



1ST EDITION

# Mastering Visual Studio 2022

Develop apps like a pro with advanced Visual Studio techniques using C# and .NET



**ROMAIN OTTONELLI DABADIE**

Foreword by Dave Callan, Microsoft MVP

# Mastering **Visual Studio 2022**

Develop apps like a pro with advanced Visual Studio techniques using C# and .NET

**Romain Ottonelli Dabadie**



# Mastering Visual Studio 2022

Copyright © 2024 Packt Publishing

*All rights reserved.* No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

**Group Product Manager:** Kaustubh Manglurkar

**Publishing Product Manager:** Bhavya Rao

**Book Project Manager:** Arul Viveaun S

**Senior Editor:** Rakhi Patel

**Technical Editor:** Simran Ali

**Copy Editor:** Safis Editing

**Indexer:** Manju Arsan

**Production Designer:** Joshua Misquitta

**DevRel Marketing Coordinator:** Nivedita Pandey

First published: December 2024

Production reference: 1291024

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK.

ISBN 978-1-83588-468-3

[www.packtpub.com](http://www.packtpub.com)

*To my wife, Cassandre, for being on my side for so many years and being a strong source of support during this journey. To my daughter and her smile, which brings the sun to each day of my life.*

*- Romain Ottonelli Dabadie*

# Foreword

As someone who frequently shares tips and insights about Visual Studio, I'm continually surprised by how many powerful features remain hidden from developers. From time-saving shortcuts to advanced tools, even experienced developers who've used Visual Studio for years often miss out on its full range of capabilities. For junior developers or those new to the tool, the sheer volume of features can feel overwhelming, leaving many unsure of where to start.

Visual Studio has also improved dramatically over the past few years, but staying updated with these advancements can feel like a full-time job. There's no single place to learn everything it has to offer, so most developers pick up their knowledge in an ad-hoc, piecemeal fashion—from blogs, YouTube videos, social media, and other sources. This book is an invaluable resource, gathering scattered knowledge, tips, and hidden gems into a cohesive guide.

What makes this book stand out is its comprehensive, holistic approach to learning Visual Studio. It doesn't just skim the surface but delves into both foundational tools and advanced capabilities that many developers might never discover on their own. The book covers essential topics such as mastering the debugger, code analysis, refactoring, and unit testing, as well as newer features like using .http files and Dev Tunnels for API work. It even includes in-depth content on starting with .NET Aspire and MAUI development in Visual Studio.

Beyond feature coverage, this book provides practical, actionable advice that developers can use immediately to enhance productivity. It includes real-world guidance on setting up effective Git workflows, integrating CI/CD pipelines with GitHub Actions and Azure DevOps, and even creating custom Visual Studio extensions and project templates to tailor the development environment to team-specific needs.

I'm thrilled to write this foreword. It's evident that Romain has poured his expertise into creating a resource that will help developers unlock Visual Studio's full potential. I'm excited to recommend this book to you and your team.

*Dave Callan, Microsoft MVP*

# Contributors

## About the author

**Romain Ottonelli Dabadie**, a seasoned Technical Expert in .NET, shapes the IT landscape in France and Canada. With over a decade of experience, he's a trusted professional, excelling in crafting robust solutions using Visual Studio. Romain's journey includes diverse roles, showcasing a commitment to excellence. Beyond his technical prowess, Romain actively engages on social media, sharing insights with the tech community. He stays abreast of Microsoft's latest developments, demonstrating a keen interest in their news and advancements. In the past decade, his proficiency in Visual Studio has driven solutions surpassing project expectations.

*I would like to thank the people who have been close to me and supported me, especially my wife and my daughter.*

## About the reviewers

**Nadir Riyani** holds a Master's in computer application and brings 15 years of experience in the IT industry to his role as an engineering manager. With deep expertise in Microsoft technologies, Splunk, DevOps automation, database systems, and cloud technologies, Nadir is a seasoned professional known for his technical acumen and leadership skills. He has published over 200 articles in public forums, sharing his knowledge and insights with the broader tech community. Nadir's extensive experience and contributions make him a respected figure in the IT world.

**Amr Saafan** is a seasoned CTO and software consultant with over 20 years of experience in software development. He specializes in designing and developing applications for multiple industries such as telecommunication, healthcare, finance, and so on. As the founder of Nile Bits, he has successfully led projects for more than 200 clients across 15+ countries. Amr's expertise includes system architecture, cloud services, software development, QA, and Agile coaching. He oversees technical teams, defines technology stacks, guides startups in product development, and is deeply involved in business development, software design, and talent acquisition.

# Table of Contents

## Preface

xiii

## Part 1: Mastering Core Development Skills

1

Unit Testing and Test-Driven Development		3	
Technical requirements	4	Creating a test with IntelliTest	15
Introducing unit testing and TDD	4	IntelliTest execution and test generation	17
The F.I.R.S.T principles of unit testing	4	Saving, executing, and reviewing	
TDD – unifying principles for		unit tests	18
software quality	5	Practicing TDD with a	
AAA pattern – a structured approach		real-world example	19
to testing	8	Automating your tests with Live	
Setting up unit testing in Visual		Unit Test	24
Studio 2022	8	Configuring Live Unit Testing	25
Creating a unit test project	9	Launching Live Unit Testing	27
Overview of the Test Explorer view	11	Summary	30

2

Advanced Debugging Strategies		31	
Technical requirements	32	Fixing bugs on the fly	40
Mastering Visual Studio Debugger	32	Advanced breakpoints and data	
Entering debug mode	32	inspections	42
Advanced debug navigation	34	Understanding the types of breakpoints	42
Understanding debugger tools	36	Organizing our breakpoints	48

Inspecting the data	49	External Sources	53
Elevating debugging with auto-decompilation and External Sources	52	Concurrency debugging	54
Auto-decompilation	52	Remote debugging	58
		Summary	60

## 3

### **Advanced Code Analysis and Refactoring** 61

---

Technical requirements	62	Predicting code with whole-line autocompletion	74
Understanding code analysis in Visual Studio	62	Accessing GitHub documentation	75
How Roslyn works	63	<b>Code metrics, maintainability, and security</b>	77
How does Visual Studio 2022 use Roslyn?	64	Understanding the metrics	77
	65	Using code metrics in Visual Studio 2022	77
<b>Utilizing static code analysis for quality assurance and security</b>		<b>Refactoring case studies</b>	79
Understanding how to use code analysis in Visual Studio	66	Handling common bad practice	80
Adjusting the level of severity	67	Generating an interface	82
Generating a .editorconfig file	70	File-scoping our namespace	83
<b>Leveraging IntelliCode for code refactoring</b>	73	<b>Summary</b>	85

## 4

### **Performance Optimization and Profiling** 87

---

Technical requirements	88	Analyzing database performance	95
Introduction to performance optimization		Instrumenting our .NET applications	95
<b>Utilizing Visual Studio profiling tools</b>	88	<b>Analyzing CPU Usage</b>	96
Analyzing .NET asynchronous events	90	<b>Memory profiling and optimization</b>	102
Monitoring with .NET Counters	91	Using the Memory Usage tools	103
Tracking .NET Object Allocation	93	Exploring Memory Usage while debugging	104
Viewing the event	94	<b>Optimizing database interactions</b>	107
Analyzing File I/O	94	<b>Summary</b>	111

## Part 2: Advancing Development Horizons

**5**

### Multi-Platform App UI Development **115**

Technical requirements	115	The Live Visual Tree	126
An introduction to MAUI	116	Debugging on devices	128
The evolution of cross-platform development	116	Enabling Developer Mode on our device	128
What is MAUI?	116	Networking device	129
The key features of MAUI	116	Launching the debugging session	130
The architecture of MAUI	117	Migrate from Xamarin	131
Exploring the tools for MAUI	118	Understanding the key differences	131
Creating a simple MAUI app	118	Using the .NET Upgrade Assistant	131
XAML Live Preview	123	Summary	133

**6**

### Advanced Web Development Tools **135**

Technical requirements	135	Configuring Dev Tunnel	142
Real-time web previews powered by Web Live Preview	136	Node.js integration with Visual Studio	144
Using API Explorer and Dev Tunnel with Visual Studio	138	Exploring JavaScript project templates	144
Generating .http files with Endpoints Explorer	138	Managing npm packages	146
		Debugging JavaScript applications	148
		Summary	150

**7**

### Machine Learning Integration **151**

Technical requirements	152	Creating an ML model with ML.NET and the Model Builder UI	153
Introduction to ML	152		

Deploying a model in an ASP.NET Core web API	158	Deploying a model in Azure Functions	161
		Summary	165

## 8

<b>Advanced Cloud Integration and Services</b>	<b>167</b>		
Technical requirements	168	Exploring Google Cloud Tools for Visual Studio 2022	177
Exploring .NET Aspire	168	Exploring the AWS Toolkit	182
Exploring Azure Functions development in Visual Studio	173	Summary	186

## Part 3: Streamlining Collaborative Development with DevOps Practices

## 9

<b>Handling Advanced Git Workflows</b>	<b>189</b>		
Technical requirements	189	Resolving conflicts through Visual Studio	197
Managing a repository through Visual Studio	190	Exploring interactive staging in Visual Studio	200
Exploring the Manage Branches window	190	Summary	205
Looking at Checkout(-- detach)	194		
Handling multiple repositories	195		

## 10

<b>Continuous Integration with GitHub Actions</b>	<b>207</b>		
Technical requirements	207	Generating GitHub Actions file with Visual Studio	212
Understanding GitHub Actions for CI/CD	208	Summary	216
Configuring workflows in GitHub Actions	209		

---

**11****Collaborative Development with Azure DevOps 217**

---

Technical requirements	218	Introducing Agile development	222
Introduction to Azure DevOps	218	Managing Work Items through Visual Studio	223
Setting up team projects	219		
Implementing Agile development practices	221	Integrating Azure Pipelines for continuous integration	227
		Summary	230

**12****Visual Studio Container Tools for Docker 231**

---

Technical requirements	232	Dockerizing applications with Visual Studio	237
Introduction to Docker and Visual Studio integration	232	Deploying containerized applications	240
Setting up Docker environments in Visual Studio	233	Deploying in Container Registry	240
Creating a project with Docker support	234	Deploying as a service in Azure	243
Adding Docker support to an existing project	235	Summary	245

**Part 4: Mastering Core Development Skills****13****Writing Your Own Project Template 249**

---

Technical requirements	250	Integrating template parameters and variables	256
Understanding project template structure	250	Extending project templates with advanced features	262
Building a basic project template	252	Summary	264
Customizing project templates for different workflows	255		

**14**

<b>Writing Your Own Visual Studio Extensions</b>	<b>265</b>
Technical requirements	266
Understanding Visual Studio extension architecture	266
Building your first extension	267
Advanced extension features	272
Deploying and sharing your extensions	275
Summary	276

**15**

<b>Creating and Publishing Powerful NuGet Packages for the Community</b>	<b>277</b>
Technical requirements	278
Introduction to NuGet and package management	278
Creating your first NuGet package	280
Versioning and dependency management	281
Publishing and distribution	284
Advanced NuGet features	287
Target frameworks	287
Pre-release versions	288
Custom scripts	288
Summary	289
<b>Index</b>	<b>291</b>
<b>Other Books You May Enjoy</b>	<b>302</b>

# Preface

Visual Studio is an essential tool for professional application development, particularly when working with C# and .NET. It provides developers with a comprehensive suite of features that simplify the process of creating, testing, and deploying robust applications. This book aims to guide developers in leveraging the full potential of Visual Studio's advanced capabilities.

In this book, we will focus on four critical aspects of modern software development:

- Core development skills, such as testing, optimization, debugging, and refactoring, which are crucial for writing clean and efficient code
- Advanced development techniques, including multi-platform app development, web tools, machine learning integration, and cloud services, enabling you to push your development capabilities further
- DevOps practices, which streamline collaborative development using Visual Studio's built-in tools for advanced Git workflows, continuous integration with GitHub Actions, and Azure DevOps for effective team collaboration
- Customization and enhancement, empowering you to tailor Visual Studio to your unique workflow by creating your own templates, extensions, and powerful NuGet packages

This book draws on my years of experience in software development, as well as insights from leading professionals in the field who have mastered Visual Studio's most advanced features. As Visual Studio continues to evolve, developers will increasingly rely on its tools for efficient and scalable development. The key to success lies not just in writing code but in mastering the full development life cycle, from design to deployment, with the help of Visual Studio's powerful features.

By mastering these techniques, you'll be well-equipped to meet the challenges of modern application development, enhance your productivity, and deliver professional-grade applications with ease.

## Who this book is for

This book is designed for developers, team leads, and IT professionals who want to elevate their expertise in Visual Studio and C#/.NET application development. The following groups will benefit the most from this content:

- **Professional developers:** Developers looking to advance their knowledge of Visual Studio's advanced features and improve their skills in areas such as testing, optimization, debugging, and DevOps integration. This book will help them build more efficient, scalable, and professional-grade applications.

- **Development team leads:** Team leaders seeking to streamline workflows, implement effective DevOps practices, and enhance collaboration within their teams using Visual Studio's powerful tools. This book will help them manage multi-platform and cloud-based projects more efficiently.
- **IT and DevOps professionals:** Those responsible for managing development environments, automating deployment pipelines, or supporting development teams will gain valuable insights into advanced Git workflows, continuous integration, and containerization with Docker. This book provides practical guidance for optimizing Visual Studio for team collaboration and project success.

Whether you're focused on coding, managing development teams, or handling DevOps tasks, this book offers the tools and knowledge needed to master professional application development with Visual Studio.

## What this book covers

*Chapter 1, Unit Testing and Test-Driven Development*, introduces the importance of unit testing and TDD principles. It covers how to write effective unit tests using Visual Studio and integrate testing seamlessly into your development workflow.

*Chapter 2, Advanced Debugging Strategies*, explores debugging techniques beyond the basics. You'll learn how to utilize Visual Studio's advanced debugging tools to identify, isolate, and resolve complex issues in your code.

*Chapter 3, Advanced Code Analysis and Refactoring*, focuses on analyzing and refactoring code to improve quality and maintainability. This chapter covers tools and methods to identify code smells and enhance your code's structure.

*Chapter 4, Performance Optimization and Profiling*, teaches you how to use Visual Studio's profiling tools to analyze performance bottlenecks and optimize application speed, memory usage, and efficiency.

*Chapter 5, Multi-Platform App UI Development*, delves into building responsive, cross-platform applications. You'll learn how to develop user interfaces that run smoothly on different devices and platforms using Visual Studio.

*Chapter 6, Advanced Web Development Tools*, introduces advanced features of Visual Studio to build modern web applications. This chapter covers tools and frameworks that enable scalable, high-performance web development.

*Chapter 7, Machine Learning Integration*, explores how to incorporate machine learning models into your applications. You'll learn to leverage Visual Studio to develop intelligent applications powered by machine learning.

*Chapter 8, Advanced Cloud Integration and Services*, focuses on integrating cloud services into your applications. You'll explore advanced cloud tools and services available through Visual Studio for scalable cloud-based applications.

*Chapter 9, Handling Advanced Git Workflows*, introduces advanced Git concepts and workflows. You'll learn how to manage complex Git operations and collaborate effectively using Visual Studio's integrated Git tools.

*Chapter 10, Continuous Integration with GitHub Actions*, covers setting up continuous integration pipelines using GitHub Actions. This chapter explains how to automate testing and deployment processes directly from Visual Studio.

*Chapter 11, Collaborative Development with Azure DevOps*, explores how to manage collaborative projects using Azure DevOps. You'll learn how to integrate Azure DevOps with Visual Studio to streamline team collaboration and project management.

*Chapter 12, Visual Studio Container Tools for Docker*, teaches you how to use Visual Studio's built-in container tools to develop, test, and deploy applications with Docker. This chapter covers containerization best practices and workflows.

*Chapter 13, Writing Your Own Project Template*, covers how to create custom project templates in Visual Studio to streamline development workflows. You'll learn to tailor templates to meet your project's specific needs.

*Chapter 14, Writing Your Own Visual Studio Extensions*, explains how to develop Visual Studio extensions to enhance your development environment. This chapter provides step-by-step instructions to build and publish your own extensions.

*Chapter 15, Creating and Publishing Powerful NuGet Packages for the Community*, focuses on creating reusable, shareable NuGet packages. You'll learn how to package and publish your code to the community or enterprise to streamline development for other developers.

## To get the most out of this book

You should have a basic understanding of the C# and .NET platforms. Some sections may reference cloud subscription tiers, but a subscription is not mandatory to follow along with most of this book.

Software/hardware covered in the book	Operating system requirements
Visual Studio	Windows
.NET	
C#	
<b>Amazon Web Service (AWS)</b>	
Azure	
<b>Google Cloud Platform (GCP)</b>	

As Visual Studio evolves quickly, I recommend using the latest preview version of the IDE for the best experience and to experiment with the latest features.

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

## Download the example code files

You can download the example code files for this book from GitHub at <https://github.com/PacktPublishing/Mastering-Visual-Studio-2022>. If there's an update to the code, it will be updated in the GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

## Conventions used

There are a number of text conventions used throughout this book.

**Code in text:** Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: “Use the `nuget push` command or integrate with your CI/CD pipeline to automate the process.”

A block of code is set as follows:

```
<PropertyGroup>
    <TargetFrameworks>net46;netstandard2.0</TargetFrameworks>
</PropertyGroup>
```

Any command-line input or output is written as follows:

```
git add Car.cs
```

**Bold:** Indicates a new term, an important word, or words that you see on screen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: “Right-click on the project in **Solution Explorer**.”

Tips or important notes

Appear like this.

## Get in touch

Feedback from our readers is always welcome.

**General feedback:** If you have questions about any aspect of this book, email us at [customercare@packtpub.com](mailto:customercare@packtpub.com) and mention the book title in the subject of your message.

**Errata:** Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit [www.packtpub.com/support/errata](http://www.packtpub.com/support/errata) and fill in the form.

**Piracy:** If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at [copyright@packt.com](mailto:copyright@packt.com) with a link to the material.

**If you are interested in becoming an author:** If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit [authors.packtpub.com](http://authors.packtpub.com).

## Share your thoughts

Once you've read *Mastering Visual Studio 2022*, we'd love to hear your thoughts! Please click here to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

## Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/978-1-83588-468-3>

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

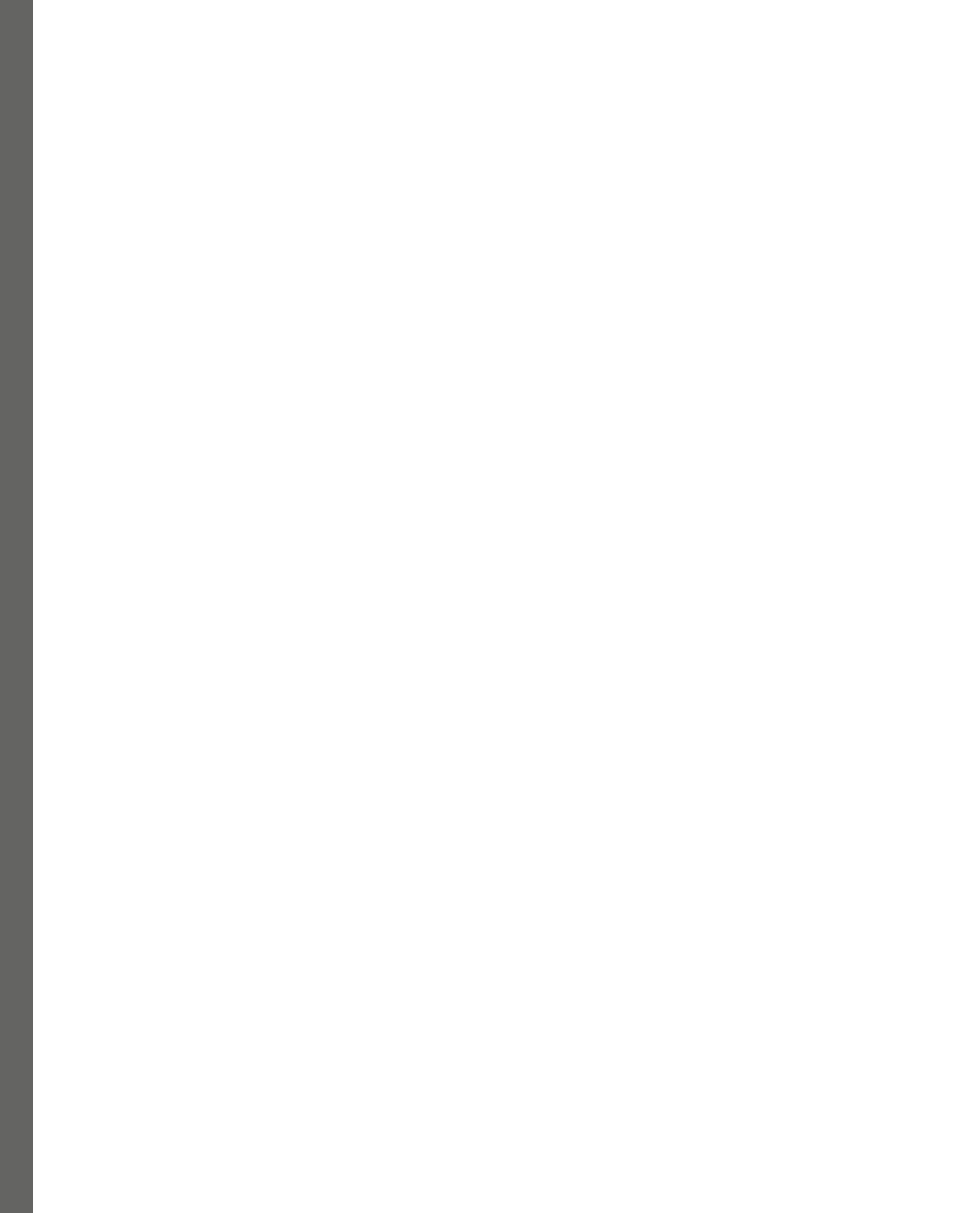
# **Part 1:**

## **Mastering Core Development Skills**

In this first part, we explore the essential skills needed to elevate your software development practices to a professional level. By breaking down core development skills, including testing, optimization, debugging, and refactoring, each of these areas will sharpen your ability to write cleaner, more efficient code and tackle complex challenges with confidence, ultimately improving your overall development process, thanks to Visual Studio's powerful assets and tools.

This part has the following chapters:

- *Chapter 1, Unit Testing and Test-Driven Development*
- *Chapter 2, Advanced Debugging Strategies*
- *Chapter 3, Advanced Code Analysis and Refactoring*
- *Chapter 4, Performance Optimization and Profiling*



# 1

# Unit Testing and Test-Driven Development

This chapter delves into the foundational concepts of unit testing and **Test-Driven Development (TDD)** within Visual Studio 2022. You will learn about the significance of writing tests before code and explore the integration of testing frameworks. Practical examples will guide you through creating effective unit tests, ensuring code reliability, and fostering a test-driven mindset.

We will embark on a journey to understand the essentials of unit testing and TDD. We'll start by gaining an overview of these foundational concepts and exploring their significance in the software development life cycle.

Following this introduction, we will delve into the practical aspects of setting up unit testing within Visual Studio Code. This section will guide you through the initial steps of configuring your development environment, laying the groundwork for the integration of testing frameworks into your workflow.

Once the setup is complete, we will transition into creating our first test using IntelliTest, a feature of Visual Studio that automates the generation of unit tests. This hands-on experience will demonstrate how IntelliTest can streamline the testing process, making it more efficient and less time-consuming.

Next, we will apply what we've learned in a real-world example, practicing TDD from the ground up. This section will walk you through the process of writing tests before code, a core principle of TDD, and how it can lead to more robust and reliable software.

Finally, we will explore how to automate your testing process with Live Unit Testing, a feature that runs your unit tests in the background as you code. This section will show you how to enable and utilize Live Unit Testing to enhance your development workflow, ensuring that your code remains reliable and bug-free throughout the development process.

In this chapter, we're going to cover the following main topics:

- Introducing unit testing and TDD
- Setting up unit testing in Visual Studio

- Creating tests with IntelliTest
- Practicing TDD with a real-world example
- Automating your tests with Live Unit Test

## Technical requirements

While writing this chapter, I used the following version of Visual Studio:

- Visual Studio Enterprise 2022 version 17.12.0
- Preview 1.0

The code files for this chapter can be found at <https://github.com/PacktPublishing/Mastering-Visual-Studio-2022/tree/main/ch01>

## Introducing unit testing and TDD

Before delving into the usage of unit testing and the TDD approach through Visual Studio, let's begin with a refresher on what they are and why it's valuable to consider them in your project.

Unit testing is a focused software testing method that targets individual units or components within a software system. The primary aim of unit testing is to ensure that each software unit functions as intended, meeting specified requirements. Typically conducted by developers, unit testing takes place early in the development stage, preceding the integration and testing of the entire system.

Automated unit tests are executed whenever code undergoes modification, ensuring that new changes do not disrupt existing functionality. These tests are meticulously crafted to validate the smallest conceivable unit of code, such as a function or method, in isolation from the broader system. This approach empowers developers to swiftly identify and address issues in the initial phases of development, thereby enhancing overall software quality and reducing the time needed for subsequent testing phases.

## The F.I.R.S.T principles of unit testing

By conscientiously embracing the **Fast, Isolated/Independent, Repeatable, Self-Validating, Timely (F.I.R.S.T.)** principles in unit testing, developers are guided in the meticulous crafting of effective tests. These principles serve as a compass, steering the testing process toward excellence by instilling key criteria that underpin the reliability and robustness of unit tests.

Let's look at each of the factors of F.I.R.S.T. in more detail:

- **Fast:** The speed of execution is a fundamental characteristic of effective unit tests. It is imperative that unit tests run swiftly, as developers frequently execute them throughout the development process. A fast test suite provides rapid feedback, allowing developers to promptly identify and rectify issues. This not only accelerates the development cycle but also cultivates a culture of

responsiveness, where developers are encouraged to run tests frequently. The ability to obtain quick feedback enhances the early detection of potential problems, fostering a proactive approach to software quality.

- **Isolated/independent:** Independence is a cornerstone principle in unit testing. Each unit test should operate in isolation, devoid of dependencies on other tests. The order in which tests are executed or the success or failure of one test must not impact the results of another. This isolation ensures that developers can pinpoint and address issues in a focused manner, simplifying the debugging process. By adhering to the principle of independence, developers gain the advantage of precisely identifying the source of a problem, thereby expediting the resolution process.
- **Repeatable:** The repeatability of unit tests is paramount for maintaining the integrity of the testing process. A unit test should consistently produce the same result every time it is executed. This consistency ensures reliability in the test results, facilitating a dependable and predictable testing environment. If a test fails, developers should be able to reproduce the failure consistently, enabling them to investigate and resolve issues with confidence. Repeatability is a cornerstone for establishing trust in the testing suite and the overall software development process.
- **Self-validating:** Unit tests should possess self-contained criteria for success or failure, requiring no manual interpretation. A self-validating test ensures unambiguous results, reducing the potential for misinterpretation. Developers can quickly understand the state of the code based solely on the test results, streamlining the debugging and issue-resolution processes. The self-validating nature of unit tests contributes to the clarity and effectiveness of the testing suite, enabling developers to make informed decisions based on the unambiguous outcomes of each test.
- **Timely:** Timeliness in test writing is a critical factor in the effectiveness of unit testing. Ideally, tests should be authored before the corresponding code is implemented. This proactive approach ensures that tests serve as a living specification for the desired behavior, guiding the implementation process. Writing tests in a timely manner establishes a foundation for a well-defined and controlled development cycle. Timely testing also plays a pivotal role in identifying and addressing issues early in the development cycle, reducing the likelihood of defects propagating to later stages of the software development process. Embracing timely testing enhances the overall efficiency and reliability of the software development life cycle.

Now that we've established a foundational understanding of unit testing and its principles, let's turn our attention to the broader context of TDD.

## TDD – unifying principles for software quality

These F.I.R.S.T principles seamlessly pave the way to an overarching methodology known as TDD. TDD merges design, development, and testing into a unified framework, providing developers with a comprehensive approach to not only crafting simple and clean code but also ensuring thorough testing. The incremental development approach systematically tests all facets of business logic, making

TDD the gold standard for creating high-quality software. It embodies the best coding and design practices throughout the entire process.

The main purpose of TDD is to achieve simpler and more reliable code. For that, TDD follows a simple and effective process, illustrated by the **red-green-refactor cycle**:

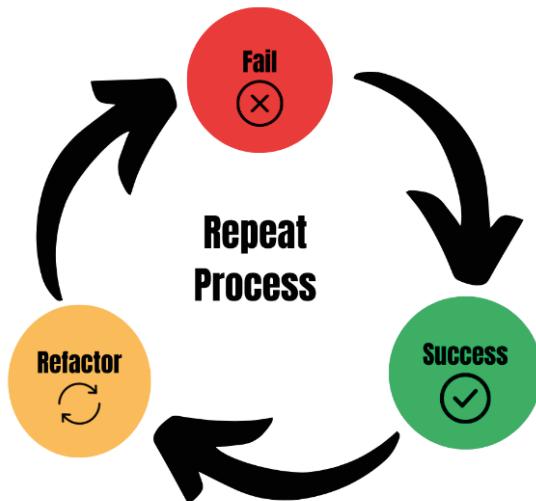


Figure 1.1 – The TDD cycle

Let's understand this TDD cycle in more detail:

- **Red – writing a failing test:**

In the first phase of TDD, known as the Red phase, developers embark on the journey by articulating their intent through the creation of a failing test. This test specifically targets a small unit of functionality that has not yet been implemented. It serves as a tangible expression of the desired behavior, essentially outlining the expectations for the code that is yet to be written. It is important to note that, at this stage, the test is expected to fail since the corresponding functionality is absent from the code base.

The primary purpose of the Red phase is to set a clear objective for the subsequent development process. By initially focusing on the expected outcome without any code in place, developers define a roadmap for the functionality they are about to implement. This intentional act of writing a test before writing the code not only establishes a specification for the upcoming functionality but also helps solidify the developer's understanding of the problem at hand.

- **Green – writing the minimum amount of code to pass the test:**

Following the Red phase, the development process proceeds to the Green phase. Here, the developer crafts the minimal amount of code necessary to make the previously written test pass successfully. The emphasis during this phase is on simplicity and efficiency. The goal is not to create an elaborate solution but to address the immediate requirement highlighted by the failing test.

By concentrating on the bare minimum code required for success, developers foster a culture of simplicity and avoid unnecessary complexities. This focused approach encourages the creation of code that meets the specific needs outlined by the test, laying the foundation for a solution that is both functional and concise. The success of the test in this phase signifies the achievement of the targeted functionality, validating the initial expectations set in the Red phase.

- **Refactor – refactoring the code:**

With a passing test in place, the development process enters the Refactor phase. At this juncture, the developer takes a step back to review and enhance the code that has been implemented. The primary objectives during refactoring are to improve the maintainability, readability, and extensibility of the code without altering its functionality.

Refactoring is a critical aspect of TDD as it ensures that the code base remains clean and adaptable to future changes. Developers strive to eliminate redundancies, improve naming conventions, and apply design patterns where appropriate. This phase reinforces the commitment to producing not only functional but also well-crafted code. The successful completion of this phase sets the stage for a code base that is not only effective but also sustainable in the long run.

- **Repeat – iterative process:**

TDD follows an iterative approach, and the Repeat phase encapsulates the cyclical nature of this development methodology. The entire Red-Green-Refactor cycle is repeated for each new piece of functionality that needs to be implemented. This iterative process ensures a systematic and gradual evolution of the code base, with each cycle contributing to the overall development of the software.

By repeating the cycle, developers continually refine and expand the application, responding to evolving requirements and ensuring that the code remains aligned with the project's objectives. This iterative nature of TDD promotes adaptability and agility, making it a valuable methodology for projects with changing or evolving requirements.

This approach not only ensures the creation of functional software but also cultivates a mindset of continuous improvement and adaptability in the development process.

## AAA pattern – a structured approach to testing

Building upon the principles of TDD and F.I.R.S.T, another key methodology in the realm of unit testing is the **Arrange-Act-Assert (AAA)** pattern. This pattern provides a uniform structure for organizing tests within a suite. The AAA pattern contributes significantly to the readability and maintainability of test suites, aligning seamlessly with the overarching goals of TDD.

The AAA pattern breaks down a test into three distinct sections:

- **Arrange:** This section is about setting up the test environment.

In the Arrange section, the focus is on preparing the environment for the test. This involves setting up the objects to be tested, bringing the **System under Test (SUT)** to a specific state, and configuring any dependencies. Whether instantiating objects directly or preparing test doubles dependencies, the goal is to establish a controlled and consistent starting point for the test.

- **Act:** This section is about performing the action on the SUT.

The Act section is where the actual interaction with the SUT takes place. This involves invoking a method or action on the SUT, passing any required dependencies, and capturing the output value, if applicable. The Act phase is crucial for simulating the real-world usage of the system and observing its behavior.

- **Assert:** This section focuses on verifying the expected outcome.

In the Assert section, the test makes explicit claims about the expected outcome. This may include checking the return value, inspecting the final state of the SUT and its collaborators, or verifying the methods called on them. The Assert phase is the culmination of the test, ensuring that the behavior aligns with the anticipated results.

The AAA pattern's structured approach provides clarity and consistency across all tests in a suite. By adhering to this pattern, developers can easily comprehend and navigate through tests, ultimately reducing the maintenance cost of the entire test suite. Whether starting with Arrange, Act, or Assert, the AAA pattern accommodates different testing styles while promoting a unified and systematic testing methodology.

Now that we've had a tour of unit testing and TDD, let's dive in and see how to create a unit test project within Visual Studio.

## Setting up unit testing in Visual Studio 2022

In this section, we will first create a unit test project within Visual Studio. Once that is done, we'll go through an overview of the **Test Explorer** view and its main options. The **Test Explorer** view serves as a central hub for all testing activities within the IDE, offering a comprehensive overview of our test suite's status and performance. It allows us to easily navigate through test results, identify failures, and access detailed information about each test case.

## Creating a unit test project

Visual Studio and .NET support three testing frameworks:

- **MSTest:** This is the default testing framework provided by Microsoft for .NET applications. It's fully integrated into Visual Studio and offers features for writing and running tests within the IDE.
- **NUnit:** This is a popular open-source testing framework for .NET applications. It provides a flexible and extensible platform for writing and executing tests and is widely used in the .NET community.
- **xUnit:** This is another open-source testing framework for .NET applications. It follows a more modern and flexible approach compared to MSTest and NUnit and is gaining popularity among .NET developers.

First, you need to create a new project for your test set, in the solution containing the project we want to test. Right-click on your solution and select **Add | New Project....** This will take you to the following screen:

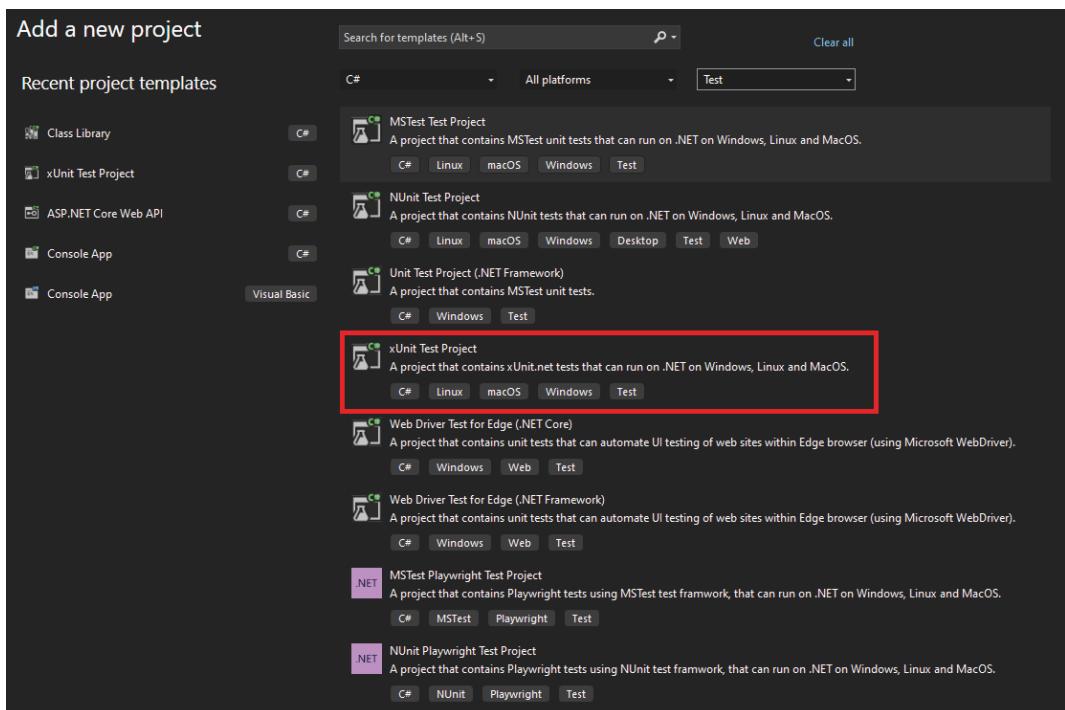


Figure 1.2 – Adding a new test project

I filtered this view by the three top dropdowns to keep displaying the template projects for testing available by default in Visual Studio. You may notice that the last two templates on the list mention **Playwright**. Playwright is a library that enables end-to-end testing for modern web apps. With it, you can create unit tests to assess the functionality of your user interface.

Since xUnit is the most used framework in the industry, we will focus on it for our examples.

Another way to create your unit test project is by right-clicking directly on your class in your code base and selecting **Create Unit Tests** from the menu:

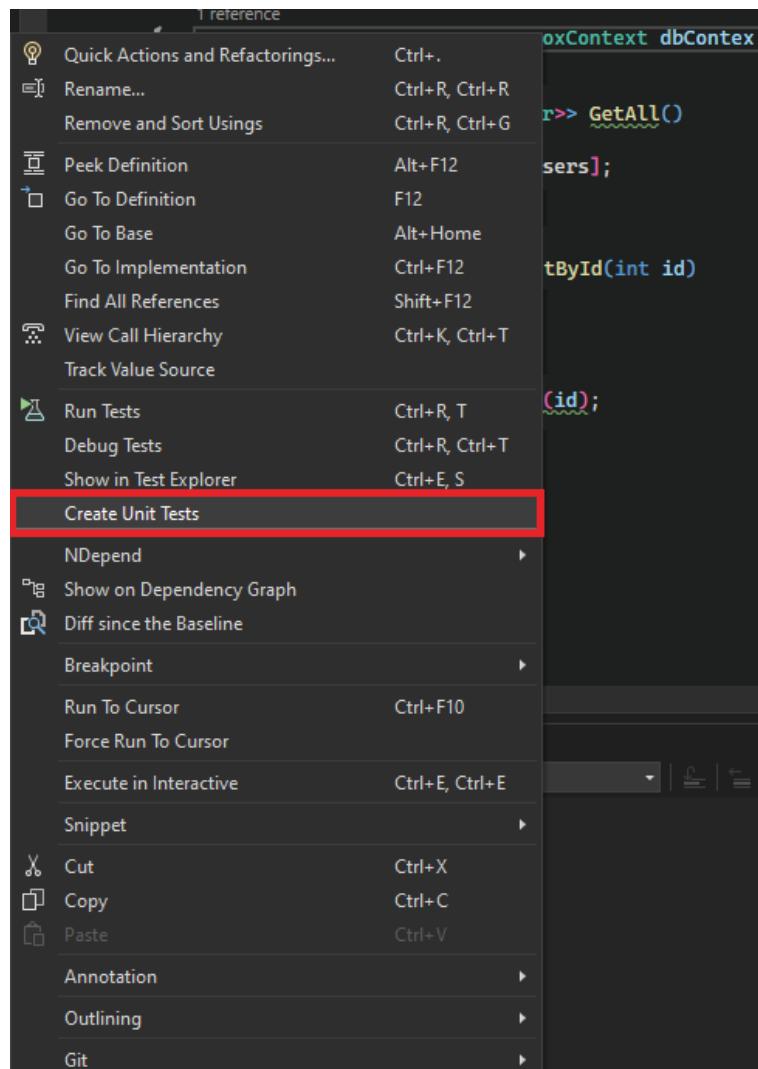


Figure 1.3 – Selecting Create Unit Tests from the menu

This action will open a menu that allows you to configure your unit test project:

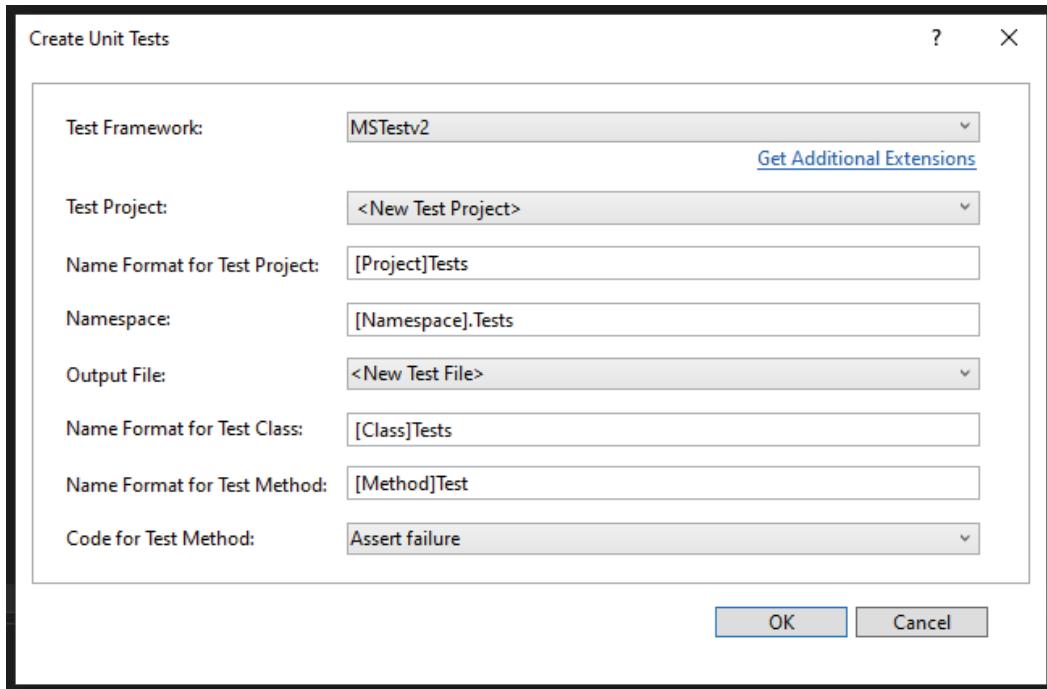


Figure 1.4 – The Create Unit Tests window

By default, this allows you to create an MSTestv2 project, but you can install additional extensions to use other frameworks. If you already have a unit tests project created, you may add your test methods to it.

Regardless of how you create them, all the tests in this project can be managed in the **Test Explorer** view and we will explore it in the next section.

## Overview of the Test Explorer view

Test Explorer serves as a powerful and centralized tool in Visual Studio 2022, facilitating the management and execution of unit tests. This feature-rich window allows developers to seamlessly view, organize, and run their unit tests within the **Integrated Development Environment (IDE)**.

To access Test Explorer, there are two options:

- You can navigate to the **Test** menu in Visual Studio or utilize the convenient keyboard shortcut accessed by pressing *Ctrl + E*, releasing them, and then pressing *T*.

- Since version 17.6, you can use the **all-in-one search** feature to quickly access all the features provided by Visual Studio. Simply open an all-in-one search using the *Ctrl + Q* shortcut, type **Test Explorer**, then press *Enter*, and you'll be directed to the **Test Explorer** window.

Here's a visualization of the **Test Explorer** window:

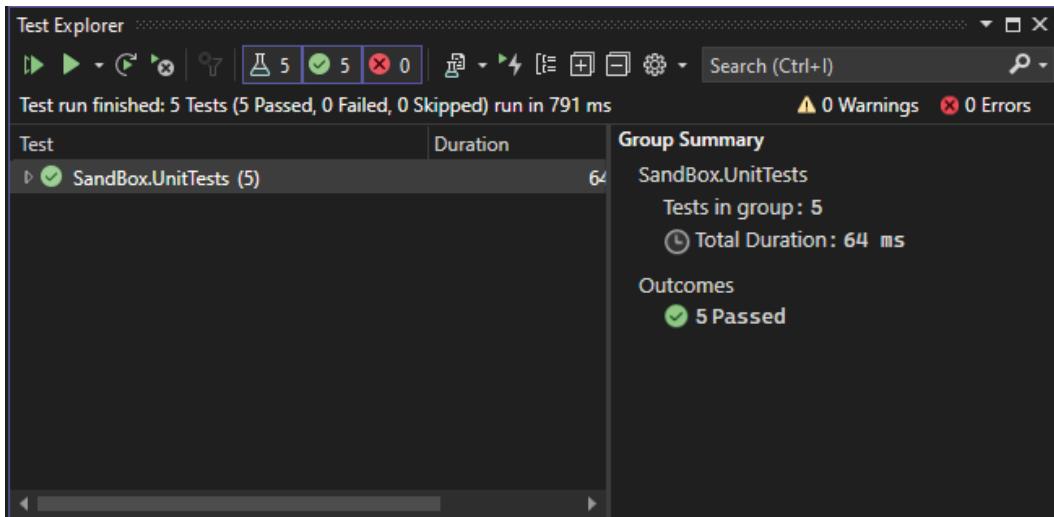


Figure 1.5 – The Test Explorer window

The **Test Explorer** console provides a clear and organized view of all available tests, making it easy for developers to navigate through their test suites. This interface offers advanced functionality for test management, such as the following:

- **Categorize and group tests:** Utilize attributes and categories to group tests based on functional areas, features, or specific requirements. This helps in creating a logical hierarchy and makes it easier to focus on specific subsets of tests. By default, tests are grouped based on **Project**, followed by **Namespace**, and then **Class**. If you wish to modify the organization of your tests, simply click on the **Group By** button in Test Explorer and choose a new grouping criterion:

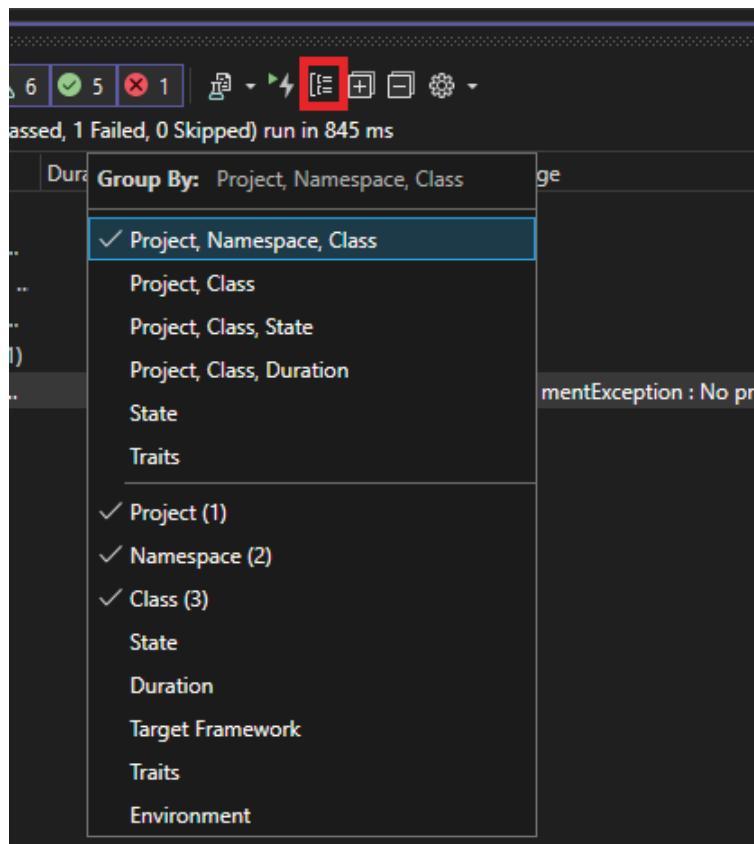


Figure 1.6 – Test Explorer – Group By

Here is the description of the groups you can choose from:

- **Duration:** Groups tests by execution time: **Fast**, **Medium**, and **Slow**
- **State:** Groups tests by execution results: **Failed Tests**, **Skipped Tests**, **Passed Tests**, and **Not Run**
- **Target Framework:** Groups tests by the framework their projects' target
- **Namespace:** Groups tests by the containing namespace
- **Project:** Groups tests by the containing project
- **Class:** Groups tests by the containing class

- **Filtering and searching:** With a large code base, locating specific tests can be challenging. Test Explorer offers filtering options and a powerful search functionality.

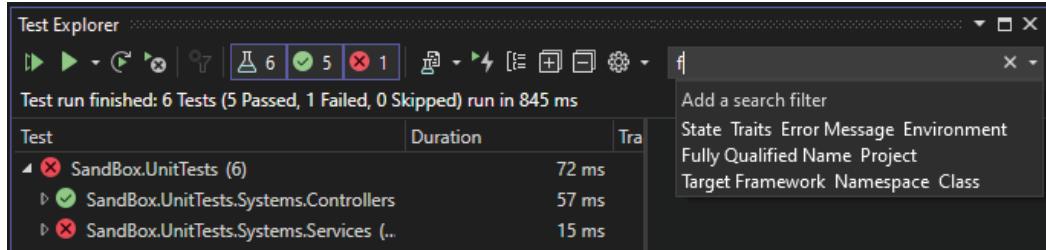


Figure 1.7 – Test Explorer – applying a search filter

You can enter a command in the textbox, such as `Namespace : " "` or any other filter criteria you wish to use for your search. Additionally, you can directly click on the provided command to initiate your filtering.

- **Creating a playlist:** You can create and manage playlists of tests in Visual Studio to organize and run specific sets of tests. To create a playlist, select one or more tests in Test Explorer, right-click, and choose **Add to Playlist | New Playlist**.

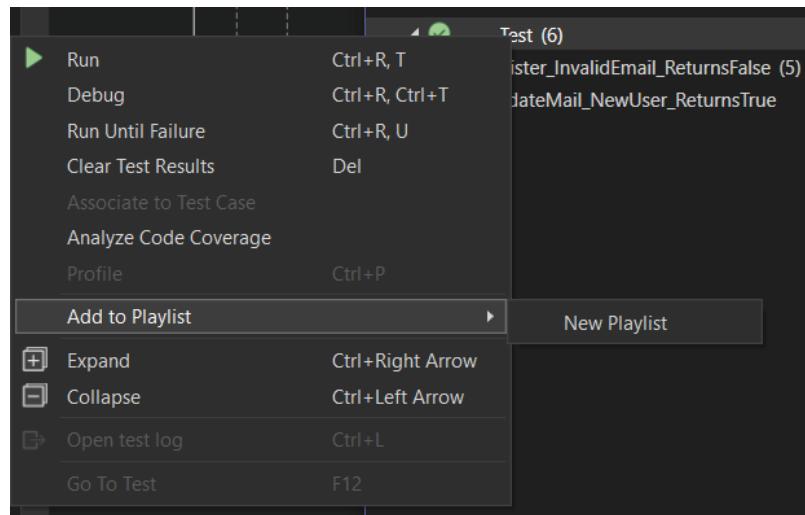


Figure 1.8 – Test Explorer – Add to Playlist

The playlist opens in a new **Test Explorer** tab, where you can save it with a name and location. You can edit playlists by adding or removing tests, and you can use the **Edit** button to manage tests more conveniently with checkboxes. Playlists can be dynamic, and automatically updated based on the tests included or excluded. They can be saved as XML files and edited manually if needed.

Now we have had an overview of the **Test Explorer** view, let's explore a preview functionality of Visual Studio 2022 and create a test using it.

## Creating a test with IntelliTest

In this section, our goal is to explore how to use IntelliTest for creating tests. For that, we will first learn how to enable the IntelliTest functionality. Next, we will learn how to execute it and generate tests with it. At the end of the section, we will learn how to organize our generated unit tests by saving, executing, and reviewing them.

IntelliTest offers characterization tests, allowing you to understand how code behaves through a suite of traditional unit tests. This suite can serve as a regression suite, aiding in handling the challenges of refactoring legacy or unfamiliar code.

Through guided test input generation, IntelliTest employs an open-code analysis and constraint-solving approach to automatically produce precise test input values, often without requiring user intervention. It generates factories for complex object types and allows customization of these factories to meet specific requirements. Assertions specified as correctness properties in the code are utilized to further direct test input generation.

Integrated seamlessly into the Visual Studio IDE, IntelliTest provides a cohesive environment for test suite generation. Information gathered during test suite creation, including automatically generated inputs, code outputs, and generated test cases, as well as their pass or fail statuses, is readily available within Visual Studio. This integration enables easy iteration between code refinement and IntelliTest reruns without leaving the IDE. Test results can be saved as a unit test project within the solution and are automatically recognized by Visual Studio Test Explorer.

IntelliTest is available in Visual Studio Enterprise Edition. It systematically explores the code base, generating test data and unit tests for each method to ensure comprehensive code coverage and validate code behavior. As of the time of writing, IntelliTest is a preview feature in Visual Studio 17.9, so you must enable it from the **Options** menu by following these steps:

1. Go to **Tools | Options | Manage Preview Features**.
2. Check the **IntelliTest support for NetFx and Net6 using Z3 v4** option under **Preview Features**.

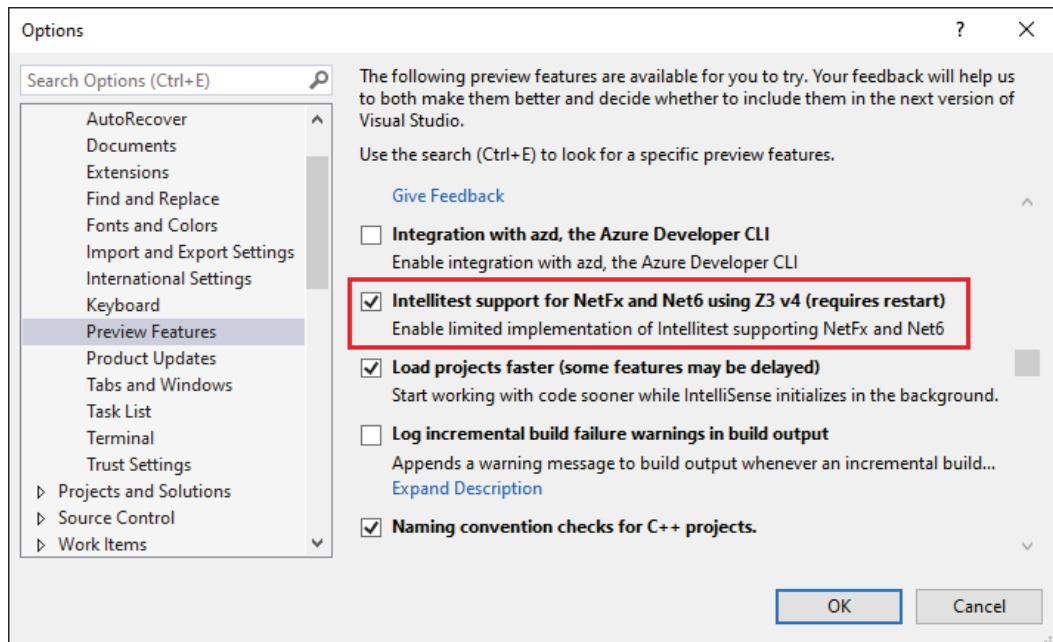


Figure 1.9 – Enable Preview Features – Intellitest

Once enabled, we can explore how IntelliTest works. For example, I've created a public class implementing `FizzBuzz`. To generate tests with IntelliTest for the class, right-click on the class name, find the **IntelliTest (Preview)** option, and then select **Generate Tests**:

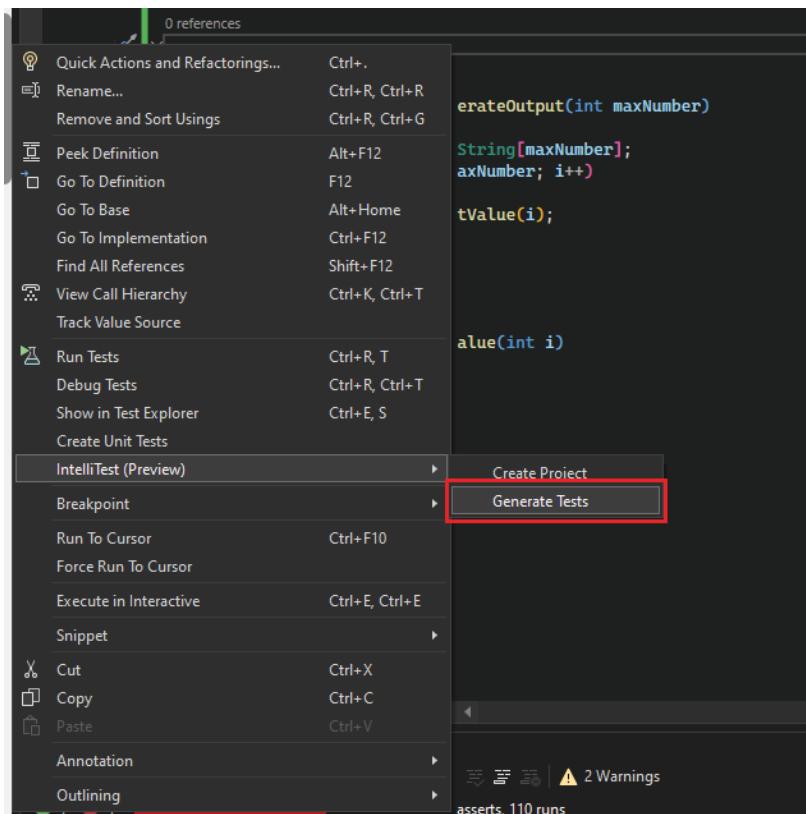


Figure 1.10 – IntelliTest – Generate Tests

## IntelliTest execution and test generation

IntelliTest executes our code multiple times using various inputs. Each execution is documented in a table, displaying the input test data along with the corresponding output or any exceptions that occur.

IntelliTest Generate Tests - stopped				
FizzBuzz.GenerateOutput(Int32)				
12/12 blocks, 0/0 asserts, 110 runs				
▲	▲	maxNumber	result	Summary / Exception
✓	1	0	{}	
✓	2	1	{"1"}	
✗	3	int.MinValue		OverflowException Arithmeti...
✓	4	2	{"1", "2"}	
✓	5	3	{"1", "2", "F...	
✓	6	7	{"1", "2", "F...	

Details

```

[TestMethod]
[PexGeneratedBy(typeof(FizzBuzzTest))]
public void GenerateOutput4050()
{
    string[] ss;
    ss = this.GenerateOutput(0);
    PexAssert.IsNotNull((object)ss);
    PexAssert.AreEqual<int>(0, (int)(ss.Length));
}

```

Figure 1.11 – Generating tests with IntelliTest

After test generation, we can access the test project and examine the parameterized unit tests that have been generated.

## Saving, executing, and reviewing unit tests

We may choose the unit tests we want to keep and use the save option to save them.

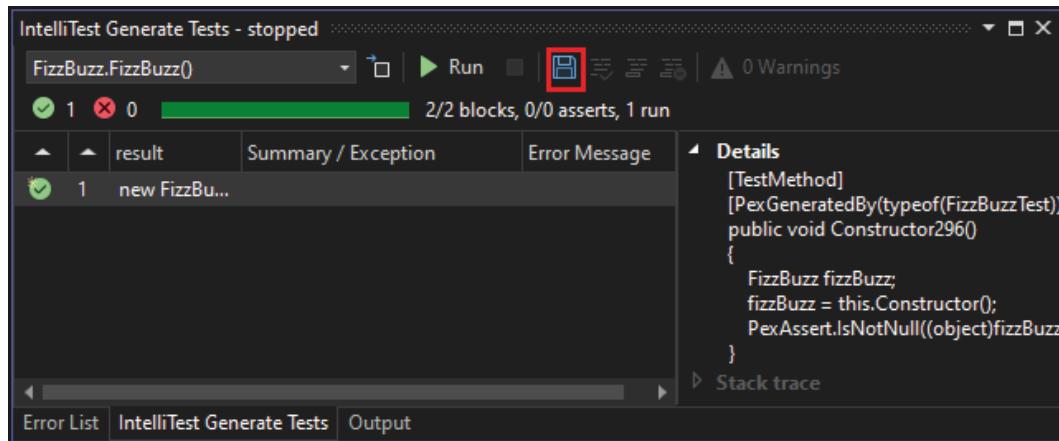


Figure 1.12 – IntelliTest’s save option

The select box in the top-left corner of the window allows you to filter through all the methods of the class. The individual unit tests, associated with each row, are stored in the `.g.cs` file within the test project, while the parameterized unit test is stored in its respective `.cs` file.

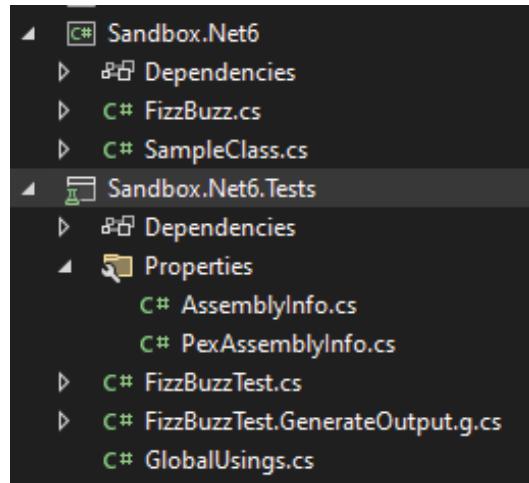


Figure 1.13 – IntelliTest – unit test project

---

We can execute these unit tests and review the outcomes using Test Explorer, just as we would handle manually created unit tests.

Similar to classic unit tests, we can generate the project without running IntelliTest first by choosing the option in the menu.

Using IntelliTest to generate tests from your code base is a valuable tool, especially for legacy projects and non-regression testing. Now, let's move forward with a real-world example to illustrate how to implement TDD within Visual Studio.

## Practicing TDD with a real-world example

In TDD, everything begins with the specification and a thorough understanding of the intended functionality. It's essential to grasp the meaning of what we want to accomplish in order to write effective tests. With that in mind, let's establish a scenario for our example.

We'll create a `ValidateMail` method that will process email verification. Our goal here is to explore the tools available in Visual Studio to enhance our TDD experience.

Now that we have outlined our requirements, our next step is to begin by creating our unit test project. For this, we'll choose xUnit, as discussed earlier in this chapter. We will create an empty Class Library Project:

1. We'll start by writing a test, in our xUnit project, that specifies the behavior or functionality we want to implement:

```
[Fact]
public void ValidateMail_NewUser_ReturnsTrue()
{
    // Arrange
    var user = new User();

    // Act
    bool result =
        user.ValidateMail("john@example.com");

    // Assert
    Assert.True(result);
}
```

2. The first thing we'll notice when writing this code is that our test doesn't compile because the `User` class doesn't exist yet in our Class Library Project.

3. We'll leverage the power of IntelliSense to create our class and its method. IntelliSense is an integrated code completion tool within Visual Studio, offering various features such as List Members, Parameter Info, Quick Info, and Complete Word. These functionalities facilitate a deeper understanding of the code being utilized, assist in managing typed parameters, and enable the quick addition of calls to properties and methods with minimal keystrokes.
4. Place the cursor on the `User()` class and expand the IntelliSense menu, selecting the **Generate new type...** quick action:

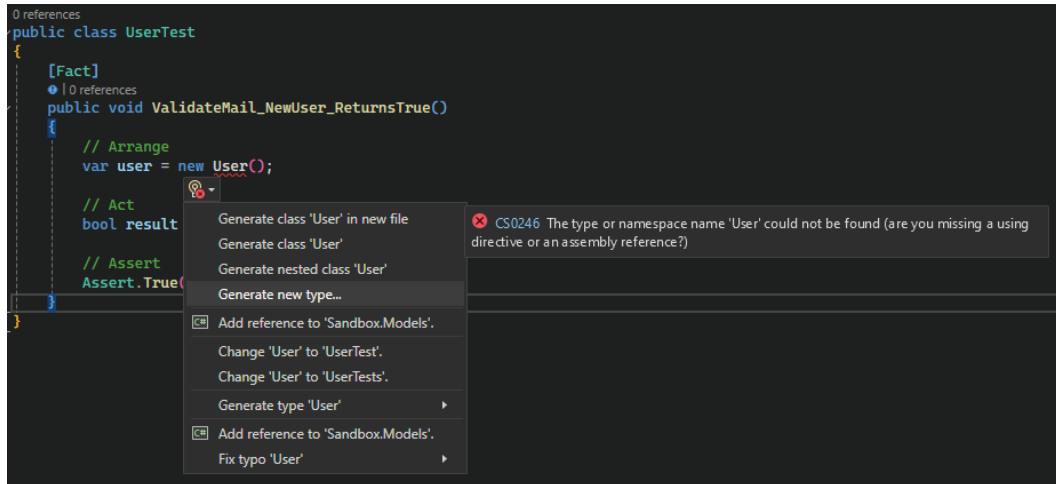


Figure 1.14 – Generate new type...

5. This will open a window where we can choose the accessibility (**Default**, **Internal**, or **Public**), the kind (**Class** or **Struct**), the project, and whether to create a new file or use an existing one. Since we selected our Class Library project, the accessibility of our new class is obviously set to **Public**.

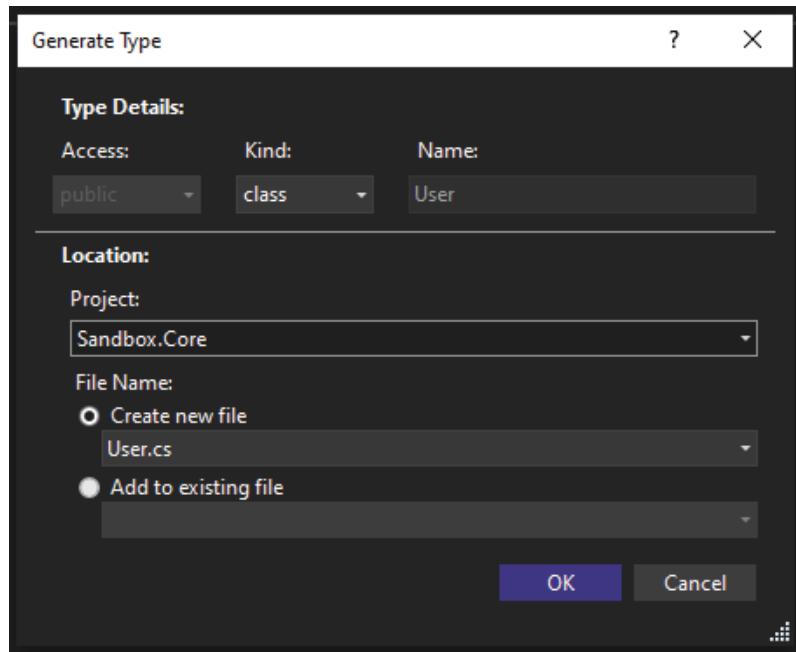


Figure 1.15 – The Generate Type window

6. Now, in the same way, we will use IntelliSense to create our `ValidateMail()` method:

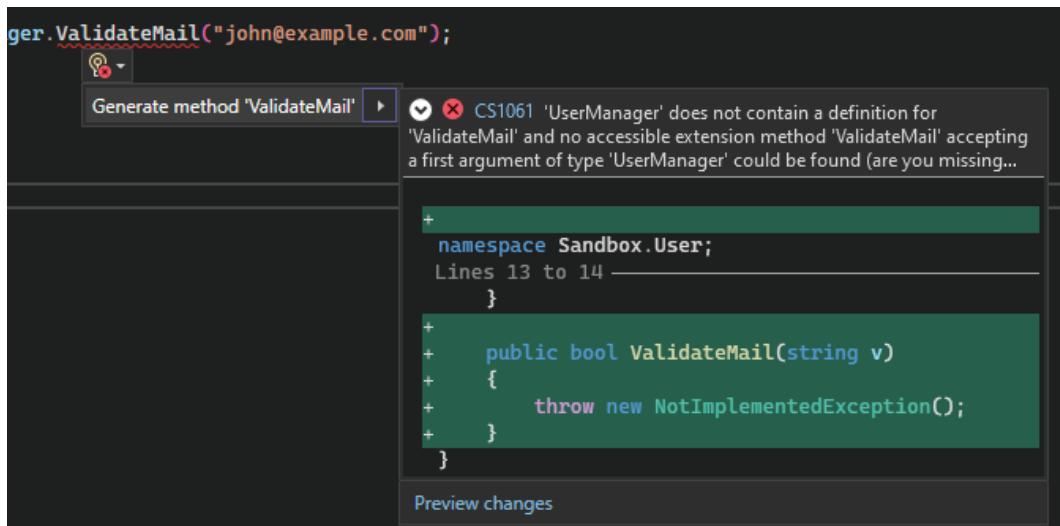


Figure 1.16 – Generate method ValidateMail'

7. With that done, our test is built, but it still fails because we have yet to fix its behavior.

Here's our `User` class with the minimum amount of code to make our test pass:

```
public class User
{
    public User()
    {
    }

    public bool ValidateMail(string mail)
    {
        return true;
    }
}
```

Now we have a `ValidateMail` method returning `true` that should respect the specification of our test.

You can use the quick access feature, above your method, to run the unit test without going to the **Test Explorer** view.

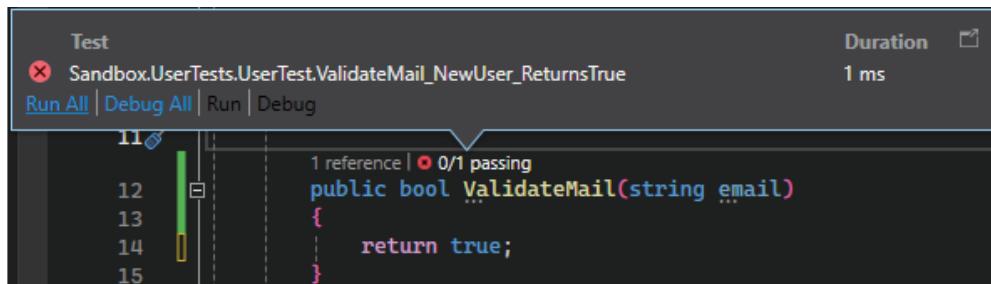


Figure 1.17 – Using the quick access tool to run a test

Here we are; our first test is green. We can write the second testing class that will handle the cases where our `ValidateMail()` must return `false`. For that, we will use the `Theory` and `InlineData` xUnit attributes:

```
[Theory]
[InlineData("invalidemail")]
[InlineData("invalidemail@")]
[InlineData("invalidemail@example")]
[InlineData("invalidemail@example.")]
[InlineData("invalidemail@.com")]
public void ValidateMail_InvalidEmail_ReturnsFalse(string email)
{
```

```
// Arrange
var userManager = new User();

// Act
bool result = userManager.ValidateMail(email);

// Assert
Assert.False(result);
}
```

We are up for another iteration, from red to green. Here's the method with the minimum amount of code to make the unit test pass:

```
public bool ValidateMail(string email)
{
    if (!email.Contains('@'))
        return false;

    if (email.EndsWith('@'))
        return false;

    if (!email.Contains('.'))
        return false;

    if (email.EndsWith('.'))

        return false;

    if (email.Contains("@."))
        return false;

    return true;
}
```

Here, we are checking each condition of our test method, with one `If` for each test to make it pass.

As you can see, our approach maintains a naïve and simple perspective. Now, it's time to enter the refactor phase of our TDD process. For that, we will explore a feature of Visual Studio named Live Unit Test.

## Automating your tests with Live Unit Test

In this section, we will explore the capabilities of automating testing with Live Unit Test. First, we will learn how to configure Live Unit Testing. Then, we will see how to launch it.

Live Unit Testing revolutionizes the testing process by automatically executing unit tests in real-time as you make code changes. This dynamic feature provides developers with the ability to refactor and modify code with increased confidence. By automatically running all affected tests during code editing, Live Unit Testing ensures that any changes made do not introduce regressions.

Furthermore, Live Unit Testing offers insights into the adequacy of test coverage for your code base. It visually presents code coverage in real-time, allowing developers to quickly identify areas where tests are lacking.

Let's activate this feature by navigating to **Test | Live Unit Testing | Start**.

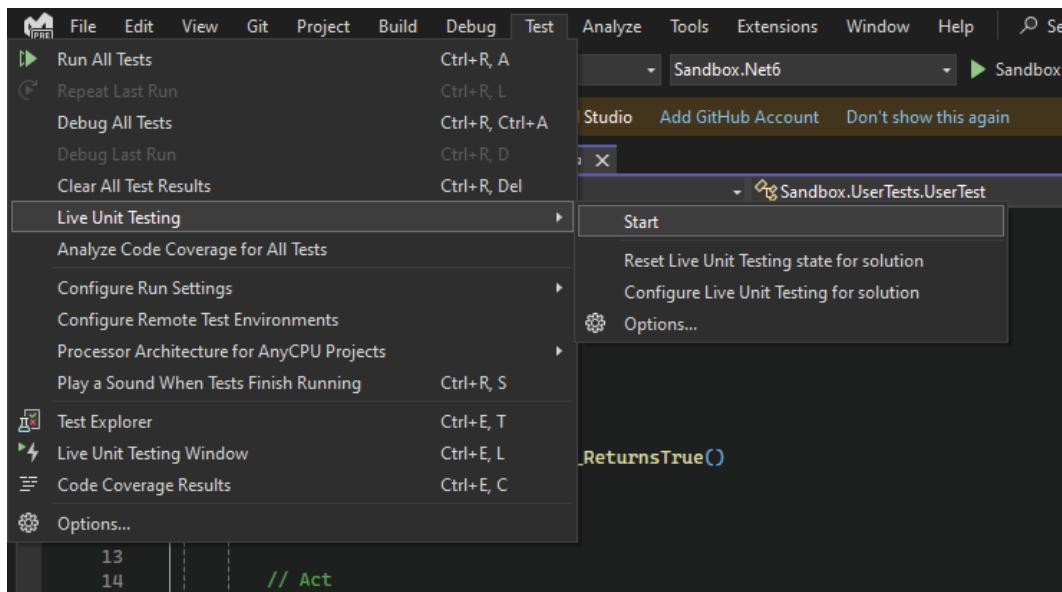


Figure 1.18 – Start Live Unit Testing

When you launch Live Unit Testing for the first time, the configuration window will open up:

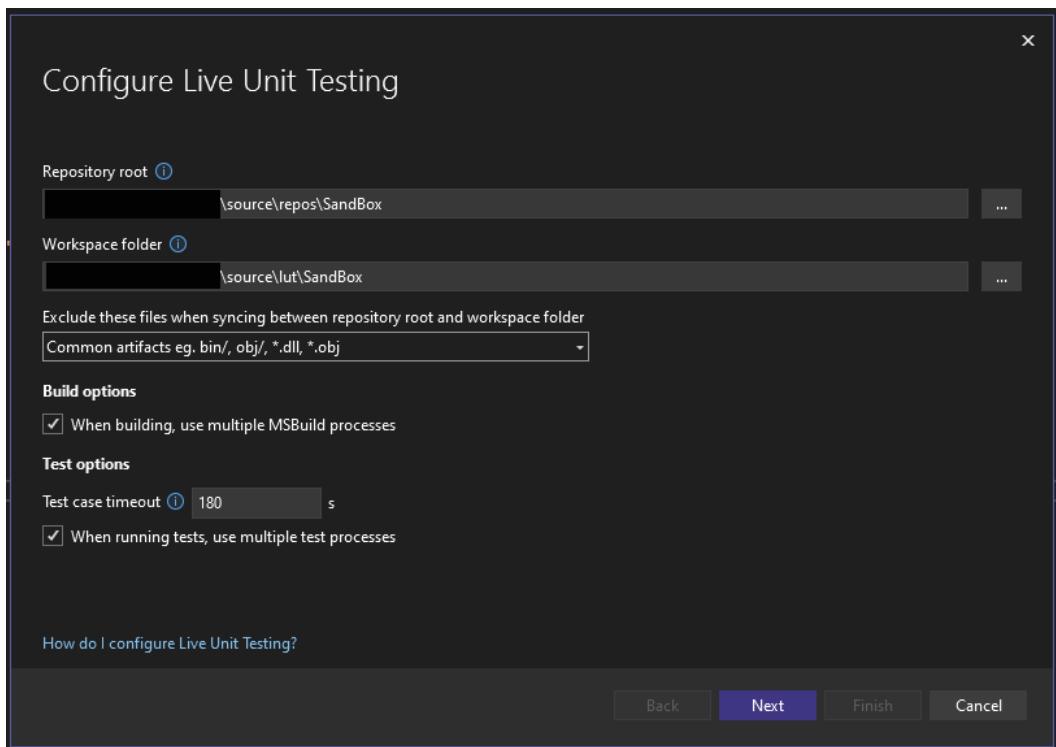


Figure 1.19 – Configure Live Unit Testing

## Configuring Live Unit Testing

When Live Unit Testing is deactivated, you can access the setup wizard by navigating to **Test | Live Unit Testing | Configure Live Unit Testing**.

During a Live Unit Testing operation, a workspace — a duplicate of the original repository — is established. Any modifications made in Visual Studio that are not yet saved are integrated into this workspace. Subsequently, Live Unit Testing initiates a build, conducts a test run, and furnishes the latest code coverage report.

The initial setup in the wizard should address file copying origins and destinations.

Let's understand more about the settings in this wizard:

- **Repository root:** This is the main folder that contains all the necessary files for Live Unit Testing. It should include all the source code, binaries, and tools required for testing. If the solution file is not located within this root folder, adjustments may need to be made to ensure proper setup.
- **Workspace folder:** This (set, by default, to **lut** for Live Unit Testing) is where Live Unit Testing stores a copy of the repository. By default, this root is created in the user's home folder, but it can be customized based on preference or specific requirements.
- **Excluded files:** Exclude generated artifacts from being copied to the Live Unit Testing workspace to prevent interference with regular builds.
- **Build options:** By default, Live Unit Testing utilizes multiple CPU cores for faster builds.
- **Test case timeout:** This allows setting a specific time duration, after which tests are automatically aborted if they exceed the allotted time.
- **Use multiple processors:** By default, Live Unit Testing attempts to utilize multiple processors for faster test execution. However, deselect this option if your machine experiences slowdowns or if parallel test execution causes issues, such as file conflicts.

If we select <custom> for the **Excluded** files, we can define our own rules. The setup wizard will bring us to the **lutignore** file editor without default values filled in.

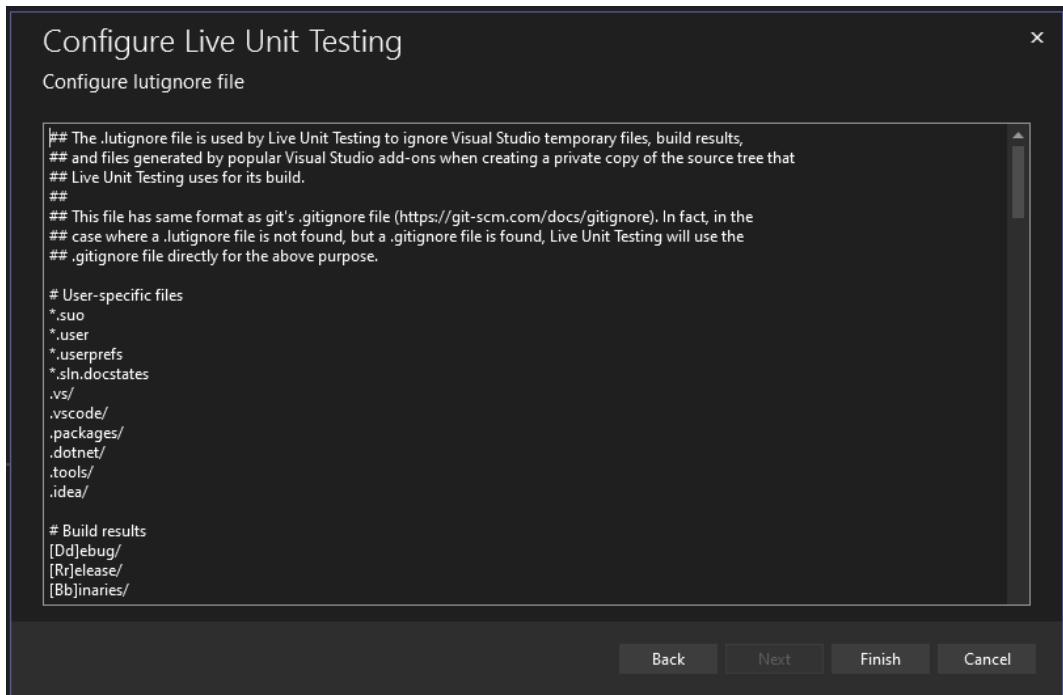


Figure 1.20 – Configure lutignore file

Let's get an overview of the `lutignore` file structure according to the Microsoft documentation.

The `lutignore` file follows the same format as a `gitignore` file. It should include rules that correspond to folders or files created during the build process, ensuring they are not copied into the workspace for Live Unit Testing. For most default project templates, the provided `lutignore` file is adequate:

```
[BB] IN  
[OO] BJ
```

These rules prevent the copying of any `BIN` or `OBJ` folders to the Live Unit Testing workspace.

If your repository has a single build folder, specify that folder in the `ignore` file:

```
[AA] RTIFACTS/
```

This rule ensures that the `ARTIFACTS` folder is not copied to the Live Unit Testing workspace.

In cases where your repository contains additional tools within the build folder, those tools should also be excluded using matching patterns:

```
[AA] RTIFACTS/  
! [AA] RTIFACTS/TOOLS/
```

The first rule excludes the `ARTIFACTS` folder. The second ensures that the `TOOLS` subfolder within `ARTIFACTS` is copied, which may contain necessary tools and utilities.

## Launching Live Unit Testing

Now that we have configured Live Unit Testing, it is open and awaiting a playlist of tests to run.

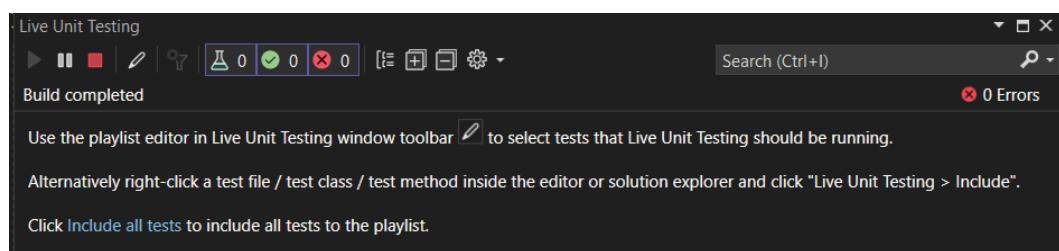


Figure 1.21 – Live Unit Testing console

Use the pen icon to add your set of tests. Alternatively, you can add all your tests by clicking on the link at the bottom of the message.

Returning to our previous example and our `ValidateMail()` function, you can see green checkmarks on the left side of our code lines. These indicate that the code is covered by passing unit tests.

```
2 references | 6/6 passing
public bool ValidateMail(string email)
{
    if (!email.Contains('@'))
        return false;

    if (email.EndsWith('@'))
        return false;

    if (!email.Contains('.'))
        return false;

    if (email.EndsWith('.'))
        return false;

    if (email.Contains("@."))
        return false;

    return true;
}
```

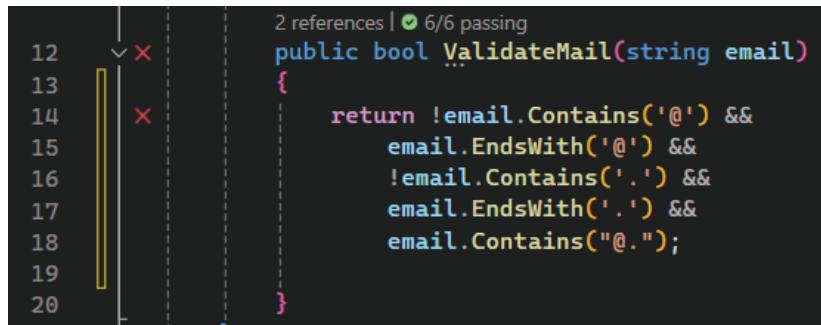
Figure 1.22 – Live Unit Testing checkmarks

Now, we can proceed to the refactoring step of our TDD iteration.

First, we will concatenate all the conditions and intentionally let the tests fail:

```
public bool ValidateMail(string email)
{
    return !email.Contains('@') &&
           email.EndsWith('@') &&
           !email.Contains('.') &&
           email.EndsWith('.') &&
           email.Contains("@.");
}
```

When we update the document, Live Unit Testing automatically rebuilds and updates the test status indicators, changing checkmarks to red crosses.



```
12     v X
13     X
14
15
16
17
18
19
20 }
```

```
2 references | 6/6 passing
public bool ValidateMail(string email)
{
    return !email.Contains('@') &&
        email.EndsWith('@') &&
        !email.Contains('.') &&
        email.EndsWith('.') &&
        email.Contains("@.");
}
```

Figure 1.23 – Live unit test red crosses

Now, we can fix our code and the checkmarks will appear automatically.

In our case, the process is very smooth, but in a larger code base with more unit tests, they may not have finished running while we have already made changes to the code. For this issue, you have an option in the top-level menu of Visual Studio: **Tools | Options | Live Unit Testing | General**.

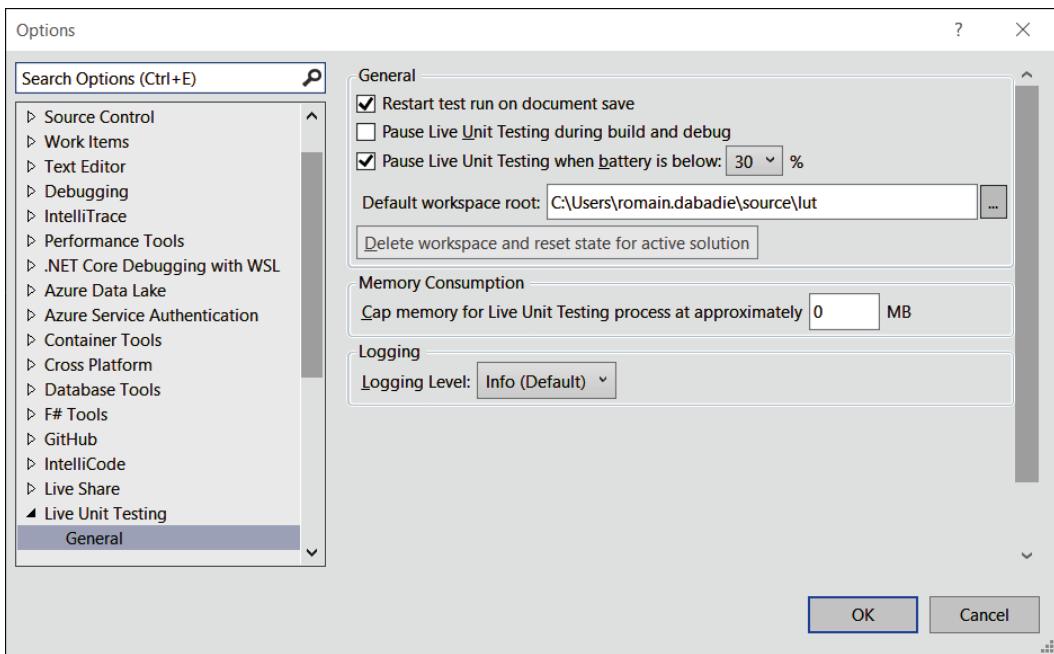


Figure 1.24 – Additional configuration – Live Unit Testing

Thanks to this window, you can customize your Live Unit Testing experience according to your needs.

