Git Repository window by selecting the View menu and clicking *Git Repository* as seen in Figure [5-53](#).



**Figure 5-53** View the Git Repository

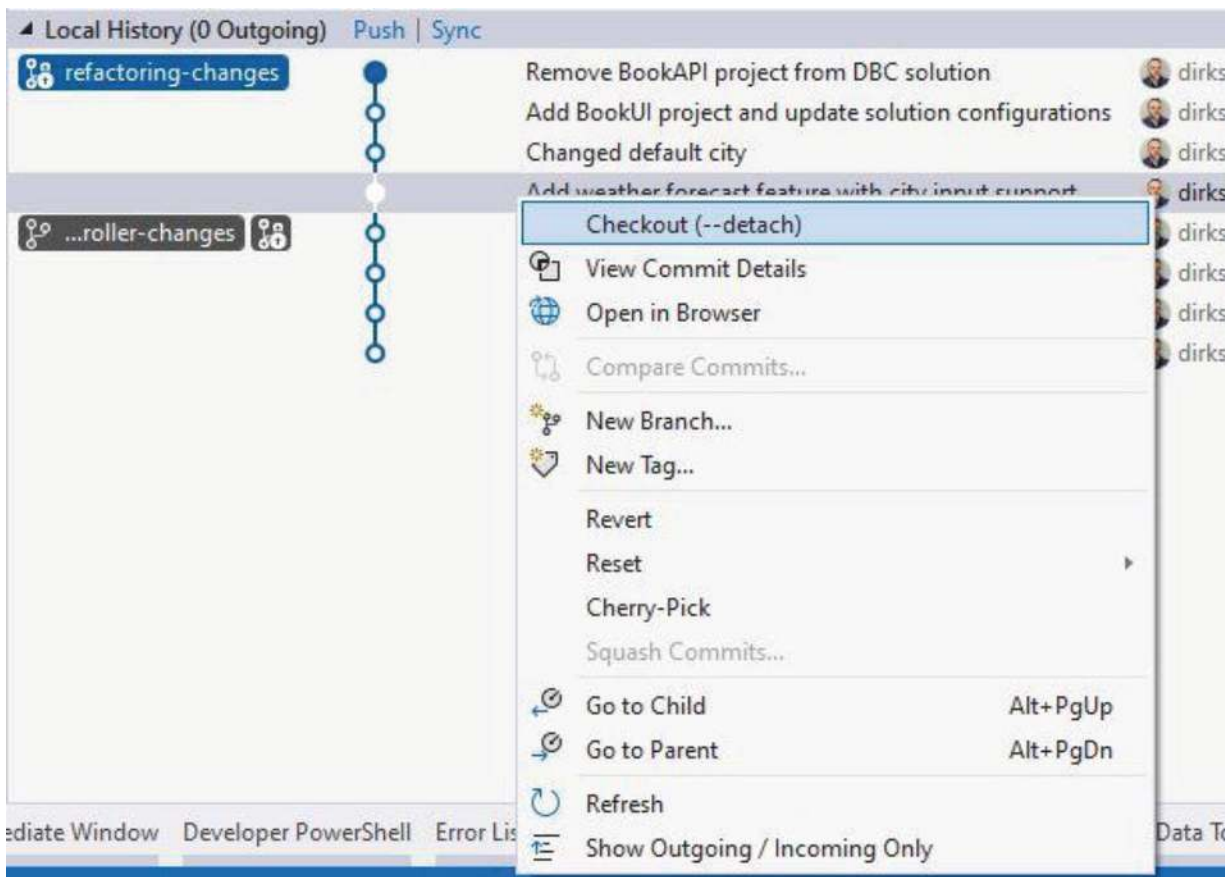
You can also hold down Ctrl+O, Ctrl+R to open the Git Repository, which can be seen in Figure [5-54](#).





**Figure 5-54** The Git Repository Window

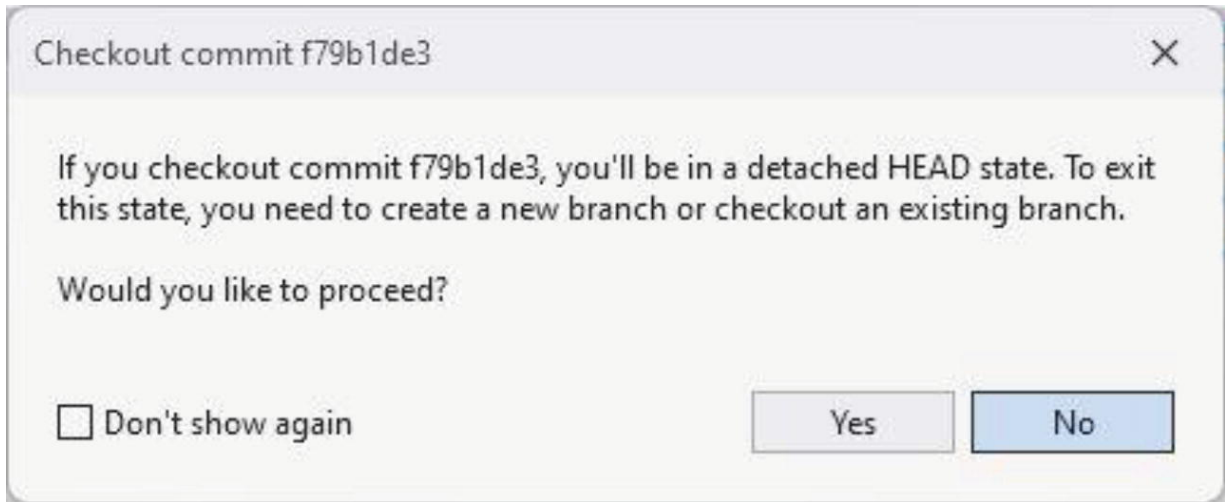
To check out a specific commit, right-click the commit in the Git Repository window shown in Figure 5-55 and select Checkout (--detach) from the context menu.



**Figure 5-55** Check Out a Commit

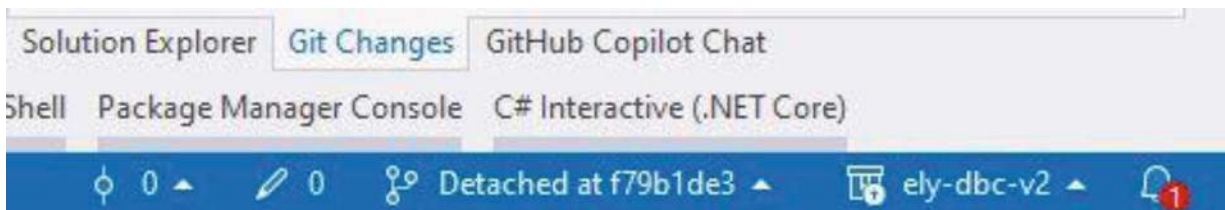


Visual Studio now displays a confirmation window (Figure 5-56) to inform you that by checking out this commit, you will be in a detached HEAD state. In other words, your repo's HEAD will be pointing to a specific commit instead of a branch.



**Figure 5-56** The Checkout Commit Confirmation

The status bar of Visual Studio will now show that you are currently pointed to a specific commit instead of to a branch (Figure 5-57).



**Figure 5-57** The Visual Studio Status Bar

Checking out a commit provides the following benefits:

- *Debug historical issues* – Specifically when a regression appears, you can check out the commit where the code last worked, allowing you to compare it with later commits to identify the issue.
- *Reproducing bugs* – Customer reported issues might be tied to a specific version of the code. You can therefore reproduce the bug by checking out the commit matching the deployed build.
- *Code “archaeology”* – Being able to understand why a section of code was written in a specific way. Being able to check out a specific

commit allows you to see the code in the exact context of that decision.

- *Avoiding unnecessary branch creation* – If all you need to do is review or test code without making any modifications, checking out the commit directly is faster and keeps the structure of your branches clean.

