

Mastering Visual Studio 2019

Second Edition

Become proficient in .NET Framework and .NET Core by using advanced coding techniques in Visual Studio



Packt

www.packt.com

Kunal Chowdhury

Mastering Visual Studio 2019

Second Edition

Become proficient in .NET Framework and .NET Core by using advanced coding techniques in Visual Studio

Kunal Chowdhury

Packt

BIRMINGHAM - MUMBAI

Mastering Visual Studio 2019

Second Edition

Copyright © 2019 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.

Commissioning Editor: Richa Tripathi

Acquisition Editor: Chaitanya Nair

Content Development Editor: Tiksha Sarang

Senior Editor: Afshaan Khan

Technical Editor: Romy Dias

Copy Editor: Safis Editing

Project Coordinator: Prajakta Naik

Proofreader: Safis Editing

Indexer: Tejal Daruwale Soni

Production Designer: Nilesh Mohite

First published: July 2017

Second edition: August 2019

Production reference: 1080819

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78953-009-4

www.packtpub.com



Packt.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 www.packt.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 for more details.

At www.packt.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.

Contributors

About the author

Kunal Chowdhury is an author, a passionate blogger, and a software engineer by profession. He was a Microsoft MVP from 2010 to 2018. Over the years, he has acquired profound knowledge of various Microsoft products and services, and has helped developers and consumers throughout the world.

As a tech buff, Kunal has in-depth knowledge of C#, XAML, .NET, WPF, Visual Studio, Windows 10, and Azure. He has written many articles on his technical blog (www.kunal-chowdhury.com) for developers and consumers. You can contact him on Twitter (@kunal2383) and become one of his fans.

He has also authored the books *Mastering Visual Studio 2017* and *Windows Presentation Foundation Development Cookbook*, both available from Packt Publishing.

As always, I would like to thank my wife, Manika Paul Chowdhury, and my parents for their continuous support while writing this book. I would also like to thank the publisher for the opportunity, and the reviewers for their valuable feedback.

About the reviewers

Atul Verma is currently a senior consultant with Microsoft and is a graduate from NIT Hamirpur. He has more than 13 years' experience of working on Microsoft technologies and open source software. He has expertise in development of polyglot solutions including Azure, mobile apps, desktop apps, ASP.NET Core, Azure DevOps, Elasticsearch, Kafka, Docker, and Kubernetes across domains including property automation, data visualization, and law enforcement.

His recent technical certifications include Microsoft Certified Azure Developer Associate, MCSD: App Builder and MCSA: Universal Windows Platform. He also contributes to technical communities, blogs, and GitHub repositories.

Jaliya Udagedara is a software engineer originally from Sri Lanka, and is currently based in Auckland, New Zealand. He is a Microsoft MVP and a technology evangelist, mostly working in the Microsoft technology stack. He has found his most significant interest in the world of .NET/.NET Core/ASP.NET/ASP.NET Core and Azure. He has more than 10 years' experience in software development.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit 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.

Table of Contents

Preface	1
<hr/>	
Section 1: What's New in Visual Studio 2019?	
<hr/>	
Chapter 1: What's New in the Visual Studio 2019 IDE?	9
Technical requirements	10
The Visual Studio 2019 installation experience	10
Overview of the installation experience	12
Installing using the online installer	13
Creating an offline installer for Visual Studio 2019	19
Installing Visual Studio 2019 from the command line	20
Modifying your existing Visual Studio 2019 installation	23
Uninstalling Visual Studio 2019	25
Overview of the new Start window	26
Cloning a repository from the Start window	27
Opening an existing project or solution from the Start window	30
Opening a local folder from the Start window	31
Creating a new project from the Start window	32
Improved search in Visual Studio 2019	34
Quick actions improvements for code refactoring	37
Inverting if statements	37
Wrapping arguments	38
Unwrapping arguments	38
Introducing inline variables	39
Pulling a member to its base type	40
Adjusting namespaces to match the folder structure	41
Converting foreach loops into generalized LINQ queries	42
Visual Studio IntelliCode	43
Installing the IntelliCode component	44
Understanding how IntelliCode helps	44
One-click code cleanup	45
Clipboard History	47
Searching in debugging windows	48
Time Travel Debugging	49
Live Share	51
Creating a new live sharing session	52
Joining an existing live sharing session	54
Managing a live sharing session	57
Product Updates	58

Summary	60
Section 2: Building and Managing Applications	
Chapter 2: Building Desktop Applications for Windows Using WPF	62
Technical requirements	63
Understanding the WPF architecture	63
An overview of XAML	65
The object element syntax	65
The property attribute syntax	66
The property element syntax	66
The content syntax	66
The collection syntax	67
The event attribute syntax	68
Building your first WPF application	68
Getting started with the WPF project	70
Understanding the WPF project structure	73
Getting familiar with XAML Designer	75
Adding controls in XAML	76
Exploring layouts in WPF	79
Using Grid as a WPF panel	81
Using StackPanel to define the stacked layout	83
Using Canvas as a panel	84
Using WPF DockPanel to dock child elements	85
Using WrapPanel to automatically reposition elements	86
Using UniformGrid to place elements in uniform cells	87
The WPF property system	89
Data binding in WPF	90
Using converters while data binding	96
Using triggers in WPF	99
The property trigger	99
The MultiTrigger property	100
The data trigger	101
The multi data trigger	102
The event trigger	104
Summary	106
Chapter 3: Accelerate Cloud Development with Microsoft Azure	107
Understanding cloud computing basics	108
IaaS	109
PaaS	110
SaaS	110
Creating your free Azure account	110
Configuring Visual Studio 2019 for Azure development	112
Creating an Azure website from the portal	114

Creating a web application	115
Creating an App Service plan	118
Managing Azure websites (web apps) from the portal	119
Creating an Azure website with Visual Studio 2019	123
Creating an ASP.NET web application	123
Publishing the web application to the cloud	125
Updating an existing Azure website with Visual Studio	131
Building a mobile app	133
Creating an Azure mobile app	134
Preparing your Azure mobile app for data connectivity	136
Adding a SQL data connection	137
Creating a SQL database	138
Integrating the mobile app service in a Windows application	143
Creating the model and service client	143
Integrating the API call	144
Scaling an App Service plan	150
Summary	153
Chapter 4: Building Applications with .NET Core	154
Technical requirements	155
Overview of .NET Core	155
Installing .NET Core with Visual Studio 2019	157
A quick lap around .NET Core commands	159
Creating a .NET Core console app	160
Creating a .NET Core class library	162
Creating a solution file and adding projects to it	162
Resolving dependencies in .NET Core applications	165
Building a .NET Core project or solution	166
Running a .NET Core application	168
Publishing a .NET Core application	169
FDD	170
SCD	171
Creating an ASP.NET Core application	173
Creating a unit testing project	174
Creating .NET Core applications using Visual Studio 2019	175
Publishing .NET Core applications using Visual Studio 2019	177
FDD	178
SCD	180
Creating, building, and publishing a .NET Core web app to Microsoft Azure	186
Summary	195
Chapter 5: Web Application Development Using TypeScript	196
Technical requirements	197
Setting up the development environment by installing TypeScript	197

Installing through NPM	197
Installing through Visual Studio 2019's installer	199
Building your first Hello TypeScript application	200
Understanding the TypeScript configuration file	204
Declaring variables in TypeScript	205
Declaring variables using the var keyword	206
Problems with using the var keyword	206
Declaring variables using the let keyword	207
Declaring constants using the const keyword	208
Working with the basic datatypes	208
Using the number type	209
Using the string type	209
Using the bool type	210
Using the enum type	210
Using the void type	211
Using the undefined type	211
Using the null type	212
Using the any type to declare a variable	212
Using the never type	212
Using the array type	213
Using the tuple type	214
Working with classes and interfaces	214
Defining classes in TypeScript	214
Initiating an instance of a class	216
Defining abstract classes in TypeScript	216
Working with inheritance in TypeScript	217
Defining interfaces in TypeScript	218
Classes implementing interfaces	218
An interface extending another interface	219
Interfaces extending multiple interfaces	220
Interfaces extending classes	220
Summary	221
Chapter 6: Managing NuGet Packages	222
Technical requirements	223
Overview of NuGet Package Manager	223
Creating a NuGet package library for .NET Framework	227
Creating the metadata in the nuspec file	229
Building the NuGet package	232
Building NuGet packages for multiple .NET Frameworks	234
Building a NuGet package with dependencies	236
Publishing a NuGet package to the NuGet store	237
Managing your NuGet packages	240
Summary	242

Section 3: Debugging and Testing Applications

Chapter 7: Debugging Applications with Visual Studio 2019	244
Technical requirements	245
Overview of Visual Studio debugger tools	245
Debugging C# source code using breakpoints	246
Organizing breakpoints in code	247
Debugger execution steps	250
Adding conditions to breakpoints	253
Using conditional expressions	254
Using breakpoint hit counters	255
Using breakpoint filters	255
Adding actions to breakpoints	256
Adding labels to breakpoints	258
Managing breakpoints using the Breakpoints window	259
Exporting/importing breakpoints	260
Using DataTips while debugging	260
Pinning/unpinning DataTips for better debugging	261
Inspecting DataTips in various Watch windows	263
The Autos window	263
The Locals window	264
The Watch window	264
Using visualizers to display complex DataTips	268
Importing/exporting DataTips	269
Using the debugger to display debugging information	270
Using the Immediate Window while debugging your code	272
Using the Visual Studio Diagnostics Tools	273
Summary	278
Chapter 8: Live Unit Testing with Visual Studio 2019	279
Technical requirements	279
Overview of Live Unit Testing in Visual Studio 2019	280
Unit testing framework support	281
Understanding the coverage information shown in the editor	281
Integration of Live Unit Testing in Test Explorer	282
Configuring Visual Studio 2019 for Live Unit Testing	283
Installing the Live Unit Testing component	283
General settings of Live Unit Testing in Visual Studio	284
Starting and pausing Live Unit Testing	286
Including and excluding test methods/projects	287
Live Unit Testing with Visual Studio 2019	289
Getting started with configuring the testing project	289
Understanding the packages.config file	292
Live Unit Testing with an example	293
Navigating failed tests	297

Summary	298
Section 4: Source Control	
Chapter 9: Exploring Source Controls in Visual Studio 2019	300
Technical requirements	301
Installing Git for Visual Studio 2019	301
Connecting to the source control servers	302
Getting started with Git repositories	303
Creating a new repository	304
Cloning an existing repository	305
Working with Git branches	307
Creating a new local branch	307
Switching to a different branch	309
Pushing a local branch to the remote repository	310
Deleting an existing branch	311
Working with changes, staging, and commits	312
Staging changes to your local repository	313
Committing changes to your local repository	314
Discarding uncommitted changes	315
Amending messages to an existing commit	317
Syncing changes between local and remote repositories	317
Pushing changes to the remote repository	318
Fetching changes in the remote repository	319
Merging changes in the remote repository with your local repository	320
Resolving merge conflicts	321
Working with pull requests for a code review	324
Creating pull requests for a code review	325
Reviewing an existing pull request	326
Working with the Git commit history	330
Rebasing changes to rewrite the commit history	330
Copying commits using cherry-pick	333
Undoing your changes	334
Resetting a local branch to a previous state	335
Reverting changes from a remote branch	337
Tagging your commits	338
Summary	341
Other Books You May Enjoy	342
Index	345

Preface

Day by day, a revolution is happening in the computer world; existing technologies are becoming older and obsolete, making way for newer ones. To learn about and work with modern technologies, you will need an updated IDE. Microsoft provides this with what is the most popular IDE among developers, named Visual Studio.

Microsoft released Visual Studio for developers in the year 1997. In the 2002, it first received a flavor of .NET and then it started to be revolutionized, with many new features added to every major version. In Visual Studio 2015, Microsoft added support for .NET Core, which is a cross-platform, free, and open source managed software framework, such as .NET Framework.

Visual Studio 2017, initially known as Visual Studio 15, was released on March 7, 2017. It featured a new installation experience, with which you could install a specific workload or a component that you need to accomplish your work. Apart from this, it also included features such as .NET Core, and support for NGen, EditorConfig, Docker, and Xamarin. It not only had support for Microsoft platforms but also supports Linux app development, C/C++, Cordova, Python, Node.js, and tooling for data science and analytical applications.

Microsoft has now improved Visual Studio 2017, releasing a new version of the IDE, named Visual Studio 2019, which was released on April 2, 2019. With this release, Microsoft improved the IDE and tools needed to build and test applications and games to make Visual Studio enterprise-grade. With new features such as IntelliCode and Live Share, it's now easy to build applications quickly and collaborate easily with team members.

In this book, we will cover most of the changes that have been made to move you one step ahead of the advancements. From the installation changes to new features introduced in the IDE, we'll give you a base from which to start working with Visual Studio 2019. Then, we will move forward to learn about building apps for Windows and the web using WPF, .NET Core, and TypeScript. We will learn about NuGet, debugging and unit testing applications, cloud development with Azure, and source control (version control) integration.

The examples given in this book are simple and easy to understand, providing you with a heads-up on how to develop skills with and master the new version of the IDE, Visual Studio 2019. By the time you come to the end of this book, you will be a proficient user with deep knowledge of each of the chapters covered. You will enjoy reading this book, with lots of graphical and textual steps to help you gain confidence in working with this IDE.

Who this book is for

.NET developers who would like to master Visual Studio 2019, and would like to delve into newer areas such as .NET Core, TypeScript, WPF, and Git, will benefit from this book. Basic knowledge of C#, XAML, and previous versions of Visual Studio is assumed.

What this book covers

Chapter 1, *What's New in the Visual Studio 2019 IDE?*, focuses on the new IDE-specific changes incorporated into Visual Studio 2019 and how these will help developers to improve their productivity. Starting with installation, it covers various workloads and component parts of the installer and then guides you through the all-new start window, followed by other key new features.

Chapter 2, *Building Desktop Applications for Windows Using WPF*, focuses on building XAML-based **Windows Presentation Foundation (WPF)** applications for Windows. This will help you to learn about the WPF architecture, XAML syntax, various layouts, data bindings, converters, and triggers and guide you through building professional applications from scratch.

Chapter 3, *Accelerate Cloud Development with Microsoft Azure*, enables you to easily understand cloud computing basics, including Microsoft Azure, which is an open, flexible, enterprise-grade cloud computing platform. It basically delivers **Infrastructure as a Service (IaaS)**, **Platform as a Service (PaaS)**, and **Software as a Service (SaaS)**. This chapter provides guidance on how to create Azure websites and mobile app services and then integrate them into Windows applications.

Chapter 4, *Building Applications with .NET Core*, takes you on a quick lap around the new framework and guides you through how to create, build, run, and publish .NET Core applications. This chapter includes in-depth explanations of framework-dependent deployments and self-contained deployments. It also provides guidance on how to publish ASP.NET applications to Windows Azure.

Chapter 5, *Web Application Development Using TypeScript*, provides guidance on setting up the development environment by installing TypeScript. Then, we will build our first TypeScript application by discussing the TypeScript configuration file, variable declarations, basic datatypes, classes, and interfaces. This will give you a kick-start to build web applications using TypeScript, an open-source programming language that is developed and maintained by Microsoft.

Chapter 6, *Managing NuGet Packages*, focuses on the NuGet package manager for the Microsoft development platform, including .NET. The NuGet client tools provide the ability to produce and consume packages. The NuGet Gallery is the central package repository used by all package authors and consumers. In this chapter, we will learn how to create a NuGet package, publish it to a gallery, and test it.

Chapter 7, *Debugging Applications with Visual Studio 2019*, focuses on giving you an in-depth understanding of the different debugging tools available in Visual Studio. It's the core part of every code development. The more comfortable you are with code debugging, the better the code you will write/maintain. This chapter will help you to learn the debugging process of Visual Studio 2019.

Chapter 8, *Live Unit Testing with Visual Studio 2019*, provides a deeper insight into Live Unit Testing, which is a new module introduced in Visual Studio 2017. It automatically runs impacted unit tests in the background as you edit code and then visualizes the results with code coverage, live in the editor. This chapter will help you to become proficient at using Live Unit Testing with Visual Studio 2019.

Chapter 9, *Exploring Source Controls in Visual Studio 2019*, demonstrates the steps to take to manage your code with versioning support in a source control repository. Source control is a component of the software configuration management, source repository, and version management system. If you are building enterprise-level applications in a distributed environment, you must use source control to keep your code in a safe vault. This chapter demonstrates how easy it is to use Git to manage your code directly from Visual Studio 2019.

To get the most out of this book

To get the most out of this book, you will need Microsoft Visual Studio 2019 installed on your system. Choosing the right version of Visual Studio 2019 can be done as follows.

Microsoft Visual Studio 2019 comes in three different editions: **Visual Studio Community 2019**, **Visual Studio Professional 2019**, and **Visual Studio Enterprise 2019**.

The Visual Studio Community Edition is a free, fully-featured IDE for students, open source developers, and individual developers. In each of these cases, you can create your own free or paid apps using Visual Studio 2019 Community Edition. Organizations are also able to use the Community Edition, but only under the following conditions:

- In an enterprise organization, an unlimited number of users can use the community edition if they are using it in a classroom-learning environment, for academic research, or for an open source project. An organization is defined as an enterprise organization if they have more than 250 computers or \$1 million in annual revenue.
- In a non-enterprise organization, the Community Edition is restricted to up to five users.

If you are a professional in a small team, you should go for Visual Studio Professional 2019. If you are part of a large organization, building end-to-end solutions in a team of any size, and if the price does not matter to you, then Visual Studio Enterprise 2019 is the right choice as it includes all the features that Visual Studio offers.



Note that you can install multiple editions of Visual Studio 2019 side by side. You can also install multiple instances of the same edition on a single system. So, feel free to install any or all editions based on your needs.

Download the example code files

You can download the example code files for this book from your account at www.packt.com. If you purchased this book elsewhere, you can visit www.packt.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packt.com.
2. Select the **Support** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Mastering-Visual-Studio-2019-Second-Edition>. In case there's an update to the code, it will be updated on the existing 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!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: https://static.packt-cdn.com/downloads/9781789530094_ColorImages.pdf.

Conventions used

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

CodeInText: 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: "The `dotnet restore` command restores the dependencies and tools of a project."

A block of code is set as follows:

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
<files>
  <file src="bin\Release\NuGetDemoLibrary.dll"
        target="lib\NuGetDemoLibrary.dll"/>
</files>
```

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

```
dotnet sln <SolutionName> add <ProjectName>
dotnet sln <SolutionName> add <ProjectOneName> <ProjectTwoName>
dotnet sln <SolutionName> add **/**
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "In the **New Project** dialog, navigate to **Installed | Templates | Visual C# | .NET Core**."

Warnings or important notes appear like this.



Tips and tricks 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, mention the book title in the subject of your message and email us at customercare@packtpub.com.

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.packt.com/submit-errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

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 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.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.

1

Section 1: What's New in Visual Studio 2019?

This section focuses on the new IDE-specific changes incorporated into Microsoft Visual Studio 2019 and how these will help developers to improve their productivity. Starting from installation, this section covers the various workloads and the component parts of the installer and then guides you through the new core features specific to the IDE.

The following chapter will be covered in this section:

- Chapter 1, *What's New in Visual Studio 2019 IDE?*

1

What's New in the Visual Studio 2019 IDE?

Visual Studio 2019 is the new IDE for developers and was released by Microsoft on 2 April 2019. Just like Visual Studio 2017, this IDE also focuses on building applications using C#, VB.NET, F#, C++, Python, Java, Xamarin, and more. In short, it is an IDE for every developer who needs to build applications for any platform.

This IDE will help you to save time and effort for all of the tasks that you want to complete with your code, be it code navigation, refactoring, code debugging, code fixes, IntelliSense, code sharing with team members, or unit testing your modules.

Using Xamarin, you can build mobile applications for Android, iOS, and Windows. You can also build mobile apps with Visual C++ or Apache Cordova.

Visual Studio 2019 can also help you to streamline your real-time architectural dependency validations and provide you with stronger support for the integration of source code repositories such as **GitHub** and **Azure DevOps**.

In this chapter, we are going to cover the key new features and enhancements that Microsoft has added to the Visual Studio 2019 IDE. The following topics are going to be covered in this chapter:

- The Visual Studio 2019 installation experience
- Overview of the new Start window
- Improved search in Visual Studio 2019
- Quick actions improvements for code refactoring
- Visual Studio IntelliCode
- One-click code cleanup
- Clipboard History
- Searching in debugging windows
- Time Travel Debugging
- Live Share
- Product Updates

Technical requirements

To complete this chapter, you need to have basic knowledge of C# and Visual Studio. Installing Visual Studio 2019 (any edition) is mandatory, but to work with Time Travel Debugging, you will need Visual Studio 2019 Enterprise Edition.

The Visual Studio 2019 installation experience

With the launch of Visual Studio 2017, Microsoft redesigned the Visual Studio installation experience. If you've skipped Visual Studio 2017 and jumped directly into Visual Studio 2019, this section will guide you through how you can install, update, and uninstall the product.

To begin with, we will discuss the various workloads and components of the Visual Studio 2019 installation experience. The basic installer that comes in web-only mode allows you to select the components that you want to install before it downloads them. This saves you a lot of bandwidth.

Then, we will discuss how to create an offline installer for installing Visual Studio without internet connectivity. We need to create the offline installer because, beginning with Visual Studio 2017, no offline installer is shipped out by Microsoft. This can be done by creating a layout using the web installer. The download size of the offline installer is big, but saves you time and bandwidth when you want to install it on multiple devices.

In this section, we will also learn how to configure and install different workloads or components using the online and offline installers. Then, we will learn how to modify or uninstall Visual Studio.

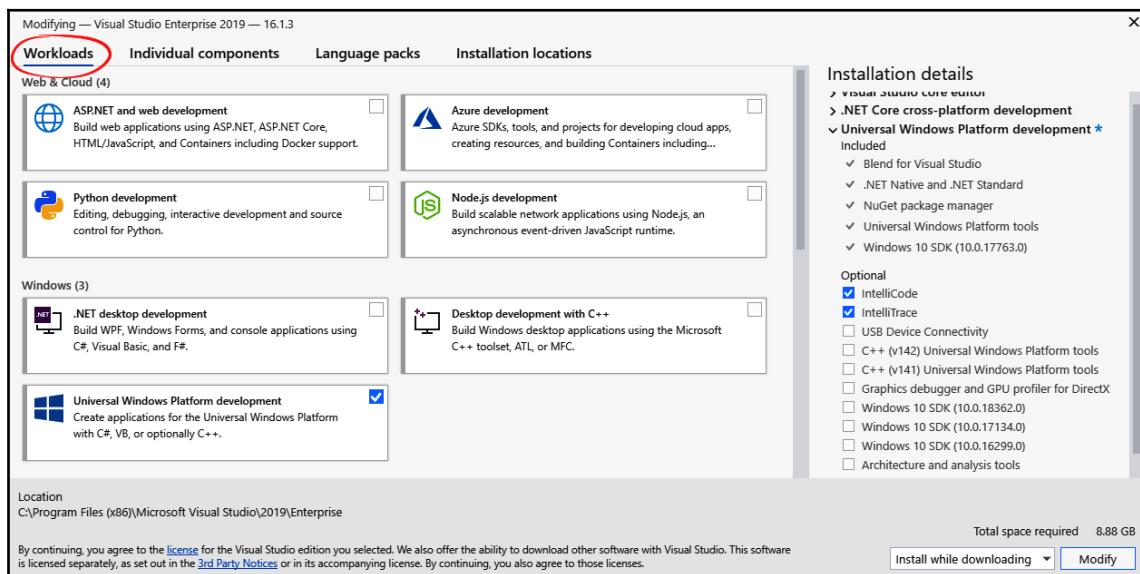
Before we go in depth, let's take a look at the system requirements for installing Visual Studio 2019:

- You can install and run Visual Studio 2019 on the following operating systems:
 - Windows 10 version 1703 or higher to build apps for the **Universal Windows Platform (UWP)**
 - Windows Server 2016 and Windows Server 2019
 - Windows 8.1 (with Update 2919355)
 - Windows Server 2012 R2 (with Update 2919355)
 - Windows 7 SP1 (with the latest Windows Updates)
- The following are the hardware requirements:
 - You should have a 1.8 GHz or faster processor. It's recommended to have a quad-core processor or higher.
 - You will need at least 2 GB of RAM, though Microsoft recommends 8 GB of RAM. If you want to run Visual Studio 2019 in a VM system, a minimum of 2.5 GB of RAM is recommended.
 - You should have at least 800 MB HDD space for a basic installation. The minimum space can vary, depending on the workloads you select.
 - Visual Studio 2019 will work best at a resolution of WXGA (1,366 by 768) or higher, but you can install on a 1,280 by 720 resolution too.

Overview of the installation experience

The new version of the installer that's used to install Visual Studio 2017 or higher allows you to control the individual workloads/modules that you need. Unlike the previous versions of the installer, it doesn't take up more installation space; instead, it allows you to do a basic installation since you only have a few hundred MBs for the core editor to install. You can select the workload or the individual module on a need to need basis.

The **Workloads** screen will allow you to select the module that you want to install. If you want to build applications targeting Windows 10 only, you should go with **Universal Windows Platform development**:



If you want to build applications for Python or Node.js, the respective workloads are there to help you to install the required components.

We will discuss the installation steps more in the next section, where we will see how to install Visual Studio 2019 using the online installer.

Installing using the online installer

You can go to <https://www.visualstudio.com/downloads/> to select and download the Visual Studio 2019 edition that best suits your needs. There are three different editions available, that is, Visual Studio Community Edition 2019, Visual Studio Professional Edition 2019, and Visual Studio Enterprise Edition 2019.

The **Visual Studio Community Edition** is a free, fully-featured IDE for students, open source developers, and individual developers. In all these cases, you can create your own free or paid apps using Visual Studio 2019 Community Edition. Organizations will also be able to use the Community Edition, but only under the following conditions:

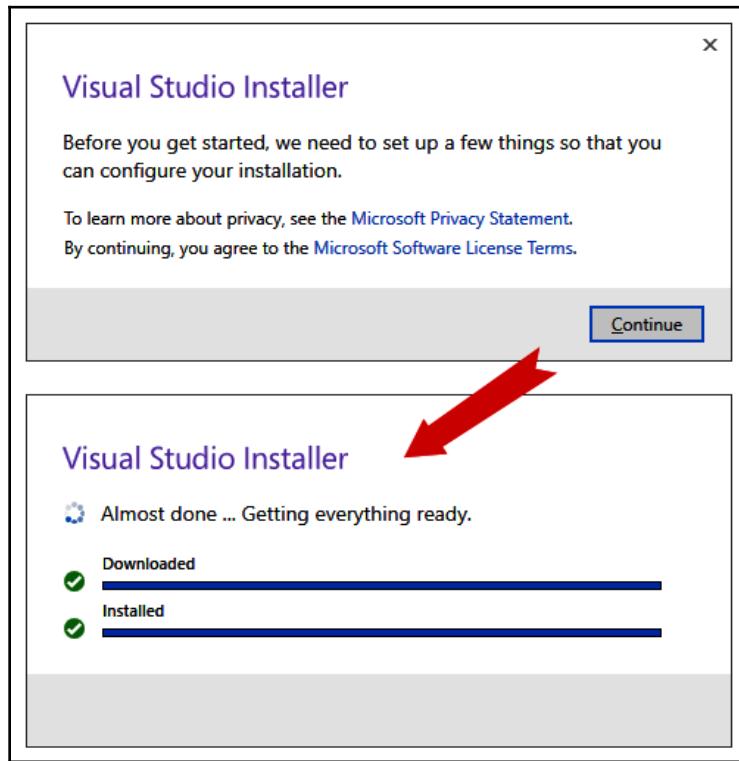
- In an enterprise organization, an unlimited number of users can use the Community Edition if they are using it in a classroom learning environment, academic research, or an open source project. An organization is defined as an enterprise organization if they have more than 250 computers or one-million-dollar annual revenue.
- In a non-enterprise organization, the Community Edition is restricted to five users.

To find out more about the Visual Studio Community license terms, check out this page:
<https://www.visualstudio.com/license-terms/mlt553321/>.

If you are a professional working within a small team, you need to select Visual Studio Professional Edition 2019. If you need to create an end-to-end solution for a team of any size, you need to select Visual Studio Enterprise Edition 2019.

Once you have downloaded the online/web installer, double-click it to start the installation process. This will show a screen where you can read the license terms and privacy statement, which you need to agree to before continuing with the installation process.

Once you click the **Continue** button, the installer will take a few minutes to prepare itself, as shown in the following screenshot:

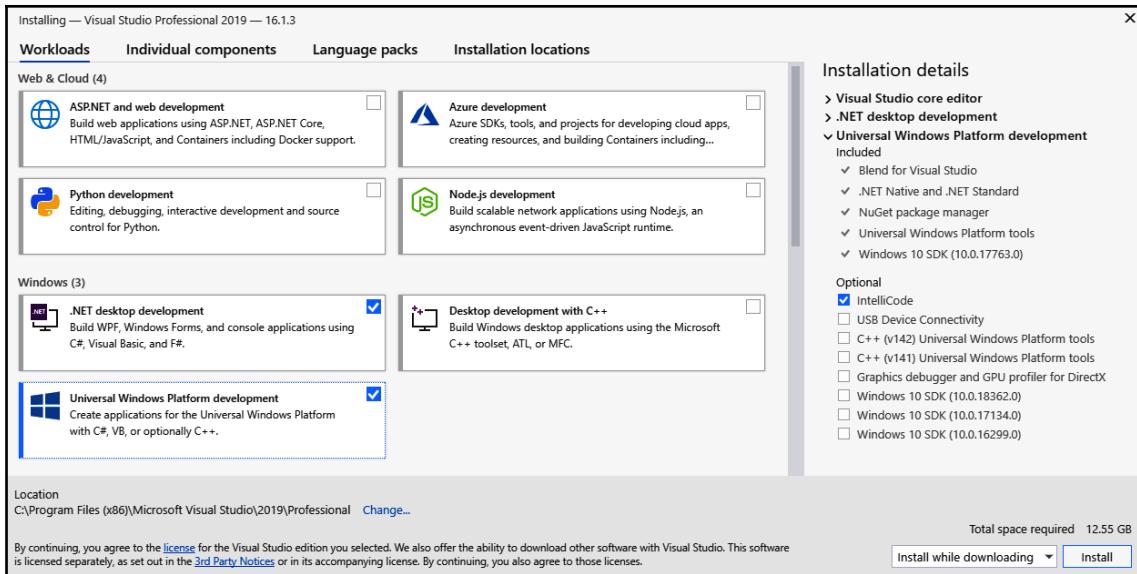


The main screen of the installer has four different tabs: **Workloads**, **Individual Components**, **Language Packs**, and **Installation locations**.

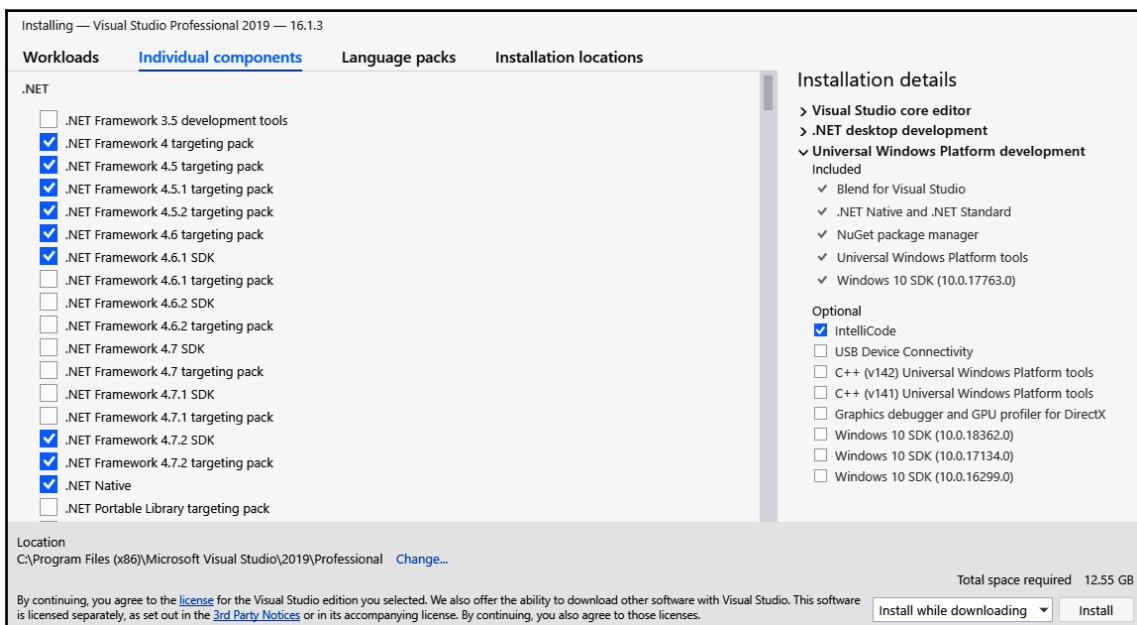
The **Workloads** tab allows you to select the group of components that come under a single module. In other words, each workload contains the features you need for the programming language or platform you prefer.

For example, if you like to build WPF applications, you need to select **.NET desktop development**; to build ASP.NET web applications, you need to select **ASP.NET and web development module** under the **Workloads** tab.

To install and build applications for both WPF and Windows 10, select **.NET desktop development** and **Universal Windows Platform development**, as shown in the following screenshot. For each individual workload, the selected components will be listed on the right-hand panel of the screen:



The **Individual components** tab lists all of the components that are a part of individual workloads, in categories. The components part of the selected workloads will be auto-checked by default. This is shown in the following screenshot:





Only use this section if you are an advanced user. Some components may be dependent on one or more workloads. Deselecting one of them can cause the other workloads to unload from the installation process. So, be cautious while selecting/deselecting any of them.

The third tab is called **Language packs** and allows you to choose the language that you want to use with Visual Studio 2019. By default, it's the system default language, which is English in my case; however, you can opt for Chinese, French, German, or any other language from the available list:

Installing — Visual Studio Professional 2019 — 16.1.3

Workloads Individual components Language packs Installation locations

You can add additional language packs to your Visual Studio installation.

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

Installation details

> Visual Studio core editor
> .NET desktop development
✓ Universal Windows Platform development included
 ✓ Blend for Visual Studio
 ✓ .NET Native and .NET Standard
 ✓ NuGet package manager
 ✓ Universal Windows Platform tools
 ✓ Windows 10 SDK (10.0.17763.0)

Optional

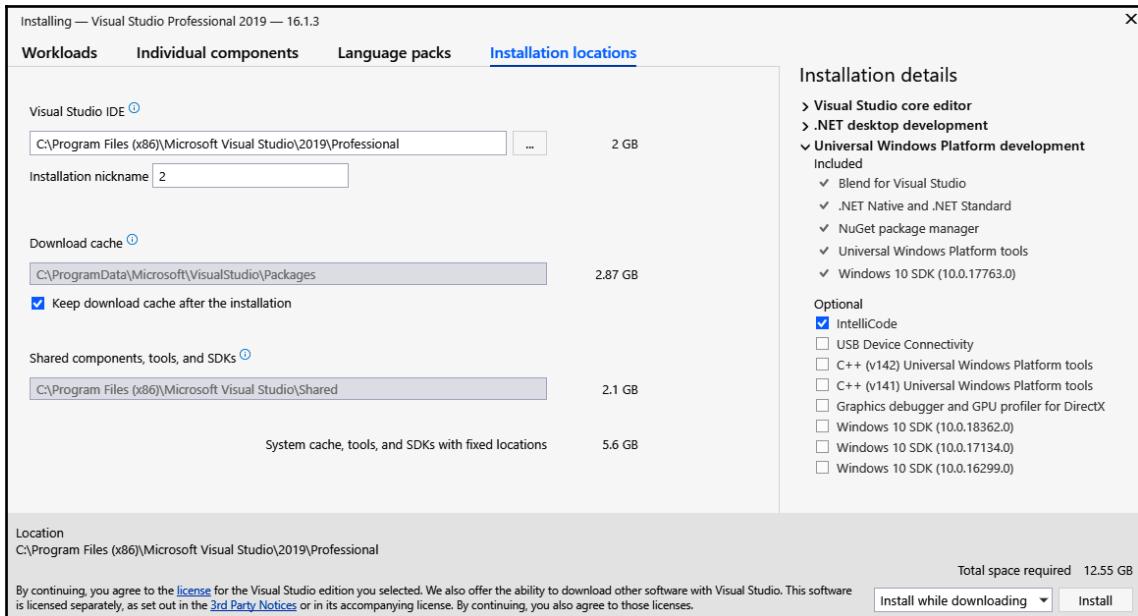
IntelliCode
 USB Device Connectivity
 C++ (v142) Universal Windows Platform tools
 C++ (v141) Universal Windows Platform tools
 Graphics debugger and GPU profiler for DirectX
 Windows 10 SDK (10.0.18362.0)
 Windows 10 SDK (10.0.17134.0)
 Windows 10 SDK (10.0.16299.0)

Location C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional Change... Total space required 12.55 GB

By continuing, you agree to the [license](#) for the Visual Studio edition you selected. We also offer the ability to download other software with Visual Studio. This software is licensed separately, as set out in the [3rd Party Notices](#) or in its accompanying license. By continuing, you also agree to those licenses.

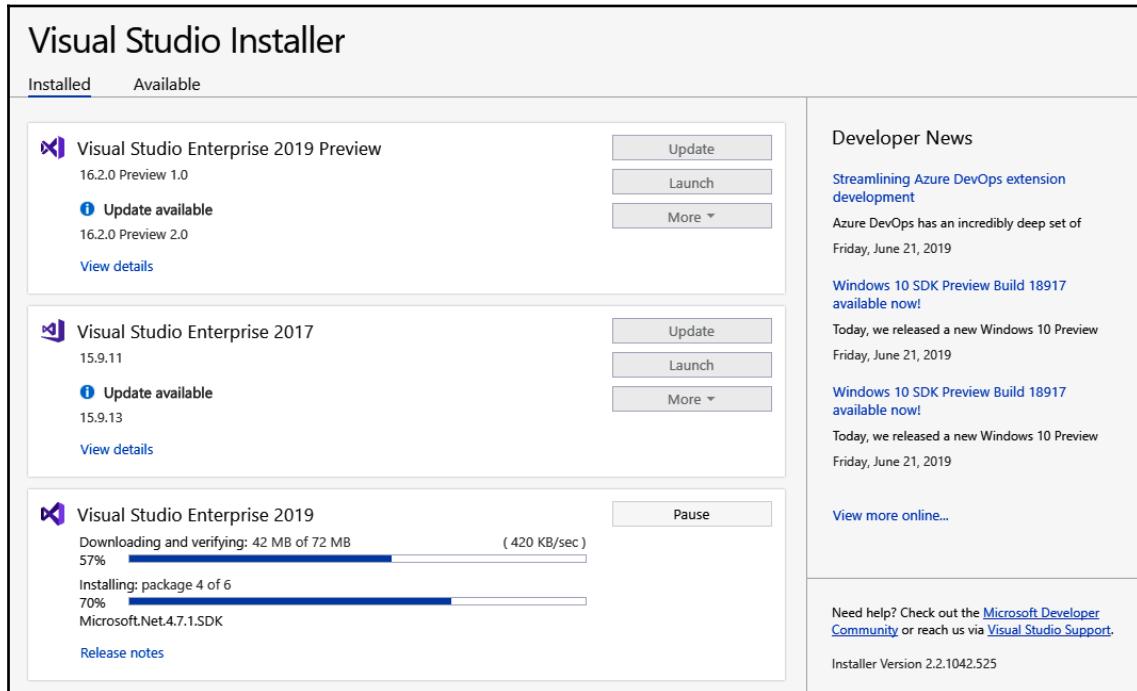
Install while downloading ▾ Install

By default, a file location is prepopulated for the installer to install Visual Studio 2019, but you can change it to a different folder if you wish. The fourth tab, called **Installation locations**, allows you to select the IDE installation path, installation cache path, and shared components path, as shown in the following screenshot:



Did you know that you can install more than one instance of the same edition of Visual Studio 2019 on a single system? For each installation, set the **Installation nickname** in the **Installation locations** tab for identification.

Once you are done customizing the installation components, click on the **Install** button. This will start the actual installation process. If you are using the web installer, it will download the individual modules from the Microsoft server and install them gradually. This may take some time, based on your selected workloads/components and internet bandwidth:



Did you just notice that the **Visual Studio Installer** lists all of the editions (including previews) of Visual Studio 2017 and Visual Studio 2019 that are already installed on your system? This was made possible starting with Visual Studio 2017.

Once the installation has been completed, it may ask you to restart your system for the changes to take effect. If you see such a message on your screen, make sure to restart your computer by clicking the **Restart** button.

Creating an offline installer for Visual Studio 2019

Sometimes, we may need to have an offline copy of the installer so that we can install it on multiple devices without any active internet connection. This will save your bandwidth from downloading the same copy multiple times over your network. The offline installer is big, so before you learn how to create the offline copy, make sure that you have an active internet connection available, with no limitations in terms of download bandwidth:

1. First, download the Visual Studio setup executable file (web installer) to a drive on your local machine.
2. Now, run the downloaded setup executable with the following arguments (switches) from Command Prompt:
 - Add `--layout <path>`, where `<path>` is the location you want the layout to be downloaded to. By default, all languages will be downloaded, along with all of the packages.
 - If you want to restrict the download to a single language only, you can do so by providing the `--lang <language>` argument, where `<language>` is one of the ISO country codes given in the following list. If it's not specified, support for all localized languages will be downloaded.

The following is a list of such localized languages:

ISO code	Language
cs-CZ	Czech
de-DE	German
en-US	English
es-ES	Spanish
fr-FR	French
it-IT	Italian
ja-JP	Japanese
ko-KR	Korean
pl-PL	Polish
pt-BR	Portuguese – Brazil
ru-RU	Russian
tr-TR	Turkish
zh-CN	Chinese – Simplified
zh-TW	Chinese – Traditional

If you want to download Visual Studio 2019 Enterprise Edition under the C:\VS2019\ local path, you need to use the following command:

```
vs_enterprise.exe --layout "C:\VS2019\"
```

To download the English localized edition to the C:\VS2019\ local path, use the following command:

```
vs_enterprise.exe --layout "C:\VS2019\" --lang en-US
```

To download only the .NET desktop development workload, use the following command:

```
vs_enterprise.exe --layout "C:\VS2019\" --add  
Microsoft.VisualStudio.Workload.ManagedDesktop
```

To download .NET desktop development and Azure development workloads, use the following command:

```
vs_enterprise.exe --layout "C:\VS2019\" --add  
Microsoft.VisualStudio.Workload.ManagedDesktop  
Microsoft.VisualStudio.Workload.Azure
```

It will start downloading all of the packages that are part of Visual Studio 2019 and part of your selection. Since the offline installer size is big, it will take a lot of time, depending on the speed of your internet network and the workloads/components that you have selected.

Once the download completes, go to the folder where you downloaded the packages (in our case, it's C:\VS2019\) and run the installer file (that is, vs_enterprise.exe, for example). Then, follow the same steps that were mentioned earlier to select the required **Workflows** and/or **Individual components** to start the installation process.

Installing Visual Studio 2019 from the command line

You can use command-line parameters/switches to install Visual Studio 2019. Make sure to use the actual installer (for example, vs_enterprise.exe for Visual Studio 2019 Enterprise Edition) and not the bootstrapper file, which is named vs_setup.exe. The bootstrapper file loads the MSI for the actual installation.

You can also run C:\Program Files (x86)\Microsoft Visual Studio\Installer\vs_installershell.exe to install Visual Studio components from the command line.

The following is a list of the command-line parameters/switches that you can use:

Parameters/switch	Description
[--catalog] <uri> [<uri> ...]	Required: One or more file paths or URIs to catalog is needed.
--installDir <dir>	Required: The target installation directory is needed.
--installationDirectory <dir>	
-l <path>, --log <path>	Specify the log file; otherwise, one is automatically generated.
-v, --verbose	Display verbose messages.
-?, -h, --help	Display parameter usage.
--instanceId <id>	Optional: Add the instance ID to install or repair.
--productId <id>	Optional: Add the product ID to install. Otherwise, the first product that's found is installed.
--all	Optional: Specify whether to install all workloads and components for a product.
--add <workload or component ID> ...	Optional: This specifies to one or more workload or component IDs to add.
--remove <workload or component ID> ...	Optional: This specifies to one or more workload or component IDs to remove.
--optional, --includeOptional	Optional: This specifies to whether to install all optional workloads and components for the selected workload.
--lang, --language <language-locale> ...	Optional: Install/uninstall resource packages with the specified language(s).
--sharedInstallDir <dir>	Optional: This is the target installation directory for shared payloads.
--compatInstallDir <dir>	Optional: This is the target installation directory for legacy compatibility payloads.
--layoutDir <dir> --layoutDirectory <dir>	Optional: This is the layout directory where you will find packages.
--locale <language-locale>	Optional: Change the display language of the user interface for the installer. This setting will persist.
--quiet	Optional: Do not display any user interface while performing the installation.
--passive	Optional: Display the user interface, but do not request any interaction from the user.

The following is a list of workload IDs that you need to provide while installing Visual Studio 2019 from the command line:

- `Microsoft.VisualStudio.Workload.CoreEditor`: This is the core part of Visual Studio 2019 and contains the core shell experience, syntax-aware code editing, source code control, and work item management.
- `Microsoft.VisualStudio.Workload.Azure`: This contains the Azure SDK, tools, and projects for developing cloud apps and creating resources.
- `Microsoft.VisualStudio.Workload.Data`: Using this workload, you can connect, develop, and test data solutions using SQL Server, Azure Data Lake, Hadoop, or Azure Machine Learning.
- `Microsoft.VisualStudio.Workload.ManagedDesktop`: To build WPF, Windows Forms, and console applications using the .NET Framework, you will need this workload.
- `Microsoft.VisualStudio.Workload.ManagedGame`: If you are a game developer, you can create 2D and 3D games with Unity, a powerful cross-platform development environment.
- `Microsoft.VisualStudio.Workload.NativeCrossPlat`: Do you want to create and debug applications running in a Linux environment? This workload will allow you to build native cross-platform apps.
- `Microsoft.VisualStudio.Workload.NativeDesktop`: Classic Windows-based applications using the power of the Visual C++ toolset, ATL, and optional features like MFC and C++/CLI can be built using this workload.
- `Microsoft.VisualStudio.Workload.NativeGame`: If you are a game developer, you can use the full power of C++ to build professional games powered by DirectX, Unreal, or Cocos2d.
- `Microsoft.VisualStudio.Workload.NativeMobile`: Using this, you can build cross-platform applications for iOS, Android, or Windows using the C++ APIs.
- `Microsoft.VisualStudio.Workload.NetCoreTools`: .NET Core is a new edition to Visual Studio. You can build cross-platform applications using .NET Core, ASP.NET Core, HTML, JavaScript, and CSS.

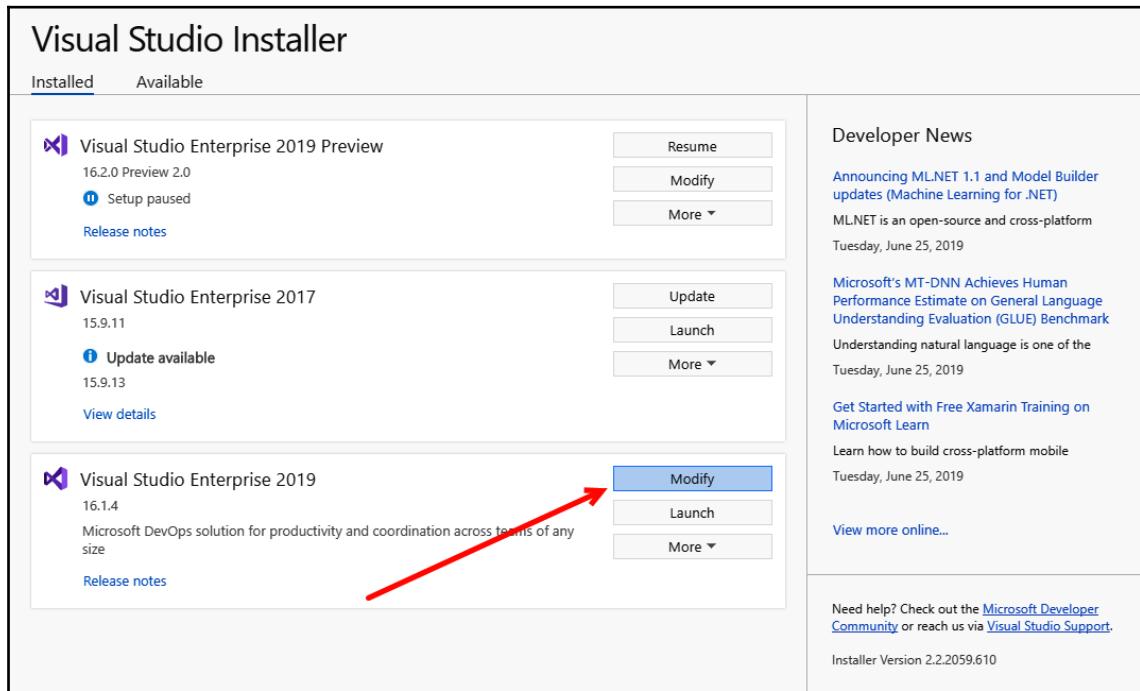
- Microsoft.VisualStudio.Workload.NetCrossPlat: To build cross-platform applications for iOS, Android, or Windows using Xamarin, you will need to have this workload installed on your development environment.
- Microsoft.VisualStudio.Workload.NetWeb: You can build web applications using ASP.NET, ASP.NET Core, HTML, JavaScript, and CSS using the NetWeb workload.
- Microsoft.VisualStudio.Workload.Node: To build scalable network applications using Node.js and an asynchronous event-driven JavaScript runtime, you will need this workload.
- Microsoft.VisualStudio.Workload.Office: To create Office and SharePoint add-ins, SharePoint solutions, and VSTO add-ins using C#, VB, and JavaScript, you will need this Office workload.
- Microsoft.VisualStudio.Workload.Universal: To create applications targeting the Universal Windows Platform with C#, VB, JavaScript, or C++ (optional), you need to install this workload.
- Microsoft.VisualStudio.Workload.VisualStudioExtension: If you want to create add-ons and extensions for Visual Studio, you will need to install this workload. This also includes new commands, code analyzers, and tool windows.
- Microsoft.VisualStudio.Workload.WebCrossPlat: To build Android, iOS, and UWP apps using Tools for Apache Cordova, you will need this workload.

Each of the preceding workloads has its own set of components, which you can refer to by going to Microsoft's official page: <https://bit.ly/vs2019componentids>.

Modifying your existing Visual Studio 2019 installation

If you decide to modify the existing installation to add/or remove any components or uninstall the installation, you can do so from the control panel by going to **Add/Remove Programs**.

Alternatively, you can launch the Microsoft Visual Studio Installer and click on the **Modify** button, as shown in the following screenshot:

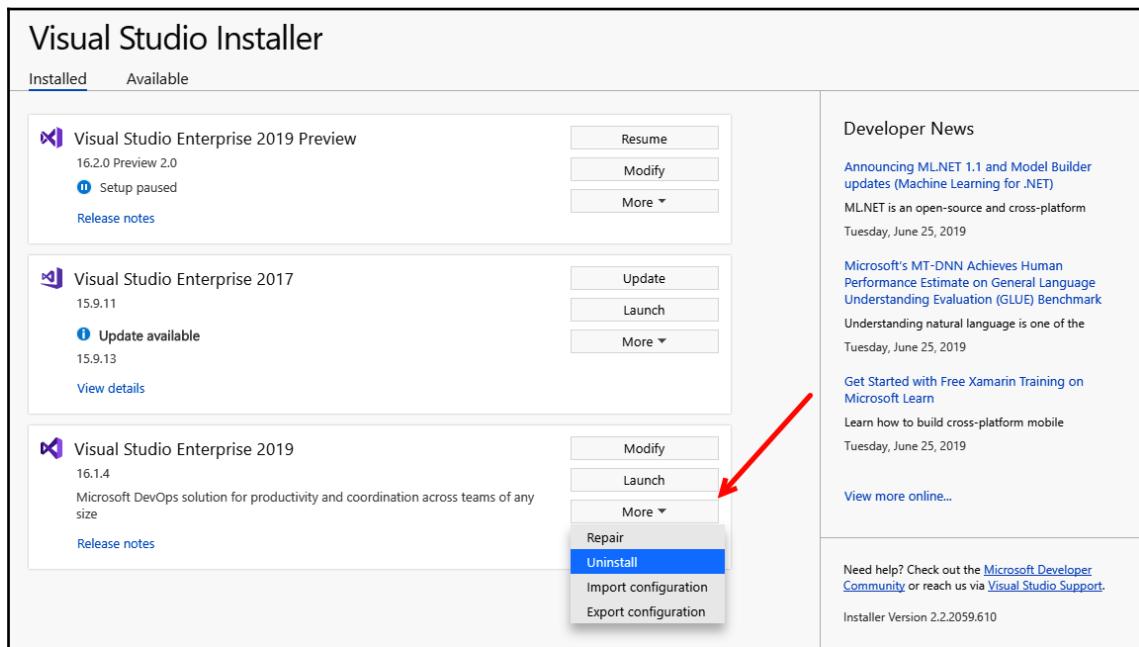


To add new modules, check the new workflow(s). To remove modules, uncheck the existing workflow(s) and continue modifying the existing installation.

Uninstalling Visual Studio 2019

In case you set your mind on uninstalling all of the packages that Visual Studio 2019 installed, the new installer that comes with it can help you to completely uninstall the packages without keeping any traces of the components.

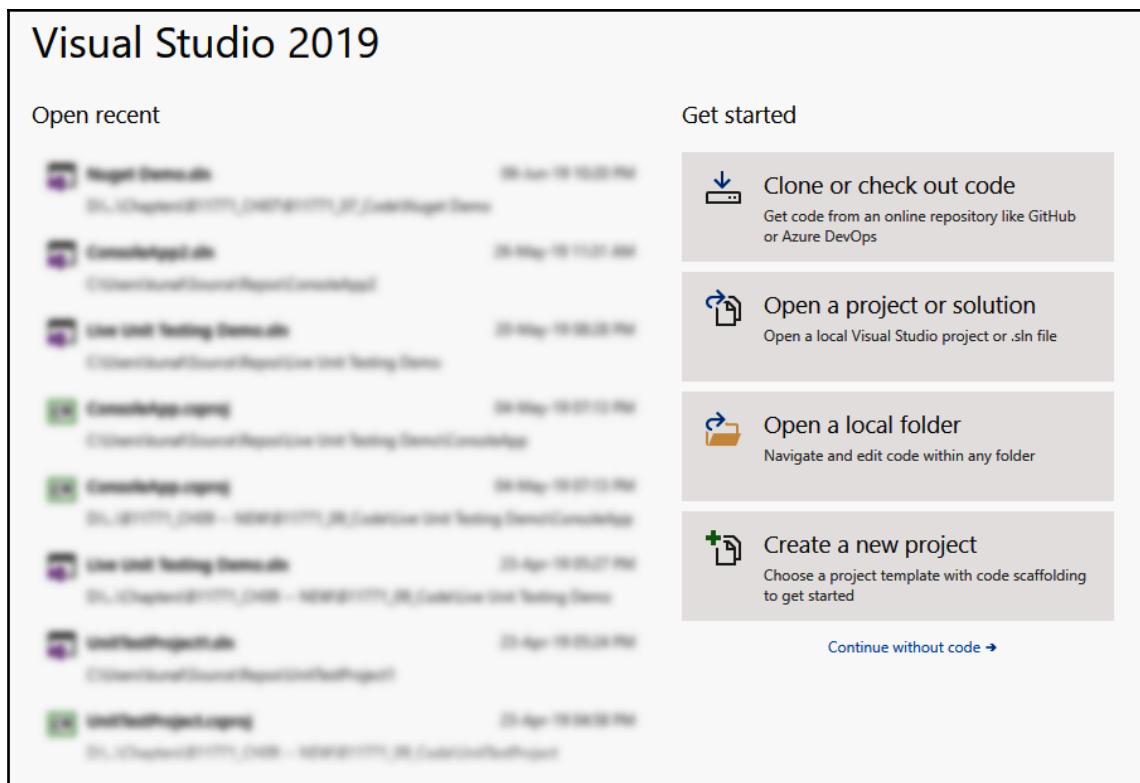
You can also launch the Microsoft Visual Studio Installer by clicking the **More** drop-down menu and then clicking **Uninstall**, as shown in the following screenshot. Then, you can click on **OK** when asked:



Now, we will move on and understand the other updates that are available in Visual Studio 2019.

Overview of the new Start window

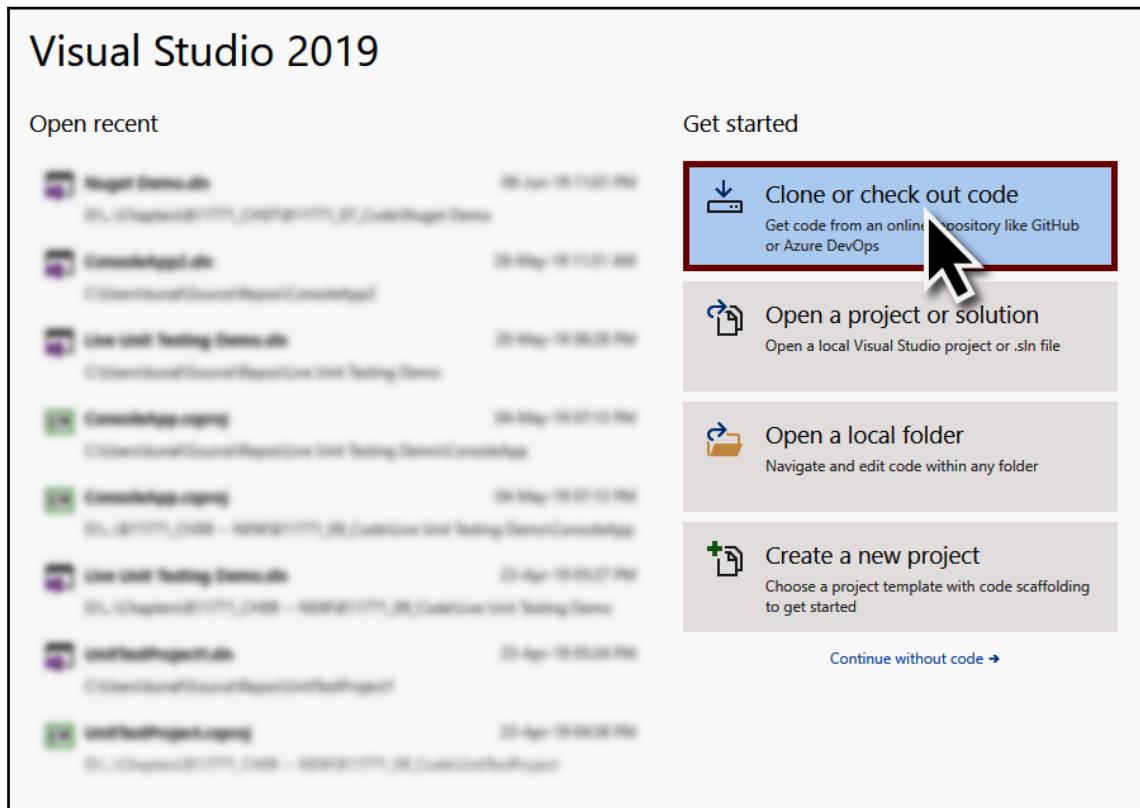
When you launch Visual Studio 2019, the new Start window is the first thing that you will see. It offers you a focused experience to get you started with the IDE. Here's a screenshot of the all-new Visual Studio 2019 Start window:



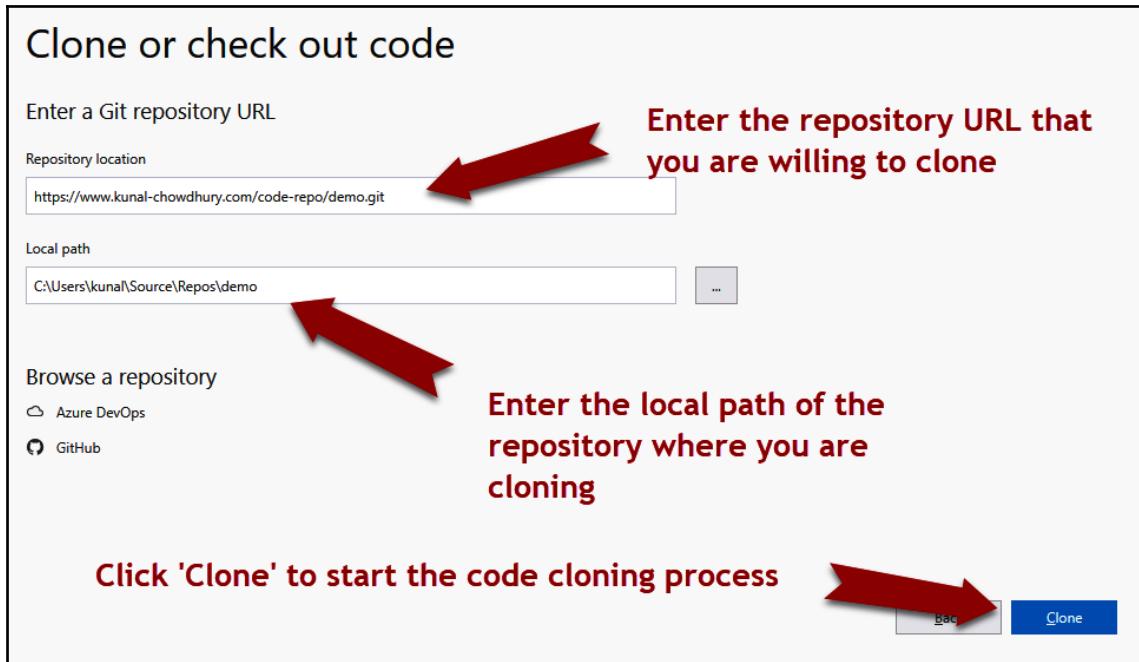
Two sections are available. The left-hand section lists the recent activities and allows you to open recent projects/solutions. It's the most common way a developer opens the code. The right-hand section allows you to perform some quick operations, such as cloning a repository, opening an existing project or solution, creating a new project, or opening the IDE without using any code.

Cloning a repository from the Start window

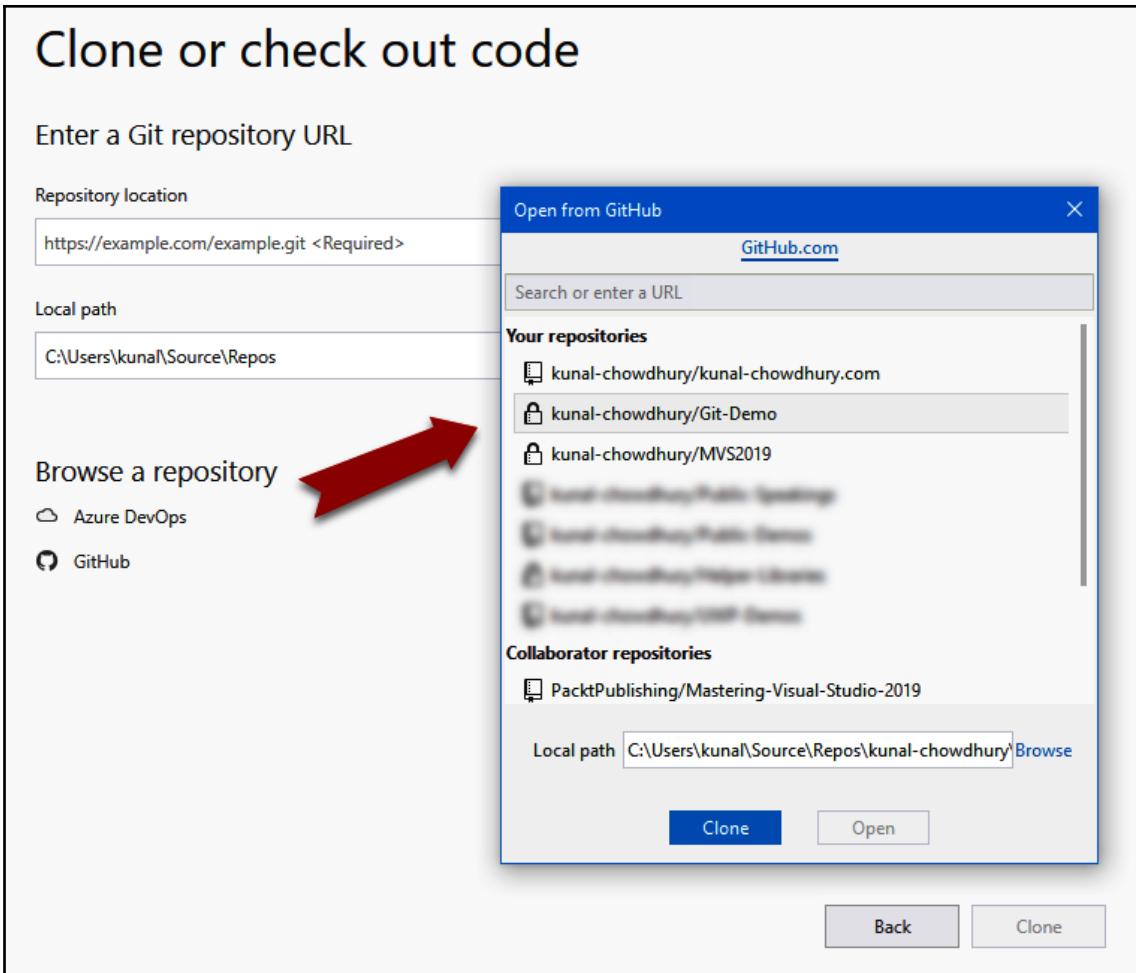
With Visual Studio 2019, it's very easy to clone an existing source control repository to a local path. You can achieve this from the Start window. To begin, click on the **Clone or check out code** navigation link, as shown in the following screenshot:



This will open another dialog window, which will help you to select the remote repository that you would like to clone locally. If you know the remote URL, as shown in the following screenshot, enter the URL in the **Repository location** field, select the **Local path**, and click the **Clone** button:



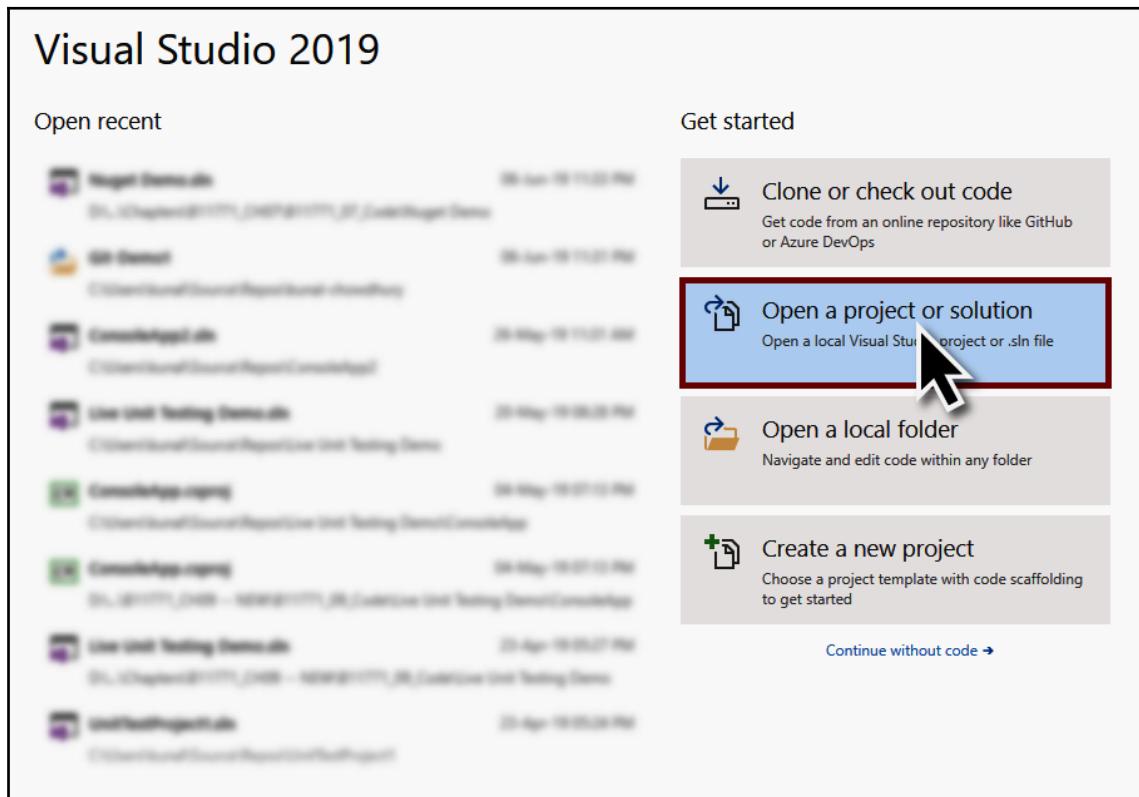
The Start window also allows you to browse well-known repositories, such as **Azure DevOps** or **GitHub**. Select the desired repository option, authenticate yourself (when asked), select the repository that you want to clone, and then hit the **Clone** button:



Once you hit the **Clone** button, the Visual Studio 2019 IDE will be launched, and the process to clone the code files will be triggered. It will take some time, based on the repository's size, to complete the process.

Opening an existing project or solution from the Start window

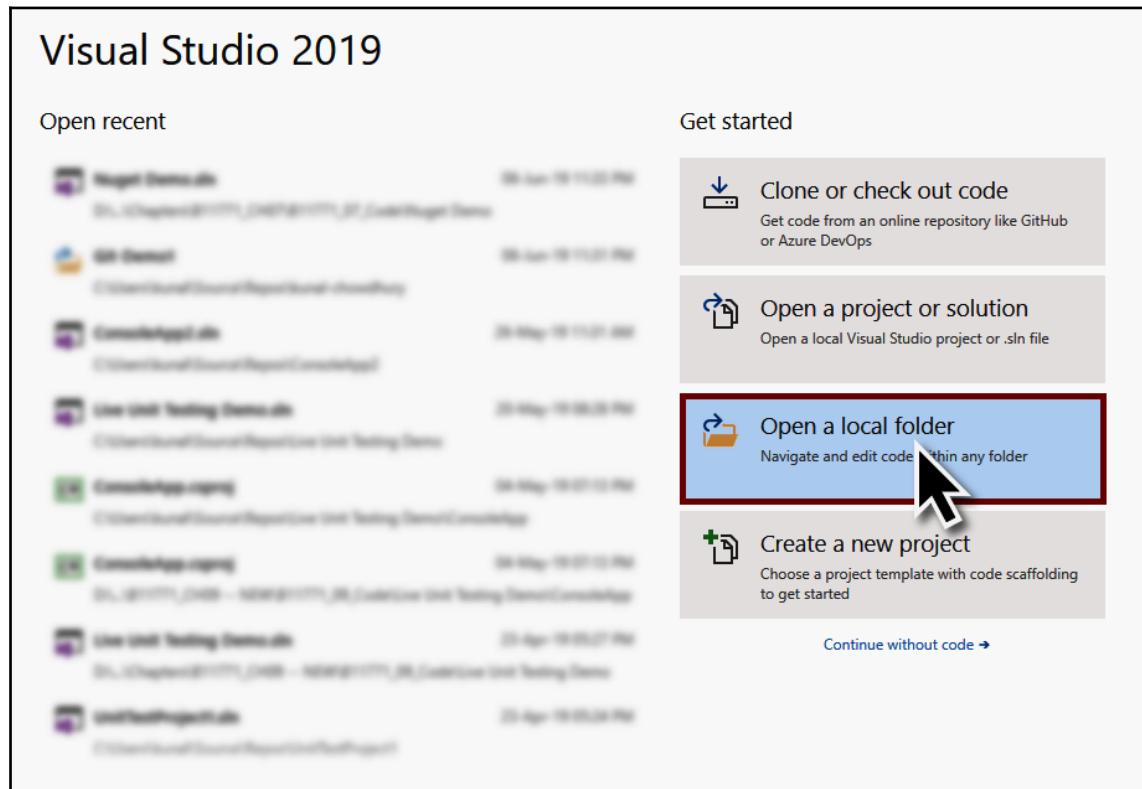
Most of the time, a developer opens an existing project that he/she is working on. The new Visual Studio 2019 Start window provides you with easy access so that you can select the desired project or solution file. To do this, click on the **Open a project or solution** navigation button, as shown in the following screenshot:



Then, navigate through your local filesystem and select the desired project/solution file to open it.

Opening a local folder from the Start window

Though not a common thing to do, the Start window allows you to open a local folder. This is often useful when you want to browse, edit, build, and debug any code files that don't have a project or solution file:

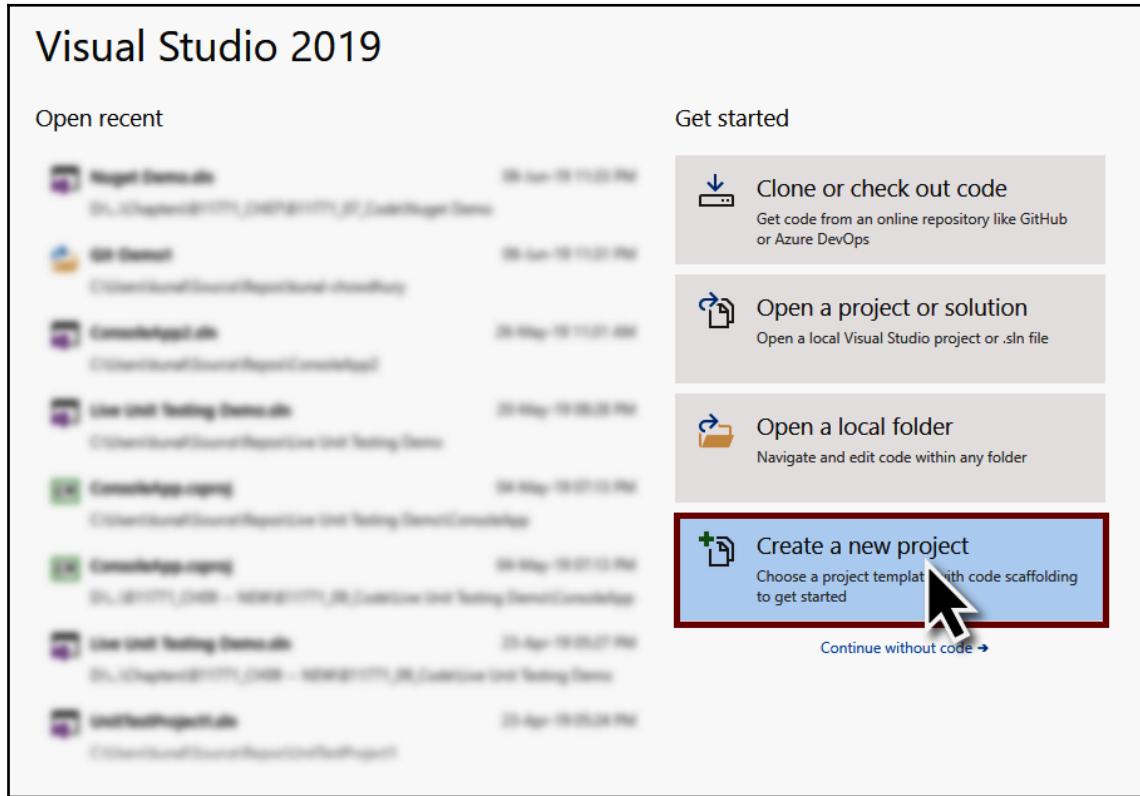


You can also open a local folder and pick up the file from the folder view of the **Solution Explorer** in Visual Studio 2019.

Creating a new project from the Start window

The Start window of Visual Studio 2019 also allows you to directly create a project. The new project creation experience is fast and easy enough to provide you with the right project template so that you can begin writing your code.

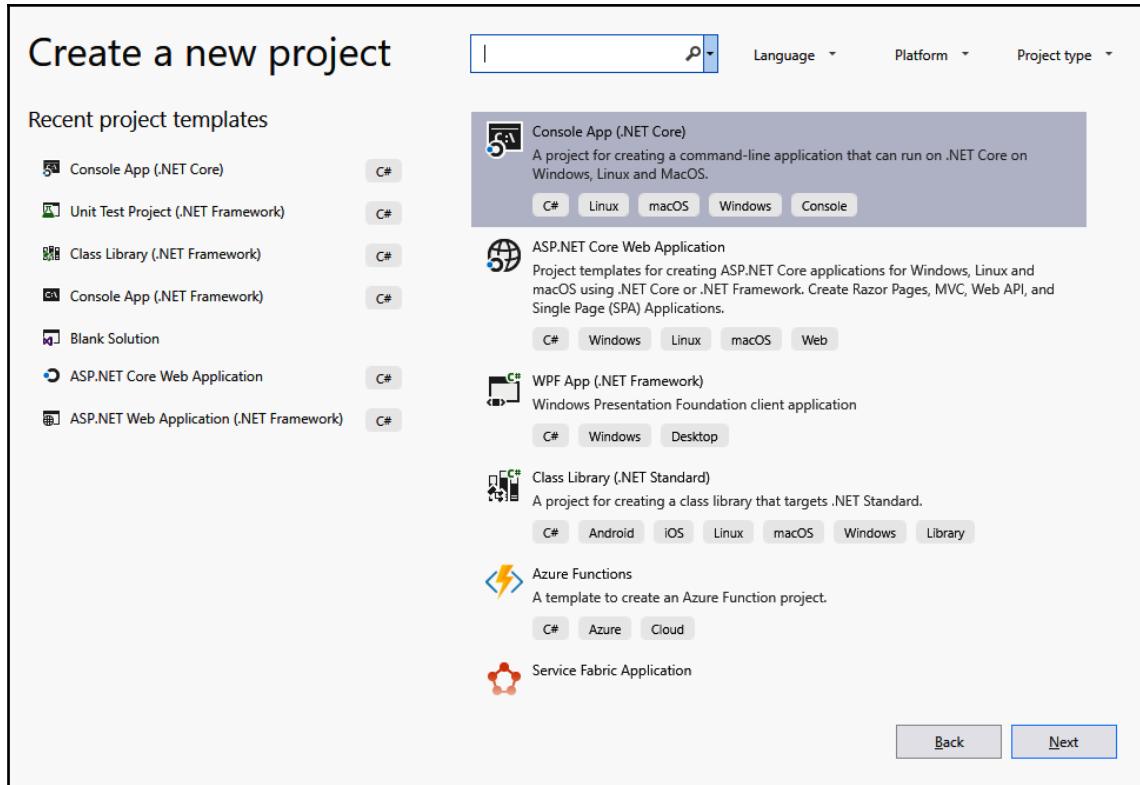
As shown in the following screenshot, click on the **Create a new project** button to begin with the selection of the project template:



The screen that follows will allow you to select the project template. Microsoft removed the old language-centric tree hierarchy here and has given priority to the option of selecting the application project type first. Based on the installed workload, the most commonly used project type will be at the top of the list.

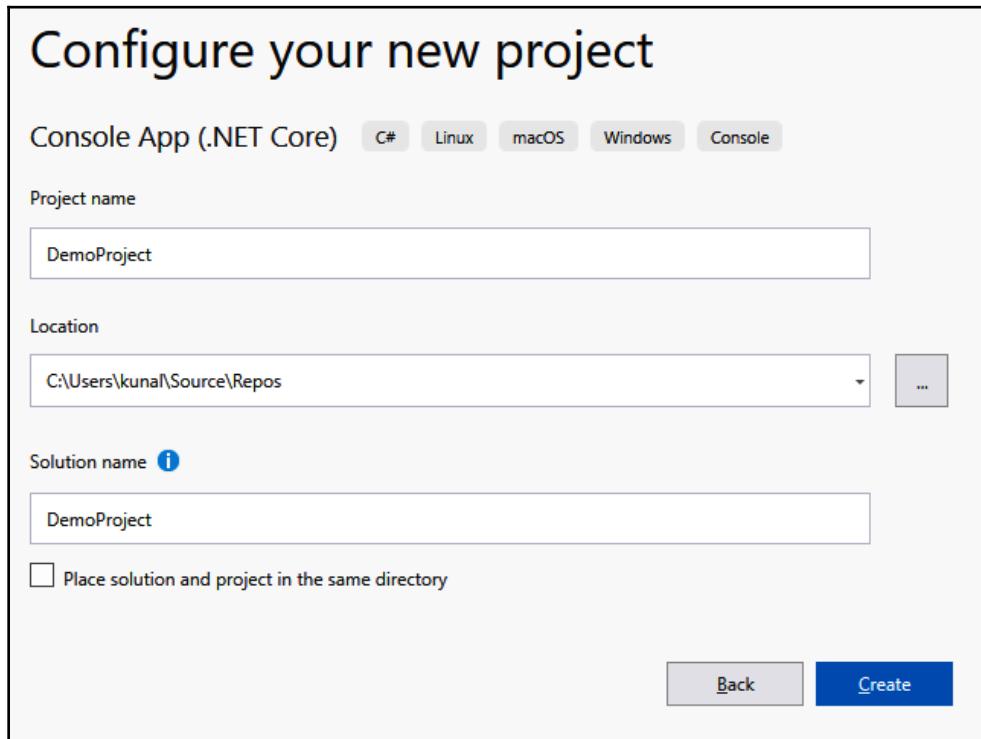
You can also find a **Most Recently Used (MRU)** list of recent project templates that you have accessed and filter the templates based on search terms, language, platform, and project type.

Here's a sneak peek of the new project dialog window of Visual Studio 2019:



Once you have selected the project template, clicking on the **Next** button will navigate you to a new screen, where you will be allowed to set a name for the project and solution and select the project's location. It also provides you with the option to place the project and solution in the same directory.

Once you are ready, click on the **Create** button to start creating the project based on the selected template:



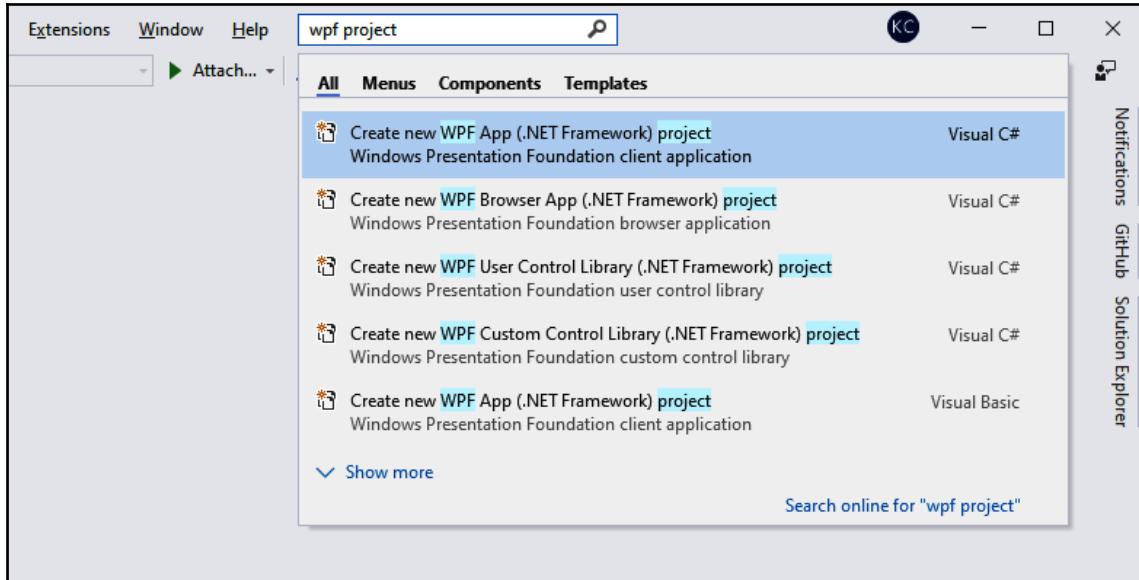
Once you are within the IDE, you can navigate to **File | Start Window** to relaunch the Start window.

Improved search in Visual Studio 2019

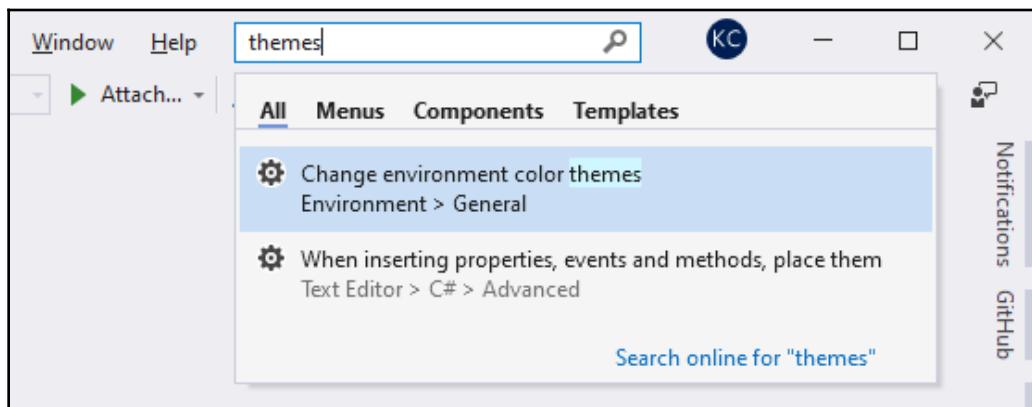
You may have already experienced the Visual Studio **quick launch** bar for searching within Visual Studio commands, menus, and so on. With the release of Visual Studio 2019, Microsoft has improved that search experience. It now gives results faster and more effectively.

To invoke the search, click on the **Search Visual Studio** input box (or use *Ctrl + Q*) and enter the search term. A drop-down menu will appear with the available commands, menus, settings, templates, and so on and will be populated dynamically, based on what you are typing. Select the one that is the most suitable for you.

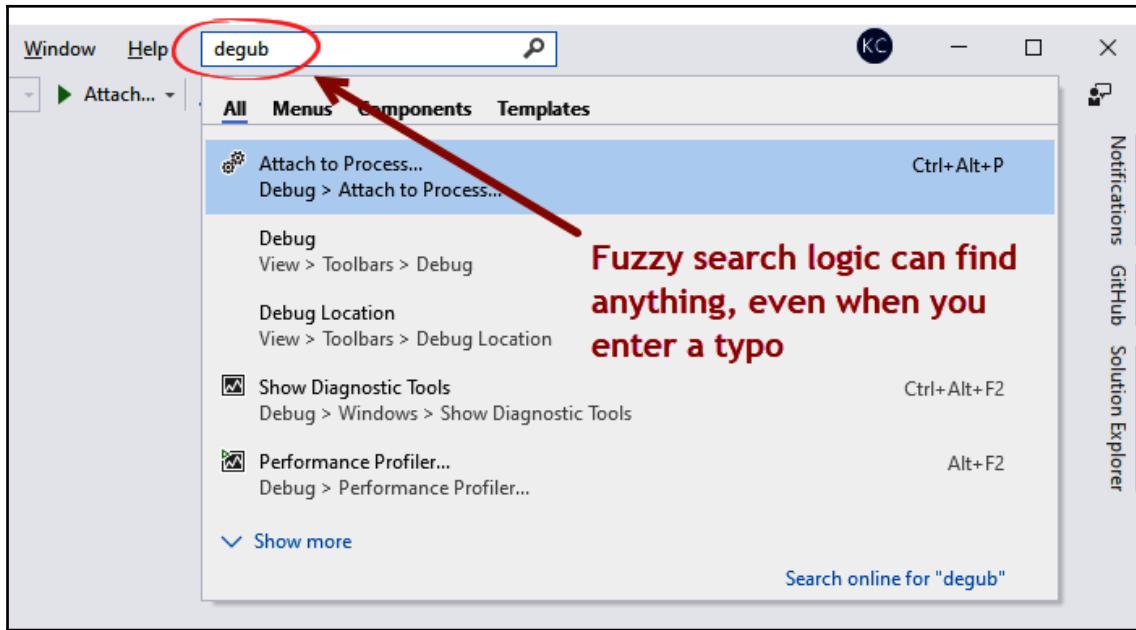
For example, if you want to create a new WPF project, enter wpf project in the search box. This will display the available templates with proper highlighting of the searched term, as shown in the following screenshot:



Similarly, if you are looking for any settings (let's say, themes), just enter the search term. It will display the available settings:



You can further drill down to it by navigating to either **Menus**, **Components**, or **Templates** from the drop-down. Interestingly, the new fuzzy search logic can find what you are looking for, even when you enter a typo. The following screenshot shows how it displays the correct search result when you search for `degub` instead of `debug`:



When you want to install a missing component, you can also invoke this search and directly open the installer. Give it a try from your end and find out how easy it is! Due to this, the new search feature in Visual Studio 2019 helps you to easily find what you are looking for.

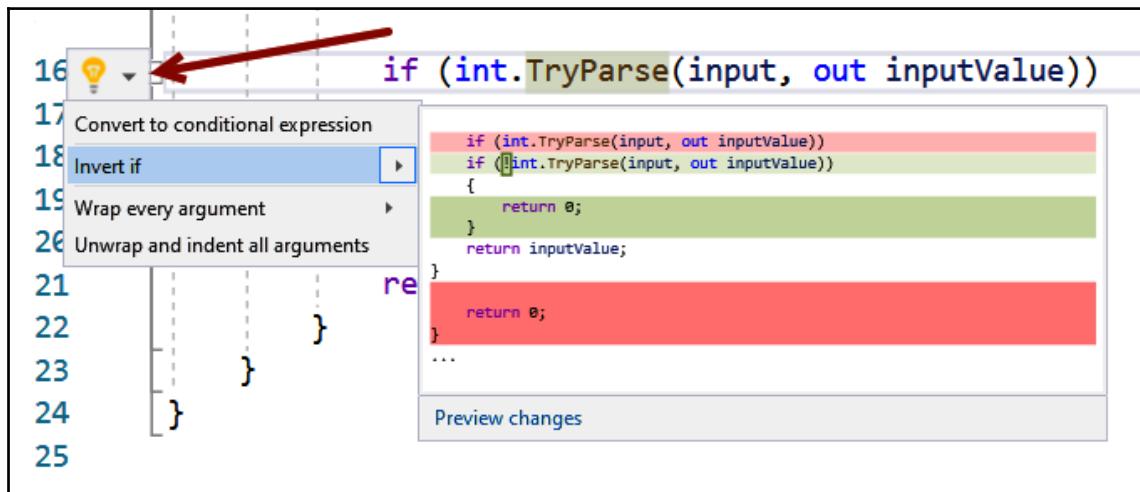
Quick actions improvements for code refactoring

You might already know about the **quick actions**. This is a tiny light bulb or a screwdriver icon that pops up on the left-hand side of the Visual Studio IDE, which allows you to apply quick actions so that you can refactor and optimize your code by displaying some suggestions. With Visual Studio 2019, Microsoft has improved it by adding more suggestions for code refactoring.

When you see a light bulb on the current line, click on the bulb or just press *Ctrl + .* (dot) to see the list of suggestions. Select the one that you would like to apply to the current code block.

Inverting if statements

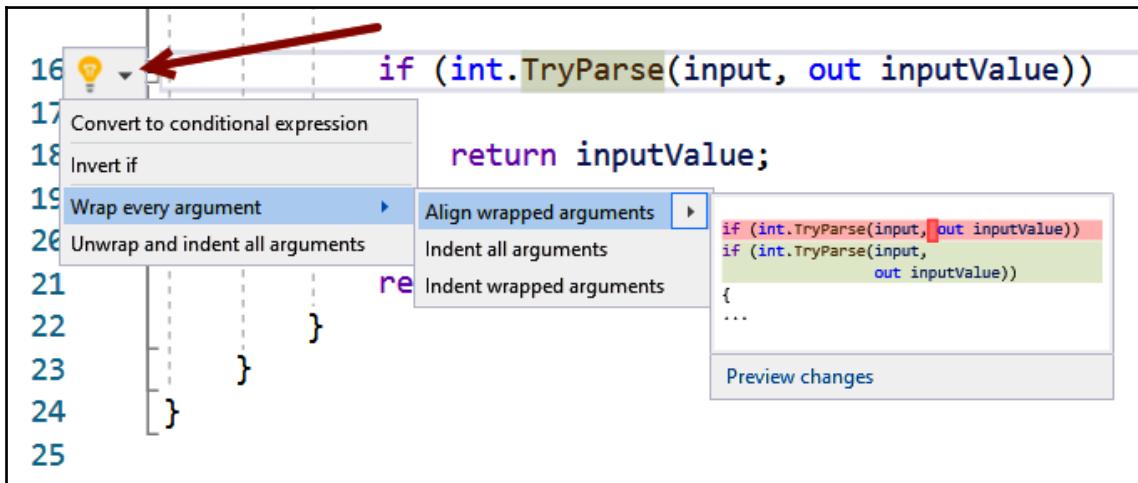
Visual Studio 2019 now suggests when you should invert an *if* condition. When you hover over the suggestion, it gives you a preview of the changes, as shown in the following screenshot:



You can accept the changes that are suggested by Visual Studio 2019 if they satisfy your needs.

Wrapping arguments

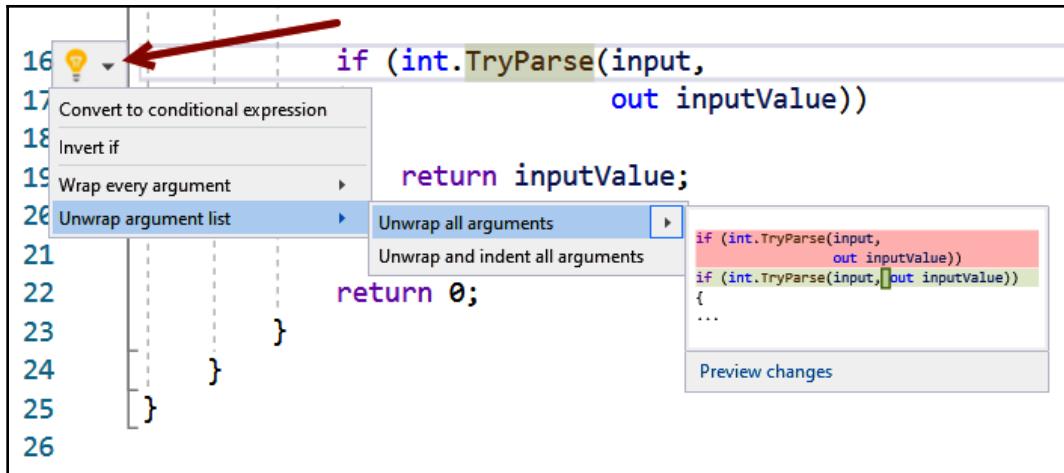
When a method accepts more than one argument, it is sometimes useful to place the arguments in separate lines to increase the visibility of the code within a single page. Visual Studio 2019 now suggests that you wrap every argument and align arguments properly. Here's a screenshot for reference:



As shown in the previous screenshot, you can wrap the arguments in three different ways: **Align wrapped arguments**, **Indent all arguments**, and **Indent wrapped arguments**.

Unwrapping arguments

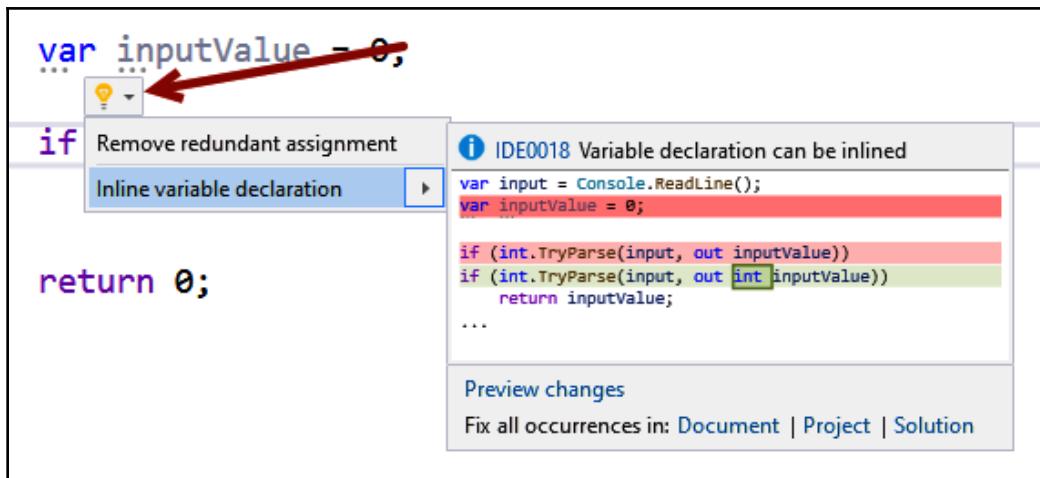
You can also unwrap the argument list and place it in a single line using the quick actions menu, as shown here:



It supports the use of **Unwrap all arguments** and **Unwrap and indent all arguments**.

Introducing inline variables

Visual Studio 2019 also suggests that you modify your code to include inline variables, which was first introduced with C# 7 in Visual Studio 2017. The following screenshot demonstrates how easy it is to automatically refactor your code to introduce inline variables instead of separately defining it before passing it as an argument:



You can use this to refactor it and reduce the lines of code that you use.

Pulling a member to its base type

You can now easily pull any member to its base class. When you click on a member, Visual Studio 2019 will suggest moving that member to its base class (if it exists). Here's a screenshot for reference:

The screenshot shows a code editor with two classes defined:

```
4 {  
5     class Person  
6     {  
7     }  
8  
9     class Employee : Person  
10    {  
11        public string GetName()  
12    }
```

A red arrow points from the text "Pull 'GetName' up to 'Person'" in the context menu to the `GetName()` method signature in the code.

The context menu options shown are:

- Replace 'GetName' with property
- Use expression body for methods
- Pull 'GetName' up to 'Person'** (highlighted)
- Pull members up to base type...** (highlighted)

A preview window on the right shows the resulting code after the refactoring:

```
{  
}  
...  
{  
    public string GetName()  
    {  
        return string.Empty;  
    }  
}  
...  
{  
}  
...
```

At the bottom of the preview window is a "Preview changes" button.

You can either select **Pull '[MEMBER_NAME]' up to '[CLASS_OR_INTERFACE_NAME]'** or **Pull members up to base type...**, where the second choice will give you more options to select.

Adjusting namespaces to match the folder structure

Visual Studio 2019 suggests that you adjust namespaces to match the folder structure. For example, if you have a class under the `Models.Core` folder but the namespace doesn't include the structural namespace, you can easily change it, as shown in the following screenshot:



This is often useful for refactoring your code and keeping it in a structural manner.

Converting foreach loops into generalized LINQ queries

The new Visual Studio 2019 IDE also suggests that you refactor your `foreach` loop and convert it into a generalized LINQ statement. Here's how it modifies the code when you click on the **Convert to LINQ** menu:

```
14 foreach (var item in items)
15     Use explicit type instead of 'var'
16 Convert to LINQ >
17 Convert to LINQ (call form)
18 Convert to 'for'
19
20 }
21 }
22 }
23 }
24 }
```

```
using System.Text;
using System.Linq;
...
var items = new List<string>();
foreach (var item in items)
{
    if (item.Equals("ERROR"))
        foreach (var _ in from item in items
                           where item.Equals("ERROR")
                           select new {} // code logic
                           )
    {
        // code logic
    }
}
```

Preview changes

Here's how it modifies the code when you click on the **Convert to LINQ (call form)** menu:

The screenshot shows a code editor with several numbered lines (13-24) and a context menu open at line 11. The menu items are:

- Use explicit type instead of 'var'
- Convert to LINQ
- Convert to LINQ (call form)** (highlighted with a red box)
- Convert to 'for'

The code being refactored is as follows:

```
13
14 foreach (var item in items)
15 Use explicit type instead of 'var'
16 Convert to LINQ
17 Convert to LINQ (call form) ▾
18 Convert to 'for'
19
20     var items = new List<string>();
21
22     foreach (var item in items)
23     {
24         if(item.Equals("ERROR"))
25             foreach (var item in items.Where(item => item.Equals("ERROR")).Select(item => new { } // code logic
26         {
27             // code logic
28         }
29     }
30 }
```

A red box highlights the 'Convert to LINQ (call form)' option in the context menu. A green box highlights the 'using' statements and the first line of the converted LINQ code. A pink box highlights the entire converted LINQ block. A blue box highlights the 'Preview changes' button at the bottom.