We can see that Live Unit Testing offers a quick visualization of our code coverage. You can access a deeper analysis of the coverage of your code base in Visual Studio by navigating to the **Test** menu and selecting **Analyze Code Coverage for All Tests**. For more information, you can consult the Microsoft documentation at <https://learn.microsoft.com/en-us/visualstudio/test/using-code-coverage-to-determine-how-much-code-is-being-tested?view=vs-2022&tabs=csharp>.

## Summary

In this chapter, we covered the importance of unit testing as a non-regression or development process. We began with a reminder of good unit testing practices and TDD before exploring the functionalities offered by Visual Studio 2022 to enhance our productivity.

We provided an overview of the **Test Explorer** view, demonstrating how to manage and organize unit tests. Then, we delved into the new IntelliSense feature, which provides tools to easily add non-regression tests to legacy code bases with minimal effort, freeing our minds for future refactoring. Finally, we learned how to leverage IntelliSense tools and Live Unit Testing to facilitate our TDD experience.

In the next chapter, we will dive deeper into the world of debugging and explore all that Visual Studio 2022 offers for advanced strategies.

# 2

# Advanced Debugging Strategies

We explored unit testing and **Test-Driven Development (TDD)** in the last chapter, establishing a sturdy groundwork for crafting dependable code. Now, our focus shifts toward the realm of advanced debugging strategies within Visual Studio. The aim of this chapter is to equip ourselves with the requisite tools and methodologies to effectively tackle bugs.

Throughout this chapter, we'll cover the following topics:

- Mastering Visual Studio Debugger
- Advanced breakpoints and data inspections
- Elevating debugging with auto-decompilation and External Sources
- Remote debugging
- Extending debugging capabilities beyond our codebase
- Mastering remote debugging

By the end of this chapter, we'll be able to resolve complex bugs significantly faster, saving ourselves time and effort. We will also gain a deeper understanding of our code, leading to more robust and maintainable applications. Finally, we will understand how to work confidently in production environments, knowing we can handle unexpected issues with ease.

So, let's set off on this journey of advanced debugging mastery together. With the knowledge and skills we'll gain, we'll be able to face any debugging challenge head-on, ensuring quality and performance in our code.

## Technical requirements

While writing this chapter, I used the following version of Visual Studio:

- Visual Studio Enterprise 2022 Version 17.12.0
- Preview 1.0

## Mastering Visual Studio Debugger

Debugging is the process of finding and fixing errors or bugs in the source code of any software. There are several steps to follow to correctly accomplish this task.

The first step in debugging is defining the problem. This entails identifying its symptoms, comparing expected versus actual outcomes, determining its scope, assessing its severity and impact, and noting the steps to reproduce it. This clarity streamlines the troubleshooting process.

Reproducing the bug is often the most effective way to pinpoint its cause. However, if this can't be done, try checking the environment where it occurred, searching for the error message online, assessing the system's state at the time, noting how often it happens, and identifying any recurring patterns. Effective debugging skills are crucial for improving software quality and developer productivity.

### Entering debug mode

To aid us in these debugging processes, modern **Integrated Development Environments (IDEs)** provide powerful debuggers. As a developer, if you have ever used Visual Studio, you should have already used the debugger available in Visual Studio. Now, let's delve into its most valuable features.

I've set up a project to walk through using the debugger functionality to analyze how the code is working, I'm sure you can play with what we will see in one of your current projects.

There are two common options to launch the debugger:

- Launch the entire solution in debug mode

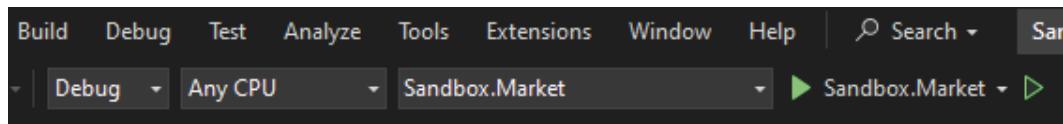


Figure 2.1 – Debug mode

- Debug unit test for the debug-specific public method

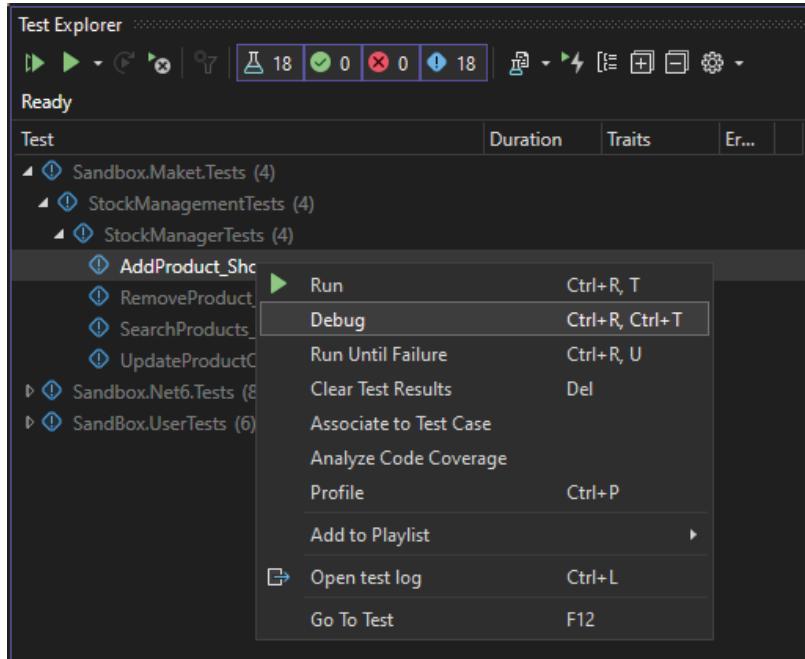


Figure 2.2 – Debugging a unit test

The second option could be a good choice to save compile time in our development process. Regardless of our choice, the first thing to do is set a breakpoint by left-clicking on the left side of our source file to navigate through the code. We will discuss the details of breakpoints later in this chapter.

Simply put, a breakpoint is an intentional stopping or pausing place in a program.

As we set the breakpoint and launch the debugger (*F5*), we can observe the behavior of the method. Here's a reminder of the three main options to navigate on our codebase:

- **Step Into (F11):** This command steps into the code, meaning it goes into the method or function that is being called on the current line. If there are nested calls, it steps into the most deeply nested one.
- **Step Over (F10):** This command steps over the current line of code, which means it executes the current line and then stops at the next line in the current method or function. It does not step into any methods or functions.
- **Step Out (Shift + F11):** This command steps out of the current method or function, which means that it will execute the rest of the method and then stop at the next line after the method call.

In debug mode, Visual Studio 2022 provides several ways that help us navigate through our code application during debugging sessions.

## Advanced debug navigation

Visual Studio 2022 allows us to play with the debugger cursor to explore different areas of our codebase, with two features, described as follows.

### **Run To Cursor (Ctrl + F10)**

The **Run To Cursor** feature allows us to quickly navigate to a specific location in our code and execute the code up to that point. To use **Run To Cursor**, we simply place the cursor on the line of code where we want execution to stop, then right-click and select **Run To Cursor** from the context menu. Alternatively, we can use the keyboard shortcut *Ctrl + F10*.

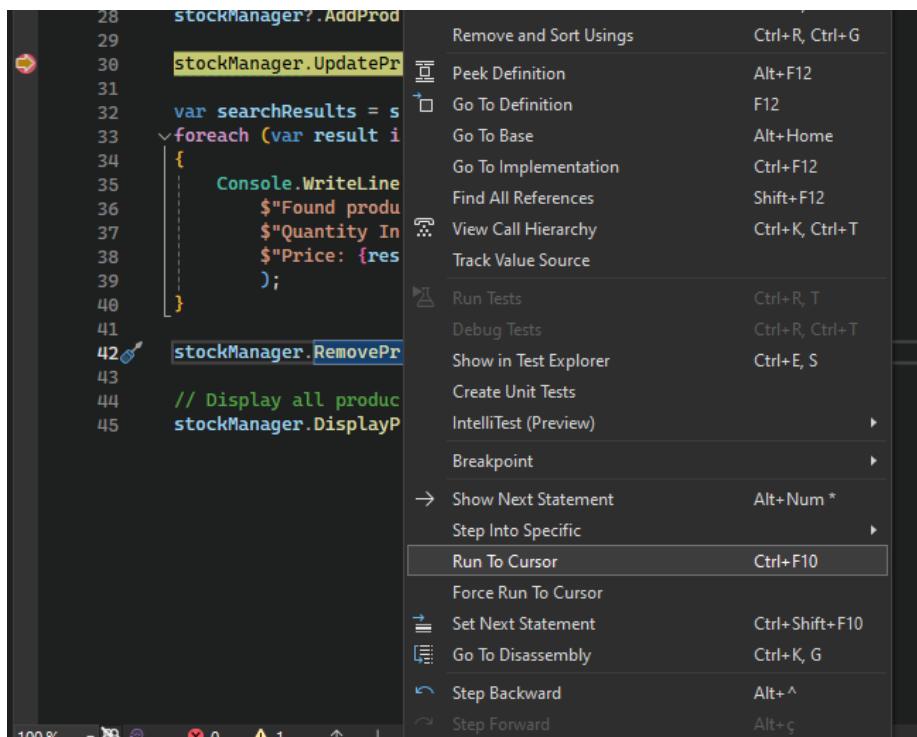


Figure 2.3 – Run to Cursor

It's important to note that the line of code where we place the cursor must be reachable, meaning it should be part of the code that gets executed when the application runs. If the line is unreachable, the **Run To Cursor** command will not work.

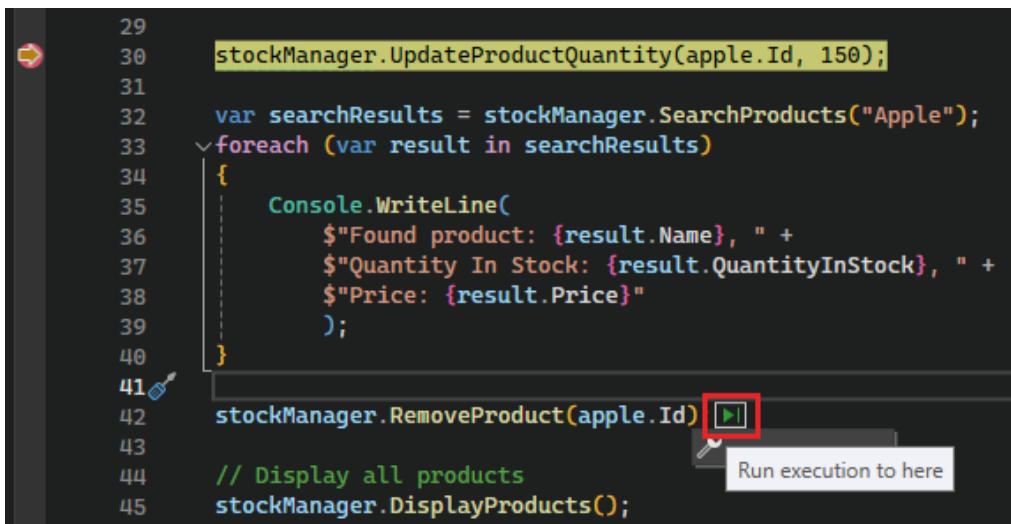
This command will run the code until it reaches the cursor's position, but if there's a breakpoint set at a location before the cursor, the debugger will stop at that breakpoint.

If we want to directly reach a line of code with the debugger, since Visual Studio 2022, we have been able to use the **Force and Run to Click** feature.

### Force And Run to Click

The **Force Run to Click** feature, available from Visual Studio 2022 onward, is a powerful tool that allows us to skip over breakpoints and any first-chance exceptions that occur during the execution of our code. This is particularly useful when we want to test updated code or focus on a specific area of our program without being interrupted by breakpoints.

To use the **Force Run to Click** feature, we need to be in a paused state in the debugger. While the debugger is paused, we can hover over a statement in the source code, press the **Shift** key, and then select **Run execution to here** (represented by a green arrow icon).



A screenshot of the Visual Studio code editor showing a C# script named 'StockManager.cs'. The code contains several lines of code related to managing product stock. A cursor is positioned on the line 'stockManager.RemoveProduct(apple.Id)'. A context menu is open at this cursor position, with a red box highlighting the 'Run execution to here' option. The menu also includes other options like 'Step Into' (with a green arrow icon), 'Step Over', and 'Step Out'.

```
29
30     stockManager.UpdateProductQuantity(apple.Id, 150);
31
32     var searchResults = stockManager.SearchProducts("Apple");
33     foreach (var result in searchResults)
34     {
35         Console.WriteLine(
36             $"Found product: {result.Name}, " +
37             $"Quantity In Stock: {result.QuantityInStock}, " +
38             $"Price: {result.Price}"
39         );
40     }
41
42     stockManager.RemoveProduct(apple.Id) ➔
43
44     // Display all products
45     stockManager.DisplayProducts();
```

Figure 2.4 – Run execution to here

When we choose this option, the application will continue to run until it reaches the cursor location, and any breakpoints and first-chance exceptions that occur during this process will be temporarily disabled.

This feature is especially convenient when we have multiple breakpoints set in our application and we want to skip them all to quickly reach a specific point in our code for testing or debugging purposes.

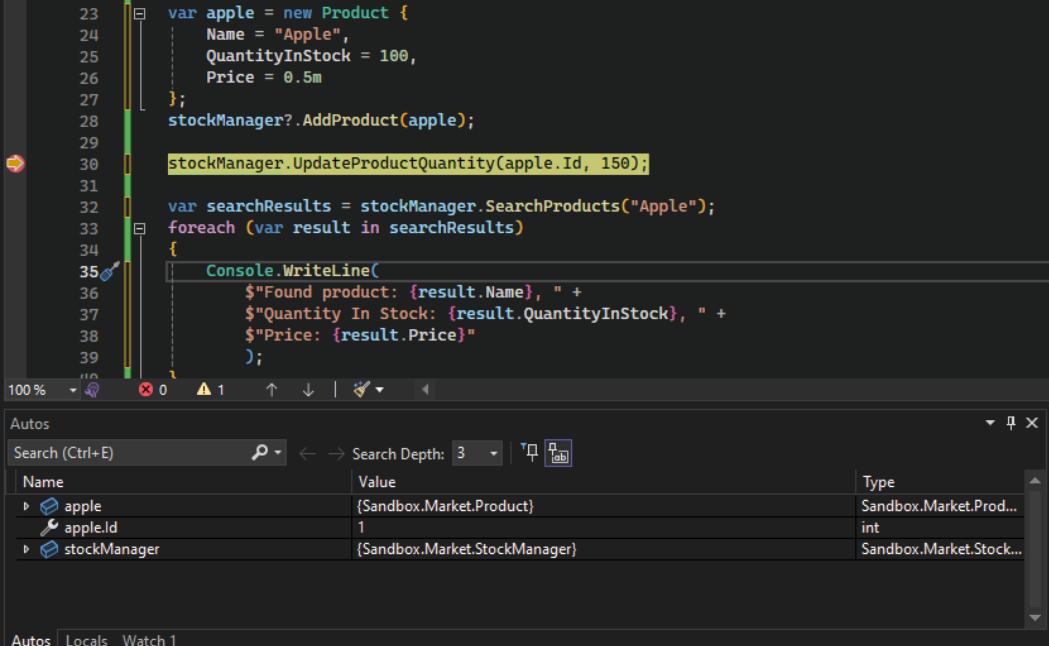
Now that we've navigated through our codebase using the debugger, we need to understand what's happening at each step. Visual Studio provides several tools to help us examine the state of our application during debugging sessions.

## Understanding debugger tools

The Visual Studio debugger offers several windows, some more recent than others, to explore our variables and objects. Let's explore these windows in detail.

### Autos window

The **Autos** window displays variables and expressions that are automatically evaluated by the debugger. It shows the values of local variables and expressions that are currently in scope. The **Autos** window is useful when we have many variables and we don't want to clutter the **Locals** window with all of them. It evaluates only the most relevant variables based on our current execution context.



The screenshot shows a code editor with C# code and an attached debugger. The code creates a product object and adds it to a stock manager, then prints its details. Below the code is the 'Autos' window, which lists variables and their values:

Name	Type
apple	Sandbox.Market.Product
apple.Id	int
stockManager	Sandbox.Market.StockManager

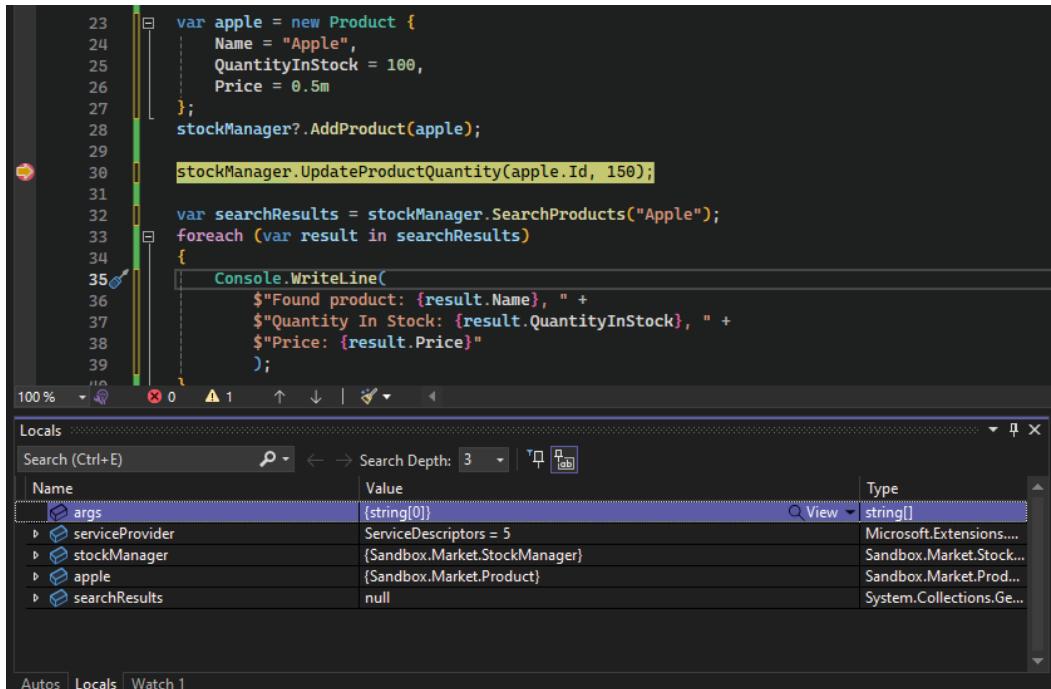
Figure 2.5 – The Autos window

In this example, we have a `int apple.Id = 1;` variable and the `apple` and `stockManager` objects in use; the **Autos** window might show just them along with their values, rather than showing every single variable in the current scope.

Next, let's take a look at the **Locals** window.

## Locals window

The **Locals** window shows all local variables in the current scope. Unlike the **Autos** window, which only shows a subset of variables, the **Locals** window lists all local variables regardless of their relevance. This can be helpful when we need to see the exact state of all local variables at a particular breakpoint.



The screenshot shows a code editor with C# code and a Locals window below it. The code is as follows:

```
23 var apple = new Product {  
24     Name = "Apple",  
25     QuantityInStock = 100,  
26     Price = 0.5m  
27 };  
28 stockManager?.AddProduct(apple);  
29  
30 stockManager.UpdateProductQuantity(apple.Id, 150);  
31  
32 var searchResults = stockManager.SearchProducts("Apple");  
33 foreach (var result in searchResults)  
34 {  
35     Console.WriteLine(  
36         $"Found product: {result.Name}, "  
37         $"Quantity In Stock: {result.QuantityInStock}, "  
38         $"Price: {result.Price}"  
39 );  
40 }
```

The Locals window displays the following variables:

Name	Type	Value
args	string[]	{string[0]}
serviceProvider	Microsoft.Extensions...	ServiceDescriptors = 5
stockManager	Sandbox.Market.Stock...	{Sandbox.Market.StockManager}
apple	Sandbox.Market.Prod...	{Sandbox.Market.Product}
searchResults	System.Collections.G...	null

Figure 2.6 – The Locals window

Here, the **Locals** window allows you to see the value of each variable at the time the code execution is paused.

Next, we'll look at the **Watch** window.

## Watch window

We use the **Watch** window to monitor specific variables or expressions. We can manually add variables or expressions to the **Watch** window, and the debugger will evaluate them whenever the execution is paused. This is particularly useful when we want to keep track of the changes in the value of a variable over time or when we want to evaluate complex expressions that involve multiple variables.

To add an expression to the **Watch** window, right-click on the code and select **Add Watch**. We can also type directly or code in the **Watch** window.

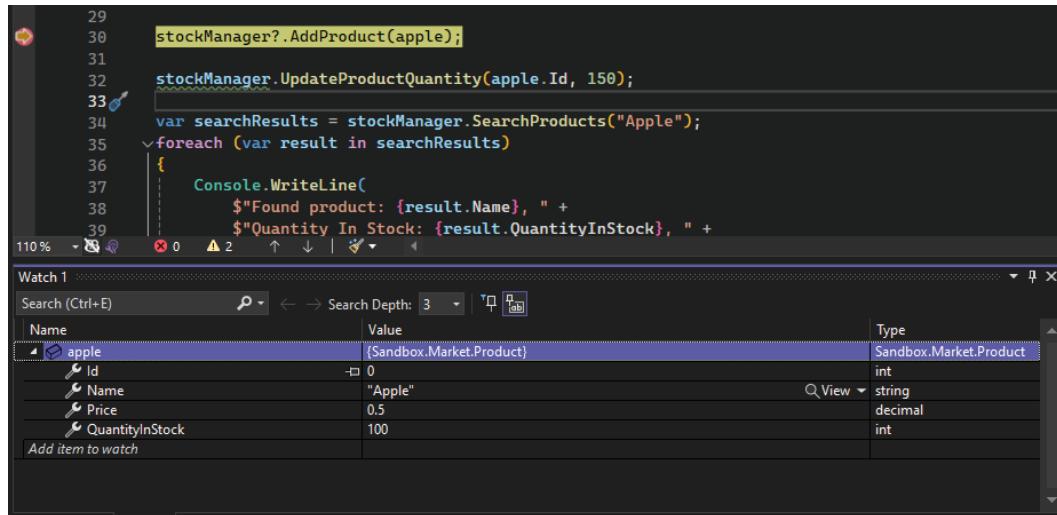


Figure 2.7 – The Watch window

The **Watch** window can also be used to modify the value of a variable during debugging, which can be helpful for testing different scenarios.

### **Call Stack** window

Now, if we enter the `UpdateProduct()` method, we can follow the sequence in the **Call Stack** window.

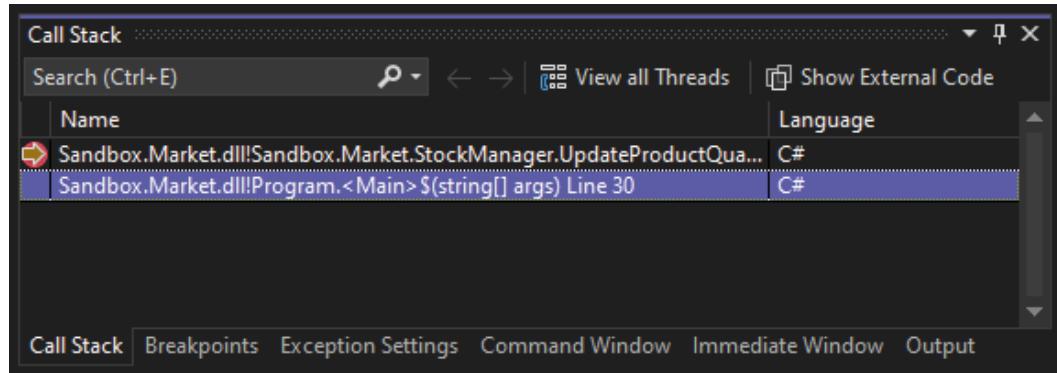


Figure 2.8 – The Call Stack window

The **Call Stack** window in Visual Studio is a critical tool for debugging as it allows us to view the sequence of function or method calls that led to the current point of execution in our program. This window is essential for understanding the flow of execution and for diagnosing issues within our code.

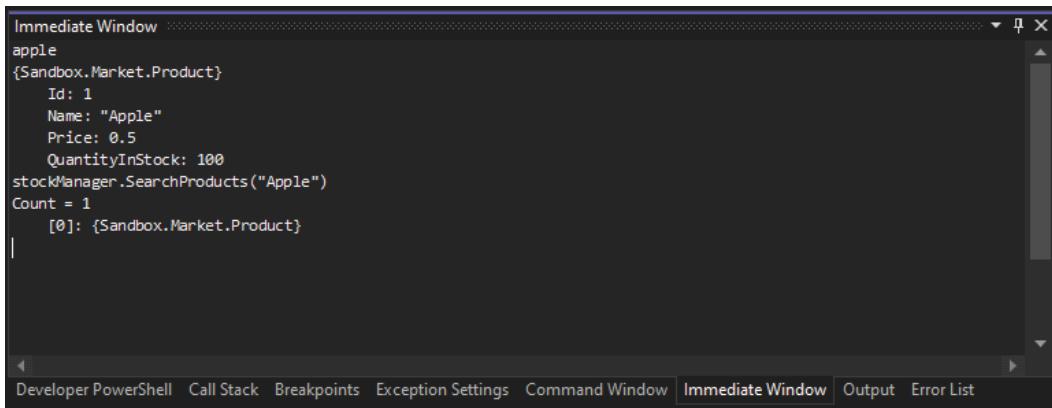
Here are some key functionalities of the **Call Stack** window in Visual Studio:

- **Viewing Call Stack:** We can see the sequence of method and function calls that led to the current point of execution in our program.
- **Switching stack frames:** We can switch to a different stack frame by right-clicking a frame in the **Call Stack** window and selecting **Switch to Frame**, or by double-clicking the frame. This allows us to inspect the code and data within that frame.
- **Disassembly code view:** To view the disassembly code for a function on the call stack, simply right-click the function and select **Go To Disassembly**.
- **Loading Symbols:** The **Call Stack** window in Visual Studio has the capability to load debugging symbols for code that currently lacks them. These symbols may include .NET or system symbols obtained from Microsoft's public symbol servers, or symbols located within a symbol path on our computer.
- **Code map integration:** Visual Studio Enterprise provides the functionality to visually map the call stack during debugging. This feature enables us to observe the current call stack in a graphical format on a new code map, which updates automatically as debugging progresses. This visual representation facilitates a more intuitive understanding of the structure and flow of our code, aiding in the identification of potential issues or areas for optimization. Please note that adding the **CodeMap** extension to Visual Studio is necessary to utilize this feature.

Finally, let us look at the **Immediate Window**.

### **Immediate window**

The **Immediate** window in Visual Studio is an old, little-known debugging tool that allows us to execute code, evaluate expressions, and print variable values on the fly during a debugging session.



```
apple
{Sandbox.Market.Product}
  Id: 1
  Name: "Apple"
  Price: 0.5
  QuantityInStock: 100
stockManager.SearchProducts("Apple")
Count = 1
[0]: {Sandbox.Market.Product}
```

The screenshot shows the Immediate Window in Visual Studio. It displays a session where a variable 'apple' is defined as a Sandbox.Market.Product object. The object has properties: Id (1), Name ("Apple"), Price (0.5), and QuantityInStock (100). A call to stockManager.SearchProducts("Apple") was made, resulting in a Count of 1, with the first result being the [0] element of the array, which is also a Sandbox.Market.Product object. The window has tabs at the bottom: Developer PowerShell, Call Stack, Breakpoints, Exception Settings, Command Window, Immediate Window (which is selected and highlighted in blue), Output, and Error List.

Figure 2.9 – The Immediate window

It is designed to help with dynamic code evaluation and the quick testing of code snippets without having to stop the execution of the application. To use the **Immediate** window, open it by going to the **Debug** menu, selecting **Windows**, and then choosing the **Immediate** window, or by pressing *Ctrl + Alt + I*. This window is particularly useful when debugging an application, as it allows us to check the value of variables, call functions, and execute statements without stepping through the code line by line.

Here are some valuable tricks for using the **Immediate** window effectively during debugging:

- **Debugging methods from the Immediate window:** We can place breakpoints in methods and call them from the **Immediate** window to debug them, even if we've already passed that point in the code. Note that we can use the **Run to a cursor location** (*Ctrl + Shift + F10*) feature to return to any line of code in the same scope and step into methods with *F11* to debug as usual.
- **Bypassing access rules:** The **Immediate** window does not enforce class accessibility rules, allowing us to call private, protected, and internal members that would not be accessible in regular code. However, IntelliSense (the built-in autocompletion tool) will still show only the public methods.
- **Evaluating expressions without side effects:** Evaluating expressions in the **Immediate** window can change variable values and call methods, potentially causing side effects. To avoid this, append **No Side Effects** (`nse`) to an expression, which will evaluate it without altering the application state.
- **Accessing special debugging variables:** The **Immediate** window can display values of special debugging variables such as `$exception` (the currently thrown exception), `$returnValue` (the return value of the currently returned method), and `$user` (the current OS user and process information).
- **Evaluating Make Object ID:** When debugging, you can **Make Object ID** for a variable in **DataTip**, which creates a unique identifier for the object. This identifier can be evaluated in the **Immediate** window at any time to see changes in the object's value or existence in memory.

Identifying bugs is an essential step in our debugging process, but the goal is to fix them. Let's explore how we can enhance productivity by fixing them on the fly with Visual Studio.

## Fixing bugs on the fly

To enhance our productivity during debugging and avoid the need to stop and start the debugger to fix a bug, Visual Studio 2022 allows us to do so on the fly with the following features.

### *Edit and Continue*

**Edit and Continue** is a feature that allows you to modify your source code while your application is in break mode. When you apply the changes, Visual Studio attempts to recompile the modified code and apply those changes to the running application. This means that you can fix bugs, add features, or experiment with code changes without interrupting the flow of your debugging session.

Here's how it works:

1. We start debugging your application.
2. We hit a breakpoint and pause the execution.
3. While paused, we can edit the code in the editor.
4. After editing, we can choose to continue execution with the new changes applied.

We can even move the yellow execution pointer back to change the execution flow and execute our edited code.

This feature is particularly useful when we find a bug during a debugging session, and we immediately know how to fix it. Instead of stopping the debugger, making the change, and then restarting the app, you can simply edit the code and continue.

### ***Hot Reload***

**Hot Reload** is a newer feature introduced in .NET 6 and supported in Visual Studio 2022. It allows us to make changes to our code while our application is running, and those changes are reflected in the running application almost instantly. This is like Edit and Continue, but with a few key differences:

- Hot Reload supports UI updates, whereas Edit and Continue does not. If we change the layout or appearance of our UI, Hot Reload will update the running app to reflect those changes.
- Hot Reload is designed to work with .NET applications, including ASP.NET Core web apps and Blazor WebAssembly apps.
- Hot Reload requires a bit more setup compared to Edit and Continue. We need to enable it in our project settings and ensure that our app is compatible with the feature. We will delve deeper into it in the next chapter.
- With Hot Reload, we can see the effects of our changes immediately, which can speed up the development process, especially when working on UI or front-end code.

Both Edit and Continue and Hot Reload are powerful tools that can save time and reduce the frustration associated with stopping and restarting our application during the debugging process. They allow us to maintain a smooth workflow and quickly iterate on our code, which can lead to more efficient problem-solving and faster development cycles.

Now that we've conquered the art of navigating the debugger, taken an overview of their windows, and performed live fixes, let's delve deeper into two powerful tools that elevate your debugging game: advanced breakpoints and data inspection. While we've already touched upon them, consider this a deep dive, equipping you to wield them with precision and unlock invaluable insights into your code's behavior.

## Advanced breakpoints and data inspections

Advanced breakpoints in Visual Studio 2022 are powerful tools that allow us to inspect and control the execution flow of their programs during debugging sessions.

### Understanding the types of breakpoints

Breakpoints in Visual Studio can be divided into several types. Those are as follows.

#### *Conditional breakpoints*

We can control when a breakpoint executes by setting conditions. That feature has been available since Visual Studio 2005. We can right-click the breakpoint symbol and select **Conditions**, or right-click in the margin next to a line of code and select **Insert Conditional Breakpoint** from the context menu. In the **Breakpoint Settings** window, select **Conditional Expression**, **Hit Count**, or **Filter**, and set the conditional expression in the text box accordingly.

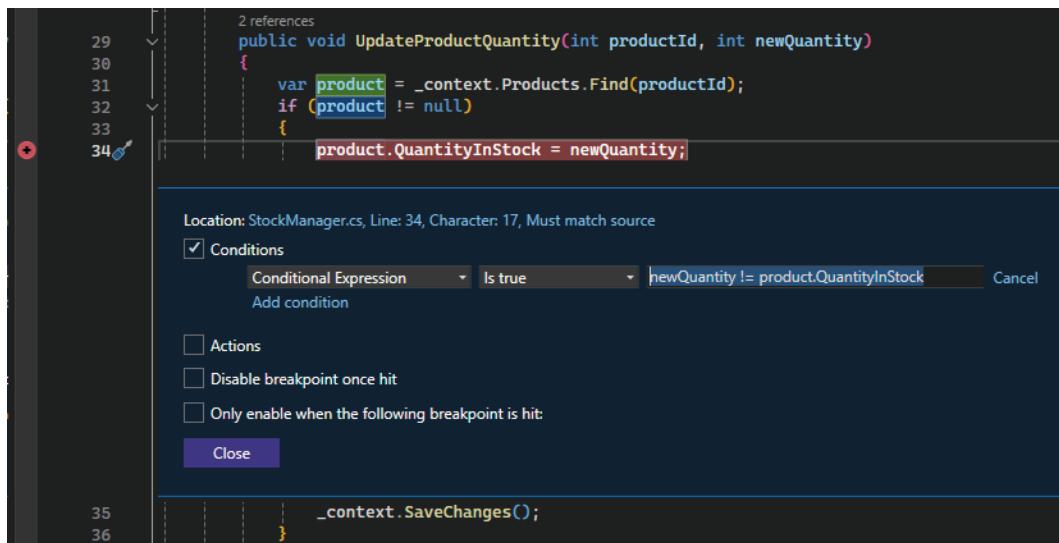


Figure 2.10 – Conditional breakpoint

In this example, with our setup, the debugger will stop if `newQuantity` is not equal to `product.QuantityStock`.

#### *Tracepoints*

Introduced with Visual Studio 2005, a **tracepoint** serves as a breakpoint variant enabling users to log information to the **Output** window based on customizable conditions as conditional breakpoints without altering or pausing the code execution. Tracepoints are compatible with managed languages such as C#, Visual Basic, F#, and native code, as well as languages such as JavaScript and Python.

To set a tracepoint, simply click on the gutter located to the left of the desired line number. Hover over the ensuing red circle and click on the gear icon to unveil the **Breakpoint Settings** window. Proceed by selecting the **Action** checkbox. This action will transform the red circle into a diamond shape, signifying the transition from a traditional breakpoint to a tracepoint.

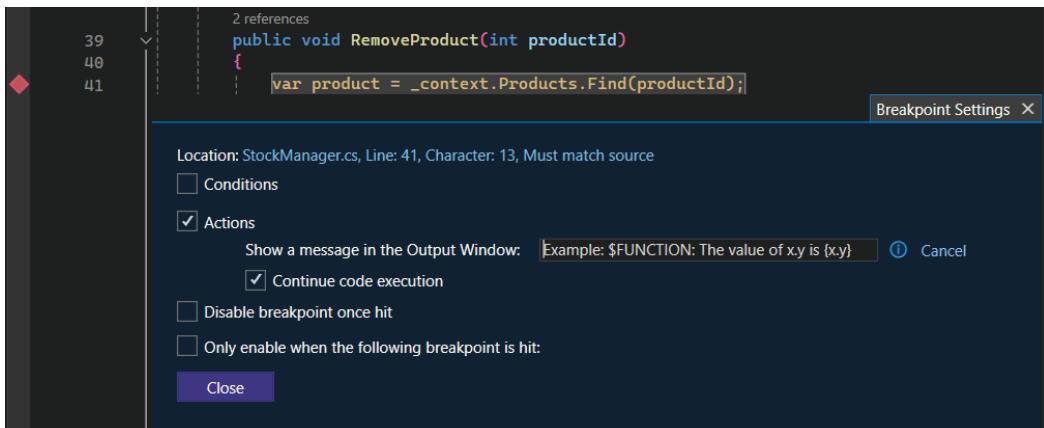


Figure 2.11 – Tracepoint

Enter the message we want to log into the **Show a message in the Output Window** text box. If we want to add conditions that determine whether our message is displayed, we could select the **Conditions** checkbox. As for conditional breakpoints, we have three choices for conditions: **Conditional Expression**, **Filter**, and **Hit Count**. Tracepoints are useful for debugging because they allow us to log information without cluttering our code with print statements or functions such as `Debug.WriteLine()`.

### **Data breakpoints**

Data breakpoints allow us, as developers, to pause execution when the value of a specific object's property changes. This feature is particularly useful for debugging scenarios where we want to monitor changes to data without having to manually step through code. Data breakpoints can be set for .NET Core 3.x or .NET 5+ projects, and they are particularly useful for tracking changes in object properties.

To set a data breakpoint in a .NET Core or .NET 5+ project, we start debugging and wait until a breakpoint is reached. Then, in the **Autos**, **Watch**, or **Locals** window, we right-click a property and select **Break when value changes** from the context menu.

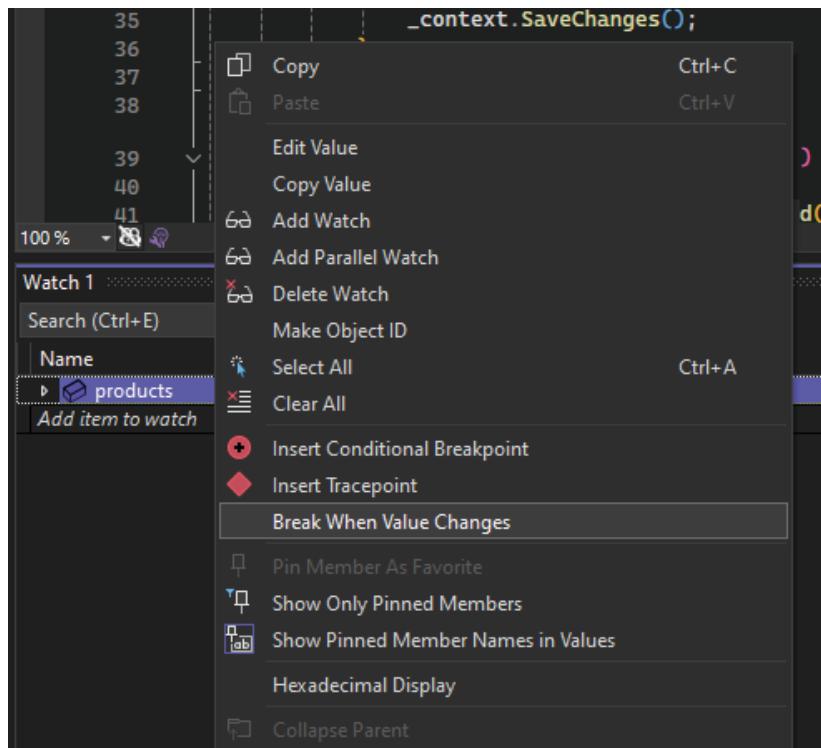


Figure 2.12 – Break When Value Changes

This will set a data breakpoint that triggers when the selected property's value changes.

Data breakpoints are subject to hardware and kernel limitations. The Windows kernel and the underlying hardware impose a maximum number of data breakpoints that can be set simultaneously. These limitations ensure that our debugging experience remains efficient and responsive.

It's important to note that data breakpoints depend on specific memory addresses, and that the address of a variable can change from one debugging session to the next. Therefore, data breakpoints are automatically disabled at the end of each debugging session. If we set a data breakpoint on a local variable, the breakpoint remains enabled when the function ends, but the memory address is no longer applicable, leading to unpredictable behavior. It's recommended to delete or disable such breakpoints before the function ends to avoid confusion.

The introduction of data breakpoints for .NET Core in Visual Studio 2019 marked the start of this feature, making it a valuable tool for us when working with .NET Core 3.x or .NET 5+.

## Function breakpoints

Since Visual Studio 2012, we have been able to set breakpoints on functions, which is useful when we know the function name but not its location, or when we have overloaded functions. To set a function breakpoint, select **Debug | New Breakpoint | Function Breakpoint**, or press *Ctrl + K*, then *B*.

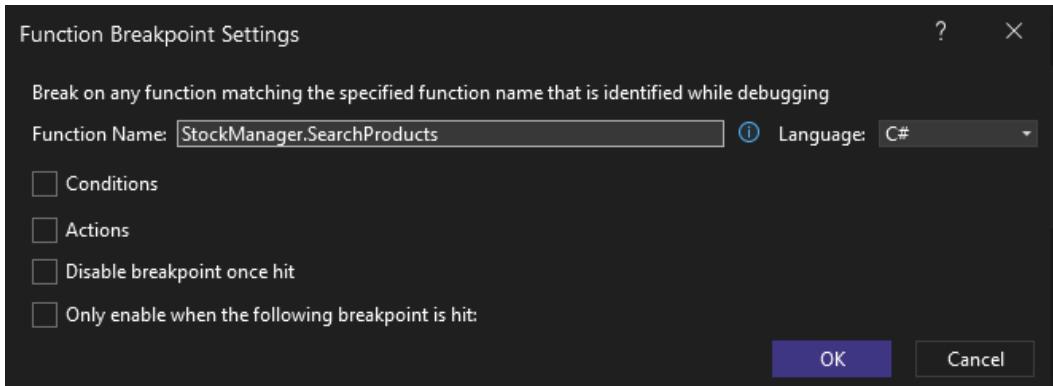


Figure 2.13 – Function breakpoints

Enter the fully qualified function name, including parameter types for overloaded functions, or use the `!` symbol to specify the module. Here, we set a breakpoint to the `SearchProducts` method of the `StockManager` class.

## Dependant breakpoints

Visual Studio 2022 introduced us to the possibility of setting up a dependent breakpoint, which is a powerful debugging feature that allows us to pause the execution of our program only when another specific breakpoint has been hit first. This is particularly useful in complex debugging scenarios, such as when we are working with multi-threaded applications or when we want to focus on debugging code in specific parts of our application.

To set up a dependent breakpoint, we first need to identify the breakpoint that our current breakpoint will depend on. Then, we hover over the breakpoint symbol, choose the settings icon, and select **Only enable when the following breakpoint is hit** in the **Breakpoint Settings** window.

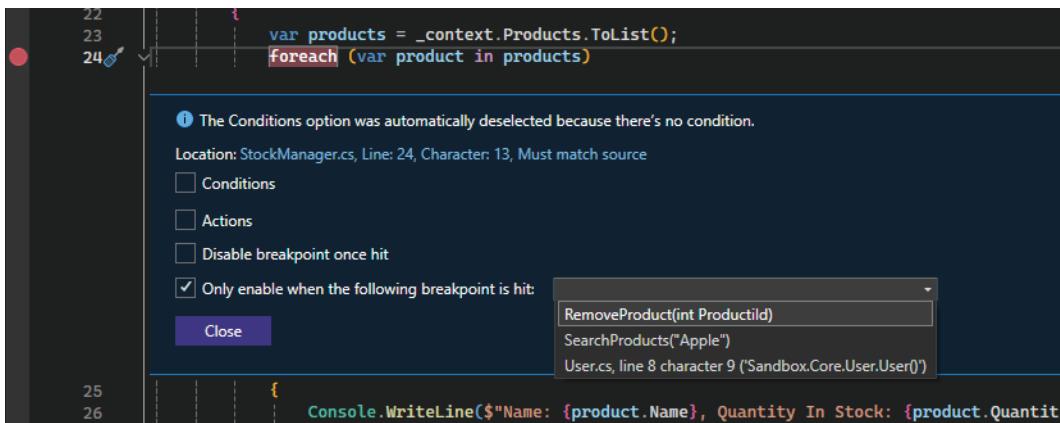


Figure 2.14 – The Only enable when the following breakpoint is hit checkbox

From the dropdown, we select the prerequisite breakpoint that we want our current breakpoint to be dependent on. After setting this up, we can close the dialog.

Another way to set a dependent breakpoint is by using the right-click context menu. We right-click in the far-left margin next to a line of code and select **Insert Dependent Breakpoint** from the context menu.

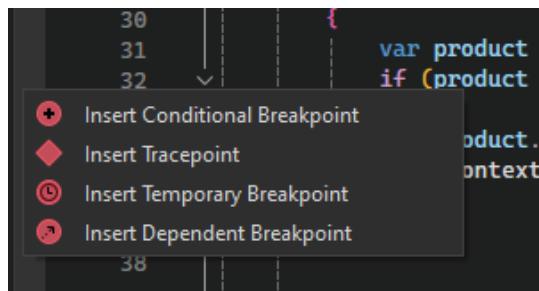


Figure 2.15 – Insert Dependent Breakpoint

It's important to note that dependent breakpoints do not work if there is only a single breakpoint in our application. If the prerequisite breakpoint is deleted, the dependent breakpoint will be converted to a normal line breakpoint.

This feature is designed to streamline our debugging process by allowing us to focus on specific parts of our code that are relevant to the debugging task at hand, thereby potentially saving time and effort during the debugging process.

## Temporary breakpoints

Temporary breakpoints in Visual Studio 2022 are a powerful debugging feature that allows us, as developers, to pause the execution of our code only once at a specific location. Once the breakpoint is hit during our debugging session, it automatically disables itself, making it ideal for scenarios wherein we want to inspect a specific condition or behavior without repeatedly hitting the same breakpoint.

To set up a temporary breakpoint, we first need to identify the breakpoint that our current breakpoint will depend on. Then, we hover over the breakpoint symbol, choose the settings icon, and select **Disable breakpoint once hit** in the **Breakpoint Settings** window. This action configures the breakpoint to automatically disable itself after it's hit during our debugging session.

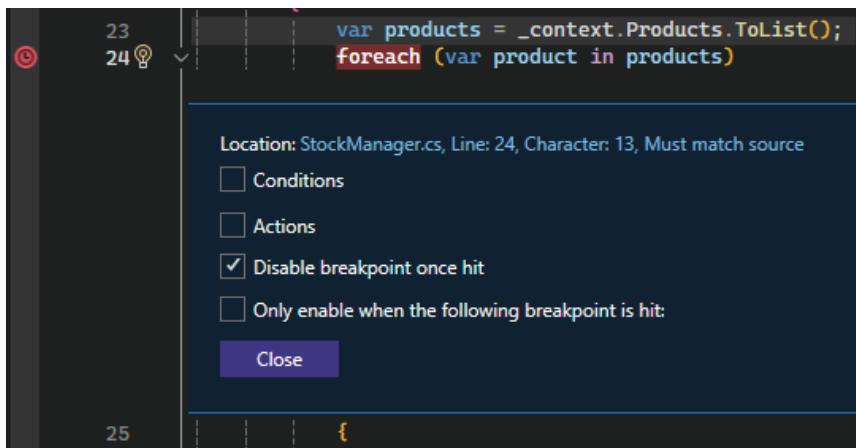


Figure 2.16 – Disable breakpoint once hit

Alternatively, we can directly set a temporary breakpoint by right-clicking in the breakpoint gutter and selecting **Insert Temporary Breakpoint** from the context menu.

The introduction of temporary breakpoints in Visual Studio 2022 enhances our debugging experience by providing us with a tool to inspect specific points in the code without the need to manually remove or disable breakpoints after they have been hit. This feature is particularly useful for validating assumptions, checking the flow of execution, or verifying the state of variables at specific moments during our debugging process.

Now that we've learned all about breakpoints, let's learn how we can organize them.

## Organizing our breakpoints

We can use labels to sort and filter breakpoints in the **Breakpoints** window. Right-click a breakpoint and select **Edit labels...** to add or change labels.

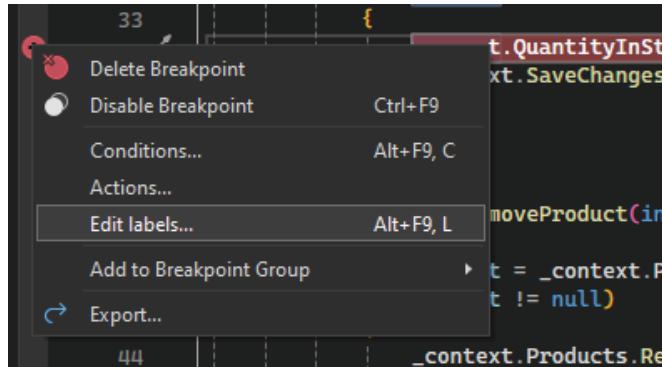


Figure 2.17 – Edit labels...

Then, in the **Breakpoints** window, we can retrieve our settled labels to filter and organize them.

Name	Labels	Condition	Hit Count
<input checked="" type="checkbox"/> Program.cs, line 30 character 1			break always
<input checked="" type="checkbox"/> RemoveProduct(int ProductId)			break always
<input checked="" type="checkbox"/> SearchProducts("Apple")			break always
<input checked="" type="checkbox"/> StockManager.cs, line 34 character 17	UpdateQuantity	when 'newQuantity != product.QuantityInStock' is true	break always
<input checked="" type="checkbox"/> User.cs, line 8 character 9			break always

Figure 2.18 – Breakpoints with labels

In this example, I labeled my conditional breakpoint **UpdateQuantity**.

Now that we know which breakpoints to use according to our needs, let's see how to inspect the data efficiently.

## Inspecting the data

As we have seen before, the **Watch** window in Visual Studio is a powerful debugging tool that allows us to monitor the values of variables and expressions during the execution of our program. It is particularly useful when we want to keep an eye on the changes in variable values as your code executes, which can be especially helpful in debugging complex logic or performance issues.

While the **Watch** window offers a dedicated space for monitoring variables, Visual Studio also empowers us with a more immediate option: **DataTips**. These handy popups appear right within the code editor, revealing variable values directly where they're used.

To leverage DataTips, simply hover our mouse over any variable name in our code. Then, a tooltip emerges, displaying the current value of that variable.

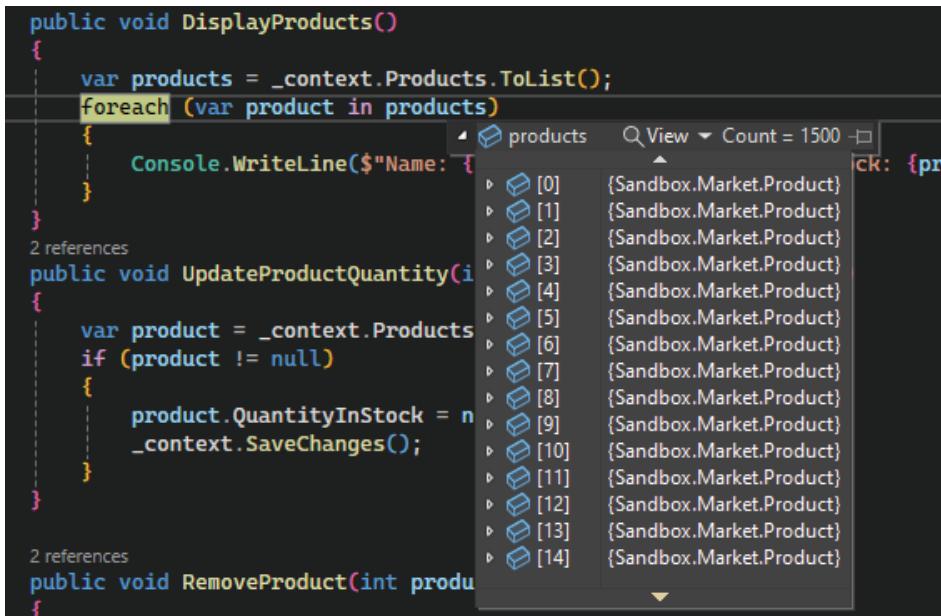


Figure 2.19 – Displaying the datatips

In certain scenarios, such as the one shown in *Figure 2.19*, it can be challenging to locate specific items of interest within a large list.

## Pinning properties

When looking at a data list with a huge list of complex data, it could be difficult to focus on the part of the data we need. The solution is to utilize the **Pinnable Properties** feature, also available in the **Watch** window. To accomplish this, we expand a variable and pin the property that interests us. For example, if we're interested in the name of the product, we can pin it for easy reference.

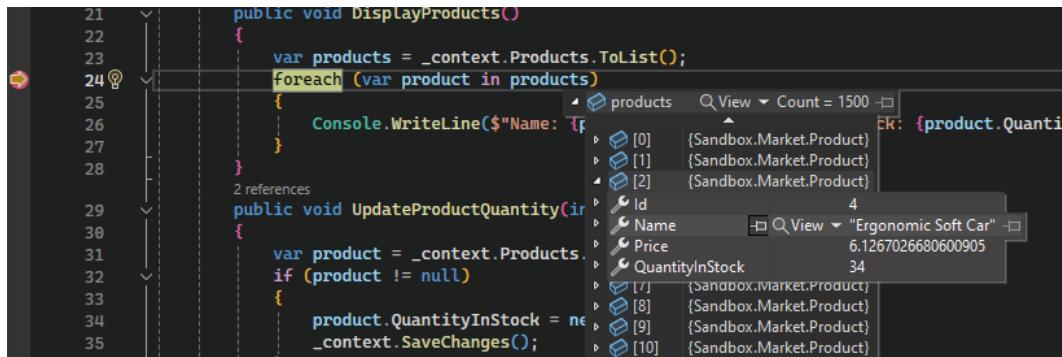


Figure 2.20 – Pinning the Name property

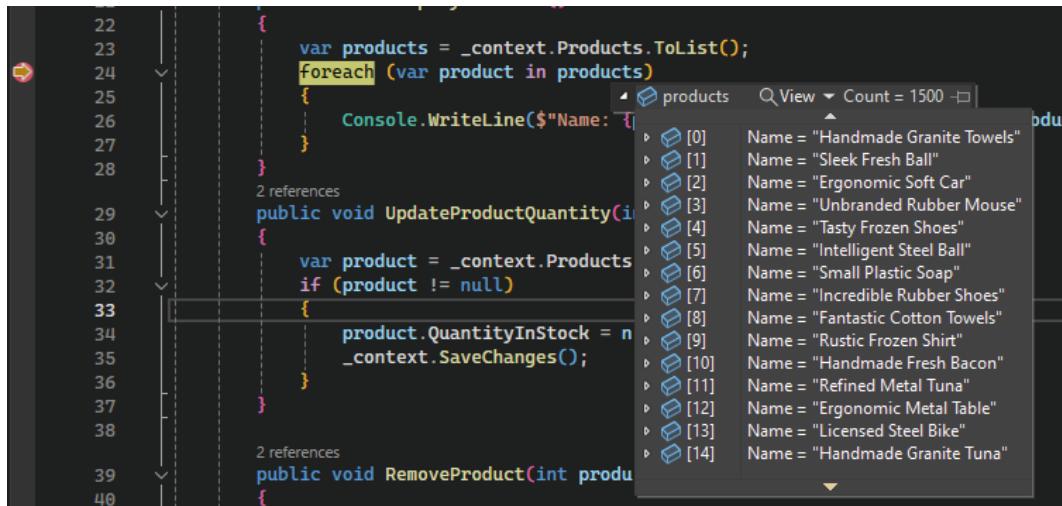


Figure 2.21 – A list of products with pinned names

Now, whenever we inspect this object with DataTips, even in the **Watch** window, the pinned property replaces the name of the object until we unpin it.

Name	Value	Type
products	Count = 1500	System.Collections.Generic.I
[0]	Name = "Sleek Cotton Fish"	Sandbox.Market.Product
[1]	Name = "Practical Rubber Soap"	Sandbox.Market.Product
[2]	Name = "Incredible Wooden Pants"	Sandbox.Market.Product
[3]	Name = "Generic Steel Mouse"	Sandbox.Market.Product
[4]	Name = "Intelligent Concrete Shirt"	Sandbox.Market.Product
[5]	Name = "Fantastic Concrete Chicken"	Sandbox.Market.Product
[6]	Name = "Generic Fresh Towels"	Sandbox.Market.Product
[7]	Name = "Refined Soft Shirt"	Sandbox.Market.Product
[8]	Name = "Small Wooden Table"	Sandbox.Market.Product
[9]	Name = "Handmade Concrete Table"	Sandbox.Market.Product
[10]	Name = "Incredible Granite Pizza"	Sandbox.Market.Product
[11]	Name = "Generic Granite Ball"	Sandbox.Market.Product

Figure 2.22 – Pinned properties in the Watch window

Now, we can easily access the product we want to inspect. Additionally, we can use the pin icon located next to our product object to keep the popup visible.

```
1 reference
public void DisplayProducts()
{
    var products = _context.Products.ToList();
    foreach (var product in products)
    {
        Console.WriteLine($"Name: {product.Name} | Id: {product.Id}");
    }
}
```

Figure 2.23 – The DataTip pin

In this example, I pin the `Name` and `Id` properties of the product so we can monitor its behavior during the execution of our loop.

### Editing value

DataTips enables us to edit the values of non-read-only variables, which can be useful for validating certain scenarios. To do this, we simply click on the value of the properties to update it.

```
ToList();
```

```
product.Name, Quantity In Stock: {product.QuantityInStock}
```

```
productId, int newQuantity)
```

```
ind(productId);
```

```
Quantity;
```

Figure 2.24 – Editing value on DataTips

DataTips also allows us to edit the values of our non-read-only variables, which can be useful for validating certain scenarios. This feature is also available in the **Watch** window, although it is not present in the **Local** or **Autos** windows.

While meticulous debugging practices within our own code are crucial, sometimes the true culprit lies hidden within external components. Understanding their behavior is the key to unlocking elusive bugs and crafting truly robust solutions. Let's explore how Visual Studio, starting from version 17.7, assists us in this endeavor.

## Elevating debugging with auto-decompilation and External Sources

Visual Studio 2022 has significantly enhanced the debugging experience with the introduction of auto-decompilation and External Sources. These features allow developers to debug external code, such as .NET libraries or **NuGet** packages, with the same ease as their own code.

### Auto-decompilation

Auto-decompilation is a feature that transforms compiled binary code into a higher-level programming language, such as C#, allowing us to examine, troubleshoot, and fix issues in external code as if they were working with their own code. This is particularly useful for debugging scenarios where the source code is not available or when we need to inspect code from a third-party library.

#### *How auto-decompilation works*

The debugger uses the **ILspy** decompiler engine to decompile external code in real time and incorporate it into the debugging session. The debugger first looks for local external sources on our machine, then uses **Source Link** or **Source Server** information from PDB files to load the source code. If these methods fail, the debugger falls back to decompiling the code.

#### *Controlling auto-decompilation*

.NET package authors could, in the past, control whether their code could be decompiled by implementing the `SuppressIldasmAttribute` attribute. Although this attribute is obsolete as of .NET 6+, Visual Studio still honors it.

#### *Limitations of auto-decompilation*

There can be issues when trying to decompile .NET assemblies, such as inaccurate variable names or unavailable variables for evaluation. These limitations can be more pronounced when debugging optimized or release assemblies.

## External Sources

The External Sources feature in Visual Studio allows us to debug and step into code from dependent NuGet or .NET libraries that are not part of our solution. This is facilitated by the addition of an External Sources node in **Solution Explorer**, which appears during debugging and shows sources for managed modules with loaded symbols containing Source Link or Source Server information.

We can find them below the top node of our solution in **Solution Explorer**:

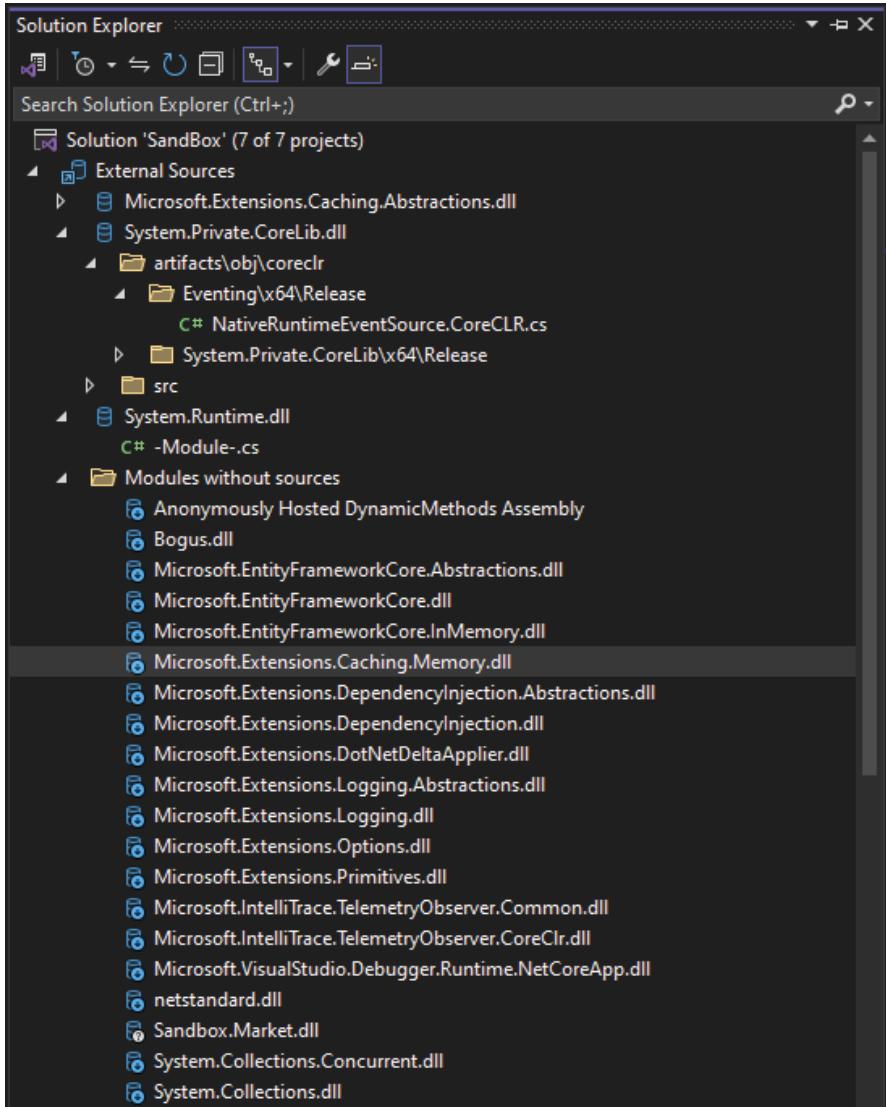


Figure 2.25 – External Sources nodes

Here's a comprehensive guide on utilizing this feature:

- **Using External Sources:** The External Sources node organizes decompiled external code modules from different call stacks, allowing us to navigate and set breakpoints within this external code. This makes debugging external code as seamless as debugging our own code.
- **Enabling External Sources:** To debug external sources, we may need to enable **Load All Modules** to allow the debugger to load symbols for all modules. We can also manually load modules from the **Modules** window if necessary from the top bar menu by going to **Debug | Window | Modules**.
- **Downloading source code:** If we double-click an item in the External Sources node, we may be prompted to download the source from the server. After accepting, we can view the source code in the editor.

Auto-decompilation and External Sources make it easier for us to debug external code, which can be particularly beneficial when dealing with complex libraries or when trying to resolve issues in third-party code. These features streamline our debugging process, reducing the time we spend on understanding and navigating external code. Package authors can prevent their code from being decompiled by using the `SuppressILDasmAttribute` attribute, ensuring that they maintain control over their intellectual property. However, debugging decompiled code may not always be as accurate as debugging source code, with potential issues such as inaccurate variable names or unavailable variables. Additionally, stepping through decompiled code may not always align with the original source code.

By leveraging these features, Visual Studio 2022 offers a more comprehensive debugging experience, enabling us to diagnose and resolve issues in their applications more effectively.

## Concurrency debugging

Multithreading is a technique whereby a process splits into multiple threads, allowing for better performance, especially on systems with multiple processors or cores. However, managing multiple threads can be challenging as they may need to access shared resources concurrently, leading to potential bugs such as deadlocks where threads are unable to progress. Debugging such issues can be difficult and time-consuming.

In this section, we will learn how to tackle this by using the **Threads** window and handling parallel debugging.

For this, we will create a simple console application calling ten simple threads. Here's the code we will use for our example :

```
for (int i = 0; i < 10; i++)
{
    CreateThreads();
```

```
}

static void CreateThreads()
{
    Dummy dummy = new Dummy();

    Thread dummyCaller = new Thread(
        new ThreadStart(dummy.Instance)
    );
    dummyCaller.Start();

    Console.WriteLine("New thread called "
        + "starting the new Dummy thread.");
}

public class Dummy
{

    static int count = 0;
    public void Instance()
    {
        Console.WriteLine(
            "Dummy.Instance is running on another
            thread.");

        int data = count++;
        Thread.Sleep(3000);
        Console.WriteLine(
            "The instance method called by the worker
            thread has ended. " + data);
    }
}
```

The code demonstrates the creation and execution of multiple threads in C#, each running an instance of a `Dummy` class's `Instance` method.

First, we will set a breakpoint on `Thread.Sleep(3000)` on the `Instance` method and launch our project in debug mode.

We can track the thread by activating the **Show Threads in Source** button on the debug toolbar.

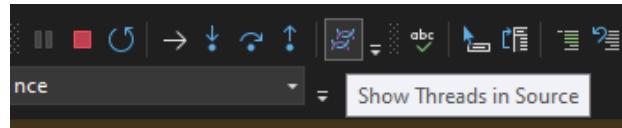


Figure 2.26 – Show Threads in Source

That will make **thread maker icons** appear on the left side of the window. When we hover over them, we can view the name and thread ID number for each stopped thread.

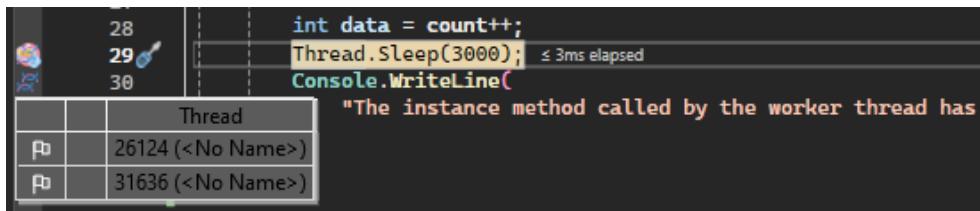


Figure 2.27 – List of threads

By right-clicking on one of the threads, we can access different types of action to navigate through the threads, such as the following:

- Flag/Unflag
- Freeze/Thaw
- Switch To Thread

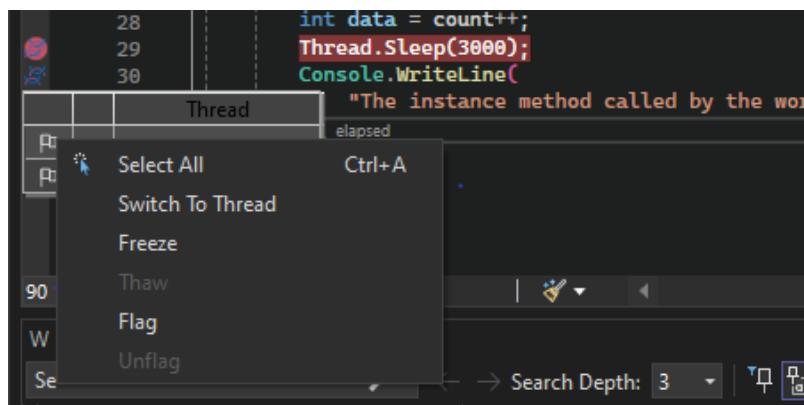


Figure 2.28 – Manipulating threads

When we flag threads, it enables us to focus on one thread while ignoring the others.

To resolve concurrency issues and control the order in which threads perform work, we must use the **Freeze** and **Thraw** features, allowing us to suspend and resume threads.

Additionally, **Switch To Thread** allows us to jump from one thread to another for step-by-step debugging.

Regarding the common debugging process, Visual Studio offers a **Watch** window named **Parallel Watch**. We access it from the top bar menu by clicking on **Debug | Window | Parallel Watch | Parallel Watch 1**.

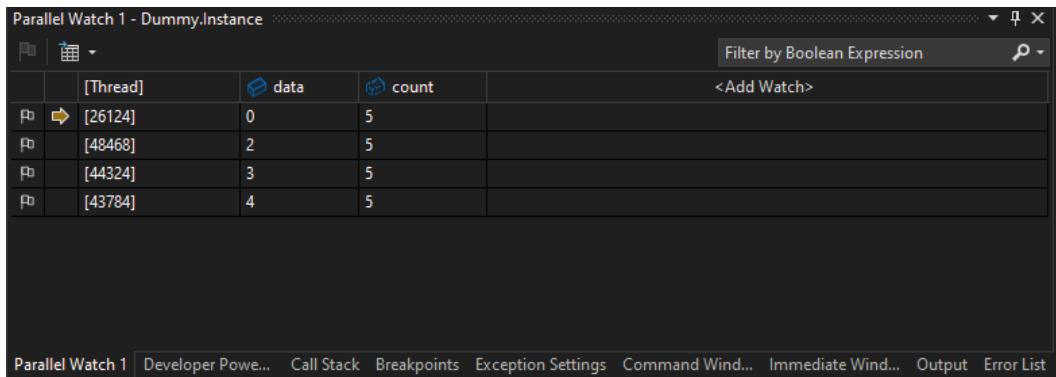


Figure 2.29 – Parallel Watch

Here, we can add data to watch by entering their names in the **<Add Watch>** cell. Additionally, we can easily switch to a thread by double-clicking on the thread line we are interested in. Right-clicking allows us to access the **Flag** and **Freeze** features. The flag button in the top left corner enables us to filter and display only the flagged threads.

Another view offered by Visual Studio to efficiently track threads is the **Parallel Stacks** window. To access it, we use the debug menu, which you can reach by clicking **Debug | Window | Parallel Stacks**.

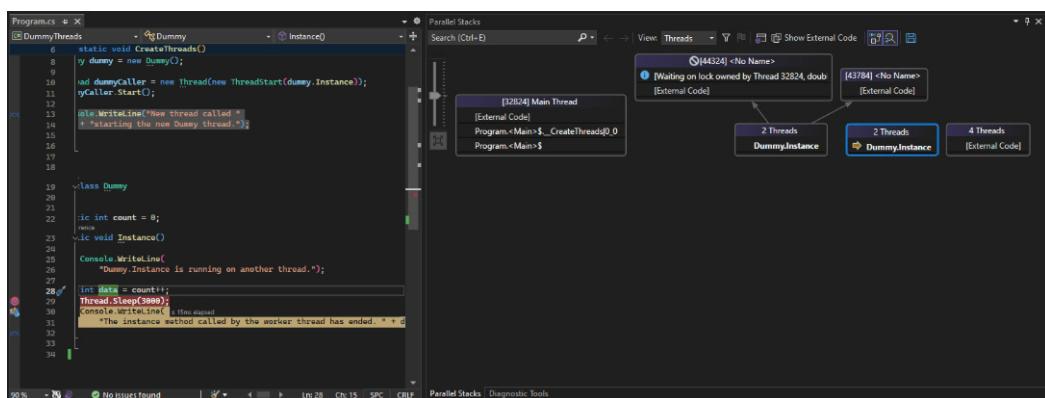


Figure 2.30 – Parallel Stacks

This window provides both a **Thread** and a **Tasks** view, showing the call stack information for each thread. The current thread is indicated by a yellow arrow, allowing us to easily follow the path of each thread. This information can also be viewed in a list format in the **Threads** window by going to **Debug | Windows | Threads**.

Finally, we can utilize conditional breakpoints, which allow us to set breakpoints based on various conditions, such as the thread name or ID. This feature is particularly handy when setting conditions on data unique to each thread. It's a common approach during debugging, especially when focusing on specific data values rather than individual threads.

Now that we have gone on a journey exploring the techniques and tools of Visual Studio 2022 concerning debugging, let's see how we can resolve the famous "it works on my laptop" issues by leveraging the remote debugging feature.

## Remote debugging

Remote debugging is a powerful feature in Visual Studio that allows us to debug applications running on different machines, devices, or environments. This is particularly useful when we need to troubleshoot issues that occur in a production environment or on a different platform that we don't have access to locally.

To leverage this functionality effectively, we need to ensure that Visual Studio is installed on both the local and remote machines. We need to configure the remote machine by installing the compatible Visual Studio version and allowing remote connections via the firewall settings.

On the local machine, we enable remote debugging in Visual Studio's settings by specifying the remote machine's address and credentials, if necessary. To do so, we go to the project properties of the application we want to debug and navigate to the **Debug** tab.

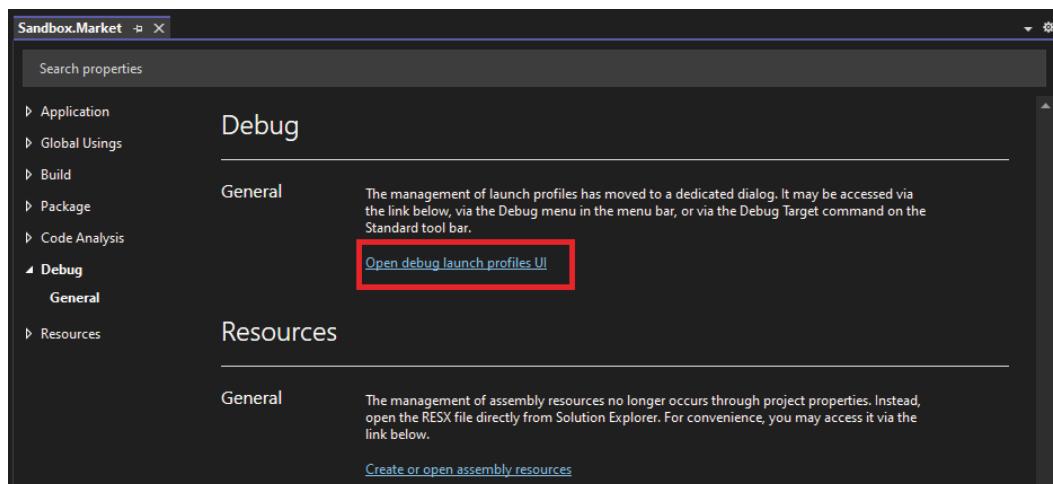


Figure 2.31 – The Project properties | Debug tab

Here, we can click on the **Open debug launch profiles UI** link. Once in the **Launch Profiles** window, we can check the **Use remote machine** checkbox that displays the two more fields that allow us to configure the remote connection:

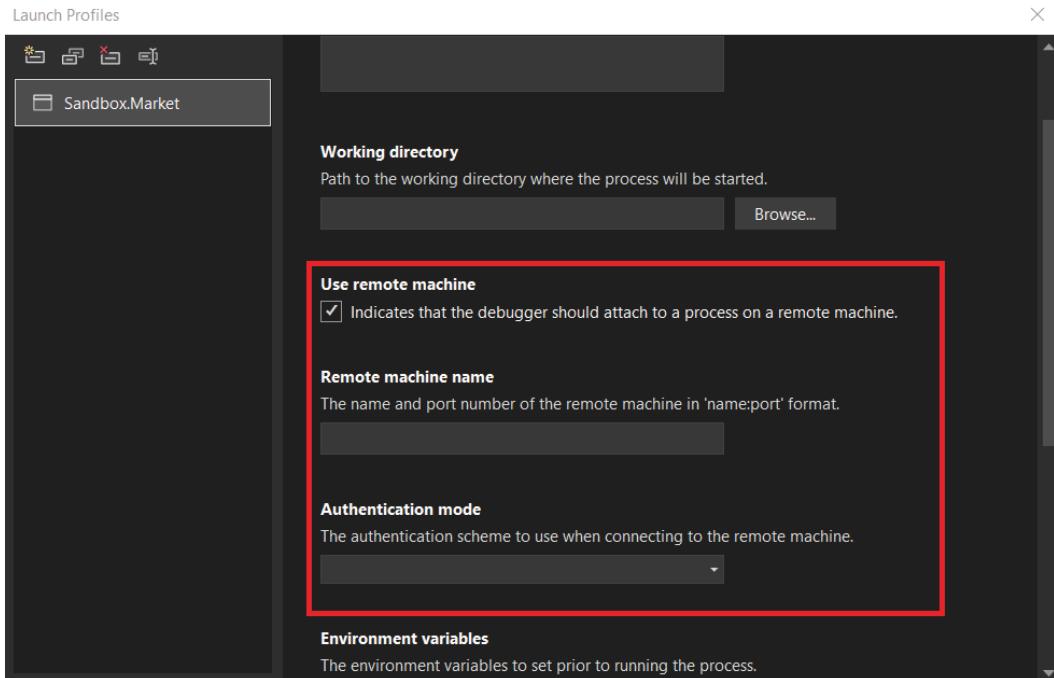


Figure 2.32 – The Use remote machine checkbox

In a critical situation wherein debugging a web application on a live production server, it's becomes essential to be able to access to the debugger on production server.

Deploy the application to the production server, ensuring that all necessary components are installed. After that, we need to configure Visual Studio on the server for remote debugging and adjust the firewall settings accordingly. In Visual Studio on the local machine, we attach to the remote process by entering the remote machine's address and credentials.

Then we can debug the application, select the process for the web application, and debug as usual, including setting breakpoints and inspecting variables. Finally, we can reproduce the issue while debugging, fix it once it has been identified, and deploy the updated application.

Remote debugging facilitates diagnosing issues in production-like environments without disrupting live traffic. Accessing production data aids in understanding the issue's context. Security measures should be implemented to safeguard sensitive information. Be mindful of performance impacts, scheduling debugging during off-peak times or in staging environments. Utilizing remote debugging enhances

application reliability by addressing issues in environments inaccessible through traditional means, bolstering overall quality and stability.

By using remote debugging, we can effectively diagnose and fix issues that occur in environments where we don't have direct access to the code or data, thereby improving the reliability and quality of our applications.

## Summary

This chapter delved into the powerful world of advanced debugging strategies in Visual Studio 2022, equipping you with the skills to tackle even the most complex code effectively. We explored various debugging tools and techniques that are essential for identifying and resolving issues in your code.

We began by mastering the art of navigating your code, where we focused on breakpoints, step execution modes, and the call stack, all of which are crucial for pinpointing the source of errors. The chapter also introduced the concept of live code modification, allowing you to dynamically fix bugs and experiment with code changes on the fly through features like Edit and Continue and Hot Reload. We didn't stop at debugging your own code; we also explored the advantages of auto-decompilation and External Sources, which enable seamless debugging of external libraries and NuGet packages. Furthermore, the chapter covered remote debugging, a technique that allows you to diagnose and fix issues in production-like environments without impacting live traffic. By mastering these advanced debugging techniques, you can resolve complex bugs more quickly, thereby reducing development time and frustration.

In the next chapter, we'll continue our journey according to the TDD cycle seen in the first chapter. After fixing our code with debugging, we'll delve into the art of shaping our code into its best form. We'll be embarking on a journey of advanced code analysis and refactoring in Visual Studio 2022.

# 3

## Advanced Code Analysis and Refactoring

This chapter elevates our skills by focusing on advanced code analysis and refactoring techniques in Visual Studio 2022.

Understanding the fundamentals of code analysis, particularly through the lens of static code analysis powered by Roslyn, lays a robust foundation for identifying vulnerabilities and ensuring code correctness. Moreover, with built-in IntelliCode, which harnesses machine learning to optimize code bases, you'll learn how to enhance maintainability and scalability effectively.

Beyond these foundational concepts, we'll also delve into the importance of code metrics, providing invaluable insights into evaluating the maintainability and security postures of your projects. By using these metrics, you'll gain the strategic foresight required to make informed decisions and enact targeted enhancements to your code base.

Throughout this chapter, I'll illustrate these concepts with real-world code review refactoring case studies. By exploring the practical applications of Visual Studio tools, you'll not only grasp theoretical concepts but also understand how to implement them effectively in your day-to-day development workflows.

By the end of this chapter, you'll emerge equipped with the knowledge and required tools to elevate the quality of your code, enhance security practices, and streamline maintenance efforts—all within the familiar and powerful environment of Visual Studio.

In this chapter, we're going to cover the following main topics:

- Understanding code analysis in Visual Studio
- Utilizing static code analysis for quality assurance and security
- Leveraging IntelliCode for code refactoring
- Using code metrics for maintainability and security
- Refactoring case studies

## Technical requirements

While writing this chapter, I used the following version of Visual Studio:

- Visual Studio Enterprise 2022, version 17.12.0
- Preview 1.0

The code files for this chapter can be found at <https://github.com/PacktPublishing/Mastering-Visual-Studio-2022/tree/main/ch03>

## Understanding code analysis in Visual Studio

First, let's discuss how Visual Studio and .NET work to analyze our code.

**Code analysis** in Visual Studio 2022 is designed to help us improve the quality of our code. It provides several tools and metrics to analyze and enhance code maintainability, readability, and performance. By identifying potential issues and suggesting improvements, it aims to streamline the development process, reduce bugs, and improve overall code quality.

Visual Studio 2022 can perform code analysis in two primary ways:

- **Legacy analysis (FxCop static analysis):** This method analyzes compiled code to identify issues. It's an old approach that only checks the code after it has been compiled.
- **.NET Compiler Platform-based code analyzers:** These are modern analyzers that analyze your code live as you type. They are dynamic and can provide real-time feedback, making it easier to catch and fix issues early in the development process.

Now, let's delve into the .NET Compiler Platform, also known as **Roslyn**, which revolutionizes code analysis capabilities in Visual Studio 2022. Roslyn is a platform that exposes the C# and Visual Basic compilers' code analysis capabilities to developers. It provides a set of APIs that allow the creation of tools and applications focused on code analysis, refactoring, and transformation.

Here's some benefit of Roslyn:

- **Rich language support:** Roslyn supports the C# and Visual Basic languages, providing comprehensive language features and syntax support for building custom developer tools and extensions.
- **Programmatic code manipulation:** With Roslyn, we can programmatically analyze, refactor, and generate code using .NET APIs, empowering them to automate repetitive tasks and improve code quality and consistency.
- **IDE extensibility:** Roslyn enables the development of custom IDE extensions and tools that enhance the functionality of Visual Studio and other .NET IDEs, extending their capabilities to support specialized workflows and development scenarios.

- **Open source community:** Roslyn is an open source project hosted on GitHub that fosters collaboration and contributions from the developer community. Developers can contribute enhancements, bug fixes, and new features to the Roslyn code base, driving the innovation and evolution of the platform.

Roslyn significantly reduces the difficulty of creating tools and applications focused on code, paving the way for innovation in various domains, including meta-programming, generating and transforming code, integrating interactive features into C# and Visual Basic languages, and embedding these languages into specialized domains.

## How Roslyn works

Roslyn revolutionizes the traditional compiler structure by decomposing it into distinct components. Moreover, it provides access to each phase of the compiler pipeline through APIs that mirror the compiler's internal processes.

Here's an illustration of how this mirroring is organized:

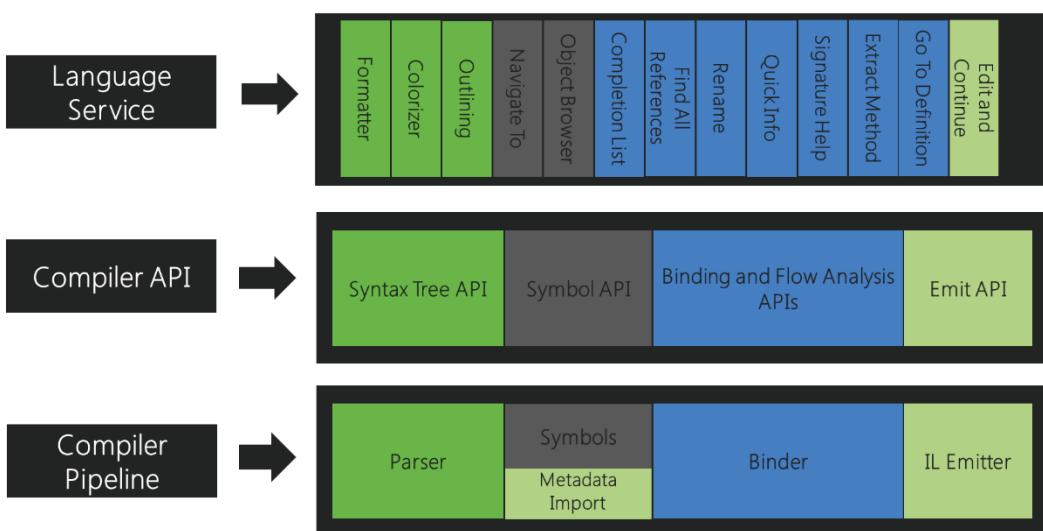


Figure 3.1 – Compiler API and Complier Pipeline

The compiler API and compiler pipeline provide access to several key phases:

- **Parse Phase:** Source code undergoes tokenization and parsing, resulting in a syntax tree that adheres to the language's grammar.
- **Declaration Phase:** Declarations from both source code and imported metadata are analyzed to construct named symbols. These symbols are organized into a hierarchical symbol table.

- **Bind Phase:** Identifiers within the code are matched to symbols, generating a semantic model that reflects the compiler's analysis.
- **Emit Phase:** The compiler accumulates all relevant information and emits it as an assembly. This assembly is represented through an API that produces **Intermediate Language (IL)** byte code.

This modular approach grants comprehensive access to information at each stage of the compiler pipeline. It empowers developers with rich capabilities for code analysis and manipulation. The availability of three-phase compiler APIs (syntax trees, semantic model, and workspace level) is particularly instrumental in facilitating robust code analysis.

First, the **syntax trees** serve as a foundational representation of the structure of source code, encompassing both lexical and syntactic elements. They are pivotal in various stages of software development, including compilation, code analysis, binding, refactoring, and IDE functionalities. Offering a complete reflection of the source information, syntax trees maintain fidelity with the original text, ensuring immutability and thread safety. This facilitates the natural manipulation of source code without direct textual edits.

Moving to semantics, the **semantic model** delves deeper into the meaning behind the code. It captures all semantic details within a single source file, shedding light on symbol references, expression types, diagnostics (errors and warnings), variable flow within source regions, and other nuanced aspects. By encapsulating language rules, this model enables clear distinctions between program elements in source code and those from precompiled libraries.

At the workspace level, a central hub organizes information across entire solutions, streamlining code analysis and refactoring processes. This layer consolidates project data into a cohesive object model, offering seamless access to compiler layer object models without the need for file parsing, configuration adjustments, or inter-project dependency management. Crucially, it underpins the development of robust code analysis and refactoring tools within integrated development environments such as Visual Studio.

## How does Visual Studio 2022 use Roslyn?

Visual Studio 2022 uses the .NET Compiler Platform (Roslyn) to analyze and refactor C# or Visual Basic code in real time as developers write code. This process is integrated into the development environment, providing immediate feedback on code style, quality, maintainability, design, and other issues. In the subsequent sections, we'll learn how Visual Studio 2022 utilizes Roslyn for analysis and refactoring.

### Analyzing with Roslyn analyzers

Visual Studio 2022 includes built-in code style analyzers (IDExxxx, e.g., IDE0001) and code quality analyzers (CAxxxx, e.g., CA1822) that inspect your code during design time in all open files. These analyzers are part of the .NET 5 SDK and are enabled by default.

We can install external analyzers, such as **StyleCop**, **Roslynator**, **XUnit Analyzers**, and **SonarAnalyzer**, as NuGet packages or Visual Studio extensions. These analyzers extend the built-in capabilities, allowing more specialized checks and rules.

Furthermore, we can create our own custom analyzer using the **Analyzer with Code Fix** template, which includes a VSIX extension and a separate project for the analyzer. This can be deployed via NuGet. That allows us to create our own custom analyzer. The analyzer can show a squiggly line below the matching code and an entry in the error list, with optional code fixes. This approach allows a more integrated development experience, with feedback and fixes available directly within the IDE.