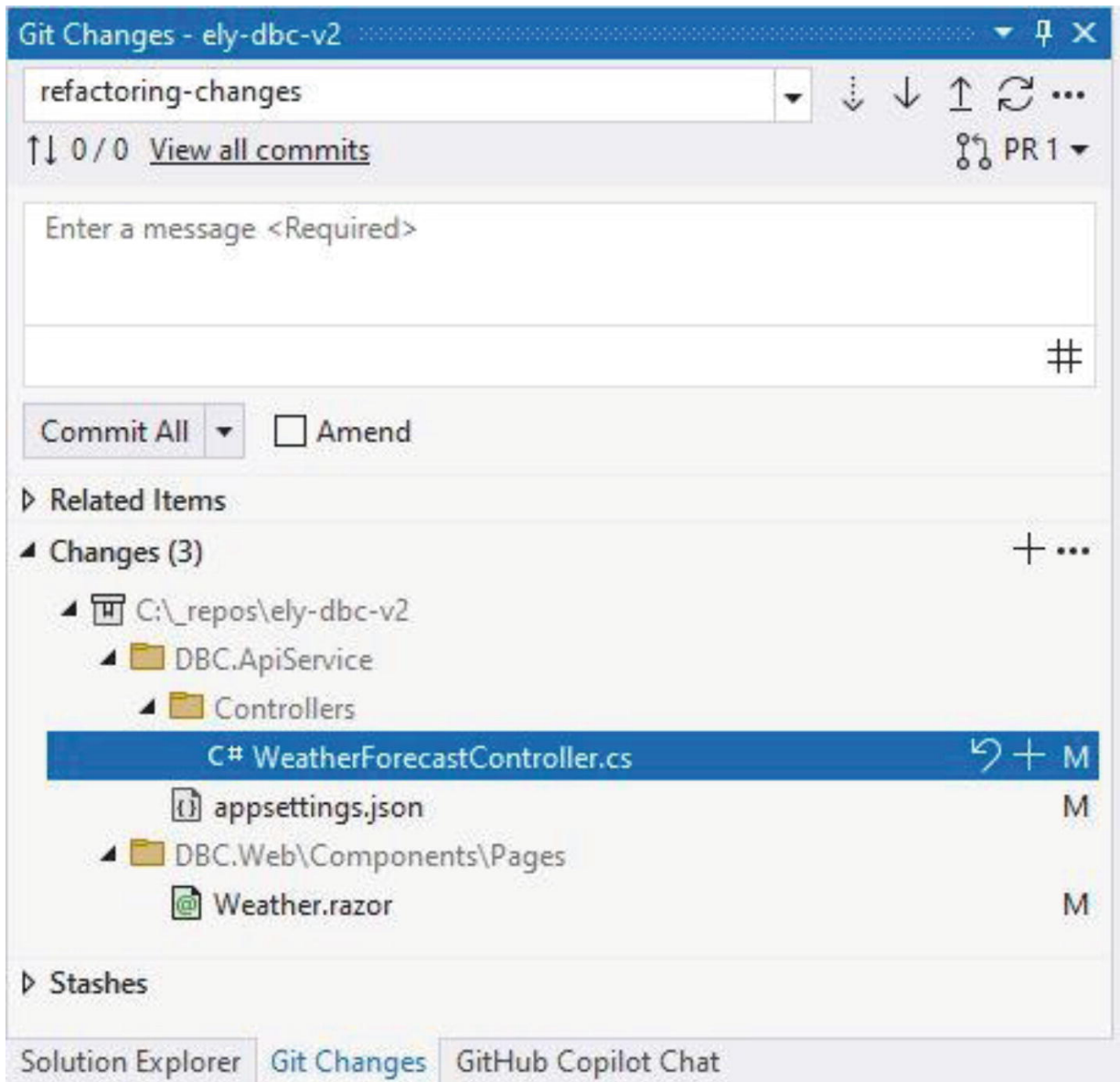
Being able to check out a specific commit is more than just going back in time. It can be a precise tool used for debugging, analysis, and validation without the clutter of temporary branches. To get back to your branch, simply select it from the branch selector in the Visual Studio status bar.

---

## Line Staging

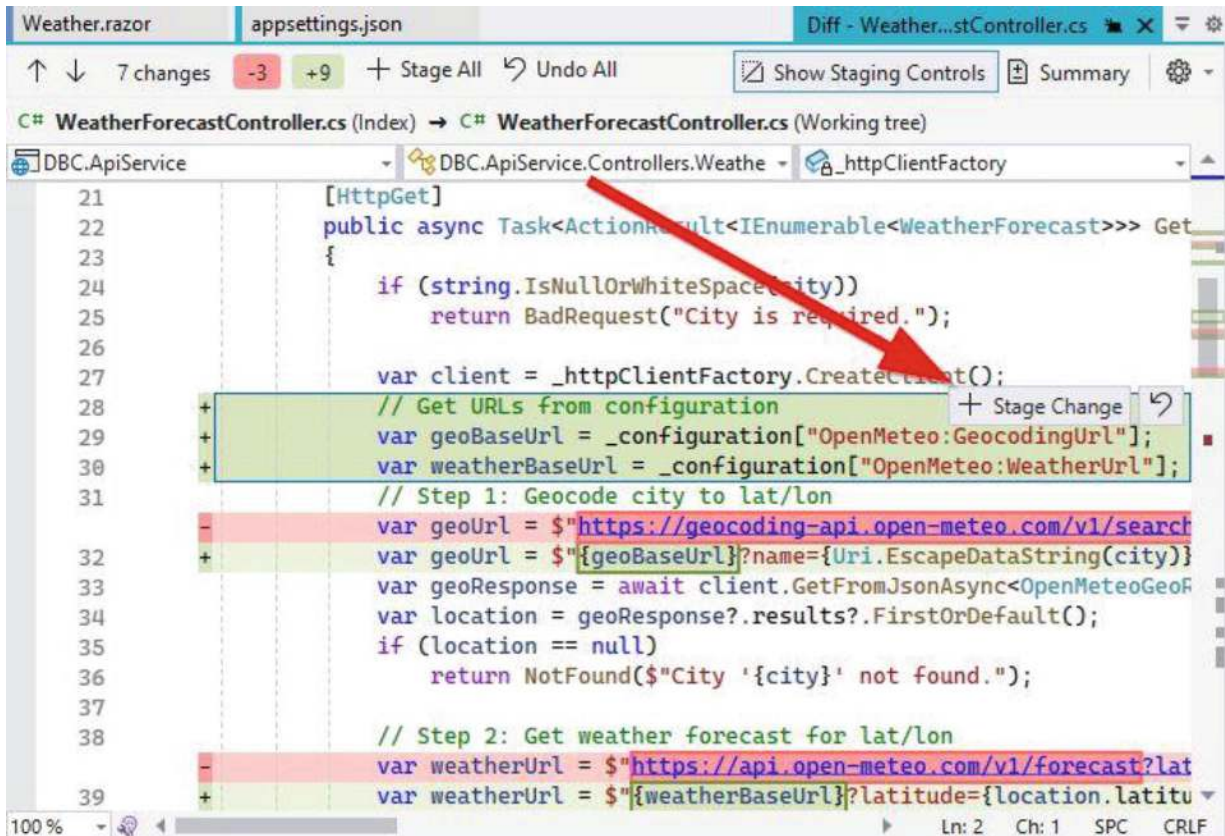
Also known as interactive staging, line-staging allows you to stage selected lines or portions of a file instead of the entire file at once. This level of precision is valuable when your edits span multiple concerns. You might be fixing a bug while also refactoring a small piece of code, and adding some experimental code, all at the same time. With line staging, you do not need to bundle all these unrelated changes together. You can selectively stage lines that belong together and leave the rest for later.

To see line staging in action, make some changes to one of your files, and then view the file in the *Git Changes* window as seen in Figure [5-58](#).



**Figure 5-58** Git Changes Showing Changed Files

Next, double-click the file that you want to apply line staging to. Visual Studio will display a diff view of the selected file, allowing you to hover over the lines that you want to stage, and stage them individually as shown in [Figure 5-59](#).