This isn't all you can do, though—you can also convert a LINQ query into a `foreach` statement using Visual Studio 2019. Try it out and see how it works.

Visual Studio IntelliCode

Visual Studio IntelliCode is an **artificial intelligence (AI)** assisted IntelliSense for Visual Studio that enhances software development by delivering context-aware code completions. IntelliCode suggestions appear at the top of the completion list with a star icon next to them, which gives you easy access to the APIs within that context without scrolling through the entire list.

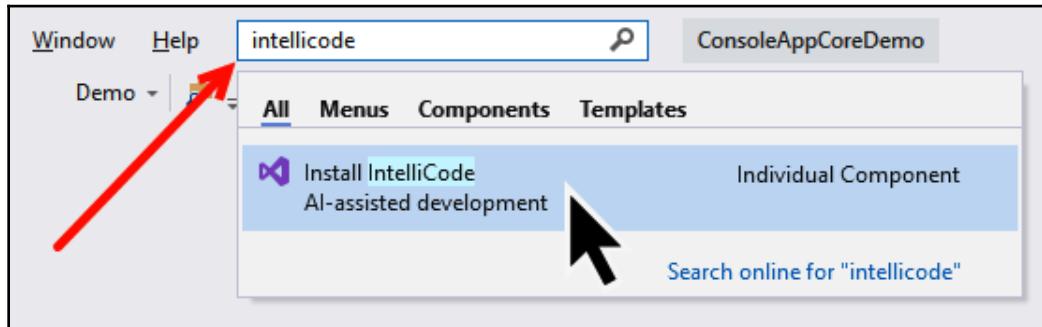


Visual Studio IntelliCode is generally available with Visual Studio 2019 version 16.1, which was released on 21 May 2019.

Currently, the following languages are supported by the IntelliCode feature: C#, XAML, C++ (Preview), JavaScript, and TypeScript (Preview). There are around 2,000 open source projects on GitHub, each of which has more than 100 stars to generate the recommendations for Visual Studio IntelliCode. You need to train your IntelliCode model to get the AI-assisted IntelliSense recommendations.

Installing the IntelliCode component

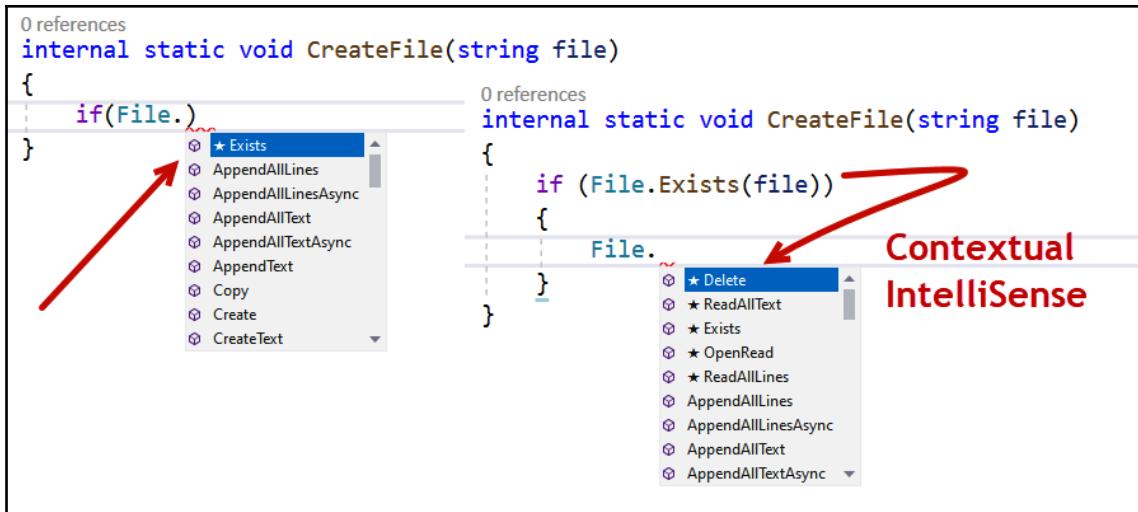
Visual Studio IntelliCode comes as a separate component and is not part of any workload. To install the component from the Visual Studio 2019 IDE, search for `intellicode` in the Quick Search box, as shown in the following screenshot:



Click on the **Install IntelliCode** link to start the Visual Studio 2019 installer. Follow the instructions to complete the installation. Once done, click the **Close** button. You may need to restart Visual Studio 2019 for the changes to take effect.

Understanding how IntelliCode helps

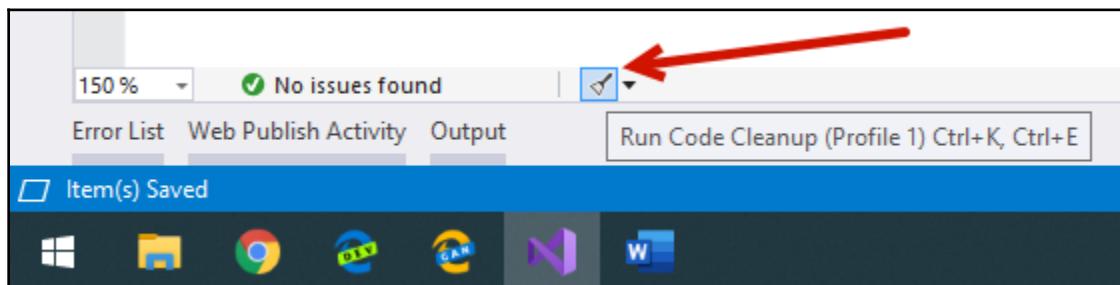
IntelliCode helps by adding context-aware code completions to your project so that you can focus on your code instead of searching and scrolling through the IntelliSense menu. It adds a * (star) symbol to the IntelliCode items, as shown in the following screenshot:



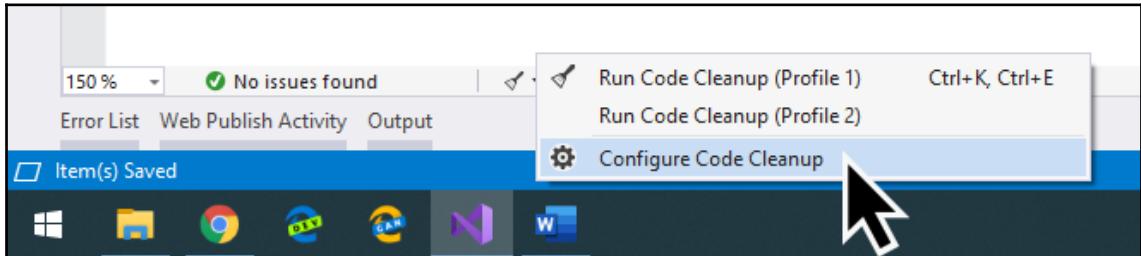
In the preceding example, since we have already inserted the `if` condition to check whether the file exists, on the next line, it adds common file operations such as **Delete**, **ReadAllText**, **OpenRead**, and **ReadAllLines** to the menu. Hence, common operations are within your focus to select so that you can complete your code easily.

One-click code cleanup

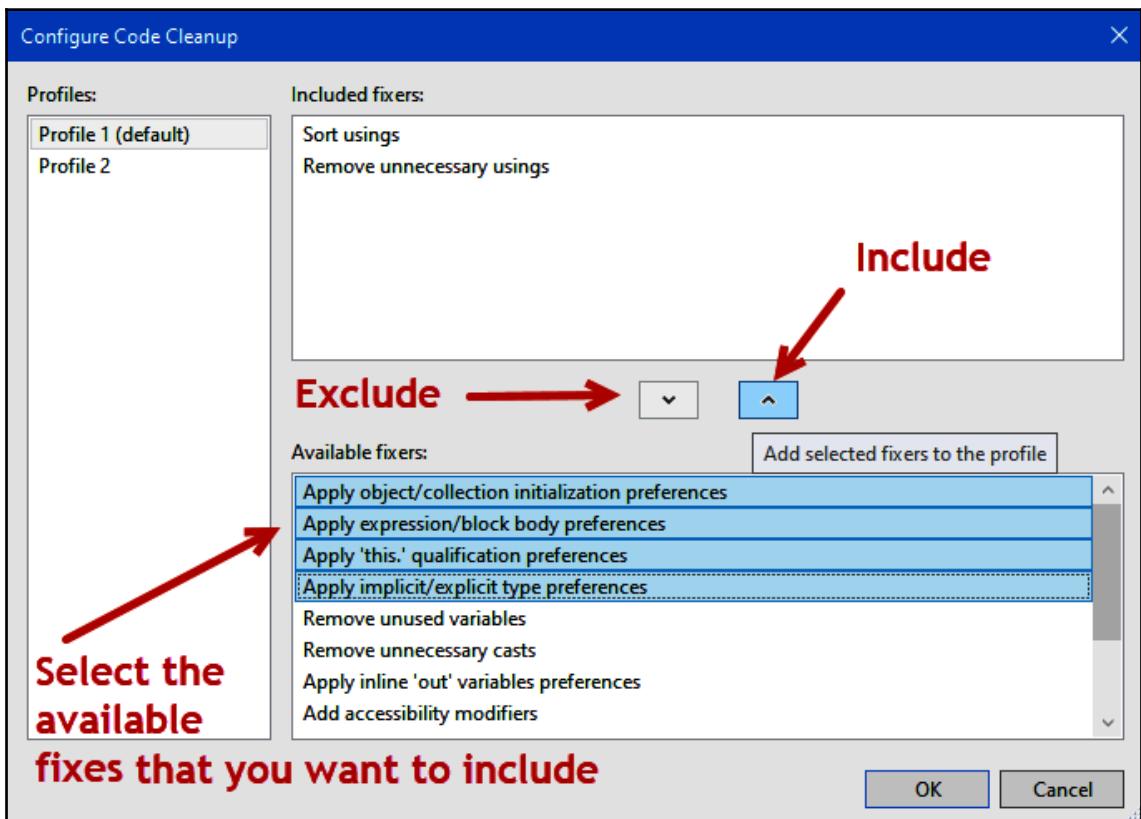
Visual Studio 2019 introduces a new one-click code cleanup option to the bottom-left of the IDE. As shown in the following screenshot, clicking on the code cleanup icon will execute the predefined rules set in the solution and perform the cleanup:



You can also configure what rules you would like to include/exclude in the one-click cleanup operation. To do this, click on the arrowhead and then click the **Configure Code Cleanup** menu item, as shown in the following screenshot:



This will open the following dialog window, where you will be allowed to select/deselect the rules to include or exclude from the one-click cleaning operation:

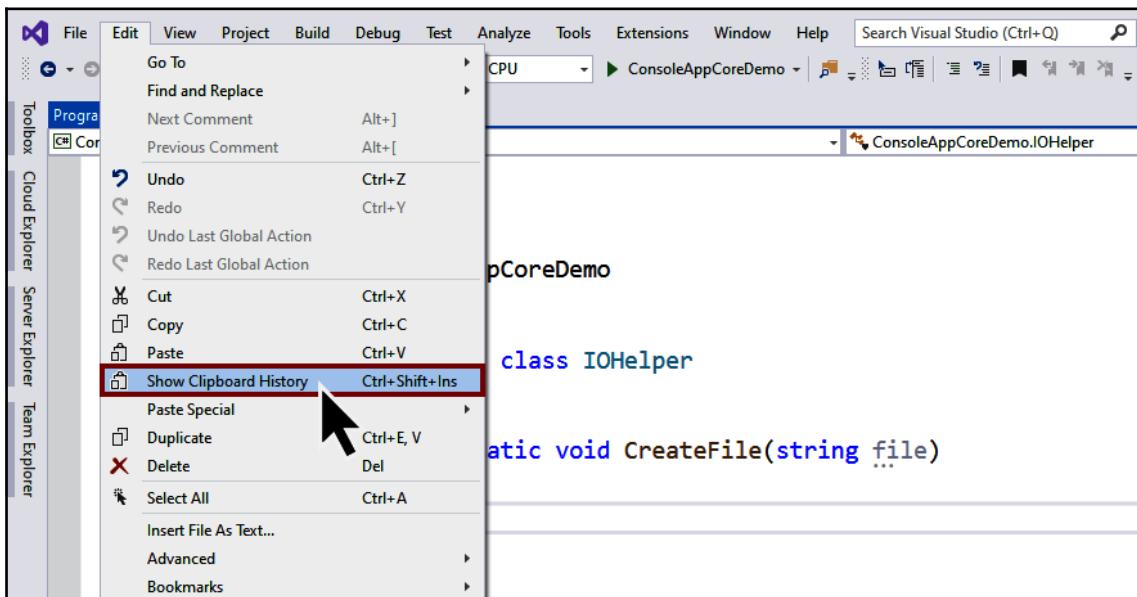


To include a new rules set, select the **Available fixers** and click the include button. To exclude the existing rules set, select the **Included fixers** and click the exclude button. Once done, click the **OK** button to save the changes.

Clipboard History

With the launch of Visual Studio 2017, a **Clipboard Ring** was added that allows you to cycle through your clipboard and paste the code block that you want to insert from your history. In Visual Studio 2019, Microsoft changed the behavior of the cycling ring. Instead of having that, they added **Clipboard History**, which allows you to select copied content from the displayed list.

When you have multiple pieces of copied content/code and you want to insert any of them in the editor window, either press *Ctrl + Shift + Ins* or navigate to the **Edit | Show Clipboard History** menu item, as shown in the following screenshot:



As shown in the following screenshot, this will display a list of copied content that's available in the **Clipboard**:

The screenshot shows a code editor window with the following code:

```
0 references
internal static class IOHelper
{
    0 references
    internal static void CreateFile(string file)
    {
        }
    }
}

if(File.Exists(file))
{
    ...
}
```

A context menu is open at the bottom of the code editor, titled "Clipboard". It contains three items:

- 1: internal-static-void>CreateFile(string-file) ...
- 2: if-(File.Exists(file)){
- 3: Click-on-the-...

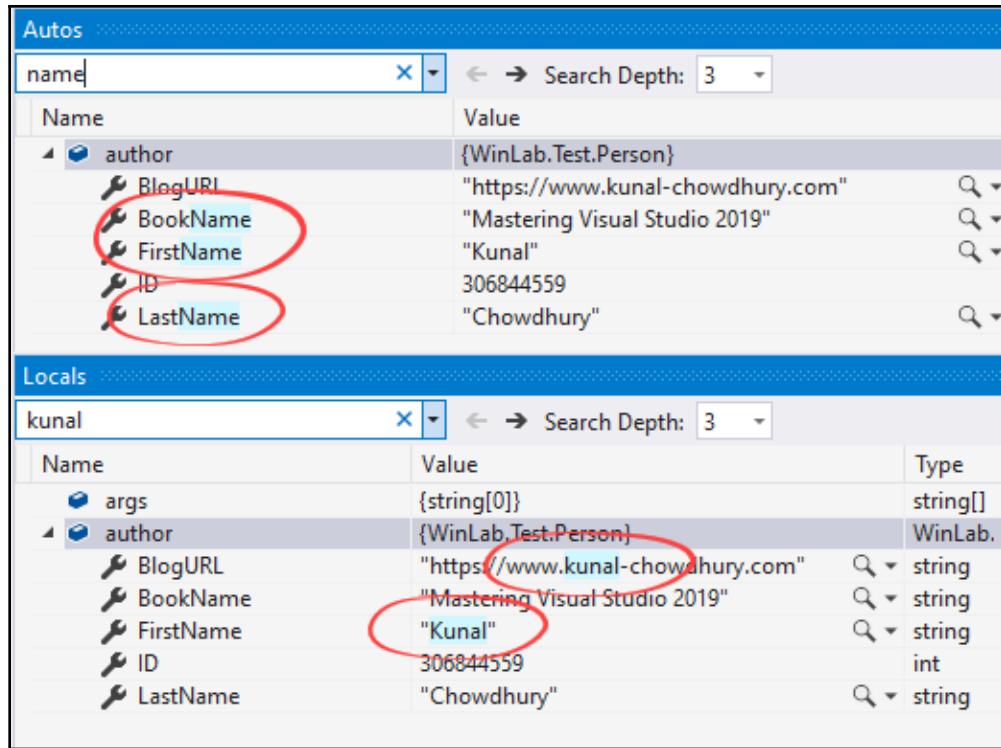
A red arrow points from the text "Select the copied content that you want to paste from the Clipboard History" to the second item in the list.

Select the copied content that you want to paste from the Clipboard History

Select the one that you want to insert. A preview of the content will be available when you hover over the **Clipboard** item.

Searching in debugging windows

Visual Studio 2019 has added new capabilities inside the Autos, Locals, and Watch windows to help you to find the objects, properties, and values that you are looking for. Here's a screenshot of how it displays the searched term:



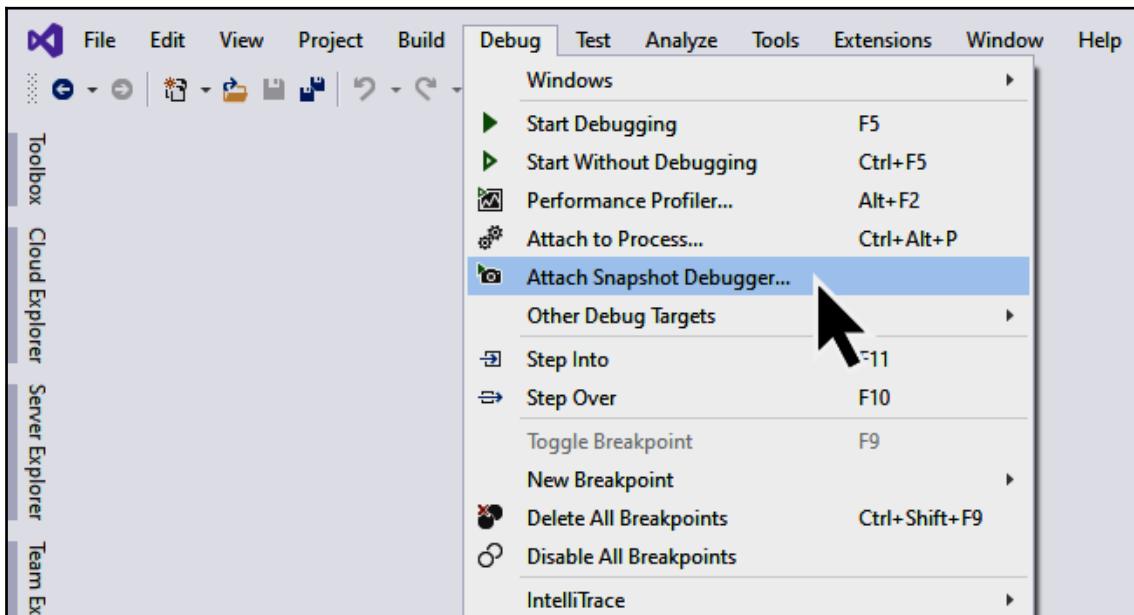
It also allows you to define the search depth. You can set it by selecting the depth level from the drop-down menu.

Time Travel Debugging

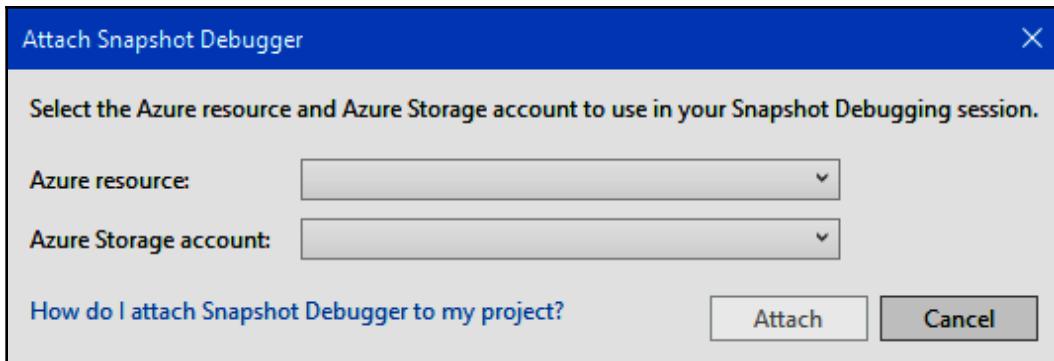
Time Travel Debugging allows you to record the execution of code in an application and replay it, either forward or backward, so that you can understand the condition that leads to a specific bug. Visual Studio 2019 Enterprise Edition added this feature to help developers perform reverse debugging on a project hosted on the cloud.

To perform Time Travel Debugging, follow these steps:

1. Open the project in Visual Studio 2019 Enterprise Edition. Make sure that you have the same version of the source code that has been deployed to your Azure Virtual Machine.
2. From the Visual Studio editor, select the **Debug | Attach Snapshot Debugger...** menu, as shown in the following screenshot:



3. This will open the following **Attach Snapshot Debugger** window:



4. Select the desired **Azure resource** and **Azure Storage account** and click the **Attach** button to continue.
5. Create a snap point by selecting the code that needs to be debugged and then clicking **Start Collection**.
6. When it completes creating the snap point, you can enable Time Travel Debugging for it.
7. Once done, click **View Snapshot**.

Now, you will be allowed to step forward and backward in order to debug the collected snapshot and find the issue in your code.

Live Share

Microsoft has made it easy for developers to collaborate with each other by sharing a **Live Share** session within Visual Studio 2019. If you are using Visual Studio 2017, you must download the extension from the Visual Studio Marketplace.

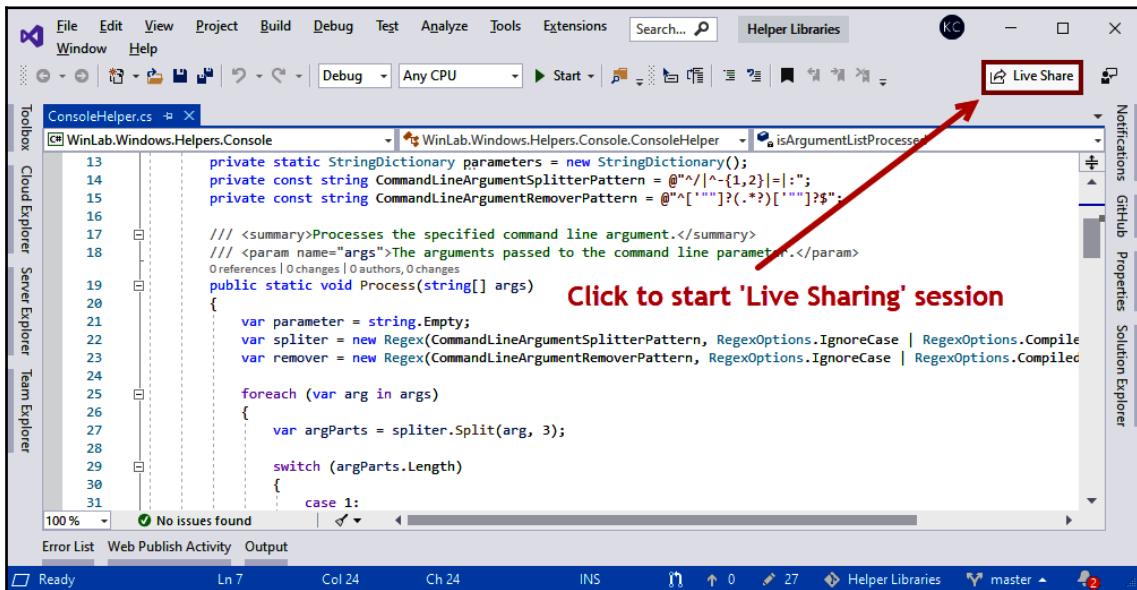
When you want to collaborate with your team members, the Live Share feature doesn't care about what type of application you are building and which operating system you are currently working on. Just focus on your work and co-edit and co-debug the code.

When team members need to collaborate on work from different locations, you can create a live sharing session and the other member(s) can join it by launching the session link.

Creating a new live sharing session

To create a new live sharing session and invite other members, follow these steps:

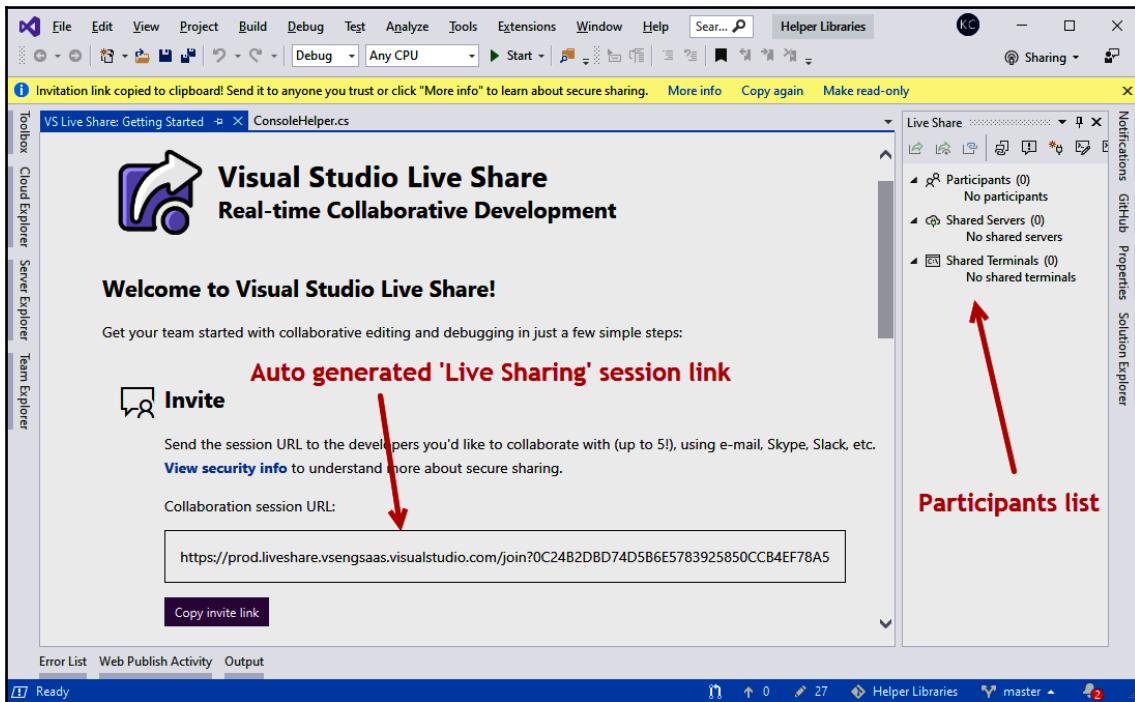
1. At the top-right corner of the Visual Studio 2019 IDE, you will find a button called **Live Share**. Click on it to begin the sharing process:



2. A message (**Visual Studio Live Share Firewall Access**) will pop up, informing you about firewall access, which you must accept to continue and start/join the live sharing session. Click on the **Ok** button on this screen to launch the firewall prompt:



3. Windows Security Alert will ask you to approve the firewall permission. Click on the **Allow access** button to continue.
4. After a few moments, it will begin the final steps and create a live session. A live sharing link will be generated, which you can share with your team members so that they can join the session.
5. As shown in the following screenshot, a **Live Share** panel will display the list of participants, shared servers list, and shared Terminal list:



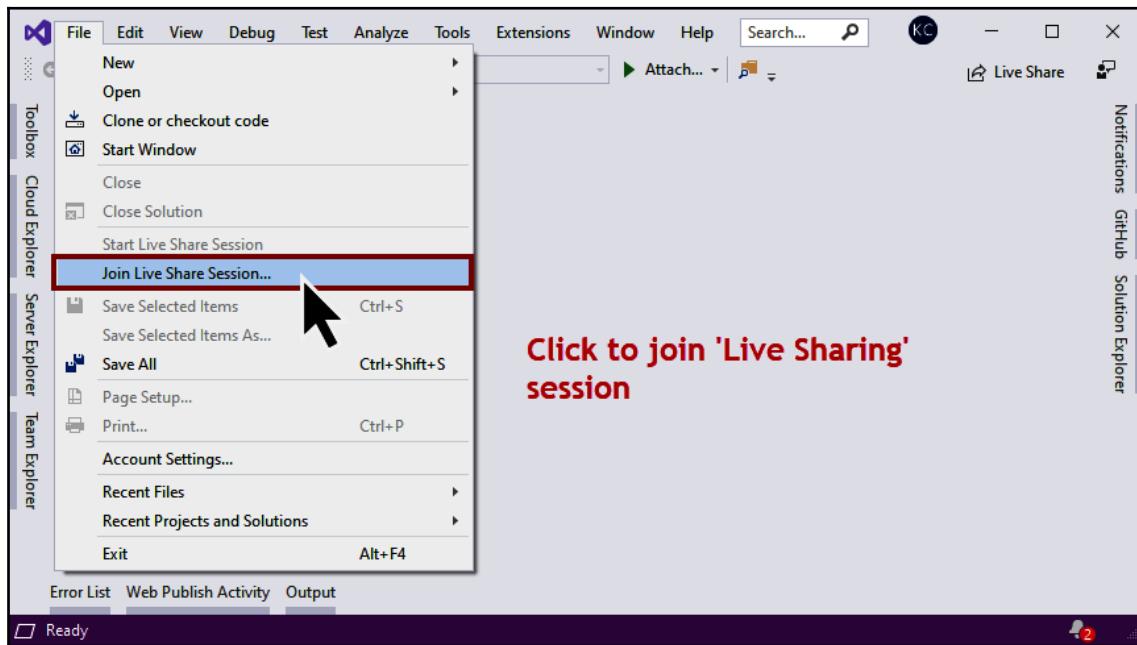
This ends the creation of the live sharing session URL in Visual Studio 2019. Once your participants have received the invitation link, they can join and start collaborating with you and other team members.

In the next section, we will learn how to join an existing live sharing session from Visual Studio 2019.

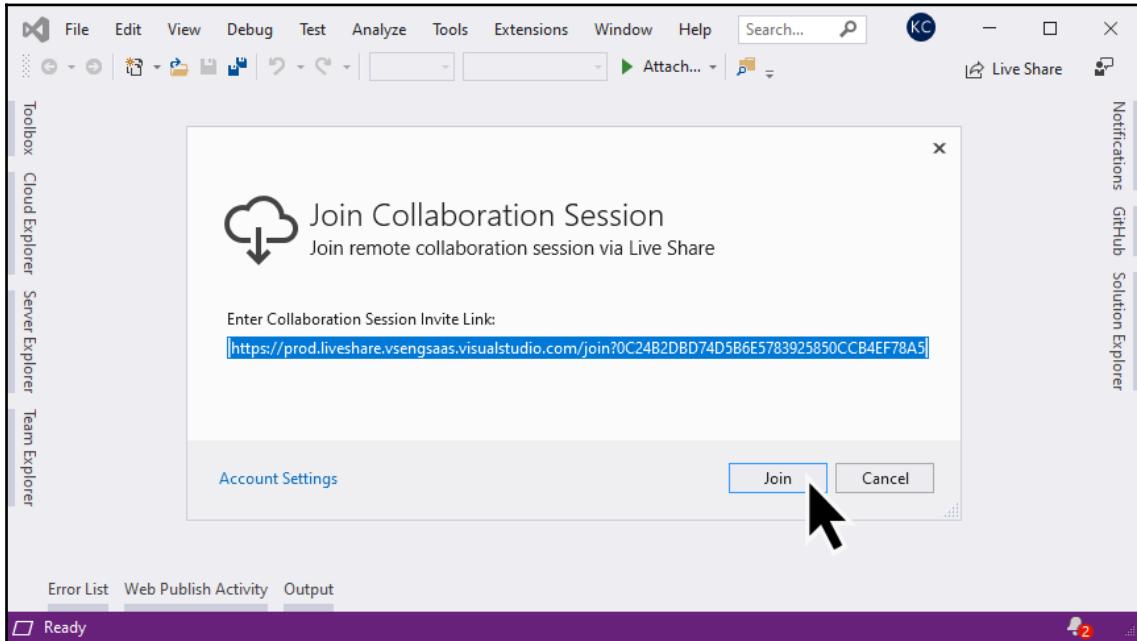
Joining an existing live sharing session

Now that you have a Visual Studio live sharing session invitation from your team member, you can join the session by following these steps:

1. Copy the link that you have received.
2. Open your Visual Studio 2019 IDE.
3. As shown in the following screenshot, navigate to the **File | Join Live Share Session...** menu item:

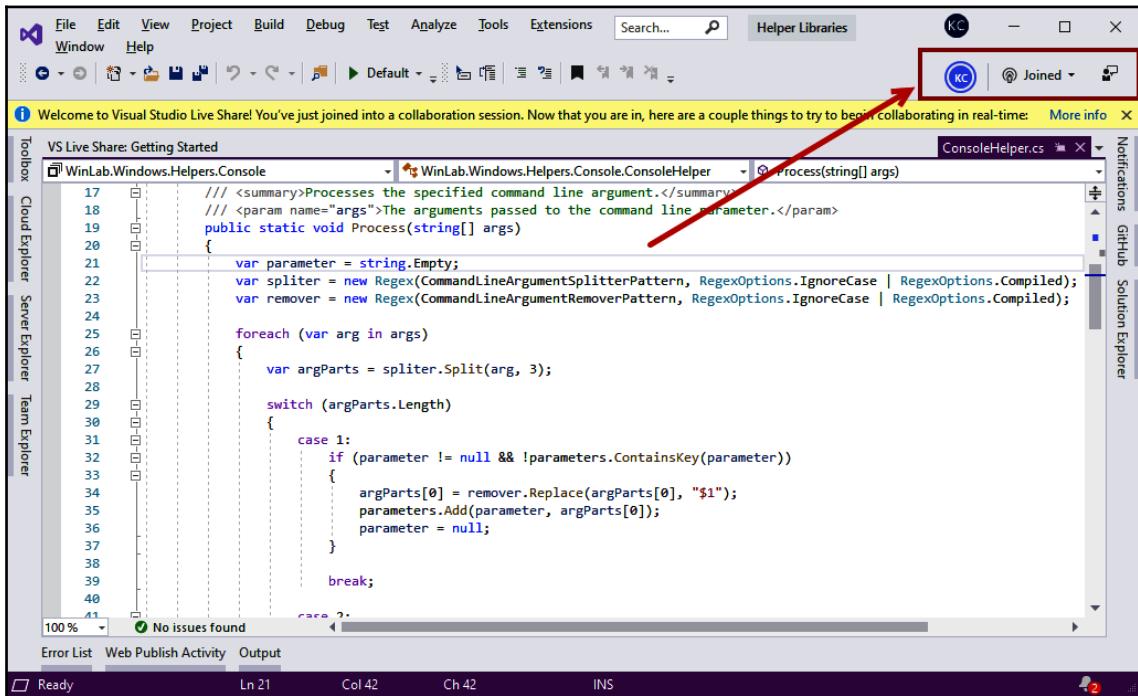


4. The link will be automatically populated from your clipboard content. If it isn't, enter the link into the field and click the **Join** button, as shown here:

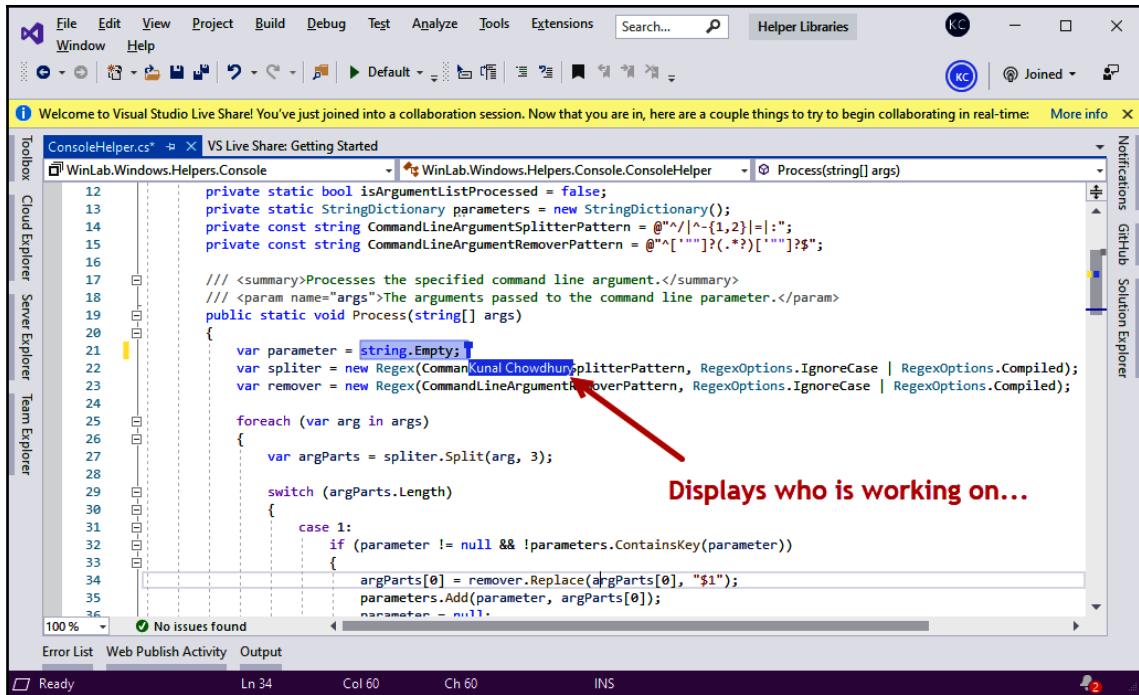


5. Once you have successfully joined the live sharing session, a banner will be displayed, welcoming you to the Visual Studio Live Share session.

6. Another panel will be added to the top-right corner of the IDE, where you can see your gravatar and a button labeled **Joined**, as shown here:



Once you've joined the live sharing session, you will be able to see who is working on what. If a member scrolls through the editor and adds/modifies some code, this will be visible on your screen. A marker will display the other participants' names at the cursor's position:



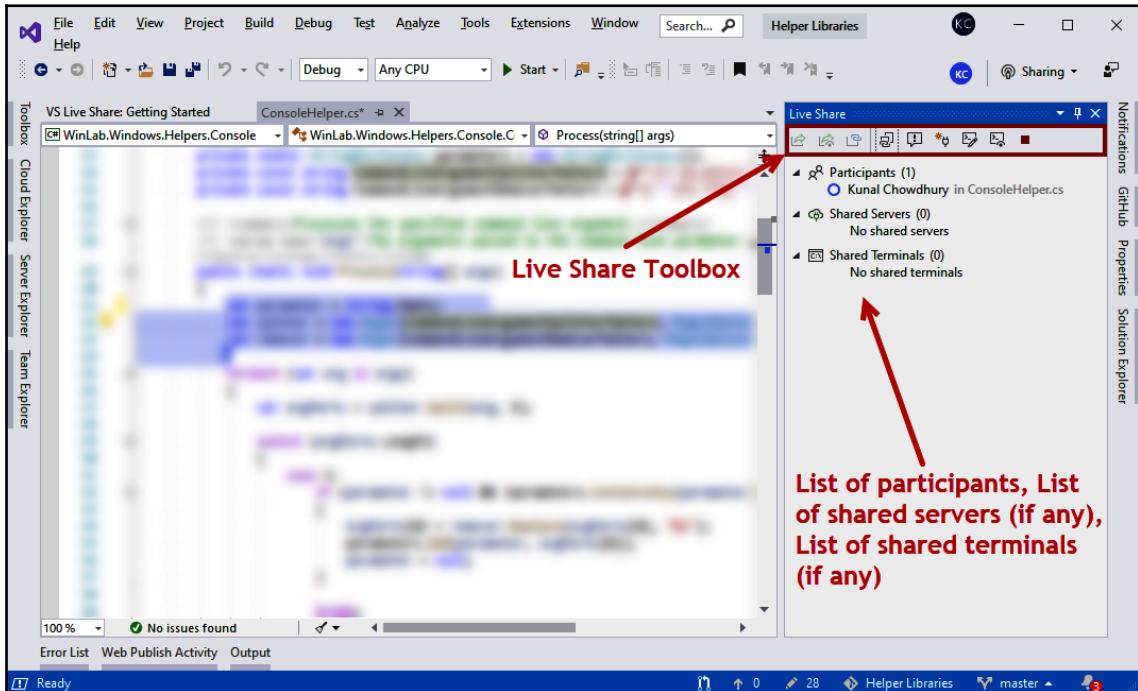
Similarly, if you select or modify content, the same will be visible to the other members as well.

Managing a live sharing session

You're not just limited to creating, inviting, and joining a live sharing session. As an initiator, you can also manage it. The **Live Share** panel will help you with this. From this panel, you are allowed to see the list of participants, shared servers, and shared Terminals. The toolbox at the top of the panel will allow you to launch a Terminal screen with all other participants and control, view, or read/write to the Terminal screen.

You can also send a notification to all of the participants so that they can come and look at something that you are doing, giving them full focus of the current session.

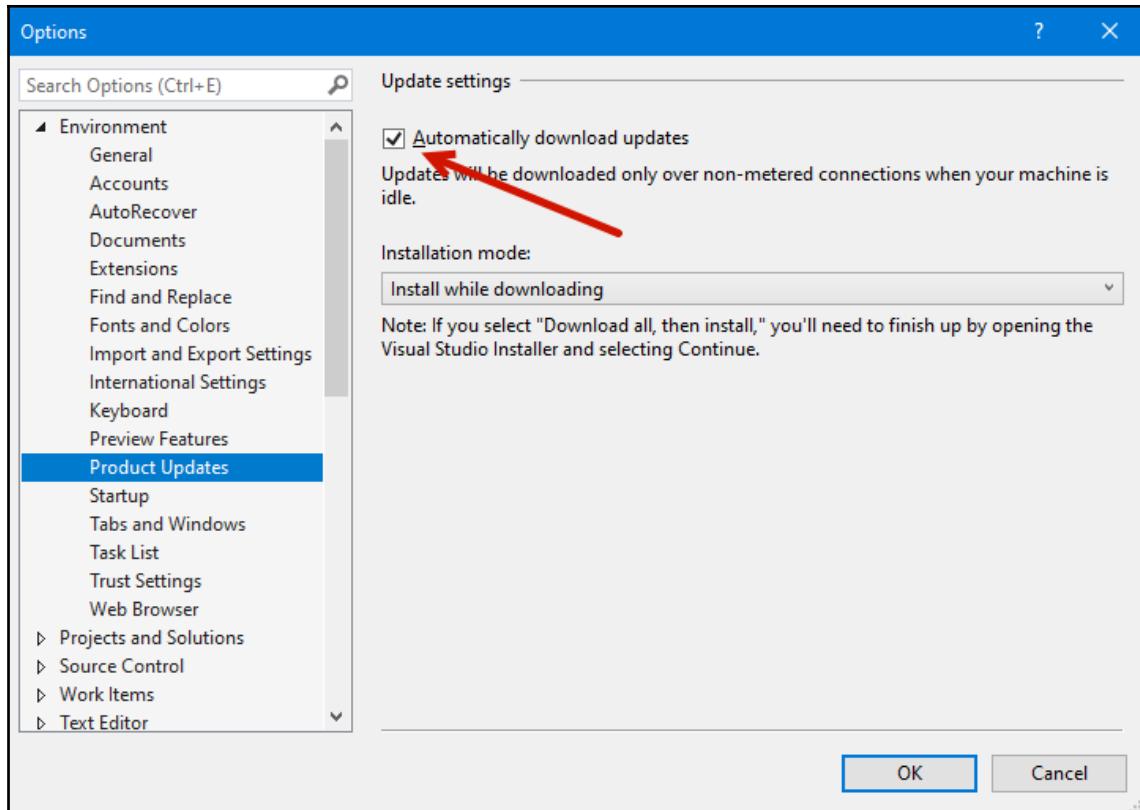
Here's a screenshot of the **Live Share** panel:



When you want to end the live sharing session, you can either click on the stop button from the **Live Share Toolbox** menu or click the tiny **Sharing** drop-down and end it.

Product Updates

Visual Studio 2019 provides you with options for controlling the update installation mode. You can either choose **Download all, then install** or **Install while downloading**. You can find these settings by going to Visual Studio's **Tools | Options** menu and then navigating to **Environment | Product Updates**, as shown in the following screenshot:



The default mode is **Download all, then install**, which allows you to continue using the Visual Studio editor when the download is in progress. If this option is selected, you will need to finish up by opening the Visual Studio installer and selecting the **Continue** button.

You can also select the **Install while downloading** option. If this option is selected, Visual Studio will continue downloading the updates in the background while the update is being installed.

Summary

In this chapter, we learned about the Visual Studio 2019 installation experience, followed by the key features that were newly introduced in Visual Studio 2019. We also discussed the new Start window, quick actions improvements, and Visual Studio IntelliCode, which will not only improve your productivity but will also save you time.

Later, we discussed one-click code cleanup, Clipboard History, and debugger improvements, which will make it easier for you to write and debug code. The all-new Live Share feature will make it easy for you to do peer coding. Finally, we discussed the background update process for keeping your IDE always updated with new features and fixes.

In the next chapter, we will discuss building Windows applications using Windows Presentation Foundation XAML tools.

2

Section 2: Building and Managing Applications

Visual Studio 2019 is an IDE for developers to build applications that can not only target Microsoft platforms, but can also be deployed to Android, iOS, Linux, and macOS by using .NET Core. You can build applications for desktop, web, cloud, and mobile applications or games using C#, VB.NET, F#, C++, Python, TypeScript, ASP.NET, Xamarin, Unity, the Android SDK, and more.

In this section, we will cover how to build applications using **Windows Presentation Foundation (WPF)**, focusing on building XAML-based desktop applications for Windows.

Next, we will gain an understanding of the cloud computing basics, including Microsoft Azure, which is an open, flexible, enterprise-grade cloud computing platform. We will guide you through creating Azure websites and mobile app services, and then integrating those with a Windows application.

We will cover .NET Core, giving you a quick lap around the framework, and guide you on how to create, build, run, and publish .NET Core applications. Then, we will discuss the fundamentals needed to start building web applications using TypeScript and, at the end, will cover the NuGet Package Manager for the Microsoft development platform. Here, you will learn how to create a NuGet package, publish it to a gallery, and test it.

The following chapters will be covered in this section:

- Chapter 2, *Building Desktop Applications for Windows Using WPF*
- Chapter 3, *Accelerate Cloud Development with Microsoft Azure*
- Chapter 4, *Building Applications with .NET Core*
- Chapter 5, *Web Application Development Using TypeScript*
- Chapter 6, *Managing NuGet Packages*

2

Building Desktop Applications for Windows Using WPF

Windows Presentation Foundation (WPF) provides developers with a unified programming model to build dynamic, data-driven desktop applications for Windows. WPF supports a broad set of application development features that include application models, controls, layouts, data binding, graphics, resources, security, and more. WPF is a graphical subsystem for rendering rich **User Interfaces (UIs)** and is part of .NET Framework, as WPF was actually first released along with .NET Framework 3.0. With the release of .NET Core 3.0, WPF will also be part of the .NET Core family.

WPF is a resolution-independent framework that uses vector-based rendering engines and the **Extensible Application Markup Language (XAML)** to create stunning UIs.

The runtime libraries required for WPF and XAML to execute have been included with Windows since the release of Windows Vista and Windows Server 2008. If you are using Windows XP with SP2/SP3 or Windows Server 2003, then you can optionally install the necessary libraries.

Functions such as data binding, customization of styles and templates, animations and triggers, clear separation between the UI and logic, and the ability to run inside web browsers make **Windows Forms (WinForms)** a strong technology that every desktop application developer working on Microsoft platforms must learn. As you continue learning this platform, it will also help you to develop the base of your skills in building any application using XAML tools. Though WinForms still exists in Visual Studio 2019, and despite the fact that it is still used in many enterprise-grade desktop applications, it is now slowly becoming outdated.

Before you start building WPF applications, you must learn about the basic architecture of WPF. In this chapter, we will begin discussing it. Then, we will learn how to create layouts and properties, and how to use data binding and triggers. At the end of this chapter, we will have learned about designing and developing desktop applications with WPF.

In this chapter, we are going to start building Windows applications using the WPF framework, and will discuss the following points:

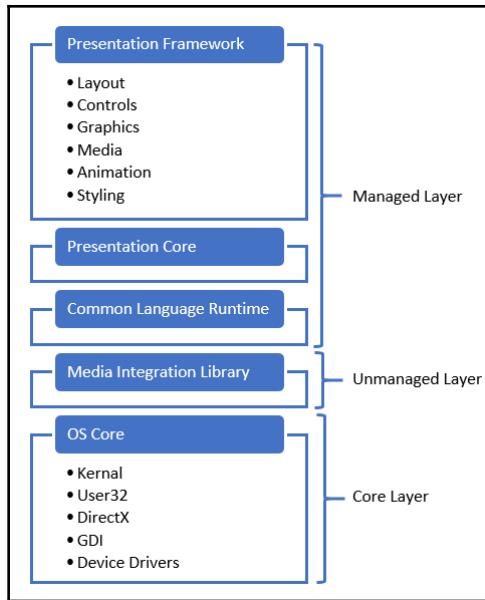
- Understanding the WPF architecture
- An overview of the XAML
- Building your first WPF application
- Exploring layouts in WPF
- The WPF property system
- Data binding in WPF
- Using converters while data binding
- Using triggers in WPF

Technical requirements

To begin this chapter, you should have a basic knowledge of C# and Visual Studio. The installation of Visual Studio 2019 is mandatory with the .NET desktop development workflow.

Understanding the WPF architecture

WPF is made up of a layered architecture that includes managed, unmanaged, and core APIs, as shown in the following diagram, where the programming model is exposed through the managed code:



The diagram demonstrates the basic layers of the framework. Each point is explained in the following sections and they are essential for your understanding of WPF:

- The **Presentation Framework** (`presentationframework.dll`) provides the required components to build WPF applications (such as layouts, controls, graphics, media, data bindings, documents, animations, styling, and more).
- The **Presentation Core** (`presentationcore.dll`) provides you with the wrapper around the **Media Integration Library (MIL)** to provide your public interface to access the milcore (covered later). It also provides you with the visual system to develop a visual tree, which contains visual elements and rendering instructions.
- The **Common Language Runtime (CLR)** provides several features to build robust applications. These include the **Common Type System (CTS)**, error handling, memory management, and more.
- The **Media Integration Library** (`milcore.dll`) is part of the unmanaged layer and provides you with access to these unmanaged components. This is to enable tight integrations with DirectX, which is used to display all the graphics that are rendered through the DirectX engine. The milcore provides a performance gain to the CLR with the rendering instructions from the visual system.
- The next layer is **OS Core**, which provides you with access to low-level APIs to handle the core components of the operating system, including the **Kernel**, **User32**, **DirectX**, **Graphics Device Interface (GDI)**, and **Device Drivers**.

An overview of XAML

XAML is an XML-based markup language to declaratively create the UI of the WPF application. You can create visible UI elements in the declarative XAML syntax and then write the backend code to perform the runtime logic.

Although it is not mandatory to use XAML to create the UI, it is widely accepted in order to make the process easier. This is because the creation of an entire application UI is much more difficult when using C# or VB.NET. It is as easy as writing an XML node with a few optional attributes to create a simple button in the UI. The following example demonstrates how you can create a button using XAML:

```
<Button />
<Button Content="Click Here" />
<Button Height="36" Width="120" />
```

You can either compile an XAML page or render it directly onto the UI. When you compile a XAML file, it produces a binary file of the type, **Binary Application Markup Language (BAML)**, stored as a resource inside the assembly file. When it loads into the memory, the BAML gets parsed at runtime.

There are few syntax terminologies available to define an element in XAML in order to create an instance of it. Here's a few of them to get started.

The object element syntax

An XAML object element declares an instance of a type. Each object element starts with an opening angular bracket (<) followed by its name. Optionally, it can have a prefix to define its namespace outside of the default scope. The object element can be closed by a closing angular bracket (>) or with a self-closing angular bracket (/>). When the object element has a child element, we use the first example (<) followed by an end tag (<Button>Click Here</Button>). For the other case, we use (>) and, additionally, a self-closing element (<Button Content="Click Here" />) is used.

When you specify the object element in an XAML page, the instruction to create an instance of the element is generated, and when you load the XAML, it creates the instance by calling the default constructor of the same object.

The property attribute syntax

Each element can have one or more properties. You can set the properties in XAML as an attribute. The syntax of the property attribute starts with the property name, an assignment operator, and a value within quotation marks. The following example demonstrates how to define XAML elements with property attribute syntax:

```
<Button />
<Button Content="Click Here" />
<Button Content="Click Here" Background="Red" />
```

The first example defines a `Button` element without specifying a property attribute, whereas the other two elements have the attribute defined.

The property element syntax

The properties can also be defined in an element when you cannot assign the value within the quotation marks. This generally starts with `<element.PropertyName>` and ends with `</element.PropertyName>`. The following example demonstrates how to achieve this:

```
<Button>
    <Button.Background>
        <SolidColorBrush Color="Red" />
    </Button.Background>
</Button>
```

In this example, we have defined a `SolidColorBrush` to the `Background` property as an element to `Button` control.

The content syntax

An XAML element can have content within it. It can be set as the value of the child elements. Let's refer to the following code blocks. The example shows how to set the `Content` text property of `Button` instead of specifying it with the attribute syntax (`<Button Content="Click Here" />`):

```
<Button>
    <Button.Content>
        Click Here
    </Button.Content>
</Button>
```

In the following example, when the `Button` element is wrapped by a `Border` panel, it is defined as a `Content` element of the said panel and omits the explicit definition of the `Content` property:

```
<Border>
    <Button Content="Click Here" />
</Border>
```

The preceding code can be rewritten with a `content` property (`Border.Child`) that has a single element in it:

```
<Border>
    <Border.Child>
        <Button Content="Click Here" />
    </Border.Child>
</Border>
```

Based on the requirement and place, this is often used by the developers/designers.

The collection syntax

Sometimes, it's necessary to define a collection of elements in XAML. This is done using the collection syntax to make it more readable. For example, `StackPanel` can have multiple elements defined inside the `Children` property:

```
<StackPanel>
    <StackPanel.Children>
        <Button Content="1" />
        <Button Content="2" />
    </StackPanel.Children>
</StackPanel>
```

The preceding example can also be written as the following, where the parser knows how to create and assign the elements to the panel:

```
<StackPanel>
    <Button Content="1" />
    <Button Content="2" />
</StackPanel>
```

You can use either of the formats to define your elements in the XAML page.

The event attribute syntax

In XAML, you can also define events for a specific object element. Although it looks like a property attribute, it is used to assign the event. If an attribute value of an element is the name of an event, it is treated as an event. In the following code snippet, the `Click` attribute defines the `Click` event of the buttons:

```
<Button Click="Button_Click">Click Here</Button>
<Button Click="Button_Click" Content="Click Here" />
```

The implementation of the event handler is generally defined in the backend code of the XAML page. The event implementation for the preceding `Button_Click` event looks like the following:

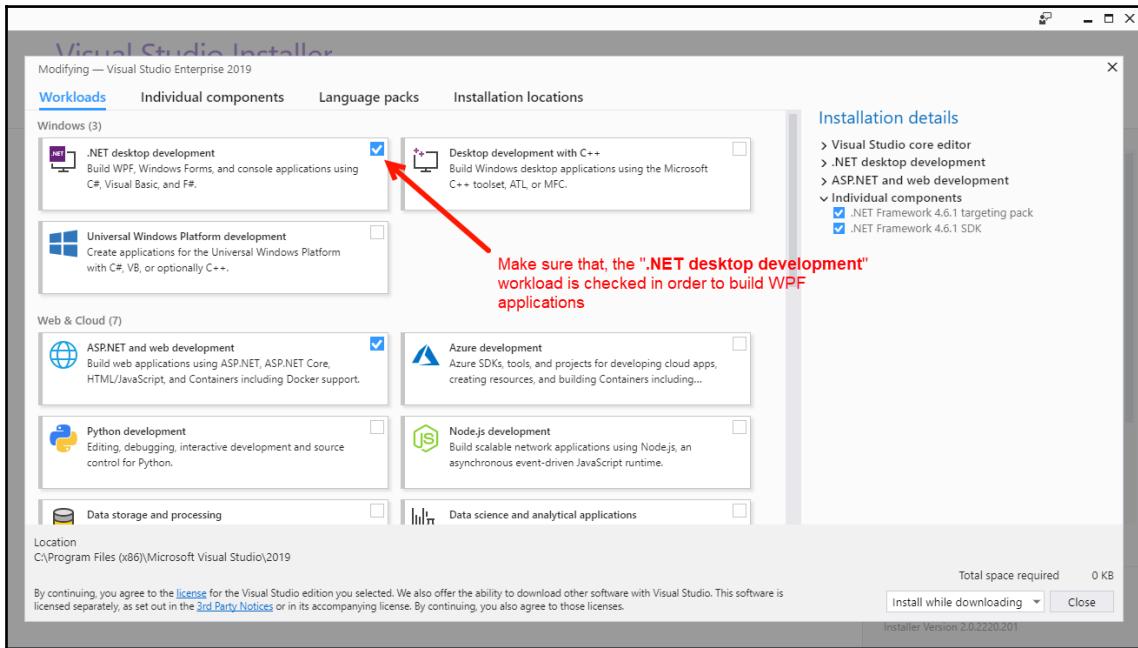
```
void Button_Click(object sender, RoutedEventArgs e)
{
    // event implementation
}
```

In this section, we have discussed object element syntax, property attribute syntax, property element syntax, content syntax, collection syntax, and event attribute syntax, which gave you a basic overview of the XAML syntax to define the UI of the WPF application. In the next section, we will start building our first WPF application.

Building your first WPF application

Before starting with the WPF application development, make sure that you have installed the **.NET desktop development** workflow on your system. Run your Visual Studio 2019 installer and confirm that the **.NET desktop development** workflow is already checked. If not, check it and proceed toward the installation of the required components.

The following screenshot illustrates how to check the **.NET desktop development** box:

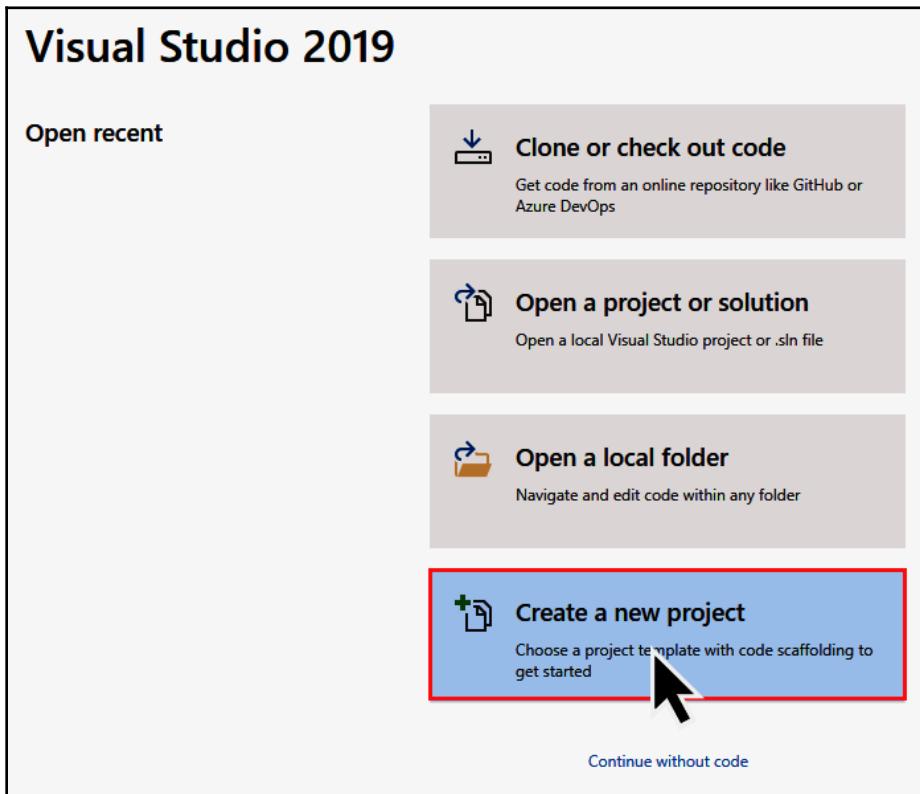


We hope you have already configured your Visual Studio installation to build desktop applications using WPF. Let's start building our first WPF application using XAML and C#.

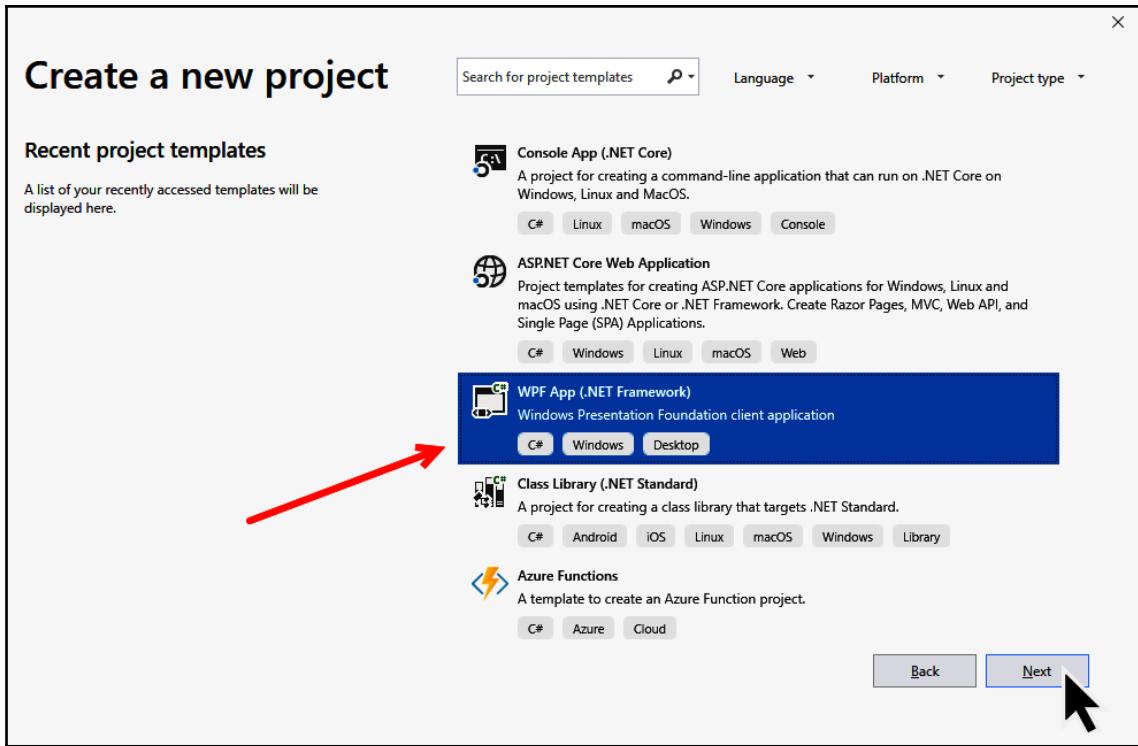
Getting started with the WPF project

To get started, open your Visual Studio 2019 IDE and perform the following steps:

1. Click on **Create a new project** from the start screen, as shown in the following screenshot:

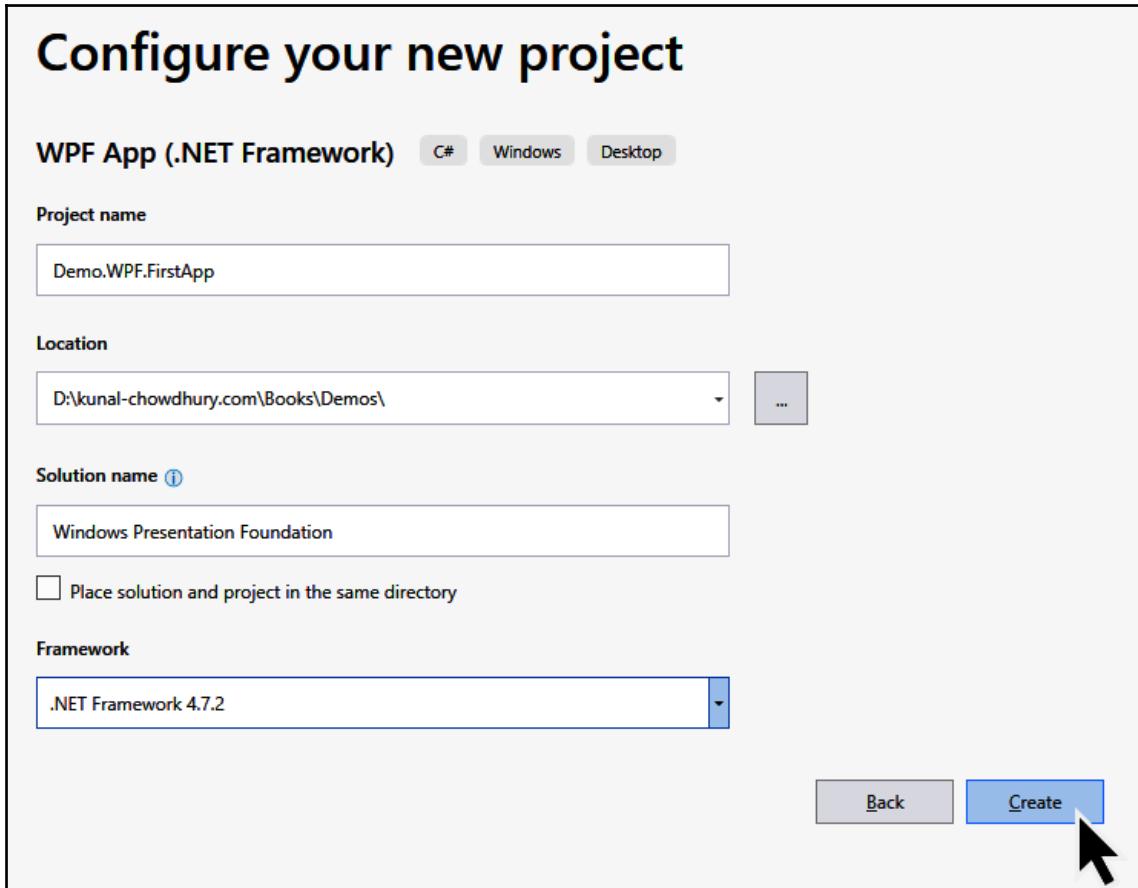


This will open the new project window on your screen. In the **Create a new project** dialog window, select the **WPF App (.NET Framework)** template from the list and click on **Next**. If you are not able to locate the template in the list, you can search for `wpf app` in the search box:



2. In the next screen, you need to enter a project name, select the location where you would like to create it, optionally provide a different solution name, and select the framework that you would like the application to target.

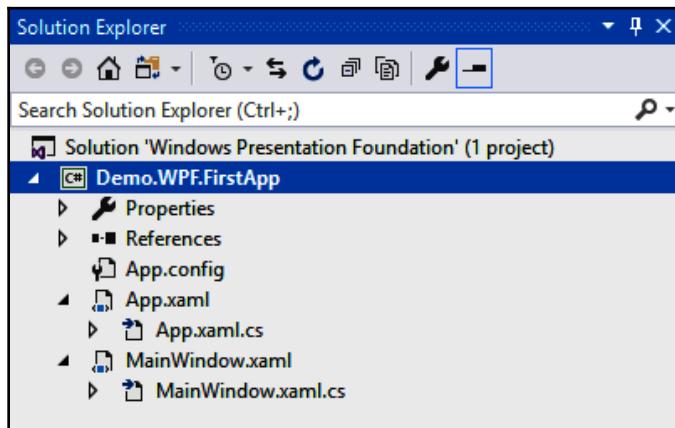
- Once you are done entering the metadata, you are ready to create the project. Hit the **Create** button to continue creating the project. Here's a screenshot of the screen for your reference:



Here, we have created the basic project.

Understanding the WPF project structure

Once the project is created from the selected template, you will see that the project has a few files in it already. Among them, you will find two XAML files (`App.xaml` and `MainWindow.xaml`) and two associated C# files (`App.xaml.cs` and `MainWindow.xaml.cs`) automatically created by Visual Studio:



`App.xaml` is the declarative start point of your application, which extends the `Application` class and subscribes to application-specific events, unhandled exceptions, and more. From this class, it gets the starting instruction to navigate to the initial window or page.

Here's the structure of the `App.xaml` file:

```
<Application x:Class="Demo.WPF.FirstApp.App"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:local="clr-namespace:Demo.WPF.FirstApp"
StartupUri="MainWindow.xaml">
    <Application.Resources>
        </Application.Resources>
</Application>
```

Here, you can see that `StartupUri` is set to `MainWindow.xaml`, which means that `MainWindow.xaml` will launch on startup. If you want to launch a different window or page at startup, then replace this with the desired filename.

By default, the other file, `MainWindow.xaml`, consists of a blank `Grid` instance (one of the WPF panels), where you can specify your layouts and add your desired controls to design the UI:

```
<Window x:Class="Demo.WPF.FirstApp.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
xmlns:local="clr-namespace:Demo.WPF.FirstApp"
mc:Ignorable="d"
Title="MainWindow" Height="350" Width="525">
    <Grid>
    </Grid>
</Window>
```

The related logic, that is, properties and events, can be defined in the associated backend code the `MainWindow.xaml.cs` class file. It's a partial class and all the autogenerated UI-related initialization is written in the `InitializeComponent()` method, which is defined in the other partial `MainWindow.g.i.cs` class file:

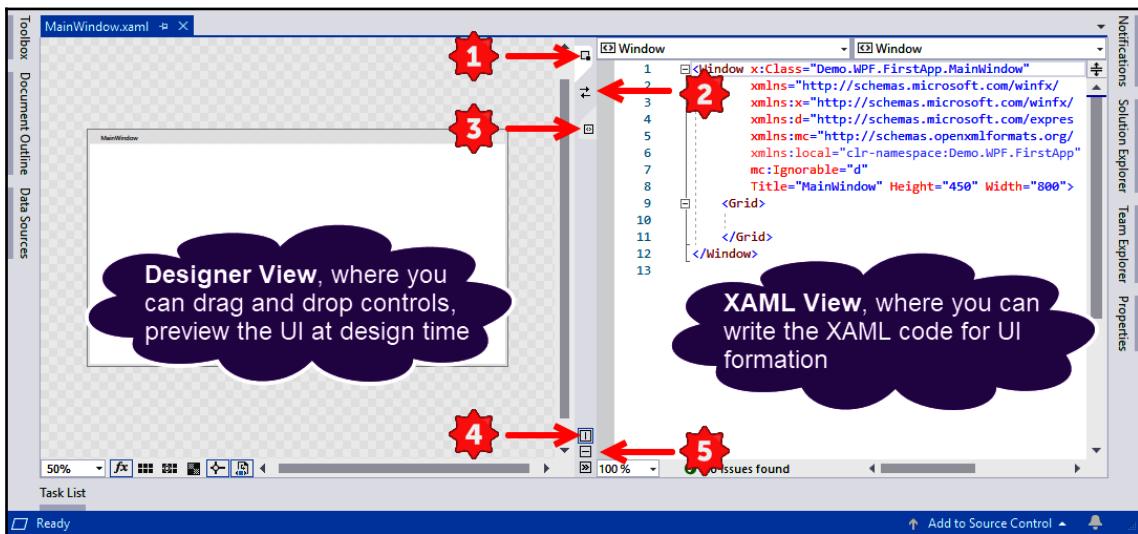
```
using System.Windows;

namespace Demo.WPF.FirstApp
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
    }
}
```

When you build any WPF desktop application project, it generates a `.exe` file along with the `.dll` file (that is, if you have any in-house class libraries and/or third-party libraries).

Getting familiar with XAML Designer

When you open an XAML page, the Visual Studio 2019 editor opens it in two different views, side by side, within the same screen. Those two views are named **designer view** and **XAML view**. In the designer view, you will be able to design your UI by dragging, dropping, and resizing elements; while on the other hand, the XAML view will allow you to write XAML code directly to create the UI. You can see the Visual Studio 2019 editor here:



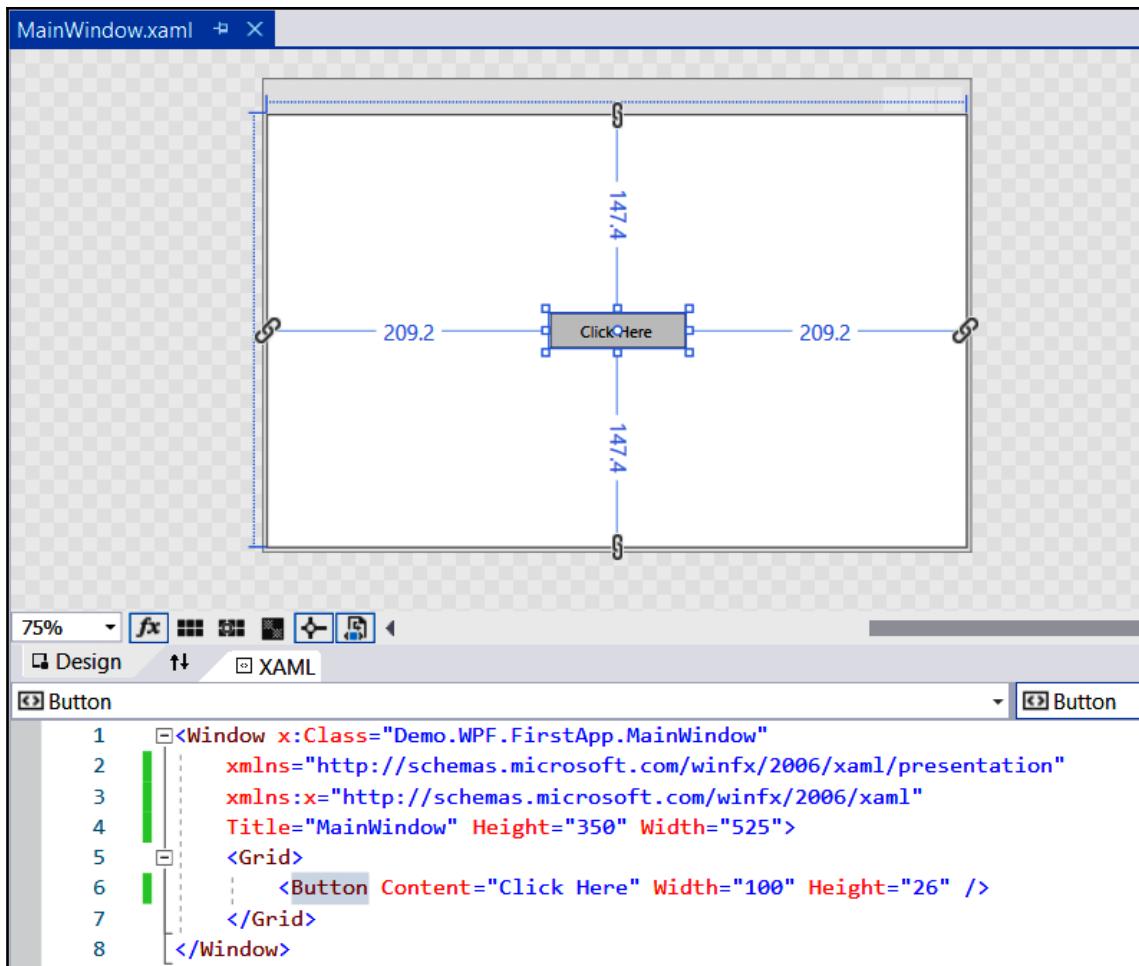
Check out the preceding screenshot to see how Visual Studio loads the XAML page and provides you with a variety of options to change the view. Some of these options are as follows:

1. **Switch to design view:** Double-clicking on this icon will change the view to a full screen designer view, hiding the XAML editor.
2. **Swap panes:** This will allow you to swap the views, either from left to right (a vertical split) or from top to bottom (a horizontal split).
3. **Switch to XAML view:** If you double-click this icon, it will change the view to a full screen XAML view, hiding the designer.
4. **Vertical split:** When you do a vertical split, the designer view and the XAML editor view will be set side by side (as in the preceding screenshot).
5. **Horizontal split:** When you do a horizontal split, the designer view and the XAML editor view will align in the top and bottom positions in a stacked fashion.

Adding controls in XAML

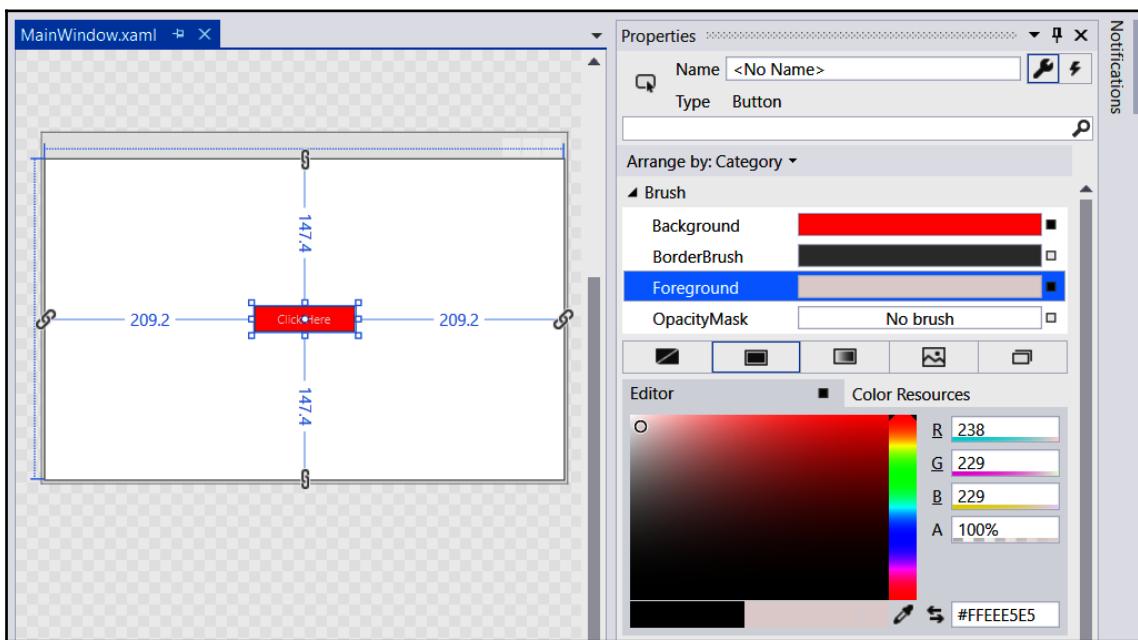
Every application needs some kind of UI so that the user can interact with the application easily. In WPF, you can add various controls to create your application's UI. As we have already created our first WPF project and become familiar with the project structure and XAML designer, let's add a button control on the UI, as follows:

1. To do this, open `MainPage.xaml` and, inside the `Grid` panel, add the `Button` tag with its content and dimensions, as shown in the following screenshot:



Once this is done, the UI will show you a preview in the designer view.

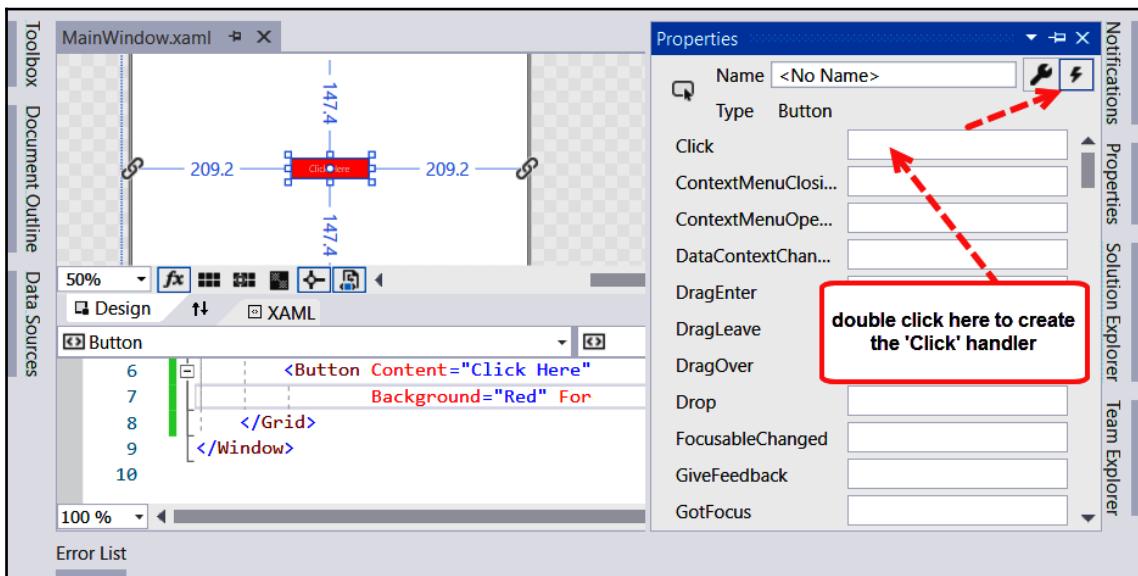
2. Now, let's add some color to the button. You can either do this by writing XAML attributes for the button in the XAML view, or you can utilize the Visual Studio property window from the designer view.
3. Let's add the color from the **Properties** window and we'll see how the Visual Studio editor adds the XAML attributes. To do this, select the button in the designer view and open the **Properties** window. Here, you will see a category called **Brush**. Under this, you will be able to change the color of the selected UI element. First, select the **Background** and move the color slider to select red. Then, select **Foreground** and choose the color white. You will notice that the preview in the designer view automatically updates with the selected colors:



Here's the XAML code for your reference:

```
<Grid>
    <Button Content="Click Here" Width="100" Height="26"
        Background="Red" Foreground="#FFEEE5E5" />
</Grid>
```

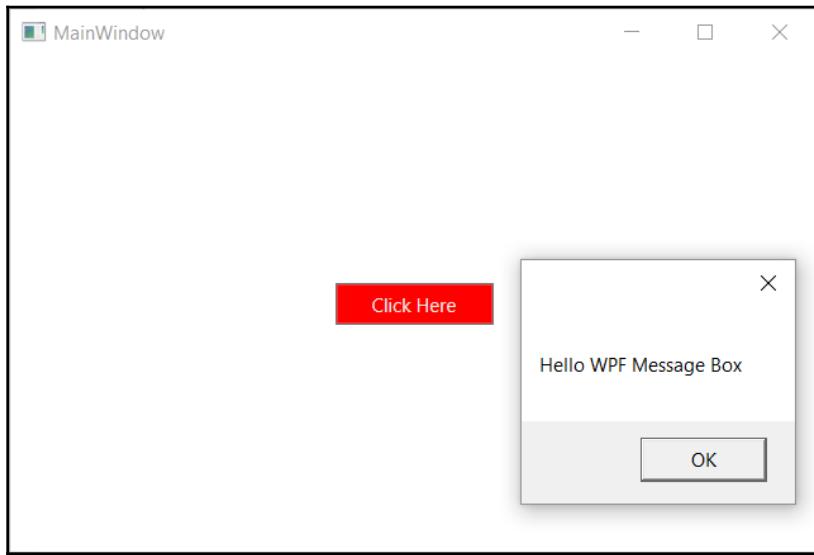
4. Now, when you build and run the app, you will see a button with a red background and white-colored text on the screen. If you click on that button, there will be no action, as we have not added any event handler to it.
5. To do this, go to the design view, select the button control, navigate to the **Properties** window, and click on the Navigate Event Handler icon present in the right-hand corner, as shown in the following screenshot:



6. Here, you will find an input box called **Click**. Double-click on it to create the click handler of the button in the associated code file and register it in the XAML page.
7. Alternatively, you can write `Click="Button_Click"` in the XAML against the `Button` tag to register the event first, and then press the `F12` keyboard shortcut on the event name to generate the associated event handler in the backend code.
8. Now, navigate to the event handler implementation and add a message box to be shown when you click on the button:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show("Hello WPF Message Box");
}
```

Let's build and run the code. If the build succeeds, it will show you the following window onscreen with the button. Clicking on the button will show you this message:



As you are now comfortable with adding controls to the application window and customizing the controls, let's jump into the next section to explore the various layouts that are available in WPF.

Exploring layouts in WPF

Layout is an important part of any GUI-based application, particularly for usability purposes. You must arrange your controls in a proper position, so that users find it easy to work with. You should also keep arranging your controls properly to scale them to different screen resolutions and/or different font sizes. WPF provides built-in support of various panels to help you with this:

- **Grid:** This defines a flexible area to position UI elements in rows and columns.
- **StackPanel:** This defines an area where the child elements can be arranged in a stacked manner, either horizontally or vertically.
- **Canvas:** This defines an area to position the UI elements by coordinates relative to the area specified.
- **DockPanel:** This defines an area that you can arrange horizontally or vertically, relative to each other.

- **WrapPanel:** This defines an area where child elements can position themselves sequentially, from left to right. When it reaches the edge of the panel box, it breaks to the next line.
- **VirtualizingPanel:** This defines the panel to virtualize the children's data.
- **UniformGrid:** This defines the panel to have a uniform cell size.

Each of the preceding panel elements derives from the `Panel` base and allows you to create a simple and better layout design for your app. You can set controls or child panels inside a panel to design your view.

Note that designing a layout is an intensive process and so, if you have a large collection of children, it will require a higher number of calculations to set the size and position of the elements. Therefore, the UI complexity will increase, impacting the performance of the application.

Whenever a child element changes its position, it automatically triggers the layout system to revalidate the UI and arranges the children according to the layout defined. The layout system does this in two phases. The `Measure` pass recalculates each member of the children collection with a call to the `Measure` method of the panel. This sets the height, width, and margin of the controls in the UI.

Then, the `Arrange` pass, which generally begins with a call to the `Arrange` method, generates the bounds of the child and passes them to the `ArrangeCore` method for processing. It then evaluates the desired size of the child and calls the `ArrangeOverride` method to determine the final size. Finally, it allocates the desired space and completes the process of layouting.

Here are some quick tips to remember:

- `Canvas` is a simple panel and has a better performance than a complex panel such as `Grid`.
- When using `Canvas`, avoid the fixed positioning of UI elements. Rather, align them with the margins and padding.
- Whenever possible, set the `Height` and `Width` properties to `Auto` instead of a fixed size.
- Avoid unnecessary calls to `UpdateLayout` as it forces a recursive layout update.
- Use `StackPanel` to lay out a list of elements horizontally or vertically.
- When working with a large children collection, consider using `VirtualizingStackPanel`.
- Use `GridPanel` to lay out static data in the UI.

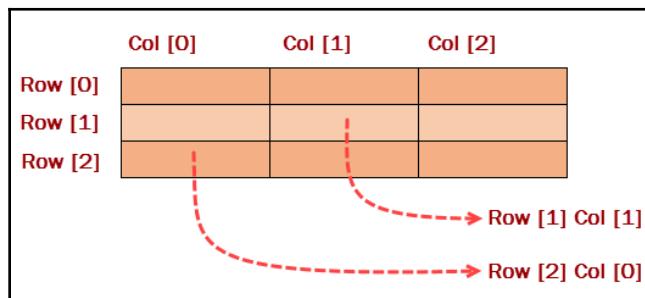
- `ItemControl` can be used with a `Grid` panel in `DataTemplate` to lay out dynamic data.
- Use `Margin` to add extra space around the control.
- Use `Padding` to add extra space inside the control.

As you have already seen, there are many panels in WPF that can be used to create the UI layout. Let's now discuss each of them in detail. This will help you to decide which one to choose, when to choose them, and how to use them.

Using Grid as a WPF panel

Let's begin with the `Grid` layout panel. You will find it the most useful panel while building the UI of any WPF application. When you create a new window, page, or `UserControl`, it's the default panel that gets inserted inside every XAML page. You can see that there was a `Grid` panel inserted when we created our first WPF application, where we placed a button control inside it.

When you would like to show some controls, some data in the matrix, or a tabular format, the `Grid` layout control allows you to define the elements in rows and columns, which is demonstrated as follows:



In XAML, you need to design your `Grid` cells using `RowDefinitions` and `ColumnDefinitions`. Each definition group can have multiple definitions. `RowDefinition` can have a height set to it and `ColumnDefinition` can have a width set to it. Each definition can have three different sets of values:

- **Auto:** When it is set to `Auto`, the row height or the column width will be set depending on the content.
- **A numeric value:** When you set a positive numeric value, it will have a fixed size.

- * (asterisk): When you provide an asterisk, it will take the maximum available space in the ratio specified. For example, consider the following:
 - If you have two rows and you specify both of their heights as *, they will have the row height equally distributed in the available space.
 - If you have two rows and you specify one as * and the other as 2*, then the row height will be divided by a 1:2 ratio of the available space.



If you don't specify any height or width, then the row or column will, by default, have an equal distribution of the available space.

Let's discuss the following code snippet, which shows four row definitions and three column definitions. The first two rows will automatically set their heights based on the dimension of the content, the fourth row will have a height of 30 px, and the third row will have the remaining space as its height. Similarly, the first column will have an automatic (Auto) width depending on its contents, the second column will have a space of 20 px, and the third column will occupy the remaining space available to it:

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>           
        <RowDefinition Height="Auto"/>           
        <RowDefinition Height="*"/>              
        <RowDefinition Height="30"/>             
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"/>          
        <ColumnDefinition Width="20"/>            
        <ColumnDefinition Width="*"/>              
    </Grid.ColumnDefinitions>

    <!-- First Row -->
    <TextBlock Text="First Name" Grid.Row="0" Grid.Column="0"/>
    <TextBlock Text=":" Grid.Row="0" Grid.Column="1"/>
    <TextBox Grid.Row="0" Grid.Column="2"/>

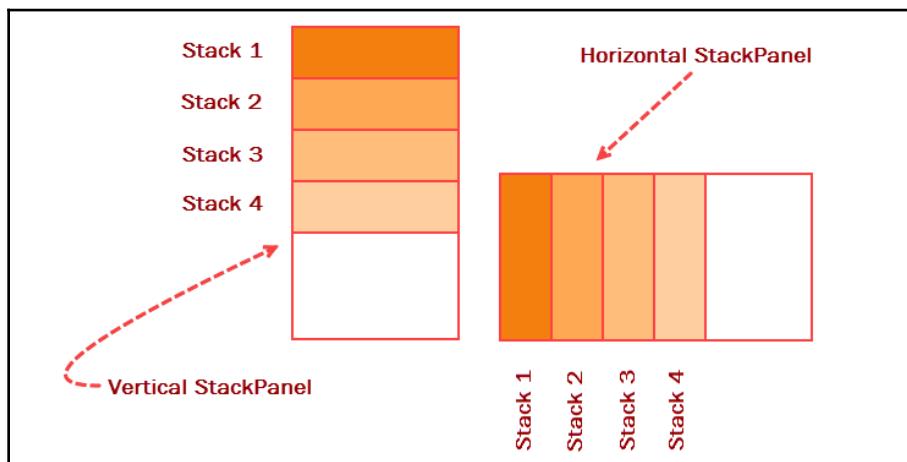
    <!-- Second Row -->
    <TextBlock Text="Last Name" Grid.Row="1" Grid.Column="0"/>
    <TextBlock Text=":" Grid.Row="1" Grid.Column="1"/>
    <TextBox Grid.Row="1" Grid.Column="2"/>

</Grid>
```

Once you divide your `Grid` panel into rows and columns, you can place your elements in the appropriate cell by using the `Grid.Row` or `Grid.Column` properties. The `Grid.RowSpan` and `Grid.ColumnSpan` properties are used to span an element across multiple rows and columns.

Using StackPanel to define the stacked layout

When you want to add elements in a stack fashion, either horizontally or vertically, you need to use `StackPanel`. You need to specify the `Orientation` property to set whether it will be a horizontal `StackPanel` or a vertical `StackPanel`. The following figure demonstrates both a horizontal and a vertical `StackPanel`:



If you don't specify any orientation, it will be automatically set as a vertical `StackPanel`. You can add multiple UI elements (controls or panels) as children to it:

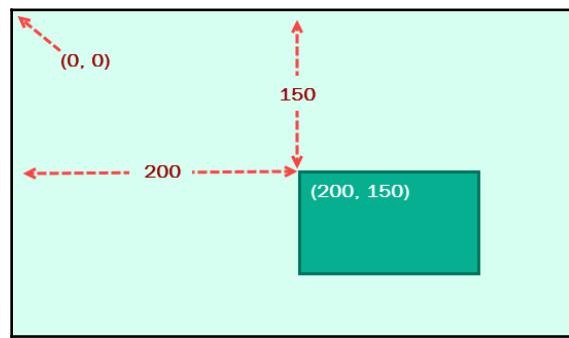
```
<!-- Horizontal StackPanel -->  
  
<StackPanel Orientation="Horizontal">  
    <TextBlock Text="Stack 1"/>  
    <TextBlock Text="Stack 2"/>  
    <TextBlock Text="Stack 3"/>  
    <TextBlock Text="Stack 4"/>  
</StackPanel>  
<!-- Vertical StackPanel -->  
<StackPanel Orientation="Vertical">  
    <TextBlock Text="Stack 1"/>  
    <TextBlock Text="Stack 2"/>
```

```
<TextBlock Text="Stack 3"/>
<TextBlock Text="Stack 4"/>
</StackPanel>
```

While using `StackPanel`, do note that the content does not resize when you resize the panel.

Using Canvas as a panel

`Canvas` is a lightweight panel that can be used to position the children at a specific coordinate position within the view. You can imagine it as an HTML `div` element and position the elements at (x, y) coordinates. The attached properties, `Canvas.Left` and `Canvas.Top`, can be used to position the elements in relation to the left and top corners; in comparison to this, the `Canvas.Right` and `Canvas.Bottom` properties can be used to position the child elements in relation to the right and bottom corners. Here's a representation of placing an element in `Canvas`:



Here's a simple example that demonstrates `TextBlock` positioned at $(75, 30)$ and $(220, 75)$, respectively:

```
<Canvas>
    <TextBlock Text="(75, 30)"
        Canvas.Left="75" Canvas.Top="30"/>
    <TextBlock Text="(220, 75)"
        Canvas.Left="220" Canvas.Top="75"/>
</Canvas>
```

This is not very flexible, as you have to manually move the child controls around and make them align in the way you want them to. We recommend that you use it only when you want complete control over the position of the child controls.

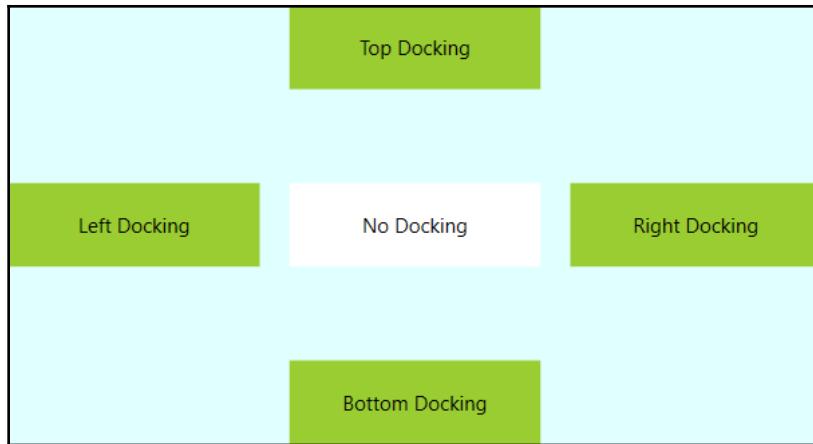
Using WPF DockPanel to dock child elements

DockPanel allows you to dock the child elements to any one of the four sides (that is, top, right, bottom, or left). By default, the last element (if not given any specific dock position) fills the remaining space. You can use it when you need to divide your window into regions and/or when you want to place one or more elements on any of the sides.

As shown in the following code snippet, you need to set the DockPanel.Dock attribute of the child elements to one of the four values:

```
<DockPanel Background="LightCyan">
    <Border Width="150" Height="50"
            DockPanel.Dock="Bottom" Background="YellowGreen">
        <TextBlock Text="Bottom Docking"
                    HorizontalAlignment="Center"
                    VerticalAlignment="Center"/>
    </Border>
    <Border Width="150" Height="50"
            DockPanel.Dock="Top" Background="YellowGreen">
        <TextBlock Text="Top Docking"
                    HorizontalAlignment="Center"
                    VerticalAlignment="Center"/>
    </Border>
    <Border Width="150" Height="50"
            DockPanel.Dock="Right" Background="YellowGreen">
        <TextBlock Text="Right Docking"
                    HorizontalAlignment="Center"
                    VerticalAlignment="Center"/>
    </Border>
    <Border Width="150" Height="50"
            DockPanel.Dock="Left" Background="YellowGreen">
        <TextBlock Text="Left Docking"
                    HorizontalAlignment="Center"
                    VerticalAlignment="Center"/>
    </Border>
    <Border Width="150" Height="50"
            Background="White">
        <TextBlock Text="No Docking"
                    HorizontalAlignment="Center"
                    VerticalAlignment="Center"/>
    </Border>
</DockPanel>
```

When you run the preceding example, the result will look like the following diagram, which positions all the child elements depending on the value specified in the DockPanel.Dock attribute:

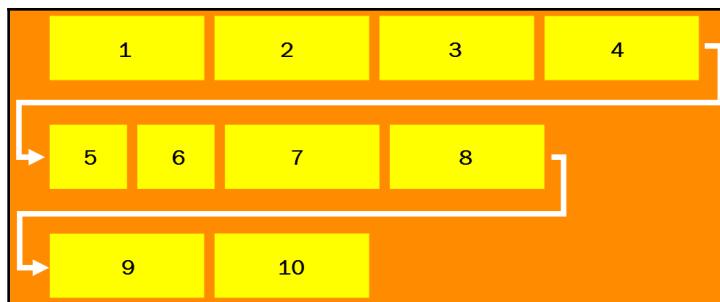


Note that the last element, which does not have any docking position set, has taken the remaining space and is placed at the center position of the window.

Using WrapPanel to automatically reposition elements

Inside a panel, when you want to position each of its child elements next to the other, either horizontally (the default) or vertically until there is no more room, you will need to use the `WrapPanel` function. This just looks like a combination of `Grid` and multiple `StackPanel` elements, with variable cell sizes. Use it when you want a vertical or horizontal list to automatically wrap elements to the next line if there's no more room for the controls to accommodate them in the same line.

Let's consider the following diagram as an example of `WrapPanel` with horizontal (the default) orientation:



Here, you can see that the child elements positioned themselves in a row, and, when `WrapPanel` does not have room to accommodate them all on the same line, it wraps it onto a new line. For example, in the preceding diagram, the fifth element wrapped itself onto the second line due to lack of space in the first line. Similarly, the ninth element positioned itself on the third line as the second line does not have enough space for that element. You can also see that the items can have variable sizes and that does not affect the behavior of the positioning. As a result, there are no big gaps between the elements in the grid.

Here's an XAML code snippet to demonstrate the preceding example:

```
<WrapPanel Background="DarkOrange">
    <Border Width="120" Height="50"
        Background="Yellow" Margin="4"/>
    <Border Width="60" Height="50"
        Background="Yellow" Margin="4"/>
    <Border Width="60" Height="50"
        Background="Yellow" Margin="4"/>
    <Border Width="120" Height="50"
        Background="Yellow" Margin="4"/>
    <Border Width="120" Height="50"
        Background="Yellow" Margin="4"/>
</WrapPanel>
```

After placing the code, run your application and observe the behavior on resizing the window.

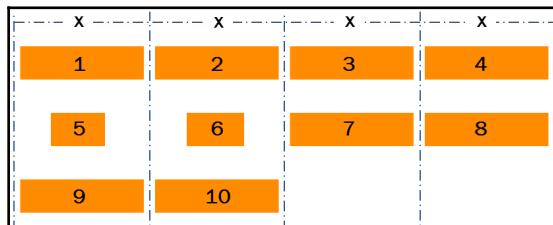
Using UniformGrid to place elements in uniform cells

The `UniformGrid` panel is just like the `WrapPanel`, but with the additional catch that it will have child elements placed in uniform cells. The rows and columns in `UniformGrid` have the same size. By default, the rows and columns automatically resize to give space to more elements. The elements also resize automatically to accommodate themselves within that space.