For scenarios where analysis does not require integration with Visual Studio, such as on a build server, we can use the **Standalone Code Analysis Tool** template. This tool opens a solution workspace and analyzes projects without requiring VSIX extensions or NuGet packages to be installed in the project or IDE.

### ***Refactoring with Roslyn***

Roslyn's Syntax API parses C# code into a tree of nodes (class nodes, method nodes, etc.), enabling detailed manipulation of the code structure. This API is crucial for automating code refactoring, such as automating the migration of applications to .NET Core.

The **SyntaxEditor** class is used to apply changes to the code tree. It ensures that when a node is replaced or deleted, all its child nodes are updated accordingly, avoiding contention and exceptions. This is particularly useful for refactoring scripts that need to modify the code based on certain patterns or rules.

The **Code Refactoring** template in Visual Studio allows the creation of a VSIX extension that integrates with the **Quick Actions** menu in the IDE. This enables developers to apply quick code fixes directly from the editor without needing to define additional analyzer IDs or show entries in the error list.

Now that we have learned how Visual Studio uses Roslyn to help us analyze and refactor our code, let's dive deeper into the use of static code analysis for quality assurance and security.

## **Utilizing static code analysis for quality assurance and security**

The built-in **code quality analysis** feature is enabled by default in Visual Studio for projects that target .NET 5 or later. However, we can enable it for our older .NET project adding and setting to `true` the `EnableNetAnalyzers` property in the `.csproj` file.

Static code analysis fosters a culture of continuous improvement and collaboration within development teams. By providing actionable insights and recommendations, these tools facilitate constructive code reviews, foster knowledge sharing, and ultimately elevate the skill and proficiency of developers.

## Understanding how to use code analysis in Visual Studio

The code quality analysis will inspect our codebase for security, performance, design, and other potential areas for improvement. By default, the analysis runs automatically, so we can see errors, warnings, and information directly when we are typing through squiggles under our code or on the error list windows.

If we write code like the one shown in *Figure 3.2*, we will see squiggles appear:

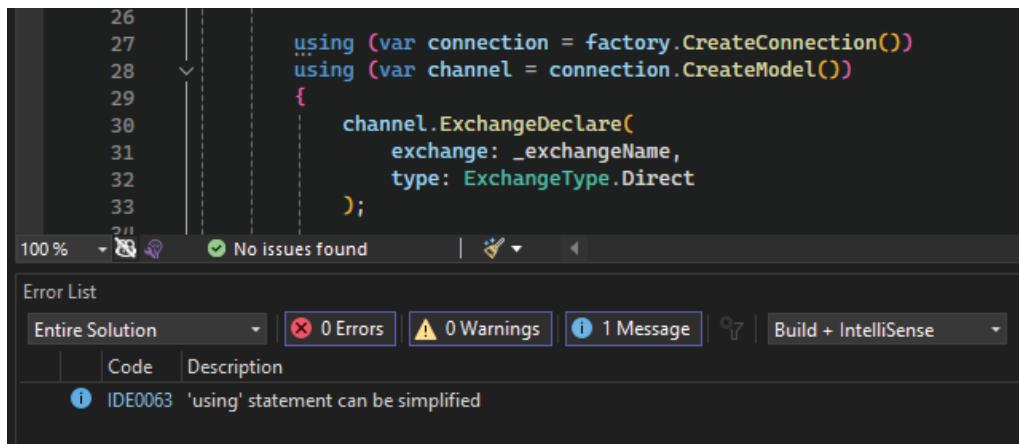


Figure 3.2 – Information squiggles

In this example, the analyzer informs us that our `using` statement can be simplified according to the feature introduced by C# 8.0.

The analysis rules are organized into categories: Design, Documentation, Globalization, Portability and Interoperability, Maintainability, Naming, Performance, Reliability, Security, Style, and Usage. You can find detailed information about each rule category in the Microsoft documentation: <https://learn.microsoft.com/en-us/dotnet/fundamentals/code-analysis/categories>.

Each rule has a severity level that determines its behavior:

- **Default:** This is the default severity level for rules that are enabled but do not have a specific severity set. It typically means that the rule is active and will report issues, but the exact behavior (e.g., whether it's treated as a warning, error, or suggestion) depends on the rule's configuration or the default behavior of the tool.
  - **Error:** When a rule is set to the Error severity level, any violation of the rule will be reported as an error. This means that the code will not be compiled until the issue is resolved. Errors are typically used for critical issues that must be fixed before the code can be considered correct.
  - **Warning:** A rule set to Warning will report violations as warnings. Warnings do not prevent the code from compiling, but they are typically used to indicate potential issues that should be addressed to improve code quality or maintainability. Warnings can be configured to be treated

as errors in certain build configurations, allowing developers to enforce stricter standards for production code.

- **Suggestion:** Rules with the Suggestion severity level report issues that are not critical but could improve the code. These rules are typically used for stylistic issues or practices that are not required but are recommended. Suggestions are often used in conjunction with code fixes that automatically apply the recommended changes, making it easier for developers to improve their code quality.
- **Silent:** A rule set to Silent will not report any issues, effectively disabling the rule. This is useful for rules that are not relevant to your project or when you want to temporarily disable a rule without removing it from your configuration.
- **None:** This severity level is like Silent but is used to explicitly indicate that a rule should not be applied. It's a way to make the intention clear that the rule is intentionally disabled.

Let's see how we can use the level of severity to enhance the quality of our project.

## Adjusting the level of severity

We can adjust the severity level of rules by right-clicking on the suggestion through the **Error List** and selecting **Set Severity**.

This allows us to prioritize and customize how Visual Studio presents and addresses potential issues identified by the rules. Depending on the project's requirements and development context, adjusting the severity levels can focus attention on critical issues while minimizing distractions from less impactful ones.

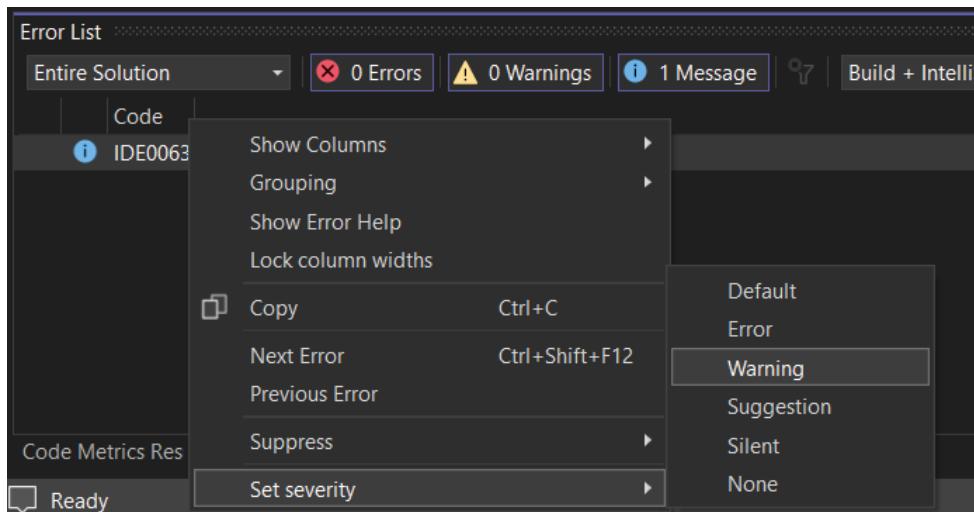
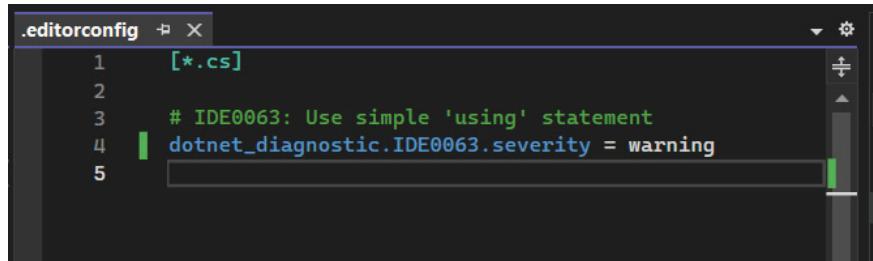


Figure 3.3 – The Set severity option

This action generates an `.editorconfig` file in our solution's root containing the overridden severity rule, which we can share with our team to ensure consistent coding practices.



```
.editorconfig
[*.cs]
# IDE0063: Use simple 'using' statement
dotnet_diagnostic.IDE0063.severity = warning
```

Figure 3.4 – `.editorconfig` file generated

An alternative method of adjusting the severity level is through the *light bulb* context menu, which also offers code fixes through Quick Actions.

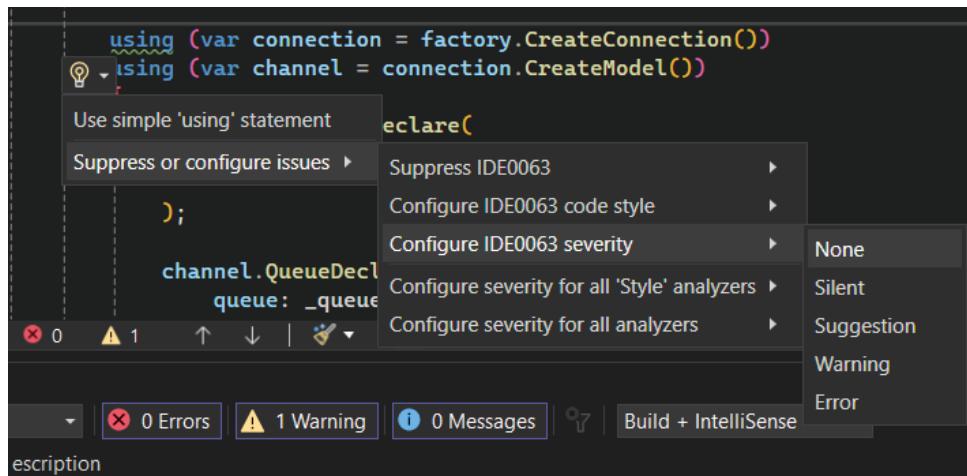


Figure 3.5 – Setting the severity using the light bulb

During a code review, we might encounter practices that could be improved, as shown in *Figure 3.6*.

```
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
```

```
private int Do(int a, bool b)
{
    if (a > 0)
    {
        if (b)
        {
            if (a == 42)
            {
                return -1;
            }
            return 1;
        }
        return 2;
    }
    return 0;
}
```

Figure 3.6 – Pyramid code

For instance, a pyramid code structure might not trigger a squiggle, but the IDE might suggest simplifying an `if` statement (IDE0046). If you want to enforce a specific coding standard, you can set the severity level of this rule to **Error**.

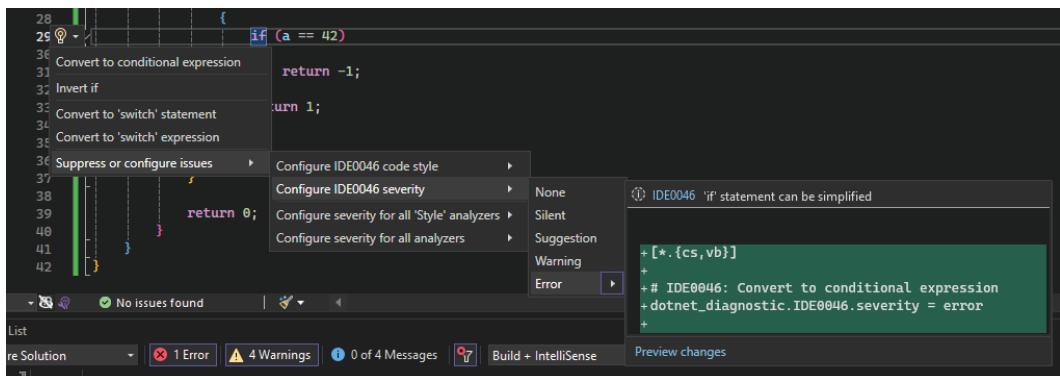


Figure 3.7 – Change the severity level of IDE0046

Likewise, when we and our teammates encounter this rule, we are forced to refactor our code. We can notice that Visual Studio provides four code fixes for this Rule:

- **Convert to a conditional expression:** This fix transforms an `if-else` statement into a conditional (ternary) expression.
- **Invert if:** This fix inverts the condition of an `if` statement and swaps the code blocks.
- **Convert to a switch statement:** This fix is applicable when you have multiple conditions based on the same variable or expression. It converts an `if-else` chain into a `switch` statement, which is more readable and maintainable for such case, like handling multiple conditions based on the same variable.
- **Convert to a switch expression:** Like converting to a `switch` statement, this fix converts an `if-else` chain into a `switch` expression, which is a more concise and functional way to handle multiple conditions. It's available in C# 8.0 and later.

For better readability, I prefer to use `Invert if` in this example, but the IDE0046 rule force is used to turn it into a conditional expression. Keep in mind that this is just an example, and we can adapt the level of severity of the analysis rules in our solution according to our needs.

As we have seen, adjusting the severity level generates a `.editorConfig` file containing our custom configurations. Let's explore how we can generate such a file to easily share our Visual Studio settings with our team.

## Generating a `.editorconfig` file

What I recommend if you work in a team environment is to generate a `.editorconfig` file that you can tweak according to your needs.

To generate a `.editorconfig` file, we *right-click* on our solution, and in the contextual menu, select **Add | New Editor Config**.

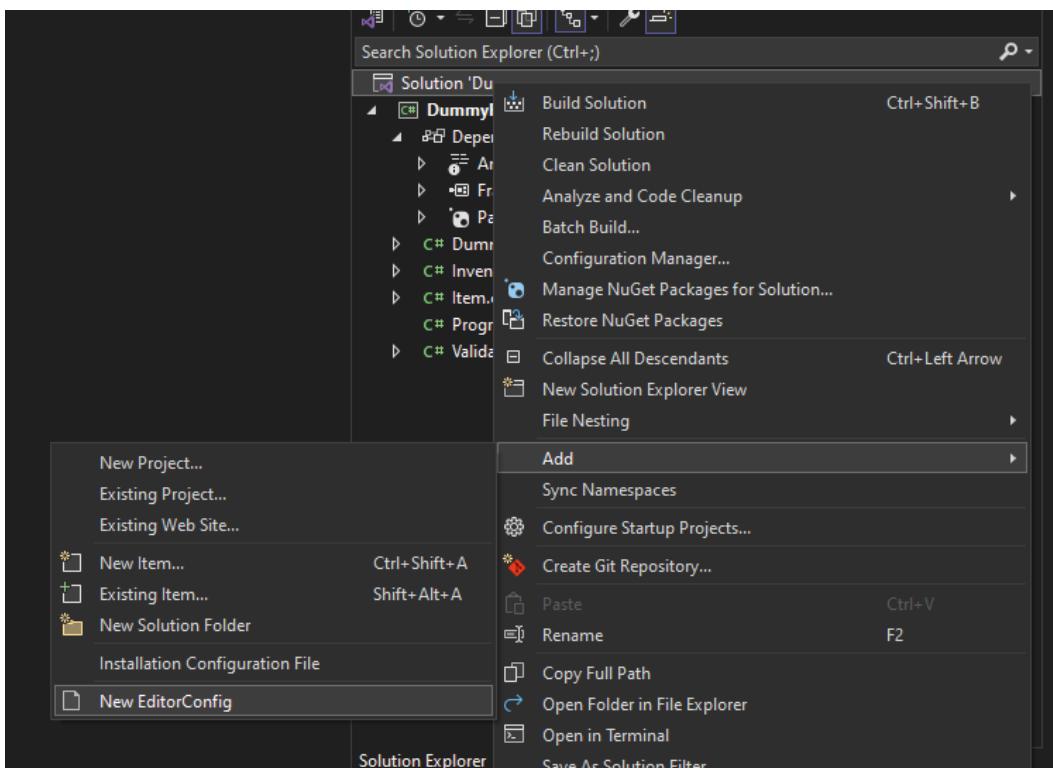
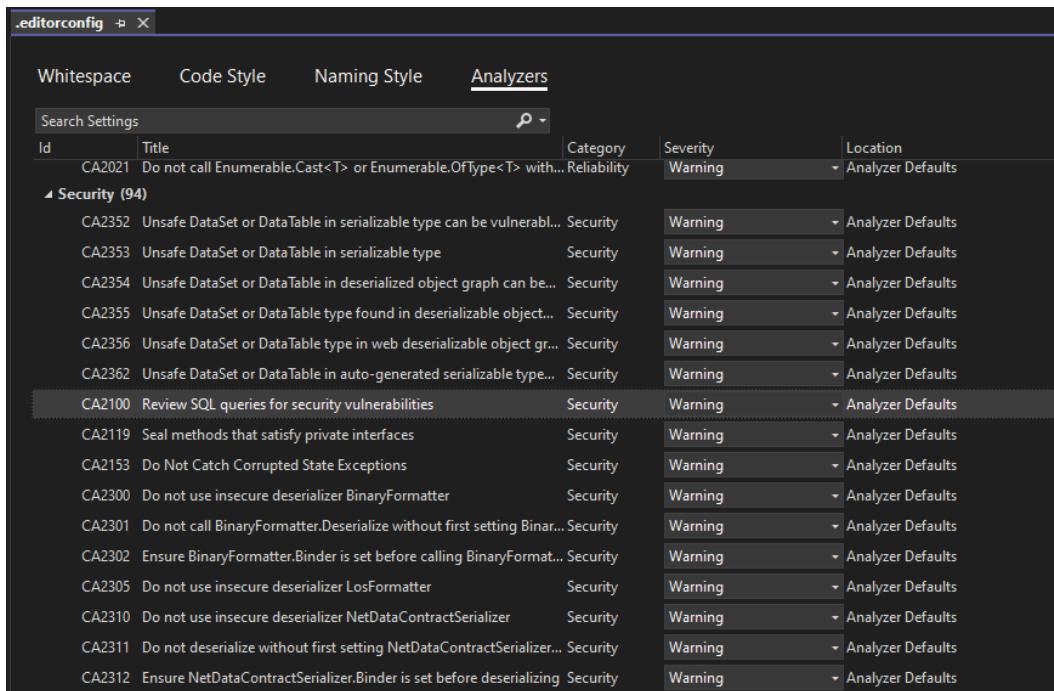


Figure 3.8 – New EditorConfig

This will generate a new file that will contain all configurations of our IDE. In this file, we will retrieve all the rules of the **Analyzer**.



The screenshot shows a table titled 'Analyzer' with columns: Id, Title, Category, Severity, and Location. The 'Category' column is currently set to 'Security'. There are 94 rows listed under the 'Security' category. Each row contains a rule ID (e.g., CA2021, CA2352, CA2353, etc.), a brief description of the rule, its category (Security), its severity level (Warning), and its location (Analyzer Defaults). The rows are ordered by rule ID.

Analyzer					
Search Settings		Category	Severity	Location	
Id Title					
CA2021	Do not call Enumerable.Cast<T> or Enumerable.OfType<T> with...	Reliability	Warning	Analyzer Defaults	
CA2352	Unsafe DataSet or DataTable in serializable type can be vulnerabl...	Security	Warning	Analyzer Defaults	
CA2353	Unsafe DataSet or DataTable in serializable type	Security	Warning	Analyzer Defaults	
CA2354	Unsafe DataSet or DataTable in deserialized object graph can be...	Security	Warning	Analyzer Defaults	
CA2355	Unsafe DataSet or DataTable type found in deserializable object...	Security	Warning	Analyzer Defaults	
CA2356	Unsafe DataSet or DataTable type in web deserializable object gr...	Security	Warning	Analyzer Defaults	
CA2362	Unsafe DataSet or DataTable in auto-generated serializable type...	Security	Warning	Analyzer Defaults	
CA2100	Review SQL queries for security vulnerabilities	Security	Warning	Analyzer Defaults	
CA2119	Seal methods that satisfy private interfaces	Security	Warning	Analyzer Defaults	
CA2153	Do Not Catch Corrupted State Exceptions	Security	Warning	Analyzer Defaults	
CA2300	Do not use insecure deserializer BinaryFormatter	Security	Warning	Analyzer Defaults	
CA2301	Do not call BinaryFormatter.Deserialize without first setting Binar...	Security	Warning	Analyzer Defaults	
CA2302	Ensure BinaryFormatter.Binder is set before calling BinaryFormat...	Security	Warning	Analyzer Defaults	
CA2305	Do not use insecure deserializer LosFormatter	Security	Warning	Analyzer Defaults	
CA2310	Do not use insecure deserializer NetDataContractSerializer	Security	Warning	Analyzer Defaults	
CA2311	Do not deserialize without first setting NetDataContractSerializer...	Security	Warning	Analyzer Defaults	
CA2312	Ensure NetDataContractSerializer.Binder is set before deserializing	Security	Warning	Analyzer Defaults	

Figure 3.9 – EditorConfig Analyzer

This allows us to list and set the severity level of each rule easily.

Here, we can see that Visual Studio Analyzer provides 94 rules about security. These rules cover all the topics of the OWASP Top Ten and more. The OWASP Top 10 serves as a widely recognized reference document for developers and web application security, highlighting the most significant security vulnerabilities faced by web applications based on a collective agreement within the industry.

### Further reading

We can find all the security rules in Microsoft documentation:

<https://learn.microsoft.com/en-us/visualstudio/code-quality/security-rules-rule-set-for-managed-code?view=vs-2019&viewFallbackFrom=vs-2022>.

Now that we have learned how we can use static code analysis, let's dive into a new feature. Indeed, Visual Studio 2022 now integrates IntelliCode as a built-in feature, available to all subscriptions. Let's explore how it works and how we can leverage its capabilities.

## Leveraging IntelliCode for code refactoring

As developers, we need to constantly refactor our code to improve its structure, readability, or performance, especially when we are working with TDD, with respect to the flow explained in *Chapter 1*.

One tool that has significantly enhanced our refactoring process is **IntelliCode**, which uses artificial intelligence and machine learning to offer intelligent suggestions and automate repetitive tasks. IntelliCode is now integrated into Visual Studio 2022 for C#. In this section, we'll explore strategies for effective code refactoring using IntelliCode, drawing from our experience and insight.

First of all, we need to ensure that we have IntelliCode installed in our Visual Studio by going to the **Options** menu through the top bar menu, which is **Tools | Options | IntelliCode**.

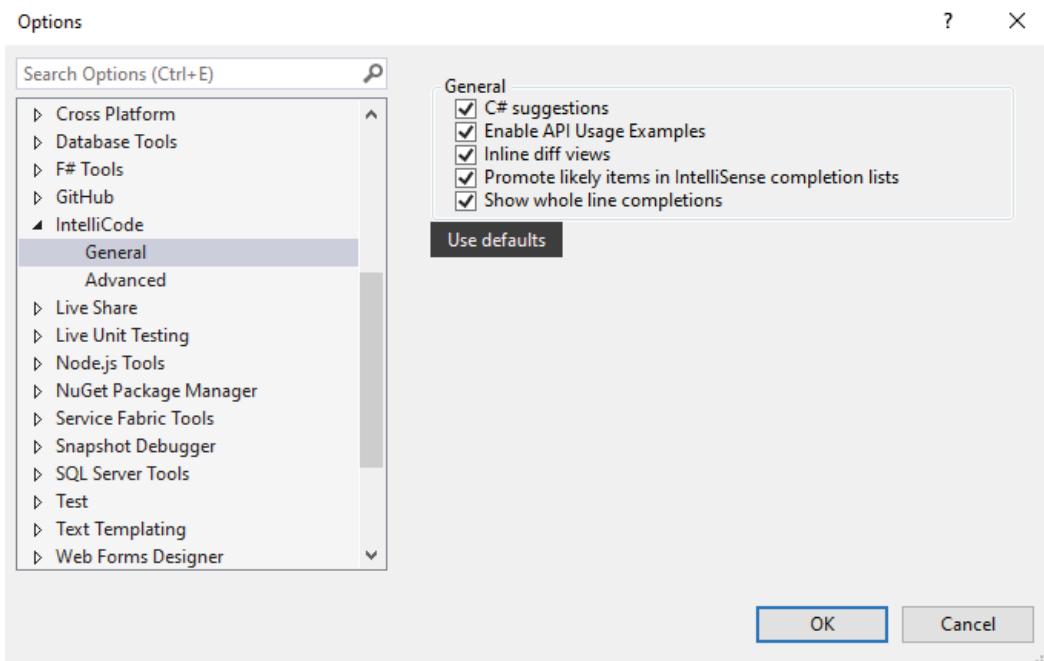


Figure 3.10 – IntelliCode Options

If you can't find IntelliCode in the **Options** menu, install it with your Visual Studio Installer by ticking the **IntelliCode** checkbox.

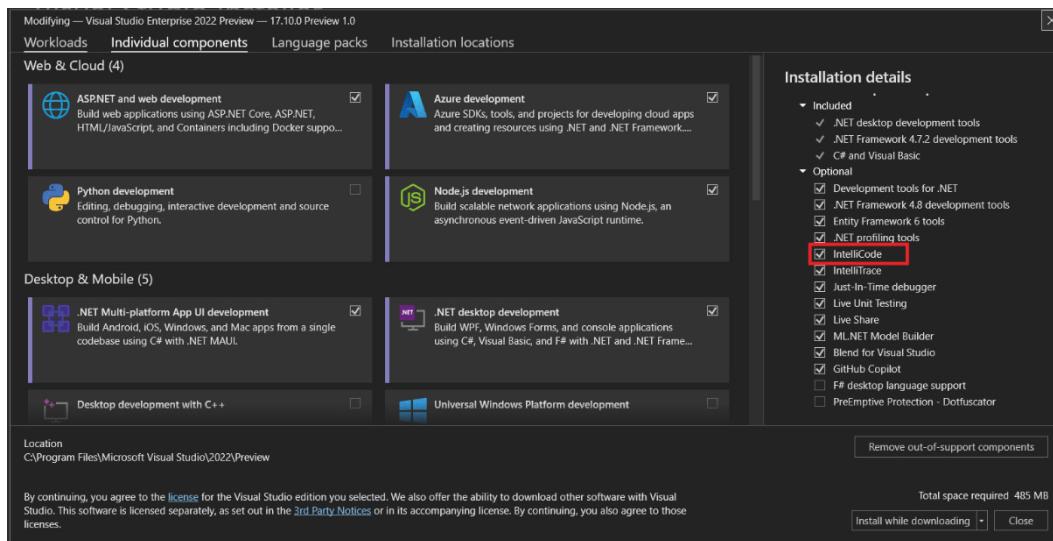


Figure 3.11 – Visual Studio Installer IntelliCode

Now that we've ensured that IntelliCode is enabled in Visual Studio, let's explore how it can enhance our coding experience.

## Predicting code with whole-line completion

IntelliCode, introduced in Visual Studio 2022, provides whole-line completion with suggestions. This feature is designed to streamline the development process, improve code quality, and increase productivity.

The following image shows whole-line completion:

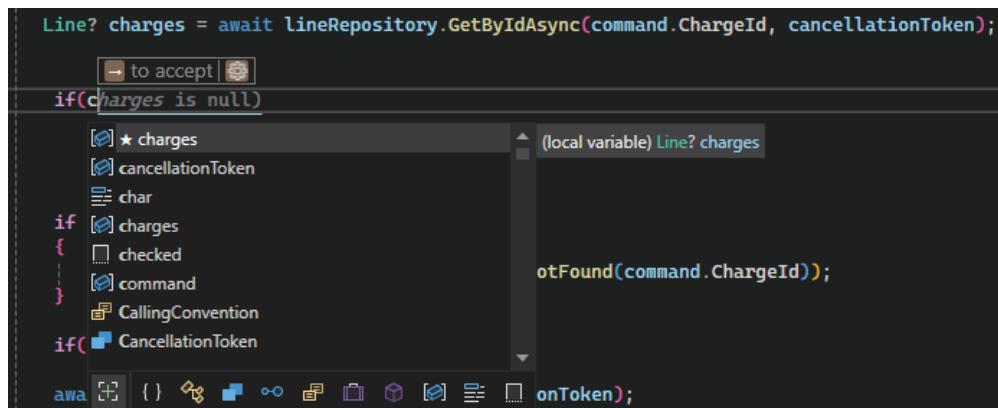


Figure 3.12 – Whole-line completion from typing

In *Figure 3.12*, we can see gray text *line prediction* generated according to the context. The prediction is based on a large amount of public, open source GitHub repositories. The IntelliCode suggestions are seamlessly integrated with those from in the IntelliSense. These IntelliCode recommendations are highlighted and easily identifiable by the *black star* icon located on the left of the suggestion.

If we select another suggestion than one provided by the whole-line completion, IntelliCode generates a new prediction according to our choice.

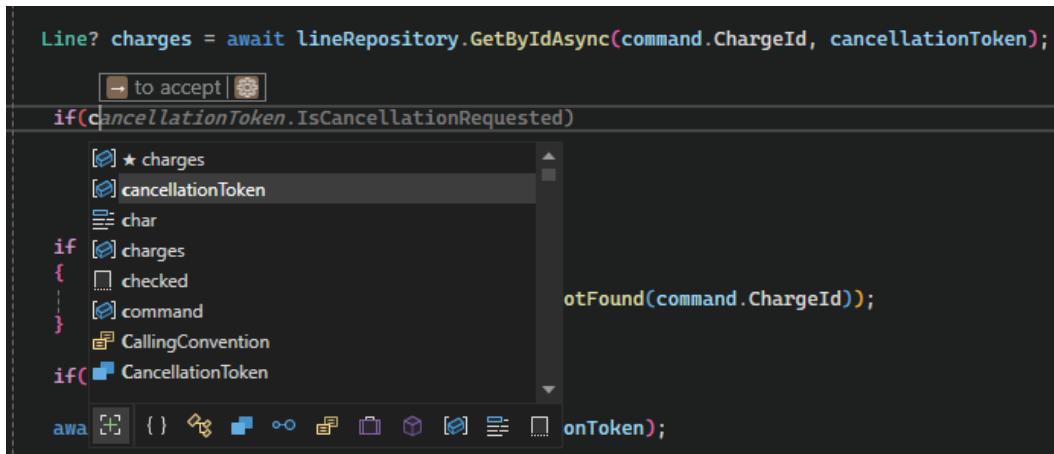


Figure 3.13 – Whole-line completion from IntelliSense

To accept the whole-line completion, we just need to press *Tab* or click on the *arrow* that pops above the suggestion. We can press *Esc* or *Delete* to dismiss it.

IntelliCode runs locally on the machine, providing whole-line completion, which enhances code security and consideration for properties, unlike ChatGPT and other AI predictions tools, which are generated inline.

Now that we've seen how IntelliCode can help us generate code, let's explore how it can assist us in understanding the code by providing direct access to documentation.

## Accessing GitHub documentation

When using external libraries or nuggets, we can encounter methods that we don't know how to use efficiently. IntelliCode provides a helpful feature called **API Usage Examples**, which provides practical examples of how to use these methods effectively.

To activate this feature, go to the **Options** menu (see *Figure 3.10*) and tick **Enable API Usage Examples**.

Now, we can hover our cursor on the method we want to know more about, and we get a link labeled **GitHub Examples and Documentation**:

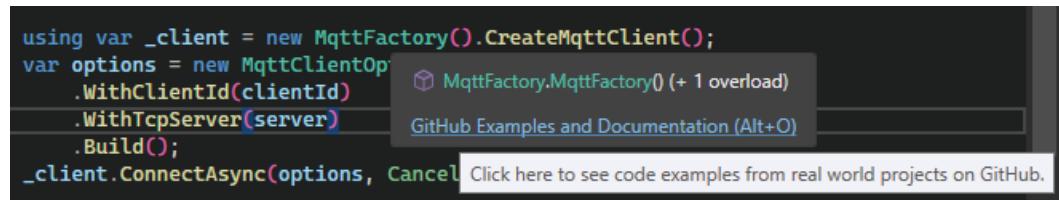


Figure 3.14 – GitHub Examples and Documentation link

Upon clicking on the link (**GitHub Examples and Documentation**), a side pin window opens with an example of the implementation of the method through several GitHub repositories:

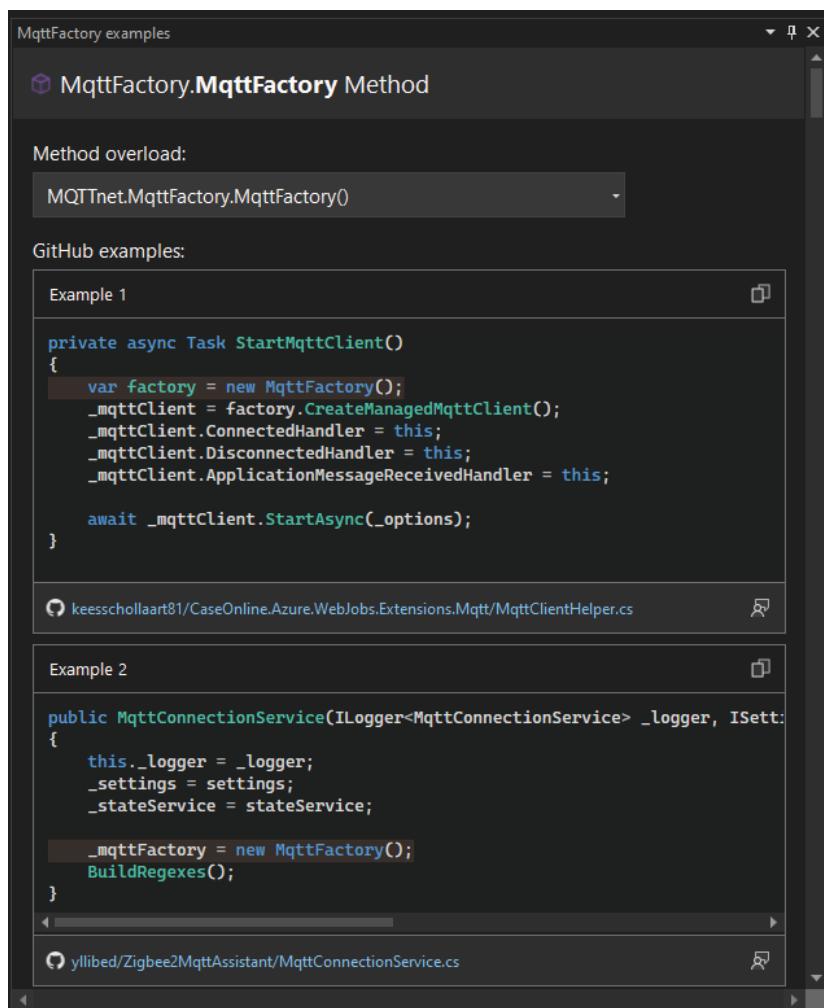


Figure 3.15 – GitHub examples

This feature helps us to plan our refactoring according to the examples of implementation in other projects, using the experience of other developers.

In the next section, let's dive into code metrics, which point to the areas of our project where we need to do some refactoring to improve maintainability and quality.

## Code metrics, maintainability, and security

As developers, we rely on **code metrics** to evaluate the quality of a software system. These quantitative measures provide insights into various aspects of our code, such as complexity, maintainability, and security. By understanding these metrics, we can identify areas that may need improvement, ensuring that our code base remains manageable, secure, and efficient.

### Understanding the metrics

Visual Studio allows us to generate a report containing a list of metrics, such as the following:

- **Maintainability Index** assesses how easy it is to maintain code, giving a score from 0 to 100, where higher scores indicate better maintainability. Ratings are color-coded: green (20-100) for *good*, yellow (10-19) for *moderate*, and red (0-9) for *low*.
- **Cyclomatic Complexity** measures a code's structural complexity by counting its different flow paths. High complexity suggests more tests are needed for good coverage and lower maintainability.
- **Depth of Inheritance** gauges how many classes inherit from one another, with lower values being better to prevent widespread changes.
- **Class Coupling** measures how tightly classes are connected, with high coupling indicating poor design.
- **Lines of Source Code** counts all lines in a source file, while **Lines of Executable Code** approximates the number of executable lines.

Now that we understand the metrics, let's delve deeper into how to use them through Visual Studio.

### Using code metrics in Visual Studio 2022

Visual Studio 2022 provides a powerful suite of tools for analyzing code metrics. Here are two ways to use them:

- In the top bar menu, select **Analyze | Calculate Code Metrics** and choose if you want to calculate the metrics on a specific project or on the solution

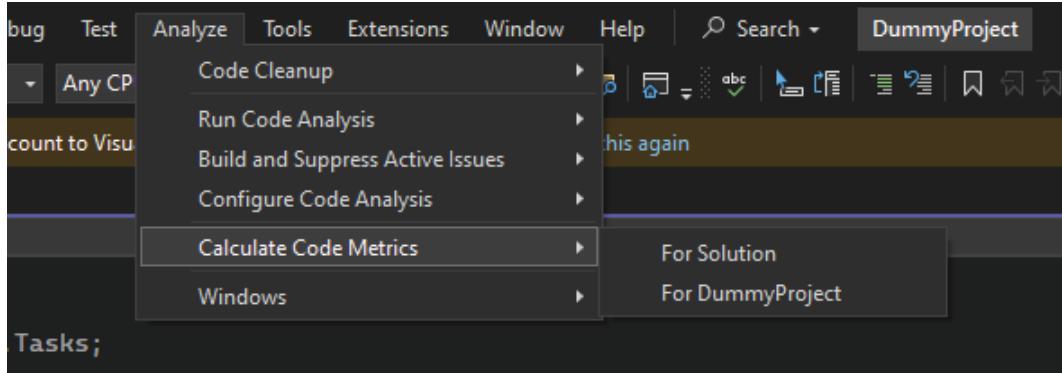


Figure 3.16 – Calculate Code Metrics from the top menu

- Alternatively, you can *right-click* directly on the solution or project you want to measure and, from the contextual menu, choose **Analyze and Code Cleanup| Calculate Code Metrics**

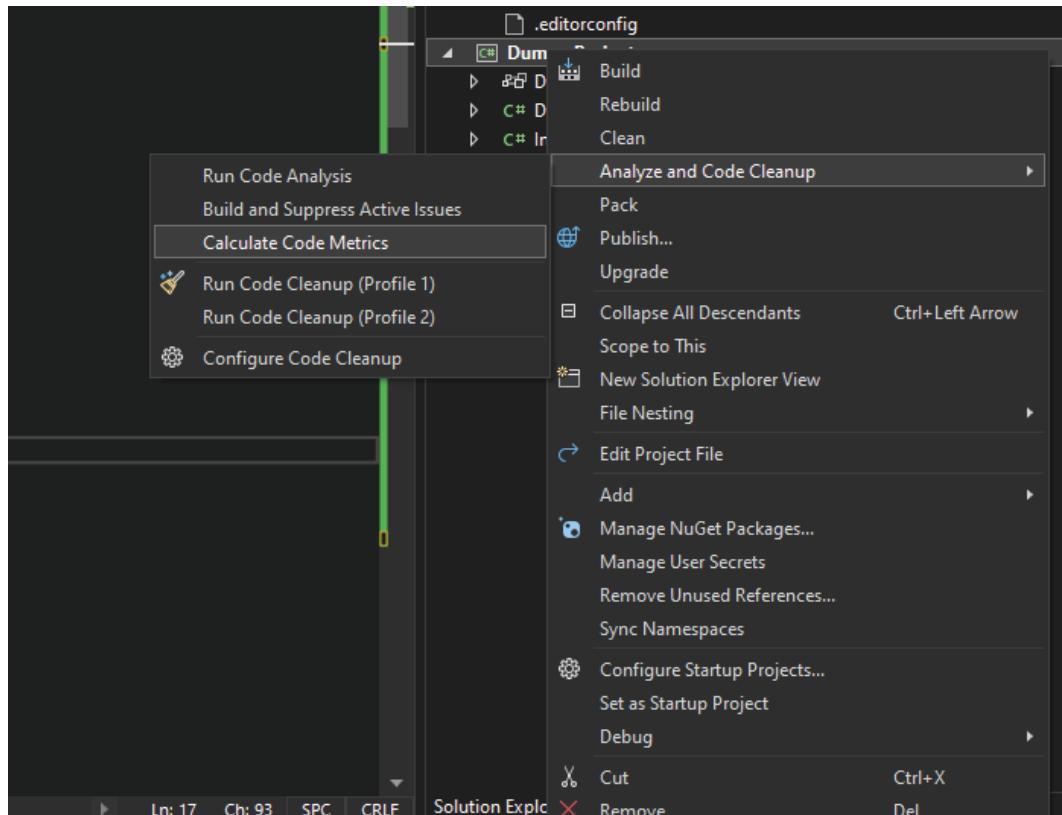


Figure 3.17 – Calculate Code Metrics for a project

After the analysis is complete, a window will open displaying the code metrics results:

Code Metrics Results							
Hierarchy	Maintainability Ind...	Cyclomatic Comp...	Depth of Inherita...	Class Coupling	Lines of Source c...	Lines of Executable c...	
ApiDummy (Debug)	74	9	1	34	77	38	
MyGroups	81	2	1	6	21	6	
GroupItemV1(this RouteGroupBuilder)	81	1		6	5	3	
GroupItemV2(this RouteGroupBuilder)	81	1		6	5	3	
Program	47	4	1	30	52	30	
<Main>\$string[] : void	55	2		30	52	15	
WeatherForecast	94	3	1	1	4	2	
WeatherForecast(DateOnly, int, string?)	100	1		1	4	0	
Date : DateOnly	100	0		1	1	0	
TemperatureC : int	100	0		0	1	0	

Figure 3.18 – Code Metrics Results

Here, we can view metrics such as **Cyclomatic Complexity**, **Depth of Inheritance**, **Class Coupling**, **Lines of Source code**, and **Lines of Executable code**. We can apply a custom filter to clamp the result between the minimum and maximum values for a metric. Furthermore, for better manipulation of the data, we can export the code metric results to an Excel file.

Understanding and utilizing code metrics is essential for maintaining high-quality software. Visual Studio 2022 provides powerful tools for analyzing code metrics, helping developers identify and address issues relating to security and maintainability. By focusing on these aspects, developers can ensure that their code bases remain robust, secure, and easy to manage.

In this section, we've navigated through tools that identify parts of code that need improvement and used IntelliCode for refactoring. In the final section, let's delve into a practical use case of refactoring.

## Refactoring case studies

**Refactoring** entails restructuring code while preserving its initial functionality. The objective is to enhance the internal structure through incremental modifications without affecting the code's external behavior. To check the integrity of our code, we use unit tests (see *Chapter 1* of this book). Refactoring is the last step in the test-driven design process.

This section showcases a small piece of code containing bad practices. We will see how to fix these issues efficiently with the help of Visual Studio. Remember that norms and practices must be discussed with all teams and can change from one team to another.

## Handling common bad practice

Here's a code with bad naming and a condition check that can be easily simplified:

```
public static bool canregister(int age)
{
    if(age > 18)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

We will use the *light bulb* to fix the naming violation (**IDE1006**) provided by the live code analyzer:

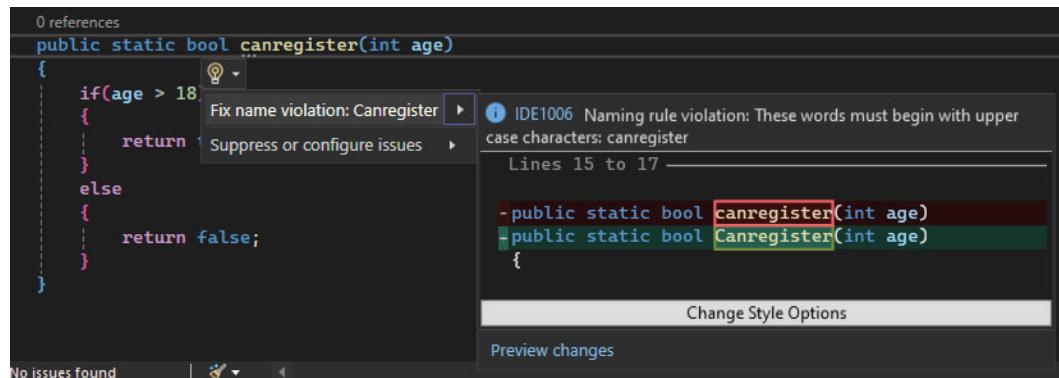


Figure 3.19 – Fixing the name violation

After that, we can simplify the age check with the *light bulb*:

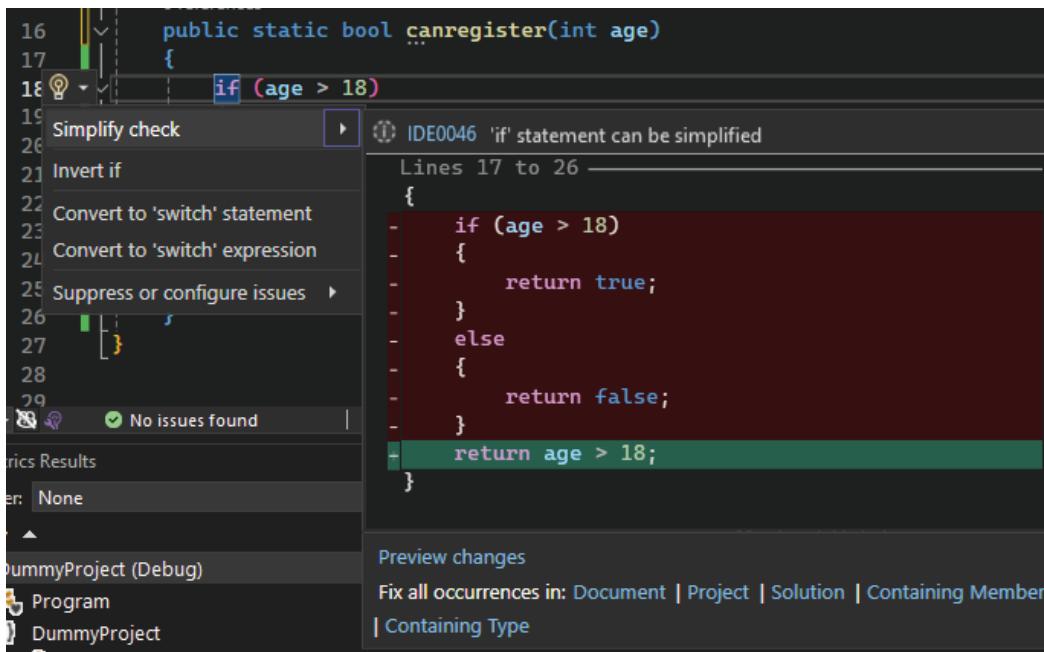


Figure 3.20 – Simplifying the age check

We can see that the **Preview changes** window allows us to be more aggressive in fixing all occurrences in the different levels of our solution.

Next, as our method contains only `return`, we can improve the readability by turning it into an **expression body** (a concise way to define the body of a method). We will use the *screwdriver* to do this:

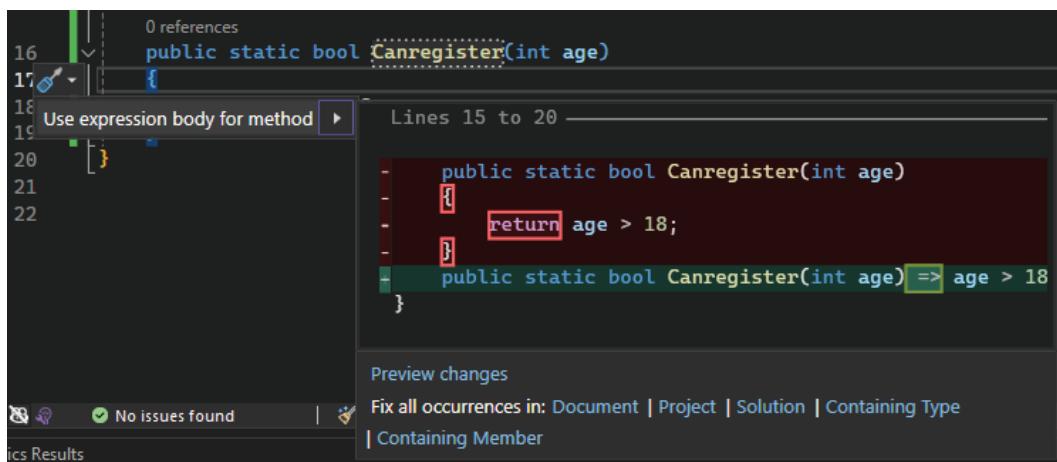


Figure 3.21 – Using an expression body

The last bad practice to handle is the usage of the magic number. A magic number refers to a numeric literal that appears directly in the source code without any explanation of its meaning or purpose. It can be challenging to change or understand without proper context. So, we will introduce a constant with a human-readable name to explain the purpose of this number. We will once again use the *light bulb* to do this:



Figure 3.22 – Introducing a constant

Now we get a simple expression body, which is well named, and one that respects clean code practices.

## Generating an interface

For instance, imagine we get a `AirTraffic` class with a `RegisterAircraft` method. Now to add abstraction to our code we want to create an `IAirTraffic` interface. We'll use the *screwdriver* to effortlessly extract the interface.

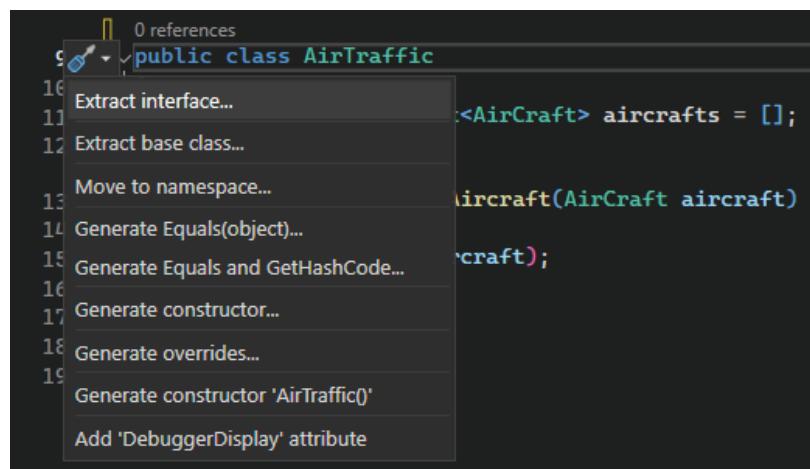


Figure 3.23 – Extracting an interface

This will open a window with options to allow us to choose the configuration of our interface. We will keep the default choices to create the interface in a new file, named `IAirTraffic.cs`.

Now, we are adding a new method, `SendMessage`, and we want it to belong to the interface as well. Again, we can use the *screwdriver* to pull the new method up to the interface:

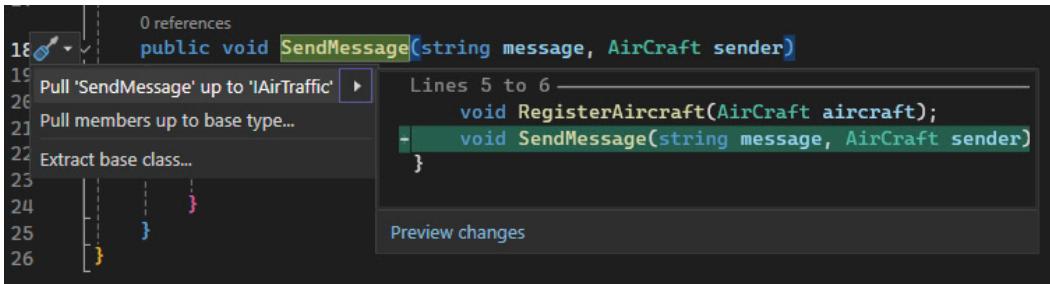


Figure 3.24 – Pulling the method up to interface

This refactoring ensures that all classes implementing `IAirTraffic` will have access to the `SendMessage` method, promoting a consistent interface and code reuse.

## File-scoping our namespace

The **file-scoped namespace** feature, introduced in C# 10, offers a more concise approach to declaring namespaces. It eliminates the need for curly braces to enclose the namespace body, streamlining the syntax and enhancing readability.

By default, when we create classes in Visual Studio, it uses the older version of namespace, which wraps the entire content of the file using curly braces. We can add a semicolon at the end of the declaration of the namespace and Visual Studio will automatically understand that you want to turn it in file-scoped namespace.

To avoid this repetitive action, we can set file-scoped namespace as the default in Visual Studio by using the top menu bar: select **Tools** | **Options** | **Text Editor** | **C#** | **Code Style** | **General** and change the namespace declaration value to **File scoped**:

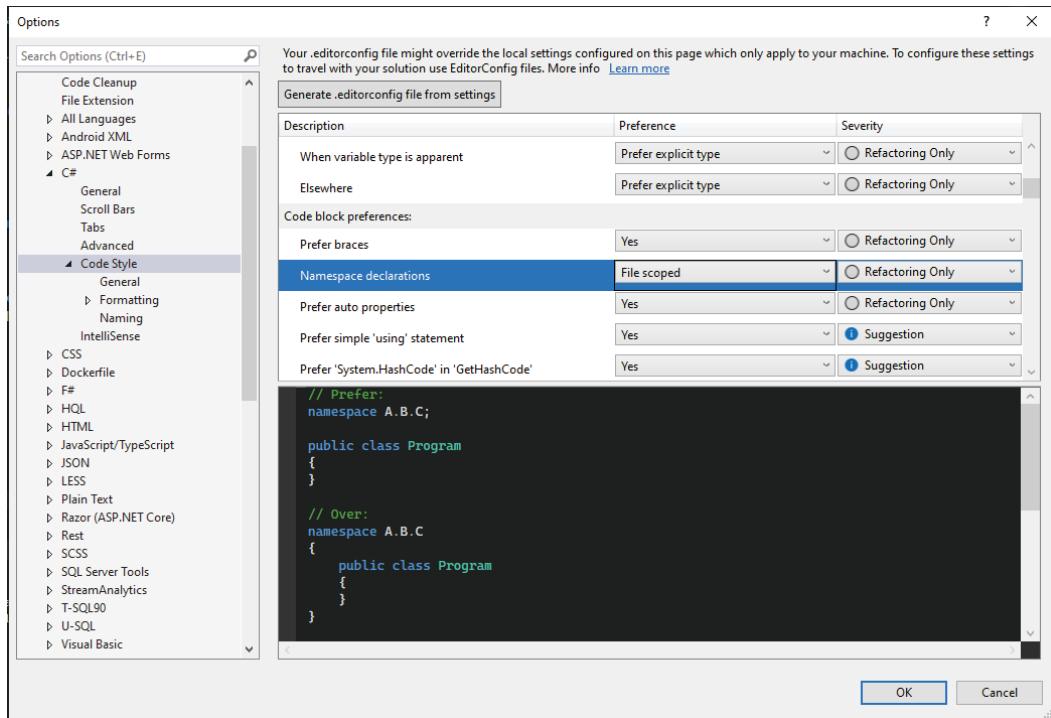


Figure 3.25 – Making namespace declarations file scoped

If we work in teams, we can generate the `.editorconfig` file directly for the **Options** window by clicking on the **Generate .editor file from setting** option, as we can see in *Figure 3.25*.

If we already have a `.editorconfig` file, (created manually or generated by changing the severity level, as seen in the *Utilizing static code analysis for quality assurance and security* section in this chapter), we can add the following line to set namespace declarations to `file_scoped`:

```
csharp_style_namespace_declarations = file_scoped
```

If we want Visual Studio to trigger compile-time errors when file-scoped namespaces are not used, the syntax would be as follows:

```
csharp_style_namespace_declarations = file_scoped:error
```

This allows us to commit the configuration to our Git repository, ensuring it's shared with our team.

## Summary

In this chapter, we explored the inner workings of Roslyn within Visual Studio 2022, empowering developers to use its capabilities fully. By mastering static code analysis, we can enhance quality assurance and security measures in our projects through nuanced control over severity levels and generating `.editorconfig` files and spreading them to our team.

We saw how IntelliCode redefines productivity, offering predictive coding via whole-line autocompletion and seamless access to GitHub documentation.

We navigated code metrics of maintainability and security, which equip us with invaluable insights that allow us to evaluate and improve code bases. Armed with Visual Studio 2022's built-in tools, projects can be optimized for long-term sustainability and robustness.

We concluded this chapter with practical refactoring case studies, addressing common pitfalls and demonstrating techniques such as interface generation and namespace refinement. By analyzing real-world scenarios, we gained hands-on experience in applying advanced refactoring principles.

In the following chapter, we explore strategies for identifying performance bottlenecks, optimizing code execution, and leveraging Visual Studio 2022's profiling tools to ensure peak performance in our applications.



# 4

# Performance Optimization and Profiling

In this chapter, we delve into the crucial aspect of ensuring that our code not only functions correctly but also runs efficiently. While writing code that works is essential, optimizing its performance is equally vital, especially in today's fast-paced digital landscape where users expect swift and responsive applications.

In the preceding chapters, we laid the groundwork by mastering unit testing, **test-driven development (TDD)**, advanced debugging strategies, and code analysis. Now, we shift our focus to the optimization and profiling tools available within Visual Studio 2022, empowering ourselves to fine-tune our applications for optimal performance.

Throughout this chapter, we will explore various techniques and methodologies aimed at enhancing the speed, responsiveness, and resource efficiency of our software. We'll begin by introducing the fundamentals of performance optimization and the importance of utilizing profiling tools to identify bottlenecks and areas for improvement.

Key topics covered in this chapter include the following:

- Introduction to performance optimization
- Utilizing Visual Studio profiling tools
- Analyzing CPU usage
- Memory profiling and optimization
- Optimizing database interaction

By mastering these concepts and techniques, we'll learn how to pinpoint performance issues, optimize critical sections of our code base, and ensure that our applications deliver a seamless user experience under various workloads and conditions.

Let's begin our journey toward building faster, more efficient software together.

## Technical requirements

While writing this chapter, I used the following version of Visual Studio:

- Visual Studio Enterprise 2022 Version 17.12.0
- Preview 1.0

The code files for this chapter can be found at <https://github.com/PacktPublishing/Mastering-Visual-Studio-2022/tree/main/ch04>.

## Introduction to performance optimization

Performance optimization in software development involves refining an application to operate with maximum efficiency, minimizing resource consumption such as memory, CPU, and bandwidth. This process includes analyzing performance at various development stages, often focusing on enhancing efficiency once a stable version of the product has been established.

The main step in our performance optimization process is *identifying bottlenecks*, as it allows us to pinpoint the specific areas of our code that are causing performance issues. **Bottlenecks** are points in our code where the execution slows down significantly, often due to resource constraints or inefficient algorithms. By identifying these bottlenecks, we can focus our optimization efforts on where they will have the most impact, leading to more efficient and faster applications. This targeted approach not only improves the performance of our application but also enhances the overall user experience by reducing load times and improving responsiveness. Furthermore, identifying bottlenecks early in the development process can prevent costly rework and delays, as it becomes more economical to address performance issues before they become entrenched in our application's architecture. In essence, the ability to identify and address bottlenecks is a key skill for us as developers aiming to create high-performance applications, ensuring that our software runs smoothly and efficiently.

Performance optimization could be performed at different levels and offers different paths of exploration:

- At the design level, the architecture of the system plays a crucial role in its performance. Designing with performance in mind involves making strategic decisions about how the system interacts with hardware and network resources. For instance, reducing network latency can be achieved by minimizing network requests, ideally making a single request instead of multiple. This approach not only reduces the load on the network but also simplifies the application's architecture, making it easier to maintain and scale.
- Implementation choices in the source code also have a significant impact on system optimization. Employing efficient coding practices is crucial for achieving system optimization. This includes avoiding unnecessary computations, which can significantly reduce the computational overhead of the application. For example, using **Language-Integrated Query (LINQ)** for data manipulation can lead to code that is more readable yet potentially more efficient than traditional loops. Additionally, utilizing C#'s asynchronous programming features, such as `async` and `await`, can

help to improve the responsiveness of applications by allowing them to perform other tasks while waiting for long-running operations to complete.

- The choice of algorithms and data structures is a critical factor in system performance. Efficient algorithms and data structures can significantly reduce the time complexity of operations, allowing the system to handle larger datasets and more complex tasks with ease. Ideally, algorithms should operate at constant ( $O(1)$ ), logarithmic ( $O(\log n)$ ), linear ( $O(n)$ ), or log-linear ( $O(n \log n)$ ) time complexities. Algorithms with quadratic complexity ( $O(n^2)$ ) can struggle to scale efficiently, especially as the size of the dataset grows. Similarly, abstract data types, which encapsulate data and operations in a single entity, can be more efficient for system optimization compared to more complex data structures.
- Optimizing at the build level involves making decisions during the build process that can tailor the application for specific processor models or environments. This can include disabling unnecessary software features, which can reduce the size of the executable and improve its performance. Additionally, build-level optimizations can involve the use of compiler flags that enable specific optimizations, such as loop unrolling or function inlining, which can improve the efficiency of the generated code.

You may have noticed that I categorized the algorithm by complexity using Big O notation; let's take a refresher about this notation.

Big O notation serves as a mathematical representation utilized to characterize the performance or complexity of algorithms, particularly concerning their runtime behavior with increasing input size.

Proficiency in understanding Big O notation holds significant importance for software engineers. It equips us with the ability to evaluate and contrast the efficiency of diverse algorithms. Consequently, we can make well-informed decisions regarding the selection of algorithms suitable for specific scenarios.

The following points outline the well-known Big O notations:

- **Constant time  $O(1)$ :** Execution time remains unchanged irrespective of input data
- **Logarithmic time  $O(\log n)$ :** Complexity increases by one unit for each doubling of input data
- **Linear time  $O(n)$ :** Execution time increases linearly with the size of the input data
- **Log-linear time  $O(n \log n)$ :** Complexity grows as a combination of linear and logarithmic
- **Quadratic time  $O(n^2)$ :** Time taken is proportional to the square of the number of elements
- **Cubic time  $O(n^3)$ :** Execution time is proportional to the cube of the number of elements
- **Exponential time  $O(2^n)$ :** Time doubles for every new element added
- **Factorial time  $O(n!)$ :** Complexity grows factorially based on the size of the dataset

Now that we've explored the fundamental principles of performance optimization in software development, let's delve into practical methods for identifying and addressing performance issues. Some invaluable tools for our endeavor are Visual Studio profiling tools. By leveraging the capabilities of these tools, we can gain deeper insights into our application's performance and streamline the optimization process. Let's examine how Visual Studio profiling tools can be effectively utilized to enhance the performance of our software applications.

## Utilizing Visual Studio profiling tools

Visual Studio profiling tools are a suite of performance measurement and diagnostic tools integrated into Visual Studio. In this section, we will explore how to use it and explore what tools are offered to explore and monitor our applications.

To open the Performance Profiler, we go to **Debug | Performance Profiler** or press **Alt + F2**.

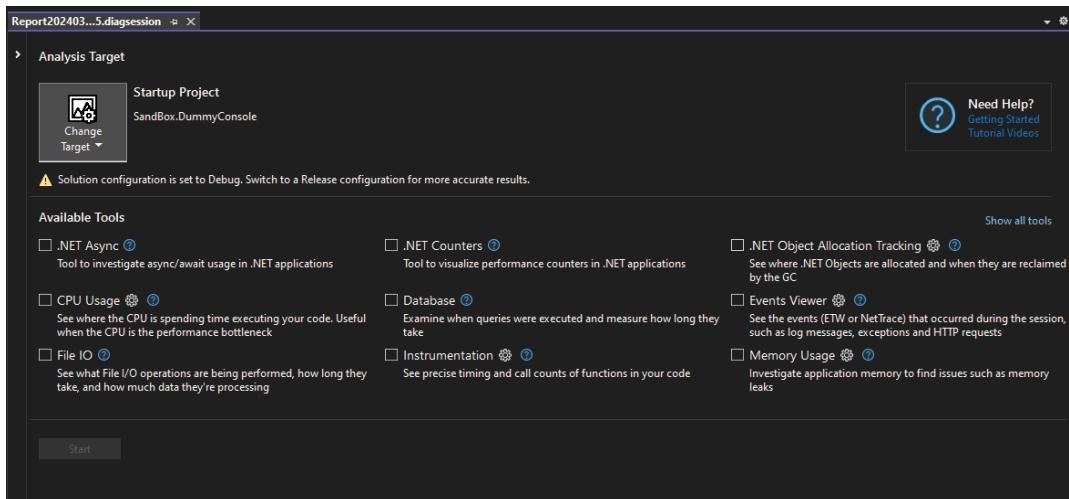


Figure 4.1 – Performance Profiler

Before clicking on the **Start** button, let's review the various options the Performance Analyzer feature offers us for profiling our applications.

### Analyzing .NET asynchronous events

.NET's async and await features allow us to analyze the asynchronous events that are organized chronologically, displaying start time, end time, and duration, in a table of activities that occurred during our profiling session. Tasks are labeled in the **Name** column.

Name	Count	Start Time (ms)	End Time (ms)	Total Time (ms)
System.Runtime.CompilerServices.AsyncTaskMethodBuilder<System.Threading.Tasks.Voi...	1	401,99	48 129,76	47 727,78
Task.WhenAll	1	400,37	48 125,74	47 725,37
Program.Main.AnonymousMethod_10	10	401,00	48 125,66	21 468,44 (avg.)
[Details]	10	401,00	48 125,66	21 468,44 (avg.)
6 within normal variance	6	401,00	48 125,66	20 454,64 (avg.)
[Task] Program.Main.AnonymousMethod_10	1	401,00	403,27	2,27
[Task] Program.Main.AnonymousMethod_10	1	401,15	3 268,54	2 867,39
[Task] Program.Main.AnonymousMethod_10	1	403,62	48 125,66	47 722,04
[Task] Program.Main.AnonymousMethod_10	1	403,73	41 768,54	41 364,81

Figure 4.2 – .NET Async

If a task isn't complete within the collection session, an *Incomplete* label appears in the **End Time** column.

To investigate a specific task or activity, we can right-click the row and select **Go To Source File** to see where in our code that activity happened.

Understanding the characteristics of async performance is crucial. While async methods are meant to enhance application responsiveness and scalability, they do introduce some overhead due to the state machine created by the compiler. However, this overhead is generally minimal and efficient for high-throughput scenarios.

When comparing the performance of async versus sync code, we need to consider the nature of the operations being performed. For operations inherently asynchronous (such as **input/output (I/O)**-bound operations), async methods can provide significant performance benefits by freeing up threads to handle other requests while waiting. However, for CPU-bound operations, the performance difference between async and sync methods may be negligible.

For effective performance monitoring of our application using .NET's async and await features, we can utilize the **.NET Async** tool in Visual Studio for detailed analysis of asynchronous code execution. Additionally, external tools such as Stackify Retrace offer comprehensive monitoring capabilities for .NET applications, including support for async/await. Understanding the performance characteristics of async methods and the nature of the operations being performed is crucial for making informed decisions about when to employ async programming patterns.

## Monitoring with .NET Counters

Visual Studio 2022 integrates the **.NET Counters** tool, which is an advanced feature that allows developers like us to visualize performance counters over time directly within the Visual Studio profiler. This tool proves to be particularly useful for monitoring and analyzing various metrics of .NET applications, such as CPU usage, garbage collector heap size, and any custom counters we might have implemented in our applications. This integration enables us to leverage the power of .NET Counters directly from within the Visual Studio environment, providing a more seamless and integrated experience for performance monitoring and analysis.

Visual Studio 2022 enhanced the .NET Counters tool to support two innovative metrics: UpDownCounter and ObservableCounter. UpDownCounter enables real-time tracking of values with both incremental and decremental changes, which is ideal for monitoring dynamic values such as user interactions in web applications. On the other hand, ObservableCounter autonomously manages aggregated totals, offering customizable callback delegates for precise control. This feature can be particularly useful for optimizing server resources by efficiently managing active session totals.

Additionally, a filter flyout feature in the tool allows us to conveniently filter data points based on tags. This dynamic adjustment feature significantly enhances the flexibility and streamlining of monitoring dynamic values in our projects.

While collecting data, we can see live values of .NET Counters and view graphs of multiple counters simultaneously.

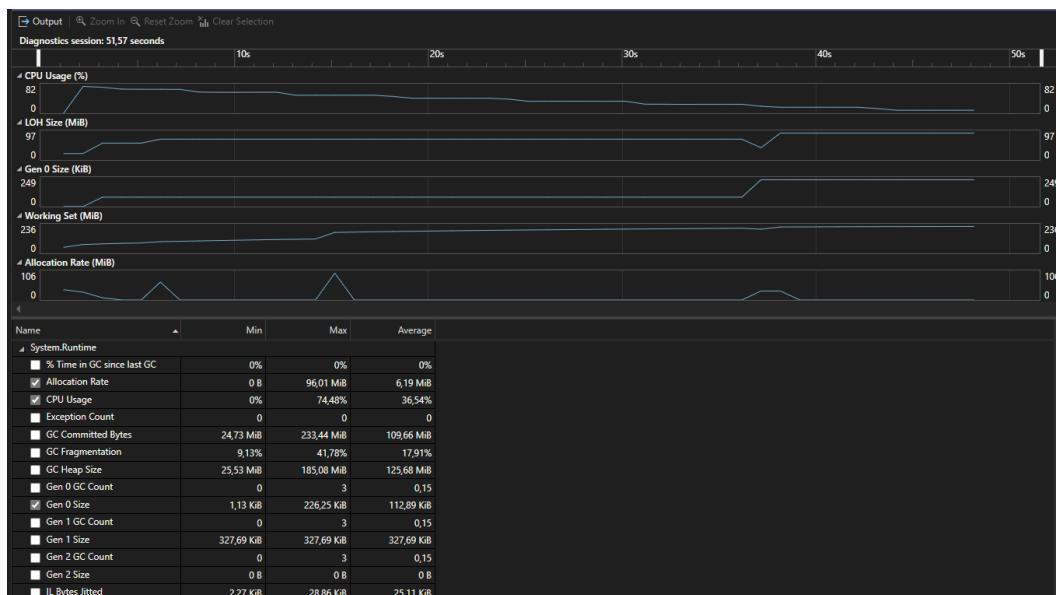


Figure 4.3 – .NET Counters

Once we stop collection, we get a detailed report showing minimum, maximum, and average values for each counter in the selected time range. This report provides us with a comprehensive overview of the performance metrics of our application, helping us identify and address performance bottlenecks more effectively.

## Tracking .NET Object Allocation

The **.NET Object Allocation Tracking** tool proves particularly valuable for understanding allocation patterns in .NET code and optimizing an application's memory usage by identifying the most memory-intensive methods. However, it's important to note that while this tool can reveal where objects are allocated, it does not elucidate why an object remains in memory.

We initiate the tool by clicking on the **Start** button, and the tool offers the **Start with collection paused** option before starting the profiler if we prefer to commence with data collection paused.

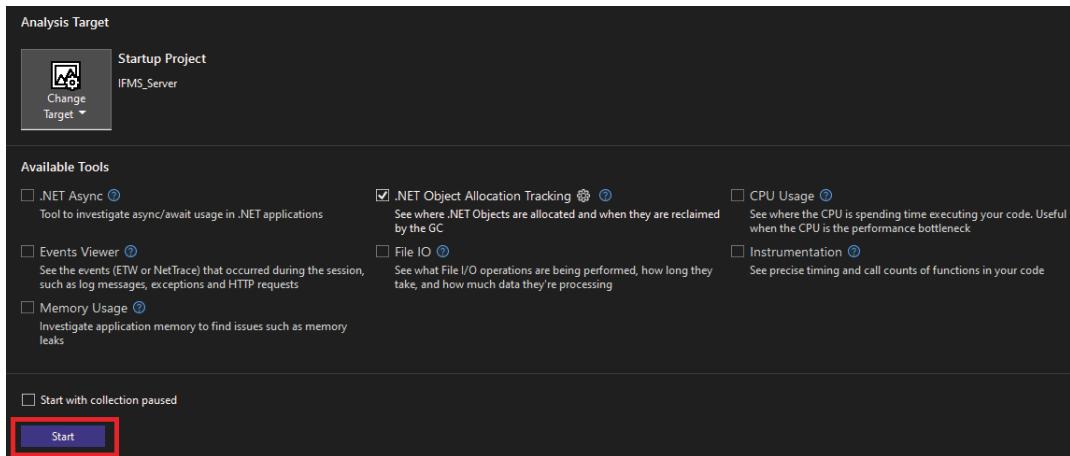


Figure 4.4 – Start analysis

This allows us to manually commence data collection by clicking the **Record** button in the diagnostic session view.

After executing the tool, we can halt the collection or close our application to review the data. The tool furnishes comprehensive memory allocation data, including the location of objects that are allocating memory and the amount of memory those objects are allocating. It also displays the number of objects that occupy memory within a specific allocation type or function, as well as the amount of memory consumed instead of the number of objects.

Additionally, the tool presents a **Collections** view, which illustrates how many objects were collected or retained during garbage collection, offering insights into garbage collection events such as the type of garbage collection, the reason for the event, and the size of the **large object heap (LOH)** and **pinned object heap (POH)** after the garbage collector was executed.

## Viewing the event

The **Events Viewer** tool allows us to examine the collected information after our application has stopped, like a post-mortem analysis. It displays a list of events such as module load, thread start, and system configuration, aiding in diagnosing our application's performance within the Visual Studio profiler.

To enable custom **Event Tracing for Windows (ETW)** events, we can instrument our code with custom events and configure them to appear in the Events Viewer. This involves setting up the provider's name and GUID for our custom event code. For C# custom event code, we set the same provider's name value that we used when declaring our event code, and for the native custom event code, we set the provider GUID based on the GUID for the custom event code. Once configured, these custom events will appear in the Events Viewer when we collect a diagnostics trace.

The Events Viewer can display up to 20,000 events at a time. To focus on specific events, we can filter the display by selecting the **Event** filter. Additionally, we can see the percentage of the total number of events that occurred for each provider, providing a breakdown of where our time is being spent. This filtering and breakdown help in identifying the most relevant events for our analysis.

For example, to enable custom ETW events:

1. First, build our custom event code.
2. Then, in the **Performance Profiler** window (accessed via *Alt + F2*), enable **Events Viewer** and select the **Settings** icon next to it.
3. In the dialog box, enable the first row under **Additional Providers** and configure it according to our custom event code.
4. For native custom event code, we set the **Provider GUID** value and leave the **Provider Name** value empty or use its default value.
5. For C# custom event code, we set the same **Provider Name** value that we used when declaring our event code, leaving the **Provider GUID** empty. After configuration, our custom events will appear in the Events Viewer when we collect a diagnostics trace.

This tool is particularly useful for us as developers looking to diagnose performance issues or understand the behavior of our applications in detail. It provides a comprehensive view of our application's activity and performance metrics.

## Analyzing File I/O

The **File IO** tool in Visual Studio is a powerful profiling tool designed to help us optimize our file I/O operations, thereby improving the performance of our applications. This tool is particularly useful for diagnosing slow loading times and inefficient data read or write patterns. It provides detailed information about file read and write operations during a profiling session, including the files accessed, the target process for each file, and aggregate information for each file. The tool also offers features such as the **Duplication Factor**, which helps us identify whether more data is being read or written than necessary, indicating potential areas for optimization, such as caching results of file reads.

The File IO tool provides file read and write information with files read during the profiling session. The files are autogenerated in a report after collection and arranged by their target process with aggregate information displayed. If we right-click on one of the rows, we can go to the source in our code. If an aggregate row was read multiple times, we can expand it to see the individual read operations for that file with its frequency, if they were read multiple times.

## Analyzing database performance

The **Database Profiler** tool in Visual Studio is a feature designed to help us developers diagnose and optimize the performance of database operations within our applications. It proves particularly useful for applications that use .NET Core with either ADO.NET or Entity Framework Core, offering insights into database activities such as query execution times, the connection strings used, and where in the code these queries are being made. This tool is part of the Performance Profiler in Visual Studio and has been available since Visual Studio 2019 version 16.3 onwards.

Once we start the profiling session, we interact with our application as we would normally, performing actions that we suspect might be causing database performance issues. After completing our actions, we stop the collection in Visual Studio. The tool then processes the collected data and displays a table of the queries that occurred during our profiling session, along with a graph showing the timing and frequency of these queries. This information can help us identify long-running queries, inefficient connection strings, or other performance bottlenecks in our database operations.

Furthermore, the Database Profiler tool supports analyzing traces collected using dotnet trace, allowing us to collect data from anywhere .NET Core runs, including Linux, and analyze it in Visual Studio. This feature is particularly useful for diagnosing performance issues in environments where Visual Studio is not installed or for scripting the collection of performance traces.

In summary, the Database Profiler tool in Visual Studio is a powerful diagnostic tool for us developers working with .NET Core applications that interact with databases. It provides detailed insights into database operations, helping us identify and resolve performance issues more effectively.

## Instrumenting our .NET applications

In Visual Studio, we utilize instrumentation tools for collecting precise call counts and call times, which are crucial for performance profiling and optimization. There are two main types of instrumentation methods available:

- **Static instrumentation:** This method involves modifying the program's files before they run. We use a tool called VSInstr to insert instrumentation code into the application's binaries. Static instrumentation is effective for collecting detailed timing data but can break strong name signing due to file modification. It also requires files to be deployed in a specific order, which can be cumbersome for complex programs.

- **Dynamic instrumentation:** Introduced in Visual Studio 2022 version 17.5, this method does not alter the program's files. Instead, it loads the files into memory and modifies them at runtime to collect instrumentation information. Dynamic instrumentation provides more accurate information, especially for smaller parts of the program, and allows for the investigation of specific code sections. It avoids the issue of breaking strong name signing since the instrumentation happens at runtime. This approach simplifies the process of finding and instrumenting files, especially in complex programs.

This tool is like the CPU Usage tool but focuses on wall clock time instead of CPU utilization, making it suitable for scenarios where understanding the execution time of functions is critical.

By using the aforementioned tools, we can gather valuable insights into our application's performance.

However, to ensure optimal accuracy in our performance measurements, it's advisable to profile our applications in the **Release** mode rather than the **Debug** mode. Debug builds can introduce additional overhead, potentially skewing our results.

When we find ourselves needing to inspect variable values or use breakpoints during analysis, we should consider leveraging the debugger-integrated tools found in the **Diagnostic Tools** window. These tools are tailored for such tasks and may offer a more suitable environment for our analysis.

To gain a holistic understanding of our application's performance, we can take advantage of Visual Studio's capability to utilize multiple profiling tools simultaneously. This approach allows us to examine our application's performance from different perspectives, providing a more comprehensive analysis.

You may have noticed that I have missed three tools offered by the Profiler Performance tools. I will highlight them in the next sections, beginning with the CPU Usage analyzer.

## Analyzing CPU Usage

The **CPU Usage** tool in Visual Studio is designed to help us identify high CPU utilization and other related performance issues in our applications. It can be used for both local trace sessions and production environments, providing insights into where optimizations might be needed. To use the CPU Usage tool without the debugger, we should set the solution configuration to **Release** and select **Local Windows Debugger (or Local Machine)** as the deployment target. Under available tools, we select **CPU Usage**, and then we select **Start**.

If we enable the start with collection paused option, data collection will not begin until we select the **Record** button in the diagnostic session view. After the app starts, the diagnostic session begins, displaying CPU usage data. Once we're finished collecting data, we select **Stop Collection**. The tool then analyzes the data and displays a report, which can be filtered and searched for specific threads or nodes.

The CPU Usage tool is particularly useful for diagnosing performance issues in our code base, identifying bottlenecks, and understanding CPU usage patterns. It provides automatic insights and various views of our data, enabling us to analyze and diagnose performance issues effectively. This tool is beneficial in production and difficult to debug at the moment but can be captured and analyzed using this tool to understand potential causes and suggest fixes.

The CPU Usage tool is particularly useful for diagnosing slow-downs, process hangs, and identifying bottlenecks in your code base, making it an essential tool for optimizing application performance.

When running this profiler tools spot the usage of CPU/second and collect traces generating a report with a graph to visualize the peak and valley of CPU usage.

When we first start the CPU Usage toll, it will collect a large amount of data per second to analyze what is going on in our application, and by default, it's set at 1000 samples/second.

We can personalize the rate of the number of samples collected by second, by clicking on the gear at the right of the **CPU Usage** label in the profile console (*Figure 4.1*) before hitting the **Start** button. Depending on our needs, we can adjust the accuracy of their results and the data collection time.

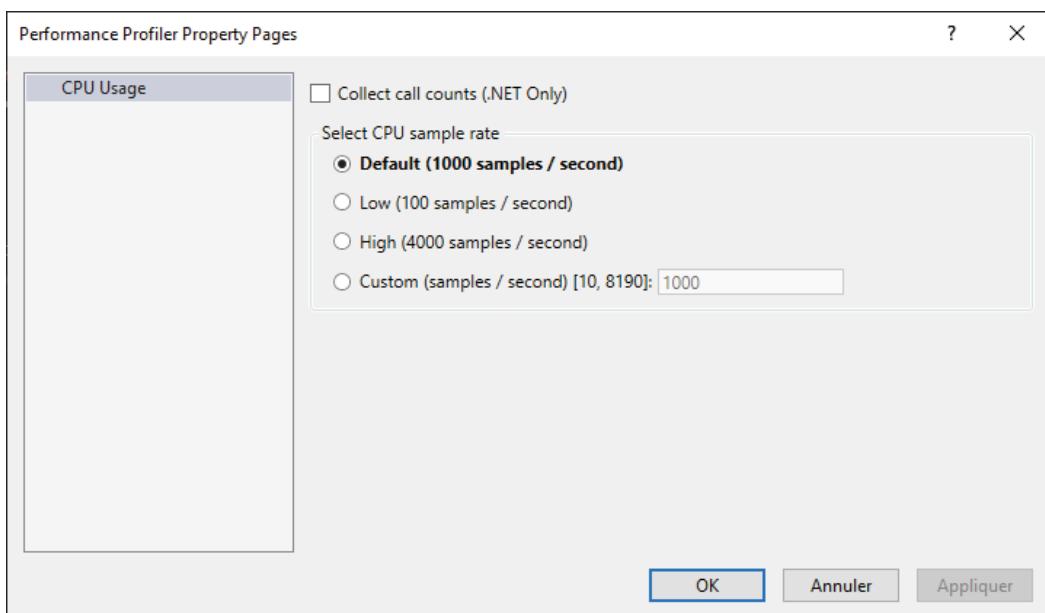


Figure 4.5 – The CPU Usage settings

When we stop the collection or shut down our application, CPU usage tools generate reports. Initially, we land on the summary page, which displays a swim lane graph, the **Top Functions** section, and the **Hot Path** section.

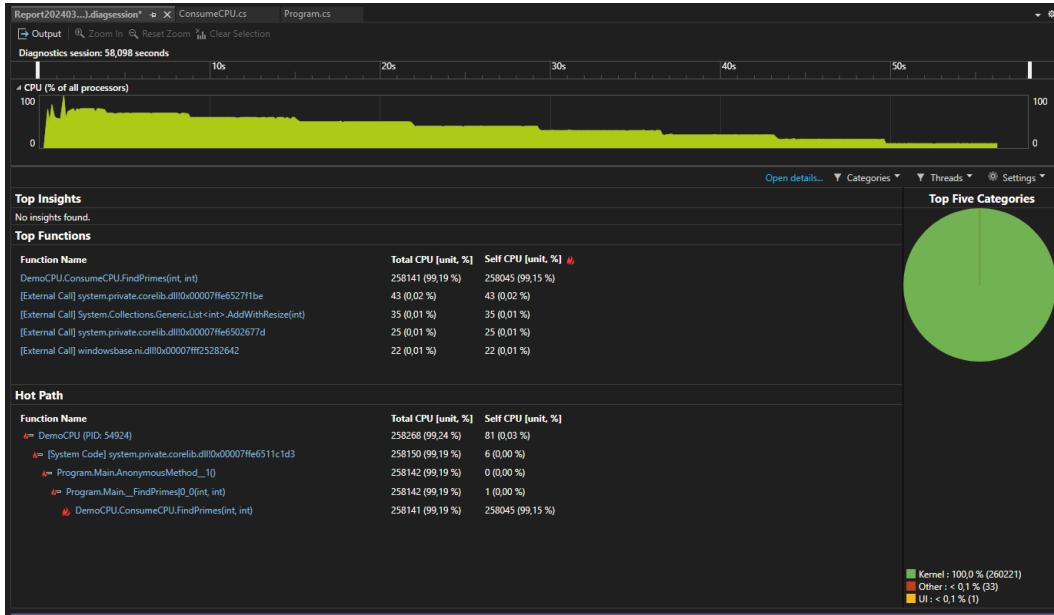


Figure 4.6 – The Summary page

Here, we can narrow down the potential bottleneck by right-clicking and dragging on the graph to surround the peak we want to focus on. By doing that, we effectively filter the graph by time.

By clicking on the **Open details...** link, we can digg deeper into the five other views:

- **Caller/Callee**
- **Call Tree**
- **Modules**
- **Functions**
- **Flame Graph**

Let's understand each of these in detail:

- In the **Caller/Callee** view, we can observe the relationship between a selected function and the functions that called it (**Calling Functions**) as well as the functions it called (**Called Functions**). It offers insights into the total time taken by the selected function and its percentage of the overall app running time. Additionally, it provides information on the time spent exclusively in the function body (**Function Body**). This view helps us understand the impact of a function on the application's performance and identify potential bottlenecks.



Figure 4.7 – Caller/Callee

- The **Call Tree** view presents a hierarchical representation of the function calls in our application, starting from the top-level pseudo-node. It includes system and framework code (under an **[External Code]** node) as well as user-code methods.

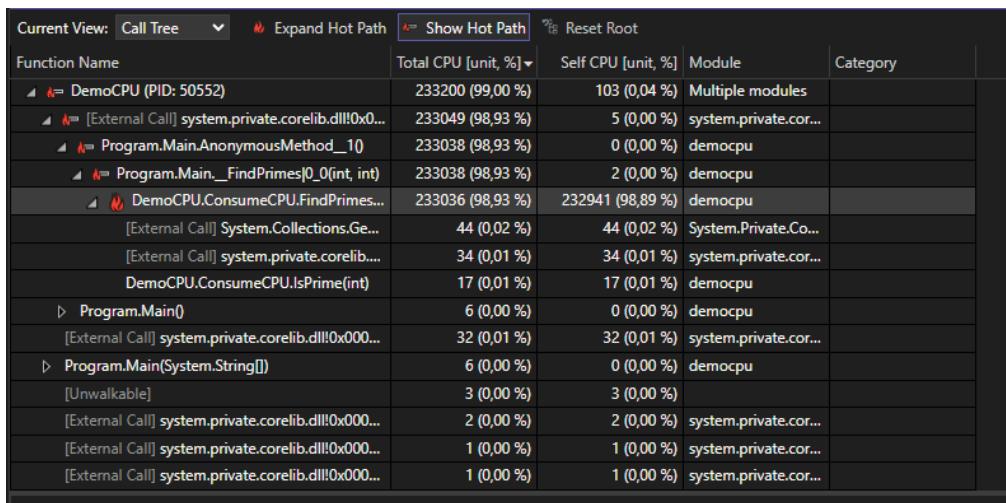


Figure 4.8 – The Call Tree view

This view is useful for understanding the sequence and nesting of function calls, aiding in the identification of the most CPU-intensive paths in our application.

- In the **Modules** view, we can see a list of modules containing functions, which can be particularly useful when analyzing external code.

Current View: Modules			
Name	Total CPU [unit, %]	Self CPU [unit, %]	Inline
DemoCPU (PID: 50552)	233200 (99,00 %)	103 (0,04 %)	
system.private.corelib	233090 (98,95 %)	80 (0,03 %)	
democpu	233050 (98,94 %)	232961 (98,90 %)	
System.Private.CoreLib.il	50 (0,02 %)	47 (0,02 %)	
system.console	6 (0,00 %)	6 (0,00 %)	
Idle (PID: 0)	1465 (0,62 %)	1465 (0,62 %)	
StandardCollector.Service (PID: 26084)	199 (0,08 %)	199 (0,08 %)	
chrome (PID: 55652)	188 (0,08 %)	188 (0,08 %)	
chrome (PID: 15948)	96 (0,04 %)	96 (0,04 %)	
devenv (PID: 21316)	76 (0,03 %)	39 (0,02 %)	
dwm (PID: 1828)	58 (0,02 %)	58 (0,02 %)	
System (PID: 4)	32 (0,01 %)	31 (0,01 %)	
explorer (PID: 12580)	31 (0,01 %)	31 (0,01 %)	
NisSrv (PID: 5596)	20 (0,01 %)	20 (0,01 %)	
chrome (PID: 17880)	20 (0,01 %)	20 (0,01 %)	

Figure 4.9 – The Modules view

It helps us understand which modules are contributing the most to CPU usage, assisting in the identification of third-party libraries or system components that might be impacting performance.

- The **Functions** view lists all the functions in our application, sorted by their CPU usage. It provides detailed information such as **Total CPU** (the time spent by the function and any functions it called) and **Self CPU** (the time spent exclusively in the function body).

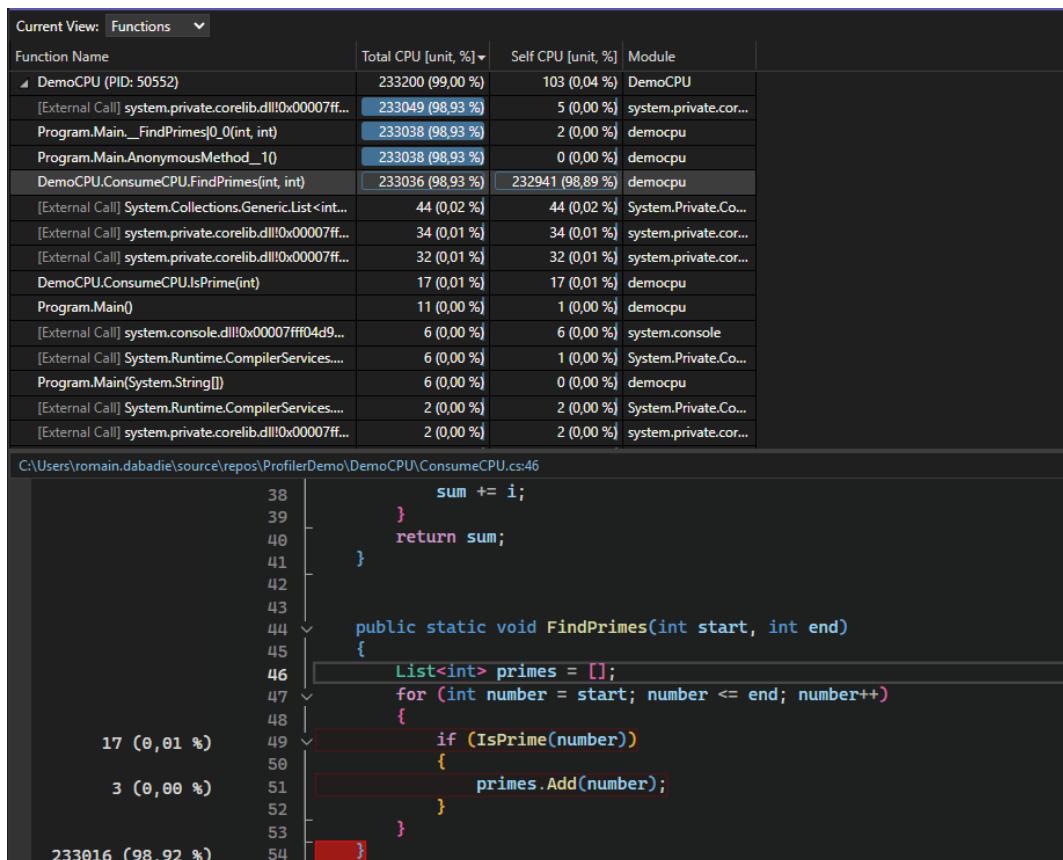


Figure 4.10 – The Functions view

This view is essential for identifying the most resource-intensive functions in our application and focusing on optimization efforts.

- A **flame graph** is a visualization that represents the call stack of our application over time. It helps in identifying hot paths, that is sequences of function calls that consume a significant amount of CPU time. The width of each function in the graph corresponds to the amount of CPU time it consumes, making it easier to spot performance bottlenecks. The **Flame Graph** view is particularly useful for understanding the overall CPU usage patterns of our application and pinpointing specific areas for optimization.

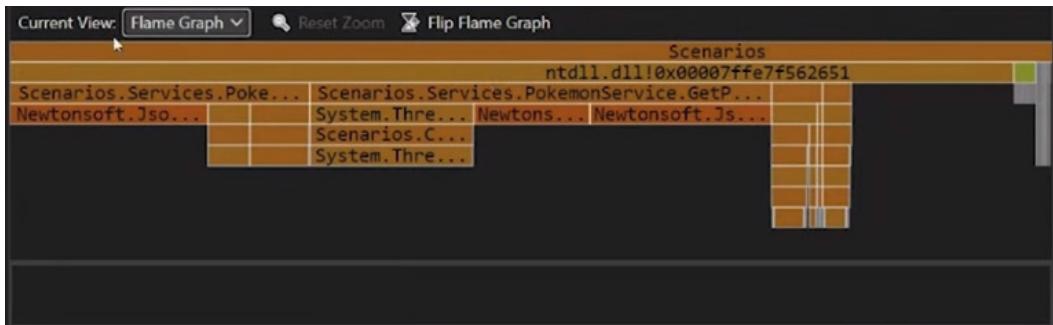


Figure 4.11 – Flame Graph

We can flip the view according to our preferences by using the **Flip Flame Graph** option and zooming in on our point of interest.

For debugging sessions, the CPU Usage tool can be accessed through the **Diagnostic Tools** window, which appears automatically unless turned off. We can select whether to see **CPU Usage**, **Memory Usage**, or both, with the **Select Tools** setting on the toolbar. The tool is enabled by default for CPU utilization analysis.

When the debugger pauses, the CPU Usage tool in the **Diagnostic Tools** window collects information about the functions executing in our application, listing the functions performing work and providing a timeline graph for focusing on specific segments of the sampling session.

While CPU usage and memory usage are related, they do not have a direct correlation. The impact of one on the other depends on the specific tasks being performed by the applications running on the system. Monitoring both metrics is crucial for optimizing system performance and effectively managing energy consumption.

Now that we've explored how to analyze CPU usage, let's continue our journey by monitoring memory allocation.

## Memory profiling and optimization

Just as a reminder, a memory leak occurs when a computer program mishandles memory allocations, leading to unreleased memory that is no longer needed. .NET applications are generally less vulnerable to memory leaks due to automatic garbage collection and the fact that .NET applications are written in managed code. This means that the runtime has control over memory allocation and deallocation. However, if we produce code with smells or misuse the disposable pattern, memory leaks can still occur.

In this section, we will explore how we can leverage Visual Studio to resolve memory leaks using the **Memory Usage** profiling tools and then the diagnostic tools available while debugging.

## Using the Memory Usage tools

To find and resolve a memory leak, we can use the **Memory Usage** tool in Visual Studio. It is a robust profiling feature designed to monitor and analyze our application's memory usage effectively. It supports various application types, including .NET, ASP.NET, C++, and mixed-mode applications. This versatile tool can be utilized both with and without the debugger, catering to different development scenarios. One of its key strengths lies in its ability to identify memory leaks and inefficient memory usage patterns.

During our diagnostic sessions, the Memory Usage tool provides a timeline graph illustrating memory fluctuations as our application runs. This graphical representation aids in pinpointing areas of our code that may be collecting or generating data inefficiently, potentially leading to memory leaks or excessive memory usage.

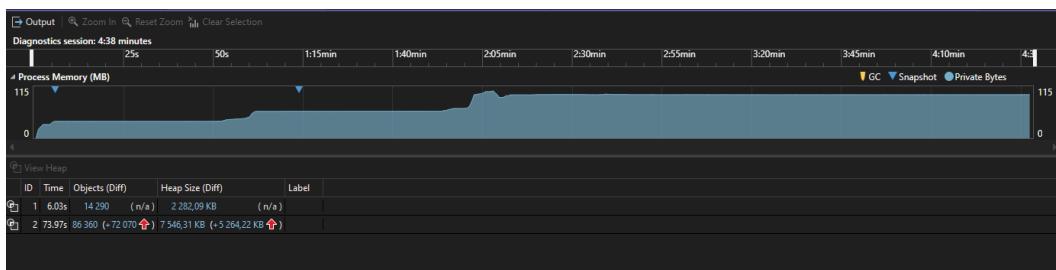


Figure 4.12 – Memory usage graph

We can take detailed snapshots of our application's memory state at different intervals using this tool. These snapshots can then be compared to pinpoint the root causes of memory issues. They showcase critical metrics such as the total number of objects and bytes in memory, along with the differences between consecutive snapshots.

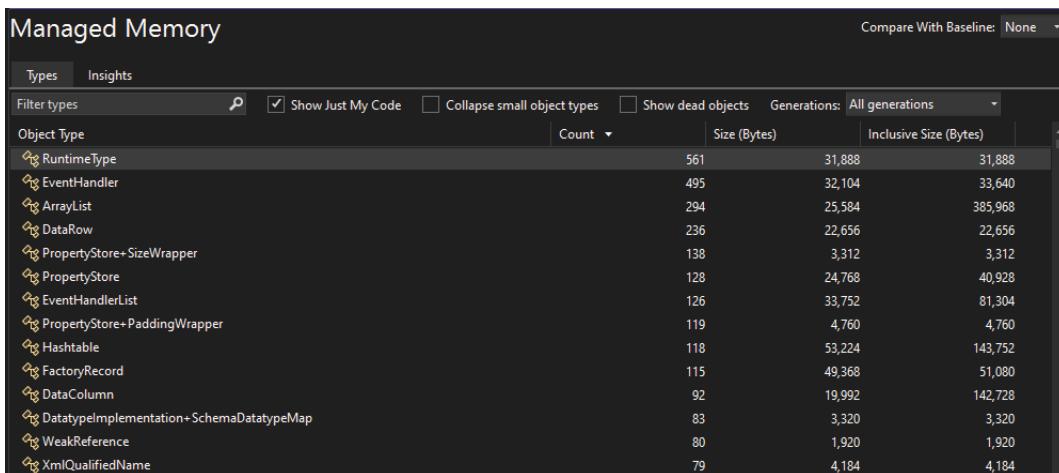


Figure 4.13 – The Memory Usage report types

We can delve deeper into these snapshots through detailed Memory Usage report views, gaining insights into the types and instances present in each snapshot or the variances between two snapshots.

Once data collection is stopped, the Memory Usage tool presents an overview page containing memory usage data. This overview helps us grasp the memory impact of our application and spot areas that could benefit from optimization.

For more advanced analysis, the Memory Usage tool provides insights into various memory issues such as duplicate strings, sparse arrays, and event handler leaks, particularly beneficial for managed memory analysis.

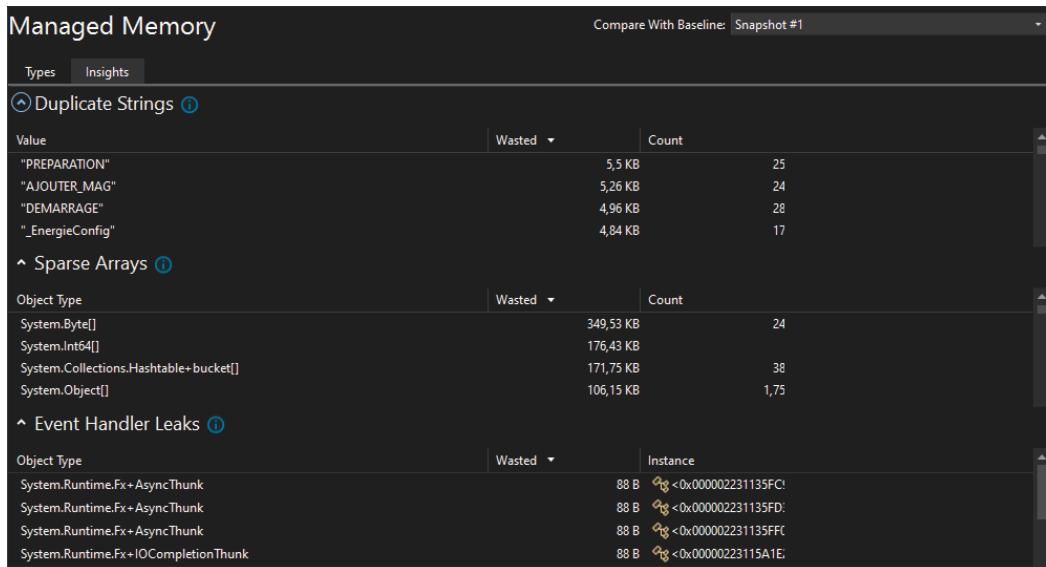


Figure 4.14 – The Memory Usage report insights

Leveraging these insights allows us to identify and resolve common memory problems more efficiently.

Some scenarios could need to focus and dig deeper into some specific part of our code base, and for that, we can use the diagnostic tools available in debugging mode.

## Exploring Memory Usage while debugging

For this part, I will create a small console app that you can retrieve on GitHub. The following code includes a `while` loop that fills a List of string with a large random string:

```
List<string> list = new();
```

```
while (true)
{
    list.Add(GenRandomStr(10000));
    Thread.Sleep(1);
}

static string GenRandomStr(int length)
{
    Random rnd = new();
    var chars = new List<char>();

    for (var i = 0; i < length; i++)
    {
        var a = (char)rnd.Next(65, 122);
        chars.Add(a);
    }

    return string.Concat(chars);
}
```

In the preceding code block, we begin by setting breakpoints in our application where we suspect the memory leak might be occurring. This could be at the start of a function or a region of code that we suspect is causing the memory leak.

For this example, since the code is straightforward, we will set breakpoints at the closing bracket of the loop.

Additionally, in more complex scenarios, we can utilize other profiler tools to identify suspect locations in our code base.

Next, we will use the **diagnostic tools**, and by default, it opens at the launch of the debugger. If not, you can reach it by navigating to the top bar menu and clicking **Debug | Windows | Show Diagnostic Tools**.

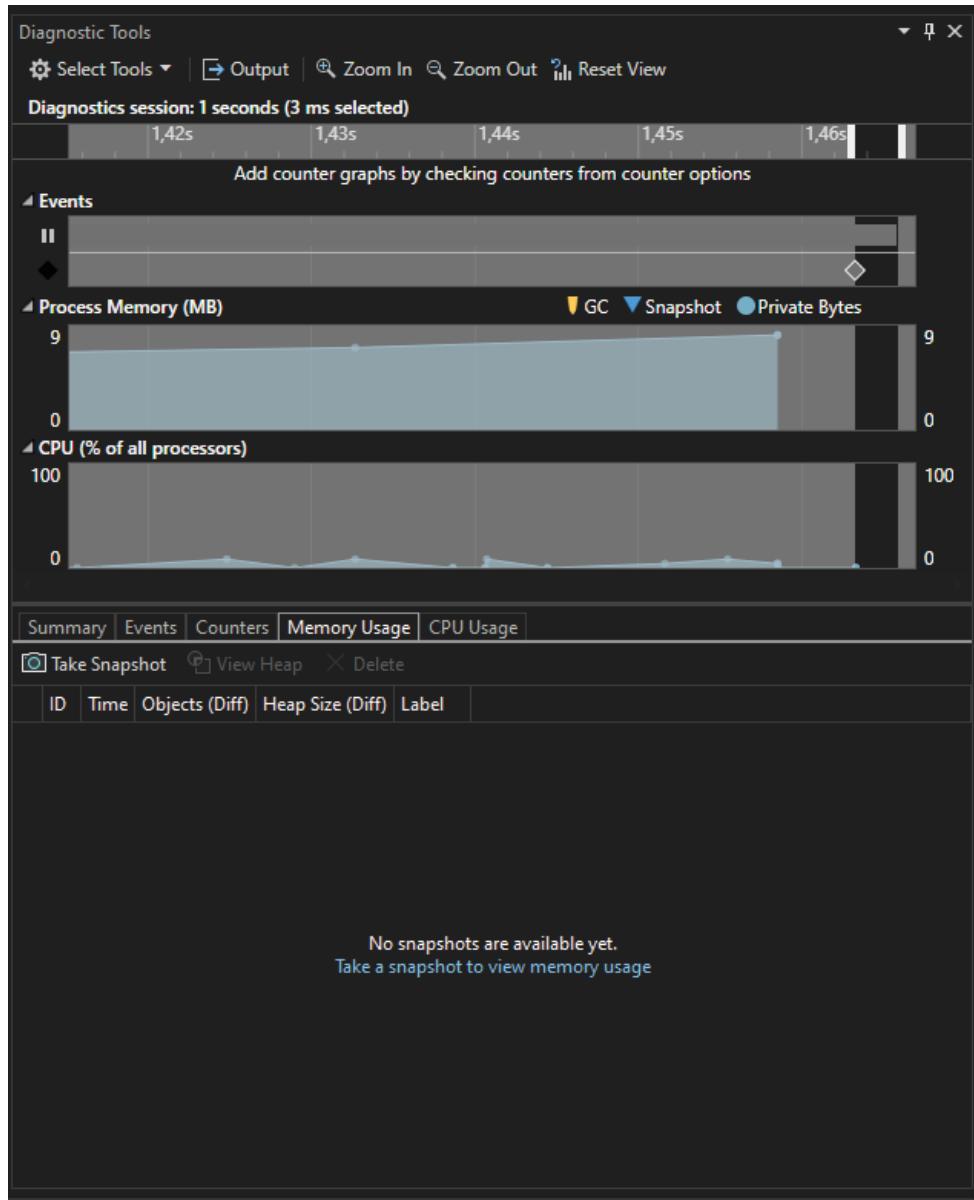


Figure 4.15 – Diagnostic Tools

In the **Diagnostic Tools** window, we retrieve the **Events**, **Process Memory**, and **CPU** usage graph. For our example, we will observe the **Process Memory** section, where memory usage increases as the loop iterates. Additionally, we can monitor the work of the garbage collector to assess its impact on memory allocation.

To analyze the heap stack, we can take a snapshot with the **Take Snapshot** option. This action will generate a report of the memory state at different intervals, as we observed in the profiler tools description previously.

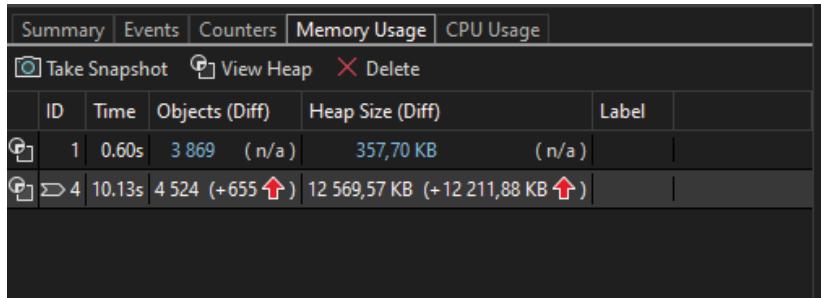


Figure 4.16 – A Memory Usage snapshot

In *Figure 4.16*, we can observe that the **Heap Size** and number of **Objects** dangerously increase between our snapshots. This indicates poor deallocation, or in other words, a memory leak.

We can dig deeper by clicking on the **View Heap** option to explore **Object Type** by size and number.

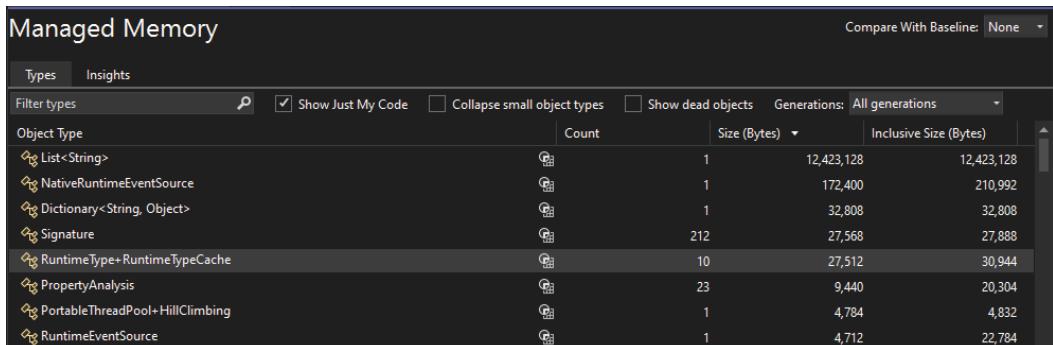


Figure 4.17 – Memory Usage

Here, we can see that the cause of our memory leak is `List<String>`, which will never deallocate.

While CPU and memory can provide insight into the performance of our application, most of the latency observed in software comes from the database interactions and in the next section, we will see how to optimize them.

## Optimizing database interactions

Visual Studio 2022 introduces a new analyzer tool named **Database Profiler**. It allows us to explore the database interactions in our application.

In this section, we will explore how to use the Database Profiler tools and how they can help us identify query optimization opportunities in our code base.

To open it, we go through **Performance Profiler** and select **Database**, where we can combine it with **CPU Usage** for more insight.

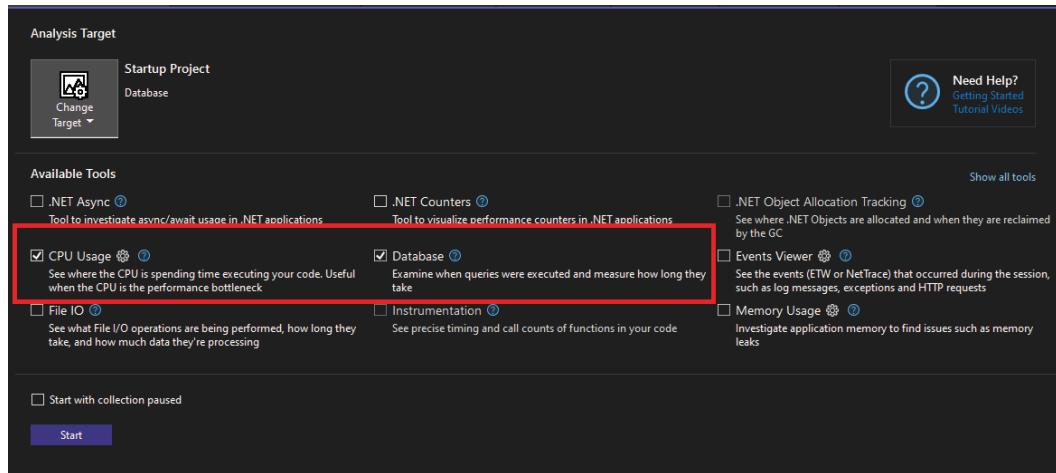


Figure 4.18 – Selecting the Database and CPU Usage tools

When we click on the **Start** button, the profiler will launch our application and start collecting data. During this time, we can perform long-running actions on our application to identify the root cause of latency.

After clicking on the **Stop Collection** button, we launch the generation of the report.

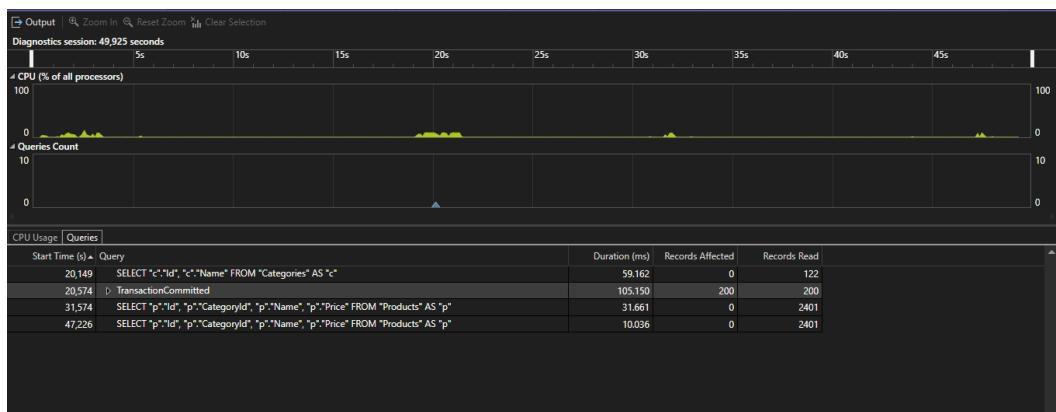


Figure 4.19 – Database report

The database report will show us a table with information about queries executed; by default, it shows columns with the following information:

- The start time of the query
- The SQL code of the query
- The duration of execution
- The number of records affected
- The number of records reads

In larger applications, we can incorporate multiple databases, such as when utilizing the **Command-Query Responsibility Segregation (CQRS)** pattern alongside database replication, for example. To help us in our investigation, we can display three more columns by right-clicking on the display one and checking which one we need.

Duration (ms)	Records Affected	Start Time (s)
59.162		<input checked="" type="checkbox"/> Start Time (s)
105.150		<input checked="" type="checkbox"/> Records Affected
31.661		<input checked="" type="checkbox"/> Records Read
10.036		<input type="checkbox"/> Database <input type="checkbox"/> Connection String <input type="checkbox"/> Query Source

Figure 4.20 – Database report managing column

The additional columns are as follows:

- **Database**
- **Connection String**
- **Query Source** displaying the data provider used (EFCore, Dapper, ADO.NET, or Others)

Coupling with the CPU Usage tools, we can easily zoom on the consuming period to examine query launch in this time.

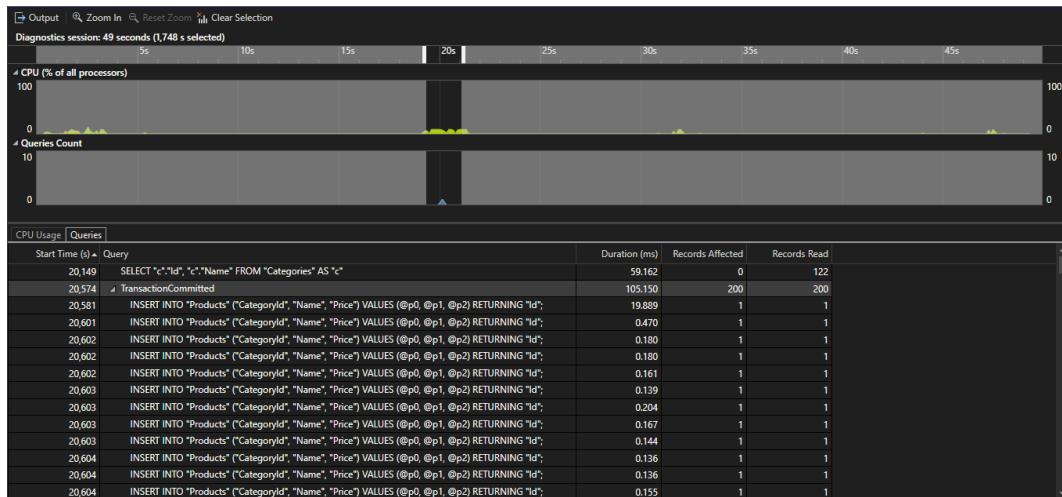


Figure 4.21 – Database report time filtered

When we identify a query that might catch our attention due to its long transaction duration or large number of associated queries, we could investigate.

Finally, in the list of queries, we can easily jump into the code source for further investigations or even refactoring, by right-clicking on the row we are interested in and selecting **Go To Source File**

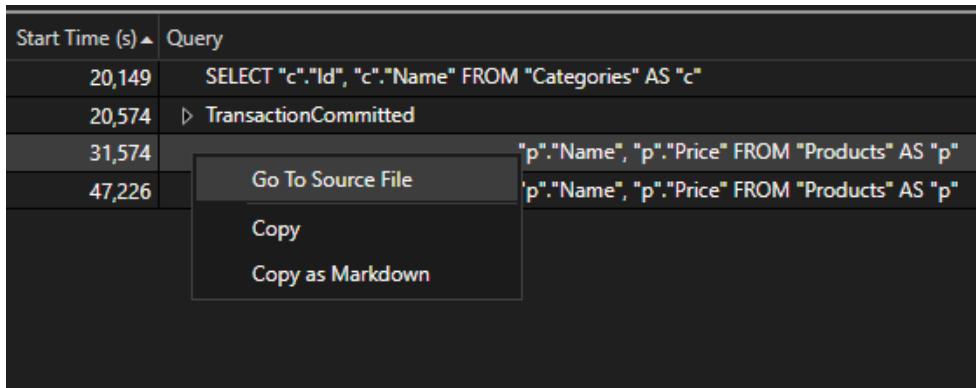


Figure 4.22 – Go To Source File

The power of Visual Studio profiling tools lies in the ability to combine some of its tools for a comprehensive investigation. My advice is to utilize the three tools highlighted in this chapter: the Memory Usage, CPU Usage, and Database tools. This will help in quickly identifying issues such as queries that generate excessive memory allocation and CPU utilization, especially when working with **Object Relational Mapping (ORMs)**, such as EF Core or Dapper, that need to instantiate objects to run queries.

## Summary

This chapter provided key insights to leverage Visual Studio profiling tools to aid in our investigation and optimization of performance bottlenecks, resulting in improved application performance.

Throughout this chapter, we've covered a range of topics, from understanding the fundamentals of performance optimization to utilizing Visual Studio's profiling tools effectively. We've learned how to analyze CPU usage and identify memory and database bottlenecks to identify and optimize critical sections and our code base for improved performance.

As we conclude this chapter, we mark the end of the first part of our journey in mastering core development skills. From unit testing and TDD to advanced debugging strategies, code analysis, and now performance optimization and profiling, you have laid a solid foundation for your development journey.

In the upcoming chapters, we'll continue to expand our horizons, delving into advanced topics such as multi-platform app UI development, advanced web development tools, machine learning integration, and advanced cloud integration and services.

To start the second part, we will dive into the world of cross-platform development by exploring tools offered by Visual Studio for MAUI.

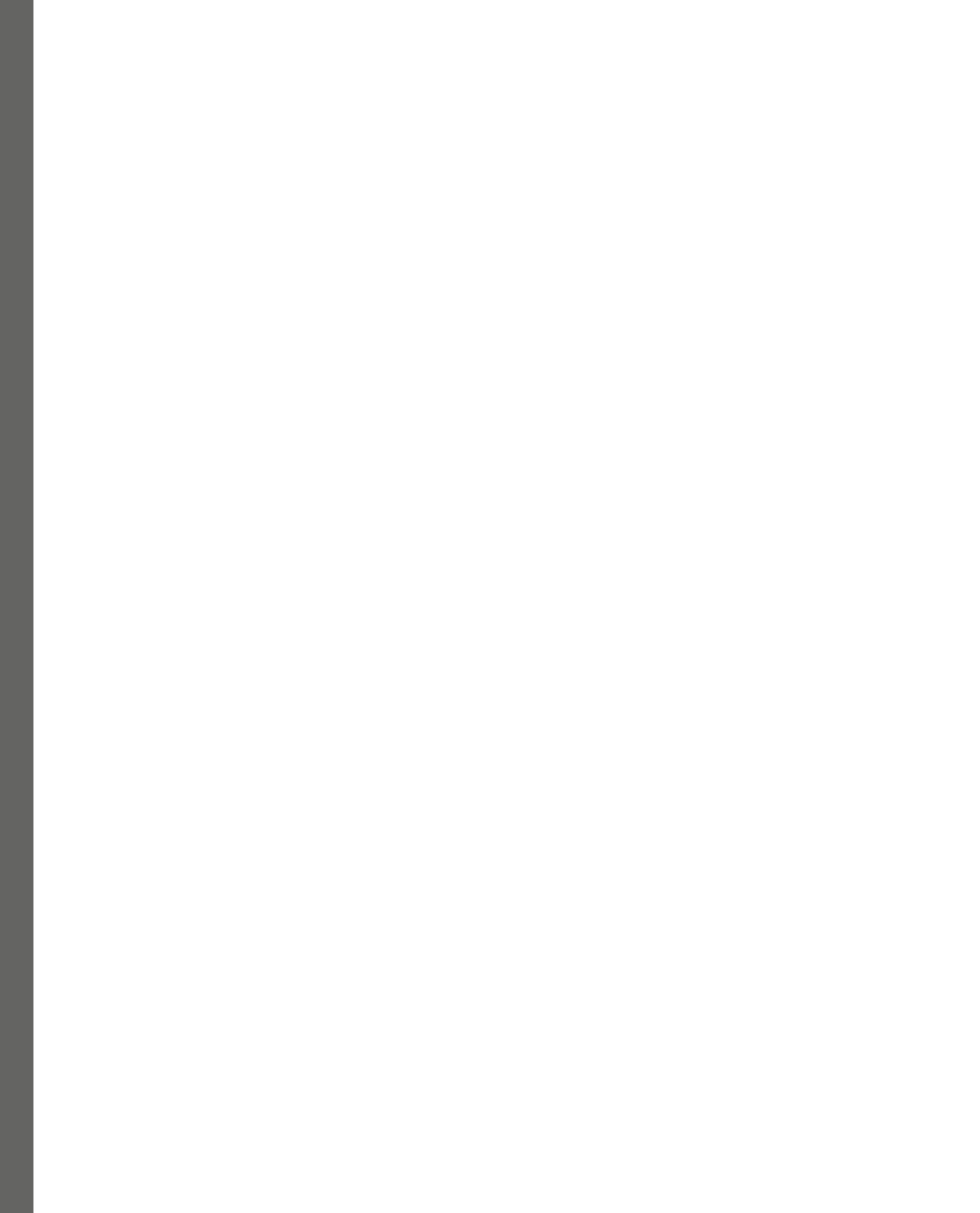


# Part 2: Advancing Development Horizons

In this second part, we focus on expanding your development expertise with advanced techniques for building versatile applications leveraging Visual Studio. From multi-platform app UI development and advanced web tools to machine learning integration and cloud services, these chapters equip you with the skills to build modern, scalable, and intelligent applications, pushing the boundaries of your development capabilities.

This part has the following chapters:

- *Chapter 5, Multi-Platform App UI Development*
- *Chapter 6, Advanced Web Development Tools*
- *Chapter 7, Machine Learning Integration*
- *Chapter 8, Advanced Cloud Integration and Services*



# 5

# Multi-Platform App UI Development

This chapter delves into the realm of cross-platform development using Visual Studio 2022, with a focus on **.NET MAUI (Multi-Platform App UI)**. We will gain comprehensive insights into building applications that seamlessly run on multiple platforms using .NET MAUI. The topics cover an introduction to .NET MAUI, an exploration of the essential tools for efficient development, debugging on various devices, and practical guidance on migrating from Xamarin to .NET MAUI.

Throughout this chapter, we'll explore the following:

- An introduction to MAUI
- Exploring tools for MAUI
- Debugging on devices
- Migration from Xamarin

By the culmination of this chapter, we'll be equipped to productively and playfully craft compelling multi-platform applications.

So, let's embark on this expedition of mastering Visual Studio for MAUI development together.

## Technical requirements

While writing this chapter, I used the following version of Visual Studio:

- Visual Studio Enterprise 2022 Version 17.12.0
- Preview 1.0

The code files for this chapter can be found at <https://github.com/PacktPublishing/Mastering-Visual-Studio-2022/tree/main/ch05>.

To access the WordPuzzle project in this chapter, check out the following GitHub repository at <https://github.com/dotnet/maui-samples/tree/main/8.0/Apps/WordPuzzle>.

## An introduction to MAUI

**MAUI**, is a versatile framework that enables developers to craft native applications for both mobile and desktop platforms, using C# and XAML. With MAUI, developers can target Android, iOS, macOS, and Windows devices from a unified code base.

This open-source framework builds upon the foundation of Xamarin.Forms, extending its capabilities to cover desktop scenarios while aiming for enhanced performance, flexibility, and the ability to consolidate app logic and UI layout within one code base.

## The evolution of cross-platform development

Cross-platform development frameworks have revolutionized app development by enabling code sharing across multiple platforms. Xamarin.Forms, for instance, has empowered companies to share over 95 percent of their code base, maximizing development investments. However, the landscape has evolved, prompting the emergence of frameworks such as MAUI to address previous limitations and adapt to the evolving demands of app development.

## What is MAUI?

Developed by Microsoft, MAUI is an open source framework designed to facilitate the creation of native cross-platform apps using a single code base. It builds upon the foundation of Xamarin.Forms, enriching it with additional features, such as a unified project structure, and flexibility.

Leveraging .NET 6, MAUI integrates the latest advancements, performance enhancements, and security updates from the .NET framework, enabling developers to harness the extensive ecosystem of .NET libraries and tools to elevate their applications.

## The key features of MAUI

Some of the key features of MAUI are as follows:

- **A single code base for multiple platforms:** MAUI streamlines project management by consolidating code, resources, and UI elements within a single project structure, reducing complexity, and saving time
- **Enhanced performance and responsiveness:** Engineered for improved performance, MAUI ensures smooth and efficient operation across all supported platforms
- **Support for native features and controls:** MAUI prioritizes native user interfaces, delivering a seamless user experience on all platforms
- **Improved developer productivity:** Features such as Hot Reload for XAML and CSS enable real-time visualization of UI changes, accelerating the development process significantly

Now that we've explored the key features and architecture of .NET MAUI, it's clear that this framework represents a significant leap forward in app development.

## The architecture of MAUI

MAUI adopts a strategy where it leverages the native capabilities of each platform while maintaining a single code base. This approach grants developers access to native platform features and UI controls through a unified API, facilitating the delivery of uncompromised user experiences while achieving higher code reuse.

.NET MAUI offers a unified framework for constructing user interfaces across both mobile and desktop applications.

The diagram in *Figure 5.1* (from [learn.microsoft.com](https://learn.microsoft.com/en-us/dotnet/maui/architecture/high-level-structure)) illustrates the high-level structure of a .NET MAUI application:

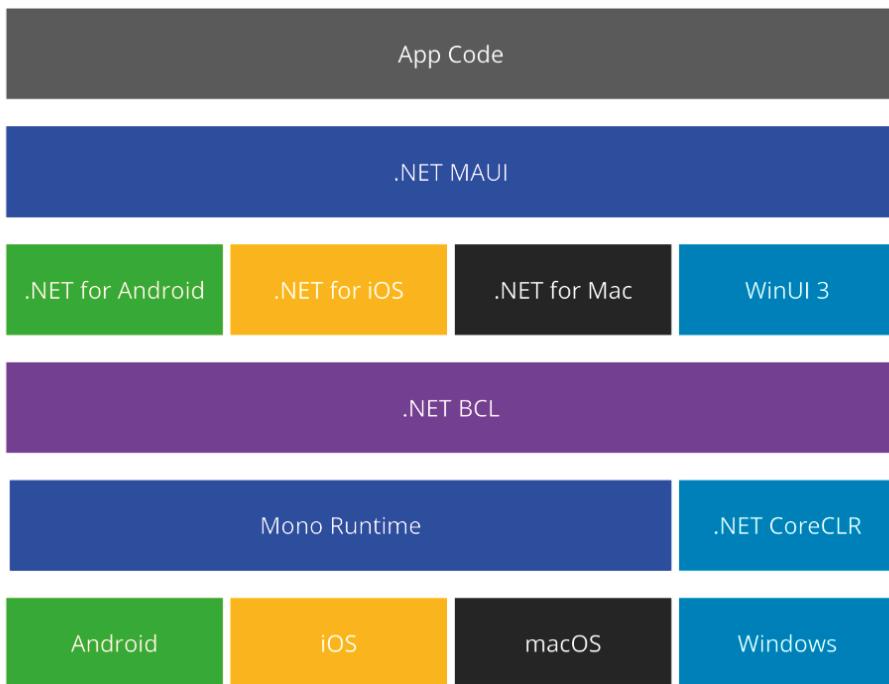


Figure 5.1 – The MAUI high-level structure (source: [learn.microsoft.com](https://learn.microsoft.com/en-us/dotnet/maui/architecture/high-level-structure))

We can craft .NET MAUI applications on either a PC or a Mac. In our .NET MAUI app, our code will primarily interact with the .NET MAUI API. Then, the .NET MAUI subsequently accesses the native platform APIs directly. Moreover, our application code may directly engage with platform APIs when necessary.

By the end, the code is compiled into native app packages:

- .NET MAUI facilitates the development of Android applications by compiling them from C# into an **intermediate language (IL)**. This IL code is subsequently JIT-compiled into a native assembly upon an application's launch. In contrast, iOS applications developed with .NET MAUI undergo **ahead-of-time (AOT)** compilation directly from C# into native ARM assembly code before deployment.
- For macOS applications, .NET MAUI taps into Mac Catalyst, a feature provided by Apple. This feature permits iOS apps, constructed using UIKit, to operate on macOS platforms. Additionally, it supports the integration of AppKit and necessary platform-specific APIs.
- When it comes to Windows applications, .NET MAUI employs the **Windows UI 3 (WinUI 3)** toolkit. This approach enables the creation of native Windows desktop applications.

Now that we have an understanding of what MAUI is, let's dive into exploring the tools offered by Visual Studio to enhance our MAUI experience.

## Exploring the tools for MAUI

In this section, to explore the tools offered by Visual Studio for MAUI, we will start by creating a basic application. By building this application, we will gain hands-on experience with the various tools and features available in Visual Studio for MAUI development. This practical approach will ensure that we not only understand the purpose of these tools but also learn how to effectively utilize them, enhancing our development workflow.

### Creating a simple MAUI app

First, we must ensure that we have the latest version of Visual Studio 2022 with the MAUI workload installed. If you don't have it installed, you can update your Visual Studio from the Visual Studio Installer or with the following command:

```
dotnet workload install maui
```

You can also use the Visual Studio Installer to install the MAUI workload or check whether the version you have is up to date, by launching your Visual Studio Installer and clicking on the button to modify your installed Visual Studio (see *Figure 5.2*):

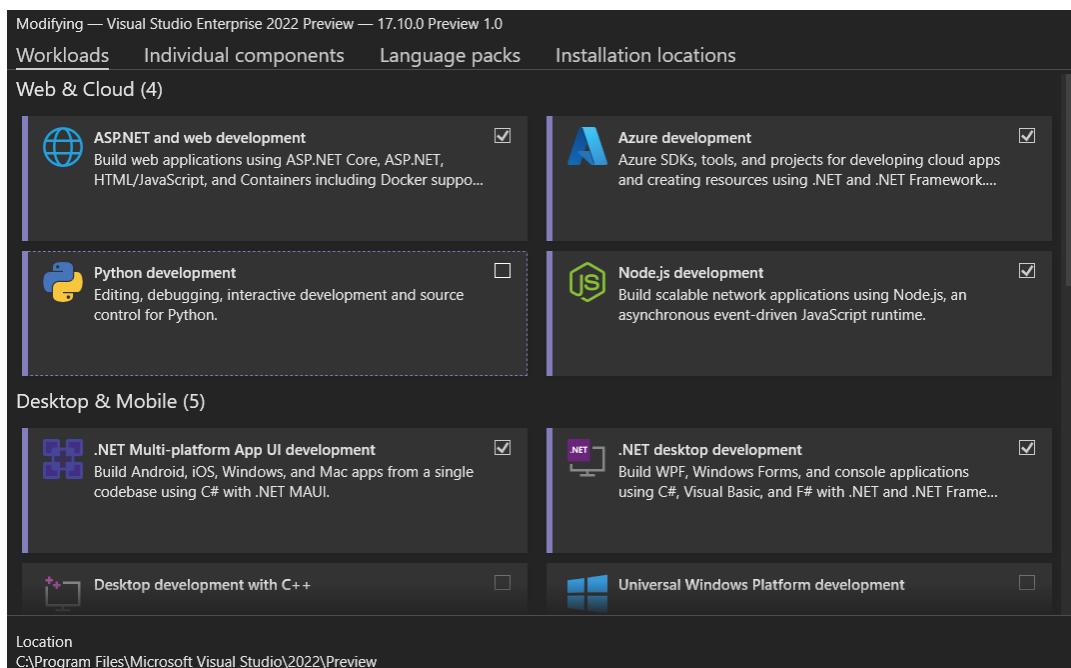


Figure 5.2 – Modifying Visual Studio’s workload

Now, we are all set. We can create our new MAUI project by following these steps:

1. Open Visual Studio and create a new project. Select **MAUI** as the project type.

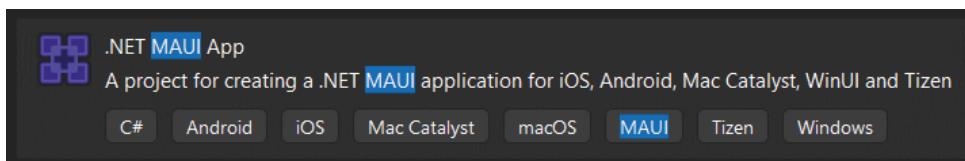


Figure 5.3 – The .NET MAUI App template

2. We can see that an MAUI project has a shared code base that targets multiple platforms. In a .NET MAUI setup, the common code base resides within the shared project. Conversely, each platform-specific project houses the unique code tailored to its respective environment (e.g., **Platforms | Android**, **Platforms | iOS**, **Platform | Tizen**, **Platforms | MacCatalyst**, and **Platforms | Windows**), as shown in *Figure 5.4*:

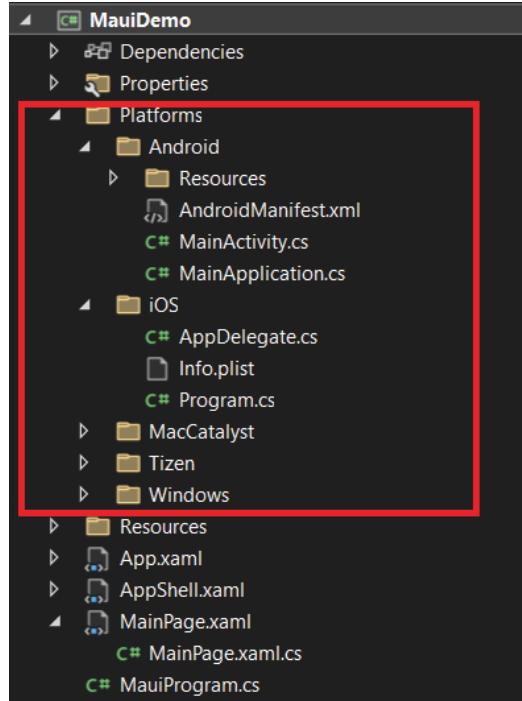


Figure 5.4 – The MAUI folder architecture

3. In the **Shared** project, we find the main page's XAML file, under `MainPage.xaml`. This is where we design our app's UI.

For our example, we will simply build an app, with a button to click on to show a text message.

We will begin by designing the UI. To do that, we open `MainPage.xaml` in the **Shared** project and replace its content with the following XAML code:

```
<ContentPage xmlns=>http://schemas.microsoft.com/dotnet/2021/maui
    xmlns:x=>http://schemas.microsoft.com/winfx/2009/xaml
    x:Class="MyMauiApp.MainPage">

    <StackLayout>
        <Button x:Name="MyButton"
            Text="Click Me"
            Clicked="OnButtonClicked"/>
        <Label x:Name="MessageLabel"
```

```
        Text="Welcome to MAUI!"/>
    </StackLayout>

</ContentPage>
```

This XAML code snippet defines the basic structure of `ContentPage` in our .NET MAUI application. This page contains `StackLayout`, which is a simple layout that arranges its children in a single line that can be oriented horizontally or vertically.

After this, we implement the logic by opening `MainPage.xaml.cs` in the **Shared** project and adding the following C# code:

```
using Microsoft.Maui.Controls;

namespace MyMauiApp
{
    public partial class MainPage : ContentPage
    {
        public MainPage()
        {
            InitializeComponent();
        }

        private void OnButtonClicked(
            object sender,
            EventArgs e)
        {
            MessageLabel.Text = "You clicked the button!";
        }
    }
}
```

This is C# code-behind for the `MainPage` class in a .NET MAUI application. This code complements the XAML markup you shared earlier, defining the behavior of `ContentPage`. Specifically, it includes the implementation of the `OnButtonClicked` method, which updates the text of a label named `MessageLabel` when a button is clicked.

Finally, we can run the app. The **Start** button allows us to select an emulator or device as the target (see *Figure 5.5*):

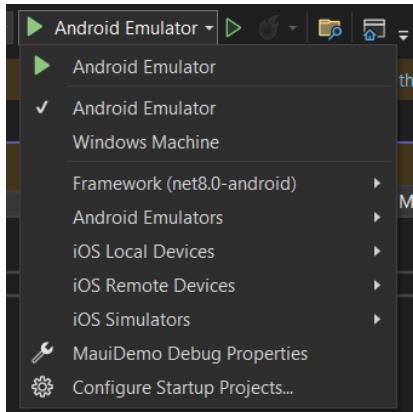


Figure 5.5 – Launching Android Emulator

For instance, we can choose **Android Emulator**; when we launch it for the first time, Visual Studio will prompt us with a wizard to install it.

Now, when we launch our app, we will see a button labeled **Click Me** (see *Figure 5.6*).

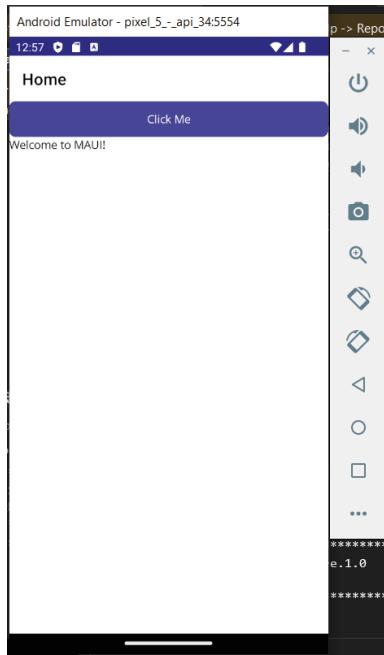


Figure 5.6 – Android Emulator

When we click the button, the text below it changes to **You clicked the button!**.

This is a basic example to get us started with MAUI. Now that we have a running app, we can explore all the tools offered by Visual Studio to enhance our MAUI development experience.

Next, let's explore XAML Live Preview to leverage instant UI modification.

## XAML Live Preview

The **XAML Live Preview** feature in Visual Studio, when used with the Windows machine executer, allows developers to see real-time updates of their XAML UI changes without the need to manually save or rebuild a project. This feature is particularly useful for quickly iterating on UI designs and ensuring that the changes are reflected immediately in the preview window.

To enable **XAML Live Preview**, look for the *video camera* icon in the *Visual Studio toolbar*. This icon is the **XAML Live Preview** button. Clicking on it will enable the Live Preview feature.



Figure 5.7 – The XAML Live Preview button

Note that we can open the **XAML Live Preview** directly through the Visual Studio top bar menu in **Debug** mode; navigate to **Debug | Windows | XAML Live Preview**.

This action allows us to dock an XAML preview window alongside Visual Studio, adjacent to our code base. As we modify the XAML code, the changes are reflected in the XAML Live Preview window almost instantly. This allows us to see the impact of our changes immediately, without the need to save or rebuild a project.

While hovering over an element, we can retrieve a variety of style information and identify the file in which the element is defined.

In this example, we hover over the `MyButton` element:

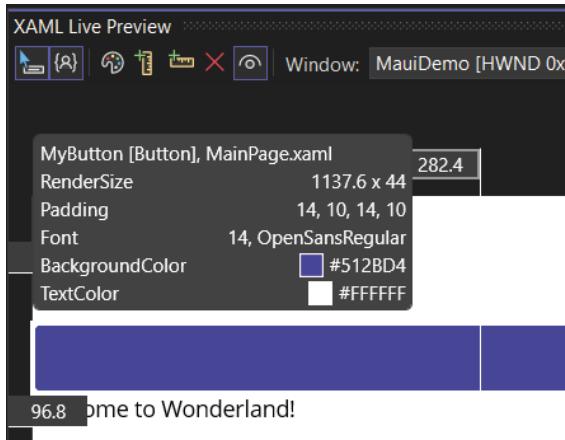


Figure 5.8 – An element's style properties

In *Figure 5.8*, we can retrieve properties such as `TextColor` and `BackgroundColor` value of the `MyButton` define in `MainPage.xaml`.

Further, we can smoothly interact with the preview by scrolling and zooming. Besides scrolling with the scroll bars, we can also utilize the following interactions:

- A *mouse wheel*, both vertical and horizontal (if our mouse supports it)
- A *touchpad* two-finger scroll, both vertical and horizontal
- A *Ctrl* key press, paired with a *mouse drag* action

In addition, for zooming, we can use the following interactions:

- The **Zoom in** or **Zoom out** buttons in the bottom-left corner.
- A *Ctrl + plus sign (+)* or *Ctrl + minus sign (-)* keyboard shortcut press, if we prefer using a keyboard.
- A *Ctrl* key press paired with a *mouse wheel* action, or a *pinch-to-zoom* action with the *touchpad*.  
A bonus of using a mouse is maintaining more precise control.

Additionally, we can add rulers to help us to define the size and position of our element. **Rulers** assist us in aligning elements within our application. They present distances, measured in application units. This feature helps us verify the spacing between different components of our application.

The second set of toolbar buttons governs the rulers, as shown in *Figure 5.9*:

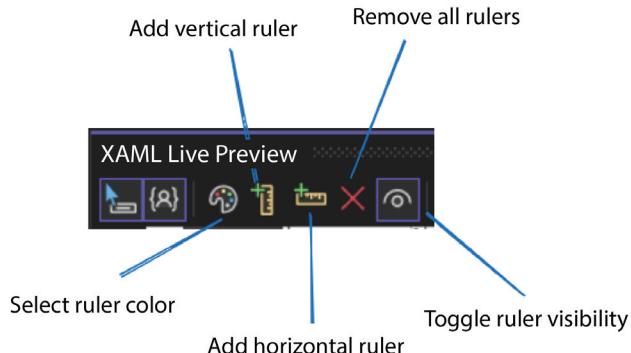


Figure 5.9 – The rulers toolbar

Here's a list of the main functionalities offered by the toolbar:

- **Add vertical ruler:** This adds a single vertical ruler. Clicking this button multiple times consecutively will position new rulers to avoid overlapping existing ones.
- **Add horizontal ruler:** This inserts a single horizontal ruler, functioning similarly to the vertical ruler.
- **Remove all rulers:** This clears all rulers at once.
- **Select ruler color:** This adjusts the color of the rulers.
- **Toggle ruler visibility:** This toggles the visibility of all rulers with a single click.

Rulers are designed to be keyboard-friendly. We can navigate through them using the *Tab* key. Utilizing the *arrow* keys allows us to move rulers one pixel at a time, or we can press *Ctrl* along with the *arrow* keys to move them by 10 app units at a time. Pressing the *Del* key deletes the currently selected ruler. Additionally, we can delete a ruler using the mouse, by selecting the **Delete Ruler** button.

We can also add rulers to an element while using **Element Selection**. Right-clicking adds vertical rulers, and holding the *Shift* key while right-clicking adds horizontal rulers.

When working in an application with multiple windows, we can choose which window to display by using the **Window combo** box. Alternatively, we can use the **Show in XAML Live Preview** button in the application toolbar that's on the window you want to preview, as we saw previously.

The XAML Live Preview feature significantly speeds up the development process by providing immediate feedback on UI changes, making it easier to design and debug XAML-based applications such as MAUI.

XAML Live Preview provides an selecting elements feature that mirrors the selection process in a running application. This functionality enables us to locate elements in either the Live Visual Tree or within the source XAML. Live Visual Tree is another tool that helps us understand the XAML structure of our MAUI applications.

Let's explore the possibilities offered by the Live Visual Tree.

## The Live Visual Tree

The **Live Visual Tree** feature in Visual Studio provides a real-time, hierarchical view of the UI elements in our application, allowing us to inspect and modify the UI structure directly within the IDE. This feature is particularly useful for debugging layout issues, understanding the visual tree, and making dynamic changes to your UI during development.

If you want to follow with the same code base to illustrate the functionality of this feature, with a more complex project, I cloned the WordPuzzle application of the GitHub repository: <https://github.com/dotnet/maui-samples/tree/main/8.0/Apps/WordPuzzle>.

### *Using the Live Visual Tree feature*

To utilize the Live Visual Tree feature, we need to debug our application. By default, the **Live Visual Tree** window is located on the left side of the IDE for Windows and on the right side for Mac. If you can't find it, you can display it by navigating to the top menu bar – **Debug | Window | Live Visual Tree**.

While open, the **Live Visual Tree** window shows the hierarchy of the UI elements that we can easily explore and follow, seeing how our application layouts are presented (see *Figure 5.10*):

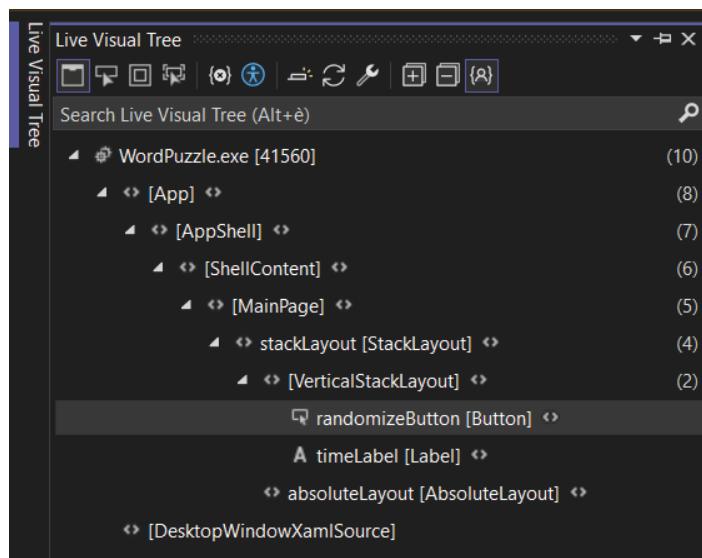


Figure 5.10 – Live Visual Tree

Now that we have introduced the Live Visual Tree, let's explore the top menu bar.

### **Exploring the Live Visual Tree top menu bar**

The Live Visual Tree *top menu bar* provides some handy features to navigate through the code base and help us debug our layouts.

*Figure 5.11* shows the Live Visual Tree top menu bar:

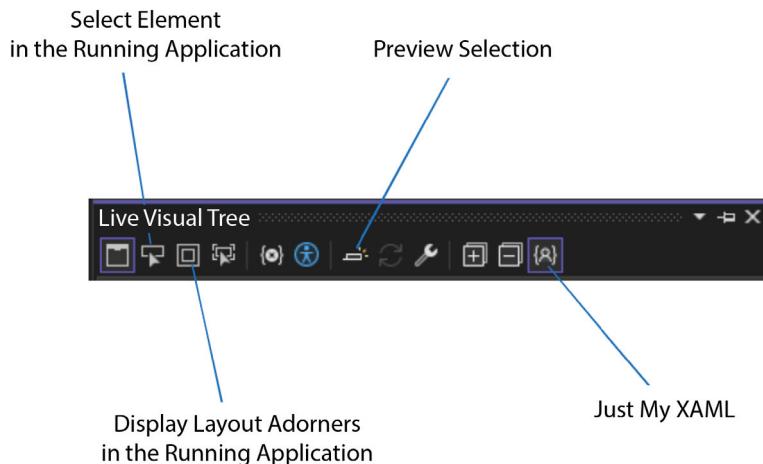


Figure 5.11 – The Live Visual Tree menu

Let's explore these features of the Live Visual Tree with a comprehensive overview of how we can utilize them:

- **Select Elements in the Running Application:** This feature enables us to pick a UI element within the application.

The Live Visual Tree then automatically updates to display the corresponding node in the tree, along with its properties. This functionality is invaluable for swiftly identifying and inspecting specific elements within our application.

- **Display Layout Adorners in the Running Application:** This mode showcases horizontal and vertical lines outlining the bounds of the selected object, along with rectangles indicating the margins. These visual aids make it easy to align and space UI elements, aiding in the identification of layout issues.
- **Preview Selection:** This mode reveals the XAML declaration of the selected element, provided we have access to an application's source code. It's a convenient feature for swiftly navigating to the source code of a selected element in the Live Visual Tree.

- **Just My XAML:** By default, the Live Visual Tree streamlines the view of XAML elements using the **Just My XAML** feature. This functionality hides elements that we're probably not directly interested in, simplifying navigation through the tree. We can toggle this feature on and off using the **Show Just My XAML** button on the **Live Visual Tree** toolbar. Note that Visual Studio for Mac does not currently support this feature.

Both the Live Tree View and XAML preview features are seamlessly supported by the Hot Reload feature provided by Visual Studio 2022. **Hot Reload** works by monitoring the source code for changes. When a change is detected, Hot Reload applies the changes to the running application without requiring a full rebuild. This means we can see the effects of our changes immediately, without losing the current state of the application.

Now that we have explored the main tooling to enhance our experience with MAUI development, in the next section, we'll see how we can debug our application directly on devices.

## Debugging on devices

As we have seen earlier in this chapter, we can easily run our app as a **universal windows platform (UWP)** or through an emulator of each existing device's OS. Debugging a .NET MAUI app directly on a device using Visual Studio involves several steps. This process allows us to test our app on a real device, which can be particularly useful for testing features that require specific hardware capabilities, or for performance testing.

In this section, we will follow a step-by-step guide on how to do debugging on a device.

### Enabling Developer Mode on our device

To debug your MAUI application on a device, it's essential to first activate **Developer Mode**. However, the steps to do this differ, based on the device's OS.

Let's look at the steps to enable Developer Mode on Windows, Android, and iOS devices:

- **For Windows devices:** Go to **Settings | Update and Security | For Developers** and select **Developer Mode**.
- **For Android devices:** Navigate to **Settings | About phone | Software information** and select the **Build number** option, tapping it seven times to unlock **Developer Options**. Afterward, return to **Settings | Developer options** and turn on USB debugging.
- **For iOS devices:** You need to have a Mac with Xcode installed. Connect your iOS device to your Mac, open Xcode, go to **Window | Devices and Simulators**, select your device, and enable **Connect via network**.

The position of the **Developer Mode** option can vary, depending on the user interface of your device. If you're having trouble locating USB debugging, it is helpful to refer to your device's manual for guidance.

## Networking device

The final step is to connect the device to the computer. This process will depend on our device. Let's explore both main use cases.

### Android

When using Android, it's preferable to use USB, as it's the easiest and most reliable method.

Once connected, your device will ask you to confirm whether you trust the computer if it's the first time you've debugged on it. You also have the option to select **Always allow from this computer** to avoid this prompt in the future.

Alternatively, it's possible to debug an Android device over Wi-Fi, eliminating the need for a physical connection to the computer. Although this method requires more effort to set up, it can be beneficial when the device is situated far from the computer and a constant cable connection is impractical.

### iOS with Hot Restart

**Hot Restart** is a feature in Visual Studio 2022 that allows you to quickly redeploy your app to an iOS device without having to go through the full build and deploy process. This can significantly speed up your development cycle, especially when you're making frequent changes and need to test them on a device. Here's how to use Hot Restart for iOS deployment in Visual Studio 2022.

To set up Hot Restart, follow these steps:

1. First, in the Visual Studio toolbar, choose **iOS Local Devices** from the **Debug** target drop-down menu, and then select **Local Device**.

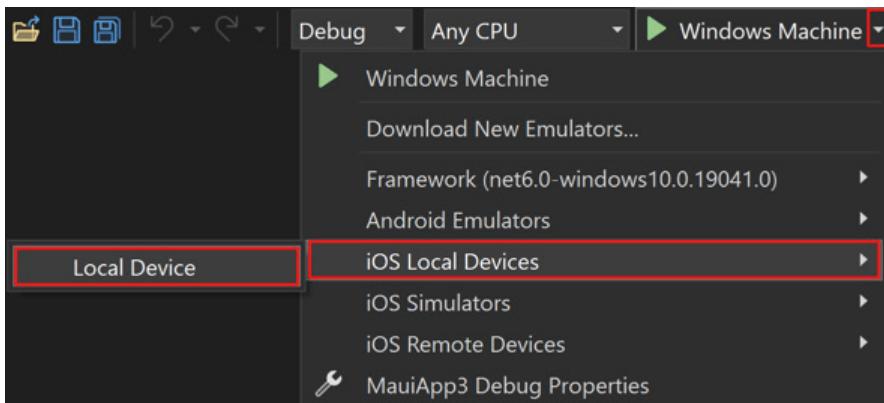


Figure 5.12 – iOS Local Devices

This action prompts the **Setup Hot Restart** setup wizard to launch on the start of the debugging, guiding us through the process of setting up a local iOS device for Hot Restart deployment. Then, select **Next** to proceed.

2. If iTunes isn't installed, the setup wizard prompts us to download it. We can install iTunes either from the Microsoft Store or Apple's website.
3. Next, connect your iOS device to your development machine using a USB cable, and trust your development machine when prompted on the device. Once your local iOS device is detected, click **Next** in the setup wizard.
4. Configure **Hot Restart** to use your individual Apple Developer Program account by clicking the **Sign in with an individual account** hyperlink and providing your App Store Connect API key data. Clicking **Finish** completes the setup wizard, adding your Apple Developer Program account to Visual Studio.
5. In **Solution Explorer**, right-click on your project and select **Properties**. Under **iOS | Bundle Signing**, choose **Automatic Provisioning** from the **Schema** drop-down menu and click **Configure Automatic Provisioning**.
6. In the **Configure Automatic Provisioning** dialog, select the team for your Connect API key, and then Visual Studio completes the automatic provisioning process.
7. Finally, click **OK** to dismiss the dialog.

The final step is to launch the debugging session targeting our device.

## Launching the debugging session

Now, Visual Studio will recognize our device, and the option to debug directly on it appears.

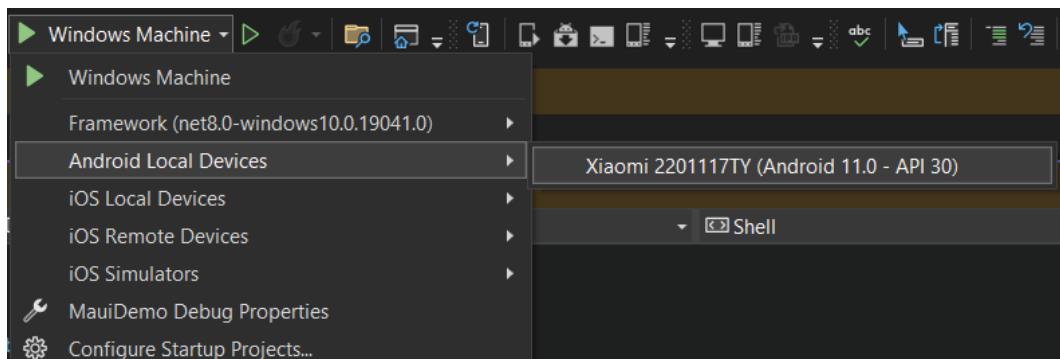


Figure 5.13 – Android Local Devices

When we select the **Debug-on-Device** option, our application installs on the phone. Subsequently, we must initiate the application to commence debugging, similar to how we do it on emulators or Windows machines.

After learning about MAUI, exploring the tools available in Visual Studio, and debugging on local devices, you may find yourself with an existing Xamarin project to migrate to MAUI. Let's explore how to leverage Visual Studio extensions to achieve this effortlessly.

## Migrate from Xamarin

In this section, we will see how to migrate a legacy project written with Xamarin to a cutting-edge MAUI application. First, we will explore the key difference between MAUI and Xamarin. Then, finally, we will install the .NET Upgrade Assistant to use to migrate our project.

### Understanding the key differences

MAUI is considered the successor to Xamarin; however, there are some key differences between them.

**Xamarin** is a framework within the .NET ecosystem that enables developers to create cross-platform applications using C# and .NET. It supports major platforms such as Android, iOS, and Windows (UWP). Xamarin facilitates the development of native applications, ensuring high performance and native user experiences by compiling code into platform-specific native binaries.

Conversely, MAUI is the next iteration of Xamarin.Forms, designed to streamline cross-platform development further. It expands platform support to include Android, iOS, macOS, and Windows, aiming for a unified code base across these platforms. MAUI prioritizes modern development practices, focusing on performance optimization and delivering native-like experiences.

Xamarin.Forms provide a comprehensive set of UI controls and layouts that are rendered natively on each supported platform. While developers share code across platforms, Xamarin.Forms ensure that UI elements conform to the native look and feel of each platform, ensuring consistency across devices.

Further to this, MAUI introduces a fresh collection of controls and layouts designed to enhance efficiency and flexibility. It aims to deliver a modern, unified UI experience across all supported platforms, bridging the gap between native and cross-platform app development.

In order to migrate our legacy Xamarin project, we have two options – doing it manually or using the Upgrade Assistant. Let's explore how to use the .NET Upgrade Assistant.

### Using the .NET Upgrade Assistant

One of the options to migrate our project is to modify each file of a Xamarin project to MAUI by creating a new MAUI project. This way could be time-consuming and need a complete understanding of the project. However, it's a great option for large and complicated projects.

Alternatively, we can use the **.NET Upgrade Assistant** to facilitate our process. The .NET Upgrade Assistant can be installed either as a Visual Studio extension or a .NET command-line tool. Here, in our example, we will use it as a Visual Studio extension.

The .NET Upgrade Assistant is available for installation as a Visual Studio extension or a .NET command-line tool through the following methods:

1. While Visual Studio is active, navigate to **Extensions | Manage Extensions** to open the **Manage Extensions** window.
2. In the **Manage Extensions** window, search for upgrade in the search box.
3. Choose the **.NET Upgrade Assistant** option, and then click **Install**.

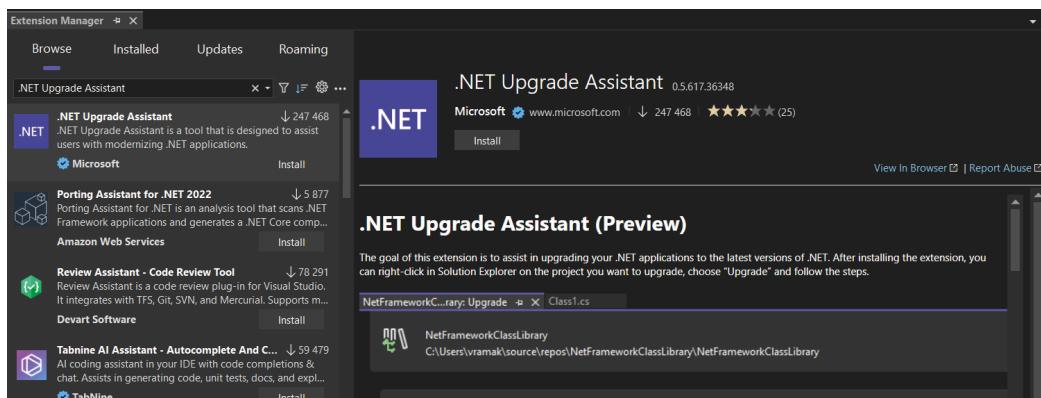


Figure 5.14 – Installing the extension .NET Upgrade Assistant

4. Once the extension finishes downloading, the installation will be launched when Visual Studio is closed, and then follow the instructions to install it.
5. After installation, right-click on the project node in the **Solution Explorer** and select **Upgrade**.

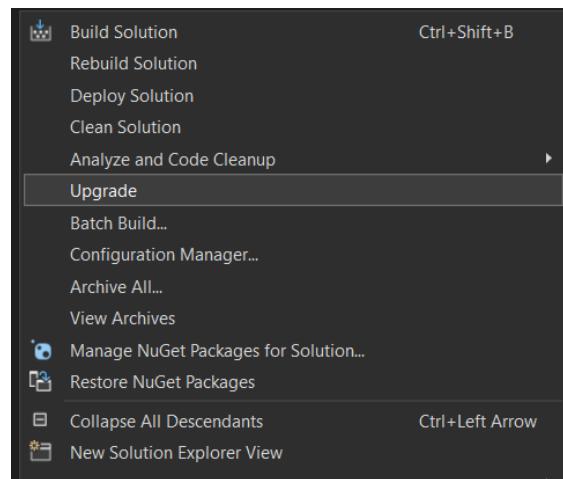


Figure 5.15 – Select Upgrade

6. That will open up a window to select between two options (see *Figure 5.16*):
- **In-place project upgrade:** This will replace the legacy project with the new one.
  - **Side-by-side project upgrade:** This will create a new project containing the new MAUI project.

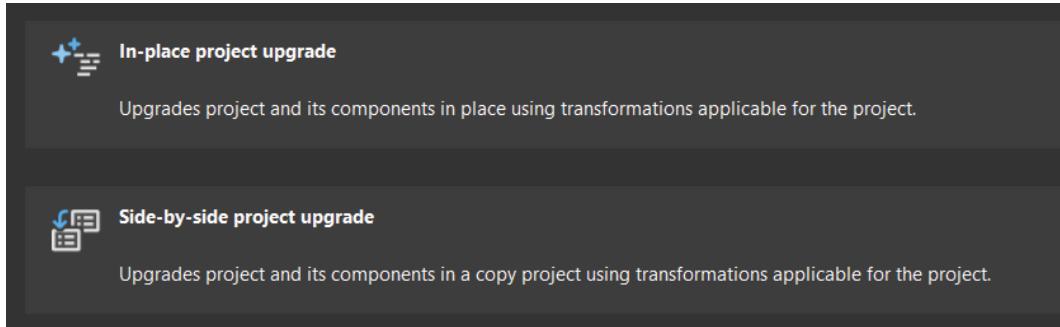


Figure 5.16 – The upgrade options

I advise choosing the **Side-by-side project upgrade** option to ensure the safety of our working Xamarin project until the upgrade has been successfully completed.

Then, we just need to follow the steps of the wizard to complete the upgrade. The upgrade assistant is a good option for relatively simple projects. However, at the end of the process, we may need to make some manual adjustments.

For complex and large projects, the upgrade should be done manually, following the step-by-step instructions found in books dedicated to MAUI.

## Summary

In this chapter, we embarked on a journey into the depths of multi-platform app UI development using Visual Studio 2022, with a focus on .NET MAUI. From unraveling the essence of .NET MAUI to mastering essential tools for efficient development, debugging across diverse devices, and seamlessly transitioning from Xamarin, we've equipped ourselves with the knowledge and skills necessary to craft compelling multi-platform applications.

Throughout our exploration, we've gained a comprehensive understanding of .NET MAUI's core principles and significance in modern application development. We've delved into the arsenal of tools tailored for .NET MAUI development within Visual Studio 2022, harnessing their power to create, preview, and debug applications, with confidence and precision.

In the next chapter, we'll explore the latest features of Visual Studio 2022 to enhance our web development experience.



# 6

# Advanced Web Development Tools

In this chapter, we will embark upon an advanced exploration of web development tools within the robust environment of Visual Studio 2022. Our exploration will begin with an examination of a web form designer powered by Web Live Preview, which offers us a way to visualize and iterate on our designs while working on legacy ASP.NET web form applications. Next, we will turn our attention to the dynamic capabilities of Visual Studio's API Exploration and Dev Tunnel functionalities, enhancing our API development experience. Further enriching our toolkit, we will investigate the Node.js workload of Visual Studio.

In this chapter, we'll cover the following topics:

- Real-time web previews powered by Web Live Preview
- API Exploration and Dev Tunnel with Visual Studio
- Node.js integration with Visual Studio

By the end of this chapter, armed with these advanced web development tools and techniques, we will be equipped to elevate our productivity for our web projects to new heights.

## Technical requirements

While writing this chapter, I used the following version of Visual Studio:

- Visual Studio Enterprise 2022 version 17.12.0
- Preview 1.0

The code files for this chapter can be found at <https://github.com/PacktPublishing/Mastering-Visual-Studio-2022/tree/main/ch06>

## Real-time web previews powered by Web Live Preview

Microsoft announced that ASP.NET, when used with old frameworks, including web form applications, will continue to support old .NET Framework versions. Following that, Visual Studio 2022 has introduced a designer for Web Forms projects that are powered by Web Live Preview. In this section, we will explore how to utilize the new Web Forms designer along with additional features offered by Web Live Preview. This feature is specifically tailored for teams handling legacy projects.

As this feature is currently dedicated only to Web Forms in .NET Framework, if you are using MVC and .NET Core, there is no forms designer available. To stay up to date with the latest web standards and browser support, Web Live Preview in Visual Studio uses WebView2, which is powered by Microsoft Edge.

To illustrate our examples, we will create a project using *ASP.NET (.NET Framework)*.

If we open the default .aspx file or any other .aspx file, we'll notice two buttons at the bottom left:

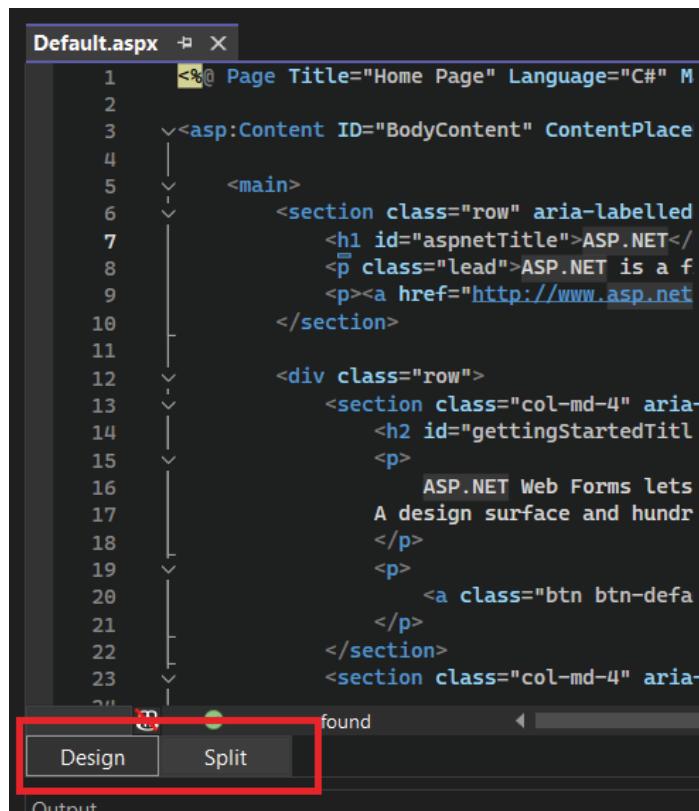


Figure 6.1 – The Design and Split buttons

These buttons allow us to display the designer, powered by Web Live Preview, in two different ways: full-size or split next to the code. We will click on the **Split** button to observe the impact the designer has on our code base:

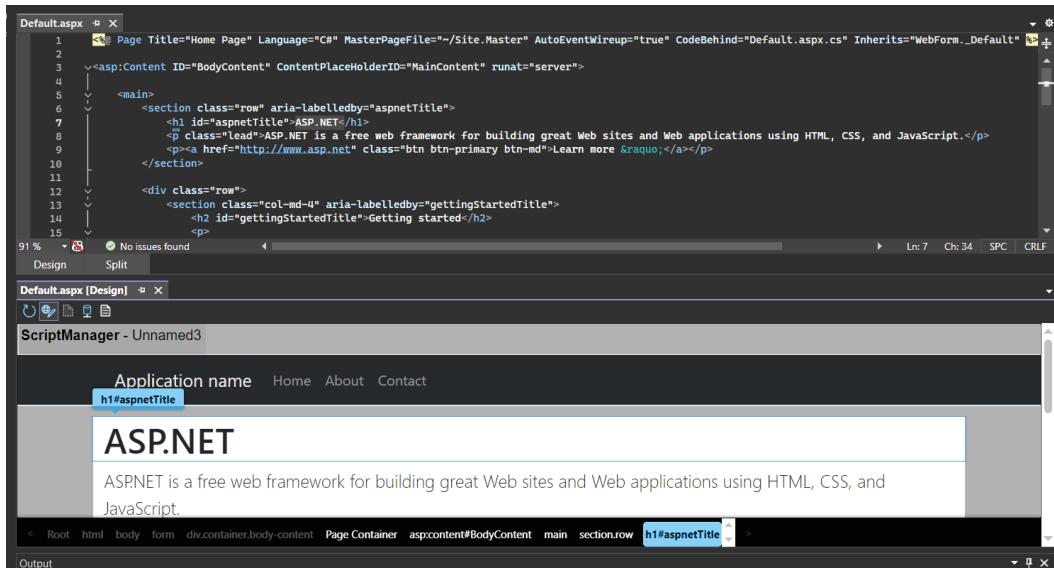


Figure 6.2 – The Split view

By clicking on the component, we can edit the label. Additionally, like in the designer for WinForm, we can drag and drop items in the toolbox to place them in the designer.

The following options are offered by this toolbox:

- **Refresh:** Updates the live preview of your web application to reflect any changes made in the editor
- **Toggle design mode:** Activates or deactivates the visual designer interface, allowing us to switch between designing our UI visually and editing the code directly
- **Show empty element:** Displays an empty UI component on the page, ready for us to configure its properties and add content
- **Use live data:** Binds our application's UI elements to data sources dynamically, enabling us to see how our application interacts with data in real-time during development

- **Show source view:** Opens the code-behind file associated with the selected UI element or control, allowing us to view or edit the source code directly:

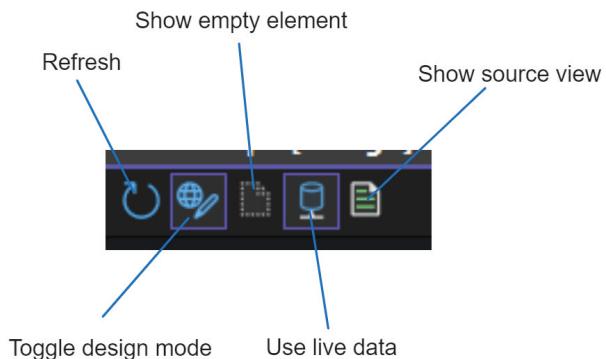


Figure 6.3 – Designer web toolbox

As shown in *Figure 6.2*, the designer displays live data by default to provide a valuable preview. In some cases, it could be valuable to show a placeholder. That's why the designer tools offer the **Use live data** toggle option.

In this section, we saw how Visual Studio 2022 can enhance our experience working with the legacy web framework. Now, let's discover what Visual Studio 2022 brings for cutting-edge API development.

## Using API Explorer and Dev Tunnel with Visual Studio

When we talk about web development, APIs come to mind for handling communication regarding each component. In this section, we will explore new Visual Studio features to enhance our experience during API development. First, we will learn how to use HTTP files to explore our endpoint, after which we will configure Dev Tunnel so that we can share our in-development APIs with clients, managers, and colleagues.

### Generating .http files with Endpoints Explorer

Using .http files to test and document API endpoints is incredibly convenient and efficient. This approach allows us to streamline API testing and development directly within our **integrated development environment (IDE)** – for example, Visual Studio, Visual Studio Code, or IntelliJ.

With .http files, we can write HTTP requests directly within our IDE and execute them without switching to a separate tool or browser. This integration keeps our development workflow within the IDE, enabling quick iterations and debugging, which ultimately boosts productivity.

These .http files adhere to the standards of RFC 9110 HTTP Semantics (the last version of the document, which establishes a solid foundation for understanding and implementing the protocol: <https://www.rfc-editor.org/rfc/rfc9110.html>), ensuring that our requests are correctly formatted and understood by the HTTP clients that are integrated into our IDEs. This standardization also makes the files portable and easily shareable among team members, and they can be seamlessly integrated into our CI/CD pipelines for efficient development.

Furthermore, integrating .http files into our existing development processes, including version control and code reviews, simplifies the management of API testing code, making it more maintainable and readable. The support for various HTTP methods, headers, and authentication mechanisms provides us with flexibility and power in terms of API testing.

Regarding Visual Studio, the support for .http files has significantly simplified the API testing process. We can create and execute HTTP requests directly within Visual Studio, eliminating the need to recreate requests for testing purposes, which makes the whole process more efficient and user-friendly.

First, we can list all the endpoints of our application with the **Endpoint Explorer** view. To open it, we can use the top bar menu and go to **View | Other Windows | Endpoints Explorer**:

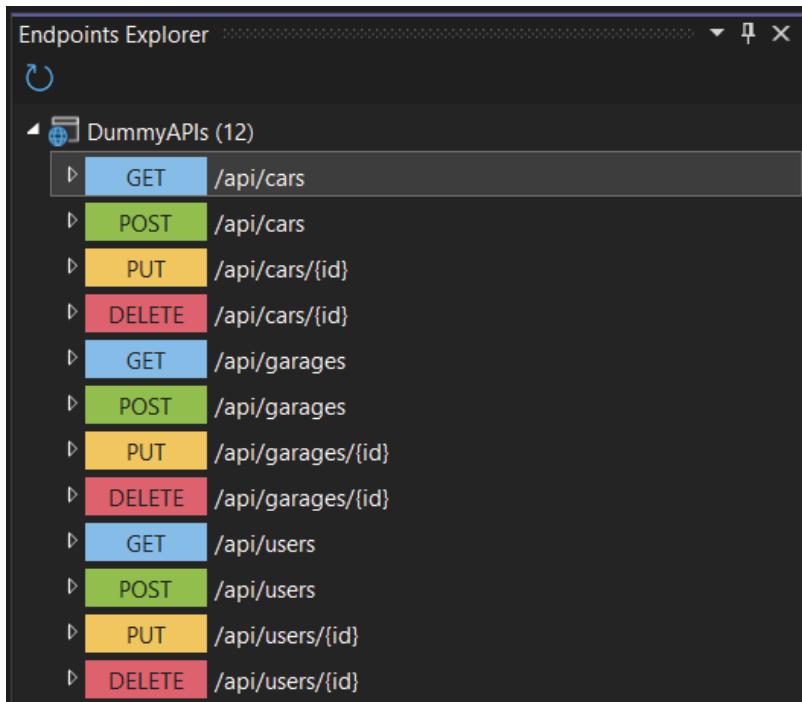


Figure 6.4 – Endpoints Explorer

This view shows us a list of all the endpoints we've built in our application. Here, we can right-click on an endpoint, and it will show us the following two options:

- **Open in the editor:** To jump to the code where the endpoint is defined
- **Generate Request:** To write the necessary code in the `.http` file:

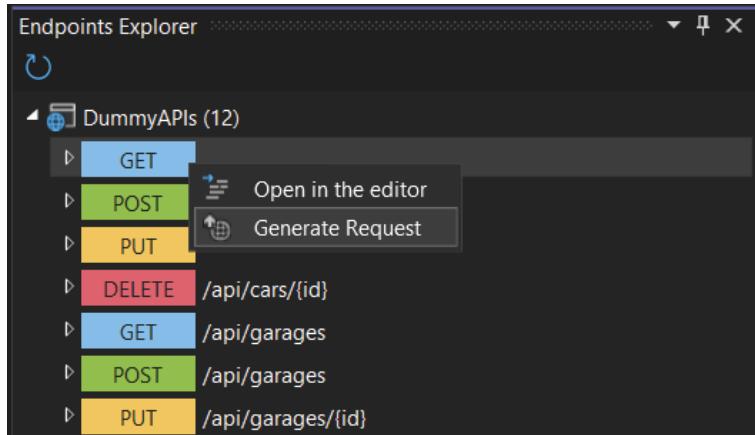


Figure 6.5 – Generate Request

In this example (*Figure 6.5*), we're generating the request for the `/api/cars` endpoint:

```
1  GET {{DummyAPIs_HostAddress}}/api/cars
2
3
4
5  @DummyAPIs_HostAddress = http://localhost:5128
6
7  Send request | Debug
8  GET {{DummyAPIs_HostAddress}}/api/cars
9
10 ####
```

The screenshot shows the 'DummyAPIs.http\*' file in Visual Studio. It contains a single line of code: `GET {{DummyAPIs_HostAddress}}/api/cars`. The host address is set to `http://localhost:5128`. There is a comment `Send request | Debug` preceding the line. The file has 7 lines of content, numbered 1 through 7.

Figure 6.6 – Generated .http request

Now, we have at least one request in the `.http` file and we can use it to test and debug our endpoint. Visual Studio offers a built-in interface to explore the response of the sent request:

```

Program.cs          DummyAPIs.http      AppDbContext.cs
GET {{DummyAPIs_HostAddress}}/api/cars    env: <none>
1   @DummyAPIs_HostAddress = http://localhost:5128
2
3
4
5   Send request | Debug
6   GET {{DummyAPIs_HostAddress}}/api/cars
7
8   ###

Status: 200 OK  Time: 646.2 ms  Size: 379 bytes

Formatted Raw Headers Request

Body

application/json; charset=utf-8, 379 bytes

[
  {
    "id": 0,
    "name": "fugit",
    "email": null
  },
  {
    "id": 0,
    "name": "odit",
    "email": null
  }
]

```

Figure 6.7 – .http file response

When testing APIs, we'll want to use a different environment. HTTP files allow us to define environment variables in an external file. For this, we can create a file named `http-client.env.json`. This file must be placed in the same folder as the `.http` file or a parent folder. We must set the different environments in the JSON file like so:

```
{
  "dev": {
    "HostAddress": "https://localhost:5128"
  },
  "remote": {
    "HostAddress": "https://contoso.com"
  }
}
```

Now, we can select the environment where we want to send the request from the top-right corner of the file:

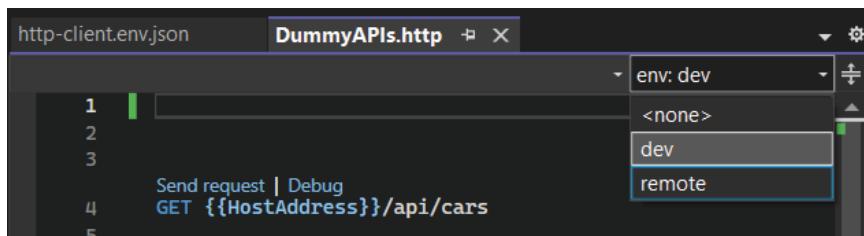


Figure 6.8 – Selecting an environment

Now, we can build a file that we can share with teams or clients to test and document our API. In some scenarios, we may want to debug our API through another device. Visual Studio 2022 provides a feature named **Dev Tunnel** that allows us to do this.

## Configuring Dev Tunnel

In Visual Studio 2022, Dev Tunnel allows developers to create temporary connections between machines that can't connect directly. This feature is great for debugging and testing web APIs and ASP.NET Core applications, especially when these applications need to be accessed from various devices, such as mobile emulators or physical devices. It's a valuable tool for us when we're working on applications that require testing across different machines or devices.

Here are some use cases where Dev Tunnel could be useful:

- Communication between web apps and mobile phones or tablets
- Port-forwarding solutions
- Communication with external services (for example, Twilio webhooks)

We can create a Dev Tunnel by clicking on the debug mode button and selecting **Dev Tunnels (no active tunnel) | Create a Tunnel...**:

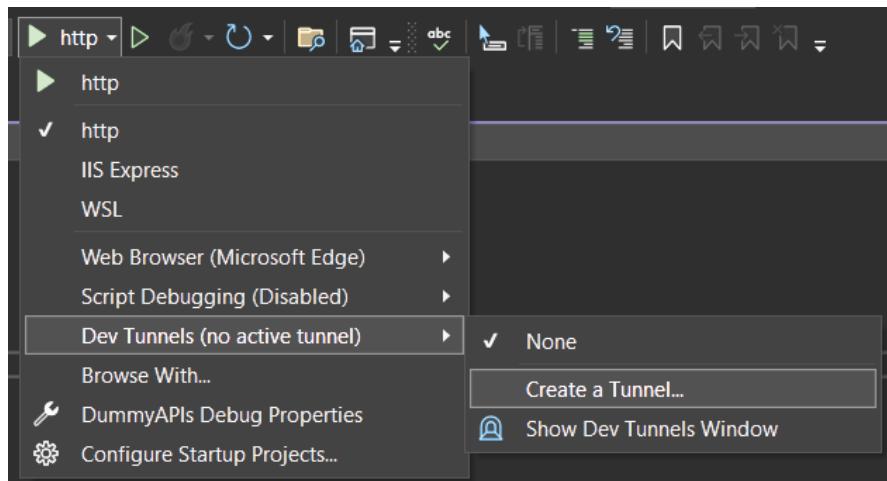


Figure 6.9 – Create a Tunnel...

After that, we get access to the configure window for our Dev Tunnel:

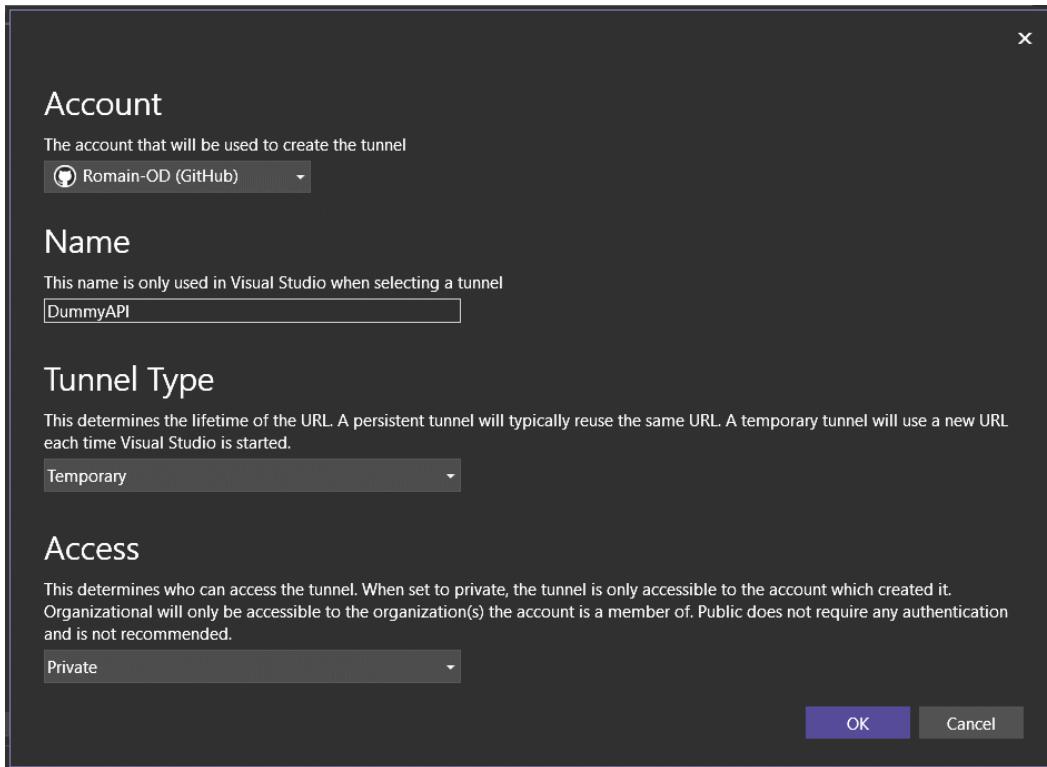


Figure 6.10 – Configuring our Dev Tunnel

In this window, we can set the following parameters:

- **Account:** We need to select an account to create the Dev Tunnel. This could be Azure, Microsoft Account (MSA), or GitHub.
- **Name:** The name that will be used to identify the Dev Tunnel in Visual Studio.
- **Tunnel Type:** Here, we can choose between two types – **Temporary** or **Persistent**:
  - **Temporary:** A new URL is generated every time Visual Studio is started
  - **Persistent:** The same URL will be presented every time Visual Studio is started
- **Access:** This allows us to set the level of access. Here, we have three options:
  - **Private:** Accessible only by the creator
  - **Organization:** Accessible by all users from the same organization
  - **Public:** Free and can be used by anyone

Now, we can get a URL from our local host up and ready to be shared on any device. For instance, we can use the URL API in a mobile to test our application, and during this time we can debug the API call using Visual Studio 2022.

When we talk about advanced web development these days, modern JavaScript frameworks are at the top of the list of frontend tools. Let's see how we can leverage Visual Studio for Node.js development.

## Node.js integration with Visual Studio

To develop a modern JavaScript frontend and backend, we must explore some features offered by Visual Studio. In this section, we will dive into how to create a JavaScript project with a Visual Studio workload. Then, we will learn how to manage the `npm` package so that we can debug our JavaScript application directly through Visual Studio.

### Exploring JavaScript project templates

In Visual Studio 2022, a new project type called **JavaScript Project System (JSPS)** has been introduced that utilizes the `.esproj` file format. This system allows us to create independent Angular, React, and Vue projects directly within Visual Studio. These frontend projects leverage the CLI tools of the respective frameworks that are installed on our machine, allowing us to choose the template version we prefer.

First, we need to verify whether the Node.js workload has been installed properly in our Visual Studio instance. To do this, open the Visual Studio Installer and check the installed workloads (see *Figure 6.11*) by clicking on the **Update** button for your desired Visual Studio instance:

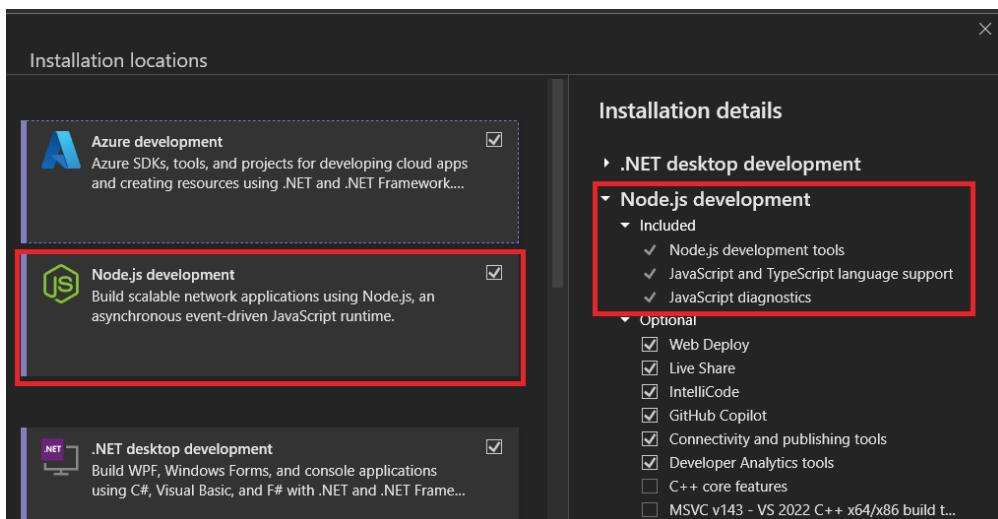


Figure 6.11 – Node.js development workload

Having ensured that we have the necessary workload installed, we can explore the template offered by Visual Studio 2022 for JavaScript development. For that, we will create a new project and select **JavaScript** in the **Language combo** box. Once you've done that, you will be presented with the following screen:

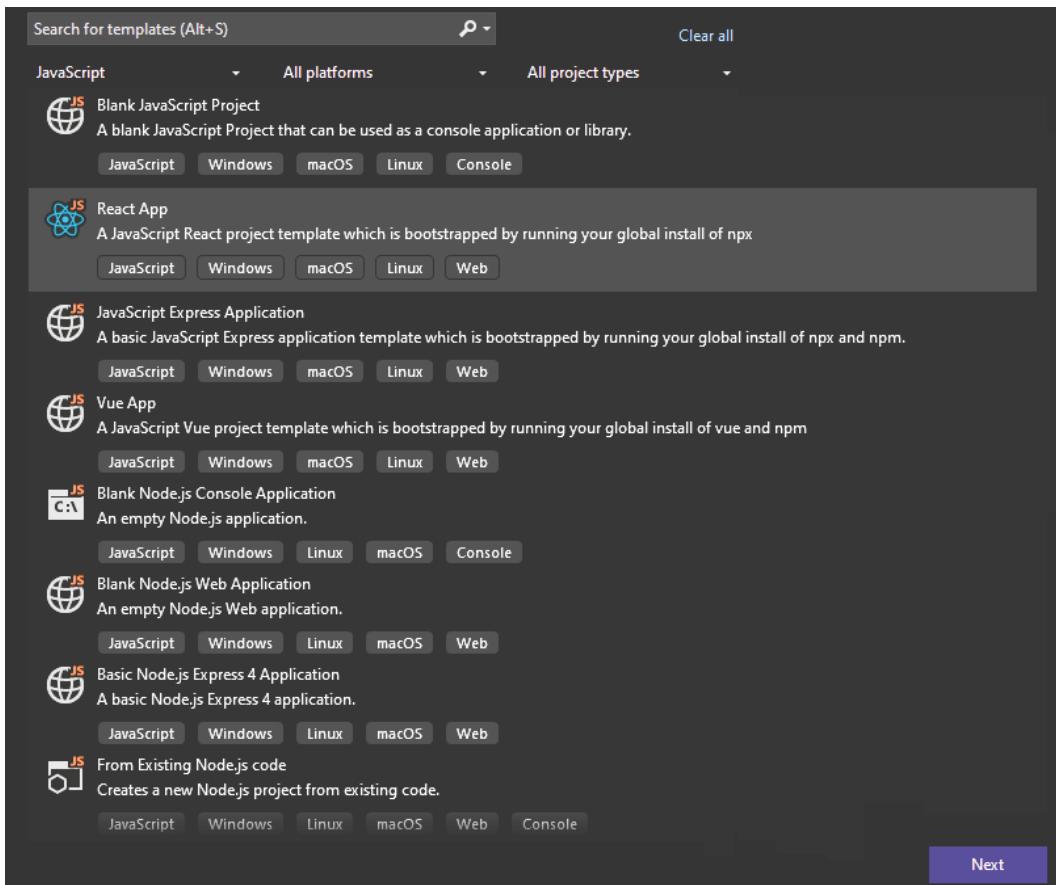


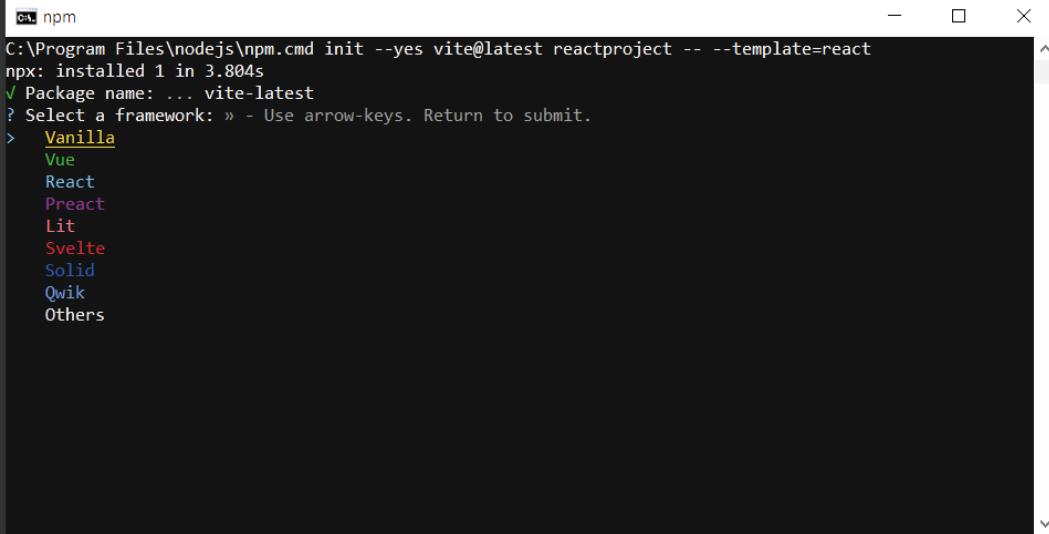
Figure 6.12 – JavaScript template

Here, we can see that Visual Studio offers several JavaScript base templates that we can split into three categories:

- ASP.NET Core combines with the modern JavaScript framework
- Standalone JavaScript project (client-side)
- Backend Node.js project

For our example, we will pick the **ReactApp** template as standalone JavaScript project.

When creating the project, we'll see that Visual Studio launches a console prompting for the `npm cli` command so that it can create the application:



The screenshot shows a terminal window titled 'npm' with a black background. The command entered is 'C:\Program Files\nodejs\npm.cmd init --yes vite@latest reactproject -- --template=react'. The output shows 'npx: installed 1 in 3.804s' and a green checkmark next to 'Package name: ... vite-latest'. Below this, a question 'Select a framework: » - Use arrow-keys. Return to submit.' is displayed, followed by a list of frameworks: Vanilla (selected), Vue, React, Preact, Lit, Svelte, Solid, Qwik, and Others. The 'Vanilla' option is underlined in blue, indicating it is the current selection.

Figure 6.13 – npm CLI

Once we've set up the project, we can develop our application while adhering to React or other modern JavaScript frameworks. One of the initial steps involves installing the necessary npm packages. While we can certainly do this directly using the npm CLI, let's explore how Visual Studio can assist us in this process.

## Managing npm packages

Managing npm packages in Visual Studio 2022 involves using the npm package manager.

Beginning with Visual Studio 2022, we have access to the npm package manager for CLI-based projects. This means that we can now download npm modules like how we can download NuGet packages for ASP.NET Core projects. We can then utilize the package `.json` file to make modifications to packages and remove them as needed.

We can access the npm package manager by right-clicking on the npm node of the folder structure:

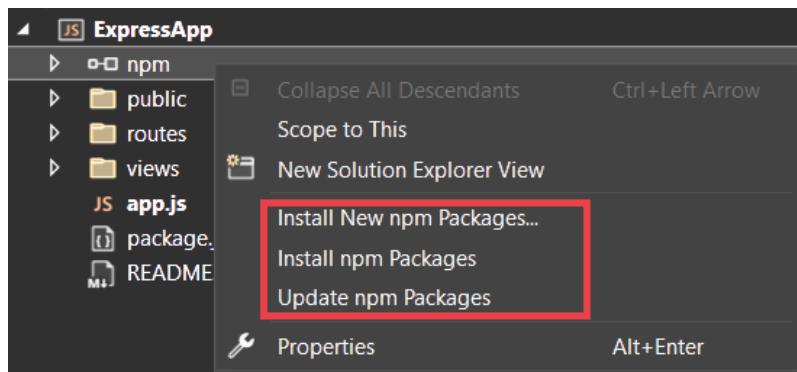


Figure 6.14 – Debugging a JavaScript application

Then, we can select **Install New npm Packages...**, which will open the manager:

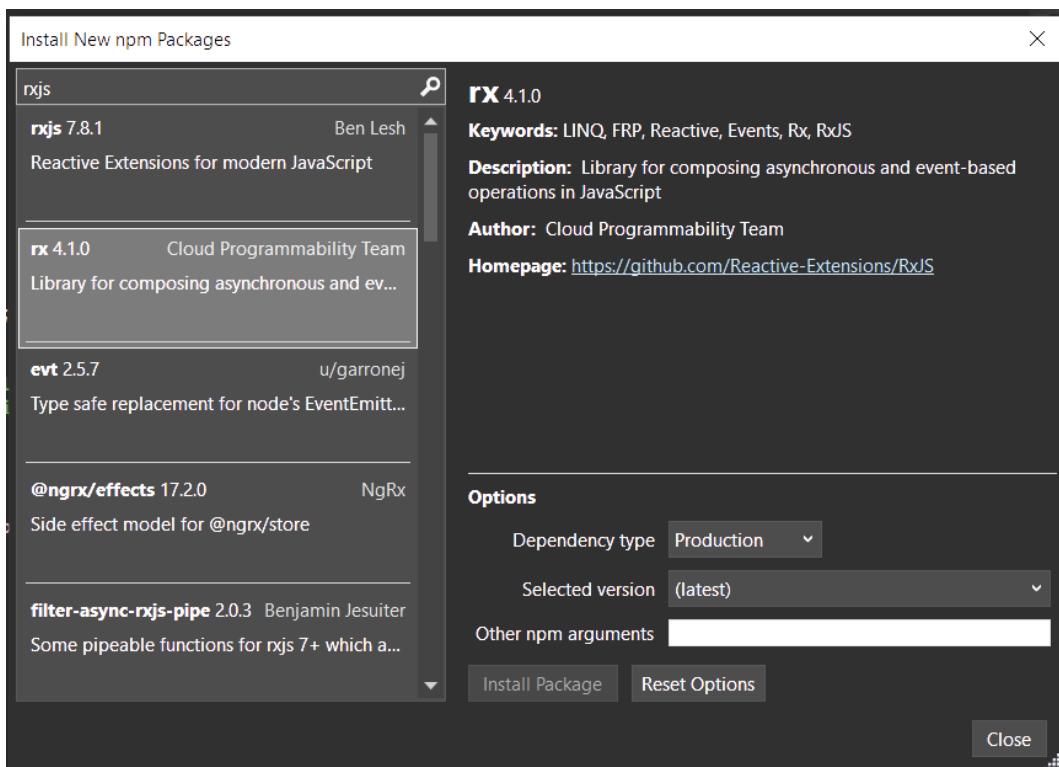


Figure 6.15 – Install New npm Packages

In this window, we can find all the packages we need to build our modern JavaScript application. As with any application development process, there comes a time when we need to debug. Let's explore how Visual Studio can enhance our experience of debugging modern JavaScript applications.

## Debugging JavaScript applications

Working with server-side JavaScript allows us to debug our application similarly to debugging a C# application. To do so, we must follow these steps:

1. **Set breakpoints:** Open your server-side JavaScript file (for example, `server.js`) in Visual Studio and click in the gutter area to set a breakpoint.
2. **Run the application in debug mode:** Press *F5* or go to **Debug | Start Debugging** to run our application in debug mode. Visual Studio will pause execution at the breakpoint we set.
3. **Inspect the application's state:** While paused at the breakpoint, we can inspect our application's state by hovering over variables in scope or using debugger windows such as **Locals** and **Watch**.
4. **Continue execution:** To continue running our application after inspecting it, press *F5* again.

As you can see, debugging server-side JavaScript applications is straightforward. Now, let's explore debugging client-side scripts.

### *Debugging client-side scripts*

In Visual Studio, we have debugging support specifically for Chrome and Microsoft Edge (Chromium) when it comes to client-side debugging. Sometimes, our debugger will automatically stop at breakpoints in JavaScript, TypeScript, and embedded scripts within HTML files.

To debug JavaScript within ASP.NET using Chrome, Edge (Chromium), and Internet Explorer, we need to navigate to the debugging option by selecting **Tools | Options | Debugging | General**, and then check the **Enable JavaScript debugging for ASP.NET (Chrome, Edge, and IE)** box:

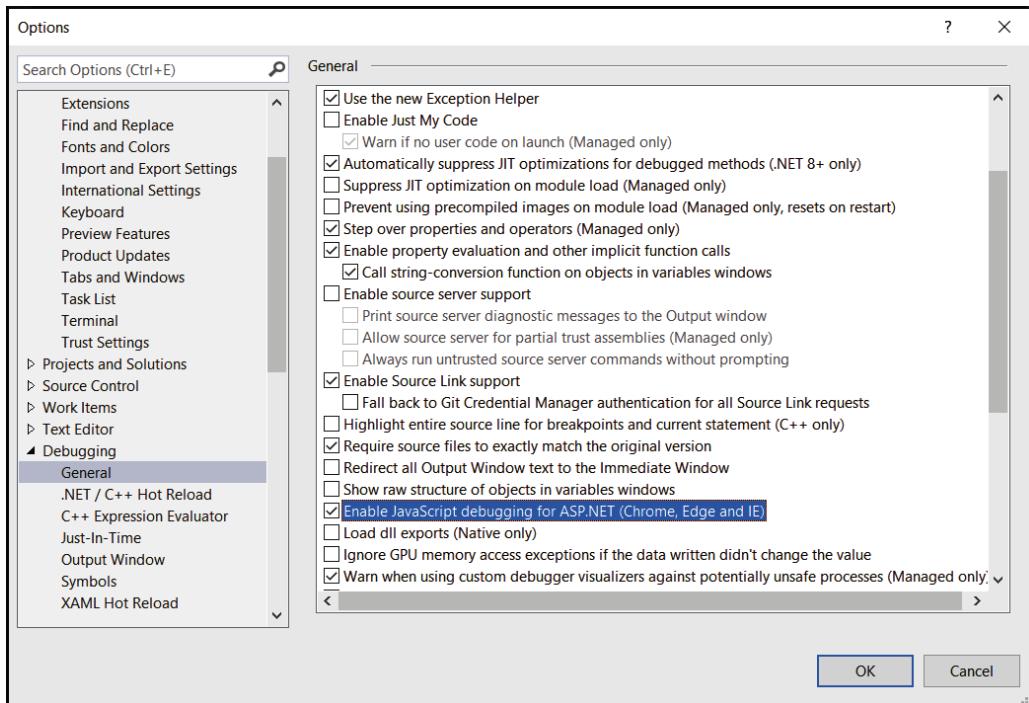


Figure 6.16 – Debugging options

Enabling this option ensures that Visual Studio supports debugging client-side code effectively within ASP.NET projects.

In our previous example, we created a standalone React application that required our source code to be transpiled by tools such as TypeScript or Babel. Utilizing source maps during debugging ensures an optimal experience by bridging the gap between minified or transpiled code and its original, human-readable form. This facilitates debugging by allowing developers to set breakpoints and inspect variables in their original state within the browser's developer tools or Visual Studio's debugging environment.

Visual Studio automatically generates source maps for TypeScript projects. However, for JavaScript projects, we need to configure build tools such as webpack to produce source maps. For TypeScript projects, include a `tsconfig.json` file with the `sourceMap` compiler option set to true. For JavaScript projects using webpack, configure it with `devtool: "source-map"`.

Next, ensure that Visual Studio is properly configured to use source maps. This may involve adjusting paths within generated source map files to accurately reference source files. For webpack users, remove the `webpack://` prefix from source map paths.

Once source maps have been set up, debugging a React project in Visual Studio becomes straightforward. Simply attach the debugger to the running application – this allows you to set breakpoints and step through the code as if it were running in its original, un-minified form. This streamlined approach helps with quickly identifying and fixing issues.

For more complex debugging scenarios, additional adjustments to the project configuration or the use of supplementary tools may be necessary. Ensure that webpack is configured properly to generate source maps. Additionally, enable JavaScript source maps in your browser's Developer Tools settings to fully utilize source maps during debugging.

## Summary

In this chapter, we embarked on an advanced exploration of web development tools within the robust environment of Visual Studio 2022. Our journey began with an in-depth look at Web Live Preview, a cutting-edge tool enabling real-time web previews to streamline design visualization and iteration for Web Forms projects.

We then delved into the dynamic capabilities of Visual Studio's API Exploration and Dev Tunnel functionalities, empowering developers to gain insights into service endpoints and seamlessly integrate with external APIs for interactive development and testing.

Furthermore, we explored Node.js integration with Visual Studio, uncovering JavaScript project templates, effective npm package management techniques, and the intricacies of debugging JavaScript applications within the Visual Studio ecosystem.

In the next chapter, we will delve into machine learning integration with Visual Studio 2022, exploring how to leverage the latest features to seamlessly integrate machine learning models into applications.

# 7

## Machine Learning Integration

In this chapter, we will embark on an exploration of integrating **machine learning (ML)** seamlessly into software development workflows using Visual Studio. As we continue to advance in the digital age, the integration of ML capabilities into applications has become increasingly pivotal, enabling intelligent decision-making and automation within our software systems.

Our journey begins with an introduction to ML, demystifying its core concepts and applications. Next, we will dive into the practical aspects by harnessing the power of **ML.NET** and **Model Builder** within Visual Studio. Through hands-on examples and guided tutorials, we'll demonstrate how to create and train ML models directly within the familiar Visual Studio environment.

As we progress, the focus shifts toward deployment strategies for ML models. We'll explore how to deploy a trained model within an ASP.NET Core web API, enabling real-time inference and integration with web-based applications. Additionally, we'll explore deploying models within Azure Functions, leveraging serverless computing for scalable and cost-effective deployment scenarios.

Key topics covered in this chapter include the following:

- Introduction to ML
- Creating machine learning models with ML.NET and Model Builder
- Deploying models in an ASP.NET Core web API
- Deploying models in Azure Functions

On this journey, we will unlock the potential of integrating ML into our software projects using Visual Studio, empowering us to build intelligent and adaptive applications that resonate with today's technology landscape.

## Technical requirements

I wrote this chapter with the following version of Visual Studio in mind:

- Visual Studio Enterprise 2022 Version 17.12.0
- Preview 1.0

The code files for this chapter can be found at <https://github.com/PacktPublishing/Mastering-Visual-Studio-2022/tree/main/ch07>

## Introduction to ML

ML is a branch of **artificial intelligence (AI)** focused on developing algorithms and statistical models that enable computers to perform tasks without explicit programming. These tasks can range from basic pattern recognition to complex decision-making processes. At its core, ML involves machines learning from data to improve the algorithms' performance over time.

There are three primary types of ML:

- **Supervised learning:** This involves training a model on labeled data to predict outputs based on input data, such as regression (predicting continuous values) and classification (predicting discrete labels)
- **Unsupervised learning:** This involves training a model on unlabeled data to identify patterns or structures within the dataset, such as clustering (grouping similar data points) and dimensionality reduction (reducing input variables)
- **Reinforcement learning:** Models learn to make decisions by interacting with an environment, and receiving feedback in the form of rewards or penalties to maximize the cumulative reward

ML utilizes various algorithms such as decision trees, **support vector machines (SVMs)**, and neural networks, chosen based on specific problems, data characteristics, and desired outcomes.

ML.NET, an open source, cross-platform ML framework by Microsoft, simplifies ML for .NET developers, enabling the integration of ML capabilities into applications without requiring deep expertise in ML or data science.

Key features of ML.NET include the following:

- **Cross-platform compatibility:** It works seamlessly on Windows, Linux, and macOS
- **Integration with .NET:** It easily incorporates ML capabilities into .NET projects
- **Customizable models:** Developers can tailor models to their needs, including training on custom datasets
- **Transparency and explainability:** It provides tools for understanding how models make predictions, essential for building trust in AI
- **Open source and community-driven:** It benefits from community contributions, leading to ongoing improvements and new features

**ML.NET** is a versatile framework that simplifies the integration of ML capabilities into .NET applications. Features such as **automated ML (AutoML)** and tools such as ML.NET CLI and Model Builder make the process straightforward, even for developers without extensive data science experience. It supports cross-platform development and seamlessly integrates with popular Python libraries such as TensorFlow and ONNX through NimbusML.

The Model Builder tool within ML.NET streamlines model creation with AutoML, enabling developers to deploy models swiftly by loading their data. This tool automates the entire model-building process, including code generation for consuming these models. Combined with its flexibility to leverage existing ML libraries, ML.NET offers a robust solution for .NET developers seeking to incorporate ML into their applications.

Now, let's dive into creating an ML model using ML.NET and the intuitive Model Builder **user interface (UI)**.

## Creating an ML model with ML.NET and the Model Builder UI

Initially introduced as a preview feature in Visual Studio 2019, Model Builder has transitioned to a stable feature as of 2022. In this section, we will explore how to use Model Builder to create an ML model with ML.NET.

First, let's ensure that the Model Builder component is installed correctly in our instance of Visual Studio. To do so, we open the Visual Studio Installer and confirm that ML.NET Model Builder is selected under the **Individual components** tab:

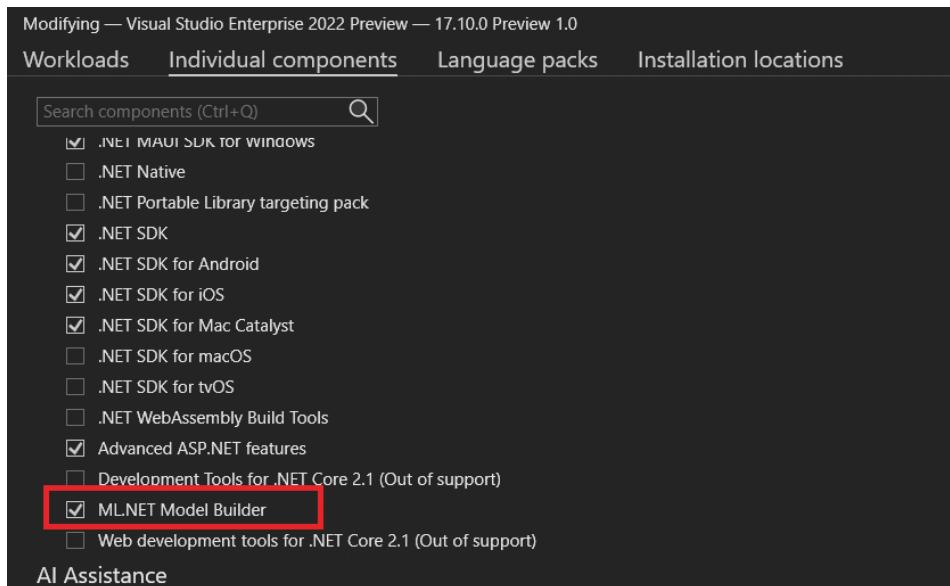


Figure 7.1 – Installer – ML.NET Model Builder

Then, we can create a new empty console project as a foundation for supporting our ML process. Now, we are all set, and we can create our ML.NET by using the file-adding menu by right-clicking on the project and selecting **Add | Machine Learning Model....**:

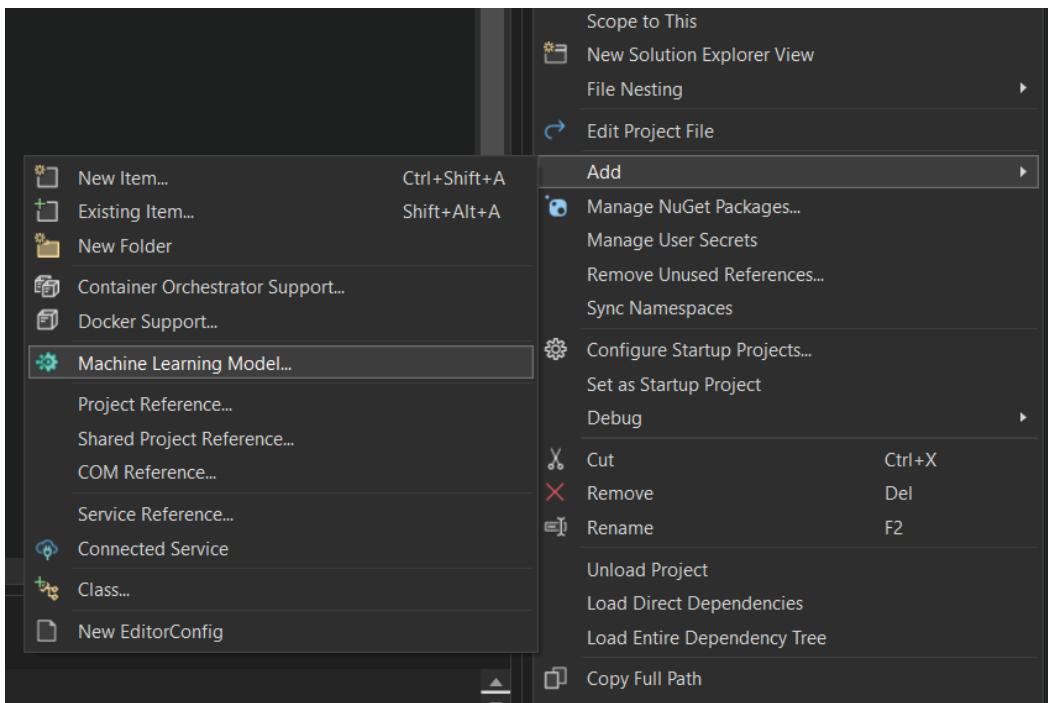


Figure 7.2 – Add an ML model

The preceding action will initiate the `.mbconfig` file. This file is a JSON file that keeps track of the state of the UI, including the model's configuration, data transforms, algorithm, and settings such as learning rate, number of layers, and number of neurons.

The first step prompted by Model Builder, after choosing a name for the `.mbconfig` file, is to select a **Scenario** option for our ML model.

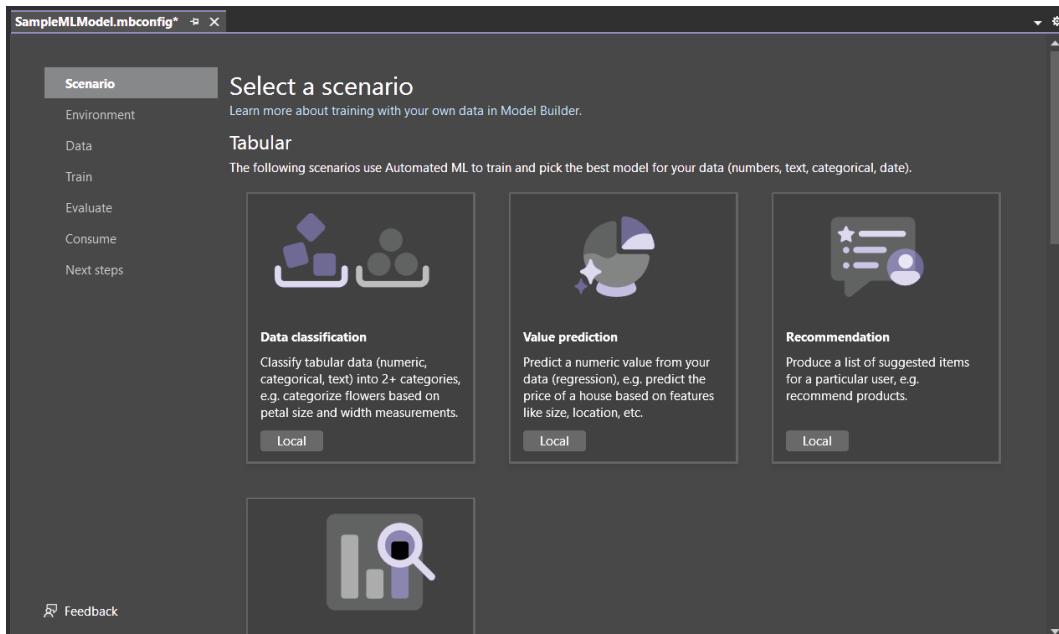


Figure 7.3 – Configuring `.mbconfig`

The selection window of Model Builder offers several scenarios organized into three main categories:

- **Tabular:** This category includes scenarios for tasks such as regression, classification, and clustering, where the data is organized in a tabular format

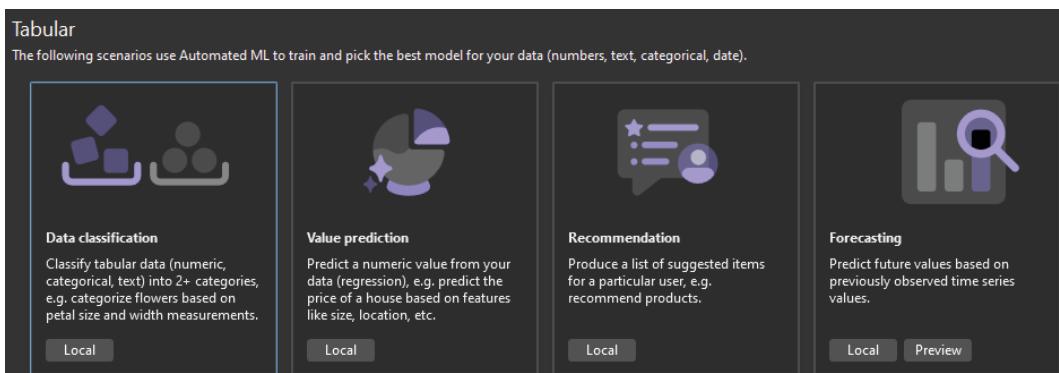


Figure 7.4 – Tabular

- **Computer Vision:** This category encompasses scenarios related to image classification, object detection, and other computer vision tasks

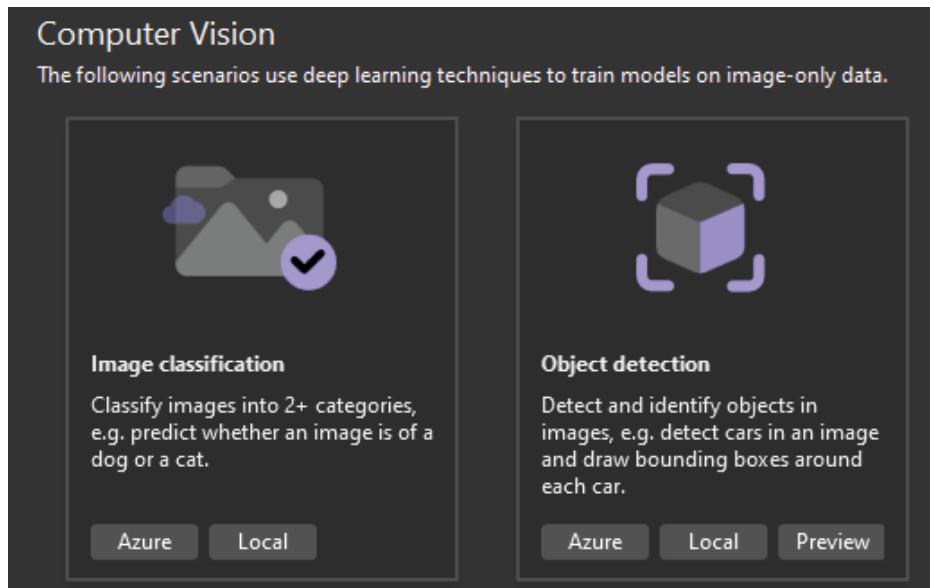


Figure 7.5 – Computer Vision

- **Natural Language Processing:** This category includes scenarios for natural language processing tasks, such as sentiment analysis, text classification, and language translation

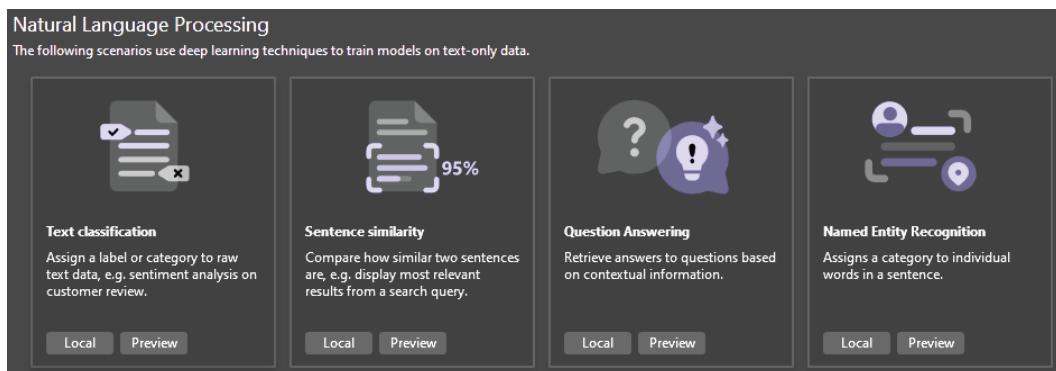


Figure 7.6 – Natural Language Processing

In our example, we will choose the **Data Classification** scenario under the **Tabular** category.

The second step is to select the training environment. We have the option to train our ML model either locally on our machines or in the cloud on Azure, depending on the situation. When we train locally, we operate within the limitations of our computer resources (CPU, memory, and disk). However, when we train in the cloud, we can scale up our resources to address the requirements of our scenario, particularly for handling large datasets. Notice that the availability of the environment depends on the scenario we choose. Here's a table of the current availability combination:

Scenario	Local CPU	Local GPU	Azure
Data classification	✓	✗	✗
Value prediction	✓	✗	✗
Recommendation	✓	✗	✗
Forecasting	✓	✗	✗
Image classification	✓	✓	✓
Object detection	✗	✗	✓
Text classification	✓	✓	✗

Now that we have chosen our scenario and training environment, we must gather the data that will be used for training. Model Builder will guide us through the process according to our selected scenario, helping us upload the data.

Finally, we can launch the training of our ML model. We can set a specific time for training to begin. Model Builder automatically chooses a training time based on our dataset size.

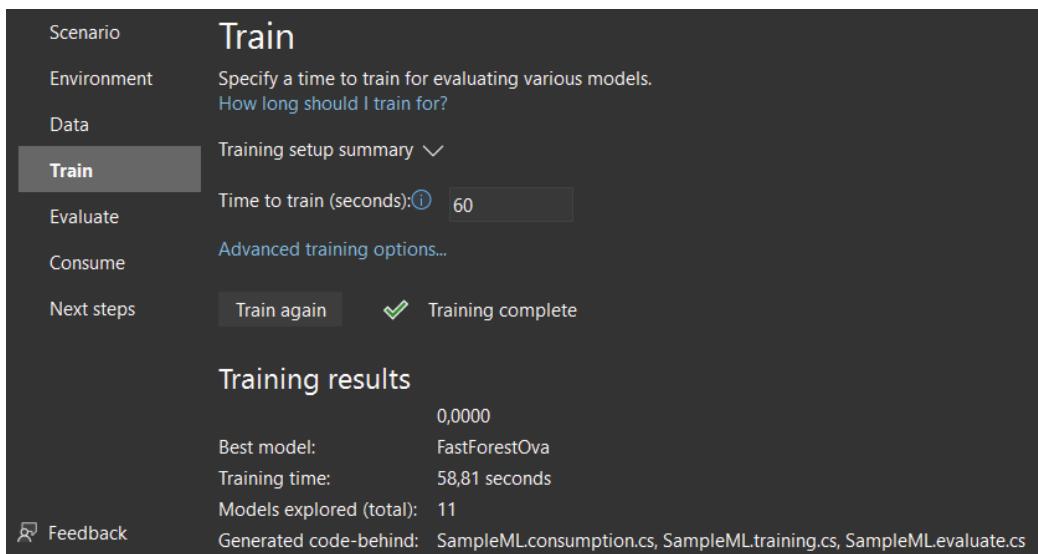


Figure 7.7 –Train

During the **Evaluate** step, we'll discover the best-performing algorithm and its highest accuracy, providing valuable insights into our model's performance. This step also enables us to experiment with the model directly within the UI.

In the **Try your model** section, we can input sample data to generate predictions. The textbox comes pre-filled with the first line of data from our dataset, but we have the flexibility to modify the input and click the **Predict** button to observe different price predictions.

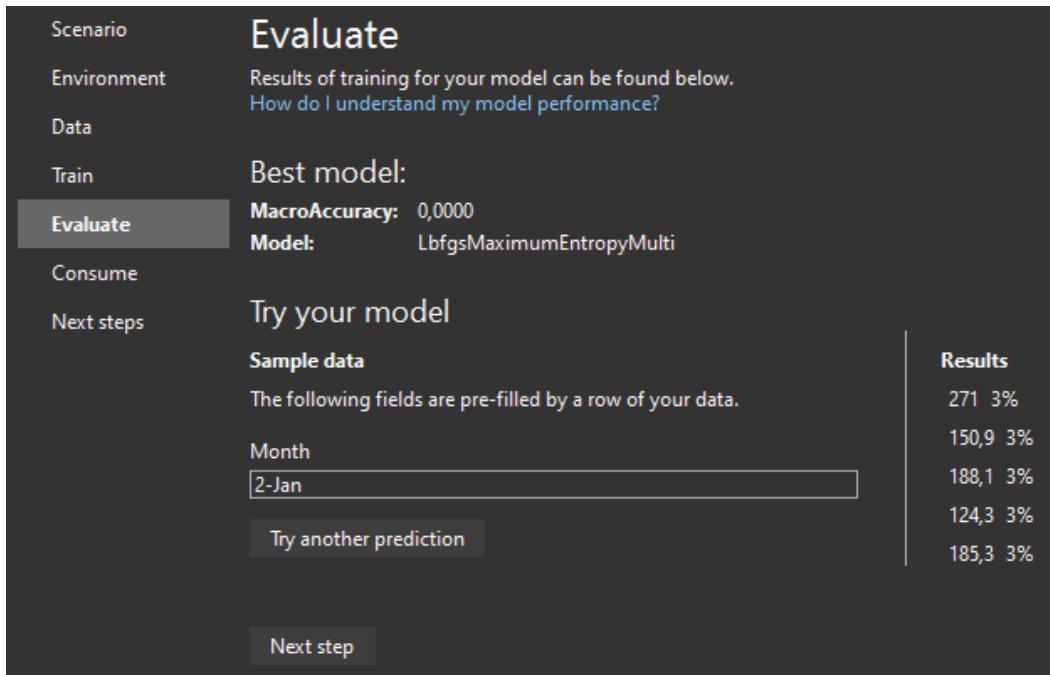


Figure 7.8 – Evaluate

The last panel of Model Builder is about consuming the model we just created. Let's explore how we can integrate our model into a web API using Visual Studio.

## Deploying a model in an ASP.NET Core web API

In this section, we will learn how to integrate our ML.NET model into an existing web API. To do so, we will jump to the **Consume** panel of Model Builder.

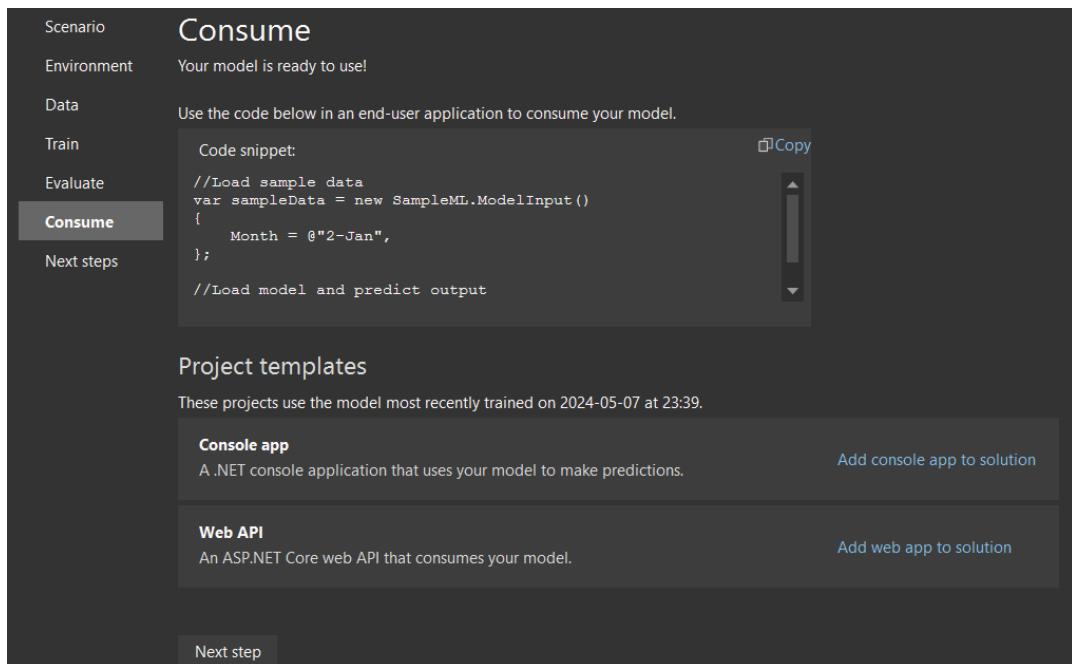


Figure 7.9 – Consume

The **Consume** panel within Model Builder is a critical tool for integrating our trained ML models into .NET applications. Once we've completed the evaluation phase, Model Builder generates a model file and the necessary code to incorporate the model into our application. These models are saved as a `.zip` file, and the code to load and use our model is added as a new project within our solution. Additionally, Model Builder provides a sample console application that we can run to see our model in action.

The **Consume** panel gives us options to create projects that consume our model, such as the following:

- **Console app:** Generates a .NET console application specifically for making predictions using our model
- **Web API:** Sets up an ASP.NET Core web API, allowing us to consume our model over the internet

These projects are crucial for deploying our model in various environments, whether for local testing or web-based applications accessible online.

Let's click on the **Add web app to the solution** link to create the new ASP.NET Core web API exposing our model. That will set up a working web API, enabling us to expose the model we just trained, using the minimal API way.

Now, to understand how it's articulate, we will take a look at what the wizard generates for us. Let's explore the project structure:

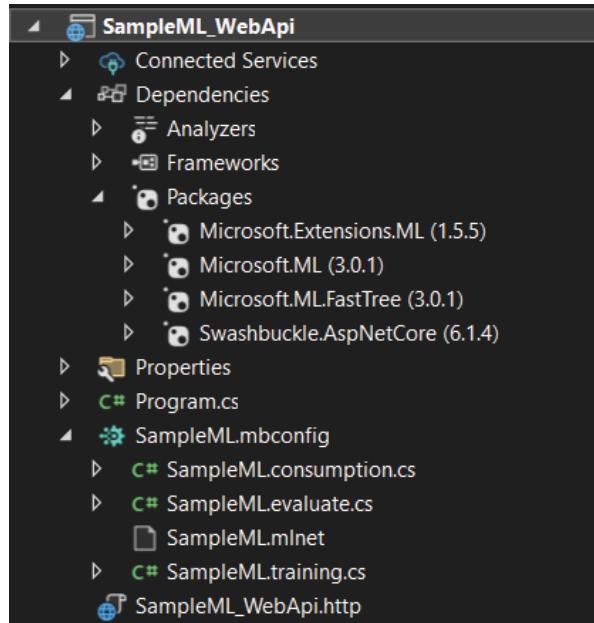


Figure 7.10 – Generating a web API project structure

The template utilizes the following packages to make ML.NET models accessible for ML:

- Microsoft.ML
- Microsoft.Extensions.ML

Among these libraries, in the `Program.cs` file, we observe that Visual Studio generates a POST request. Here's the code snippet:

```
app.MapPost("/predict",
    async (PredictionEnginePool<SampleML.ModelInput,
        SampleML.ModelOutput> predictionEnginePool,
        SampleML.ModelInput input) =>
    await Task.FromResult(
        predictionEnginePool.Predict(input)
    )
);
```

This code snippet uses the models generated by the model builder to set up an HTTP POST endpoint `/predict` in an ASP.NET Core minimal API application.

This is possible because of the following lines of code:

```
builder.Services
    .AddPredictionEnginePool<SampleML.ModelInput,
        SampleML.ModelOutput>()
    .FromFile("SampleML.mlnet");
```

This code is configuring an ASP.NET Core application to use an ML model stored in a file named `SampleML.mlnet` for making predictions. The model expects inputs of the `SampleML.ModelInput` type and produces outputs of the `SampleML.ModelOutput` type. This setup allows the application to efficiently manage and reuse the ML model across different parts of the application.

Now, we can test the endpoints using the endpoints explorer and `.http` file, as we saw in *Chapter 6*. Since the HTTP verb used for this endpoint is POST, we have to add a body to our request.

Here's the proper syntax to set up our working request in the `.http` file:

```
@SampleML_WebApi_HostAddress = https://localhost:63555

POST {{SampleML_WebApi_HostAddress}}/predict
Content-Type: application/json

{
    "Month": "2-Jan"
}
```

Now, we can validate that we own a web API able to consume our model generated by Model Builder. This web API can be seamlessly deployed like any other API in the .NET ecosystem.

There is another way to consume our generated ML model, and that is to consume it through Azure Functions.

## Deploying a model in Azure Functions

In this section, we will learn how to integrate the ML model generated by Model Builder into Azure Functions. This process will be less straightforward as no template is provided.

**Azure Functions** is a serverless computing service offered by Microsoft Azure. Serverless computing is a cloud computing execution model where the cloud provider runs the server, and dynamically manages the allocation of machine resources. Pricing is based on the actual number of resources consumed by an application, rather than pre-purchased units of capacity.

First, we need to ensure that the Azure development workload is well installed in our Visual Studio instance by navigating to the Visual Studio Installer.

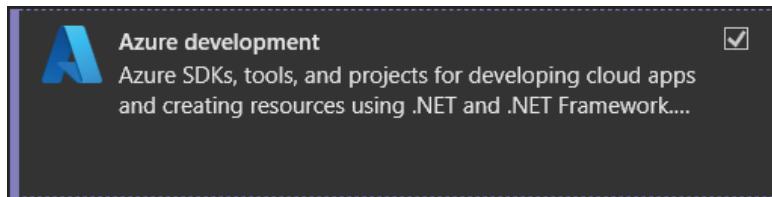


Figure 7.11 – Azure development workload

Once the workload is installed, we can start by creating a new Azure Functions project, for example, we will name it `SampleML_AzureFunction`, and we will keep all parameters by default.

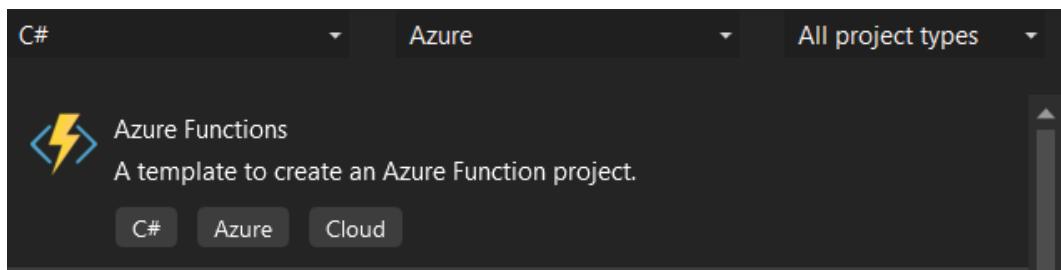


Figure 7.12 – Azure Functions project

This time, we need to add both libraries by ourselves, as seen in the previous section, using the NuGet Manager. We will open the NuGet Manager by right-clicking on the project name, selecting **Manage NuGet Packages...**, and then browsing for our needed libraries, to finally install them:

- `Microsoft.ML`
- `Microsoft.Extension.ML`

Now, to be able to use our pre-trained model, all we need to do is copy the `.mbconfig` file to Azure Functions. After that, to be sure you can access the `.mlnet` package after compiling, we need to set the **Copy to Output Directory** properties to **Copy if newer**.

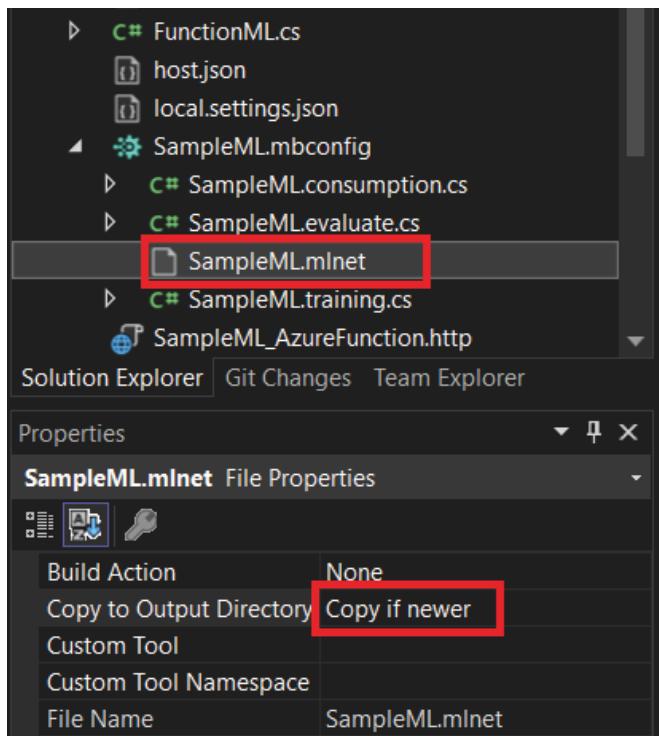


Figure 7.13 – The SampleML.Net properties

Once we are all set, we can develop our function following these simple steps:

1. Add `startup.cs` to configure our Azure Functions project to use `SampleML.mlnet`:

```
[assembly: FunctionsStartup(typeof(Startup))]  
namespace SampleML_AzureFunction;  
public class Startup : FunctionsStartup  
{  
    public override void Configure(  
        IFunctionsHostBuilder builder)  
    {  
        builder.Services  
            .AddPredictionEnginePool<  
                SampleML.ModelInput,  
                SampleML.ModelOutput>()  
            .FromFile("SampleML.mlnet");  
    }  
}
```

2. Call the predict method of the ML.NET package in the Azure's run method:

```
public class FunctionML
{
    private PredictionEnginePool<
        SampleML.ModelInput, SampleML.ModelOutput>
        _predictionEnginePool;

    public FunctionML(
        PredictionEnginePool<SampleML.ModelInput,
            SampleML.ModelOutput>
            predictionEnginePool)
    {
        _predictionEnginePool =
            predictionEnginePool;
    }

    [FunctionName("FunctionML")]
    public async Task<IActionResult> Run(
        [HttpTrigger(
            AuthorizationLevel.Anonymous,
            "post",
            Route = null)] HttpRequest req)
    {
        string requestBody = await new
            StreamReader(req.Body)
            .ReadToEndAsync();
        SampleML.ModelInput input = JsonConvert
            .DeserializeObject<
                SampleML.ModelInput>(
            requestBody);

        SampleML.ModelOutput responseMessage =
            await Task.FromResult(
                _predictionEnginePool
                .Predict(input));