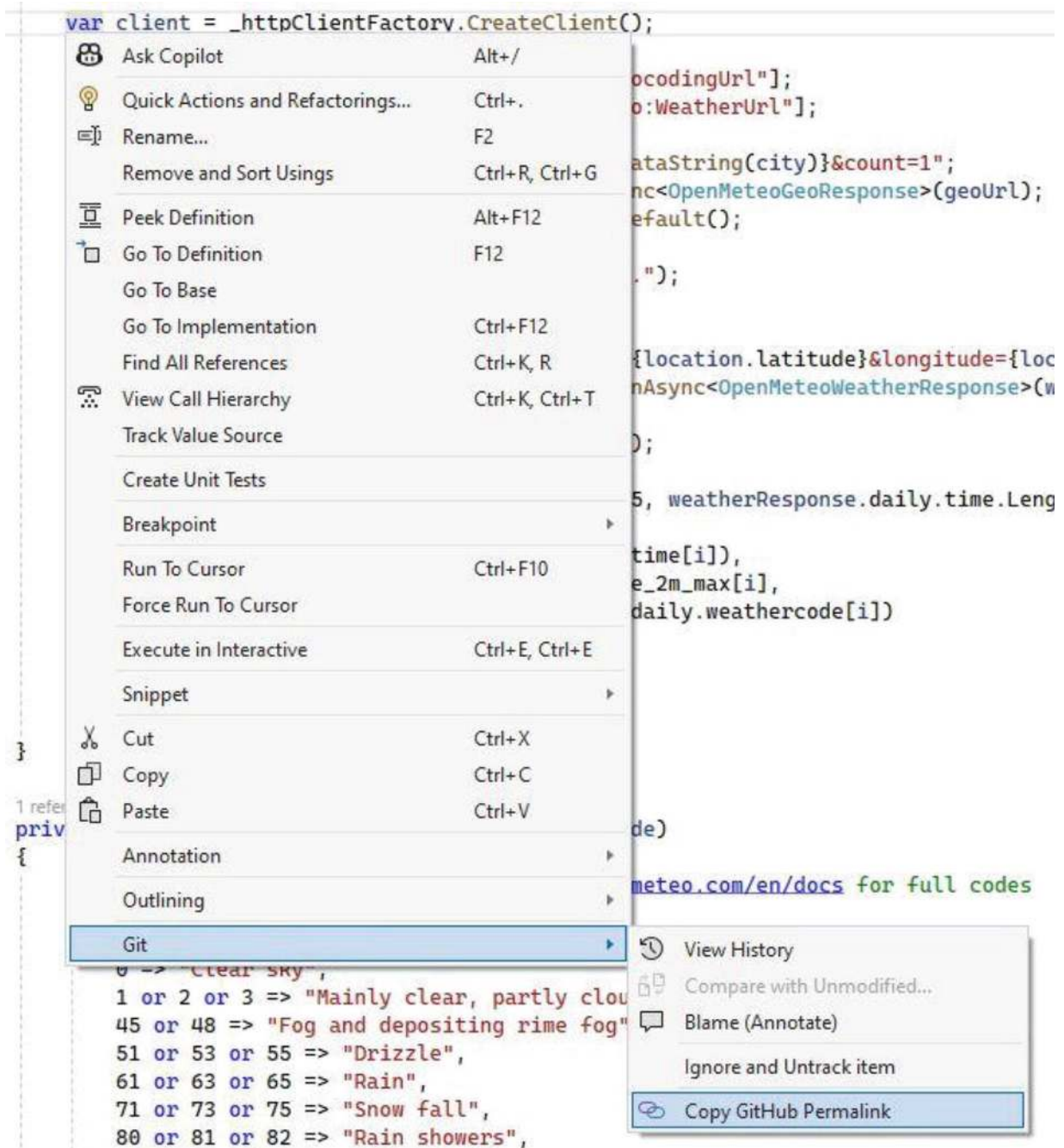
**Figure 5-59** Staging a Specific Line of Code

Line staging also supports side-by-side and inline diff modes. Line staging turns staging into a surgical tool rather than a blunt instrument. It helps developers maintain discipline in version control, produces a cleaner history, and ultimately leads to more maintainable codebases.

## Copy Git Permalink in Visual Studio

Visual Studio 2022 introduces the ability to copy a Git permalink directly from the editor. A permalink is a URL that points to a specific line or block of code at a given commit. This is quite useful when reviewing code with a colleague or documenting behavior in a wiki. Instead of having to refer to a line number in code, you can simply share a link that takes the recipient to the exact line of code in question.





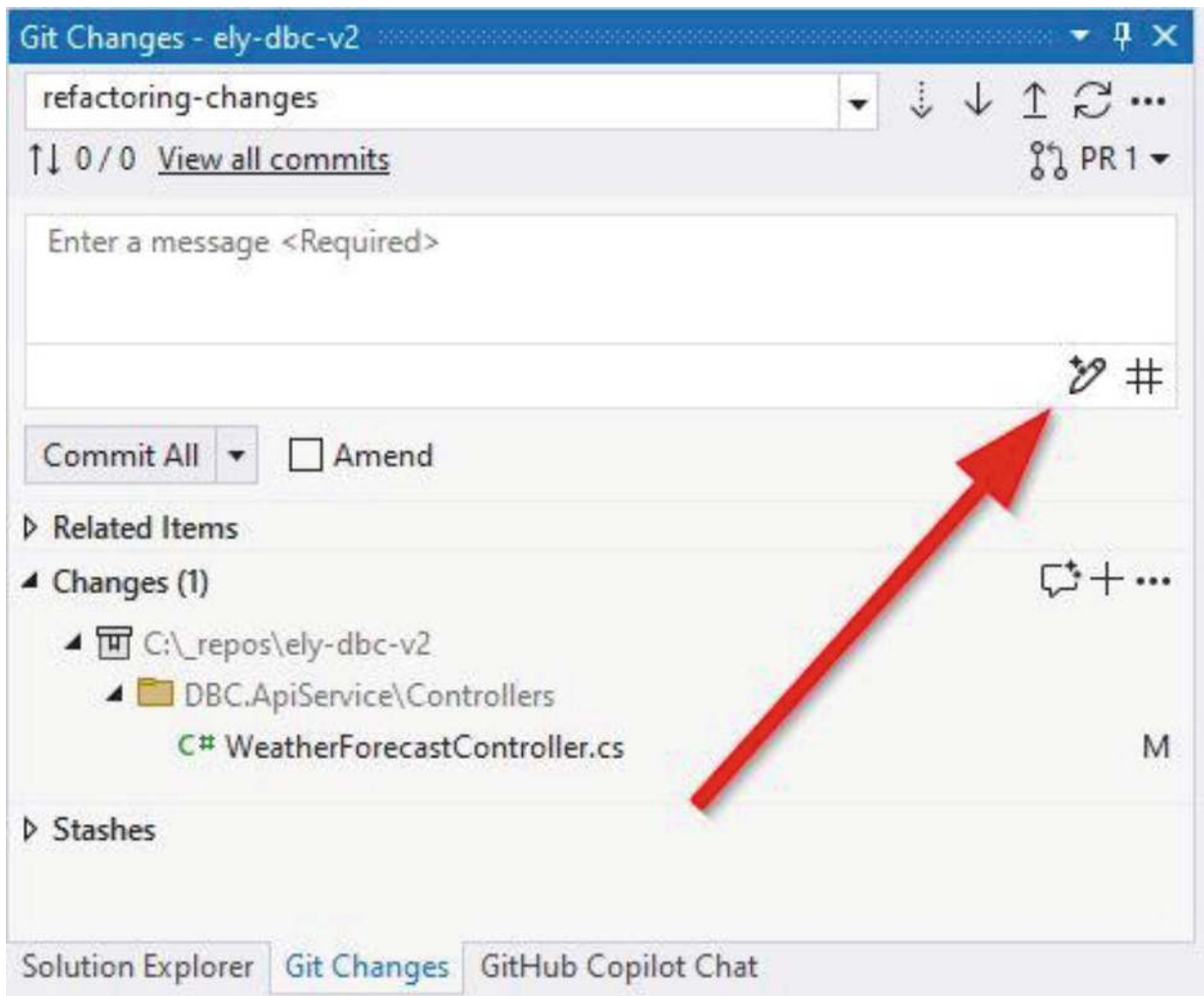
**Figure 5-60** Copying a Permalink

To copy a permalink, right-click a line of code and select **Git** ➤ **Copy GitHub Permalink** from the context menu. Your colleague can then open the link you copied to view the specific line of code on the web. This allows them to gain valuable context of the issue at hand without disrupting their local dev environment or workflow.

Note that this feature is supported in both Azure DevOps and GitHub Repositories.