Here's a code snippet demonstrating the use of UniformGrid:

```
<UniformGrid>
    <Border Width="150" Height="40"
            Background="DarkOrange" Margin="4"/>
    <Border Width="150" Height="40"
            Background="DarkOrange" Margin="4"/>
    .
    .
    .
    <Border Width="150" Height="40"
            Background="DarkOrange" Margin="4"/>
    <Border Width="150" Height="40"
            Background="DarkOrange" Margin="4"/>
</UniformGrid>
```

The following diagram is a UniformGrid panel, showing all the columns automatically resized to the same size, x:



In the preceding example, notice the elements **5** and **6**. Unlike the WrapPanel, they don't sit side by side in a condensed manner to reduce the gap between them; rather, they still position themselves in their originally defined column cells.

You can define how many rows or columns you want to show in the Grid panel by specifying the Rows and Columns properties of UniformGrid.

When you change the row count to 1, all the elements inside UniformGrid will accommodate themselves in a single row by reducing their size. When you define more rows, based on the number of row counts, it will reduce the column count:

```
<UniformGrid Rows="1">
    <Border Width="150" Height="40"
            Background="DarkOrange" Margin="4"/>
    .
    .
    <Border Width="150" Height="40"
            Background="DarkOrange" Margin="4"/>
</UniformGrid>
```

Similarly, you can specify the `Columns` count; when specified, depending on the column count, it will generate the number of rows.

The WPF property system

WPF provides a new property system that extends the functionality of the default **Common Language Runtime (CLR)** property and is known as the **dependency property**. A normal CLR property is a wrapper on top of its getter (`get { ... }`) and setter (`set { ... }`) implementation. A dependency property extends the CLR property to provide you with a way to compute the value based on the user inputs.

In addition to this, it also provides you with a way to do self-contained validation, set default values, monitor changes to its value, and do a callback.

When you want to define a dependency property in a class, you must derive that class from `DependencyObject`, which will act as an observer, and `DependencyObject` holds the new property system for a faster execution.

A CLR property looks like this:

```
private string m_AuthorName;

public string AuthorName
{
    get { return m_AuthorName; }
    set { m_AuthorName = value; }
}

public string BookName { get; set; } // auto properties
```

A dependency property extends the CLR property further and provides you with more options to set default values, pass a callback method, and more:

```
public static readonly DependencyProperty AuthorNameProperty =
DependencyProperty.Register("AuthorName", typeof(string),
typeof(MainWindow), new PropertyMetadata("Kunal Chowdhury"));

public string AuthorName
{
    get { return (string)GetValue(AuthorNameProperty); }
    set { SetValue(AuthorNameProperty, value); }
}
```

The first line is used to register the dependency property into the WPF property system. This has been done to ensure that the object contains the property and we can call the getter/setter methods easily to access the property value.

You can also wrap a dependency property using a normal CLR property. You can then use the `GetValue` and `SetValue` methods to get and set the value passed to the dependency property.

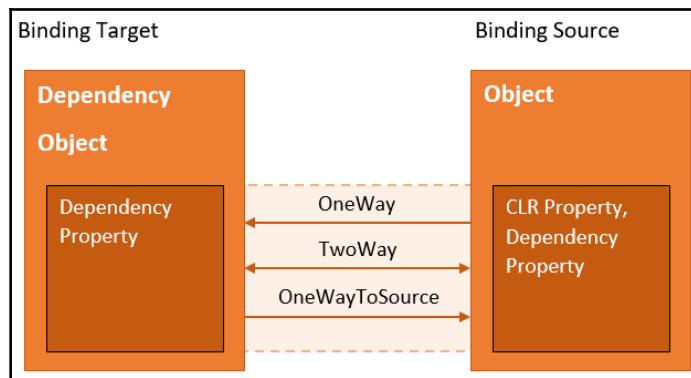
The `Register` method uses four parameters. The first one is the CLR property name that you defined for the getter and setter. The second parameter is the return type of the property. The third is the class handler (derived from `DependencyObject`), where you declare the dependency property. The fourth one is the extended property metadata, where you can set the default value.

Data binding in WPF

Data binding is a technique to establish a connection between the UI of the application and the business logic, in order to create data synchronization between them. Although you can directly access UI controls from the code to update their content, data binding became the preferred way to update the UI layer due to its automatic update notification mechanism.

To make data binding work, both sides of the binding must provide a change notification to the other side. The source property of data binding can be a normal .NET CLR property or a dependency property, but the target property must be a dependency property.

Here's a diagram detailing the various modes of data binding in WPF:



Data binding is typically done in XAML by using the `{Binding}` markup extension. It can be unidirectional (that is, source > target or target > source) or bidirectional (source < > target), known as Mode, and is defined in four categories:

- **OneWay**: This type of unidirectional data binding causes the source property to automatically update the target property. The reverse is not possible here. For example, if you want to display a label/text in the UI based on some condition in the backend code or business logic, you need to use one-way data binding as you don't need to update the property back from the UI:

```
<TextBlock Text="{Binding AuthorName}"/>
<TextBlock Text="{Binding AuthorName, Mode=OneWay}"/>
<TextBox Text="{Binding AuthorName}"/>
<TextBox Text="{Binding AuthorName, Mode=OneWay}"/>
```

- **TwoWay**: This type of binding is bidirectional data binding, where both the source property and the target property can send update notifications. This is also applicable for editable forms where users can change the value displayed in the UI. For example, the `Text` property of a `TextBox` control supports this type of data binding:

```
<TextBox Text="{Binding AuthorName, Mode=TwoWay}"/>
```

- **OneWayToSource**: This is another unidirectional data binding that causes the target property to update the source property (the reverse of `OneWay` binding). Here, the UI sends notifications to the data context and no notifications are generated if the data context changes:

```
<TextBox Text="{Binding AuthorName, Mode=OneWayToSource}"/>
```

- **OneTime**: It causes the source property to initialize the target property. After that, no notifications will be generated. You should use this type of data binding when the source data does not change.



Remember, `OneWay` binding is the default data binding. When you set `Mode=OneWay` to the `Text` property of `TextBox`, the notification does not generate to the source property if the user updates the value of the `Text` property in the UI.

Let's discuss data binding with a small and simple demonstration. First, create a new WPF project using Visual Studio 2019 and name it `Demo.WPF.DataBinding`. It will have already created XAML files named `MainWindow.xaml` and `MainWindow.xaml.cs`.

Open the `MainWindow.xaml.cs` file and add three dependency properties of the string type, named `AuthorName`, `BookName`, and `PublishDate`. You can use the `propdp` snippet to generate the dependency properties. After the addition of the three properties, your code will look like this:

```
namespace Demo.WPF.DataBinding
{
    public partial class MainWindow : Window
    {
        public string AuthorName
        {
            get { return (string)GetValue(AuthorNameProperty); }
            set { SetValue(AuthorNameProperty, value); }
        }

        public string BookName
        {
            get { return (string)GetValue(BookNameProperty); }
            set { SetValue(BookNameProperty, value); }
        }

        public string PublishDate
        {
            get { return (string)GetValue(PublishDateProperty); }
            set { SetValue(PublishDateProperty, value); }
        }

        public static readonly DependencyProperty AuthorNameProperty =
            DependencyProperty.Register("AuthorName", typeof(string),
            typeof(MainWindow), new PropertyMetadata(
                "Kunal Chowdhury"));
        public static readonly DependencyProperty BookNameProperty =
            DependencyProperty.Register("BookName", typeof(string),
            typeof(MainWindow), new PropertyMetadata(
                "Mastering Visual Studio"));
        public static readonly DependencyProperty PublishDateProperty =
            DependencyProperty.Register("PublishDate", typeof(string),
            typeof(MainWindow), new PropertyMetadata("2019"));

        public MainWindow()
        {
            InitializeComponent();
        }
    }
}
```

Now, as we have three dependency properties already defined, we can start creating the object binding. First, let's give the name `mainWindow` to `Window` to easily identify the context of the backend code:

```
<Window x:Class="Demo.WPF.DataBinding.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Name="mainWindow" Title="Data Binding Demo"
    Height="350" Width="700" FontSize="12">
```

Then, we need to set the data context of the `Grid` panel. We can use a data binding here, and can tell the `Grid` panel to use the same class by assigning `mainWindow` as the element name of the context: `<Grid DataContext="{Binding ElementName=mainWindow}"`.

Now divide the `Grid` panel into a four rows and five columns. Add the following `TextBlock` and `TextBox` controls inside the `Grid` and align those in rows and columns properly:

```
<TextBlock Text="Author Name" Grid.Row="0" Grid.Column="0"/>
<TextBlock Text=":" Grid.Row="0" Grid.Column="1"
    HorizontalAlignment="Center"/>
<TextBox Text="{Binding AuthorName, Mode=OneWay}"
    Grid.Row="0" Grid.Column="2"/>
<TextBlock Text="(Mode=OneWay)" Grid.Row="0" Grid.Column="4"/>

<TextBlock Text="Book Name" Grid.Row="1" Grid.Column="0"/>
<TextBlock Text=":" Grid.Row="1" Grid.Column="1"
    HorizontalAlignment="Center"/>
<TextBox Text="{Binding BookName, Mode=TwoWay}" Grid.Row="1"
    Grid.Column="2"/>
<TextBlock Text="(Mode=TwoWay)" Grid.Row="1" Grid.Column="4"/>

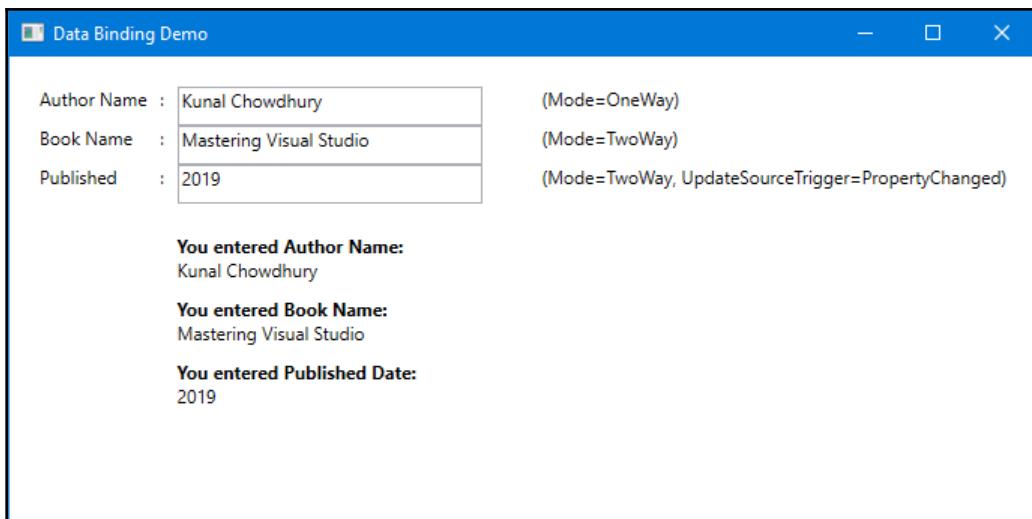
<TextBlock Text="Published" Grid.Row="2" Grid.Column="0"/>
<TextBlock Text=":" Grid.Row="2" Grid.Column="1"
    HorizontalAlignment="Center"/>
<TextBox Text="{Binding PublishDate, Mode=TwoWay,
    UpdateSourceTrigger=PropertyChanged}"
    Grid.Row="2" Grid.Column="2"/>
<TextBlock Text="(Mode=TwoWay,
    UpdateSourceTrigger=PropertyChanged)" Grid.Row="2" Grid.Column="4"/>
```

For the first `TextBox` instance, the `Text` property has been bound to `AuthorName` using one-way data binding. The second `TextBox` instance uses the two-way data binding mode with the `BookName` property. The third `TextBox` instance also uses a two-way data binding process with the `PublishDate` property, but has specified `UpdateSourceTrigger=PropertyChanged`. This tells the system to notify us whenever the property changes.

Now add the following `StackPanel` with some additional `TextBlock` controls and align them vertically:

```
<StackPanel Grid.Row="3" Grid.Column="2" Margin="0 20 0 0">
    <TextBlock Text="You entered Author Name:">
        <TextBlock>FontWeight="Bold"/>
    <TextBlock Text="{Binding AuthorName}" />
    <TextBlock Text="You entered Book Name:">
        <TextBlock>FontWeight="Bold" Margin="0 10 0 0"/>
    <TextBlock Text="{Binding BookName}" />
    <TextBlock Text="You entered Published Date:">
        <TextBlock>FontWeight="Bold" Margin="0 10 0 0"/>
    <TextBlock Text="{Binding PublishDate}" />
</StackPanel>
```

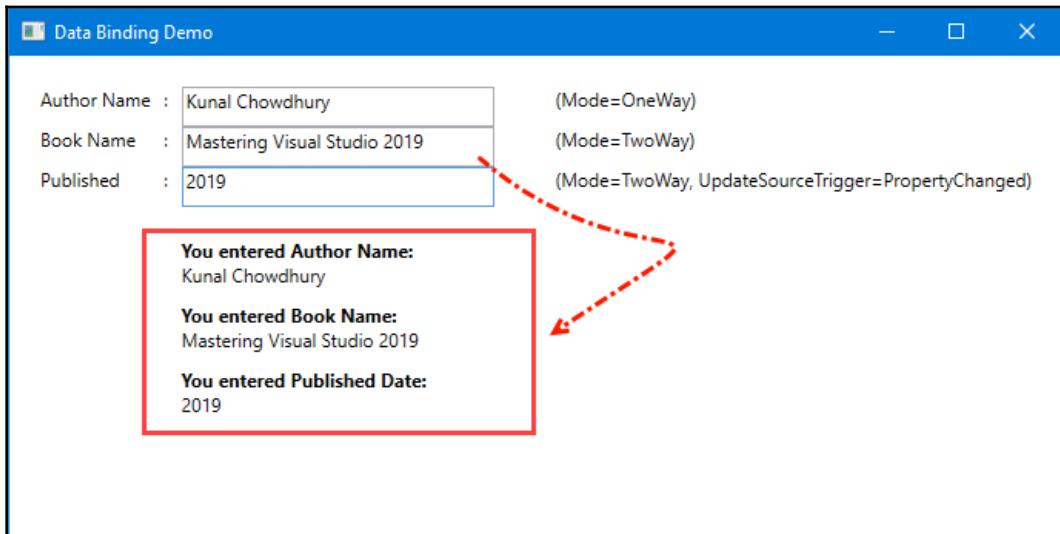
When you build and execute the preceding code, you will see the following window pops up with a few `TextBlock` and `TextBox` instances as defined in the XAML. The values in the `TextBox` instance controls are populated from the default value that we specified at the time of declaring the dependency properties:



Now, when you try modifying the text of the first `TextBox` (**Author Name**), it will not update the label that we placed below. As it has a `OneWay` data binding associated with it, no update notification will be sent out, even if you trigger a `LostFocus` on that `TextBox` control.

If you change the text of the second `TextBox` (**Book Name**), no notification will be carried out unless you trigger a `LostFocus` on it. The `TwoWay` data binding will act here to update the associated `TextBlock` automatically.

The third `TextBox` control also has `TwoWay` data binding associated with it, but it also has a binding property that says to update the source when the property changes. So, if you start typing in the third box, you will see that the `TextBlock` control will immediately start getting the notification and update itself:



You can assign four sets of values to `UpdateSourceTrigger`, as follows:

- **Default:** When specified, it triggers based on the default association set by the target. For example, the default trigger for the `TextBox.Text` property is `LostFocus`.
- **LostFocus:** It triggers when the associated control loses its focus.
- **PropertyChanged:** Triggers when the associated property gets a change notification.
- **Explicit:** It only triggers when the application explicitly calls `UpdateSource` of the binding.

Using converters while data binding

In data binding, when the source object type and the target object type differ, value converters are used to manipulate data between the source and the target. This is done by writing a Converter class, implementing the `IValueConverter` interface. It consists of two methods, as follows:

- `Convert(...)`: This gets called when the source updates the target object.
- `ConvertBack(...)`: This gets called when the target object updates the source object.

The implementation of the value converter is simple. First, create a class in your project. We named it as `BoolToColorConverter` and implemented it from `IValueConverter`. Implement the preceding two methods that are part of the interface.

For demonstration purposes, the class will accept Boolean values as input and convert them to a `SolidColorBrush` object. This needs to be done in the `Convert` method. You can also add the revert conversion to the `ConvertBack` method; however, for this demonstration, we will not implement it. Instead, we will throw `NotImplementedException`.

Here's the converter code for your reference:

```
using System;
using System.Globalization;
using System.Windows.Data;
using System.Windows.Media;

namespace Demo.WPF.Converters
{
    public class BoolToColorConverter : IValueConverter
    {
        public object Convert(object value, Type targetType,
            object parameter, CultureInfo culture)
        {
            var boolValue = value is Boolean ?
                bool.Parse(value.ToString()) : false;
            return new SolidColorBrush(boolValue ? Colors.Red :
                Colors.Green);
        }

        public object ConvertBack(object value, Type targetType,
            object parameter, CultureInfo culture)
        {
            throw new NotImplementedException();
        }
    }
}
```

```
    }  
}
```

Now we need to attach the Converter class to our XAML page, load it to memory, and create the data binding. Let's first declare the `xmllns` namespace for our converter. As it is in the same project, a namespace declaration will be sufficient:

1. Once the `xmllns` declaration is done, we need to load the converter into the window's resource and define a key for it, so that it can be easily accessible:

```
<Window ...  
    <Window.Resources>  
        <converters:BoolToColorConverter  
            x:Key="BoolToColorConverter"/>  
    </Window.Resources>  
    .  
    .  
    .  
</Window>
```

2. The next step is to create the UI and complete the data binding. Let's first add one border and a checkbox in the XAML page. Here, instead of creating any property in the backend code, we will directly use the control properties. As per this demonstration, the `Background` property of `Border` will change based on the checked status of the checkbox named `chkColor`.
3. Let's create a data binding for the border's `Background` property. The binding path will be the `IsChecked` property of the checkbox, whose name we defined as `chkColor`. Therefore, the `ElementName` will point to the name of the checkbox control, as shown in the following line of code:

```
<Border Background="{Binding Path=IsChecked,  
    ElementName=chkColor}"/>
```

4. Now, as the `IsChecked` property returns the Boolean value and the `Background` property accepts the `Brush` value, we need to use the converter with the binding expression. As we have already defined the converter as a resource of the window, we just need to reference it here as `StaticResource`. The code can now be written as follows:

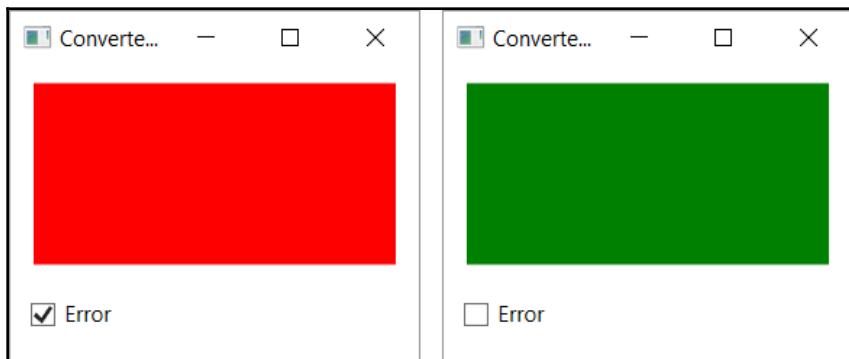
```
<Border Background="{Binding Path=IsChecked, ElementName=chkColor,  
    Converter={StaticResource BoolToColorConverter}}"
```

Here is the complete code for your easy reference:

```
<Window x:Class="Demo.WPF.Converters.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:converters="clr-namespace:Demo.WPF.Converters"
Title="Converter Demo" Height="200" Width="240">
    <Window.Resources>
        <converters:BoolToColorConverter
            x:Key="BoolToColorConverter"/>
    </Window.Resources>
    <StackPanel Orientation="Vertical" VerticalAlignment="Top">
        <Border Background="{Binding Path=.IsChecked,
            ElementName=chkColor, Converter={StaticResource
            BoolToColorConverter}}" Width="200"
            Height="100" Margin="10"/>
        <CheckBox x:Name="chkColor" IsChecked="True"
            Content="Error" Margin="10"/>
    </StackPanel>
</Window>
```

Now, if we run the preceding sample code, it will first load the checkbox as checked because we specified the `.IsChecked` property of the control as `True`. As the value is `True`, it will be passed to the converter and will return a red brush that will be assigned as the background color of the border control (as shown on the left-hand side of the screenshot).

Uncheck the checkbox to set its value to `False`, resulting in the converter returning the green brush, and you will see the color changed to green (as shown in the right-hand side of the screenshot). If you check it again, you will see it as red:



When you work in a medium-or large-scale application, you will find it very useful to convert a value from one type to another. The most common converters are `BoolToVisibilityConverter` and `VisibilityToBoolConverter`.

Using triggers in WPF

A trigger enables you to change property values when certain conditions are satisfied. It can also enable you to take actions based on property values by allowing you to dynamically change the appearance and/or the behavior of your control without writing additional codes in the backend code classes.

In WPF, the triggers are usually defined in a style or in the root of an element that is applied to that specific control. There are five types of triggers:

- The property trigger
- The multi trigger
- The data trigger
- The multi-data trigger
- The event trigger

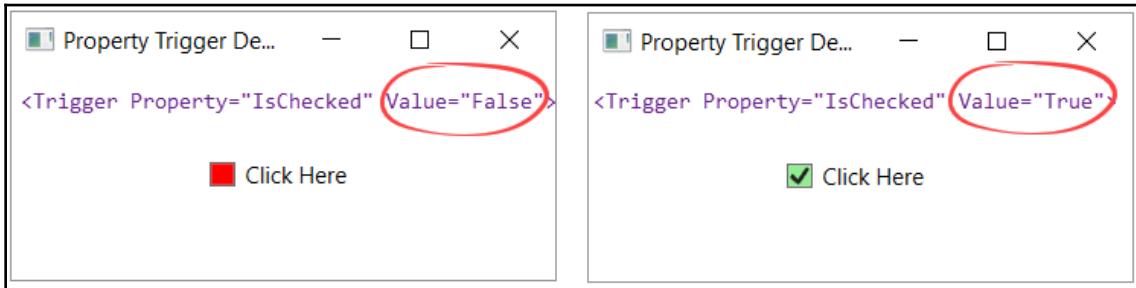
The property trigger

The most common trigger is the property trigger, which can be simply defined in XAML with a `<Trigger>` element. It triggers when a specific property on the owner control changes to match a specified value:

```
<Style TargetType="{x:Type CheckBox}">
    <Style.Triggers>
        <Trigger Property="IsChecked" Value="True">
            <Setter Property="Background" Value="LightGreen"/>
        </Trigger>
        <Trigger Property="IsChecked" Value="False">
            <Setter Property="Background" Value="Red"/>
        </Trigger>
    </Style.Triggers>
</Style>
```

The preceding XAML code snippet creates `Style` for the `CheckBox` control having two triggers associated with its `.IsChecked` property. When the value is `True`, the background of the checkbox will set to `LightGreen`, and when it is `False`, the background will set to `Red`.

If we run the example with a checkbox control in it, we will see the following results based on its respective checked status:



Try changing the checked status and see the result.

The MultiTrigger property

This is almost like the property trigger, but, here, it is used to set an action on multiple property changes and will execute them when all the conditions within `MultiTrigger.Conditions` are satisfied:

```
<Style TargetType="{x:Type CheckBox}">
    <Style.Triggers>
        <MultiTrigger>
            <MultiTrigger.Conditions>
                <Condition Property="IsEnabled" Value="True"/>
                <Condition Property="IsChecked" Value="False"/>
            </MultiTrigger.Conditions>
            <MultiTrigger.Setters>
                <Setter Property="Background" Value="Red"/>
                <Setter Property="Opacity" Value="0.5"/>
            </MultiTrigger.Setters>
        </MultiTrigger>
    </Style.Triggers>
</Style>
```

Here, in this example, we have a multi-trigger associated with the CheckBox control's style. When the control is enabled and unchecked, the trigger will get fired, setting the background of the control to the color red and the opacity to 50%:



This way, you can use the property trigger to fire based on a specific property value and keep your backend code (that is, the backend code file) clean.

The data trigger

As the name suggests, `DataTrigger` applies the property value to perform a set of actions on the data that has a data binding with the UI element. This is represented by the `<DataTrigger>` element.

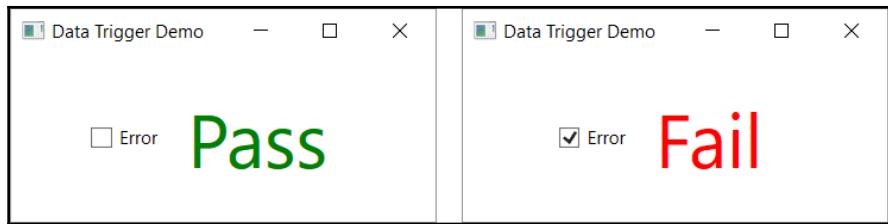
Here's an example of writing a data trigger to a UI element:

```
<StackPanel Orientation="Horizontal"
            HorizontalAlignment="Center"
            VerticalAlignment="Center">
    <CheckBox Name="chkError" Content="Error"
              VerticalAlignment="Center"/>
    <TextBlock Margin="20, 0" FontSize="50">
        <TextBlock.Style>
            <Style TargetType="TextBlock">
                <Setter Property="Text" Value="Pass" />
                <Setter Property="Foreground" Value="Green" />
                <Style.Triggers>
                    <DataTrigger Binding="{
                        Binding ElementName=chkError, Path=IsChecked}">
                        <Value>True</Value>
                        <Setter Property="Text"
                               Value="Fail" />
                        <Setter Property="Foreground"
                               Value="Red" />
                    </DataTrigger>
                </Style.Triggers>
            </Style>
        </TextBlock>
    </TextBlock>
</StackPanel>
```

```
</DataTrigger>
</Style.Triggers>
</Style>
</TextBlock.Style>
</TextBlock>
</StackPanel>
```

In the preceding example, we have two UI controls: a checkbox and a text block. The text block has a data trigger set to it, which has a data binding with the `.IsChecked` property of the checkbox.

When you run the preceding example, we will see the following result based on the checked status of the `CheckBox` control:



When the property value matches `True`, it will trigger and set the text as `Fail` with a foreground color of red. In default cases (that is, when the value matches `False`), the normal case will execute and show the text `Pass` in the color green.

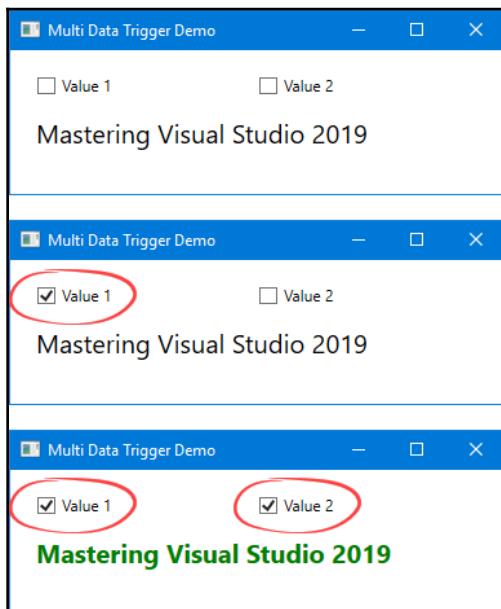
The multi data trigger

This is the same as the data trigger, but you can set property values based on multiple conditions defined in `MultiDataTrigger.Conditions`. Property values are defined in `MultiDataTrigger.Setters`, as shown in the following code snippet:

```
<StackPanel Orientation="Vertical" Margin="20">
    <UniformGrid>
        <CheckBox x:Name="chkOne" Content="Value 1"/>
        <CheckBox x:Name="chkTwo" Content="Value 2"/>
    </UniformGrid>
    <TextBlock Text="Mastering Visual Studio 2019" FontSize="20">
        <TextBlock.Style>
            <Style TargetType="TextBlock">
                <Style.Triggers>
                    <MultiDataTrigger>
                        <MultiDataTrigger.Conditions>
                            <Condition Binding="{Binding
```

```
        ElementName=chkOne,
        Path=IsChecked} " Value="True"/>
    <Condition Binding="{Binding
        ElementName=chkTwo,
        Path=IsChecked} " Value="True"/>
    </MultiDataTrigger.Conditions>
    <MultiDataTrigger.Setters>
        <Setter Property="FontWeight"
            Value="Bold"/>
        <Setter Property="FontSize" Value="36"/>
        <Setter Property="Foreground"
            Value="Green"/>
    </MultiDataTrigger.Setters>
</MultiDataTrigger>
</Style.Triggers>
</Style>
</TextBlock.Style>
</TextBlock>
</StackPanel>
```

When you run the preceding code, there will be two unchecked checkbox controls and a text block with a font size of 20. As per the conditions mentioned in the code, you need to check both the checkboxes to trigger the MultiDataTrigger conditions, and change the font weight, font size, and foreground color of the text:



Check and uncheck the checkboxes in the UI to see the results. The output will be based on multiple data values.

The event trigger

Event triggers are generally used to perform actions when the routed events of the associated `FrameworkElement` rises. This is mainly used in animations to control the look of the control when a certain UI event is raised.

In the following example, we have a `TextBlock` control. By default, its size is set to 30 and it has an opacity level of 20%. It has two events, `MouseEnter` and `MouseLeave`, associated with it. Now, when hovering over the `TextBlock`, we need to specify an animation to grow the font size to 50 and the opacity to 100%. Similarly, when the mouse leaves, we need to bring the text back to its initial state.

Here, the event trigger will help you to perform this within the XAML page:

```
<TextBlock Text="Hover here" FontSize="30" Opacity="0.2"
           HorizontalAlignment="Center"
           VerticalAlignment="Center">
    <TextBlock.Style>
        <Style TargetType="TextBlock">
            <Style.Triggers>
                <EventTrigger RoutedEvent="MouseEnter">
                    <EventTrigger.Actions>
                        <BeginStoryboard>
                            <Storyboard>
                                <DoubleAnimation Duration="0:0:0.500"
                                                    Storyboard.TargetProperty=
                                                    "FontSize" To="50" />
                                <DoubleAnimation Duration="0:0:0.500"
                                                    Storyboard.TargetProperty=
                                                    "Opacity" To="1.0"/>
                            </Storyboard>
                        </BeginStoryboard>
                    </EventTrigger.Actions>
                </EventTrigger>
                <EventTrigger RoutedEvent="MouseLeave">
                    <EventTrigger.Actions>
                        <BeginStoryboard>
                            <Storyboard>
                                <DoubleAnimation Duration="0:0:0.500"
                                                    Storyboard.TargetProperty=
                                                    "FontSize" To="30" />
                                <DoubleAnimation Duration="0:0:0.500"
```

```
        Storyboard.TargetProperty=
        "Opacity" To="0.2"/>
    </Storyboard>
</BeginStoryboard>
</EventTrigger.Actions>
</EventTrigger>
</Style.Triggers>
</Style>
</TextBlock.Style>
</TextBlock>
```

When you run the sample, by default, the text will be visible with a 20% opacity level. Now, if you hover the mouse over the text, you will see a smoothing animation that gradually changes the size of the font and the opacity level. When you move your mouse away from the text, it will smoothly move back to the initial state:



Event triggers are mostly used for animations, but you can also use them depending on your specific business needs. This removes the clutter of writing huge code in the backend code file.

Summary

In this chapter, we learned about WPF, its architecture, the XAML syntaxes, the WPF project structure, and the XAML designer. We have also covered how to work with the XAML markup to design a Windows application by discussing WPF controls and the various layout panels.

In addition to this, we discussed the new WPF property system using an example of a dependency property, looked at data binding and its implementation, along with considering converters and various types of triggers.

You can now confidently use WPF to build desktop applications for Windows. Now, let's move on to the next chapter, where we will discuss Microsoft Azure and learn how to accelerate cloud development to build and manage Azure websites and Azure App services.

3

Accelerate Cloud Development with Microsoft Azure

Microsoft Azure is an open, flexible, enterprise-grade cloud computing platform from Microsoft that was first released as **Windows Azure** on February 1, 2010 and then was renamed as **Microsoft Azure** on March 25, 2014. You can build, deploy, and manage applications and services using the Azure portal, which is globally available through the Microsoft data centers.

It basically delivers **Infrastructure as a Service (IaaS)**, **Platform as a Service (PaaS)**, and **Software as a Service (SaaS)** and supports different programming languages, tools, and frameworks that help us to build and manage applications/services.

In this chapter, we will discuss the following topics:

- Understanding cloud computing basics
- Creating your free Azure account
- Configuring Visual Studio 2019 for Azure development
- Creating an Azure website from the portal
- Managing Azure websites (web apps) from the portal
- Creating an Azure website from Visual Studio 2019

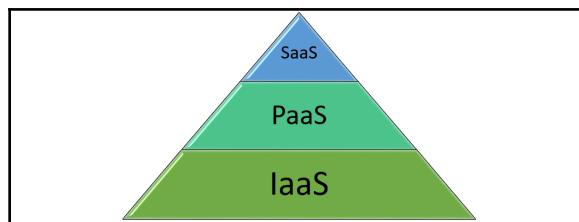
- Updating an existing Azure website from Visual Studio
- Building a mobile app
- Integrating Mobile App Service in a Windows application
- Scaling an App Service plan

Understanding cloud computing basics

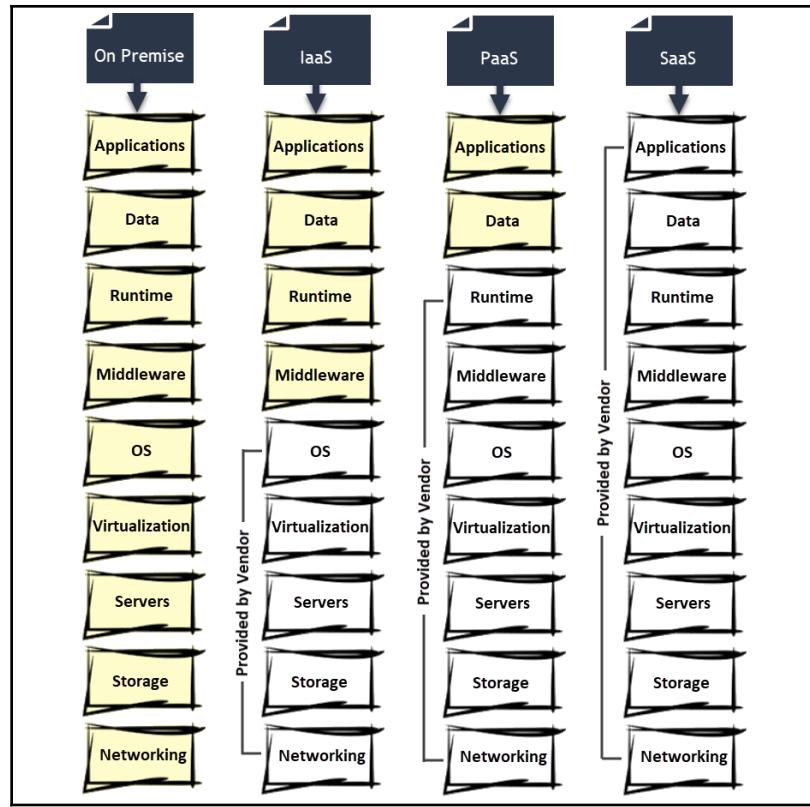
Cloud computing is a very broad concept. When you or your business needs to consider cloud services for your infrastructure or application deployment, you should understand the basics of it. Generally, there are three types of cloud computing models:

- IaaS
- PaaS
- SaaS

Consider the following diagram:



IaaS creates the main building blocks and PaaS sits on top of it, giving you a platform from which to use SaaS. SaaS is at the top of the cloud computing system. Let's illustrate all these basic service blocks with the following diagram:



Let's discuss each of these blocks with some suitable examples.

IaaS

In the fundamental building blocks of cloud computing resources, **IaaS** uses the physical computer hardware to build a virtual infrastructure in order to utilize the resources. You can create, reconfigure, resize, or remove any of the virtual resources in the data centers within a few minutes and then monitor them remotely.

IaaS is the most flexible cloud computing model. It allows you to automate the deployment of servers, process power, storage, and networking. In this service model, you don't have to purchase any hardware because the virtual ecosystem is provided by the vendor. **Amazon Web Service (AWS)**, **Azure Virtual Machines**, **Azure Container Service**, and **Google Compute Engine (GCE)** are some examples of well-known IaaS providers.

PaaS

PaaS is one step above IaaS. It provides a platform where you can develop and deploy software. It makes development, testing, and deployment simpler, faster, and more cost effective by providing the programming language to interact with services, databases, servers, and/or storage without having to deal with the infrastructure in which it is being used.

Since PaaS is built on top of virtualization technology, the vendor of such services provides the physical/virtual environment with frameworks and runtimes to manage your applications and data. **Google App Engine**, Red Hat's **OpenShift**, and **Heroku** are some examples of well-known PaaS services. **Azure App Service** and **Azure CDN** are examples of PaaS on Azure.

SaaS

SaaS is the top layer of cloud computing and is typically built on top of the solution given by PaaS. SaaS uses the web to deliver software applications, which are managed by third-party vendors, to end users. Thus, it's the most popular cloud service for consumers because it reduces the cost of software ownership by removing the need for technical staff to manage the installation, upgrades, and licensing of the software.

The service is typically charged on a per-user or per-month basis and provides the flexibility to add or remove users at runtime without additional costs. **Office 365**, **Google Apps**, **DropBox**, and **OneDrive** are some well-known examples of SaaS.

Creating your free Azure account

To get started with application development with Microsoft Azure, you will first need to have an Azure account and have a basic understanding of the Azure portal. Before we start, let's create an Azure account.



If you want to learn about and try using Azure, Microsoft provides you 30 days' free trial to explore the cloud platform with \$200 of free credit to your new account.

To get started with the \$200 of free credit, go to <https://azure.microsoft.com/en-us/free/> and click on the **Start free** button:

The screenshot shows the Microsoft Azure landing page for free accounts. It highlights three key benefits: \$200 free credit, trying any Azure service for 30 days, and paying nothing at the end. A prominent green button invites users to 'Start free'. At the bottom, links provide alternative purchase options and contact information. A small inset window shows the Azure Marketplace interface, displaying various compute offerings like Windows Server 2012 R2 Datacenter and Ubuntu 16.04 LTS.

Use your Microsoft account (formerly your Windows Live ID) to log in to the portal. If you don't have a Microsoft account username, you will be able to create one from the same screen.

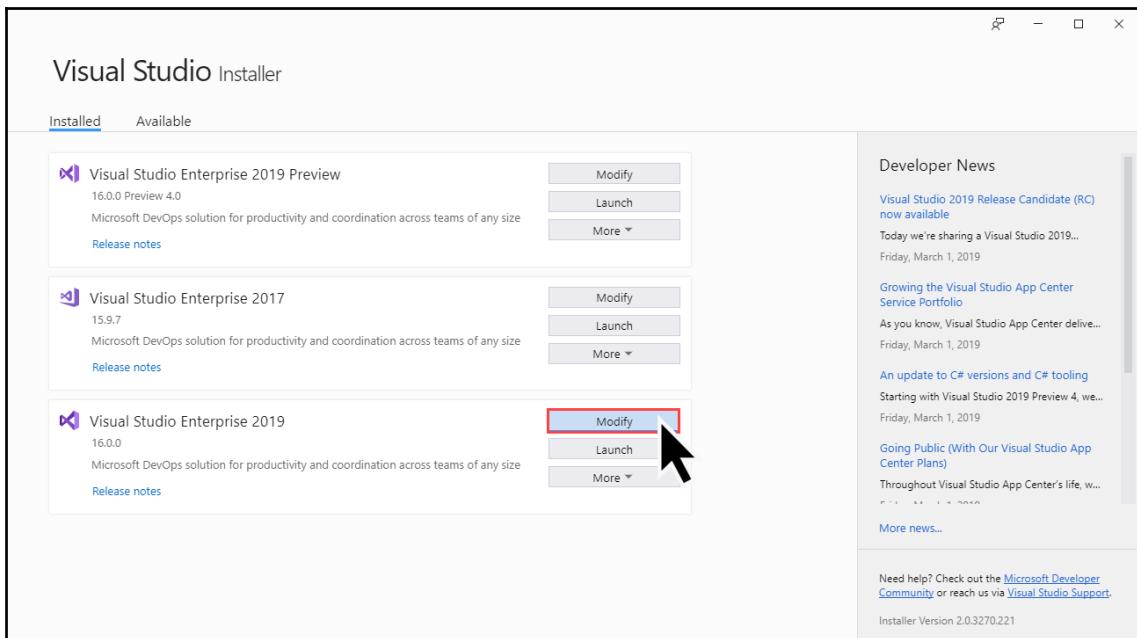
Once you log in to the portal for the first time, it will ask you to verify your identity. Enter your mobile number and credit card details to verify that you are a real person. Microsoft will not charge anything on your card, but a nominal \$1 charge may hold initially on your card for verification and will be removed within 3-5 days.

Once your free credit ends or you reach the expiration date, you won't be able to use the services that Azure offers unless you manually go and pay for your subscription. You can also opt for a Pay-As-You-Go subscription and set a monthly spending cap. In this case, when you reach that monthly spending limit, it will automatically suspend the service and you won't incur any additional costs.

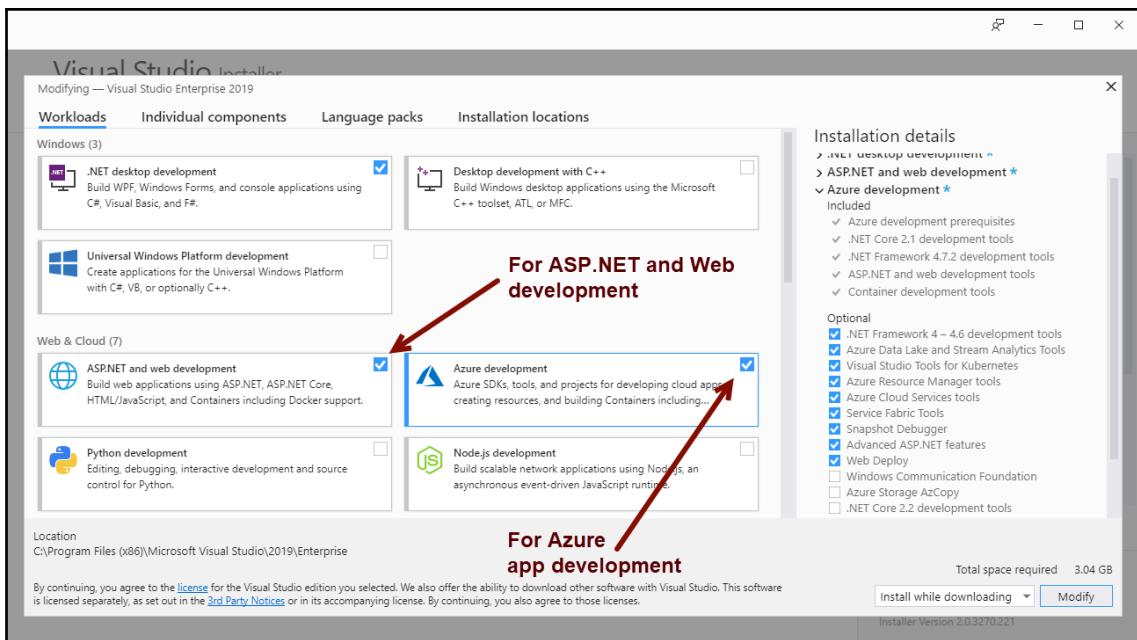
Configuring Visual Studio 2019 for Azure development

Before you start building Azure applications with Visual Studio 2019, you need to configure it by installing the required workloads. If you have not already installed the **Azure development** workload, open your Visual Studio 2019 installer:

1. As shown in the following screenshot, click on **Modify** to start customizing the instance of the IDE:



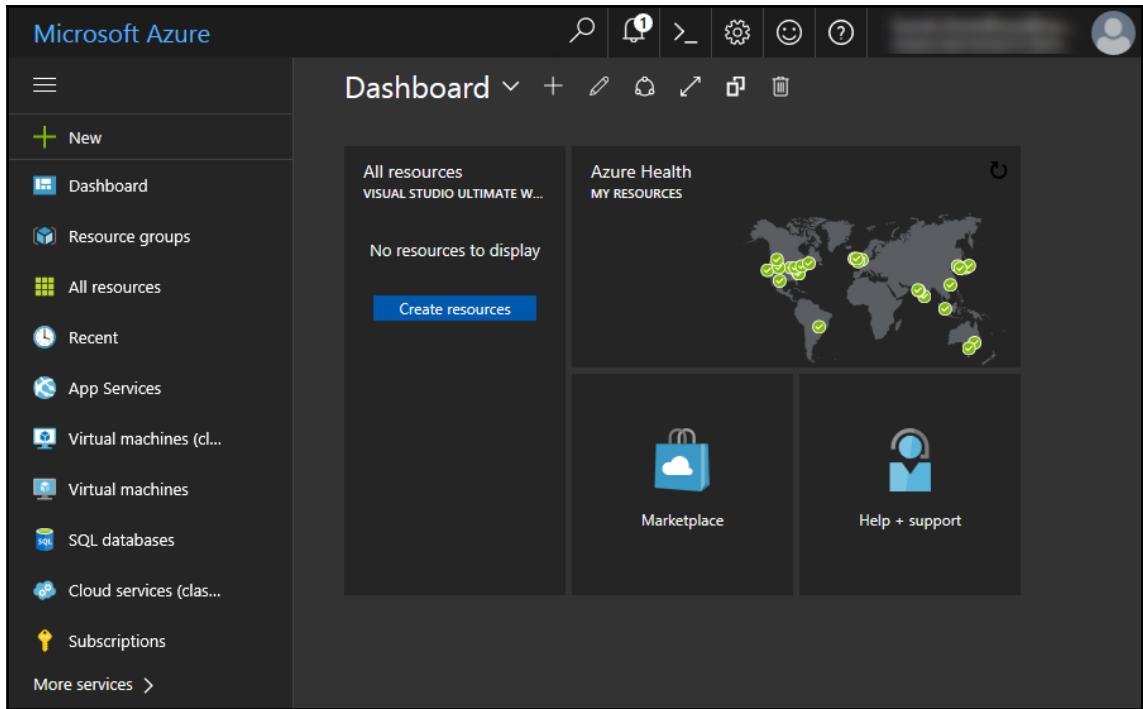
2. This will open the customization screen with the **Workloads** tab open. Scroll down to find the **Azure development** workload and select it. If you want to build ASP.NET applications, select the **ASP.NET and web development** workload, as shown in the following screenshot:



When you are ready, click the **Modify** button to start the installation process. It will take some time, depending on your internet bandwidth, to download and install the required components.

Creating an Azure website from the portal

Once you have your Azure account, you can visit <https://portal.azure.com> to start using the Microsoft Azure portal. When you log in to the portal, a dashboard will be shown on the screen, as follows:

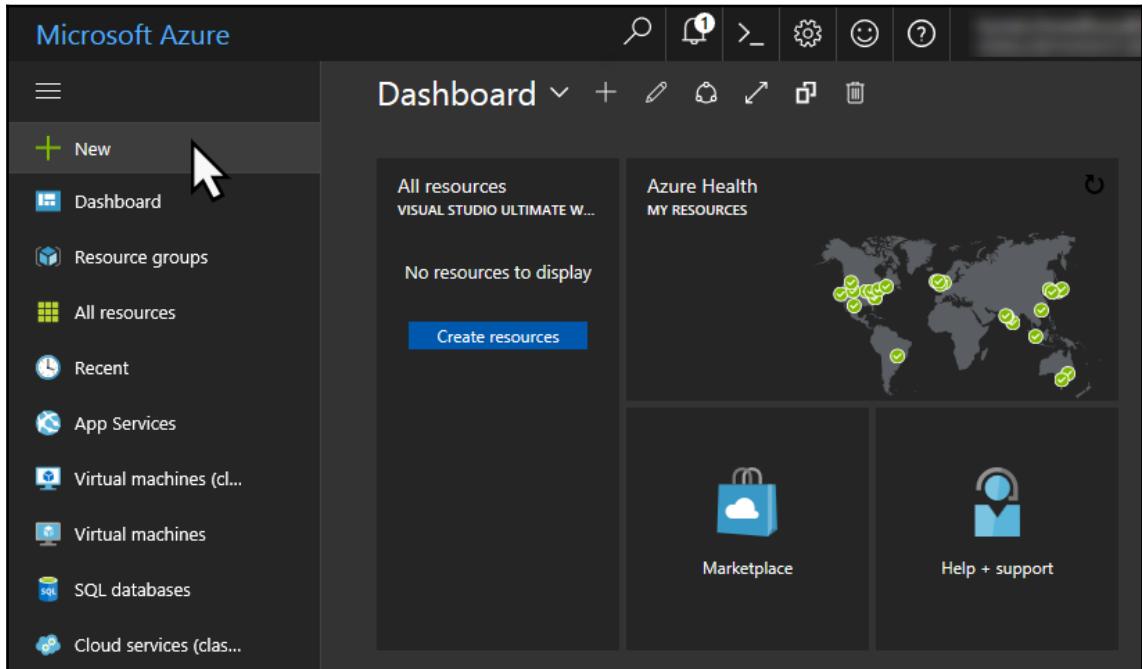


The categories on the left side allow you to create or manage your App Services, websites, virtual machines, databases, networks, IoT devices, and many other services that Azure supports.

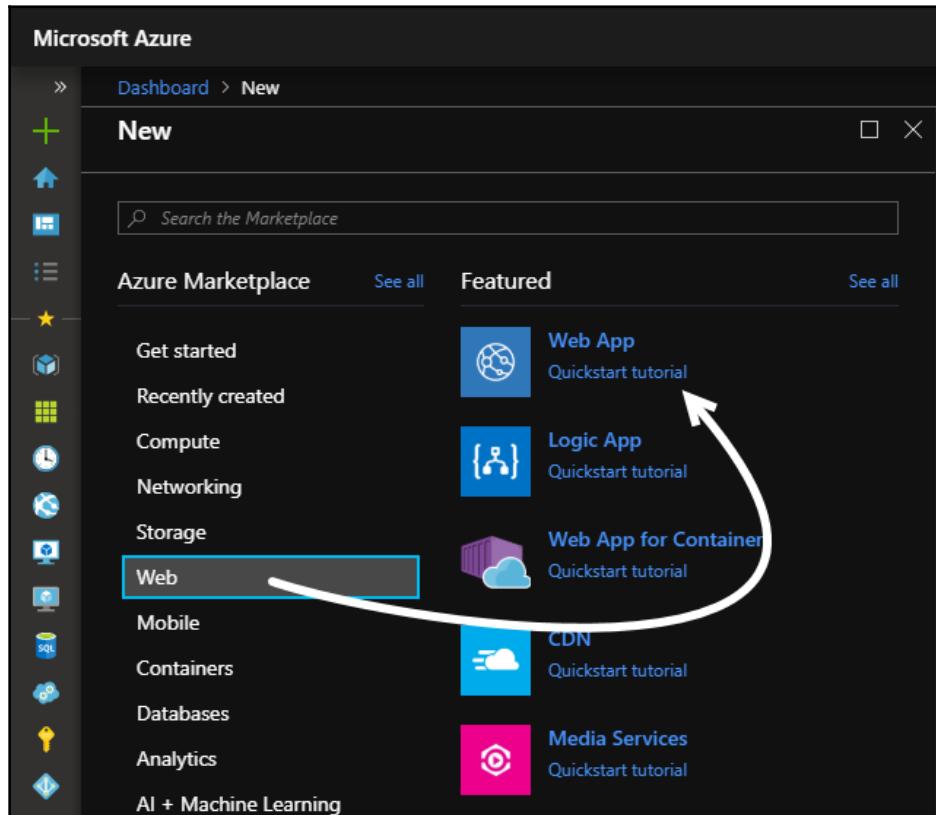
The Azure website comes under PaaS and provides an effortless way to build and deploy web applications in the management portal, under **Marketplace**, named as **Web App**. Let's start creating our first Azure website.

Creating a web application

Once you log in to the Azure management portal, click on the + icon or the + New label, as shown in the following screenshot. This will guide you through the creation of any services/resources that are currently available in Microsoft Azure:

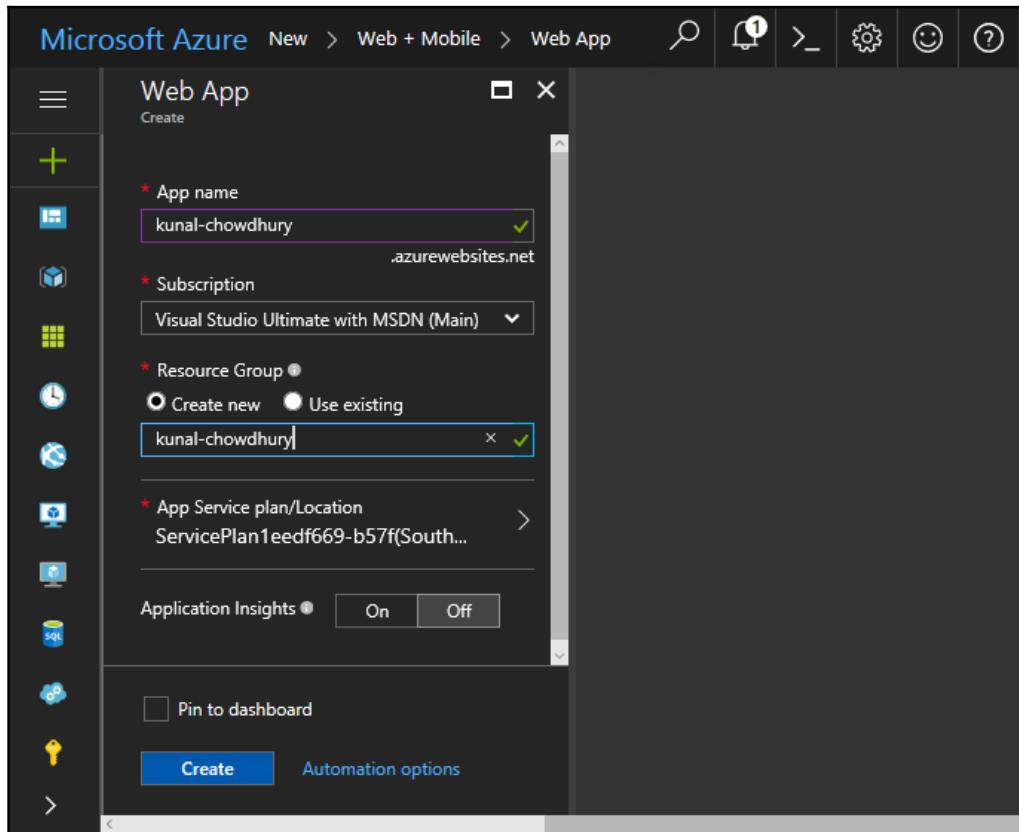


This will open a panel on the right-hand side with a list of the resource categories available from the Azure Marketplace. Select the one that you want to create. In our case, because we are going to create our website, we will select **Web** from the list:



Once you select **Web**, it will open another panel on the right side with a list of featured apps. This includes **Web App** (for websites), **Logic App** (for automated access and use of data), **CDN** (for globally distributed edge servers), **Media Services** (for encoding, storing, and streaming audio/video), and more.

As we are going to create a website, let's click on the featured app titled **Web App**. This will open another new panel on the right of the screen:



On the **Web App** screen, the wizard will ask you to input some details. Enter a name for the application. Make sure that it is globally unique because it's going to create a subdomain with the same name under `azurewebsites.net`.

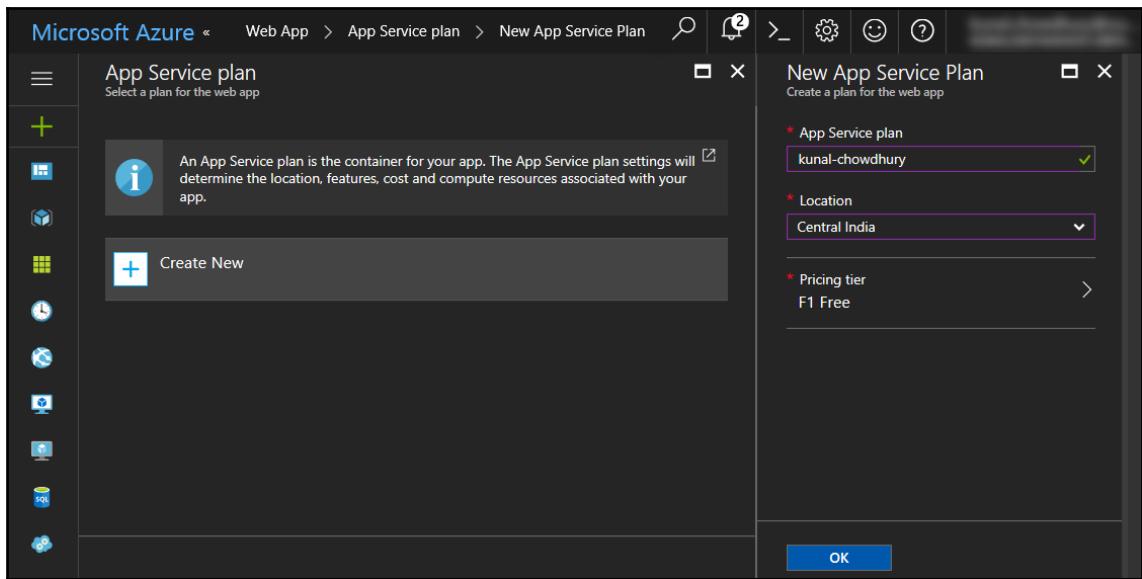
Select the subscription model that you are going to use. In your case, it will be different to the one shown in the preceding screenshot.

Then, select the **Create new** radio button to create a new **Resource Group**. Give it a new name. Generally, it's going to create the resource group with the same name as the application.

Creating an App Service plan

Next, you should select the **App Service plan/Location** from the list, if it already exists. An App Service plan is the container for your app, which will determine the location, features, cost, and compute resources associated with your app. So, set it appropriately to optimize the compute cost.

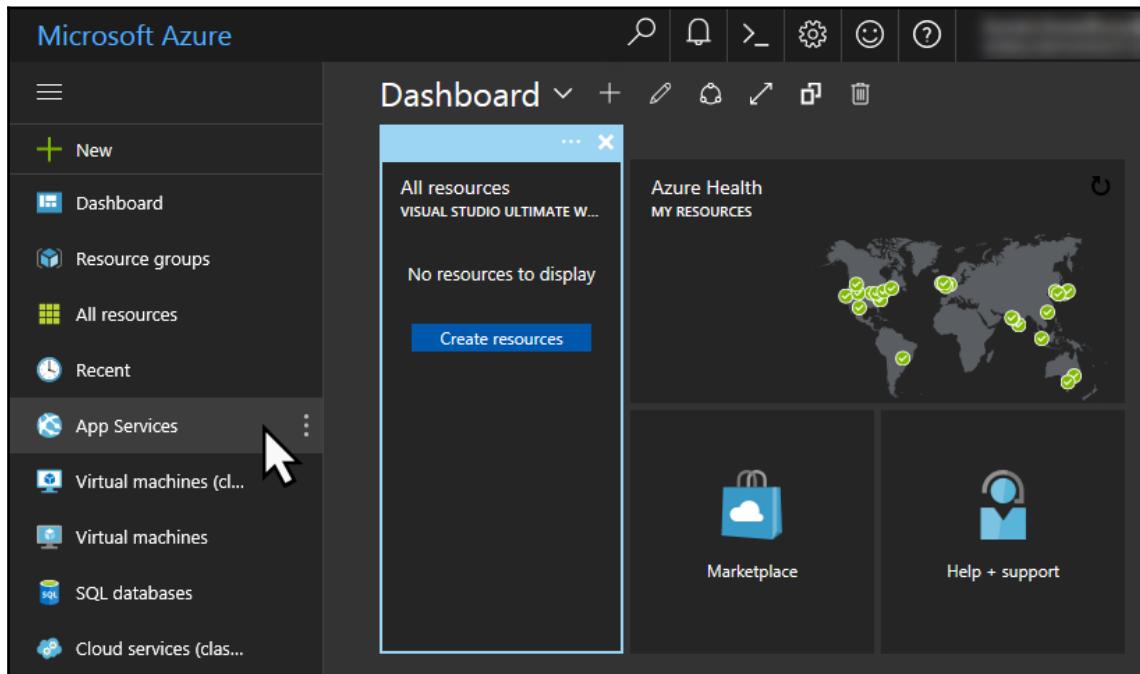
If there is no App Service plan in the list, create a new one and select it. In our case, we are going to create an App Service plan with a free pricing tier so there is no cost associated with hosting this demonstration app:



Once you select the App Service plan, you may want to detect and diagnose the quality issues in your web applications and web services. In this case, enable **Application Insights**, and leave everything else as is. When you are ready, click on the **Create** button. It will take some time for the web app to be created. Once it is created, move on the next point, managing Azure websites from the web portal.

Managing Azure websites (web apps) from the portal

Once the web app is created, navigate to **App Services** in the category list on the left-hand side, as shown in the following screenshot:



Here, it will list all the App Services that you have currently hosted on your Azure account. If you have a large list, you can easily search/filter it based on the options on this screen.

The website that we just created will be listed on this screen. Click on the name of the app that you used to create it:

The screenshot shows the Microsoft Azure App Services dashboard. On the left, there's a sidebar with various icons for different services like Storage, Functions, and Logic Apps. The main area is titled "App Services" and shows one item: "kunal2383@yahoo.com (Default Directory)". Below this, there are buttons for "+ Add", "Columns", and "Refresh". A search bar says "Filter by name..." and dropdowns show "Visual Studio Ultimate with MSDN (Main)", "All locations", and "No grouping". A table lists one item: "NAME" (kunal-chowdhury), "STATUS" (Running), "APP TYPE" (Web app), "APP SERVICE PLAN" (kunal-chowdhury), "LOCATION" (Central India), and "SUBSCRIPTION" (Visual Studio Ultim...). The row for "kunal-chowdhury" is highlighted with a blue background.

This will open another screen with detailed information about the application that you have selected. On this screen, you can check the activity log, monitor the requests and errors, modify the access control, diagnose and solve problems, create a backup of the site, update the custom domain, and more:

The screenshot shows the Microsoft Azure App Services Overview screen for the 'kunal-chowdhury' app service. The left sidebar contains a navigation menu with icons for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Deployment (Quickstart, Deployment credentials, Deployment slots, Deployment options, Continuous Delivery (Preview)), Settings (Application settings, Authentication / Authorization, Backups, Custom domains), and more.

Toolbar:

- Browse
- Stop
- Swap
- Restart
- Delete

Essentials:

- Resource group (change) **kunal-chowdhury**
- Status **Running**
- Location **Central India**
- Subscription name (change) **Visual Studio Ultimate with MSDN (Main)**
- Subscription ID **1fa4f721-0c6d-48ac-aea8-18d94ba910c0**
- URL **http://kunal-chowdhury.azurewebsites.net**
- App Service plan/pricing tier **kunal-chowdhury (Free)**
- FTP/deployment username **kunal-chowdhury\cwc-kc**
- FTP hostname **ftp://waws-prod-pn1-001.ftp.azurewebsites...**
- FTPS hostname **https://waws-prod-pn1-001.ftp.azurewebsite...**

Monitoring:

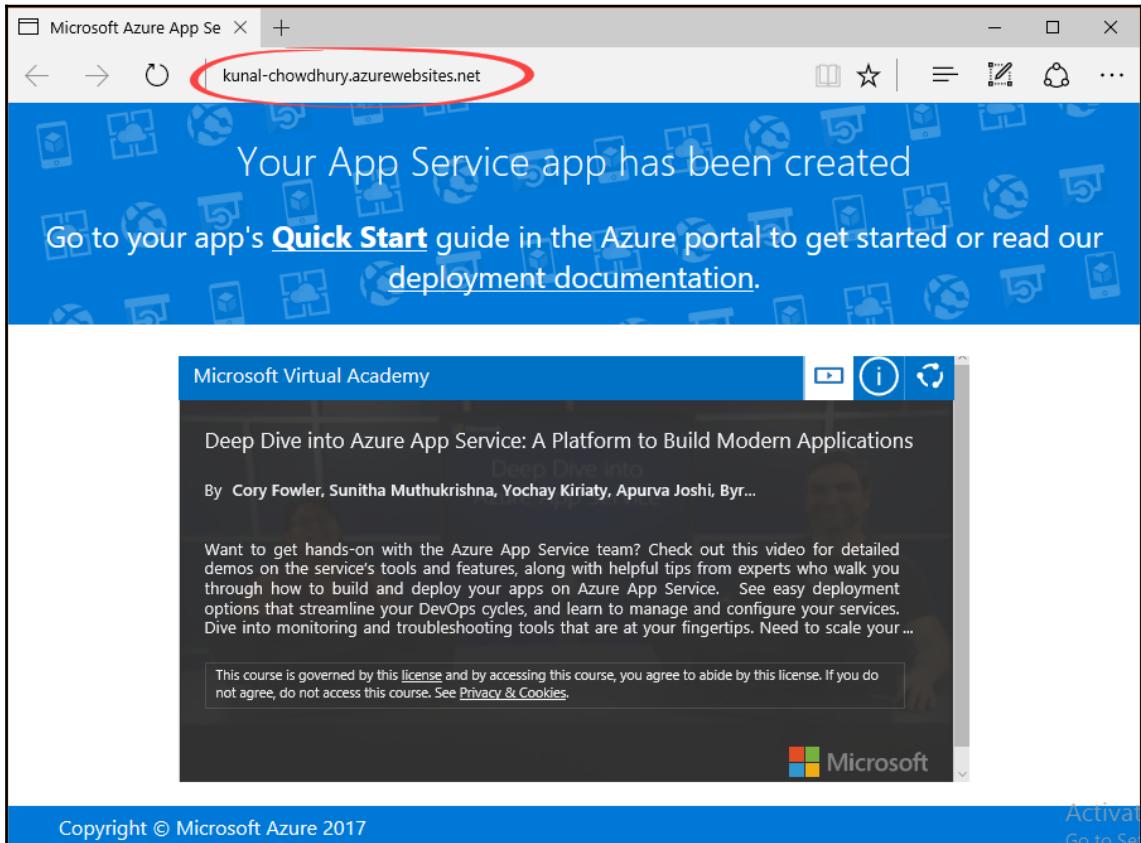
Requests and errors

The monitoring chart shows two spikes in requests around 22:30 and 22:45. The Y-axis represents the number of requests (0 to 1), and the X-axis shows the time from 22:00 to 22:45. A legend indicates that blue lines represent REQUESTS and red lines represent HTTP SERVER ERRORS.

Time	REQUESTS	HTTP SERVER ERRORS
22:00 - 22:30	~0.1	0
22:30	~0.9	0
22:45	~0.9	0
22:45 - 22:48	~0.1	0

At the top of the **Overview** screen, you will find few toolbar buttons that will allow you to **Browse** the site, **Start/Stop/Restart** the web application, and **Delete** it.

The URL was created by the name of the application (in our case, it is `http://kunal-chowdhury.azurewebsites.net`), and, when it is clicked, it will launch the website in a browser window. A generic website from the Microsoft template will be launched, as shown in the following screenshot:



It's a default page that Microsoft provides as part of the default template. You can redesign this page and publish the entire website from Visual Studio. In the next sections, we are going to see how this can be done.

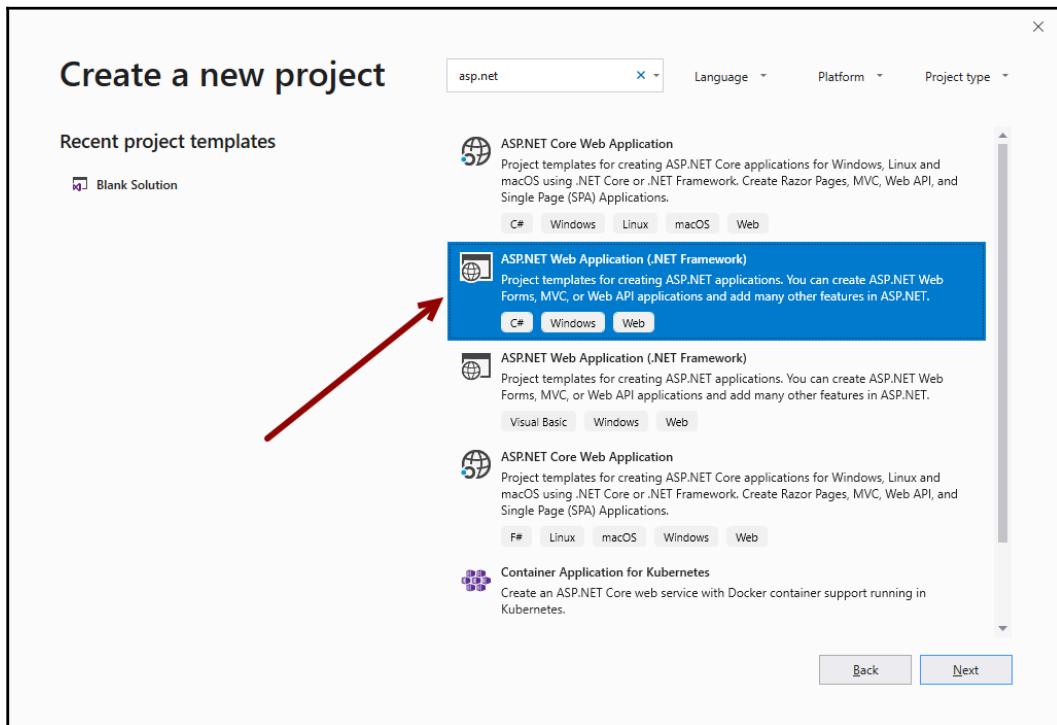
Creating an Azure website with Visual Studio 2019

Since we have already learned how to create an Azure website from the Azure management portal, let's learn how to create one with Visual Studio 2019. Make sure that you have the required components/workloads already installed (refer to the *Configuring Visual Studio 2019 for Azure development* section in this chapter).

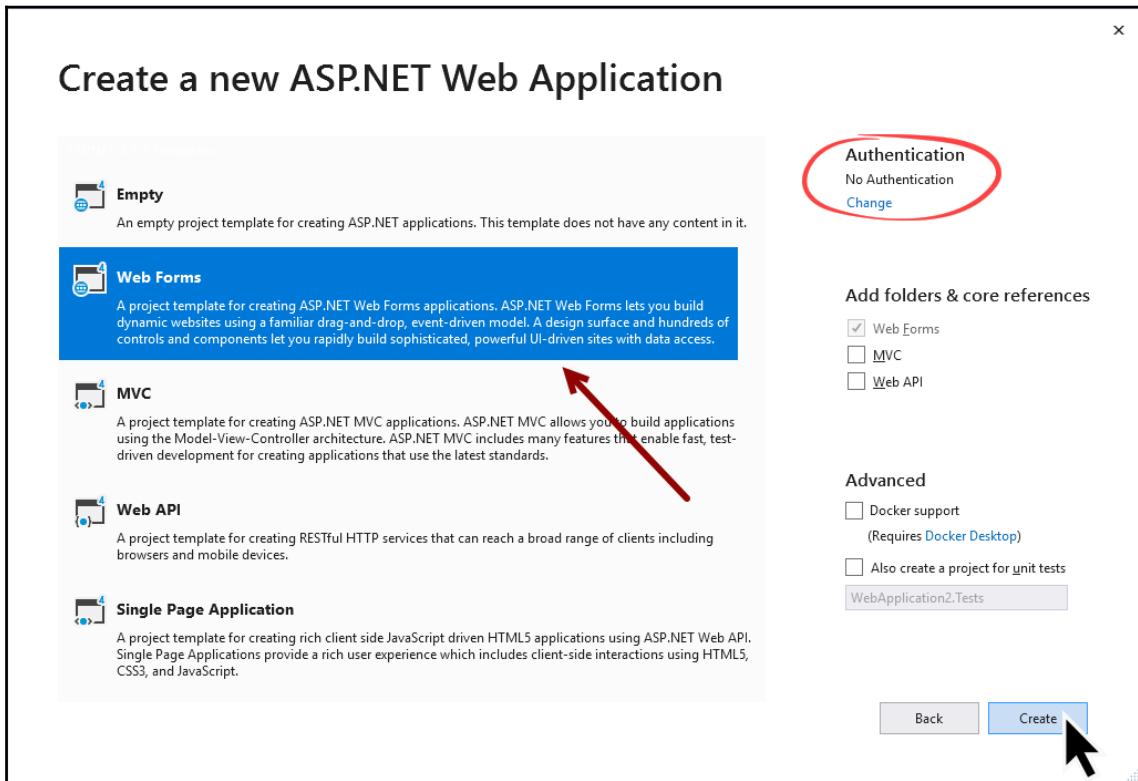
Creating an ASP.NET web application

Once you are ready to create an ASP.NET website and deploy it to Azure, open your Visual Studio 2019 instance. Follow these steps:

1. Create a new project by searching for `asp.net` and then selecting **ASP.NET Web Application (.NET Framework)**, as shown in the following **Create a new project** dialog:



2. In the next screen, provide a name for your project and click the **Create** button to start creating your web application from a template. The next screen will guide you through selecting the template that you want to use. There are a number of templates (**Empty project**, **Web Forms**, **MVC**, **Web API**, **Single Page Application**, and more) available for you to select.
3. We will select **Web Forms** here, with **No Authentication** support. When you are ready, click **Create** to continue:



4. Visual Studio will then create the project based on the template and the authentication configuration that you selected on the previous screen.
5. Now, from the **Solution Explorer**, open the **Default.aspx** page. It will already have a design and content specified by Visual Studio in its template. Replace the Content tag with the following code:

```
<asp:Content ID="BodyContent" ContentPlaceHolderID="MainContent"  
runat="server">
```

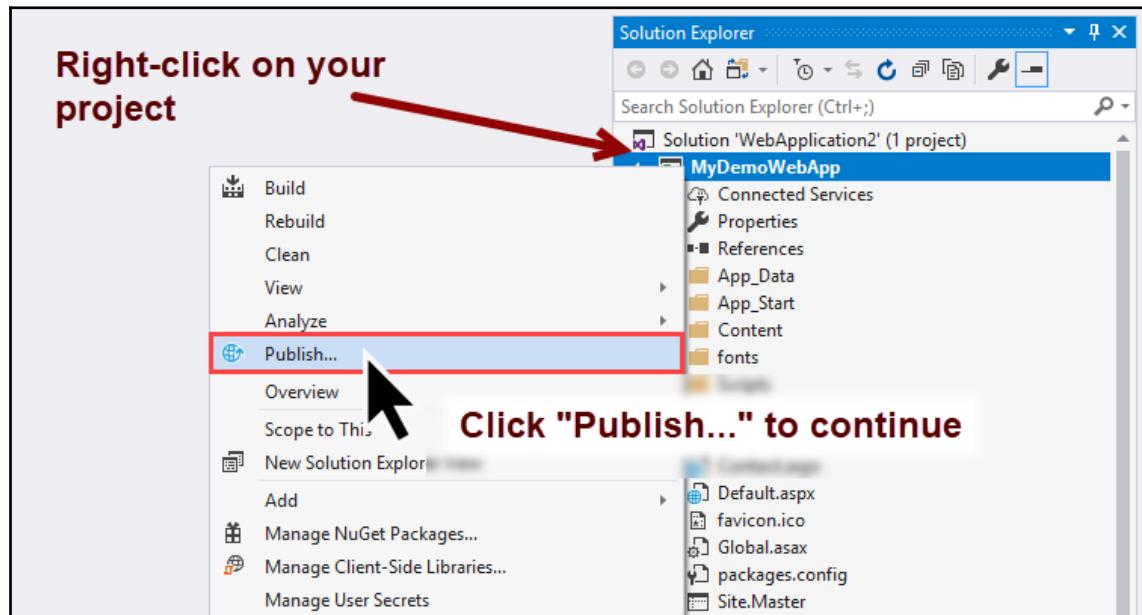
```
<div class="jumbotron">
    <h1>Hello Azure Website</h1>
    <p class="lead">It is a demo application to build Azure
website.</p>
    <p><a href="http://www.kunal-chowdhury.com" class="btn btn-
primary btn-lg">Visit site &raquo;</a></p>
</div>

</asp:Content>
```

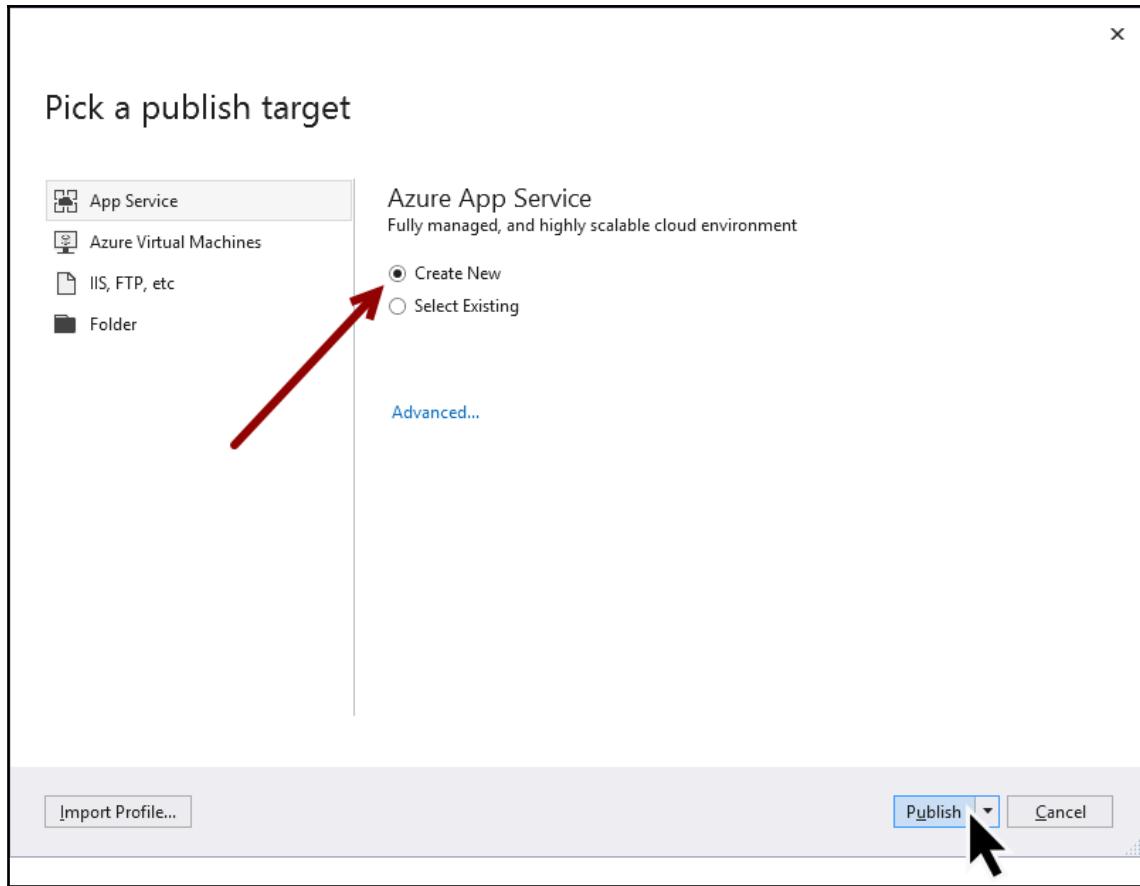
As we have just created an ASP.NET web application, let's jump into the next section to publish it to the cloud.

Publishing the web application to the cloud

When you are ready to publish your web application to the cloud as an Azure website, right-click on the project in **Solution Explorer**. A context menu will pop up on the screen. Click **Publish...**, as shown in the following screenshot:



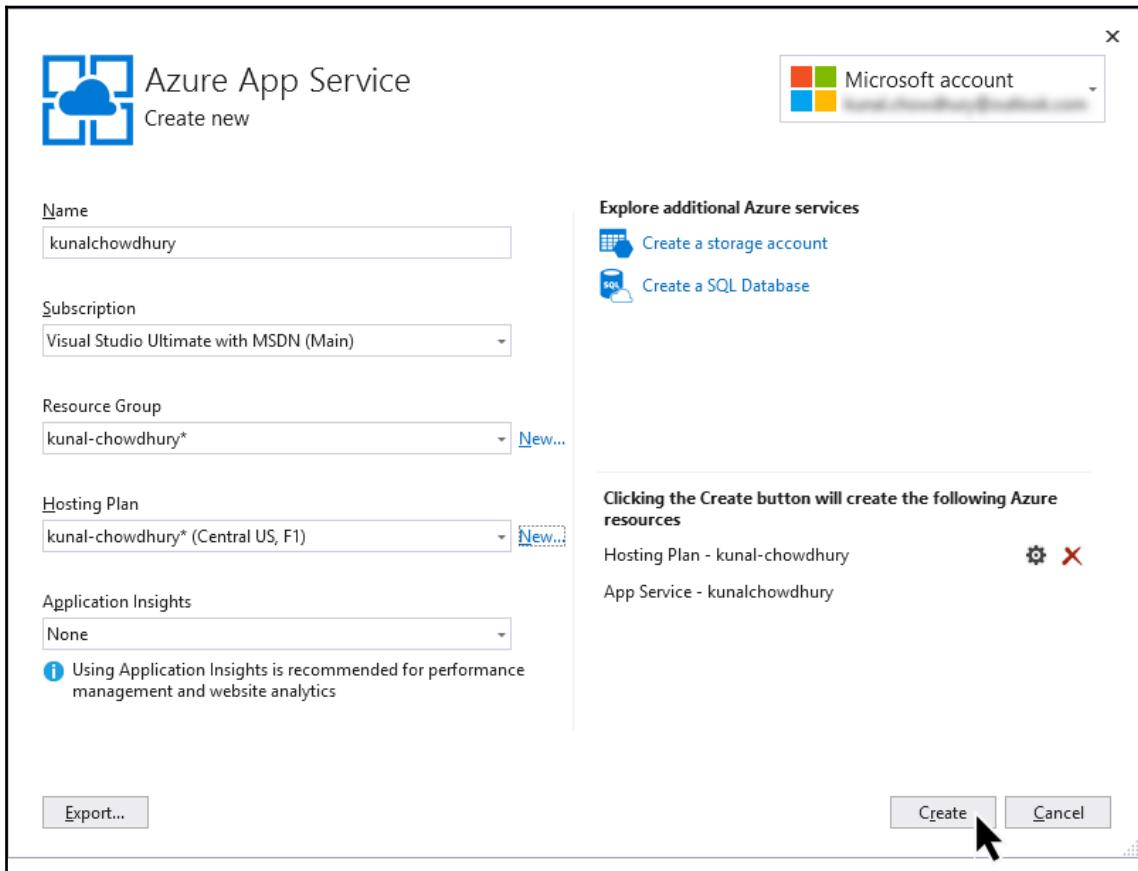
This will open the web app publishing wizard. As shown in the following screenshot, click on the **App Service** tab. There are two radio buttons, **Create New** and **Select Existing**. The first one will help you to create a fresh new Azure website, whereas the other will enable you to select an existing Azure website that's already hosted on the cloud:



As we are going to create a new website on the cloud, with a new URL, let's select the **Create New** radio option and click on the **Publish** button.

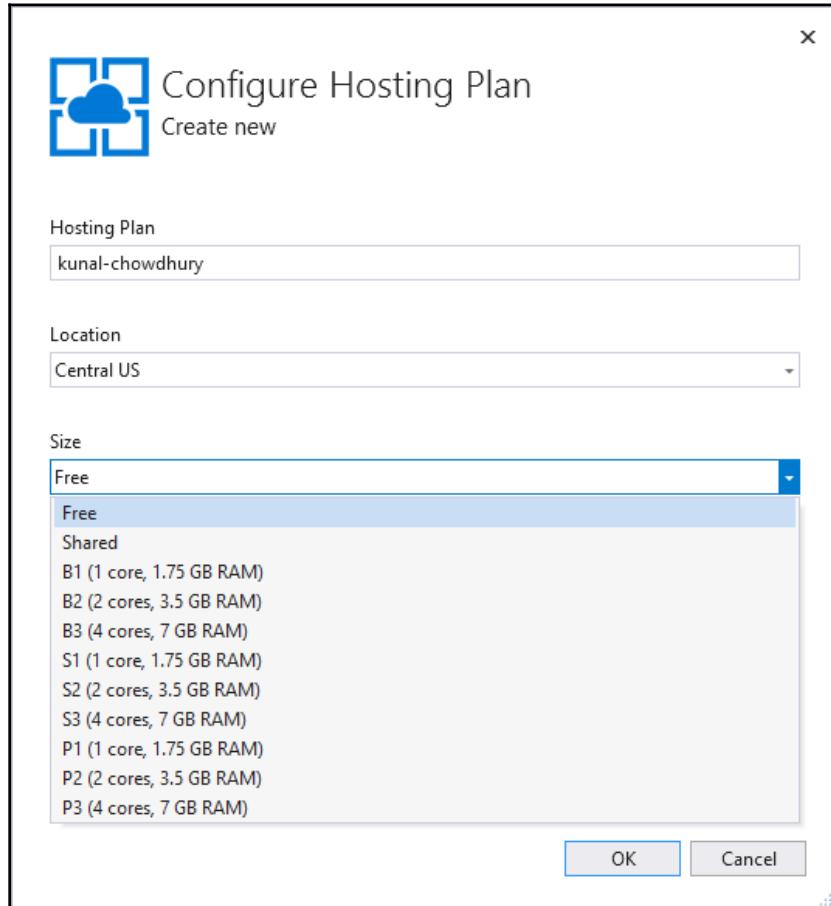
Now, a dialog window titled **Azure App Service Create new** will pop up. On this screen, you need to first log in to your Azure account if you are not already logged in.

Provide a globally unique name for your web app, which will be used to create the website's URL. Select the subscription type that you want to use to host this site:



You need to select the **Resource Group** from the list. If no resource group is available for you to use or you want to host it from a new one, click on the **New...** button next to the drop-down arrow and enter a name.

Now select the **Hosting Plan** from the list. If there are no hosting plans available or you want to create a new plan, click on the **New...** button next to the **Hosting Plan** drop-down arrow. A new window will pop up that enables you to configure a new service plan:

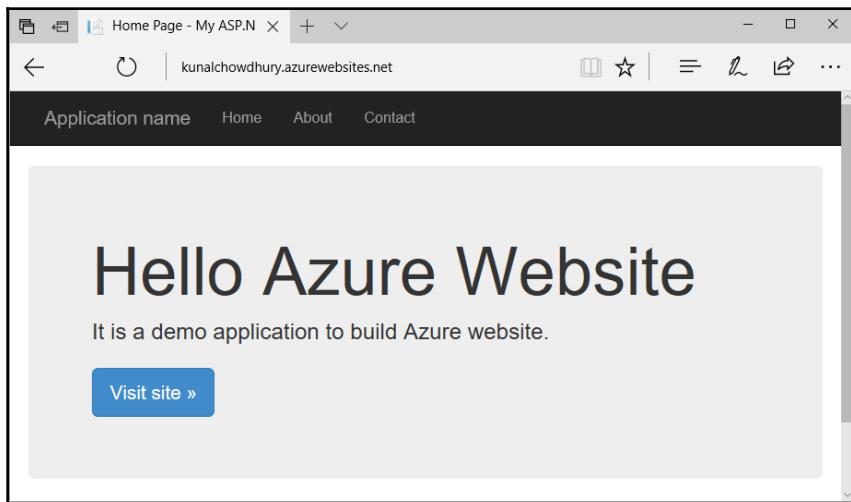


On this screen, first give your hosting plan a name so that you can easily identify it later. Select the **Location** at which you want to host it. Select a proper location to optimize its usages. Lastly, select the size of the processing unit that you want to configure for this website. Select **Free** if you want it for testing/learning purposes and don't want to cut your pocket for the usages.



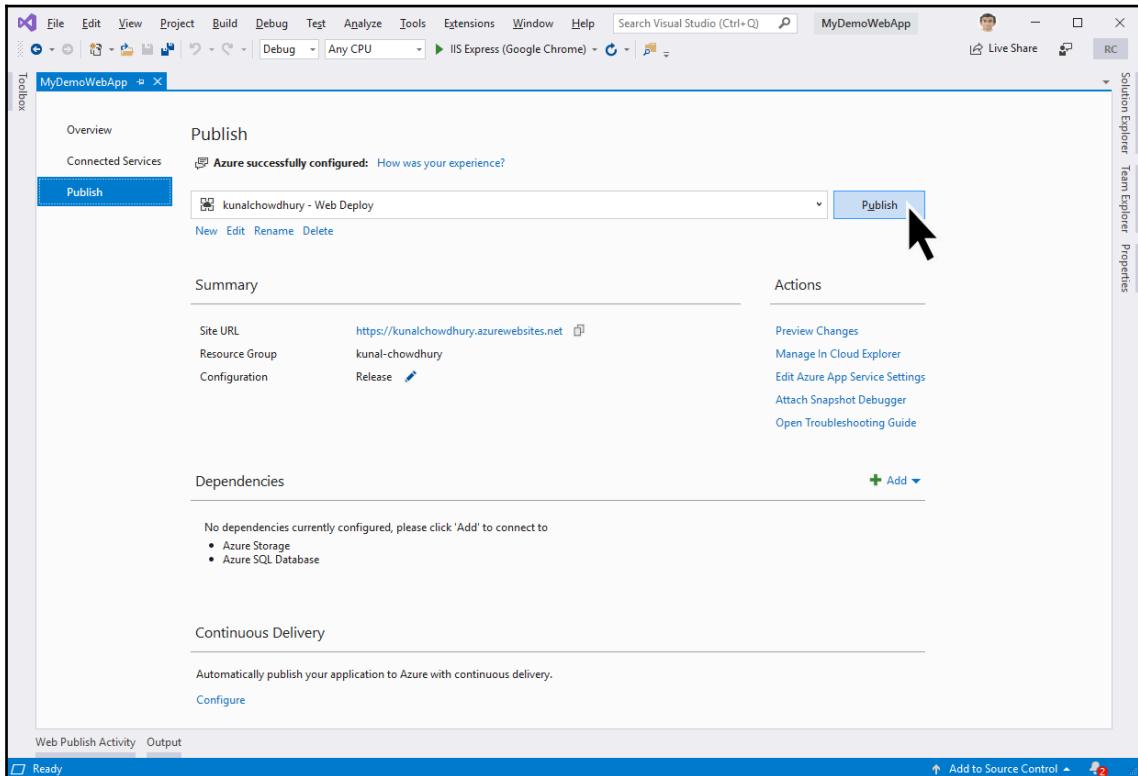
Remember that the higher the processing unit or RAM, the better the performance, but it will cost you more money because it requires more computing power.

Once you are done configuring your App Service plan, click on the **Create** button in your App Service window. This will start building your project using the **Release** configuration and start deploying it to Azure by creating a new website based on the name that you supplied. When the publishing is complete, it will open the browser at the URL that you just deployed:



Here, you can see that the URL (`kunalchowdhury.azurewebsites.net`) that it navigated to has the same name that we provided as the app name when publishing the web app to the cloud. All done! Your web app has been deployed to Azure and is running.

When you update your web application project and want to republish the updated content to the web, right-click on the project again and click on **Publish....** As the publishing configuration has been already created for this project, it will open the following configuration screen. You may want to change something here. Once you are ready, click on the **Publish** button:

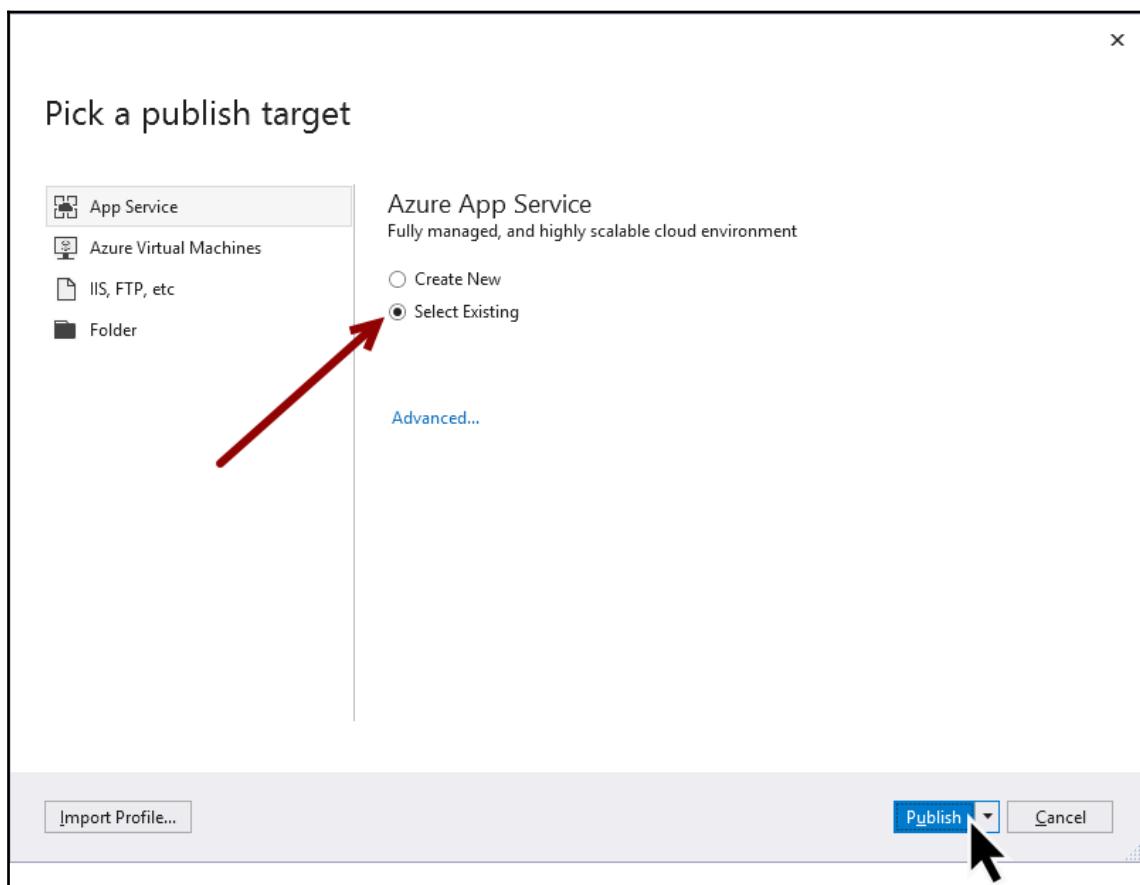


It will build again, and if there are no errors during compilation, it will be published to the cloud account and will open the browser at the same URL that the app is hosted on.

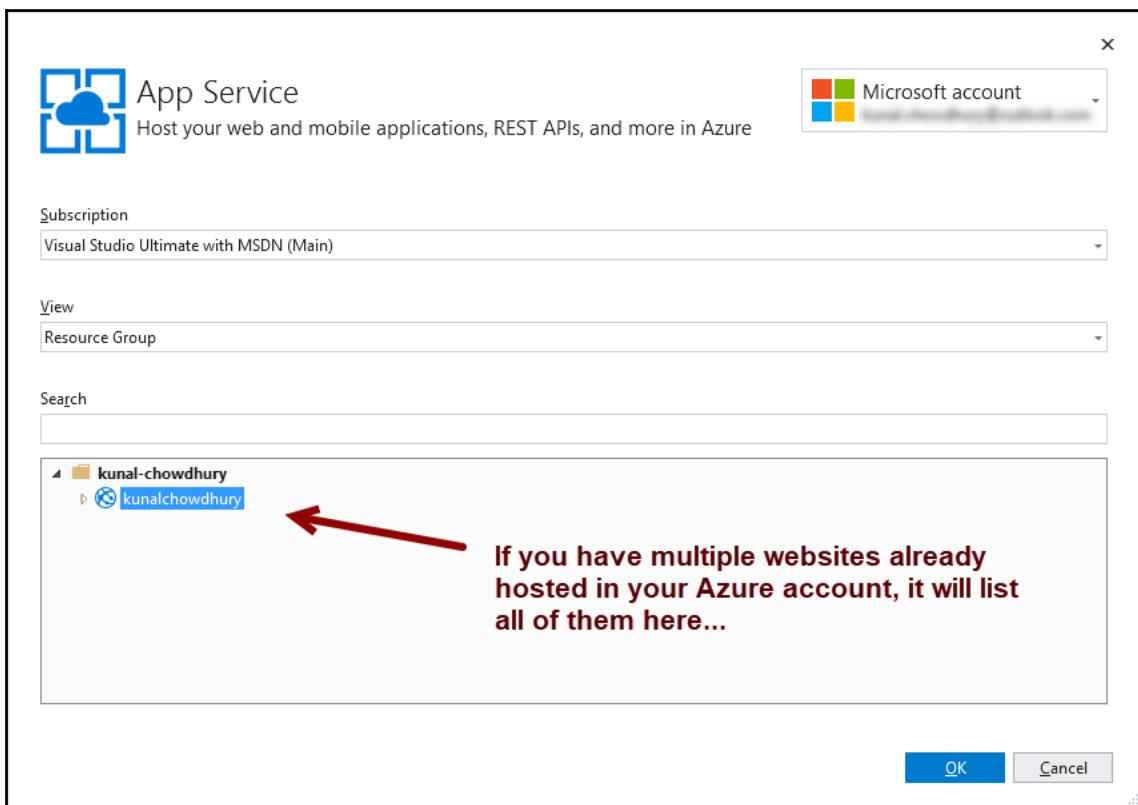
Updating an existing Azure website with Visual Studio

There may be various occasions when you have a website that's already running on Azure and you want to update it using Visual Studio, but the publishing configuration for the project is missing. In that case, follow these simple steps:

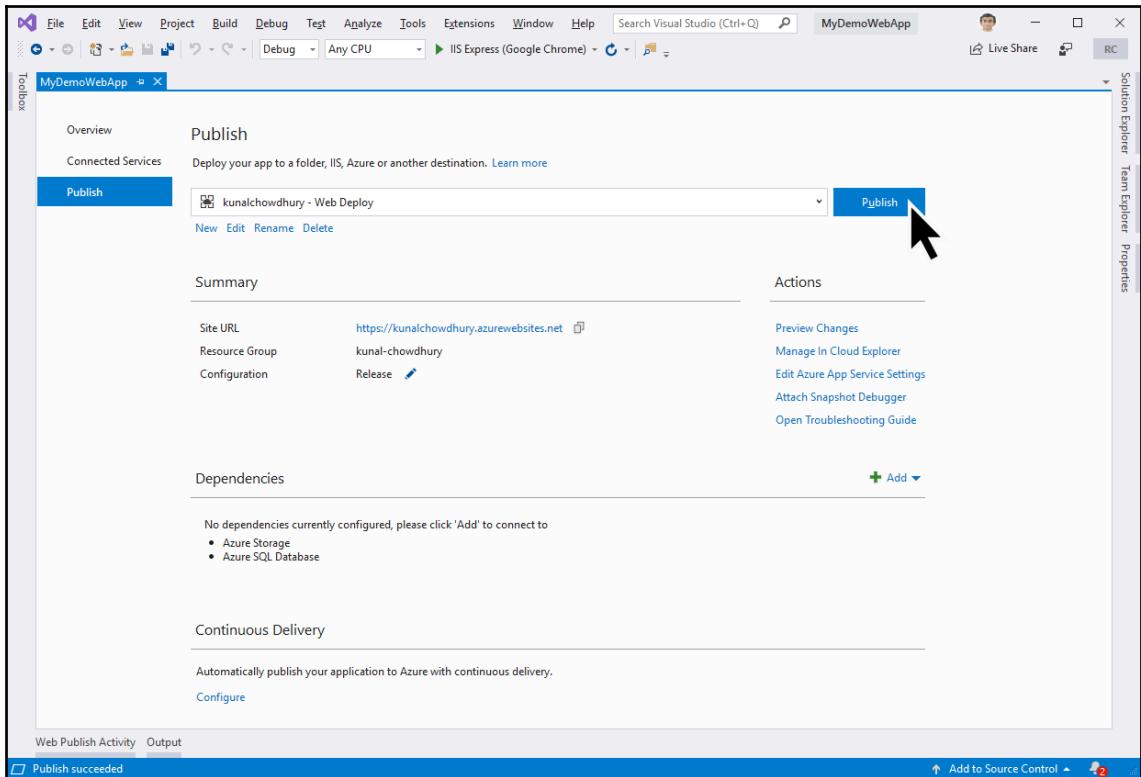
1. When you are ready with your changes inside your Visual Studio project, right-click on it and click **Publish...** from the context menu. The following publishing wizard will open on the screen:



2. Inside the wizard, navigate to the **Publish** tab and select the **App Service** category button, as shown in the preceding screenshot.
3. Now, instead of selecting the **Create New** radio button, select the one labeled **Select Existing** and click **Publish**.
4. The following **App Service** window will be launched. Here, you can select the already-running website on which to publish your current project. If you are already logged in to your Azure account, the subscription type will be populated automatically. Select the subscription that you want to use for this web app and select the **Resource Group** from the list. As shown in the following screenshot, another tree view list will be populated automatically based on the apps already hosted on your Azure account. If there are no sites running, you need to go back and start from the beginning to create a new App Service. Otherwise, select where you want to host it and click **OK**:



5. Next, Visual Studio will show you the following screen so that you can review the configuration details. When you are ready, just click on the **Publish** button to let it compile and publish it to the website that you have selected:

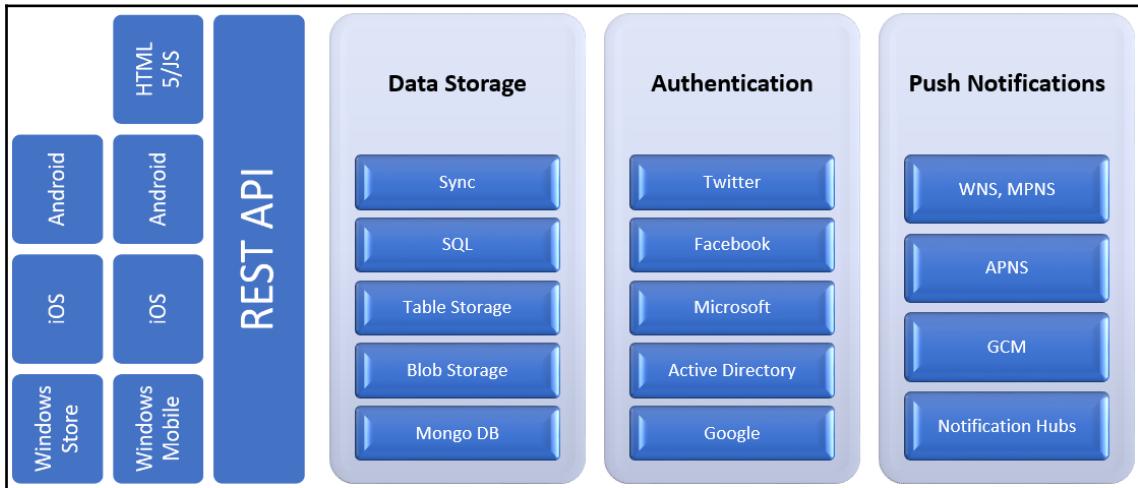


Once the publishing wizard deploys to the cloud, it will open the browser window and navigate to the URL where the application was deployed.