        return new OkObjectResult(
            responseMessage);
    }
}
```

Finally, we can test the Azure function using the Endpoints Explorer and .`http`, like how you tested the web API in the previous section.

## Summary

In this chapter, we explored the integration of ML capabilities into software development workflows using Visual Studio, equipping developers with essential skills to leverage intelligent decision-making within applications.

We began by introducing the fundamental concepts of ML, providing an overview to understand its applications and significance in modern software development. Next, we delved into practical implementation by demonstrating how to create and train ML models using ML.NET and Model Builder within Visual Studio. Moving forward, we explored deployment strategies for trained models, showcasing how to deploy ML models within an ASP.NET Core web API for real-time inference and integration with web applications. Additionally, we discussed deploying models within Azure Functions to leverage serverless computing for scalable and efficient deployment scenarios.

In this chapter, you gained hands-on experience in building, training, and deploying ML models using Visual Studio, enabling you to infuse intelligent capabilities into your applications.

In the upcoming chapter, we'll continue on track toward advanced cloud integration and services through Visual Studio 2022.



# 8

## Advanced Cloud Integration and Services

In this chapter, we will delve into the advanced realm of cloud integration and services, highlighting how Visual Studio 2022 serves as a powerful tool to develop and manage cloud-based applications. In today's digital ecosystem, cloud computing has become a cornerstone, enabling scalable, resilient, and highly available applications. To stay ahead, developers must not only build robust applications but also seamlessly integrate them with various cloud platforms.

Throughout this chapter, we'll learn how to harness the power of various cloud services, streamline our development workflows, and ensure that our applications are optimized for cloud environments. Whether it's deploying serverless functions on Azure, integrating Google Cloud services, or leveraging the extensive tools provided by AWS, this chapter will equip you with the knowledge and skills needed to excel in advanced cloud development.

Key topics covered in this chapter include the following:

- Exploring .NET Aspire
- Exploring Azure Functions development in Visual Studio 2022
- Exploring Google Cloud Tools for Visual Studio
- Exploring the AWS Toolkit

Let's begin our journey toward mastering cloud integration and building cutting-edge cloud-based applications together.

## Technical requirements

While writing this chapter, I used the following version of Visual Studio:

- Visual Studio Enterprise 2022 Version 17.12.0
- Preview 1.0

For an optimal experience, consider having the following:

- An Azure subscription
- A Google Cloud Platform subscription
- An AWS subscription

The code files for this chapter can be found at <https://github.com/PacktPublishing/Mastering-Visual-Studio-2022/tree/main/ch08>

## Exploring .NET Aspire

In November 2023, as part of .NET 8, Microsoft introduced .NET Aspire. The motivation of Microsoft is to make .NET one of the most productive platforms for building cloud-native applications. This technology seamlessly integrates with popular containerization platforms such as Docker and Kubernetes, facilitating the management of applications and their deployment in cloud-native environments. In this section, we will explore what .NET Aspire is and how Visual Studio guides us through its development process.

.NET Aspire represents a sophisticated suite of tools engineered to facilitate the construction of observable, robust, and scalable applications tailored for cloud environments. Designed with a focus on cloud-specific requirements, this cohesive framework seeks to ease the process of developing cloud-native solutions by offering a coherent array of utilities and methodologies. Central aspects of .NET Aspire encompass the following:

- **A cloud-specific framework:** Tailored to address the unique hurdles encountered during cloud application development, .NET Aspire underscores the importance of being easily observable, ready for deployment, and capable of operating across distributed systems.
- **Component-based design:** Presented as a series of NuGet packages, these components cater to distinct needs within cloud computing, allowing developers to incorporate only those elements essential for their projects. This modular structure not only enhances adaptability but also boosts productivity.
- **A streamlined development workflow:** By introducing a uniform set of tools and practices, .NET Aspire simplifies the assembly of .NET applications suited for the cloud. It features pre-configured modules for databases such as Redis and PostgreSQL, thereby easing their integration into projects.

- **Enhanced developer tools:** The toolkit enriches the development life cycle through project templates and integrations with Visual Studio and the .NET CLI, making the initiation and administration of applications more straightforward.

.NET Aspire allows us to manage the organization and linkage of different parts of an application, making the assembly of interconnected services and the identification of available resources less cumbersome. It provides standardized components by delivering NuGet packages that resolve common issues associated with cloud deployments, offering predefined setups for tasks such as monitoring, diagnostics, and data transmission.

By introducing well-structured project templates to enhance the organization of applications and expedite the setup phase, and featuring pivotal projects such as Application Host and Default Services, .NET Aspire caters to professionals looking to construct applications that are scalable, fault-tolerant, and maintainable within cloud infrastructures.

Its dedication to cloud-centric development positions it as a significant asset within the .NET community, empowering developers to harness the full potential of cloud technologies for their software solutions.

Let's explore what Visual Studio has prepared for us regarding .NET Aspire. To do this, we will create a new project with the **.NET Aspire Starter Application** template. For this example, we will name it `SampleAspireProject`.

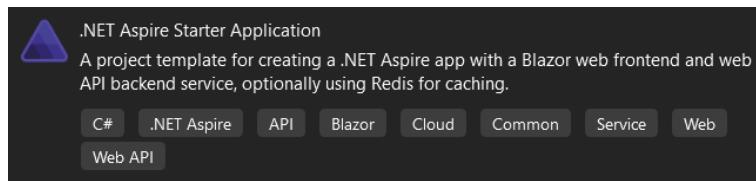


Figure 8.1 – The .NET Aspire Starter Application template

When you create a new project, Visual Studio will open an **Overview** page, providing links to Microsoft documentation about building and deploying our app and service discovery:

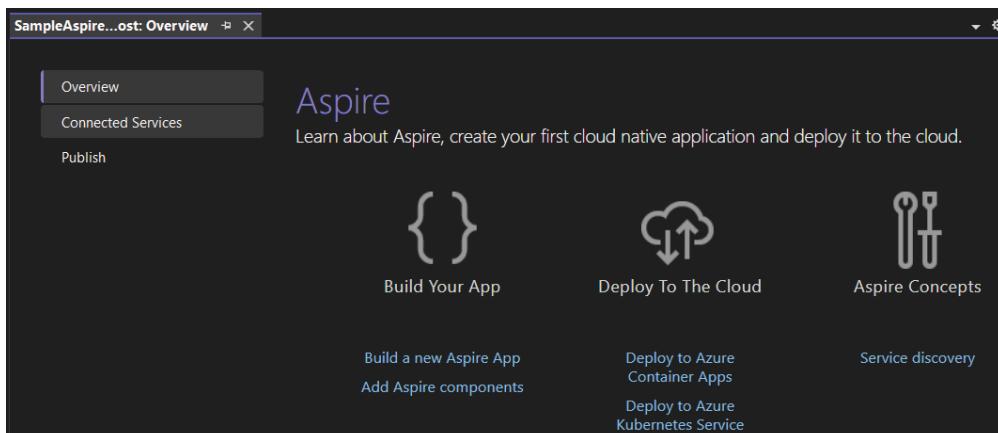


Figure 8.2 – The Aspire Overview page

On the same page, we can find more tabs; the second one is **Connected Services**. By clicking on this option, we can access two options that allow us to add service dependencies and service references.

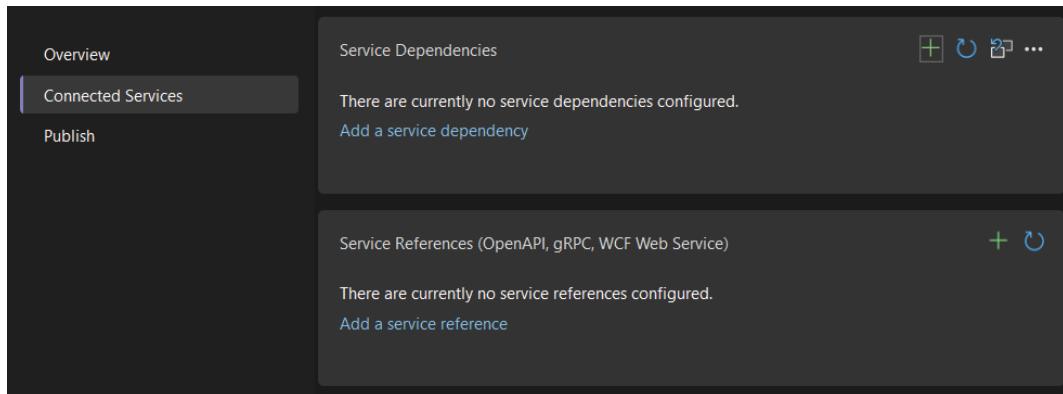


Figure 8.3 – Connected Services

Upon clicking the **Add a service dependency** link, a window opens up that allows us to select an Azure service to add to our solution.

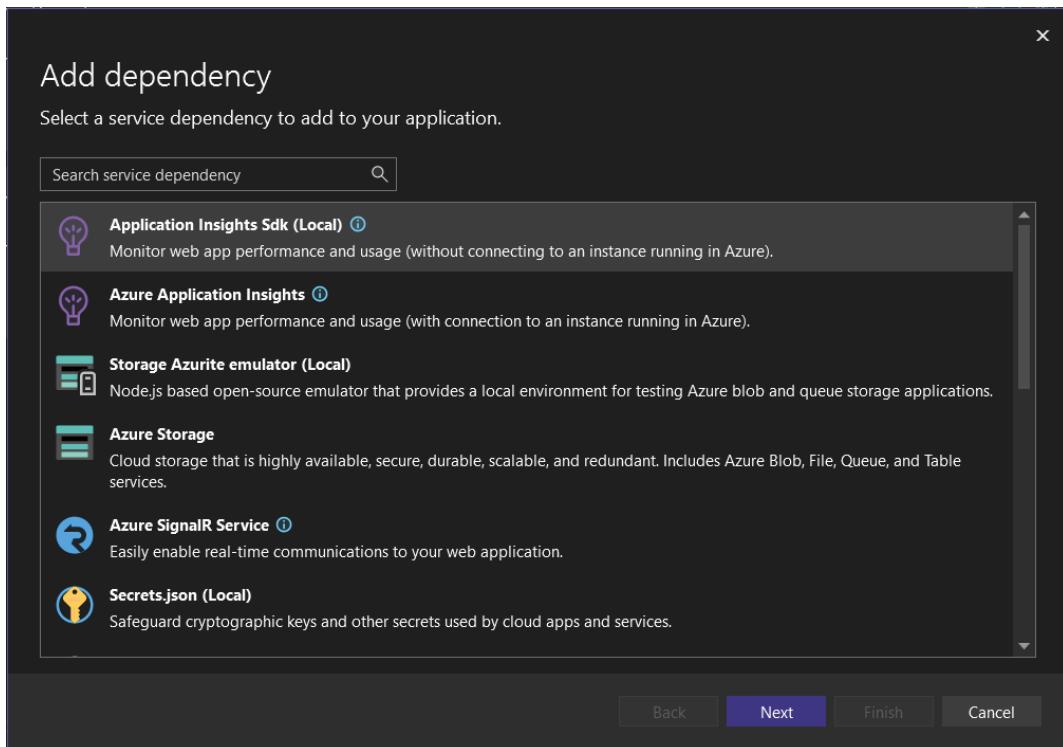


Figure 8.4 – Adding a dependency

Note that to be able to complete the process of adding a dependency, we need a valid Azure subscription.

Go back to the **Connected Service**, and click on the **Add a service reference** link. This will give us access to a window that enables us to select between three API specifications – **OpenAPI**, **gRPC**, and **WCF Web Service**.

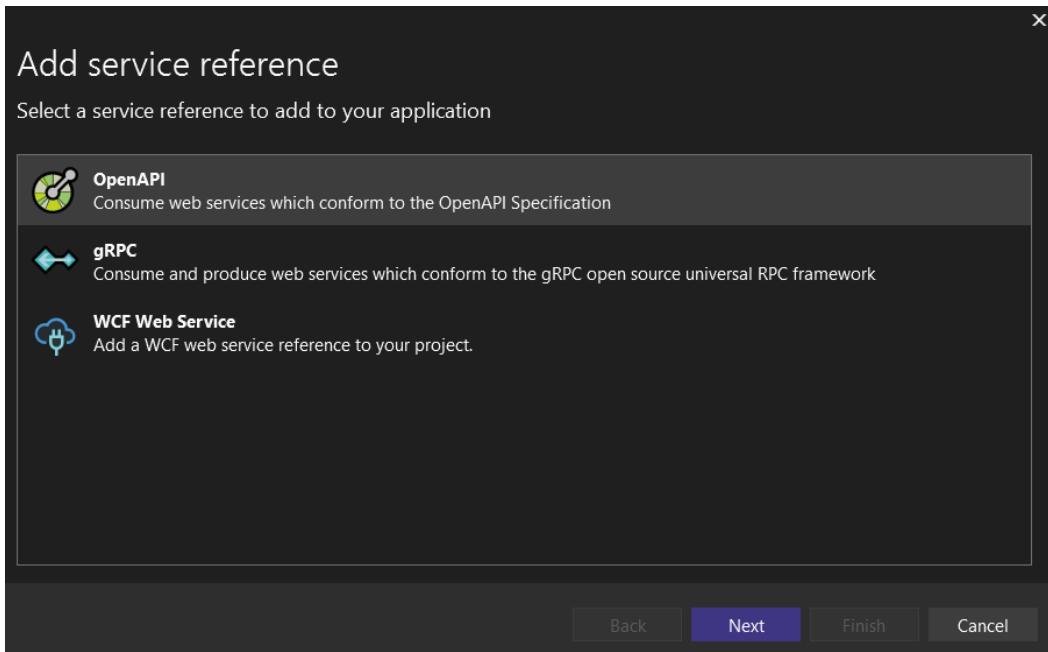


Figure 8.5 – Adding a service reference

The template we have used sets up four projects for our solution:

- **ApiService**: ASP.NET Core minimal APIs, serving as the backend logic layer
- **AppHost**: This is the project that hosts the configuration of our solution
- **ServiceDefaults**: Represents the shared project that contains default configurations and potential objects used across multiple services in the solutions

- **Web:** This Blazor project is dedicated to frontend development.

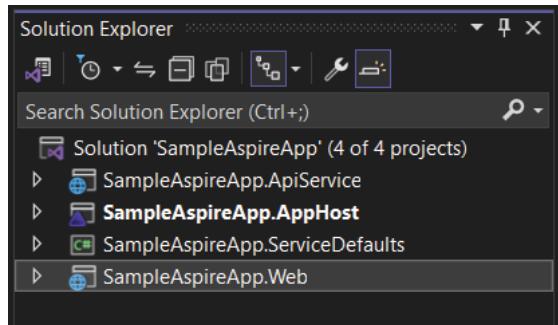


Figure 8.6 – Starter application components

The **AppHost** project is set as the starting project by default. So, when we launch the solution, the .NET Aspire template opens the **SampleAspireApp dashboard** window.

The screenshot shows the .NET Aspire dashboard titled "SampleAspireApp dashboard". On the left is a sidebar with "Resources", "Monitoring" (Console logs, Structured logs, Traces, Metrics), and other tabs. The main area is titled "Resources" with a table:

Type	Name	State	Start time	Source	Endpoints	Logs	Details
Project	apiservice	Running	2024-05-18 02:49:11	SampleAspireApp.ApiServic...	http://localhost:5437/weat...	View	View
Project	webfrontend	Running	2024-05-18 02:49:12	SampleAspireApp.Web.cspr...	http://localhost:5074	View	View

Figure 8.7 – The .NET Aspire dashboard

The dashboard provides crucial information about our solution through different metrics, such as logs, traces, and environment configuration. In the **Resources** panel, we can access each endpoint of our solution. In this template project, we retrieve **apiservice** and **webfrontend** in the dashboard.

After the local development of our cloud-native solution, we can directly publish it to our Azure platform. Note that we need an Azure Developer CLI installed. To create a publish profile, we can reopen the **Overview** window from the top bar menu (**Project | Overview**).

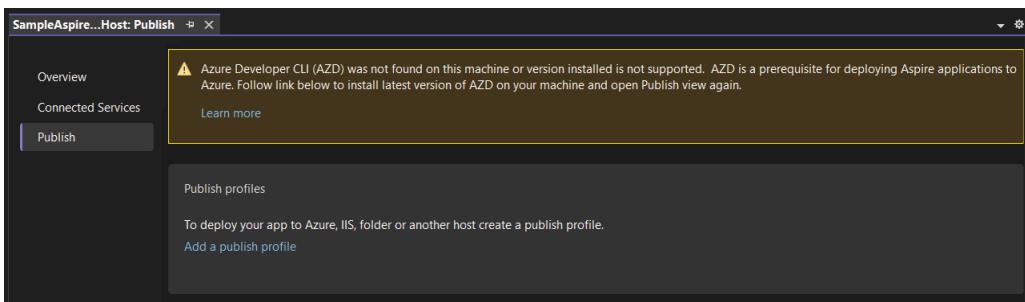


Figure 8.8 – Publish profiles

By using Visual Studio through .NET Aspire, we have set up a cloud-native application ready to be deployed. In the next section, we will explore Azure Functions development in Visual Studio.

## Exploring Azure Functions development in Visual Studio

Visual Studio provides several built-in templates for the Azure platform. As Azure Functions is the most popular cloud-native feature of Microsoft, we will focus on it in this section.

Put simply, Azure Functions is a serverless computing service offered by Microsoft Azure that allows developers to run small pieces of code without worrying about the underlying infrastructure.

Azure Functions is designed to respond to events from a variety of sources, including HTTP requests, timers, database changes, and many other Azure services. This event-driven model enables us to build applications that react to real-time data changes, automate tasks, and integrate with other systems seamlessly.

First, we need to ensure that the Azure development workload is installed on our Visual Studio instance by navigating to the Visual Studio Installer.

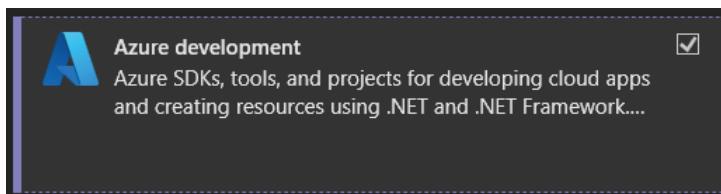


Figure 8.9 – An Azure development workload

Once the workload is installed, we can start by creating a new Azure Functions project, and for this example, we will name it `SampleAzureFunction`.

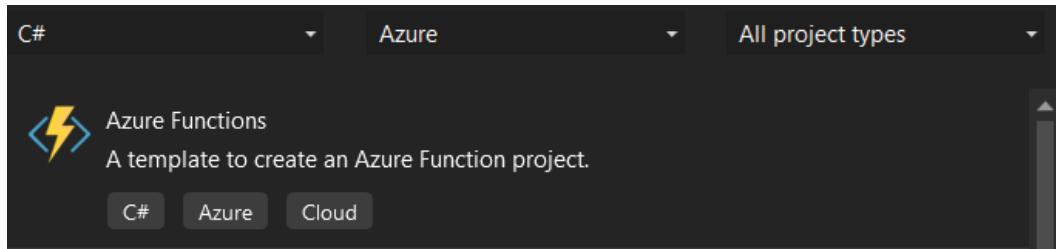


Figure 8.10 – An Azure Function project

For the next step, Visual Studio will lead us to an **Additional information** window, where we are asked to configure the worker, trigger, and authorization to use.

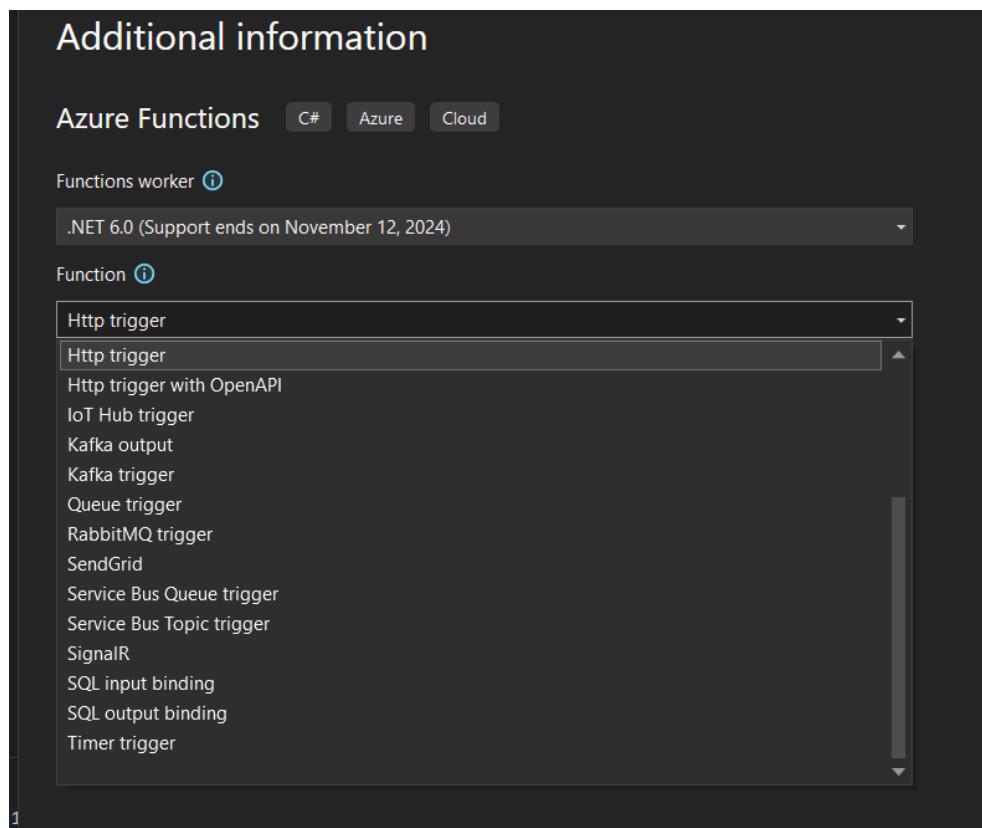


Figure 8.11 – Choosing an Azure Functions trigger

Let's explore and understand the different trigger options offered by Azure Functions:

- **Http trigger:** Allows you to create an HTTP endpoint that can be called to execute the function. This is useful for creating RESTful APIs.
- **Http trigger with OpenAPI:** Similar to the HTTP trigger but includes built-in support for OpenAPI (formerly Swagger), making it easier to design and document APIs.
- **IoT Hub trigger:** Enables functions to respond to events sent to an IoT Hub event stream. Useful for processing telemetry data from IoT devices.
- **Kafka output:** Writes a message to a Kafka topic. This is used when you want to publish messages to a Kafka topic from an Azure Functions.
- **Kafka trigger:** Consumes messages from a Kafka topic. This is useful for processing incoming messages from Kafka topics within an Azure Function.
- **Queue trigger:** Executes a function whenever a new message is added to an Azure Queue Storage queue. Ideal for processing queued tasks.
- **RabbitMQ trigger:** Consumes messages from a RabbitMQ queue. This is useful for integrating with RabbitMQ-based applications.
- **SendGrid:** Allows you to send emails directly from an Azure Function using SendGrid's email service. This is particularly useful for sending notifications or alerts via email.
- **Service Bus Queue trigger:** Responds to messages from a Service Bus queue. This is ideal for processing messages asynchronously.
- **Service Bus Topic trigger:** Responds to messages published to a Service Bus topic. This is useful for implementing pub/sub patterns.
- **SignalR:** Enables running real-time messaging web apps in a serverless environment. It's great for scenarios requiring real-time updates to clients.
- **SQL input binding:** Retrieves data from a SQL database and passes it to a function's input parameters. This is useful for querying databases and processing the results.
- **SQL output binding:** Writes data to a SQL database. This is useful for updating databases based on a function's execution logic.
- **Timer trigger:** Runs a function on a schedule, such as every minute, hour, or day. This is perfect for periodic tasks such as backups or reports.

Once we have chosen the function type, we have to select the level of authorization to rule our Azure Function. The three types are as follows:

- **Function:** This level restricts access to only those users who have been granted specific permissions to invoke a function. It's typically used when we want to control access to our function more granularly than what's provided by the other two levels.

- **Anonymous:** This level allows any client (authenticated or not) to call a function without needing to provide credentials. It's useful for public APIs or functions that don't require authentication.
- **Admin:** This level grants full administrative privileges to the function, allowing it to perform actions on behalf of the caller as if they were logged in as an administrator. It should be used cautiously due to its broad permissions.

Once we are all set, we can develop the logic of our function, and to respond to our stakeholders, the next step is to publish it to our Azure subscription:

1. Right-click on the project in **Solution Explorer** and select **Publish**.
2. Then, choose **Azure** as the target and click **Next**. Finally, select **Azure Function App (Windows)** for the specific target.

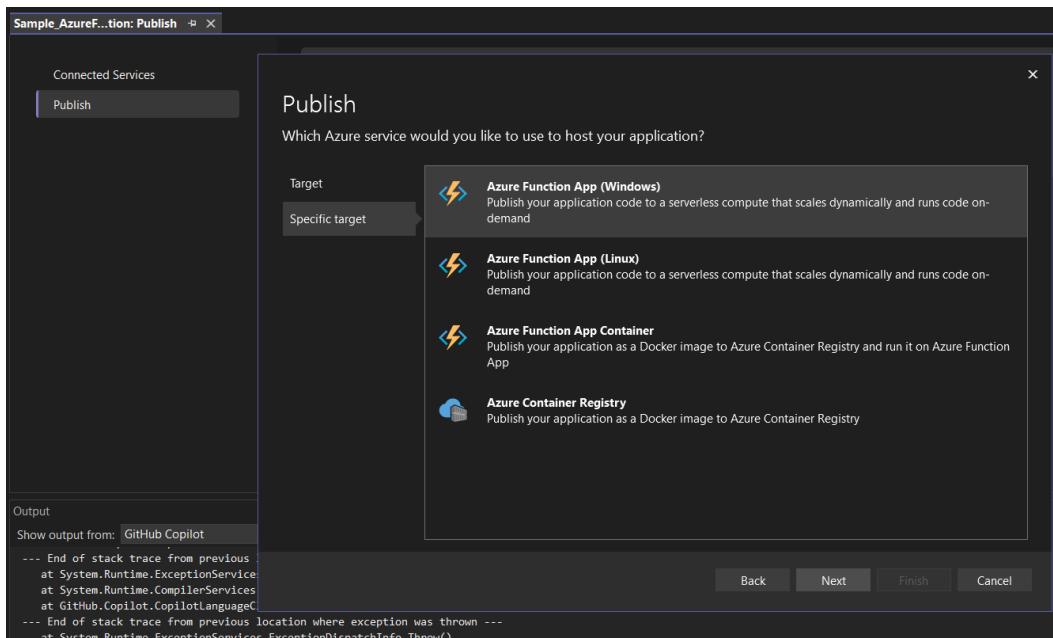


Figure 8.12 – Publishing a specific target

3. Then, we need to choose a unique name for our function app to ensure that it doesn't duplicate any existing names globally.
4. Next, we'll select our Azure subscription from the available options. We can either pick an existing resource group or create a new one. For the plan type, we'll opt for the **Consumption** plan to guarantee cost-effective execution based on actual usage.
5. We will choose a region that is geographically close to our users or services for better performance. If required, we will set up a general-purpose storage account. We'll also enable **Application Insights** to monitor and diagnose our function app.

6. During deployment, we need to make sure to select the **Run from package** option. This allows our function app to execute directly from the deployment package, enhancing performance and simplifying the deployment process. After reviewing all our settings, we'll click **Publish** to begin the deployment. A success message will appear once the deployment is completed successfully.
7. Finally, we will go to the Azure portal and navigate to our function app to ensure that it is running correctly. Through the portal, we can monitor logs, test endpoints, and manage our function.

As Azure is part of the Microsoft ecosystem, other cloud platforms can be used in our overall enterprise solution. Next, let's begin to explore how we can enhance our Google Cloud Platform development with Visual Studio.

## Exploring Google Cloud Tools for Visual Studio 2022

**Google Cloud Platform (GCP)** is a public cloud computing service developed by Google, offering a variety of cloud-based solutions. Just like other cloud platforms, these include services for computing, storage, networking, big data, machine learning, and the **Internet of Things (IoT)**. GCP allows us and businesses to create and deploy applications and services using Google's robust infrastructure, benefiting from the scalability, flexibility, and security of Google's worldwide network. In this section, we will explore the GCP extension for Visual Studio 2022.

First, we need to install the extension by going to **Extension Manager** through the top-bar menu (**Extension | Manage extensions...**) and searching for **Google Cloud Tools**:

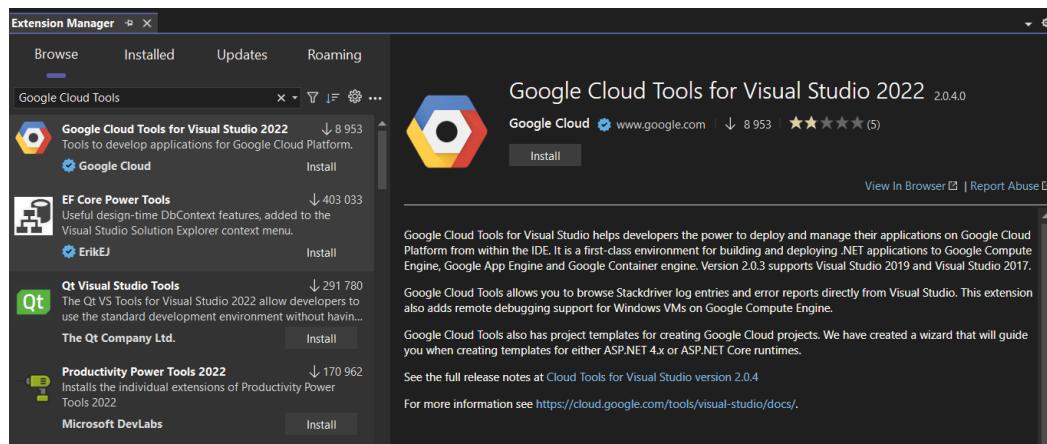


Figure 8.13 – Extension Manager | Google Cloud Tools

As with each installation of new Visual Studio extensions, we need to close our instance after we have clicked on the **Install** button to launch the beginning of the modification.

Since we have installed the Google Cloud Tools extension, we can connect to our Google account to deploy our project to App Engine. We achieve that by launching Google Cloud Explorer through the top-bar menu (**Tools | Google Cloud Tools | Show Google Cloud Explorer**):

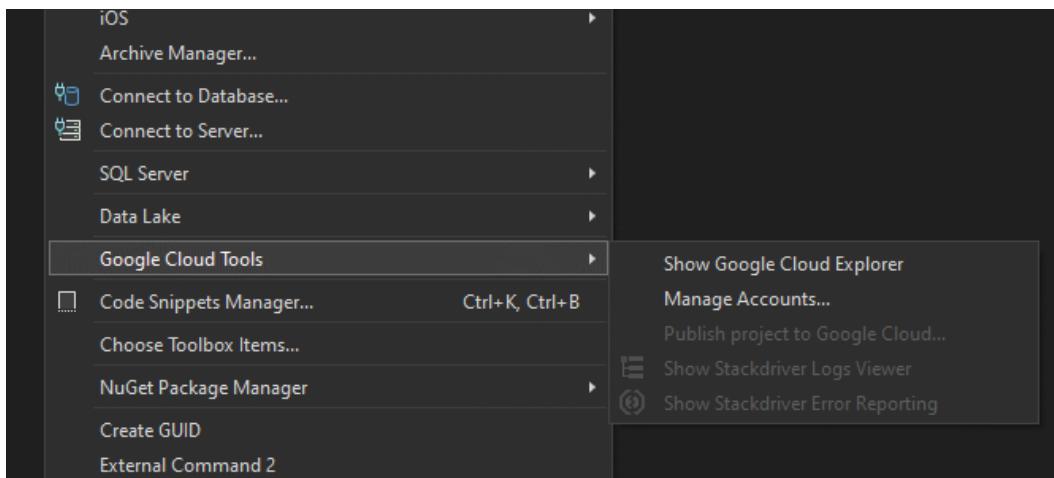


Figure 8.14 – Show Google Cloud Explorer

That opens up a new window to enter the information for our valid Google account.

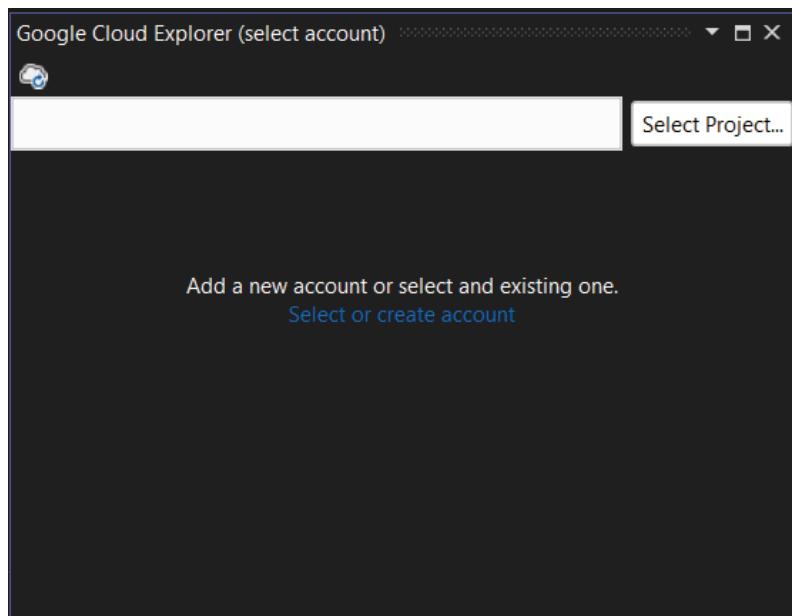


Figure 8.15 – Add a new account

We now have an integrated deployment assistant within the extension that deploys your application to the following compatible targets:

- Compute Engine for ASP.NET applications
- Flexible App Engine and Google Kubernetes Engine for ASP.NET Core applications

The deployment assistant automatically detects compatible targets for our project and guides us through the deployment process.

The deployment assistant can be called by selecting **Tools | Google Cloud Tools | Publish [PROJECT-NAME] to Google Cloud**, or by right-clicking on the project node in **Solution Explorer** and clicking on **Publish [PROJECT-NAME] to Google**.

Note that the **Publish [PROJECT-NAME] to Google Cloud** menu item will only be enabled if the start-up project of the solution is compatible with Google Cloud.

The deployment assistant displays the deployment targets on Google Cloud that are compatible with the selected project.

If we wish to change the project, we can open Cloud Explorer by clicking on **Tools | Google Cloud Tools | Open Cloud Explorer** and select the project we want to deploy.

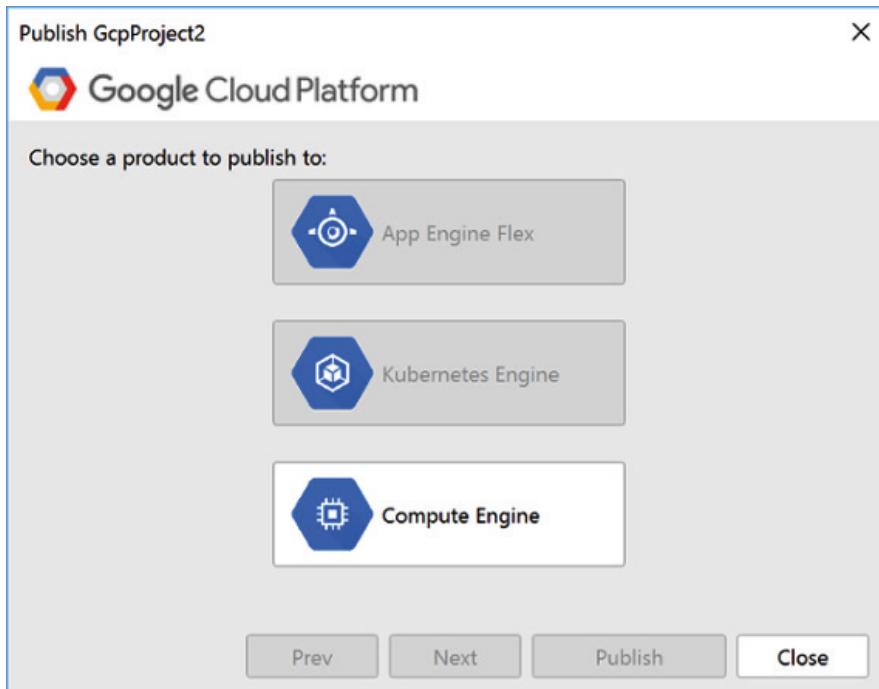


Figure 8.16 – Choosing a product to publish

Note that ASP.NET 4.x applications only run on Windows VMs on Compute Engine. To deploy our ASP.NET 4.x application on Compute Engine, follow these steps:

1. Open the deployment assistant by clicking on **Tools | Google Cloud Tools | Publish [PROJECT-NAME] to Google Cloud**.
2. Select **Compute Engine**.
3. Select the **Windows VM** instance and deployment credentials. The **Windows VM** must run **Internet Information Services (IIS)** and be able to run ASP.NET 4.x applications, such as a VM ASP.NET created by the deployment manager.
4. Select the credentials for deployment. To create Windows credentials, click on **Manage Credentials**.
5. Click on **Publish** to create our application and deploy it to the selected VM.

The deployment progress is displayed in the Visual Studio output window, and a progress indicator is shown in the system tray status bar of Visual Studio.

The Deployment of ASP.NET Core applications can be run in a Docker container so that your application can be deployed in the flexible App Engine and **Google Kubernetes Engine (GKE)** environments.

To deploy to the flexible environment, follow these steps:

1. Open the deployment assistant by clicking on **Tools | Google Cloud Tools | Publish [PROJECT-NAME] to Google Cloud**.
2. Select **App Engine Flex** to deploy the app to App Engine.
3. Enter the name of our application's version and traffic management choice.

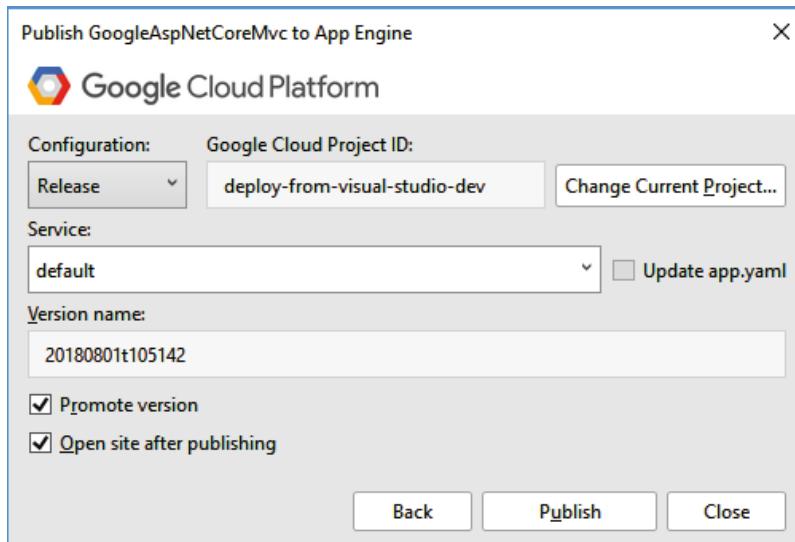


Figure 8.17 – Publishing AspNetCore to App Engine

The default version name is based on the current system time. We can specify another name. Note that, if we specify an existing version name, the previous version will be overwritten. The **Promoting version** checkbox allows us to choose whether this application version should receive 100% of the traffic. If this box is checked, the new application will receive all the traffic immediately after deployment.

4. Click on **Publish** to create your application and deploy it to the flexible App Engine environment.

And there we go – our application is deployed, and the progress is displayed in the Visual Studio output window.

To deploy to **GKE**, follow these steps:

1. Open the deployment assistant by clicking on **Tools | Google Cloud Tools | Publish [PROJECT-NAME] to Google Cloud**.
2. Select **Container Engine** to deploy our app.
3. Select a cluster for deployment, and enter the deployment name of your application, its version, and the number of replicated instances.

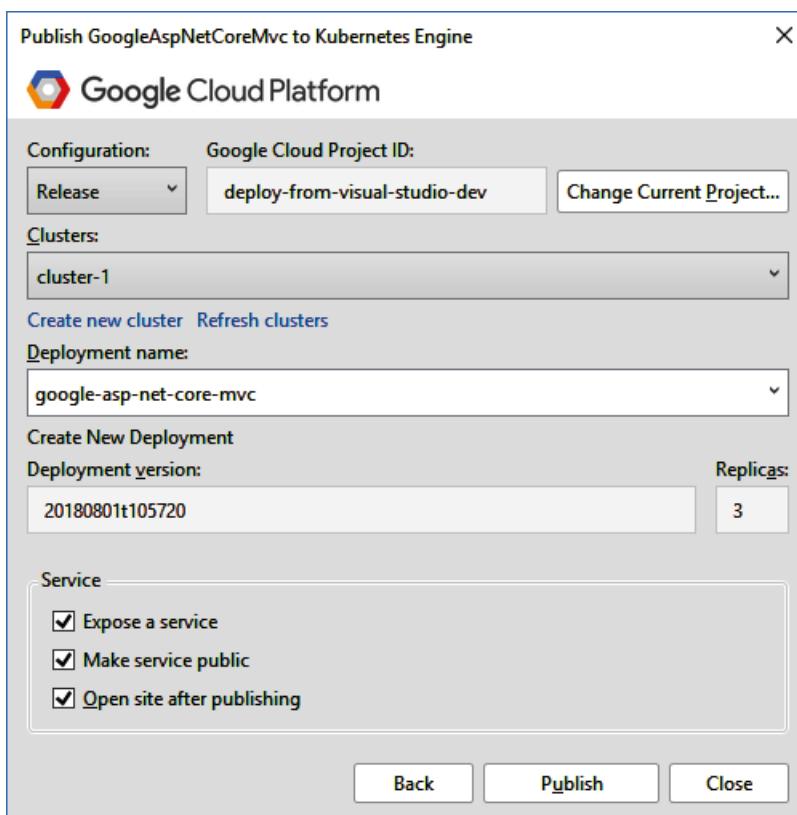


Figure 8.18 – Publishing AspNetCore to Kubernetes Engine

Note the three checkboxes allowing us to fine-tune our service:

- The **Expose a service** option refers to the ability to make our application accessible from outside the Kubernetes cluster. Essentially, it creates a Kubernetes service that exposes our application to external traffic.
- The **Make service public** option is closely related to **Expose a service** and specifically refers to making the exposed service publicly accessible on the internet. When we choose to expose our application as a service and make it public, GKE assigns a public IP address to your service, allowing external clients to access it.
- Selecting **Open site after publishing** will automatically open a web browser window pointing to the URL of our newly deployed application.

To create a cluster, follow these steps:

- I. Click on **Create new cluster**. We will be redirected to the cluster creation page in the Google Cloud console.
- II. To display the cluster in Visual Studio, click on **Refresh clusters**.

The deployment name is used when creating the Kubernetes deployment or, if selected, the name of the Kubernetes service that will run our application on the cluster. We can modify the name to make it more descriptive.

Note that if we use a name that already exists, the old deployment will be updated instead of creating a new one. The default version name is based on the current system time. We can specify another name.

We can choose to expose a Kubernetes service on the internet. By exposing a public service on the internet, we get a public IP address that we can use to access our service outside of our cluster.

4. Click on **Publish**.

And there we go – our application is containerized in a Docker image and deployed within our container. If our application is an exposed service, Visual Studio will wait for the service's IP address to become available.

Now that we have seen how easy it is to deploy our applications to Google Cloud Platform gracefully with Google Cloud Tools, we will move on to the cloud platform of Amazon through the AWS Toolkit in the next section.

## Exploring the AWS Toolkit

The AWS Toolkit for Visual Studio is an extension designed to enhance the development experience for creating, testing, and deploying .NET applications on **WS**. In this section, we will explore how this extension simplifies working with AWS services within Visual Studio.

First, we need to install the extension by going to **Extension Manager** through the top-bar menu (**Extension | Manage extensions...**) and searching for AWS Toolkit with Amazon Q.



Figure 8.19 – AWS Toolkit with Amazon Q

As with each installation of new Visual Studio extensions, we need to close our instance after we have clicked on the **Install** button to launch the begin the modification.

Key AWS services that are integrated through the toolkit include **Amazon Simple Storage Service (S3)**, **Amazon Elastic Compute Cloud (EC2)**, AWS Elastic Beanstalk, and Amazon DynamoDB.

Once the extension is installed, we can start to configure it through the top-bar menu – **Extensions | AWS Toolkit | Getting Started**.

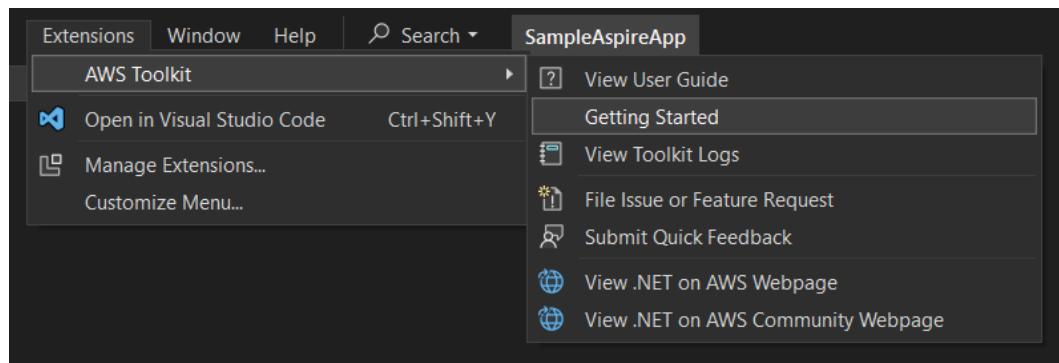


Figure 8.20 – AWS Toolkit | Getting Started

That will allow us to connect to our AWS subscription to set up both of the main features of the AWS Toolkit:

- **AWS Explorer:** A central hub for navigating through AWS services, monitoring storage, and managing resources directly from Visual Studio.
- **Amazon Q:** Represents a comprehensive suite of AI-powered tools designed to facilitate better decision-making, increase productivity, and streamline operations across various domains within an organization, leveraging the power of generative AI and deep integration with AWS services.

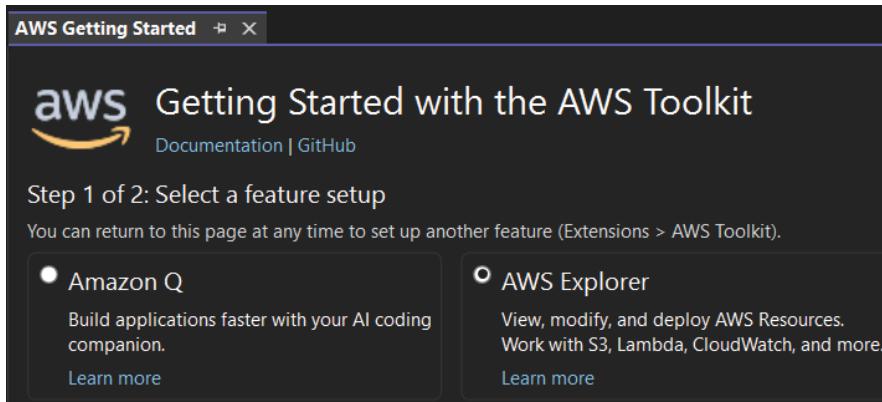


Figure 8.21 – AWS Getting Started

AWS Explorer is designed to provide us with a seamless way to interact with various AWS services directly from our development environment. It acts as a bridge between the local development environment and the cloud, allowing us to perform tasks such as the following:

- **Creating and managing AWS resources:** It allows us to easily create new instances on Amazon EC2, manage storage buckets in Amazon S3, and configure settings without leaving Visual Studio. We can now retrieve four project templates about AWS by adding a new project to our solution.

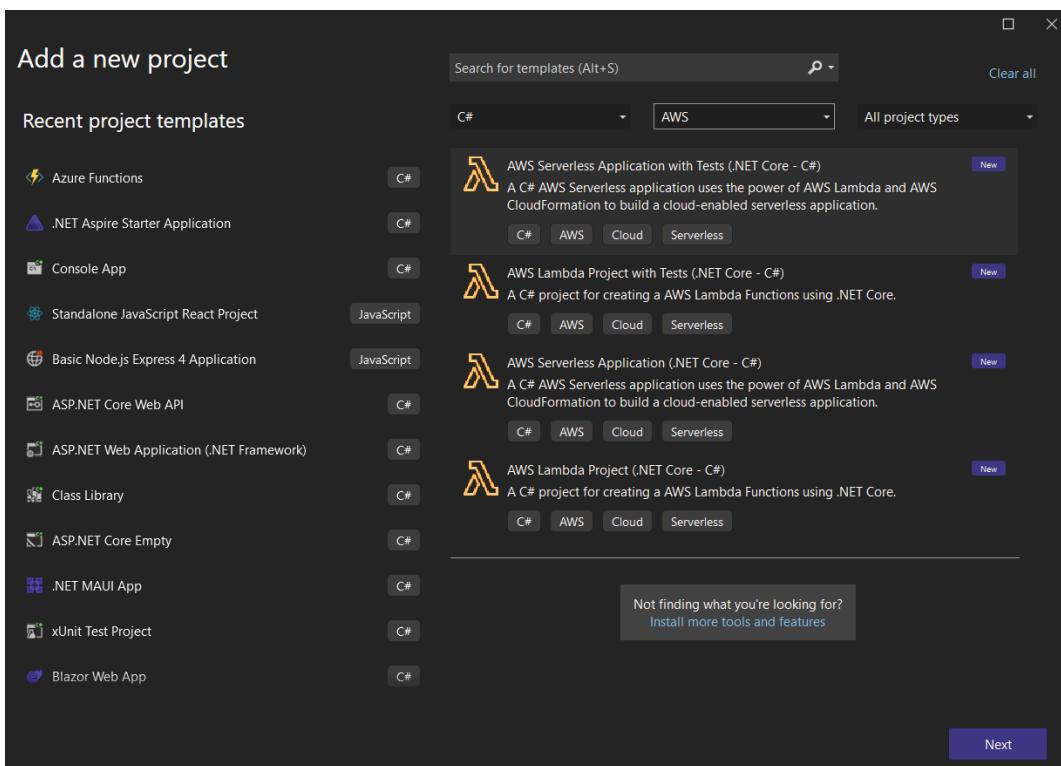


Figure 8.22 – An AWS project template

- **Deploying applications:** With support for AWS Lambda, we can deploy functions and applications to our AWS subscription.
- **Monitoring and management:** Through integration with AWS CloudFormation, we can manage infrastructure as code, ensuring consistency, repeatability, and version control over our AWS deployments.

Let's consider a scenario where we want to deploy a simple serverless function using AWS Lambda with this toolset. Here's how we might do it:

1. **Write the function locally:** First, we write a .NET Core function locally, testing it thoroughly to ensure that it works as expected.
2. **Configure the deployment settings:** Using AWS Explorer, we configure the deployment settings, including selecting the appropriate runtime (.NET Core), setting up any necessary environment variables, and specifying the IAM role that will execute the function.

3. **Deploy the function:** With just a few clicks or commands, we deploy the function to AWS Lambda. The AWS Explorer handles the packaging of the function code, uploading it to AWS, and configuring the necessary triggers or event sources.
4. **Monitor and manage:** Post-deployment, we can use the AWS Explorer to monitor the function's performance, view logs, and make updates or changes as needed, all from within our development environment.

This example illustrates how the AWS Explorer integrates with AWS services to simplify the development life cycle for .NET Core applications on AWS, providing a unified interface to create, deploy, and manage cloud resources.

## Summary

In this chapter, we explored the advanced aspects of cloud integration and services using Visual Studio 2022. Emphasizing the importance of cloud computing in modern application development, we provided a detailed guide on leveraging Visual Studio's powerful tools and extensions to build and manage cloud-based applications directly through the IDE.

As we conclude this chapter, we mark the end of the second part of our journey in mastering core development skills. From advanced web development to multi-platform, machine learning, and now the advanced cloud, we have covered how Visual Studio enhances our cutting-edge development.

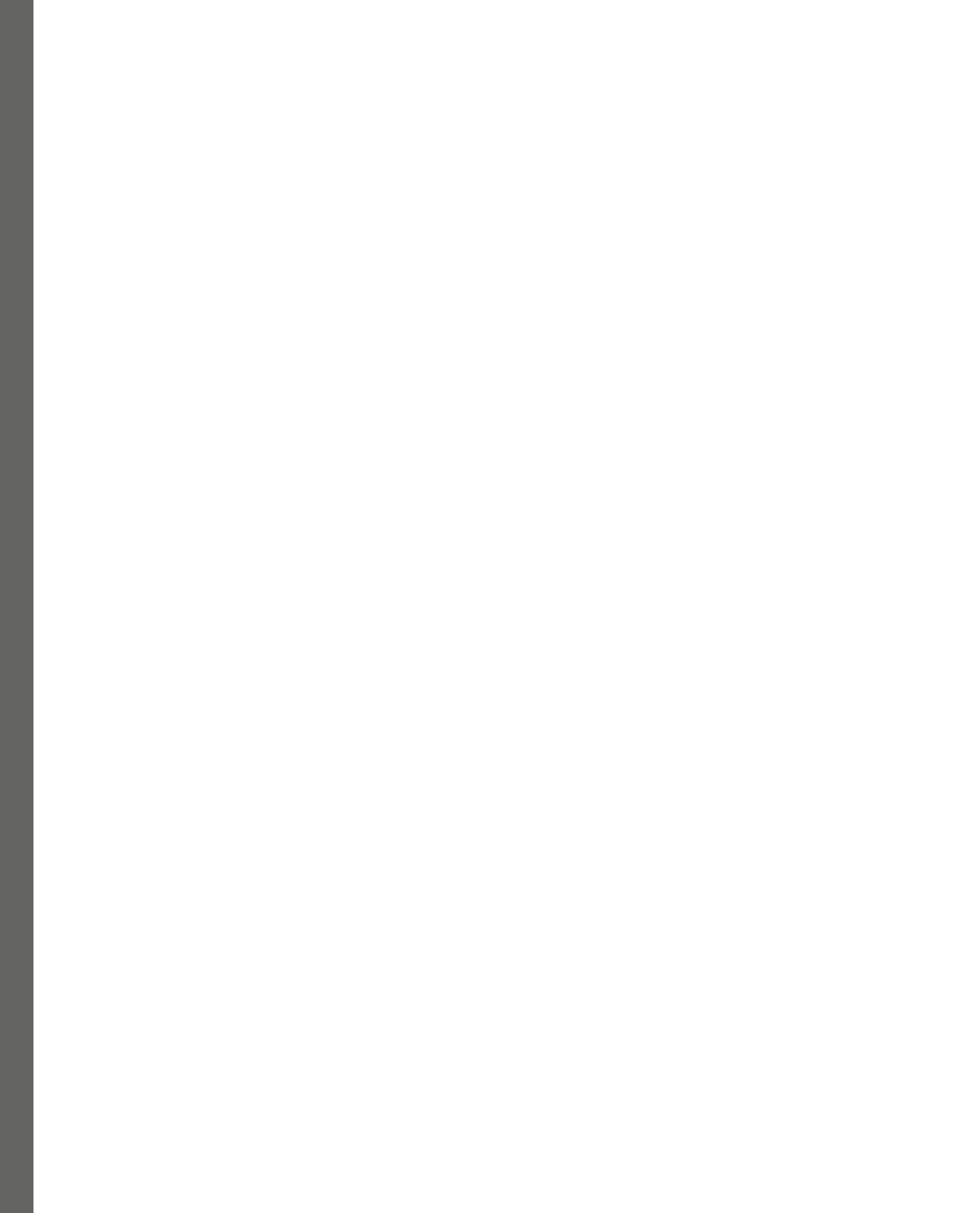
In the upcoming chapter, we'll continue to expand our horizons, delving into the world of DevOps, starting by handling advanced Git Workflow directly within Visual Studio.

# Part 3: Streamlining Collaborative Development with DevOps Practices

In this third part, we focus on how Visual Studio streamlines modern DevOps workflows. You'll explore how Visual Studio simplifies advanced Git workflows, automates continuous integration with GitHub Actions, and facilitates seamless collaboration through Azure DevOps. Additionally, you'll leverage Visual Studio's container tools for Docker, empowering you to optimize development, testing, and deployment within a unified DevOps environment.

This part has the following chapters:

- *Chapter 9, Handling Advanced Git Workflows*
- *Chapter 10, Continuous Integration with GitHub Actions*
- *Chapter 11, Collaborative Development with Azure DevOps*
- *Chapter 12, Visual Studio Container Tools for Docker*



# 9

## Handling Advanced Git Workflows

In this chapter, we will dive deep into the world of Git integration within Visual Studio, focusing on practical skills that will enhance our software development workflows. We will focus on advanced features provided by Visual Studio 2022. This chapter is designed to equip you with the tools and techniques needed to manage repositories efficiently, resolve conflicts effectively, and leverage the power of interactive staging.

In this chapter, we're going to cover the following main topics:

- Managing a repository through Visual Studio
- Resolving conflicts through Visual Studio
- Exploring interactive staging in Visual Studio

In order to streamline our process for collaborative development experiences, we will explore mastering the management of branches, resolving conflicts, and implementing interactive staging.

### Technical requirements

While writing this chapter, I used the following versions of Visual Studio:

- Visual Studio Enterprise 2022 version 17.12.0
- Preview 1.0

## Managing a repository through Visual Studio

If you're familiar with development, you'll have likely already used tools such as **GitKraken** or **Sourcetree**, among others. For now, all our daily usage of Git and repository management can be handled entirely within Visual Studio. In this section, we'll explore how we can leverage Visual Studio to manage our repositories without having to exit our favorite IDE.

### Exploring the Manage Branches window

Visual Studio 2022 introduced a window named **Manage Branches**. To open it, we have several options. The first option is using the top **Git | Manage Branches** menu option:

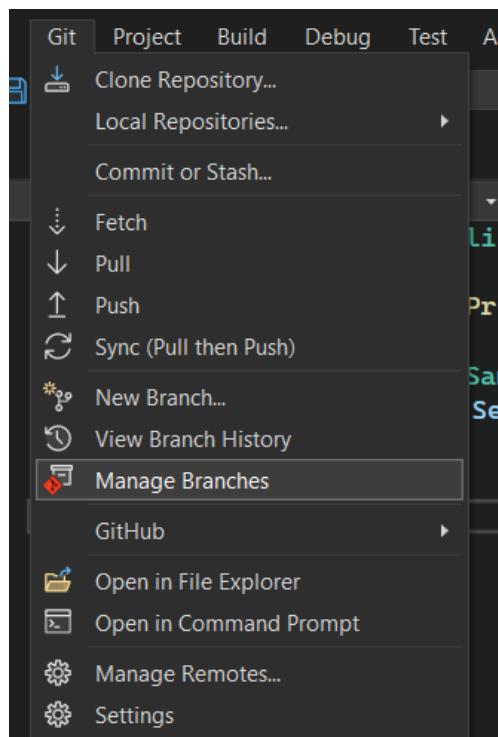


Figure 9.1 – Manage Branches top-bar menu option

The second option is going to **Git Changes** windows, then using the three dots in the top-right corner of the window and selecting **Manage Branches**:

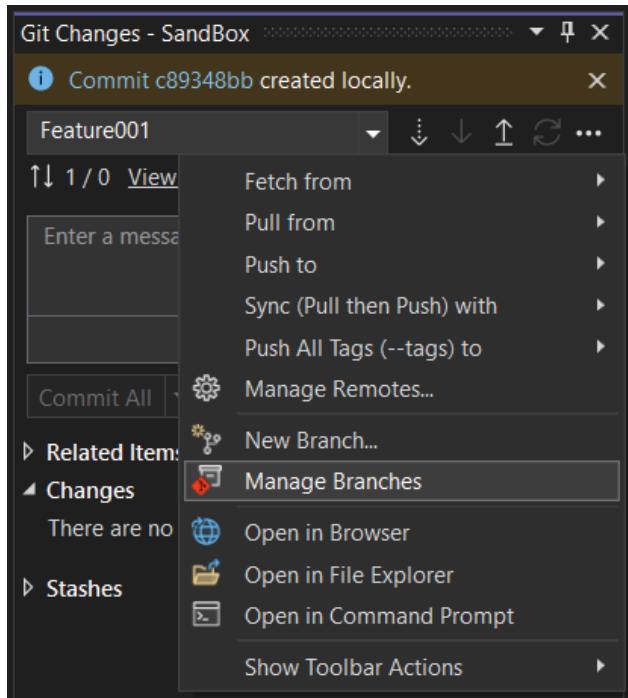


Figure 9.2 – Manage Branches Git Changes menu

Furthermore, in the **Git Changes** window, if we have pending outgoing or incoming commits, we can directly click on the **View Commits** link below the **Branches Combo** box.

Notice that we can search for this feature by using the **Feature Search** box with the *Ctrl + Q* shortcut.

This action will open the **Git Repository** window:

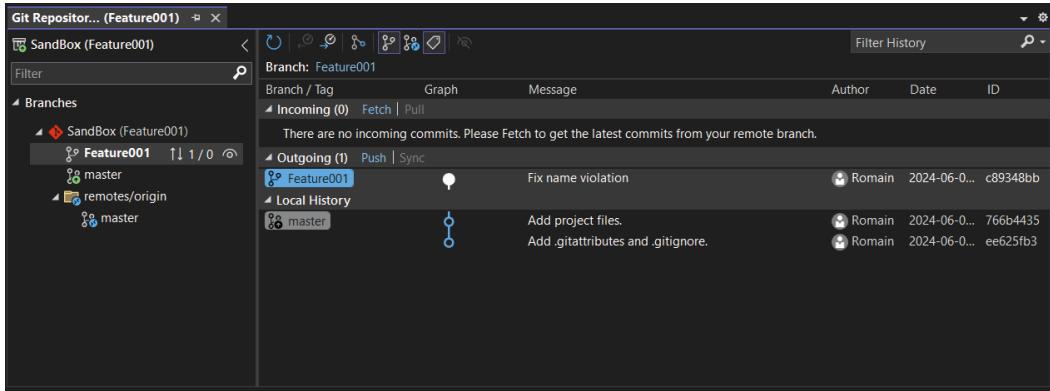


Figure 9.3 – Git Repository window

In this window, we can find all the branches of our repository. When we click on a branch, we can see details about its state, in the right part of the window, which is organized into three sections:

- **Incoming:** This section shows changes from other branches that have not yet been merged into the currently selected branch. These changes could come from any remote branch that has updates compared to our local branch.
- **Outgoing:** This section displays changes that have been made locally but have not yet been pushed to the remote repository. This includes commits that we've made since our last synchronization with the remote branch.
- **Local History:** This section offers a detailed view of the commit history for the currently selected branch. Unlike the **Incoming** and **Outgoing** sections, which focus on changes relative to other branches, the **Local History** section focuses on the chronological progression of commits within the selected branch.

Through this window, we can organize our view using the features offered by the toolbox menu:

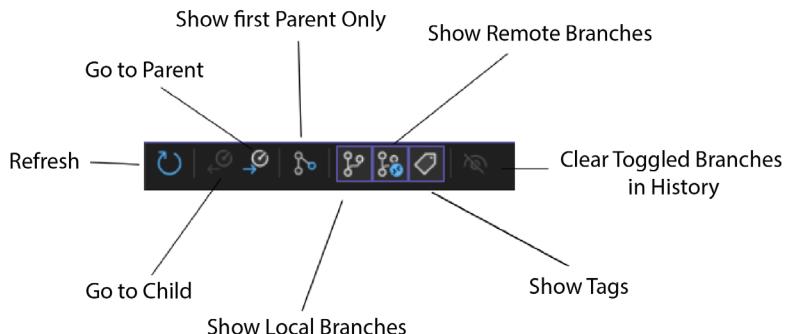


Figure 9.4 – Manage Branches toolbox

This toolbox offers us the following options:

- **Refresh**: Updates the list of branches, tags, and other repository objects displayed.
- **Go to Child**: Navigates directly to a child branch of the currently selected branch. A child branch is typically created from another branch (the parent) as a result of operations such as branching or merging.
- **Go to Parent**: Enables navigation to the parent branch of the currently selected branch. The parent branch is the branch from which the current branch was originally created.
- **Show first Parent Only**: Filters the view to display only the immediate parent branch of the current selection.
- **Show Local Branches**: Hides any remote branches from the view, allowing you to concentrate on branches that exist only on our local machine.
- **Show Remote Branches**: Filters the view to display only the branches that exist on the remote repository.
- **Show Tags**: Makes Git tags visible in the **Git Repository** window alongside branches. Git tags are references to specific points in our repository's history, often used to mark release versions.
- **Clear Toggled Branches in History**: Provides a way to clean up branches' histories by removing the toggles next to branches in the history panel. This can make it easier to see the commit history of the currently selected branch without being distracted by toggles indicating the presence of other branches in the history.

With this toolbox, we can interact with different branches and perform various Git operations. To access these options, we open the context menu by right-clicking on the desired branch or commit.

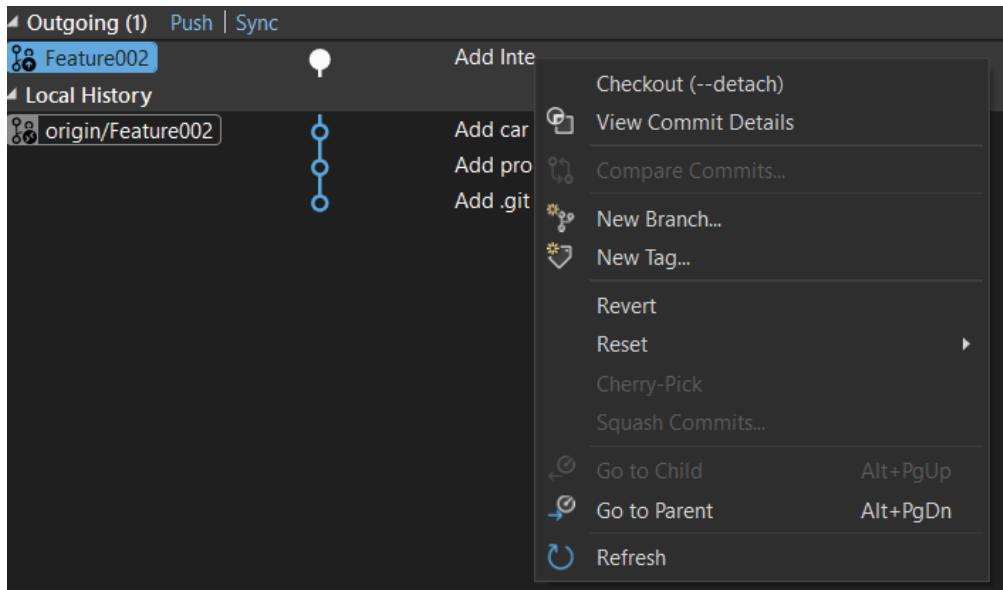


Figure 9.5 – Git command menu

From this menu, we can process classic Git commands, such as creating new branches and tags, viewing commit details, reverting commits, and even deleting changes. Some of the options may be disabled depending on the state of the commit.

One of the handy options provided is **Checkout(--detach)**.

## Looking at Checkout(-- detach)

In the Git command menu (*Figure 9.5*), we can find the **Checkout (--detach)** option, which allows us to revert to an earlier state of our repository, which is particularly useful for testing or running code as it existed at a particular moment.

In some cases, we might want to check out the latest commit of a remote branch to quickly review a pull request and evaluate the most recent updates. To do this, we first need to ensure that we have fetched and updated our local copy of the branch. Then, we can right-click on the remote branch of interest and select **Checkout Tip Commit (--detach)**:

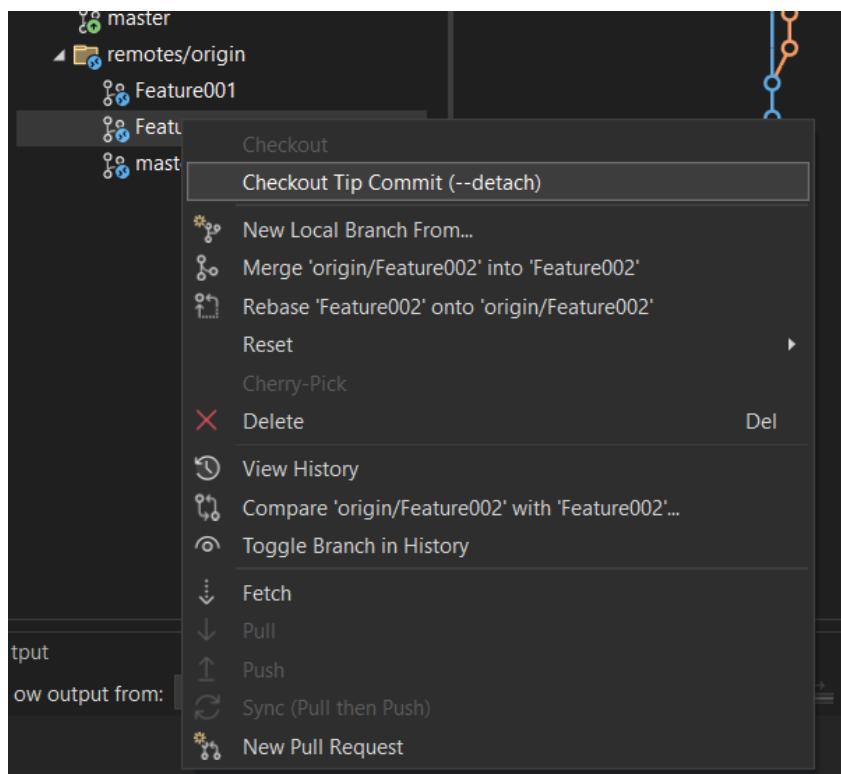


Figure 9.6 – Checkout Tip Commit (--detach)

It's crucial to note that any commits made while in a `detached HEAD` state aren't linked to a specific branch. As a result, Git might remove these commits once you switch to another branch, as they become vulnerable to being deleted. Therefore, to safeguard our work, it's advisable to start a new branch before moving away from a detached HEAD state if we want to keep potential changes.

## Handling multiple repositories

For greater convenience, or depending on our team's organization, we might come across solutions built across several repositories. This could be a nightmare to deal with. Visual Studio introduces multi-repo branching, which streamlines this use case.

Both the status bar and the **Git Changes** tool window in Visual Studio now include enhanced branch selection features that support working with multiple repositories. These tools allow for easy switching between branches and facilitate branch management across all our currently active repos. To swiftly change branches within any active repository, simply expand the repository tree in the branch picker and select the branch you wish to switch to.

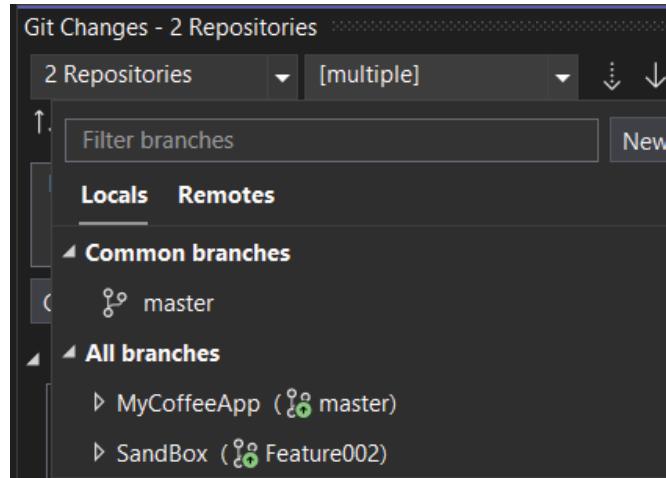


Figure 9.7 – Select repository

For better efficiency, Visual Studio offers us the capability to work with multiple repositories as if they were one. Indeed, we can create a new branch across our multiple repositories by using the top **Git | New branch...** menu option.

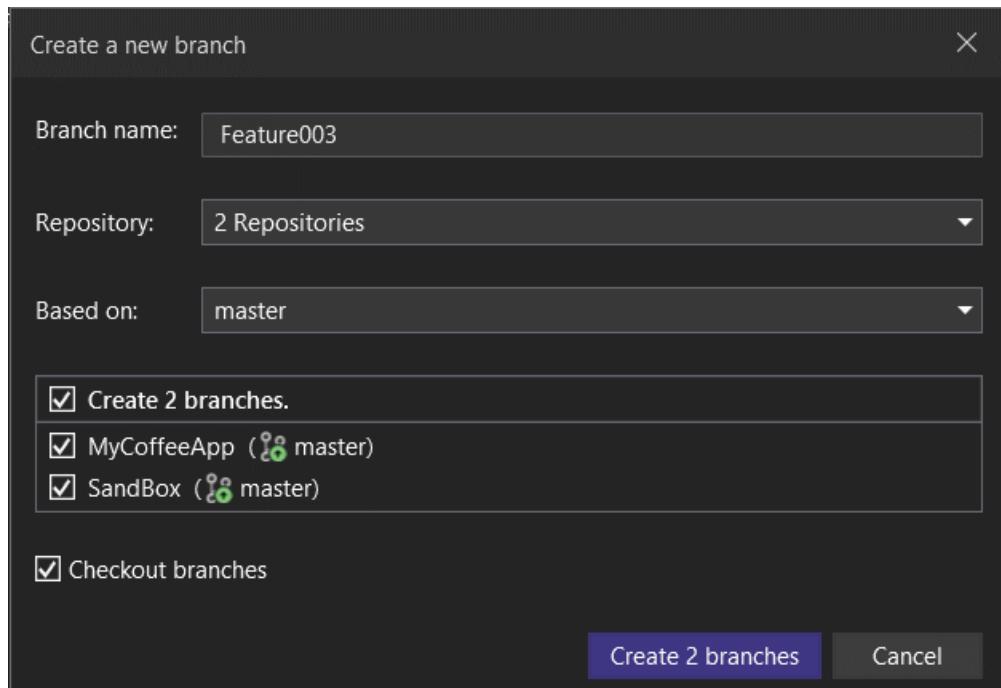


Figure 9.8 –Creating a new branch in multi-repositories

Here, we can choose which repository(s) to include when creating the new branch using the checkboxes.

Finally, we can access and manage all repositories and branches through the **Manage Branches** feature, allowing us to interact with them seamlessly.

Now that we have learned how to leverage Visual Studio 2022 to manage branches, using the **Manage Branches** and multi-repository features, in the next section, we will explore an important part of a shared code base: resolving conflicts.

## Resolving conflicts through Visual Studio

Resolving conflicts involves identifying and merging changes made by multiple contributors to the same lines of code. This process ensures that everyone's contributions are integrated smoothly, maintaining the integrity and consistency of the shared code base. By understanding and effectively managing these conflicts, teams can ensure that their software development process remains efficient and productive, fostering a collaborative environment where contributions from all team members are valued and integrated seamlessly. In this section, we will see how Visual Studio 2022 allows us to handle this process.

Git excels at seamlessly integrating file modifications under normal conditions, provided that the content of the files doesn't undergo significant alterations between updates. When our branch lags significantly behind the primary branch, it's advisable to rebase our branches prior to initiating a pull request. This process ensures that our branch can be smoothly incorporated into the main branch without encountering conflict issues.

Despite its proficiency in resolving changes using the history in our repository, merge changes are sometimes not clear, and Git stops the merge and informs us about file conflicts.

So, in that case, when we pull the remote branch to our local repository, Visual Studio will warn us in the **Git Changes** window with a message and list the files that experiencing conflicts with the merge.

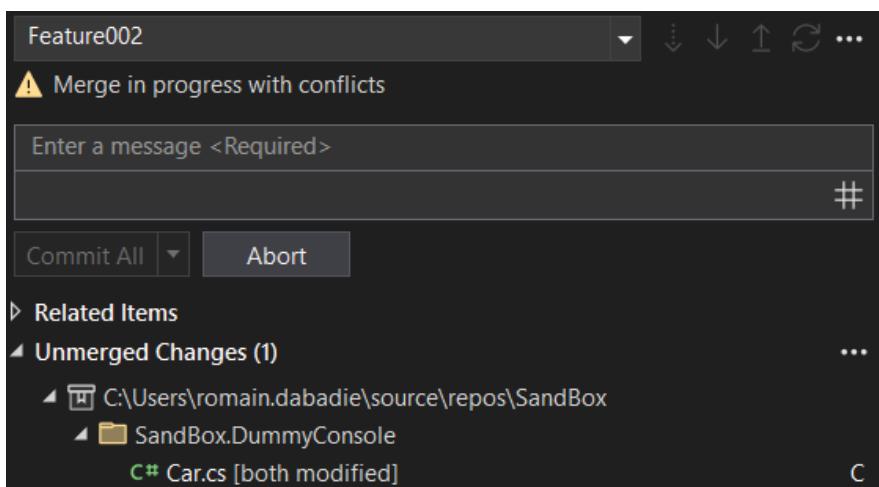


Figure 9.9 – Merge conflicts arising

In this example, the `Car.cs` file was modified on both the remote and local branches. In such a case, we must complete the merging process by ourselves. To do that, we will double-click on the concerned file, which will open the file in resolve conflict mode.

```

Car.cs* # X
↑ ↓ Show Conflicts Only Take Incoming Take Current Compare Accept Merge
[< ->] 1 Conflicts (1 Remaining)
    Incoming: Remote
10 {
11     public int Id { get; set; }
12     public string Name { get; set; }
13     public string Description { get; set; }
14    public string Models { get; set; }
15 }
16
17

    Current: Local
10 {
11     public int Id { get; set; }
12     public string Name { get; set; }
13     public string Description { get; set; }
14 ! public string Model { get; set; }
15
16 }
17

Result: SandBox.DummyConsole/Car.cs
10 {
11     public int Id { get; set; }
12     public string Name { get; set; }
13     public string Description { get; set; }
14
15 }

110% [gear icon]

```

Figure 9.10 – Resolve conflict mode

In this mode, we can see three sections:

- **Incoming:** These are the modifications made in the branch we are trying to merge with our current branch.
- **Current:** These changes refer to the modifications we have made in our current branch.
- **Result:** This section consolidates the result of the merging. Notice that we can manually edit this part at our convenience.

We may have unique preferences for how conflict windows are displayed. To adjust these settings according to personal convenience, simply click on the gear icon located in the upper-right corner of the interface.

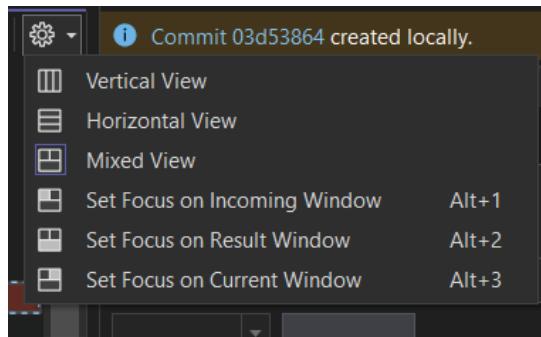


Figure 9.11 – Change the resolve conflict view

This way, we can switch between three modes:

- **Vertical View:** The **Result** section is placed between the **Incoming** and **Outgoing** sections
- **Horizontal View:** The **Result** section is placed below the **Incoming** and **Outgoing** sections
- **Mixed View:** Here, the **Incoming** and **Outgoing** sections are side by side and the **Result** section is below them

The resolve conflict mode also provides a toolbox to enable us to easily resolve conflicts.

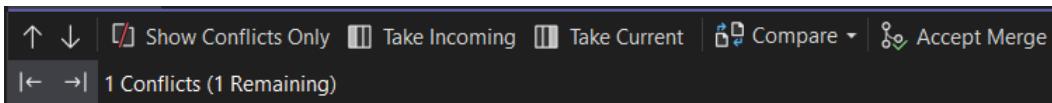


Figure 9.12 – Resolve conflict toolbox

This toolbox allows us to use the **Take Incoming** button (or press *F10*) to automatically accept all changes coming from the other branch. Alternatively, we can click the **Take Current** button (or press *F11*) to retain our current version of all conflicting changes.

In the left corner, the arrows enable us to navigate through the differences and conflicts across the file. This way, we can resolve conflicts line by line. To combine both modifications, we can use the checkboxes on the left side of each section. We can see them in *Figure 9.10*.

After successfully resolving all conflicts, a notification indicating **0 Remaining** will appear near the arrows in the top-left corner of the screen. This signifies that there are no unresolved conflicts. To finalize the merge operation, click on the **Accept Merge** button.

Finally, after we accept the merge and repeat the process in all conflict files, we use the **Git Changes** window to create a merge commit and resolve the conflict. In our daily use cases, we sometimes write code that we don't want to commit immediately or at all. Git offers us fine-grained control over which changes are committed, a feature known as staging. In the next section, let's explore how Visual Studio streamlines this process.

## Exploring interactive staging in Visual Studio

The staging area is where Git stores information about what will go in our next commit.

Staging allows us to select which changes we want to include in the next commit. It is a crucial Git feature. The main reason is that it provides a level of granularity and control over what gets committed. By staging only the necessary changes, we make the project history cleaner and easier to understand.

For instance, imagine we work on a feature that involves several changes in two functions. We might have finished making changes for one function but are still working on the other. With staging, we can commit the first function without including the second. This way, our commit reflects the state of the feature at that point in time, and our teammates can integrate the stable part without getting distracted by incomplete tasks.

Since version 17.6, Visual Studio 2022 has integrated interactive staging features. In this section, we will explore how.

When we make modifications to the files, these modifications will appear above the **Changes** section in the **Git Changes** window.

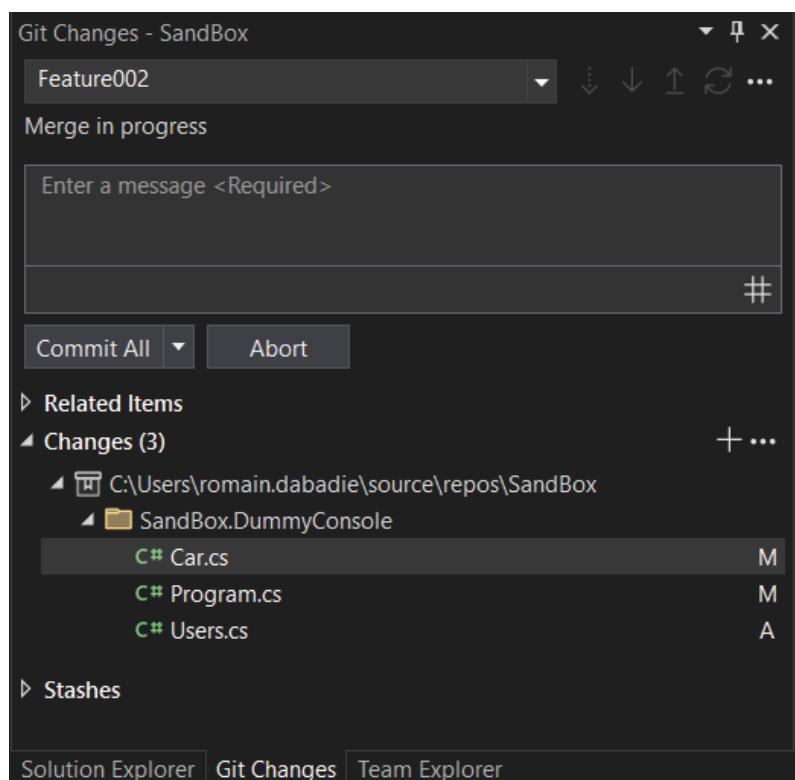


Figure 9.13 – Changes section

In this example, we made changes to three files. By default, Visual Studio displays the **Commit All** button, which **stage all** changes in the files and then commits them using the message entered in the text box. As if we executed directly those command:

```
git add .
git commit -m "Your commit message here"
```

If we want to stage only the `Car.cs` file, we can right-click on it and select **Stage**; alternatively, we can use the + button that appears on the right when the file is selected:

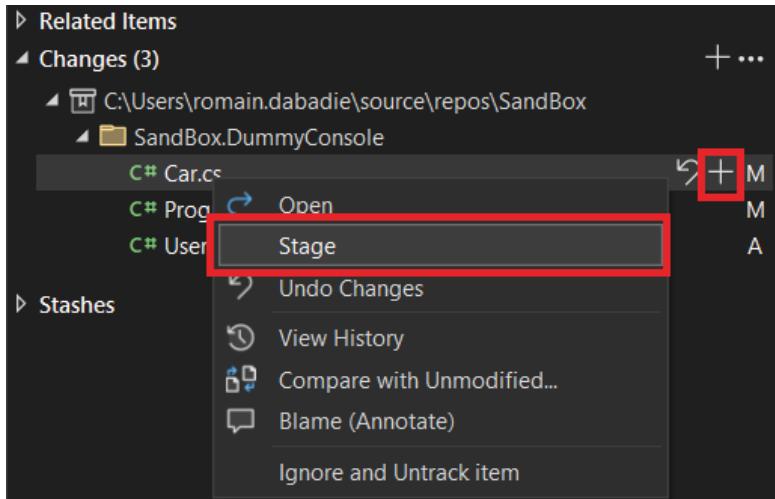


Figure 9.14 – Staging a file

With this, the **Staged Changes** section will appear, and we can find the `Car.cs` file listed under it:

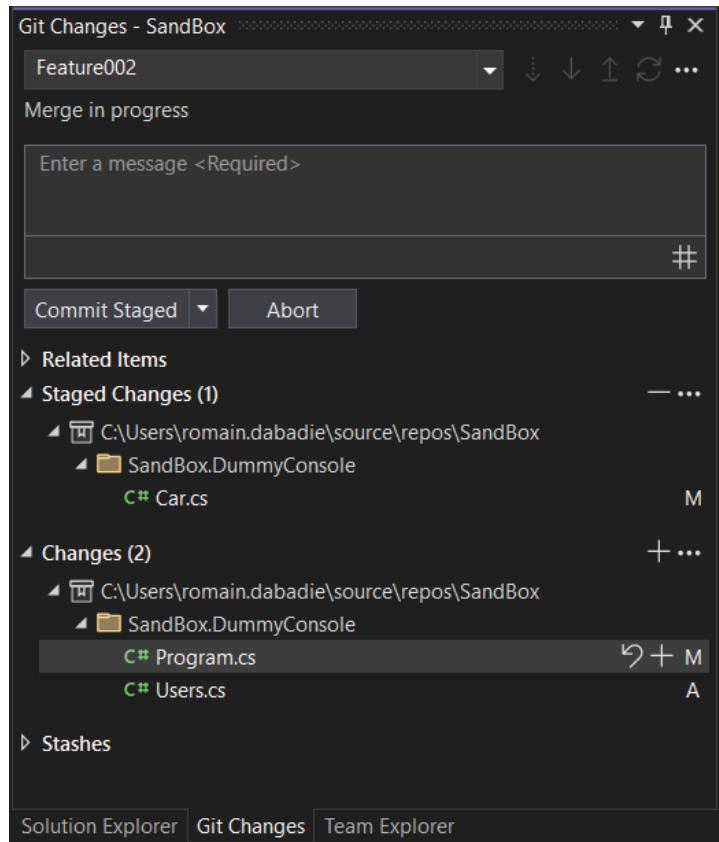


Figure 9.15 – Staged Changes section

You might notice that now the **Git Changes** window will display a **Commit Staged** button instead of the **Commit All** one. Under the hood, Visual Studio will execute the following `git` command to stage our specified file:

```
git add Car.cs
```

If we go back to our initial example, both features can be in the same file. For instance, we finish implementing the `UpdateModel()` function and leave `DeleteCar()` in a standby state. Here, we will take advantage of the interactive staging feature, by selecting the change we want to stage line by line, following these steps:

1. In the **Git Changes** window, we double-click on the `Car.cs` file to open the **Diff** window. This will show us the difference between the remote repository and our local repository.

```

Diff - Car.cs
Car.cs (Index)
Car.cs (Working tree)

public class Car
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
    public CarModel Model { get; set; }
}

public void UpdateModel(CarModel carModel)
{
    Model = carModel;
}

public void DeleteCar()
{
    // TODO: Implement the logic
}

```

Figure 9.16 – Diff window

- Once you have identified the lines you want to stage, select them and click on the **+ Stage Lines** pop-up button to add them:

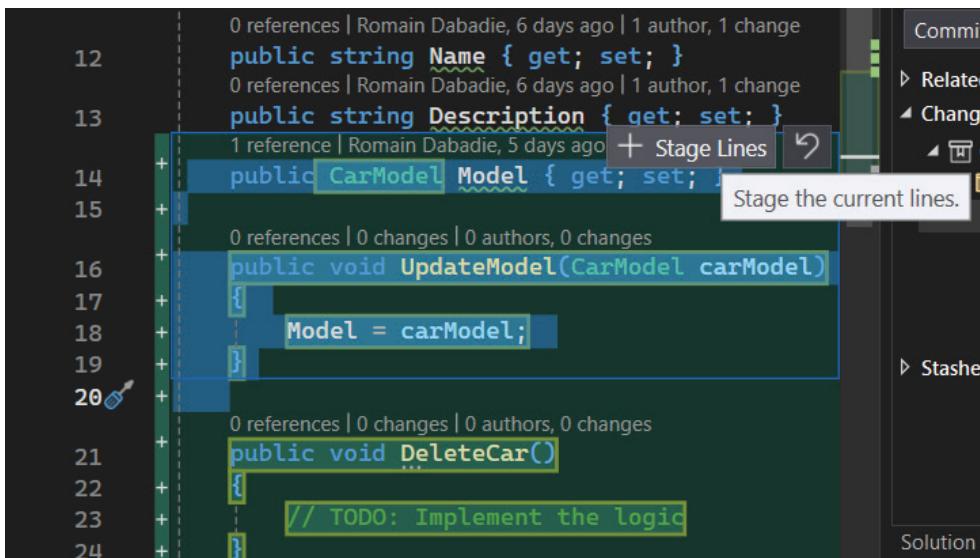


Figure 9.17 – Stage Lines

3. Now, in the **Git Changes** window, we can find two versions of our `Car.cs` file: one with the lines we picked to stage in the **Staged Changes** section and another with our complete pending changes in the **Changes** section.

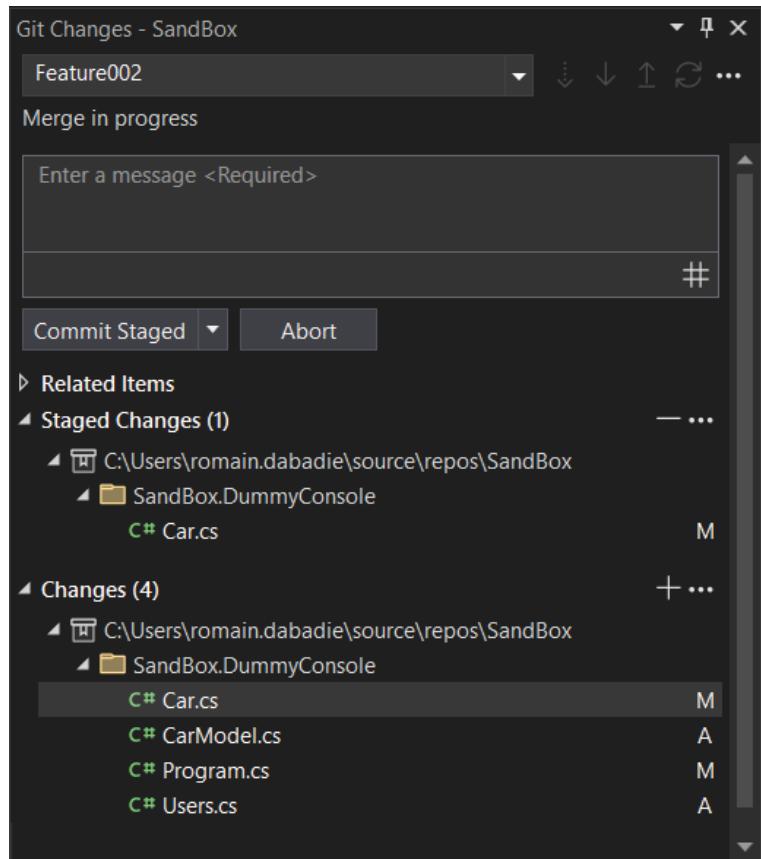


Figure 9.18 – Staged by line file

After staging our changes, we can commit them using the standard commit workflow with the **Commit Staged** button. This ensures that our commits are clean and logically organized, reflecting the changes we intended to share.

In this section, we explored how we can properly stage commits by selectively choosing specific lines of changes.

## Summary

This chapter delved into essential aspects of managing software development projects using Visual Studio, focusing on repository management, resolving conflicts, and leveraging the interactive staging feature. These lessons are crucial as they equip us with the tools and techniques necessary to maintain clean, efficient code bases and ensure smooth collaboration among team members.

We started by diving deep into the capabilities of Visual Studio for managing repositories, underscoring its critical role in simplifying version control processes. Following this, we delved into the art of resolving conflicts within Visual Studio, offering valuable insights into navigating the challenges of the merge conflicts that commonly surface during teamwork. Our journey concluded with an examination of the interactive staging feature, which provides a more user-friendly method for preparing commits.

As we move forward to the next chapter, we will build upon the foundational knowledge acquired in this chapter. The upcoming discussion will focus on automating the integration process, further enhancing project efficiency and reliability. This transition marks the next logical step in our journey toward mastering modern software development practices through Visual Studio 2022, where continuous integration plays a pivotal role in delivering high-quality software consistently.



# 10

## Continuous Integration with GitHub Actions

Welcome to the chapter on **Continuous Integration (CI)** with GitHub Actions. As we delve into this chapter, we will gain a comprehensive understanding of how GitHub Actions can enhance our development workflow, streamline our processes, and ensure our code base remains robust and reliable, as well as how Visual Studio helps us with this.

First, we will explore the fundamentals of GitHub Actions, a powerful CI/CD tool integrated directly into GitHub. Next, we will dive into configuring workflows in GitHub Actions. This section will guide you through setting up and managing workflows tailored to your project's needs. Finally, we will focus on generating GitHub Actions files using Visual Studio. Visual Studio provides robust support for creating and managing GitHub Actions for Azure deployment, making it easier for us to integrate CI/CD into your development environment. We'll cover the following topics:

- Understanding GitHub Actions for CI/CD
- Configuring workflows in GitHub Actions
- Generating GitHub Actions files with Visual Studio

Mastering GitHub Actions for CI/CD is crucial for any modern developer. By the end of this chapter, you will have the knowledge and skills required to set up and manage automated workflows, significantly improving your development efficiency.

### Technical requirements

While writing this chapter, I used the following version of Visual Studio:

- Visual Studio Enterprise 2022 Version 17.12.0
- Preview 1.0

To fully follow the chapter, you will also need a valid GitHub account, which you can set up at <https://github.com/>.

## Understanding GitHub Actions for CI/CD

In this section, we will dive into GitHub Action to recognize its role in automating software workflows directly within GitHub repositories and facilitating CI and **continuous deployment (CD)** processes. CI/CD practices aim to accelerate development cycles, enhance code quality, and streamline application deployment processes.

GitHub Actions enables the direct automation of software workflows within GitHub repositories, encompassing CI/CD processes. By automating these workflows, development cycles can be expedited, code quality can be ensured, and application deployment processes can be streamlined. An example of this automation could be running tests on code automatically upon pushing changes to the repository.

In the context of CI/CD, CI involves regularly merging all developers' working copies to a shared mainline several times a day. This helps to detect and address bugs quickly. By integrating code frequently, teams can identify and fix issues early in the development cycle, reducing the cost and effort required to resolve them later. For instance, we can set up a GitHub Action to run unit tests every time a pull request is merged into the main branch.

CD takes CI one step further by automating the deployment of code changes to selected infrastructure environments after they pass through the CI pipeline. CD ensures that new features and fixes are released quickly and reliably, improving the overall efficiency of the software release process. GitHub allows us to automatically deploy a web application to a staging environment whenever there's a successful build from the CI pipeline.

GitHub Actions offers several benefits for implementing CI/CD pipelines, including ease of setup, integration with GitHub's ecosystem, and the ability to customize workflows to fit specific project needs. These benefits make it accessible for teams of all sizes, from individuals working on personal projects to large organizations managing complex software products. A team can easily set up a CI/CD pipeline without needing extensive DevOps expertise, leading to faster development cycles and higher-quality releases.

GitHub Actions streamlines the setup of CI/CD pipelines, removing the need for manual configurations such as setting up webhooks, purchasing hardware, and managing security patches. Its seamless integration with GitHub allows it to respond to any webhook, enabling flexible event triggers for automation or CI/CD pipelines. The GitHub community contributes a vast array of pre-built CI/CD workflows through the GitHub Marketplace, making it easier for users to use existing solutions or share their own. Additionally, GitHub Actions supports any platform, language, and cloud, offering unparalleled flexibility for various technologies.

For a better understanding of the process, we will explore the different components of GitHub Actions:

- **Workflows:** A workflow is a configurable automated process that you can set up in your repository to build, test, package, release, or deploy any project on GitHub. Workflows are defined using **YAML** files stored in the `.github/workflows` directory of your repository. YAML originally stood for **Yet Another Markup Language**, but it later became the recursive acronym **YAML Ain't Markup Language**.
- **Events:** An event is something that happens in a GitHub repository, which can trigger a workflow. Examples include pushing code (`push`), opening a pull request (`pull_request`), and creating a new issue (`issue_comment`).
- **Jobs:** A job is a set of steps that execute on the same runner. Each job runs in its own fresh instance of the virtual environment specified by `runs-on`.
- **Actions:** Actions are reusable units of code that perform a specific task, such as building a Docker image, running tests, or deploying to a server. They can be written in JavaScript (using Node.js) or TypeScript and can be hosted in GitHub Marketplace or self-hosted.
- **Runners:** Runners are servers where the jobs live while they are being executed. There are two types of runners: GitHub-hosted runners and self-hosted runners.

After this overview of GitHub Actions, we will learn how to configure workflows.

## Configuring workflows in GitHub Actions

All components of GitHub can be configured through **YAML** file. In this section, I will explain how the GitHub Actions configuration file is organized to help you understand its functionality.

The **YAML** files are placed in the `.github/workflows` directory of our repository. These files, known as workflow files, define the scenarios of our CI/CD pipelines.

A typical GitHub Actions workflow configuration file consists of several key sections:

- `name`: A human-readable name for the workflow.
- `on`: Specifies the event(s) that trigger the workflow.
- `jobs`: Defines the jobs that make up the workflow. Each job runs on a runner environment specified by `runs-on`.
- `steps`: Within each job, steps are executed sequentially. Steps can run commands, set up tasks, or perform an action in your repository, a public repository, or an action published in a Docker registry.
- `env`: Allows you to set environment variables for all steps in a job.

- **defaults:** Sets default behaviors for all jobs and steps in the workflow.
- **permissions:** Controls the permissions granted to the GitHub Actions runner during the job execution.

Let's break down these key sections with a real-world CI workflow for a C# application.

Here's the content of a workflow file that we will name `CI.yaml`:

```
name: C# CI

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  build:
    runs-on: windows-latest

    steps:
      - uses: actions/checkout@v2
        with:
          fetch-depth: 0 # Fetches all history for all tags
                          # and branches

      - name: Setup .NET
        uses: actions/setup-dotnet@v1
        with:
          dotnet-version: '3.1.x' # Specify the .NET version
                                  # you need