## Commit with GitHub Copilot

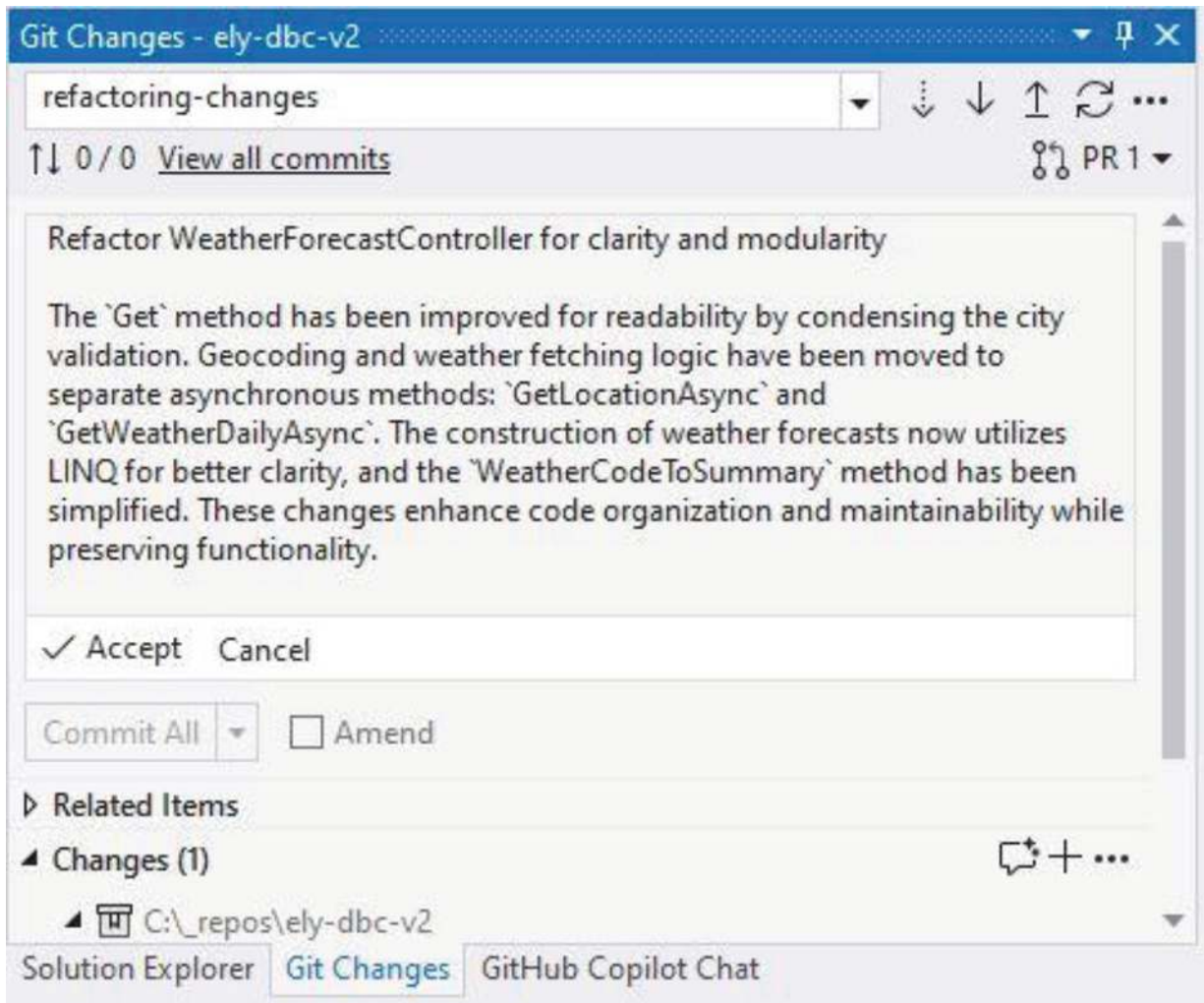
GitHub Copilot can generate commit messages directly from your changes. By clicking on the pencil icon in the Git Changes window (Figure 5-61), Copilot inspects the diff and proposes a commit message.



**Figure 5-61** Generate Commit Message

Shown in Figure 5-62, the commit message proposed by Copilot does save you time, but relying solely on the default output can lead to inconsistent messages, especially if your team likes to create commit messages in a specific format.



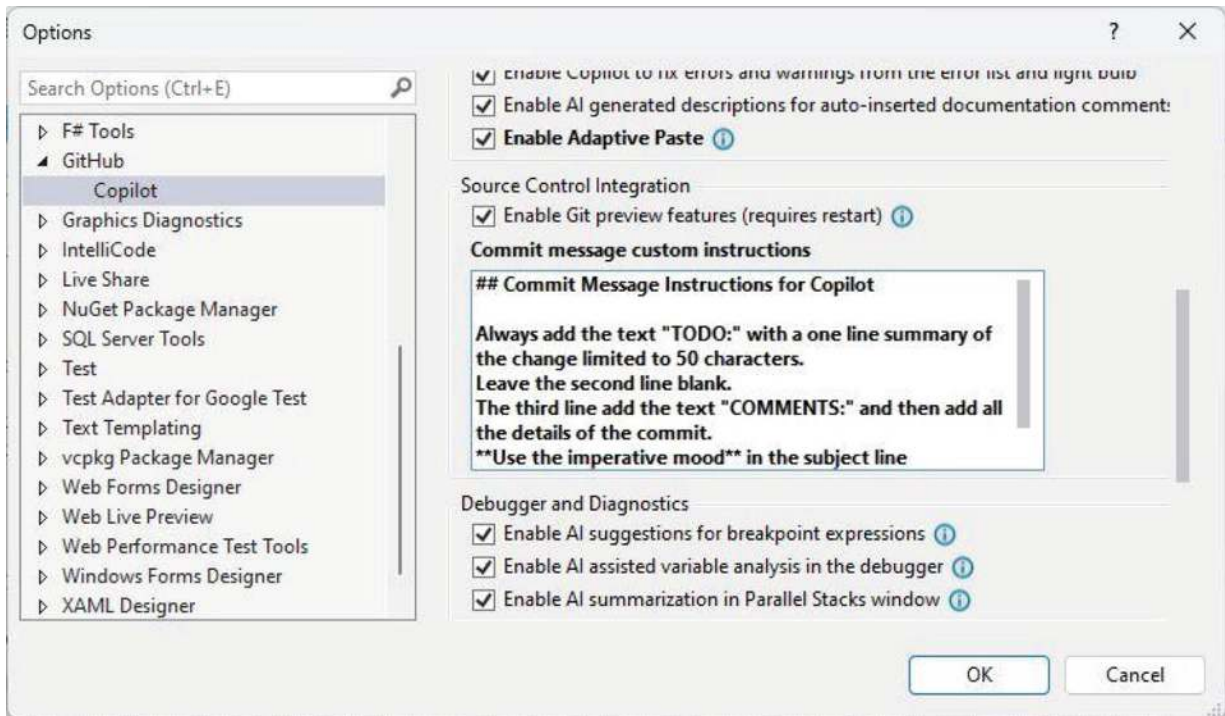


**Figure 5-62** Generated Commit Message

Visual Studio provides a way to enforce a specific standard when it comes to Copilot-generated commit messages by allowing you to define custom instructions.

As seen in Figure 5-63, I can specify the custom instructions by going to Tools ► Options ► GitHub ► Copilot and specifying a set of rules that help guide Copilot's generation process.

Think of the custom instructions as a style guide that Copilot applies every time you ask it to write a commit message. This ensures that your commit messages remain structured and consistent.



**Figure 5-63** Commit Message Custom Instructions

For example, consider the custom instructions shown in Listing [5-1](#).

```
## Commit Message Instructions for Copilot
```

```
Always add the text "TODO:" with a one-line
summary of the change, limited to 50 characters.
Leave the second line blank.
```

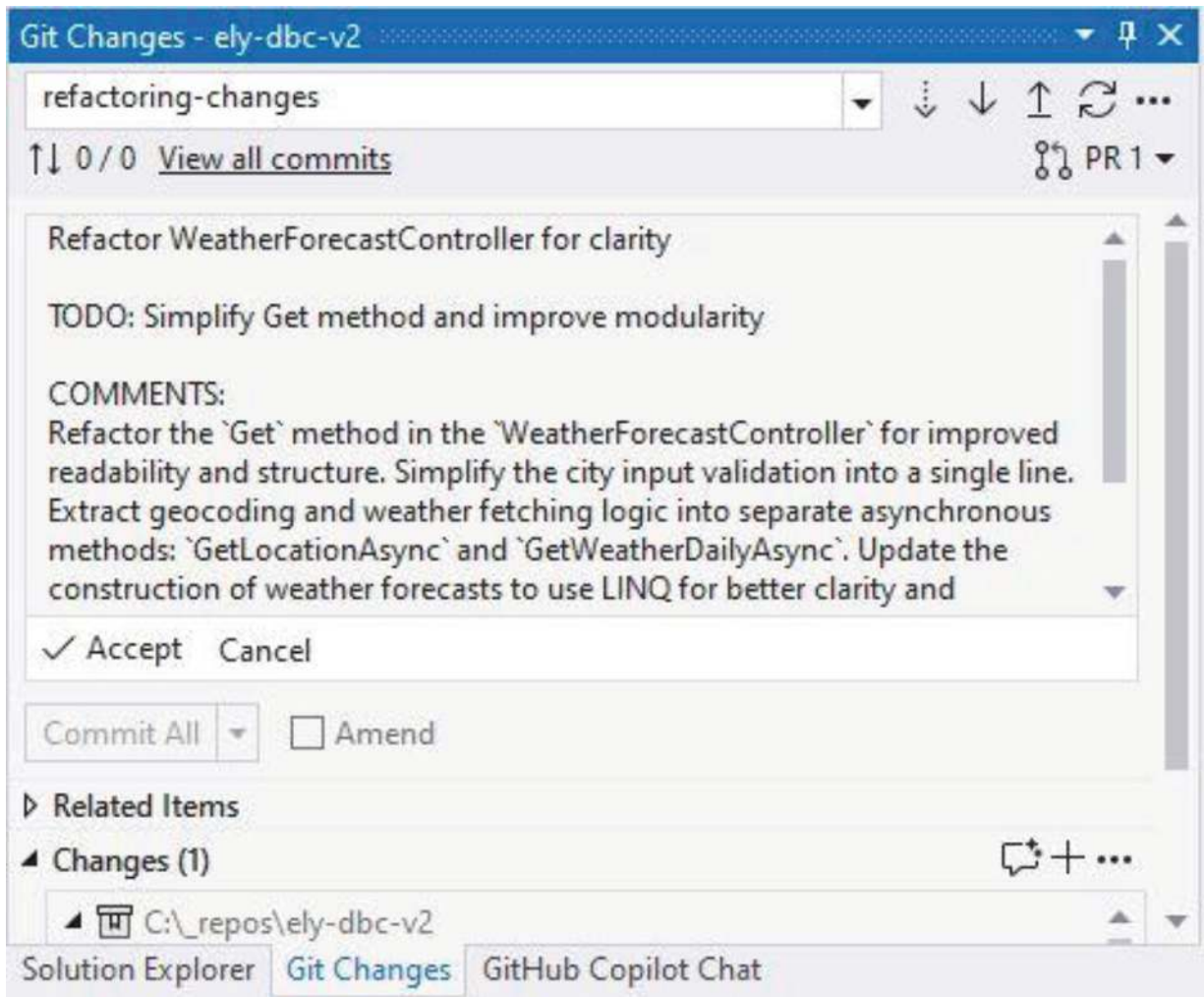
```
The third line adds the text "COMMENTS:" and then
adds all the details of the commit.
```

```
**Use the imperative mood** in the subject line
```

```
- Example: *"Add logging for API requests"* not
*"Added logging"*
```

**Listing 5-1** The Commit Message Custom Instructions

With these instructions in place, Copilot’s generated commit message follows a predictable format. Defining a structured commit message format enforces industry-accepted conventions, ensuring each change is documented in a consistent and predictable way. This makes the rationale behind a change clear and easier for the team to follow.



**Figure 5-64** Commit Message Using Custom Instructions

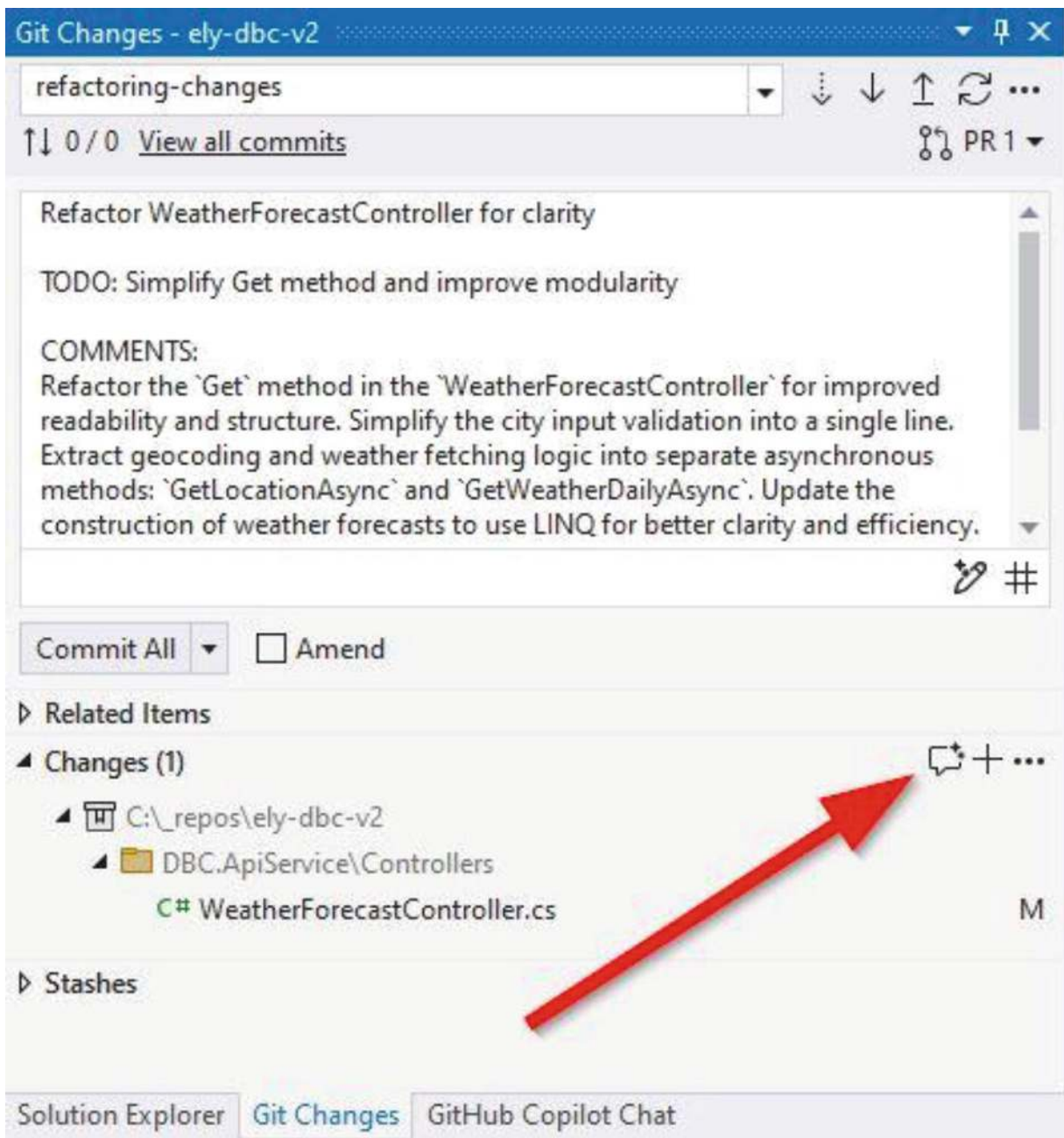
When you generate the same commit message using the custom instructions, the difference is striking. Instead of generic or verbose commit messages, Copilot now produces entries that align with established best practices, reducing ambiguity during code reviews and making history easier to navigate.

For teams working at scale, this consistency is not just a convenience; it's a necessity for long-term maintainability. Custom Instructions allow you to codify your team standards directly into the tooling, ensuring that message quality is enforced automatically rather than relying on manual discipline.

---

## Stop Issues Before Committing to Git

Visual Studio now includes a feature that helps you catch potential problems before your code ever leaves your machine. This is called Local Code Review, and it integrates directly with GitHub Copilot. Instead of waiting until a pull request review identifies gaps, or worse, letting issues slip all the way into the remote repo, you can surface suggestions at commit time.



**Figure 5-65** Running Local Code Review



This process fits seamlessly into the Git workflow. Once you have a change ready to commit, you can click on the “sparkle comment” icon (for lack of a better word) shown in Figure [5-65](#). This triggers Copilot to review your changes and provide feedback about the changes you’ve made. What makes this especially valuable is that it happens within Visual Studio, allowing you to address the issues immediately before committing to the remote repo.

---

## Summary

In this chapter, we looked at the full Git and GitHub workflow inside Visual Studio 2022. We started with account creation and multi-account support. Then we had a look at cloning repositories, committing changes, branching, and handling pull requests. These are the tools that developers use every day. We also looked at some of the advanced features such as working with stashes, managing multiple repositories in the same solution, comparing branches, checking out specific commits, and staging changes at a line level.

Finally, we saw how Visual Studio integrates tightly with GitHub Copilot to generate commit messages, enforce team standards, and even perform local code reviews to catch issues before they reach your remote repository.

Together, these features make source control in Visual Studio more than just a way of keeping code safe. It’s a productivity engine that enforces best practices, supports collaborative workflows, and gives teams confidence that the code they are shipping is both maintainable and secure.

# Index

## A

AddCategoryEntity [163](#)

AddDatabase [159](#)

Added NuGet Packages [157](#)

AdditionalData [321](#)

AddServiceDefaults() extension method [125](#)

AddServiceDefaults References [126](#)

AddSqlServer [158](#)

AI assistance [292](#), [293](#)

AppHost Project

    AddProject<...>("apiservice") [124](#)

    AddProject<...>("webfrontend") [124](#)

    AddRedis("cache") [124](#)

    Builder.Build().Run() [124](#)

    DistributedApplication [124](#)

    Program.cs file [123](#)

AppHost, SQL Server

    AddDatabase [159](#)

    add migrations [162](#)

    AddSqlServer [158](#)

    connect application via EF Core [161](#), [162](#)

    First Run [159](#), [160](#)

    migrations folder [164](#)

    Program.cs File [158](#)

    run Aspire application

        Connect to Server Dialog [167](#)

        eShopLite database [168](#)

        Product Store [169](#)

        run containers, console [165](#)

        run Docker containers [165](#)

        SQL Server Password [166](#)

    WaitFor [159](#)

    WithLifetime [158](#), [159](#)

    WithReference [159](#)



## Application projects

- Aspire dashboard [128](#)

- HelloAspire.ApiService [127](#)

- HelloAspire.Web [126](#)

- .NET Aspire Starter application [127](#)

- Returned Weatherforecast JSON [129](#)

- run Blazor Application [129](#)

- Weatherforecast API [130](#)

## Arrange-Act-Assert [314](#)

## Ask Copilot [75](#)

## Aspire dashboard [128](#), [150](#), [160](#)

## Aspire.Hosting.SqlServer Package [156](#)

## Aspire.Microsoft.Data.SqlClient [156](#)

## Aspire.Microsoft.EntityFrameworkCore.SqlServer package [156](#)

## Aspire Solution [122](#)

- HelloAspire.AppHost [122](#)

- HelloAspire.ServiceDefaults [123](#)

- HelloAspire.Web and HelloAspire.ApiService [123](#)

## ASP.NET Core Web App (Model-View-Controller) [116](#)

## ASP.NET Core Web App (Razor Pages) [116](#)

## Assert.AreEqual [315](#)

## Attaching to a running process

- connection type [299](#)

- filter process text box [300](#)

- local machine/remote computer [298](#)

- reattach to process [301](#)

- remote process [300](#), [301](#)

- w3wp.exe process [300](#)

## Azure-specific packages [153](#)

## B

## Blazor server project [126](#)

## Blazor Web App [115](#)

## Bookmarks

- collection [188](#)

- delete [188](#)

- rename [187](#)

- Visual Studio [186](#)
  - window [186](#), [187](#)
- Branch creation
  - branches view [377](#)
  - code changes [379](#)
  - Commit All and Push [379](#), [380](#)
  - GitHub [380](#), [381](#)
  - isolation [376](#)
  - local branch [379](#)
  - master branch [377](#)
  - naming [378](#)
  - New Branch button [377](#), [378](#)
- Breakpoint actions [265-267](#)
- Breakpoints
  - action expression [266](#)
  - actions [265-267](#)
  - add and remove [269](#)
  - conditional [263-267](#), [287](#)
  - debug toolbar when breakpoint hit [255](#)
  - dependent [268](#), [269](#)
  - drag [269](#)
  - Force Run to Cursor [259](#)
  - import/export [273](#)
  - labels [271](#), [272](#)
  - multiple [269](#), [270](#)
  - Run to Click [257](#), [258](#)
  - Run to Cursor [258](#), [259](#)
  - setting [254](#), [255](#)
  - start debugging button [255](#)
  - step buttons [256](#)
  - step into specific [256](#), [257](#)
  - temporary [268](#)
  - visualizers [260-263](#)
  - window [270](#), [271](#)
  - window with labels set [272](#)
- builder.AddServiceDefaults() [126](#)
- the builder.AddServiceDefaults() [127](#)

Built-in color themes [234](#), [235](#)

Built-in themes [238](#)

## C

Check out commit [403–408](#)

C# Interactive

- code results [207](#)

- install C# REPL [208](#)

- IntelliSense [208](#)

- invoke C# REPL [208](#)

- REPL [207](#)

- run LINQ query [207](#)

Class Library

- .NET Framework [117](#), [118](#)

- .NET Standard [117](#)

- projects [117](#)

- Razor [118](#)

- Windows Forms [118](#)

- WPF [118](#)

Client-side web applications, # c, [115](#)

Cloud-agnostic packages [153](#)

Code archaeology [406](#)

Code coverage

- analysis [350](#)

- in blocks [352](#)

- vs. bugs [349](#)

- change fonts and colors [351](#)

- expressed in lines [352](#)

- results [350](#), [351](#)

- toggle code coverage coloring [351](#)

Code metrics results, Visual Studio Windows [209](#), [215](#)

- Calculate Code Metrics [209](#), [210](#)

Cyclomatic Complexity

- calculation [213](#)

- decision-heavy methods [214](#)

- refactor and quality gates [214](#)

- testability/maintainability, code [213](#)

depth of inheritance

analyzer rule CA1506 [222](#)

calculation [215](#), [216](#), [219](#)

Car class [218](#)

Class Coupling [219](#)

code analysis rule CA1501 [217](#)

the “Gang of Four,” state [217](#)

High Class Coupling [221](#)

high Class Coupling count [219](#), [221](#)

inheritance hierarchy [215](#)

object composition [217](#)

red flag [216](#)

refactor and quality gates [218](#), [219](#), [222](#)

lines of code

calculation [222–225](#)

refactor and quality gates [226](#)

maintainability index

calculation [211](#), [212](#)

Go to Source Code [211](#)

low score [212](#)

refactor and quality gates [212](#)

vibe coding [210](#)

Code shortcuts

added indicator [189](#)

Task List [189](#)

TODO comment, Task List [189](#), [190](#)

Visual Studio [188](#)

Code snippets

basic logging class [182](#)

basic snippet template [181](#)

Ctrl+K, Ctrl+X [180](#), [181](#)

example, inserted code snippet [184](#)

inserted code snippet [184](#)

insert try-catch [179](#)

insert try code [179](#)

Manager [179](#), [180](#)

new code snippet [184](#), [185](#)

- Visual Studio [178](#), [185](#)
- Collaborative workflows [415](#)
- Comparing branches
  - diff configuration options [403](#)
  - master branch [401](#), [402](#)
  - pull request [400](#)
  - The Branch Diff [402](#)
  - view branches [401](#)
- Conditional breakpoints [287](#)
  - conditional expression [263](#), [264](#)
  - context menu [263](#)
  - Filter [264](#), [265](#)
  - Hit Count [264](#)
- Console Application [111](#)
- Console page [137](#)
- context.Database.MigrateAsync() [164](#)
- Copilot [46](#), [67](#), [97](#), [100](#), [102](#), [131](#), [140](#), [141](#)
  - unit tests
    - chat window [340](#), [341](#)
    - documenting [341](#)
    - generate [339](#)
    - GetRiskCategory method [339](#)
    - inline chat [339](#), [340](#)
    - InsuranceCalculator [338](#)
    - InsuranceCalculatorTests [339](#)
    - risk category [338](#)
    - Test Explorer [340](#)
- Copilot Agent [79](#), [80](#), [93](#), [101](#)
  - in action [81](#)
- Copilot Agent Diff [82](#)
- Copilot Chat [92](#)
- Copilot response [77](#)
- Copy Git permalink [408](#), [409](#)
- CPU usage [291-293](#)
- CreateDbIfNotExistsAsync method [163](#)
- Creating and running unit tests
  - Ask Copilot Option [317](#)

- Failed tests [317](#)
- playlist [318](#), [319](#)
- Test Explorer Window [315](#), [316](#)
- test frameworks [311-313](#)
- writing [313-315](#)
- writing acts [311](#)
- Custom instructions [93](#), [94](#)
- Custom token [191](#)
  - Task List [191](#)
  - Visual Studio [192](#)
- Custom Visualizers [263](#)
- Cyclomatic Complexity [212](#)
  - calculation [213](#)
  - decision-heavy methods [214](#)
  - refactor and quality gates [214](#)

## D

- Database engines [200](#)
- Data-driven tests [321](#)
- DataTips [274](#), [275](#)
- DebuggerDisplay Attribute [276-278](#)
- Debugging code
  - application [251](#)
  - attaching to a running process [298-301](#)
  - breakpoints [254-273](#)
  - DataTips [274](#), [275](#)
  - DebuggerDisplay Attribute [276-278](#)
  - Diagnostic Tools [288-296](#)
  - error copying improvements [281](#), [282](#)
  - evaluate functions [278-281](#)
  - GitHub Copilot [282-287](#)
  - Immediate Window [296-298](#)
  - inline return values [273](#), [274](#)
  - remote debugging [301-309](#)
  - toolbar location [251-254](#)
  - Watch window [275](#), [276](#)
- Dependent breakpoints [268](#), [269](#)



## Diagnostic Tools

- CPU usage [291-293](#)

- debug session [288](#)

- enable [288](#)

- execution timeline [290](#)

- memory usage [294-296](#)

- Select Tools [289](#), [290](#)

- window [288](#), [289](#)

## Dirty indicator [229](#)

- default [230](#)

- dot [230](#)

## Docker [165](#)

### Docker desktop [160](#)

### Document unit tests

#### MCP server

- API key [343](#), [344](#)

- config JSON [345](#), [346](#)

- configuration [344](#), [345](#), [347](#)

- Docker MCP Toolkit [342](#)

- documentation [349](#)

- host and run [342](#)

- installation [343](#)

- .mcp.json file [346](#)

- Obsidian knowledge base [342](#)

- prompting Copilot [347](#), [348](#)

- REST API Plugin [342](#), [343](#)

- tool selection [347](#)

## Drag breakpoints [269](#)

## Dynamic data driven tests [321-324](#)

## E

### EnsureCreatedAsync() method [162](#)

### eShopLite [143](#), [154](#)

### eShopLite.AppHost project [156](#)

### eShopLite Database [168](#)

### eShopLite solution

- Added Aspire Projects [146](#)

- Add .NET Aspire Orchestrator Support [146](#)
- add .NET Aspire to Products [147](#)
- AppHost Program.cs File [147](#)
- Aspire Dashboard with Copilot [152](#)
- Aspire NuGet Package Version [151](#)
- benefit [153](#)
- configure Multiple Startup Projects [144](#)
- DataEntities [144](#)
- modified AppHost Program.cs file [148](#)
- Products [144](#)
- products page [145](#)
- run Store App [150](#)
- Store [144](#)
- update AppHost NuGet package [152](#)
- Evaluate functions
  - Counter class [278](#), [279](#)
  - Format specifiers [280](#), [281](#)
  - side effects [280](#)
  - suppress [279](#)
- Exception handling
  - GitHub Copilot [284–286](#)

## **F**

- Filter condition [264](#), [265](#)
- Filtered solution [59](#)
- Format specifiers
  - ac [280](#)
  - C# [280](#)
  - d [280](#)
  - dynamic [280](#)
  - h [280](#)
  - hidden [281](#)
  - Immediate and Command window [280](#)
  - nq [280](#)
  - nse [280](#)
  - raw [281](#)
  - results [281](#)

Free subscription [357](#)

## G

GetCustomDynamicDataDisplayName [322–324](#)

Ghost Text Code Suggestion [73](#)

GitHub [83](#)

- account creation [356](#), [357](#)

- Microsoft stock [355](#)

- Visual Studio 2022 [355](#)

GitHub account [66](#)

GitHub account management [358–360](#)

GitHub Copilot [74](#), [79](#), [88](#), [360](#), [415](#)

- app [97](#)

- commit message custom instructions [411–413](#)

- commit messages [410](#)

- and conditional breakpoints [287](#)

- debugger-aware AI [282](#)

- default output [410](#)

- exception handling [284–286](#)

- long-term maintainability [413](#)

- model selection [71](#)

- Settings [70](#)

- specific standard [411](#)

- structured commit message format [412](#)

- variable analysis [282–284](#)

GitHub Copilot Integration [141](#)

GitHub Copilot Settings [73](#)

GitHub MCP Tools [86](#)

GitHub repository cloning

- location [374](#)

- Team Explorer [375](#), [376](#)

- Visual Studio [373](#)

GitHub repository creation

- account connection [363](#), [364](#)

- Account drop-down [362](#)

- commit changes locally [367](#), [368](#)

- commit options [366](#), [367](#)

- experimental changes [372](#)
- Git Changes [364](#), [365](#)
- Output Window [362](#), [363](#)
- push changes [368](#), [369](#)
- pushed changes [369](#), [370](#)
- push remote repo [369](#)
- repository window [361](#), [362](#)
- Solution Explorer [364](#)
- source control [363](#)
- UI or code changes [365](#), [366](#)
- undo changes [371](#), [372](#)
- Visual Studio [360](#)
- WeatherApiClient.cs file [370](#), [371](#)
- GitHub workflow [415](#)
- GPT-4.1 code completion model [71](#)
- GrandParent [215](#)

## H

- HelloAspire.ApiService [127](#)
- HelloAspire.Web [126](#)

## I

- IEnumerable Visualizer [260](#), [261](#)
- Immediate Window [296–298](#)
- Import/Export Breakpoints [273](#)
- Inline return values [273](#), [274](#)
- InsuranceCalculatorTests [314](#)
- IntelliCode [60](#)
- InvalidOperationException [89](#)

## J

- JSON Visualizer [261](#), [262](#)

## K

- Keep Staged [393](#)

## L

- Lines of Executable Code [222](#), [224](#), [225](#)
- Lines of Source Code [222](#), [225](#)

Line-staging [406](#)

Live unit testing (LUT)

- artifacts [336](#)

- benefits [332](#)

- breaking change [337](#)

- configure [332](#), [333](#), [335](#)

- Configure Lutignore file* [336](#), [337](#)

- frameworks [332](#)

- multiple test processes [336](#)

- playlist editor [334](#)

- in real time [332](#)

- Repository root [335](#)

- results updated [337](#), [338](#)

- selection [334](#)

- test options [336](#)

- window [333](#), [334](#)

- workspace folder [336](#)

Local Code Review [414](#)

Low Maintainability Index [212](#)

## M

Maintainability index [212](#)

- calculation [211](#)

- refactoring and quality gates [212](#)

Managed memory analysis [295](#), [296](#)

Master/main [376](#)

MCP

- See* Model Context Protocol (MCP)

Memory usage [294–296](#)

Metrics page [139](#), [140](#)

Microsoft Visual Studio 2022 [248](#)

MigrateAsync() method [162](#)

Model Context Protocol (MCP) [83](#)

- configurations [84](#)

- server tools [88](#)

Model-View-Controller approach [116](#)

MSTest [312](#), [315](#), [319](#), [322](#), [324](#), [326](#), [327](#)

Multiple GitHub accounts [358](#), [360](#)

Multi-repo support

branches selection [400](#)

changes [399](#)

changes committed locally [399](#)

Git repositories [397](#)

multiple repositories [400](#)

project selection [397](#)

Visual Studio [398](#)

## N

.NET [112](#), [117](#)

.NET 6.0 [112](#)

.NET 7 [112](#)

.NET 8 [112](#)

.NET 9 [112](#)

.NET 10 [112](#)

.NET Aspire

Aspire dashboard [150](#)

benefits [153](#)

eShopLite solution

*See also* eShopLite solution

packages [143](#), [144](#), [153](#), [153](#), [154](#)

Store appsettings.json File [149](#)

WithExternalHttpEndpoints Extension Method [148](#)

.NET Aspire application [118](#), [119](#)

.NET Aspire Community Toolkit [154](#)

.NET Aspire dashboard

Aspire login page [132](#)

Console Window [132](#)

Copilot [131](#)

overview [131-133](#)

resources page

Actions [133](#)

Actions Menu [134](#)

Graph Menu [135](#)

Graph view [134-136](#)



- Name [133](#)
- Source [133](#)
- Start time [133](#)
- State [133](#)
- URLs [133](#)
- standalone dashboard mode [141](#), [142](#)
- view logs and telemetry
  - console page [137](#)
  - Copilot [140](#), [141](#)
  - metrics page [139](#), [140](#)
  - structured logs page (structured) [137](#), [138](#)
  - traces page [138](#), [139](#)
- .NET Aspire Orchestration [147](#)
- .NET Aspire Orchestrator project [147](#)
- .NET Aspire Project
  - Aspire Solution [122](#)
  - Container Runtime [119](#)
  - IDE or Editor [119](#)
  - .NET Aspire Workload [120](#)
  - .NET SDK [119](#)
  - solution structure
    - AppHost Project [123](#), [124](#)
    - application projects [126–130](#)
    - ServiceDefaults Project [124–126](#)
- Visual Studio 2022 [120–123](#)
- .NET Aspire Starter application [127](#)
- .NET Core [112](#)
- .NET Framework [111](#), [118](#)
- .NET SDK [119](#)
- .NET Standard [117](#)
- .NET Upgrade Tool [99](#)
- npm CLI [177](#)
- npm in .NET projects
  - ASP.NET Core web apps [170](#)
  - built-in user interface [177](#)
  - CLI [175](#)
  - client-side JavaScript packages management [174](#)

- install gulp-scss-lint [176](#)
- install new npm packages [174](#)
- Node.js [171](#)
- NuGet [171](#)
- package.json file [172](#), [173](#)
- run npm Audit [178](#)
- search, npm packages [174](#), [175](#)
- security considerations [177](#), [178](#)
- updated package.json file [176](#)
- Visual Studio [170](#)
- Visual Studio 2022 [171](#)
- NuGet packages [153](#), [154](#), [171](#)
  - developers [169](#)
  - private hosting [170](#)
  - ZIP file, .nupkg extension [169](#)

## O

- Object composition [217](#)
- Once per class [319](#)

## P, Q

- package.json file [172](#), [173](#)
- Parent inherits [215](#)
- Personalize Visual Studio
  - adjust line spacing [227](#), [228](#)
  - document management customizations
    - colorize document tabs [231](#), [232](#)
    - dirty indicator [229](#)
    - document close button [229](#)
    - invisible tabs in italics [230](#), [231](#)
    - tab placement [232-234](#)
  - Visual Studio themes
    - See* Visual Studio themes
- ProductDataContext class [161](#)
- Products project [156](#)
- Program.cs [126](#)
- Project templates
  - Class Library

- .NET Framework [117](#), [118](#)
- .NET Standard [117](#)
- projects [117](#)
- Razor [118](#)
- Windows Forms [118](#)
- WPF [118](#)
- Console Application [111](#)
- .NET Aspire [118](#), [119](#)
- Service
  - Windows Service .NET Framework [112](#), [113](#)
  - Worker Service [112-114](#)
- Visual Studio 2022 [110](#)
- Web Applications
  - ASP.NET Core Web App (Model-View-Controller) [116](#)
  - ASP.NET Core Web App (Razor Pages) [116](#)
  - Blazor Web App [115](#)
  - types [115](#)
- Windows Forms application [111](#), [112](#)
- Pull Requests (PRs) creation
  - code review [383](#)
  - feedback [384](#)
  - New Pull Request window [382](#), [383](#)
  - preview [383](#)
  - Visual Studio notification [381](#), [382](#)
- Pull Requests (PRs) handling
  - code review [387-389](#)
  - comments [390](#)
  - feedback [388](#)
  - GitHub [384](#), [385](#)
  - GitHub integration [391](#)
  - integration Visual Studio [384](#)
  - reviews [386](#), [387](#)
  - team's workflow [391](#)
  - viewing [386](#)

## **R**

- Razor Class Library [118](#)

Razor pages [116](#)

Read-Eval-Print Loop (REPL) [207](#)

Remote debugging

- configuration [304](#), [305](#)

- download and install remote tools [301](#), [303](#), [304](#)

- port assignments [301](#)

- running remote tools [304–306](#)

- start

  - application [307](#), [308](#)

  - breakpoint [306](#)

  - launch profiles [307](#)

  - project properties page [306](#)

  - session connected [308](#)

  - Visual Studio 2022 [306](#)

- system requirements [302](#), [303](#)

REPL

*See* Read-Eval-Print Loop (REPL)

## S

SeniorJunior.playlist [318](#)

Server Explorer [193](#)

- add connection [194](#), [195](#)

- choose data source [193](#), [194](#)

- create table statement [197](#)

- database [195](#), [196](#)

- Data Tools Operations [199](#), [200](#)

- functionality [205](#)

- modified create table statement [197](#)

- preview database updates [198](#), [199](#)

- run SQL queries [201](#)

  - functionality [200](#)

  - INSERT statements [202](#)

  - new user history entry insert [203](#)

  - select statement [202](#)

  - SQL select statement [201](#)

  - SQL update statement [203](#)

  - TestUser entry update [204](#)

- table designer [196](#), [197](#)
- table design update [198](#)
- ServiceDefaults Project [124–126](#)
- Shipment Locator application [9](#)
- ShipmentLocator project [13](#), [14](#)
- ShipMethodLogic projects [54](#)
- Sign-in process [359](#)
- Slash commands [78](#)
- .SLNX file [240](#)
  - alignment with project files and MSBuild [244](#)
  - compare SLN to SLNX [241](#), [242](#)
  - content simplicity [240](#)
  - definition [240](#)
  - format [240](#)
  - manual edits and tooling [243](#)
  - merge conflicts [243](#)
  - readability [240](#)
  - reducing file bloat and duplication [243](#)
  - source control friendliness [243](#)
  - Visual Studio 2022 [249](#)
- .SLNX solution files
  - .NET CLI [247](#), [248](#)
  - Preview Features [244](#), [245](#)
  - Visual Studio [245–247](#)
- Solution Explorer [364](#)
- Solution Filter [53](#), [57](#), [59](#)
- SQL Server
  - Added NuGet Packages [157](#)
  - AppHost
    - See AppHost, SQL Server
  - install Aspire.Hosting.SqlServer [155](#), [156](#)
  - local SQLite database [155](#)
- Standalone dashboard mode [141](#), [142](#)
- Stash All [393](#)
- Stashes
  - Applying/Popping [396](#)
  - Git Changes window [392](#)

- multiple branches [392](#)
- new branch [395](#)
- options [392-394](#)
- production bug fix [397](#)

Structured logs page (structured) [137](#), [138](#)

## T

Tab placement [232-234](#)

Task List [189](#)

- custom token [191](#), [192](#)
- add NOTE token comment [192](#)

Team and Enterprise subscriptions [357](#)

Team subscription [357](#)

Temporary breakpoints [268](#)

TestDynamicDataCalucaltions [322](#)

Test frameworks [311-313](#)

TestTimeoutOfMethod [324](#)

Theme Selector [235](#)

Toolbox Context Menu [21](#)

Tracepoint [266](#)

Traces page [138](#), [139](#)

## U

Unit testing

- attributes
  - Ignore [330](#), [331](#)
  - data-driven tests [321](#)
  - dynamic data driven tests [321-324](#)
  - metadata [327-330](#)
  - Retry [326](#), [327](#)
  - setup and teardown methods [319](#), [320](#)
  - Testing Timeouts [324-326](#)
- code coverage [349-352](#)
- Copilot [338-341](#)
- creating and running [311-331](#)
- document [342-349](#)
- LUT [332-338](#)

Up() and Down() methods [164](#)



## V

Variable analysis [282-284](#)

Version number [173](#)

Visualizers

- custom [263](#)

- debug session [260](#)

- IEnumerable [260](#), [261](#)

- JSON [261](#), [262](#)

Visual Studio [22-24](#), [64](#), [170](#)

- GitHub [355](#)

- tokens [190](#)

Visual Studio 2022

- \$shared environment [50](#)

Visual Studio 2022 [6](#), [9](#), [25](#)

- ASP.NET Core API project [47](#)

- C# projects and VB.NET projects [10](#)

- development environment [5](#)

- dev environment [51](#)

- evolution [1](#)

- features and productivity

  - cleanup on save [42](#)

  - clipboard history [37](#)

  - customize file encoding [44](#)

  - File Explorer [36](#)

  - Go To commands [38](#)

  - horizontal scrollbar [45](#)

  - JSON [42](#)

  - keyboard shortcuts [36](#)

  - matching selections [40](#)

  - namespaces in sync [42](#)

  - pasting copied code [43](#)

  - toolbar [34](#)

  - Track Active Item [34](#)

  - wrapped lines [44](#)

- Find in Files feature [28](#)

- GitHub Copilot [66](#), [70](#)

- Go To commands [30](#)

- Go To Definition [30](#)
- hardware [4](#)
- installer [2](#)
- languages [5](#)
- Live Share Chat [64](#)
- navigation bar [26](#)
- .NET 9 projects [6](#)
- .NET Aspire Project [120–123](#)
- .NET languages [11](#)
- operating systems [4](#)
- Peek Definition [31](#), [32](#)
- project types [66](#)
- references [26](#)
- references highlighting [28](#)
- solution file [56](#)
- Start screen [106](#)
- subword navigation [33](#)
- system requirements [3](#)
- toolbar [25](#)
- toolbox [21](#)
- tools and features [1](#)
- tool window [27](#)
- top-level assignments [49](#)
- variables [47](#)
- versions [3](#)
- Windows machines [2](#)
- workloads [6](#)
- Visual Studio Code [63](#)
- Visual Studio Community 2022 [3](#)
- Visual Studio Enterprise 2022 [3](#)
- Visual Studio files [17](#)
- Visual Studio IDE [9](#)
- Visual Studio Installer [7](#), [68](#)
- Visual Studio IntelliCode [59](#)
- Visual Studio Live Share [61](#)
- Visual Studio Professional 2022 [3](#)
- Visual Studio project types

- create a new project [109](#)
- filter project templates [109](#), [110](#)
- Get Started section [108](#)
- New Project button [108](#)
- recent projects [106](#), [107](#)
- Visual Studio Search [52](#)
- Visual Studio Theme Pack [239](#)
- Visual Studio themes
  - built-in color themes [234](#), [235](#)
  - customization colors and fonts [236](#), [237](#)
  - customization windows layouts [238](#)
  - extensions and additional themes [238](#), [239](#)
- Visual Studio Windows
  - C# Interactive
    - code results [207](#)
    - install C# REPL [208](#)
    - IntelliSense [208](#)
    - invoke C# REPL [208](#)
    - REPL [207](#)
    - run LINQ query [207](#)
  - code metrics results
    - See* Code metrics results, Visual Studio Windows
  - other windows [206](#)
  - send feedback [226](#), [227](#)
- vs-solutionpersistence [244](#)

## W

- WaitFor [159](#)
- Watch window [275](#), [276](#)
- Weatherforecast API [130](#)
- Web API template [116](#)
- Web Applications
  - ASP.NET Core Web App (Model-View-Controller) [116](#)
  - ASP.NET Core Web App (Razor Pages) [116](#)
  - Blazor Web App [115](#)
  - types [115](#)
- Web projects [127](#)

Windows Forms application [111](#), [112](#)  
Windows Forms Class Library [118](#)  
Windows Presentation Foundation (WPF) [118](#)  
Windows Service .NET Framework [112](#), [113](#)  
WinForms application [11](#), [20](#), [22](#)  
WinForms Toolbox [20](#)  
WithExternalHttpEndpoints() extension method [148](#)  
WithLifetime(ContainerLifetime.Persistent) [158](#), [159](#)  
WithReference [159](#)  
Worker Service [112–114](#)  
Workloads [6](#), [8](#)  
WPF  
    *See* Windows Presentation Foundation (WPF)

## **X, Y, Z**

XML-based solution file format [240](#)