Building a mobile app

Azure Mobile App Service is another type of Azure App Service that runs as a PaaS and runs on any platform or device, offering high scalability and global availability for its enterprise-ready mobile application development platform. Using this, you can create a rich set of functionalities in your mobile applications.

Using Azure Mobile App Service, you can build native and cross-platform apps targeting Windows, iOS, and Android. You can also connect to your enterprise system or social networking sites in minutes, build offline-ready applications with data synchronization, and leverage the push notification services to engage your customers. Here's an App Service ecosystem:

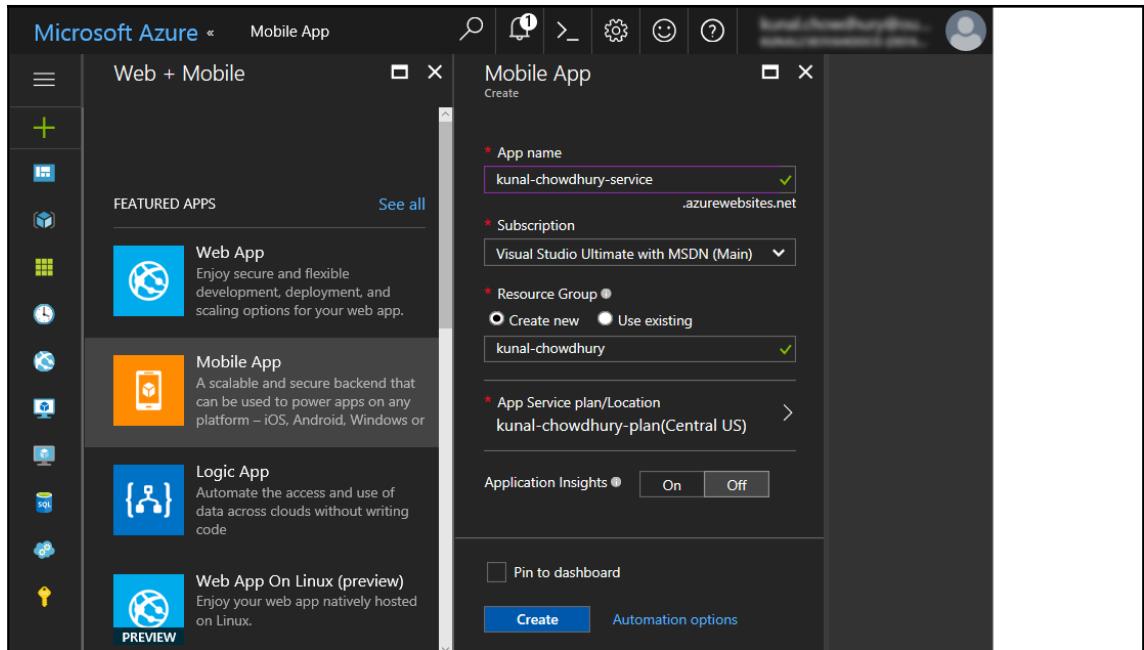


Let's create an Azure mobile app and connect it to a SQL database, which will be used later when demonstrating integration with a Windows application.

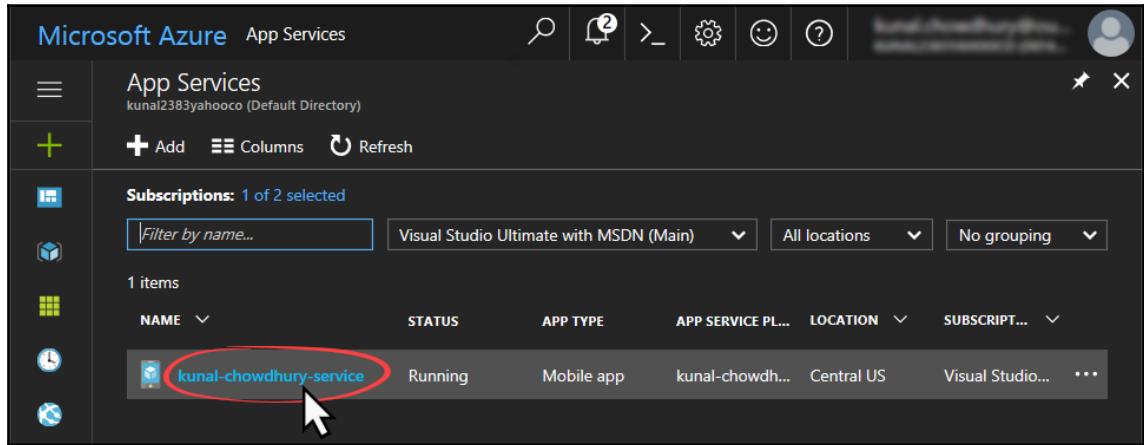
Creating an Azure mobile app

To get started, you need to create an Azure mobile app. Follow these steps:

1. Log in to your Microsoft Azure portal account and click on the + or the + New button.
2. In the wizard, select **Mobile** and then **Mobile App**. This will open a new screen where you can enter the details of your mobile app.
3. Give the app a name, which should be globally unique because it's going to create a subdomain at `azurewebsites.net` with the name of the app.
4. Select the appropriate **Subscription**, create/select the **Resource Group**, and select the appropriate **App Service plan/Location**.
5. When you are ready, click on the **Create** button to start creating the Azure mobile app. This will now provision an Azure Mobile App backend that you can use in your mobile client applications. It will take some time to complete this operation:



6. Once it is provisioned, navigate to the **App Services** screen and click on the app that you have just created:

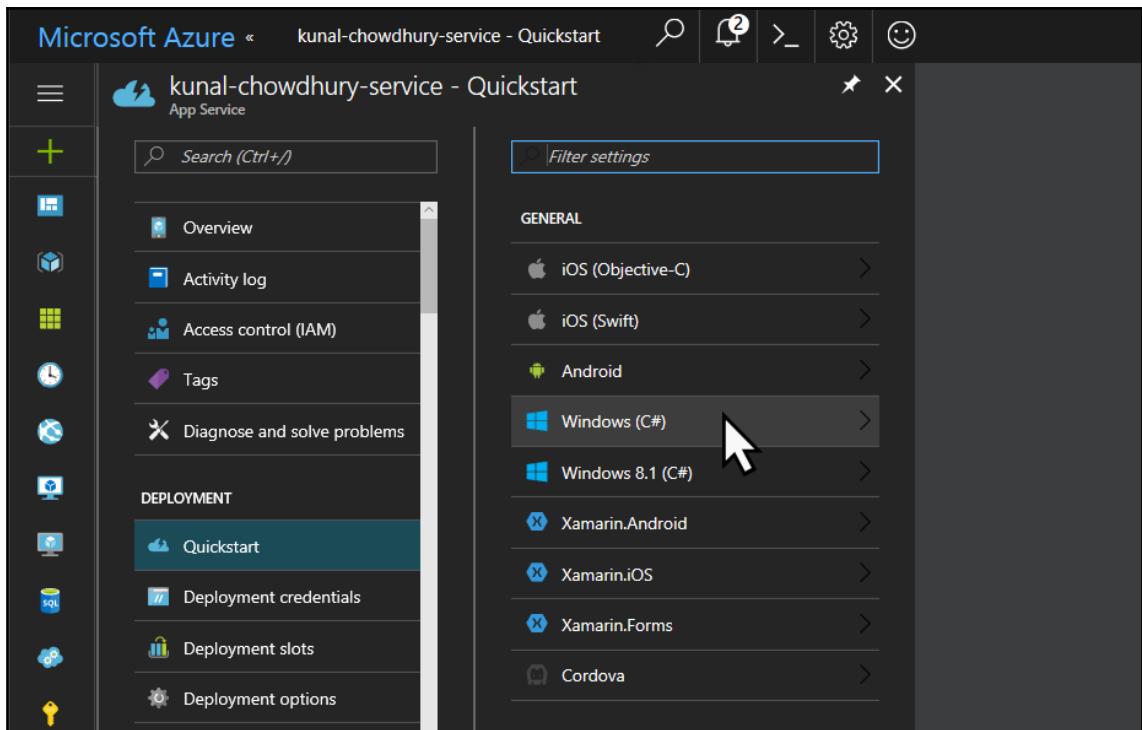


This will open the overview page of **App Services**, where you can see the details of the app. On this screen, you can also stop, restart, or delete the app when you need to.

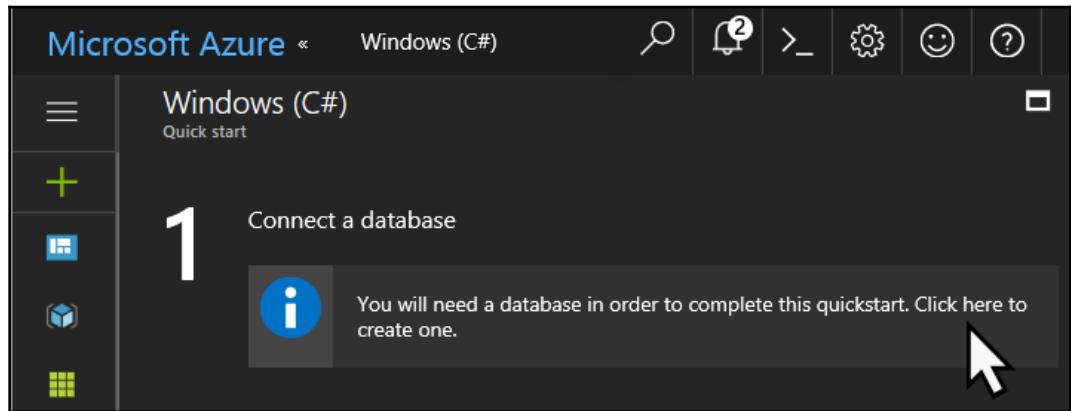
Preparing your Azure mobile app for data connectivity

You may want your Azure mobile app to connect to a SQL database that's hosted on Azure. To do this, you need to create the connectivity of the database, and you may need to create a database server too if one has not already been created:

1. On the service **Overview** page, click on **Quickstart** and then select the project template that you want to integrate with.
2. You can select **Android**, **Windows (C#)**, **Xamarin.iOS**, **Xamarin.Android**, **Xamarin.Forms**, **Cordova** or any other language listed in this screen:



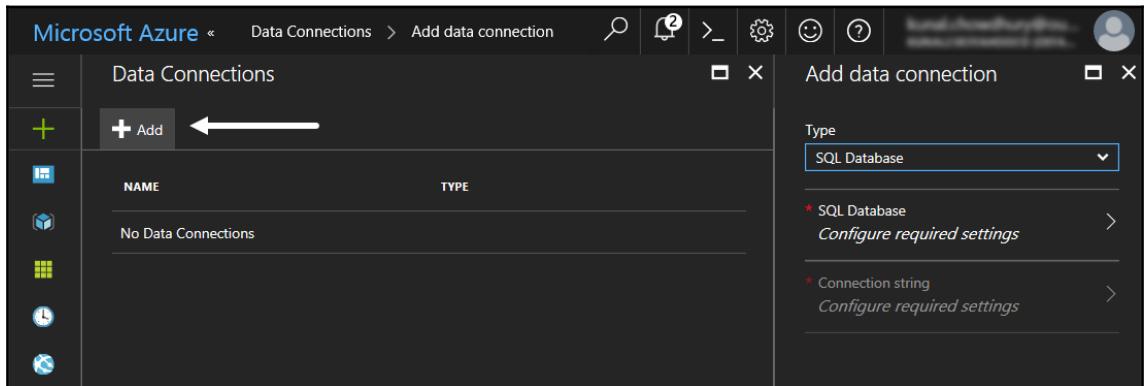
Assuming that you don't have a SQL database on Azure, you will see the following screen:



Click on the message to start creating your SQL database and then connect it with your mobile app service.

Adding a SQL data connection

The following screen will pop up, asking you to create a data connection. Click on the **Add** button to add a new connection. As shown in the following screenshot, select **SQL Database** as the connection type and then click on the next item to start configuring the database settings:

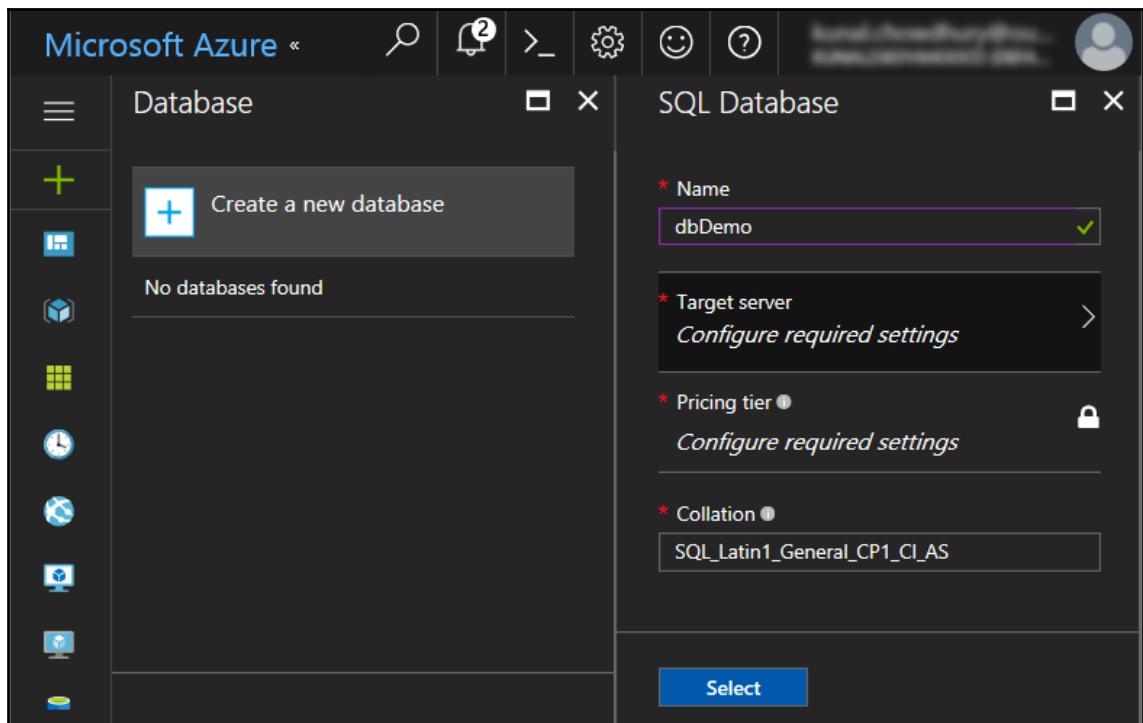


If you don't have a SQL database hosted on Azure, you need to create one. If you don't have a database server, you need to create one of these too. Finally, you need to generate the connection string.

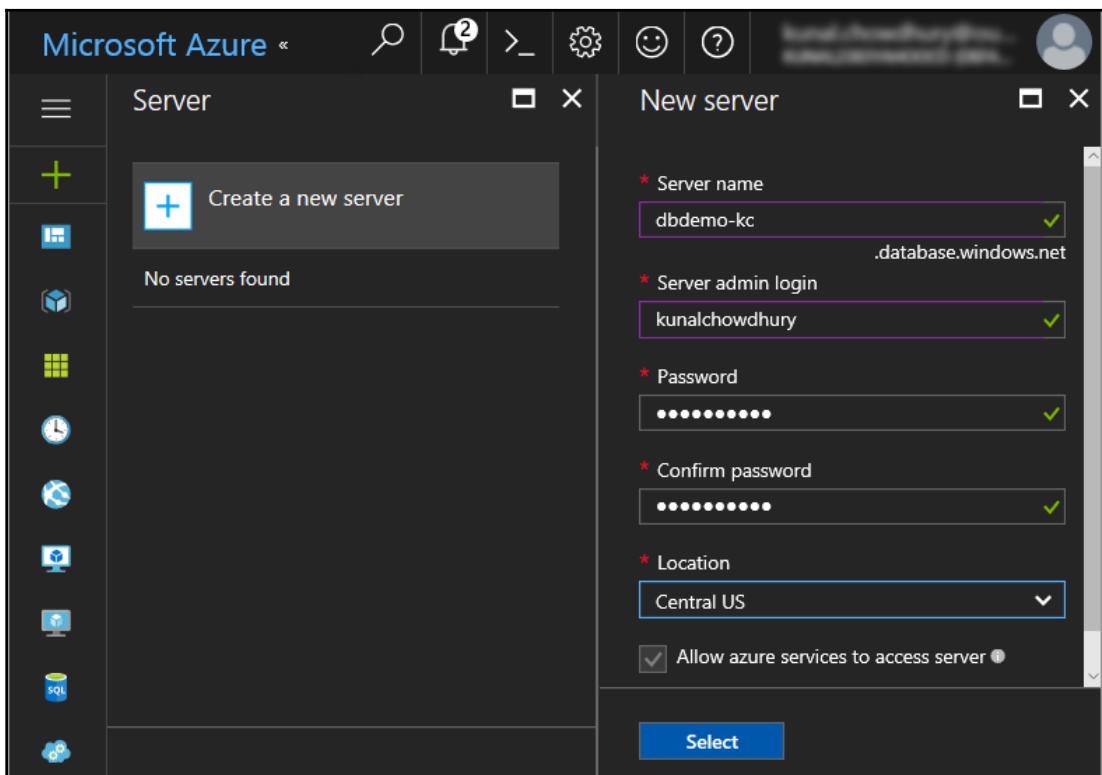
Creating a SQL database

Azure portal provides a straightforward way to create a SQL database:

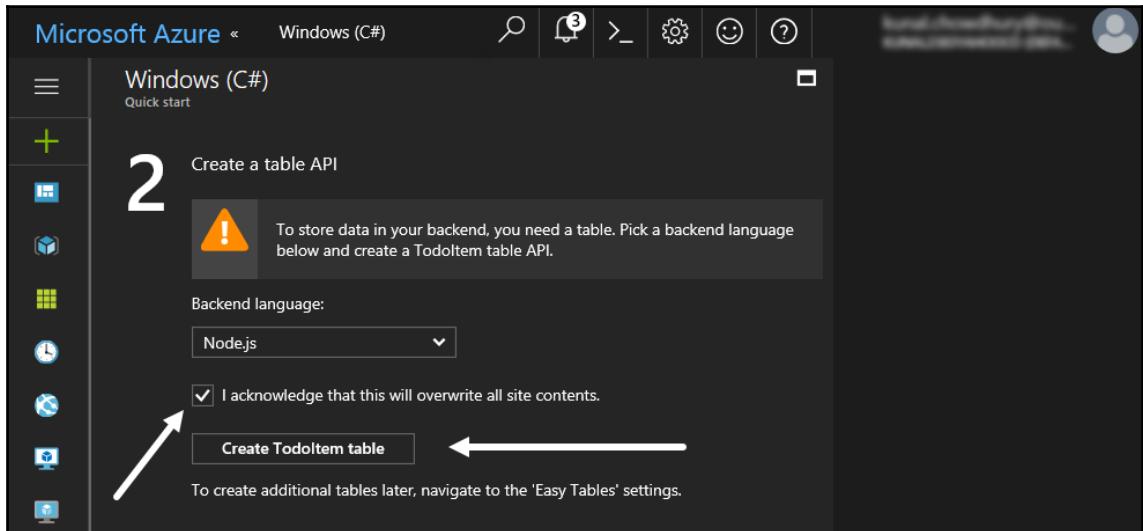
1. As shown in the following screenshot, click on the **Create a new database** button, which will open another screen that asks you to enter the database name, target server, pricing tier, and more:



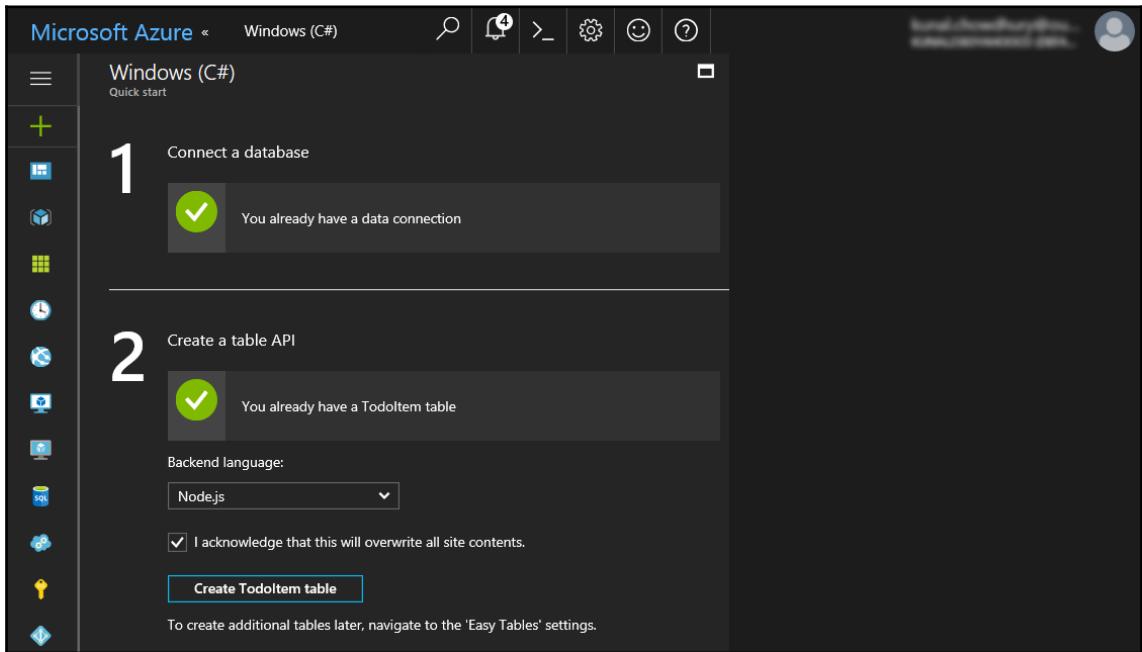
2. Clicking on the **Target server** will navigate you to a different screen, which will ask you to create a new database server if one has not been created.
3. As shown in the next screenshot, click on the **Create a new server** button to open the **New server** form, where you need to enter the details of the database server.
4. Enter the **Server name**, which must be globally unique because it will create the database server in the `database.windows.net` domain.
5. Enter data in the other fields, such as **Server admin login**, **Password**, and **Location**, and click on the **Select** button to continue. This will create the server and provision it so that you can access the hosted database server:



6. The next screen will ask you to create a table API. You will need it to store your data to the backend. Pick the backend language from the list. You can either select **Node.js** or **C#**. Let's select **Node.js** here.
7. Now, click the checkbox to acknowledge that the settings will overwrite all site contents. Click on **Create TodoItem table** to continue. The TodoItem table is a sample database table that the wizard will generate, but if you want to create any additional tables later, navigate to the **Easy Tables** settings:

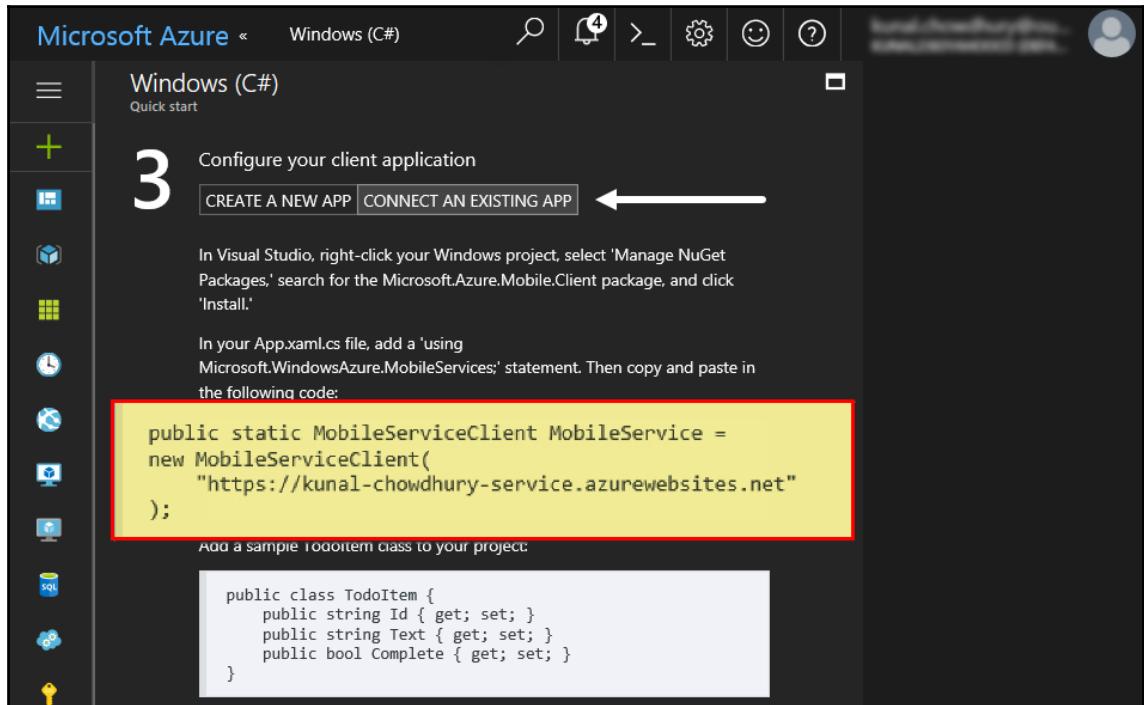


8. When it successfully generates a table inside your database, it will show you a green checkmark beside both settings, as shown in the following screenshot:



This confirms that the database has been successfully configured. Scroll down to view the third section, where it will generate the configuration settings and a sample model for you to start the integration of your application.

If you are creating the application from scratch, the **CREATE A NEW APP** tab will provide a sample Visual Studio project to get started with. If you have an existing application and/or you want to manually insert the API configurations, navigate to the **CONNECT AN EXISTING APP** tab, as shown in the following screenshot:



Note the highlighted section, which we will later need to create `MobileServiceClient` to integrate the mobile service in our project. The `TodoItem` model has also been generated for you to easily start with the development for the first time.

Integrating the mobile app service in a Windows application

Let's get started with integrating the mobile app service that we just created into an application. You can use any application, but here we will use a WPF application. Open your Visual Studio IDE and navigate to **File** | **New** | **Project...** to create a WPF project.

Creating the model and service client

Once the project has been created, you need to create the `TodoItem` model in the project:

1. Right-click on the project, navigate to **Add** | **Class...**, and call it `TodoItem`.
2. Now copy the definition of the class from the Azure portal and replace it in the code file. Here is the code for reference:

```
public class TodoItem
{
    public string Id { get; set; }

    public string Text { get; set; }

    public bool Complete { get; set; }
}
```

3. Now we need to create the instance of the mobile service client, so that you can interact with Azure.
4. Open the `App.xaml.cs` file and enter the following lines of code:

```
public static MobileServiceClient MobileService = new
MobileServiceClient("https://kunal-chowdhury-service.azurewebsites.net");
```

As our service client is hosted at

`https://kunal-chowdhury-service.azurewebsites.net`, we have provided it as the endpoint address to the service client object. If you have hosted it at a different endpoint, you need to enter this endpoint instead. Please update the entry accordingly, as highlighted earlier.

The `MobileServiceClient` class is part of the `Microsoft.Azure.Mobile.Client` DLL, which you need to reference in your project. You can get it from NuGet by clicking on the lightbulb tooltip and following the context menu, as shown in the following screenshot:



This will install the latest version of the DLLs and automatically add them as a reference in your current project. Alternatively, you can get them from <https://www.nuget.org/packages/Microsoft.Azure.Mobile.Client/> using the NuGet Package Manager. Once they are downloaded, build the solution to confirm that there are no compiler errors.

Once your service client instance has been created, you can call its API methods to do CRUD operations on the Azure database that you have created.

Integrating the API call

Let's first design our app UI so it has a `TextBox` control where you can enter the description of a to do item, a checkbox to mark it as complete, and some buttons to perform add, delete, and refresh data functionalities.

Open the `MainPage.xaml` file. This is where we can edit the UI. Replace the default `Grid` panel with the following XAML code snippet:

```
<Grid>
    <StackPanel Width="270" Margin="10">
        <TextBlock Text="Task description:"/>
        <TextBox x:Name="txbTaskDescription" Height="26" />
        <CheckBox x:Name="chkComplete" Content="Complete?" Margin="0 10"/>
        <StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
            <Button Content="Save" Width="100"
```

```
        Margin="10" Click="OnSaveButtonClicked"/>
    <Button Content="Refresh" Width="100"
        Margin="10" Click="OnRefreshButtonClicked"/>
</StackPanel>
<ListBox x:Name="lstDetails" Height="100">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <StackPanel Orientation="Horizontal">
                <CheckBox IsChecked="{Binding Complete}"/>
                <TextBlock Text="{Binding Text}"
                    Margin="10 0"/>
            </StackPanel>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
<StackPanel Orientation="Horizontal"
    HorizontalAlignment="Center">
    <Button Content="Delete" Width="80"
        Margin="5" Click="OnDeleteButtonClicked"/>
</StackPanel>
</StackPanel>
</Grid>
```

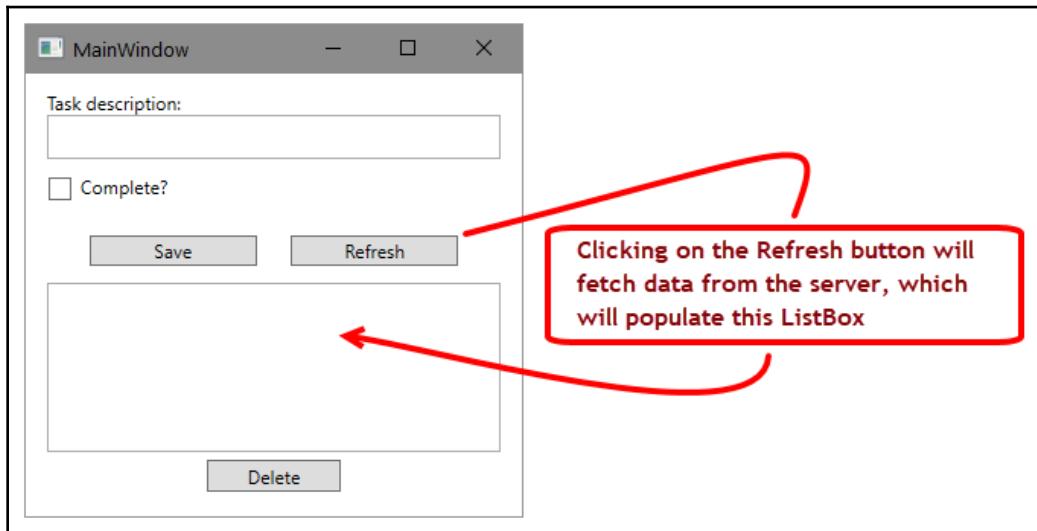
In the code-behind file of `MainPage`, that is, the `MainPage.xaml.cs` file, define the button click events that we associated in the XAML page. Mark all of them with the `async` keyword. This will now look like the following code snippet:

```
private async void OnSaveButtonClicked(object sender,
                                         RoutedEventArgs e)
{
}

private async void OnRefreshButtonClicked(object sender,
                                         RoutedEventArgs e)
{
}

private async void OnDeleteButtonClicked(object sender,
                                         RoutedEventArgs e)
{
}
```

Now, build the solution to check for any errors and correct them if you encounter anything. Once this is done, run the application, which will look like the following screenshot:



Now it's time to integrate the mobile service API to perform CRUD operations on the Azure database from your application. The `GetTable` method of the service client object returns the handle of the table where you want to perform database operations as `IMobileServiceTable`. Generally, it takes the table name as an argument. Alternatively, you can call strongly typed data operations by the type of the instance of the table:

```
App.MobileService.GetTable("TodoItem"); // untyped data operation  
App.MobileService.GetTable<TodoItem>(); // typed data operation
```

To fetch the contents of the remote table, we need to call the `ReadAsync()` method on top of the mobile service table instance and set the response as the `ItemsSource` method of the list box that we have added in the UI. Here is the code for reference:

```
lstDetails.ItemsSource = await  
App.MobileService.GetTable<TodoItem>().ReadAsync();
```

Similarly, when you want to insert an item to the remote database table, you should call the `InsertAsync` method and pass the instance of the model, as follows:

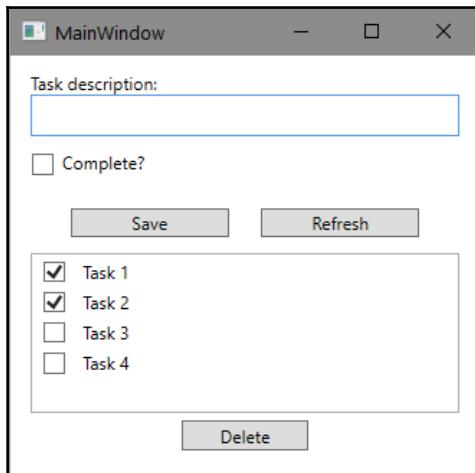
```
await App.MobileService.GetTable<TodoItem>().InsertAsync(todoItem);
```

For delete operations, call the `DeleteAsync` method by passing the instance of the model. Here's how to call it:

```
await App.MobileService.GetTable<TodoItem>().DeleteAsync(item);
```

Additionally, you may want to add validations while performing add or delete operations. When you compile and run the application, enter the task description and optionally select the **Complete?** checkbox before clicking on the **Save** button.

Once the save operation completes, or when you click on the **Refresh** button, the API call will retrieve the latest data available on the Azure database. If you want to delete a record, select the desired data in the list box and click on the **Delete** button:



Here is the complete code snippet of the code-behind file, for reference:

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
        OnRefreshButtonClicked(this, new RoutedEventArgs());
    }

    private async void OnSaveButtonClicked(object sender,
                                         RoutedEventArgs e)
    {
        var taskDescription = txbTaskDescription.Text;

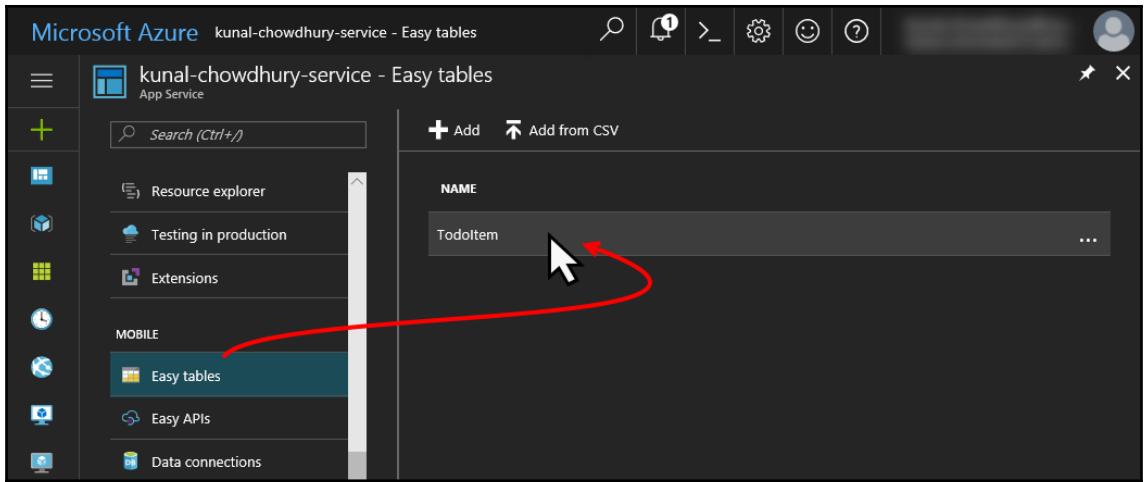
        if (!string.IsNullOrWhiteSpace(taskDescription))

```

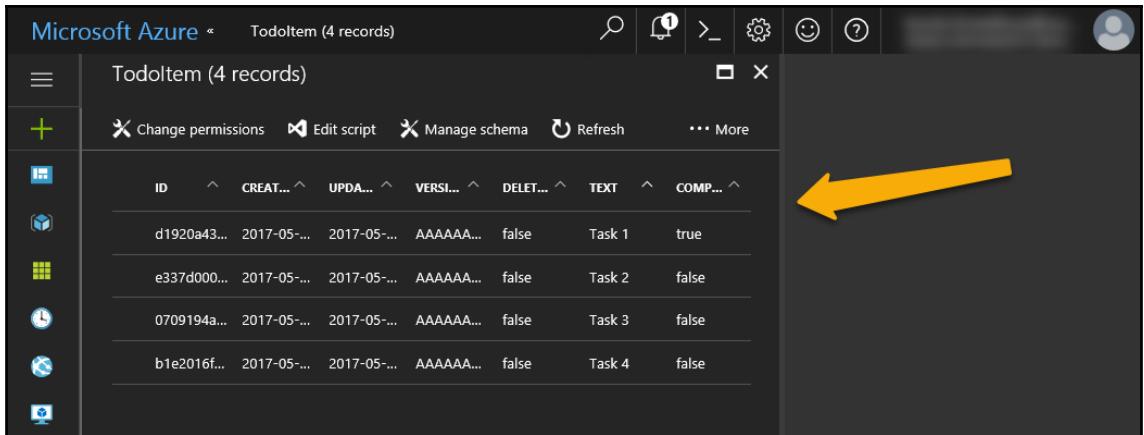
```
{  
    var isComplete = chkComplete.IsChecked == true;  
    var todoItem = new TodoItem { Text = taskDescription,  
                                Complete = isComplete };  
  
    txbTaskDescription.Text = string.Empty;  
    chkComplete.IsChecked = false;  
  
    await App.MobileService.GetTable<TodoItem>()  
        .InsertAsync(todoItem);  
    OnRefreshButtonClicked(sender, e);  
}  
}  
  
private async void OnRefreshButtonClicked(object sender,  
                                         RoutedEventArgs e)  
{  
    lstDetails.ItemsSource = await  
        App.MobileService.GetTable<TodoItem>().ReadAsync();  
}  
  
private async void onDeleteButtonClicked(object sender,  
                                         RoutedEventArgs e)  
{  
    if(lstDetails.SelectedItem is TodoItem item)  
    {  
        await App.MobileService.GetTable<TodoItem>()  
            .DeleteAsync(item);  
        OnRefreshButtonClicked(sender, e);  
    }  
}
```

You can now navigate to the Azure portal to view the data that's been entered in the table. Log in to the Azure dashboard, navigate to **App Service**, and select the mobile app service that you created.

Now, as shown in the following screenshot, scroll down the panel to find the **Easy tables** option. Click on the **TodoItem** table to examine the data that it contains. From this screen, you can also create a new table:



When you click on the table, it will open another screen, where you can see the records that the table has. On this screen, you can also change the permissions to access the table, edit the script, and manage the schema of the table:

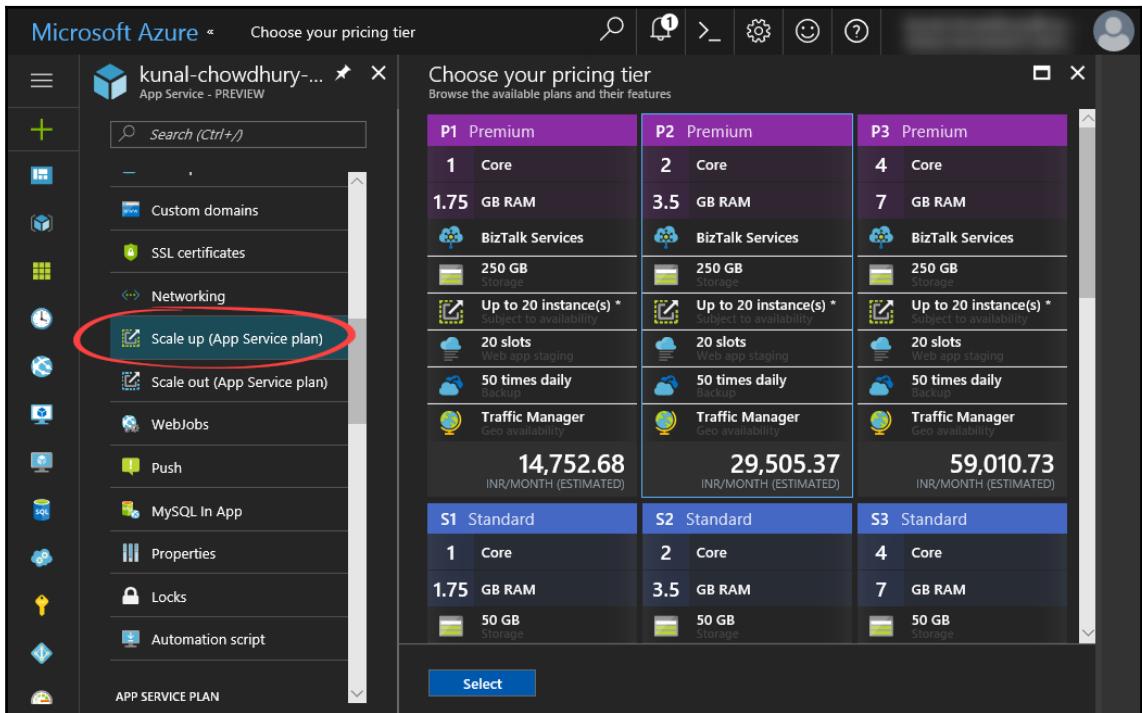


In this section, we have learned how to integrate a mobile app service into a Windows application. If you want to integrate it into an ASP.NET or **Universal Windows Platform (UWP)** application, the procedure is almost the same.

Scaling an App Service plan

The portal allows you to easily scale up or scale down the App Service plan that you are using. It also allows you to automate the process to allow you to scale the service plan on demand.

First, log in to the Azure portal and navigate to the App Service that you want to scale. In the left-hand panel, scroll down to find **Scale up (App Service plan)**. When you click on it, a new screen will appear that will allow you to select the pricing tier for your application:

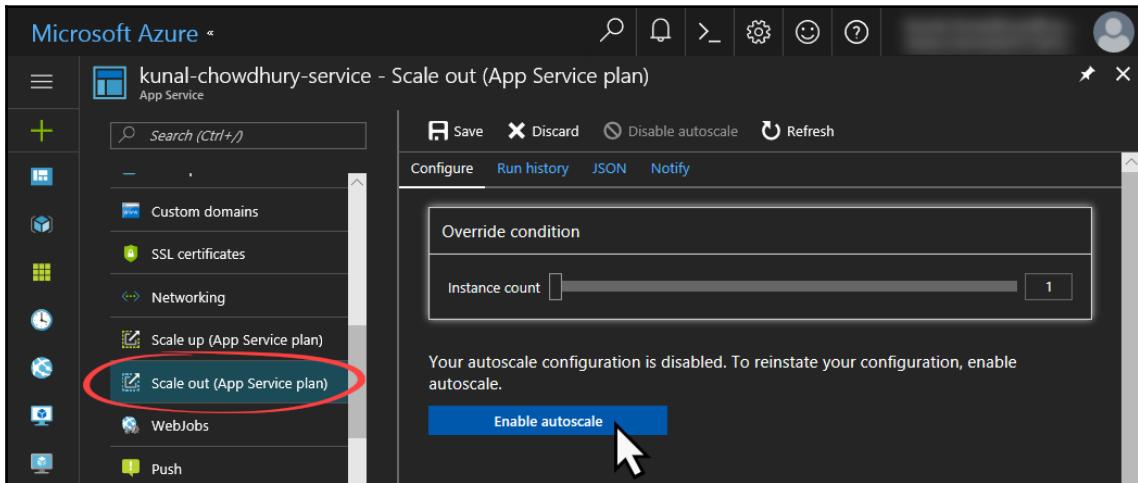


The screenshot shows the Microsoft Azure portal interface. On the left, there's a sidebar with various icons and a search bar. In the main area, a modal window titled "Choose your pricing tier" is open. This window displays different pricing tiers with their features and estimated monthly costs. The "P1 Premium" tier is selected, showing 1 Core, 1.75 GB RAM, BizTalk Services, 250 GB Storage, Up to 20 instance(s) * Subject to availability, 20 slots Web app staging, 50 times daily Backup, and Traffic Manager Geo availability. The cost is listed as 14,752.68 INR/MONTH (ESTIMATED). Other tiers shown include P2 Premium (2 Core, 3.5 GB RAM, BizTalk Services, 250 GB Storage, Up to 20 instance(s) * Subject to availability, 20 slots Web app staging, 50 times daily Backup, and Traffic Manager Geo availability; cost 29,505.37 INR/MONTH (ESTIMATED)), P3 Premium (4 Core, 7 GB RAM, BizTalk Services, 250 GB Storage, Up to 20 instance(s) * Subject to availability, 20 slots Web app staging, 50 times daily Backup, and Traffic Manager Geo availability; cost 59,010.73 INR/MONTH (ESTIMATED)), S1 Standard (1 Core, 1.75 GB RAM, 50 GB Storage), S2 Standard (2 Core, 3.5 GB RAM, 50 GB Storage), and S3 Standard (4 Core, 7 GB RAM, 50 GB Storage). A "Select" button is at the bottom of the dialog.

P1 Premium	P2 Premium	P3 Premium
1 Core	2 Core	4 Core
1.75 GB RAM	3.5 GB RAM	7 GB RAM
BizTalk Services	BizTalk Services	BizTalk Services
250 GB Storage	250 GB Storage	250 GB Storage
Up to 20 instance(s) * Subject to availability	Up to 20 instance(s) * Subject to availability	Up to 20 instance(s) * Subject to availability
20 slots Web app staging	20 slots Web app staging	20 slots Web app staging
50 times daily Backup	50 times daily Backup	50 times daily Backup
Traffic Manager Geo availability	Traffic Manager Geo availability	Traffic Manager Geo availability
14,752.68 INR/MONTH (ESTIMATED)	29,505.37 INR/MONTH (ESTIMATED)	59,010.73 INR/MONTH (ESTIMATED)

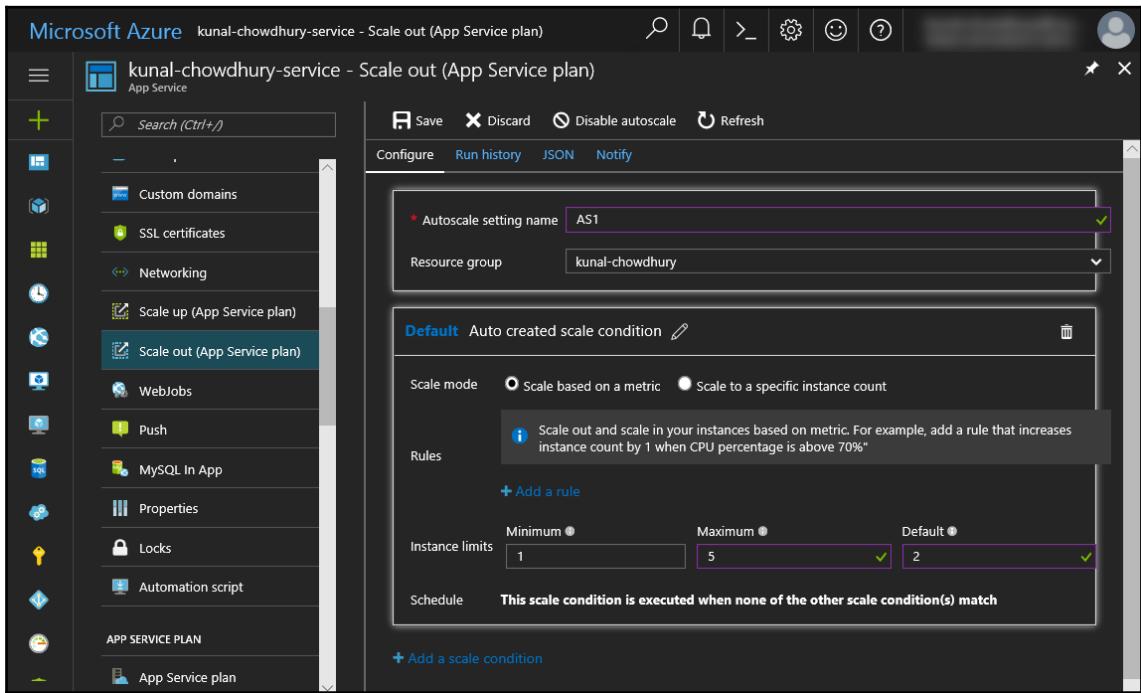
There are plenty of service plans available for you to select. The basic tier will have less performance than the standard or the premium tiers, but it will allow you to save money. The premium tier will charge you more money but will give you the best performance.

Scale out (App Service plan) allows you to override the instance count. It also allows you to enable/disable the autoscaling option. If you want to autoscale the service based on some conditions/rules, click on **Enable autoscale**:



This will open another screen where you can create a condition and ask the service plan to scale based on that condition. You can select it either based on a metric or based on a specific instance count. You can add more rules and conditions as per your requirements. Finally, click on the **Save** button to save all the changes made on this screen.

When you don't want to run the autoscaling option, come to this screen again and click on the button labeled **Disable autoscale**:



The screenshot shows the Microsoft Azure portal interface for managing an App Service plan named 'kunal-chowdhury-service - Scale out (App Service plan)'. The left sidebar lists various service configurations like Custom domains, SSL certificates, Networking, and App Service Plan settings. The main panel is titled 'Configure' and displays the 'Default' scale condition. Key settings include:

- Autoscale setting name:** AS1
- Resource group:** kunal-chowdhury
- Scale mode:** Scale based on a metric (selected)
- Rules:** A note states: "Scale out and scale in your instances based on metric. For example, add a rule that increases instance count by 1 when CPU percentage is above 70%".
- Instance limits:** Minimum 1, Maximum 5, Default 2
- Schedule:** A note says: "This scale condition is executed when none of the other scale condition(s) match".

A prominent blue button at the top right of the configuration panel is labeled 'Disable autoscale'.

Doing this will disable the autoscaling feature of the app service. If you want to enable it again in future, you will be able to re-enable it by following the same steps discussed previously.

Summary

In this chapter, we learned about the basics of cloud computing, including IaaS, PaaS, and SaaS. We learned how to create a free Azure account and how to configure Visual Studio 2019 for Azure development. Then, we discussed creating and managing Azure websites from the portal, as well as from Visual Studio. We also discussed how to update an existing website from Visual Studio.

Later, we discussed Azure Mobile App Service and created a mobile app service, configured an Azure database, and integrated the service into a Windows application. Finally, we learned about the service API calls and scaled the App Service plan using the various pricing tiers.

In the next chapter, we are going to learn about web application development using .NET Core, and we'll start building applications targeting this new .NET framework.

4

Building Applications with .NET Core

.NET Core is an open source framework (hosted on GitHub at <https://github.com/dotnet/core>), released by Microsoft and maintained by the .NET community, to build cross-platform applications for Windows, Linux, and macOS. You can get it from Microsoft's official .NET Core site (<https://www.microsoft.com/net/core>) or GitHub. If you are using Visual Studio 2019, then you can get it as part of the installer.

As Microsoft is investing more on .NET Core and announced that they are going to merge .NET Framework in .NET Core 5, it's now becoming a core framework for the Microsoft application development platform. This helps our learning of the basics of .NET Core, so we can start building cross-platform applications.

In this chapter, we will discuss the following core topics:

- Overview of .NET Core
- Installing .NET Core with Visual Studio 2019
- A quick lap around .NET Core commands
- Creating .NET Core applications using Visual Studio 2019
- Publishing .NET Core applications using Visual Studio 2019
- Creating, building, and publishing .NET Core web apps to Microsoft Azure

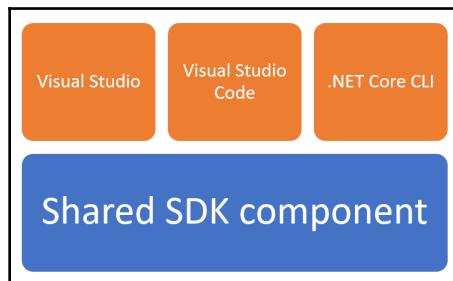
Technical requirements

To follow along with this chapter, you should have a basic knowledge of C# and Visual Studio. Installation of Visual Studio 2019 is mandatory with the **.NET Core cross-platform development** workload.

You will also need to have some basic knowledge of Microsoft Azure to deploy an ASP.NET web application to Azure App Service. You can obtain a free Azure account by visiting <https://azure.microsoft.com/en-us/free/>.

Overview of .NET Core

.NET Core version 1.0 was first released in 2016, and now, it has support to work in Visual Studio 2015 Update 3.3 or later, Visual Studio 2017, Visual Studio 2019, and Visual Studio Code only. The shared SDK component can be used to build, run, and publish applications using Visual Studio, Visual Studio Code, and the .NET **command-line interface (CLI)**:



Please note that the latest version of .NET Core is 2.2, which was released in December, 2018. .NET Core 3, which was announced at the Microsoft Build conference, is scheduled to be released in 2019.

.NET Core is a subset of .NET Framework and contains the core features from it in both runtime and libraries. .NET Framework, which was first released in 2002, now runs version 4.8.

.NET Framework was only targeted for Windows platforms, but, as time passed, there was a need to target it on different platforms. Using .NET Core components, you can build apps targeting the following platforms:

- Windows platform:
 - Windows 7, 8, 8.1, and 10
 - Windows Server 2008 R2 SP1
 - Windows Server 2012 R2 SP1
 - Windows Server 2016
- macOS platform:
 - macOS 10.11 or higher
- Linux platform:
 - Red Hat Enterprise 7.2
 - Ubuntu 14.04, 16.04, and 16.10
 - Mint 17
 - Debian 8.2
 - Fedora 23 and 24
 - Cent OS 7.1
 - Oracle Linux 7.1
 - Open SUSE 13.2 and 42.1

.NET Core is composed of a .NET runtime, a set of framework libraries, a set of SDK tools, language compilers, and the .NET app host. You can use the C# and F# languages to build apps and libraries, whereas partial support for VB.NET is currently available.

.NET Core 1.0 does not support all the app models of .NET Framework except the console and ASP.NET Core app models. It contains a smaller set of APIs from .NET Framework as it is a subset of the other, but Microsoft is making changes to expand it further. Using .NET Core, you can build apps that can execute on devices, on the cloud, and on the embedded/IoT devices. Here are the characteristics that best define .NET Core:

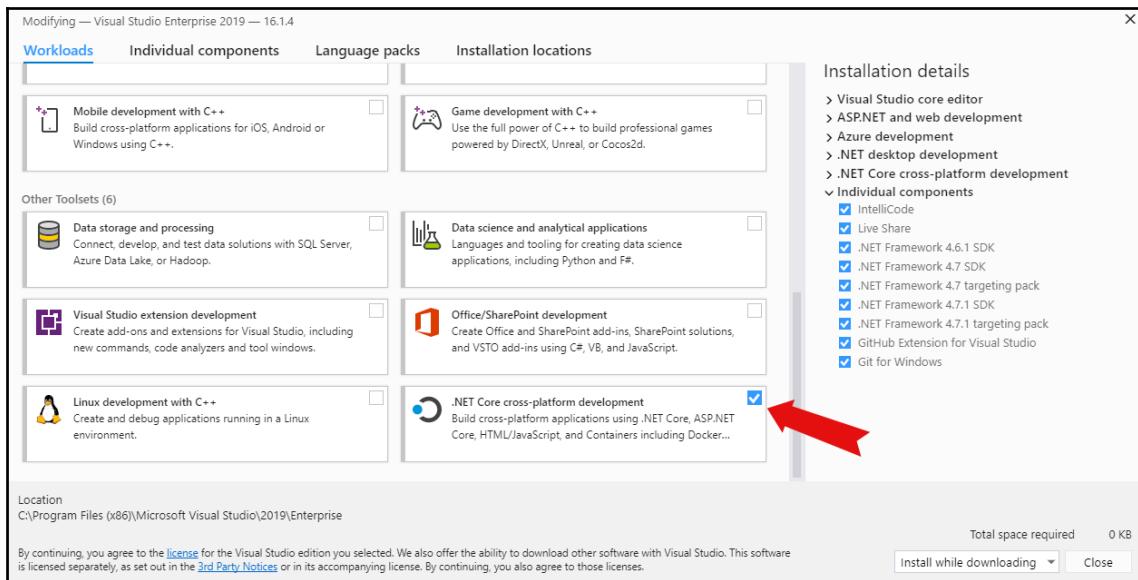
- It is an open source platform that is hosted on GitHub with the MIT and Apache 2 licenses.
- Due to its flexible deployment nature, you can include .NET Core in your app using **self-contained deployment (SCD)** or can install it side-by-side using **framework-dependent deployment (FDD)**. We are going to discuss SCD and FDD later in this chapter.

- You can build cross-platform applications, which are currently supported by a few flavors of Windows, Linux, and macOS.
- You can use a CLI to execute everything from project creation to publishing an app.
- As it is a subset of .NET Framework, it is compatible with it, along with Xamarin and Mono, via the .NET Standard library.
- Finally, it is supported by Microsoft and maintained by a vast .NET community.

Installing .NET Core with Visual Studio 2019

To install .NET Core along with Visual Studio 2019, run the Visual Studio Installer. If you have already installed Visual Studio 2019, then you can modify the existing installation to install the .NET Core workload.

When the installer starts, you will see the following screen where you can customize the workloads that you want to install. From the list, select **.NET Core cross-platform development** and click **Install** to continue:



The installer will begin the installation process and, once it completes, you may have to restart your computer for the changes to take effect.

Once the installation is successful, open a console window and type the `dotnet new` command to populate your local .NET Core package cache, which will occur only once to improve the restore speed and enable offline access.



`dotnet` is the driver of the .NET Core CLI. It accepts commands, followed by distinct options to run specific features.

When used for the first time, the command results in the following output in the console window, which may take up to a minute for the initialization to complete:

A screenshot of a Windows Command Prompt window titled 'C:\Windows\system32\cmd.exe'. The window shows the output of the command 'dotnet new'. The output includes a welcome message, telemetry information, and a progress bar for decompressing and expanding files.

```
C:\Windows\system32\cmd.exe
D:\Demo>dotnet new
Welcome to .NET Core!
-----
Learn more about .NET Core @ https://aka.ms/dotnet-docs. Use dotnet --help to see available commands or go to https://aka.ms/dotnet-cli-docs.
Telemetry
-----
The .NET Core tools collect usage data in order to improve your experience. The data is anonymous and does not include command-line arguments. The data is collected by Microsoft and shared with the community. You can opt out of telemetry by setting a DOTNET_CLI_TELEMETRY_OPTOUT environment variable to 1 using your favorite shell.
You can read more about .NET Core tools telemetry @ https://aka.ms/dotnet-cli-telemetry.
Configuring...
-----
A command is running to initially populate your local package cache, to improve restore speed and enable offline access. This command will take up to a minute to complete and will only happen once.
Decompressing 100% 5873 ms
Expanding 100% 95342 ms
```

As we have completed the installation of .NET Core, let's move on to the next point and discuss the .NET Core commands.



According to a recent announcement, Microsoft is going to merge .NET Framework and .NET Core. **The next major version will be named .NET 5 and will be released in November 2020.** A preview version will be available in early 2020.

A quick lap around .NET Core commands

The driver of the .NET Core CLI is `dotnet`. You can use this along with commands and distinct options to run the specific features that it supports. Each feature of it is implemented as a command, which you need to specify after `dotnet` in the command line.

Let's look at some .NET Core commands:

- When you use `dotnet new`, this initializes new projects based on the project template and language that you select. The default is C# and, currently, C# and F# programming languages are fully supported, whereas partial support of VB.NET is also available.
- Here is the list of project templates, their short names to pass to the command, and the language type for each template, for your reference:

Templates	Short Name	Language
Console Application	console	[C#], F#, VB
Class library	classlib	[C#], F#, VB
Unit Test Project	mstest	[C#], F#, VB
NUnit 3 Test Project	nunit	[C#], F#, VB
NUnit 3 Test Item	nunit-test	[C#], F#, VB
xUnit Test Project	xunit	[C#], F#, VB
Razor Page	page	[C#]
MVC ViewImports	viewimports	[C#]
MVC ViewStart	viewstart	[C#]
ASP.NET Core Empty	web	[C#], F#
ASP.NET Core Web App (Model-View-Controller)	mvc	[C#], F#
ASP.NET Core Web App	razor	[C#]
ASP.NET Core with Angular	angular	[C#]
ASP.NET Core with React.js	react	[C#]
ASP.NET Core with React.js and Redux	reactredux	[C#]
Razor Class Library	razorclasslib	[C#]
ASP.NET Core Web API	webapi	[C#], F#
global.json file	globaljson	
NuGet Config	nugetconfig	
Web Config	webconfig	
Solution File	sln	

- The `dotnet restore` command restores the dependencies and tools of a project.
- The `dotnet clean` command is used to clean the output of a project that was built earlier.
- The `dotnet build` command is used to build a .NET Core application.

- The `dotnet msbuild` command provides access to the `msbuild` CLI to build a project and all its dependencies.
- When you want to publish a .NET Core application as a framework-dependent or self-contained app, you should use the `dotnet publish` command.
- The `dotnet run` command is used to run the application from source.
- When you use `dotnet test`, this executes the tests using the test runner specified in the `project.json` file.
- When you want to create a NuGet package of your code, you should use the `dotnet pack` command in the CLI.
- When you specify the `dotnet sln` command, it allows you to add, remove, and list projects in a solution file.
- The `dotnet nuget delete` command deletes or unlists projects from the NuGet server.
- The `dotnet nuget locals` command is used to clear or list the local NuGet resources.
- By using the `dotnet nuget push` command, you can push a NuGet package to the server and publish it in the NuGet store.

Creating a .NET Core console app

Let's learn about how to create a .NET Core console application. Here, we will discuss how to create it from the CLI using the `dotnet new` command.

As we have already seen that the short name for the command-line console application template is `console`, we will use it to create this type of project. By default, this creates the new project in the current directory with the name of the same directory unless you specify the name.

Let's first create a console application with all default parameters/options. Open a console window and navigate to a directory of your choice. Then, enter the following command to create a console application project:

```
dotnet new console
```

In the following demonstration, we create a new folder named `Demo` in the `D:\DotNetCore` directory path, navigate to that new directory (`D:\DotNetCore\Demo`) in the console window, and then provide the `dotnet new console` command:

```
C:\Windows\System32\cmd.exe
D:\DotNetCore> dotnet new console
Content generation time: 48.0845 ms
The template "Console Application" created successfully.

Documents (D:) > DotNetCore > Demo
Name           Date modified   Type      Size
Demo.csproj    01-Mar-17 10:37 P... Visual C# Project F... 1 KB
Program.cs     01-Mar-17 10:37 P... Visual C# Source F... 1 KB
```

As we have not specified any name explicitly, it will create a project in the context of the current directory. So, you will see `Demo.csproj` in that directory, with a C# file named `Program.cs`.

Let's now create another project, but this time, we will specify a name and the language explicitly. Navigate to the `DotNetCore` directory in the console window and enter the following command:

```
dotnet new console -lang C# -n "HelloDotNetCore"
```

This will create a new folder named `HelloDotNetCore`, which contains a project that has the same name and a `Program.cs` code file in C#. If you want to build an F# project, then specify the option as `-lang F#`. This is shown in the following screenshot by specifying the language as C#:

```
C:\Windows\System32\cmd.exe
D:\DotNetCore> dotnet new console -lang C# -n "HelloDotNetCore"
Content generation time: 104.5538 ms
The template "Console Application" created successfully.

Documents (D:) > DotNetCore > HelloDotNetCore
Name           Date modified   Type      Size
HelloDotNetCore.csproj 01-Mar-17 10:46 P... Visual C# Project F... 1 KB
Program.cs     01-Mar-17 10:46 P... Visual C# Source F... 1 KB
```



Note that none of the preceding commands have created any solution file for the projects. To create a solution file from a CLI, the `dotnet new sln` command is used.

Creating a .NET Core class library

To create a .NET Core class library project, we need to use the same `dotnet new` command, but with a class library-type project template whose short name is `classlib`. Let's navigate to the folder (in our case, it's `D:\DotNetCore`) in the console window and enter the following command:

```
dotnet new classlib -n "CustomLibrary" -lang C#
```

Check the following screenshot after running the preceding command:

The screenshot shows a Windows Command Prompt window titled 'C:\Windows\System32\cmd.exe'. The command entered is `dotnet new classlib -n "CustomLibrary" -lang C#`. The output shows 'Content generation time: 119.0008 ms' and 'The template "Class library" created successfully.' Below the command line, a file explorer window is open showing the directory structure and files created. A red arrow points from the command line to the 'CustomLibrary' folder in the file explorer path. Another red arrow points from the 'CustomLibrary' folder in the path to the 'CustomLibrary.csproj' file in the file list.

Name	Date modified	Type	Size
Class1.cs	02-Mar-17 12:00 A...	Visual C# Source F...	1 KB
CustomLibrary.csproj	02-Mar-17 12:00 A...	Visual C# Project F...	1 KB

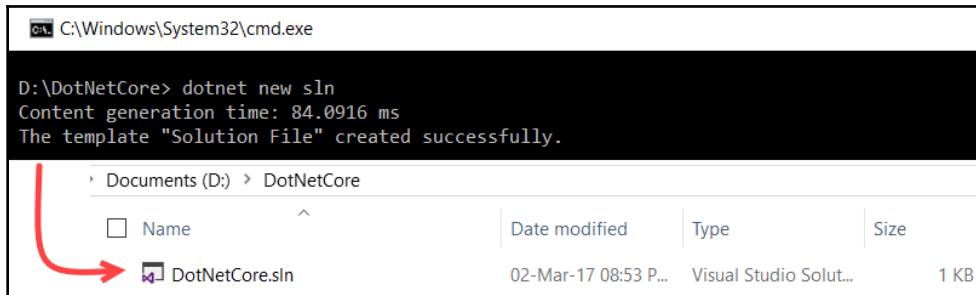
This will create a C# project named `CustomLibrary.csproj` and a C# class file named `Class1.cs` in a directory named `CustomLibrary`, which is under the current directory.

Creating a solution file and adding projects to it

Using the .NET Core CLI, you can also create a solution file and add/remove projects to/from it, respectively. To create a default solution file, enter the following command in the console window:

```
dotnet new sln
```

Upon entering the command, we see the following:



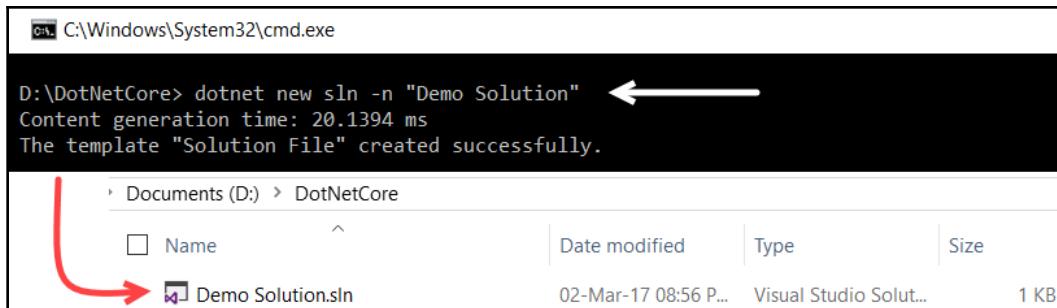
```
C:\Windows\System32\cmd.exe
D:\DotNetCore> dotnet new sln
Content generation time: 84.0916 ms
The template "Solution File" created successfully.

Documents (D:) > DotNetCore
Name          Date modified   Type      Size
DotNetCore.sln 02-Mar-17 08:53 P... Visual Studio Solut... 1 KB
```

This will create a solution file named `DotNetCore.sln` in the current directory. If you want to specify a name while creating the solution file, then enter the following command:

```
dotnet new sln -n "Demo Solution"
```

On entering the command, we get the following:



```
C:\Windows\System32\cmd.exe
D:\DotNetCore> dotnet new sln -n "Demo Solution" ←
Content generation time: 20.1394 ms
The template "Solution File" created successfully.

Documents (D:) > DotNetCore
Name          Date modified   Type      Size
Demo Solution.sln 02-Mar-17 08:56 P... Visual Studio Solut... 1 KB
```

To modify the solution file from the command line to add projects to it, enter any of the following commands, providing the proper name of the solution and project name:

```
> dotnet sln <SolutionName> add <ProjectName>
> dotnet sln <SolutionName> add <ProjectOneName> <ProjectTwoName>
> dotnet sln <SolutionName> add **/**
```

When you want to add a single project in a solution file, use the first command. Use the second command with the names of all the projects that you want to add. When you want to add all the projects that are available in the current directory, you can use `*/**`, as shown in the third command in the preceding code snippet.

Similarly, you can remove any projects from the solution file. You can use any of the following commands as per your requirement:

```
> dotnet sln <SolutionName> remove <ProjectName>
> dotnet sln <SolutionName> remove <ProjectOneName> <ProjectTwoName>
> dotnet sln <SolutionName> remove **/**
```

The first command in the preceding block is used to remove a single file. When there are multiple projects, and you want to remove them selectively, run the second command. Use the third command when you want to remove all the projects in the current directory from the solution file.

Let's do this in practice. We will first create two .NET Core projects of the **console app** and **class library** types. Then, we will add them to a newly created solution file. Now, navigate to the folder where you want to create these projects and follow these steps to build our first solution file:

1. Create a console app project by entering the following command:

```
dotnet new console -n "Demo App"
```

2. Create a class library project by entering the following command:

```
dotnet new classlib -n "Custom Library"
```

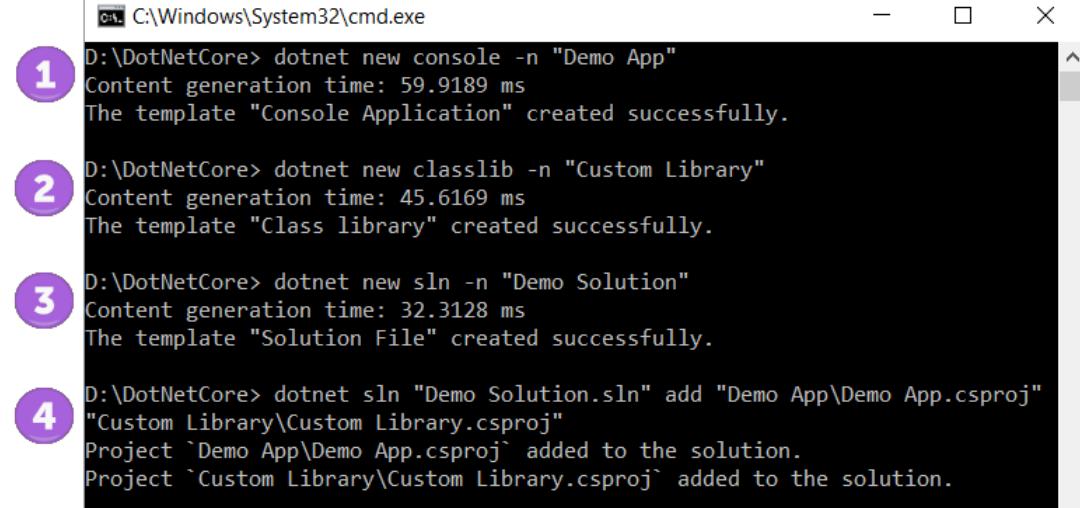
3. Create a solution file:

```
dotnet new sln -n "Demo Solution"
```

4. Add the projects to the solution file:

```
dotnet sln "Demo Solution.sln" add "Demo App\Demo App.csproj"
"Custom Library\Custom Library.csproj"
```

Here's a screenshot demonstrating the preceding steps in a console window:



The screenshot shows a Windows Command Prompt window titled 'cmd.exe' with the path 'C:\Windows\System32\cmd.exe'. The window contains four numbered steps (1 through 4) illustrating the creation of a solution and two projects using the 'dotnet new' command:

- 1** D:\DotNetCore> dotnet new console -n "Demo App"
Content generation time: 59.9189 ms
The template "Console Application" created successfully.
- 2** D:\DotNetCore> dotnet new classlib -n "Custom Library"
Content generation time: 45.6169 ms
The template "Class library" created successfully.
- 3** D:\DotNetCore> dotnet new sln -n "Demo Solution"
Content generation time: 32.3128 ms
The template "Solution File" created successfully.
- 4** D:\DotNetCore> dotnet sln "Demo Solution.sln" add "Demo App\Demo App.csproj"
"Custom Library\Custom Library.csproj"
Project `Demo App\Demo App.csproj` added to the solution.
Project `Custom Library\Custom Library.csproj` added to the solution.

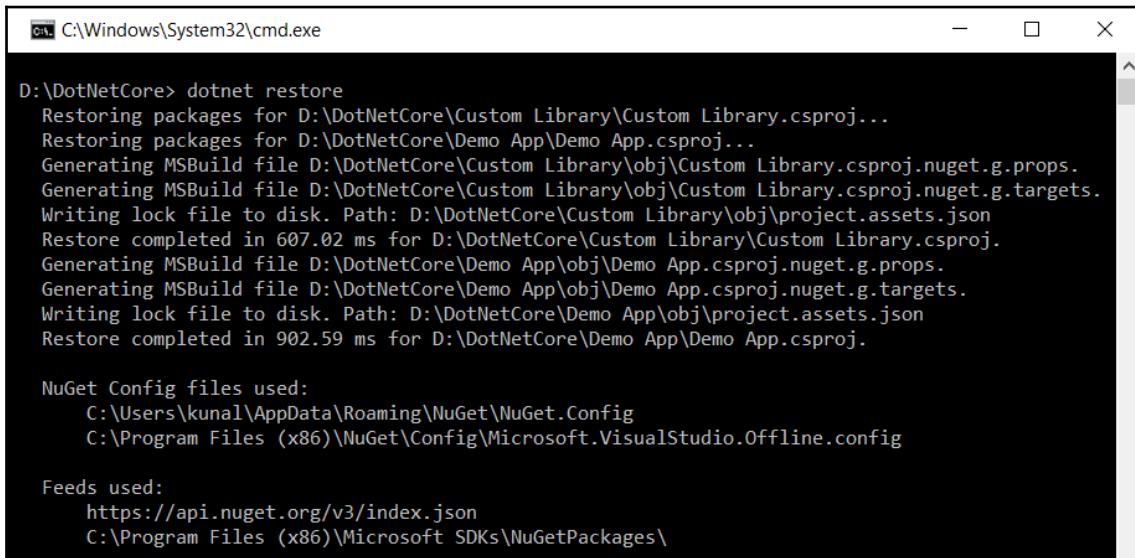
Once your project has been created from the command line, we can resolve all the dependencies related to the project, which we will discuss next.

Resolving dependencies in .NET Core applications

.NET Core uses NuGet to restore dependencies, which is done in parallel when you enter the command from a command line in a directory where your project(s)/solution(s) reside. Let's resolve the dependencies of the projects that we have created just now. Enter the following command from the current directory (D:\DotNetCore, in our case):

```
> dotnet restore
```

In the following screenshot, you can see how the command is executed in parallel to first restore its package dependencies, and then the tool-related dependencies for MSBuild:



The screenshot shows a Windows Command Prompt window titled 'C:\Windows\System32\cmd.exe'. The command 'dotnet restore' is run from the directory 'D:\DotNetCore'. The output shows the process of restoring packages for two projects: 'Custom Library' and 'Demo App'. It includes generating MSBuild files, writing lock files, and listing used NuGet Config files and feeds.

```
D:\DotNetCore> dotnet restore
Restoring packages for D:\DotNetCore\Custom Library\Custom Library.csproj...
Restoring packages for D:\DotNetCore\Demo App\Demo App.csproj...
Generating MSBuild file D:\DotNetCore\Custom Library\obj\Custom Library.csproj.nuget.g.props.
Generating MSBuild file D:\DotNetCore\Custom Library\obj\Custom Library.csproj.nuget.g.targets.
Writing lock file to disk. Path: D:\DotNetCore\Custom Library\obj\project.assets.json
Restore completed in 607.02 ms for D:\DotNetCore\Custom Library\Custom Library.csproj.
Generating MSBuild file D:\DotNetCore\Demo App\obj\Demo App.csproj.nuget.g.props.
Generating MSBuild file D:\DotNetCore\Demo App\obj\Demo App.csproj.nuget.g.targets.
Writing lock file to disk. Path: D:\DotNetCore\Demo App\obj\project.assets.json
Restore completed in 902.59 ms for D:\DotNetCore\Demo App\Demo App.csproj.

NuGet Config files used:
  C:\Users\kunal\AppData\Roaming\NuGet\NuGet.Config
  C:\Program Files (x86)\NuGet\Config\Microsoft.VisualStudio.Offline.config

Feeds used:
  https://api.nuget.org/v3/index.json
  C:\Program Files (x86)\Microsoft SDKs\NuGetPackages\
```

Unless these dependencies are resolved, you cannot proceed to build and run your application.

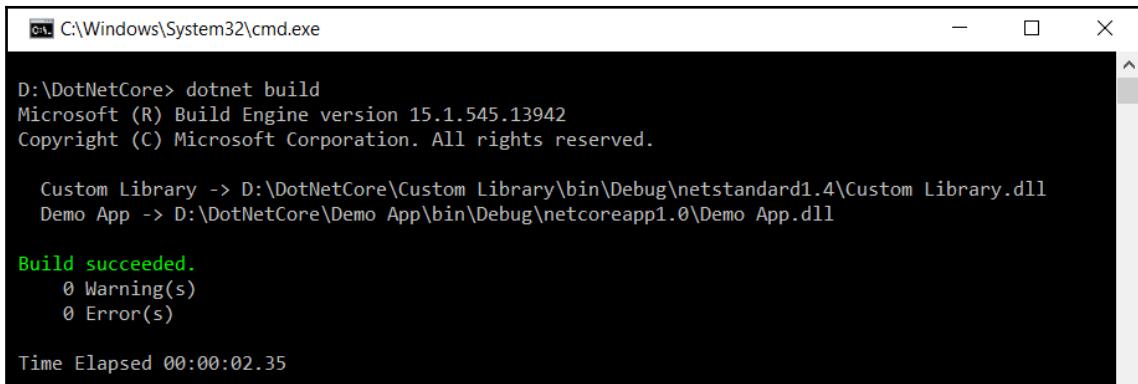
Building a .NET Core project or solution

To build a .NET Core project from the command line, you can use the `dotnet build` command. When you specify it in a directory that has multiple projects, `dotnet build` will be performed for all the projects.

When there is a solution in the directory that has at least one project linked, performing the following command will build all the linked projects in the solution file:

```
> dotnet build
```

This is the output for the `dotnet build` command:



```
C:\Windows\System32\cmd.exe

D:\DotNetCore> dotnet build
Microsoft (R) Build Engine version 15.1.545.13942
Copyright (C) Microsoft Corporation. All rights reserved.

  Custom Library -> D:\DotNetCore\Custom Library\bin\Debug\netstandard1.4\Custom Library.dll
  Demo App -> D:\DotNetCore\Demo App\bin\Debug\netcoreapp1.0\Demo App.dll

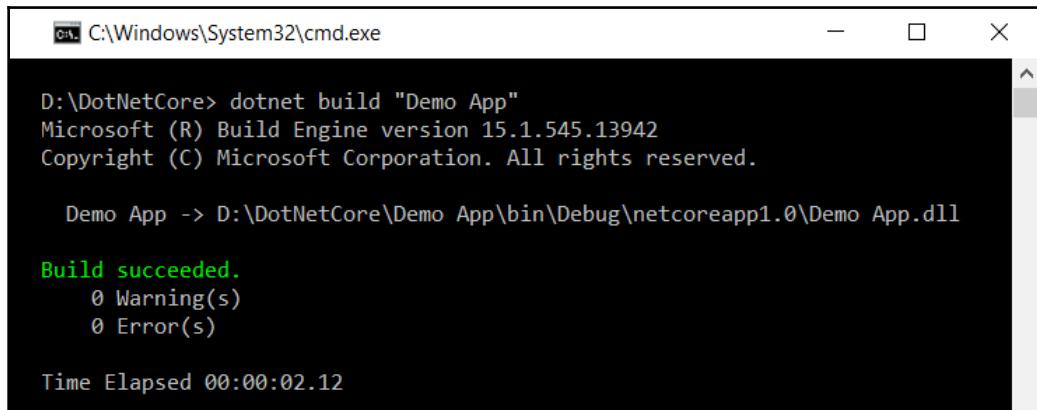
Build succeeded.
  0 Warning(s)
  0 Error(s)

Time Elapsed 00:00:02.35
```

To build a specific project or solution, provide the name or the path to it while executing the `dotnet build` command. Here's the command to build the `Demo App` project that we have created (you can use either of them):

```
> dotnet build "Demo App"
> dotnet build "Demo App\Demo App.csproj"
```

This is the output for it:



```
C:\Windows\System32\cmd.exe

D:\DotNetCore> dotnet build "Demo App"
Microsoft (R) Build Engine version 15.1.545.13942
Copyright (C) Microsoft Corporation. All rights reserved.

  Demo App -> D:\DotNetCore\Demo App\bin\Debug\netcoreapp1.0\Demo App.dll

Build succeeded.
  0 Warning(s)
  0 Error(s)

Time Elapsed 00:00:02.12
```



Building a .NET Core project requires the dependencies to be resolved first. Hence, you should execute the `dotnet restore` command at least once before building your code.

The output of the build will be generated in the `bin` folder of the project directories. By default, it builds against the `debug` configuration settings. But, to build the project(s) in `release` mode, the `--configuration Release` option needs to be specified as follows:

```
> dotnet build --configuration Release  
> dotnet build "Demo App" --configuration Release
```

You can also target the build for a different runtime by specifying `--runtime` followed by `<RuntimeIdentifier>`. If you want to build the project/solution against the runtime of Red Hat Enterprise Linux 7 (64-bit), then enter the following commands one by one:

```
> dotnet clean  
> dotnet resolve --runtime rhel.7-x64  
> dotnet build --runtime rhel.7-x64
```

The first command will clean the current workspace. The second command will build the projects/solutions against the specified runtime (`rhel.7-x64`). Then, the third command will build against the runtime specified.



Make sure to resolve the dependencies before retargeting to build to a different runtime.

When you build a project or a solution, the output files are written in the `bin` folder of the project directory. All the temporary files related to the build are generated in the `obj` folder of the project directory, just like any other projects targeting .NET Framework.

Running a .NET Core application

To run a .NET Core application, you can use the `dotnet run` command from the console window. As this command relies on the `dotnet build` command, when you specify a project to run, it automatically builds the project and then launches the application.

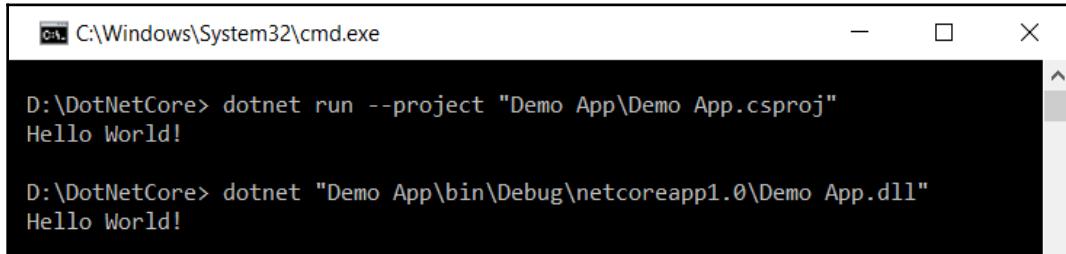
To run an app in the project context, enter the following command:

```
dotnet run --project <ProjectPath>
```

If you already have a portable application DLL, then you can run it directly by executing the DLL by the `dotnet` driver without specifying any command. For example, the following command runs the already available DLL, which was built against .NET Core 1.0:

```
dotnet "Demo App\bin\Debug\netcoreapp1.0\Demo App.dll"
```

The output is as follows:



A screenshot of a Windows Command Prompt window titled 'C:\Windows\System32\cmd.exe'. The window contains the following text:

```
D:\DotNetCore> dotnet run --project "Demo App\Demo App.csproj"
Hello World!

D:\DotNetCore> dotnet "Demo App\bin\Debug\netcoreapp1.0\Demo App.dll"
Hello World!
```

As you can now build and run .NET Core applications, it's time for you to learn about how to publish a .NET Core application. In the next section, we are going to discuss this.

Publishing a .NET Core application

When you build, debug, and test your application successfully, you need to publish or deploy your app. .NET Core provides a command, `dotnet publish`, which publishes your app for deployment.

To publish a .NET Core app in `Release` mode, enter the following command:

```
> dotnet publish <ProjectName> -c Release
```

Two kinds of deployments are available for .NET Core applications: FDD and SCD. FDD depends on a system-wide version of .NET Core being present before running the application, whereas SCD does not depend on any .NET Core versions that are already installed on your system as it includes the entire .NET Core libraries and runtime along with the resultant application files.

Let's learn more about each of the preceding deployment types with simple demo-oriented examples.

FDD

This type of deployment model executes by default while publishing a .NET Core application. In this deployment step, only your app and any third-party dependencies are deployed, and there is no reason to deploy .NET Core along with your app as your app will use the .NET Core version that's present on your system.

In this model, the output of the .NET Core application is always a .dll file. You won't find any .exe files to run.

This type of deployment has a number of benefits:

- As you don't need to deploy .NET Core along with your app, the size of your deployment package stays very small.
- It reduces disk space and memory usage on your system since multiple apps can use the single .NET Core installation.
- As .NET Core uses a common **Portable Executable (PE)** format that the underlying OS can understand, there's no need to define the target OS.

The following steps are used to create, build, run, and publish an app using the FDD model:

1. Create a directory for your solution (for example, DotNetCore) and navigate to it in a command window.
2. Enter the following command to create a new C# console application:

```
> dotnet new console -lang C# -n "FDD Demo"
```

3. Create a solution file named FDD Demo and add the project to the solution file:

```
> dotnet new sln -n "FDD Demo"  
> dotnet sln "FDD Demo.sln" add "FDD Demo\FDD Demo.csproj"
```

4. Now, resolve the dependencies by entering the following command:

```
> dotnet restore
```

5. Build the solution by running the following command, which will generate the PE version of the application in the bin\debug folder:

```
> dotnet build
```

6. If the build succeeds, then the output DLL will be generated in `DotNetCore\FDD Demo\bin\Debug\netcoreapp1.0\FDD Demo.dll`. Now, enter the following command to run it:

```
> dotnet "FDD Demo\bin\Debug\netcoreapp1.0\FDD Demo.dll"
```

7. Once you are ready to publish your app, enter the following command to create the release version of the project to the `DotNetCore\FDD Demo\bin\Release\netcoreapp1.0\publish` folder:

```
> dotnet publish "FDD Demo\FDD Demo.csproj" -c Release
```

Here, in the FDD model, you will only have your app and its dependency files (if any), along with the PDB file and JSON configuration files in the publish directory. No other files will be deployed there, thereby, keeping your deployment package very small.

SCD

In this type of deployment model, .NET Core is also deployed along with your application and dependency libraries. Thus, the size of your deployment package becomes large when compared to the package created by the FDD model. The version of .NET Core depends on the version of the framework that you build your app with.

In this model, the output of the .NET Core application is an `.exe` file, which loads the actual `.dll` file on execution.

The main advantages of this type of deployment are as follows:

- As you are bundling the version of .NET Core that your app needs to run, you can be assured that it will run on the target system.
- Only you can decide which version of .NET Core your app will support. You should select the target platform before you create the deployment package.
- The output is an EXE file, which loads the actual DLL.

Let's follow these steps to create, build, run, and publish an app using the SCD model:

1. Create a directory for your solution (for example, `DotNetCore`) and navigate to it in a command window.
2. Enter the following command to create a new C# console application:

```
dotnet new console -lang C# -n "SCD Demo"
```

3. Create a solution file named `SCD Demo` and add the project to the solution file:

```
> dotnet new sln -n "SCD Demo"  
> dotnet sln "SCD Demo.sln" add "SCD Demo\SCD Demo.csproj"
```

4. Now, open the `SCD Demo.csproj` project file in a notepad.
5. Create a `<RuntimeIdentifiers>` tag under the `<PropertyGroup>` section of your `.csproj` project file and define the platforms that your app targets:

```
<PropertyGroup>  
  <RuntimeIdentifiers>win10-x64</RuntimeIdentifiers>  
</PropertyGroup>
```

6. Here, you can also set multiple identifiers separated by semicolons.
7. Now, resolve the dependencies by entering the following command:

```
dotnet restore
```

8. Build the solution by running the following command (enter the runtime identifier that you want your app to target), which will generate the output of the application in the `bin\Debug\netcoreapp1.0` folder:

```
dotnet build -r win10-x64
```

9. If the build succeeds, then the output DLL will be generated along with the `.exe` file as `DotNetCore\SCD Demo\bin\Debug\netcoreapp1.0\SCD Demo.dll` and `DotNetCore\SCD Demo\bin\Debug\netcoreapp1.0\SCD Demo.exe`.

10. Now, enter the following command to run it:

```
dotnet "SCD Demo\bin\Debug\netcoreapp1.0\SCD Demo.dll"
```

11. You can also run the `.exe` file directly:

```
"SCD Demo\bin\Debug\netcoreapp1.0\SCD Demo.exe"
```

12. Once you are ready to publish your app, enter the following command to create the release version of the project to the `DotNetCore\SCD Demo\bin\Release\netcoreapp1.0\win10-x64\publish` folder:

```
dotnet publish -c Release -r win10-x64
```

If you open the published folder, then you will see many other files apart from the project DLL, project EXE, JSON configurations, and PDB files. You can create as many deployment packages as you want based on your targeted **runtime identifiers (RIDs)**. Separate folders will be generated for each RID package.

Creating an ASP.NET Core application

There are three different types of ASP.NET Core project templates that are available to use:

- The empty ASP.NET Core web application (identified as `web`)
- The MVC ASP.NET Core web application (identified as `mvc`)
- The web API ASP.NET Core web application (identified as `webapi`)

Among these, only the MVC web app currently has support for both the C# and F# languages.

To create an empty ASP.NET Core application, enter the following command in the console window:

```
dotnet new web -n <ProjectName>
```

To create an MVC-based ASP.NET web app, enter the following command:

```
dotnet new mvc -n <ProjectName>
```

To create an ASP.NET MVC web application with no authentication, enter the following command:

```
dotnet new mvc -au None -n <ProjectName>
```

To create an ASP.NET Core web API application, enter the following in the command line:

```
dotnet new webapi -n <ProjectName>
```

By default, the ASP.NET Core web applications support SQLite for database support. You can change it to use a LocalDB instead of SQLite. To change it, append either of the `-uld` or `--use-local-db` options with a value of `true`, as shown in the following command:

```
dotnet new mvc -n <ProjectName> -uld true -au none
```

This will give the following output:

```
C:\Windows\System32\cmd.exe
D:\DotNetCore> dotnet new mvc -n "MVC Demo" -uld true -au none
Content generation time: 1072.6964 ms
The template "MVC ASP.NET Core Web Application" created successfully.

> DotNetCore > MVC Demo >

```

Name	Date modified	Type	Size
Controllers	04-Mar-17 11:19 A...	File folder	
Views	04-Mar-17 11:19 A...	File folder	
wwwroot	04-Mar-17 11:19 A...	File folder	
.bowerrc	04-Mar-17 11:19 A...	BOWERC File	1 KB
appsettings.Development	04-Mar-17 11:19 A...	JSON File	1 KB
appsettings	04-Mar-17 11:19 A...	JSON File	1 KB
bower	04-Mar-17 11:19 A...	JSON File	1 KB
bundleconfig	04-Mar-17 11:19 A...	JSON File	1 KB
MVC Demo.csproj	04-Mar-17 11:19 A...	Visual C# Project F...	1 KB
Program.cs	04-Mar-17 11:19 A...	Visual C# Source F...	1 KB
Startup.cs	04-Mar-17 11:19 A...	Visual C# Source F...	2 KB

By using the proper option parameters, you can now select the desired ASP.NET Core template that you are going to create. In the next section, we are going to learn about how to create a unit testing project in .NET Core.

Creating a unit testing project

Unit testing projects can also be created using .NET Core. In the command line, you can specify the `mstest` or `xunit` template code to create a simple unit testing project or `xUnit` testing project, respectively.

To create a simple unit test project from the command line, enter the following:

```
dotnet new mstest -n <ProjectName>
```

To create a `xUnit` testing project, enter the following command:

```
dotnet new xunit -n <ProjectName>
```

You can also run a unit testing project from the command line. You need to execute the `dotnet test` command to do so. The unit testing frameworks are bundled as a NuGet package and can be restored as ordinary dependencies. To do so, you can execute either of the following commands:

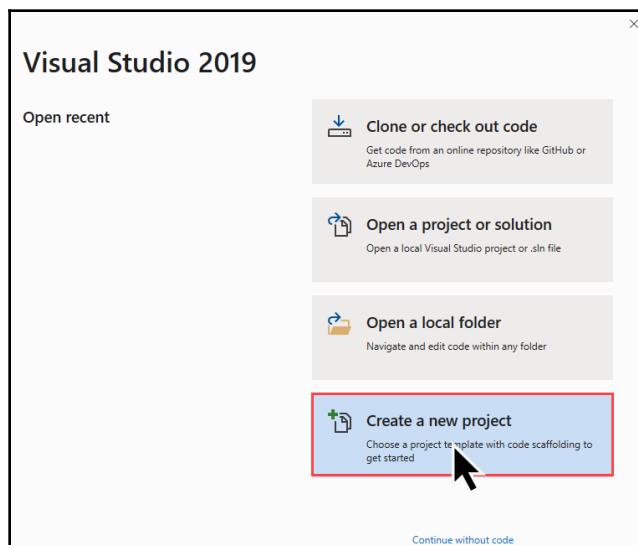
```
> dotnet test  
> dotnet test <ProjectPath>
```

As we have learned about how to create .NET Core applications from the command line, let's jump into the next section to learn about how to do this using Visual Studio 2019.

Creating .NET Core applications using Visual Studio 2019

We have spent enough time discussing the .NET Core commands to create, build, run, and publish .NET Core applications. We have also covered how to create and run ASP.NET Core applications and unit test projects.

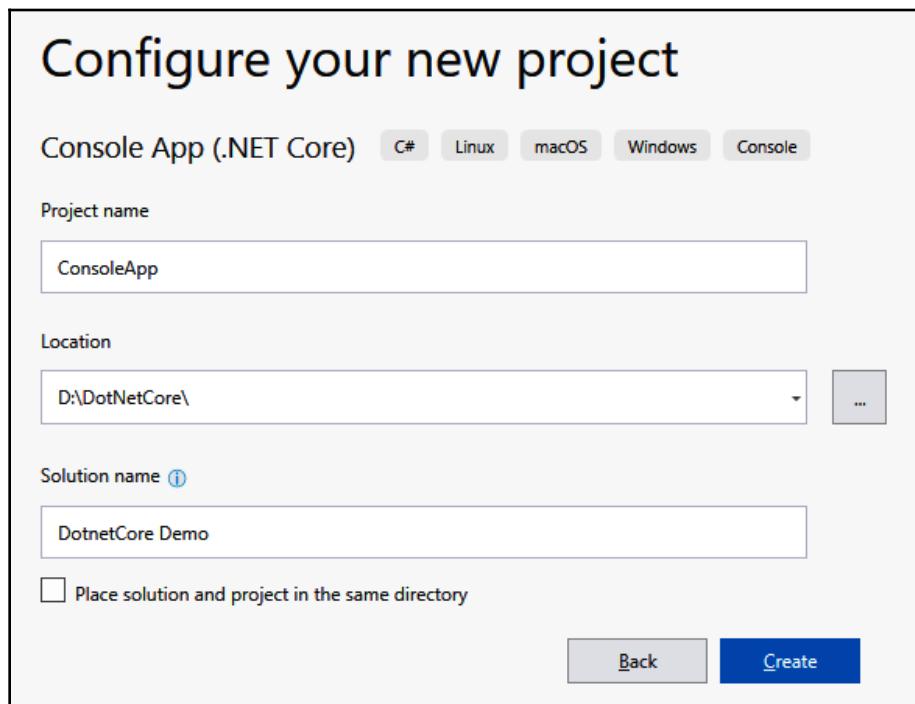
Now, let's see how to do it easily using Visual Studio 2019. To create a .NET Core application using Visual Studio 2019, open the Visual Studio 2019 instance and, from the Start Screen, select **Create a new project**. Alternatively, you can click the **File | New | Project...** (shortcut: *Ctrl + Shift + N*) menu from the Visual Studio 2019 IDE to open the **New Project** window:



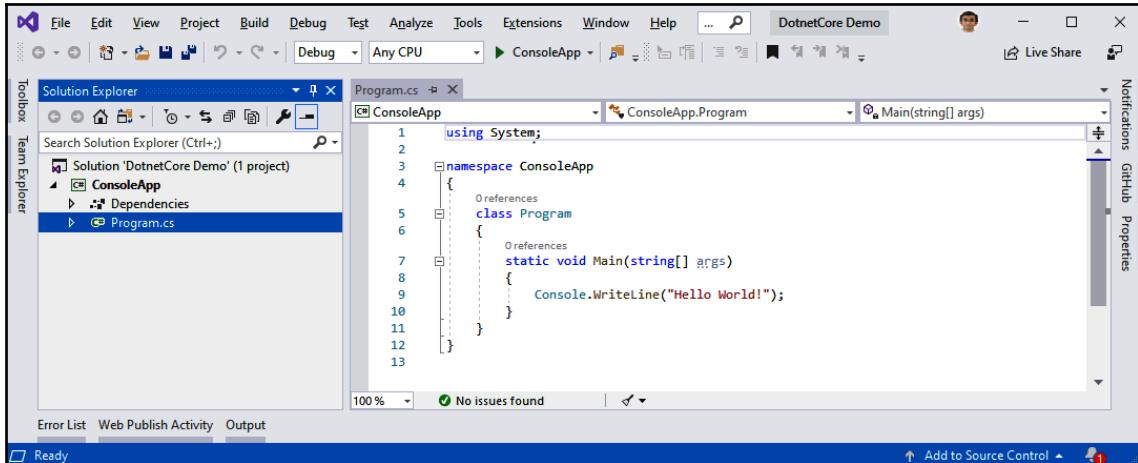
In the **New Project** dialog, search for `.net core`, which will list the following project templates:

- Console App (.NET Core)
- ASP.NET Core web application
- Class library (.NET Core)
- MSTest test project (.NET Core)
- xUnit test project (.NET Core)
- NUnit test project (.NET Core)
- Web driver test for edge (.NET Core)

You can select the template that you want to create. To demonstrate, let's create a console application first. From the available list of project templates, select **Console App (.NET Core)**. Provide a proper name for the project/solution file, select the location, and click **Create** to create the project, as shown in the following screenshot:



Like other project templates, and unlike the CLI tools for .NET Core, this wizard will create the solution and add the project to it automatically. A class file named `Program.cs` will be available, which is the entry point to the application:



Let's build the application and run it. You will see a **Hello World** string printed on the screen. If you navigate to the `bin\debug\netcoreapp2.1` folder path (as we have built on .NET Core 2.1) of the project directory, then you will see `ConsoleApp.dll`, which is our application host.

Publishing .NET Core applications using Visual Studio 2019

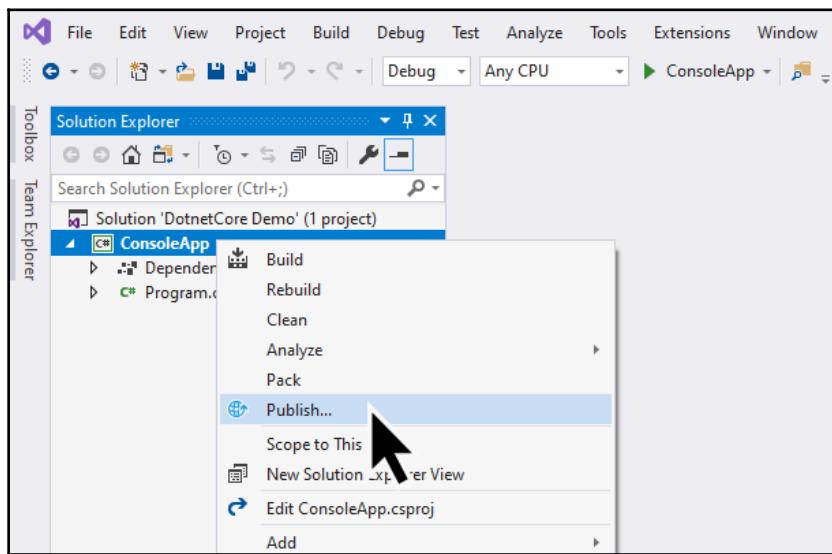
Visual Studio 2019 made it easy to publish a .NET Core application. In a CLI, it's all about commands that you need to perform one by one to publish, but, in Visual Studio 2019, it's just about a few clicks.

As on the command line, you can publish an application in both the FDD and SCD models from Visual Studio 2019. Let's learn about how to do this with both the deployment models.

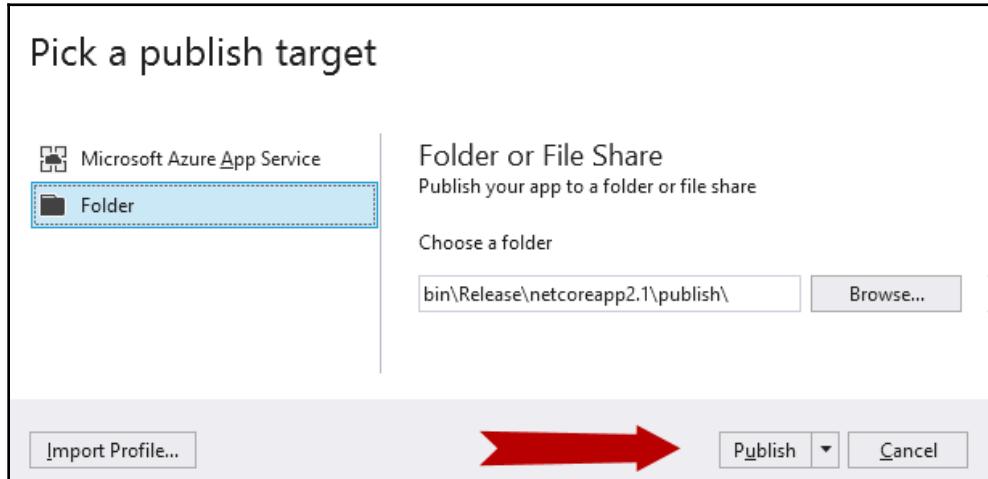
FDD

The default deployment model that Visual Studio configures for a .NET Core application is the FDD model. Here, when you publish an app, only the portable executable DLL will be generated, and you need to run this in a system where .NET Core is already installed.

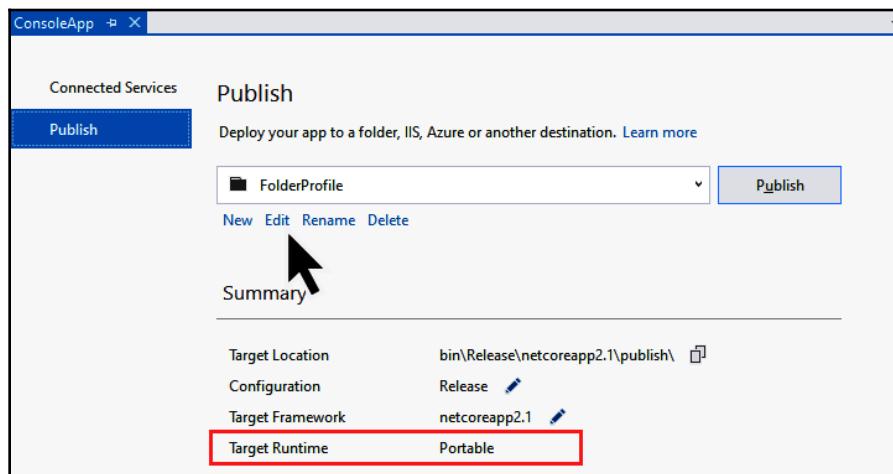
To publish an app to create the deployment package, right-click on the project and, from the context menu that pops up on the screen, click **Publish...**, as shown in the following screenshot:



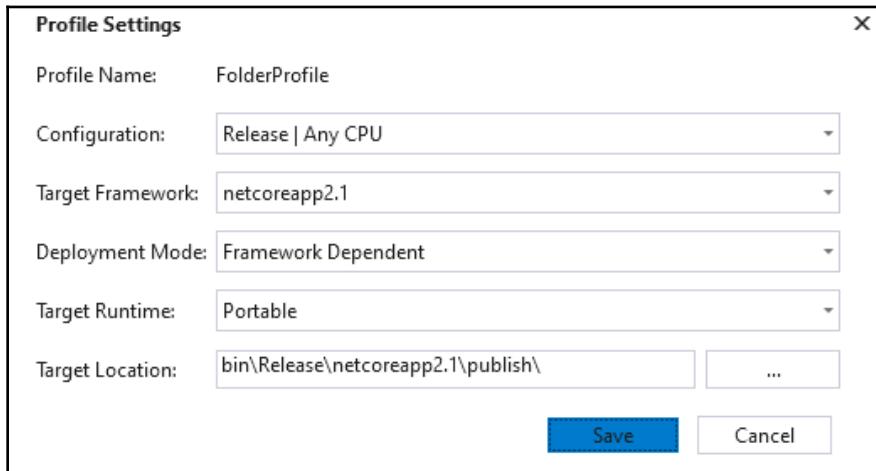
This will show the publish wizard dialog, where you can set the publish target. You can select **Microsoft Azure App Service** or **Folder** as the publish target. Here, let's select **Folder** as the publishing target:



Click the **Browse...** button to select the publish folder (the default is `bin\Release\netcoreapp2.1\publish`) and, when you are ready, click on the **Publish** button. This will publish the app to the selected folder and show you a summary:



In the summary section, you can see that the build configuration is **Release** and the **Target Runtime** is **Portable**. Click on the link labeled **Edit**, which will open the following **Profile Settings** page:



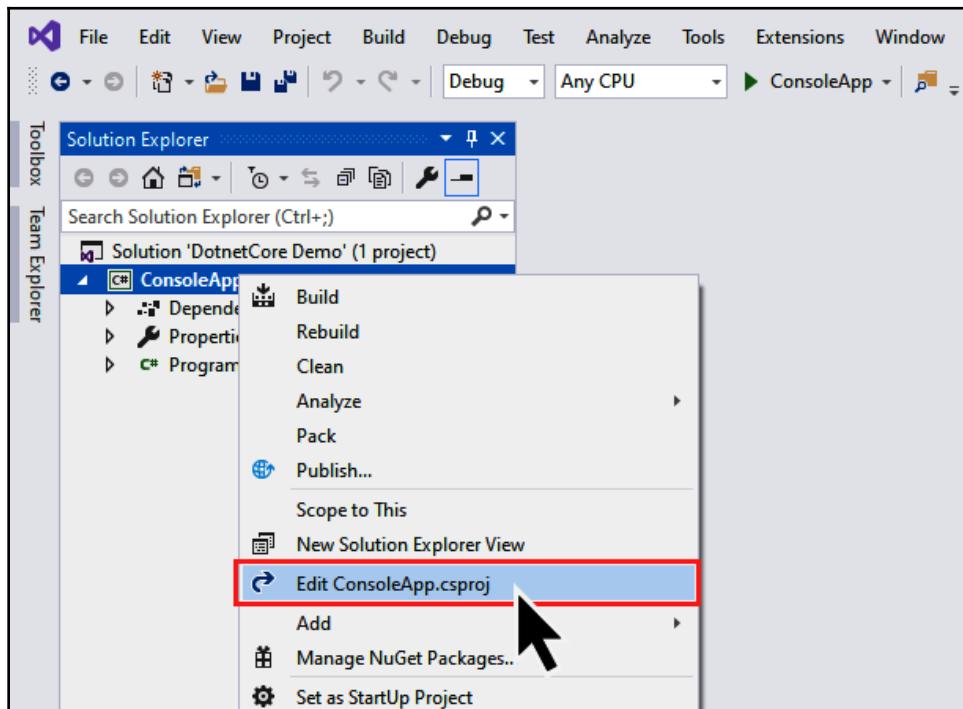
In this screen, you will be able to change the configuration, target framework, and target location. As we are publishing it as a **Portable** executable format (the FDD model), you won't be able to change the target runtime.

SCD

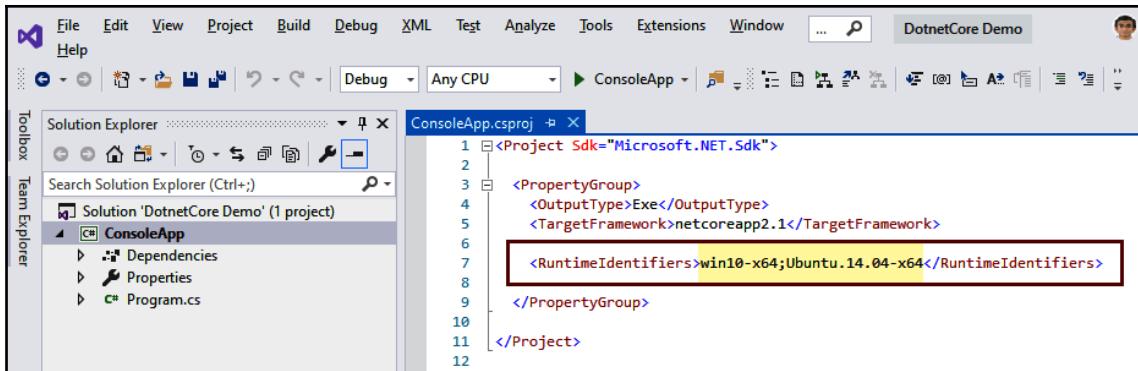
To use an SCD for your application to run in a system that doesn't have .NET Core installed, you need to first edit your project file to make some changes. In this type of deployment, the core libraries will also be copied with your application, making the deployment package larger than expected.

For SCD, first open your project file and perform the following steps:

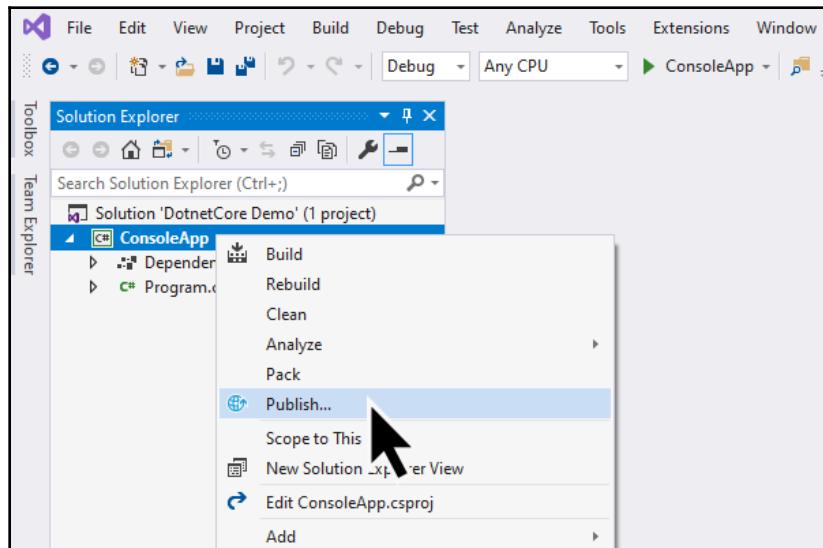
1. In Visual Studio 2019, right-click on the project in **Solution Explorer** and click the **Edit <AppName>.csproj** menu item. In our case, it's **Edit ConsoleApp.csproj**, as shown in the following screenshot:



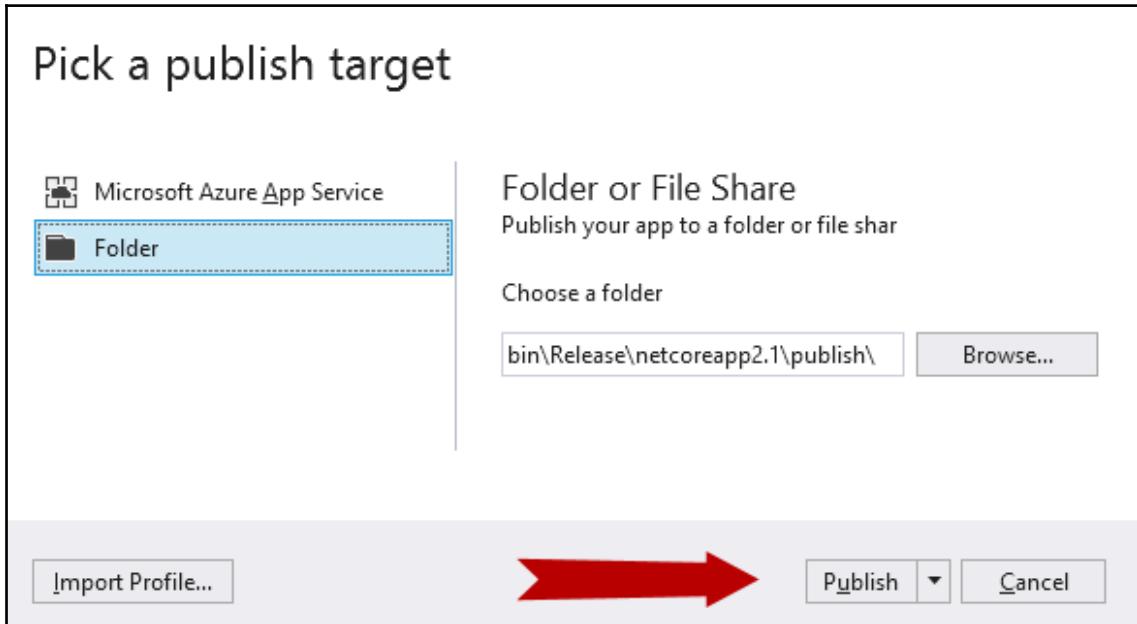
2. As shown in the following screenshot, enter the following tag, `<RuntimeIdentifiers>win10-x64;Ubuntu.14.04-x64</RuntimeIdentifiers>`, inside the `<PropertyGroup>` tag as a child, where `win10-x64` and `ubuntu.14.04-x64` are two different RIDs that we have selected for the deployment of our app. You can add more than one RID here as semicolon separated values:



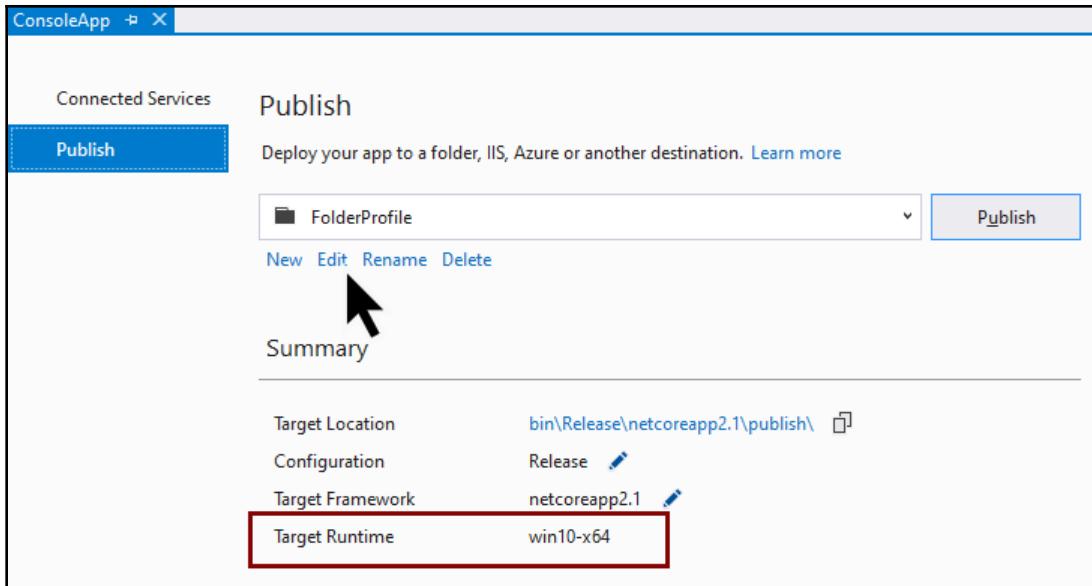
3. Now, save the .csproj file, build the solution, and right-click on the project inside **Solution Explorer**.
4. From the context menu, click on the **Publish...** menu item to start the deployment process:



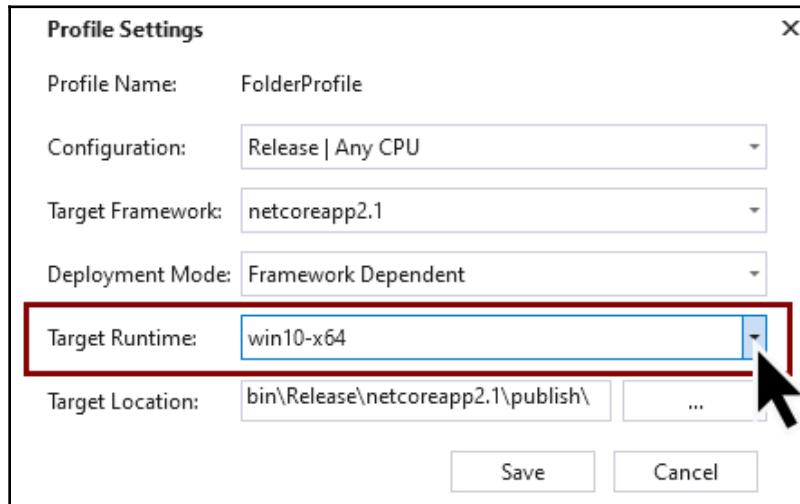
5. This will show the publish wizard dialog, where you can set the publishing target. You can select **Microsoft Azure App Service** or **Folder** as the publish target. Let's select **Folder** in this instance:



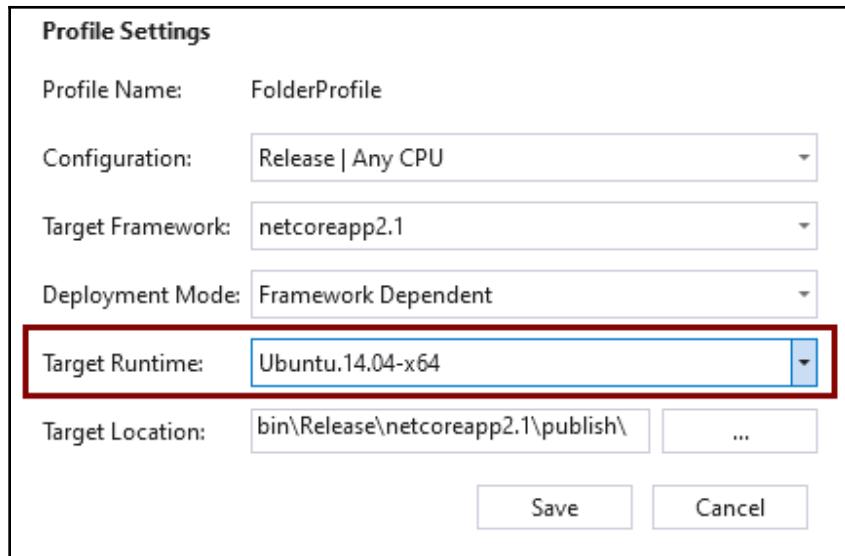
6. Click the **Browse...** button and select the publish folder (the default is `bin\Release\netcoreapp2.1\publish`), where `netcoreapp2.1` belongs to the .NET Core version that you have selected during project creation. Once you are done, click the **Publish** button.
7. This will publish your app to the target location and show you a summary.
8. Note that the configuration is, by default, selected as **Release** and the target runtime is set to your host system RID, which is the first entry that we have added as `RuntimeIdentifier` in the `.csproj` file:



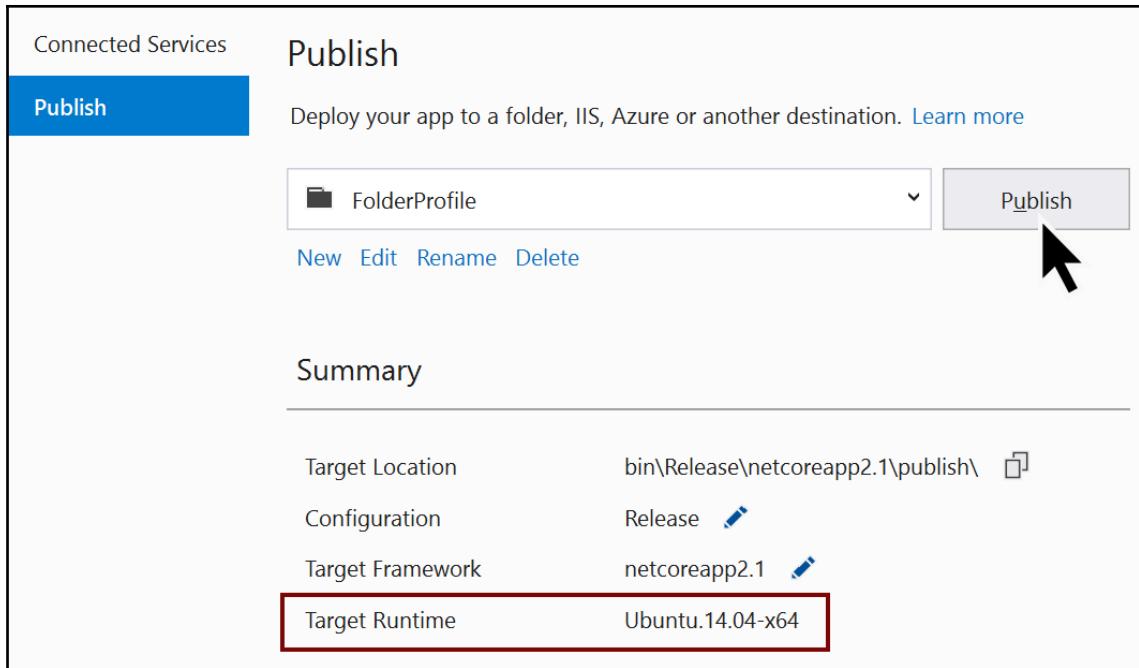
9. To edit the publish profile, click the **Edit** link. This will open the **Profile Settings** page:



10. In this page, you will be allowed to change the **Configuration**, **Target Framework**, **Target Runtime**, and **Target Location** settings. Click the **Target Runtime** dropdown. There, you will see the identifiers that we have set in the project file. In our case, they were `win10-x64` and `unbuntu.14.04-x64`.
11. To publish the application for Ubuntu Linux, select **Ubuntu.14.04-x64** from the list and click **Save**:



12. This will update the **Target Runtime** setting on the **Summary** page to Ubuntu RID. Now, click on the **Publish** button to publish the application. By default, the package will be deployed in the same publish directory.
13. If you want to place them in separate folders, then change the **Target Location** accordingly to easily identify the output:



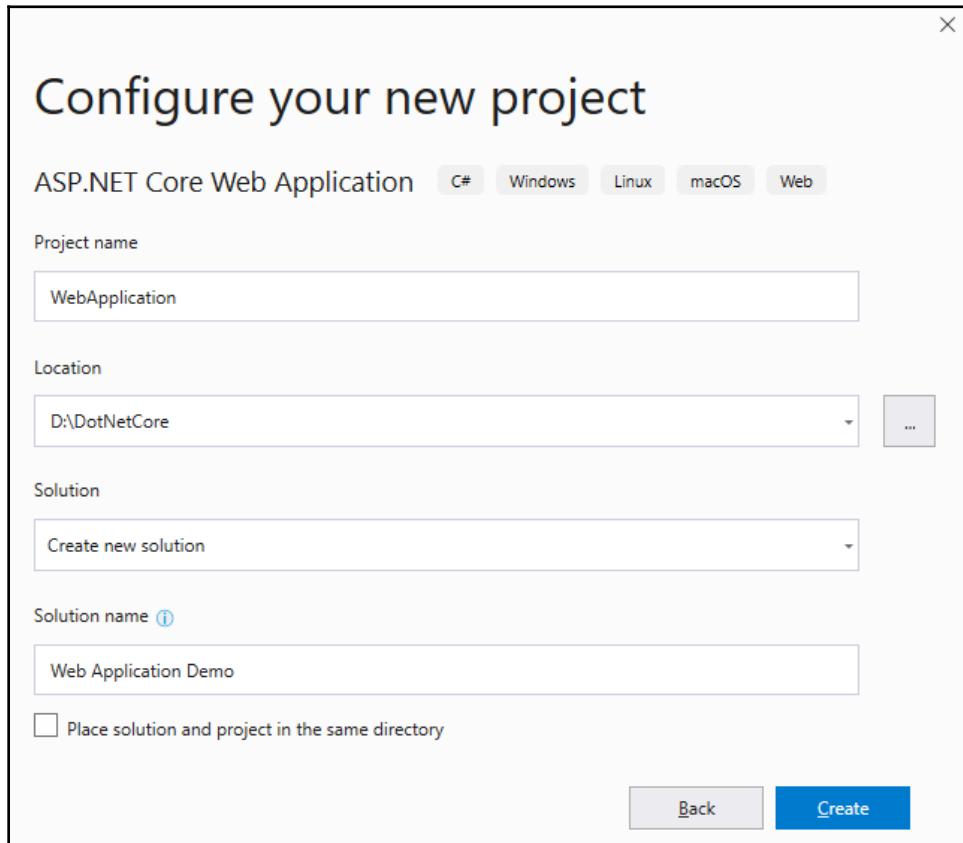
You can also publish the application to other runtimes. In our case, you can select **Portable**, **Ubuntu.14.04-x64**, **win10-x64**, **win-x86**, **win-x64**, **win-arm**, **osx-x64**, **linux-x64**, or **linux-arm** from the **Target Runtime** list.

Creating, building, and publishing a .NET Core web app to Microsoft Azure

As we have already learned about how to create a .NET Core console application, and then build and publish it using the framework-dependent model and self-contained deployment model (both using the CLI and from Visual Studio 2019), let's now begin with creating an ASP.NET Core MVC web application.

To begin with, click the **File | New | Project...** menu, which will open the **New Project** dialog on the screen. Alternatively, you can open it by pressing the keyboard shortcut, **Ctrl + Shift + N**, or from the Visual Studio 2019 start screen.

From the available .NET Core project types, click on the **ASP.NET Core Web Application** template. Enter **Project name**, **Solution name**, and **Location**, and click **Create** to begin the creation of the project:

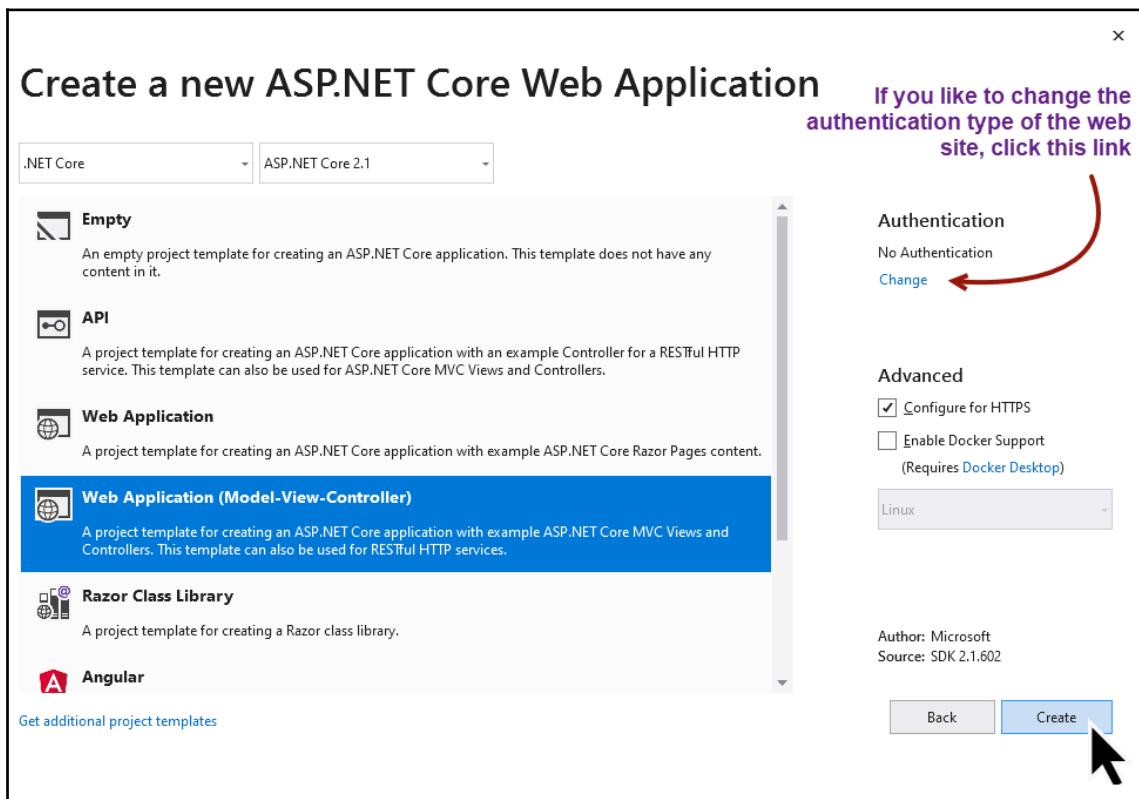


This will open another dialog on the screen, asking you to select the ASP.NET Core web template. Currently, the following templates are available for you to choose from:

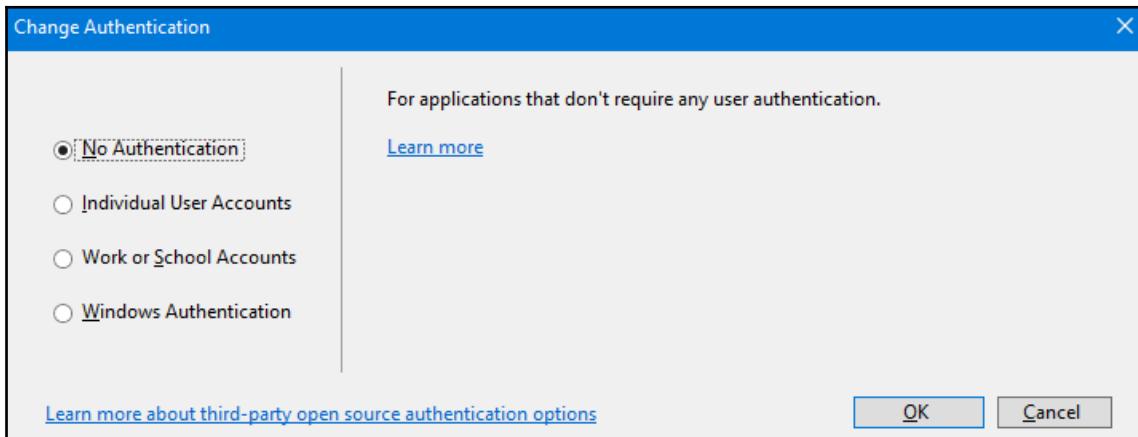
- Empty
- API
- Web application

- Web application (Model-View-Controller)
- Razor class library
- Angular
- React.js
- React.js and Redux

If you want to build a web application from scratch, then select the **Empty** web template. To create an ASP.NET Core application with the RESTful HTTP service, select an **API** template that will give you a sample app to start with. For a complete web application with MVC views and controllers, select the **Web Application (Model-View-Controller)** template, as demonstrated in the following screenshot:



You can also set authentication to your web app (the default is **No Authentication**) by clicking the **Change Authentication** button:

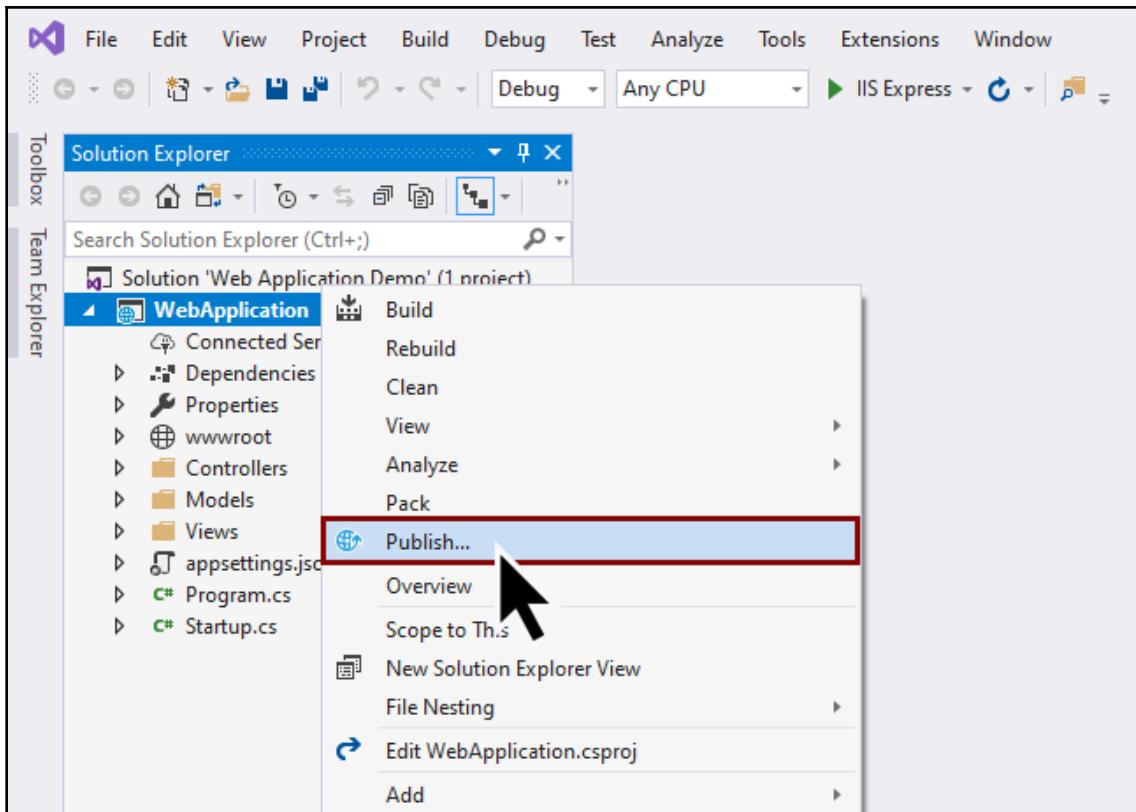


The **Change Authentication** dialog has the following four options:

- **No Authentication:** This is the default authentication type. Keep this selected if your app does not require any user authentication.
- **Individual User Accounts:** If your application stores user profiles to the SQL Server database, then you need to select this option. You can also configure your signing process by using Facebook, Twitter, Google, Microsoft, or any other provider's authentication system.
- **Work or School Accounts:** To use an Active Directory or Office 365 signing process, use this type of authentication system.
- **Windows Authentication:** If you are building an intranet application that will use a Windows authentication system, then select this.

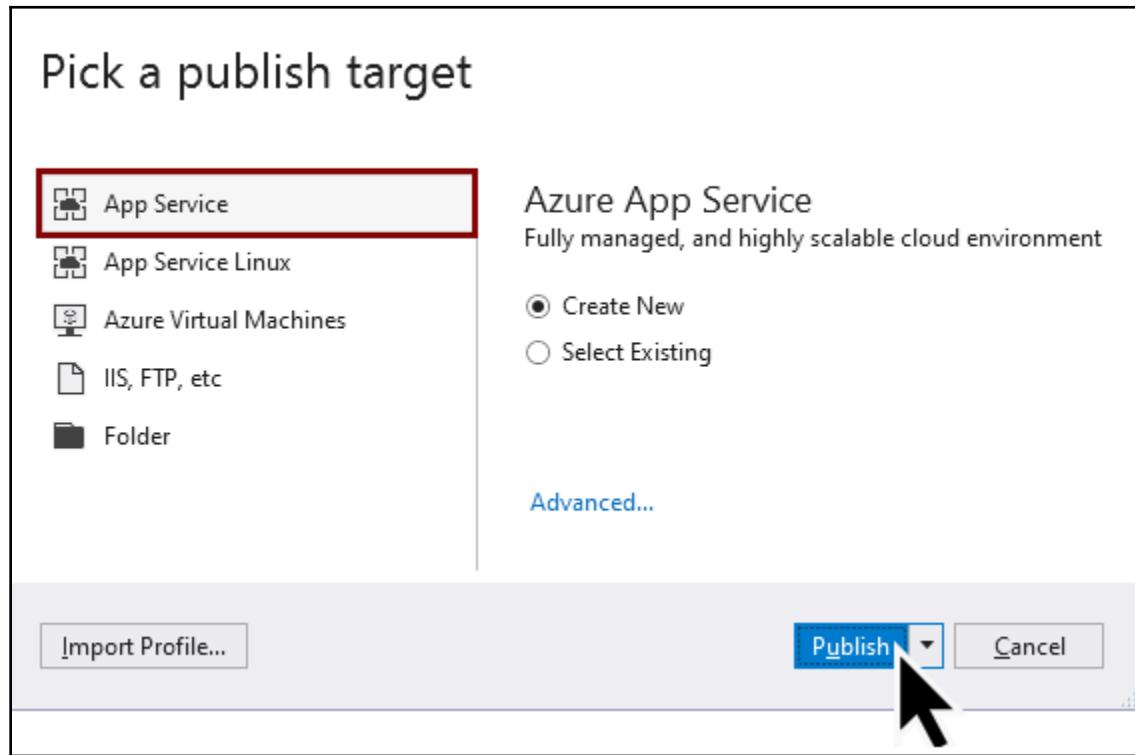
Here, we will use **No Authentication**, so keep it selected. Click **OK** to continue creating the web application.

Once the project has been created by Visual Studio 2019, you can build and run it. This will show a web page from the sample that it has created:



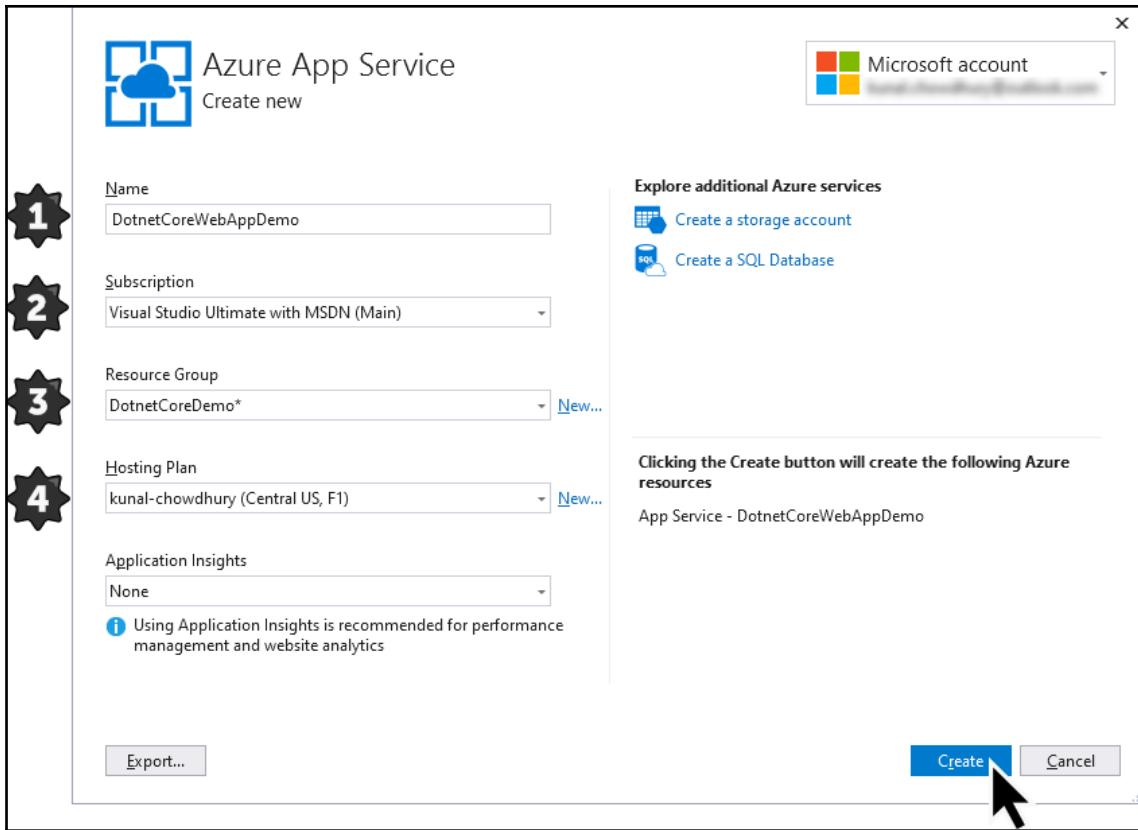
To publish the application that we have just created, right-click on the project inside **Solution Explorer** and click on **Publish**, as shown in the preceding screenshot. This will open the app publishing wizard. Here, you can select **App Service**, **App Service Linux**, **Azure Virtual Machines**, **IIS**, **FTP**, or **Folder** as the publishing target.

As we are going to publish the application to Microsoft Azure, select **App Service** and click the **Publish** button, as shown in the following screenshot:



To deploy an app to Microsoft Azure, you should already have an Azure account. To create your Azure account with \$200 free credit, visit this link: <https://azure.microsoft.com/en-us/free/>.

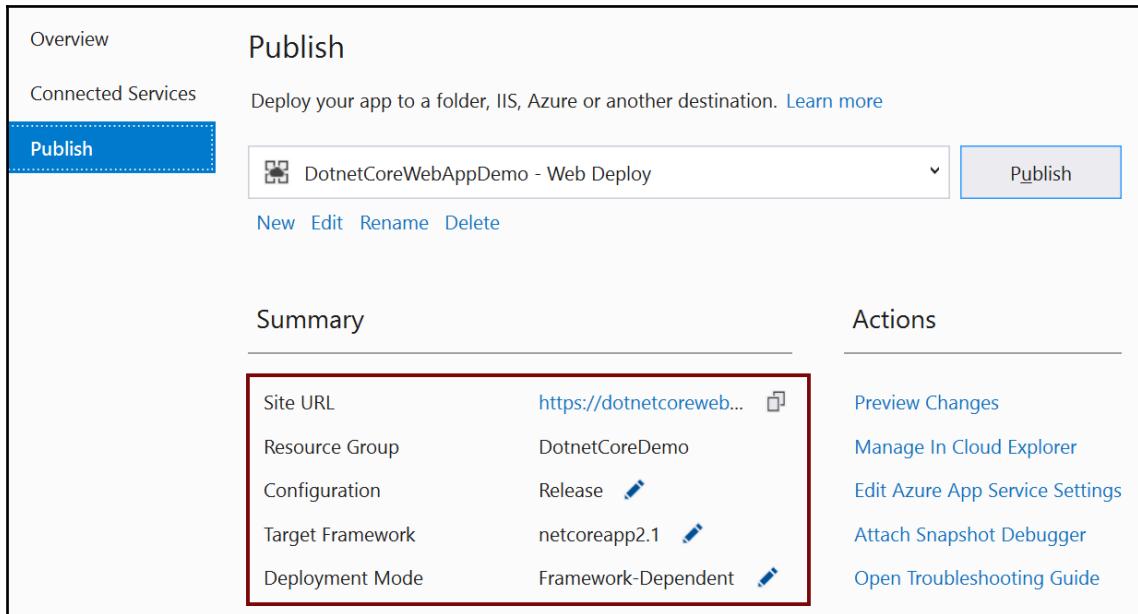
If you already have an Azure account and are already signed in with the same MSA account in Visual Studio, then the following screen will be shown, asking you to provide more details on hosting your app to Azure:



It will ask you to enter your web app name and Azure subscription details (as follows) to publish it there. Let's have a look at the following mentioned steps:

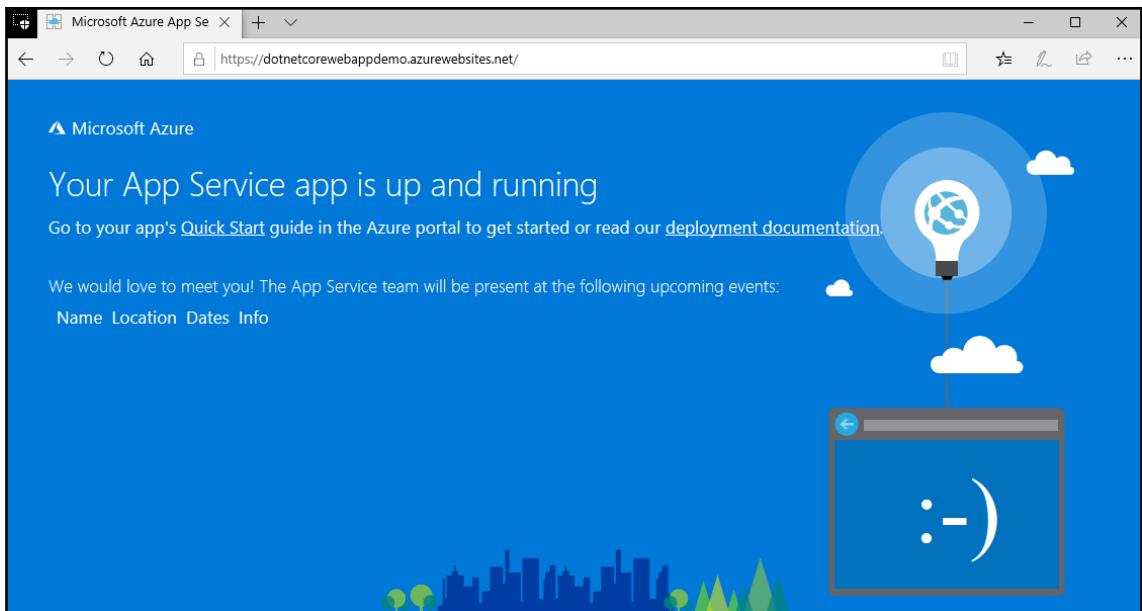
1. First, enter the web application name. This should be unique globally.
2. Select the Azure Subscription account that you want to link with.
3. Select a **Resource Group** name. If there are no entries in the list, then you need to create one by clicking the **New...** button. Fill in the required details and click **OK**.
4. Select the application hosting plan from the list. If the list is empty, then you need to click the **New...** button to create a plan and fill in the details. Then, select the correct plan that you have created.
5. Finally, click **Create** to start the final publishing and hosting process to Azure.

Once the Visual Studio starts the publishing job, it will show you the following page with details of your application hosting. Based on the web application name that you entered earlier, it will create the **Site URL** link and host to a subdomain hosted on **azurewebsites.net**:

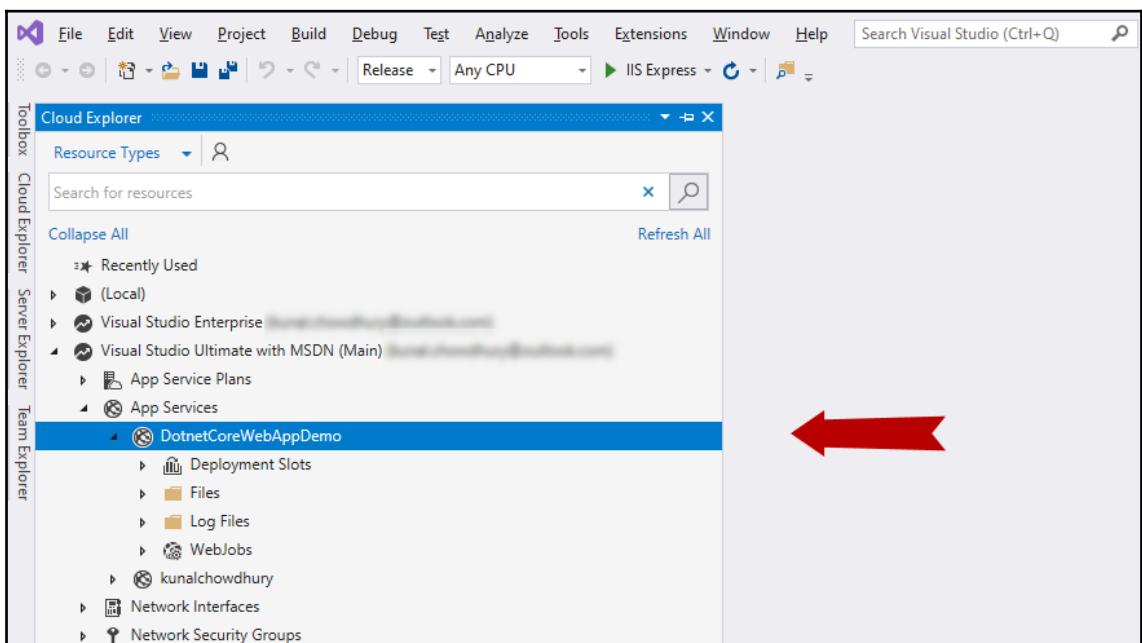


Once Visual Studio successfully publishes the application to Microsoft Azure App Service, you can click the link under **Site URL**, or you can click **Preview Changes** to launch it in your configured browser application.

When you launch the site on the browser, you will see that the site we have created runs on a public web server. As the website was created from the Visual Studio default web template, you will see the following screen:



You can also view the site details by launching **Cloud Explorer** from the Visual Studio **View** menu, which will look like this:



From **Cloud Explorer**, you will also be able to manage your web projects that are hosted on the Azure platform. Don't forget to give this a try.

Summary

In this chapter, we have learned about the basics of **.NET Core** and how to install it from the Visual Studio 2019 installer. We have discussed the commands that you must know in order to create, build, and run .NET Core applications from the CLI or console. Along with this, we have also discussed several types of application deployment models, including FDD and SCD.

As well as the core commands, we learned about how to create, build, and publish .NET Core applications using the Visual Studio 2019 IDE. We have also covered how to publish a web app to Microsoft Azure.

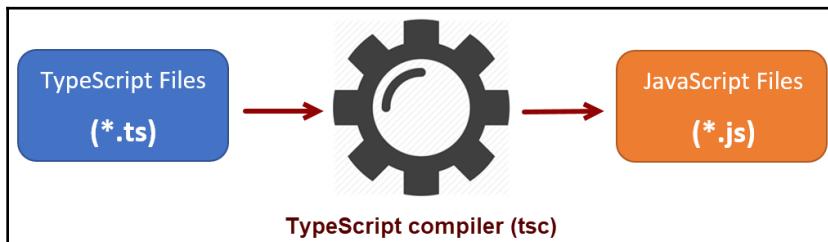
In the next chapter, we will discuss the basics of the **TypeScript** language and how to build web applications using TypeScript.

5

Web Application Development Using TypeScript

TypeScript is an open source programming language that has been developed and maintained by Microsoft and hosted in the GitHub repository. Since it is a superset of **JavaScript**, it provides optional static typing, classes, interfaces, and more to help frontend developers build web-based applications for both the client and server sides.

The name of the **TypeScript compiler** is `tsc`. When you compile a TypeScript code file (`.ts`), it outputs readable and standard JavaScript (`.js`), which can be executed by any JavaScript engine:



Microsoft released the first public version (v0.8) of TypeScript in October, 2012. In July 2014, a new TypeScript compiler was announced, and the source code was moved to GitHub. You can find it at <https://github.com/Microsoft/TypeScript>. To find out what the latest TypeScript version is, visit <https://github.com/Microsoft/TypeScript/releases>.

TypeScript simplifies JavaScript code, and thus makes it easier to read and debug code. With its static type checking, TypeScript helps you avoid the pain of bugs in your code. Since TypeScript is nothing but JavaScript with some additional features, this chapter will cover the basics of TypeScript, which will guide you through writing your code and compiling it to JavaScript so that you can use it to build web applications.

In this chapter, we will discuss the following topics:

- Setting up the development environment by installing TypeScript
- Building your first Hello TypeScript application
- Understanding the TypeScript configuration file
- Declaring variables in TypeScript
- Working with the basic datatypes
- Working with classes and interfaces

Technical requirements

To follow along with this chapter, you will need to have basic knowledge of programming and hands-on experience of using Visual Studio and/or the Visual Studio Code editor. To simplify this demonstration, we will be using the Visual Studio Code editor, and so installing the Visual Studio Code editor is required to proceed. You can download it from <https://code.visualstudio.com>.

Setting up the development environment by installing TypeScript

You can install TypeScript by using either **Node Package Manager (NPM)** or the Visual Studio installer. Let's discuss both of the ways to install it in your development environment.

Installing through NPM

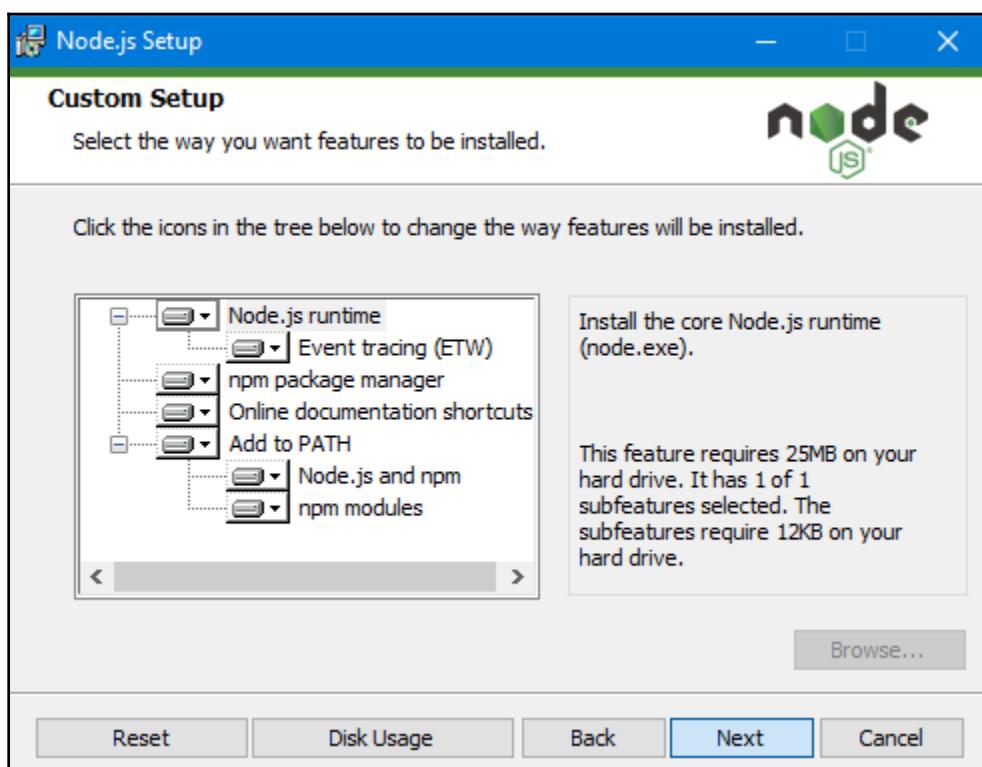
To install the TypeScript using NPM, you will need to install the **Node.js** runtime on your system. The Node.js runtime is an asynchronous event-driven JavaScript runtime that has been built on top of Chrome's V8 JavaScript engine. It uses a non-blocking **input/output (I/O)** model, which makes it simple, lightweight, and efficient enough to build scalable applications.

The Node.js installer comes with two modules: `Node` and `npm`. The `npm` module is the Node Package Manager for JavaScript. It contains a command-line client and `npm registry`, which is an online database that contains public and paid-for private packages.

If you don't have Node.js installed on your system, you can head over to <https://nodejs.org/en/> and click on the version that is tagged as **Current** to install the latest stable build of the runtime.

Alternatively, you can navigate to <https://nodejs.org/en/download/current/> and download the desired installation package that Node.js offers. From this page, you can download the 32-/64-bit versions of the Windows Installer (.msi), macOS installer, Linux binaries, and/or the source code.

Once you have downloaded the installer on your system, double-click on the installer file (.msi) to start the installation process. Then, just follow the step-by-step installation process to install Node.js on your computer. If you have gone through **Custom Setup** mode, make sure that you have selected the **Node.js runtime**, **npm package manager**, and **Add to PATH** features, as shown in the following screenshot:

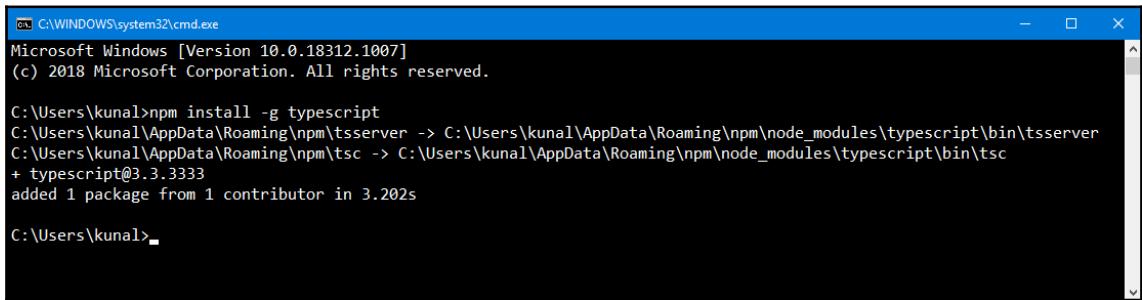


To run and complete the installation process, you will need administrative privileges. Make sure that you have the required admin rights on that system; otherwise, contact your system administrator.

Once you have installed Node.js, you can use NPM to install the TypeScript package. To install the latest version of the package from the server, open any console window (cmd.exe) and enter the following command:

```
npm install -g typescript
```

During the installation, a progress indicator will be shown in the console window to provide a visualization of the current progress. Once the installation completes, it will show the following output on the screen:



The screenshot shows a Windows Command Prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The title bar also displays 'Microsoft Windows [Version 10.0.18312.1007]' and '(c) 2018 Microsoft Corporation. All rights reserved.' The command entered is 'npm install -g typescript'. The output shows the resolution of the 'tsserver' dependency to 'C:\Users\kunal\AppData\Roaming\npm\tsserver' and the 'tsc' dependency to 'C:\Users\kunal\AppData\Roaming\npm\node_modules\typescript\bin\tsc'. It also indicates the addition of the '@3.3.333' version of 'typescript'. The total time taken is 3.202s. The command prompt ends with 'C:\Users\kunal>'.

```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 10.0.18312.1007]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\kunal>npm install -g typescript
C:\Users\kunal\AppData\Roaming\npm\tsserver -> C:\Users\kunal\AppData\Roaming\npm\node_modules\typescript\bin\tsserver
C:\Users\kunal\AppData\Roaming\npm\tsc -> C:\Users\kunal\AppData\Roaming\npm\node_modules\typescript\bin\tsc
+ typescript@3.3.333
added 1 package from 1 contributor in 3.202s

C:\Users\kunal>
```

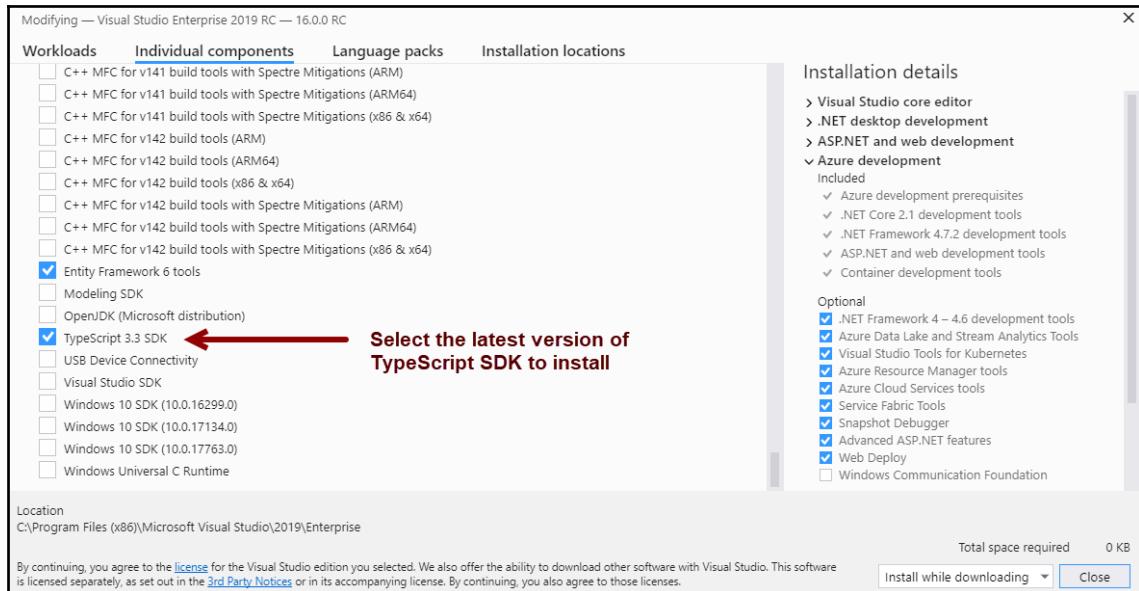
To confirm whether the installation was successful, type `tsc` in the console window (cmd.exe) and hit the *Enter* key. This will list the command-line usages of the TypeScript compiler on the console window screen.

Installing through Visual Studio 2019's installer

If you like to build TypeScript applications using Visual Studio 2019, you can directly install the compiler using the Visual Studio installer, as follows:

1. Run the Visual Studio 2019 installer and click the **Modify** button.
2. Once the installer customization window loads on the screen, navigate to the **Individual components** tab and check the desired TypeScript SDK version that you would like to install.
3. Then, click the **Modify** button to continue with the installation.

For easy reference, here's a screenshot of the view after following these steps:



If you have **ASP.NET and web development** or the **Azure development** workload already installed on your system, you won't have to explicitly install the TypeScript component, as the Visual Studio 2019 installer would have already installed that as per the default installer configuration.

Building your first Hello TypeScript application

Let's start building our first TypeScript application. For simplicity and demonstration purposes, we will be using **Visual Studio Code**. You can use Visual Studio 2019 or a plain text editor such as Notepad or Notepad++.

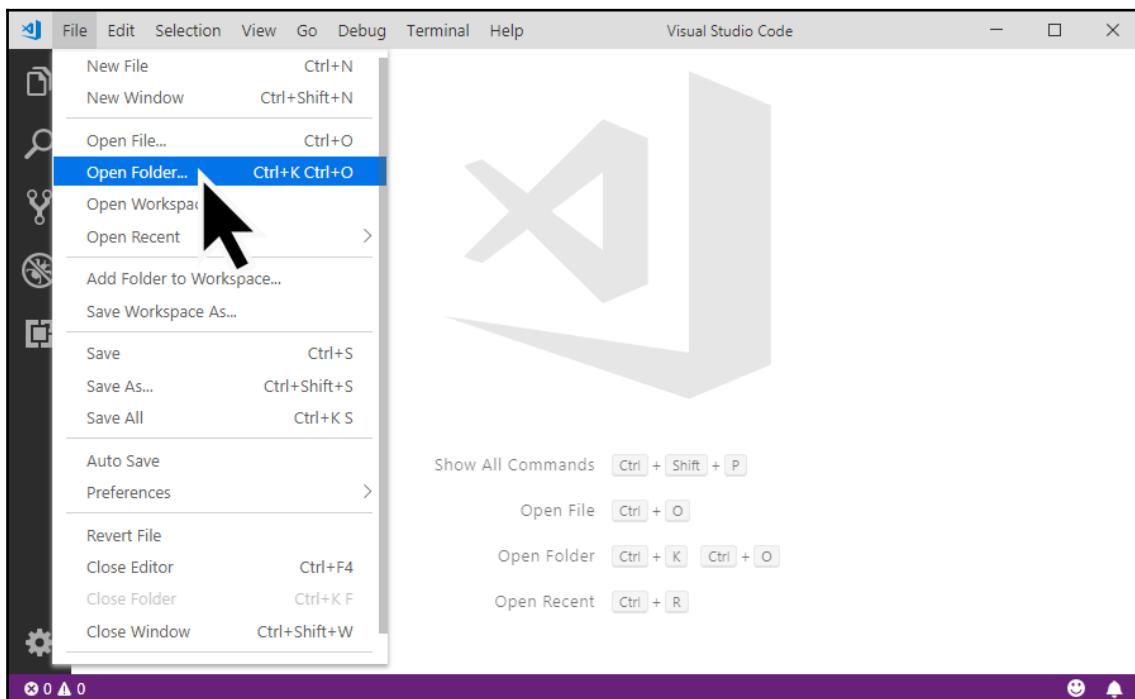
Visual Studio Code is a free, lightweight, but powerful source code editor from Microsoft, that runs on your desktop and is available for Windows, macOS, and Linux.

It comes with built-in support for JavaScript, TypeScript, Node.js, Python, PHP, Azure, Docker, and more. It has a rich ecosystem of extensions for other languages (such as C++, C#, Java, and Go) and runtimes (such as .NET and Unity).

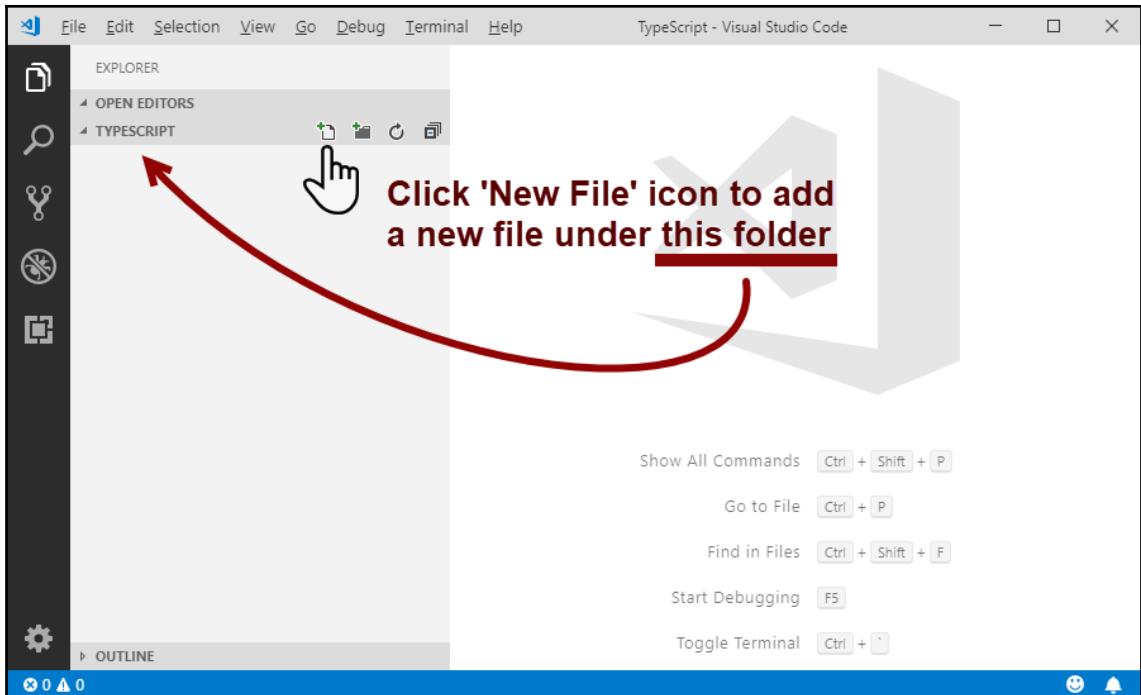
You can download the latest version of Visual Studio Code from <https://code.visualstudio.com>.

Let's start writing our first HelloTypeScript application code:

1. Once you have installed Visual Studio Code, open it. As shown in the following screenshot, click **File | Open Folder...**, or press *Ctrl + K + O* shortcut to load a blank folder as a project workspace:



2. Now, click on the **New File** icon—as shown in the following screenshot—to add a new file under the selected folder and provide it with a name (we used `HelloTypeScript.ts`). Make sure that the extension of the file is `.ts`. Based on the file extension provided, Visual Studio Code will automatically recognize it as a TypeScript file:



3. Now, inside the code editor, enter the following lines of code:

A screenshot of the Visual Studio Code interface showing a TypeScript file named 'HelloTypeScript.ts'. The code in the editor is:

```
1 // create a function that accepts a person name of type string
2 function greet(person : string) {
3     // greet the user with customized message
4     console.log(`Hello ${person}, Welcome to TypeScript`);
5 }
6
7 // call the function passing the name of the person
8 greet("reader");
```

The Explorer sidebar on the left shows the file is part of an 'OPEN EDITORS' group. The status bar at the bottom indicates the file is 3.3.3 lines long, with 8 columns, 17 spaces, and is saved in UTF-8 format.

4. Now, open a console window (`cmd.exe`) and navigate to the path where you created the `HelloTypeScript.ts` file. Inside the console window, enter the following command to compile the TypeScript file (`HelloTypeScript.ts`) that we have just created:

```
tsc HelloTypeScript.ts
```

5. This will generate the `HelloTypeScript.js` compiler output file inside the same source directory. Now, enter the `node` command (`node HelloTypeScript`) to execute the TypeScript code that we have written inside the `HelloTypeScript.ts` file. It will print the following output on the screen:

```
D:\kunal-chowdhury.com\Books\Demos\TypeScript>tsc HelloTypeScript.ts
D:\kunal-chowdhury.com\Books\Demos\TypeScript>node HelloTypeScript
Hello reader, Welcome to TypeScript
D:\kunal-chowdhury.com\Books\Demos\TypeScript>
```

When you compile a `.ts` (TypeScript source) file using the `tsc` compiler, it compiles the code and generates a JavaScript (`.js`) file. The `node` command is used from the console window to execute the JavaScript.



The `tsc` command is used for compiling TypeScript (`.ts`) files. The `node` command is used to run the output file, which is in JavaScript (`.js`) format.

Understanding the TypeScript configuration file

The **TypeScript configuration file** (`tsconfig.json`) allows you to specify the root level files that you want to include or exclude while building a TypeScript project. It also allows you to specify the compiler options that you require to compile a TypeScript project. When you place a `tsconfig.json` file in a directory, it is treated as the TypeScript project root.

For easy reference, here's a sample TypeScript configuration file:

```
{  
    "compileOnSave": true,  
    "compilerOptions": {  
        "target": "es5", /* specify ECMAScript target version */  
        "module": "system", /* specify module code generation */  
        "lib": [], /* specify library files to be included */  
        "allowJs": true, /* whether to allow JavaScript */  
        "checkJs": true, /* specify errors in .js files */  
        "jsx": "preserve", /* specify JSX code generation */  
        "removeComments": true, /* remove comments to output */  
        "noEmit": true, /* do not emit outputs */  
        "strict": true, /* enable all strict-type checking */  
        "strictNullChecks": true, /* enable strict-null check */  
        "allowUnreachableCode": false,  
        "sourceMap": true, /* generates '.map' file */  
        "rootDir": "./", /* specify the root directory */  
        "outFile": "../JS/TypeScript/HelloWorld.js",  
        "outDir": "./", /*  
    },  
    "files": [  
        "program.ts",  
        "sys.ts"  
    ],  
    "include": [  
        "src/**/*"  
    ],  
    "exclude": [  
        "node_modules",  
        "src/**/*.spec.ts"  
    ]  
}
```

Let's learn about the code, as follows:

- **compileOnSave**: The `compileOnSave` property of the configuration file allows you to specify whether you would like to automatically compile the TypeScript code files in a project. When you set the `compileOnSave` property to `true`, every time you perform a save operation on a `.ts` file, it gets compiled and generates the output.
- **compilerOptions**: The `compilerOptions` property allows you to set additional options for the TypeScript compiler. If you omit this property, the default values from the compiler will be used automatically. The `compilerOptions` property includes attributes such as `module`, `noImplicitAny`, `removeComments`, `allowUnreachableCode`, `strictNullChecks`, `outFile`, and `sourceMap`.
- **files**: The `files` property allows you to specify the relative or absolute path of the code files that you would like to ask the compiler to include.
- **include**: When you choose to include all the files in a specific directory, the `include` property is being used. You will need to specify the property using the glob wildcards pattern. If you don't specify the `files` and `include` properties, the TypeScript compiler will, by default, include all the TypeScript files. If both are specified, the compiler will include the union of the specified files.
- **exclude**: If you want to exclude a list of TypeScript files, or if you want to filter out some files that were included using the glob `include` property, you can use the `exclude` property by using the glob wildcard pattern. If you have included any files using the `files` property, the `exclude` property won't have any impact on those files.

Declaring variables in TypeScript

In any programming language, variables are used to store data/values in memory that you can refer to whenever you need to access them. It can be a local variable in which the scope is limited to the block where it has been defined, a class-level variable that can be accessed within that class, or a global variable that can be accessed from anywhere within the same application.

TypeScript also allows you to declare variables to store data/values. A variable name can contain uppercase or lowercase letters, numbers, or special characters, but can't start with a number. Special characters include the `_` (underscore) and `$` (dollar) symbols. Apart from these two special characters, other special characters are not allowed.

Declaring variables using the var keyword

Like C#, you can define variables in TypeScript using the `var` keyword. A variable can be defined in multiple ways, as shown in the following code snippet:

```
var identifier;           // var author;
var identifier = value;   // var author = "Kunal Chowdhury"
var identifier: datatype; // var author: string;
var identifier: datatype = value; // var author: string = "Kunal"
```

When you define a variable as `var identifier` without specifying the datatype and the value, the compiler assigns its datatype as `any` and sets its value as `undefined` by default.

If you are defining `var identifier = value`, although you don't specify the datatype of the variable, the compiler sets its datatype based on the value that you set. When you define `var identifier = "Kunal Chowdhury"`, the datatype will be defined as `string`; if you define it as `var identifier = 2019`, the datatype will be automatically defined by the compiler as `integer`.

If you define a variable as `var identifier: datatype`, you can only set the value of that datatype. However by default, the compiler sets its value as `undefined`, irrespective of the datatype.

When you specify both the datatype and the value as `var identifier: datatype = value`, the variable will initialize with the specified value and can only store data/values of the datatype that you defined.

Problems with using the var keyword

In TypeScript, defining variables using the `var` keyword has some problems, which causes developers to avoid it unless it's really required. Let's discuss these issues with some suitable examples.

You can define a variable inside a block and access it outside the defined scope. As a result, you may get the return value as `undefined`. Let's consider the following example to understand it:

```
function Book(publicationDate: number) {
    if (publicationDate == 2019) {
        var bookTitle = "Mastering Visual Studio 2019";
    }

    return bookTitle;
```

```
}

console.log(Book(2019)); // prints "Mastering Visual Studio 2019"
console.log(Book(2018)); // prints "undefined"
```

Another problem with defining variables using the `var` keyword is that you can define the same variable multiple times within the same code block. The TypeScript compiler won't result in any errors. Let's consider the following example in order to aid our understanding:

```
function GetBook(title: string, year: number) {
    var title = "Mastering Visual Studio 2017";
    var title = "Windows Presentation Foundation " +
        "Development Cookbook"; // no error
    if (year == 2019) {
        var title = "Mastering Visual Studio 2019"; // no error
    }
}
```

As you can see, by using the `var` keyword, you will encounter some problems that developers always face. So, what could be the right approach to declaring variables? To answer this question, continue to the next section.

Declaring variables using the `let` keyword

To overcome the problem that arises with the `var` keyword, TypeScript allows you to declare variables using the `let` keyword. Similar to defining variables using the `var` keyword, you can write the following `let` statement:

```
let title = "Mastering Visual Studio 2019"
```

The `let` statement uses block scoping, and hence a variable declared within a function or block will be limited to the same function or block. You won't be able to access it outside of that. The following code block will result in a compile time error:

```
function Book(publicationDate: number) {
    if (publicationDate == 2019) {
        let bookTitle = "Mastering Visual Studio 2019";
    }

    return bookTitle; // error: can't find name 'bookTitle'
}
```

Also, if you use the `let` keyword, you can't redeclare the same variable within the same scope. Consider the following example:

```
function GetBook(title: string, year: number) {  
    let title = "Mastering Visual Studio 2017";  
    // error: duplicate identifier 'title'  
    let title = "Windows Presentation Foundation " +  
        "Development Cookbook";  
    if (year == 2019) {  
        let title = "Mastering Visual Studio 2019"; // no error  
    }  
}
```

Here, the second reassignment to `title` results in an error as we are redeclaring it within the same scope. However, the third reassignment results in no error as its scope is different and limited to the `if` block only.

Declaring constants using the `const` keyword

TypeScript also allows you to declare constant variables. When you declare constant variables using the `const` keyword, TypeScript acts like `let`, but with a small difference—you can't change its value after initialization:

```
const author = "Kunal Chowdhury";  
author = "some other author"; // error: can't assign to 'author' because it  
is a constant
```

The preceding code will result in an error as you can't reassign a value to a constant.

Working with the basic datatypes

Every programming language provides a set of basic datatypes. TypeScript also follows that and provides datatypes such as `number`, `string`, `boolean`, `enum`, `void`, `null`, `defined`, `any`, `never`, `Array`, and `tuple`. You can declare variables using datatypes in the following fashion:

```
let identifier: datatype;  
let identifier: datatype = value;
```

In the next section, we'll learn more about TypeScript datatypes by implementing some suitable examples.

Using the number type

TypeScript allows you to define floating-point values by using the `number` type. When you define a variable of this type, you can assign any numeric values to it, including decimal, binary, hexadecimal, and octal values. Here are a few examples of defining numeric values using the `number` datatype:

```
let decimalValue: number = 100.25;           // decimal
let binaryValue: number = 0b10110010; // binary
let hexaDecimalValue: number = 0xf210b;      // hexadecimal
let octalValue: number = 0o41530;           // octal
```

To define a binary or octal number, your TypeScript version must support **ECMAScript 2015** or higher. Make sure that you are using the latest version of TypeScript.

Using the string type

Like all other programming languages, TypeScript also uses `string` datatypes to represent textual data. You can use single quotes (`'` and `'`) or double quotes (`"` and `"`) to surround the `string` value:

```
let firstName: string = "Kunal";           // using double quotes
let lastName: string = 'Chowdhury';        // using single quotes
```

TypeScript also supports **templated strings**, where you can span a string to use multiple lines and print values of embedded expressions in the form of `${expression}`. The templated strings are surrounded by backquote/backtick (``` and ```) and can be written as follows:

```
let authorName: string = "Kunal Chowdhury";
let blogURL: string = "https://www.kunal-chowdhury.com";
let message: string = `Hi, my name is ${authorName}.
I do blog at ${blogURL}. Don't forget to visit it.`;
```

You can use templated strings in your code to properly format the string using expression values.

Using the `bool` type

The `boolean` keyword is used in TypeScript to declare variables of the `bool` type. This basic datatype only accepts `true` and `false` values. Here's how to declare a `boolean` variable in TypeScript:

```
let canExecute: boolean = false;
canExecute = true;
```

Boolean types are useful when you need to check conditional values and/or define a flag.

Using the `enum` type

Enumerated datatypes are a set of numeric values with a more friendly name. You need to define an enumerated datatype using the `enum` keyword. Here's a code snippet that demonstrates how you can declare a variable of the `enum` type and use it:

```
enum MessageType {
    None,           // value is '0'
    Information,   // value is '1'
    Warning,        // value is '2'
    Error,          // value is '3'
}

let messageType: MessageType = MessageType.Information;
```

By default, the indexes of `enum` values start from 0 (zero). Just like with C#, you can manually set the values by entering the index values against its members. Consider the following two examples:

```
enum MessageType {
    None = 1,         // value is '1'
    Information,     // value is '2'
    Warning,          // value is '3'
    Error,            // value is '4'
}

enum MessageType {
    None = 1,         // value is '1'
    Information = 3, // value is '3'
    Warning = 5,      // value is '5'
    Error = 7,         // value is '7'
}
```

In the first example, you can see that the index of the first `enum` value has been set to 1. Thus, the `enum` values that follow are automatically assigned index values of 2, 3, and 4. In the second example, the index values of the `enum` values are explicitly assigned.

Using the void type

In general, when you do not want to return any value from a function, the `void` datatypes are used as the return type, as shown in the following example:

```
function showMessage(message: string) : void { ... }.
```

In TypeScript, you can also declare a variable of the `void` type, but can only assign an `undefined` or `null` value to it, as shown in the following code snippet:

```
let stability: void = undefined;
let stability: void = null;
```

As a result, you can declare variables of the `void` type, but this is not useful in real-world applications.

Using the undefined type

In TypeScript, you can use the `undefined` keyword as a datatype to store an `undefined` value. You can also assign it to a `number` or `boolean` value as it is a subtype of all other types:

```
let undefinedValue: undefined = undefined;
let undefinedNumber: number = undefined;
let undefinedBoolean: boolean = undefined;
```

If you use the `--strictNullChecks` flag, `undefined` will only be assignable to `any` and its respective type in order to avoid common errors.

Using the null type

In TypeScript, you can use the `null` keyword to declare a variable of the `null` type. When declared, you can only assign a `null` value to it. Just like `undefined`, you can also assign it to a `number` or `boolean` value as it is a subtype of all other types:

```
let nullableValue: null = null;
let nullableNumeric: number = null;
let nullableBoolean: boolean = null;
```



If a variable is assigned with `undefined`, then the variable has no value or object assigned to it. However, if a variable is assigned with `null`, then the variable is a type of an object whose value hasn't been defined.

Using the any type to declare a variable

While writing code in C#, if you are unsure about the datatype of a value due to its dynamic nature, you can use the `dynamic` keyword to declare that variable. Similarly, TypeScript also provides a datatype, called `any`, to declare a dynamic type variable.

Due to its dynamic nature, you can assign different datatypes at different points in time:

```
let dynamicValue: any = "Kunal Chowdhury";
dynamicValue = 2019;
dynamicValue = 0b100110111001;
dynamicValue = 0o411021;
dynamicValue = true;
```

If you have an array of mixed datatypes, then the `any` datatype can be used, as shown in the following code snippet:

```
let dynamicList: any[] = [ "Mastering Visual Studio 2019",
  "Kunal Chowdhury",
  2019,
  True
];
```

This is often useful when you seek input from a user or a third-party library or service.

Using the never type

A function can never return a value if it always throws an exception. In such a condition, the return datatype of that function in TypeScript is never. Here's a function declaration demonstrating the use of the never type:

```
function throwError(message: string): never {
    throw new Error(message);
}
```

In short, the `never` datatype represents a type of value that never occurs.

Using the array type

TypeScript also supports working with arrays. You can declare a variable of the array type in either of the following ways, but the second approach is a more generic way to define it:

```
// approach one:
let books: string[] = [
    "Mastering Visual Studio 2017",
    "Mastering Visual Studio 2019",
    "Windows Presentation Foundation Development Cookbook"
];

// approach two (generic way):
let books: Array<string> = [
    "Mastering Visual Studio 2017",
    "Mastering Visual Studio 2019",
    "Windows Presentation Foundation Development Cookbook"
];
```

In the first approach, you need to set the type of the elements followed by `[]` to denote an array of that element type. In the second approach, a generic array type is being used.

Using the tuple type

Tuples are used to declare the data/values more appropriately when you want to create an array with a fixed number of elements of known types, but assign them without any limit. Although the types of elements are known and don't need to be the same here, you can still define them. Consider the following example:

```
let author: [string, number] = ["Kunal Chowdhury", 2019];
let values: [string, number] = ["One", 1, "Two", 2];
```

The data of the correct type will be returned if you access an element of a tuple with a valid known index. A union type will be returned if you access an element outside the set of known indices. To understand this properly, let's consider the following example:

```
let values: [string, number] = ["One", 1];           // correct
let values: [string, number] = ["One", 1, "Two"];    // correct
let values: [string, number] = [1, "One"];          // error
let values: [string, number] = ["One", 1, 2];        // error
let values: [string, number] = ["One", 1, true];     // error
```

If you access an element outside the set of known indices, it will fail with an `error`.

Working with classes and interfaces

TypeScript supports **Object-Oriented Programming (OOP)** concepts, and thus the basic features such as classes and interfaces are available in TypeScript. A class is a blueprint for creating an object of a type, which encapsulates the data of that object.

Along with **ECMAScript 6** (more specifically, **ECMAScript 2015**), TypeScript now supports classes. Let's learn about how to define a class, its constructors, and its interfaces. We will also discuss how to define a static class and inheritance.

Defining classes in TypeScript

Just like C# and other OOP languages, you can create a class in TypeScript using the `class` keyword. Let's define a class with a few properties and methods. Consider the following example:

```
class Person {
  // properties
  FirstName: string = "Default";
  LastName: string = "User";
```

```
Age: number;
// default constructor
constructor() {
    this.FirstName = "Kunal";
    this.LastName = "Chowdhury";
    this.Age = 30;
}
// methods/functions
public getFullName(): string {
    return `${this.FirstName} ${this.LastName}`;
}
public getAge(): number {
    return this.Age;
}
}
```

If you are from a C# or Java background, you should be able to easily understand the preceding class model. Here, we have defined a class named Person. It consists of three properties (FirstName, LastName, and Age), one constructor, and two methods (getFullName(), which returns a string, and getAge(), which returns a number).

By default, all the members in a class defined in TypeScript are public.



In TypeScript, instead of defining the properties manually, you can also go for automatic property creation. In this case, the parameterized constructor plays a vital role. You will just have to define the constructor parameters as public, and the TypeScript compiler will do the rest for you to generate the properties. You don't have to define them explicitly. You can write the previous code snippet as follows:

```
class Person {
    // parameterized constructor
    constructor (public FirstName: string,
                public LastName: string,
                public Age: number) {
    }
    // methods/functions
    public getFullName(): string{
        return `${this.FirstName} ${this.LastName}`;
    }
    public getAge(): number {
        return this.Age;
    }
}
```

Once you've defined a class, you may need to create an instance of it. We'll discuss how to create an instance of a class in the next section.

Initiating an instance of a class

Just like with C# or Java, you can easily instantiate a class and access its members. To create an instance of a class, you can use the `new` operator. Here are two different ways to create class instances in TypeScript:

```
let person: Person = new Person("Kunal", "Chowdhury", 30);
console.log("Person name: " + person.getFullName());

// alternate way
let person = new Person("Kunal", "Chowdhury", 30);
console.log("Person name: " + person.getFullName());
```

The first way has the type defined for the instance variable, whereas the second way, by default, assigns the type based on the right-hand side definition.

Defining abstract classes in TypeScript

In TypeScript, you can also define a class as abstract. Although you cannot instantiate an abstract class, you can inherit it to create a derived class and create an instance of that derived class. To create an abstract class, you can use the `abstract` keyword, as shown in the following code snippet:

```
abstract class Person {
  ...
  ...
  ...
}
```

An abstract class can have abstract methods as well as non-abstract methods. An abstract method does not have an implementation body, but you must implement it in derived classes. To define an abstract method, you should use the `abstract` keyword.

As shown in the following code snippet, the `getFullName()` and `getAge()` methods are abstract methods in an abstract `Person` class:

```
abstract class Person {
  abstract getFullName(): string; // abstract method
  abstract getAge(): number; // abstract method
```

```
        toString(): string {
            return "Age of " + this.getFullName() +
        " is: " + this.getAge();
    }
}
```

Now, you should be able to create an abstract class in TypeScript and write the method implementation wherever it is necessary.

Working with inheritance in TypeScript

Inheritance is a mechanism to extend the properties of one class to another class. Since the TypeScript language supports basic object-oriented principles, you can inherit a base class and generate derived classes. To inherit a class, you will have to use the `extend` keyword. Let's learn how to do this by using a simple example:

```
class Base {
    PropertyOne: string;
    PropertyTwo: string;

    constructor(p1: string, p2: string) {
        this.PropertyOne = p1;
        this.PropertyTwo = p2;
    }
}

class Derived extends Base {
    PropertyThree: string;

    constructor(p1: string, p2: string, p3: string) {
        super (p1, p2);

        this.PropertyThree = p3;
    }

    public print() {
        console.log(`P1: ${this.PropertyOne}
                    P2: ${this.PropertyTwo}
                    P3: ${this.PropertyThree}`);
    }
}
```

Here, the `Base` class contains two properties (`PropertyOne` and `PropertyTwo`) of the `string` type. The constructor of the base class takes two parameters of the `string` type and populates the defined properties.

A new class named `Derived` extends the `Base` class. It contains a property named `PropertyThree` of the `string` type. Here, the constructor of the `Derived` class takes three parameters. The first two parameters are passed to the `Base` class using the `super` keyword, while the third parameter populates the new `PropertyThree` property.

Constructors for `Derived` classes must contain a `super` call.



The `print()` method of the `Derived` class prints the values of all three properties to the console log. Since the access modifiers of all the `Base` class members are `public`, you can access them from the `Derived` class.

Defining interfaces in TypeScript

An interface is like an abstract class with a difference—all the members are abstract. This means that an interface only contains the declaration of the methods and properties. It is the responsibility of the class that implements the interface to write the actual implementation of all the members of the interface.

TypeScript also supports interfaces and is denoted by the `interface` keyword. By default, all the members in an interface are `public`. Here's how to define an interface:

```
interface IPerson {  
    FirstName: string;  
    LastName: string;  
  
    getDetails(): string;  
}
```

In TypeScript, a class can implement an interface, and an interface can extend one or more interfaces. Not only that, but an interface can also extend a class in TypeScript. Let's discuss this in depth with some suitable examples.

Classes implementing interfaces

In TypeScript, a class can implement an interface by using the `implements` keyword. Here's an example of defining an interface and implementing it in a class:

```
interface IPerson {
    FirstName: string;
    LastName: string;

    getDetails(): string;
}

class Employee implements IPerson {
    EMP_ID: string;

    FirstName: string;
    LastName: string;

    getDetails(): string { // method implementation
        return `${this.FirstName} ${this.LastName} -
${this.EMP_ID}`;
    }
}

class Customer implements IPerson {
    CUST_ID: string;

    FirstName: string;
    LastName: string;

    getDetails(): string { // method implementation
        return `${this.FirstName} ${this.LastName} -
${this.CUST_ID}`;
    }
}
```

An interface extending another interface

An interface can be extended from another interface by using the `extends` keyword. As shown in the following code snippet, the `ICricket` and `IFootball` interfaces extend the `ISports` interface:

```
interface ISports {
    ...
    ...
}
```

```
interface ICricket extends ISports {  
    ...  
    ...  
}  
  
interface IFootball extends ISports {  
    ...  
    ...  
}
```

Interfaces extending multiple interfaces

It is also possible to extend multiple interfaces. The previous example can be rewritten by extending multiple interfaces. Consider the following example, where the `ICricket` and `IFootball` interfaces extend the `ISports` and `IEvents` interfaces:

```
interface ISports {  
    ...  
    ...  
}  
  
interface IEvents {  
    ...  
    ...  
}  
  
interface ICricket extends ISports, IEvents {  
    ...  
    ...  
}  
  
interface IFootball extends ISports, IEvents {  
    ...  
    ...  
}
```

Interfaces extending classes

Unlike other programming languages, TypeScript allows you to extend an interface from a class. In such cases, only the declarations of the class members are inherited by the interface, without their implementations:

```
class Cricket {  
    play(): void {  
        ...  
    }
```

```
...
}

getUmpireDetails(): IPerson {
...
...
}
}

class Football {
    play(): void {
        ...
        ...
    }

    getRefereeDetails(): IPerson {
...
...
    }
}

interface ISports extends Cricket, Football {
    play(): void; // common to both Cricket and Football class
    getUmpireDetails(): IPerson; // inherits from Cricket class
    getRefereeDetails(): IPerson; // inherits from Football class
}
```

We have successfully learned about how interfaces can be defined and used in TypeScript.

Summary

In this chapter, we learned about the basics of TypeScript, which included setting up the development environment with Node.js, TypeScript SDK, and building our first Hello TypeScript application. Later, we discussed the TypeScript configuration file (`tsconfig`), as well as various ways to declare variables, different datatypes, classes, and interfaces.

Now, you should be able to write code in TypeScript and compile it to JavaScript (`.js`) files, which you can then call from your web application.

In the next chapter, we are going to learn about **NuGet packages**, which is a free and open source package manager designed for the Microsoft development platform.

6

Managing NuGet Packages

NuGet is the free and open source package manager for the Microsoft development platform. It was first introduced in 2010 and has now evolved into a larger ecosystem for the platform. Starting with Visual Studio 2012, it comes preinstalled as a Visual Studio extension.

You can create packages of your libraries and distribute them to the **NuGet package gallery** (<http://www.nuget.org/packages>) or to your local NuGet store, which, when published, can be downloaded again using the **NuGet Package Manager** tool inside Visual Studio. As the NuGet packages are becoming more popular to build your own packages/components and distribute it with others, it's now important to learn how to create, publish and manage these packages.

In this chapter, we will cover the following topics:

- Overview of NuGet Package Manager
- Creating a NuGet package library for .NET Framework
- Publishing a NuGet package to the NuGet store
- Managing your NuGet packages

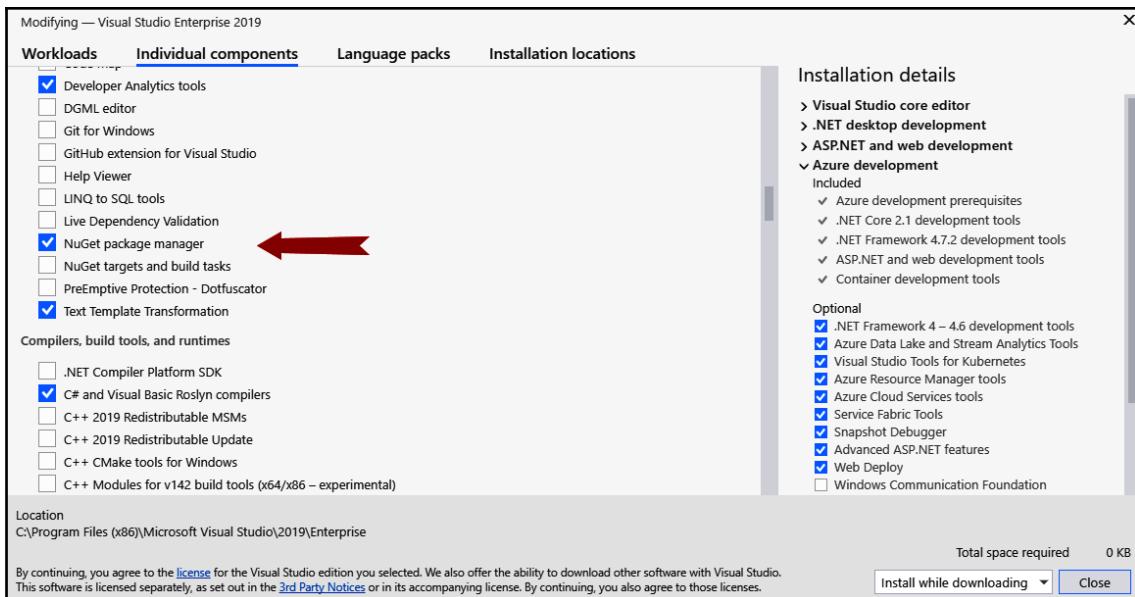
Technical requirements

To embark on this chapter, you will need to have Visual Studio 2019 installed on your system. You will also need the NuGet Package Manager tool, which you can install by running Visual Studio Installer. A basic understanding of the IDE and the C# language is also recommended.

Overview of NuGet Package Manager

A NuGet package is a .nupkg file that contains all the components needed to reference an external library in a project. Typically, this component consists of one or more DLLs and the metadata describing the package. It can consist of other components, source code, and debug symbols, but the debug symbols are very rarely found.

When you install any of the Visual Studio 2019 workloads, by default, they install the support for NuGet along with the NuGet Package Manager tool, which you can find listed under the **Individual components** tab of the installer. If you uninstall it, all the dependent workloads will be automatically uninstalled from your system:

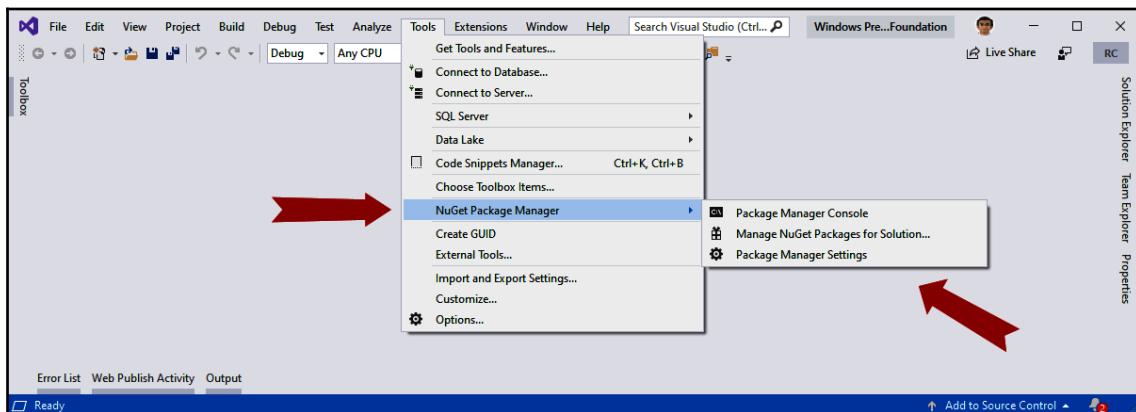


NuGet Package Manager comes with both console and GUI mode, integrated as part of the Visual Studio IDE. As the GUI interface provides you with easier access to searching, installing, updating, and uninstalling a package, it is always preferable and, most of the time, a developer interacts with it while using any library package through NuGet.

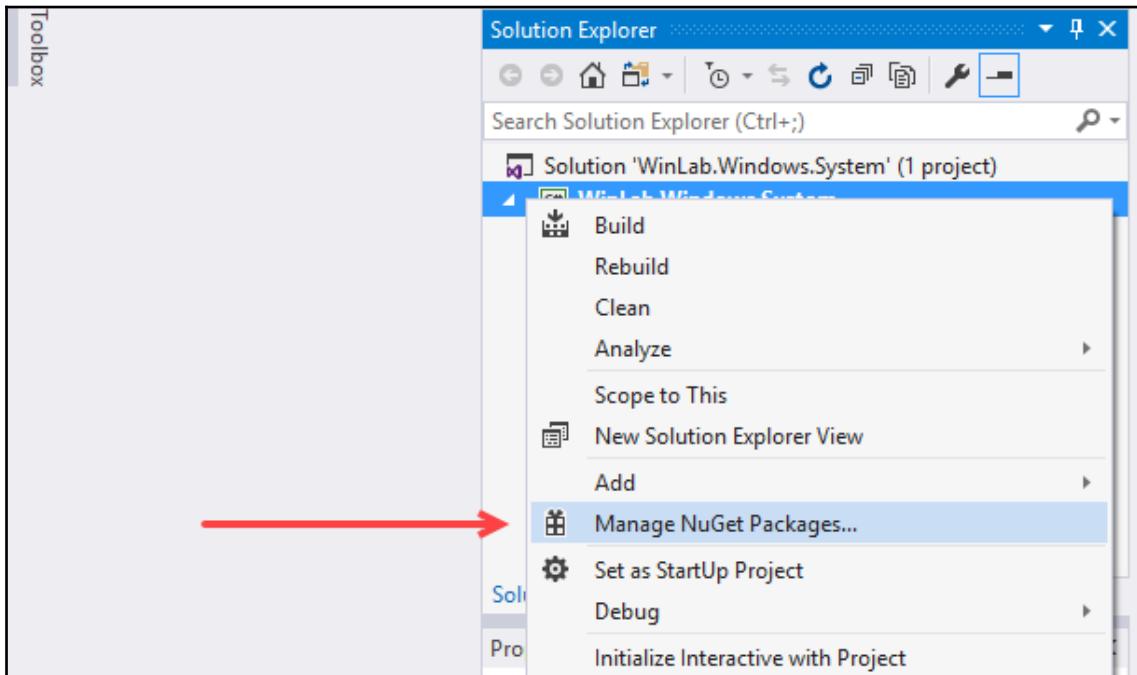
When you install a NuGet package, the following things happen:

1. If you are using .NET Framework, the package gets downloaded in the packages folder under your solution directory. If you are using .NET Core, the package gets downloaded to the %UserProfile%\nuget\packages folder.
2. Any other NuGet packages that this package is dependent on are also downloaded automatically.
3. The references to the package content (that is, the DLL references) are automatically added to the project.

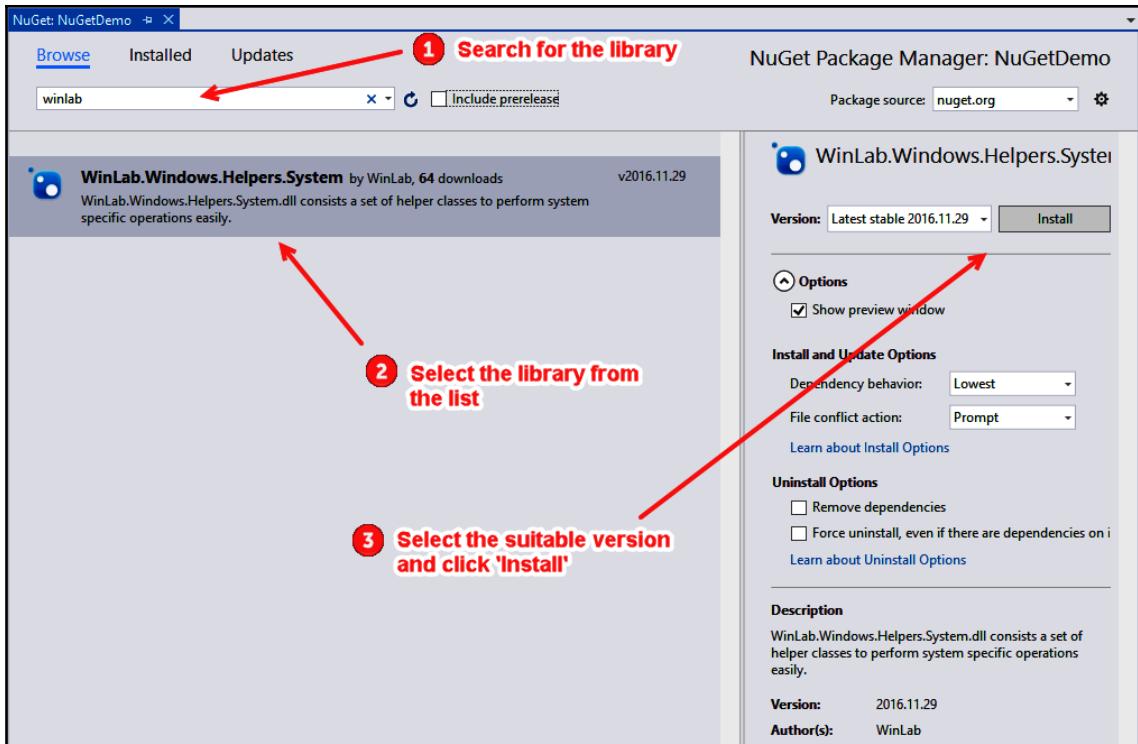
To open NuGet Package Manager, click **Tools** | **NuGet Package Manager**, and then select **Package Manager Console** for a console window or **Manage NuGet Packages for Solution...** for a GUI window:



Alternatively, you can right-click on the project/solution to launch the NuGet Package Manager GUI from the context menu entry:

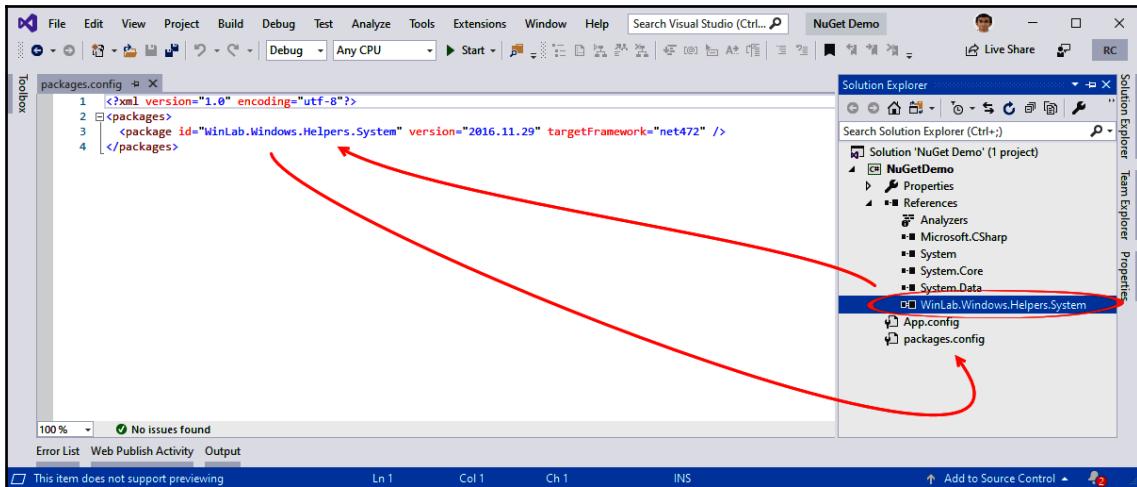


In the **Package Manager** window, search for the library that you want to install. It will give you a list of packages based on your search term. Select the one that you want to install, select the suitable version, and then click **Install** to continue:



A dialog will pop up, showing you the list of packages (including dependent packages) that it's going to install. Review the changes and click **OK** to start the download process.

Once downloaded, it will add the assembly references to your project. If you are using .NET Framework, an entry of the assembly reference will be added to the packages.config file under your project directory:



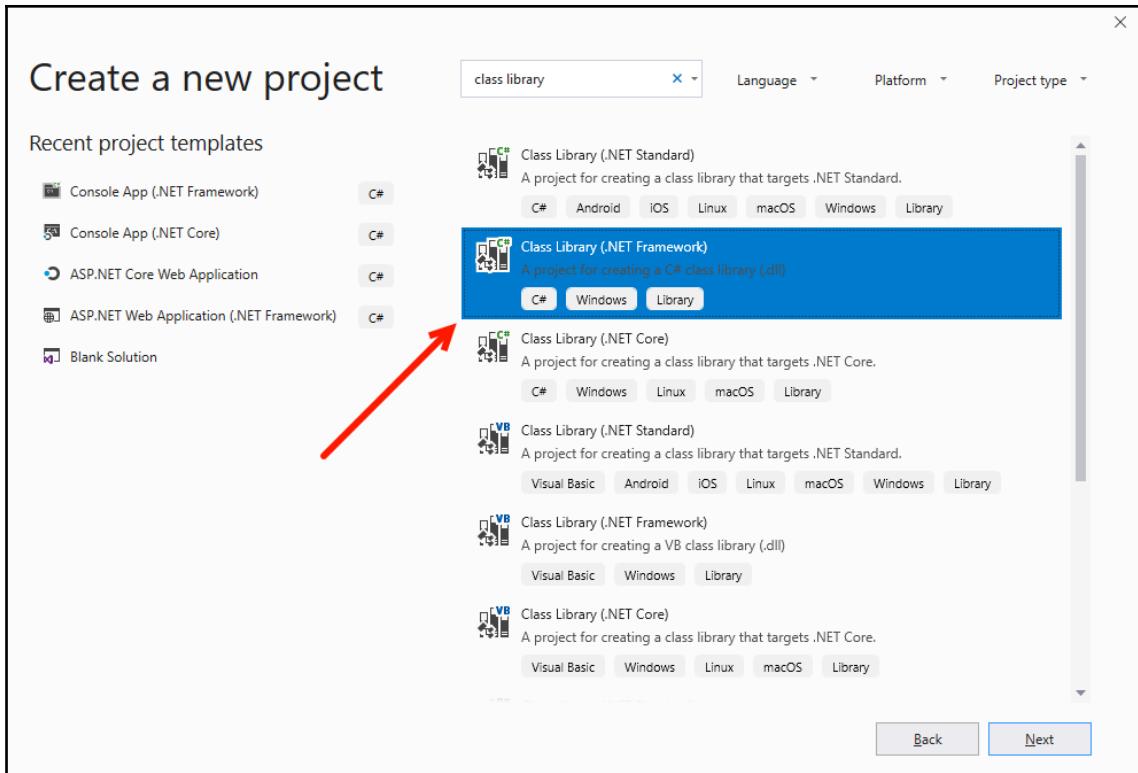
If you are using .NET Core, there will be a slight difference. You won't see the packages.config file there, but an entry to the assembly reference is added to the project file itself (.csproj if it's a C# project). Check it out for yourself by editing the .NET Core project file in Notepad.

Creating a NuGet package library for .NET Framework

As we have discussed the basics of the NuGet package, let's start creating our first NuGet package for libraries targeting .NET Framework. We will use the NuGet **Command-Line Interface (CLI)** to build the package for class libraries targeting .NET Framework. The NuGet CLI tool is a single .exe file, which you can download from the NuGet site, directly from <https://www.nuget.org/nuget.exe>.

First, let's create a class library project, which we are going to pack later:

1. Open your Visual Studio 2019 IDE and create a new project by selecting the **Class Library (.NET Framework)** from the available templates. Fill in all the required metadata of the project and click **Create**:



- Once the project is created, add a few classes and code inside it, as per your requirements. Let's add a class file called `LogHelper.cs` and add the following code to it:

```
namespace NuGetDemoLibrary
{
    public class LogHelper
    {
        public static void Log(string message)
        {
            Console.WriteLine(message);
        }
    }
}
```

Now, build the project in **Release** mode and ensure that there are no errors. As our demo class library (`NuGetDemoLibrary.dll`) is ready, it's time for us to create the NuGet package.

Creating the metadata in the nuspec file

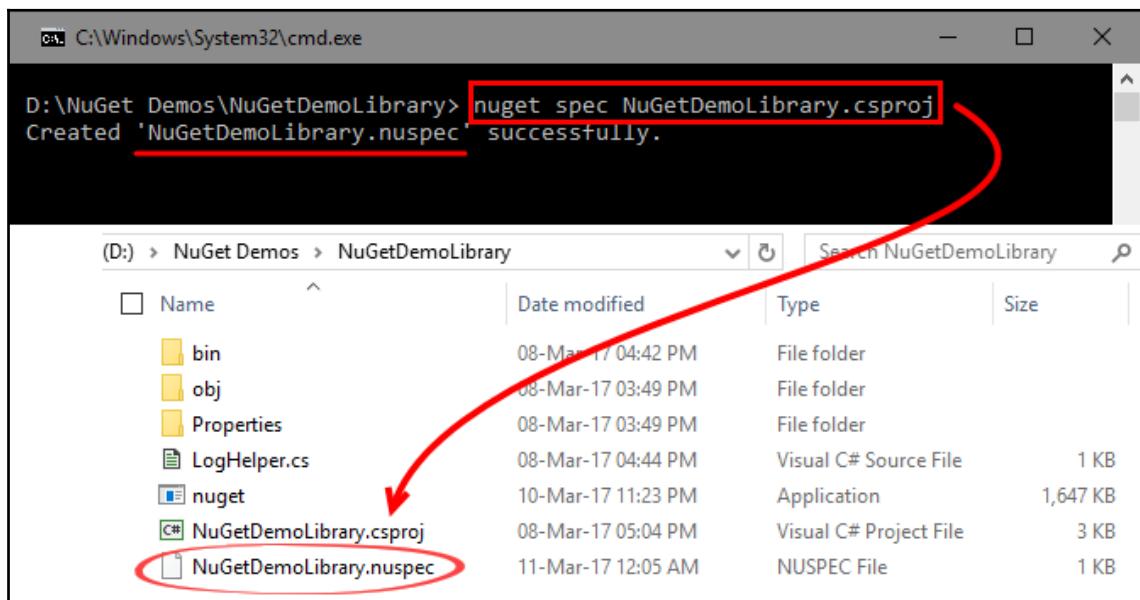
Go to the root directory of your project and copy the downloaded nuget.exe CLI file in there. Open a console window, navigate to the project directory, and enter either of the following commands to create the nuspec file:

```
nuget spec  
nuget spec NuGetDemoLibrary.csproj  
nuget spec bin\Release\NuGetDemoLibrary.dll
```

For the previous NuGet commands, please note the following:

- If you use the first command, `nuget spec`, it will create the nuspec file based on the project files that are in the directory where it was executed.
- `nuget spec <ProjectFilePath>` will only create the nuspec file for the project that you specified.
- If you use a specific DLL in this manner, `nuget spec <DllFilePath>`, the nuspec file will be created based on the specified DLL. This command is recommended as it creates the metadata information of the nuspec file from the DLL metadata.

Have a look at the following screenshot to aid your understanding of it:



This will create the NuGetDemoLibrary.nuspec file with default values:

```
<?xml version="1.0"?>
<package>
  <metadata>
    <id>$id$</id>           → Must be unique, globally in NuGet Store
    <version>$version$</version>
    <title>$title$</title>
    <authors>$author$</authors>
    <owners>$author$</owners>
    <licenseUrl>http://LICENSE_URL_HERE_OR_DELETE_THIS_LINE</licenseUrl>
    <projectUrl>http://PROJECT_URL_HERE_OR_DELETE_THIS_LINE</projectUrl>   → optional
    <iconUrl>http://ICON_URL_HERE_OR_DELETE_THIS_LINE</iconUrl>
    <requireLicenseAcceptance>false</requireLicenseAcceptance>
    <description>$description$</description>
    <releaseNotes>Summary of changes made in this release of the package.</releaseNotes>
    <copyright>Copyright 2017</copyright>
    <tags>Tag1 Tag2</tags>
  </metadata>
</package>
```

If you create the nuspec file from the DLL, most of the metadata entries will have values picked from the generated DLL. As we have created the nuspec file from the project, it has a placeholder field against most of the entries. You must replace these default values before generating the package.

The `id` field must be unique across the NuGet global store or the gallery where you are going to host the package. Generally, the `id` field gets populated from the `AssemblyName` field of the DLL. Perform a check in the store or the gallery before assigning this field.

`licenseUrl`, `projectUrl`, and `iconUrl` are optional. If you don't want to input these links, you can remove the entire lines.

NuGet uses semantic versioning, where it has three parts for the `version` field in the `Major.Minor.Patch` convention. Hence, incorporate a three-field version number against it. Whenever you upload a new version to NuGet, it decides that this is the latest stable version.

If you are not releasing a final version, you can inform NuGet to mark it as a prerelease version by suffixing the `-prerelease` switch, or by adding `-alpha`, `-beta`, or `-gamma` to the version field. Technically, you can add any string there, but it is always preferable to use a meaningful convention. For example, if your build version is `4.5.2` and you want to set it as a prerelease beta version, change the value of the version field to `4.5.2-beta`.



Note that NuGet prioritizes the version field in reverse chronological order. Thus, `-beta` will get a higher preference than `-alpha` and the final release version will get the highest preference automatically.

Fill in all the other fields mentioned under the `metadata` tag inside the `nuspec` file and save it. Now, you need to specify which files you are going to bundle and which folder you want to place them in when they have downloaded from NuGet. Generally, when you download files from NuGet, they are placed inside the `package` folder of your solution directory and are categorized by package type. Here, you can also define a framework-specific folder path, which we are going to discuss later, in the *Building NuGet packages for multiple .NET Frameworks* section of this chapter.

Now, let's add the output of our class library project and mark it to place under the `lib` folder of the `target` directory. Enter the following XML tag inside the `package` element with the path of the precompiled binary assembly:

```
<files>
  <file src="bin\Release\NuGetDemoLibrary.dll"
        target="lib\NuGetDemoLibrary.dll"/>
</files>
```

Let's update the details and save the file. The resultant output of our `.nuspec` file will look like this:

```
<?xml version="1.0"?>
<package>
  <metadata>
    <id>Demo.Packt.Kunal.NuGetDemoLibrary</id>
    <version>1.0.0-alpha</version>
    <title>NuGet Demo Library</title>
    <authors>Kunal Chowdhury</authors>
    <owners>WinLab</owners>
    <description>This is for demo purpose only</description>
    <releaseNotes>Change 1, Change 2, ...</releaseNotes>
    <copyright>Copyright 2017</copyright>
    <tags>Demo, Packt</tags>
  </metadata>
  <files>
```

```
<file src="bin\Release\NuGetDemoLibrary.dll"
      target="lib\NuGetDemoLibrary.dll"/>
</files>
</package>
```

You can add multiple precompiled assemblies inside the package by defining them inside the `<files>...</files>` element:

```
<files>
  <file src="Assembly-1.dll" target="lib\Assembly-1.dll"/>
  <file src="Assembly-2.dll" target="lib\Assembly-2.dll"/>
  <file src="Assembly-3.dll" target="lib\Assembly-3.dll"/>
</files>
```

You can also use wildcards in the `<files>` section, as follows:

```
<files>
  <file src="*.dll" target="lib " />
</files>
```

As we have created the NuGet metadata, let's move on to the next section and build a NuGet package.

Building the NuGet package

As our nuspec file is now ready, it's time for us to create the NuGet package from the nuspec file. Use the `nuget pack` command, followed by the nuspec filename, to start creating the project. Enter the following command in the console window:

```
nuget pack NuGetDemoLibrary.nuspec
```

This is what we get in the command window:

```
D:\NuGet Demos\NuGetDemoLibrary> nuget pack NuGetDemoLibrary.nuspec
Attempting to build package from 'NuGetDemoLibrary.nuspec'.
Successfully created package 'D:\NuGet Demos\NuGetDemoLibrary\Demo.Packt.Kuna.1.NuGetDemoLibrary.1.0.0-alpha.nupkg'.

WARNING: 1 issue(s) found with package 'Demo.Packt.Kunal.NuGetDemoLibrary'.
Issue: Assembly not inside a framework folder.
Description: The assembly 'lib\NuGetDemoLibrary.dll' is placed directly under 'lib' folder. It is recommended that assemblies be placed inside a framework-specific folder.
Solution: Move it into a framework-specific folder. If this assembly is targeted for multiple frameworks, ignore this warning.
```

Name	Date modified
bin	08-Mar-17 04:42 PM
obj	11-Mar-17 11:15 AM
Properties	08-Mar-17 03:49 PM
<input checked="" type="checkbox"/> Demo.Packt.Kunal.NuGetDemoLibrary.1.0.0-alpha.nupkg	11-Mar-17 12:40 PM
LogHelper.cs	08-Mar-17 04:44 PM
nugget	10-Mar-17 11:23 PM
NuGetDemoLibrary.csproj	08-Mar-17 05:04 PM
NuGetDemoLibrary.nuspec	11-Mar-17 12:24 PM

This command will create a NuGet package file with the .nupkg extension on the same directory. The filename will be a combination of the ID and the version that we specified in the .nuspec file. In our case, it generates the Demo.Packt.Kunal.NuGetDemoLibrary.1.0.0-alpha.nupkg file, as shown in the preceding screenshot.

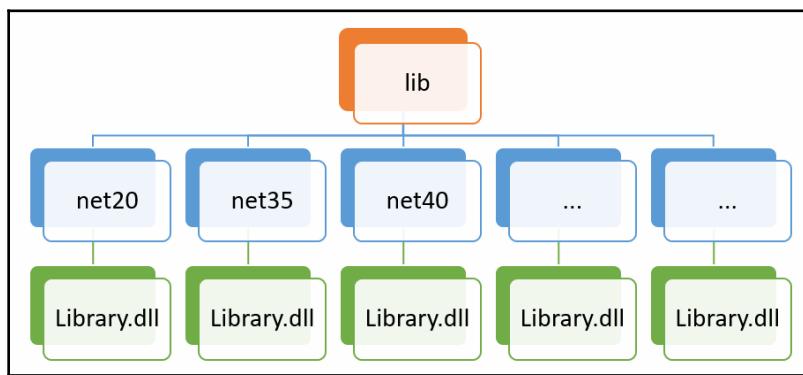
You will also see a warning message on the screen, which says one issue has been found, as the assembly is not inside a framework folder. As we have placed the DLL under the lib folder of the target location directly, it generated this error. NuGet recommends placing the assemblies inside a framework-specific folder. You can ignore this if this assembly is targeted for multiple frameworks. But if you want to build multiple binaries for different frameworks with the same name and embed them in the same package, continue reading the next section.

Building NuGet packages for multiple .NET Frameworks

NuGet supports multiple .NET Framework-targeting platforms. If your binary targets a specific .NET Framework, you can ask NuGet to handle it accordingly by specifying these platforms in the nuspec file—for example, your binary might support only .NET Framework 3.0, or can support any version higher than .NET 3.0.

To do this, NuGet uses a proper directory structure under the `lib` folder, categorized by the target framework in an abbreviated format. For example, for a .NET 2.0 targeted library, place it inside a subfolder named `net20`; for a .NET 3.5 targeted library, place it inside a subfolder named `net35`. You can find a list of supported frameworks and their abbreviated codes at: <http://bit.ly/nuget-supported-frameworks>.

When you install the assembly from a NuGet package, first, it checks for the target .NET Framework version of the project where you are going to install the package, and then it selects the correct assembly version from the appropriate subfolder under the `lib` folder and adds it as a reference. Here's a structural representation of the `lib` folder:



Now, let's modify our existing `nuspec` file to add support for framework-specific assemblies. Since our project targets .NET 4.0 and above, let's specify it in the `nuspec` file. Open the existing `.nuspec` file and modify the target of our assembly file to deploy for .NET 4.0 and above. Here's the modified version of our `nuspec` file:

```
<?xml version="1.0"?>
<package>
  <metadata>
    <id>Demo.Packt.Kunal.NuGetDemoLibrary</id>
    <version>1.0.0-alpha</version>
    <title>NuGet Demo Library</title>
```

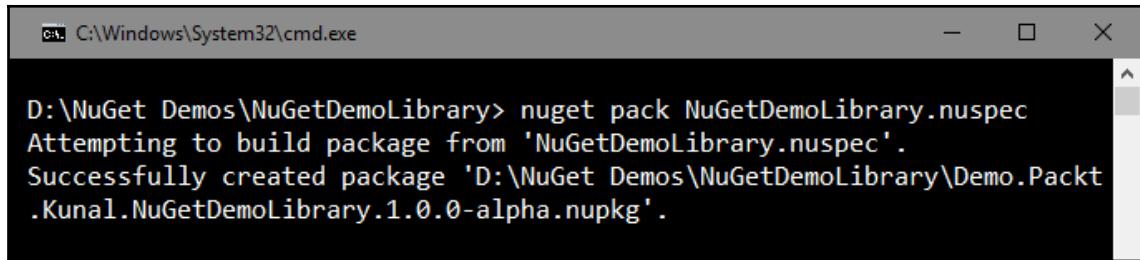
```
<authors>Kunal Chowdhury</authors>
<owners>WinLab</owners>
<description>This is for demo purpose only</description>
<releaseNotes>Change 1, Change 2, ...</releaseNotes>
<copyright>Copyright 2017</copyright>
<tags>Demo, Packt</tags>
</metadata>
<files>
    <file src="bin\Release\NuGetDemoLibrary.dll"
          target="lib\net40\NuGetDemoLibrary.dll"/>
</files>
</package>
```

Here, you can see that the target of the assembly has been set to `lib\net40\NuGetDemoLibrary.dll`. The `net40` subfolder under the `lib` root defines that the assembly can be deployed in projects targeted for .NET Framework 4.0 and above.

Now, open the console window in that directory and use the following `nuget pack` command to create/update the package:

```
nuget pack NuGetDemoLibrary.nuspec
```

The output for this is as follows:



A screenshot of a Windows Command Prompt window titled "cmd C:\Windows\System32\cmd.exe". The window shows the command "nuget pack NuGetDemoLibrary.nuspec" being run from the directory "D:\NuGet Demos\NuGetDemoLibrary". The output indicates that the package is being built from the specified nuspec file, and it successfully creates a package named "Demo.Packt.Kunal.NuGetDemoLibrary.1.0.0-alpha.nupkg".

```
D:\NuGet Demos\NuGetDemoLibrary> nuget pack NuGetDemoLibrary.nuspec
Attempting to build package from 'NuGetDemoLibrary.nuspec'.
Successfully created package 'D:\NuGet Demos\NuGetDemoLibrary\Demo.Packt.Kunal.NuGetDemoLibrary.1.0.0-alpha.nupkg'.
```

This time, you will see that no warning messages have been generated, as we have specified the correct target framework.

You can also append a dash and add a profile name to the end of the folder to define a target-specific framework profile. There are currently four profiles available and supported by NuGet:

- client for the client profile
- full for the full profile
- wp for Windows Phone
- cf for the compact framework

For example, to target the .NET Framework 4.0 full profile, place your assembly in a folder named `net40-full`; to target the .NET Framework 4.0 client profile, place your assembly in `net40-client`.

Building a NuGet package with dependencies

When you create a NuGet package, you can specify dependencies that your package should refer to and download automatically. You can define it using the `<dependency>` tag inside the `<dependencies>` tag under `<metadata>`. When installing or reinstalling the package, NuGet, by default, downloads the exact version of the dependency specified in the `nuspec` file:

```
<?xml version="1.0"?>
<package>
    <metadata>
        <id>Demo.Packt.Kunal.NuGetDemoLibrary</id>
        <version>1.0.0-alpha</version>
        <title>NuGet Demo Library</title>
        <authors>Kunal Chowdhury</authors>
        <owners>WinLab</owners>
        <description>This is for demo purpose only</description>
        <releaseNotes>Change 1, Change 2, ...</releaseNotes>
        <copyright>Copyright 2017</copyright>
        <tags>Demo, Packt</tags>
        <dependencies>
            <dependency id="WinLab.Windows.Helpers.System"
                version="2016.11.29"/>
        </dependencies>
    </metadata>
    <files>
        <file src="bin\Release\NuGetDemoLibrary.dll"
            target="lib\net40\NuGetDemoLibrary.dll"/>
    </files>
</package>
```

You can define dependencies to support a minimum version, a maximum version, a version range, or an exact version when downloading via a package. In general, developers only specify a minimum version of a referenced dependency, but any of them can be possible.

If you specify `version="1.0"` for a dependency, the `nuspec` file marks it as a minimum required version. When you specify `version="[1.0]`", the `nuspec` file marks it as an exact version match. You can also define a prerelease version while defining a dependency (such as `version="1.0-prerelease"`).



Please note that a stable release version of a package cannot have a prerelease dependency.

Here is a table that describes how you can specify a minimum version, a maximum version, or a version range when defining a dependency in NuGet:

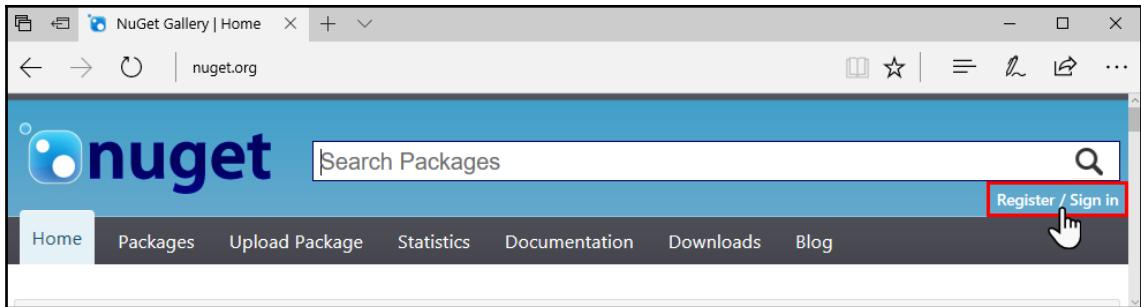
Notation	Description
1.0	Minimum version, inclusive
(1.0,)	Higher than the minimum version specified
[1.0]	Exact version
(,3.0]	Maximum version, inclusive
(,3.0)	Lesser than the maximum version specified
[1.0,3.0]	Version ranging between 1.0 and 3.0, inclusive
(1.0,3.0)	Version higher than 1.0, but less than 3.0
[1.0,3.0)	At least version 1.0, but less than 3.0
(1.0,3.0]	Version higher than 1.0, but a maximum version of 3.0

As we have learned how to create and build NuGet packages, let's now move on to the next section to learn how to publish a NuGet package to the NuGet store.

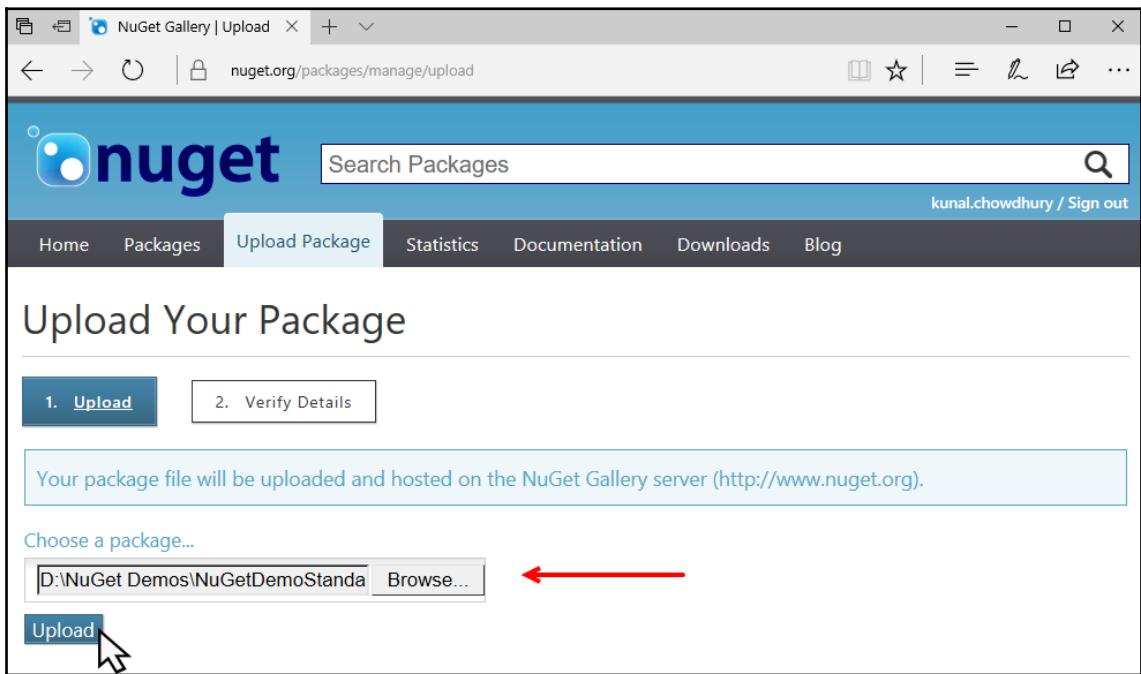
Publishing a NuGet package to the NuGet store

Once you are done with your class library and have generated the NuGet package and tested it locally, you may want to push it to the NuGet gallery for public availability so that others can find it, and install and use your library from the NuGet store.

To begin publishing a NuGet package, open any browser window and navigate to <https://www.nuget.org>. If you don't have an account, you need to register in the portal; otherwise, you can sign in with your account credentials. The **Register / Sign in** link can be found in the top-right corner of the website:



Once you are signed in to NuGet, click **Upload Package**. Under the **Choose a package...** label, click on the **Browse...** button to select the .nupkg file that you want to upload and publish to the NuGet store. Click the **Upload** button to continue:



Once the package has been uploaded, verify all the details. You may want to change the metadata information from this screen. Click on **Submit** to finally publish it.

If you have uploaded any prerelease packages, the site will inform you about the package. Also, a label will be displayed, reading **This package has not been indexed yet**, as it takes some time to index your package and show it in the search results:

The screenshot shows a NuGet package page for 'Demo.Packt.Kunal.NuGetDemoStandardLibrary'. It highlights that the package is a prerelease version and has not been indexed yet. A command-line interface shows the installation command.

This is a prerelease version of Demo.Packt.Kunal.NuGetDemoStandardLibrary.

This package has not been indexed yet. It will appear in search results and will be available for install/restore after indexing is completed.

Demo.Packt.Kunal.NuGetDemoStandardLib...

1.0.0-prerelease

Demo Library

To install Demo.Packt.Kunal.NuGetDemoStandardLibrary, run the following command in the [Package Manager Console](#)

```
PM> Install-Package  
Demo.Packt.Kunal.NuGetDemoStandardLibrary -Pre
```

Once the package has been uploaded and published, you will be able to download and install the package directly from NuGet Package Manager, pointing your gallery to `nuget.org` as the package source. You can also use the **Package Manager Console** to download it via the command line.

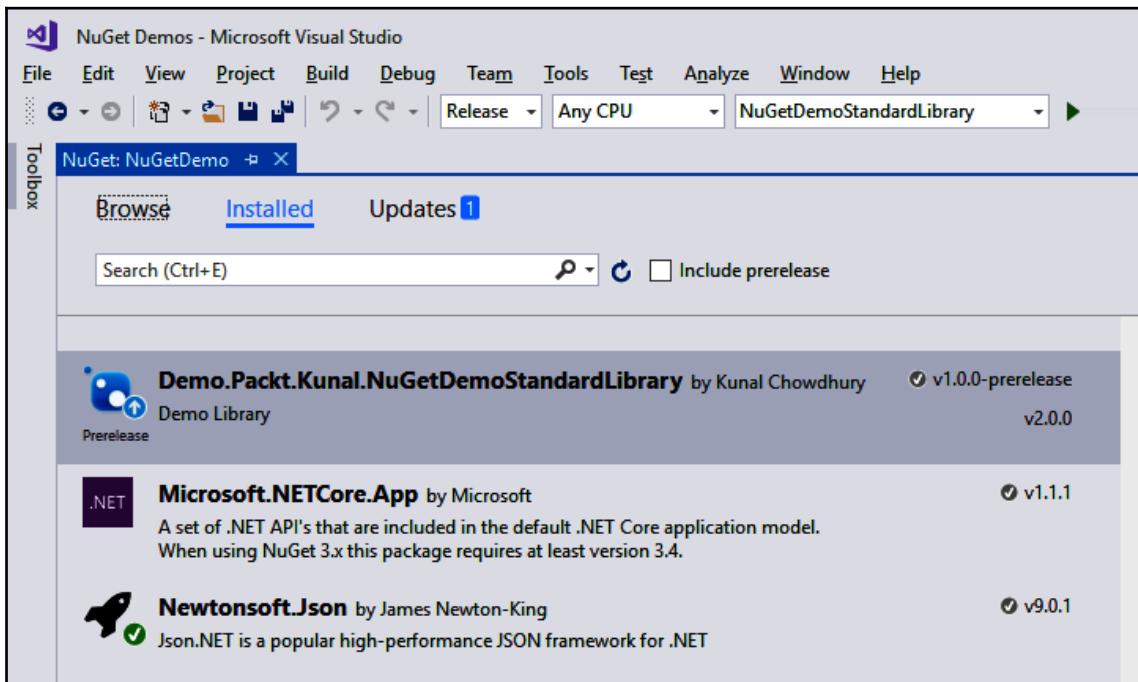
After reading this part of the chapter, you should now be familiar with publishing NuGet packages to the NuGet store. If you have an in-house store or a third-party store, you can publish your packages in a similar way.

Managing your NuGet packages

You can manage all your NuGet packages from the NuGet Package Manager, which you can invoke from the **Tools** menu of Visual Studio: **NuGet Package Manager | Manage NuGet Packages for Solution....**

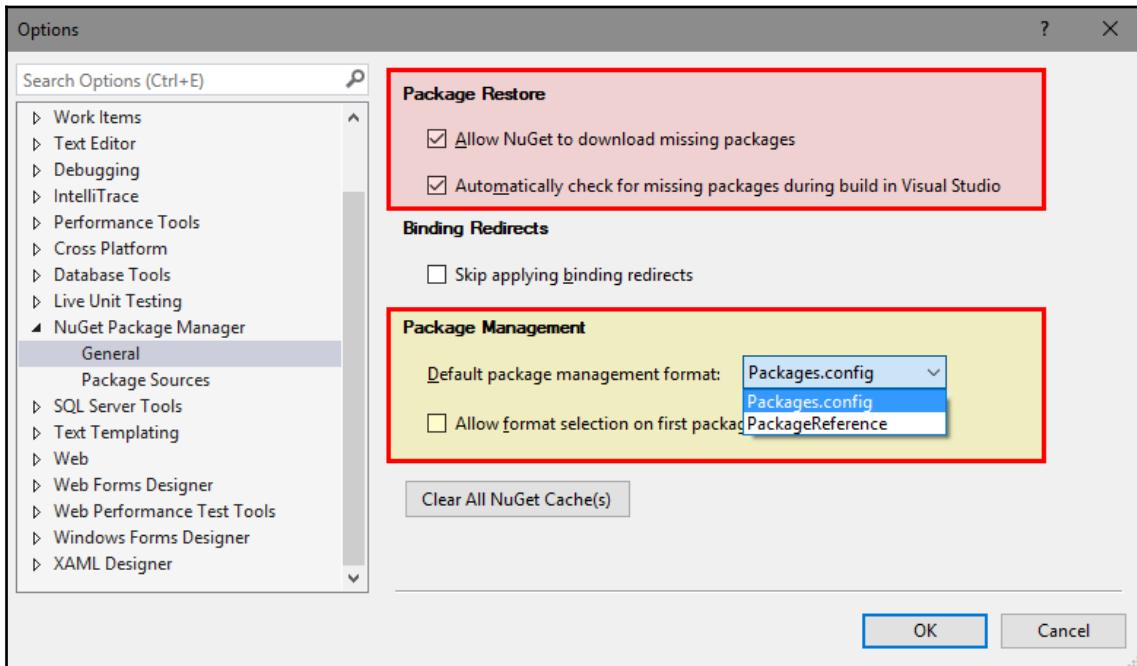
All the installed packages will be listed under the **Installed** tab, whereas the packages that have new updates on the server will be listed under the **Updates** tab.

The installed packages may have overlay icons to denote the status of the package. For example, the blue arrowhead (as shown in the following screenshot for our package library) denotes that an update is available, and the green check mark (as shown in the following screenshot for the `Newtonsoft.Json` library) denotes that the library is installed:



You can configure NuGet Package Manager to allow NuGet to automatically download the missing packages of a project/solution. You can also ask it to automatically check for any missing packages during the build in Visual Studio. To enable these settings, open the NuGet Package Manager settings.

From this screen (as shown in the following screenshot), you can also configure the default package management format to either **Packages.config** or **PackageReference**. .NET Core projects generally use the **PackageReference** format:



This screen also allows you to clear the NuGet cache. Click the **Clear All NuGet Cache(s)** button to continue. After reading this section of this chapter, you are now familiar with managing NuGet packages using the NuGet Package Manager. You have learnt how to configure NuGet Package Manager to automatically download missing packages of a project/solution. You have also learnt configuring the default package management format.

Summary

In this chapter, we have gained an overview of NuGet Package Manager, its usage, and the process of using it in projects/solutions. We also learned about how to create a NuGet package library for .NET Framework. We discussed the `nuspec` file, versioning mechanism, dependencies, conditional dependencies, and how to target the NuGet package for multiple .NET Frameworks.

Finally, we discussed how to create a local package source to test a package before publishing it to the store, and how to publish it to the NuGet gallery using the `nuget.org` portal. As you are now familiar with NuGet packages, you can create your components easily and publish it to the public store, so that developers can download it and easily use them in their projects.

In the next chapter, we will learn about the new debugging techniques in Visual Studio 2019, along with Breakpoints, DataTips, Diagnostic Tools, and XAML UI debugging.

3

Section 3: Debugging and Testing Applications

Debugging and testing applications plays a vital role in building enterprise-grade games and applications. To write better code and fix bugs, you should learn the art of debugging and writing unit test cases. In this section, we focus on giving you an in-depth understanding of the different debugging tools inside Visual Studio. The more comfortable you are with code debugging, the better the code that you write or maintain will be.

Later in this section, we will jump into a deep-dive section on Live Unit Testing, which automatically runs the impacted unit tests in the background as you edit code. This chapter will help you become proficient in building Live Unit Testing with Visual Studio 2019.

The following chapters will be covered in this section:

- Chapter 7, *Debugging Applications with Visual Studio 2019*
- Chapter 8, *Live Unit Testing with Visual Studio 2019*

7

Debugging Applications with Visual Studio 2019

Debugging is the core part of any application development. It is a process that helps you to quickly look at the current state of your program by walking through the code line by line. Developers generally start debugging the application while writing code. Some developers start debugging even before writing the first line of code to find out about its logic and functionalities. It's a rare case scenario when a developer completes writing code without debugging the application.

Visual Studio provides us with details about the running program as much as possible. It also helps to change value of variables and properties while the application is already running. Hence, debugging inside the Visual Studio IDE is becoming more popular every day.

In this chapter, we are going to cover the following topics:

- Overview of Visual Studio debugger tools
- Debugging C# source code using breakpoints
- Using DataTips while debugging
- Using the Immediate window while debugging your code
- Using the Visual Studio Diagnostics Tools

Technical requirements

To begin this chapter, you will need Visual Studio 2019 installed on your system. A basic understanding of the IDE and C# language is also recommended.

Overview of Visual Studio debugger tools

The debugger is a Visual Studio tool that works as a background process to inspect the execution of a program. Breakpoints are used to notify the debugger to pause the execution of the program when it hits a certain line. **Program database (PDB)** files are used to store the debugging information. It stores the line number, DataTips information, and other related information of the source code, which are required to debug the application. Visual Studio then reads it to pause the debugging process and provide more details of the execution.

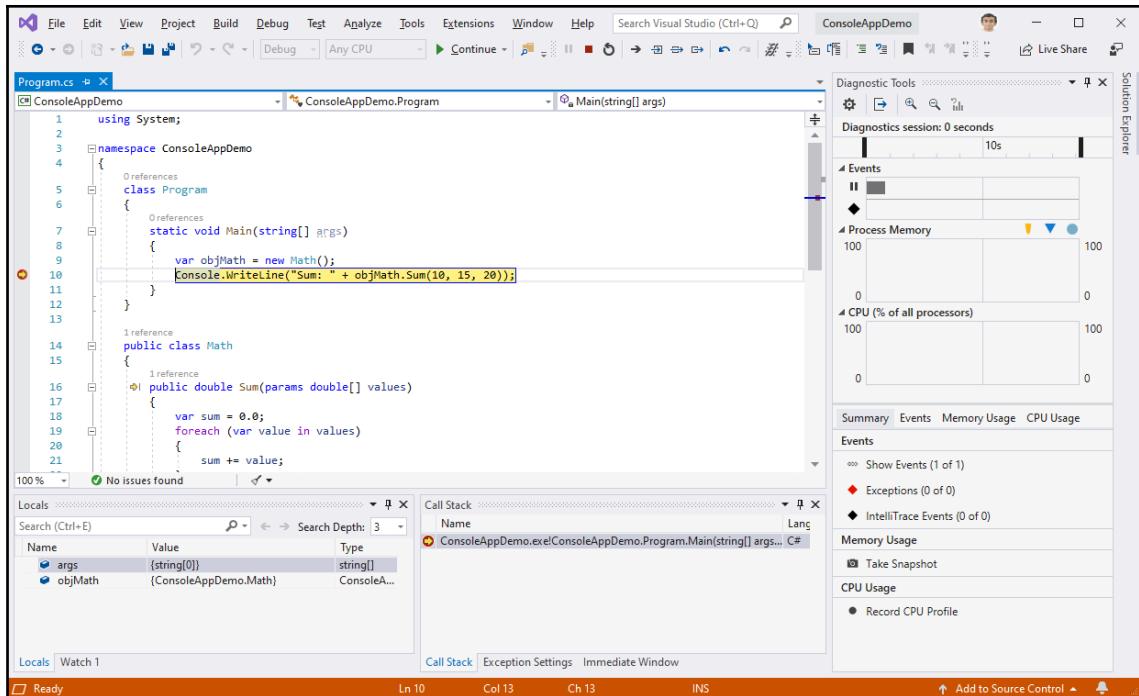


When you build the application in debug mode, the PDB file has more information than in release mode.

It is not definite that, in all cases, the debugger will work. There are certain situations when your debugging information is invalid, and, in those cases, the Visual Studio debugger will fail to attach to the running process; for example, if the binary version of your code is different than the actual code. This often happens when you've compiled your code and attached the binary in a modified version of the original code.

When you run your application in debug mode, you will see a different layout of the Visual Studio IDE. It automatically adjusts the window layout to provide you with a better environment for debugging.

In debug mode, you will not generally see the Solution Explorer, Team Explorer, Toolbox, and Properties window. Instead, Visual Studio creates a layout with C# Interactive, Call Stack, Diagnostics Tools, IntelliTrace, Watch window, Exception Settings, Immediate window, and all other related information:



In this chapter, we will demonstrate some of the important tools and techniques to master when it comes to code debugging.

Debugging C# source code using breakpoints

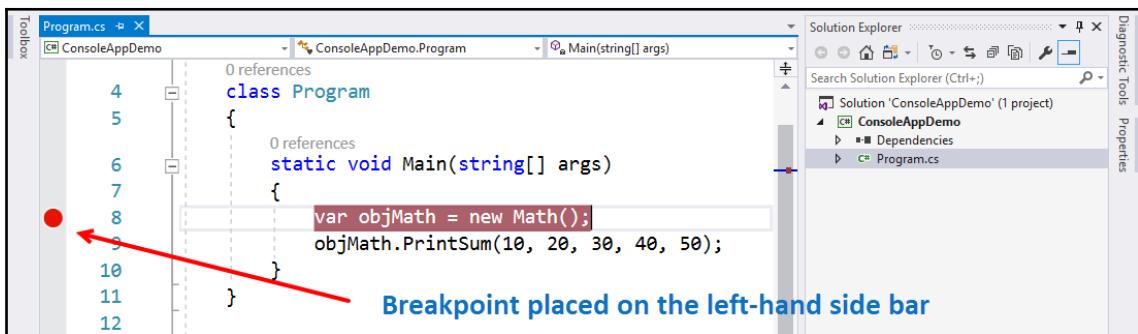
A **breakpoint** is a signal to tell the debugger to pause the current execution of the application at certain points defined in the code editor and wait for further commands. It does not terminate the execution; instead, it waits until the next instruction is sent to it and then resumes at any time.

The Visual Studio Code debugger is a powerful tool that helps you break the execution where and when you want to start the debugging process. It helps you to observe the behavior of the code at runtime and find the cause of the problem.

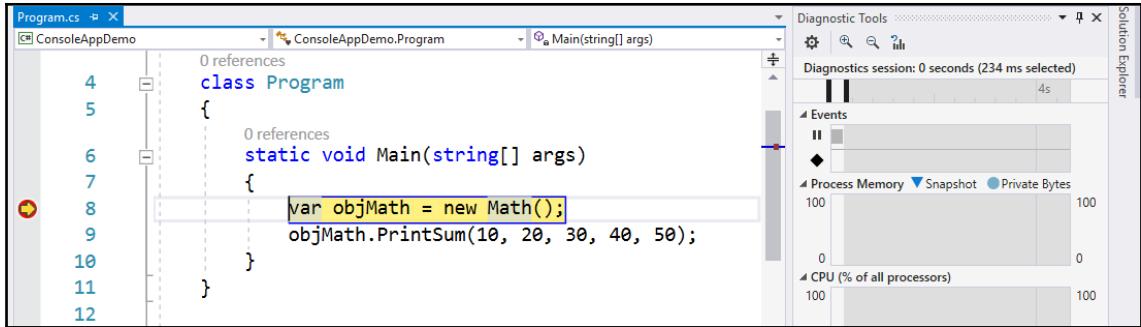
Organizing breakpoints in code

Let's start by placing our first breakpoint in the code and gradually move forward with code debugging:

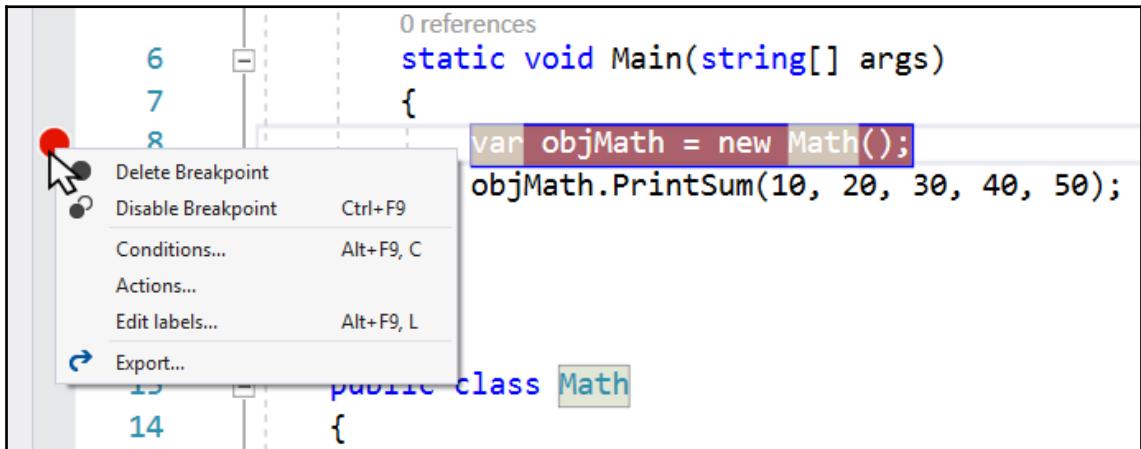
1. Click on the left-hand sidebar of the code file in Visual Studio to place the breakpoint. Alternatively, you can press *F9* to toggle the breakpoints.
2. Once you've placed a breakpoint, a red circle will be generated in the sidebar, and the entire code block in the line will have a dark red background:



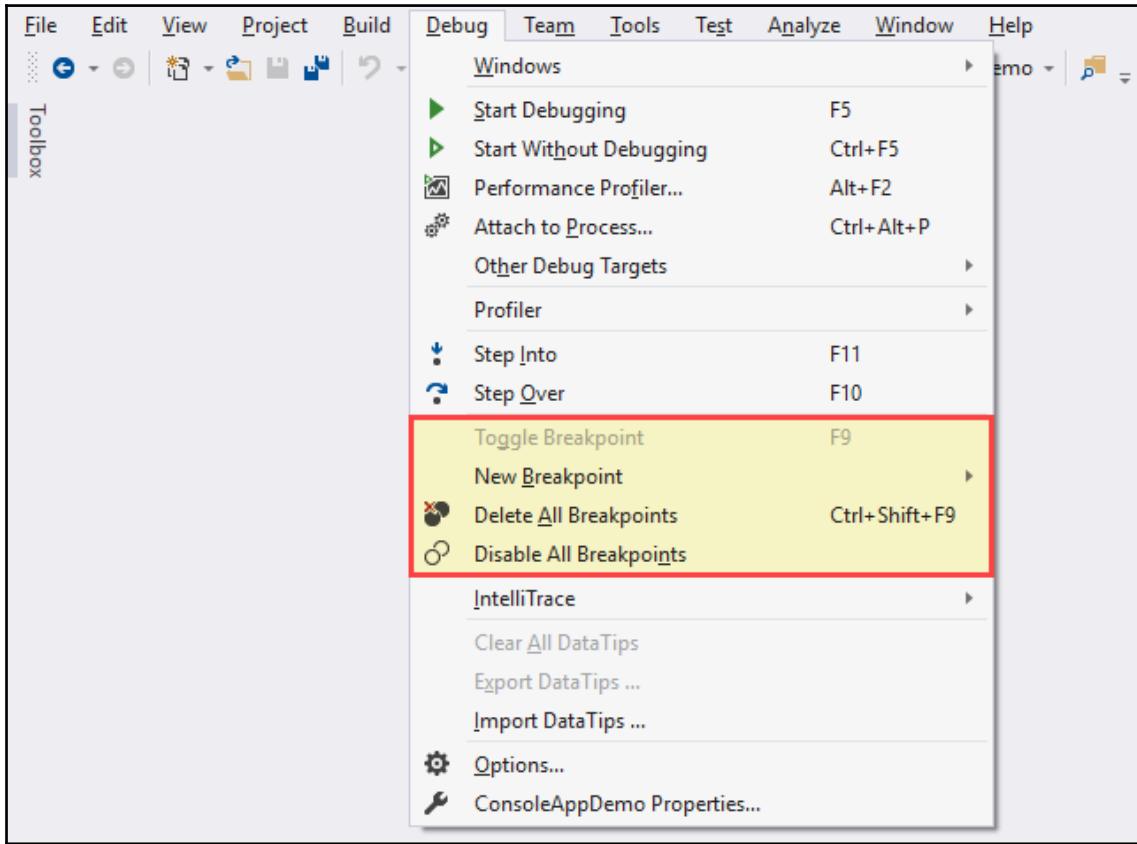
3. When you press *F5* or start the application in debug mode, it will break at the point where you placed it.
4. A yellow arrowhead in the left-hand sidebar represents a marker for the line that is currently being executed, with a yellow block on the execution line, which you can drag up or down within the same debugging context to restart or continue the execution of the code block:



- When you right-click on the breakpoint circle on the left-hand sidebar, a context menu will pop up on the screen, which allows you to delete or disable the breakpoint:



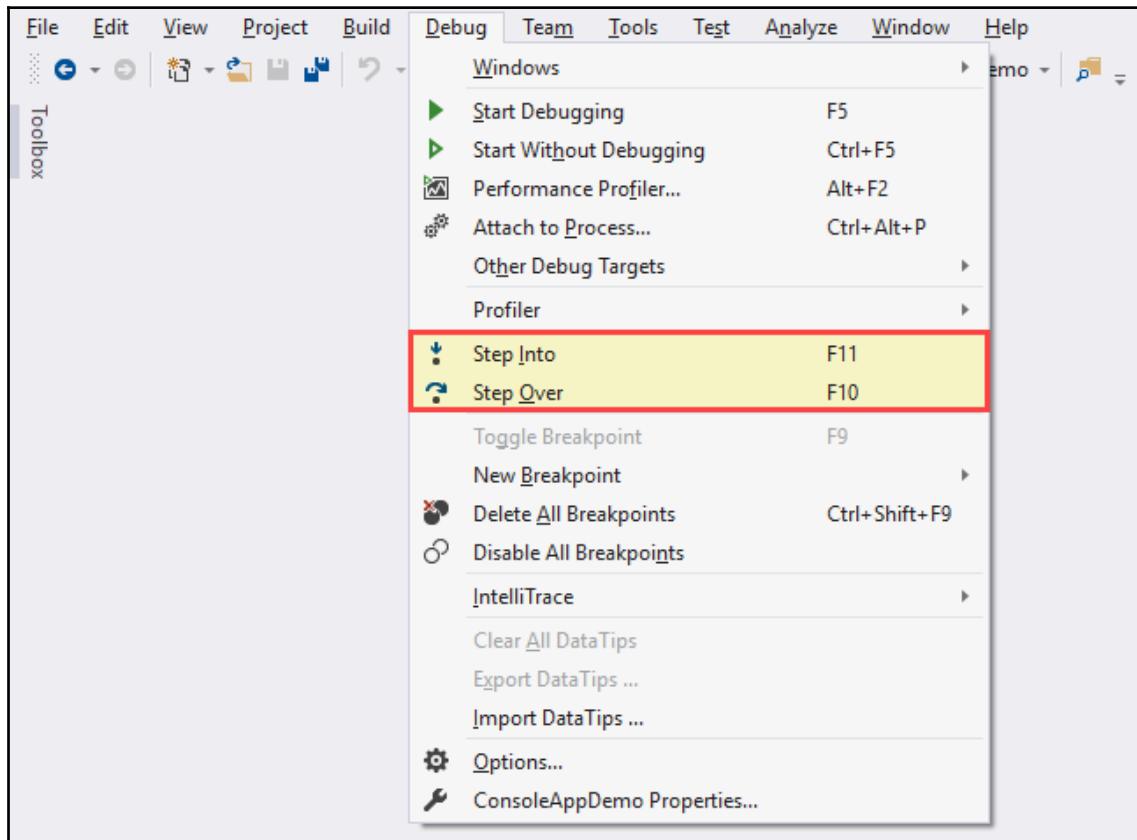
6. If you want to delete or disable all of the breakpoints in the currently loaded solution, you can navigate to the Visual Studio **Debug** menu and click **Delete All Breakpoints** or **Disable All Breakpoints**, respectively.
7. From this menu, you can also create a new breakpoint or toggle an existing breakpoint:



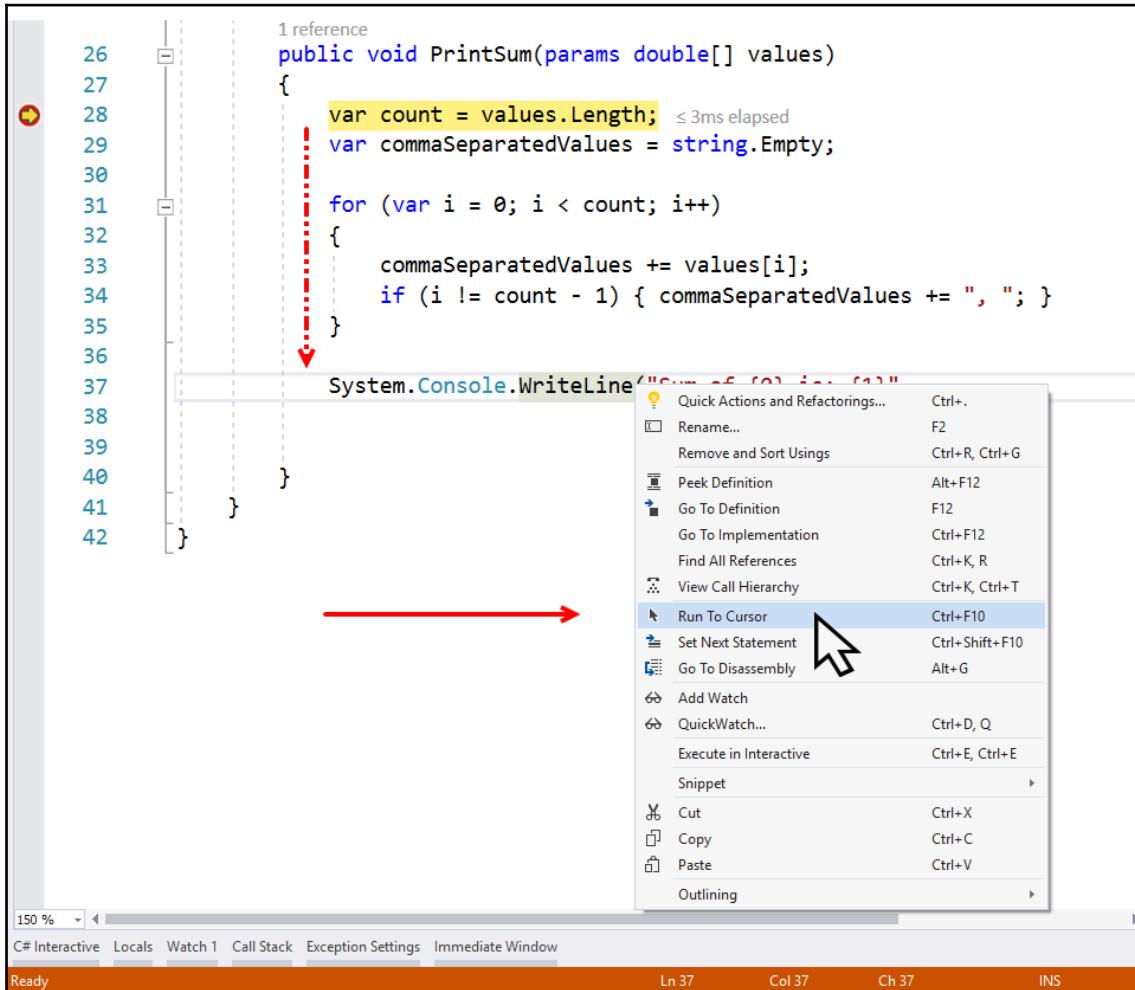
Now that you have become familiar with the breakpoints in the code and the breakpoint menus, let's begin with the debugger execution steps in the next section.

Debugger execution steps

When you are debugging code, you may want to debug it line by line. Hence, you will need to execute the current line and step over to the next one. Sometimes, you may want to go deep into the method or property to debug the code. All of this can be done from the Visual Studio **Debug** menu:

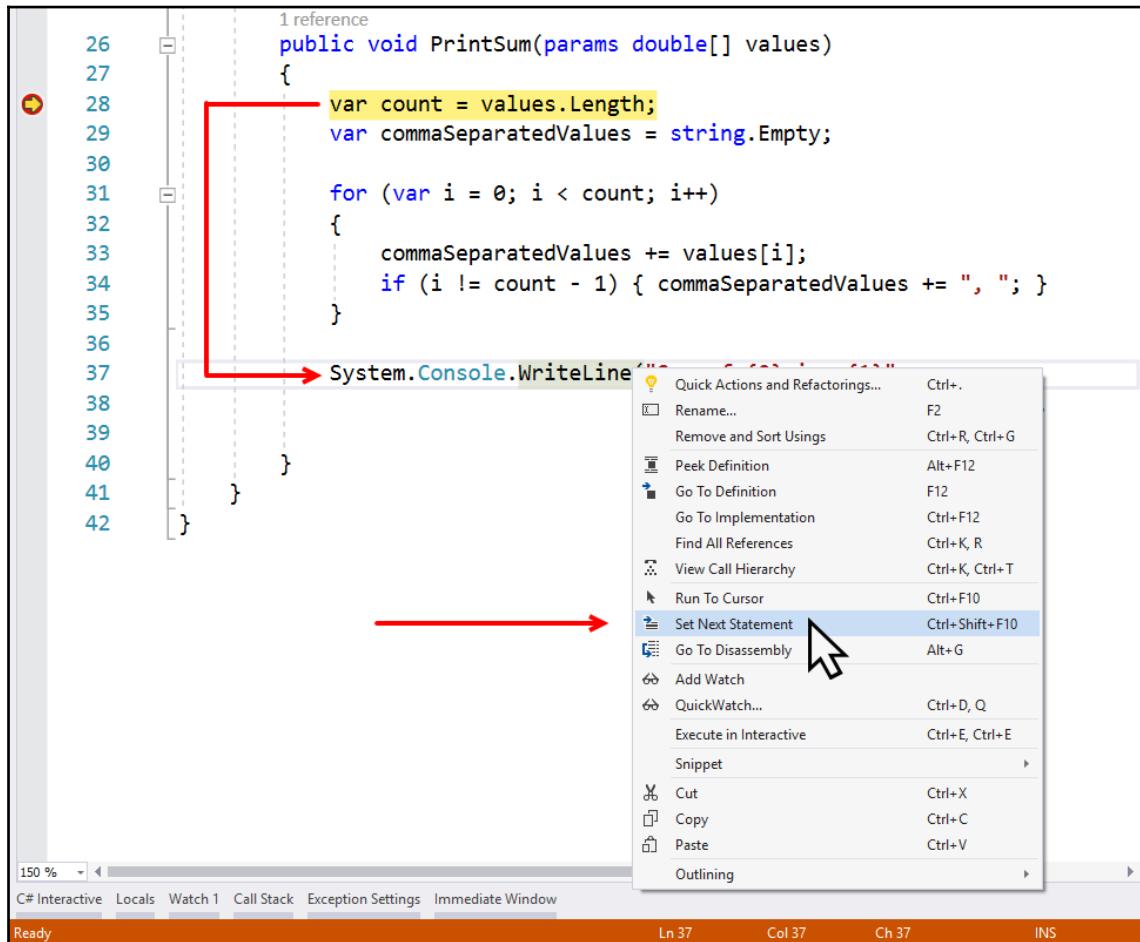


You can click the **Debug | Step Over** menu or press the *F10* keyboard shortcut to execute the current line and jump into the next line for execution. Click the **Debug | Step Into** menu item or press the *F11* key to step into any property or method for debugging. You can then continue line-by-line execution by pressing *F10* or continue executing the program to the end or to the next breakpoint by pressing *F5*:



Visual Studio also provides some smart menu entries to help you debug your code. When you are at any breakpoint, you can continue the execution to a specific line and break again. To do so, right-click on any line within that debugging context and choose **Run to Cursor**, as shown previously. Alternatively, you can click on the desired line and press *Ctrl + F10* to continue the execution.

In this case, all of the lines between the executing line and select line will execute before it breaks. This is the same as placing another breakpoint on that line:



You can also ask Visual Studio to jump into a specific line without executing certain lines of code. This is sometimes needed when you smell bad code that is causing a bug to occur, and you want to check it without removing/commenting/changing that portion of code.

A point to remember is that **Run To Cursor** executes lines of code in-between, whereas **Set Next Statement** skips code in-between.

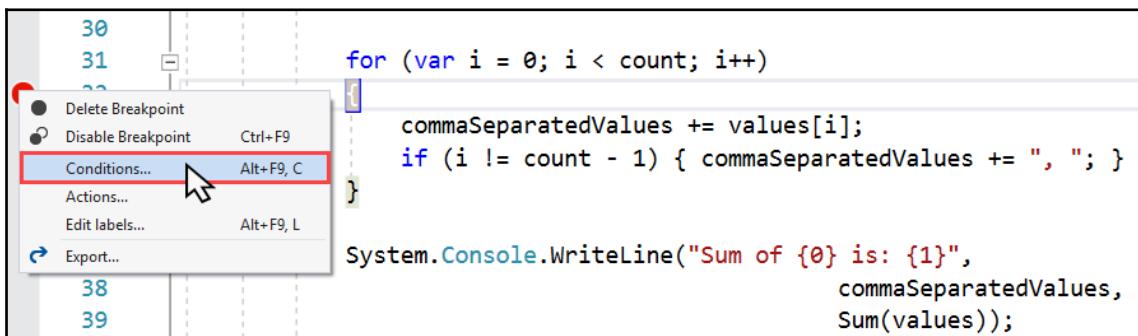


While you are at a specific breakpoint, you can right-click on any line within that debugging context and select **Set Next Statement**. You can also press *Ctrl + Shift + F10* to move the current marker of your debugger to that selected line. Alternatively, you can also drag the marker arrow of the current line to the new execution line to set it as the next statement. None of the code written within those two lines will execute during that debugging instance.

Adding conditions to breakpoints

You can add conditions to break the execution when a certain condition is met. This is often useful when you are iterating a list and want to check the runtime value for a specific condition.

To add a condition to a breakpoint, right-click on the red circle on the left-hand sidebar. A context menu will pop up on the screen. Click on the **Conditions...** menu to open the inline dialog. You can also press *Alt + F9 + C* to open it:



There are three different types of conditions that you can place on a breakpoint:

- Conditional expressions
- Hit counters
- Filters

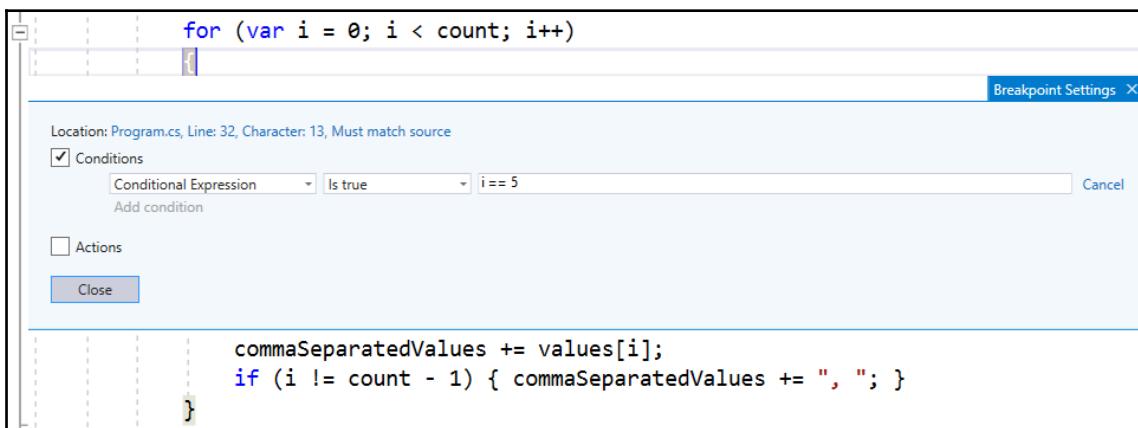
We'll learn to understand all of these in the following sections.

Using conditional expressions

Conditional expressions are often useful to hit a breakpoint on some conditions. For example, perhaps you would like to debug a `for` statement when a specific index count is reached or an item in a collection meets a specific value.

To add a conditional expression to a breakpoint inside Visual Studio 2019, do the following:

1. Select the **Conditions** checkbox from **Breakpoint Settings** and set the condition type to **Conditional Expression** from the drop-down menu.
2. The next drop-down menu allows you to select the comparer (**Is true** or **When changed**). The textbox next to it allows you to enter the condition. In this case, we entered `i == 5`:

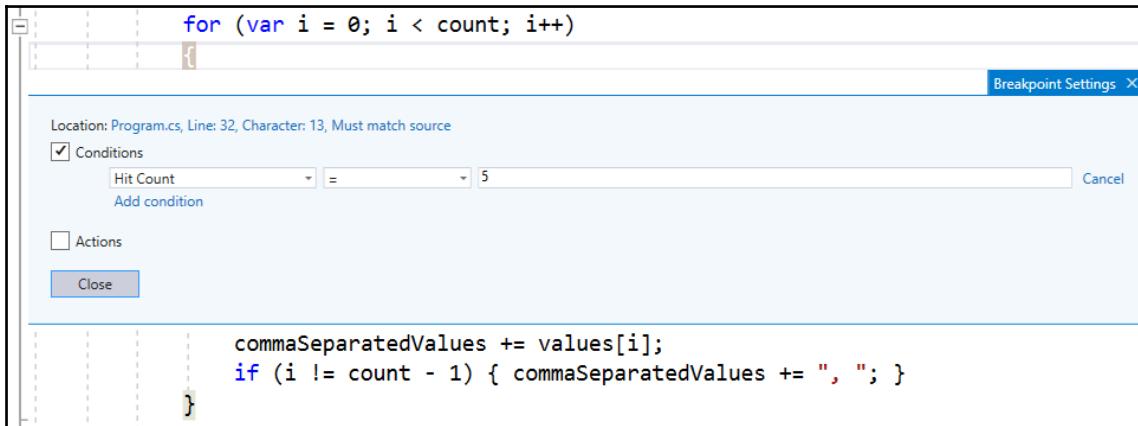


3. In the preceding example, the breakpoint will directly hit when the `i` variable of the `for` loop is set to 5. You can then start debugging your code when the said condition is met.

You can add multiple conditions by clicking the **Add condition** link, which will be clubbed with the **AND** operator. Once a condition has been added to a breakpoint, the red breakpoint circle will show a plus (+) symbol.

Using breakpoint hit counters

A hit counter is used to determine the count of breakpoint hits on a specific line and, based on that, break the debugger. This is useful in iterations where you want to stop only when the number of hits it encounters reaches a threshold value:



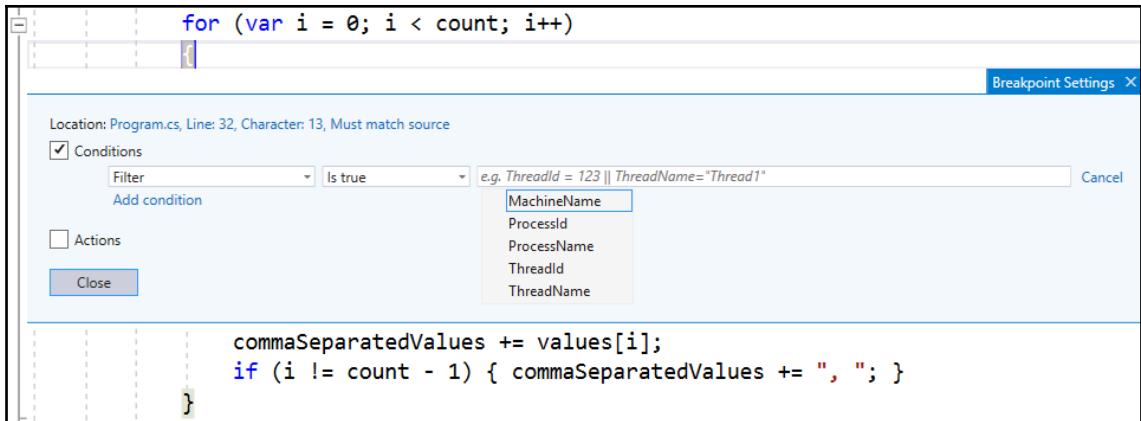
In the **Breakpoint Settings** dialog, select the **Conditions** checkbox and, from the drop-down menu, select **Hit Count**. From the next drop-down menu, select either **=**, **Is a multiple of**, or **>=** as the type of comparer, and then enter the value in the text field.

When the hit count is reached, the debugger will stop at that breakpoint and wait for further commands. Like conditional expressions, you can add multiple hit counters, along with other conditions, with the **AND** operator.

Using breakpoint filters

Breakpoint filters are useful when you want to specify any additional criteria for the breakpoint. You can specify the breakpoint to hit only when a specific `MachineName`, `ProcessId`, `ProcessName`, `ThreadId`, or `ThreadName` matches.

For example, you can select **Filter** from the first drop-down menu, **Is true** from the second drop-down menu, and then specify the conditional value in the associated input box, as shown in the following screenshot:

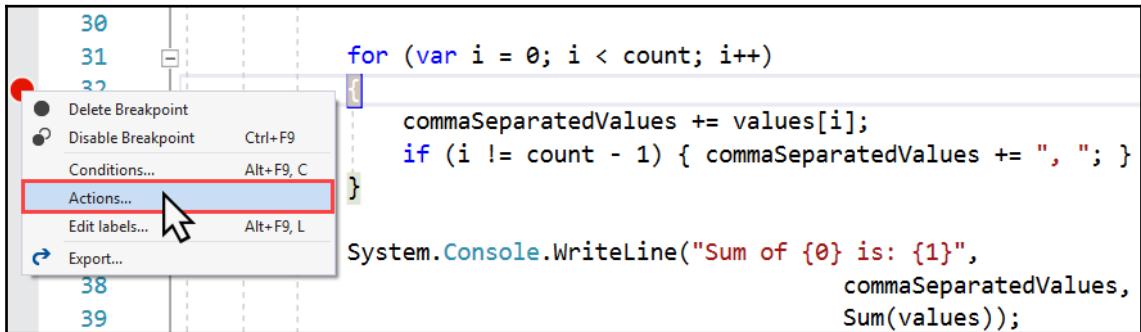


This way, you can manage the breakpoint filters for your code inside the Visual Studio IDE. Make sure to select the proper conditions based on your actual needs.

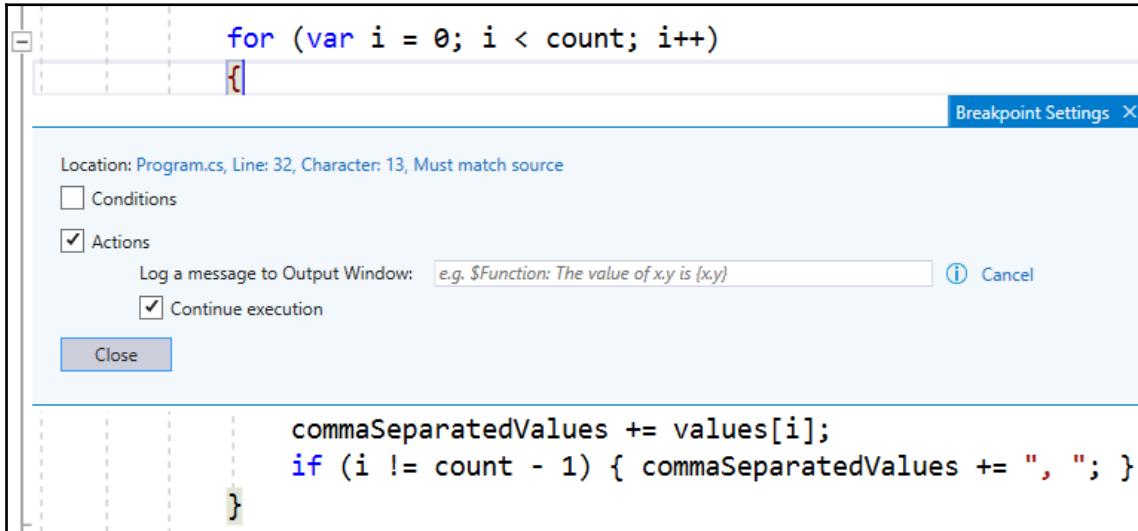
Adding actions to breakpoints

You can specify some actions to a breakpoint to log a message to the output window or run a specific function when the breakpoint is reached. This is generally done to trace an execution log and print various values at different moments of the process.

To add actions to a breakpoint, right-click on the red circle on the left-hand sidebar and click on **Actions...** from the context menu that pops up on the screen:



This will open the **Breakpoint Settings** dialog onscreen, with the **Actions** checkbox checked. In the input box, you can either enter the name of the function that you want to execute for tracing or enter the string with the variable that you want to print:



You can use curly braces to print the value of a variable. For example, the following expression, `The value of 'i' is {i}`, will print the value of the `i` variable, along with the text that we specified.

You can also use some special keywords, mentioned as follows, to print their current values:

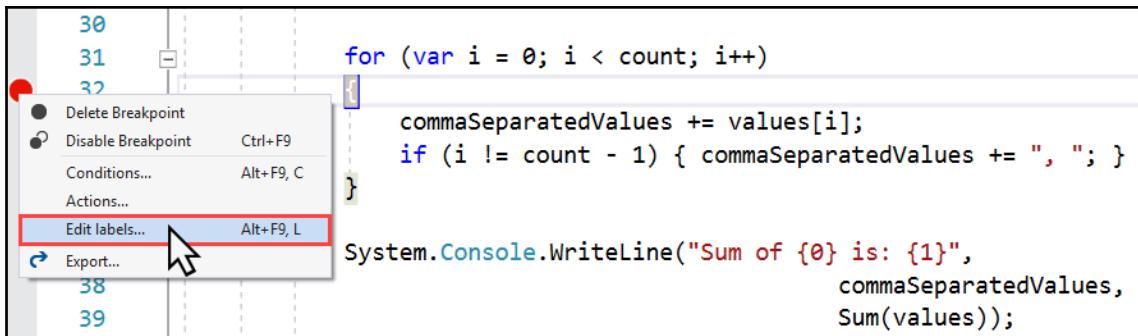
- `$ADDRESS`: Current instruction
- `$CALLER`: Previous function name
- `$CALLSTACK`: Call stack
- `$FUNCTION`: Current function name
- `$PID`: Process ID
- `$PNAME`: Process name
- `$TID`: Thread ID
- `$TNAME`: Thread name

By default, the **Continue execution** checkbox is selected. This means the actions that we selected will be executed without stopping the code execution.

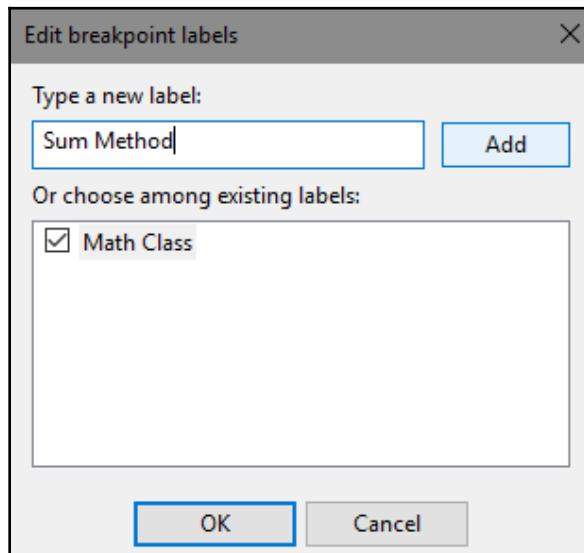
Adding labels to breakpoints

When you are working with multiple breakpoints set over multiple code files of a large project or solution, it is sometimes difficult to identify and manage them properly. Visual Studio provides a better way to organize them.

As shown in the following screenshot, right-click on a specific breakpoint and click on **Edit Labels...**. Alternatively, you can press **Alt + F9 + L** to open the breakpoint labels dialog:



This will open the **Edit breakpoint labels** dialog window, as shown in the following screenshot:



If you have already defined any labels within the same solution, this will list all of them here. You can add a new label to your selected breakpoint or link a label with other breakpoint labels listed on this screen.

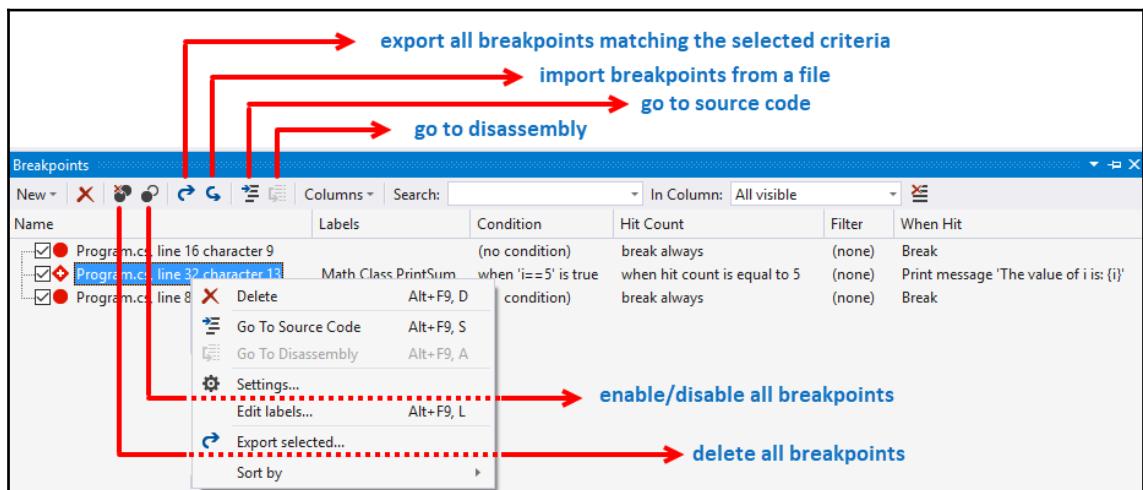


You can add multiple labels to a single breakpoint from the **Edit breakpoint labels** dialog box.

Managing breakpoints using the Breakpoints window

Visual Studio provides a straightforward way to manage all of your breakpoints under a single toolbox window, which you can find by navigating to **Debug | Windows | Breakpoints** (or use *Ctrl + D + B*).

When you place multiple breakpoints in your code file or solution, this window helps you to navigate between them very easily. Here's a screenshot of the **Breakpoints** window, demonstrating all of the toolbar icons and menus:

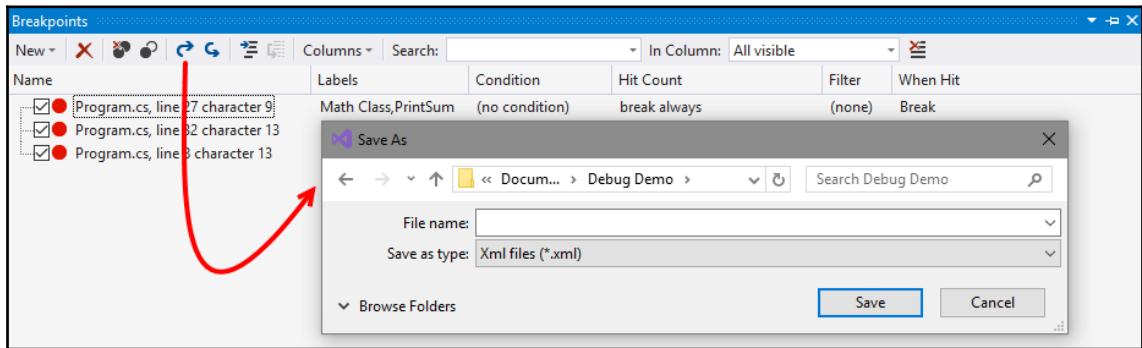


You can add a new breakpoint, delete a breakpoint, or enable/disable it from here. This also lists the label, conditions, hit counter, filter, complete function name, process, and many other details that are set to each breakpoint; they are easily searchable.

Exporting/importing breakpoints

Visual Studio allows you to export/import breakpoints. This is useful when multiple developers are working on the same code and want to debug at the same breakpoints set by other developers.

You can either export individual breakpoints by right-clicking on the breakpoint circle present on the left-hand sidebar or can export all of the listed breakpoints in the **Breakpoints** window by clicking the export button, as follows:



This will save the details in an XML file, which you can share with others. They need to import it by clicking the import breakpoints from the breakpoint window. All of the detailed information shared by you will be automatically loaded in the editor.

Using DataTips while debugging

During code debugging, **DataTips** are used to provide more information about an object/variable in the current scope of execution, and they work only in break mode. The DataTip uses the data type information to display each value that has a type associated with it. The debugger loads the object information recursively in a hierarchical structure and displays it in the editor.

Here's an example of how DataTips loads an object:

The screenshot shows a code editor window with a C# file. A breakpoint is set at line 54. A tooltip for the variable 'values' is displayed, showing its type as 'double[5]' and its contents as '[0] 10, [1] 20, [2] 30, [3] 40, [4] 50'. A red arrow points from the text 'Data Tips' to the tooltip.

```
1 reference
public void PrintSum(params double[] values)
{
    var count = values.Length;
    var commaSeparatedValues = string.Empty;
    for (var i = 0; i < count; i++)
    {
        commaSeparatedValues += values[i];
        if (i != count - 1) { commaSeparatedValues += ", ";}
    }
}
```

To display a DataTip, place a breakpoint in your application code and run it in debug mode. When the breakpoint hits, hover over an object/variable to display the DataTip associated with that object/variable. It is a tree of members that you can expand to get more details associated with debugging information. When you hover out of the object/variable, the DataTips disappear.



DataTips are evaluated in the current execution context where the breakpoint hits and the execution moves into a suspended state. Hence, hovering over a variable in another function, which is in the current context, will show the same value in the DataTips that are being displayed in the current execution context.

Pinning/unpinning DataTips for better debugging

When you hover out from the variable, the DataTips disappear. However, we need to keep it displayed. The Visual Studio editor provides a way to keep it visible: you can pin a value from a DataTip to the editor screen and drag it to any position. To do this, just click on the pin to source icon in the tooltip and a pin icon will appear on the left-hand sidebar on the same line where you positioned the pinned value. This is shown in the following screenshot:

A screenshot of the Visual Studio code editor during a debugging session. The code shown is:

```
1 reference
public void PrintSum(params double[] values)
{
    var count = values.Length;
    var commaSeparatedValues = string.Empty;
    for (var i = 0; i < count; i++)
    {
        commaSeparatedValues += values[i];
        if (i != count - 1) { commaSeparatedValues += ", ";}
    }
}
```

The variable `values` is highlighted in yellow. A tooltip for `values[1]` shows the value `20`. A red arrow points from the pinned icon on the left sidebar to this tooltip. Another red arrow points from the pinned icon to a pinned DataTip window titled `values[1] 20`, which contains a list of values: [0] 10, [1] 20, [2] 30, [3] 40, [4] 50. The entry for index 1 is highlighted. The title of the window includes the text **Pinned Data Tips**.

You can also add a comment to pinned DataTips. Hover over the pinned value and expand the arrowhead. It will provide you with a space to enter a comment, as shown here:

A screenshot of the Visual Studio code editor during a debugging session. The code shown is:

```
1 reference
public void PrintSum(params double[] values)
{
    var count = values.Length;
    var commaSeparatedValues = string.Empty;
    for (var i = 0; i < count; i++)
    {
    }
```

The variable `values` is highlighted in yellow. A tooltip for `values[1]` shows the value `20`. A red arrow points from the pinned icon on the left sidebar to this tooltip. The pinned DataTip window is open, showing the value `20` and a comment input field with the placeholder `Type a comment here`.

When you are debugging your code, the value stored in the pinned entry will change as per the current context. The IDE stores the DataTip information in persistent storage for later use. When the debugging session is over, you can hover over the pinned icon present on the left-hand sidebar to view the value from the last debugging session. This is often used to share debugging details with other members of the team.

When you pin a DataTips value, the icon changes to an unpinned state. To unpin a DataTip, click on the unpin from source icon.

Inspecting DataTips in various Watch windows

Visual Studio provides various Watch windows to simplify the debugging process so that you can investigate objects in a fixed debugging window, just like the DataTips. There are three distinct types of Watch windows available:

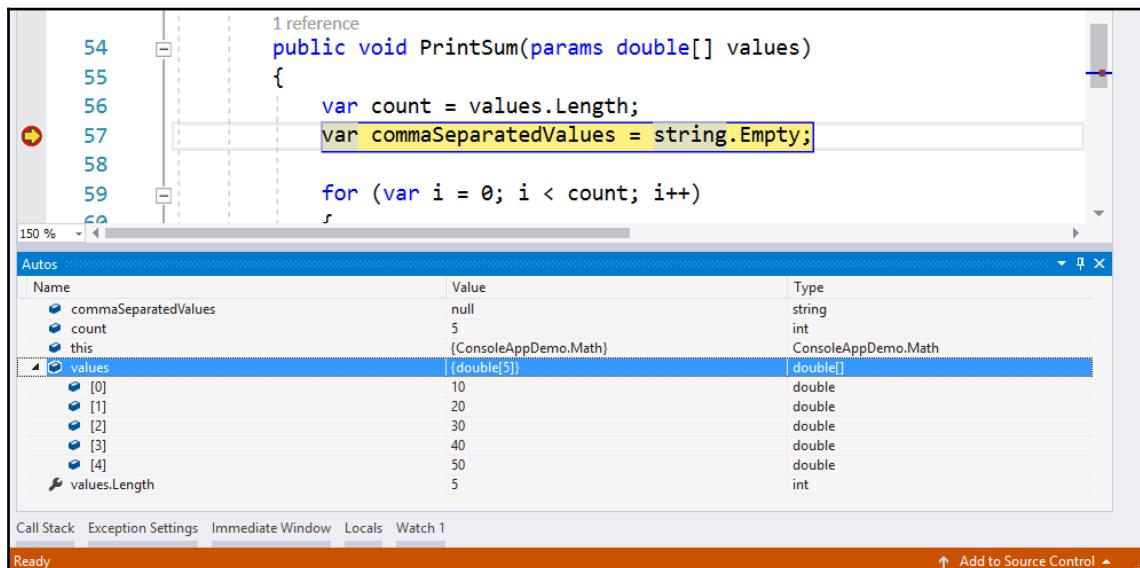
- **Autos**
- **Locals**
- **Watch**

We'll look at each of these in the upcoming sections.

The Autos window

The **Autos** window shows all of the objects and variables information from the current executing context. This information is loaded when the debugger hits a breakpoint. Generally, Visual Studio automatically generates this list and updates based on the context of the debugger.

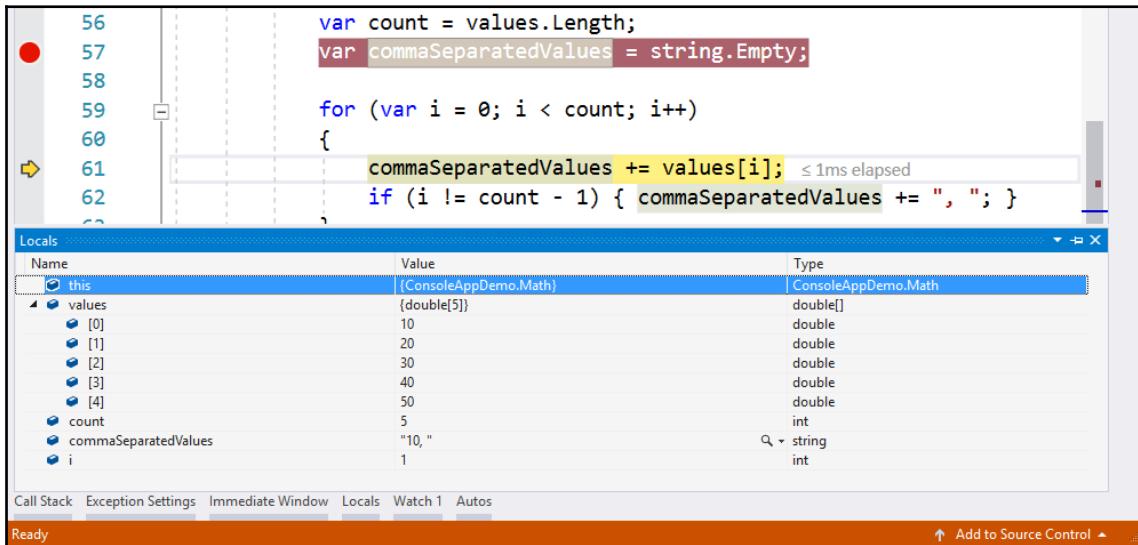
Here is a screenshot of the **Autos** window, which you can manually open from the Visual Studio **Debug | Windows | Autos** menu path:



You can also invoke the **Autos** window by pressing **Ctrl + D + A**.

The Locals window

The **Locals** window displays the information of the local variables and objects based on the current thread execution context. You can manually invoke this window from the Visual Studio **Debug** | **Windows** | **Locals** menu path:



You can also invoke the **Locals** window by pressing **Ctrl + D + L**.

The Watch window

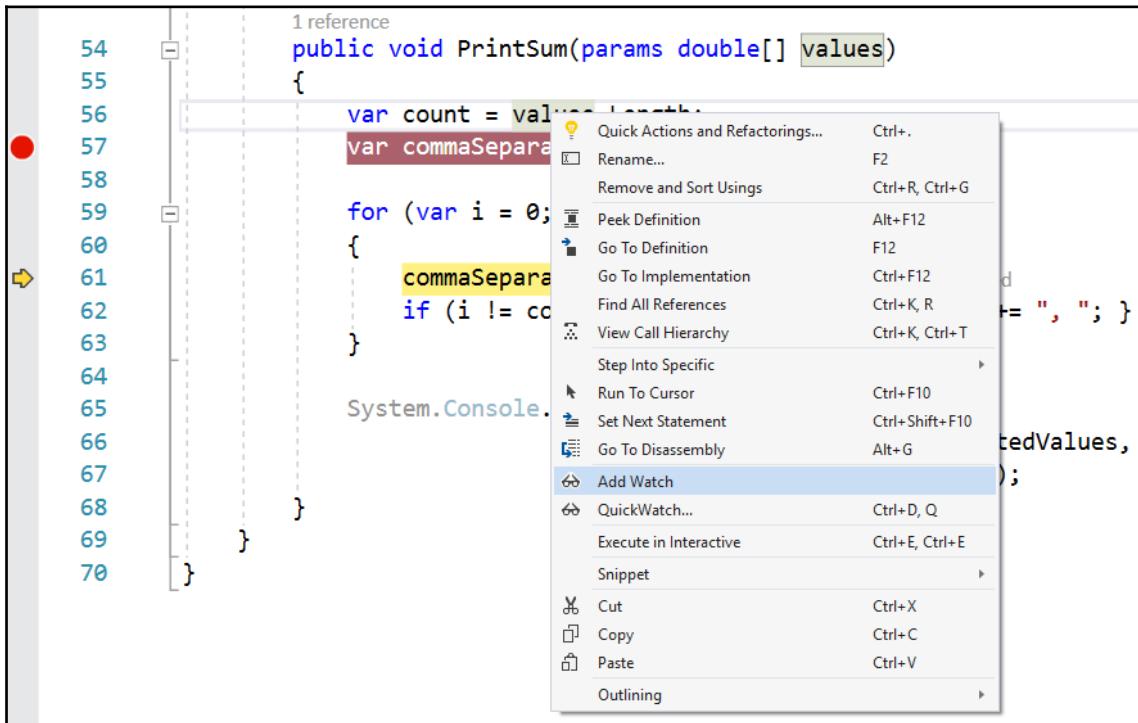
You can have four **Watch** windows, named **Watch 1**, **Watch 2**, **Watch 3**, and **Watch 4**. These are customized windows that show information about objects and variables based on what you have added to them.

In general, when you add a variable to a Watch window, it gets added to **Watch 1**. If you want to move it to a different Watch window, you need to drag it from **Watch 1** to another one.



While you are in debugger break mode, you will see a blank Watch window. However, you can right-click on any object or variable in that context and click **Add Watch** from the context menu to add the selected variable in the watch list.

Not only can you add objects and variables, but you can also add any expression to it. Simply select an expression, right-click on it, and from the context menu, click the **Add Watch** menu item, as shown here:



You can drag a variable, object, or expression from the code editor to the **Watch** window. You can also add a new item to the **Watch** window grid by double-clicking on a new grid row.

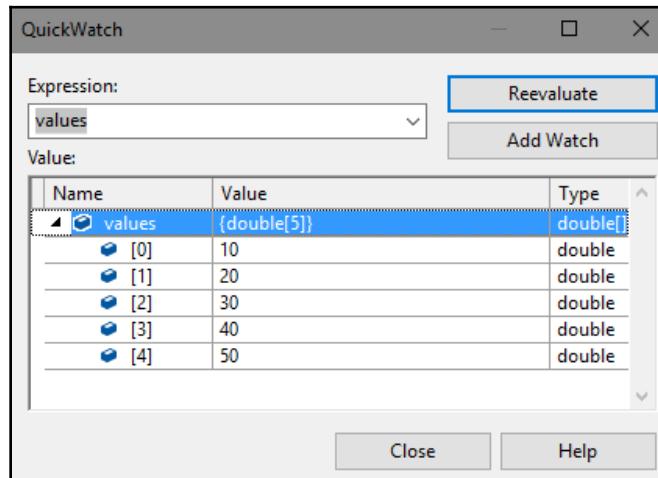
Like other windows, it also displays the data in three columns, that is, **Name**, **Value**, and **Type**, and the tree can be expanded to get more details about a complex object:

The screenshot shows the Visual Studio 2019 interface during a debug session. The code editor displays a C# method named `PrintSum` with a break point at line 54. The `Watch 1` window is open, showing the variable `values` and its elements [0] through [4]. The expression `commaSeparatedValues += values[i]` is being evaluated, showing the result `"10, 20"`. The status bar at the bottom indicates the message `≤ 1ms elapsed`.

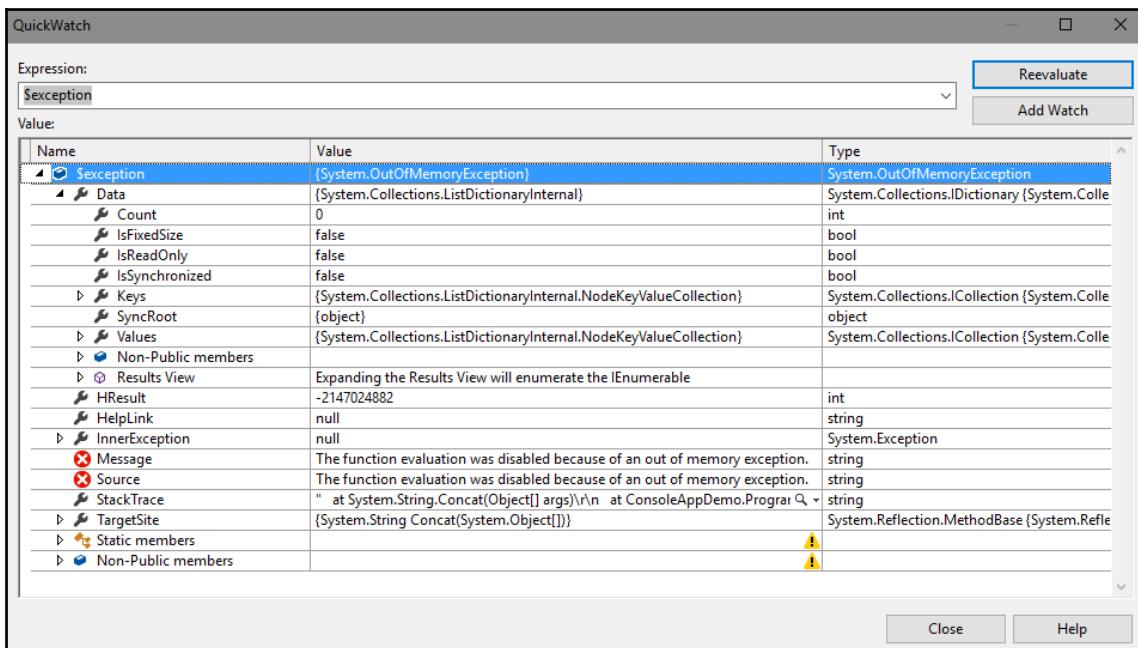
You can also add a variable, an object, or an expression to **QuickWatch** to simplify the value inspection process. From the right-click context menu, click on **QuickWatch...** to add it for inspection:

The screenshot shows the Visual Studio 2019 interface during a debug session. The code editor displays the same `PrintSum` method. A context menu is open over the variable `commaSeparatedValues` at line 56. The **QuickWatch...** option is highlighted in blue, indicating it is selected.

Alternatively, you can press **Ctrl + D + Q** to add it to the **Quick Watch** window. You can then expand an object to display its properties or values:



When you have a more complex object, like an exception object, the **QuickWatch** window is more useful to get each and every detail out of the object in a single debugging tool window:

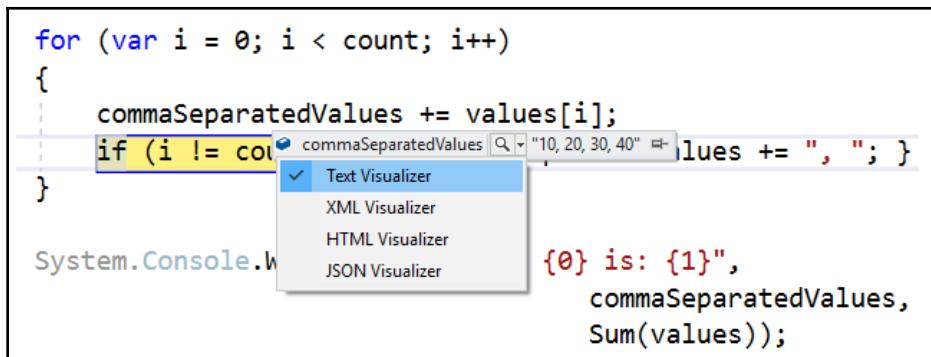


Now that we have learned about inspecting DataTips in various Watch windows, let's jump into the next section and use visualizers to display complex DataTips.

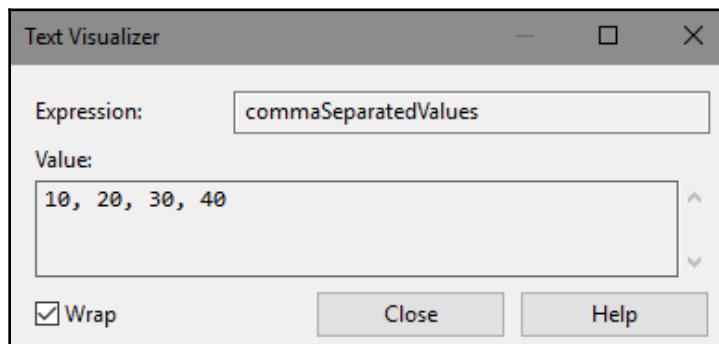
Using visualizers to display complex DataTips

Visual Studio provides a set of visualizers to help you display complex DataTips while you debug your code. When you hover over a variable, the DataTip can contain a find icon based on the data type associated with the debugger visualizer for that variable. Clicking on the arrowhead opens a menu with a list of visualizers for that debugging instance.

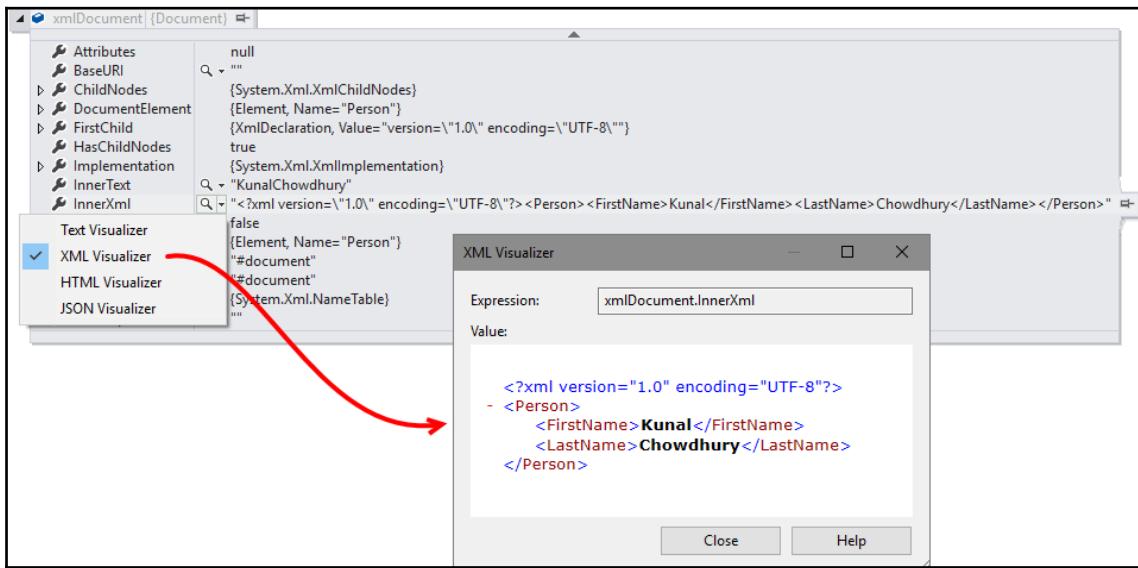
For example, check out the following screenshot, where we have four visualizers named **Text Visualizer**, **XML Visualizer**, **HTML Visualizer**, and **JSON Visualizer**:



Here, if you click on **Text Visualizer**, a dialog box will appear showing the text representation of the debugging value. The text visualizer shows it in text format, whereas the XML, HTML, and JSON visualizers show it in their respective format based on the associated data type. Based on the content of the variable, you should use the respective visualizer:



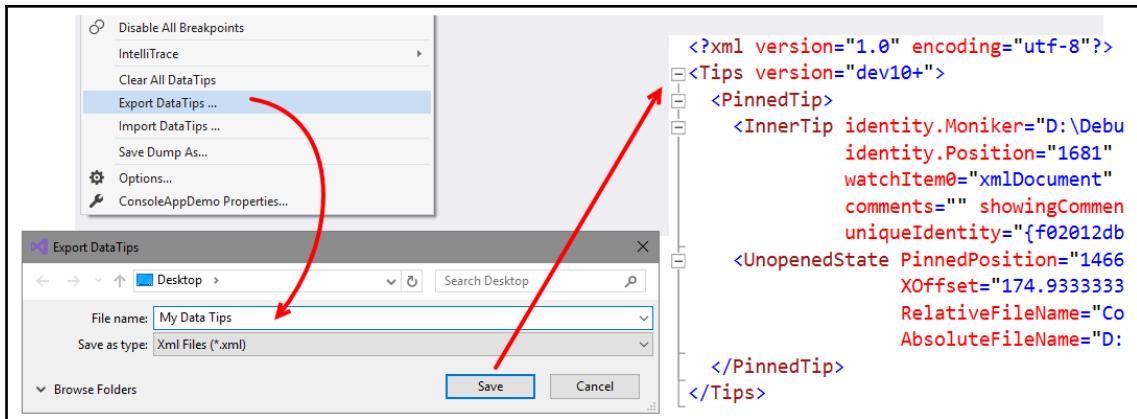
Here's an example of how the Visual Studio **XML Visualizer** represents the content of an XML document:



In the next section, we will learn how to import and export DataTips.

Importing/exporting DataTips

Like breakpoints, you can also export DataTips to an XML file, which you can share with your team. Others can import it to the same source and debug it further. To export DataTips, go to the Visual Studio **Debug | Export DataTips...** menu, which will open the **Export DataTips** dialog. Select the folder where you want to save the XML file, give the file a name, and click **Save**. This will produce an XML file, which will look like the one shown in the following screenshot (on the right):

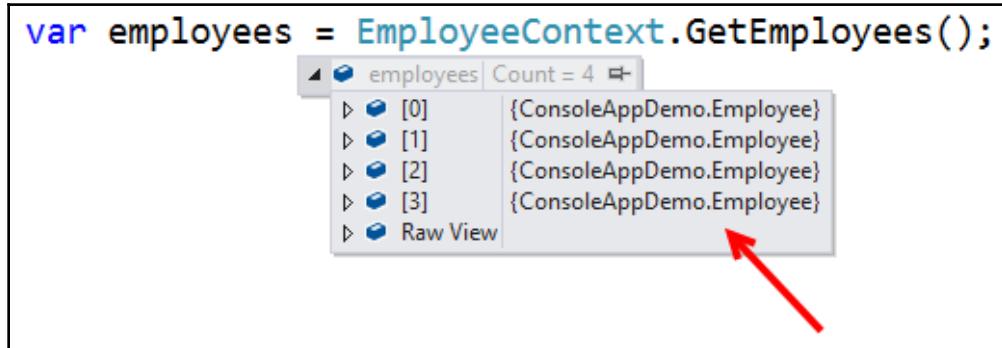


It is a SOAP-based XML file and includes all of the related information that you need to reload the entire DataTip.

On the other hand, to import DataTips, navigate to **Debug | Import DataTips...** and select the desired file.

Using the debugger to display debugging information

Sometimes, it is difficult to debug a complex data type value. For example, with a collection of employees, if you see it inside a DataTip, you will see that each item of the list displays the object by default. To view each property, you need to expand it, which becomes difficult in some scenarios where you have a lot of data:



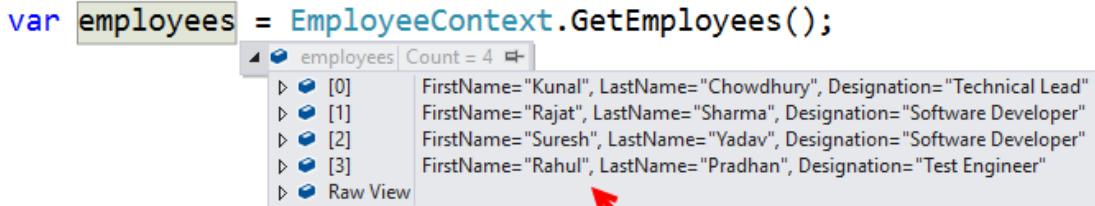
As shown in the preceding screenshot, we are expanding the object to check the property values of each object. Since it's a simple object, we may find this easy, but think about a scenario where you have multiple properties and variables inside it. Debugging that becomes very difficult, which you already know.

So, what can be done to simplify the debugging process? Visual Studio provides an attribute, `DebuggerDisplay`, in the `System.Diagnostics` namespace which, when set to a class, controls how the member value is going to be displayed in the debugger window.

To implement it, set the `DebuggerDisplay` attribute to a class and pass the string that you want to display as an argument to it. The value inside {} (curly braces) defines the property that you want to show in the debugger window. You can add multiple properties to display in the DataTips for an easier debugging experience:

```
[DebuggerDisplay("FirstName={FirstName}, LastName={LastName}, Designation={Designation}")]
public class Employee
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Designation { get; set; }
}
```

Now, when your debugger hits the breakpoint and you hover over it to see the DataTip, you will see that the string that you passed as an argument to the `DebuggerDisplay` attribute (along with the property details in curly braces) gets printed in the DataTips instead of the object representation. Here's a screenshot of how it will look in our example, which we used in the preceding screenshot:





In C#, you can also use an expression inside the curly braces that has implicit access to the member variable of the current instance of the target type.

These were some of the DataTips we studied; now, let's move on to using the Immediate window while debugging.

Using the Immediate Window while debugging your code

The **Immediate Window**, which is present under the Visual Studio **Debug | Windows | Immediate** menu and can be invoked using the *Ctrl + D + I* keyboard shortcut, is used while debugging applications. You will find it useful while executing statements, evaluating expressions, and/or printing any values in the debugging context.

For example, let's take the previous example and populate the `employees` object. Once your debugging context has evaluated said object, you can perform an operation on that object within the **Immediate Window**.

In general, entering the `employees` object name will print the object information that's available in that collection. When `DebuggerDisplay` is set to the model class, entering the same `employees` name will print the entire details of the object in a formatted string, as defined in the debugger display attribute. Here's a screenshot of this:

```
Immediate Window

employees;
Count = 4
[0]: {ConsoleAppDemo.Employee}
[1]: {ConsoleAppDemo.Employee}
[2]: {ConsoleAppDemo.Employee}
[3]: {ConsoleAppDemo.Employee}

employees; ← when debugger display is set
Count = 4
[0]: FirstName="Kunal", LastName="Chowdhury", Designation="Technical Lead"
[1]: FirstName="Rajat", LastName="Sharma", Designation="Software Developer"
[2]: FirstName="Suresh", LastName="Yadav", Designation="Software Developer"
[3]: FirstName="Rahul", LastName="Pradhan", Designation="Test Engineer"

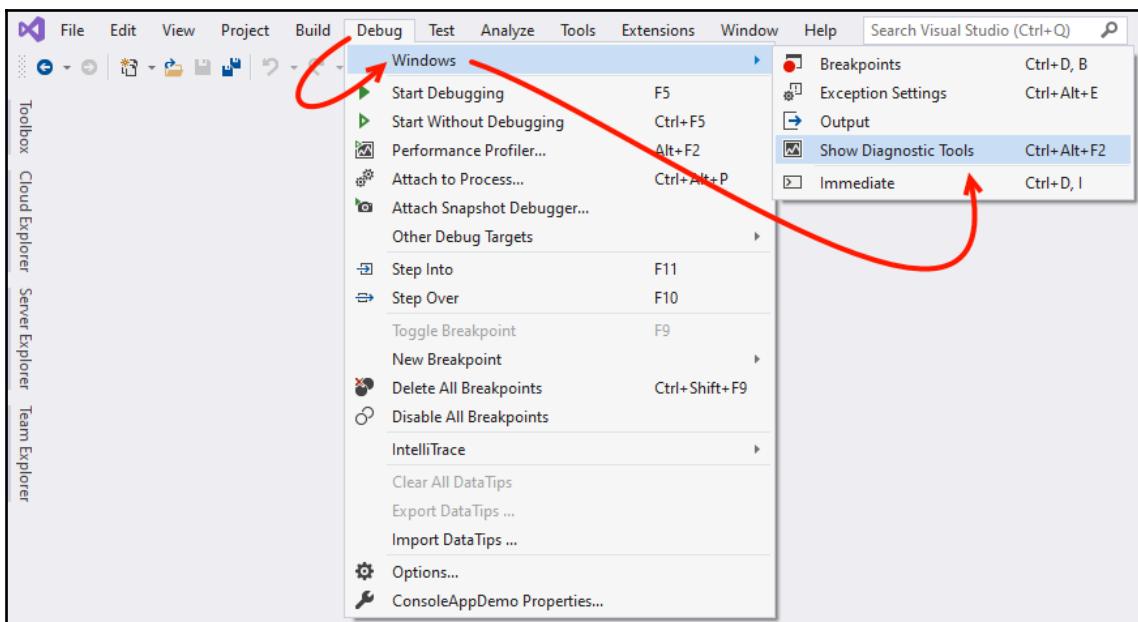
employees[0].FirstName + " " + employees[0].LastName
"Kunal Chowdhury"
```

You can also evaluate an expression by defining it in the **Immediate Window** if the same is available within the same debugging context. Next, we move on to the **Diagnostics Tools** of Visual Studio.

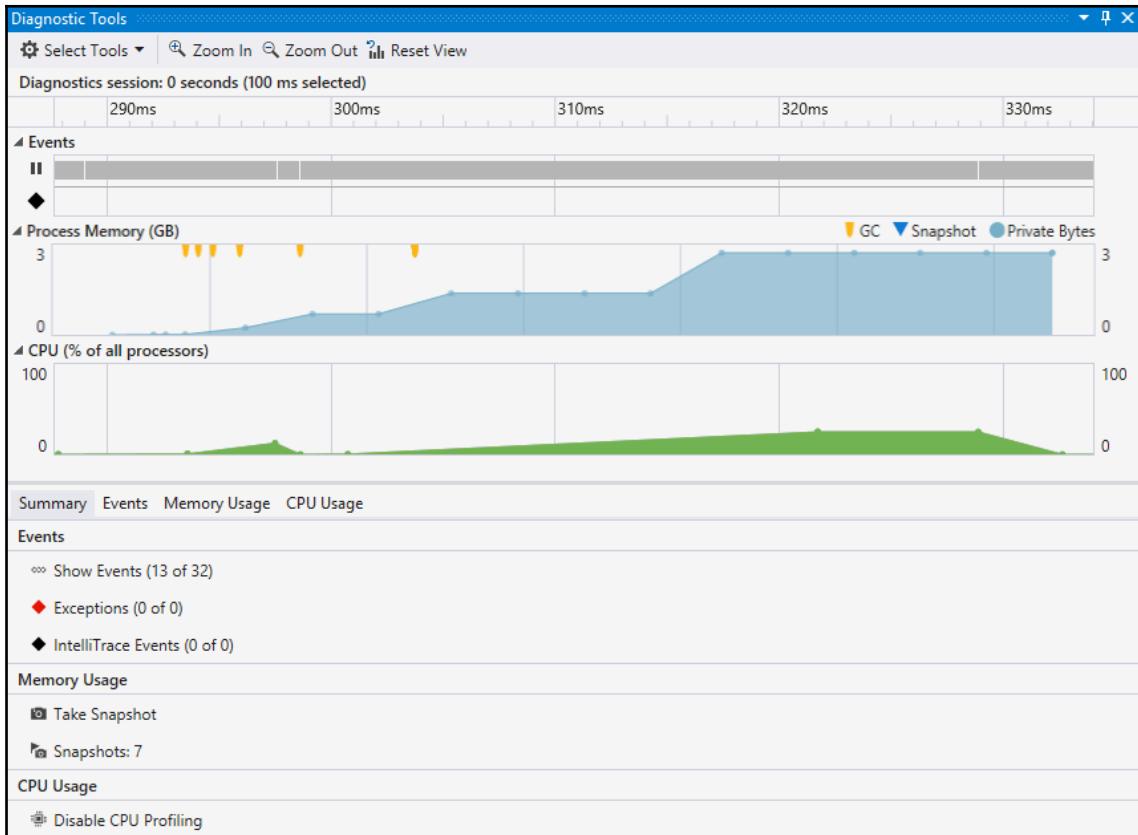
Using the Visual Studio Diagnostics Tools

The **Diagnostics Tools** of Visual Studio provide you with historical information about your application in a debugging session. Along with Visual Studio 2013, Microsoft first introduced the **Performance and Diagnostics** hub, which changed over time and was relaunched as **Diagnostics Tools** in Visual Studio 2015 with more limited options than the version currently available in Visual Studio 2019.

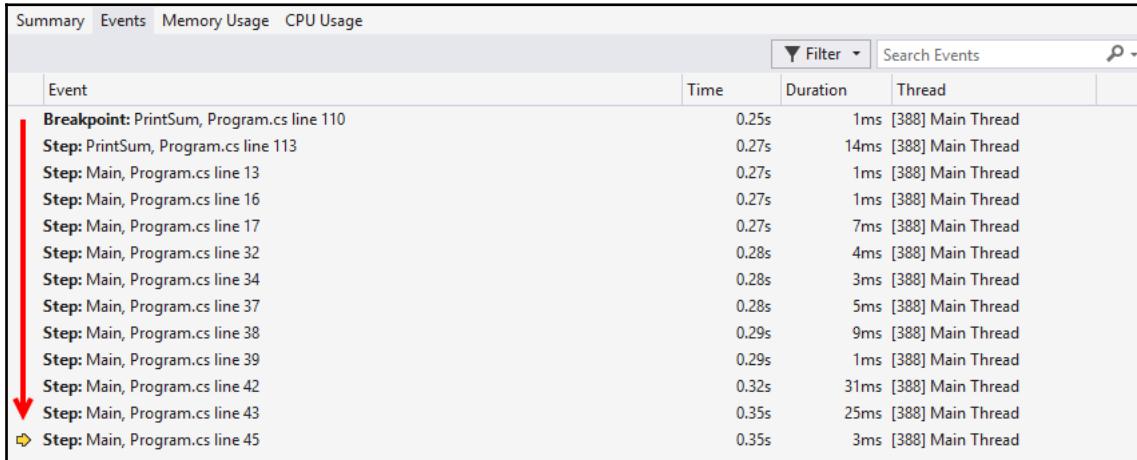
When you start a debugging session, the **Diagnostics Tools** window will automatically launch and appear side-by-side with your code window. If it is unavailable, you can launch it from the Visual Studio **Debug | Windows | Show Diagnostics Tools** menu or, press **Ctrl + Alt + F2**. The following screenshot shows this:



This window shows you detailed historical information about your application, called PerfTips, in the events graph and events table. It also allows you to correlate execution time with the memory and CPU utilization graph. You can take snapshots of the current memory utilization to enable/disable the CPU profiling in this screen:



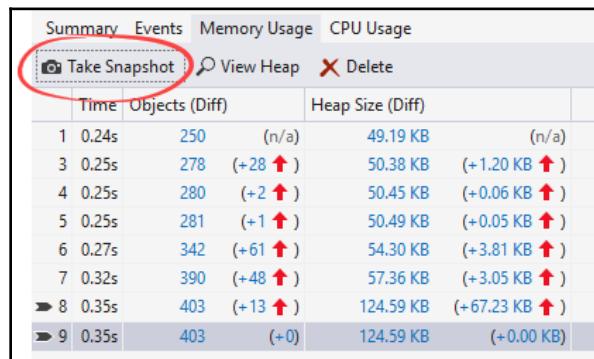
In the preceding screenshot, check how the memory was utilized over time. The yellow arrowhead indicates the time when the **Garbage Collector** was called, either automatically by the system or forcefully from the code:



Event	Time	Duration	Thread
Breakpoint: PrintSum, Program.cs line 110	0.25s	1ms	[388] Main Thread
Step: PrintSum, Program.cs line 113	0.27s	14ms	[388] Main Thread
Step: Main, Program.cs line 13	0.27s	1ms	[388] Main Thread
Step: Main, Program.cs line 16	0.27s	1ms	[388] Main Thread
Step: Main, Program.cs line 17	0.27s	7ms	[388] Main Thread
Step: Main, Program.cs line 32	0.28s	4ms	[388] Main Thread
Step: Main, Program.cs line 34	0.28s	3ms	[388] Main Thread
Step: Main, Program.cs line 37	0.28s	5ms	[388] Main Thread
Step: Main, Program.cs line 38	0.29s	9ms	[388] Main Thread
Step: Main, Program.cs line 39	0.29s	1ms	[388] Main Thread
Step: Main, Program.cs line 42	0.32s	31ms	[388] Main Thread
Step: Main, Program.cs line 43	0.35s	25ms	[388] Main Thread
Step: Main, Program.cs line 45	0.35s	3ms	[388] Main Thread

To view all of the events that were performed while you were debugging the context, switch to the **Event** tab. It gives you a new IntelliTrace experience in Visual Studio and saves your valuable debugging time. It captures additional events and useful information about the execution of your application. Hence, it allows you to identify the potential root causes of any issues. When an exception happens, it also maintains the history of the events where it occurred.

When you want to know more about a specific event, click on an item from the list and activate **Historical debugging** to set the debug window back to a time when the event occurred. You can then see the call stack, the values of local variables, and other important information that was available at the time when the event occurred:



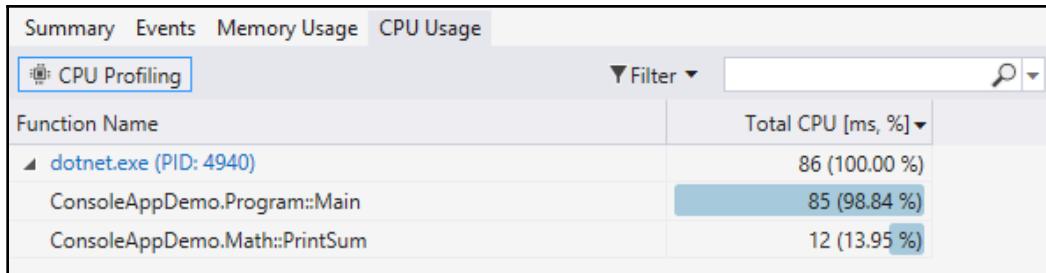
Time	Objects (Diff)	Heap Size (Diff)
1 0.24s	250 (n/a)	49.19 KB (n/a)
3 0.25s	278 (+28 ↑)	50.38 KB (+1.20 KB ↑)
4 0.25s	280 (+2 ↑)	50.45 KB (+0.06 KB ↑)
5 0.25s	281 (+1 ↑)	50.49 KB (+0.05 KB ↑)
6 0.27s	342 (+61 ↑)	54.30 KB (+3.81 KB ↑)
7 0.32s	390 (+48 ↑)	57.36 KB (+3.05 KB ↑)
8 0.35s	403 (+13 ↑)	124.59 KB (+67.23 KB ↑)
9 0.35s	403 (+0)	124.59 KB (+0.00 KB)

To find out more about memory utilization, navigate to the **Memory Usage** tab. Here, you need to take snapshots of the current memory usages by clicking the **Take Snapshot** button, as shown in the preceding screenshot.

Using this tab, you can monitor the memory usage (increased/decreased) and identify memory issues while you are debugging your code. Clicking on the individual links in each item will give you more details about the snapshot and heap objects, as shown in the following screenshot:

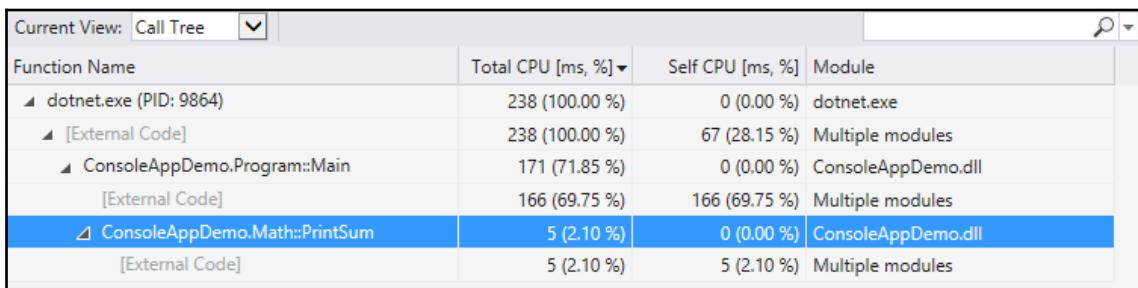
Snapshot #1 Heap dotnet.exe (1.72s)				
Managed Memory (dotnet.exe)		Compare to: Select baseline		
Object Type	Count	Size (Bytes)	Inclusive Size (Bytes)	
AppDomain	1	19,896	21,088	
OutOfMemoryException	2	6,064	6,064	
CultureData	2	2,728	2,728	
StreamWriter	1	1,464	1,720	
NumberFormatInfo	1	1,056	1,056	
CalendarData	1	960	960	
Dictionary<String, Object>	2	536	536	
AppDomainSetup	1	512	728	
RuntimeType	10	480	480	
ThreadAbortException	2	304	304	
CultureInfo	2	304	1,976	
Dictionary<String, CultureData>	1	280	304	
Exception	1	152	152	
StackOverflowException	1	152	152	
ExecutionEngineException	1	152	152	
StringBuilder	1	144	144	
EncoderReplacementFallback	4	128	128	
DecoderReplacementFallback	4	128	128	
CompareInfo	2	112	112	
UnicodeEncoding	2	112	240	
UTF8Encoding	2	96	224	
TextInfo	1	88	88	
Paths to Root Referenced Types				
Object Type	Reference Count			
OutOfMemoryException	2			
OutOfMemoryException [Strong Handle]				

The **CPU Usage** tab provides you details about CPU utilization and tells you how many CPU resources were used by your code. You can correlate this information with the CPU usage graph shown in the tool window and find out the spike where a higher utilization took place:



You should start **CPU Profiling** to grab this information about CPU utilization by the program currently in the debugging context.

If you spot a potential issue when debugging, you can check this tab to get the per-function breakdown to identify the problem. Double-clicking on any item will give you details about the call tree, as shown in the following screenshot:



This provides the following details about each function that is available in the call tree:

- **Total CPU %:** This provides you with the CPU activity in the selected function and the functions it called. The information is provided as a percentage value.
- **Self CPU %:** This provides the percentage of CPU activity in the selected function, but not in the functions where it has been called.
- **Module:** This provides the name of the module where the call has been made.

By investigating these column values, you can easily identify the code block where CPU utilization was higher and, based on that, optimize your code.

Summary

In this chapter, we learned about the debugger execution steps and how to debug C# code using breakpoints. Here, we covered organizing breakpoints and setting conditional breakpoints, hit counters, breakpoint filters, actions, and labels.

We also covered how to manage breakpoints in code using the **Breakpoints** window of the Visual Studio debugger tools and discussed how to import/export them.

Apart from these, we discussed DataTips in detail. This included pinning/unpinning DataTips, a discussion of various Watch windows and visualizers, how to import/export DataTips, and the usages of the debugger display attribute.

Finally, we discussed the Immediate window and Visual Studio Diagnostics Tools.

In the next chapter, we will cover testing applications using Visual Studio 2019. There, we will discuss how to use the Live Unit Testing feature to automatically run the impacted unit tests to visualize the result and code coverage in the background, as and when you are editing the code.

8

Live Unit Testing with Visual Studio 2019

In computer programming, **unit testing** is a software development and testing process by which the smallest testable parts of source code, called **units**, are tested to determine whether they are performing as per design. Unit testing is generally part of an automation process, but you can run it manually too.

Visual Studio 2019 supports **Live Unit Testing**, which was first introduced with Visual Studio 2017 and is available in the Enterprise Edition for C#/VB.NET projects.

In this chapter, to keep a baseline on understanding the basics of unit testing processes, we are only going to discuss the Live Unit Testing feature, and we will cover the following points:

- Overview of Live Unit Testing in Visual Studio 2019
- Configuring Visual Studio 2019 for Live Unit Testing
- Live Unit Testing with Visual Studio 2019
- Navigating to failed tests

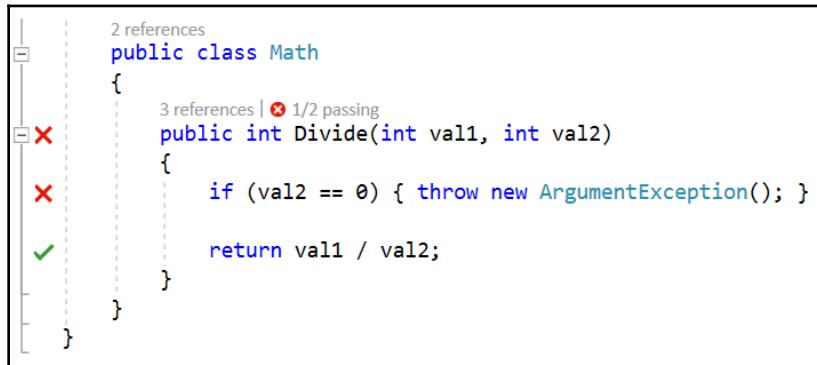
Technical requirements

To follow along with this chapter, you will need Visual Studio 2019 Enterprise Edition installed on your system. A basic understanding of the IDE, the C# language, and unit testing is also recommended.

Overview of Live Unit Testing in Visual Studio 2019

The Live Unit Testing feature in Visual Studio 2019 allows you to quickly see the code coverage details and the unit test case execution results without leaving the code editor window. The Live Unit Testing feature automatically runs the impacted unit tests in the background as we edit the source code, and, in real time, it visualizes the unit testing result and coverage within the editor window.

It currently supports C#/VB.NET projects that target .NET Framework and .NET Core, but only in Visual Studio 2019 Enterprise Edition. When enabled, the unit test results and visualization of the code coverage results appear on a line-by-line basis in the editor, as shown in the following screenshot:



The live feedback notifies us in real time of the change that has broken the program. This way, it helps you to maintain the quality of the code by making the tests pass as you make any changes for a new feature or a bug fix.

In the preceding screenshot, check the left side bar, which has green tick marks and red × icons and provides live notifications of the unit test result.

Unit testing framework support

At present, Live Unit Testing in Visual Studio 2019 Enterprise Edition only supports three popular unit testing frameworks: xUnit, NUnit, and MSTest. However, a few supportive version specifications do exist, which you must meet for the Visual Studio unit testing adapter and the unit testing framework, as mentioned here:

- **xUnit.net:** Framework version 1.9.2 or higher, adapter version 2.2.0 or higher
- **NUnit:** Framework version 3.5.0 or higher, NUnit3Test adapter version 3.5.1 or higher
- **MSTest:** Framework version 1.0.5 or higher, MSTest test adapter version 1.1.4 or higher

If you have an older version of the test framework references and/or adapter version in your existing projects, make sure to remove those and add the new references for Live Unit Testing.

Understanding the coverage information shown in the editor

When you enable Live Unit Testing, the Visual Studio code editor provides you with sufficient information to notify you of the code coverage changes as you work. It also provides you with real-time unit test results by showing some symbolic icons in the left sidebar. This is known as coverage visualization and you can visualize it on a line-by-line basis in the editor. Here are a few points that you should know about the coverage information:

- The blue dash () indicates that the line of executable code does not have any test coverage.
- The green tick mark () indicates that the line of executable code is covered by unit test case passes.
- The red cross mark () indicates that the line was executed, but at least one unit test case failed.
- The blue dash with a clock icon () indicates that the line of executable code does not have any test coverage at this moment, but it is processing the changes that have been made and is going to update the visualization with refreshed data.

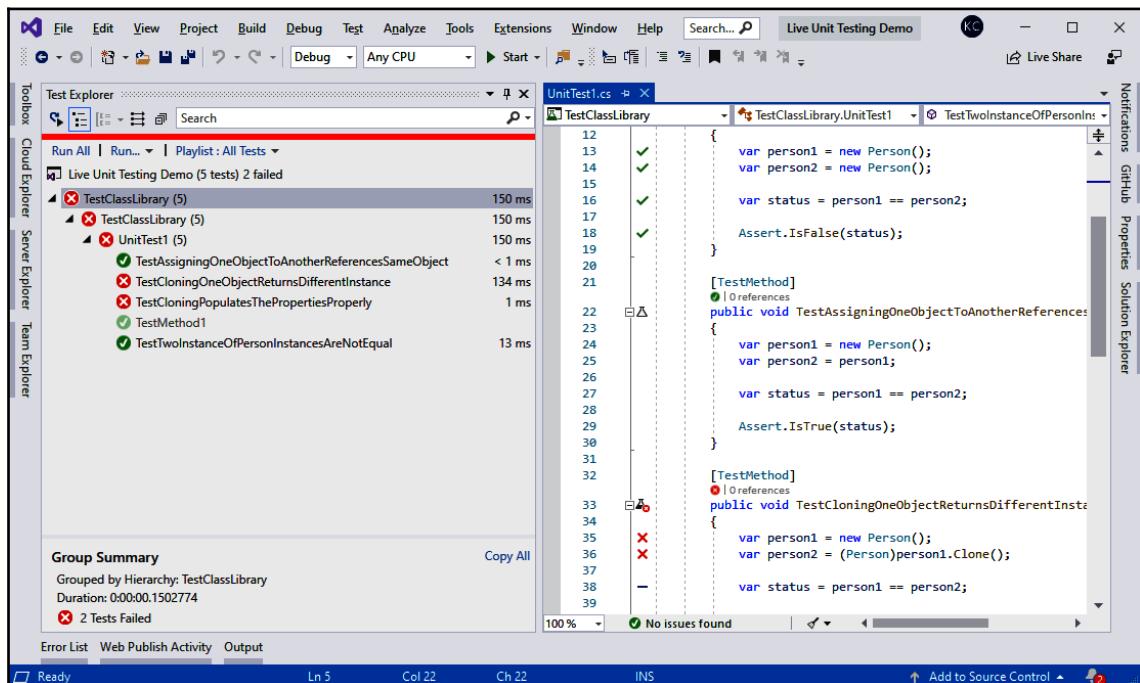
- The green tick mark with a clock icon (⌚) indicates that the data is not up to date for the test case that passed earlier.
- The red cross mark with a clock icon (⌚) denotes a failed test case for which the data is not yet up to date. When processing the changes, the visualization will update automatically with new data.

Integration of Live Unit Testing in Test Explorer

Visual Studio 2019 provides a seamless coding and testing environment. The **Live Unit Testing** and **Test Explorer** functionalities inside Visual Studio are synchronized to give a proficient way to perform unit test execution throughout the project/solution.

While you are modifying the existing code, the Live Unit Testing process executes in the background for the impacted test cases for which you are changing the code, and, based on that, it lists the result in the **Test Explorer** automatically and highlights it in a bright color.

When some non-impacted test cases are available, their listings are grayed out, as shown here for `TestMethod1`:



As we now have a basic overview of Live Unit Testing, let's start configuring Visual Studio 2019 so that we can begin working with Live Unit Testing.

Configuring Visual Studio 2019 for Live Unit Testing

The Live Unit Testing component comes with **Visual Studio 2019 Enterprise Edition** only. To use it, you must first install the component from the Visual Studio installer and configure it as an optional setting.

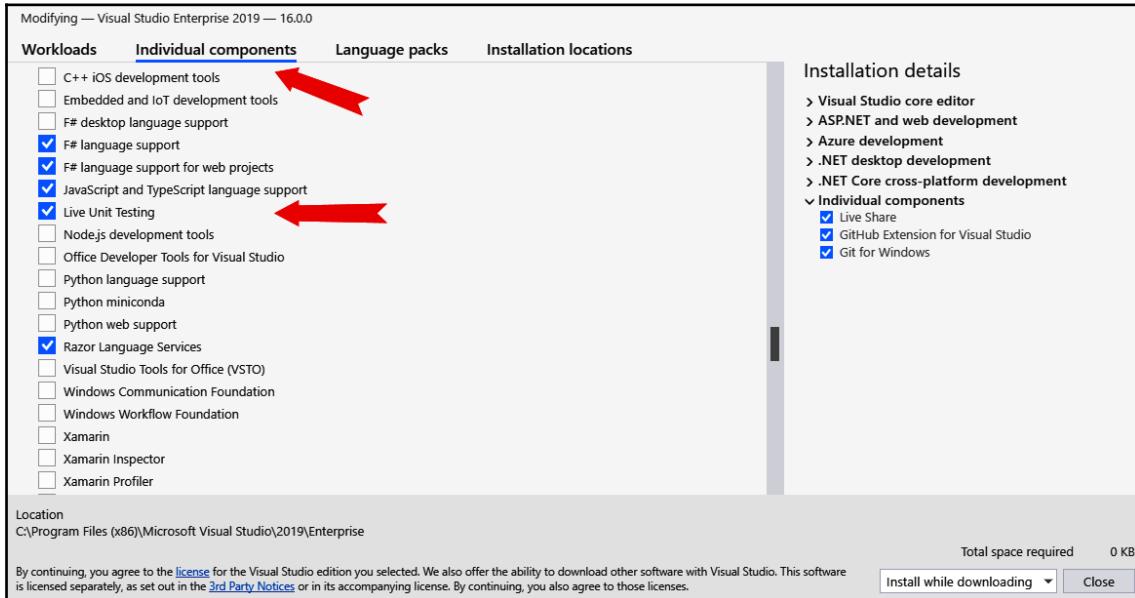
You can start/pause/stop the Live Unit Testing module at any point. You can also include/exclude unit test cases to be run as part of Live Unit Testing.

In this section, we are going to discuss all of these topics. Let's first start with the installation of the component.

Installing the Live Unit Testing component

To install the Live Unit Testing component in an existing installation of Visual Studio 2019 Enterprise Edition, run the installer and modify the existing installation.

Now, navigate to the **Individual components** tab. Then, scroll down to the **Development activities** section and select the **Live Unit Testing** component, as shown in the following screenshot:

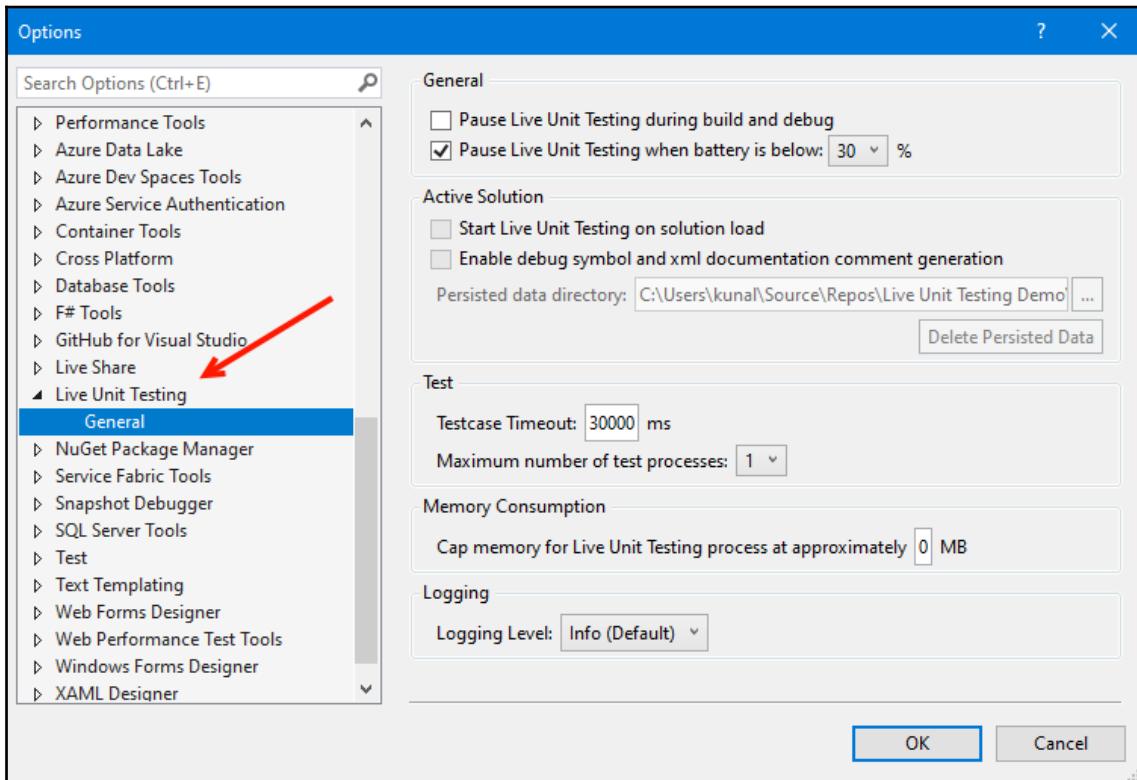


If Live Unit Testing is already checked, then this means the component is already installed. Based on your selection, click on either the **Modify** or **Close** buttons. The component will not take up more than 3 MB of installation space.

General settings of Live Unit Testing in Visual Studio

There are a few configurable options available for Live Unit Testing, which you can optionally modify. To do so, navigate to **Options** from the **Tools** menu. Now, from the left-hand pane, select **Live Unit Testing | General**.

In the right-hand pane, there are a couple of settings available, as shown in the following screenshot:



From the preceding screenshot, we understand the following:

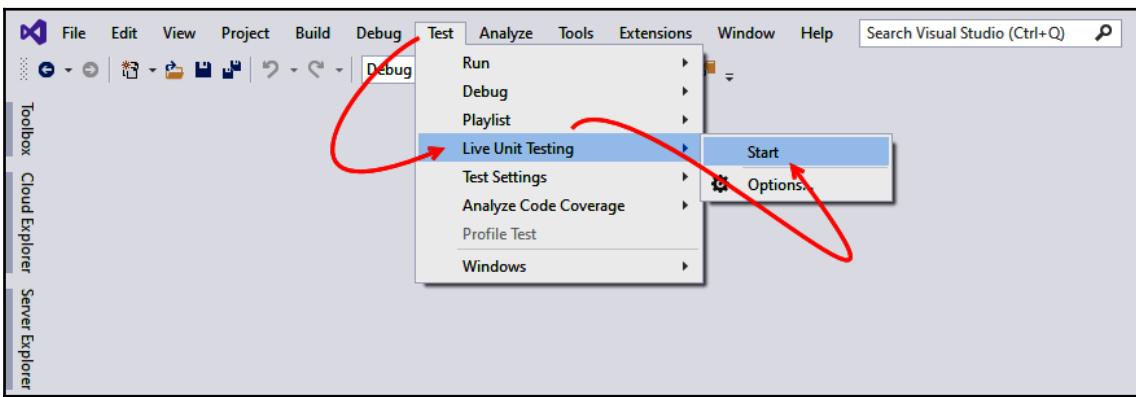
- The first option enables you to control whether you can pause the Live Unit Testing process during the build and/or when debugging is in progress.
- The next option allows you to pause the Live Unit Testing process when you have a low battery, which you can configure by selecting the value from the adjacent dropdown. By default, this is set to 30%.
- The third option enables you to control whether Live Unit Testing runs automatically on the solution load.

- The fourth option enables you to control whether you can enable/disable the debug symbol and the XML documentation comment generation during this process.
- On this screen, you can also control the test case timeout in milliseconds, which is **30000 ms** (30 seconds) by default. You can also set the number of test processes that the Live Unit Testing process will create.

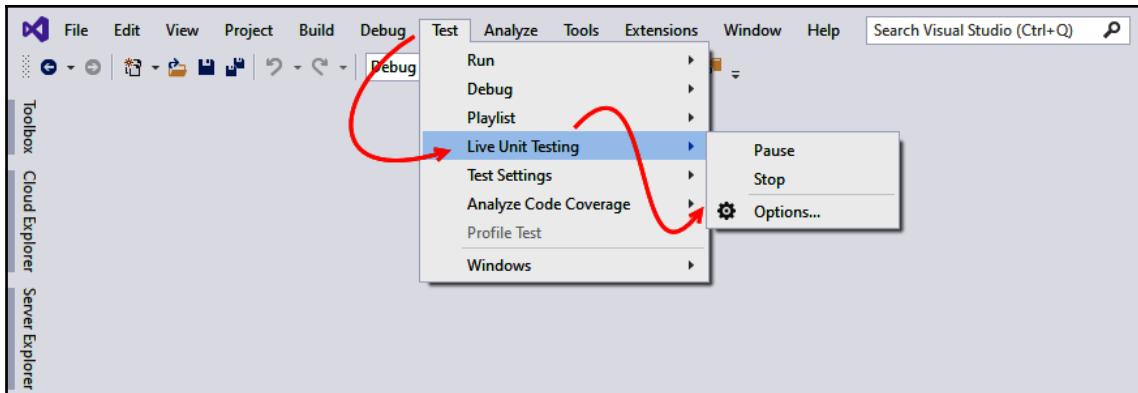
Apart from these, you can limit the memory consumption of Live Unit Testing processes in MB. You can also set the logging level of Live Unit Testing. When it is set to **None**, there won't be any automatic logging performed; when it is set to **Error** or **Info**, only error messages or informational messages will get logged based on the selection. Set it to **Verbose** if you want to log every detail. The default logging level is **Info**.

Starting and pausing Live Unit Testing

To enable Live Unit Testing to work, navigate to **Test | Live Unit Testing** and click on **Start**, as shown in the following screenshot:



In some cases, you may want to pause or stop the automatic execution of the Live Unit Testing process. Visual Studio 2019 allows you to temporarily pause or stop it. To invoke any of the previously mentioned commands, navigate to the **Test | Live Unit Testing** menu and click on the respective menu items, as shown in the following screenshot:



When you pause the Live Unit Testing process, there won't be any coverage visualization in the Visual Studio editor as the process will go into a temporary suspended state. When you want to resume, click on **Continue** and it will do the necessary work to update the visualization as soon as it can.

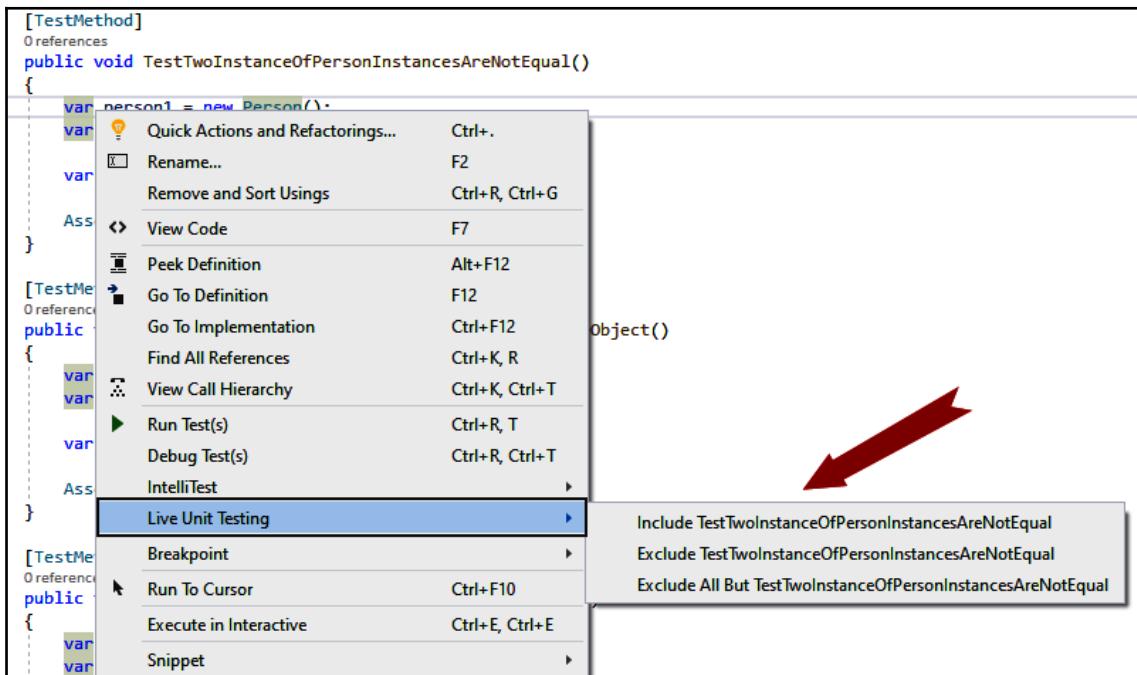
To completely stop the Live Unit Testing process, click on **Stop**. This will remove all the collected data from the process. When you start the process again, it takes longer to load the data and update the visualization.

Including and excluding test methods/projects

Live Unit Testing always runs in the background to give you real-time data on the unit testing results and code coverage. In some cases, however, you may not want to run all the cases. This may be due to some projects, classes, or methods in a solution that you haven't modified for a prolonged period. Unnecessarily running all those cases is just overkill.

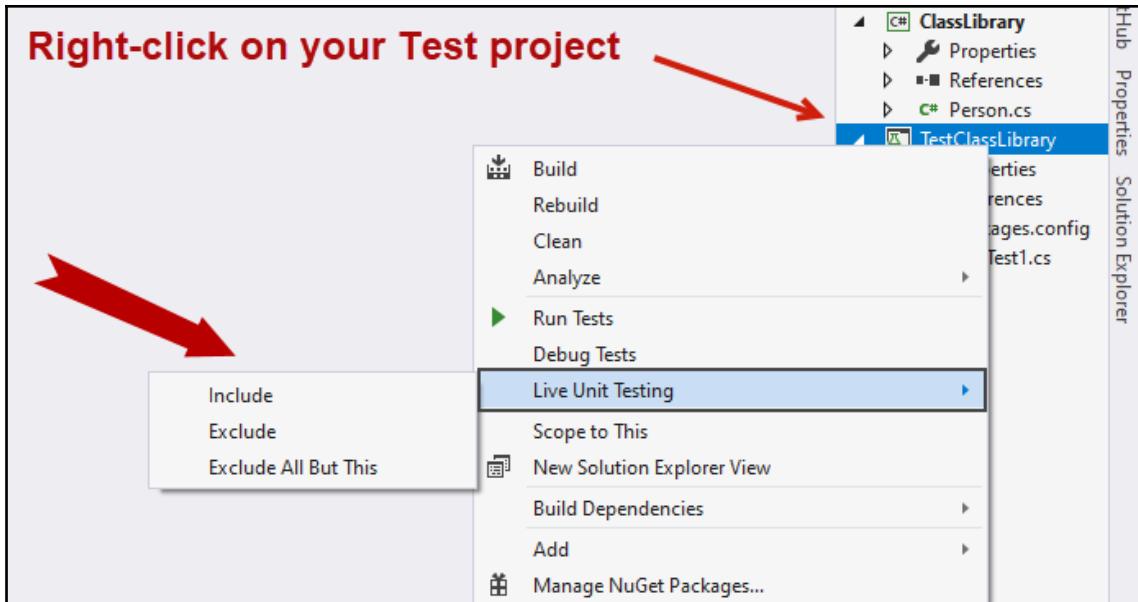
Visual Studio 2019 provides you with an option to selectively include/exclude a specific method, class, or test project. You can right-click inside a method, and, from the context menu, select **Live Tests** and then **Include [Class OR Method NAME]**, **Exclude [Class OR Method NAME]**, or **Exclude All But [Class OR Method NAME]**.

This will internally mark the selected method to include or exclude from Live Unit Testing as per your choice, and save the information in user settings. When you reopen the solution, the same information will be remembered by Visual Studio. Here's a screenshot for your quick reference:



If you want to include/exclude a specific class, do the same steps by right-clicking inside a class, but outside a method. To include/exclude an entire file, right-click outside the class but within the file. All test cases in that file will either be included or excluded based on your choice.

You can also individually include/exclude a test project. Right-click on that project and, from the context menu, select **Live Tests** and then the **Include**, **Exclude**, or **Exclude All But This** options, as shown in the following screenshot:



As we have now learned about the configurations of Visual Studio 2019 Enterprise Edition to run Live Unit Testing, let's start with some simple demonstrations of how to execute them.

Live Unit Testing with Visual Studio 2019

As we have already discussed the feature, its configuration settings, and how to start and stop Live Unit Testing, let's begin demonstrating it in action with a live example.

Getting started with configuring the testing project

To get started with Live Unit Testing demonstration, we will create a project with some code and then write the unit testing cases. The steps to begin with are mentioned as follows:

1. Open your Visual Studio 2019 (Enterprise Edition) IDE and create a new project of the **Class Library (.NET Framework)** type.

2. Create the project by giving it a name (in this example, we name it `ClassLibrary`).
3. Now, create a class named `Person` and inherit it from the `ICloneable` interface (just for this demonstration).
4. Implement the interface to generate a `Clone()` method that will, by default, throw `NotImplementedException`. Leave it as it is. We are going to revisit this method later.
5. Add a few string properties named `ID`, `Name`, and `Address`. Here's the implementation of the class, for reference:

```
public class Person : ICloneable
{
    public string ID { get; set; }

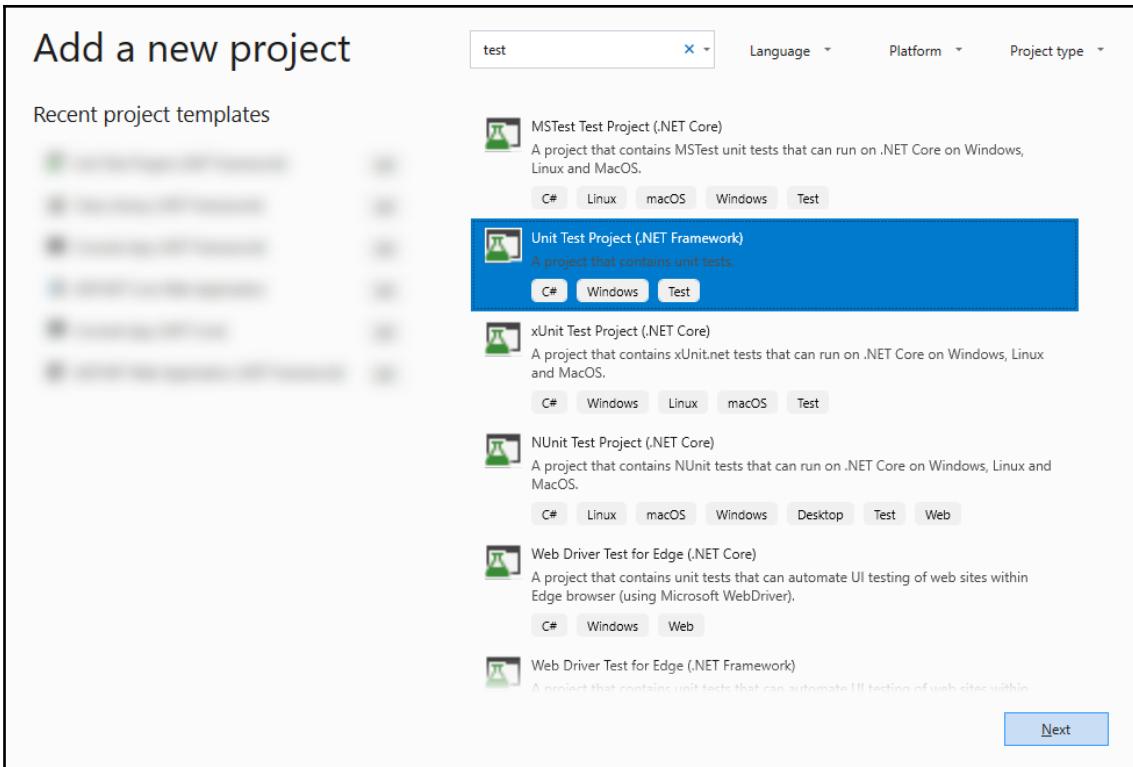
    public string Name { get; set; }

    public string Address { get; set; }

    public object Clone()
    {
        throw new NotImplementedException();
    }
}
```

As we have our `Person` class in our application project, let's create the unit testing project for testing and code coverage.

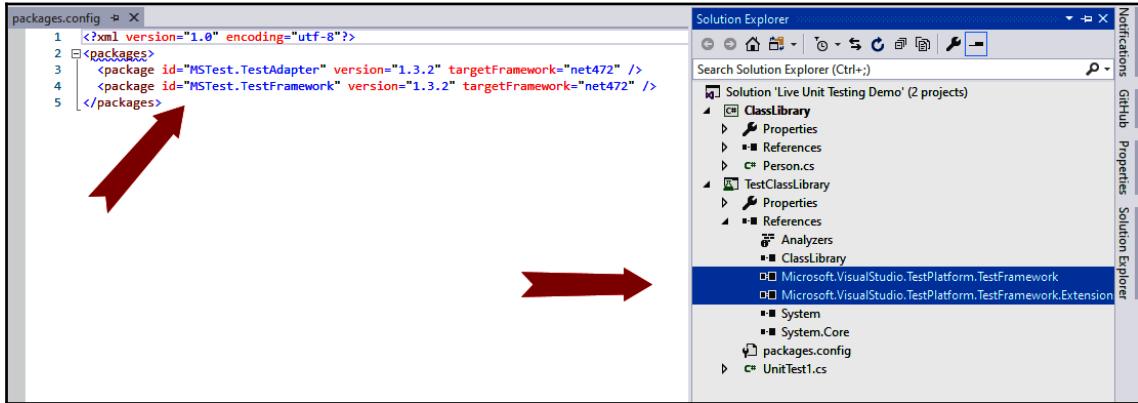
6. Right-click on the solution file in **Solution Explorer** and navigate to **Add | New Project...** from the Visual Studio context menu. This will open the **Add New Project** dialog on the screen.
7. Select the **Unit Test Project (.NET Framework)** project type, give it a name (for example, `TestClassLibrary`) on the next screen, and click **Create** to create the unit testing project in the same solution:



8. Once the unit testing project is created in the solution, add the assembly reference of the main project into it in order to refer the classes available from there.
9. Right-click on the unit testing project and, from the context menu, select **Add | Reference**.
10. From the **Reference Manager** dialog window, navigate to **Projects | Solution** and then select the projects that you want to add as a reference.
11. Finally, click on **OK** to continue to complete the test project setup.

Understanding the packages.config file

Once you add the project reference to the unit testing project, the IDE adds a few additional assemblies to the testing project. These are used to add support to the testing framework. Let's look into this:



If you expand the **References** folder of the **TestClassLibrary** project, you will see two assembly references, **Microsoft.VisualStudio.TestPlatform.TestFramework**, and **Microsoft.VisualStudio.TestPlatform.TestFramework.Extensions**, along with the reference of our main project. These two DLL references are the core files needed for the unit testing project to function.

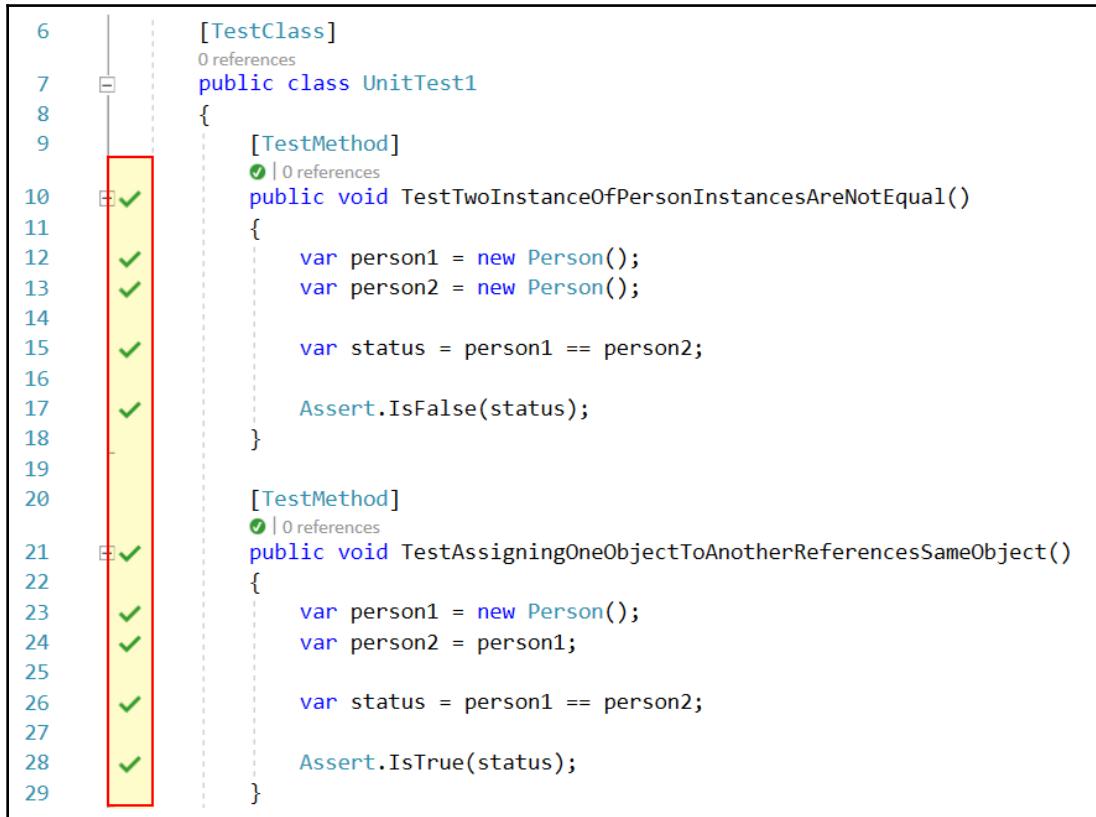
In the same project, you will find a file named **packages.config**, which defines the packages required for the unit testing to create the test adapter with the referenced project. The file content will look like the following XML content, with **MSTest.TestAdapter** and **MSTest.TestFramework** defined in it:

```
<?xml version="1.0" encoding="utf-8"?>
<packages>
  <package id="MSTest.TestAdapter" version="1.3.2"
targetFramework="net472" />
  <package id="MSTest.TestFramework" version="1.3.2"
targetFramework="net472" />
</packages>
```

As we have learned about the **packages.config** file, let's now move to the next section to begin with the Live Unit Testing steps.

Live Unit Testing with an example

Let's open the `UnitTest1.cs` file and add the following two test methods to it. The first method will create two instances of the `Person` class and check to ensure that both instances are different. The second method will create one instance of the `Person` class, assign it to another variable, and check to ensure that both instances are equal. The following screenshot shows this:



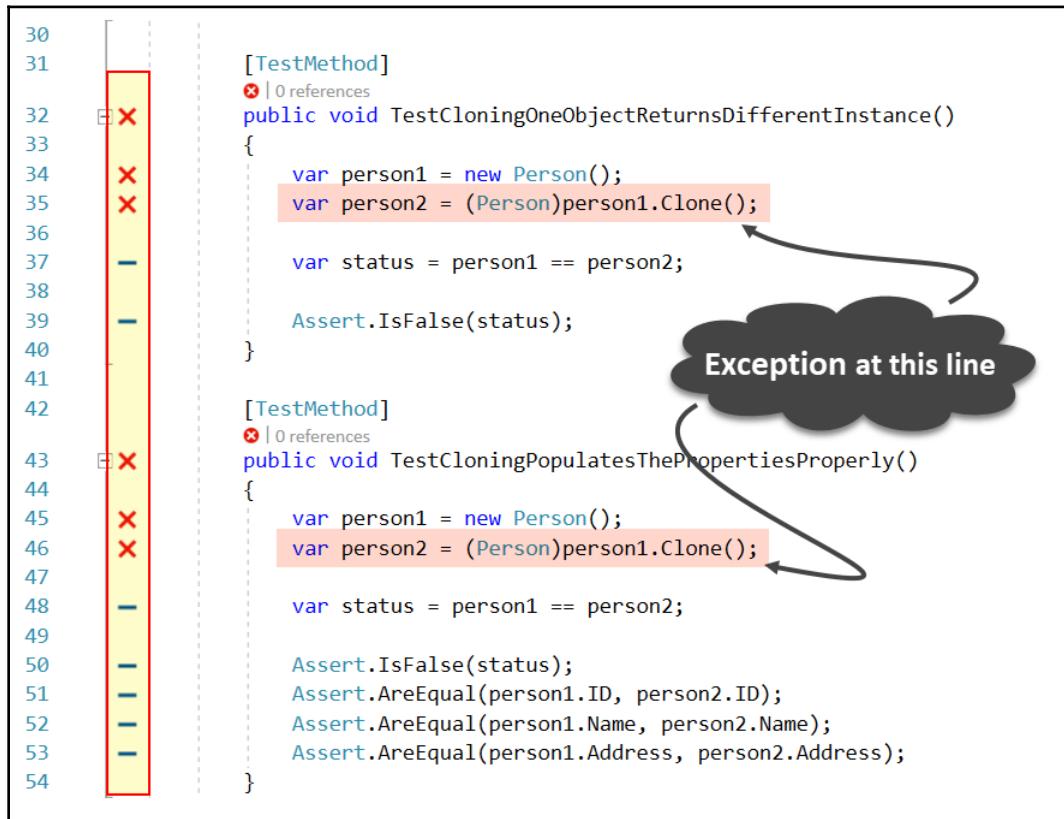
```
6 [TestClass]
7 public class UnitTest1
8 {
9     [TestMethod]
10    public void TestTwoInstancesOfPersonInstancesAreNotEqual()
11    {
12        var person1 = new Person();
13        var person2 = new Person();
14
15        var status = person1 == person2;
16
17        Assert.IsFalse(status);
18    }
19
20    [TestMethod]
21    public void TestAssigningOneObjectToAnotherReferencesSameObject()
22    {
23        var person1 = new Person();
24        var person2 = person1;
25
26        var status = person1 == person2;
27
28        Assert.IsTrue(status);
29    }
}
```

When you start writing the code line by line, you will see that the Live Unit Testing process will run in the background and provide the status of the code coverage and the test result in the left side bar. In the previous screenshot, every line generates a green tick mark (✓), indicating that the written test cases passed and all the lines have been covered.

If you are unable to see the live unit test working, refer to the *Configuring Visual Studio 2019 for Live Unit Testing* section of this chapter.

When any unit test method has failed, the icon will change to a red cross mark () up to the line where it failed. Refer to the *Overview to Live Unit Testing in Visual Studio 2019 | Understanding the coverage information shown in editor* section of this chapter for more details about the various icons shown in the editor.

Now, let's create another two test methods that will call the `Clone()` method of the class. Here, the Live Unit Testing process will break at the same line where the method has been called and notify us with a red cross mark, as you can see in the following screenshot:



```
30
31 [TestMethod]
32     public void TestCloningOneObjectReturnsDifferentInstance()
33     {
34         var person1 = new Person();
35         var person2 = (Person)person1.Clone();
36
37         var status = person1 == person2;
38
39         Assert.IsFalse(status);
40     }
41
42 [TestMethod]
43     public void TestCloningPopulatesThePropertiesProperly()
44     {
45         var person1 = new Person();
46         var person2 = (Person)person1.Clone();
47
48         var status = person1 == person2;
49
50         Assert.IsFalse(status);
51         Assert.AreEqual(person1.ID, person2.ID);
52         Assert.AreEqual(person1.Name, person2.Name);
53         Assert.AreEqual(person1.Address, person2.Address);
54 }
```

In the preceding example, the `Clone()` method call breaks as it throws `NotImplementedException`. The rest of the test method lines will be decorated with a blue dash mark (), as those are all unreachable code.

Let's revisit the `Person` class, where you will see the following status notification from the Live Unit Testing framework:

```
7 references
public class Person : ICloneable
{
    2 references | ✘ 0/1 passing
    public string ID { get; set; }

    2 references | ✘ 0/1 passing
    public string Name { get; set; }

    2 references | ✘ 0/1 passing
    public string Address { get; set; }

    2 references | ✘ 0/2 passing
    public object Clone()
    {
        throw new NotImplementedException();
    }
}
```

Now, move ahead and implement the body of the `Clone()` method, which will now return an object by calling the `MemberwiseClone()` method. Momentarily, the Live Unit Testing process will execute automatically and show you the status in the left side bar of the editor. This time, everything will be represented with green tick marks:

```
7 references
public class Person : ICloneable
{
    ✓ 2 references | 1/1 passing
    public string ID { get; set; }

    ✓ 2 references | 1/1 passing
    public string Name { get; set; }

    ✓ 2 references | 1/1 passing
    public string Address { get; set; }

    ✓ 2 references | 2/2 passing
    public object Clone()
    {
        return MemberwiseClone();
    }
}
```

Let's navigate to the test class. You will see that the code coverage is now 100% for those two methods. Also, all the test methods passed. This way, the framework ensures that the cases execute while you write the code, and the framework gives you live feedback on the changes that you made, thereby reducing the extra effort of executing the test cases manually:

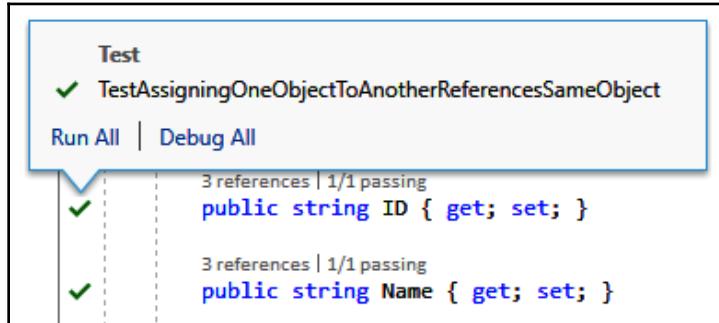
```
31 [TestMethod]
32     public void TestCloningOneObjectReturnsDifferentInstance()
33     {
34         var person1 = new Person();
35         var person2 = (Person)person1.Clone();
36
37         var status = person1 == person2;
38
39         Assert.IsFalse(status);
40     }
41
42 [TestMethod]
43     public void TestCloningPopulatesThePropertiesProperly()
44     {
45         var person1 = new Person();
46         var person2 = (Person)person1.Clone();
47
48         var status = person1 == person2;
49
50         Assert.IsFalse(status);
51         Assert.AreEqual(person1.ID, person2.ID);
52         Assert.AreEqual(person1.Name, person2.Name);
53         Assert.AreEqual(person1.Address, person2.Address);
54     }
```

Now that you are familiar with Live Unit Testing, let's jump into the next section where we will discuss how easy it is to quickly navigate failed tests in Visual Studio 2019.

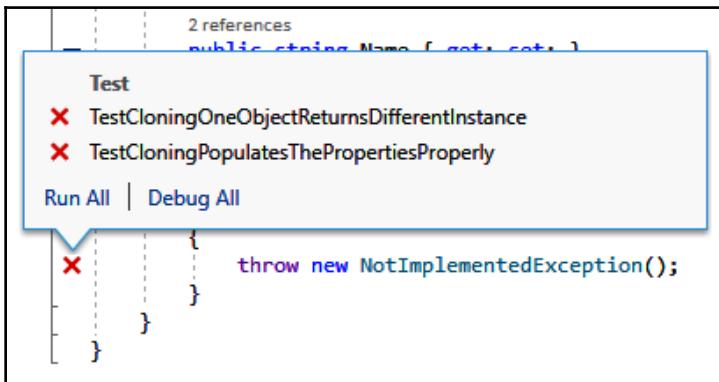
Navigating failed tests

Visual Studio 2019 allows you to quickly navigate to the failed tests and get insight's into them. You can click on '✓' or '✗' to see how many tests were being hit by a given line.

When you click on the green tick mark (✓), as shown in the following screenshot, Visual Studio lists the names of the test methods where the test case passed:



When you click on the red cross mark (✗), as shown in the following screenshot, Visual Studio lists the names of the test methods where the test case failed:



If you hover over a failed test in the tooltip, it provides additional information with StackTrace, including further insights into the nature of the failure.

Summary

In this chapter, we learned about the Live Unit Testing feature of Visual Studio 2019 Enterprise Edition. We discussed the supported unit testing framework and adapters for running Live Unit Testing inside the IDE. We also discussed the coverage information shown in the editor and integration with Test Explorer.

Then, we learned how to configure Visual Studio 2019 for Live Unit Testing. There, we discussed how to install the component by running the installer. We also discussed how to start/stop/pause the Live Unit Testing process and how to include any specific test methods/projects to show the real-time unit testing status and code coverage.

Later in this chapter, we demonstrated how to create a unit testing project, configure the unit testing framework, and perform it in real time with a simple example. Finally, we discussed the methods to navigate to failed tests and obtain more details from them.

In the next chapter, we will learn about source control integration in Visual Studio 2019. We will demonstrate this through the use of Git projects.

4

Section 4: Source Control

Source control is a component of software configuration management, source repositories, and version management systems. If you are building enterprise-level applications in a distributed environment, you must use it to keep your code in a safe vault. This section demonstrates the necessary steps to manage your code with versioning support in a source control repository.

The following chapter will be covered in this section:

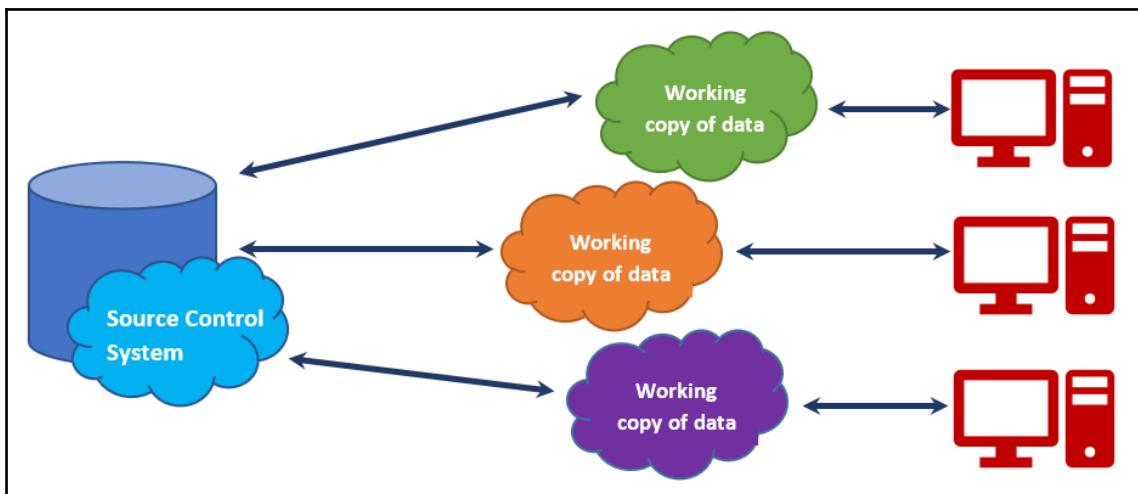
- Chapter 9, *Exploring Source Controls in Visual Studio 2019*

9

Exploring Source Controls in Visual Studio 2019

A **Source Code Control System (SCCS)** is a component of a source repository and version management system. If you are building enterprise-level applications in a distributed environment, you will want to keep your source code in a safe vault with easy-to-manage, easy-to-integrate, and easy-to-create versions of each check-in. A source control repository will help you to manage your code.

There are plenty of source control repositories on the market. Some of the most common repositories are Git, TFS, and SVN. Here's a small representation of how an SCCS works:



In general, repositories are hosted in on-premise environments or in cloud-hosted environments such as **Microsoft Azure DevOps**, **Team Foundation Server**, **GitHub**, and **BitBucket**. Developers connect to the remote repository and clone the code changes to their system in order to start coding. When the changes are complete and well tested, they push those changes to the remote repository.

In this chapter, we are going to learn about the following core points in order to use a source control repository such as **Git** in a Visual Studio 2019 environment:

- Installing Git for Visual Studio 2019
- Connecting to the source control servers
- Getting started with Git repositories
- Working with Git branches
- Working with changes, staging, and commits
- Syncing changes between local and remote repositories
- Working with pull requests for code review
- Working with Git commit history
- Undoing your changes
- Tagging your commits

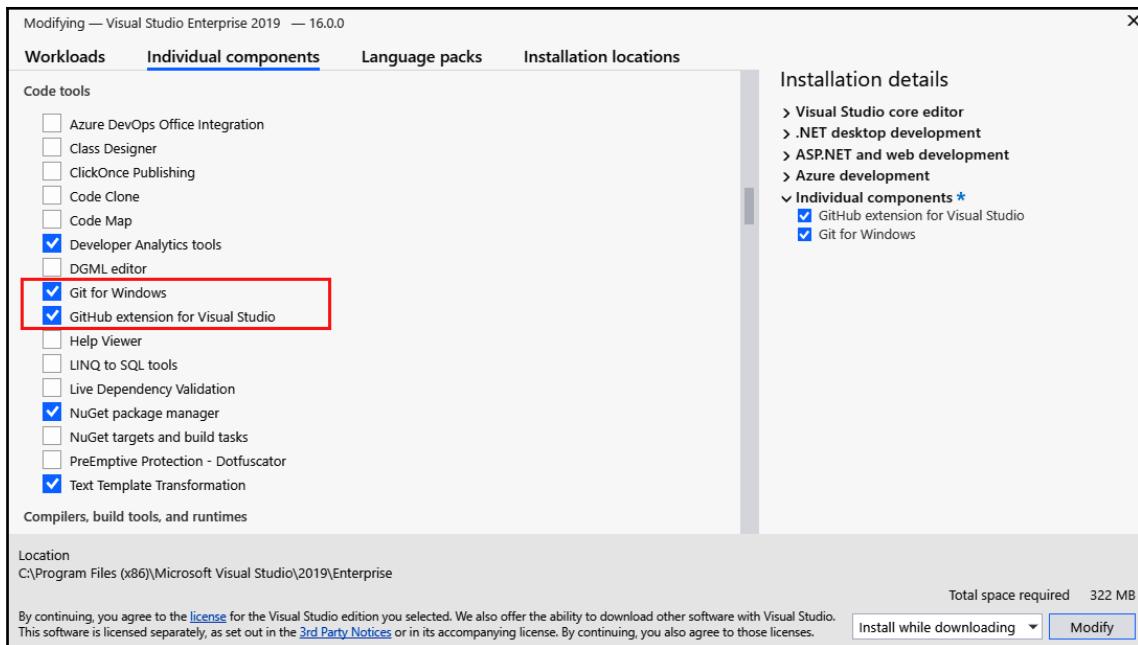
Technical requirements

To embark on this chapter, you need to have Visual Studio 2019 installed on your system. You also need Git for Visual Studio 2019, which you can install by using Visual Studio installer. A basic understanding of the IDE and the C# language is also recommended.

Installing Git for Visual Studio 2019

Git for Visual Studio 2019 comes as an optional component, and you need to manually install it to work with Git servers such as Team Foundation Services, GitHub, and BitBucket.

To install the Git plugin for Visual Studio, run the Visual Studio 2019 installer and click on **Modify**. Once the screen loads, navigate to the **Individual Components** tab, as shown in the following screenshot. Scroll down to the **Code tools** section and select **Git for Windows** and **GitHub extension for Visual Studio**. Click on the **Modify** button to continue with the installation:



Git for Windows will allow you to perform Git commands and access both local and remote Git repositories. If you are going to deal with GitHub, you must check **GitHub extension for Visual Studio** to have easy access to your repository.

If you are installing it from an offline installer, it will momentarily install the required components. In other cases, the installer will first download it from the Microsoft server and then proceed with the installation.

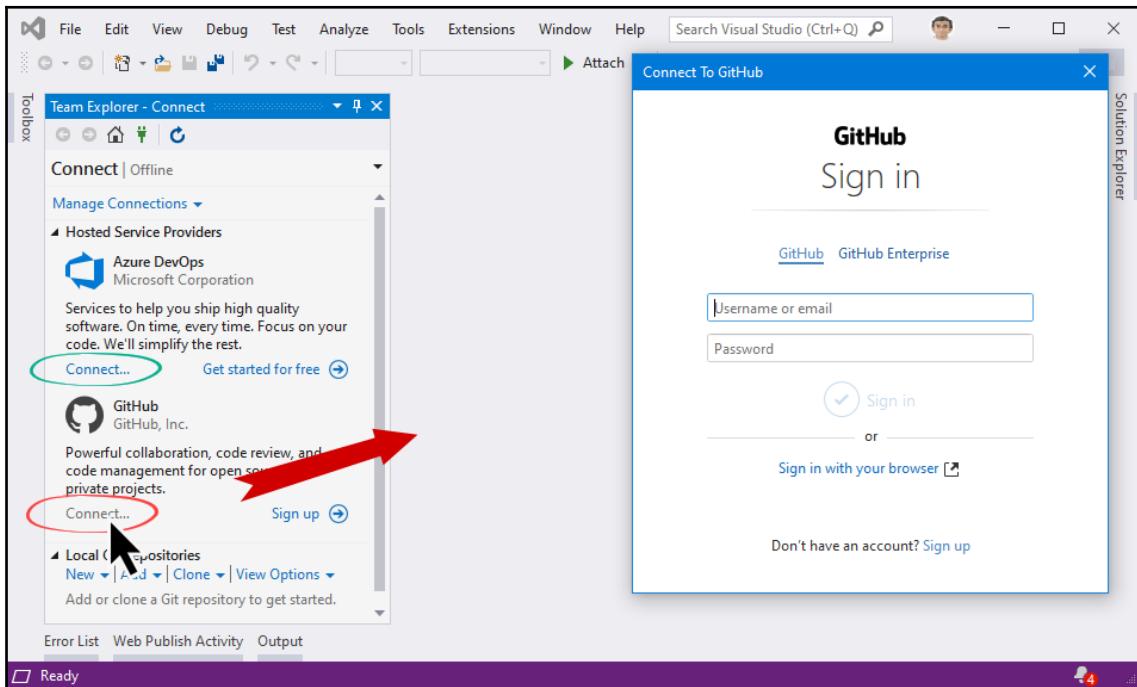
Once the installation succeeds, you are good to go with connecting to your Git repository. The next section will help you to connect with Azure DevOps and/or the GitHub server.

Connecting to the source control servers

If you have an account on Azure DevOps (previously known as **Visual Studio Team Services (VSTS)**) or GitHub, you can easily connect to it directly from the Visual Studio 2019 IDE. All these connection settings are found under **Team Explorer**.

To connect to Azure DevOps, open **Team Explorer** and click the **Connect...** link, which is under **Azure DevOps**. You will need to log in to the portal to get access.

If you have a GitHub account and you would like to connect to it, click the **Connect...** link, which is under the **GitHub** panel of the **Team Explorer** (**View | Team Explorer**) window. You will need to log in with your GitHub account credentials:



Once you log in to the source control repository of your choice, you will be able to perform common operations such as creating a new repository, cloning an existing repository, pulling, and pushing. Let's begin our journey with Git repositories.

Getting started with Git repositories

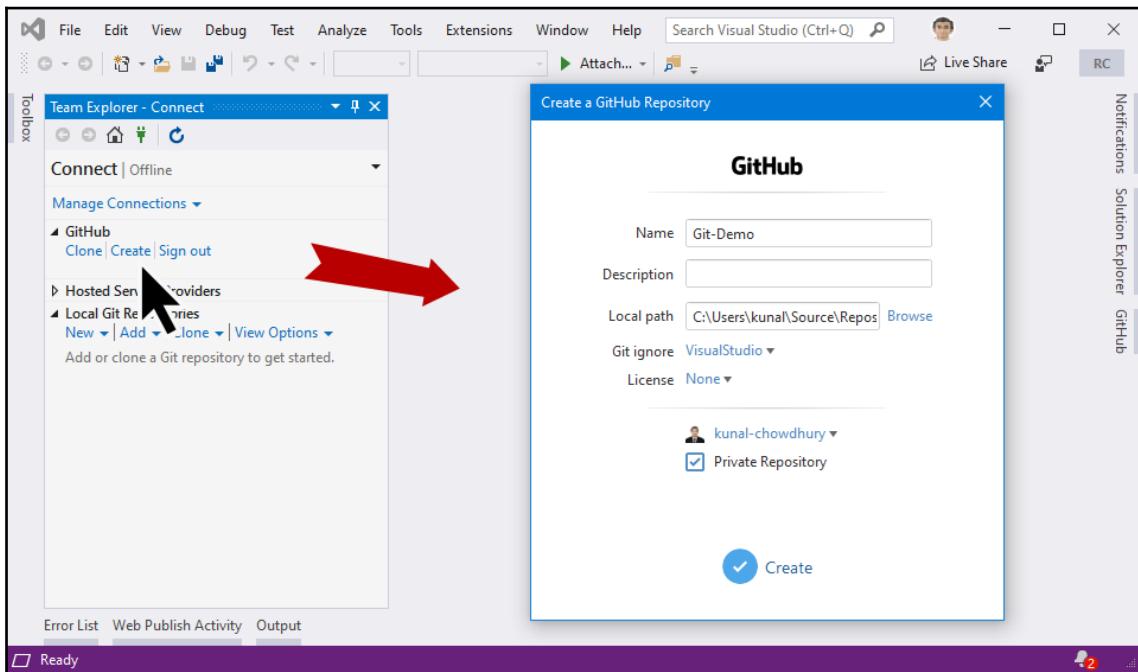
Once you have signed in to the desired server of your choice, you may want to create a new code repository or clone an existing one from the remote server to your local repository. As the process is similar for both Azure DevOps and GitHub, we are going to demonstrate this using the GitHub account.



For details on how to use the Git command from the command line or bash, refer to git.kunal-chowdhury.com.

Creating a new repository

To create a new repository on a remote GitHub server, you can use Visual Studio 2019 Team Explorer directly. Once you have logged in, click on the **Create** link, as shown in the following screenshot. This will open the **Create a GitHub Repository** window onscreen. Enter the required details and select the local path of the repository. Once you have filled everything in, click on the **Create** button:

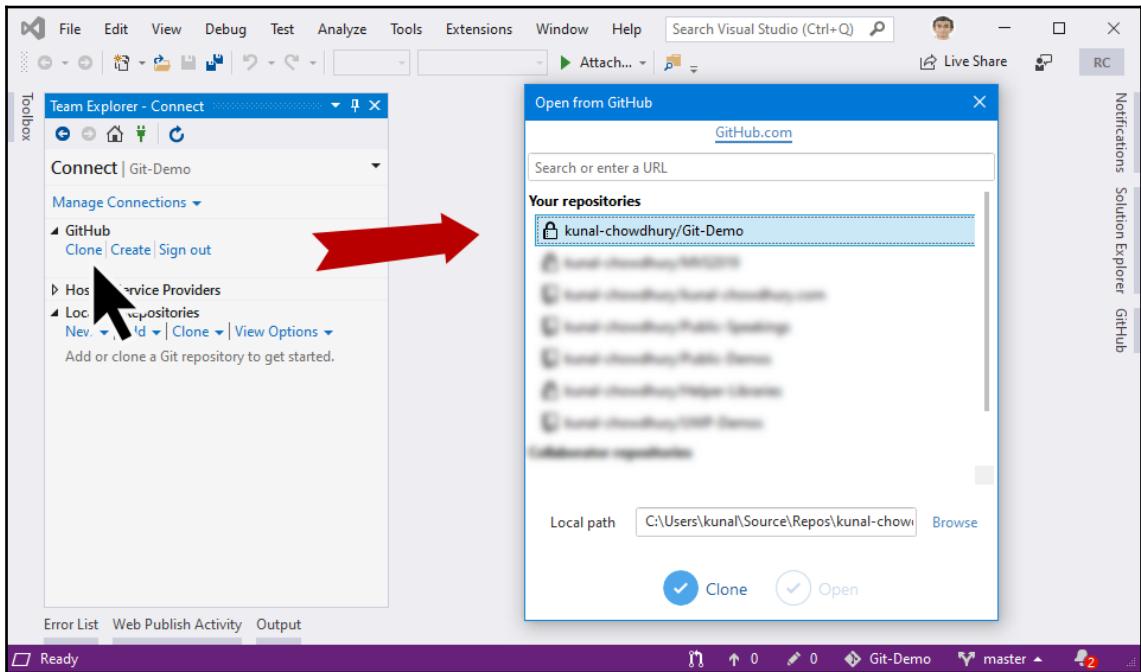


Once the process finishes creating the repository, you will find it listed on the remote server as well as in your local repository.

Cloning an existing repository

Visual Studio 2019 makes it easy to clone an existing GitHub repository. Under the **GitHub** panel in **Team Explorer**, click on the **Clone** link to continue. If you have already logged in, it will show you a screen with a list of repositories that you have on your Git server.

Select the repository that you want to clone, select the folder path of the local repository where you want to download the remote information, and finally, click on the **Clone** button to start the process:

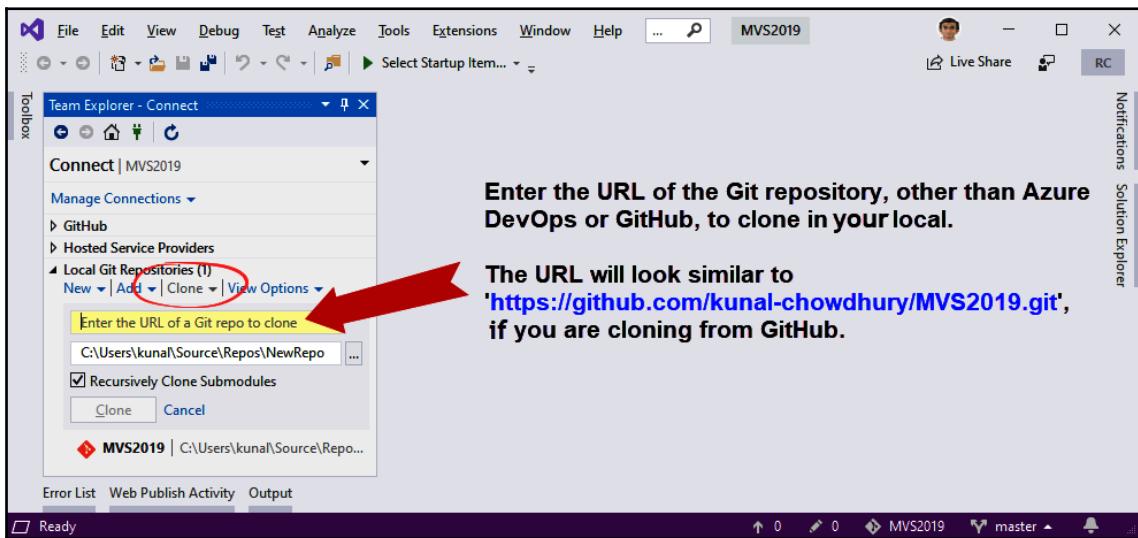


Based on the size of the remote repository and your internet bandwidth, it will take some time to download all the content. Once the process is complete, you will see it listed under your **Local Git Repositories**, as well as in GitHub.



In case you have a repository other than Azure DevOps or GitHub, you can still clone that to your local repository. In such a case, as the extension or support is not available for such services in Visual Studio, you need to click on **Clone**, which will expand a panel. Enter the URL of said repository, select the local folder path of your choice, and then click on the **Clone** button.

This will process the content available on the remote server and download a copy of it to your local repository for further processing:

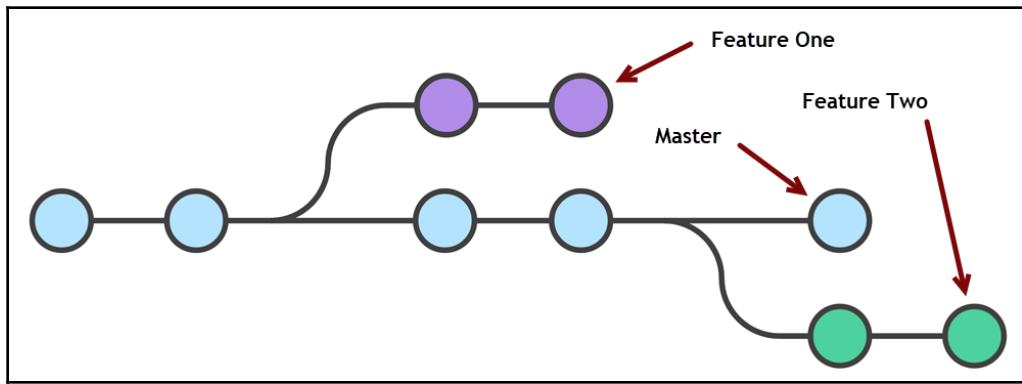


Once the cloning is complete, you will find it listed on the remote server as well as in your local repository.

Working with Git branches

A **branch** in Git is a lightweight, movable pointer to the commits that you make. The name of the default branch is `master`. Whenever you make any commit, it automatically moves forward.

In Git, you can create sub-branches to create another line of development from the master repository, to work on a new feature or to fix a bug. Once the feature or the bug fixing is complete, it is merged back to the main master branch and the sub-branch that we worked is deleted:



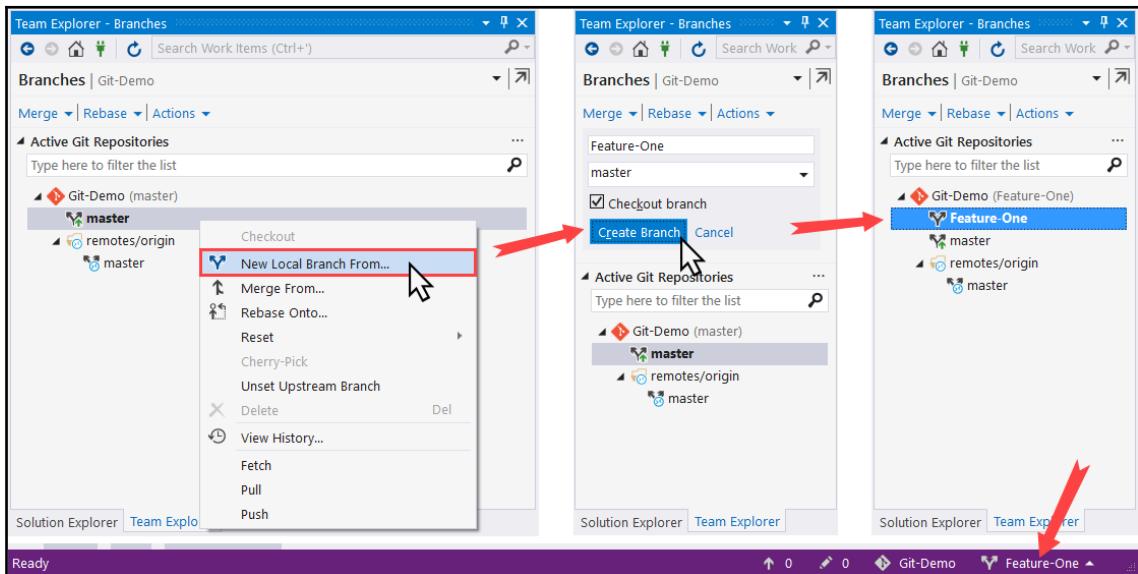
Every branch is referenced by `HEAD`, which points to the latest commit in that branch. Whenever you make a commit, `HEAD` is updated with the latest commit.

Creating a new local branch

Git branches are just a small reference to keep an exact commit history; they do not create multiple copies of your source. Git uses the history information stored in commits to recreate the files on a branch, so it's very easy to create a branch. When you commit changes to a branch, it will not affect the other branches because it creates an isolation. Later, you can share branches with other members or merge the changes into the main branch.

To create a new local branch from another one (for example, the `master` branch), right-click on that branch from Visual Studio **Team Explorer**, and then, from the context menu, select the **New Local Branch From...** menu item.

Enter the name of the new branch in the input box, and select the parent branch from the dropdown (which is automatically populated based on the selection). If you want to checkout to that new branch, make sure to check the box labeled **Checkout branch**. Once you are ready to create it, click on the **Create Branch** button:

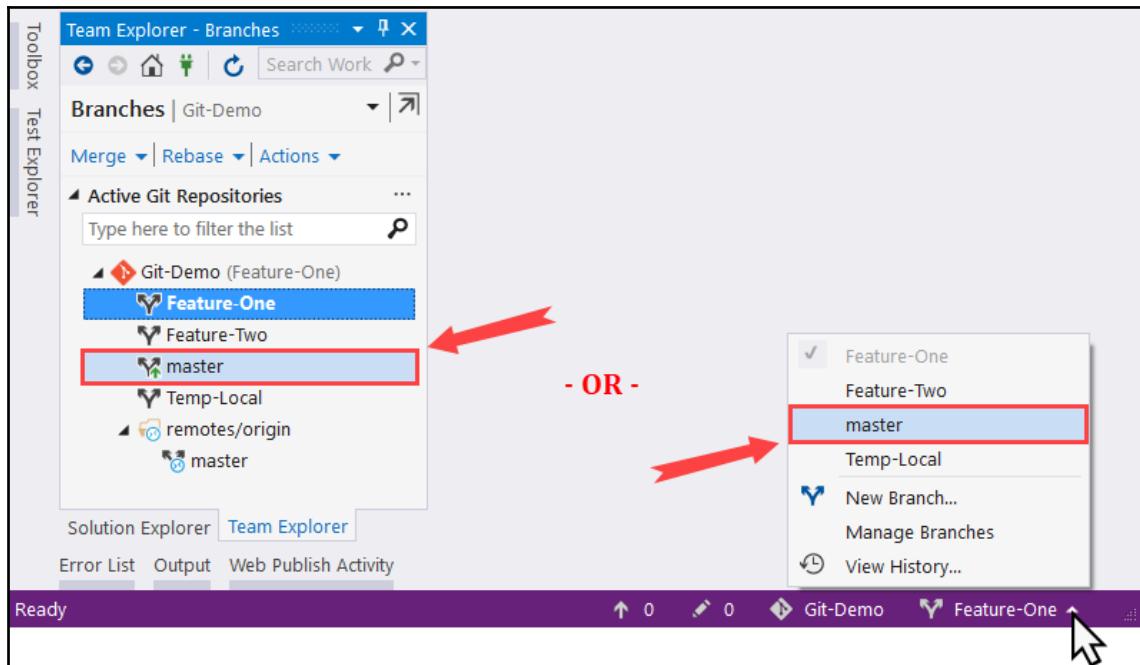


This will create the branch and checkout to it. In **Team Explorer**, you will see it listed as bold. Also, you will see the new branch name in the bottom-right corner of the Visual Studio status bar, as shown in the preceding screenshot.

Switching to a different branch

In Git terminology, switching to a branch is called a **checkout**. Since the branches are lightweight, switching between them is very quick and easy.

You can switch/checkout to a different branch either by double-clicking on it under **Team Explorer**, or by selecting the one from the drop-down menu when you click on the current branch name, which can be found in the bottom-right corner of the Visual Studio status bar:

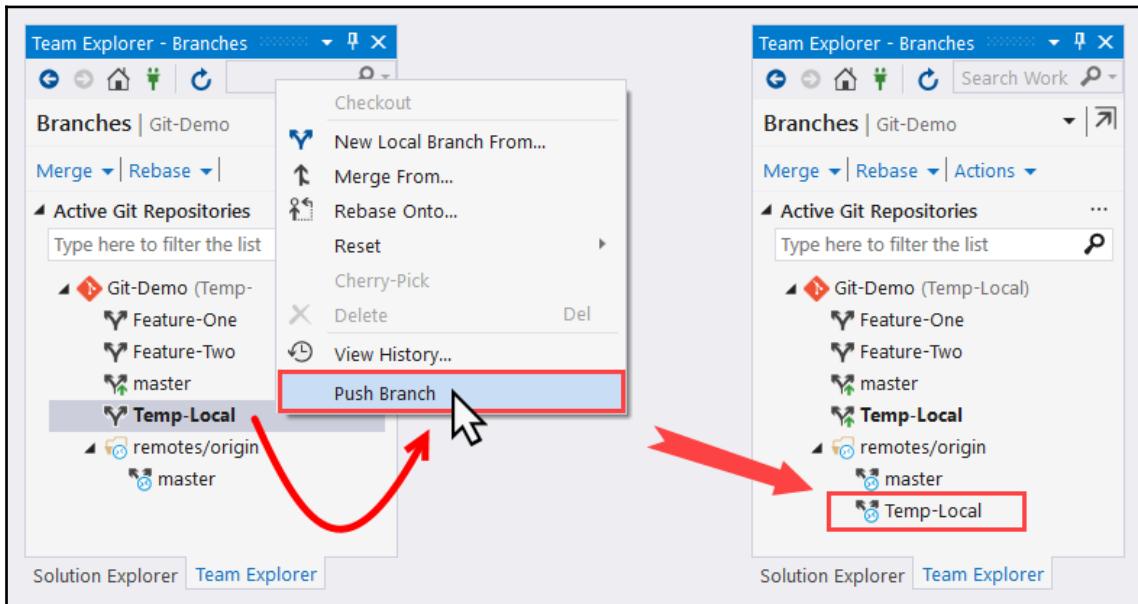


Upon checkout, the selected branch will be highlighted as bold in **Team Explorer**. Also, the name of the branch, in the bottom-right corner, will change accordingly.

Pushing a local branch to the remote repository

When you have a local branch created on your system, you may want to push the branch (with or without any changes) to the remote repository so that other users can see and get the updates.

To push a local branch to the remote repository, right-click on that branch in **Team Explorer**. From the context menu that pops up on the screen, select **Push Branch**. This will update the remote repository with the new details we have created:



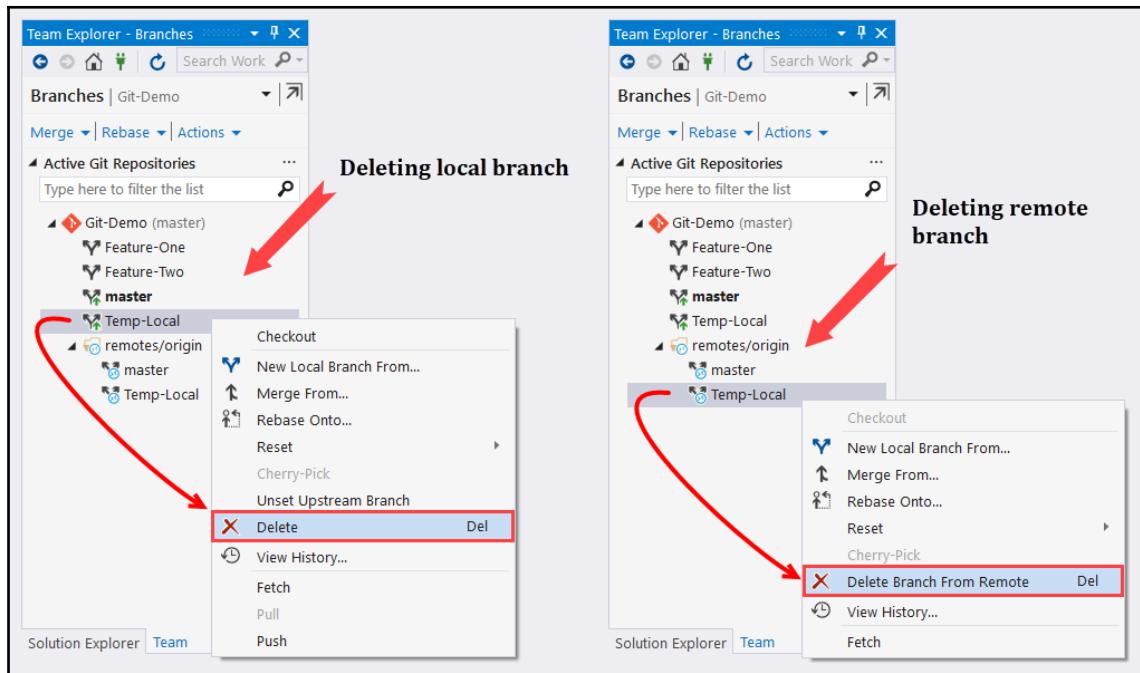
Once the remote update is completed successfully, you will see the branch details listed under **remotes/origin** of your repository in **Team Explorer**.

Deleting an existing branch

There could be some cases when you want to remove a branch. It could either be your local branch or a remote branch on your Git server repository. Visual Studio 2019 provides you with easy access to both.

To delete a local branch, right-click on it and select **Delete** from the context menu. To delete a remote branch listed under **remotes/origin**, right-click on the desired branch and select **Delete Branch From Remote** from the context menu that pops up on the screen.

Alternatively, you can select the branch that you want to delete and press the *Del* or **Delete** key on your keyboard:

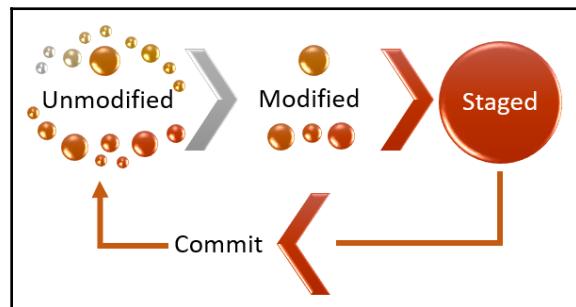


As we have now learned about the basic operations on a Git repository, we can now proceed with some advanced operations.

Working with changes, staging, and commits

Git is all about taking a snapshot of your code to your repository, as and when you commit your changes. There are three stages/areas where Git tracks the changes as you continue working on your repository:

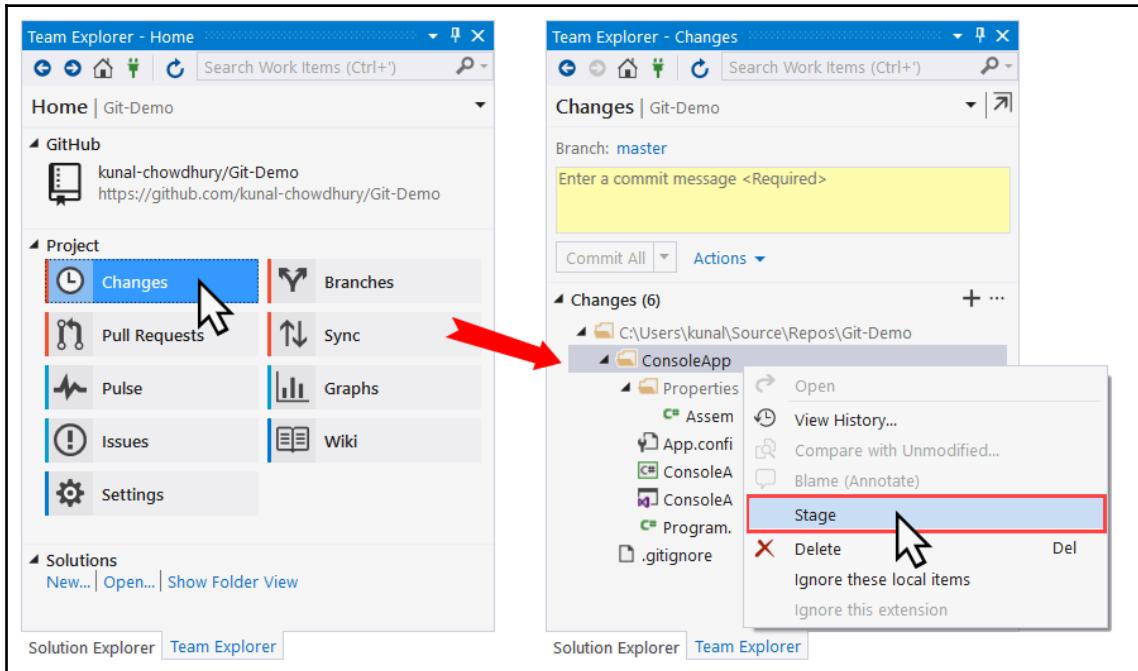
- **Unmodified:** This is the area where the available files have not yet changed since your last commit to your repository. When you change some files and save them, Git will mark them as modified.
- **Modified:** These are the files that have been changed after your last commit but not yet staged for your next commit. When you stage these files, Git will move them to the staging area.
- **Staging:** This is the area where the changed files have been added for your next commit. When you commit the staged files, they will be marked as unmodified:



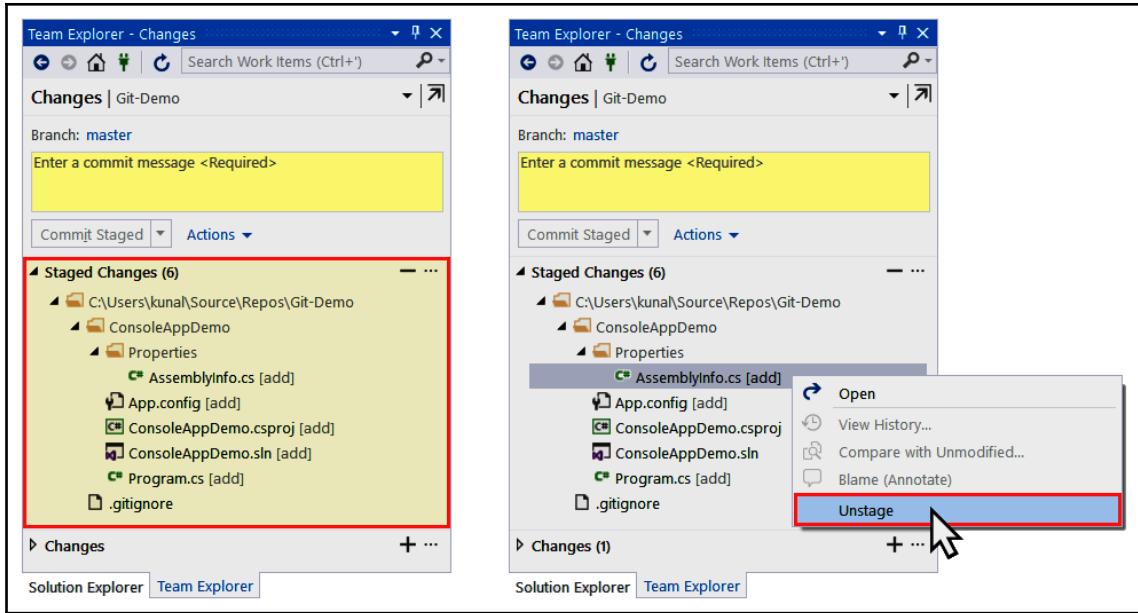
Let's learn how to stage changes, commit changes, discard uncommitted changes, and amend messages to an existing commit.

Staging changes to your local repository

When you add new files to your project, modify existing files, or delete existing files from your project or Git code base, they will be marked as modified since the last commit. You can see a list of your changes by navigating to **Project | Changes** under **Team Explorer**, as shown in the following screenshot:



Right-click on the files/folders that you want to move to the staging area, and, from the context menu, click **Stage**. Later, if you want to unmark any files/folders from the staging area, right-click on those files/folders and click on **Unstage** from the context menu, as shown in the following screenshot:

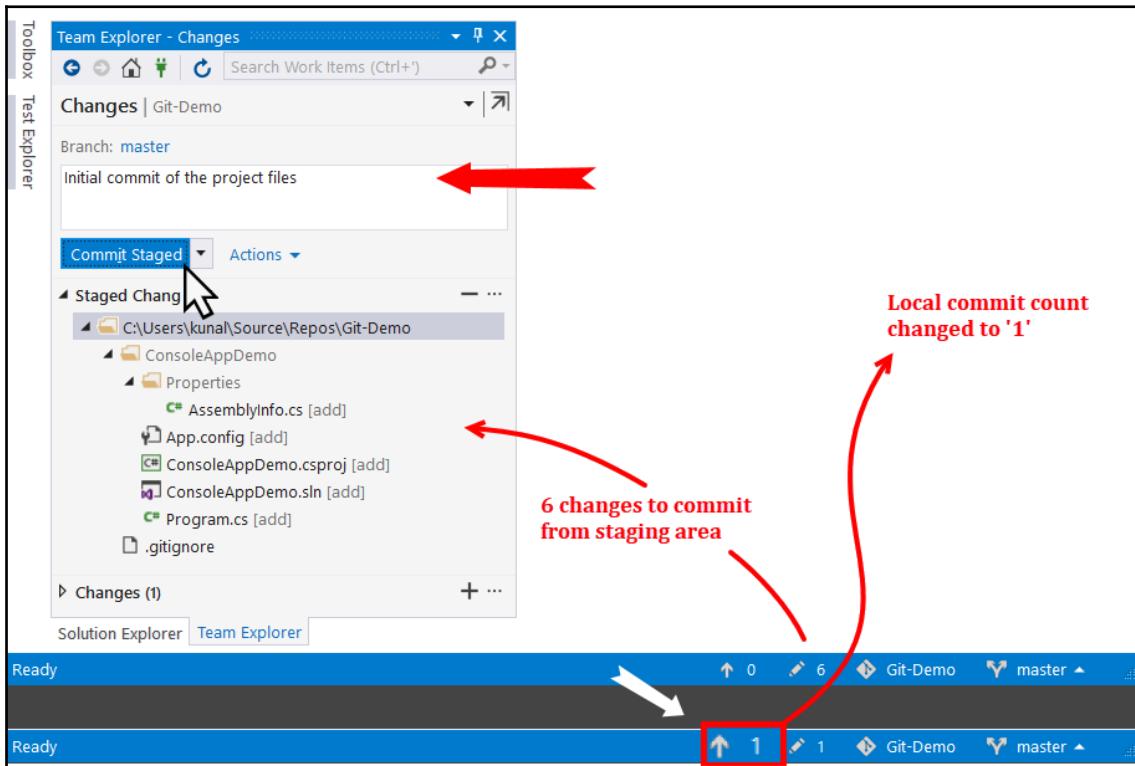


If you **Stage** any files/folders after you have already staged them, the delta will be added to the changes made in staging. When you are ready to save the changes to your repository, you need to commit those changes.

Committing changes to your local repository

To commit the staged changes to your local repository, navigate to **Team Explorer** | **Project** | **Changes**. You will see a list of staged files. You will also see a list of changed files that have not yet been pushed to the staging area.

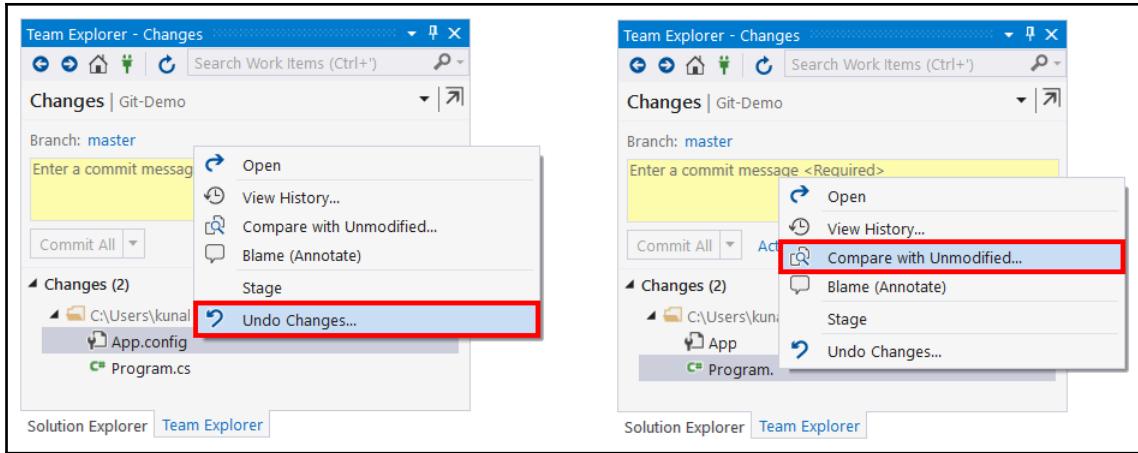
Enter a comment as your commit message and click on **Commit Staged** to perform a commit operation in your local repository. The changed files that are not yet in the staging area will not be committed:



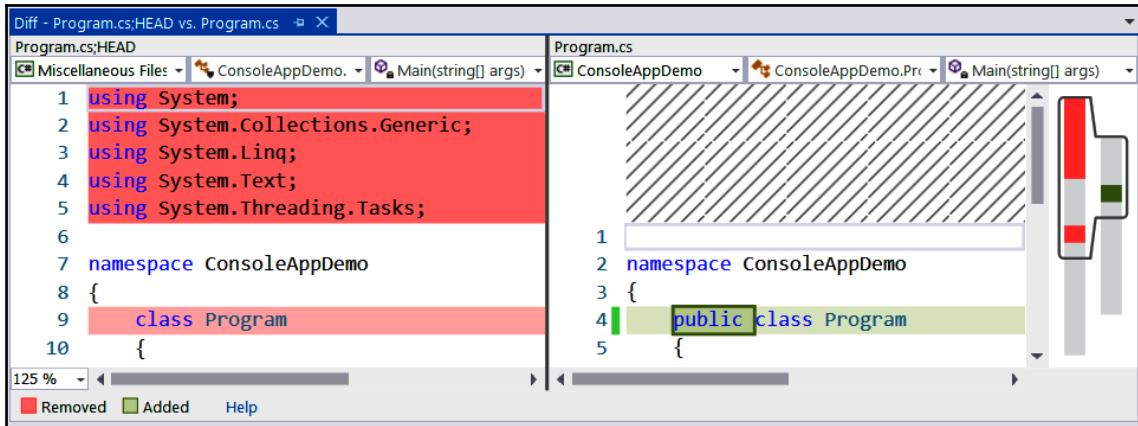
As shown in the preceding screenshot, the Visual Studio 2019 status bar will show the count of the commits that have been performed but not yet pushed to the remote repository. As and when you commit to your local repository, the count will increase by one, but when you push all your local commits, it will reset to zero again.

Discarding uncommitted changes

There could be a set of changed files that you don't want to commit, and so want to undo the changes that you have already performed on them. Select those files/folders and right-click on them to open the Git context menu, where you can click **Undo Changes...** to discard them. The files will be reset to the unmodified stage:



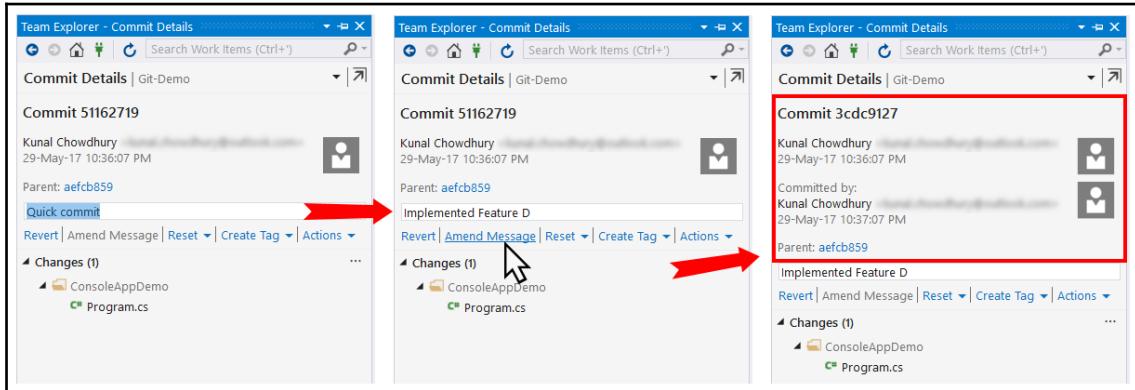
Sometimes, before performing any commit or undoing the changes, you may want to check what has been changed since your last commit. You can do so by individually selecting a file, right-clicking on it, and clicking **Compare with Unmodified...** from the context menu to compare the changes side by side:



You will find the removed lines or content marked with a reddish background, whereas the amended or added content will be marked with a greenish background.

Amending messages to an existing commit

Occasionally, in some rare cases, you may want to change the existing commit message. To do so, open the commit that you want to modify. Change the desired message and click on the **Amend Message** link, as shown in the following screenshot:



This will modify the commit message and replace the existing one with a new commit. At the end, make sure to push the changes to the remote repository in order for the changes to be available to other users.

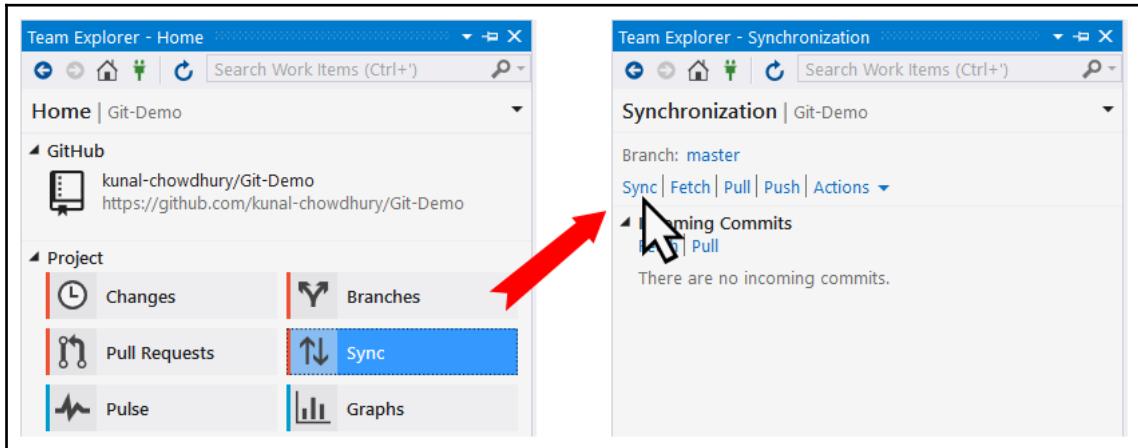


It is not advisable to modify/amend a public commit on a remote repository, as it is going to replace the existing one with a new hash code. When required, only perform this operation on a local commit.

Syncing changes between local and remote repositories

The Git extension for Visual Studio 2019 allows you to easily sync changes between local and remote repositories. If it finds any remote changes, it will first download those and merge with the changes in the local repository. If it finds any merge conflicts, it will immediately stop processing and ask you to resolve them first. Later, it will push all your changes to the remote repository.

To invoke a single sync operation, open **Team Explorer** and navigate to the **Sync** view, which may have outgoing commits ready to push:



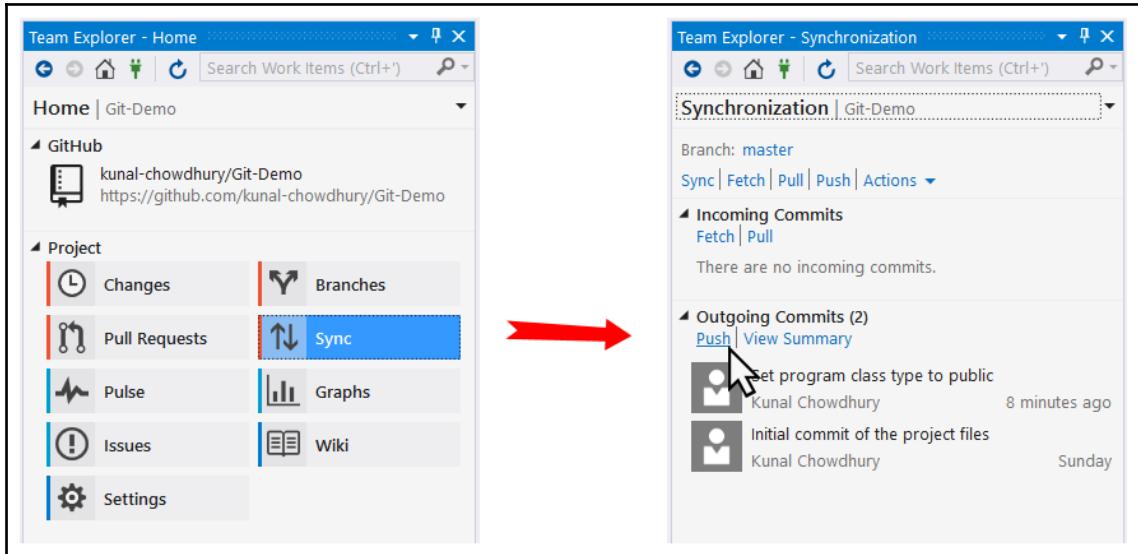
You can also perform the steps individually, rather than performing both the inbound and outbound operations in one go. **Push** will commit your local changes to the remote repository; **Fetch** will download the remote changes, but won't merge; and **Pull** will download the remote changes and proceed to merge the changes. Let's discuss these one by one.

Pushing changes to the remote repository

Pushing your changes can be done in two ways: push and publish. When there is a relationship between the local branch and the remote repository, push will commit the local changes to the remote repository. When there is no relationship between the local branch and the remote repository, it publishes the changes to the remote repository by first creating the branch with the same name of the local repository, and then pushing the local commits to it.

After you publish the branch, only push operation will happen as a branch exists in remote, with a relationship to your local repository.

To push your changes to the remote repository, open **Team Explorer** and navigate to **Sync** view. You will see a list of outgoing changes, as shown in the following screenshot. Review the changes for each commit and finally, click on **Push** to commit those on your remote branch and make them available publicly within the same project team:

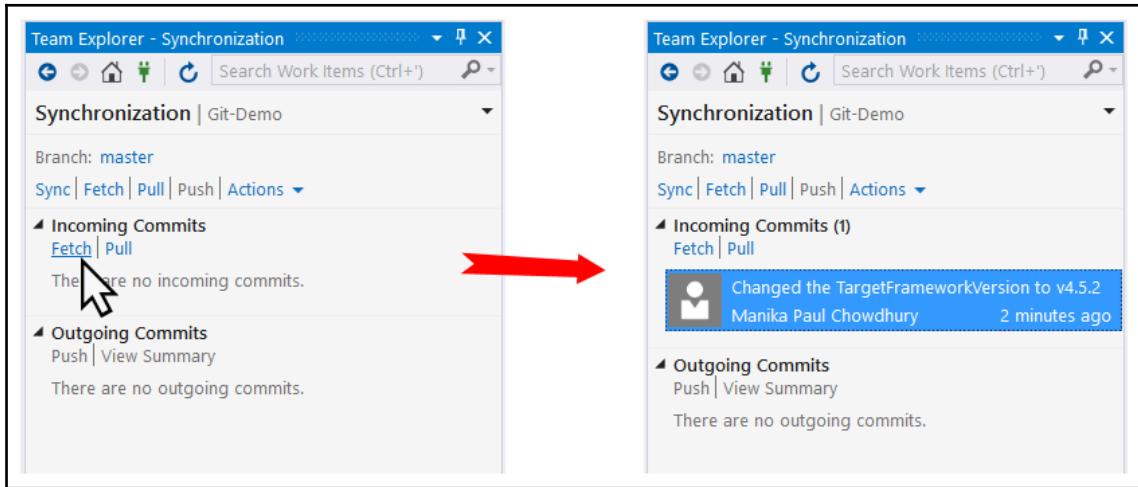


If, during the push operation, Visual Studio finds any conflicts between remote commits and local commits, it will break the operation immediately. You will first have to resolve these conflicts before you can push your changes.

Fetching changes in the remote repository

When you have the latest changes in the remote repository, but not in your local repository, you can ask Visual Studio to fetch those changes (new commits and new branches) using the Git command, **Fetch**. **Fetch** will just download the changes to your local repository but won't merge automatically. You will need to review the changes first.

To fetch the remote commits, open **Team Explorer**, navigate to **Sync** view, and then click on **Fetch**:

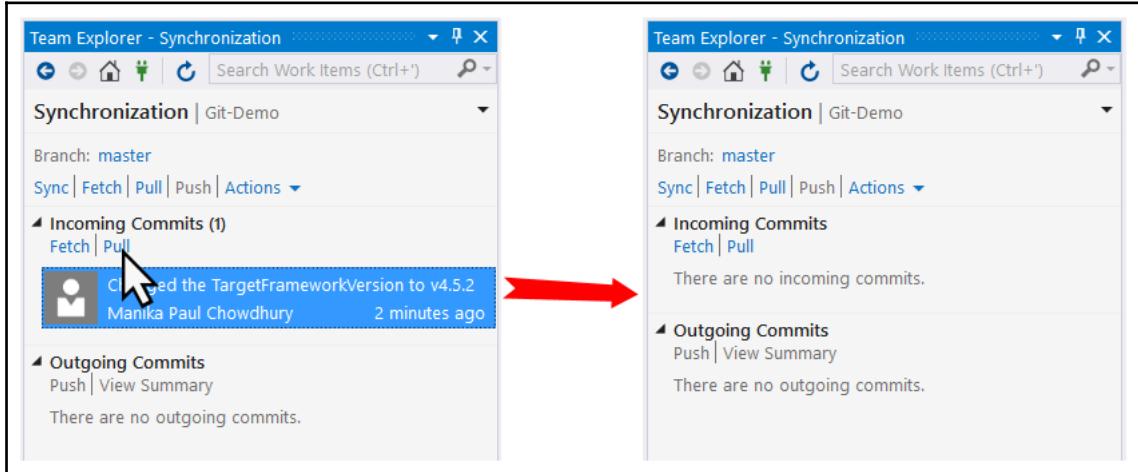


A list of incoming commits will be shown, if any. You can then review them and merge the changes with your local changes.

Merging changes in the remote repository with your local repository

Pull is just like fetch, but it merges the changes automatically after downloading them. If any merge conflicts exist, although Git takes care of most of them, it will ask you to resolve them first before continuing.

To apply the remote changes to your local repository, open **Team Explorer** and navigate to **Sync** view. Then, you can click on either **Sync** or **Pull** to download the changes and merge them:

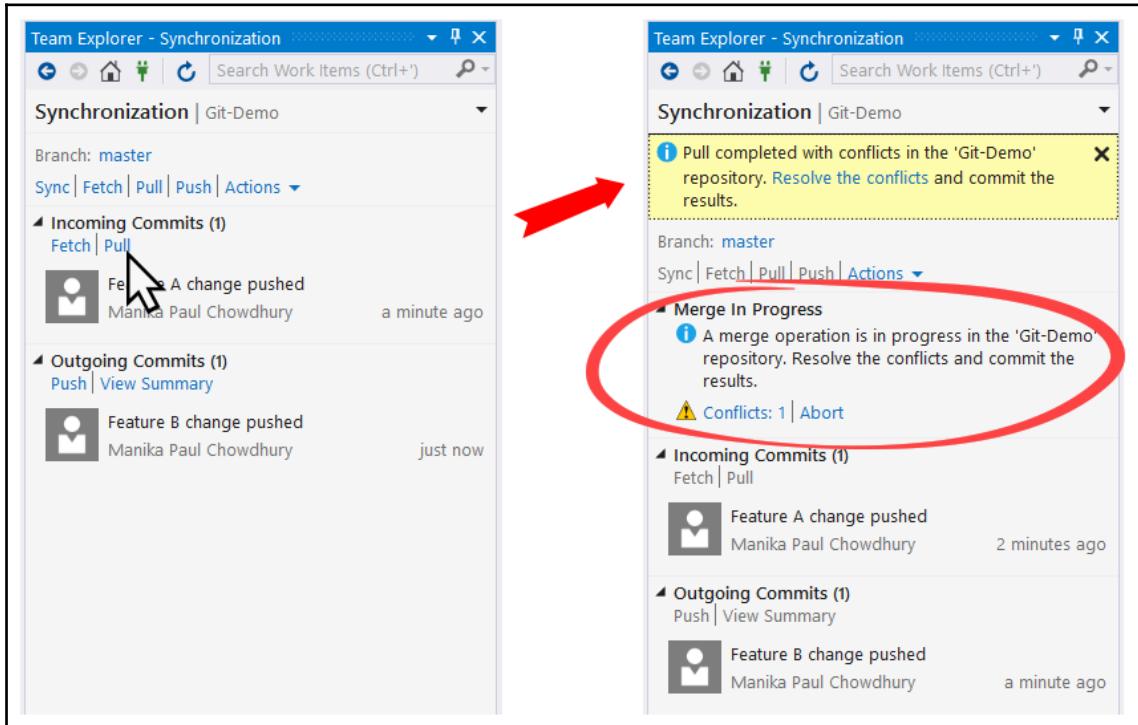


The merge operation will keep the commit history of your local changes intact so that, when you share your branch with the push command, Git will understand how others should merge your changes.

Resolving merge conflicts

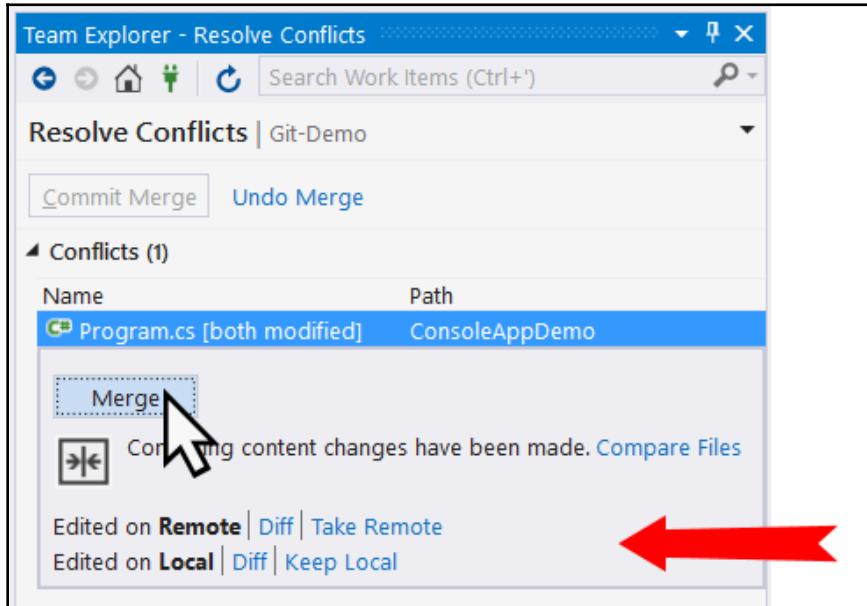
Git is good at automatically merging the file changes, but sometimes, it can throw merge conflicts. In such cases, you will have to manually resolve the conflicts before syncing your local and remote branches.

When there are merge conflicts, Visual Studio 2019 will list the conflicts under the **Merge In Progress** panel of the **Sync** view. Click on the **Conflicts** link to start resolving the file conflicts:



This will bring up a list of files with conflicts. Selecting a file lets you accept the changes in the source branch where you are merging. You can also compare the files by using the **Diff** link and compare with **Remote** or **Local**.

From the same screen, you can also take the remote changes or keep the local changes. If you want to review the changes and perform a merge operation manually, click on the **Merge** button, as shown in the following screenshot:



This will show you a side-by-side view of your source and target, along with the resultant changes to the file. Use the checkboxes next to the modified lines under merge conflicts to select between the remote and the local changes. You can also modify the lines directly in the **Result** panel.

When you are done with the changes, click on the **Accept Merge** button to accept the changes performed for the conflicts. You need to perform the same for all the conflicts that you have before continuing with the synchronization of local and remote branches:

The screenshot shows the 'Merge' tool window in Visual Studio. It displays three panes: 'Source' (ConsoleAppDemo\Program.cs\Remote), 'Target' (ConsoleAppDemo\Program.cs\Local), and 'Result' (ConsoleAppDemo\Program.cs). A red arrow points to the 'Accept Merge' button at the top left. In the 'Source' and 'Target' panes, line 8 contains a conflict: 'System.Console.WriteLine("Feature A");'. Both lines have a checked 'Accept' checkbox. In the 'Result' pane, both lines are present as 'System.Console.WriteLine("Feature A");' and 'System.Console.WriteLine("Feature B");', indicating they were merged. The 'Result' pane also has a checked 'Accept' checkbox for line 8.

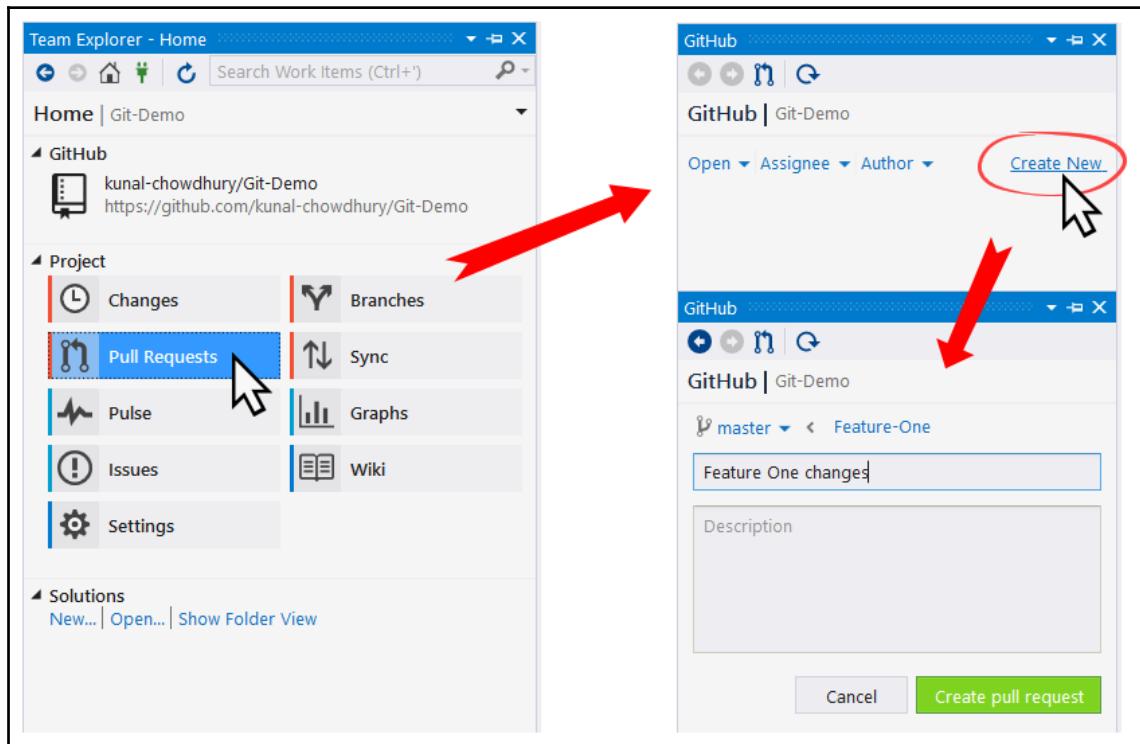
Once you are done resolving all the conflicts, you can go to the **Changes** view and commit the merged changes.

Working with pull requests for a code review

A pull request is the collaborative process for discussing the changes implemented in a branch, getting early feedback on the changes, and merging them with the master branch once everyone approves them. You can create a pull request of a work-in module even if you are not ready to merge the changes.

Creating pull requests for a code review

If your Git repository supports it, you can create pull requests directly from the Visual Studio 2019 IDE. First, you will need to push/publish the local branch changes to the remote repository. Then, open **Team Explorer** and navigate to the **Pull Requests** view, as shown in the following screenshot:

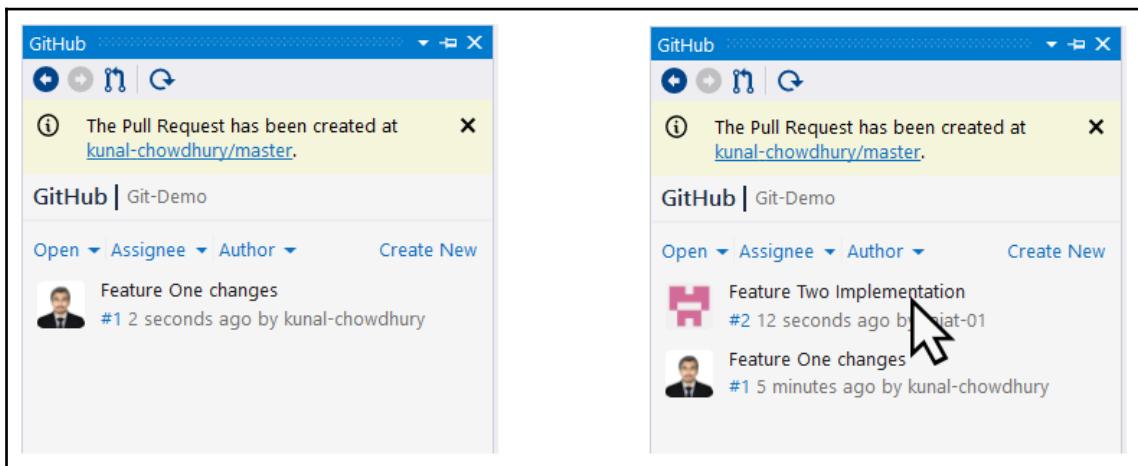


If you are using the GitHub extension, a panel will appear with a **Create New** link to create the pull request for a code review. It will automatically list the branch that you are working on and will begin to merge once the reviewers approve it. Give a title to the pull request and a description of the work that you performed in this change set. Once you are ready, click on the **Create pull request** button. All the members of the project will get a notification to review your changes.

If you are using **Team Foundation Server** or **Azure DevOps**, you may get an option—while creating the request—to enter the reviewer's name to get it reviewed by selected people on the project team.

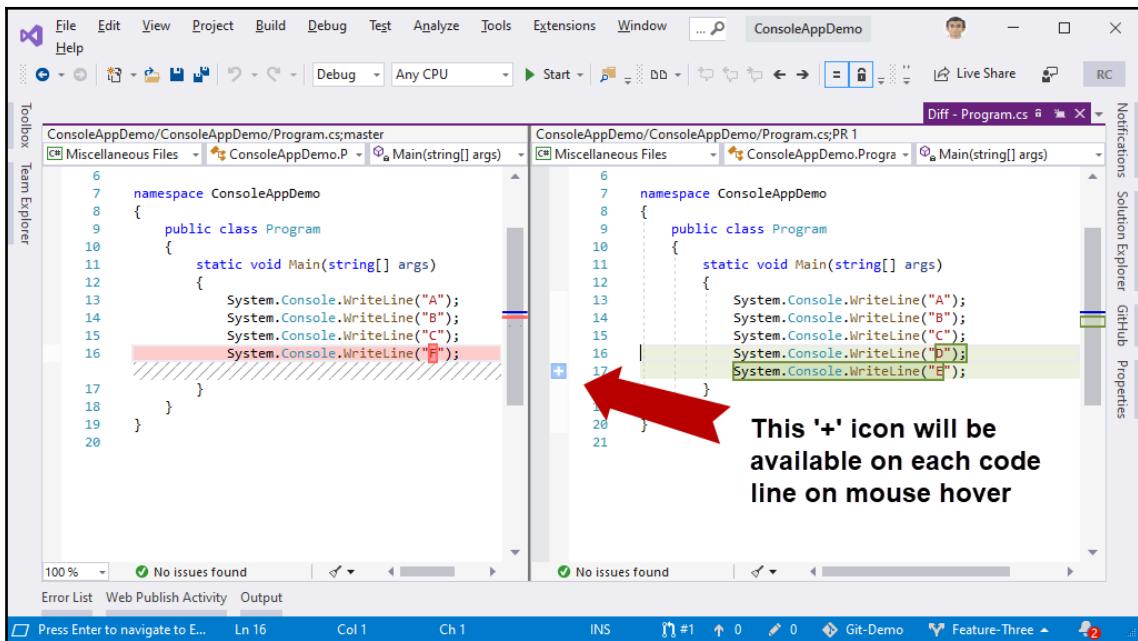
Reviewing an existing pull request

When you create a pull request, the selected reviewers will receive a notification to review your changes. They can provide feedback and/or approve your pull request to merge with the master branch:

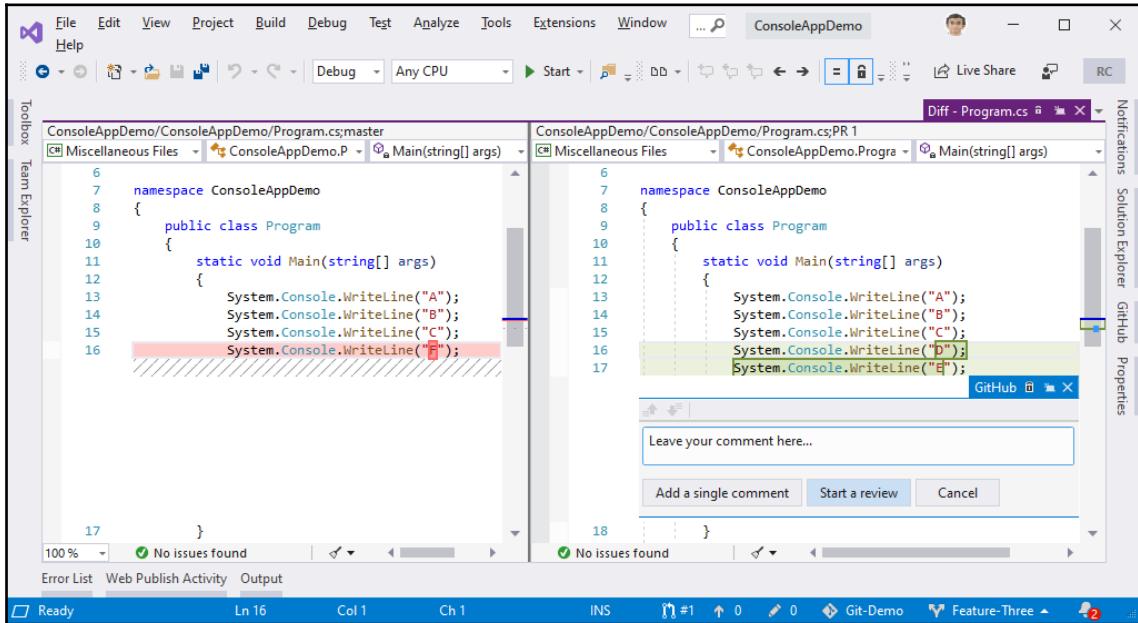


When you get a notification about someone else's pull request, you can double-click on it to open the review panel.

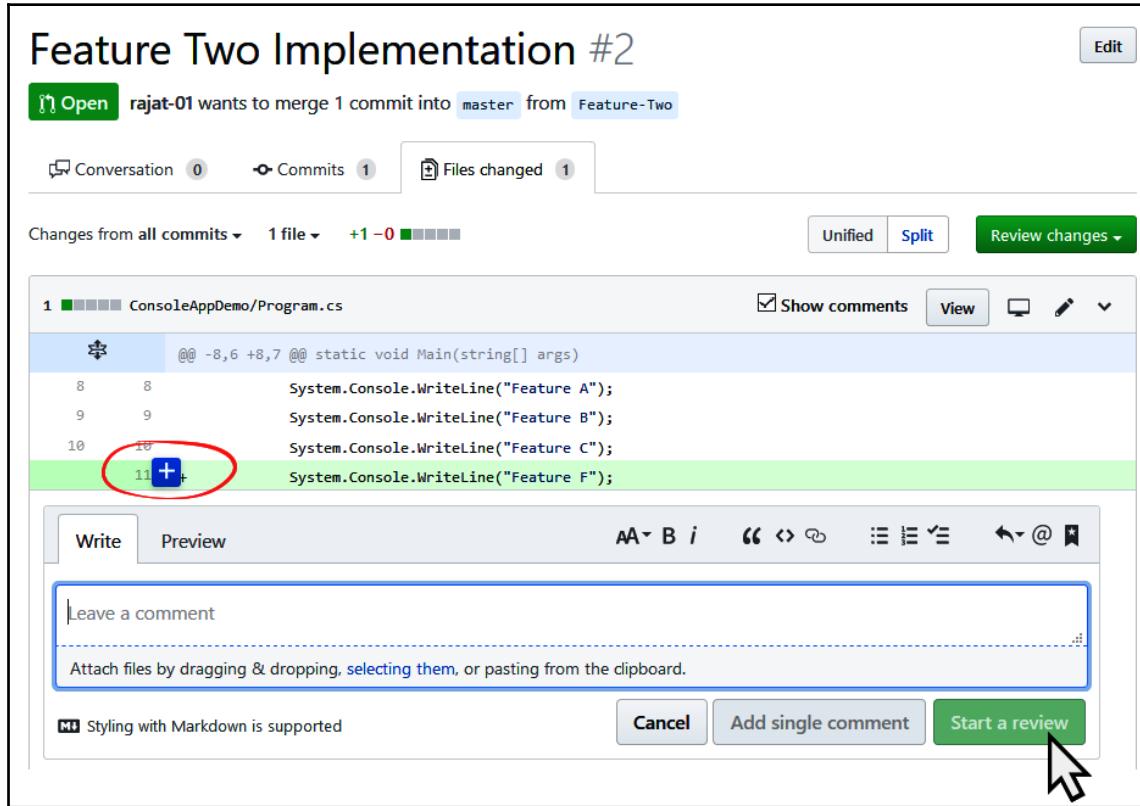
In the code lines—where you have some feedback or want to ask the author of this pull request to implement further changes—click on the + symbol, which will appear once you hover over that line:



Clicking on the + symbol icon will provide you with an area to enter your comment description. You can use various formatting options to construct it, and finally, you can click on the **Start a review** button (if you are using GitHub) to submit your feedback:



Alternatively, you can begin a review of your pending pull requests from the GitHub site too. Here is a screenshot from GitHub, demonstrating the process of adding a review comment to a pull request:



In GitHub, your comments can be of three types:

- **Comment:** To be used to submit general feedback, without providing approval to the pull request.
- **Approve:** Use this option to provide approval to the code review request and to give your approval to merge the changes to the master branch.
- **Request changes:** When you have a change request in mind, you can select this and ask the author of the code to address those changes before proceeding toward final approval.

Working with the Git commit history

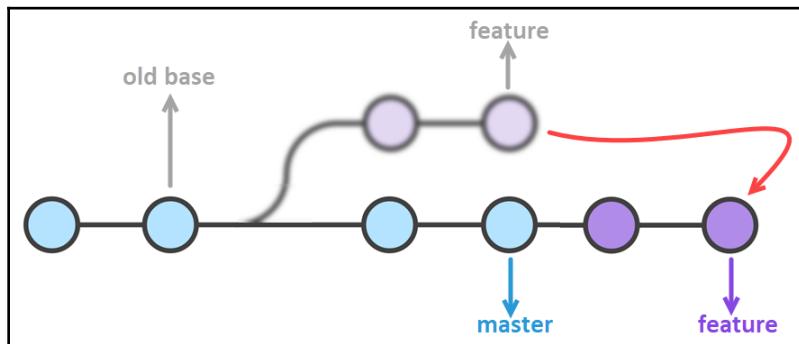
Git is used to manage the history of changes as and when you save your code as commits to your local repository, and then merge these changes with the master branch once pull requests have been approved.

The commit history becomes complicated when you pull remote changes, made by other team members, from the master branch to your feature branch. It then loses the linearity of the commit history, making it hard to follow.

This also makes it hard to keep track of the final feature changes when multiple commits exist in a single feature branch. In such a case, if you want to revert a feature at a later date, this makes it difficult to track and revert each one of them.

To resolve this, Git provides a command called `rebase`, which addresses all such issues. It takes the commits made on your current branch and replays them on a different branch. The commit history on your current branch will be rewritten to retain the granularity of the history.

For example, consider the following diagram as the commit history, where you started the feature branch from the old base and saved your changes with two commits. In the meantime, some other members of your team committed two changes in the master branch. Thus, the old base will now have two branches; one points to master, and the other to feature:

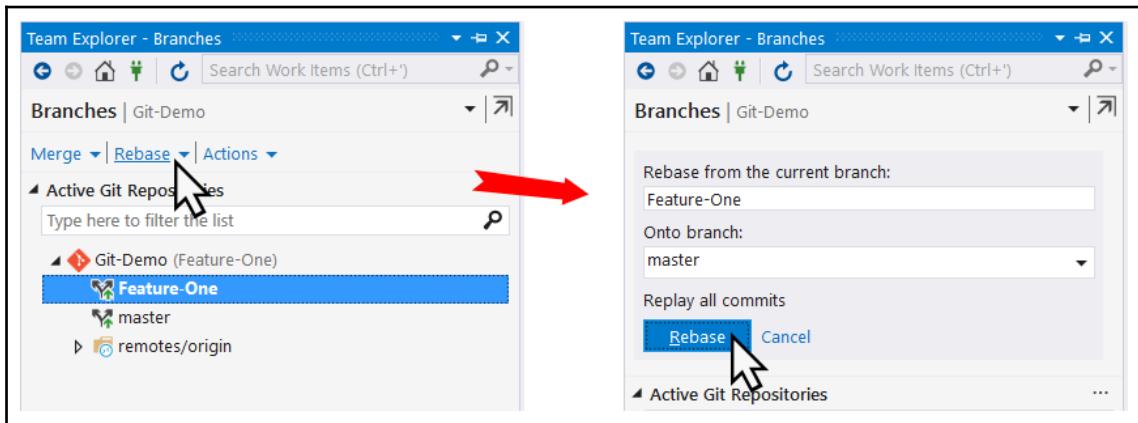


Now, when you rebase your feature branch on to the master, it will rewrite your local commit history, replaying your feature changes to the master branch. You will then see a linear commit history on your local repository along with the changes made by others.

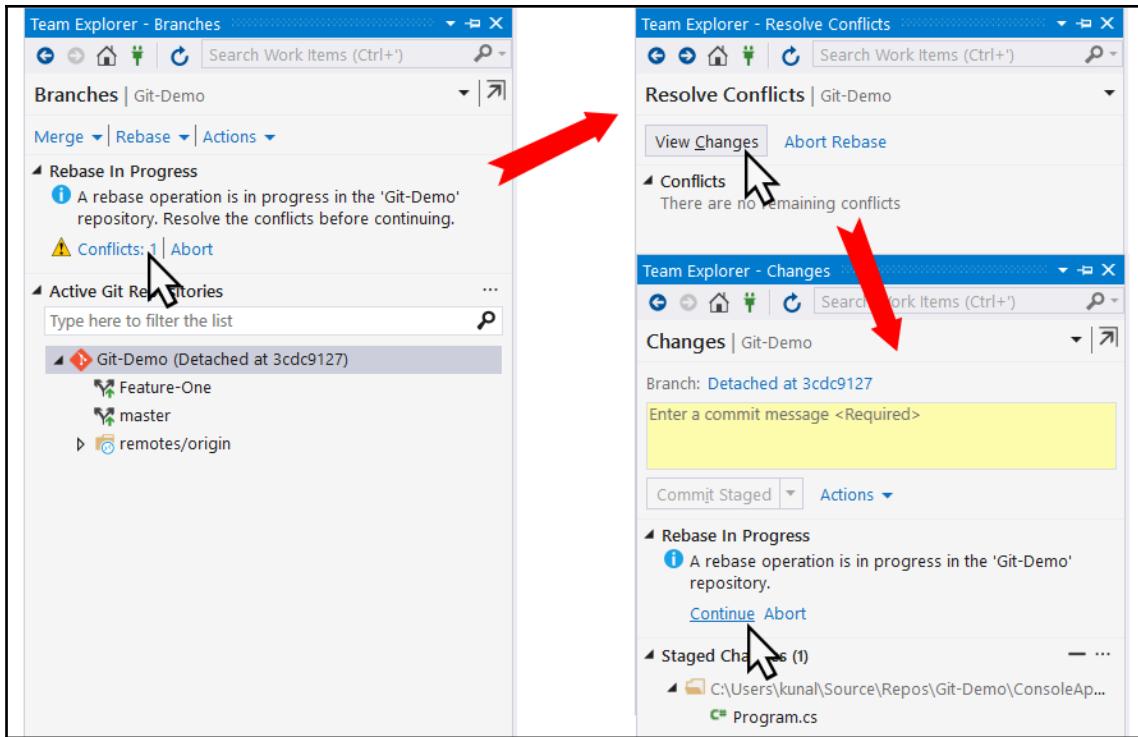
Rebasing changes to rewrite the commit history

To rebase your current feature branch on the recent changes on the master branch, first make the feature branch current and then navigate to the **Branches** view from **Team Explorer**.

There, you will find a link called **Rebase**, which, when clicked, will provide you with a view to rebase the changes from your current branch to the branch where you would like to replay the commits:



When you are ready to rebase your code, click on the **Rebase** button and wait until it completes the process. If there is a conflict, you will have to resolve those merge conflicts before completing the rebase. Click on **Continue** once you resolve all the conflicts, or click on **Abort** to terminate the rebasing process:



After a successful rebase, your local branch will have a different history than your remote branch. You will have to force push the changes in your local repository to update the changes in your remote branch.

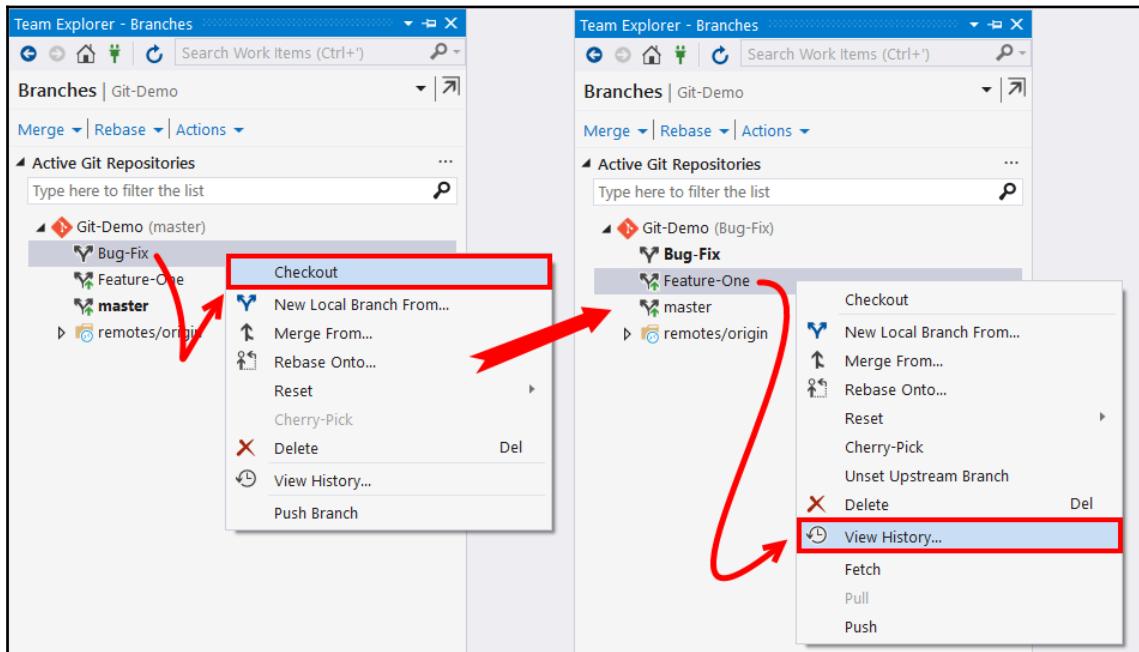


Never force push a branch that other members are working on, as this rewrites the commit history. Only perform a force push on the branches that you alone are working on.

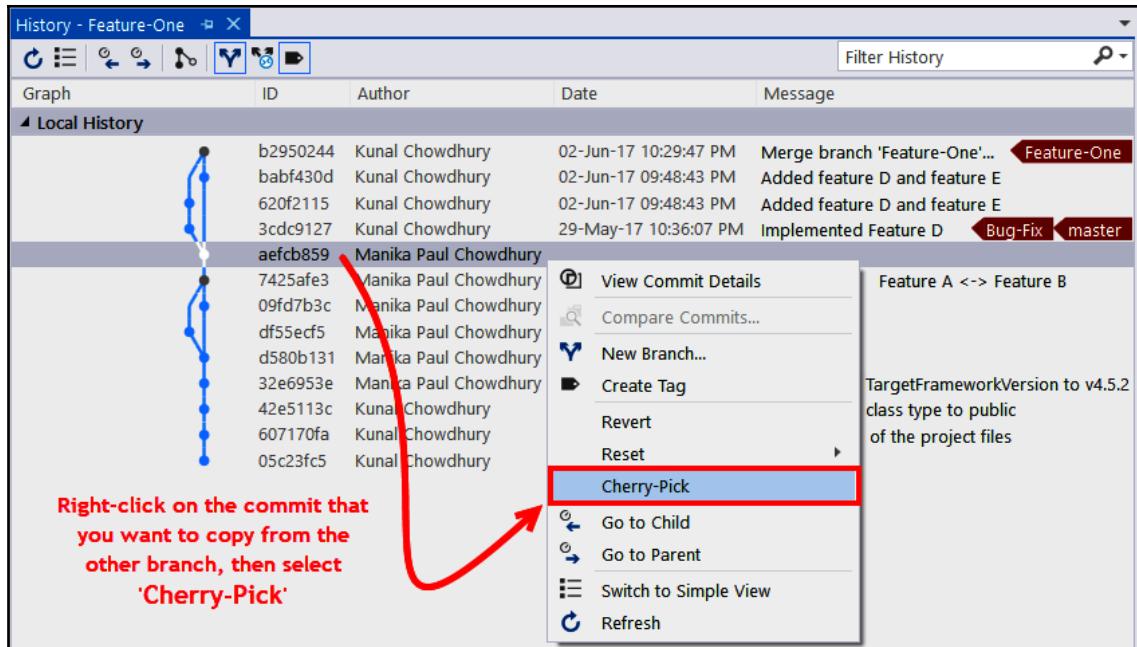
Copying commits using cherry-pick

Cherry-pick is a process that involves copying commits from one branch to another. It only copies the changes from the commits instead of copying all the changes in a branch. Thus, it is completely different from the functions that merge and rebase perform.

You will need to perform a cherry-pick on commits in cases when you have accidentally committed to the wrong branch and/or want to pull out a set of commits from your **Bug-Fix** branch to your master/feature branch, as soon as those are available:



To cherry-pick, first check out the branch to where you want to copy a set of commits from a different branch. Then, open the history of the other branch from which you want to pull by clicking on **View History...** from the context menu, as shown in the preceding screenshot. From the list of the commit history, select the one that you want to pull to your current branch. Right-click and select **Cherry-Pick** from the context menu, as shown in the following screenshot:



This will pull the changes to the current branch. If there are any merge conflicts, you must resolve them first before continuing.

Once the cherry-pick process is successful, commit the changes and push it to your remote branch. You can repeat the process for each commit that you want to bring into your current checked-out branch.

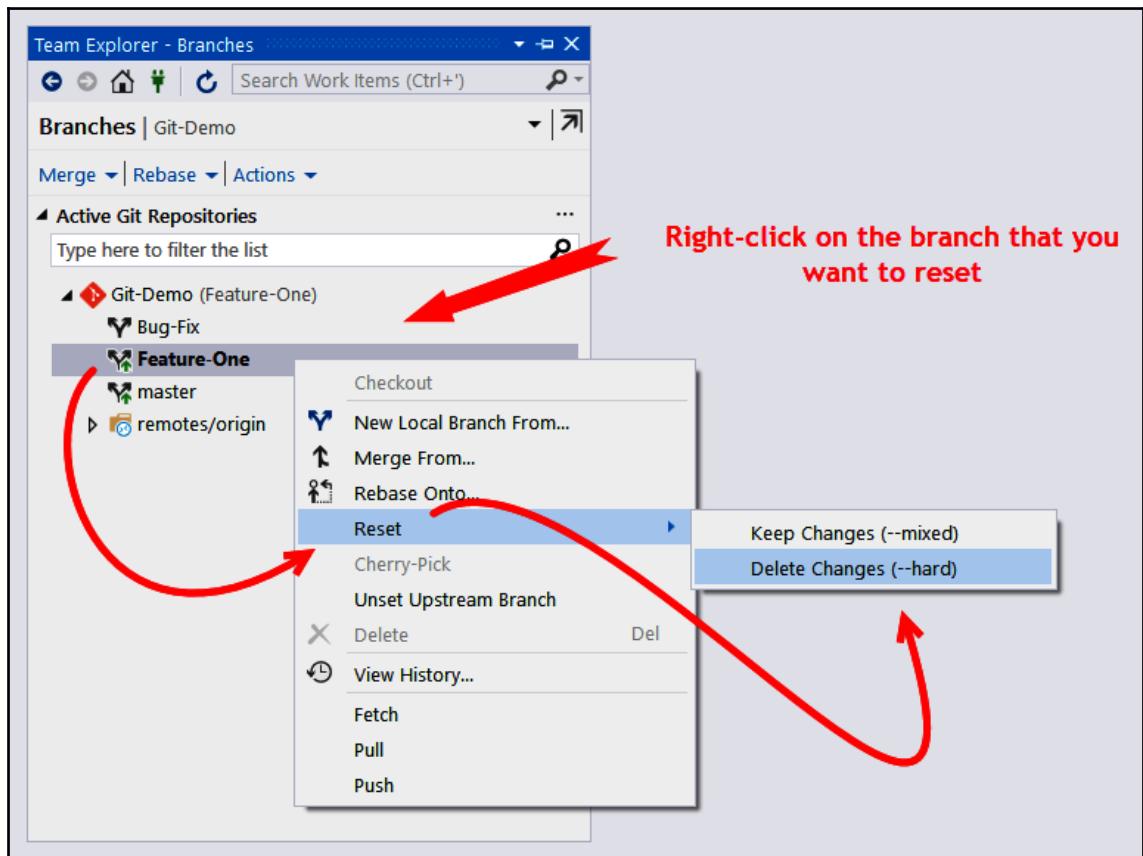
Undoing your changes

You may not need all the changes to be committed and pushed to the remote repository. There are many cases when you just need to reset the working directory/workspace to get a cleaner space, showing just the last committed changes. You may also want to revert a change from the remote branch if you have pushed a wrong commit. In this section, we are going to discuss how to reset and revert a branch.

Resetting a local branch to a previous state

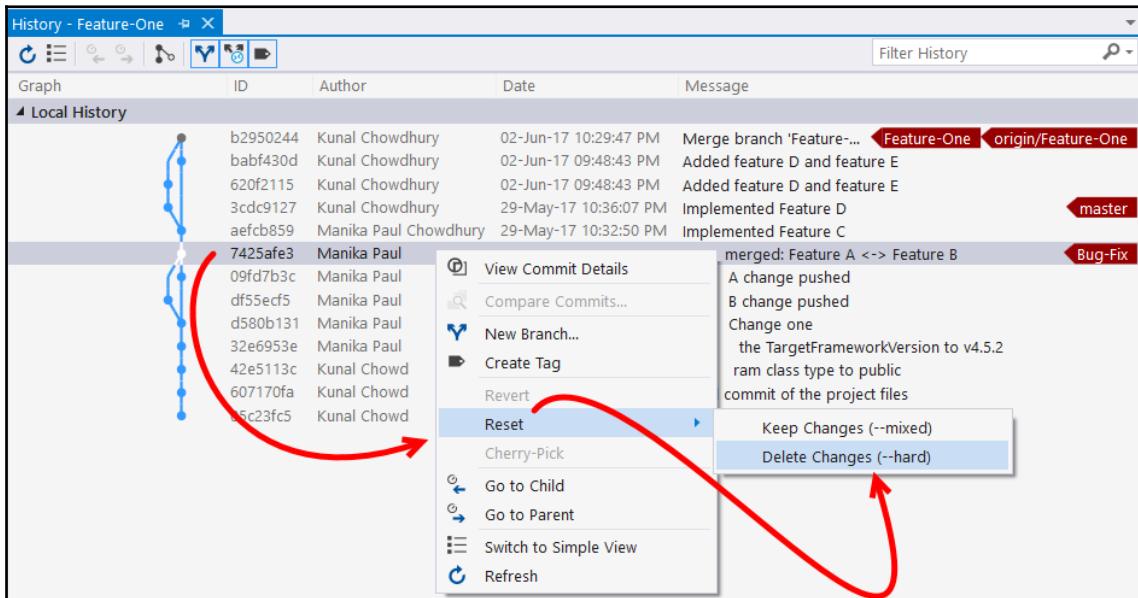
Resetting a local branch can be done in two different ways. The first option is resetting the branch to the last committed state in the local repository. During this, you can select whether to keep your changes, but its most common usage is to undo all the uncommitted changes that happened in the local branch.

To reset a branch, open **Team Explorer** and navigate to the **Branches** view. As shown in the following screenshot, right-click on the branch that you want to reset. From the context menu, select **Reset** | **Delete Changes (--hard)** to undo all uncommitted local changes:



The other option could be resetting the local branch to a specific commit. All the changes until that selected commit will be undone and you will be provided with a fresh working copy of your files.

To reset a branch to a specific commit, open the **History** page of the branch and right-click on the commit that you would like to reset to. From the context menu, select **Reset | Delete Changes (--hard)**, as shown in the following screenshot:



Note that the reset operation affects the entire branch that has been selected, not just those in your current directory.



Upon doing so, the local branch will have the changes up to the selected commit from the commit history. The history page will add another section that will list the **Incoming** commits, which, if needed, can be pulled to the local branch:

Graph	ID	Author	Date	Message
Incoming				
	b2950244	Kunal Chowdhury	02-Jun-17 10:29:47 PM	Merge branch 'Feature-One' of https://g... origin/Feature-One
	babf430d	Kunal Chowdhury	02-Jun-17 09:48:43 PM	Added feature D and feature E
	620f2115	Kunal Chowdhury	02-Jun-17 09:48:43 PM	Added feature D and feature E
	3cdc9127	Kunal Chowdhury	29-May-17 10:36:07 PM	Implemented Feature D
	aefcb859	Manika Paul Chowdhury	29-May-17 10:32:50 PM	Implemented Feature C
Local History				
	7425afe3	Manika Paul Chowdhury	29-May-17 10:30:30 PM	Code merged: Feature A <-> Featur... Bug-Fix Feature-One
	09fd7b3c	Manika Paul Chowdhury	29-May-17 10:08:28 PM	Feature A change pushed
	df55ecf5	Manika Paul Chowdhury	29-May-17 10:09:12 PM	Feature B change pushed
	d580b131	Manika Paul Chowdhury	29-May-17 10:07:02 PM	Feature Change one
	32e6953e	Manika Paul Chowdhury	29-May-17 09:54:43 PM	Changed the TargetFrameworkVersion to v4.5.2
	42e5113c	Kunal Chowdhury	29-May-17 09:09:15 PM	Set program class type to public
	607170fa	Kunal Chowdhury	28-May-17 01:31:52 PM	Initial commit of the project files
	05c23fc5	Kunal Chowdhury	27-May-17 07:30:11 PM	Initial commit

As we have just learned how to reset a branch, let's learn how to revert a commit from a remote branch.

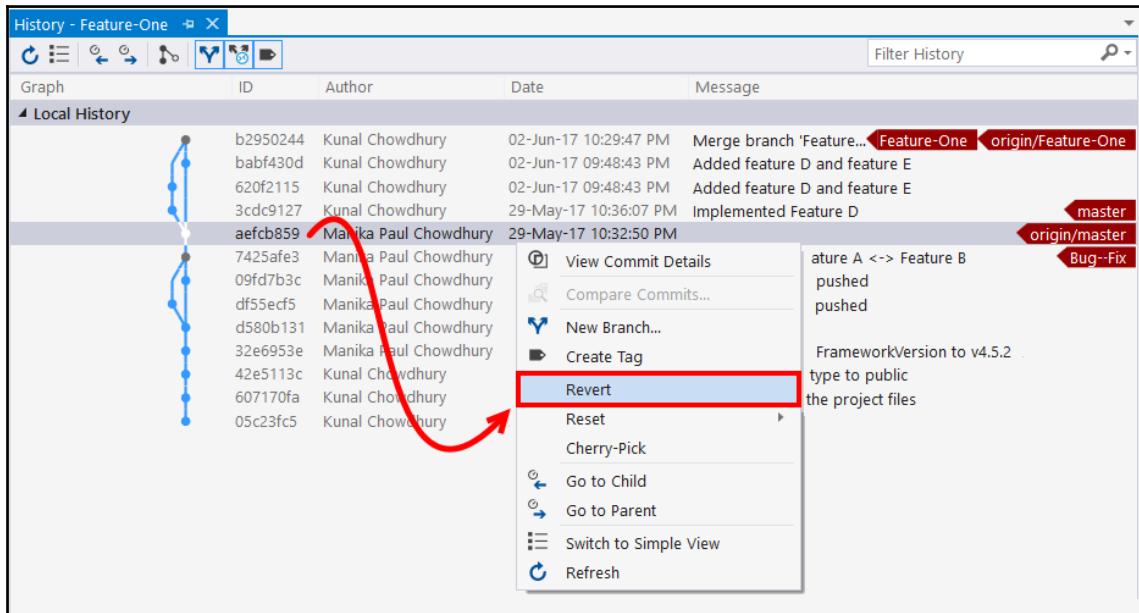
Reverting changes from a remote branch

When you have pushed some commits to the remote repository and would like to undo those changes, you need to use the **Revert** option to create a new commit that will undo all those changes.



Note that history will not be rewritten in the case of a revert. Thus, it is safe to use when working with others.

To revert a change from the shared remote branch, open **Team Explorer** and navigate to the **History** page of the branch. Now, right-click on the commit that you want to revert and click on the **Revert** option from the context menu entries, as shown in the following screenshot:

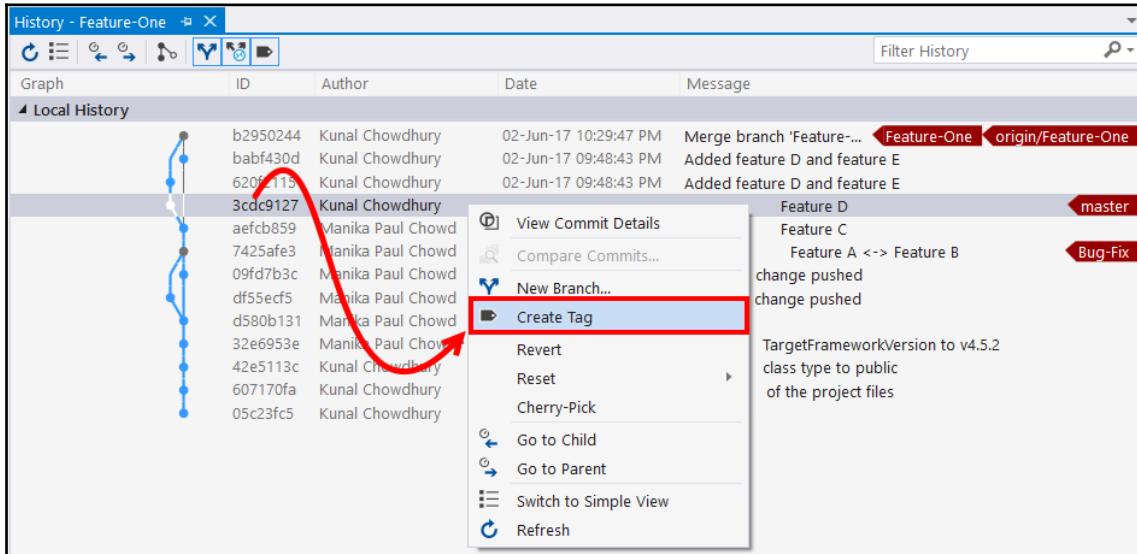


This will create a new commit to undo the changes, which you need to push to your remote branch.

Tagging your commits

Git provides you with the ability to tag some specific points in your commit history as being important. Typically, people use this functionality to mark release points of the code. When you tag a commit, it is easy to identify the last commit that went to a specific release build.

To add a tag to a specific commit, open the **History** page from **Team Explorer**. Right-click on the commit that you want to assign a tag to, and click on **Create Tag** from the context menu, as shown here:

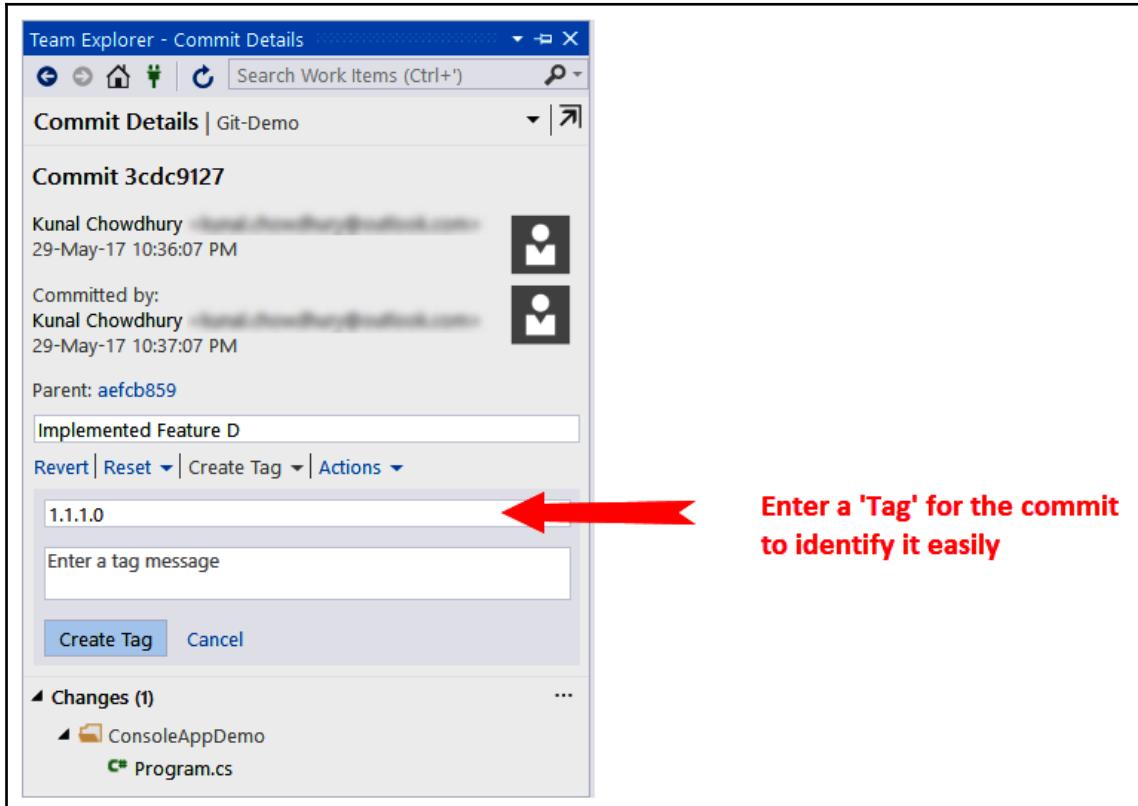


This will open the **Commit Details** dialog inside the **Team Explorer** window, with the complete details of the specific commit. Enter a tag name for it in the appropriate box; you may want to add a tag message and then click on the **Create Tag** button.

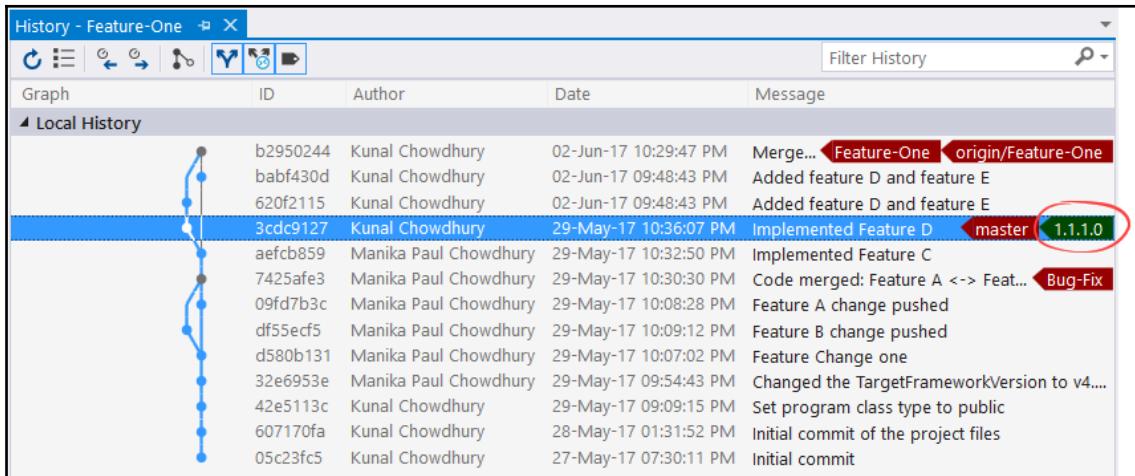
Remember that a tag cannot have a blank space and/or any special characters, and it must be unique within the same repository.



Once the tag has been locally created, you will have to push the additional details to the remote repository so that the tag information is visible to the other team members. If the same tag name has already been created by some other team member, you won't be allowed to push the tag details that you have created:



Upon creating the tag, you will see it marked alongside the commit that you have selected to perform the tagging operation. In future, when you need to pull the remote changes up to that commit history, you can utilize it to download the specific file changes to your local repository:



The screenshot shows a Git commit history for a repository named 'Feature-One'. The commits are listed in chronological order from bottom to top. A commit by Kunal Chowdhury on May 29, 2017, at 10:36:07 PM, with the ID '3cdc9127', has a green tag '1.1.1.0' next to it, which is circled in red. The commit message for this tag is 'Implemented Feature D'. Other commits in the history include merges and feature implementations by Manika Paul Chowdhury, such as 'Merge... < Feature-One < origin/Feature-One', 'Added feature D and feature E', and 'Implemented Feature C'. There are also bug fixes and feature changes like 'Code merged: Feature A <-> Feature B' and 'Feature A change pushed'. The commit history starts with an initial commit on May 27, 2017.

ID	Author	Date	Message
b2950244	Kunal Chowdhury	02-Jun-17 10:29:47 PM	Merge... < Feature-One < origin/Feature-One
babf430d	Kunal Chowdhury	02-Jun-17 09:48:43 PM	Added feature D and feature E
620f2115	Kunal Chowdhury	02-Jun-17 09:48:43 PM	Added feature D and feature E
3cdc9127	Kunal Chowdhury	29-May-17 10:36:07 PM	Implemented Feature D
aefccb859	Manika Paul Chowdhury	29-May-17 10:32:50 PM	Implemented Feature C
7425afe3	Manika Paul Chowdhury	29-May-17 10:30:30 PM	Code merged: Feature A <-> Feature B
09fd7b3c	Manika Paul Chowdhury	29-May-17 10:08:28 PM	Bug-Fix
df55ecf5	Manika Paul Chowdhury	29-May-17 10:09:12 PM	Feature A change pushed
d580b131	Manika Paul Chowdhury	29-May-17 10:07:02 PM	Feature B change pushed
32e6953e	Manika Paul Chowdhury	29-May-17 09:54:43 PM	Feature Change one
42e5113c	Manika Paul Chowdhury	29-May-17 09:15:15 PM	Changed the TargetFrameworkVersion to v4....
607170fa	Kunal Chowdhury	28-May-17 01:31:52 PM	Set program class type to public
05c23fc5	Kunal Chowdhury	27-May-17 07:30:11 PM	Initial commit of the project files
			Initial commit

When adding a tag, make sure that it provides meaningful information for when you need it in future.

Summary

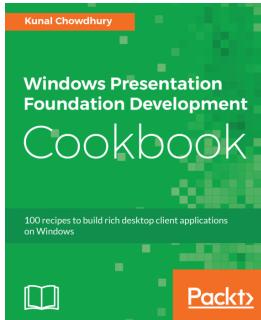
In this chapter, we first learned about the source control repository and how to install Git for Visual Studio 2019. Then, we discussed—in depth—how to use Visual Studio 2019 to interact with the GitHub repository.

During the discussion, we covered how to connect to a remote server and create/clone a remote repository. Then, we learned about Git branches, committing changes to a branch, syncing changes between local and remote repositories, creating pull requests to perform code reviews, and how to approve and merge changes with the master branch.

We have also learned about how to perform rebasing, cherry-picking, tagging a commit, and resetting and reverting changes. I hope that this information was helpful to you in a broader context.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

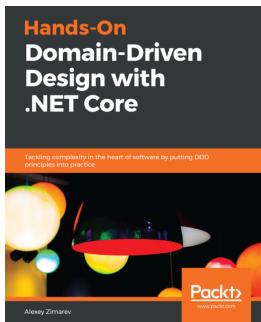


Windows Presentation Foundation Development Cookbook

Kunal Chowdhury

ISBN: 978-1-78839-980-7

- Understand the fundamentals of WPF
- Explore the major controls and manage element layout.
- Implement data binding
- Create custom elements that lead to a particular implementation path.
- Customize controls, styles, and templates in XAML
- Leverage the MVVM pattern to maintain a clean and reusable structure in your code
- Master practical animations
- Integrate WCF services in a WPF application
- Implement WPFs support for debugging and asynchronous operations.



Hands-On Domain-Driven Design with .NET Core

Alexey Zimarev

ISBN: 978-1-78883-409-4

- Discover and resolve domain complexity together with business stakeholders
- Avoid common pitfalls when creating the domain model
- Study the concept of Bounded Context and aggregate
- Design and build temporal models based on behavior and not only data
- Explore benefits and drawbacks of Event Sourcing
- Get acquainted with CQRS and to-the-point read models with projections
- Practice building one-way flow UI with Vue.js
- Understand how a task-based UI conforms to DDD principles

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

- - .NET command-line interface (CLI) 155
 - .NET Core 1.0 156
 - .NET Core application
 - creating, Visual Studio 2019 used 175, 176, 177
 - dependencies, resolving in 165, 166
 - FDD 170
 - publishing 169
 - running 168, 169
 - SCD 171, 172
 - .NET Core applications, publishing with Visual Studio 2019
 - about 177
 - FDD 178, 179, 180
 - SCD 180, 181, 183, 184, 185, 186
 - .NET Core class library
 - creating 162
 - .NET Core CLI
 - solution file, creating 162, 163, 164
 - .NET Core commands 159, 160
 - .NET Core console app
 - creating 160, 161
 - .NET Core project
 - building 166
 - .NET Core solution
 - building 167, 168
 - .NET Core web app
 - building 188, 189
 - creating 186
 - publishing, to Microsoft Azure 190, 191, 192, 193, 194
 - .NET Core
 - about 154
 - installing, with Visual Studio 2019 157, 158
 - overview 155, 156
 - unit testing project, creating 174
- URL 154
- .NET Framework
 - NuGet package library, creating 227, 228

A

- Amazon Web Service (AWS) 109
- amend messages 312
- any type
 - using 212
- App Service plan
 - scaling 150, 151, 152
- array type
 - using 213
- artificial intelligence (AI) 43
- ASP.NET Core application
 - creating 173
- ASP.NET web application
 - creating, with Visual Studio 123, 124
 - deploying, to Azure 123, 124
 - publishing, to cloud as Azure website 125, 126, 127, 128, 129, 130
- Azure account
 - creating 110, 111
- Azure App Service 110
- Azure CDN 110
- Azure Container Service 109
- Azure development
 - Visual Studio 2019, configuring for 112, 113
- Azure DevOps 9
- Azure mobile app
 - creating 133, 134, 135
 - preparing, for data connectivity 136, 137
 - SQL data connection, adding 137
 - SQL database, creating 138, 139, 140, 141, 142
- Azure Virtual Machines 109
- Azure website

creating, from Microsoft Azure portal 114
creating, with Visual Studio 123
managing, from Microsoft Azure portal 119, 120, 122
updating, with Visual Studio 131, 132
Azure websites
 updating, with Visual Studio 133

B

basic datatypes
 any type, to declare variable 212
 array type 213
 bool type 210
 enum type 210
 never type 213
 null type 212
 number type 209
 string type 209
 tuple type 214
 undefined type 211
 void type 211
 working with 208

Binary Application Markup Language (BAML) 65
bool type
 using 210
Breakpoints window
 used, for managing breakpoints 259
breakpoints
 actions, adding 256, 257
 conditions, adding to 253
 conditions, adding with breakpoint filters 255
 conditions, adding with breakpoint hit counters 255
 conditions, adding with conditional expressions 254
 exporting 260
 importing 260
 labels, adding 258, 259
 managing, with Breakpoints window 259
 organizing, in C# source code 247, 249
 used, for debugging C# source code 246

C

C# source code
 breakpoints, organizing 247, 249

debugging, with breakpoints 246
Canvas
 using, as panel 84
changes
 reverting, from remote branch 337, 338
 undoing 334
checkout 309
cherry-pick
 used, for copying commits 333, 334
classes
 abstract classes, defining in TypeScript 216
 class instance, initiating 216
 defining, in TypeScript 214
 inheritance working with, in TypeScript 217
 working with 214
Clipboard History 47, 48
Clipboard Ring 47
cloud computing models
 about 108
 IaaS 109
 PaaS 110
 SaaS 110
cloud computing
 basics 108
code review
 pull requests, creating for 325, 326
 pull requests, working with for 324
Command-Line Interface (CLI) 227
commit changes 312
commits
 messages, amending to 317
 tagging 338, 341
Common Language Runtime (CLR) 64, 89
Common Type System (CTS) 64
complex DataTips
 displaying, with visualizers 268, 269
const keyword
 used, for declaring constants 208
constant variables
 declaring, const keyword used 208
converters
 using, while data binding 96, 97, 98

D

data binding, in WPF
 OneTime 91
 OneWay 91
 OneWayToSource 91
 TwoWay 91
data binding
 converters, using 96, 97, 98
 in WPF 90, 92, 93, 94, 95
data trigger 101
DataTips
 exporting 269
 importing 269
 inspecting, in Autos window 263
 inspecting, in Locals window 264
 inspecting, in Watch window 264, 266, 268
 inspecting, in Watch windows 263
 using, during debugging 260
debugger
 about 245
 execution steps 250, 251, 252, 253
 used, for displaying debugging information 270, 272
debugging windows
 searching in 48
dependencies
 resolving, in .NET Core applications 165, 166
dependency property 89
Device Drivers 64
DirectX 64
DropBox 110

E

ECMAScript 2015 209, 214
ECMAScript 6 214
enum type
 using 210
event trigger 104, 105
Extensible Application Markup Language (XAML)
 63
 about 62
 collection syntax 67
 content syntax 66, 67
 event attribute syntax 68

object element syntax 65
overview 65
property attribute syntax 66
property element syntax 66

F

failed tests
 navigating 297
framework-dependent deployment (FDD) 156

G

Git branches
 deleting 311
 local branch, creating 307
 local branch, pushing to remote repository 310
 switching, between branches 309
 working with 307
Git commit history
 commits, copying with cherry-pick 333, 334
 rewriting, by rebasing changes 331, 332
 working with 330
Git repository
 cloning 305, 306
 creating 304
 working with 303

Git

 installing, for Visual Studio 2019 301, 302
GitHub 9
Google App Engine 110
Google Apps 110
Google Compute Engine (GCE) 109
Graphics Device Interface (GDI) 64

Grid

 using, as WPF panel 81, 82, 83

H

Hello TypeScript application
 building 200, 202, 203
Heroku 110

I

IaaS 109
Immediate Window
 using, during code debugging 272, 273

inheritance 217
input/output (I/O) model 197
interfaces
defining, in TypeScript 218
extending, another interface 219
extending, classes 220
extending, multiple interfaces 220
implementing, classes used 219
working with 214

J

JavaScript 196

K

Kernel 64

L

layouts, in WPF
about 79, 81
Canvas, using as panel 84
Grid, using as WPF panel 81, 82, 83
panels 79, 80
StackPanel, using to define stacked layout 83
UniformGrid, using to place elements in uniform cells 87, 88
WPF DockPanel, using to dock child elements 85
WrapPanel, using to reposition elements 86, 87

let keyword
used, for declaring variables 207

LINQ queries
foreach loops, converting into 42
Live Share session
about 51
creating 52, 53
joining 54, 55, 56, 57
managing 57, 58

Live Unit Testing, with Visual Studio 2019
about 289
coverage information, in editor 281
example 293, 294, 295, 296
Live Unit Testing, integrating in Test Explorer 282
packages.config file 292
testing project, configuring 291

unit testing framework support 281
Live Unit Testing
about 279
in Visual Studio 2019 280
integrating, in Test Explorer 282
test methods/projects, excluding 287, 288
test methods/projects, including 287, 288
local branch
pushing, to remote repository 310
resetting, to previous state 335, 337
local Git repository
and remote Git repository, changes syncing between 317
local repository
remote repository, changes merging with 320
staged changes, committing to 314, 315
staging changes 313, 314
uncommitted changes, discarding 315, 316

M

Media Integration Library 64
Media Integration Library (MIL) 64
merge conflicts
resolving 321, 323
metadata
creating, in nuspec file 229, 230, 231, 232
Microsoft Azure portal
App Service plan, creating 118
Azure website, creating from 114
Azure websites, managing from 119, 120, 122
reference link 114
web application, creating 115, 116, 117

Microsoft Azure
.NET Core web app, publishing to 190, 191, 192, 193, 194
mobile app service, integrating in Windows application
about 143
API call, integrating 144, 146, 149
model, creating 143, 144
service client, creating 143, 144
Most Recently Used (MRU) 33
MSTest 281
multi data trigger 102, 104
multi trigger 100, 101

multiple .NET Framework
 NuGet packages, building 234, 235, 236

N

never type
 using 213
Node Package Manager (NPM) 197
Node.js
 about 197
 download link 198
 URL 198
NuGet 222
NuGet package gallery
 reference link 222
NuGet package library
 creating, for .NET Framework 227, 228
NuGet Package Manager 222, 223, 224, 225,
 226, 227
NuGet packages
 building 232, 233
 building, for multiple .NET Frameworks 234,
 235, 236
 building, with dependencies 236, 237
 managing 240, 241
 publishing, to NuGet store 237, 238, 239
 reference link 237
NuGet store
 NuGet package, publishing to 237, 238, 239
null type
 using 212
number type
 using 209
 NUnit 281
nuspec file
 metadata, creating in 229, 230, 231, 232

O

Object-Oriented Programming (OOP) 214
Office 365 110
one-click code cleanup 45, 46, 47
OneDrive 110
OpenShift 110
OS Core 64

P

PaaS 110
pin DataTips
 using, for debugging 261, 262
Portable Executable (PE) format 170
Presentation Core 64
Presentation Framework 64
Product Updates 58, 59
Program Database (PDB) 245
property trigger 99
pull requests
 creating, for code review 325, 326
 reviewing 326, 329
 working with, for code review 324

Q

quick actions, improvements for code refactoring
 about 37
 arguments, unwrapping 38
 arguments, wrapping 38
foreach loops, converting into generalized LINQ
 queries 42, 43
if statements, inverting 37
inline variables 39
members, pulling to base type 40, 41
namespaces, adjusting to match folder structure
 41
quick launch 34

R

remote branch
 changes, reverting from 337, 338
remote Git repository
 and local Git repository, changes syncing
 between 317
remote repository
 changes, fetching in 319, 320
 changes, merging in with local repository 320
 changes, pushing to 318, 319
 local branch, pushing to 310
runtime identifiers (RIDs) 173

S

self-contained deployment (SCD) 156
Source Code Control System (SCCS) 300
source control servers
 connecting to 302, 303
SQL data connection
 adding, to Azure Mobile App 137
SQL database
 creating, in Azure Mobile App 138, 139, 140,
 141, 142
StackPanel
 using, to define stacked layout 83
stage changes 312
staged changes
 committing, to local repository 314, 315
staging changes
 to local repository 313, 314
Start window
 local folder, opening from 31
 overview 26
 project, creating from 32, 33, 34
 project, opening from 30
 repository, cloning from 27, 28, 29
 solution, opening from 30
string type
 using 209

T

templated strings 209
Test Explorer
 Live Unit Testing, integrating in 282
Time Travel Debugging
 about 49
 performing 50, 51
triggers, in WPF
 about 99
 data trigger 101
 event trigger 104, 105
 multi data trigger 102, 104
 multi trigger 100, 101
 property trigger 99
tuple type
 using 214
TypeScript compiler 196

TypeScript configuration file

 about 204
 compileOnSave 205
 compilerOptions 205
 exclude 205
 files 205
 include 205

TypeScript version

 reference link 196

TypeScript

 about 196
 classes, defining 214
 constants, declaring with const keyword 208
 installation used, for setting up development
 environment 197
 installation, through NPM 197, 198, 199
 installation, through Visual Studio installer 199,
 200
 interfaces, defining in 218
 URL 196
 variables, declaring 205
 variables, declaring with let keyword 207
 variables, declaring with var keyword 206

U

uncommitted changes
 discarding 312
 discarding, from local repository 315, 316
undefined type
 using 211
UniformGrid
 using, to place elements in uniform cells 87, 88
unit testing 279
units 279
Universal Windows Platform (UWP) 11
unpin DataTips
 using, for debugging 261, 262
UpdateSourceTrigger
 values 95
User Interfaces (UIs) 62
User32 64

V

var keyword
 issues 206

used, for declaring variables 206
variables
declaring, in TypeScript 205
declaring, let keyword used 207
declaring, var keyword used 206
Visual Studio 2019 installation experience
about 10, 11
installing, from command line 20, 21, 22, 23
installing, with online installer 13, 14, 15, 16, 17,
18
modifying 23, 24
offline installer, creating 19, 20
overview 12
uninstalling 25
Visual Studio 2019, configuring for Live Unit Testing
about 283
Live Unit Testing component, installing 283
Live Unit Testing, pausing 286, 287
Live Unit Testing, starting 286, 287
settings 284, 286
Visual Studio 2019
.NET Core, installing with 157, 158
about 9
configuring, for Azure development 112, 113
download link 13
Git, installing for 301, 302
Live Unit Testing 280
search, improving 34, 36
used, for creating .NET Core applications 175,
176, 177
used, for publishing .NET Core applications 177
Visual Studio Code
about 200
URL 201
Visual Studio Community Edition 13
Visual Studio Community license
reference link 13
Visual Studio debugger tools 245
Visual Studio Diagnostics Tools
using 273, 274, 275, 276, 277
Visual Studio IntelliCode
about 43
component, installing 44
need for 44
Visual Studio Team Services (VSTS) 302
Visual Studio
ASP.NET web application, creating with 123,
124
Azure website, creating with 123
Azure website, updating with 131, 132, 133
visualizers
used, for displaying complex DataTips 268, 269
void type
using 211

W

Windows application
mobile app service, integrating 143
WPF application
building 68, 69
WPF architecture
about 63, 64
Common Language Runtime 64
Media Integration Library 64
OS Core 64
Presentation Core 64
Presentation Framework 64
WPF DockPanel
using, to dock child elements 85
WPF project structure 73, 74
WPF project
starting 70, 71, 72
WPF property system 89, 90
WPF
data binding 90, 92, 93, 94, 95
WrapPanel
using, to reposition elements 86, 87

X

XAML Designer
about 75
controls, adding in XAML 76, 77, 78, 79
options 75
xUnit.net 281