      - name: Build
        run: dotnet build --configuration Release

      - name: Test
        run: dotnet test --no-build --verbosity normal
```

Now, let's understand how this workflow works and is articulated. There are four parts in this file, and they are organized as follows:

#### Part 1: the Workflow Metadata:

- `name: C# CI`

This is a descriptive name for the workflow, making it easier to identify in the GitHub UI.

## Part 2: the Trigger Conditions:

- `on:`
- `push:`
- `branches: [ main ]`

This means the workflow will run whenever there's a push to the main branch. Similarly, the following workflow will run when a pull request is opened, synchronized, or reopened targeting the main branch:

- `pull_request:`
- `branches: [ main ]`

## Part 3: the Jobs:

- `jobs:`
- `build:`
- `runs-on: windows-latest`

This specifies that the job should run on the latest Windows environment provided by GitHub Actions. This is important because .NET Core/.NET 5+ applications often require a Windows environment to build and run correctly.

## Part 4: the Steps:

Each job consists of a sequence of steps that are executed in order. Here's what each step does:

### 1. Checkout code:

- `uses: actions/checkout@v2`

This action checks out our repository under `$GITHUB_WORKSPACE`, allowing subsequent steps in the job to access it. The `fetch-depth: 0` option ensures that all history for all tags and branches is fetched, not just the default branch.

### 2. Set up .NET:

- `name: Setup .NET`
- `uses: actions/setup-dotnet@v1`

This action sets up the .NET environment. The `dotnet-version: '3.1.x'` input specifies which version of .NET to use. We can adjust this to match the requirements of our project.

3. Build:

- name: Build
- run: dotnet build --configuration Release

This step compiles the application using the .NET CLI. The `--configuration Release` flag indicates that the build should produce a `release` build, optimizing the output for performance.

4. Test:

- name: Test
- run: dotnet test --no-build --verbosity normal

This step runs any unit tests in the project without rebuilding the project first (`--no-build`). The `--verbosity normal` option controls the amount of logging output. This helps keep the log clean and focused on essential information.

This workflow provides a straightforward CI pipeline for a C# project, ensuring that every push or pull request to the main branch is built and tested automatically. By adjusting the .NET version and possibly adding more jobs or steps, we can tailor this workflow to fit the specific needs of our project.

After pushing our YAML file to GitHub, we might now go to the **Actions** tab in our GitHub repository to see our workflow running. When we push changes to the main branch or open a pull request, the workflow will automatically run.

## Generating GitHub Actions file with Visual Studio

In this section, we will see how Visual Studio allows us to generate our GitHub Actions file for our Azure deployment. Please note that you will need a valid subscription, as mentioned in the *Technical requirements* section, to complete this section.

First, we need an application to deploy to Azure, and for that, I simply initiated a Blazor project, with the template provided by Visual Studio, which I named `BlazorServerApp`. After that, follow these steps:

1. Right-click on the top node of our project in order to launch the publish wizard and select **Azure**:

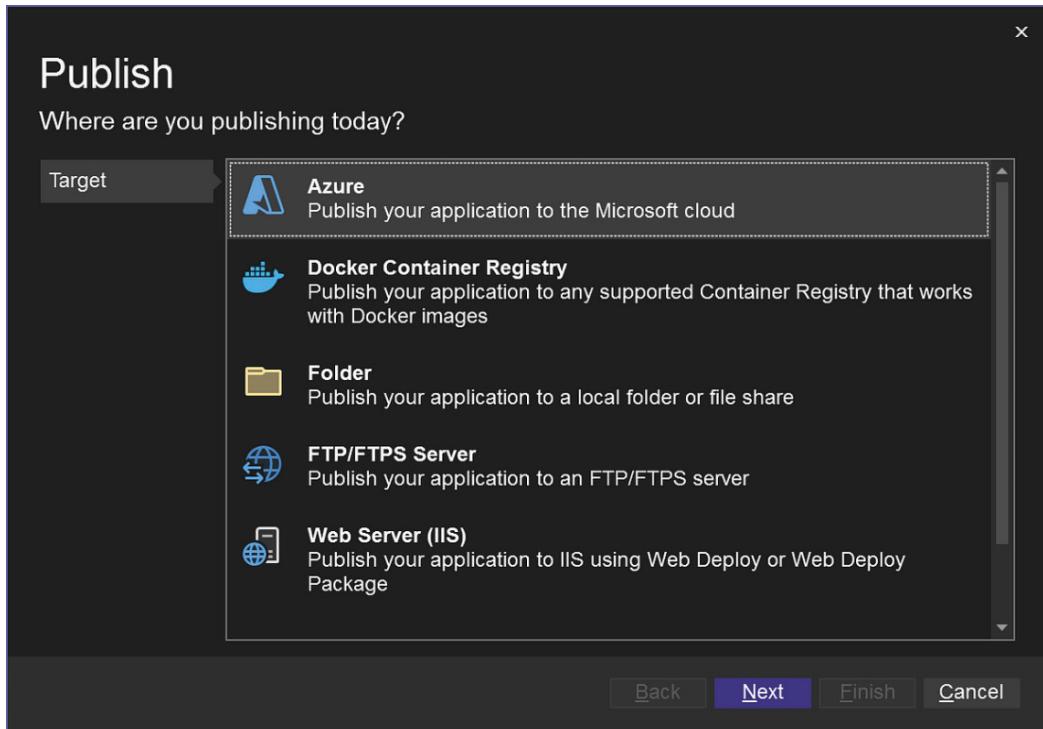


Figure 10.1 – Publish Azure

2. Then, select the type of target you want. It is up to you to choose the target that will suit your budget and company politics. For this example, we will choose **Azure App Service (Linux)**:

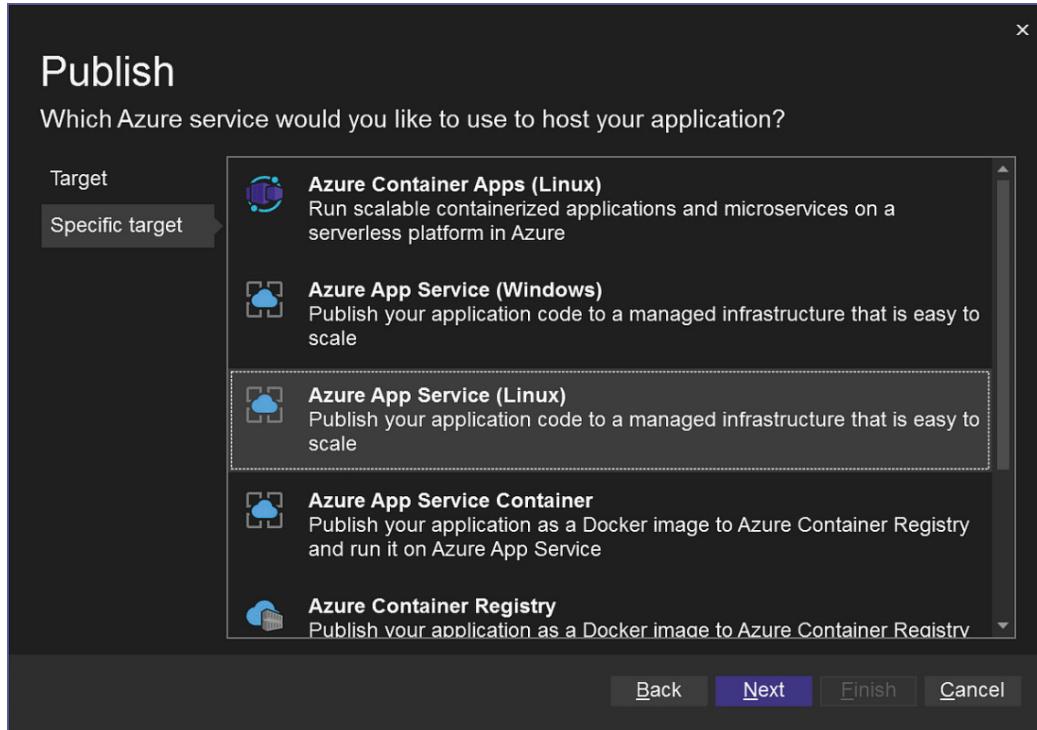


Figure 10.2 – Specific target

- Once we choose the target, we will jump to the selection of the new **App Service**. Here, we have to select an existing instance in our subscription.

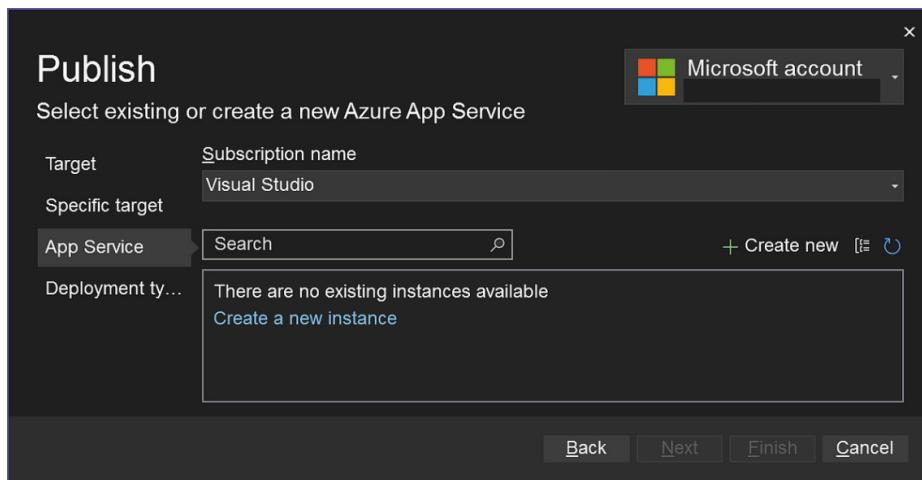


Figure 10.3 – App Service subscription

If none are available, you might need to create a new one by clicking on the **Create a new instance** link and following the instructions in the wizard.

Finally, in the last step, we will determine the **Deployment type**.

4. Choose **CI/CD using GitHub Actions workflows (generates yml file)**, which will generate the appropriate YAML file according to our configuration:

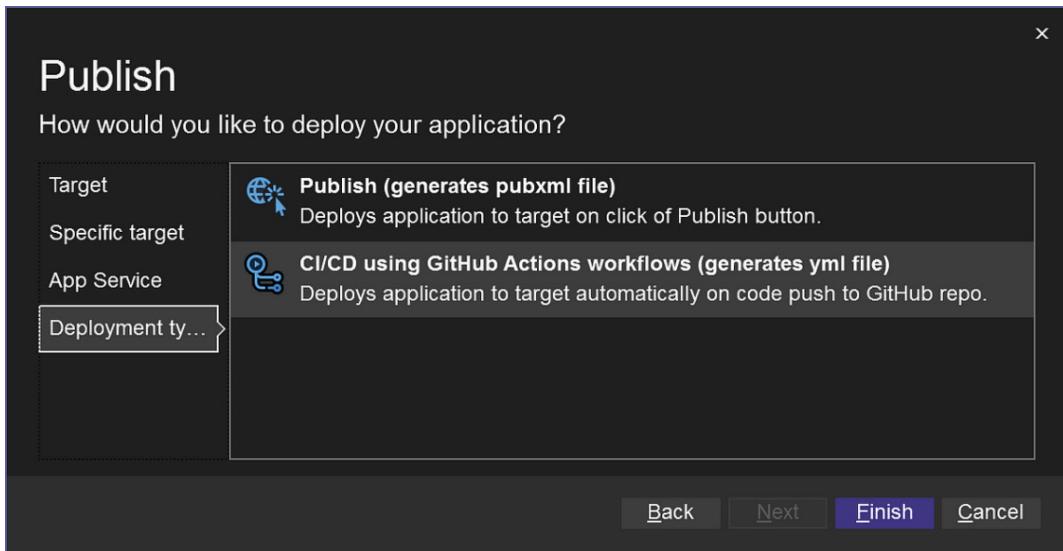


Figure 10.4 – Deployment type

5. Now in the **Solution Explorer**, we can see the Blazor Server project and also the generated `BlazorServerApp.yml` file under a **GitHub Actions** node.

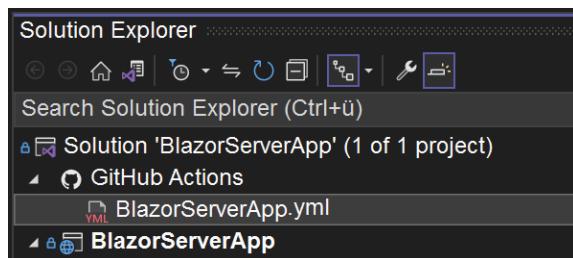


Figure 10.5 – GitHub Actions node

We have now seen how to generate a GitHub Actions file. We were able to customize it to add code quality to our pipeline, using external tools, for example SonarCloud, a widely used static analysis solution, among others. GitHub Actions is a powerful asset to explore, and I advise you to read the book *Mastering GitHub Actions* by Eric Chapman (<https://www.packtpub.com/en-us/product/mastering-github-actions-9781805128625>) to dig deeper into the subject.

## Summary

In this chapter, we explored the power of GitHub Actions for CI and CD, learning how to automate and streamline our development workflows. We began by understanding the fundamental concepts of GitHub Actions. Next, we moved on to configuring workflows in GitHub Actions. We then learned how to write and manage YAML files that define our CI/CD pipelines. In the final section, we focused on generating GitHub Actions files using Visual Studio. Visual Studio's robust support for GitHub Actions for Azure's deployment simplifies the process of integrating CI/CD into your development environment.

Having established a strong foundation in automating CI/CD with GitHub Actions, it's time to expand our collaborative capabilities. In the next chapter, we will explore how Azure DevOps can enhance team collaboration, streamline project management, and further optimize your development processes.

# 11

## Collaborative Development with Azure DevOps

In the fast-paced world of software development, collaboration and efficiency are paramount. Azure DevOps provides a comprehensive suite of tools designed to streamline and enhance team-based development efforts. This chapter will guide you through the essential aspects of leveraging Azure DevOps for collaborative development within Visual Studio.

The first section will introduce Azure DevOps, which will offer a foundational understanding of its core services and features. Next, we'll walk through the process of setting up team projects to establish a connection between Visual Studio and Azure DevOps. Then, we'll explore the principles of Agile development and how they can be integrated within Azure DevOps. We will leverage work items to increase collaboration, traceability, and productivity. Finally, in the last section, we will explore how to manage builds within Visual Studio.

In this chapter, we're going to cover the following main topics:

- Introduction to Azure DevOps
- Setting up team projects
- Implementing Agile development practices
- Integrating Azure Pipelines for continuous integration

By the end of this chapter, we will be equipped to effectively utilize Azure DevOps in our development workflow within Visual Studio.

## Technical requirements

While writing this chapter, I used the following versions of Visual Studio:

- Visual Studio Enterprise 2022 version 17.12.0
- Preview 1.0

To fully follow the chapter, you will also need a valid Azure DevOps account.

## Introduction to Azure DevOps

As experienced developers working within the .NET ecosystem, we've always sought ways to enhance our productivity, streamline our workflows, and ensure the quality of our software projects. In this chapter, we delve into the world of Azure DevOps, a suite of development tools, services, and features that empower teams to plan work, collaborate on code development, and build and deploy applications efficiently.

Azure DevOps stands out as a comprehensive platform that integrates project management, version control, reporting, and automated builds into a single service. It supports both cloud-based and on-premises deployment models, making it adaptable to various organizational needs. By leveraging Azure DevOps, we can improve our team's efficiency, enhance collaboration among team members, and accelerate the delivery of high-quality software solutions.

Here is an overview of the core components that we can leverage to transform our development practices with the help of Azure DevOps:

- **Azure Boards** for orchestrating work, planning sprints, and managing backlogs with unparalleled clarity and precision
- **Azure Repos** for harnessing the power of Git repositories, ensuring seamless version control and collaborative coding experiences
- **Azure Pipelines** for automating the build and deployment processes, facilitating continuous integration, and ensuring that our applications are always ready for release
- **Azure Test Plans** for streamlining the testing phase, allowing us to create, execute, and track tests efficiently
- **Azure Artifacts** for sharing packages and dependencies, fostering a cohesive and efficient development ecosystem

Now we have familiarized ourselves with the Azure DevOps context, let's dive into how to connect our Azure DevOps project within Visual Studio.

## Setting up team projects

To work effectively with Azure DevOps projects from within Visual Studio, we need to establish a connection between the two. This involves several steps, including setting up our environment, configuring project settings, and ensuring that both Visual Studio and Azure DevOps are properly integrated. In this section, we will navigate through this process.

Before starting, ensure you have the following:

- An active Azure subscription
- Access to an Azure DevOps organization and at least one project within it

Visual Studio allows us to connect to Azure DevOps projects through the **Team Explorer** view. This approach allows us to access and manage our Azure DevOps projects directly from within Visual Studio without needing to open a separate Azure DevOps window. To do so, follow these steps:

1. Navigate to **View | Team Explorer**:

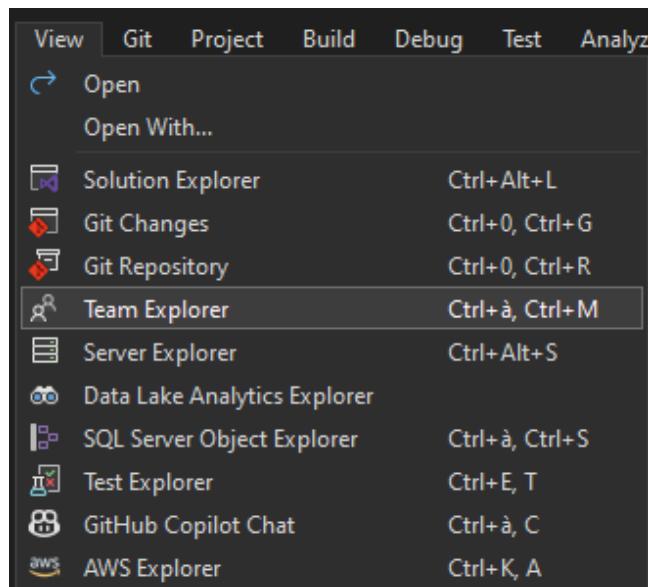


Figure 11.1 – Opening Team Explorer

2. With **Team Explorer** open, click on the *Manage Connection* button located at the top of the **Team Explorer** window, visualized as an *electric plug*:

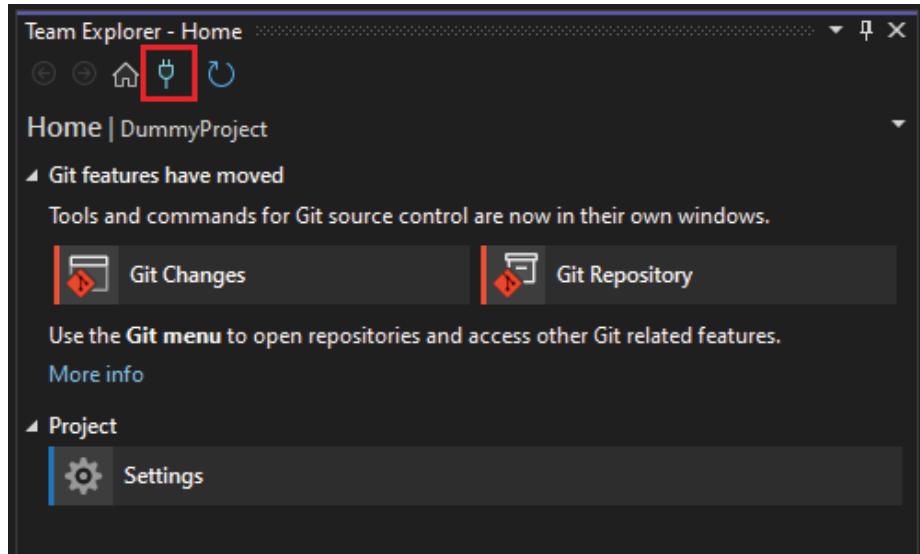


Figure 11.2 – The Manage Connection button

This will lead us to sign in to our Azure DevOps account. We can retrieve our active AzureDevOps connection here, if you haven't set one yet.

3. Next, click on the **Manage Connections** link and then on **Connect to a Project...**:

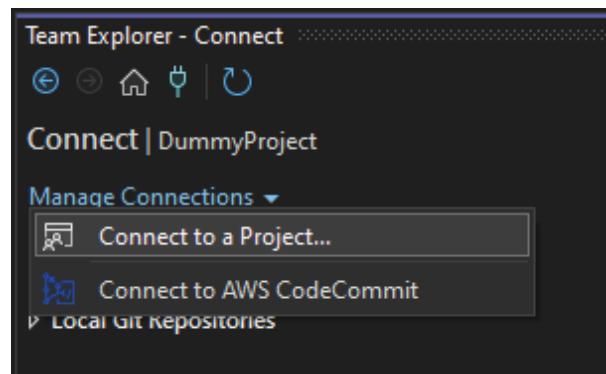


Figure 11.3 – Connect to a Project...

After signing in, we'll be presented with a list of Azure DevOps organizations and projects that we have access to:

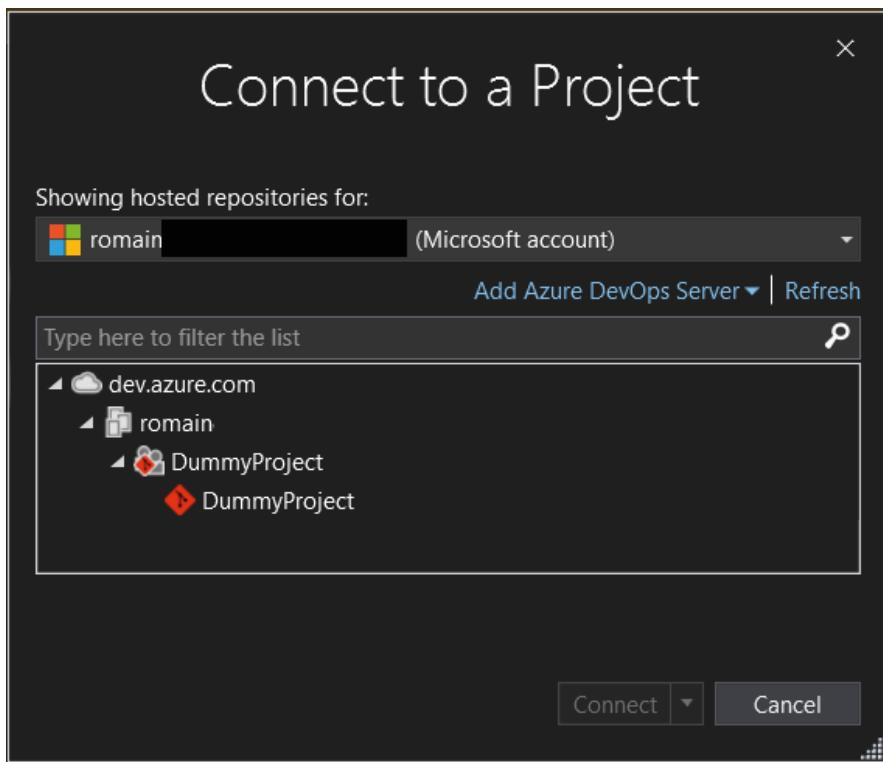


Figure 11.4 – Connect to a Project

4. Select the project you wish to connect to and click **Connect**. This action establishes a connection between Visual Studio and your selected Azure DevOps project.

Once connected, let's see now how we can leverage this connection, managing our Azure DevOps project directly from the **Team Explorer** window.

## Implementing Agile development practices

Now that we have connected Visual Studio with the Azure DevOps project, we can start managing our Azure DevOps project directly from the **Team Explorer** window. In this section, we will have a refresher on what Agile development is and how to leverage Visual Studio to streamline our Agile process.

## Introducing Agile development

Agile development is a flexible approach to managing projects and developing software that emphasizes collaboration, customer feedback, and adapting to change. It was established through the Agile Manifesto in 2001 and focuses on delivering value to customers quickly and continuously improving processes and products.

Key aspects of Agile development include the following:

- **Prioritizing people over processes:** Agile values human interaction and collaboration over rigid processes and tools, recognizing that effective communication is crucial for project success
- **Functional software over extensive documentation:** Agile teams focus on delivering working software frequently, believing that functional products provide more value than comprehensive documentation
- **Collaboration with customers:** Agile development encourages close collaboration with customers throughout the project lifecycle to better understand and meet their needs
- **Adaptability:** Agile methodologies embrace change, even late in the development process, understanding that adapting to new requirements helps deliver a product that truly meets customer needs

Agile practices involve breaking down projects into smaller, manageable units called sprints, typically lasting one to four weeks. Teams work collaboratively during these sprints to deliver functional increments of the product, allowing for regular feedback and adjustments.

In that agile context, Work Items are utilized to organize, track, and manage the various elements of work involved in a project. These elements can range from user stories and tasks to bugs and features. Work Items serve as the building blocks for planning and tracking work in Agile projects, allowing teams to capture details about the work, assign responsibilities, set priorities, and track progress.

In Agile methodologies, Work Items are typically categorized into types such as User Stories, Bugs, Tasks, Features, and Epics, each serving a specific purpose:

- User Stories represent the functionality that needs to be developed from the perspective of the end user. They help teams understand what users need and why.
- Bugs are used to track defects or issues found in the software during testing or development phases.
- Tasks are smaller units of work that need to be completed to fulfill a User Story or to address a Bug.
- Features represent larger pieces of functionality that may encompass several User Stories. They help in grouping related work together.
- Epics are even broader than Features, representing large bodies of work that can span across multiple sprints or releases.

Work Items are essential for Agile project management because they facilitate communication among team members, help in prioritizing work, and enable tracking of progress toward project goals. Teams can link Work Items to each other to show dependencies, roll up work under larger initiatives, and generate reports for better project oversight.

Agile teams often use tools such as Azure DevOps to manage Work Items effectively. These tools provide templates for different Work Item types, support customization to fit team processes, and offer features for tracking dependencies, managing workloads, and visualizing progress through dashboards and reports.

In summary, Work Items are a fundamental aspect of Agile practices, providing a structured way to plan, track, and manage work across various stages of the project lifecycle.

## Managing Work Items through Visual Studio

Back on the **Home** page of **Team Explorer**, we might notice three tiles in the **Project** section with one named **Work Items**. In this section, we will explore how we can interact with the Work Items through Visual Studio.

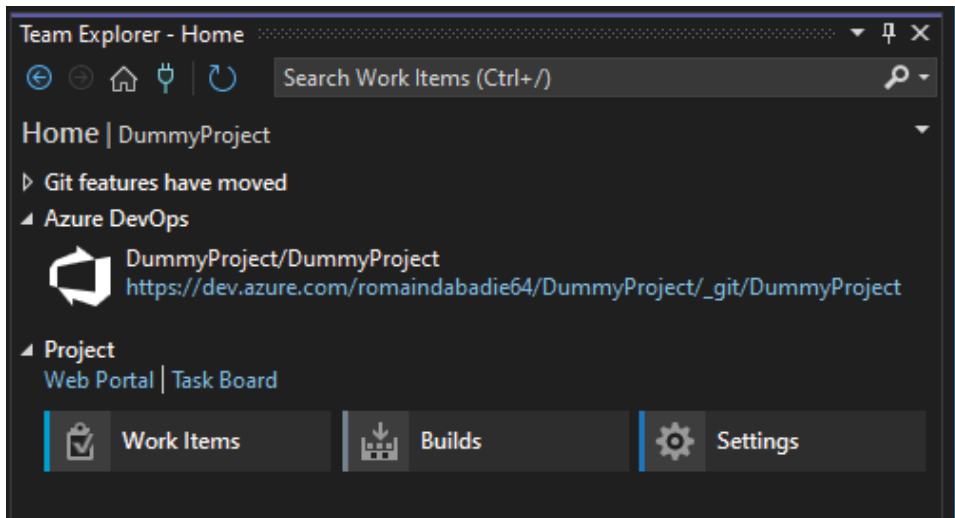


Figure 11.5 – Team Explorer Home

By clicking on the tile, we access the list of Work Items for our project:

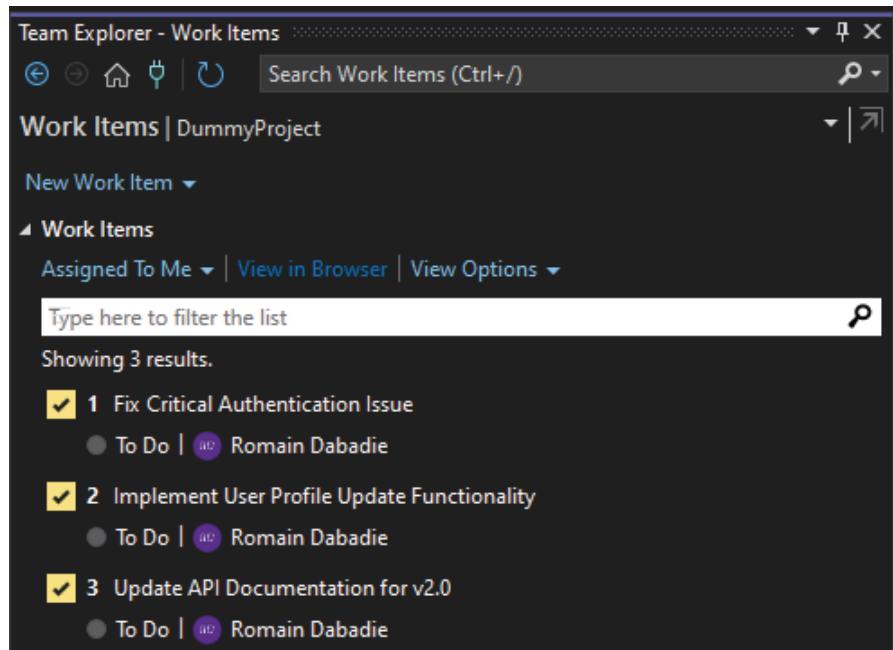


Figure 11.6 – Visualizing Work Items

In this window, we can list all our Work Items. The two drop-down options, **Assigned To Me** and **View Options**, along with the textbox allow us to search and filter the list. Additionally, the **View in Browser** link allows us direct access to the Azure dashboard via the browser.

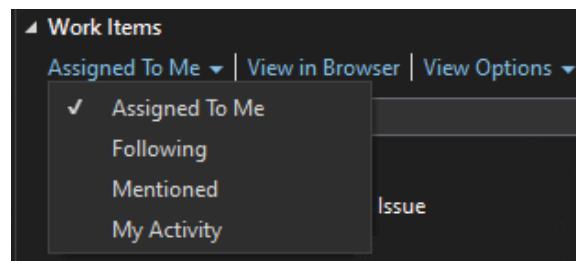


Figure 11.7 – Filter Work Items

By clicking the first link under the **Work Items** title, we get access to the following four options:

- **Assigned To Me**: This filter shows Work Items that are specifically assigned to the currently logged-in user. It helps us focus on our responsibilities and tasks without getting overwhelmed by the entire project's workload.

- **Following:** This filter displays Work Items that we have chosen to follow. This could include tasks assigned to others that we have a vested interest in or want to keep track of for collaboration or oversight purposes.
- **Mentioned:** This option filters Work Items where we have been mentioned. Mentions typically occur in comments or descriptions and serve as a way to draw someone's attention to a particular item.
- **My Activity:** This filter shows Work Items that we have interacted with in some way. This could include creating, editing, commenting on, or otherwise engaging with Work Items.

These filtering options in **Team Explorer** enhance productivity by allowing users to tailor their view of Work Items according to their roles, interests, and responsibilities within the project.

Additionally, the **View Options** link offers us options to manage the display of the Work Items.

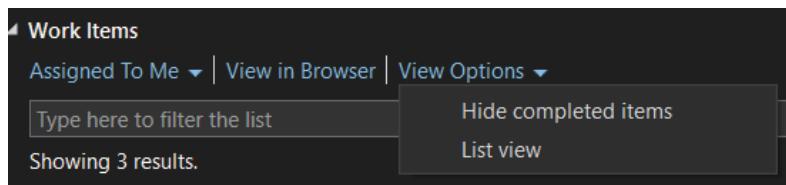


Figure 11.8 – View options

Here, we can select **Hide completed items** and switch between a **List view** display or **Detail view**.

Now we have seen how to manage the list of our Work Items, let's explore the way we can interact with these Work Items. We open the **Work Items** contextual menu by right-clicking on one of them.

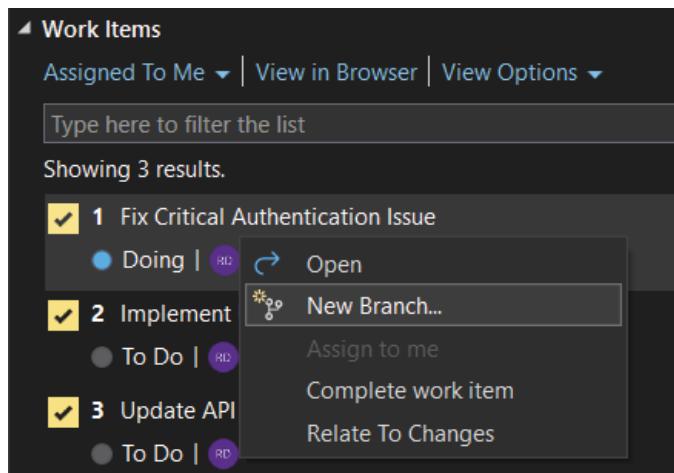


Figure 11.9 – Work Items contextual menu

This menu contains the following actions:

- **Open:** This action opens the selected work item in a detailed view directly in the Azure portal.
- **New Branch...:** This creates a new branch in our version control system specifically for the selected work item. This helps in isolating changes related to the work item, making it easier to track and review those changes later.

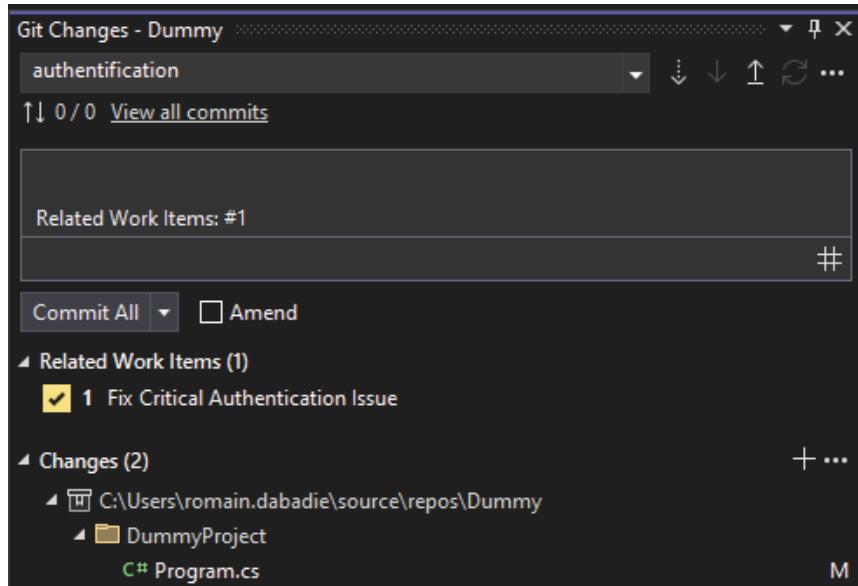


Figure 11.10 – Related Work items

This way, each commit in this branch will directly contain the tag to mention the related Work Items ID in the commit message.

- **Assign to me:** This action assigns the selected work item to the currently logged-in user. It's a way of taking ownership of a task, indicating that we will be responsible for completing it.
- **Complete work item:** This marks the selected work item as completed. This usually changes the state of the work item to indicate that all required work has been done and it's ready for review or closure.
- **Relate To Changes:** This links the selected work item to specific changesets, commits, or branches in the version control system. This creates traceability between the work being done (code changes) and the reason for those changes (the work item). This way, even though the branch was not created for this work item, we can easily link it to a commit.

If we have to create a new work item, we can quickly do it directly by using the **New Work Item** link at the top of the **Work Items** window:

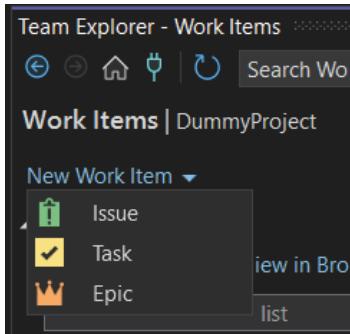


Figure 11.11 – Creating new Work Items

This will prompt us to choose between three types of Work Items (**Issue**, **Task**, or **Epic**), and then a simple form appears to set the title. Once created, the work item will appear under the **My Activity** filter.

Managing the Work Items directly in Visual Studio allows us to increase our productivity. Proper use of this feature enhances collaboration, traceability, and productivity within our development teams.

As we have seen with GitHub Actions in the previous chapter, continuous integration is a crucial part of modern application development. In the next section, we will see how to interact with Azure Pipelines within Visual Studio and manage our builds.

## Integrating Azure Pipelines for continuous integration

The **Builds** feature in the **Team Explorer** window of Visual Studio is designed to help developers manage, monitor, and interact with build processes within a team project. It provides a centralized interface for viewing build definitions, queuing new builds, monitoring ongoing builds, and reviewing completed builds. This feature supports both manual and automated build processes, making it easier for teams to ensure the quality and reliability of their software products.

We can access this interface by clicking on the **Builds** tile on the **Home** page of **Teams Explorer**:

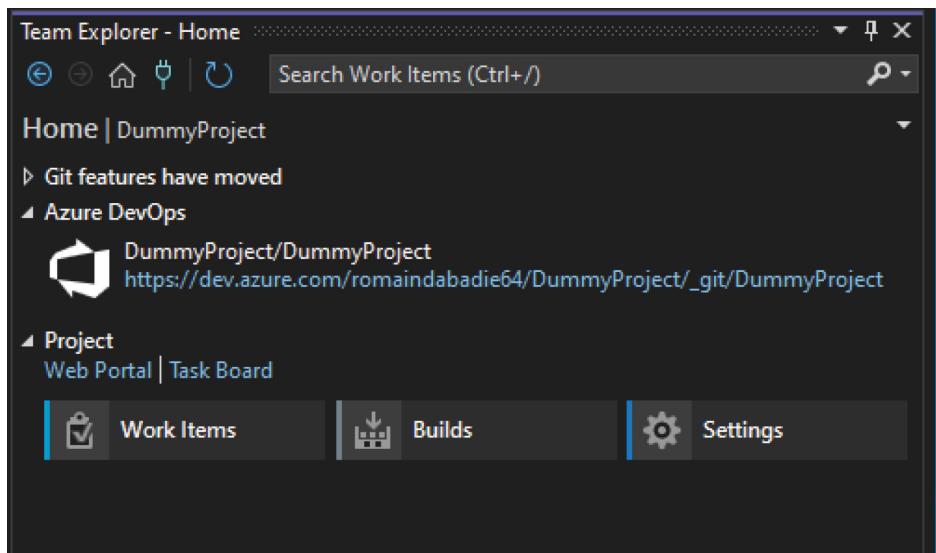


Figure 11.12 – The Builds tile in Team Explorer

In the **Builds** section of **Team Explorer**, we can see four sections that help us interact with the Azure builds.

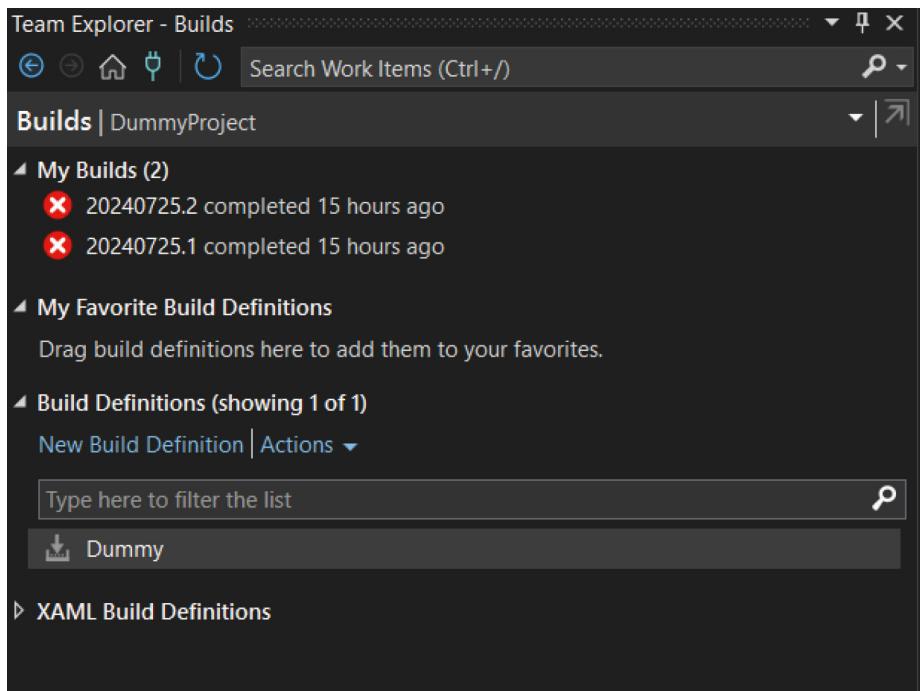


Figure 11.13 – Team Explorer Builds

The cornerstone of the feature is **Build Definitions**. These define how your application should be built, including what triggers a build (e.g., continuous integration), which source files to include, and any pre- or post-build tasks. We can find all the build definitions for our project under **Build Definitions**. By clicking one of the builds, we get access to a contextual menu.

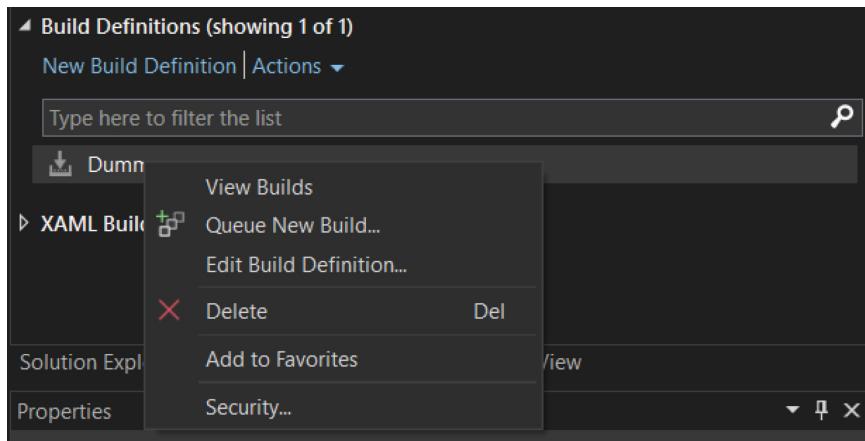


Figure 11.14 – Build definition contextual menu

The contextual menu of **Build Definitions** offers several actions, such as the following:

- **View Builds:** This opens builds in the Azure Devops interface through the browser
- **Edit Build Definition....:** This opens the build definition in the Azure Devops interface in the browser in edit mode, in order to edit it
- **Queue New Build....:** We can manually queue builds from the **Team Explorer** window, specifying parameters such as the build definition to use and whether to run the build immediately or schedule it for later
- **Add to Favorites:** Adding build definitions to favorites streamlines access to frequently used builds, enhancing productivity by reducing the time spent navigating through various build definitions

Additionally, the **Builds** feature allows users to view the status of current builds, including progress, success, or failure messages. This real-time feedback helps us quickly address issues. After a build is completed, we can review logs, test results, and other artifacts generated during the build process. This aids in troubleshooting and ensures that all components meet quality standards.

As a result, integrating the **Builds** feature in **Team Explorer** with Azure DevOps provides a powerful platform for managing end-to-end CI/CD pipelines directly from within Visual Studio. This integration enhances productivity, ensures consistency across environments, and supports scalable, secure software delivery processes.

## Summary

In this chapter, we've explored the essentials of collaborative development with Azure DevOps, equipping you with the knowledge and skills needed to streamline your development workflow within Visual Studio. We began with an introduction to Azure DevOps, highlighting its core services and demonstrating how it integrates with Visual Studio to enhance team collaboration and project management.

Next, we delved into implementing agile development practices, starting with a comprehensive introduction to Agile methodologies. We learned how to manage Work Items directly through Visual Studio, ensuring efficient tracking and resolution of tasks, bugs, and user stories.

We then learned how to integrate Azure Pipelines for continuous integration, guiding you through the management of builds within Visual Studio.

As we transition to the next chapter, we'll continue to expand our skill set. Docker containers have revolutionized the way we develop, test, and deploy applications, providing a consistent environment across different stages of development. In the upcoming chapter, you'll learn how to leverage Visual Studio's powerful container tools to work with Docker.

# 12

## Visual Studio

# Container Tools for Docker

In this chapter, we will delve into the powerful integration of Docker with Visual Studio, a synergy that simplifies container management and accelerates your development workflow. We will begin by covering the basics of Docker and how it integrates with Visual Studio. We'll understand the fundamental concepts of containerization and why it has become a cornerstone in modern development practices. Next, we will walk through the steps to configure our development environment for Docker. This includes installing necessary components, setting up Docker support in Visual Studio, and configuring your projects to use Docker. Then, we will dive into the practical aspects of converting your applications into Docker containers. Finally, we will explore the deployment phase. We'll learn various methods to deploy our containerized applications, whether to a local Docker host, a remote server, or cloud platforms.

In this chapter, we're going to cover the following main topics:

- Introduction to Docker and Visual Studio integration
- Setting up Docker environments in Visual Studio
- Dockerizing applications with Visual Studio
- Deploying containerized applications

By the end of this chapter, you will have the knowledge and skills to harness Docker within Visual Studio, making your development process more efficient and scalable, without getting out of our favorite IDE.

## Technical requirements

While writing this chapter, I used the following version of Visual Studio:

- Visual Studio Enterprise 2022 Version 17.12.0
- Preview 1.0

To fully follow the chapter, you will also need to install Docker Desktop.

## Introduction to Docker and Visual Studio integration

The integration of containerization technologies such as Docker with powerful IDEs such as Visual Studio has become essential for modern application development. This section introduces Docker and its seamless integration within Visual Studio, showcasing how this combination enhances the development workflow.

Docker, an open source platform, automates the deployment, scaling, and management of applications. Introduced in 2013, Docker utilizes containerization technology to package an application along with its environment and dependencies into a standardized unit called a **container**.

**Containers** are lightweight, standalone, and executable packages that include everything needed to run a piece of software, including the code, runtime, system tools, libraries, and settings. Sharing the host **operating system's (OS's)** kernel, containers isolate application processes from the rest of the system, ensuring consistent performance across various environments. This abstraction optimizes system resource usage and simplifies deployment and scalability.

Docker operates on a client-server model with three main components: the Docker client, the Docker host, and Docker Registry:

- **The Docker client:** The primary user interface to Docker, accessible via **command-line interface (CLI)** commands. Users can build images, download images from a registry, run containers, and manage container networks and volumes. The Docker client communicates with the Docker daemon, which handles the heavy lifting of building, running, and managing containers.
- **The Docker host:** The runtime environment for containers, comprising several key components:
  - **Docker daemon:** This runs in the background on the host machine, managing the building, running, and distribution of Docker containers. It listens to API requests from the Docker client and manages Docker objects, such as images, containers, networks, and volumes.
  - **Containers:** Runnable instances of Docker images, encapsulating an application along with its environment, libraries, and dependencies.
  - **Images:** Read-only templates used to create containers, containing the application and all its dependencies.

- **Networks:** Custom networks configured for container communication.
- **Storage:** Managed through volumes that persist data generated and used by Docker containers.
- **Docker Registry:** Stores Docker images in a central repository, facilitating the sharing and deployment of images across different hosts. Registries can be public, such as Docker Hub, or private, for secure storage and management of organizational images.

Another key component of Docker is **Docker images**. Docker images are snapshots of a container's configuration at a specific point in time, containing the application code, runtime, libraries, tools, and dependencies. They are immutable and serve as the basis for creating containers. Modifications to an image create new images, enabling consistent replication of application environments.

Docker uses Dockerfile to automate Docker image creation. **Dockerfile** is a script with instructions defining the environment in which an application will run. Common instructions include the following:

- **FROM:** This specifies the base image (e.g., `FROM mcr.microsoft.com/dotnet/sdk:9.0` uses the .NET SDK image tagged with version 9.0)
- **RUN:** This executes commands in a new layer on top of the current image, often used for installing software packages (e.g., `RUN npm install`)
- **CMD:** This provides defaults for executing a Docker container (e.g., `CMD ["npm", "start"]`)

Dockerfiles streamline the setup and configuration of containers, ensuring efficient definition and replication of application environments across development stages.

Visual Studio provides robust support for Docker, enabling developers to easily containerize their applications and manage container orchestration directly from the IDE. This integration simplifies the process of developing, testing, and deploying containerized applications, especially for projects targeting .NET Framework, .NET Core, ASP.NET, and ASP.NET Core.

Next, we will explore how to set up a Docker environment within Visual Studio.

## Setting up Docker environments in Visual Studio

In this section, we will explore how to leverage Docker support. Docker support in Visual Studio provides a seamless experience for developing applications that are intended to run inside Docker containers. Docker support can be added to a Visual Studio project during project creation or added to an existing project.

## Creating a project with Docker support

Assuming you have Docker Desktop installed and running, the first thing to do is to create a new project.

We will create an ASP.NET Core application as usual and in the **Additional information** window, we will check the **Enable container support** checkbox:

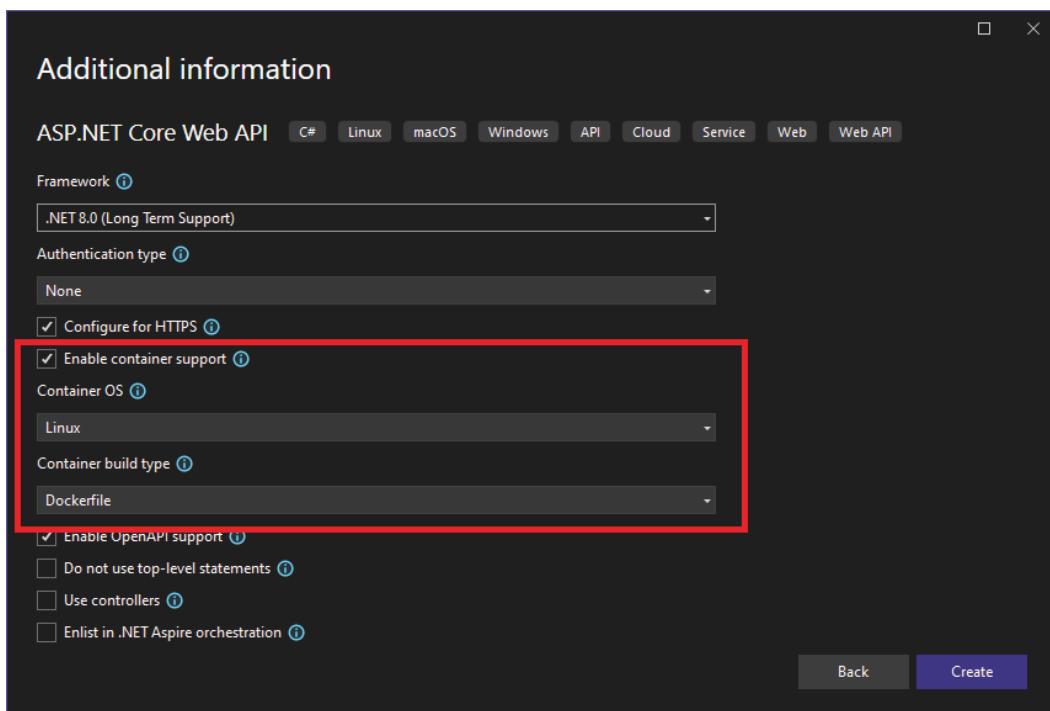


Figure 12.1 – Enable container support

Once checked, it enables the two text boxes, **Container OS** and **Container build type**, to configure the container type. We can select either **Windows** or **Linux** for the **Container OS** option based on our requirements. For **Container build type**, we can select either of the following:

- **Dockerfile:** When you choose **Dockerfile** in an ASP.NET Core application, you are opting to manually define how your container image should be built using a Dockerfile. Using a Dockerfile gives you full control over the contents of your container image, including the base image, dependencies, environment variables, exposed ports, and the specific commands to run your application.
- **.Net SDK:** Choosing **.Net SDK** simplifies the process by leveraging Visual Studio's built-in container support.

For the following example of this chapter, we will choose the **Dockerfile** option.

## Adding Docker support to an existing project

If you have an existing project and want to add Docker support to it, we follow these steps:

1. To add Docker support, right-click on the project in **Solution Explorer** and navigate to **Add | Docker Support...**:

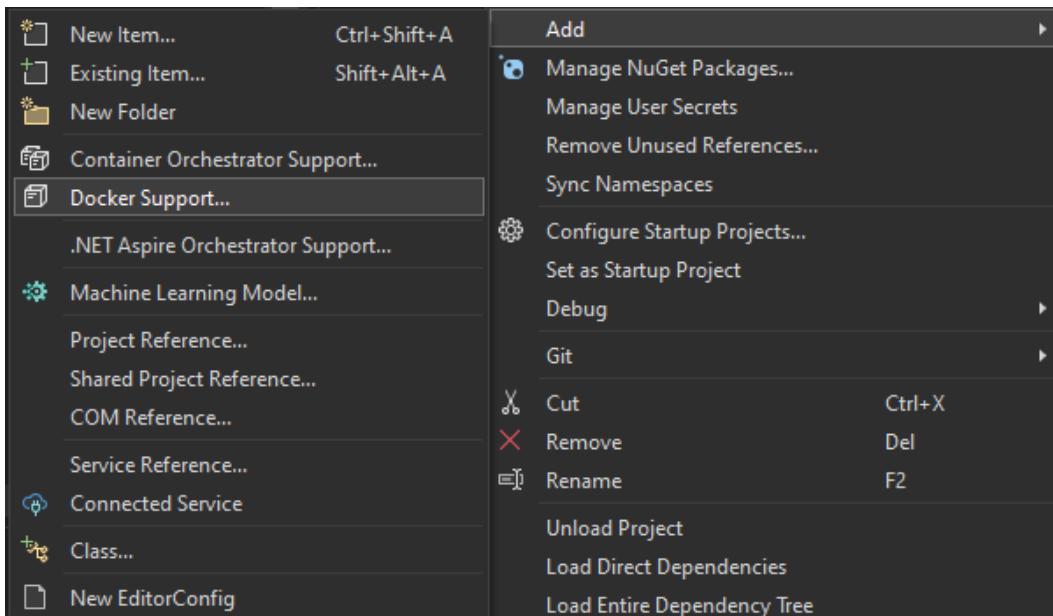


Figure 12.2 – Adding Docker support

Visual Studio will generate the necessary Docker files for our project.



Figure 12.3 – Dockerfile

2. The following dialog will appear where you will have to configure the Docker settings:

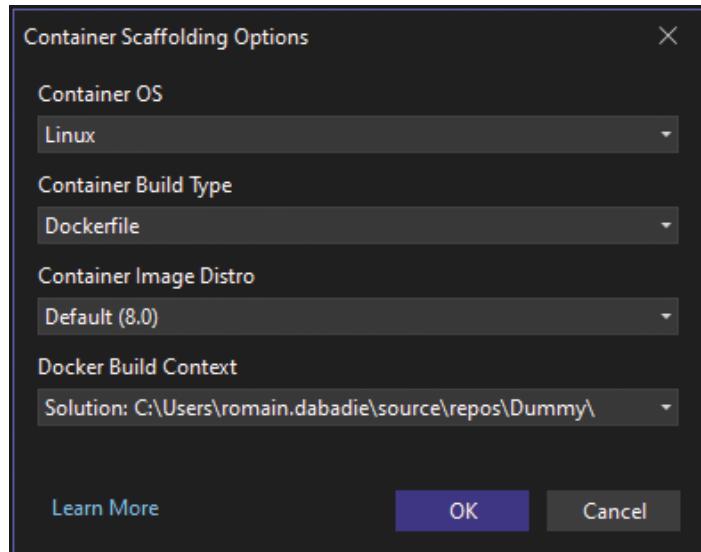


Figure 12.4 – Container Scaffolding Options

- **Container OS:** This setting allows us to choose the OS that will be used inside your Docker container. When we are adding Docker support when creating a new project, we have the choice between Linux and Windows.
  - **Container Build Type:** As previously discussed, this setting determines how your Docker image will be built (e.g. Dockerfile / .Net SDK).
  - **Container Image Distro:** This setting refers to the base image distribution for your container. It's particularly relevant when choosing a Linux-based container OS.
  - **Docker Build Context:** The Docker build context refers to the set of files located in the specified path or URL that Docker uses to build the image. Essentially, it defines the scope of files that Docker can access during the build process.
3. With Docker support added, we can now build and run our project within Docker containers. Visual Studio provides convenient buttons in the toolbar for building and running your Dockerized application.



Figure 12.5 – Container run/debug button

Now we have set Docker support up, let's learn about how it might improve our productivity.

## Dockerizing applications with Visual Studio

Now that we have set the Docker support up for our project, in this section, we will explore the Dockerizing of our application through the container window of Visual Studio. We can retrieve all the information of our container in the windows. To open this window, go to **View | Other Windows | Containers:**

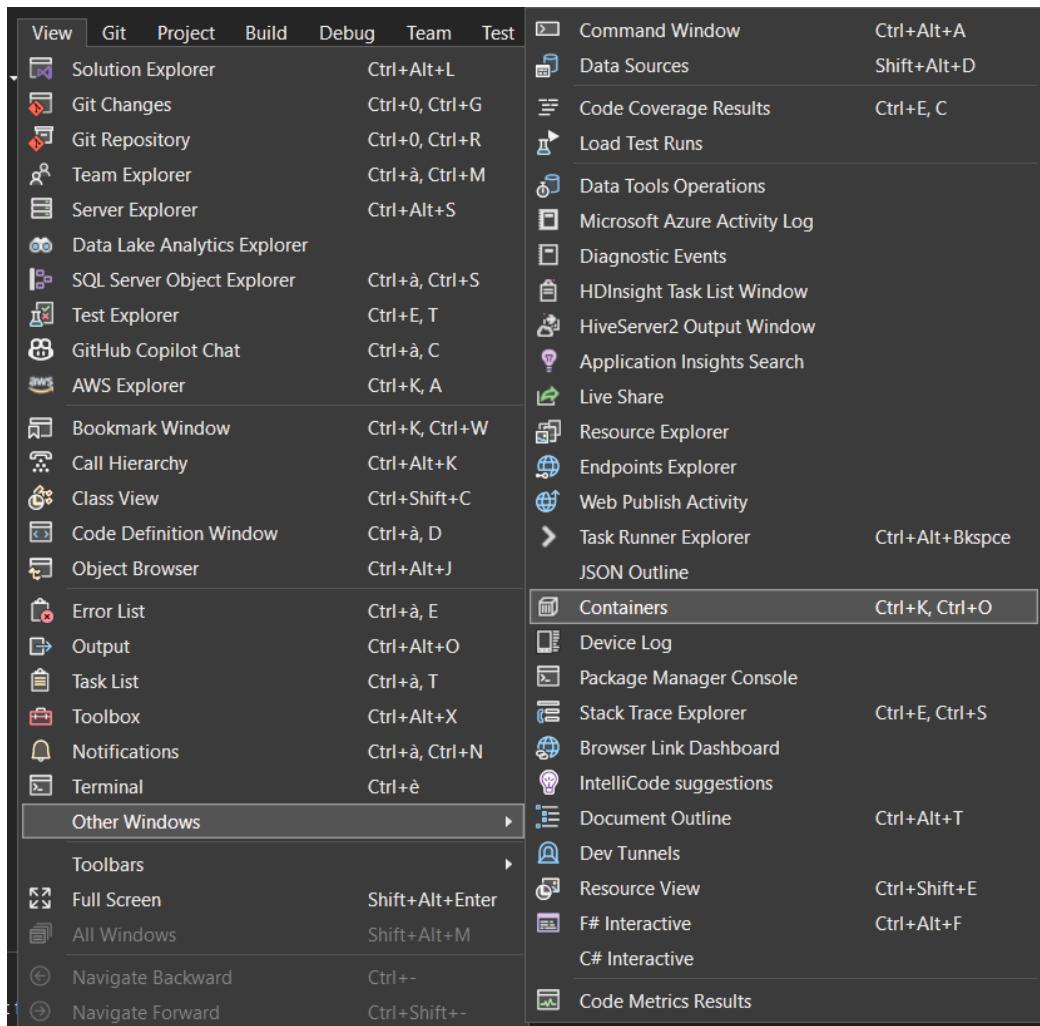


Figure 12.6 – Open containers windows

Here, we can find the information on **Solution Containers** as well as all containers in our Docker Desktop:

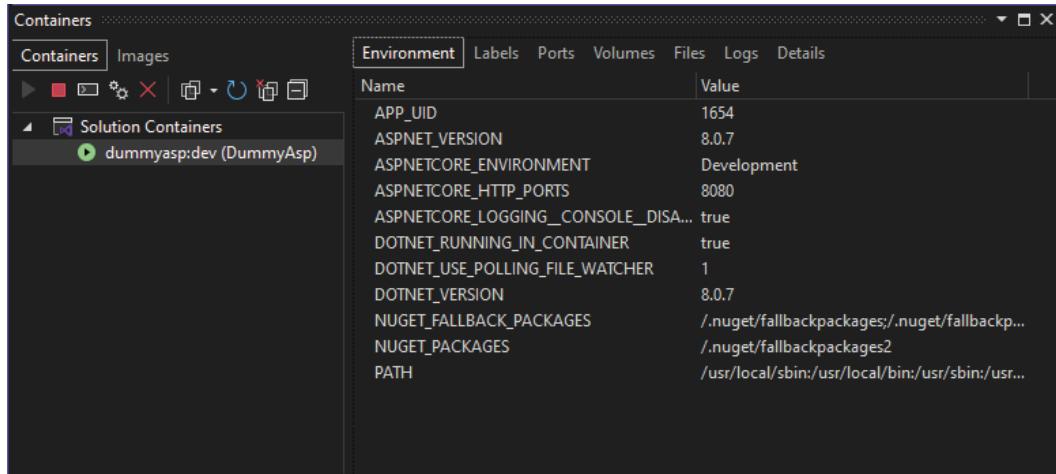


Figure 12.7 – The Containers window

Notice that Docker Desktop must be running on your computer to ensure that the Docker Engine is running and provides the necessary environment for building, managing, and running Docker containers.

The container window allows us to quickly access the containers' information directly in Visual Studio in order to monitor container status, view logs, and manage the container life cycle without leaving Visual Studio.

The toolbar allows us to manage the containers:

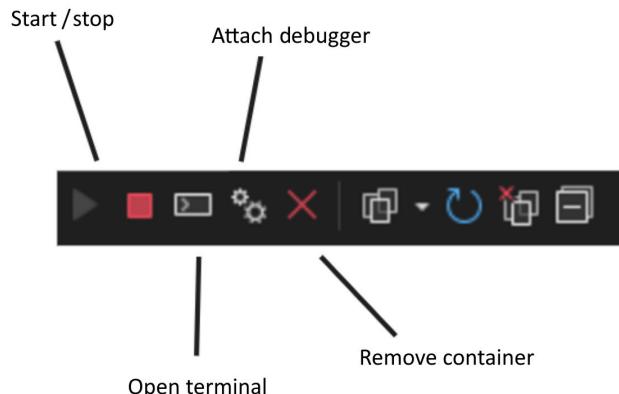


Figure 12.8 – Container toolbox

Let's understand these toolbar options in detail:

- **Start/stop:** These options allow you to manage the life cycle of the container directly from Visual Studio, without needing to use Docker CLI commands.
- **Attach debugger:** For containers running applications that support debugging (e.g., .NET Core applications), this option allows attaching a debugger to the container. This enables setting breakpoints and inspecting variables as if the application were running locally.
- **Open terminal:** This opens a terminal session inside the container. This is particularly useful for executing commands within the container's environment.
- **Remove container:** This allows you to delete the container. This is useful for cleaning up stopped containers that are no longer needed.

Additionally, we can find panels on the right, allowing us to consult all the information about the container.

The **Logs** pane displays the container's logs. It's useful for debugging and monitoring the container's output:

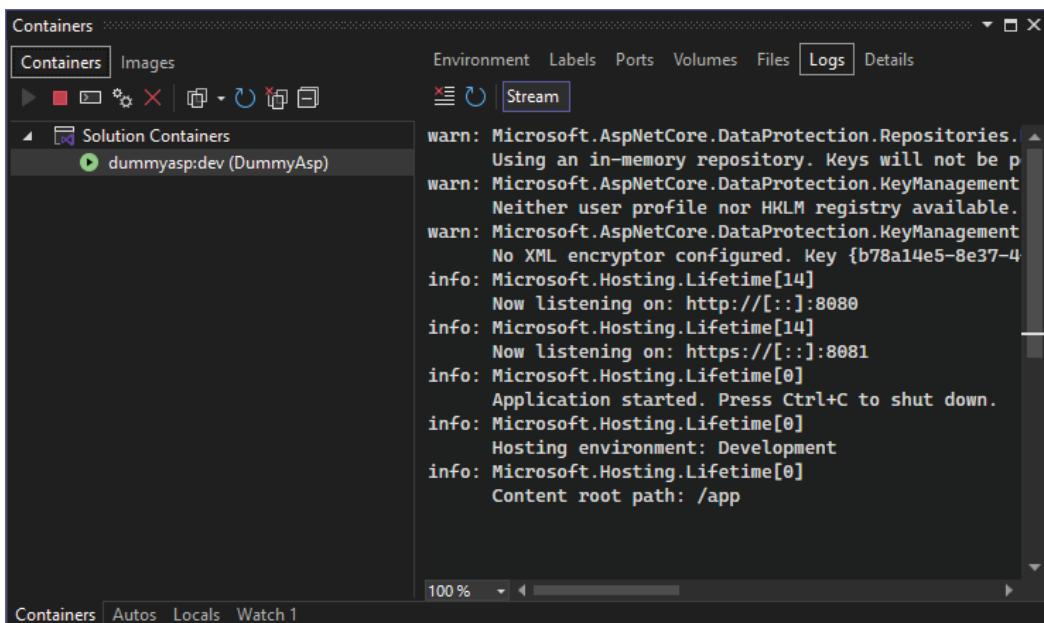


Figure 12.9 – Containers | Logs

This display automatically shows when we are launching our container in debug.

Now we can run our container without getting out of Visual Studio, let's see how to deploy it.

## Deploying containerized applications

When the application is developed, the next step is to deploy it. In this section, we will see the built-in options offered by Visual Studio to achieve that.

First, we access the publish wizard by right-clicking on the project through **Solution Explorer** and then selecting the **Publish...** option.

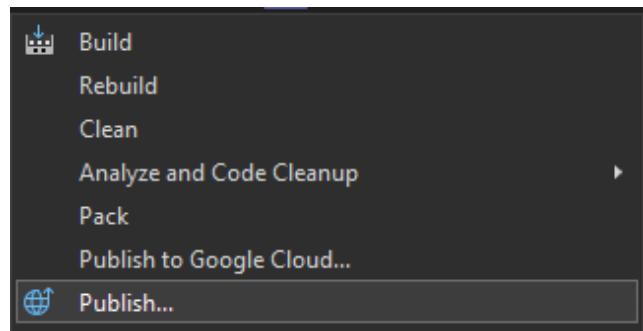


Figure 12.10 – Publish...

We already have seen in *Chapter 8* and *Chapter 10* that it will open a window allowing us to choose the destination of our publication. In this chapter, we will focus on two options, **Docker Container Registry** and **Azure**.

### Deploying in Container Registry

Let's begin with a reminder of what a container registry is. A **container registry** is a centralized storage and distribution system for named Docker images and their associated tags. It acts as a repository for container images, allowing us to build, share, and deploy containerized applications efficiently. Container registries can be either public or private, depending on whether they are accessible to everyone or restricted to authorized users.

So, in this case, in the **Target** window, we will choose **Docker Container Registry**:

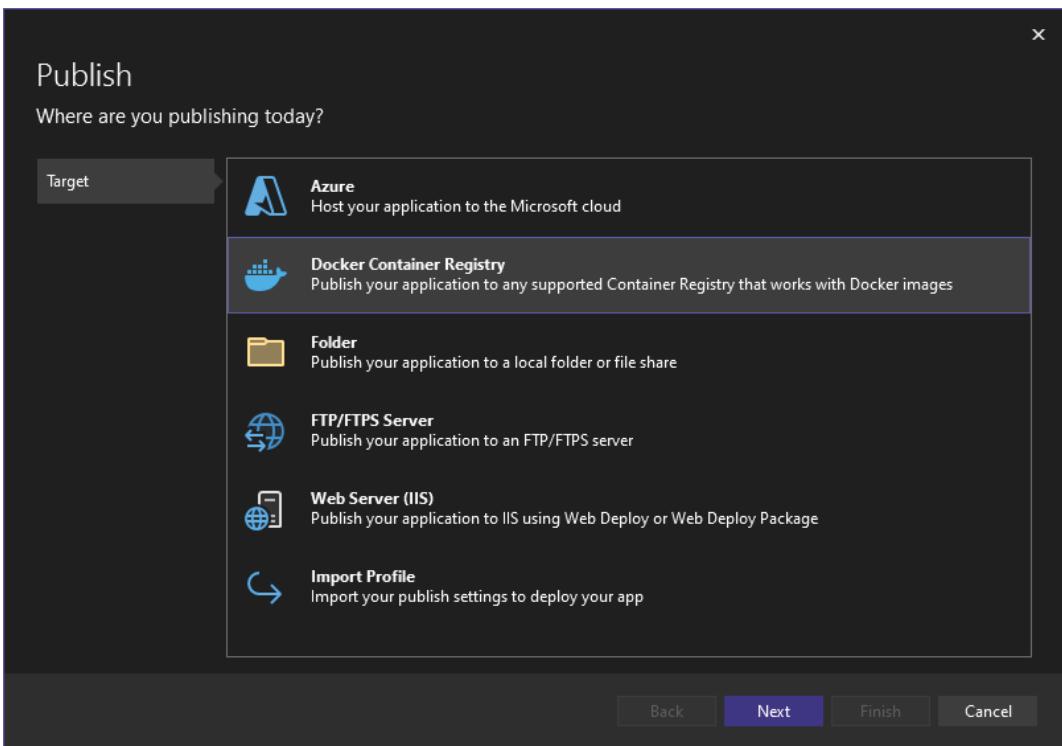


Figure 12.11 – Docker Container Registry

Then, click **Next** to jump into the **Specific target** tab to select the host of our application:

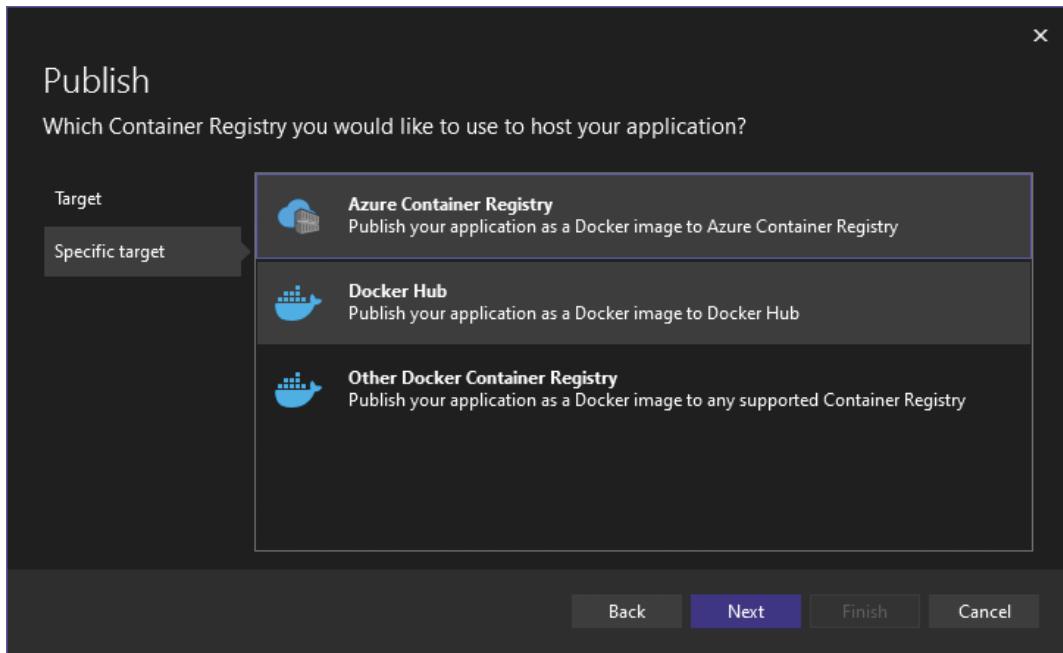


Figure 12.12 – Specific target Container Registry

Here, we have three options:

- **Azure Container Registry:** The **Azure Container Registry (ACR)** is a private, managed, and secured Docker Registry provided by Azure. It allows us to store and manage our Docker images and related artifacts in a central registry, enabling reliable, secure, and scalable deployments. ACR offers secure storage of Docker images behind Azure's network security and access control mechanisms.
- **Docker Hub:** **Docker Hub** is a public, cloud-based repository in which Docker users and partners create, test, store, and distribute container images. While not as secure or feature-rich as ACR for enterprise use, Docker Hub is widely used for sharing open source projects and base images.
- **Other Docker Container Registry:** Besides **ACR** and Docker Hub, there are several other container registries available, each with its own set of features and pricing models. Some notable ones include **Google Container Registry (GCR)** and Amazon **Elastic Container Registry (ECR)**.

For ACR, Visual Studio will ask us to use a valid Azure subscription. On the other hand, it will prompt us to provide our Docker credentials if we choose Docker Hub.

## Deploying as a service in Azure

Back to the **Target** window, we select **Azure**:

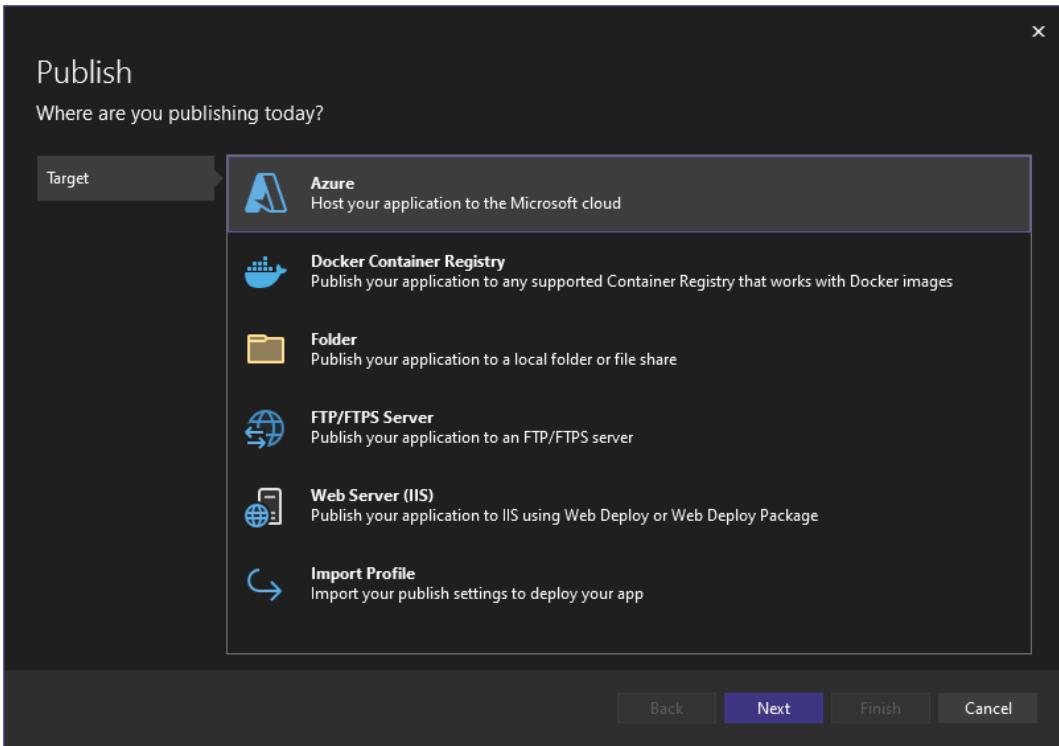


Figure 12.13 – Target Azure

This will bring us to the **Specific target** window where we find several options to deploy our application on our Azure subscription.

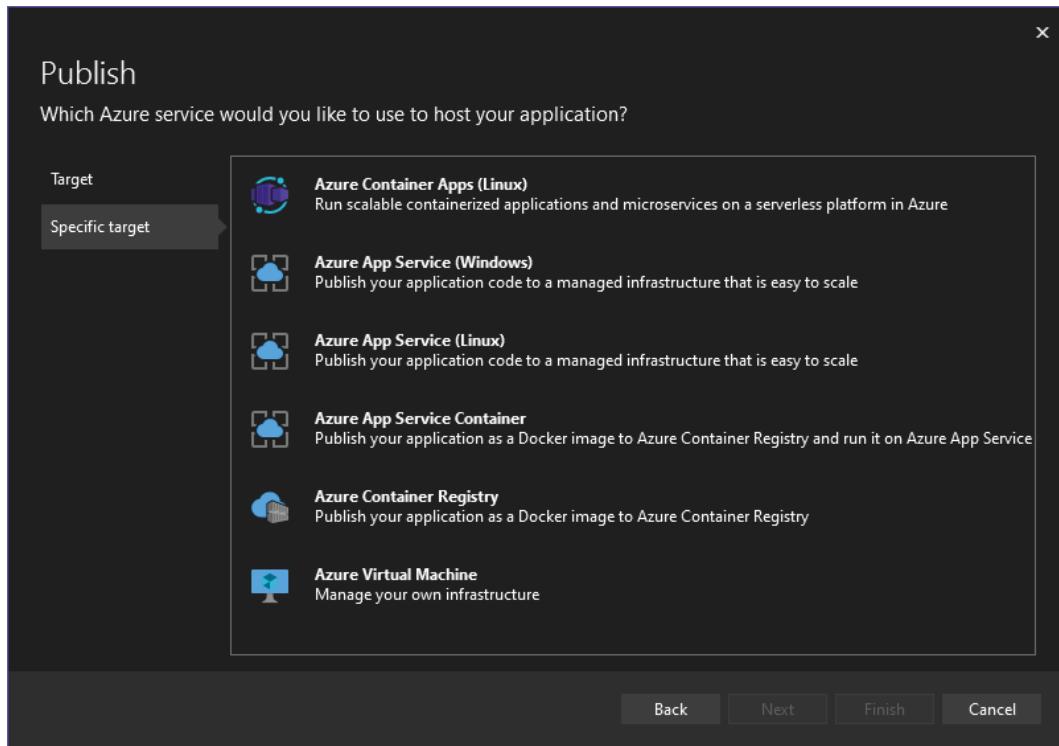


Figure 12.14 – Specific target Azure

In this chapter, we will focus on these container options:

- **Azure Container Apps (Linux):** **Azure Container Apps** is a fully managed service designed to run containers directly, without the need for orchestrators, such as Kubernetes. It's optimized for running microservices and serverless applications, offering built-in support for source code repositories, CI/CD pipelines, and automatic scaling.
- **Azure App Service Container:** **Azure App Service** is a fully managed platform for building, deploying, and scaling web apps. The container option within App Service allows us to run Docker containers, providing a simple way to deploy and manage containerized applications or websites.
- **Azure Container Registry:** Here, we retrieve the ACR, which is the private, managed, and secured Docker Registry provided by Azure.

Each option serves different needs, from serverless containers optimized for microservices to fully managed environments for web applications and secure, private registries for Docker images.

## Summary

In this chapter, we explored the robust integration between Docker and Visual Studio, a combination that significantly enhances your development workflow. We began with an introduction to Docker and Visual Studio integration, where we covered the basics of Docker and its significance in modern software development. The next section guided you through the steps to configure Docker within Visual Studio with Docker support. Next, we focused on the practical aspects of converting your applications into Docker containers. Finally, we discussed various deployment strategies available in Visual Studio.

By mastering these skills, you are now equipped to streamline your development process, improve deployment speed, and adopt modern DevOps practices effectively staying in Visual Studio.

In the next chapter, we will delve into crafting our own Visual Studio extensions, where we'll learn how to customize and extend Visual Studio to fit your unique development needs.



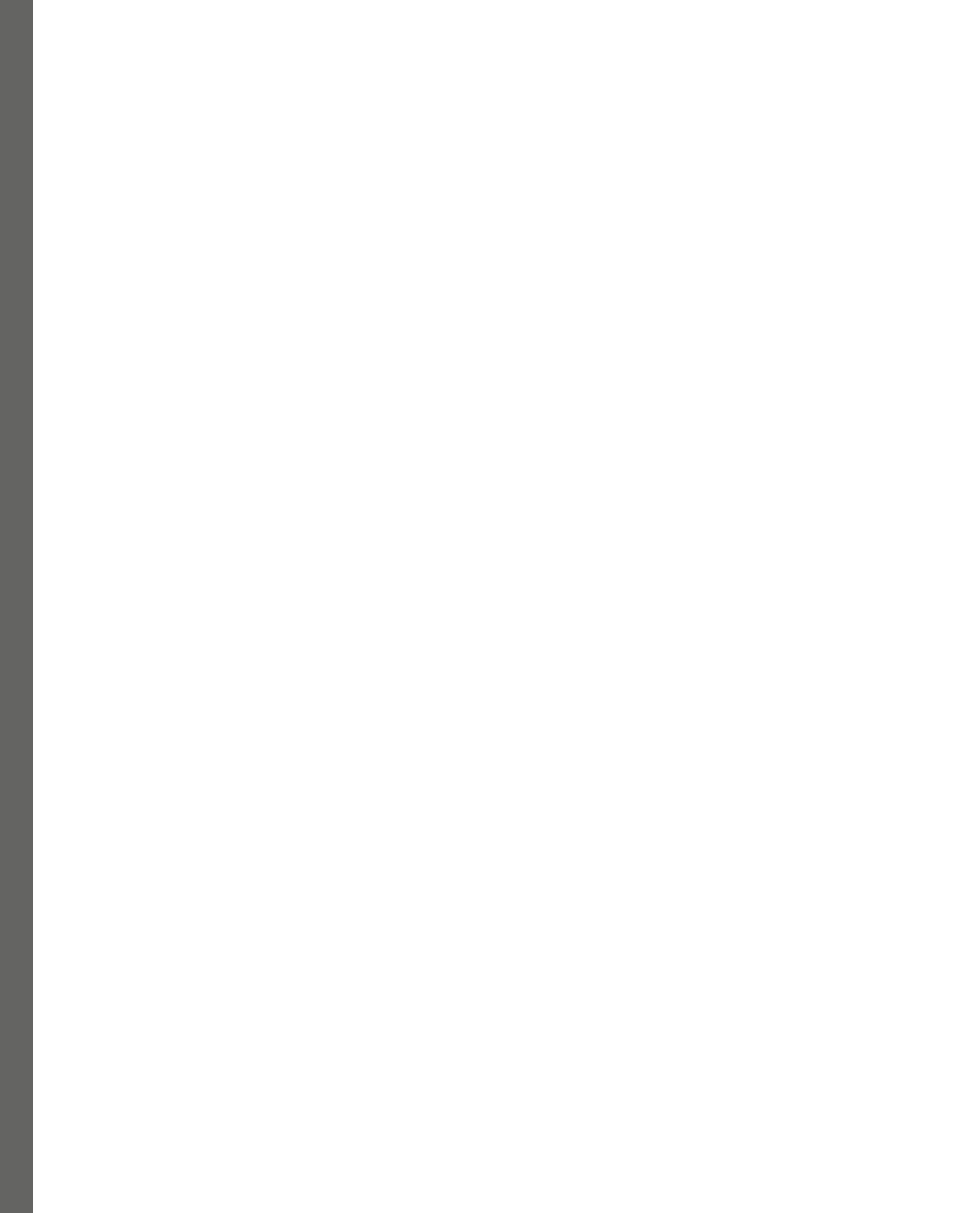
# Part 4:

## Mastering Core Development Skills

In this final part, we focus on customizing and enhancing your development environment. You'll learn how to write your own project templates, create Visual Studio extensions, and build NuGet packages to share with the development community and within your organization. These skills will empower you to tailor Visual Studio to your needs, streamlining workflows and boosting productivity.

This part has the following chapters:

- *Chapter 13, Writing Your Own Project Template*
- *Chapter 14, Writing Your Own Visual Studio Extensions*
- *Chapter 15, Creating and Publishing Powerful NuGet Packages for the Community*



# 13

## Writing Your Own Project Template

In this chapter, we will dive into the world of Visual Studio by exploring how to create custom project templates. These templates are incredibly powerful tools that allow us to streamline our development process, ensuring that our team starts every project with a consistent structure and configuration. We'll begin by breaking down the fundamental structures that make up a project template, giving you a solid foundation to build upon. As we progress, we'll discover how to integrate parameters into our templates, allowing for dynamic customization at the point of project creation. Finally, we'll delve into advanced features, teaching you how to extend your project templates into full-fledged solution templates that can handle complex, multi-project solutions.

In this chapter, we're going to cover the following main topics:

- Understanding project template structures
- Building a basic project template
- Customize project templates for different workflows
- Integrating template parameters
- Extending project templates with advanced features

By the end of this chapter, you'll not only have the knowledge to create and customize project templates, but you'll also understand how to leverage these tools to enhance productivity and maintain consistency across your projects.

## Technical requirements

While writing this chapter, I used the following version of Visual Studio:

- Visual Studio Enterprise 2022 Version 17.12.0
- Preview 1.0

The code files for this chapter can be found at <https://github.com/PacktPublishing/Mastering-Visual-Studio-2022/tree/main/ch13>.

## Understanding project template structure

We all use templates when we create a new project or a new item with Visual Studio. In this section, we will cover the difference between project and item templates and then see what files are involved in this process. Project templates in Visual Studio are powerful tools that help developers quickly set up new projects with predefined configurations and structures. On the other hand, we can create item templates for specific parts of the project. These templates provide a base that can be customized according to specific needs, streamlining the development process and ensuring consistency across projects.

Both project templates and item templates in Visual Studio are reusable structures designed to simplify development by providing basic code frameworks and configurations that can be tailored to specific needs. Despite their similarities, they differ significantly in scope and application within the Visual Studio environment.

**Project templates** serve as foundational blueprints for creating new projects. They include the entire project structure, necessary files, references, and configuration settings tailored to a specific project type (e.g., ASP.NET Core Web App, Class Library). When starting a new project in Visual Studio, we choose a project template that matches our requirements. This template establishes the initial project framework, including default namespaces, assembly references, and compiler options, saving time and ensuring consistency across projects.

**Item templates** focus on individual components within a project. These templates can range from simple file types (e.g., XML, HTML, CSS) to complex structures involving multiple files and resources. They are designed to quickly add predefined items to an existing project. We use item templates when adding new elements to our project via the **Add New Item** dialog box. This could include anything from a class file or interface to a pre-configured web page. Item templates enable the rapid inclusion of these elements without the need for manual configuration.

The creation of new templates involves working with the different files described in the following points:

- **Files to be created:**

- **Source code files:** These are the initial coding files included with the template. For example, a C# class library template might start with a default `Class1.cs` file.
- **Embedded resources:** This category includes images, configuration files, or other resources the project might require.
- **Project files:** These encompass the solution and project files (such as `.sln` and `.csproj` for C# projects) that outline the project's structure and dependencies.

- **.vstemplate:**

- This XML file is essential for defining the template. It contains metadata such as the template's name, description, icon, and project type.
- It outlines the files to be included in the project and specifies any additional parameters or wizard data that need to be processed when the template is instantiated.

- **Compressed into a ZIP file:**

- Once the template and its associated files are prepared, they are compressed into a `.zip` file. This file is placed in a specific folder where Visual Studio can recognize it.
- For project templates, the `.zip` file is placed in the `\Documents\Visual Studio <version>\Templates\ProjectTemplates` directory.
- For item templates, it goes into the `\Documents\Visual Studio <version>\Templates\ItemTemplates` directory.

Project templates in Visual Studio provide a starting point for entire projects, offering a structured foundation based on the chosen template's specifications. Conversely, item templates serve to expedite the addition of individual components or files within those projects, streamlining the development workflow by providing reusable pieces of code or resources. Understanding the distinction between these two types of templates can significantly enhance productivity and consistency in software development practices.

Now, let's learn how to create basic templates.

## Building a basic project template

In this section, we will cover the more convenient way to build a project template. The process consists of building a skeleton project with the minimum we need inside to fit with our company requirements for example. After that, we can export the existing project as a template using Visual Studio's Export Template Wizard, by using the Visual Studio top bar menu: **Project | Export Template...**:

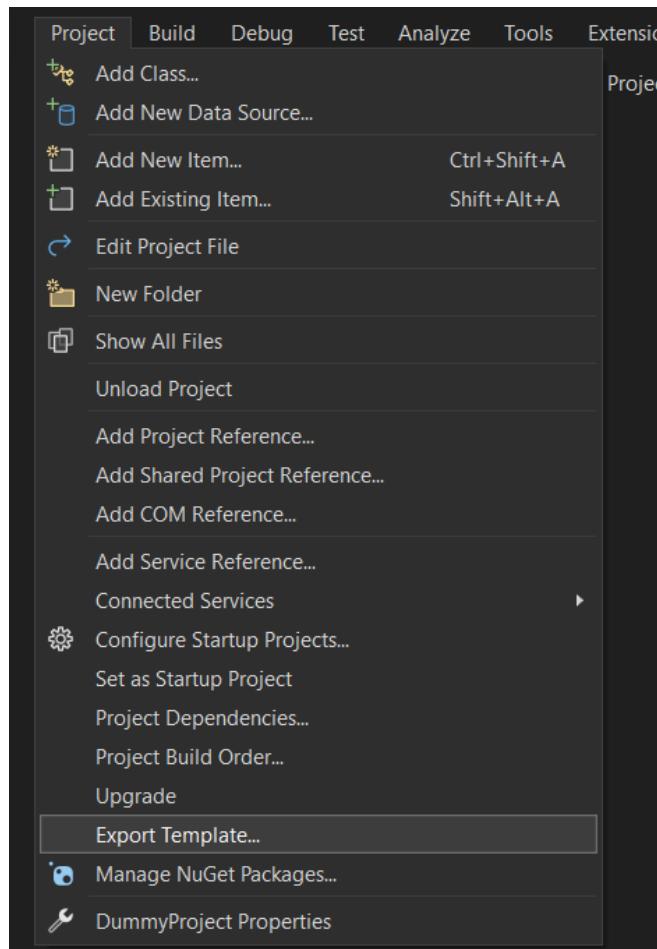


Figure 13.1 – Export Template...

The preceding action will open the **Export Template Wizard** window:

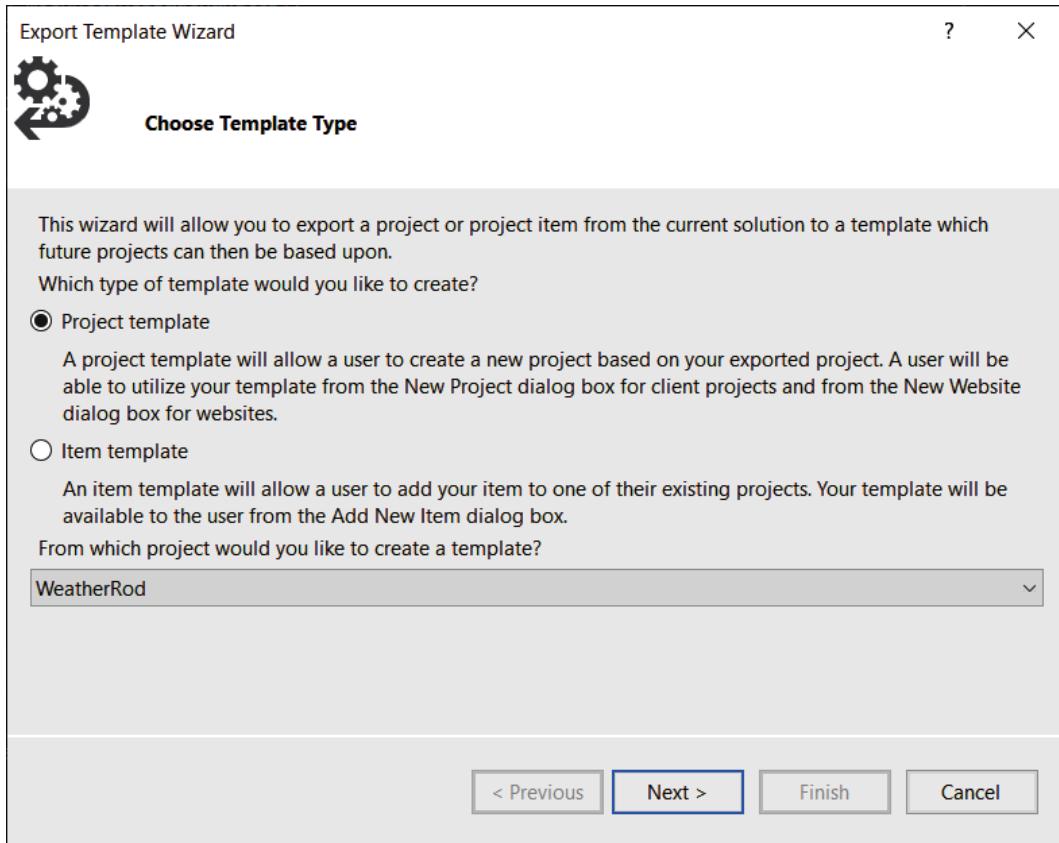


Figure 13.2 – Export Template Wizard

First, we choose the type of template we want to export; either **Project template** or **Item template**.

Then, the tool allows us to define key details, such as the template's name, description, icon, and preview images.

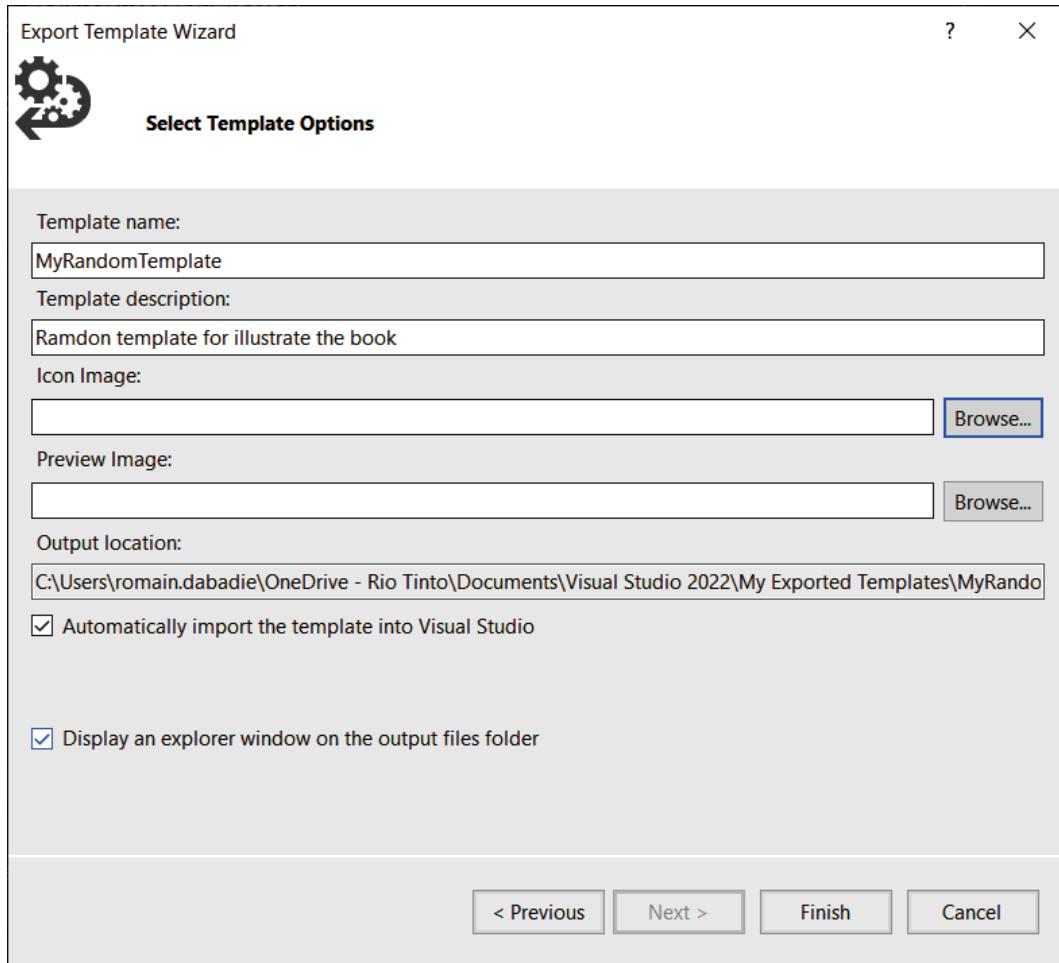


Figure 13.3 – Select Template Options

Upon completion, the project is bundled into a .zip file and saved to a chosen output location. Additionally, we can opt to directly import the template into Visual Studio for immediate utilization. To access and use our newly created template, open the dialog box for creating a new project.

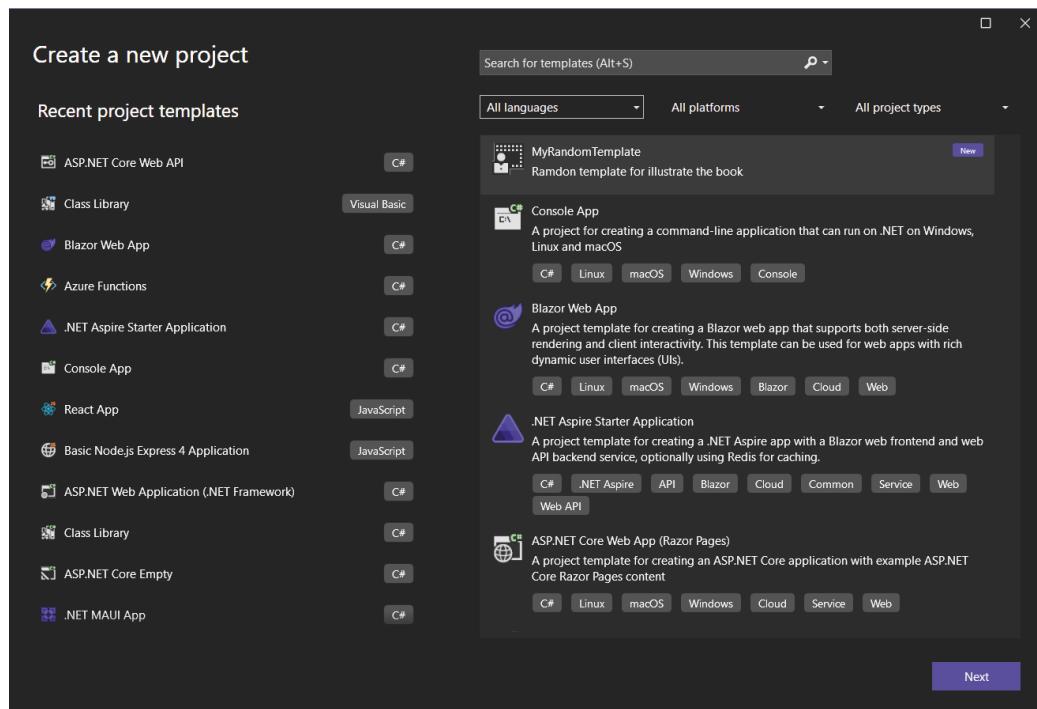


Figure 13.4 – Use the custom template

We can locate the template by searching for its name or browsing through the list. Keep in mind that filtering by language or project type may not be supported for custom templates.

Let's explore how to custom templates.

## Customizing project templates for different workflows

There are two ways to update a Visual Studio template either by using the Export Template Wizard or by manually adjusting the files within the template.

The first method is to utilize the Export Template Wizard by initiating a new project within the project template we are interested in modifying and then following these steps:

1. Within the project, carry out the modifications we desire. This could encompass changing the output type, incorporating new files, or implementing other alterations.
2. **Export the modified template:** With our project adjusted, head to **Project | Export Template** to access the Export Template Wizard.
3. **Conclude the export process:** Adhere to the wizard's instructions to successfully export our modified template as a **.zip** file.

The other option is to apply the manual modification to a template by revising the `.vstemplate` by following these steps:

1. **Identify the template:** Locate the `.zip` archive corresponding to the template we intend to alter. Typically, this resides in `%USERPROFILE%\Documents\Visual Studio <version>\Templates\ProjectTemplates`.
2. **Unpack the archive:** Extract the contents of the template archive.
3. **Edit the files:** Undertake the necessary edits, additions, or deletions to the files within the template.
4. **Revise the `.vstemplate` file:** Adjust the `.vstemplate` XML file to accurately represent the changes made.
5. **Archive the changes:** Compress all the modified files back into a `.zip` archive.
6. **Implement the updated template:** Transfer the newly compressed `.zip` file to the original directory, replacing the outdated template files.

Customizing allows us to use some parameters and variables in existing templates. Let's see in the next section how to achieve that.

## Integrating template parameters and variables

Now that we have seen how to open an existing `.vstemplate` in order to customize it, in this section, we will explore how to leverage parameters to dynamically replace values within the template when it's instantiated. Using parameters and variables in `.vstemplate` files with Visual Studio allows us to create dynamic templates that can adapt to different project names, namespaces, and other customizable aspects. This feature enhances the reusability and customization of our templates, making template creation a powerful tool for streamlining development workflows.

**Template parameters** are placeholders within our template that get replaced with actual values when the template is instantiated. These parameters can represent various aspects of our project, such as the project name, namespace, or even custom-defined values.

Visual Studio provides a set of reserved template parameters that you can use directly in the templates. Here is the list of these reserved parameters, according to the Microsoft documentation:

- `clrversion`: The current version of the **common language runtime (CLR)**.
- `ext_*`: The prefix added to any parameter to refer to variables of the parent template (e.g., `ext_safeprojectname`).
- `guid[1-10]`: A GUID used to replace the project GUID in a project file. Up to 10 unique GUIDs can be specified (e.g., `guid1`).

- **itemname:** The name of the file in which the parameter is being used.
- **machinename:** The current computer name (e.g., Computer01).
- **projectname:** The name provided by the user when the project was created. This applies only to project templates.
- **registeredorganization:** The registry key value from HKEY\_LOCAL\_MACHINE\Software\Microsoft\WindowsNT\CurrentVersion\RegisteredOrganization.
- **rootnamespace:** The root namespace of the current project followed by the subfolder of the current item, with slashes replaced by periods.
- **defaultnamespace:** The root namespace of the current project.
- **safeitemname:** The same as `itemname` but with all unsafe characters and spaces replaced by underscore characters.
- **safeitemrootname:** Same as `safeitemname`.
- **safeprojectname:** The name provided by the user when the project was created but with all unsafe characters and spaces removed. This applies only to project templates.
- **targetframeworkversion:** The current version of the target .NET Framework.
- **time:** The current time in a format based on Windows user settings (e.g., DD/MM/YYYY 00:00:00).
- **specifiedsolutionname:** The name of the solution. Behavior varies based on whether **Place solution and project in the same directory** or **Create solution directory** is checked.
- **userdomain:** The current user domain.
- **username:** The current username.
- **webnamespace:** The name of the current website, used in web form templates to guarantee unique class names.
- **year:** The current year in YYYY format.
- **solutiondirectory:** The directory of the solution.
- **destinationdirectory:** The path to the directory of the `.csproj` once created.

These reserved parameters can be used to dynamically insert project-specific information into your templates. For instance, we can use the `$safeprojectname$` and `$year$` reserved parameters to dynamically generate the namespace and add a copyright notice.

For this example, we will create a new template manually. This example will focus on creating a C# class library project template that uses several reserved template parameters to dynamically generate namespaces, class names, and file names based on user input and system information. Let's get started with the steps:

1. First, define the structure of our project template in a `.vstemplate` file. This file specifies what gets created when the template is instantiated:

```
<VSTemplate Version="3.0.0"
    xmlns="http://schemas.microsoft.com/developer/
        vstemplate/2005"
    Type="Project">
    <TemplateData>
        <Name>Masterize VS ClassTemplate</Name>
        <Description>TemplateParametree</Description>
        <ProjectType>CSharp</ProjectType>
        <SortOrder>1000</SortOrder>
        <CreateNewFolder>true</CreateNewFolder>
        <DefaultName>ClassTemplate</DefaultName>
        <ProvideDefaultName>true</ProvideDefaultName>
        <LocationField>Enabled</LocationField>
        <EnableLocationBrowseButton>
            true
        </EnableLocationBrowseButton>
        <CreateInPlace>true</CreateInPlace>
        <Icon>_TemplateIcon.ico</Icon>
    </TemplateData>
    <TemplateContent>
        <Project TargetFileName="ClassTemplate.csproj"
            File="ClassTemplate.csproj"
            ReplaceParameters="true">
            <ProjectItem
                ReplaceParameters="true"
                TargetFileName="Class1.cs">
                Class1.cs
            </ProjectItem>
            <ProjectItem
                ReplaceParameters="true">
                Properties\AssemblyInfo.cs
            </ProjectItem>
        </Project>
    </TemplateContent>
</VSTemplate>
```

Notice that it's common practice to specify `TargetFileName` for every `ProjectItem` if you need them to have a specific name in the newly created project. The second `<ProjectItem>` (`Properties\AssemblyInfo.cs`) does not have `TargetFileName`. If you want to ensure that this file has the same name in the new project, this is fine. If you want to replace parameters, it should also have the `ReplaceParameters="true"` attribute (as it currently does).

2. Next, create a C# class file named `Class1.cs` that will be part of our template. We'll use the `$safeprojectname$` and `$year$` reserved parameters to dynamically generate the namespace and add a copyright notice:

```
using System;

namespace $safeprojectname$
{
    /// <summary>
    /// Represents a simple class in the
    /// $safeprojectname$ project.
    /// Copyright (c) $year$ by Company name
    /// </summary>
    public class Class1
    {
        public Class1()
        {
            Console.WriteLine(
                "Hello from $safeprojectname$!"
            );
        }
    }
}
```

3. We will also create an `AssemblyInfo.cs` file under a `Properties` folder within our template. This file will contain assembly metadata, and we'll use the `$projectname$`, `$year$`, and `$username$` reserved parameters to customize the assembly title and copyright:

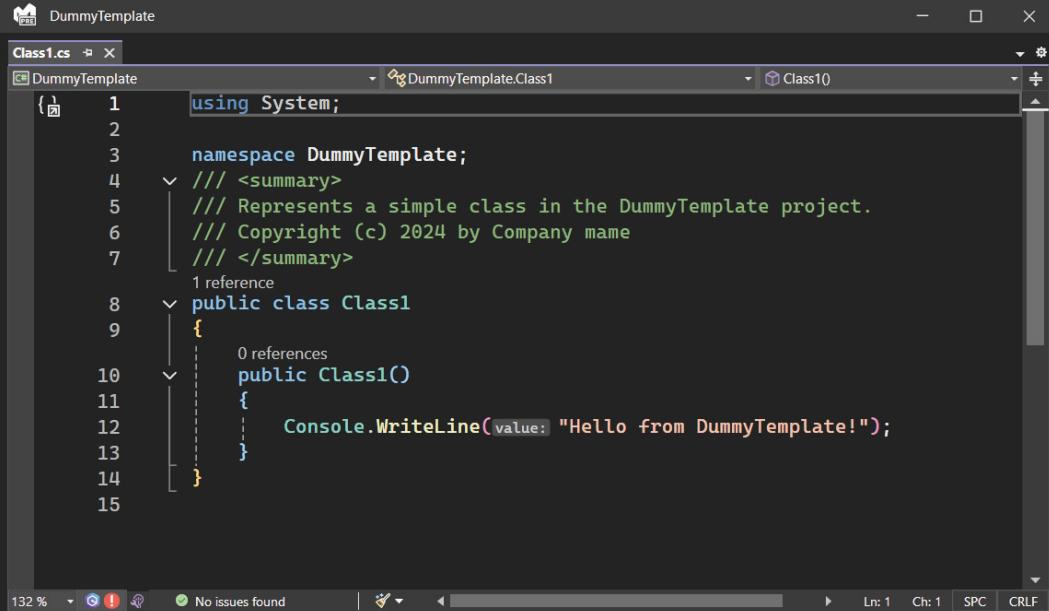
```
using System.Reflection;
using System.Runtime.CompilerServices;

[assembly: AssemblyTitle("$projectname$")]
[assembly: AssemblyDescription("A simple class library
project.")]
[assembly: AssemblyCompany("Your Company")]
[assembly: AssemblyProduct("$projectname$")]
[assembly: AssemblyCopyright("Copyright © $year$ $username$. All
rights reserved.")]
```

```
[assembly: AssemblyTrademark("")]
[assembly: AssemblyVersion("1.0.*")]
```

- After creating these files, package them into a .zip file and place them in the Visual Studio templates directory or import them through Visual Studio's template manager.

When creating a new project based on this template, Visual Studio will prompt us for the project name and location. As a result, the `Class1.cs` file will be filled with the provided values:



The screenshot shows the Visual Studio IDE with the title bar "DummyTemplate". The code editor window displays the file "Class1.cs" under the project "DummyTemplate". The code content is as follows:

```
1 using System;
2
3 namespace DummyTemplate;
4 /// <summary>
5 /// Represents a simple class in the DummyTemplate project.
6 /// Copyright (c) 2024 by Company name
7 /// </summary>
8 public class Class1
9 {
10     public Class1()
11     {
12         Console.WriteLine(value: "Hello from DummyTemplate!");
13     }
14 }
```

The code editor interface includes tabs for "Class1.cs", "DummyTemplate", "D:\DummyTemplate.Class1", and "Class10". The status bar at the bottom shows "132 %", "No issues found", and other standard build and navigation icons.

Figure 13.5 – Generate Class1

It will then replace the reserved parameters in the template files with the actual values, such as the safe project name, current year, and username. In this example, I choose `DummyTemplate` as the project name and the system takes the current year to set the `$year$` parameter.

At the point when we create a project from our custom template, you might notice that we can use the filter combo box to filter them. Within the `<TemplateData>` section of our `.vstemplate` file, we can add the following elements to specify tags:

- `<LanguageTag>`: This specifies the programming languages used in the template
- `<PlatformTag>`: This indicates the target platforms (e.g., Windows, web)
- `<ProjectTypeTag>`: This describes the type of project (e.g., Class Library, Console Application)

Each tag element should contain the name of the tag as its text content:

```
<TemplateData>
  ...
  <LanguageTag>C#</LanguageTag>
  <PlatformTag>Windows</PlatformTag>
  <ProjectTypeTag>Class Library</ProjectTypeTag>
</TemplateData>
```

When updating our template this way, we can observe the tags that appear on the description in the template list.

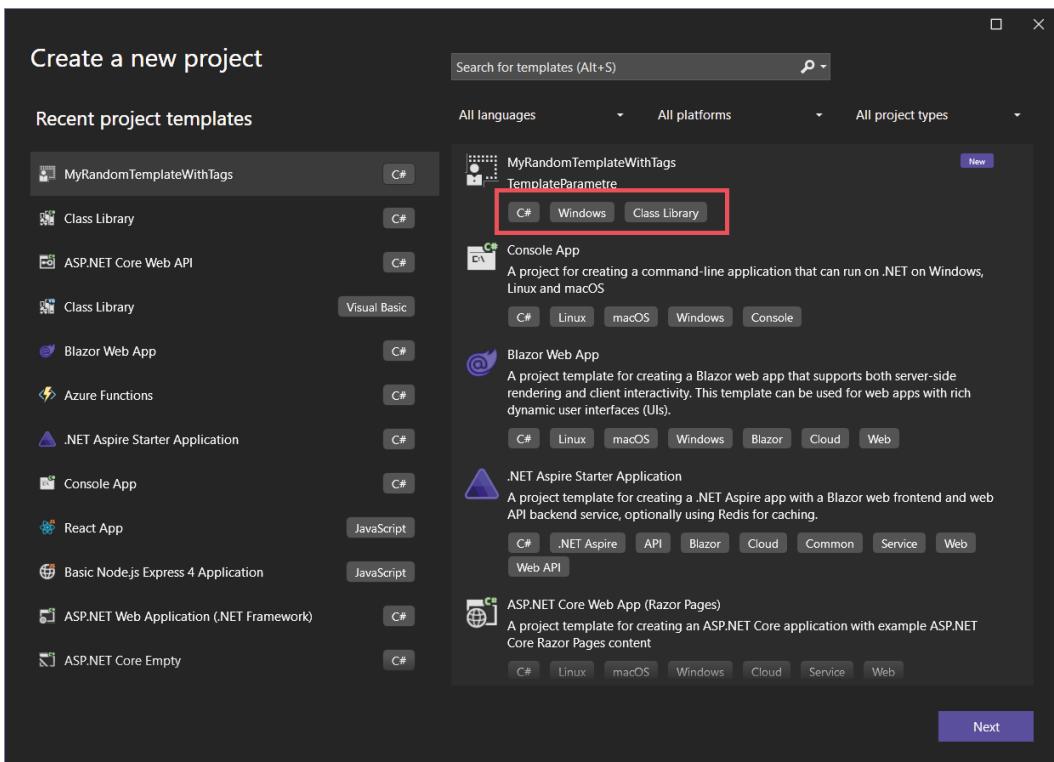


Figure 13.6 – Template list with tags

These tags allow us to leverage the three filters upon the list when we are looking for a template.

Sometimes, we need more than one project, and we expect to be more productive with a ready-to-use solution with multi-library projects to scaffold a clean architecture, for example; let's see in the next section how we can achieve that.

## Extending project templates with advanced features

In this section, we will go further by creating what we can call a solution template by combining multiple project templates.

Creating a multi-project solution template in Visual Studio involves several steps, including creating individual project templates, combining them into a single solution template, and configuring the solution template to instantiate multiple projects upon creation. Let's get started:

1. First, create the templates that will be included in the solution template. For that, follow the instructions provided in the *Building a basic project template* section of this chapter.
2. Next, prepare a directory structure for the solution template. This involves creating a folder for the solution template and copying the unzipped contents of each project template into this directory.

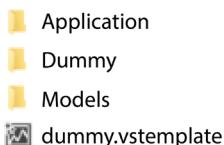


Figure 13.7 – Folder structure

3. For this example, I created three project templates to scaffold the solution.
4. Now, create or modify a `.vstemplate` file in the root of the solution template directory. This file defines the structure of our solution template, including references to the individual project templates. In this file, we set the **Project Type** option to **ProjectGroup** to indicate that this template represents a collection of projects as follows:

```
<VSTemplate Version="3.0.0"
    xmlns="http://schemas.microsoft.com/developer/
        vstemplate/2005"
    Type="ProjectGroup">
    <TemplateData>
        <Name>CleanDummyArch</Name>
        <Description>Dummy solution</Description>
        <ProjectType>Web</ProjectType>
        <ProjectSubType>CSharp</ProjectSubType>
        <SortOrder>1000</SortOrder>
        <CreateNewFolder>true</CreateNewFolder>
        <DefaultName>DummySolution</DefaultName>
        <ProvideDefaultName>true</ProvideDefaultName>
        <LocationField>Enabled</LocationField>
```

```
<EnableLocationBrowseButton>
    True
</EnableLocationBrowseButton>
<CreateInPlace>true</CreateInPlace>
<Icon>__TemplateIcon.ico</Icon>
<LanguageTag>C#</LanguageTag>
<PlatformTag>Windows</PlatformTag>
<ProjectTypeTag>Class Library</ProjectTypeTag>
</TemplateData>
<TemplateContent>
    <ProjectCollection>
        <ProjectTemplateLink
            ProjectName="$safe projectName$">
            Dummy\MyTemplate.vstemplate
        </ProjectTemplateLink>
        <ProjectTemplateLink
            ProjectName="$safe projectName$.Application">
            Application\MyTemplate.vstemplate
        </ProjectTemplateLink>
        <ProjectTemplateLink
            ProjectName="$safe projectName$.Models">
            Models\MyTemplate.vstemplate
        </ProjectTemplateLink>
    </ProjectCollection>
</TemplateContent>
</VSTemplate>
```

5. Inside `<TemplateContent>`, we use `<ProjectCollection>` instead of `<Project>` to define a collection of projects. Each child project is linked by a `<ProjectTemplateLink>` element that establishes the link, pointing to the relative path of the project template within the solution template directory.
6. Once our solution template is configured, we do the following:
  - We zip the contents of the solution template directory, including the modified `.vstemplate` file and the unzipped project templates
  - We place the zipped file in the Visual Studio templates directory or import it through Visual Studio's template manager
7. Finally, test the solution template by creating a new project from it in Visual Studio to ensure that it correctly generates a solution with all the desired projects, through **Solution Explorer**.

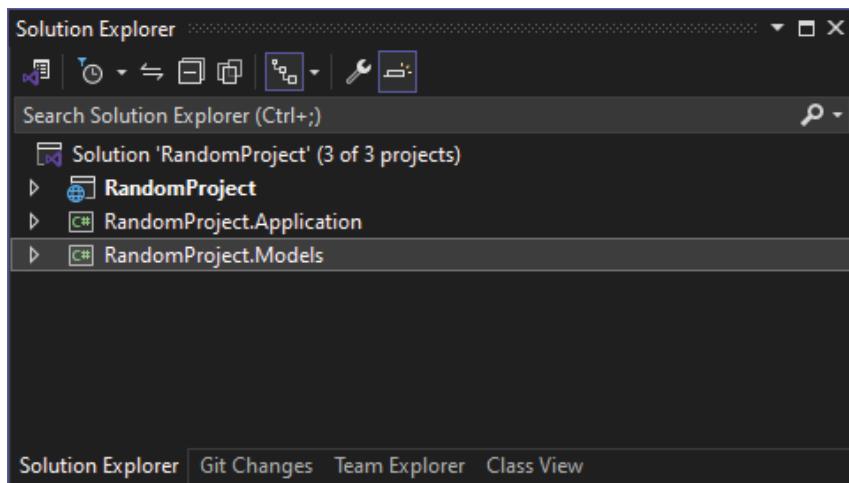


Figure 13.8 – Solution architecture

As a result, we initiate a solution with three projects as defined in the template. This process allows us to create a reusable template for a solution that includes multiple projects, streamlining the setup of new solutions based on predefined configurations, saving time and ensuring consistency across projects.

## Summary

In this chapter, we've explored the art of crafting custom project templates in Visual Studio 2022, a crucial skill for any developer aiming to standardize and accelerate their workflow. We began by understanding the fundamental structures that form the backbone of project templates. We then delved into customization, showing you how to adapt these templates to suit different workflows, ensuring they meet the specific needs of your projects. From there, we explored the integration of parameters, allowing you to add dynamic elements to your templates for greater flexibility. Finally, we expanded our focus to include advanced features, enabling you to create solution templates that can manage intricate, multi-project solutions with ease.

As we move forward, we'll build on this knowledge by exploring another powerful aspect of Visual Studio: creating your own extensions. In the next chapter, we'll dive into the world of custom tooling, where you'll learn how to enhance your development environment even further by creating and deploying extensions tailored to your unique needs.

# 14

## Writing Your Own Visual Studio Extensions

In this chapter, we will delve into the world of Visual Studio extension development, where we'll learn how to create tools that will enhance our development environment. **Visual Studio** extensions are powerful, enabling the addition of custom features, automation of repetitive tasks, and creation of tailored workflows that meet specific needs.

We will begin by breaking down the Visual Studio extension architecture, which will provide us with a solid understanding of its core components and how they function within the **integrated development environment (IDE)**. With this foundation, we'll proceed to build and test our first extension using the provided item template. As we advance, we'll explore more sophisticated features, examining the capabilities offered by preload templates. Finally, we'll cover the deployment and sharing process, ensuring we can distribute our extensions efficiently, whether for personal use, within our team, or to the broader developer community.

In this chapter, we're going to cover the following main topics:

- Understanding Visual Studio extension architecture
- Building your first extension
- Advanced extension features
- Deploying and sharing your extension

By the end of this chapter, you'll possess the knowledge and practical skills needed to create, refine, and share Visual Studio extensions that will significantly boost your productivity.

## Technical requirements

While writing this chapter, I used the following version of Visual Studio:

- Visual Studio Enterprise 2022 Version 17.12.0
- Preview 1.0

## Understanding Visual Studio extension architecture

**Visual Studio extensions** are powerful tools that enhance the capabilities of Microsoft Visual Studio, allowing developers to customize and extend the IDE to better suit their development needs. This section delves into the underlying architecture of Visual Studio extensions, highlighting key components, their interactions, and the broader ecosystem they operate within.

Visual Studio extensions are built upon a robust foundation known as the **Visual Studio Shell**, which serves as the core environment for interactions with the IDE. This shell is highly customizable, providing a framework that allows us to create and integrate new features seamlessly. The process of developing an extension begins with the **extension manifest**, a crucial file that contains essential metadata, such as the extension's name, version, description, and dependencies. This manifest is vital for the installation process within Visual Studio and plays a significant role in managing updates once the extension is deployed.

At the heart of the extension lies the **package** component, which encapsulates all the necessary functionalities and assets, including **Dynamic Link Library (DLL)** and images, into a single, deployable unit. This packaging ensures that all required files are correctly installed and configured within the Visual Studio environment. For extensions that need to interact with the **user interface (UI)**, the inclusion of menu commands and tool windows is necessary. These elements, defined using XML, can be dynamically enabled or disabled based on specific conditions or events occurring within the IDE.

When an extension operates at the project level, **project system integration** becomes essential. This involves working with the **project system object model (PSOM)**, which represents the project's structure and properties within Visual Studio. Extensions that offer code generation and analysis tools leverage APIs provided by Visual Studio to generate boilerplate code, refactor existing code, or perform comprehensive code analysis. Finally, the success of an extension often hinges on effectively utilizing various **integration points** offered by Visual Studio, such as event handlers that allow the extension to respond to specific IDE operations, such as opening documents or building solutions. These integration points enable extensions to automate tasks and enhance the user experience by responding to actions taken within the IDE.

In addition to the core components that make up a Visual Studio extension, **Visual Studio Marketplace** plays a pivotal role in the ecosystem. This platform serves as the primary distribution hub for extensions, enabling developers to easily find, install, and manage a wide array of third-party tools. Visual Studio Marketplace not only streamlines the process of accessing extensions but also fosters community engagement by providing a space for feedback and interaction between developers and users.

To support the creation of these extensions, the Visual Studio **Software Development Kit (SDK)** provides all the necessary tools and documentation. The SDK includes libraries, sample code, and comprehensive tutorials designed to help us get started with building our own extensions. This toolkit is essential for navigating the complexities of extension development and ensuring that new tools integrate smoothly with the Visual Studio environment.

With a solid understanding of the foundational components and the supportive ecosystem surrounding Visual Studio extensions, you're now well equipped to take the next step: building your first extension. In the following section, we'll go through the process of creating a basic Visual Studio extension from scratch, demonstrating how to leverage the tools and resources provided by the Visual Studio SDK to bring your ideas to life.

## Building your first extension

Let's create a simple Visual Studio extension that adds a new menu item under the **Tools** menu. When clicked, this menu item will display a message box. This example will guide us through setting up a basic extension project.

Before diving into extension development, it's crucial to set up your environment properly. The first step is to ensure that we have the right tools installed. Begin by opening the **Visual Studio Installer** and modifying the current Visual Studio installation. Within the installer, we'll want to check that the Visual Studio extension development workload is installed.

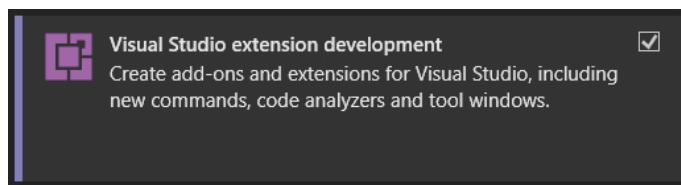


Figure 14.1 – The Visual Studio extension development workload

This workload includes the Visual Studio SDK and other essential tools required for building extensions. Having these components in place is the foundation for a smooth development process, ensuring that you have access to all the libraries, templates, and resources necessary to create and test our extension effectively.

To begin building our extension, follow these steps:

1. Start by opening Visual Studio and selecting the **Create a new project** option for any sort of project.

2. In the project creation window, search for **VSIX Project**. This template is specifically designed for creating Visual Studio extensions. Once we've found it, select the template and click **Next** to proceed.

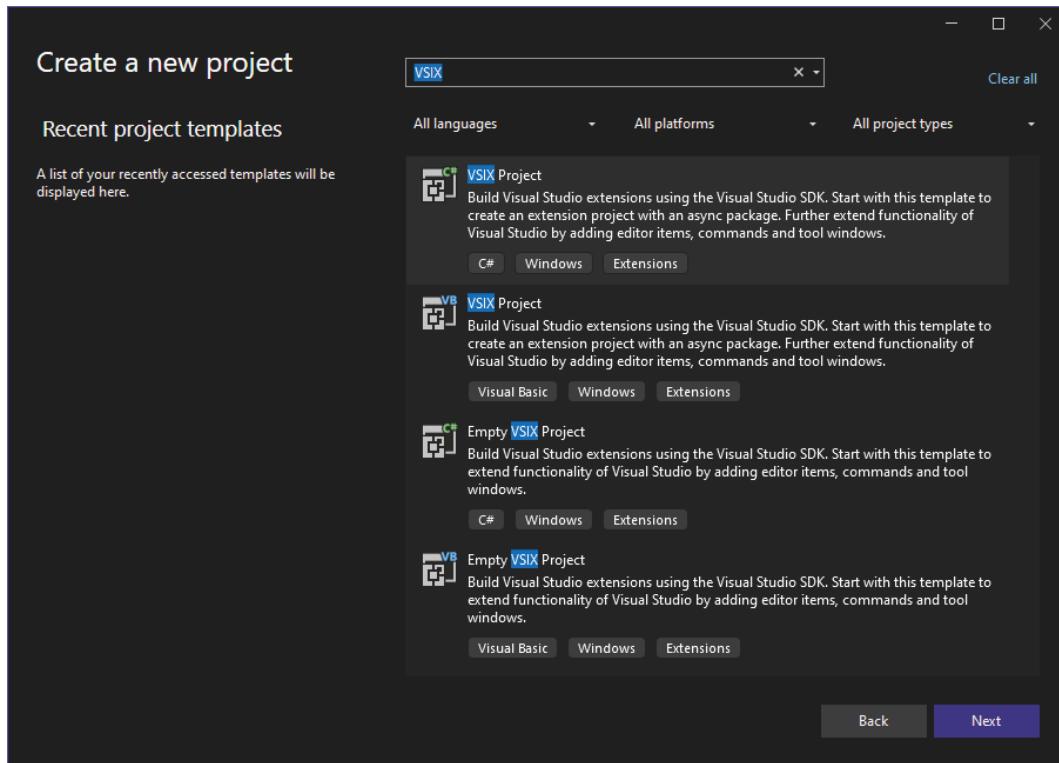


Figure 14.2 – Create a new VSIX project

3. Next, we'll be prompted to name your project. Choose a name that reflects the purpose of our extension, for our example, we will choose `MyFirstExtension`, and specify the location where you want to save the project files.

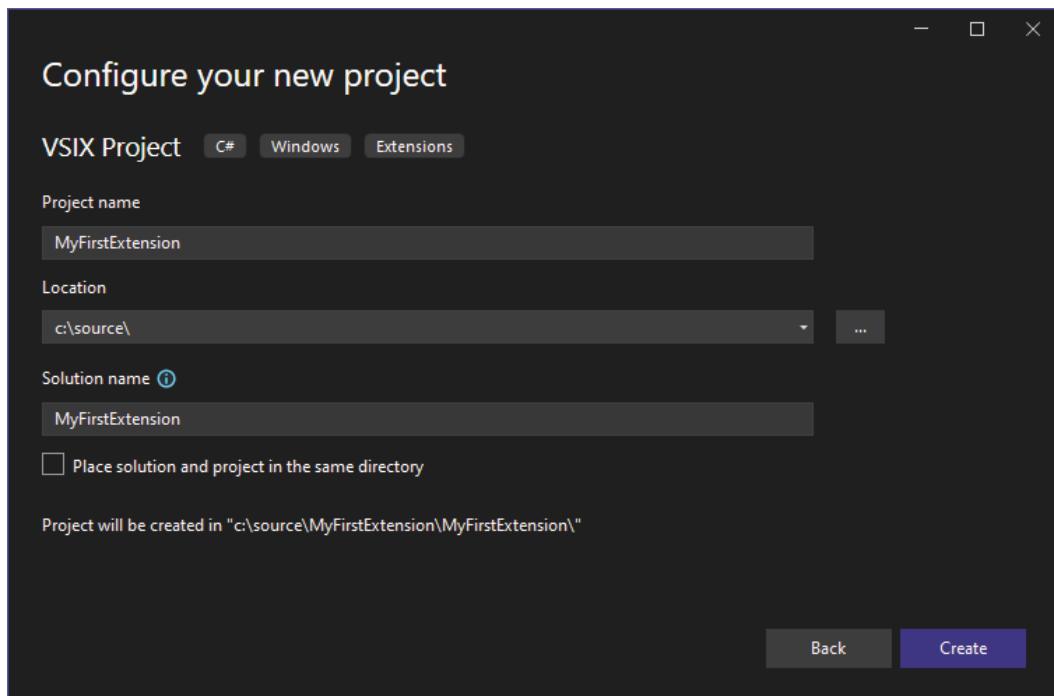


Figure 14.3 – Configure VSIX Project

4. After we've configured these settings, click on **Create** to generate the project structure. This step sets up the basic framework for our extension, allowing us to start customizing and adding functionality to our project.
5. The first step to creating a good extension is to add an entry point to our new feature. The most convenient way is to add a new command. Here, we will leverage the item template offered by the Visual Studio SDK by right-clicking on our project node, selecting **Add | New Item**, and then looking for **Command** under **C# Items | Extensibility**.

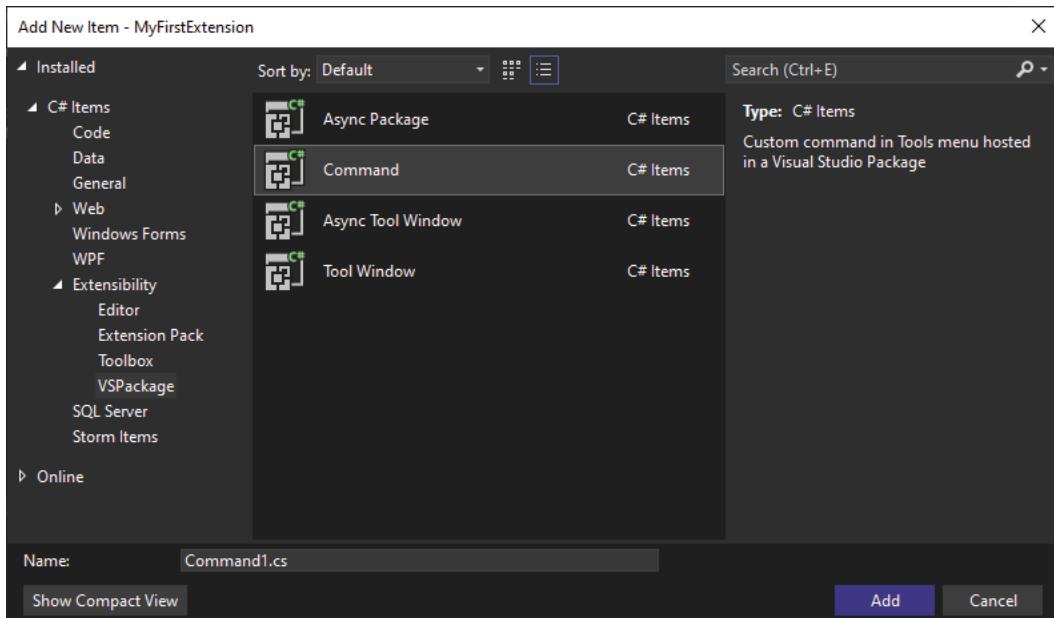


Figure 14.4 – Add New Item | Command

For this example, we choose the name `MyCustomTemplate`. This item template adds all we need to our first command:

- The `.vsct` file, short for **Visual Studio Command Table** file, is an XML file used in Visual Studio extensions to define the structure and behavior of menus, commands, and other UI elements that the extension adds to the Visual Studio environment. It acts as a blueprint for how the extension integrates with the Visual Studio UI, specifying details such as where menus and buttons should be placed, what icons they should use, and how they should behave when clicked.
  - The `MyCustomCommand.cs` file is added to your project; you'll need to replace its content with the appropriate code to define the desired functionality. For now, it's defining a command to display a message box when executed.
6. Now, when we are launching the debug mode, that will open an experiment instance of Visual Studio. In this instance, we will retrieve our installed package from the **Extensions | Manage Extension** window.

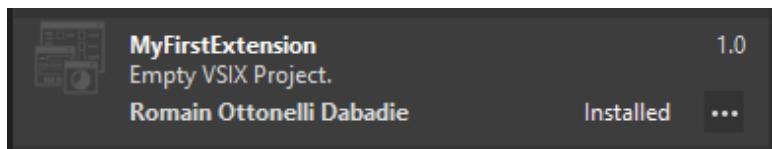


Figure 14.5 – MyFirstExtension installed

Plus, in the **Tools** menu, we find the **Invoke MyCustomCommand** option. For now, this command simply opens a message box displaying a basic message.

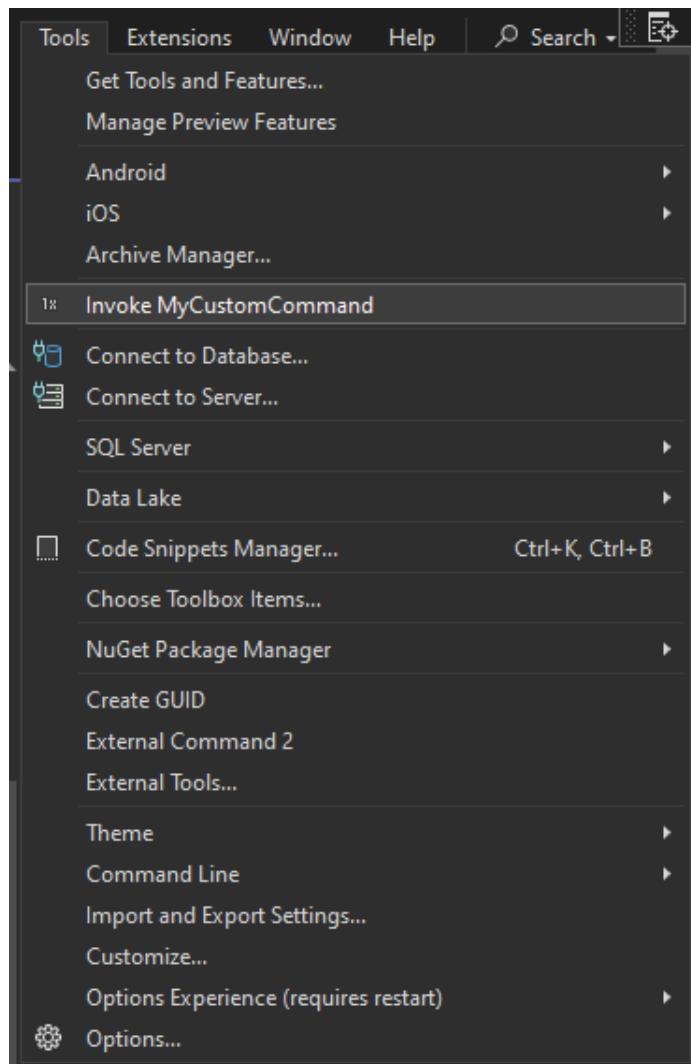


Figure 14.6 – Invoke MyCustomCommand

We have now built our first extension. Let's explore the features we can add to customize our Visual Studio extension in the next section.

## Advanced extension features

As we saw, we can easily add new commands to the menu of Visual Studio that allow us to interact with our IDE and even open external tools. In this section, we will explore the pre-set template we can leverage to extend Visual Studio.

So, if we go back to the **Add Item** window, by right-clicking on the project node and selecting **Add | New Items...**, you can find those kickstart ordered in four categories.

First, the **Editor** category encompasses templates that allow for the modification and enhancement of the Visual Studio editor itself. These modifications can range from changing how text is highlighted to adding interactive elements within the editor pane.

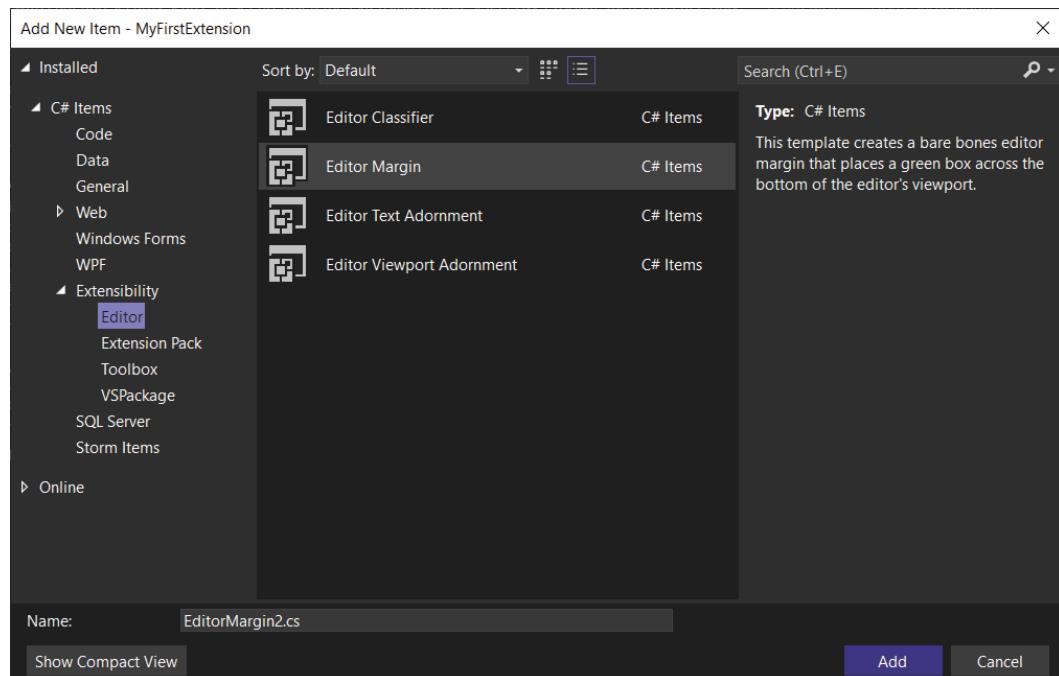


Figure 14.7 – Add a new Editor item

The following is an overview of the subcategories within the **Editor** category:

- **Editor Classifier:** An **Editor Classifier** item allows us to classify text within the editor based on custom rules. This can be used for syntax highlighting, identifying specific patterns in the code, or even providing real-time feedback on coding standards. For example, we can create a classifier that highlights all to-do comments in a bright color to ensure they catch the developer's attention.
- **Editor Margin:** The **Editor Margin** template enables the creation of custom margins around the **Editor** pane. These margins can display additional information or controls related to the code being viewed. A use case could be implementing a margin that shows a mini-map of the code file, allowing users to quickly navigate through large files.
- **Editor Text Adornment:** **Text adornments** allow for the decoration of text within the editor. This can include underlining, strikethroughs, background colors, or even inserting images or icons in line with the text. That allows us to add a red squiggly underline to deprecated methods to visually indicate they should not be used, for instance.
- **Editor Viewport Adornment:** Similar to text adornments but focused on the viewport (the visible area of the editor), **viewport adornments** can overlay graphics or UI elements on top of the editor content. For example, displaying a watermark or banner over the editor when viewing read-only files.

An **extension pack** is essentially a collection of extensions bundled together to provide a comprehensive set of functionalities around a specific theme or purpose. Creating an extension pack allows developers to distribute a suite of tools that work seamlessly together, enhancing productivity and streamlining workflows for specific development scenarios.

Then, the **Toolbox** category encompasses templates designed for creating controls that can be added to Windows Forms or **Windows Presentation Foundation (WPF)** design surfaces within Visual Studio. These controls can range from simple UI elements to complex components that encapsulate specific functionalities.

Furthermore, **VSPackages** represent a deeper level of extensibility within Visual Studio, allowing for the creation of extensions that can modify or add to the IDE's core functionalities. These packages can range from simple commands to complex tool windows and services.

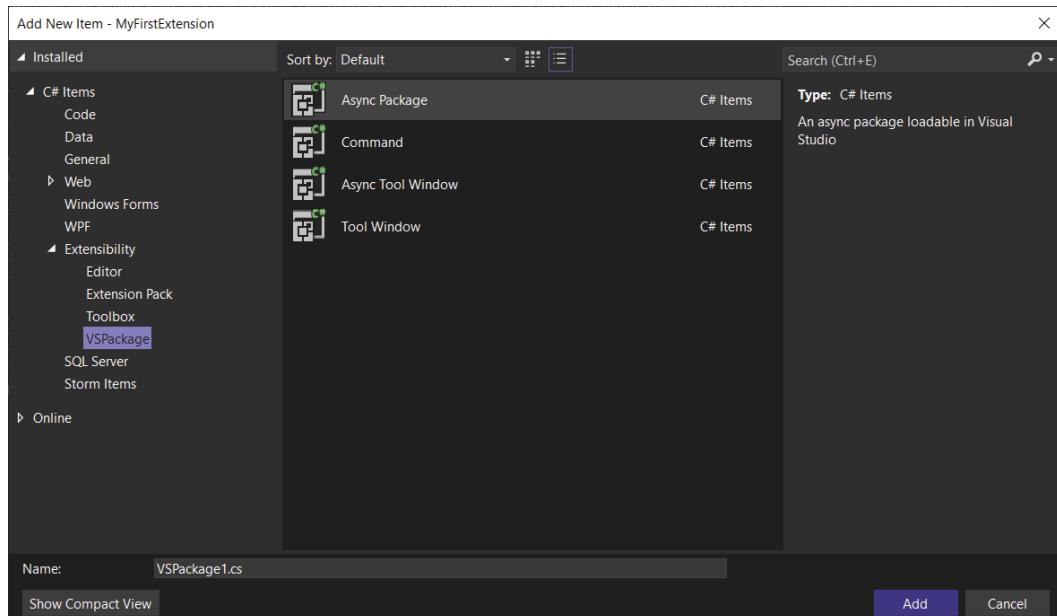


Figure 14.8 – Add new VSPackage Item

In this section, we can find the following items template:

- **Async Package:** This provides a foundation for creating asynchronous operations within Visual Studio extensions. This is particularly useful for operations that may take a significant amount of time to complete, such as fetching data from an external service or performing intensive computations.
- **Command:** As discussed in the previous section, commands allow for adding new actions to Visual Studio menus, toolbars, or even context menus. This template facilitates the creation of commands that can interact with the IDE or external tools, enabling a wide range of custom functionalities.
- **Async Tool Window:** This is like **Async Package** but specifically tailored to creating asynchronous tool windows within Visual Studio. These tool windows can perform background operations without blocking the UI, enhancing the responsiveness of the IDE.
- **Tool Window:** This template enables the creation of custom tool windows within Visual Studio. Tool windows can dock within the IDE and serve various purposes, such as displaying project properties, search results, or even custom controls and visualizations relevant to your workflow.

Additionally, you can find examples and samples in the official Visual Studio SDK repository on GitHub: <https://github.com/microsoft/VSSDK-Extensibility-Samples/tree/master>.

Remember that when developing extensions using these templates, like in other developments, it's important to consider the end-user experience, ensuring that the added functionalities integrate seamlessly with the existing Visual Studio environment. Testing and iteration are key to refining these extensions and delivering value to the developer community. Once our extension is tailored and well tested, the next step is to package and deploy it for general availability.

## Deploying and sharing your extensions

Deploying and sharing Visual Studio extensions with the community is an exciting step in the extension development process. It allows us to contribute to the ecosystem, potentially improving the development experience for thousands of Visual Studio users worldwide. In this section, we will walk through a comprehensive guide on deploying and sharing our extensions, including signing your VSIX (a .vsix file contains one or more Visual Studio extensions) packages for added security and trustworthiness.

Once we ensure it is thoroughly tested and ready for public release, we can start the deploying phase. This includes testing across different versions of Visual Studio, handling edge cases, and ensuring compatibility with various project types and configurations.

First, in order to prevent tampering and build trust with the end user, we sign our VSIX with **Sign CLI**. You need to purchase a code signing certificate from a trusted **certificate authority (CA)**. I used the .NET CLI to install Sign CLI. Notice that at the time of writing, Sign CLI is still pre-released; so, we need to include the `-prerelease` tag when we need to get the latest:

```
dotnet tool install sign --prerelease -global
```

Once installed, Sign CLI could be accessed in a Developer PowerShell instance using `sign code <command> <option>`. You can find more details in the official documentation: <https://learn.microsoft.com/en-us/visualstudio/extensibility/dotnet-sign-cli-reference-vsix?view=vs-2022>.

After signing your extension, the next step is deploying it to Visual Studio Marketplace, where it can be accessed by Visual Studio users worldwide. The process involves several key steps to ensure your extension is properly showcased and ready for installation:

1. First, create a publisher account on Visual Studio Marketplace.
2. Sign in with your Microsoft account and follow the prompts to set up your publisher profile if you haven't done so already. This account will allow you to manage and track your extensions on the platform.
3. Next, create a new extension by navigating to the **Publishers** dashboard and selecting the **New Extension** option (here is the link to the Visual Studio Marketplace dashboard: <https://marketplace.visualstudio.com/manage/createpublisher?managePageRedirect=true>). Here, you'll need to provide essential details about your extension, including the title, description, and relevant visuals, such as an icon and screenshots. These elements help potential users understand what your extension offers.

4. Once we've filled out the details, upload your signed VSIX package. Visual Studio Marketplace will automatically validate the package and its digital signature, ensuring everything is in order before proceeding.
5. After uploading, submit your extension for review. Microsoft will carefully evaluate your submission to ensure it adheres to the Marketplace's policies and guidelines.

This review process is crucial for maintaining the quality and security of the extensions available to users.

Finally, upon approval, your extension will be published on Visual Studio Marketplace. It will then be available for installation directly through the Visual Studio IDE or as a download from the Visual Studio Marketplace website, allowing developers worldwide to benefit from your contribution.

## Summary

As we conclude this chapter on Visual Studio extension development, you've acquired the knowledge and skills to create, refine, and distribute your own powerful extensions. From understanding the core architecture to building your first extension and exploring advanced features, you've learned how to enhance Visual Studio to better meet your needs and those of your team. By mastering the deployment and sharing process, you're now equipped to contribute to the broader developer community, offering tools that can streamline workflows and boost productivity.

With this solid foundation in place, we're now ready to transition to the final chapter of the book. In this concluding chapter, we'll delve into the world of NuGet package creation, where you'll learn how to package and distribute reusable code libraries.

# 15

## Creating and Publishing Powerful NuGet Packages for the Community

In this chapter, we'll embark on a comprehensive journey into the world of NuGet and package management, an essential aspect of modern .NET development. Whether you're a seasoned developer or not, understanding NuGet will elevate your ability to manage dependencies and streamline your workflow.

We'll begin by introducing the fundamentals of NuGet, covering what it is and why it's a cornerstone in the .NET ecosystem. You'll gain practical insights into creating your first NuGet package, including hands-on experience with setting up a project, packaging your code, and generating a `.nupkg` file. As we progress, we'll master versioning and dependency management, which are crucial skills to ensure that your packages are compatible and up to date. Then, we'll explore the process of publishing and distributing your packages, ensuring that our work reaches the right audience. Finally, we'll dive into advanced features and best practices, from targeting specific platforms to incorporating pre-release versions.

In this chapter, we're going to cover the following main topics:

- Introducing NuGet and package management
- Creating your first NuGet package
- Versioning and dependency management
- Publishing and distribution
- Advanced NuGet features

By the end of this chapter, you'll be equipped with the knowledge and skills to create, manage, and distribute NuGet packages effectively, empowering you to contribute to the broader .NET community with confidence.

## Technical requirements

While writing this chapter, I used the following versions of Visual Studio:

- Visual Studio Enterprise 2022 Version 17.12.0
- Preview 1.0

The code files for this chapter can be found at <https://github.com/PacktPublishing/Mastering-Visual-Studio-2022/tree/main/ch15>.

## Introduction to NuGet and package management

In this opening section, we'll lay the groundwork for everything to come. We'll explore what NuGet is, its importance in the .NET ecosystem, and how it revolutionizes the way developers share and reuse code.

NuGet is a free and open source package manager designed specifically for the Microsoft development platform, including .NET. At its core, NuGet serves as a mechanism for sharing code and speeding up the development process. It allows developers to create packages that encapsulate functionality, libraries, or frameworks and distribute them through NuGet repositories.

NuGet's importance lies in its ability to simplify the process of incorporating third-party libraries into your projects, manage dependencies, and keep them up to date. This approach significantly reduces the complexity associated with manual dependency management, which was once a common challenge in .NET development.

The NuGet ecosystem is vast and dynamic. At its center is the official NuGet Gallery ([nuget.org](https://nuget.org)), which hosts thousands of packages contributed by the community. However, NuGet isn't limited to public repositories; it also supports private feeds, allowing organizations to control access to internal packages.

One of the most significant advantages of NuGet is its ability to simplify dependency management. With NuGet, developers no longer need to manually download and install libraries. Instead, they can simply add a reference to the desired package in their project, and NuGet takes care of downloading and installing the correct version.

To effectively create and consume packages, it's crucial to set up your development environment properly. We can access the general configuration of the package manager through the top-level toolbar – **Tools | NuGet Package Manager | Package Manager Settings**.

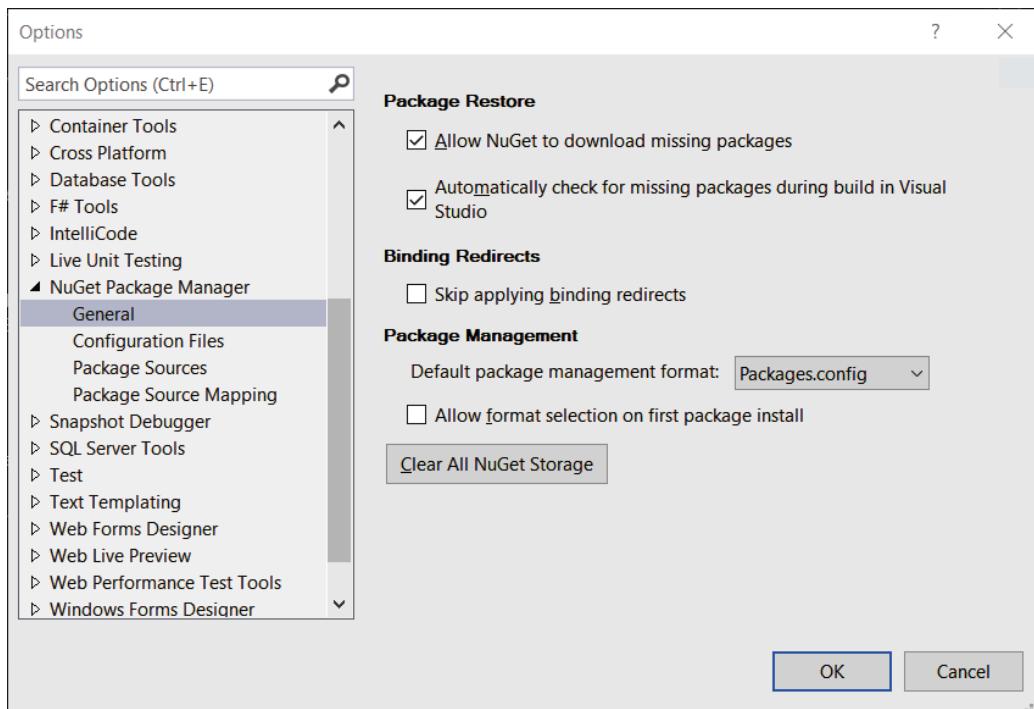


Figure 15.1 – The NuGet package manager options

Here are some details about the options available:

- **Package Restore:** Checking both the boxes under this allows us to ensure that NuGet packages are automatically restored when building a solution. This is important because it guarantees that all necessary packages are available, even if they were not included in the source control, thus avoiding missing package issues during build processes.
- **Binding Redirects:** This is a mechanism used to address version conflicts between assemblies. When you're working on a .NET project that references multiple libraries, it's common to encounter situations where different libraries depend on different versions of the same assembly. This can lead to runtime errors due to version conflicts. You can choose to skip those libraries, but it's particularly useful in scenarios where you cannot control the version dependencies of third-party libraries but need to ensure compatibility across your application.
- **Package Management:** Here, we can choose the default format of the package management file. We can also opt to select the format when installing the first package.

- **Clear All NuGet Storage:** The global NuGet cache stores copies of packages that we have downloaded or installed. Clearing this cache ensures that our development environment starts fresh, without any potentially corrupted or outdated packages. This helps to troubleshoot package-related issues and ensures that we work with clean, up-to-date packages. One other way to achieve this is by opening a Command Prompt as administrator and running the `nuget locals all -clear` command.

Now that we've established the basics of NuGet and its significance, let's move on to the practical side of things. It's time to roll up our sleeves and start creating our very own NuGet package.

## Creating your first NuGet package

Here, we will dive straight into action. We'll walk through the process of creating a simple NuGet package from scratch, covering every step from initial setup to publishing. This hands-on experience will help reinforce the concepts introduced in the previous section and prepare you for more advanced package creation.

Let's break this down further. First, we'll create a new .NET Standard library project in Visual Studio. This type of project is ideal for creating NuGet packages because it's compatible with multiple .NET frameworks. Let's get started with the steps:

1. Open Visual Studio and select **New Project**.
2. For this example, choose **.NET Standard** under the **C#** section.
3. Name the project (e.g., `MyFirstPackage`) and click **OK**.

Next, we'll write the package content by adding some sample code to our project. For this example, let's create a simple utility class that converts temperatures between Celsius and Fahrenheit:

```
using System;

namespace MyFirstPackage
{
    public static class TemperatureConverter
    {
        public static double ConvertCelsiusToFahrenheit(
            double celsius)
        {
            return (celsius * 9 / 5) + 32;
        }

        public static double ConvertFahrenheitToCelsius(
            double fahrenheit)
        {
```

```
        return (fahrenheit - 32) * 5 / 9;
    }
}
```

Now that we have our code, we need to prepare it for packaging:

1. Right-click on the project in the **Solution Explorer** window.
2. Select **Add | New Folder**.
3. Name the folder `lib`.
4. Move the `TemperatureConverter.cs` file into this folder.

With our project prepared, we can now create the NuGet package:

1. Right-click on the project in the **Solution Explorer** window.
2. Select **Pack** from the context menu.

This will generate a `.nupkg` file in the `bin/Debug` folder.

Let's verify that our package was created successfully:

1. Locate the `.nupkg` file in the `bin/Debug` folder
2. Double-click the file to view its contents.
3. Check that the package includes the `lib` folder containing our `TemperatureConverter.cs` file.

Now that we've created our first NuGet package, let's take a moment to reflect on what we've accomplished. We've gone from having no package to having a fully functional one that we could potentially share with others. Now that we have done that, there's an important aspect we need to address – how do we ensure that our package remains stable and doesn't cause issues in other people's projects?

## Versioning and dependency management

Versioning is crucial for maintaining compatibility and responsibly managing updates. Properly handling dependencies between packages is equally important, as it ensures that your package works seamlessly with other packages in a project.

Versioning is the process of assigning unique identifiers to different versions of our package. It's a way to communicate to users how much your package has changed since the last release. There are two main approaches to versioning:

- **Semantic versioning:**
  - **Major version:** Breaking changes
  - **Minor version:** New features

- **Patch version:** Bug fixes

An example is 1.0.0, 1.1.0, 1.2.0, and so on

- **Date-based versioning:**

- Uses dates to indicate when a package was released

An example is 2023.01.01.0

While date-based versioning is simpler, semantic versioning is generally preferred, as it provides more meaningful information about a package's history.

We can easily define the version of our package by navigating to the properties of our project:

1. Right-click on the project in the **Solution Explorer** window,
2. Select **Properties**.
3. Go to the **Package** tab.

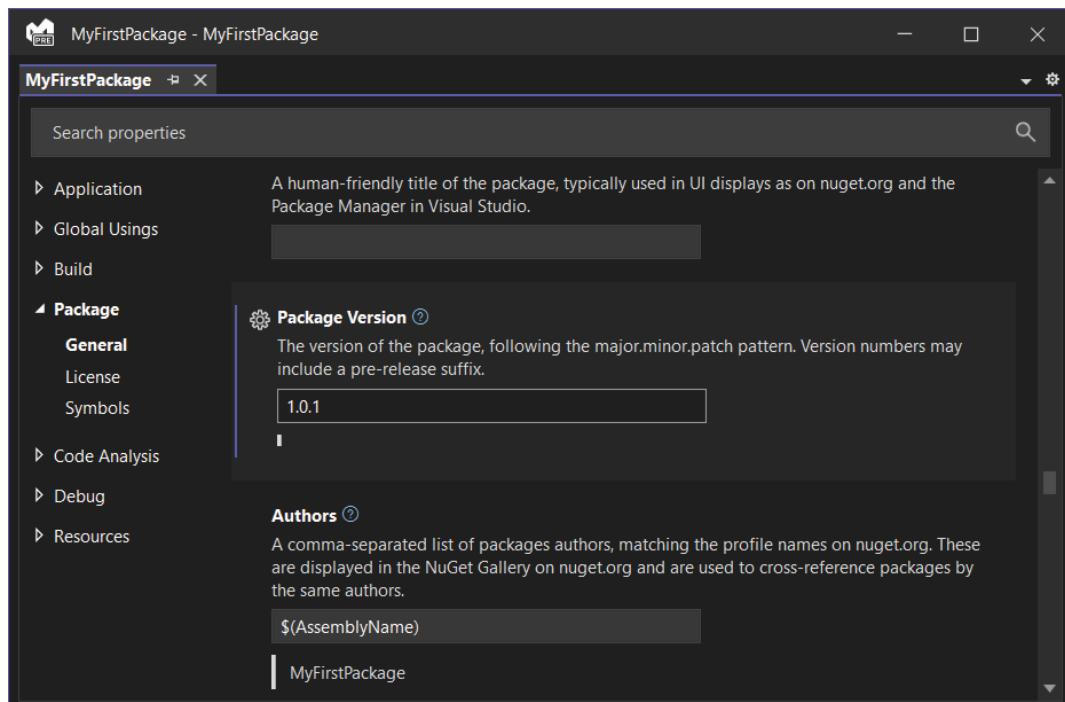


Figure 15.2 – Package Properties

Here, we define version 1.0.1 of our package; if we pack the project again, it will generate a new .nupkg file with the newly defined version.

Dependency management involves specifying which other packages your package depends on. This is crucial for ensuring that our package functions correctly in various environments.

There are two ways to declare dependencies in a NuGet package:

- **Direct dependencies:**

- Specify the exact versions of packages you depend on. Here's an example:

```
<dependencies>
  <group targetFramework="netstandard2.0">
    <dependency
      id="System.Collections"
      version="4.3.0"
    />
  </group>
</dependencies>
```

- **Range dependencies:**

- Specify a range of acceptable versions. Here's an example:

```
<dependencies>
  <group targetFramework="netstandard2.0">
    <dependency
      id="Newtonsoft.Json"
      version="[12.0,13.0)"
    />
  </group>
</dependencies>
```

Range dependencies allow for flexibility while still maintaining some level of control over the versions used.

Effective versioning and dependency management are essential practices in software development. It is crucial to maintain consistency by adhering to a chosen versioning strategy throughout the life cycle of a package.

Clearly communicating this strategy, especially through documentation such as the `README.md` file, helps ensure that all stakeholders understand the approach. Keeping dependencies up to date is equally important, as regular updates to the packages you rely on can prevent compatibility issues and enhance security.

Thorough testing against different versions of dependencies is necessary to identify potential conflicts early. Additionally, it is important to consider transitive dependencies – those indirect dependencies that may influence your package's behavior – to maintain overall stability and reliability.

Now that we've covered the essentials of versioning and dependency management, let's turn our attention to the next crucial step in getting our package out to the world – publishing and distribution. This is where we make our hard work available to others and potentially become part of the rich ecosystem of .NET packages.

## Publishing and distribution

Publishing your NuGet package makes it accessible to the entire .NET community or your entire organization. This section guides you through the process of publishing your package to the official NuGet gallery, as well as alternative distribution methods.

Let's explore in detail how to publish to the official NuGet Gallery:

1. Sign in to your account at <https://www.nuget.org/>.
2. Click on **Create a new package** and fill in the required details.
3. Upload the .nupkg file.
4. Fill in the package metadata (description, tags, etc.) .
5. Review and submit the package.

While the official gallery is the most widely recognized method for distributing packages, there are scenarios where alternative distribution methods may be more appropriate. For instance, private feeds are particularly useful for company-wide or team-specific packages. These feeds can be hosted on platforms such as Azure DevOps, GitHub, or even self-hosted servers, providing a secure and controlled environment for package distribution within an organization. Local feeds are another option, ideal for testing and development environments. These can be set up directly on your local machine or integrated within a CI/CD pipeline, offering a convenient way to manage packages during the development process. For those requiring even more control, custom servers allow for fine-grained management of package distribution, although they do come with the added responsibility of hosting infrastructure and ongoing maintenance.

Private repositories offer a secure and controlled environment for hosting NuGet packages that are intended for internal use within an organization, or among a select group of developers. Unlike public repositories such as nuget.org, which are accessible to anyone, private repositories restrict access to authorized users only. This is particularly useful for proprietary software components, beta versions, or when you need to comply with specific security policies.

**JFrog Artifactory** is one of the most popular tools for managing binary repositories, including NuGet packages. It supports both public and private NuGet feeds, making it an excellent choice for organizations looking to establish their own NuGet repository infrastructure.

Here are the details on how to set up a private NuGet Feed in Artifactory:

1. **Install Artifactory:** First, you'll need to install JFrog Artifactory on your server or cloud environment. Follow the official installation guide provided by JFrog.

2. **Create a new repository:** Once Artifactory is up and running, log in to the administration console. Navigate to the **Repositories** section and choose to create a new repository. Select **NuGet** as the package type.
3. **Configure the repository settings:** Specify the repository key, layout, and other settings according to your requirements. For a private feed, ensure that the repository is not publicly accessible.
4. **Set up access control:** Define who has access to your new NuGet repository. You can create groups for different teams or projects and assign appropriate permissions.
5. **Publish packages:** With your repository set up, you can now publish NuGet packages to it. Use the `nuget push` command or integrate with your CI/CD pipeline to automate the process.

When working with private NuGet feeds, such as Artifactory and others, we must set up a NuGet source in Visual Studio that allows us to specify where NuGet packages should be retrieved from during the package restore process, or when adding new packages to our projects.

Here's how we can set up a NuGet source through Visual Studio:

1. Go to **Tools | NuGet Package Manager | Package Manager Settings**.
2. Select **Package Sources** from the left navigation menu.

To add a new package source, click on the green plus (+) button at the top-right corner of the **Package sources** window.

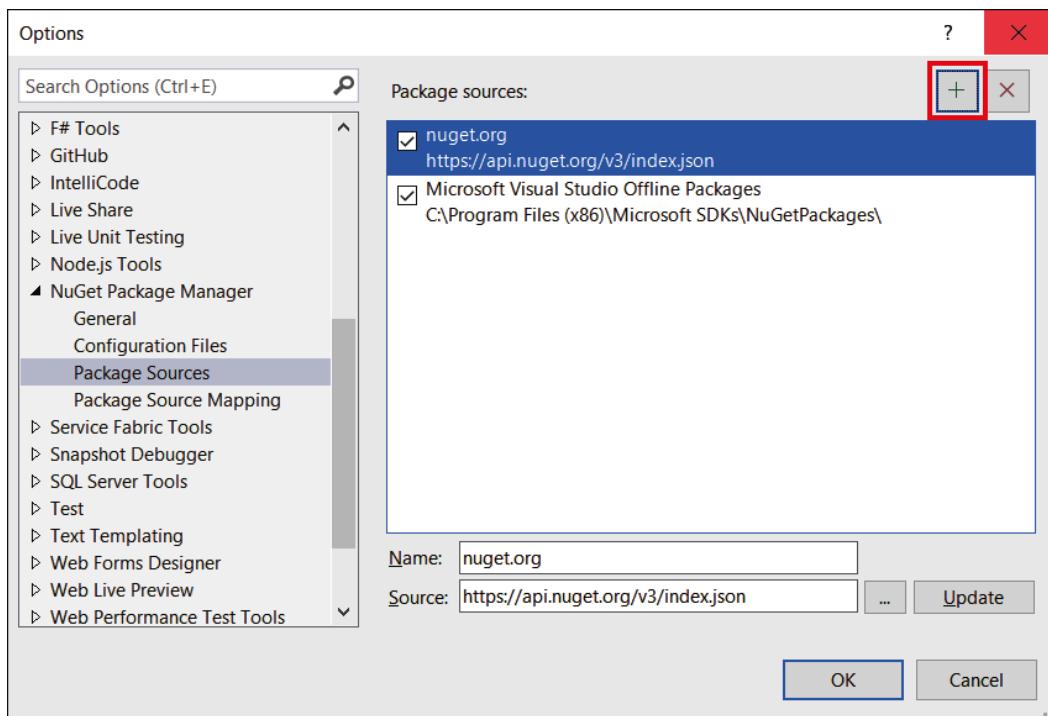


Figure 15.3 – Adding a new package source

3. Fill in the **Name** and **Source** textboxes to fit the desired NuGet feeds to be added.

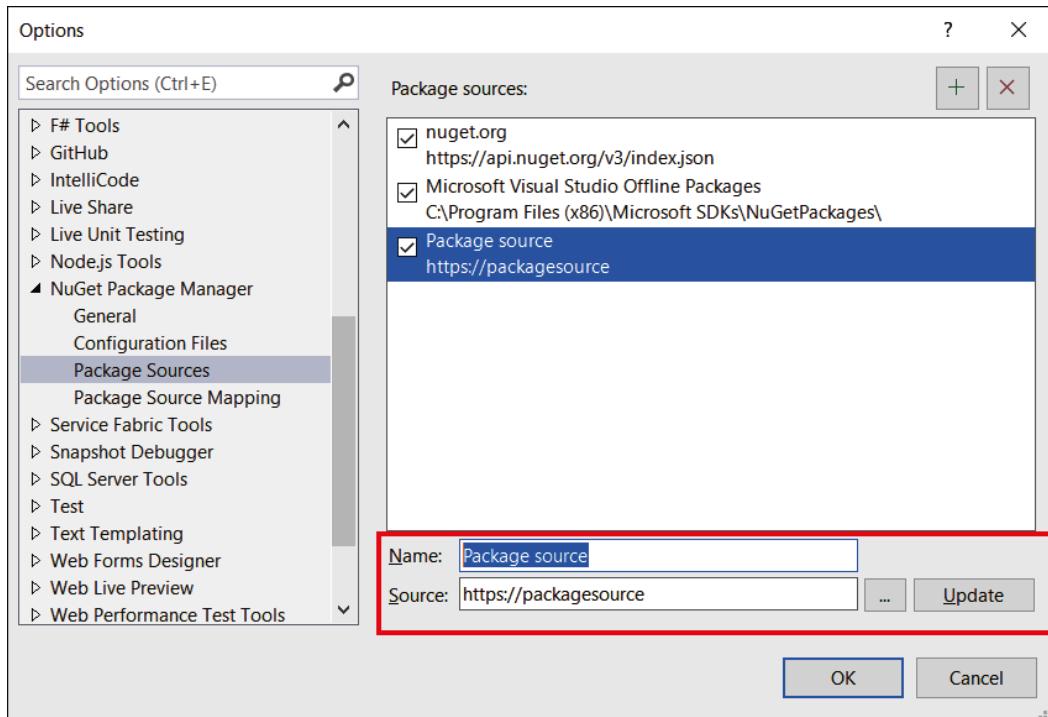


Figure 15.4 – Configuring a new package source

4. Then, validate the source by clicking on the **Update** button.

After adding and configuring your package source, you can verify that it's working correctly by attempting to restore packages or searching for packages within Visual Studio. Go to **Tools | NuGet Package Manager | Manage NuGet Packages for Solution...**, and then you can select the package source you want to use with the selection box in the top-right corner.

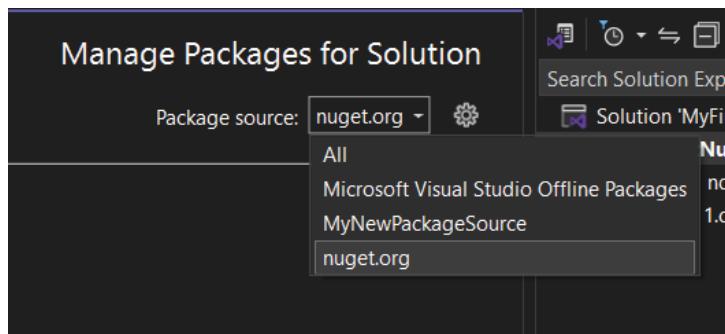


Figure 15.5 – Choosing the package source

As we conclude our journey through the process of creating and publishing a NuGet package, it's worth taking a moment to appreciate the power and flexibility of NuGet. From its humble beginnings as a simple package manager to its current status as a cornerstone of .NET development, NuGet continues to evolve and improve.

## Advanced NuGet features

In this final section, we'll explore some of the more advanced features of NuGet, such as target frameworks, pre-release versions, and custom scripts, and discuss the best practices for package development. This knowledge will help you create more robust, flexible, and maintainable packages.

### Target frameworks

Target frameworks allow us to specify which .NET platforms your package supports, ensuring compatibility across various environments. This feature helps prevent compatibility issues by allowing only the installation of your package in projects that target supported frameworks.

When creating a NuGet package, you can define multiple target frameworks within the `.csproj` or `.nuspec` file. This setup enables NuGet to pack separate binaries for each framework, ensuring that your package leverages platform-specific features when available.

Here's an example of configuration:

```
<PropertyGroup>
  <TargetFrameworks>net46;netstandard2.0</TargetFrameworks>
</PropertyGroup>
```

This configuration indicates that the project targets both .NET Framework 4.6 and .NET Standard 2.0, allowing optimal compatibility and functionality across different environments.

## Pre-release versions

Pre-release versions are a way to distribute packages that are not yet ready for production use. These versions are typically used for beta testing, release candidates, or nightly builds.

To allow package authors to distribute versions of their packages that are not considered stable or final, consumers can choose to install these pre-release versions to test new features or bug fixes early.

Pre-release versions are denoted by appending a suffix to the version number, following a hyphen. Common suffixes include `-alpha`, `-beta`, `-rc` (**release candidate**), and `-preview`. Consumers need to explicitly opt in to receive pre-release packages, usually through a setting in their package manager client.

Here's an example of configuring a `-beta` version:

```
<Version>1.0.0-beta1</Version>
```

This indicates that the package is a beta version of 1.0.0, suitable for testing but not recommended for production use.

## Custom scripts

NuGet supports the execution of PowerShell scripts during the installation or uninstallation of a package. This feature can be used for various purposes, such as modifying project files, registering **Component Object Model (COM)** components, or performing other setup tasks.

Custom scripts automate tasks that need to occur when a package is installed or uninstalled, reducing manual steps for the consumer and ensuring consistency.

As Package authors, we can include PowerShell scripts in our packages that NuGet will execute at appropriate times. There are several types of scripts, including `init.ps1` (run once per solution), `install.ps1` (run every time the package is installed), and `uninstall.ps1` (run when the package is uninstalled):

```
<packageTargetFallback  
    Condition="'$(OS)' == 'Windows_NT'">  
    win7-x64-msvc2015  
</packageTargetFallback>  
<script  
    src="tools/precompile.js"  
    condition="$( [MSBuild]::GetTargetPath(  
        '$(ProjectDir)tools\precompile.js')) != ''"  
/>
```

This configuration specifies a fallback target framework and includes a custom script (`precompile.js`) that will be executed under certain conditions. Note that custom scripts are less commonly used today, especially with the evolution of .NET Core and .NET Standard, but they remain supported for backward compatibility.

By leveraging these advanced features of NuGet, we can create more sophisticated and flexible packages that better meet the needs of the users across different environments and stages of development.

## Summary

In this chapter, we explored the essential aspects of NuGet and package management within the .NET ecosystem. We began by understanding what NuGet is and why it's a critical tool for managing dependencies in modern development. We learned how to create our first NuGet package, including setting up a project, writing the necessary code, and packaging it into a `.nupkg` file.

We then delved into versioning and dependency management, highlighting the importance of semantic versioning and strategies to maintain compatibility across different versions. The chapter also covered the steps involved in publishing and distributing NuGet packages, both to the official NuGet gallery and through alternative methods, ensuring that your packages are accessible to others.

Finally, we explored advanced features and best practices, such as targeting specific platforms, using pre-release versions, and incorporating custom scripts. Now, you have gained a comprehensive understanding of how to create, manage, and distribute NuGet packages, equipping you with the tools to enhance your .NET development practices.

As we reach the conclusion of *Mastering Visual Studio*, we've journeyed through a wide array of advanced techniques, tools, and best practices that are essential for any serious developer working within the .NET and Visual Studio ecosystem. This book has equipped you with powerful skills and knowledge to become a more efficient, versatile, and innovative developer. Whether you're building intelligent applications, optimizing performance, or contributing to the developer community, the skills you've gained will continue to drive your success in the world of software development.



# Index

## Symbols

- .editorconfig file
  - generating 70-72
- .http files
  - generating, with Endpoints Explorer 138-142
- .NET 5+ 43
- .NET 6 41
- .NET applications
  - instrumenting 95, 96
- .NET Aspire
  - cloud-specific framework 168
  - component-based design 168
  - enhanced developer tools 168
  - exploring 168-173
  - streamlined development workflow 168
- .NET asynchronous events
  - analyzing 90, 91
- .NET Async tool 91
- .NET Compiler Platform-based
  - code analyzers 62
- .NET Core 3.x 43
- .NET Counters 91
  - monitoring with 92

- .NET Object Allocation tool 93
- .NET Upgrade Assistant 131

## A

- Agile development
  - aspects 222, 223
  - practices, implementing 221
  - Work Items, managing through Visual Studio 223-227
- ahead-of-time (AOT) 118
- all-in-one search feature 12
- Amazon Elastic Compute Cloud (EC2) 183
- Amazon Elastic Container Registry (ECR) 242
- Amazon Q 184
- Amazon Simple Storage Service (S3) 183
- Analyzer with Code Fix template 65
- Android 129
- API Explorer
  - using, with Visual Studio 138
- API Usage Examples 75
- applications
  - Dockerizing, with Visual Studio 237, 238

**Arrange-Act-Assert (AAA) pattern** 8  
**artificial intelligence (AI)** 152  
**ASP.NET Core web API**  
    used, for deploying ML.NET model 158-161  
**auto-decompilation**  
    controlling 52  
    limitations 52  
    usage 52  
**automated ML (AutoML)** 153  
**Autos window** 36  
**AWS Explorer** 184  
**AWS Toolkit**  
    exploring 182-186  
    features 184  
**Azure**  
    containerized applications, deploying as service 243, 244  
**Azure App Service** 244  
**Azure Artifacts** 218  
**Azure Boards** 218  
**Azure Container Apps** 244  
**Azure Container Registry (ACR)** 242, 244  
**Azure DevOps** 218  
    components 218  
    team projects, setting up 219-221  
**Azure Functions** 161  
    used, for deploying machine learning (ML) model 161-164  
**Azure Functions development**  
    exploring in Visual Studio 173-177  
**Azure Pipelines** 218  
    integrating, for continuous integration 227-229  
**Azure Repos** 218  
**Azure Test Plans** 218

## B

**Big O notations**  
    constant time  $O(1)$  89  
    cubic time  $O(n^3)$  89  
    exponential time  $O(2^n)$  89  
    factorial time  $O(n!)$  89  
    linear time  $O(n)$  89  
    logarithmic time  $O(\log n)$  89  
    log-linear time  $O(n \log n)$  89  
    quadratic time  $O(n^2)$  89

**bottlenecks** 88

**breakpoints**  
    organizing 48  
    types 42

**breakpoints, types**  
    conditional breakpoints 42  
    data breakpoints 43, 44  
    dependant breakpoints 45, 46  
    function breakpoints 45  
    temporary breakpoints 47  
    tracepoint 42, 43

**bugs, on fly**  
    fixing, with Edit and Continue feature 40  
    fixing, with Hot Reload feature 41

**Builds feature** 227-229

## C

**Call Stack window** 38  
    functionalities 39  
**certificate authority (CA)** 275  
**Checkout (--detach) option** 194  
**client-side scripts**  
    debugging 148-150

- 
- code analysis**
    - legacy analysis (FxCop static analysis) 62
    - .NET Compiler Platform-based
      - code analyzers 62
  - CodeMap extension** 39
  - code metrics**
    - case coupling 77
    - Cyclomatic Complexity 77
    - depth of inheritance 77
    - lines of source code 77
    - maintainability Index 77
    - using, in Visual Studio 2022 77-79
  - code quality analysis feature** 65
  - code refactoring**
    - IntelliCode, leveraging 73, 74
  - Code Refactoring template** 65
  - command-line interface (CLI)**
    - commands 232
  - Command-Query Responsibility Segregation (CQRS) pattern** 109
  - common language runtime (CLR)** 256
  - compiler API**
    - Bind Phase 64
    - Declaration Phase 63
    - Emit Phase 64
    - Parse Phase 63
  - Component Object Model (COM)**
    - components 288
  - concurrency debugging** 54-58
  - conditional breakpoint** 42
  - Consume panel**
    - console app 159
    - web API 159
  - containerized applications**
    - deploying 240
    - deploying, as service in Azure 243, 244
    - deploying, in container registry 240-242
  - container registry** 240
  - containers** 232
  - container toolbox**
    - Attach debugger 239
    - Open terminal 239
    - Remove container 239
    - Start/stop 239
  - continuous deployment (CD)**
    - GitHub Actions, using for 208
  - continuous integration**
    - Azure Pipelines, integrating for 227-229
  - CPU usage**
    - analyzing 96-102
    - tool, using 96-102
  - cross-platform development**
    - evolution 116
  - custom Event Tracing**
    - for Windows (ETW) events
    - enabling 94

## D

- database interactions**
  - optimizing 107-111
- Database Profiler tool** 95, 107
- data breakpoint** 43, 44
- data inspections** 49
  - properties, pinning 50, 51
  - value, editing 51, 52
- DataTips** 49
- date-based versioning** 282
- debugger cursor, Visual Studio 2022** 34
  - Force Run to Click feature 35
  - Run to Cursor feature 34, 35
- dependant breakpoint** 45, 46
- dependency management** 283
- Designer web toolbox** 138

**Developer Mode**

- enabling, on Android 128
- enabling, on iOS 128
- enabling, on Windows 128

**devices**

- debugging 128
- Developer Mode, enabling 128
- networking device 129

**Dev Tunnel**

- configuring 142-144
- use cases 142
- using, with Visual Studio 138

**diagnostic tools 105****direct dependencies 283****Docker 232****Docker client 232****Docker daemon 232****Docker environments**

- setting up, in Visual Studio 233

**Dockerfile 233****Docker host**

- containers 232
- Docker daemon 232
- images 232
- networks 233
- storage 233

**Docker Hub 242****Docker images 233****Docker Registry 233****Docker support**

- adding, to existing project 235-237
- project, creating 234

**Duplication Factor 94****dynamic instrumentation 96****Dynamic Link Library (DLL) 266****E****Edit and Continue feature 40****Endpoints Explorer**

- used, for generating .http files 138-142

**Event Viewer tool 94****extension manifest 266****extension pack 273****External Sources feature 53, 54**

- usage 54

**F****File I/O tool 94****file-scoped namespace feature 83****F.I.R.S.T principles**

- of unit testing 4, 5

**flame graph 101****Force Run to Click feature 35****function breakpoint 45****G****GitHub Actions**

- actions 209
- events 209
- for continuous deployment (CD) 208
- jobs 209
- runners 209
- workflows 209
- workflows, configuring 209-212

**GitHub Actions file**

- generating, with Visual Studio 213-216

**GitKraken 190****Google Cloud Platform (GCP) 177****Google Cloud Tools**

- exploring, for Visual Studio 2022 177-182

**Google Container Registry (GCR)** 242  
**Google Kubernetes Engine (GKE)**  
 environments 180

## H

**Hot Reload feature** 41, 128  
**Hot Restart** 129

## I

**ILspy** 52  
**Immediate window** 39, 40  
**input/output (I/O)-bound operations** 91  
**instrumentation** 95  
 dynamic instrumentation 96  
 static instrumentation 95  
**integrated development environment (IDE)** 11, 32, 138  
**integration points** 266  
**IntelliCode**  
 code, predicting with whole-line  
 autocompletion 74, 75  
 GitHub documentation, accessing 75-77  
 leveraging, for code refactoring 73, 74  
**IntelliTest**  
 execution and test generation 17, 18  
 test, creating with 15, 16  
 unit tests, executing 18, 19  
 unit tests, reviewing 18, 19  
 unit tests, saving 18, 19  
**interactive staging**  
 in Visual Studio 200-204  
**intermediate language (IL)** 64, 118  
**Internet Information Services (IIS)** 180  
**Internet of Things (IoT)** 177  
**item templates** 250

## J

**JavaScript applications**  
 debugging 148  
**JavaScript Project System (JSPS)** 144  
**JavaScript project templates**  
 exploring 144-146  
**JFrog Artifactory**  
 private NuGet Feed, setting up 284

## L

**Language-Integrated Query (LINQ)** 88  
**large object heap (LOH)** 93  
**legacy analysis (FxCop static analysis)** 62  
**Lines of Executable Code** 77  
**Live Unit Test**  
 configuring 25-27  
 launching 27-30  
 tests, automating with 24, 25  
**Live Visual Tree** 126  
 feature 126  
 top menu bar, exploring 127, 128  
**Locals window** 37

## M

**machine learning (ML)** 151, 152  
 reinforcement learning 152  
 supervised learning 152  
 unsupervised learning 152  
**machine learning (ML) model**  
 creating, with ML.NET 153-158  
 creating, with Model Builder UI 153-158  
 deploying, in Azure Functions 161-164  
**Manage Branches window** 190-194  
**memory leak** 102

**memory usage**

exploring, while debugging 104-106

**Memory Usage tool**

using 103, 104

**ML.NET 151, 153**

features 152

used, for creating machine learning (ML) model 153-158

**ML.NET CLI 153****ML.NET model**

deploying, in ASP.NET Core

web API 158-161

**Model Builder 151, 153****Model Builder UI**

used, for creating machine learning (ML) model 153-158

**Model Builder UI, scenarios**

Computer Vision 156

Natural Language Processing 156

Tabular 155

**MSTest 9****Multi-Platform App UI (MAUI) 115, 116**

architecture 117, 118

cross-platform development, evolution 116

key features 116

**Multi-Platform App UI (MAUI) tools**

exploring 118

Live Visual Tree 126

MAUI app, creating 118-123

XAML Live Preview 123-126

**multiple repositories**

handling 195-197

**multithreading 54****N****networking device 129**

Android 129

debugging session, launching 130

Hot Restart 129

Hot Restart for iOS 129, 130

**Node.js integration**

with Visual Studio 144

**No Side Effects (nse) 40****npm packages**

managing 146-148

**NuGet 278****NuGet, advanced features**

custom scripts 288

pre-release versions 288

target frameworks 287

**NuGet Gallery**

URL 278

**NuGet package 52**

manager, options 278-280

creating 280, 281

dependencies, declaring 283

publishing 284

version, defining 282

**NuGet source**

setting up, through Visual Studio 285, 286

**NUnit 9****O****Object Relational Mapping (ORMs) 111****P****package component 266****performance optimization 88, 89****Performance Profiler 90****pinned object heap (POH) 93****Playwright 10****pre-release versions 288****private NuGet Feed**

setting up, in Artifactory 284

**project**

creating, with Docker support 234

**project system integration 266****project system object model (PSOM) 266****project template**

building 252-255

customizing, for different workflows 255, 256

extending, with advanced features 262-264

parameters and variables, integrating 256-261

structure 250, 251

**R****range dependencies 283****real-time web previews**

powered by, Web Live Preview 136-138

**red-green-refactor cycle 6****refactoring**

bad practice, handling 80-82

case studies 79

file-scoped namespace feature 83, 84

interface, generating 82, 83

**reinforcement learning 152****release candidate 288****remote debugging 58-60****repository**

managing, through Visual Studio 190

**RFC 9110 HTTP Semantics**

reference link 139

**Roslyn 62**

benefits 62

used, for refactoring 65

using, in Visual Studio 2022 64

working 63

**Roslyn analyzers**

analyzing with 65

**Roslynator 65****rulers 124****Run to Cursor feature 34, 35****S****semantic model 64****semantic versioning 281****Sign CLI 275****Software Development Kit (SDK) 267****SonarAnalyzer 65****Source Link 52****Source Server 52****Sourcetree 190****Standalone Code Analysis Tool template 65****static code analysis**

.editorconfig file, generating 70-72

level of severity, adjusting 67-70

using, for quality assurance and security 65

using, in Visual Studio 66, 67

**static instrumentation 95****StyleCop 65****supervised learning 152****support vector machines (SVMs) 152****syntax trees 64****System under Test (SUT) 8****T****target frameworks 287****template parameters 256****temporary breakpoint 47****Test-Driven Development (TDD) 3, 4, 31, 87**

AAA pattern 8

cycle 6, 7

practicing, with real-world example 19-23

principles, unifying for software quality 5-7

**Test Explorer view**

overview 11-15

**text adornments 273****thread maker icons 56****tracepoint breakpoint 42, 43****U****unit testing 4**

F.I.R.S.T principles 4, 5

**unit test project**

creating 9-11

**universal windows platform (UWP) 128****unsupervised learning 152****UpdateQuantity 48****Use live data toggle option 138****user interface (UI) 153****V****versioning 281-283**

date-based versioning 282

semantic versioning 281

**viewport adornments 273****Visual Studio 232**

applications, Dockerizing 237, 238

Azure Functions development,  
exploring 173-177

conflicts, resolving 197-199

Docker environments, setting up 233

interactive staging 200-204

NuGet source, setting up 285, 286

repository, managing 190

used, for generating GitHub

Actions file 213-216

Work Items, managing through 223-227

**Visual Studio 2022**

code metrics, using 77-79

Google Cloud Tools, exploring for 177-182

Roslyn, using 64

Test Explorer view, overview 11-15

unit test project, creating 9-11

unit test project, setting up 8

**Visual Studio Command Table file 270****Visual Studio Debugger 32**

bugs, fixing on fly 40

debug mode, entering 32-34

tools 36

**Visual Studio Debugger, tools**

Autos window 36

Call Stack window 38, 39

Immediate window 39, 40

Locals window 37

Watch window 37

**Visual Studio extensions 265**

advanced extension features 272-275

architecture 266, 267

building 267-271

deploying and sharing 275, 276

**Visual Studio Installer 267****Visual Studio Marketplace 266****Visual Studio profiling tools 90**

.NET Async 90, 91

.NET Counters 91, 92

.NET Object Allocation Tracking 93

Database Profiler 95

Event Viewer 94

File I/O 94

**Visual Studio Shell 266****W****Watch window 37****Web Live Preview**

real-time web previews, powered by 136-138

**Windows Presentation****Foundation (WPF) 273**

**Windows UI 3 (WinUI 3)** 118

**workflow files** 209

## **Work Items**

managing, through Visual Studio 223-227

# **X**

## **Xamarin**

.NET Upgrade Assistant, using 131, 133

versus MAUI 131

migrating 131

**XAML Live Preview** 123, 124

**xUnit** 9

**XUnit Analyzers** 65

# **Y**

**Yet Another Markup Language (YAML)** 209





Packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

## Why subscribe?

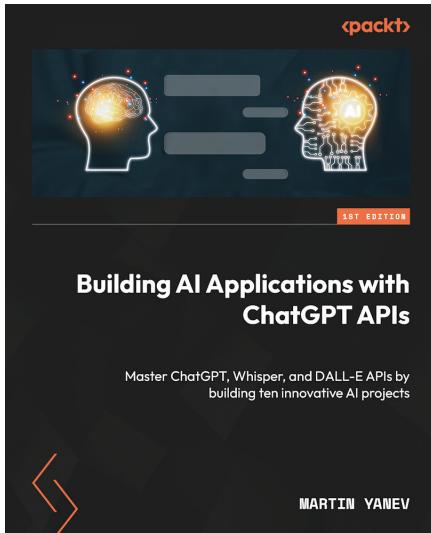
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at [packtpub.com](http://packtpub.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at [customercare@packtpub.com](mailto:customercare@packtpub.com) for more details.

At [www.packtpub.com](http://www.packtpub.com), you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

# Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

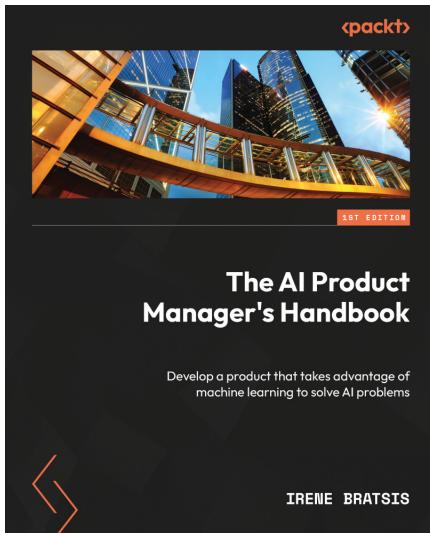


## **Building AI Applications with ChatGPT APIs**

Martin Yaney

ISBN: 978-1-80512-756-7

- Develop a solid foundation in using the ChatGPT API for natural language processing tasks
- Build, deploy, and capitalize on a variety of desktop and SaaS AI applications
- Seamlessly integrate ChatGPT with established frameworks such as Flask, Django, and Microsoft Office APIs
- Channel your creativity by integrating DALL-E APIs to produce stunning AI-generated art within your desktop applications
- Experience the power of Whisper API's speech recognition and text-to-speech features
- Discover techniques to optimize ChatGPT models through the process of fine-tuning



### The AI Product Manager's Handbook

Irene Bratsis

ISBN: 978-1-80461-293-4

- Build AI products for the future using minimal resources
- Identify opportunities where AI can be leveraged to meet business needs
- Collaborate with cross-functional teams to develop and deploy AI products
- Analyze the benefits and costs of developing products using ML and DL
- Explore the role of ethics and responsibility in dealing with sensitive data
- Understand performance and efficacy across verticals

## Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit [authors.packtpub.com](http://authors.packtpub.com) and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Hi,

I am Romain Ottonelli Dabadie author of *Mastering Visual Studio 2022*. I really hope you enjoyed reading this book and found it useful for increasing your productivity and efficiency.

It would really help me (and other potential readers!) if you could leave a review on Amazon sharing your thoughts on this book.

Go to the link below to leave your review:

<https://packt.link/r/1835884695>

Your review will help me to understand what's worked well in this book, and what could be improved upon for future editions, so it really is appreciated.

Best Wishes,

Romain Ottonelli Dabadie

---

## Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/978-1-83588-468-3>

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly