

Visual Studio Code Distilled

Evolved Code Editing for Windows,
macOS, and Linux

Alessandro Del Sole

The Apress logo consists of the word "apress" in a lowercase, sans-serif font, followed by a registered trademark symbol (®).

Visual Studio Code Distilled

**Evolved Code Editing for
Windows, macOS, and Linux**

Alessandro Del Sole

Apress®

Visual Studio Code Distilled: Evolved Code Editing for Windows, macOS, and Linux

Alessandro Del Sole
Cremona, Italy

ISBN-13 (pbk): 978-1-4842-4223-0
<https://doi.org/10.1007/978-1-4842-4224-7>

ISBN-13 (electronic): 978-1-4842-4224-7

Library of Congress Control Number: 2018965198

Copyright © 2019 by Alessandro Del Sole

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

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Joan Murray
Development Editor: Laura Berendson
Coordinating Editor: Jill Balzano

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit <http://www.apress.com/rights-permissions>.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at <http://www.apress.com/bulk-sales>.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-4223-0. For more detailed information, please visit <http://www.apress.com/source-code>.

Printed on acid-free paper

To Angelica, the love of my life.

Table of Contents

| | |
|---|-------------|
| About the Author | xi |
| Acknowledgments | xiii |
| Introduction | xv |
| | |
| Chapter 1: Introducing Visual Studio Code..... | 1 |
| Introducing Visual Studio Code | 2 |
| When and Why Visual Studio Code..... | 3 |
| Installing and Configuring Visual Studio Code | 5 |
| Installing Visual Studio Code on Windows..... | 6 |
| Installing Visual Studio Code on macOS..... | 9 |
| Installing Visual Studio Code on Linux..... | 10 |
| Localization Support..... | 12 |
| Updating Visual Studio Code..... | 13 |
| Previewing Features with Insiders Builds | 15 |
| Summary..... | 17 |
| | |
| Chapter 2: Getting to Know the Environment | 19 |
| The Welcome Page..... | 20 |
| The Code Editor..... | 21 |
| Reordering, Resizing, and Zooming Editor Windows | 22 |
| The Status Bar | 23 |
| The Activity Bar | 24 |

TABLE OF CONTENTS

| | |
|--|-----------|
| The Side Bar..... | 26 |
| The Explorer Bar..... | 26 |
| The Search Tool | 31 |
| The Git Bar | 32 |
| The Debug Bar | 34 |
| The Extensions Bar..... | 35 |
| The Settings Button | 36 |
| Navigating Between Files | 36 |
| The Command Palette..... | 37 |
| The Panels Area | 38 |
| The Problems Panel..... | 38 |
| The Output Panel..... | 40 |
| The Debug Console Panel..... | 41 |
| Working with the Terminal..... | 42 |
| Summary..... | 43 |
| Chapter 3: Language Support and Code Editing Features | 45 |
| Language Support..... | 46 |
| Working with C# and C++ | 47 |
| Basic Code Editing Features | 48 |
| Working with Text | 48 |
| Syntax Colorization..... | 49 |
| Delimiter Matching and Text Selection | 51 |
| Code Block Folding | 51 |
| Multicursors | 52 |
| Reusable Code Snippets..... | 52 |
| Word Completion | 54 |
| Minimap Mode..... | 55 |

TABLE OF CONTENTS

| | |
|--|-----------|
| Whitespace Rendering and Breadcrumbs | 56 |
| Markdown Preview..... | 58 |
| Evolved Code Editing..... | 59 |
| Working with IntelliSense..... | 60 |
| Parameter Hints..... | 62 |
| Inline Documentation with Tooltips | 62 |
| Go To Definition..... | 63 |
| Find All References..... | 65 |
| Peek Definition | 66 |
| Renaming Symbols and Identifiers..... | 67 |
| Live Code Analysis..... | 68 |
| Summary..... | 76 |
| Chapter 4: Working with Files and Folders..... | 77 |
| Visual Studio Code and Project Systems | 77 |
| Working with Individual Files | 78 |
| Creating Files..... | 79 |
| File Encoding, Line Terminators, and Line Browsing | 80 |
| Working with Folders and Projects | 82 |
| Opening a Folder | 84 |
| Opening .NET Core Solutions..... | 86 |
| Opening JavaScript and TypeScript Projects | 87 |
| Opening Loose Folders | 88 |
| Working with Workspaces..... | 89 |
| Creating Workspaces..... | 91 |
| Opening Existing Workspaces | 92 |
| Workspace Structure | 92 |
| Summary..... | 93 |

TABLE OF CONTENTS

| | |
|--|------------|
| Chapter 5: Customizing Visual Studio Code | 95 |
| Customizations and Extensions Explained..... | 95 |
| Customizing Visual Studio Code..... | 97 |
| Theme Selection..... | 97 |
| Customizing the Environment..... | 99 |
| Customizing Key Bindings | 106 |
| Summary..... | 110 |
| Chapter 6: Installing and Managing Extensions | 111 |
| Installing Extensions | 111 |
| Extension Recommendations | 115 |
| Useful Extensions | 117 |
| Managing Extensions | 118 |
| Configuring Extensions..... | 120 |
| Hints About Extension Authoring..... | 122 |
| Summary..... | 122 |
| Chapter 7: Source Control with Git | 125 |
| Source Control in Visual Studio Code..... | 125 |
| Downloading Other Source Control Providers | 126 |
| Managing Repositories | 127 |
| Initializing a Local Git Repository | 128 |
| Creating a Remote Repository..... | 130 |
| Handling File Changes | 132 |
| Staging Changes | 134 |
| Managing Commits | 135 |
| Working with the Git Command Line Interface | 137 |

TABLE OF CONTENTS

| | |
|--|------------|
| Creating and Managing Branches..... | 138 |
| Switching to a Different Branch | 140 |
| Merging from a Branch..... | 140 |
| Deleting Branches | 141 |
| Adding Power to the Git Tooling with Extensions..... | 142 |
| Git History..... | 142 |
| GitLens..... | 144 |
| GitHub Pull Requests | 148 |
| Working with Azure DevOps and Team Foundation Server | 150 |
| Summary..... | 154 |
| Chapter 8: Automating Tasks | 157 |
| Understanding Tasks..... | 158 |
| Tasks Types | 159 |
| Running and Managing Tasks..... | 159 |
| The Default Build Task..... | 164 |
| Auto-Detected Tasks..... | 164 |
| Configuring Tasks | 166 |
| Running Files with a Default Program..... | 188 |
| Summary..... | 189 |
| Chapter 9: Running and Debugging Code | 191 |
| Creating Applications | 191 |
| Creating .NET Core Projects | 192 |
| Creating Projects on Other Platforms | 195 |
| Debugging Your Code..... | 196 |
| Configuring the Debugger | 198 |

TABLE OF CONTENTS

| | |
|--|------------|
| Managing Breakpoints..... | 202 |
| Debugging an Application..... | 203 |
| Supporting Azure, Docker, and Artificial Intelligence..... | 207 |
| Summary..... | 209 |
| Index..... | 211 |

About the Author

Alessandro Del Sole is Senior Software Engineer for a healthcare company, building mobile apps for doctors and dialysis patients. He has been in the software industry for almost 20 years, focusing on Microsoft technologies such as .NET, C#, Visual Studio, and Xamarin. He has been a trainer, consultant, and a Microsoft MVP since 2008 and is the author of many technical books. He is a Xamarin Certified Mobile Developer, Microsoft Certified Professional, and a Microsoft Programming Specialist in C#.

Acknowledgments

Thanks to Joan Murray, Jill Balzano, Laura Berendson and to everyone at Apress for the opportunity and the great teamwork on this book.

Special thanks to the technical editor, Dr. James McCaffrey, who contributed to the quality and accuracy of the contents.

Special thanks to my girlfriend Angelica, who understands and never complains about the time I spend on writing books.

Introduction

One of the most common requirements in software development today is building applications and services that run on multiple systems and devices, especially with the continued expansion of cloud and artificial intelligence services.

Developers have many options to build cross-platform and cross-device software, from languages to development platforms and tools. However, in most cases such tools rely on proprietary systems, therefore creating strong dependencies. Moreover, most development tools target specific platforms and development scenarios. Microsoft Visual Studio Code makes a step forward, by providing a fully featured development environment for Windows, macOS, and Linux that not only offers advanced coding features but also integrated tools that span across the entire application lifecycle from coding to debugging to team collaboration. In this book, developers with any skill will learn how to leverage Visual Studio Code to target scenarios such as web, cloud, and mobile development with the programming language of their choice, providing guidance to build apps for any system and any device.

CHAPTER 1

Introducing Visual Studio Code

Visual Studio Code is not just another evolved notepad with syntax colorization and automatic indentation. Instead, it is a very powerful code-focused development environment expressly designed to make it easier to write web, mobile, and cloud applications using languages that are available to different development platforms and to support the application development lifecycle with a built-in debugger and with integrated support to the popular Git version control engine.

With Visual Studio Code, you can work with individual code files or with structured file systems based on folders. This chapter provides an introduction to Visual Studio Code giving you information on when and why you should use it, as well as about installing and configuring the program on the different supported operating systems.

Note Across the book, I will refer to the product with its full name, Visual Studio Code, and its friendly names VS Code and Code interchangeably.

Introducing Visual Studio Code

Visual Studio Code has been the first cross-platform development tool in the Microsoft Visual Studio family that runs on Windows, Linux, and macOS. It is free, open source (<https://github.com/Microsoft/vscode>), and it is definitely a code-centric tool, which makes it easier to edit code files and folder-based project systems as well as writing cross-platform web and mobile applications over the most popular platforms, such as Node.js and .NET Core, with integrated support for a huge number of languages and rich editing features such as IntelliSense, finding symbol references, quickly reaching a type definition, and much more.

Visual Studio Code is based on Electron (<https://electronjs.org/>), a framework for creating cross-platform applications with native technologies, and combines the simplicity of a powerful code editor with the tools a developer needs to support the application lifecycle development, including debuggers and version control integration based on Git. It is therefore a complete development tool, rather than being a simple code editor. For more advanced coding and development, you will certainly consider Microsoft Visual Studio 2017 on Windows and Visual Studio for Mac on macOS, but Visual Studio Code can be really helpful in many situations.

In this book, you learn how to use Visual Studio Code and how to get the most out of it, seeing how you can use it both as a powerful code editor and as a complete environment for end-to-end development. Except where necessary, figures are based on the Microsoft Windows 10 operating system, but there is no difference on Linux and macOS. Also, Visual Studio Code includes a number of color themes that style its layout. In this book, figures are based on the so-called Visual Studio Light Theme, so you might see different colors. Chapter 5, “Customizing Visual Studio Code,” explains how to change the theme, but if you want to be consistent with the book’s figures, simply select **File > Preferences > Color Theme** and select the Visual Studio Light Theme. It is worth mentioning that the theme you select does not affect at all the features described in this book.

When and Why Visual Studio Code

Before you learn how to use Visual Studio Code, what features it offers, and how it provides an improved code editing experience, you have to clearly understand its purpose. Visual Studio Code is not a simple code editor; rather it is a powerful environment that puts writing code at its center. The main purpose of Visual Studio Code is making it easier to write code for web, mobile, and cloud platforms for any developers working on different operating systems, such as Windows, Linux, and macOS, making you independent from proprietary development environments.

For a better understanding, let's consider an example based on ASP.NET Core, the cross-platform, open source technology able to run on Windows, Linux, and macOS that Microsoft produced to create portable web applications; forcing you to build cross-platform, portable web apps with Microsoft Visual Studio 2017 would make you dependent on this Integrated Development Environment (IDE). You could argue that the Visual Studio 2017 Community edition is free of charge, but it only runs on Windows. On the contrary, though it is not certainly intended to be a replacement for more powerful and complete environments such as its major brother, Visual Studio Code can run on a variety of operating systems and can manage different project types, as well as the most popular languages. To accomplish this, Visual Studio Code provides the following core features:

- Built-in support for coding with many languages, including those you typically use in cross-platform development scenarios, with advanced editing features and support for additional languages via extensibility
- Built-in debugger for Node.js, with support for additional debuggers (such as .NET Core and Mono) via extensibility

- Version control based on the popular Git engine, which provides an integrated experience for collaboration supporting code commits and branches, and that is the proper choice for a tool intended to work with possibly any language

In order to properly combine all these features into one tool, Visual Studio Code provides a coding environment based on folders, which makes it easy to work with code files that are not organized within projects and offers a unified way to work with different languages. Starting from this assumption, Code offers an advanced editing experience with features that are common to any supported languages, plus some features that are available to specific languages. As you learn throughout the book, Code also makes it easy to extend its built-in features by supplying custom languages, syntax coloring, editing tools, debuggers, and much more via a number of extensibility points. It is a code-centric tool, with primary focus on web, cross-platform code. That said, it does not provide all of the features you need for full, more complex application development and application lifecycle management and is not intended to be the proper choice with some development platforms. If you have to make a choice, consider the following points:

- Visual Studio Code can produce binaries and executable files only if the language you use has support to do so through a debugger. If you use a language for which there is no extensive support (e.g., Visual Basic), Visual Studio Code is not able to invoke a compiler. You can workaround this by implementing task automation, discussed in Chapter 8, "Automating Tasks," but this is different than having the compilation process integrated.

- Visual Studio Code has no designers, so creating an application's user interface can only be done by writing all of the related code manually. As you can imagine, this is fine with some languages and for some scenarios, but it can be very complicated with some kinds of applications and development platforms, especially if you are used to work with the powerful graphical tools available in Microsoft Visual Studio.
- It is a general purpose tool and is not the proper choice for specific development scenarios such as building Windows desktop applications.

If your requirements are different, consider instead Microsoft Visual Studio 2017 or Microsoft Visual Studio for Mac, which are optimized for building, testing, deploying, and maintaining multiple types of applications.

Now that you have a clearer idea of Code's goals, you are ready to learn the amazing editing features that put it on the top of any other code editor.

Installing and Configuring Visual Studio Code

Installing Visual Studio Code is an easy task. In fact, you can simply visit <https://code.visualstudio.com> from your favorite browser, and the web page will detect your operating system, suggesting the appropriate installer. Figure 1-1 shows how the download page appears on Windows.

CHAPTER 1 INTRODUCING VISUAL STUDIO CODE

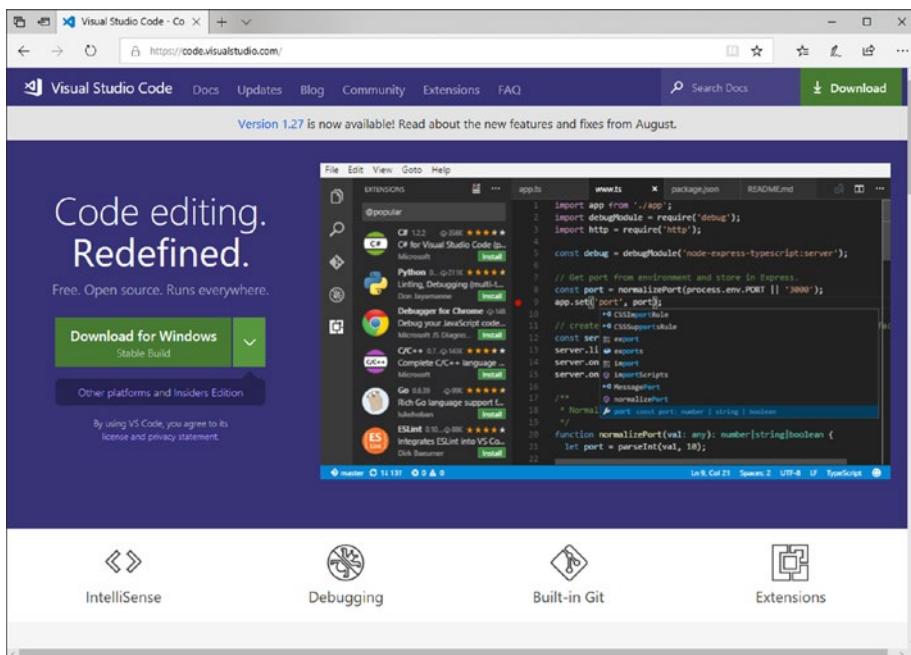


Figure 1-1. The download page for Visual Studio Code

In the next paragraphs, you will learn tips for installing Code on the various supported systems.

Note The latest stable release at the time of this writing is version 1.27.2, released in August 2018 and called August Recovery.

Installing Visual Studio Code on Windows

Visual Studio Code can be installed on Windows 7, 8, and 10. For this operating system, Visual Studio Code is available with two installers: a global installer and a user-level installer. The first installer requires administrative privileges for installation and makes Code available to all

users. The second installer makes Code available only to the currently logged user, but it does not require administrative privileges.

The latter is the choice I recommend, especially if you work within a corporate environment and you do not have administrative privileges to install software on your PC. The **Download for Windows** button that you can see in Figure 1-1 will automatically download the global installer. If you instead wish to download the user-level installer, click the arrow at the right of the button and then click the **User Installer** hyperlink. It is worth mentioning that Visual Studio Code is available in two versions, 32 bit and 64 bit. The download page will automatically suggest the version that matches your operating system architecture, but if you wish to download a different installer, you can click the arrow and then click **Other downloads**.

Once the download has been completed, launch the installer and simply follow the guided procedure as you are already used to do with most of Windows programs. During the installation, you will be prompted to specify how you want to integrate shortcuts to Visual Studio Code in the Windows' shell. In the Select Additional Tasks dialog, make sure you select (at least) the following options:

- **Add “Open With Code” action to Windows Explorer file context menu**, which allows for right-clicking a code file in the Explorer and opening such a file with VS Code.
- **Add “Open With Code” action to Windows Explorer directory context menu**, which allows for right-clicking a folder in the Explorer and opening such a folder with VS Code.
- **Add to PATH (available after restart)**, which adds the VS Code's pathname to the PATH environment variable, making it easy to run Visual Studio Code from the command line without typing the full path.

Note Some antivirus and system protection tools, such as Symantec Endpoint Protection, might block the installation of some files that are recognized as false positives. In most cases this will not prevent Visual Studio Code from working, but it is recommended that you disable the protection tool before installing Code or, if you do not have elevated permissions, that you ask your administrator to do it for you.

A specific dialog will inform you once the installation process has completed. The installation folder for the user-level installer is C:\Users\proga\AppData\Local\Programs\Microsoft VS Code, while the installation folder for the global installer is C:\Program Files\Microsoft VS Code on 64-bit systems and C:\Program Files(x86)\Microsoft VS Code on 32-bit systems. You will find a shortcut to Visual Studio Code in the Start menu and on the Desktop, if you selected the option to create a shortcut during the installation. When started, Visual Studio Code appears like in Figure 1-2.

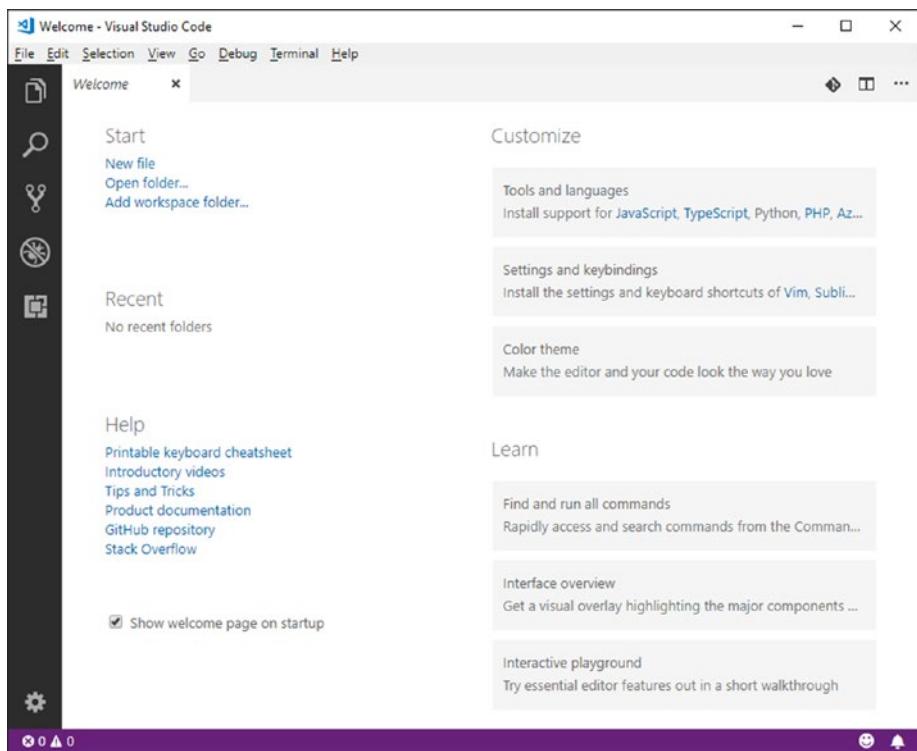


Figure 1-2. Visual Studio Code running on Windows

Installing Visual Studio Code on macOS

Installing VS Code on macOS is extremely simple. From the download page, simply click the **Download for macOS** button and wait for the download to complete. On macOS, Visual Studio Code works as an individual program, and therefore you simply need to double-click the downloaded file to start the application. Figure 1-3 shows Visual Studio Code running on macOS.

CHAPTER 1 INTRODUCING VISUAL STUDIO CODE

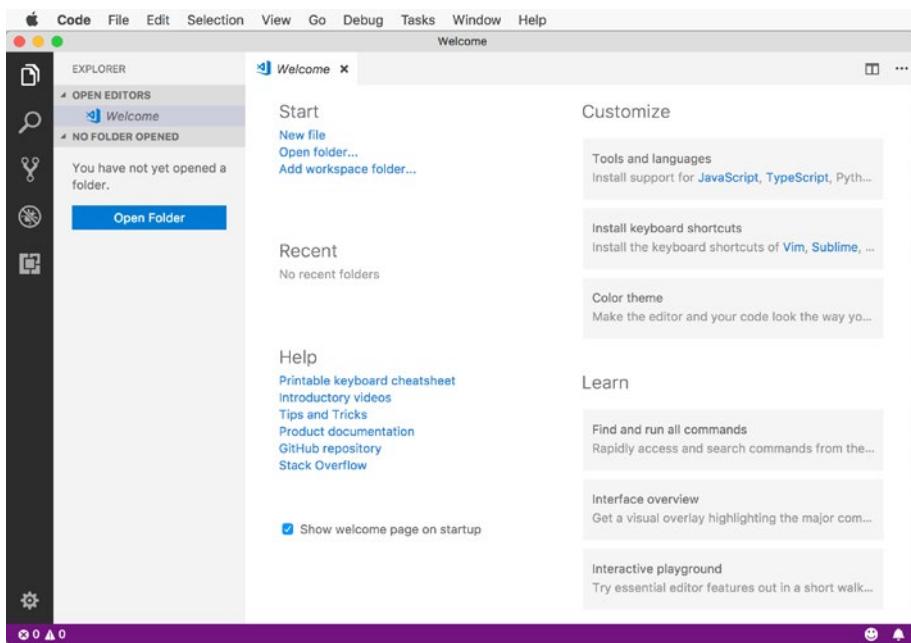


Figure 1-3. Visual Studio Code running on macOS

Installing Visual Studio Code on Linux

Linux is a very popular operating system and many derived distributions exist, so there are different installers available depending on the distribution you are using. For the Ubuntu and Debian distributions, you will need the .deb installer. For the Red Hat Linux, Fedora, and SUSE distributions, you will need the .rpm installer. This clarification is important because, differently from Windows and macOS, the browser might not be able to automatically detect the Linux distribution you are using, and therefore it will offer both options.

Once installed, you will simply need to click the Show Applications button on the desktop and then the Visual Studio Code shortcut. Figure 1-4 shows Visual Studio Code running on Ubuntu.

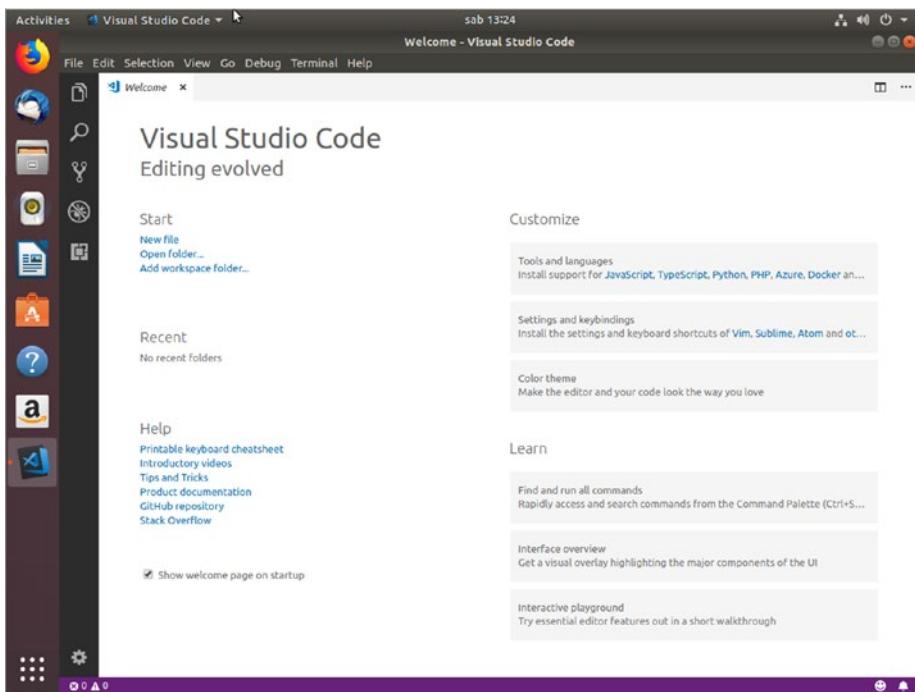


Figure 1-4. Visual Studio Code running on Ubuntu

Note If you are a Windows user and want to try Visual Studio Code on a Linux distribution, you can create a virtual machine with the Hyper-V tool. For example, you might install the latest Ubuntu version (www.ubuntu.com/download/desktop) as an ISO image and use it as an installation media in Hyper-V. On macOS, you need to purchase the Apple Parallels Desktop software separately in order to create virtual machines, but you can basically do the same.

Localization Support

Visual Studio Code ships in English, but it can be localized in many other supported languages and cultures. When started, VS Code checks for the operating system language and, if different from English, it shows a popup suggesting to install a language pack for the culture of your operating system. The localization support can be also enabled manually.

To accomplish this, select **View > Command Palette**. When the text box appears at the top of the page, type the following command:

> Configure Display Language

This will open a file called `locale.json`, which is the place where Visual Studio Code stores the localization information. Figure 1-5 shows how this file appears in the editor. As you can see in the comments, there is a link to the documentation that contains the full list of supported cultures.

Note The Command Palette will be discussed thoroughly in the next chapter.

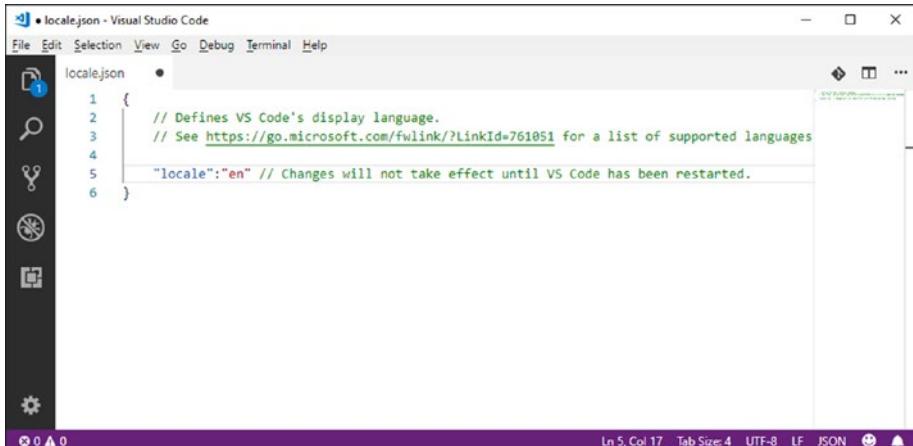


Figure 1-5. Changing the localization for Visual Studio Code

For instance, if you wanted to change the localization from English to Italian, you would replace en with it, saving your changes. At restart, Visual Studio Code will apply the new localization downloading the language pack it needs.

Updating Visual Studio Code

Visual Studio Code is configured to receive automatic updates in the background and, usually, Microsoft releases updates monthly.

Note Because VS Code receives monthly updates, some features might have been updated at the time of your reading, and others might be totally new. This is a necessary clarification you should keep in mind while reading, and it is also the reason why I will also provide links to the official documentation, so that you can stay up to date more easily.

Additionally, you can manually check for updates with **Help ▶ Check for Updates** on Windows and Linux and with **Code ▶ Check for Updates** on macOS. If you do not want to receive automatic updates and prefer manual updates, you can disable automatic updates by selecting **File ▶ Preferences ▶ Settings** and then, in the **Update** section, disable the background updates option. Figure 1-6 shows an example.

CHAPTER 1 INTRODUCING VISUAL STUDIO CODE

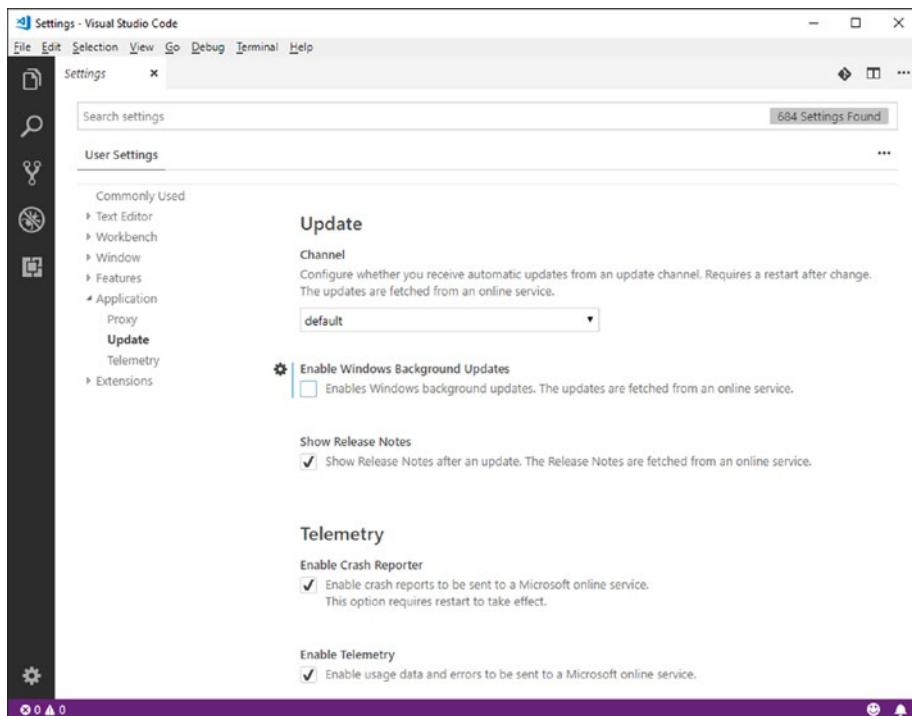


Figure 1-6. Disabling automatic updates

You will follow the same steps to re-enable updates in the background. Whenever Visual Studio Code receives an update, you will receive a notification that suggests you to restart Code in order to apply changes. The first time you restart Visual Studio Code after an update, you will see the release note for the version that was installed, as demonstrated in Figure 1-7.

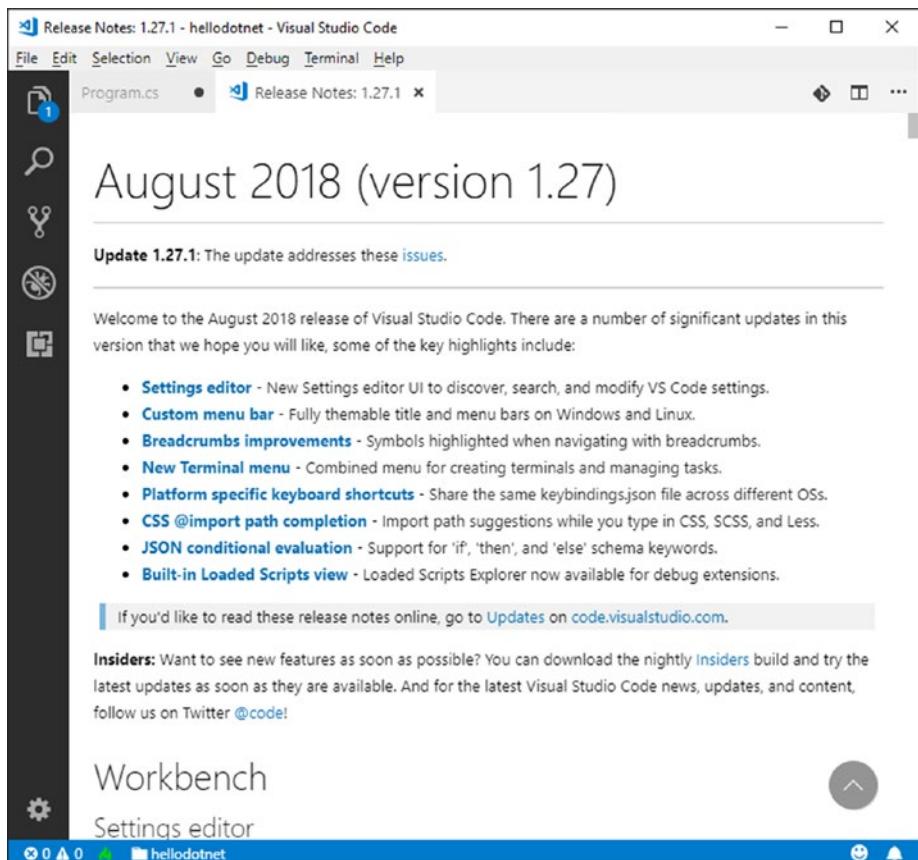


Figure 1-7. VS Code release notes

Release notes contain the list of new and updated features, as well as hyperlinks that will open the proper feature page in the documentation.

Previewing Features with Insiders Builds

By default, the download page of the Visual Studio Code's web site allows you to download the latest stable build. However, Microsoft periodically also releases preview builds of Visual Studio Code called Insiders builds that you can download to have a look at new and updated upcoming features before they are released to the general public.

CHAPTER 1 INTRODUCING VISUAL STUDIO CODE

Insiders builds can be downloaded from <https://code.visualstudio.com/insiders>, and follow the same installation rules described previously for each operating system. They have a different icon color, typically a green icon instead of a blue icon, and the name you see in the application bar is Visual Studio Code - Insiders instead of Visual Studio Code (see Figure 1-8).

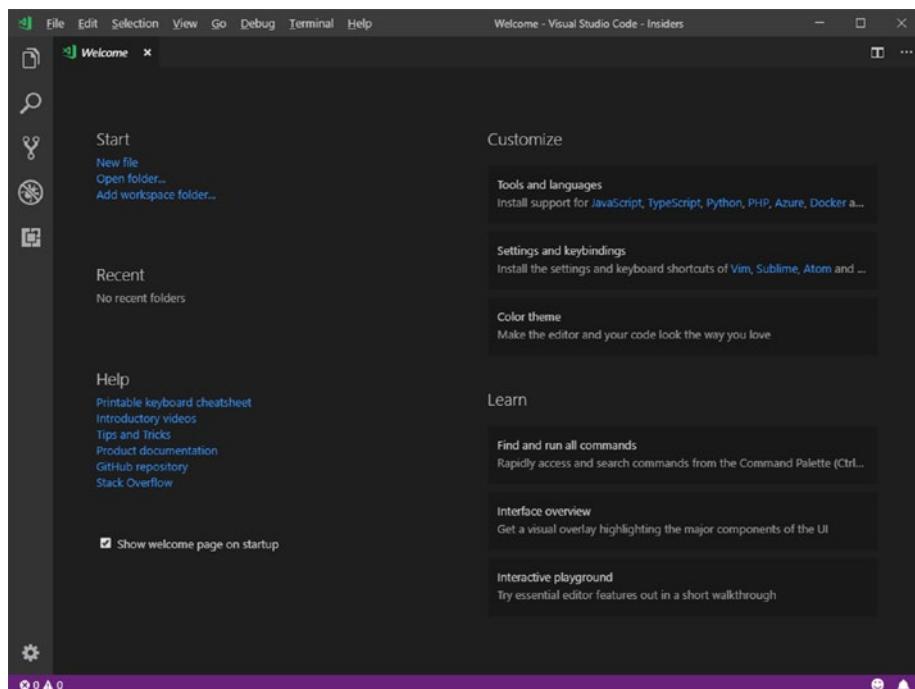


Figure 1-8. Visual Studio Code Insiders builds

Insiders builds and stable builds can work side by side without any issues. Because each lives in its own environment, your setting customizations and extensions you installed on the stable build will not be automatically available to the Insiders build and vice versa, so you will need to provide them again.

Insiders builds are a very good option to have a look at what is coming with Visual Studio Code, but because they are not stable, final builds, it is not recommended you use them in production or with code you will release to production.

Summary

Visual Studio Code is not a simple code editor but a fully featured development environment optimized for web, mobile, and Cloud development. In this chapter, you saw how to install Visual Studio Code on Windows, macOS, and Linux distributions, learning how to select the appropriate installers and fine-tune the setup process. You also saw how to configure localization and updates. Finally, you had a look at the Insiders build, which offer previews of upcoming, unreleased features.

Now that you have your environment ready for use, it is time to start discovering the amazing features offered by Visual Studio Code. The next chapter walks through the environment, then in Chapter 3, "Language Support and Code Editing Features," you will see all the amazing code editing features that make Visual Studio Code a rich, powerful cross-platform editor.

CHAPTER 2

Getting to Know the Environment

Before you use Visual Studio Code as the editor of your choice, it is convenient for you to know how the workspace is organized and what commands and tools are available, in order to get the most out of the development environment.

The VS Code's user interface and layout are optimized to maximize the space for code editing, and it also provides easy shortcuts to quickly access all the additional tools you need in a given context. More specifically, the user interface is divided into five areas: the code editor, the status bar, the activity bar, panels area, and the side bar. This chapter explains how the user interface is composed and how you can be productive getting the most of it.

Note All the features discussed in this chapter apply to any file in any language, and they will be available regardless of the language you see in the figures (normally C#). You can open one or more code files via **File** ➤ **Open File** to get some editor windows active and understand the features discussed in this chapter. Then in Chapter 4, “Working with Files and Folders,” I will discuss more thoroughly how you can work with individual files and multiple files, in one or more languages concurrently.

The Welcome Page

At startup, Visual Studio Code shows the Welcome page, visible in Figure 2-1.

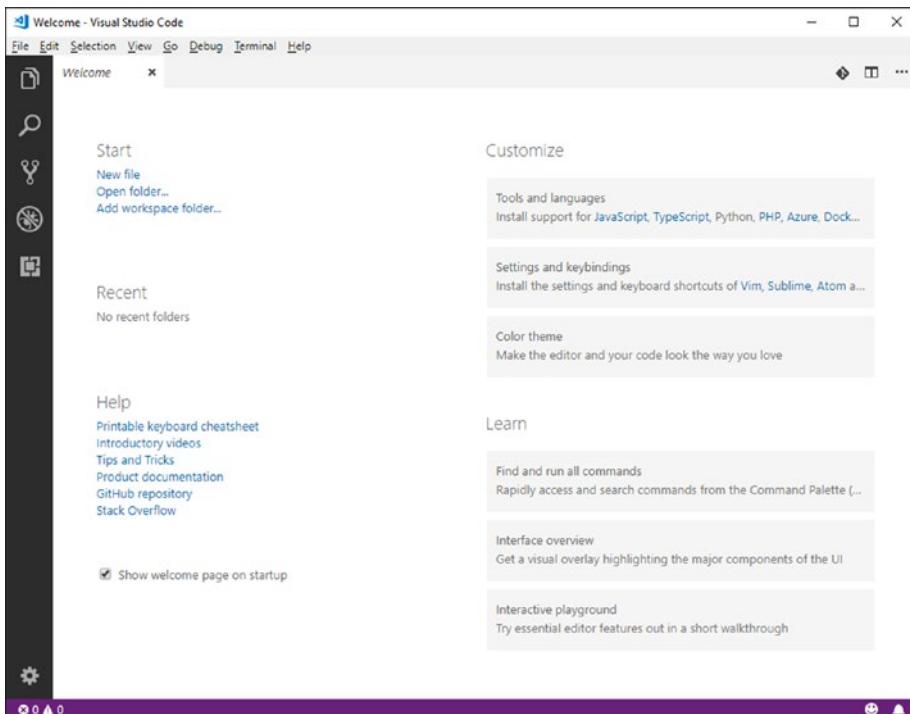


Figure 2-1. The Welcome page

On the left side of the page, under the **Start** group, you will find shortcuts for creating and opening files and folders. Under the **Recent** group, you will find a list of recently opened files and folders that you will be able to click for fast opening. Under the **Help** group, there are useful links to cheat sheets, introductory videos, product documentation, and other learning resources about Visual Studio Code. On the right side, under the **Customize** group, you can find shortcuts to customize Visual Studio Code by installing extensions, changing keyboard shortcuts and

color themes. Under the **Learn** group, you will find additional shortcuts to learning resources about commands and the user interface. Most of the features highlighted in the Welcome page will be discussed across this book. By default, the Welcome page is set to show up every time you launch Code. You can remove the flag from the **Show welcome page on startup** checkbox to change this behavior.

The Code Editor

The code editor is certainly the area where you spend most of your time in VS Code. The code editor becomes available when you create a new file or open existing files and folders. You can edit one file at a time as well as multiple files side-by-side concurrently. Figure 2-2 shows an example.

```

MainPage.xaml.cs - Visual Studio Code
File Edit Selection View Go Debug Tasks Help
Welcome    webapp.md    WebApplication1.xml    ...
1 # WebApplication1 #
2
3 ##### Method
4 Areas.HelpPage.ApiDescriptionExtensions.GetFriendlyId
5 (System.Web.Http.Description.ApiDescription)
6 Generates an URI-friendly
7 ID for the [[]
8 T:System.Web.Http.
9 Description.
10 ApiDescription]]. E.g.
11 "Get-Values-{id_name}"
12 instead of "GetValues/{id}
13 ?name={name}"
14
15 |Name | Description |
16 |----|-----|
17 |description| The [[]
18 T:System.Web.Http.
19 Description.ApiDescription]
20 ].|
21 **Returns**: The ID as a
22 string.
23
24 ===
25 ## Type
26 Areas.HelpPage.HelpPageConfig
27
28 Use this class to
29
30
31
32
33
34
<?xml version="1.0"?>
<doc>
<assembly>
<name>WebApplication1</name>
</assembly>
<members>
<member name="M:WebApplication1.<summary>Generates an UR</summary></member>
<member name="T:WebApplication1.<summary>Use this class</summary></member>
<member name="T:WebApplication1.<summary>The controller</summary></member>
<member name="M:WebApplication1.<summary>Sets the docume</summary></member>
<member name="M:WebApplication1.<summary>Sets the docume</summary></member>
<member name="M:WebApplication1.<summary>Sets the docume</summary></member>

```

```

MainPage.xaml.cs
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6 using Xamarin.Forms;
7
8 namespace App1
9 {
10     public partial class MainPage : ContentPage
11     {
12         public MainPage()
13         {
14             InitializeComponent();
15             this.Label1.Fo...
16         }
17     }
18 }
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34

```

Figure 2-2. The code editor and multiple file views

To do this, you have a couple options:

- Right-click a file name in the Explorer bar and then select **Open to Side**.
- Ctrl-click a file name in the Explorer bar.
- **Ctrl+\`\` (or `⌘+\\ on macOS)`** to split the editor in two.

Notice that if you already have three files open and you want to open another file, the editor that is active will display that file. You can quickly switch between editors by pressing **Ctrl + 1, 2, and 3**. The code editor is the heart of Visual Studio Code and provides tons of powerful productivity features that will be deeply discussed in the next chapter. For now, it is enough to know how to open and arrange editor windows.

Reordering, Resizing, and Zooming Editor Windows

Editor windows can be reordered and resized based on your preferences. Reordering editors can be done by clicking the editor's header (which is where you see the file name) and moving it to a different position. Resizing an editor can instead be accomplished by clicking the mouse left button over the editor's border, when the pointer appears as a left/right arrow pair.

You can also zoom in and out the active editor by clicking **Ctrl++** and **Ctrl+-**, respectively. As an alternative, you can select **View ▶ Zoom in** and **View ▶ Zoom out**.

Note In Visual Studio Code, the zoom is actually an accessibility feature. As an implication, when you zoom the code editor, the activity bar and side bar will also be zoomed.

The Status Bar

The status bar contains information about the current file or folder and provides shortcuts for some quick actions. Figure 2-3 shows an example of how the status bar appears.



Figure 2-3. The status bar

The status bar contains the following information, from left to right:

- Git version control information and options, such as the current branch.
- Errors and warnings detected in the source code.
- The cursor position expressed in line and column.
- Indentation information, in this case **Spaces: 4**. You can click this to change the indentation size and to convert indentation to tabs or spaces.
- The encoding of the current file.
- The current line terminator.
- The language for the open file. By clicking the current language name, you will be able to change the language from a dropdown list that will pop up.
- The project name, if you open a folder that contains a supported project system. It is worth noting that, in case the folder contains multiple project files, clicking this item will allow switching between projects.

- The feedback button, which allows sharing your feedback about Visual Studio Code on Twitter.
- The notification icon, which shows the number of new notifications (if any). Notification messages typically come from extensions or they are about product updates.

It is worth mentioning that the status bar color changes depending on the situation. For example, it is violet when you open a single file, blue when you open a folder, and orange when Visual Studio Code is in debugging mode.

The Activity Bar

The Activity bar is at the left side of the workspace and can be considered a collapsed container for the side bar. Figure 2-4 shows the Activity bar.



Figure 2-4. The Activity bar

The Activity bar provides shortcuts for the Explorer, Search, Git, Debug, Extensions, and Settings tools, each described in the next section. When you click a shortcut, the side bar related to the selected tool becomes visible. You can click again the same shortcut to collapse again the side bar.

The Side Bar

The Side bar is one of the most important tools in Visual Studio Code, certainly the tool you will interact more with together with the code editor. It is made of five tools, each enabled by the corresponding icon, described in the next subsections.

The Explorer Bar

The Explorer bar is enabled by clicking the first icon from the top and provides a structured, organized view of the folder or files you are working with. The **OPEN EDITORS** subview contains the list of active files, including open files that are not part of a project or folder or files that have been modified. These are instead shown in a subview whose name is the folder or project name. Figure 2-5 provides an example of Explorer.

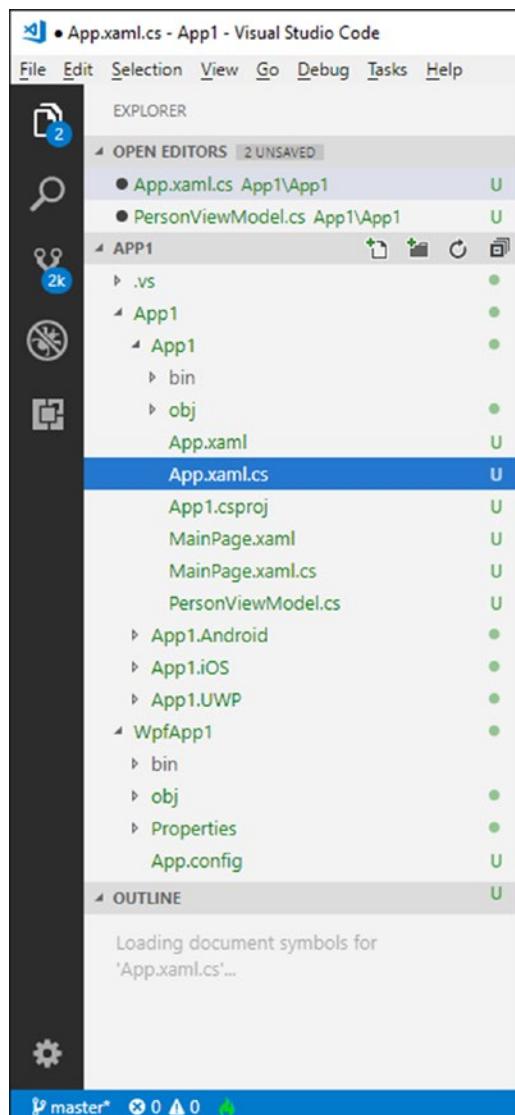


Figure 2-5. The Explorer bar

Note You must hover over the folder name (**APP1** in Figure 2-5) in order to get the four buttons visible.

The subview that shows a folder structure provides four buttons (from left to right): **New File**, **New Folder**, **Refresh**, and **Collapse All**, each self-explanatory. The **OPEN EDITORS** subview has instead three buttons (which you get when hovering over with the mouse): **Toggle Vertical/Horizontal Editor Layout**, **Save All**, and **Close All Files**. Right-clicking a folder or file name in Explorer provides a context menu that offers common commands (such as **Open to Side** you saw at the beginning of this chapter). A very interesting command is **Reveal to Explorer** (or **Reveal to Finder** on Mac and **Open Containing Folder** on Linux), which opens the containing folder for the selected item. Notice that the Explorer icon in the Activity bar also reports the number of modified files.

The Outline View

The bottom of the Explorer bar contains another group called **OUTLINE**. This group provides a hierarchical view of types and members defined within a code file or of tags within defined in a markup file. Figures 2-6 and 2-7 show the OUTLINE based on a TypeScript file and on a HTML file, respectively.

CHAPTER 2 GETTING TO KNOW THE ENVIRONMENT

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

- Title Bar:** Untitled-1 - Visual Studio Code
- File Menu:** File Edit Selection View Go Debug Tasks Help
- Explorer View:** OPEN EDITORS (1 UNSAVED), Welcome, TS Untitled-1, NO FOLDER OPENED. A message says "You have not yet opened a folder." with a "Open Folder" button.
- Outline View:** Shows the structure of the TypeScript code:
 - Student class:
 - firstName
 - fullName
 - lastName
 - middleInitial
 - constructor
 - Person interface:
 - firstName
 - lastName
 - greeter function
 - user variable
- Code Editor:** Displays the following TypeScript code:

```
1 class Student {  
2   fullName: string;  
3   constructor(firstName: string, middleInitial: string) {  
4     this.fullName = firstName + " " + middleInitial + " "  
5   }  
6 }  
7  
8 interface Person {  
9   firstName: string;  
10  lastName: string;  
11 }  
12  
13 function greeter(person: Person) {  
14   return "Hello, " + person.firstName + " " + person.lastName  
15 }  
16  
17 let user = new Student("Jane", "M.", "User");  
18  
19 document.body.innerHTML = greeter(user);
```
- Status Bar:** Ln 17, Col 27 Spaces: 4 UTF-8 CRLF TypeScript 3.0.1

Figure 2-6. The Outline view on a TypeScript file

CHAPTER 2 GETTING TO KNOW THE ENVIRONMENT

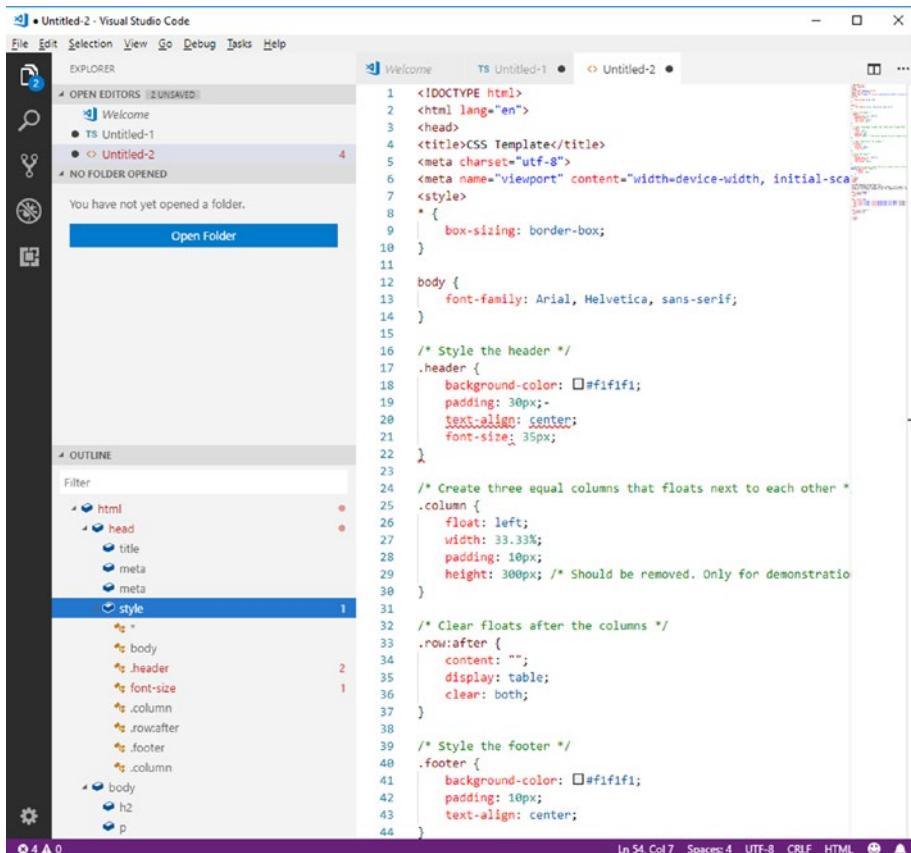


Figure 2-7. The Outline view on a HTML file

You can expand types and members to see what other objects they define, and you can click each item and get the cursor over the selected item definition in the source code. Also, you can type in the **Filter** text box to restrict the list of items based on a search criterion. It is worth mentioning that Visual Studio Code highlights with a different color (red in the case of the Visual Studio Light Theme) items that have potential problems and that are highlighted with squiggles in the code editor.

The Search Tool

The Search tool, enabled with the search icon, allows for searching and optionally replacing text across files. You can search for one or more words, including special characters (such as * and ?), and you can even search based on regular expressions. Figure 2-8 shows the Search tool in action, with advanced options expanded (files to include and files to exclude).

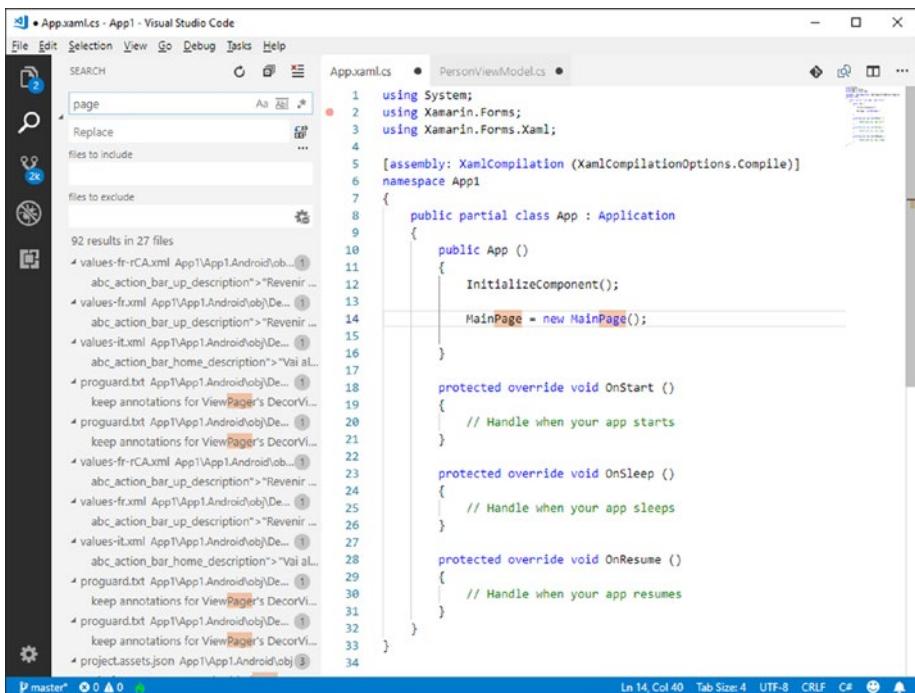


Figure 2-8. The Search tool

Search results are presented in a hierarchical view that groups all the files that contain the specified search key, showing an excerpt of the line of code that contains it. Occurrences are also highlighted in both the list of files and in the code editor. You can finally clean up search results by clicking the **Clear Search Results** button. If you instead wish to replace

some text with a new text, you can do this by entering the new text into the **Replace** text box and then by clicking the **Replace All** button.

The Git Bar

The Side bar provides access to Git integration for version control. Git integration is a core topic and will be thoroughly discussed in Chapter 7, “Source Control with Git,” but a quick look is provided here for the sake of completeness about the Side bar.

The Git bar can be enabled by clicking the third button from the top (with a kind of fork icon) and provides access to all of the common source control operations, such as initializing a repository, committing code files, and synchronizing branches. The Git icon also shows the number of files that have been modified locally. Figure 2-9 shows an example.

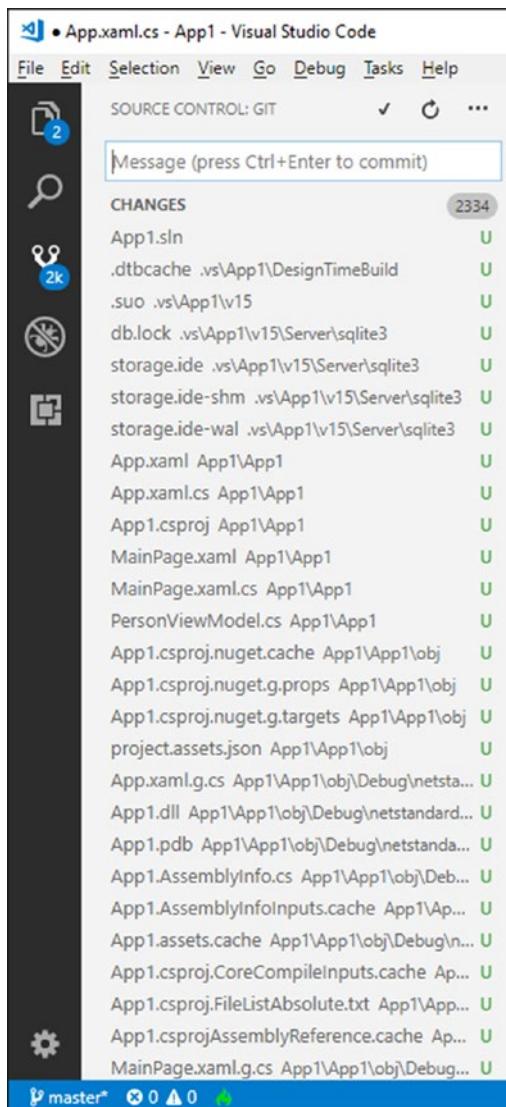


Figure 2-9. The Git bar

The Git bar also provides a popup menu that you can see by clicking the ... button at the top-right corner of the bar and that contains the list of supported Git commands in Visual Studio Code. As I said before, Git integration will be described later in the book.

The Debug Bar

Visual Studio Code is not a simple code editor, but it is also a fully featured development tool that ships with an integrated debugger for .NET Core and that can be extended with third-party debuggers for other platforms and languages. Chapter 9, "Running and Debugging Code," describes in more detail such an important part of Visual Studio Code, but for now you have to know that the debugging tools can be accessed by clicking the fourth icon from the top. This will open the Debug bar, shown in Figure 2-10.

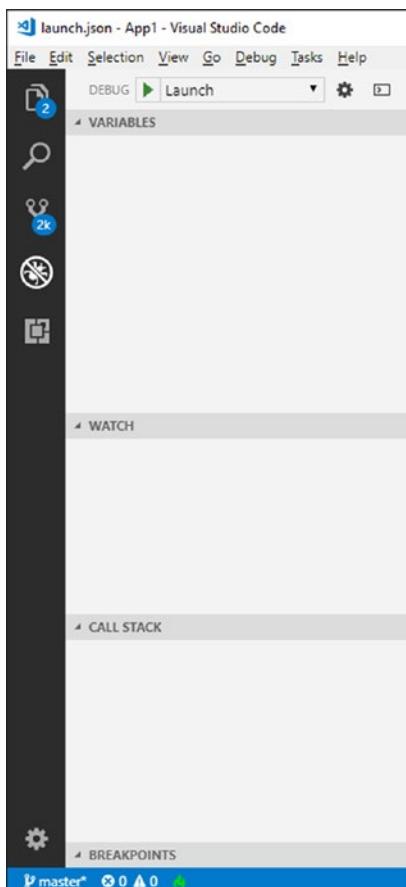


Figure 2-10. The Debug bar

In Chapter 9 you will see how powerful the debugging tools are in Visual Studio Code and how easy is installing additional debuggers.

The Extensions Bar

The Extensions bar can be enabled by clicking the fifth button from the top in the Activity bar and allows for searching and installing extensions for Visual Studio Code, which include additional languages, debuggers, code snippets, and much more. Extensibility will be discussed in Chapter 6, "Installing and Managing Extensions," but Figure 2-11 provides an example of how the Extensions bar appears.

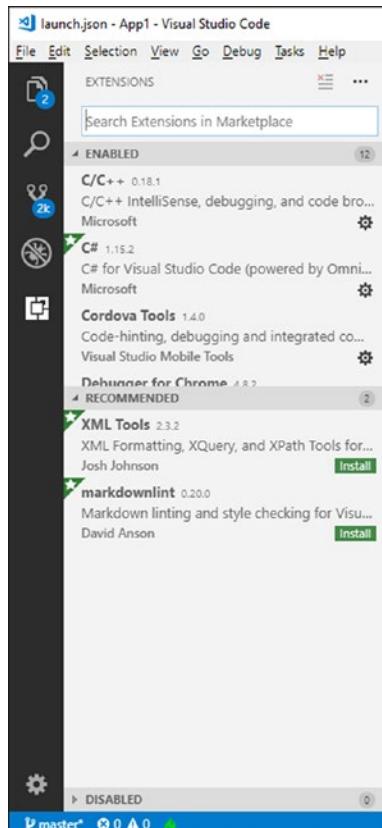


Figure 2-11. The Extensions bar

Not only you can search online for extensions, but you can see the list of installed extensions as well as disabled and recommended extensions.

The Settings Button

The Settings button is represented with the gear icon, at the bottom of the Activity bar. If you click it, you will see a popup menu with a list of commands that represent shortcuts for customizing Visual Studio Code and that will be discussed more thoroughly in Chapter 5, "Customizing Visual Studio Code." Among the others, a command in the menu allows for manually searching for product updates.

Navigating Between Files

Visual Studio Code provides two ways of navigating between files. The quickest way is pressing Alt+Left or Alt+Right to switch between active files.

If you instead press Ctrl+Tab, you will be able to browse the list of files that were opened since VS Code was launched, and you will be able to select one for editing, as shown in Figure 2-12.

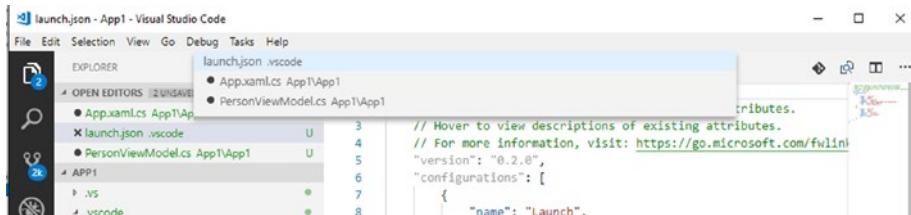


Figure 2-12. Navigating between active files

The Command Palette

Together with the code editor and the activity and side bars, the Command Palette is another very important tool in Visual Studio Code, which allows for accessing Visual Studio Code built-in commands and also commands added by extensions via the keyboard. The Command Palette can be opened with View ► Command Palette or via the Ctrl+Shift+P keyboard shortcut (⌘+P on macOS), and Figure 2-13 shows how it looks like.

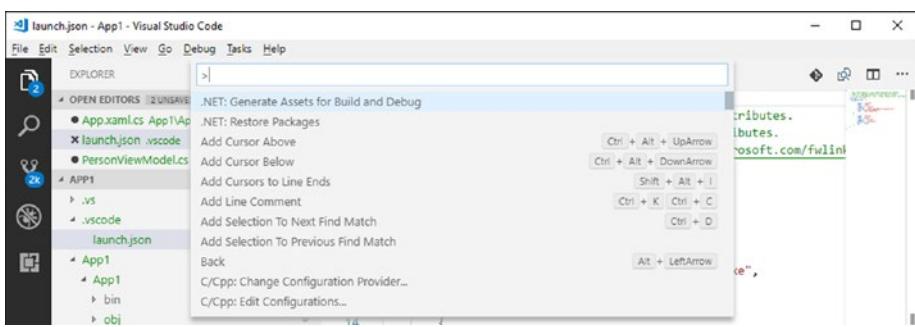


Figure 2-13. The Command Palette

The Command Palette is not just about menu commands or to user interface instrumentation but also to other actions that are not accessible elsewhere. For instance, the Command Palette allows installing extensions as well as restoring NuGet packages over the current project or folder. You can simply move up and down to see the full list of available commands, and you can type in some characters to filter the list. You will notice how many of them map actions available within menus and that, for many of them, there is a keyboard shortcut available. Other commands, such as extension, debug, and Git commands, will be discussed in the next chapters, so it is important that you get started with the Command Palette at this point.

The Panels Area

Visual Studio Code will very often need to display information about source code but also information coming from the Git engine, external tools, or debuggers. To accomplish this in an organized way, the environment provides the so-called Panels area, which appears by default at the bottom of the user interface.

The Panels area is made of four built-in panels: Problems, Output, Debug Console, and Terminal, each discussed in this section. The Panels area is not visible by default, and it usually pops up when the information they represent becomes available (such as the debugger sending information about symbols in the source code). Additionally, by default it appears at the bottom of the VS Code's user interface, but you can move it to the side of the workspace with a button called **Move to Right** that each panel provides, and then you can restore the original layout with another button called **Move to Bottom**. Let's now discuss each panel in more detail.

The Problems Panel

With languages that have built-in enhanced editing support, such as TypeScript (www.typescriptlang.org), or for which an extension has been added to provide advanced editing features, such as C#, Visual Studio Code can detect code issues as you type. In the code editor, these are usually highlighted with red squiggles (for blocking errors) and in green (for warnings). The list of errors, warnings, and informational messages is also displayed in the Problems panel. This can be enabled by clicking the number of errors at the bottom-left corner of the status bar (see Figure 2-11).

The Problems panel makes it easy to distinguish between errors and warnings due to different icons (a white x over red background for errors and a black exclamation mark over yellow background for warnings). Figure 2-14 shows an example based on some C# code that contains an unused variable (warning) and a syntax error.

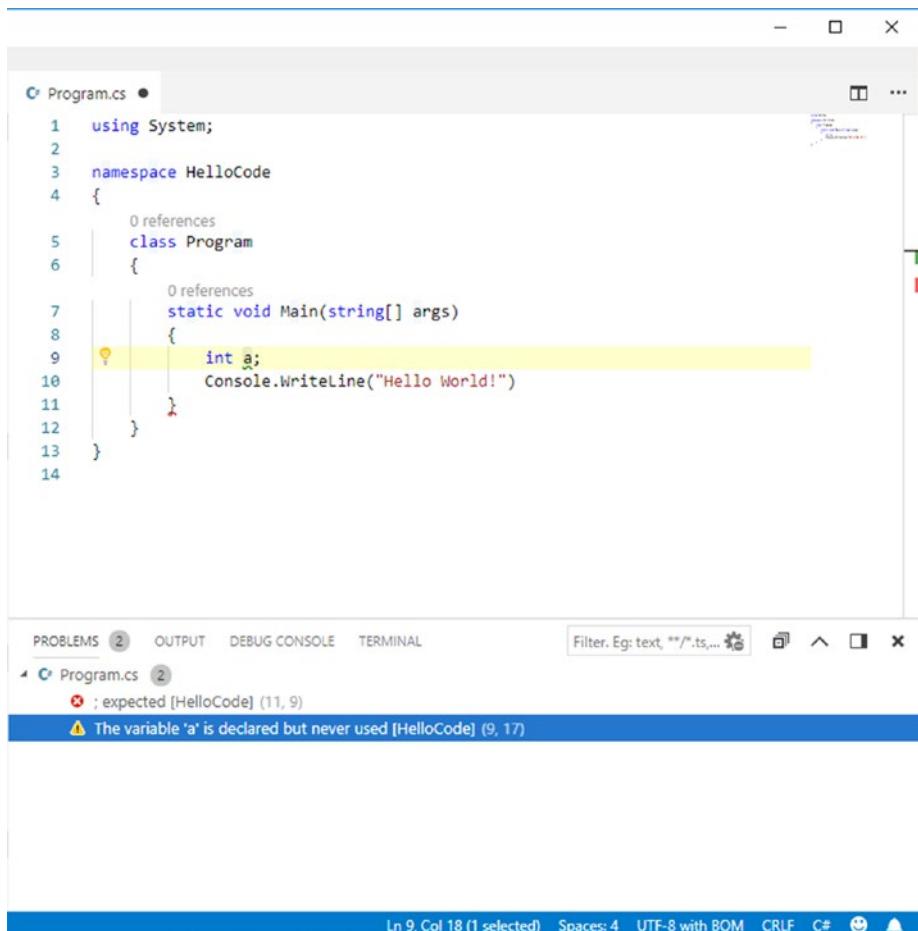


Figure 2-14. The Problems panel

In case you have multiple files opened, the Problems panel will group problems by file name. Also, for each problem, you will be able to see the folder name and the position within the source code file. Just double-click a problem, and VS Code will move the cursor to the selected item in the code editor.

Note The code editor also provides a way to quickly fix code issues while typing, but this is not related to the Problems panel and will instead be discussed in the next chapter.

The Output Panel

The Output panel is the place where Visual Studio Code displays messages from internal and external tools, such as runtime tools, Git commands, extensions, and tasks. Figure 2-15 shows an example based on the output of .NET's NuGet package manager.

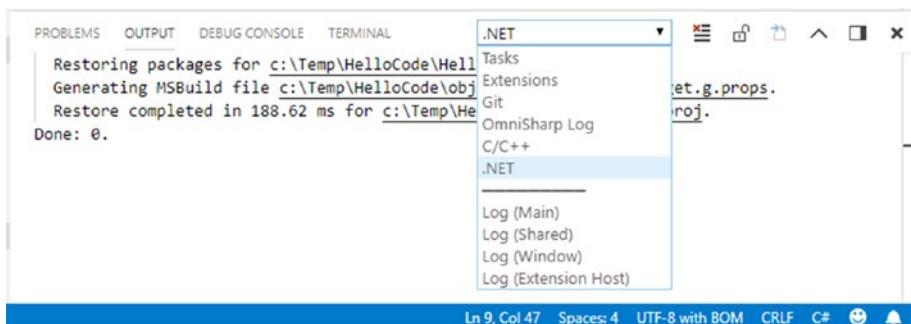


Figure 2-15. The Output panel

Because multiple tools might run concurrently during an operation against source code files (e.g., package restore and then compilation) or during the Visual Studio Code lifetime (such as extensions), you can use the dropdown box in the panel to change the view and see the output of each tool. This tool is particularly useful if the execution of external tools fails and you want to get more information about what happened.

The Debug Console Panel

As the name implies, the Debug Console panel is a specialized panel used by debuggers to display information about code execution. Figure 2-16 shows an example based on the execution of a simple C# application.

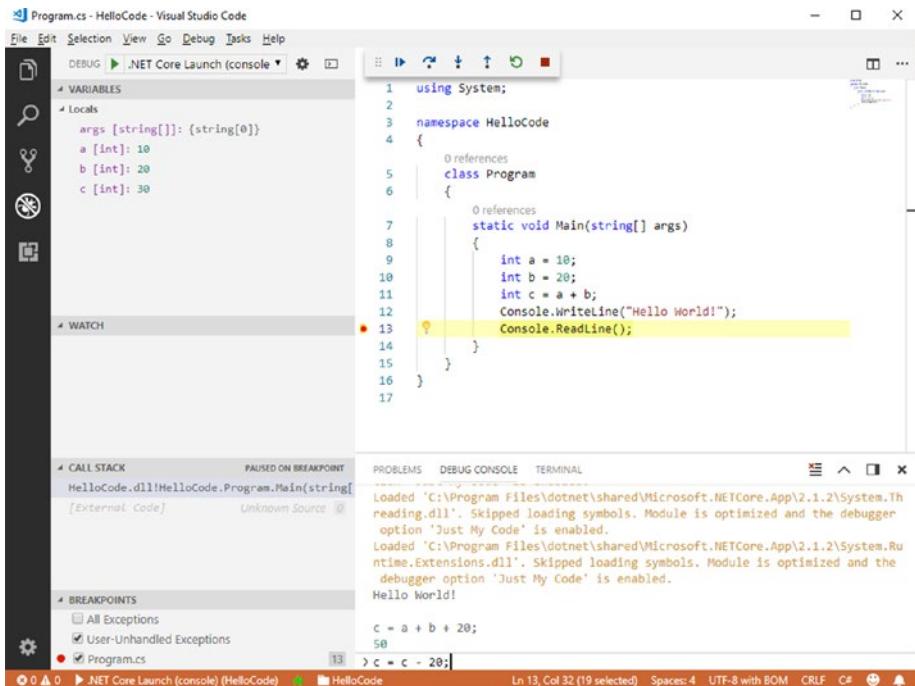


Figure 2-16. The Debug Console panel

Not only the Debug Console shows information about code execution, debug symbols, and any other information a debugger needs to display, but it also acts as an interactive console where you can evaluate expressions. If you take a look at Figure 2-16, you can see how a mathematical expression has been manually evaluated using variables defined in the code. Debugging is a very important topic in Visual Studio Code and is thoroughly discussed in Chapter 9, "Running and Debugging Code," where you will find additional information about the Debug Console.

Working with the Terminal

Visual Studio Code allows executing commands against the operating system directly from within the development environment. In fact, you can select the **Terminal > New Terminal** command to open a new terminal instance in a panel at the bottom of the work area. Figure 2-17 shows an example based on Windows.



Figure 2-17. The Terminal panel

On macOS and Linux, the terminal tool is based on the bash shell of each system. On Windows, the terminal is based on PowerShell by default. However, when you open a terminal instance, a popup message will tell you that you can select a different tool by clicking the **Customize** button on the popup itself. At this point you will be able to select, from the Command Palette, one among the Windows command prompt, PowerShell, and the Git bash command line tool. You can also open multiple terminal instances by clicking the **New Terminal** button (the icon with the + symbol). The Terminal panel is also used by Visual Studio Code to launch automatic scripts and commands against the operating system. For example, when you build a C# application, Visual Studio Code starts the .NET Core compiler whose output is displayed in the Terminal panel, as shown in Figure 2-18.

The screenshot shows the Visual Studio Code interface with the Terminal tab selected. The terminal window displays the output of a build command:

```
> Executing task: C:\Program Files\dotnet\dotnet.exe build C:\Temp\HelloCode\HelloCode.csproj <  
Microsoft (R) Build Engine version 15.8.166+gd4e8d81a88 for .NET Core  
Copyright (C) Microsoft Corporation. All rights reserved.  
  
Restore completed in 35.66 ms for C:\Temp\HelloCode\HelloCode.csproj.  
HelloCode -> C:\Temp\HelloCode\bin\Debug\netcoreapp2.1\HelloCode.dll  
  
Build succeeded.  
0 Warning(s)  
0 Error(s)
```

The status bar at the bottom indicates: Ln 13, Col 1 | Spaces: 4 | UTF-8 with BOM | CRLF | C# | 🎉 | 📡.

Figure 2-18. The Terminal panel used for automatic scripting

Summary

In this chapter, you got an overview of the workspace in Visual Studio Code and of the tools you will interact more with. You saw how to take advantage of quick shortcuts in the Welcome page and how you can arrange editor windows.

You saw how the status bar provides information about the active file and how the Activity bar is a collapsed container of shortcuts for the tools contained in the Side bar: the Explorer bar, the Search tool, the Git bar, the Debug bar, the Extensions bar, and the Settings button. You saw how to quickly navigate between files and how the Command Palette provides a way for accessing commands via the keyboard, both Visual Studio Code commands and extensions' commands. You have also walked through another important area in the environment, the Panels area, where you can get information about code issues, messages from internal and external tools and debuggers, and where you can execute commands and scripts via the Terminal. Now that you have seen how the environment is organized, it is time to get fun walking through all the powerful productivity features in the code editor. This is the topic of the next chapter.

CHAPTER 3

Language Support and Code Editing Features

Visual Studio Code is not just another evolved text editor with syntax colorization and automatic indentation. Instead, it is a very powerful code-focused development environment expressly designed to make it easier to write web, mobile, and cloud applications using languages that are available to different development platforms.

With the ambition to provide a powerful, rich development environment, Visual Studio Code integrates a number of editing features that are focused on improving the productivity and quality of your code. This chapter discusses what languages are supported in Visual Studio Code and all the available code editing features, starting from the most basic that are available to all the supported languages to the most advanced productivity tools that are available to specific languages such as C# and TypeScript.

Note Keyboard shortcuts used in this chapter are based on the default settings in Visual Studio Code.

Language Support

Out of the box, Visual Studio Code has built-in support for many languages. Table 3-1 groups supported languages by editing features.

Table 3-1. Language Support by Feature

| Languages | Editing Features |
|--|---|
| Batch, C, C#, C++, Clojure, CoffeeScript, Diff, Dockerfile, F#, Go, HLSL, Jade, Java, HandleBars, Ini, Lua, Makefile, Objective-C, Objective-C++, Perl, PowerShell, Properties, Pug, Python, R, Razor, Ruby, Rust, SCSS, ShaderLab, Shell Script, SQL, Visual Basic, XML | Common features (syntax coloring, bracket matching, basic word completion) |
| Groovy, Markdown, PHP, Swift | Common features and code snippets |
| CSS, HTML, JSON, JSON with Comments, Less, Sass | Common features, code snippets, IntelliSense, Outline |
| TypeScript, TypeScript React, JavaScript, JavaScript React | Common features, code snippets, IntelliSense, Outline, parameter hints, refactoring, Find All References, Go To Definition, Peek Definition |

Visual Studio Code can be extended with additional languages produced by the developer community and that can be downloaded from the Visual Studio Marketplace. This is discussed in more detail in Chapter 6, “Installing and Managing Extensions,” but, in the meantime, you can have a look at the available languages.

Working with C# and C++

The C# programming language deserves a more detailed version, because of its popularity and because it is now a cross-platform language that you can use not only on Windows but also on macOS and Linux. As you can see from Table 3-1, the editing experience that Visual Studio Code offers out of the box for C# is limited to common features.

However, full and rich support for the coding experience with C# is offered via the **Microsoft C#** free extension (<https://marketplace.visualstudio.com/items?itemName=ms-vscode.csharp>). This provides an optimized experience for .NET Core development and includes all the support and tools you need to build apps with C#, including the necessary support for the .NET Core debugger. With this extension, you will basically get the same experience available to TypeScript, including advanced editing capabilities based on the .NET Compiler Platform (also known as Roslyn) that makes it easier to fix code issues as you type. If you plan to work with C#, I definitely recommend you to install this extension, especially because this chapter discusses some editing features that are available only through the extension. Extensibility is explained in more detail in Chapter 6, "Installing and Managing Extensions," so the easiest way to get the extension installed without further information is opening any C# code file (.cs) and following the instructions shown by Visual Studio Code when it detects that a proper extension is available for that file type.

Similarly, you might want to install the Microsoft C/C++ extension that adds enhanced editing features to the C and C++ languages, plus debugging support for Windows (PDB, MinGW, Cygwin), macOS, and Linux. The extension is available at <https://marketplace.visualstudio.com/items?itemName=ms-vscode.cpptools>, and you can follow the same steps described before about the C# extension for easy installation, of course opening a .c, .h, or .cpp file.

Basic Code Editing Features

Visual Studio Code provides many of the features you would expect from a powerful code editor. This section describes what editing features make your coding experience amazing with this new tool. If you are familiar with Microsoft Visual Studio 2017, you will also see how some features have been inherited from this IDE. It is worth mentioning that Visual Studio Code provides keyboard shortcuts for almost all the editing features, giving you an option to edit code faster. For this reason, the keyboard shortcut is also mentioned for many of the described features.

Note Features described in this section apply to all the supported languages described in Table 3-1, except where expressly specified.

Working with Text

As you would expect, the code editor in VS Code offers commands for text manipulation and text selection. The **Edit** menu provides the **Undo**, **Redo**, **Copy**, **Cut**, **Paste**, **Find**, **Replace**, **Find in Files**, and **Replace in Files** commands. These commands are available in every text editor and do not require any further explanation, except for the find and replace tools that were described in the previous chapter.

The **Edit** menu also includes the **Toggle Line Comment** and **Toggle Block Comment** commands, which add a single line comment or a block comment, respectively, depending on the language. For instance, in C# the first command would comment a line like this:

```
// int a = 0;
```

Instead, the block comment tool would add a multiline comment as follows:

```
/* */ int a = 0;
```

The **Edit** menu also provides two commands to work with code snippets, **Emmet: Expand Abbreviation** and **Emmet...**. The first command is the menu representation of keyboard shortcuts offered by code editor to add a code snippet, whereas the second command opens the Command Palette and shows the list of code snippets that you can add. Code snippets are discussed in more detail in the “Reusable Code Snippets” section in this chapter.

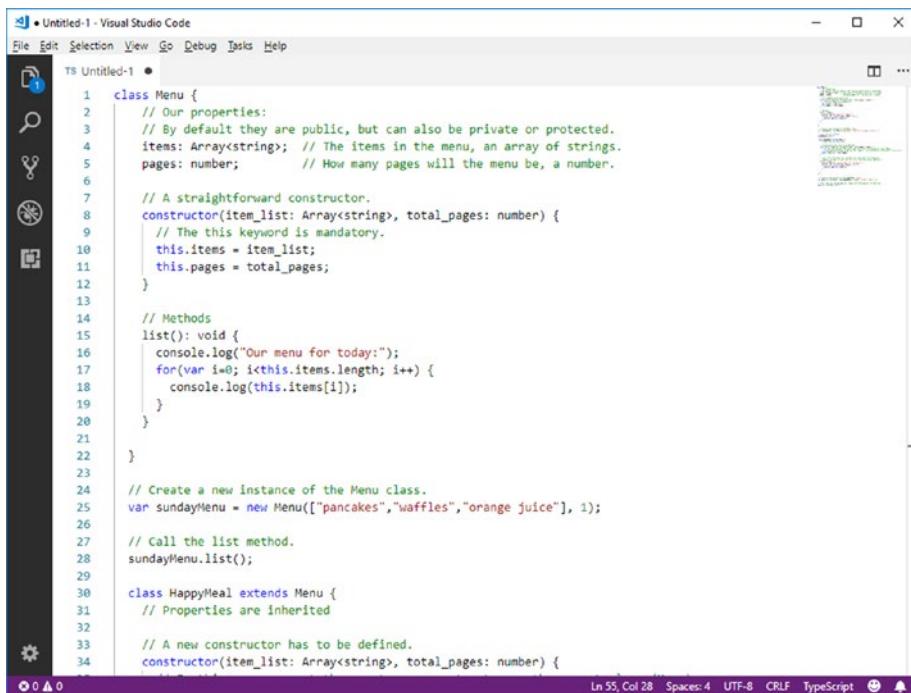
The **Selection** menu not only provides commands for text selection but also commands that make it easier to move or duplicate lines of code above and below the current line. The **Add Cursor Above**, **Add Cursor Below**, and **Add Cursors To Line Ends** commands allow working with multicursors, described in the “Multicursors” section in this chapter.

If you click an identifier, reserved word, or type name in the editor, you can use the **Add Next Occurrence**, **Add Previous Occurrence**, and **Select All Occurrences** commands that allow to quickly select occurrences of the selected word, and occurrences will be highlighted in a different color, which differs depending on the current theme.

Syntax Colorization

For all the languages summarized in Table 3-1, the code editor in Visual Studio Code provides the proper syntax colorization. Figure 3-1 shows an example based on a TypeScript code file.

CHAPTER 3 LANGUAGE SUPPORT AND CODE EDITING FEATURES



The screenshot shows a Visual Studio Code window with the title bar "Untitled-1 - Visual Studio Code". The menu bar includes File, Edit, Selection, View, Go, Debug, Tasks, Help. The main editor area contains the following TypeScript code:

```
1 class Menu {
2     // Our properties:
3     // By default they are public, but can also be private or protected.
4     items: Array<string>; // The items in the menu, an array of strings.
5     pages: number; // How many pages will the menu be, a number.
6
7     // A straightforward constructor.
8     constructor(item_list: Array<string>, total_pages: number) {
9         // The this keyword is mandatory.
10        this.items = item_list;
11        this.pages = total_pages;
12    }
13
14    // Methods
15    list(): void {
16        console.log("Our menu for today:");
17        for(var i=0; i<this.items.length; i++) {
18            console.log(this.items[i]);
19        }
20    }
21
22 }
23
24 // Create a new instance of the Menu class.
25 var sundayMenu = new Menu(["pancakes", "waffles", "orange juice"], 1);
26
27 // Call the list method.
28 sundayMenu.list();
29
30 class HappyMeal extends Menu {
31     // Properties are inherited
32
33     // A new constructor has to be defined.
34     constructor(item_list: Array<string>, total_pages: number) {
```

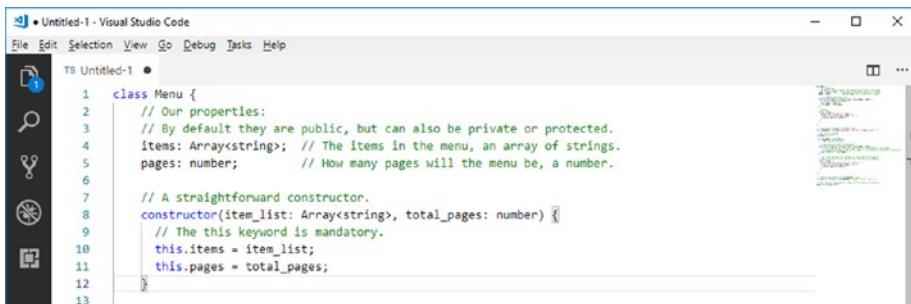
The status bar at the bottom shows "Ln 55, Col 28" and "Spaces: 4" and "UTF-8" and "CRLF" and "TypeScript". There are also icons for file operations and settings.

Figure 3-1. Syntax colorization

Syntax colorization is available for other languages via extensibility. If you need to work with a language that is not included out of the box, you can check the Visual Studio Marketplace and see if an extension is available to support such a language. See Chapter 6, "Installing and Managing Extensions," for information about extensibility. As a side note, syntax colorization is the minimum that an extension must provide to add support for a new language.

Delimiter Matching and Text Selection

The code editor can highlight matching delimiters such as brackets and parentheses (both square and round). This feature is extremely useful to delimit code blocks and is triggered once the cursor gets near one of the delimiters. Figure 3-2 shows an example based on bracket matching in a constructor definition.



The screenshot shows a code editor window titled "Untitled-1 - Visual Studio Code". The menu bar includes File, Edit, Selection, View, Go, Debug, Tasks, and Help. The code editor displays the following TypeScript code:

```

1 class Menu {
2     // Our properties:
3     // By default they are public, but can also be private or protected.
4     items: Array<string>; // The items in the menu, an array of strings.
5     pages: number; // How many pages will the menu be, a number.
6
7     // A straightforward constructor.
8     constructor(item_list: Array<string>, total_pages: number) {
9         // The this keyword is mandatory.
10        this.items = item_list;
11        this.pages = total_pages;
12    }
13

```

The cursor is positioned at the end of the first closing brace on line 12. The entire constructor definition from line 8 to line 12 is highlighted with a light blue selection, demonstrating the delimiter matching feature.

Figure 3-2. Delimiter matching

This feature is also very useful when you need to visually delimit nested blocks and with complex and long expressions. It is worth mentioning that you can press Ctrl+D to fast select a word or identifier at the right of the cursor and that you can also easily expand (Shift+Alt+Right) and shrink (Shift+Alt+Left) text selection within enclosing delimiters of a code block.

Code Block Folding

The code editor allows folding delimited code blocks. Just hover line numbers and the - symbol will appear near the start of a code block. Simply click to fold, and you will see the + symbol at this point, which you click to unfold the code block. Figure 3-3 provides an example.

CHAPTER 3 LANGUAGE SUPPORT AND CODE EDITING FEATURES

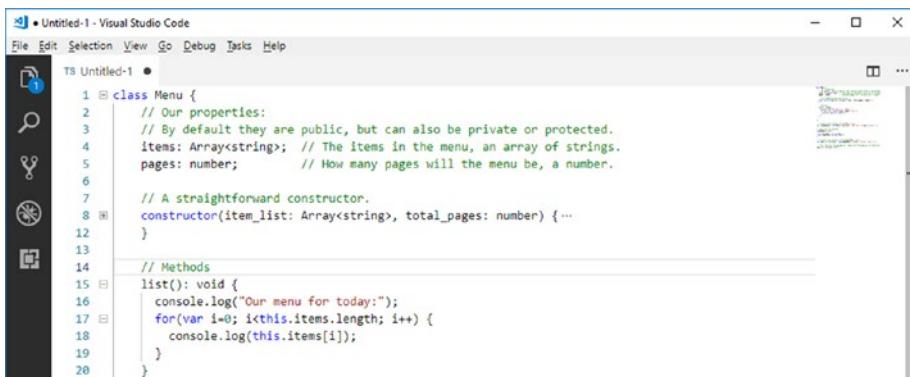
A screenshot of the Visual Studio Code interface. The title bar says "Untitled-1 - Visual Studio Code". The menu bar includes File, Edit, Selection, View, Go, Debug, Tasks, Help. The main editor area shows a TypeScript file named "TS Untitled-1". The code contains a class definition and a method. Lines 1 through 20 are visible, with some lines collapsed under fold indicators. The left sidebar has icons for file operations like Open, Save, Find, and others. A status bar at the bottom shows "VS Code 1.32.1".

Figure 3-3. Code block folding

Multicursors

The code editor supports multicursors. Each cursor operates independently, and you can add secondary cursors by pressing Alt-Click at the desired position. You will see that secondary cursors will be rendered thinner. The most typical situation in which you want to use multicursors is when you want to add (or replace) the same text in different positions of a code file.

Reusable Code Snippets

Visual Studio Code ships with a number of built-in code snippets that you can easily add by using the Emmet abbreviation syntax and pressing Tab. See the “Language Support” section to discover what languages support code snippets natively. For instance, in a Swift file, you can easily add a do...catch block definition by using the do code snippet, as shown in Figure 3-4.

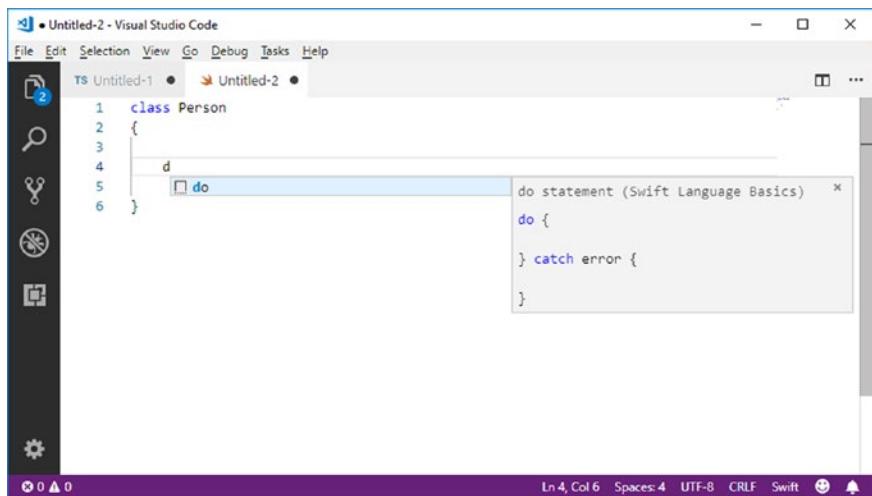


Figure 3-4. Adding code snippets

Code snippets are available as you type within the code editor, and you can recognize them by the icon representing a small, white sheet. Notice how a tooltip shows a preview of the code snippet. Pressing Tab over the previous snippet produces the result shown in Figure 3-5.

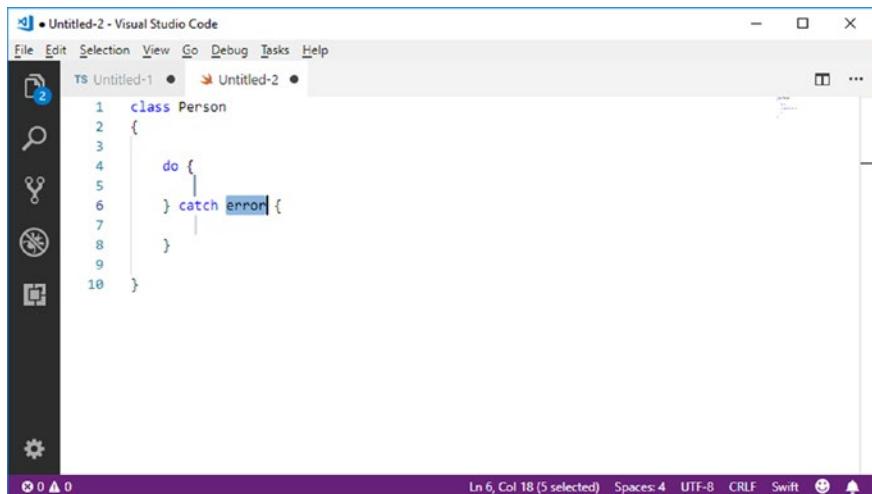


Figure 3-5. A newly added code snippet with a variable name highlighted

CHAPTER 3 LANGUAGE SUPPORT AND CODE EDITING FEATURES

Notice that if the code snippet contains variable names or identifiers, these might be highlighted suggesting that you should give them a different name (like for the error identifier in Figure 3-5). When you rename a highlighted identifier, all occurrences will be also renamed.

Visual Studio Code is not limited to built-in code snippets. You can download code snippets produced by other developers for many languages from the Visual Studio Marketplace. Actually, most of the extensions that introduce or extend support for programming languages also include a number of code snippets.

Word Completion

Out of the box, the code editor in Visual Studio Code implements basic word completion for all the supported languages. This feature helps you complete words and statements as you type. For example, if you look at Figure 3-6, you can see how the code editor suggests terminating a statement with the `Class` keyword in a Visual Basic file, based on what the developer is typing.

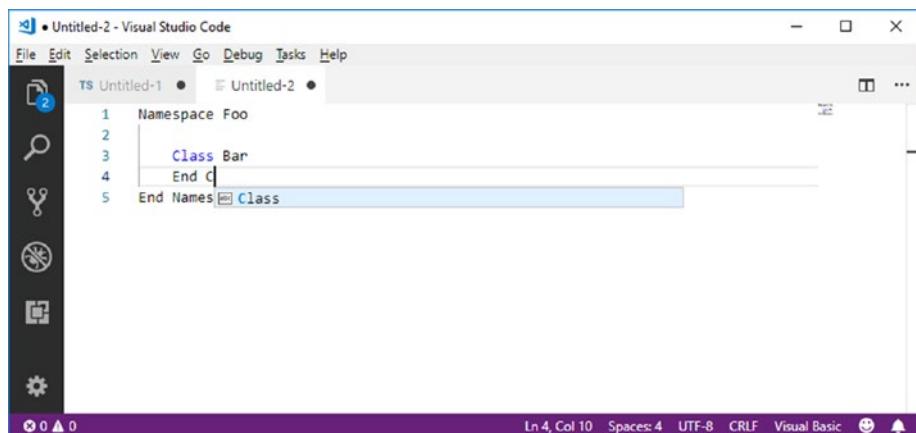


Figure 3-6. Completing a statement with word completion

Simply press Enter or Tab to insert the suggested word. The word completion engine learns as you code and can provide suggestions based on variables and member names you declare. For example, Figure 3-7 demonstrates how the editor suggests adding the name of a variable called Test, declared previously in the code.

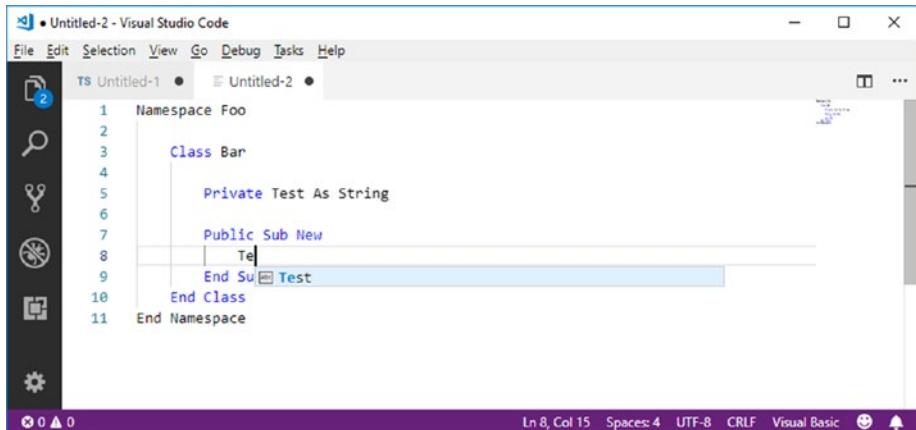


Figure 3-7. The code editor can suggest identifiers declared in the code

Minimap Mode

Sometimes it is difficult to have an idea of the position of the cursor inside a source code file, especially with very long files. Visual Studio Code provides the Minimap, a small preview of the source code file on the code editor's scrollbar. Figure 3-8 provides an example.

CHAPTER 3 LANGUAGE SUPPORT AND CODE EDITING FEATURES

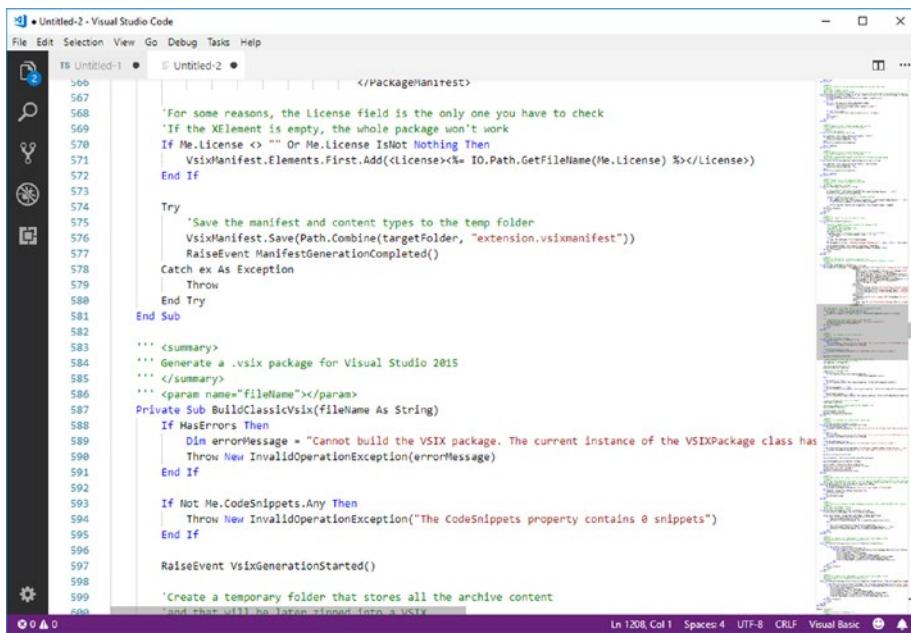
A screenshot of the Visual Studio Code interface. The main area shows a code editor with several lines of VB.NET code. The code is related to generating a VSIX package, including sections for saving manifest files and handling errors. The status bar at the bottom right indicates "Ln 1208, Col 1 Spaces: 4 UTF-8 CRLF Visual Basic". To the right of the code editor is a vertical "Minimap" window, which is a small preview of the entire document's structure and highlights the current visible region.

Figure 3-8. The Minimap allows for previewing source code on the scrollbar

If you click the Minimap, the portion of source code that is visible in the code editor is highlighted in the scrollbar, so that you can have a better perception of the current position of the cursors. The Minimap can be disabled and enabled using the **View > Toggle Minimap** command.

Whitespace Rendering and Breadcrumbs

A very common feature with text editors is the possibility of showing light dots instead of white spaces. In Visual Studio Code, this is possible for white spaces within indentations. To accomplish this, you select **View > Toggle Render Whitespace**. Figure 3-9 shows an example of how white spaces for indentations are replaced with dots.

CHAPTER 3 LANGUAGE SUPPORT AND CODE EDITING FEATURES

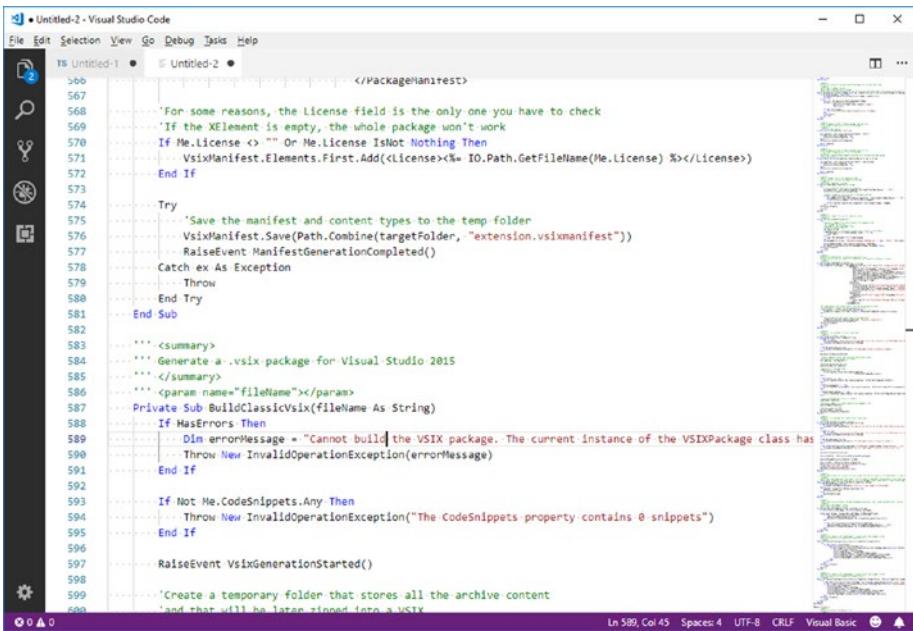


Figure 3-9. Rendering indentation spaces with dots

Simply use again the same command to return to white spaces. Another very useful command is **Toggle Breadcrumbs**, available in the **View** menu. With supported languages, such as JavaScript, TypeScript, and C# with the extension installed, it shows an icon at the upper left corner of the code editor, which you can expand to see the definition of types and members, as shown in Figure 3-10.

CHAPTER 3 LANGUAGE SUPPORT AND CODE EDITING FEATURES

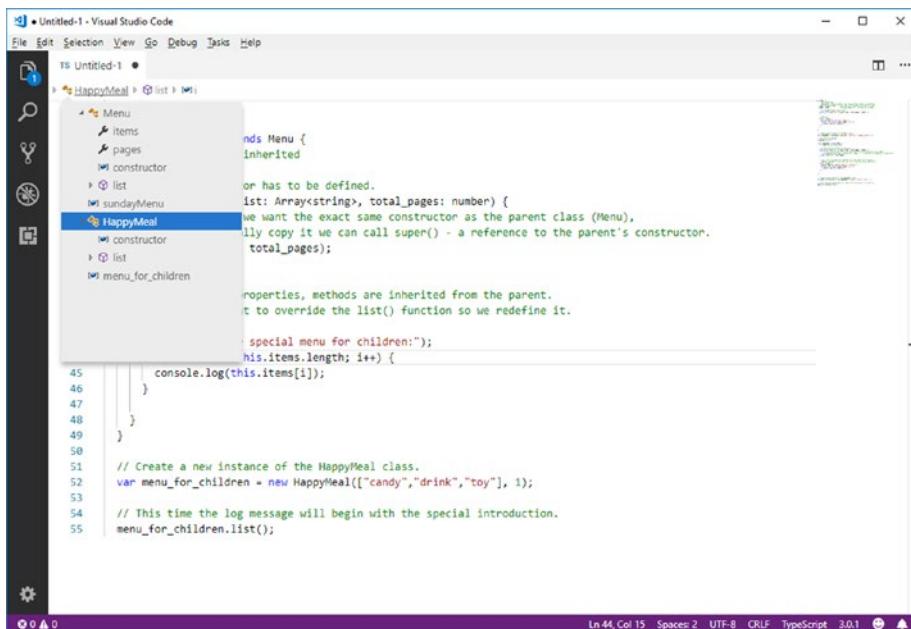


Figure 3-10. Navigating between types and members with breadcrumbs

By clicking a type or member name, the cursor will be moved to its definition, making code navigation much easier.

Markdown Preview

Visual Studio Code supports the Markdown syntax for producing documents in the very popular .md file format. Other than syntax colorization, for this particular language Visual Studio Code also provides a preview of how the document will look like. Simply press **Ctrl+Shift+V** (**Cmd+Shift+V** on macOS) in the code editor, and the preview will appear in a separate window, as demonstrated in Figure 3-11.

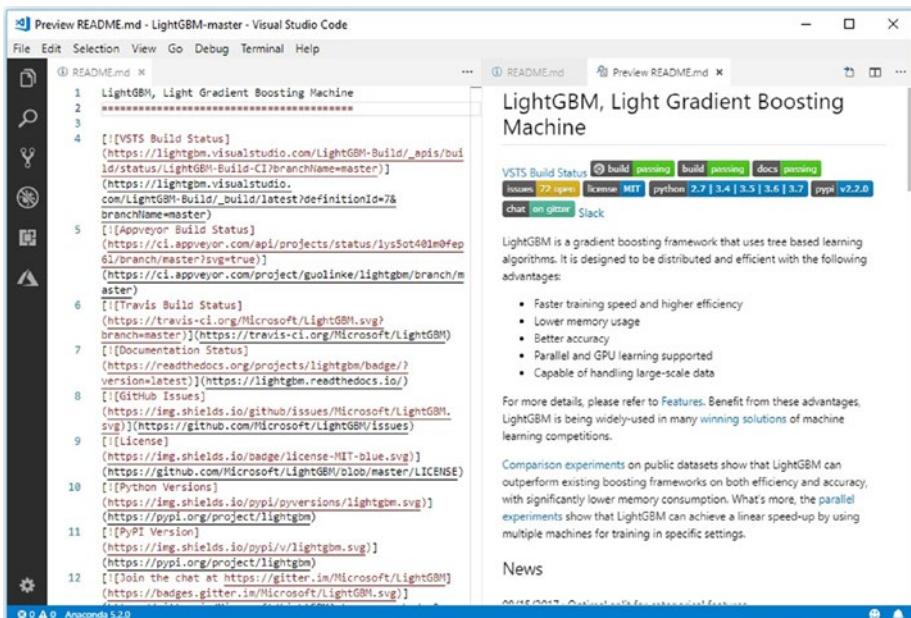


Figure 3-11. Integrated Markdown preview

This feature is very useful because it allows to preview your documents without the need of an external program such as a web browser.

Evolved Code Editing

Visual Studio Code is an extremely powerful code editing tool and brings to a cross-platform and multilanguage environment many features that have been available in Microsoft Visual Studio since many years, providing what is called *evolved code editing*. This section explains all the advanced code editing features that are available, out of the box, to languages such as TypeScript and JavaScript and to languages like C#, C++, and Python with the appropriate extensions installed.

Working with IntelliSense

IntelliSense provides rich, advanced word completion via a convenient popup that appears as you type. In the developer tools from Microsoft, such as Visual Studio, IntelliSense has always been one of the most popular features, and the reason is that it is not simply word completion. In fact, IntelliSense provides suggestions as you type, showing the documentation about a member (if available) and displaying an icon near each suggestion that describes what kind of syntax element a word represents. Figure 3-12 shows IntelliSense in action with a C# code file.

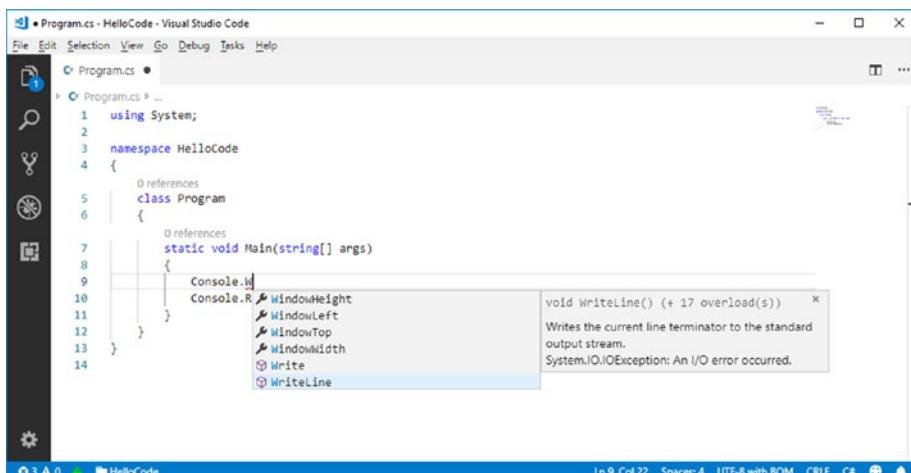


Figure 3-12. IntelliSense showing suggestions as you type and advanced word completion

As you can see in Figure 3-12, IntelliSense shows the list of available members as you write, for the given type (in this case `Console`). When you select a word from the completion list, Visual Studio Code shows the member documentation.

Note The documentation for a type or member will only be available if it has been supplied by the developers. For example, in C# the documentation for types and members must be provided with XML comments. This will enable IntelliSense to display it in a tooltip, like in Figure 3-12.

Press either Tab or Enter to complete the word insertion. Not limited to this, IntelliSense in Visual Studio code supports suggestion filtering: based on the CamelCase convention, you can type the uppercase letters of a member name to filter the suggestion list. For instance, if you are working against the `System.Console` type and you write `cv`, the suggestion list will show the `CursorVisible` property, as demonstrated in Figure 3-13.

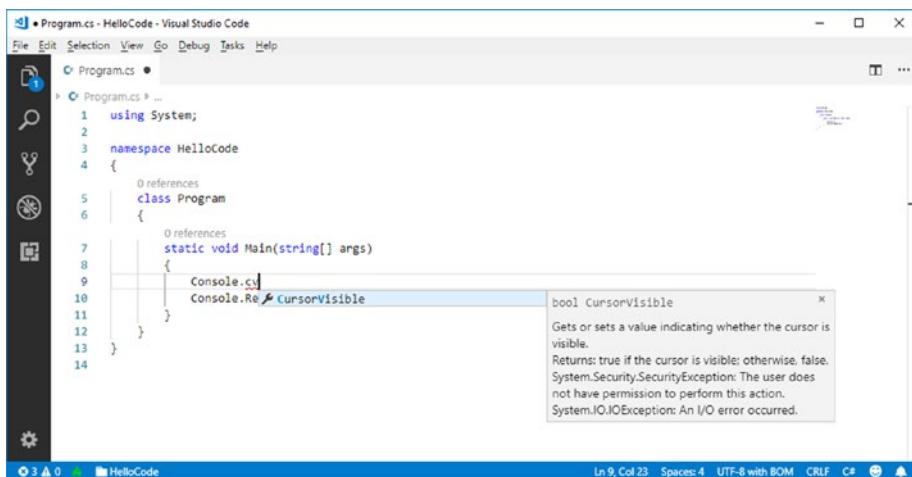


Figure 3-13. Suggestion filtering in IntelliSense

IntelliSense also provides the foundation for other advanced features in the code editor that depend on it, described in the next subsections.

Parameter Hints

When you write a function invocation, IntelliSense also shows a tooltip that describes each parameter. This feature is called *parameter hints* and is available only if the documentation for function parameters has been implemented. An example is visible in Figure 3-14.



Figure 3-14. IntelliSense shows parameter hints

For languages such as C# and TypeScript or, more generally, with languages that allow for function overloads, parameter hints show the description for the parameters of each overload. You can also scroll the list of overloads with the up and down arrow keys to select a different overload.

Inline Documentation with Tooltips

If you hover types, variables, and type members, Visual Studio Code will show a tooltip that contains the documentation for the selected object. Figure 3-15 provides an example.

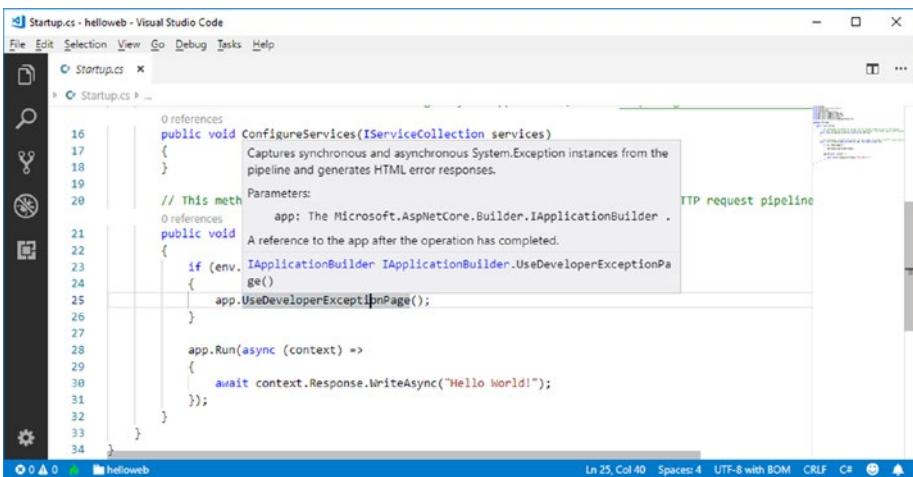


Figure 3-15. Tooltips provide quick, inline documentation

Like parameter hints, this feature is available only if the documentation has been implemented. If you instead hover a variable name, the tooltip will only show the type for the variable.

Go To Definition

Visual Studio Code provides another interesting feature called *Go To Definition*. You can hover over a symbol with the mouse and, if you press Ctrl (or ⌘ on macOS), the symbol will appear as a hyperlink; also, a tooltip will show the code that declares that symbol. You click the type name while pressing Ctrl, and you will be redirected to the code that defines that type. Figure 3-16 shows how the code editor appears when you press Ctrl and hover over a type name.

CHAPTER 3 LANGUAGE SUPPORT AND CODE EDITING FEATURES

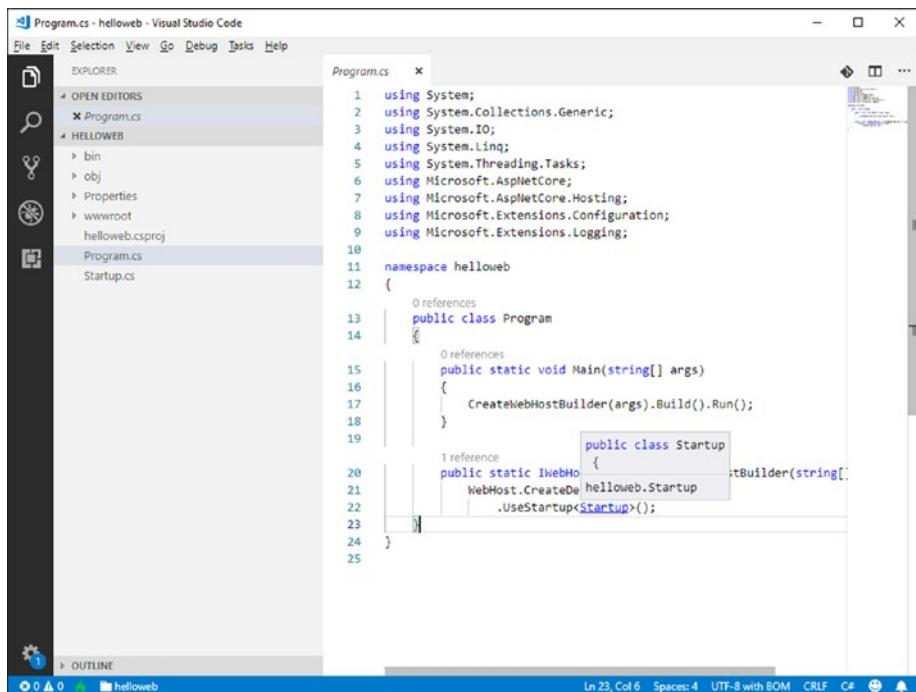


Figure 3-16. *Ctrl + hovering over a type enables Go To Definition*

The same tool is available if you select a type name and press F12 or if you right-click a type name and then select **Go To Definition** from the context menu. This is an extremely useful feature that lets you quickly browse between type definitions that are in different code files.

Note For C#, Go To Definition can also open the definition of a type exposed by the .NET Core libraries, not just your code.

Find All References

Find All References is a very useful feature that makes it easy to see how many times and where a type or member has been used across the code. For each type or member, the code editor shows how many times a member has been referenced and in which files. Figure 3-17 shows an example based on finding all references of a type called `Startup`.

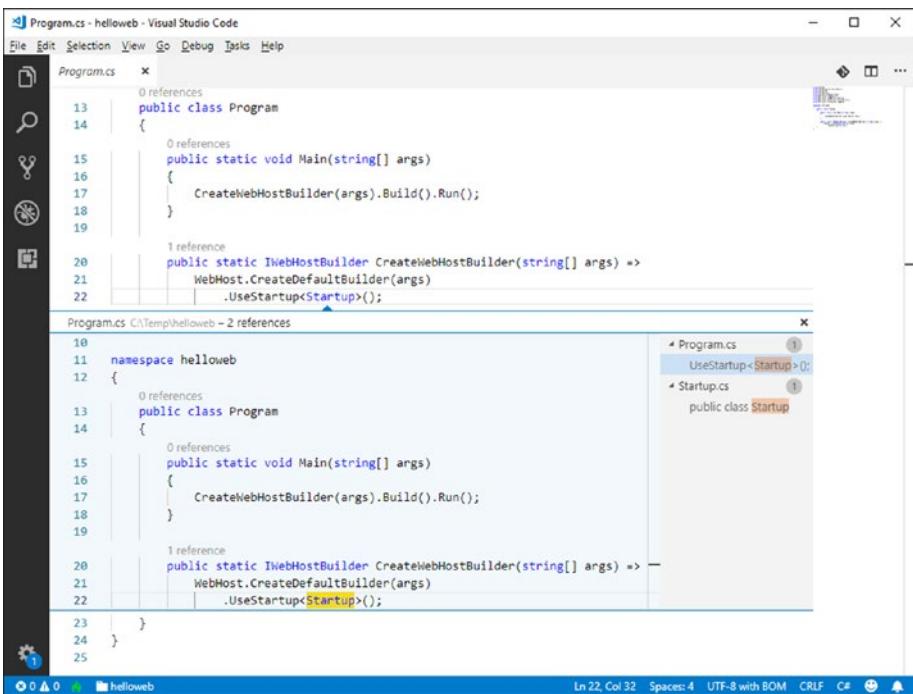


Figure 3-17. Finding all references of types and members

If you click an occurrence in the list on the right, the code editor brings up a popup containing the code where that occurrence has been found. It is very important noting that this popup is interactive, which means that you can edit the code directly, without the need of opening the containing code file separately. This allows keeping your focus on the code, saving time. Also, notice that the interactive popup shows, at the top, the file name that contains the selected reference.

Peek Definition

Suppose you have dozens of code files and that you want to see or edit the definition of a type you are currently using. With other editors, you should search among the code files, which not only can be annoying but that would also move your focus away from the original code. Visual Studio Code brilliantly solves this problem with a feature called **Peek Definition**.

You can simply right-click a type name and then select **Peek Definition** (the keyboard shortcut is Alt+F12); an interactive popup window appears, showing the code that defines the type, giving you not only an option to look at the code but also of direct editing. Figure 3-18 shows the peek window in action.

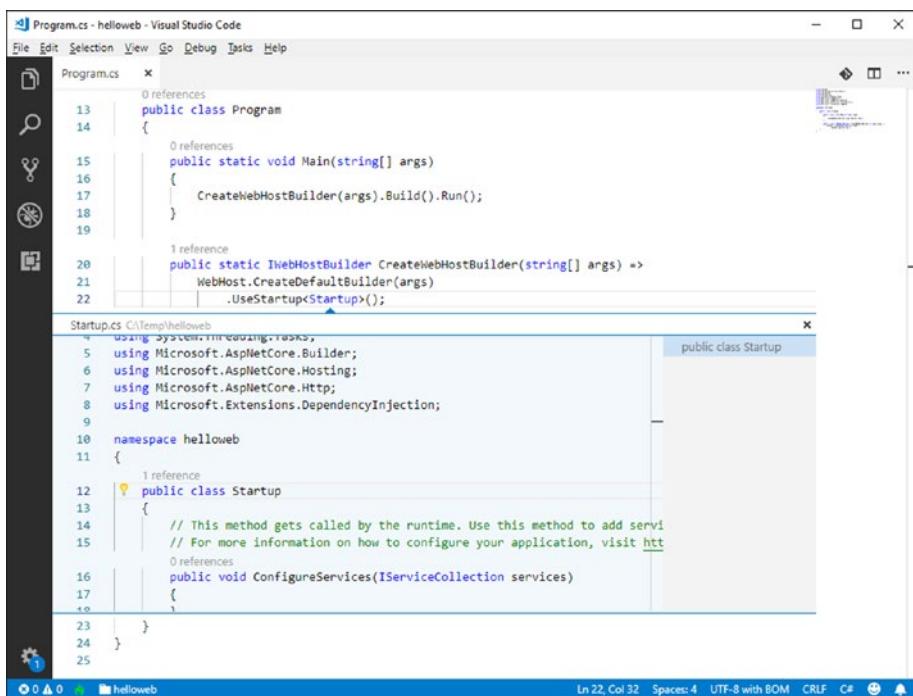


Figure 3-18. Working on a type defined in another file with Peek Definition

As you can see, the peek window is very similar to the Find All References feature, and it still shows the file name that defines the type at its top. Simply click the file name to open the code file in a separate editor.

Renaming Symbols and Identifiers

It is very frequent to rename a symbol, so Visual Studio Code offers a convenient way to accomplish this. If you press F2 over the symbol you wish to rename or right-click and then select the **Rename Symbol** command, a small interactive popup appears. Figure 3-19 shows an example based on a symbol called app. There you can write the new name without any dialogs, keeping your focus on the code.

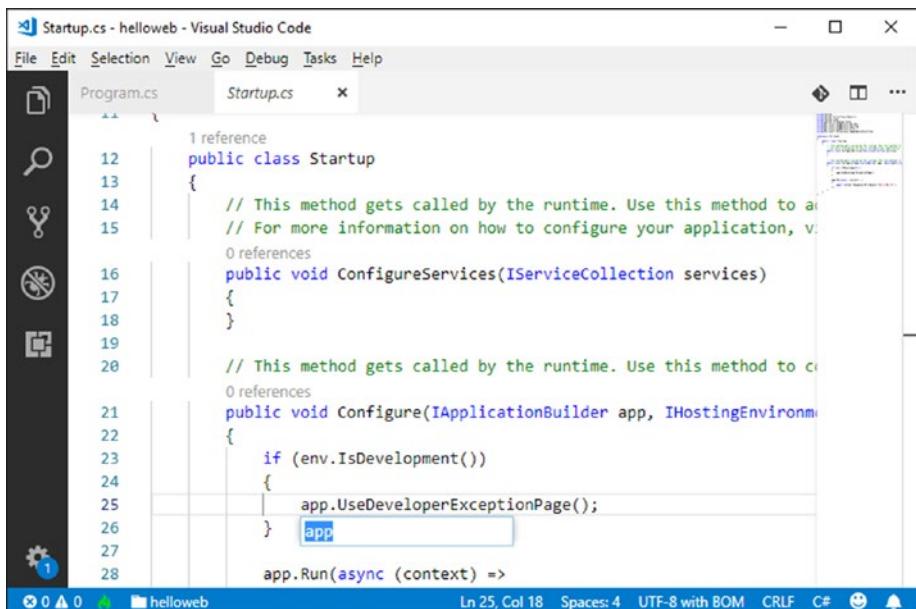


Figure 3-19. Renaming symbols

All references of that symbol will be renamed accordingly. Additionally, you can rename all the occurrences of an identifier. You simply right-click the identifier, then select **Change All Occurrences** (or press Ctrl+F2 on Windows/Linux and ⌘+F2 on macOS); all the occurrences will be highlighted and updated with the new name as you type.

Live Code Analysis

With C#, TypeScript, and with languages whose support can be enhanced via extensions like Python, Visual Studio Code can detect code issues as you type, suggesting fixes and offering code refactorings. This is one of the most powerful features in this tool, which is something that you will not find in most other code editors. The next examples are based on the C# programming language, since (together with TypeScript) this supports the richest experience possible in Visual Studio Code, and therefore it is a good choice to discuss the powerful coding features available. Of course, everything discussed here applies to all other languages that support the same enhanced features.

According to the severity level of a code issue, Visual Studio Code underlines with squiggles the pieces of code that need your attention. Green squiggles mean a warning; red squiggles mean an error that must be fixed. If you hover over the line or symbol with squiggles, you will get a tooltip that describes the issue. Figure 3-20 shows two code issues, one with green squiggles (an unused local variable) and one with red squiggles (a symbol that does not exist), plus the tooltip for the code issue with the higher severity level.

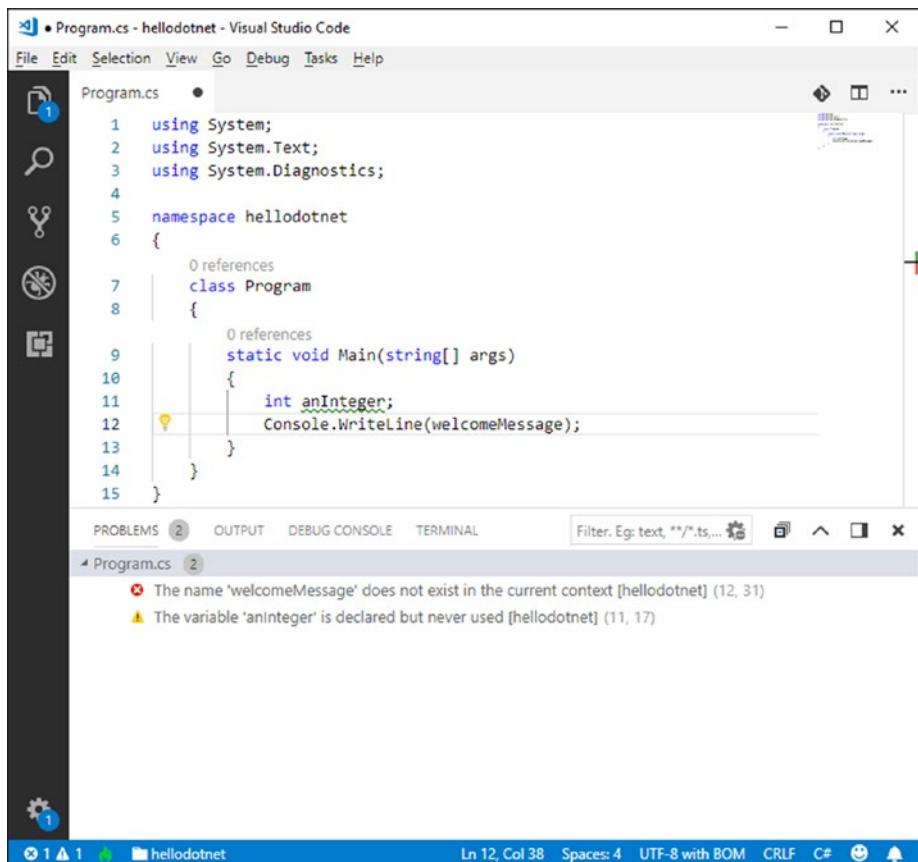


Figure 3-20. Code issue detection as you type

Code issues are detected as you type and they are also listed in the Problems panel. If you look at Figure 3-20, you can also see an icon with the shape of a light bulb. This icon is a shortcut for a tool called Light Bulb. When you click the Light Bulb, Visual Studio Code shows possible code fixes for the current context. For example, if you look at Figure 3-21, you can see the suggestions that the Light Bulb provides to fix the missing symbol underlined with red squiggles.

CHAPTER 3 LANGUAGE SUPPORT AND CODE EDITING FEATURES

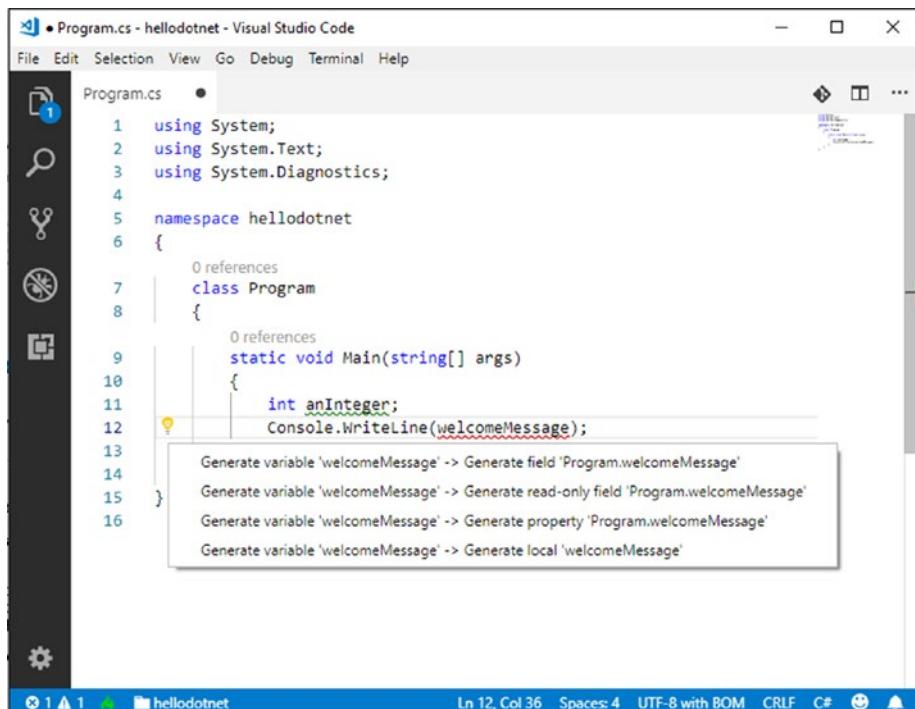


Figure 3-21. Potential fixes suggested by the Light Bulb

In this particular case, the editor suggests four options, such as creating a field, a read-only field, a property, or a local variable. In this particular case, a field would be created as follows:

```
private static bool welcomeMessage;
```

Instead, a property would be generated like this:

```
public static bool welcomeMessage { get; private set; }
```

Probably `bool` is not the type you would expect here, but Visual Studio Code has not enough information to infer a different type. However, when the code contains some information that Visual Studio Code could use to

understand the proper type, properties, fields, and local variables would be generated of the expected type. With the Light Bulb, it is also easier to generate types on the fly. Figure 3-22 shows an example based on an object called person, for which a type has not been defined yet. As you can see, for this context the code editor shows a larger list of possible fixes, including generating a new class, either in the current file or in a separate file.

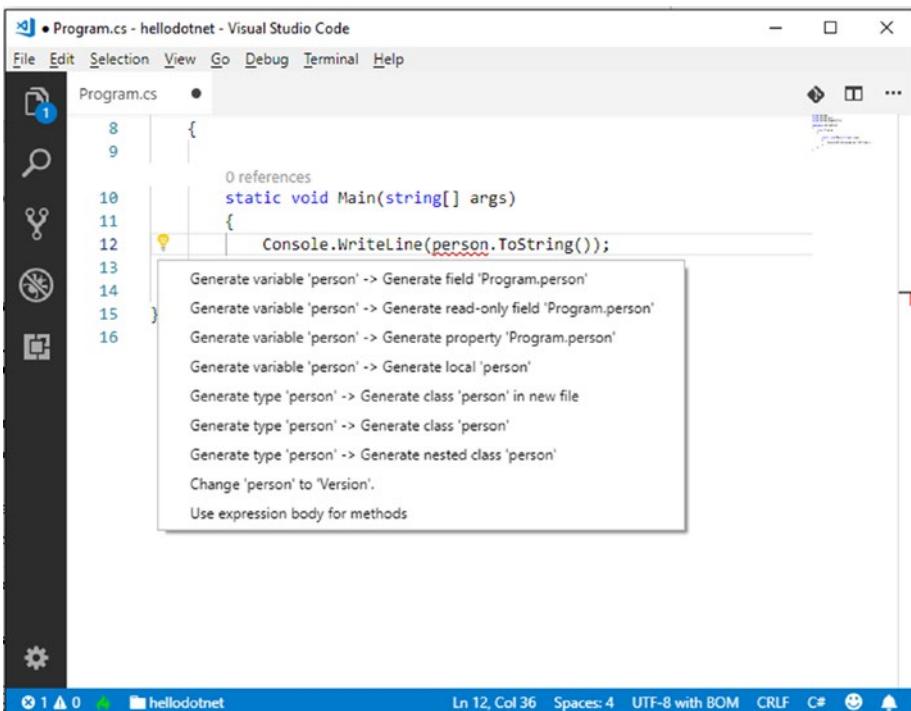


Figure 3-22. Generating types on the fly

CHAPTER 3 LANGUAGE SUPPORT AND CODE EDITING FEATURES

Not limited to this, the Light Bulb can help you refactor your code and keep it cleaner. For example, you can click any of the using directives (or equivalent in other languages) and, when the Light Bulb appears, you can see how it offers to remove unused code, as shown in Figure 3-23.

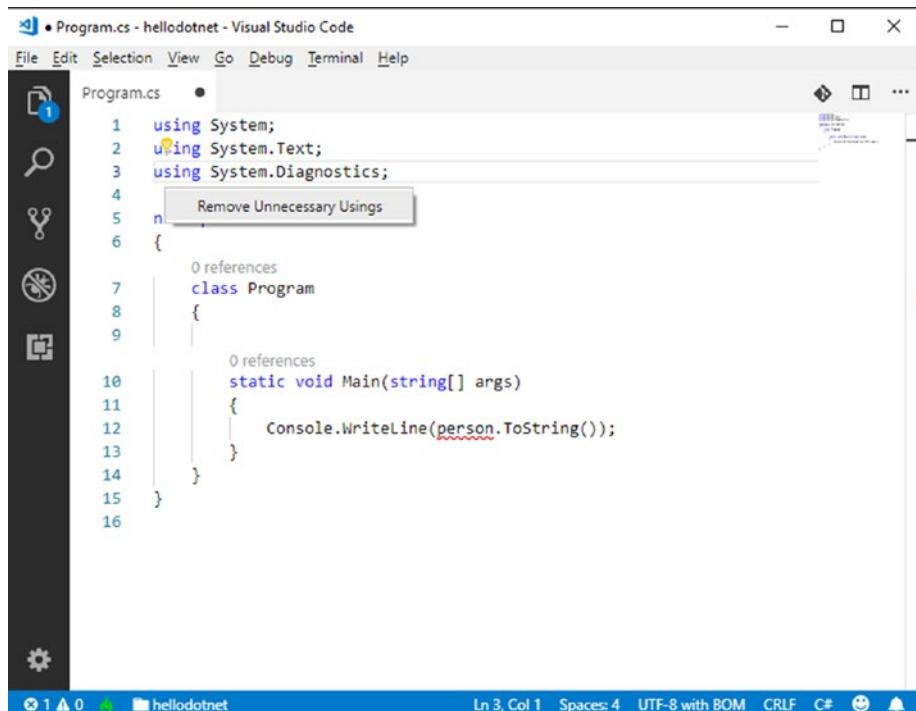


Figure 3-23. *Code refactoring made easy*

Actually, there is even more power. Suppose you want to create a class that implements the `IDisposable` interface. As you can see in Figure 3-24, the code editor cannot find the definition of such interface and shows a red squiggle, but the Light Bulb provides shortcuts for quickly fixing this issue. For example, it suggests adding a `using System;` directive, which is what the code needs.

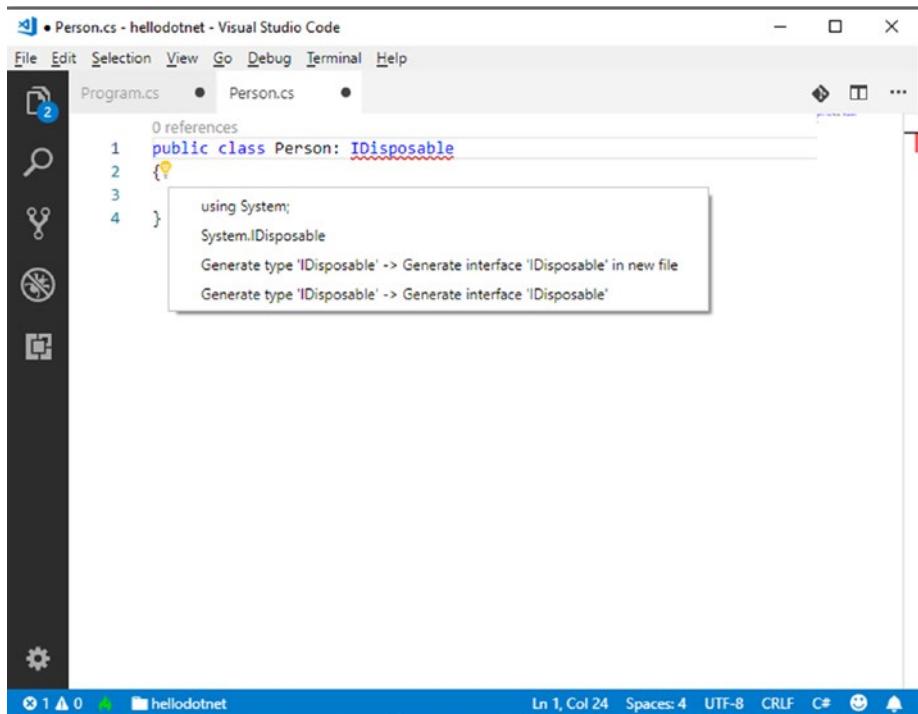


Figure 3-24. Adding missing directives

At this point, `IDisposable` will be still underlined with a red squiggle because the code is not implementing the interface yet. If you still enable the Light Bulb, you will see how the code editor suggests potential fixes based on the current context, such as implementing the interface in different ways (see Figure 3-25).

CHAPTER 3 LANGUAGE SUPPORT AND CODE EDITING FEATURES

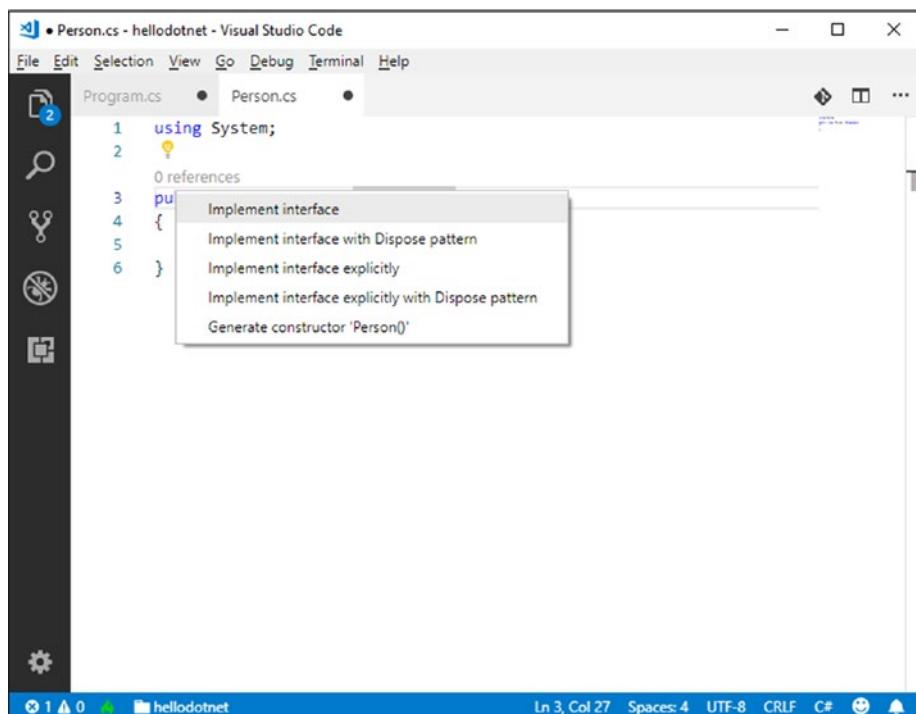


Figure 3-25. The Light Bulb provides suggestions based on the current context

Just to give you an idea of the power of this tool, following is the code that is generated if you choose to implement the interface with Dispose pattern:

```
using System;

public class Person: IDisposable
{
    #region IDisposable Support
    private bool disposedValue = false; // To detect redundant calls
```

```
protected virtual void Dispose(bool disposing)
{
    if (!disposedValue)
    {
        if (disposing)
        {
            // TODO: dispose managed state (managed objects).
        }

        // TODO: free unmanaged resources (unmanaged objects)
        // TODO: set large fields to null.

        disposedValue = true;
    }
}

// TODO: override a finalizer only if Dispose(bool disposing) above
// has code to free unmanaged resources.
// ~Person() {
//     // Do not change this code. Put cleanup code in
//     Dispose(bool disposing) above.
//     Dispose(false);
// }

// This code added to correctly implement the disposable
// pattern.
public void Dispose()
{
    // Do not change this code. Put cleanup code in
    // Dispose(bool disposing) above.
    Dispose(true);
```

CHAPTER 3 LANGUAGE SUPPORT AND CODE EDITING FEATURES

```
// TODO: uncomment the following line if the finalizer  
// is overridden above.  
// GC.SuppressFinalize(this);  
}  
#endregion  
}
```

You would get a similar result, but with different implementation, if the choice was one of the other possible code fixes. Though it is not possible to show examples for all the code fixes that Code can apply, what you have to keep in mind is that suggestions and code fixes are based on the context for the code issue, which is a very powerful feature that makes Visual Studio Code a unique editor.

Summary

Visual Studio Code is a code-centric tool that supports out of the box a wide variety of languages, offering coding features such as syntax colorization, delimiter matching, code block folding, multicursors, code snippets, and code completion that are common to all the supported languages.

In addition, languages such as TypeScript and C# provide the so-called evolved code editing experience via integrated tools such as IntelliSense, Go To Definition and Peek Definition, Find All References, and the extremely powerful Light Bulb that detects code issues as you type and that suggests potential fixes based on the context. Now that you have knowledge of the powerful coding features that Visual Studio Code offers, it is time to see how to use them with individual source code files and structured folders.

CHAPTER 4

Working with Files and Folders

Being the powerful editor it is, Visual Studio Code provides a convenient way of working with code files and folders containing both loose files and projects. In this chapter you will learn how to work with individual files, with folders containing source code files, and with workspaces. You will also learn about VS Code's independency from proprietary project systems as well as the built-in support for a few, popular project types.

Visual Studio Code and Project Systems

Visual Studio Code is file and folder based. This means that you can open one or more code files distinctly, but it also means that you can open a folder that contains source code files and treat them in a structured, organized way. When you open a folder, Visual Studio Code searches for one of the following files:

- Tsconfig.json
- Jsconfig.json
- Package.json
- Project.json
- .sln Visual Studio solutions for .NET Core with the C# extension installed

If Code finds one of these files, it is able to organize the file structure into a convenient editing experience and can offer additional rich editing features such as IntelliSense and code refactoring. If a folder only contains source code files, without any of the aforementioned .json or .sln files, it still opens and shows all the source code files in that folder, providing a convenient way to switch between all of them. This chapter describes how to work with individual files and with folders in Visual Studio Code, and more details about how it manages projects will be provided in the subsection “Working with Folders and Projects.”

Working with Individual Files

The easiest way to get started editing with Visual Studio Code is working with one code file. You can open an existing supported code file with File ➤ Open (Ctrl+O or ⌘+O on macOS). Visual Studio Code automatically detects the language for the code files and enables the proper editing features. Of course, you can certainly open more files and easily switch between files by pressing Ctrl+Tab (or ⌘+Tab on macOS). As you can see in Figure 4-1, a convenient popup will show the list of open files; by pressing Ctrl+Tab, you will be able to browse files, and when you release the keys, the selected file will become the active editing window.

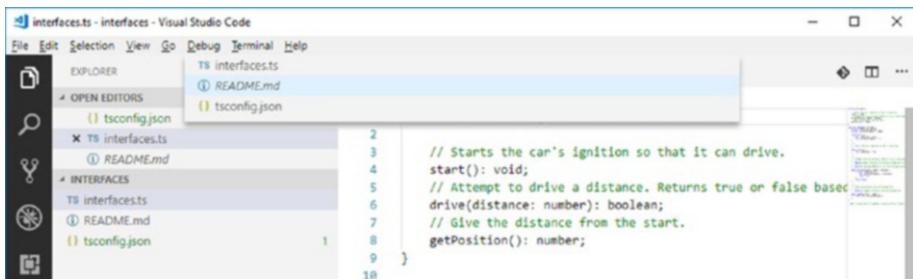


Figure 4-1. Quickly navigating between open editors

Simply close an editor by using the **Close** button at the upper right corner, or use the **Close All Files** command in the **File** menu.

Note In the Visual Studio Code terminology, it is common to refer to open files as active editors or open editors. This is because editor windows are not limited to code files, but they can also display documentation files or provide formatted previews of the content of other types of files (e.g., images and spreadsheets).

Creating Files

You have several ways to create a new file:

- Via **File ▶ New File**
- By pressing **Ctrl+N** (**⌘+N** on macOS)
- By using the **New File** shortcut in the Welcome page
- By clicking the **New File** button in the Explorer bar when a folder is currently opened

By default, new files are treated as plain text files. In order to change the language for a new file, click the **Select Language Mode** item in the right corner of the status bar, near the smile icon. In this case, you will see Plain Text as the current mode, so click it. As you can see in Figure 4-2, you will be presented with a list of supported languages where you can select the new language for the current file. You can also start typing a language name to filter the list.

CHAPTER 4 WORKING WITH FILES AND FOLDERS

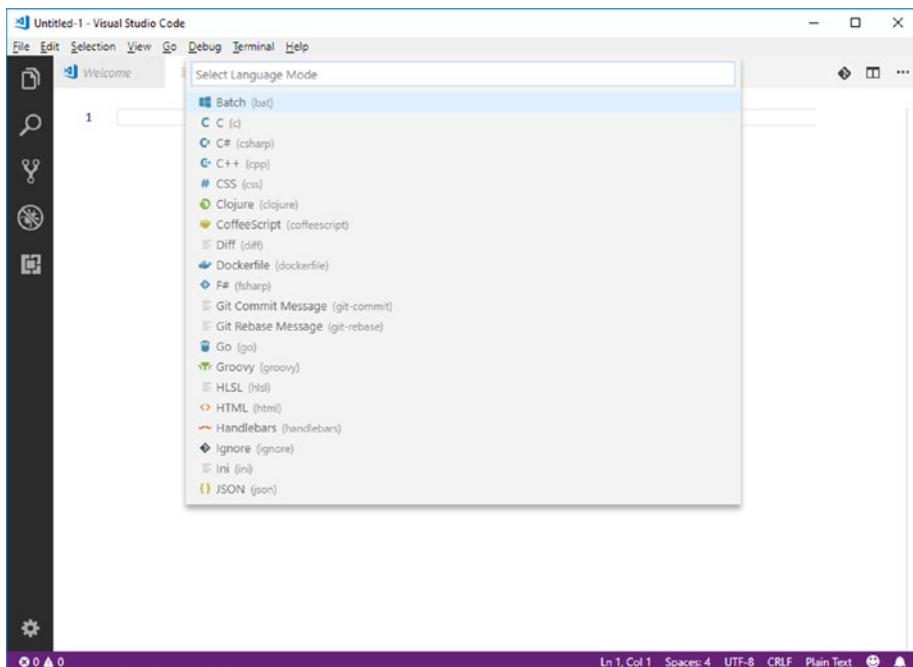


Figure 4-2. Selecting the language for a new file

When you select a new language, the **Select Language Mode** item is updated with the current language, and the editor enables the supported features for the selected language, such as syntax colorization, word completion, and code snippets.

Obviously, you can change the language of any open code file, not just new files.

File Encoding, Line Terminators, and Line Browsing

Visual Studio Code allows you to specify an encoding for new and existing files. Default encoding for new files is UTF-8. You can change the current encoding by clicking the **Select Encoding** item in the status bar (in the previous figures, it is represented with **UTF-8**, the current encoding). You

will be presented with a long list of supported encodings and a search box where you can filter the list as you type (see Figure 4-3).

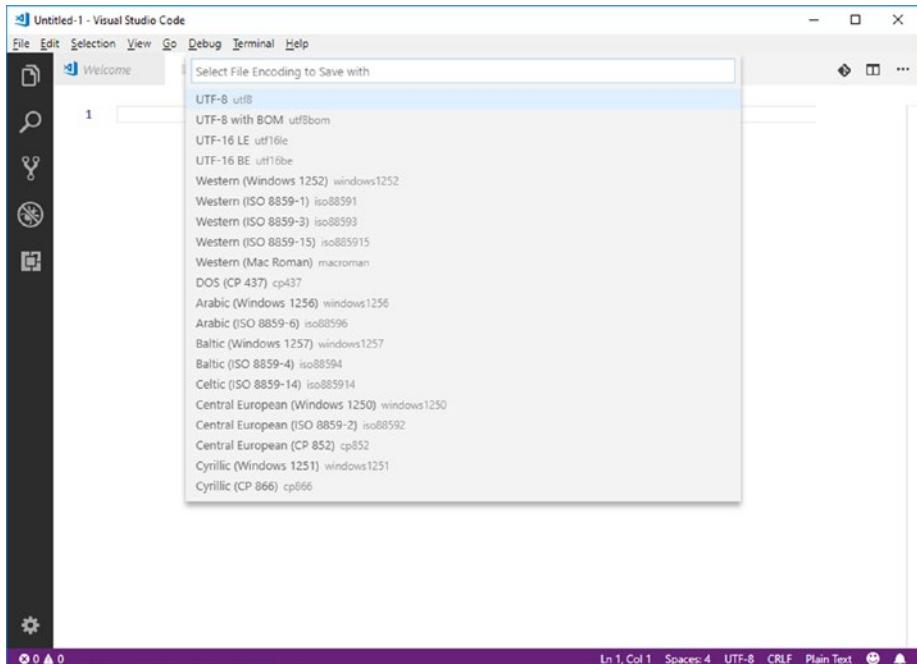


Figure 4-3. Selecting the file encoding

Similarly, you can change the line terminator by clicking the **Select End of Line Sequence** item (in previous figures it's represented by **CRLF**). Visual Studio Code supports CRLF (Carriage Return and Line Feed) and LF (Line Feed), and the default selection is CRLF. You can also move fast to a line of code by clicking the **Go to Line** item, represented by the line number/column group in the status bar. This will open up a search box where you can write the line number you want to go to, and the line of code will be immediately highlighted as you type (see Figure 4-4).

CHAPTER 4 WORKING WITH FILES AND FOLDERS

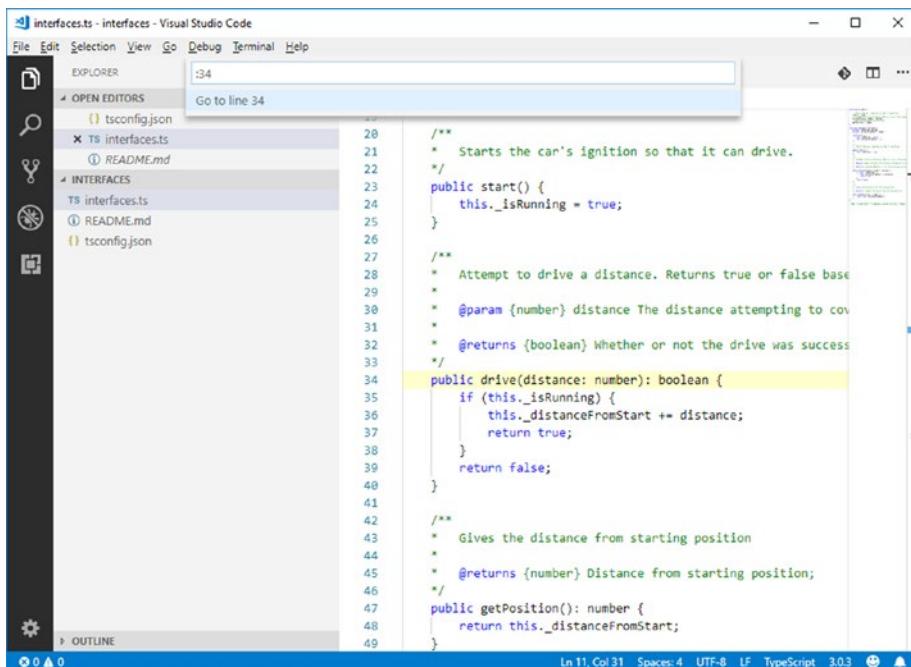


Figure 4-4. Quickly moving to a specific line of code with *Go to Line*

Working with Folders and Projects

Differently from other development environments, such as Microsoft Visual Studio, Visual Studio Code is folder based, not project based. This makes Visual Studio Code independent from proprietary project systems. VS Code can open folders on disk containing multiple code files and organize them the best way possible in the environment, and it also supports a variety of project files. More specifically, when you open a folder, VS Code first searches for:

- *MSBuild solution files (.sln)*: In this case, Visual Studio Code expects to find a .NET Core solution made of C# projects, so it scans the referenced projects (*.csproj files) and organizes files and subfolders in the proper

way. Remember that Visual Studio Code needs the Microsoft C# extension installed in order to properly treat solution files. It is worth mentioning that VS Code can open any .sln solution, but full support is currently offered only for .NET Core. An example of this scenario will be offered in Chapter 8, “Automating Tasks.”

- *tsconfig.json files*: If found, Visual Studio Code knows this represents the root of a TypeScript project, so it scans for the referenced files and provides the proper file and folder representation.
- *jsconfig.json files*: If found, Visual Studio Code knows this represents the root of a JavaScript project. So, similarly to TypeScript, it scans for the referenced files and provides the proper file and folder representation.
- *package.json files*: These are typically included with JavaScript projects and .NET Core projects, so Visual Studio Code automatically resolves the project type based on the folder’s content.
- *project.json files*: If found, Code treats the folder as a legacy DNX project written in C#. DNX stands for .NET Execution Environment and represents the runtime with Software Development Kit (SDK) built on top of .NET Core 1.0 and 1.1. Project.json has been discontinued as a project system starting with .NET Core 2.0 so, if you have projects written with earlier versions, I recommend you to read the migration guide from Microsoft (<https://docs.microsoft.com/en-us/dotnet/core/migration/from-dnx>).

Note Opening a .sln or .json file directly will result in editing the content of the individual file. For this reason, you must open a folder, not a solution or a project file.

Additional project systems might be supported via extensibility. If no one of the supported projects is found, Visual Studio Code loads all the code files in the folder as a loose assortment, organizing them into a virtual folder for easy navigation. Now let's discover how Visual Studio Code allows you to work with folders and supported projects providing appropriate examples.

Opening a Folder

You open a folder via **File ▶ Open Folder** or via the **Open Folder** shortcut in the Welcome page. You can also drag and drop a folder name from Windows' Explorer or macOS' Finder onto Visual Studio Code.

Note On Windows, the VS Code installer will also provide an option to enable a shortcut called **Open With Code** when you right-click a folder or file name in File Explorer.

Whatever folder you open, VS Code organizes files and subfolders into a structured view represented in the Explorer side bar. Figure 4-5 shows an example based on a C# project.

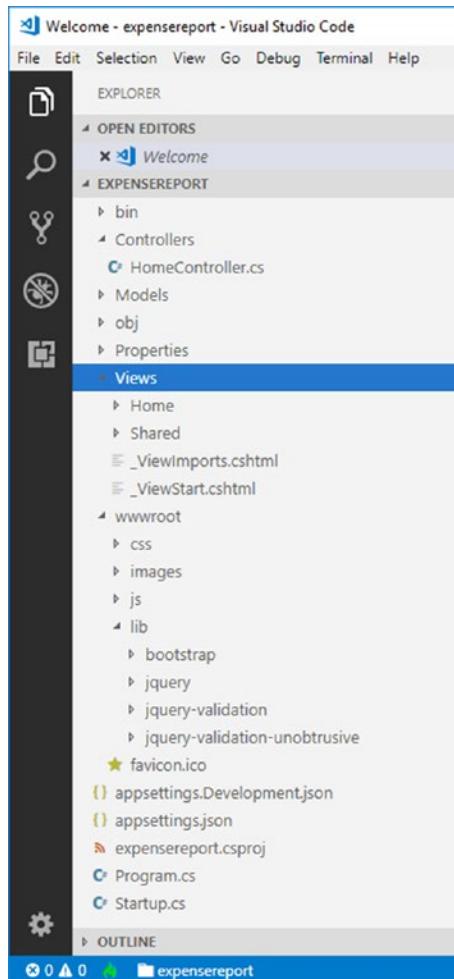


Figure 4-5. The structured view of files and folders inside the Explorer

The root container is the folder name. Nested you see files and subfolders, and you can expand each subfolder to browse every file it contains. Simply click a file to open an editor window on it.

Opening .NET Core Solutions

When you open a folder that contains a .NET Core solution based on the MSBuild project system (.sln file), Visual Studio Code organizes all the code files into the Explorer bar and enables all the available editing features for C#. Figure 4-6 shows an example.

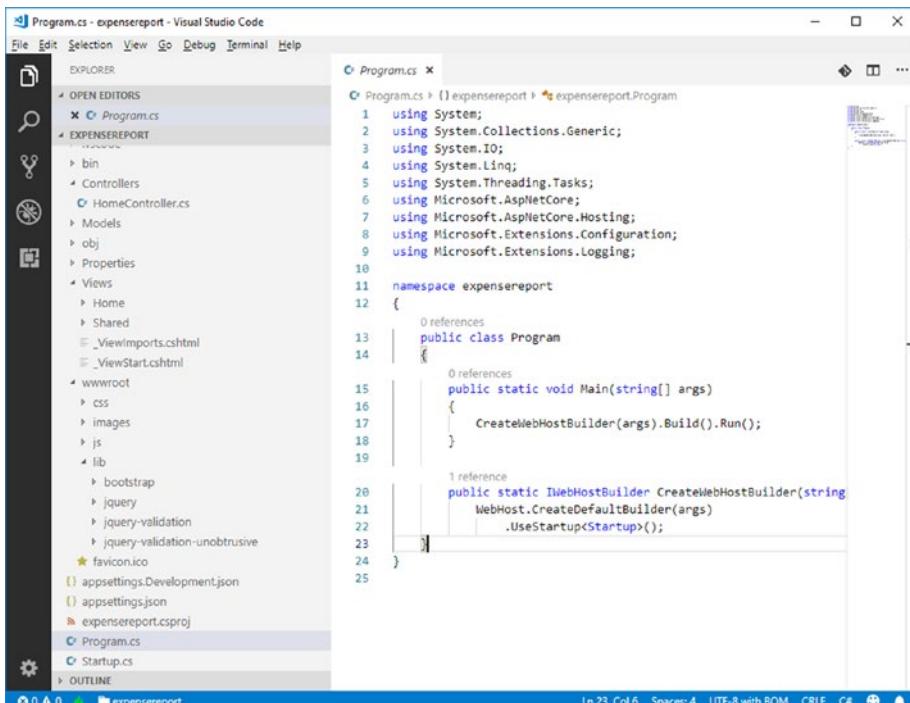


Figure 4-6. A .NET Core solution opened in Visual Studio Code

Notice how the root level in Explorer is the project name. You can browse folders, code files, and edit anything that Visual Studio Code can properly recognize. It is worth mentioning that Visual Studio Code can certainly open any MSBuild solution, not only .NET Core solutions, but it will only be able to run and debug .NET Core applications. For instance, if you open a Windows Presentation Foundation (WPF) solution in VS Code, you will

still benefit the structured folder view in the Explorer bar and then full C# language support, but you will not be able to build, run, and debug the code. Instead, with .NET Core you also have integrated debugging support which allows running, debugging, and testing code directly within VS Code. This will be discussed in Chapter 9, "Running and Debugging Code."

Opening JavaScript and TypeScript Projects

Similarly to .NET Core solutions, Visual Studio Code can manage JavaScript folders by searching for `jsconfig.json` or `package.json` files. If found, Code organizes the list of folders and files the proper way and enables all the available editing features for all the files it supports, as shown in Figure 4-7.

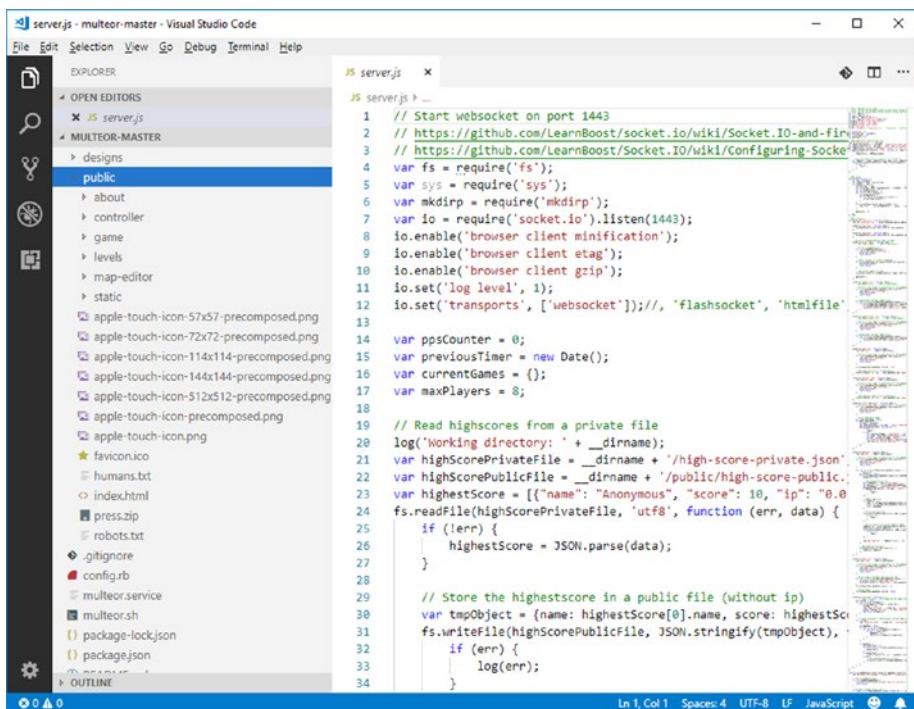


Figure 4-7. A JavaScript project opened in Visual Studio Code

CHAPTER 4 WORKING WITH FILES AND FOLDERS

TypeScript projects' behavior is the same as for JavaScript, except that Visual Studio Code will search for a file called `tsconfig.json` as the root.

Opening Loose Folders

Visual Studio Code allows opening folders that contain unrelated, loose assortments of files. Visual Studio Code creates a logical root based on the folder name, showing files and subfolders. Figure 4-8 shows an example based on a sample folder called `MyFiles` that contains files in different languages.

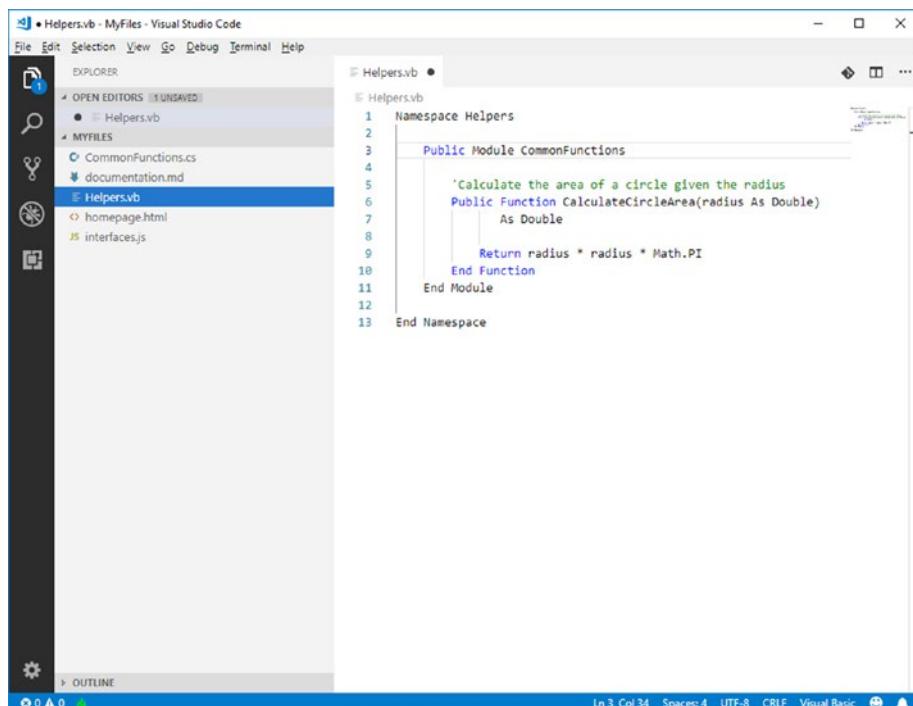


Figure 4-8. A folder containing a loose assortment of files

With this option, you can basically open any folder in VS Code and edit all supported files taking advantage of the code editing features for each file individually.

Working with Workspaces

Visual Studio Code has recently introduced the concept of *workspace*. A workspace can be thought of as a logical container of folders.

Note If you have experience with Microsoft Visual Studio, a workspace in Visual Studio Code can be compared to a Visual Studio solution as a container of projects.

Workspaces are extremely useful to organize multiple projects and/or folders into one place. For example, you might have a .NET Core Web API project, a JavaScript application that consumes such API and a folder containing documentation. Instead of working on each folder separately, you can put them all under the same workspace and have them all available in Visual Studio Code at the same time. Figure 4-9 shows a workspace, called SampleWorkspace, which includes the projects and folders shown in the previous figures.

CHAPTER 4 WORKING WITH FILES AND FOLDERS

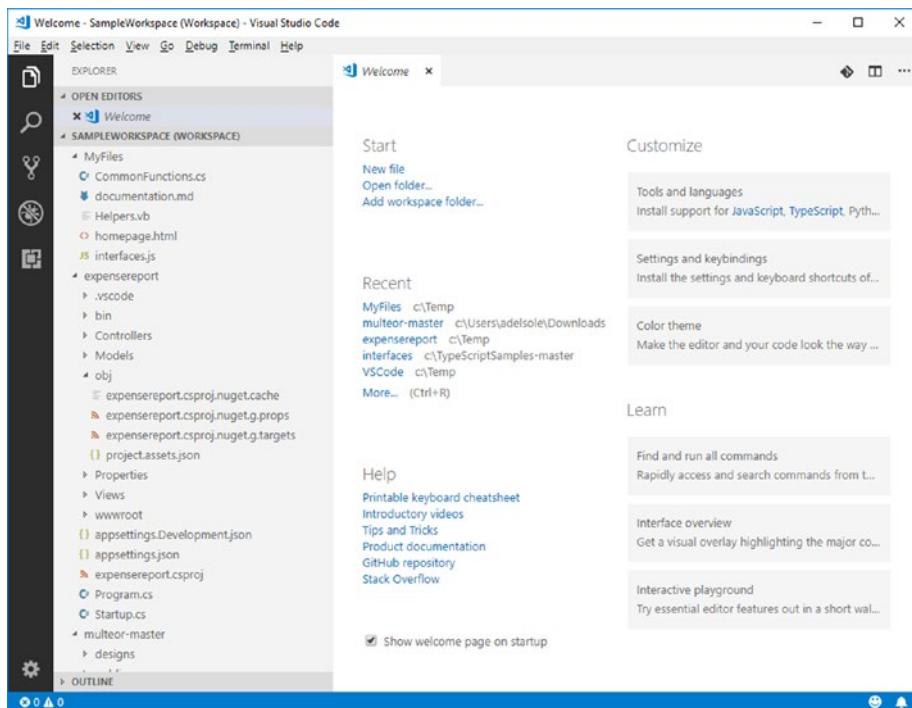


Figure 4-9. A workspace can group multiple projects and folders into one logical container

The multeor-master folder is referred to a sample open-source project called multeor that you can download for instructional purposes from <https://github.com/filidorwiese/multeor>. The Explorer bar shows the name of the workspace in uppercase together with the **(WORKSPACE)** literal so that it's easier to recognize it. In the next paragraphs, I will explain in more detail how to create and open workspaces and what is the structure of a workspace file.

Creating Workspaces

Creating a workspace can be done whether you already have any folders opened or not. If you already have any folders opened, you can select **File ➤ Save Workspace As**. VS Code will ask you to specify the location and file name for the new workspace. A workspace is represented by a JSON file with `.code-workspace` extension, whose structure will be explained shortly.

The workspace name is simply the file name without the `.code-workspace` extension and will be shown in the Explorer bar (see Figure 4-9). Then you can add other folders to the workspace by selecting **File ➤ Add Folder to Workspace** or via the **Add workspace folder** shortcut in the Welcome page. Added folders will be displayed in the Explorer bar under the workspace root. If you do not have any folders already opened, you can either start with **File ➤ Save Workspace As** or with **File ➤ Add Folder to Workspace**. With the first option, you will basically create an empty workspace with a name, and then you will add folders as described in the preceding text. With the second option, you will instead create an empty, untitled workspace starting from an existing folder. In this case, in fact, the Explorer bar will show **UNTITLED (WORKSPACE)** as the new workspace name. When you save the workspace like described in the preceding text, the Explorer bar will show the new name based on the workspace file name. Remember that workspaces are only logical containers and do not affect the structure or behavior of your projects and folders in any manner.

Note Folders you add to a workspace can be anywhere on disk; Visual Studio Code will be able to group their content under the workspace root and let you work like if they were in the same location.

Opening Existing Workspaces

You can open an existing workspace via **File ➤ Open Workspace**. You can also drag and drop a workspace file name from your operating system's file browsing program onto the Visual Studio Code surface. Opening a .code-workspace file directly will simply result in viewing the file content, not opening the workspace. Similarly, opening a folder that contains a .code-workspace file will only result in opening the folder, not the workspace. You can only use the specific commands described at the beginning of this paragraph.

Workspace Structure

The information of a Visual Studio Code workspace is stored inside a file with .code-workspace extension. A workspace file is a JSON file with a root element called **folders**. This is an array of path elements, each assigned with the name of a folder that is included in the workspace. The following JSON markup represents the workspace file of the example shown in Figure 4-9:

```
{  
  "folders": [  
    {  
      "path": "MyFiles"  
    },  
    {  
      "path": "ExpenseReport"  
    },  
  ]}
```

```
{  
    "path": "C:\\Users\\adelsole\\Downloads\\\\  
    multeor-master"  
}  
]  
}
```

Notice that the full pathname of a folder is only provided if such a folder is not in the same location of the workspace file. In this case, the .code-workspace file, the MyFiles folder, and the ExpenseReport folders are all in the same location; instead, the multeor-master folder is located under a different folder, C:\Users\adelsole\Downloads. If you want to see yourself the structure of a workspace file, you can open it within Visual Studio Code via **File > Open File**.

Summary

Visual Studio Code is file and folder based, and it allows for working with individual files as well as with folders that contain source code files and treat them in a structured, organized way.

It also supports a number of project systems such as .NET Core, TypeScript, and JavaScript, and it allows for creating and managing workspaces, logical containers of folders that make it easy to have multiple projects and folders under the same visual root. Visual Studio Code is not only a very powerful code editor but also a very flexible environment which can be customized in many ways. Customization is the topic of the next chapter.

CHAPTER 5

Customizing Visual Studio Code

Visual Studio Code is an extremely versatile development tool that can be customized and extended in many ways. In fact, you can customize its appearance, the code editor, and key shortcuts to make your editing experience extremely personalized.

Additionally, you can install third-party extensions such as new languages, debuggers, themes, linters, and code snippets. This chapter explains how to customize Visual Studio Code, explaining the difference between customizations and extensions. Then, in the next chapter, you will learn how to work with extensions.

Customizations and Extensions Explained

You can personalize the environment of Visual Studio Code with both customizations and extensions. The difference is that extensions add new instrumentation or they add functionalities to a tool or change the behavior of existing functionalities. Implementing IntelliSense for a language that does not have it by default, adding commands to the status bar, and adding custom debuggers are examples of extensions.

Customizations are instead related to environment settings and do not add functionalities to a tool. Table 5-1 summarizes customizations and extensions in VS Code.

Table 5-1. *Customizations and Extensions*

| Feature | Description | Type |
|--|--|---------------|
| Color themes | Style the environment layout with different colors. | Customization |
| User and workspace settings | Specify environment preferences. | Customization |
| Key bindings | Redefine keyboard shortcuts. | Customization |
| Language grammar and syntax colorizers | Add support to additional languages with syntax colorizers. | Customization |
| Code snippets | Add TextMate and Sublime Text snippets and write repetitive code faster. | Customization |
| Debuggers | Add new debuggers for specific languages and platforms. | Extension |
| Language servers | Implement your validation logic for files opened in VS Code. | Extension |
| Activation | Load an extension when a specific file type is detected or when a command is selected in the Command Palette. | Extension |
| Editor | Work against the code editor's content, including text manipulation and selection. | Extension |
| Workspace | Enhance the status bar, working file list, and other tools. | Extension |
| Eventing | Interact with Code's lifecycle events such as open and close. | Extension |
| Evolved editing | Improve language support with IntelliSense, Peek Definition, Go To Definition, and all the advanced, supported editing capabilities. | Extension |

In this chapter, you will see how to customize Visual Studio Code by changing the existing preferences. Then in the next chapter, you will see how to install extensions, including extensions that add new customizations to the development environment, such as themes and key bindings.

Customizing Visual Studio Code

In this section, you will discover how easy it is customizing Visual Studio Code, walking through the customization types described in Table 5-1.

Theme Selection

You can select among several themes to give Visual Studio Code a different look and feel. You select a color theme with **File ▶ Preferences ▶ Color Theme** or by clicking the **Settings** button and then **Color Theme**. The list of available color themes will be shown in the Command Palette, as you can see in Figure 5-1.

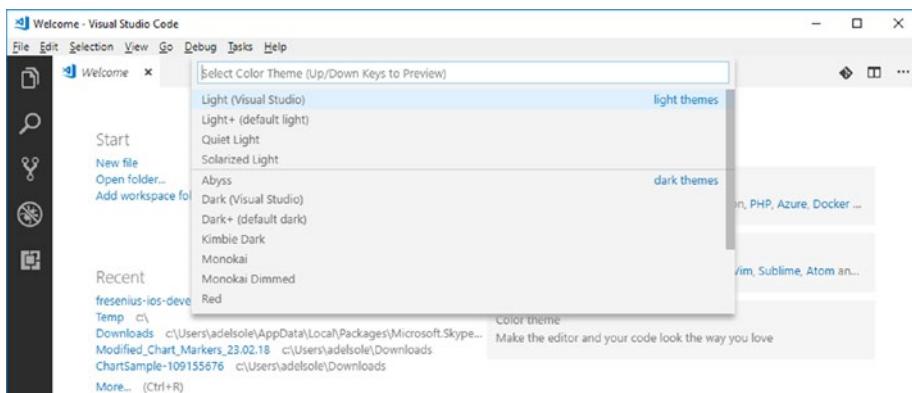


Figure 5-1. Selecting a theme

CHAPTER 5 CUSTOMIZING VISUAL STUDIO CODE

Once you select a different color theme, this will be applied immediately. Also, you can get a preview of the theme as you scroll the list with the keyboard. Figure 5-2 shows the **Dark (Visual Studio)** theme applied to VS Code, which is a very popular choice, while you can try yourself the other themes.

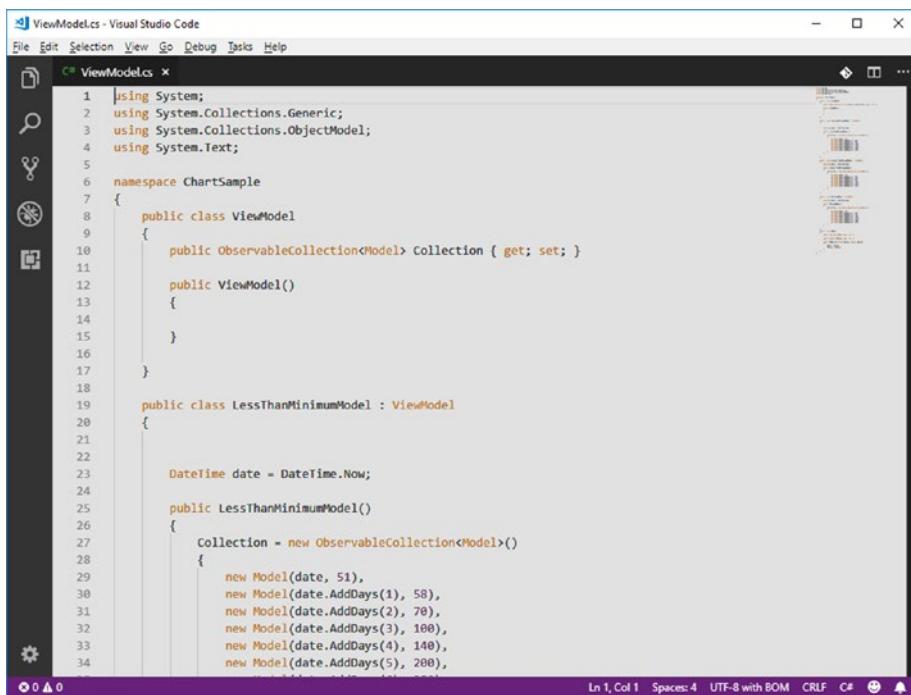


Figure 5-2. The Dark (Visual Studio) theme applied to Visual Studio Code

As you might expect, applying a theme also affects colors used in the code editor so that there is an appropriate brightness and contrast balance. In the next chapter, you will see how to install additional themes as extensions.

Customizing the Environment

In most applications, including other IDEs, you set environment settings and preferences via a convenient user interface, and VS Code is no less. There are two different types of settings: user settings and workspace settings. User settings apply globally to the development environment, while workspace settings only apply to the current project or folder. I will now cover both user and workspace settings.

Understanding User Settings

User settings globally apply to the VS Code's development environment.

Customizing user settings is accomplished by selecting **File > Preferences > Settings**.

Preferences > Settings. When you do this, the settings editor appears, as represented in Figure 5-3.

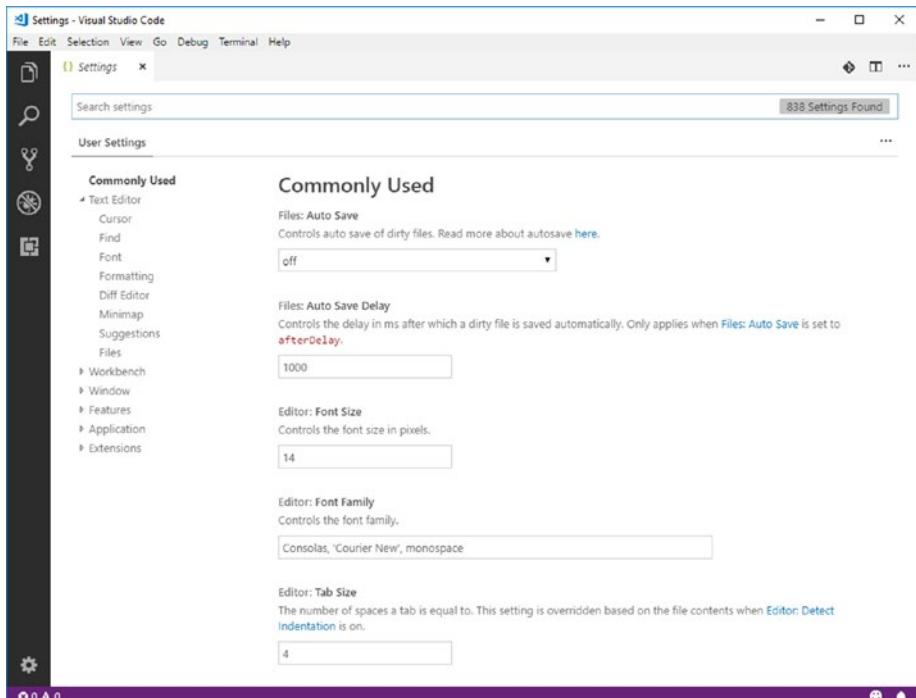


Figure 5-3. Working with user settings

CHAPTER 5 CUSTOMIZING VISUAL STUDIO CODE

On the left side of the editor, settings are grouped by category. In the **Search settings** bar, you can quickly search settings based on what you type, and you can also see the number of total settings found, which varies depending on the version of VS Code and on the number of extensions you have installed. You can manually expand setting categories manually, or you can just scroll the list of settings, and the related category is automatically expanded as you scroll. For instance, you could control the behavior of the Explorer bar by locating and selecting **Explorer** under the **Features** category, and here you could change the current settings, as shown in Figure 5-4.

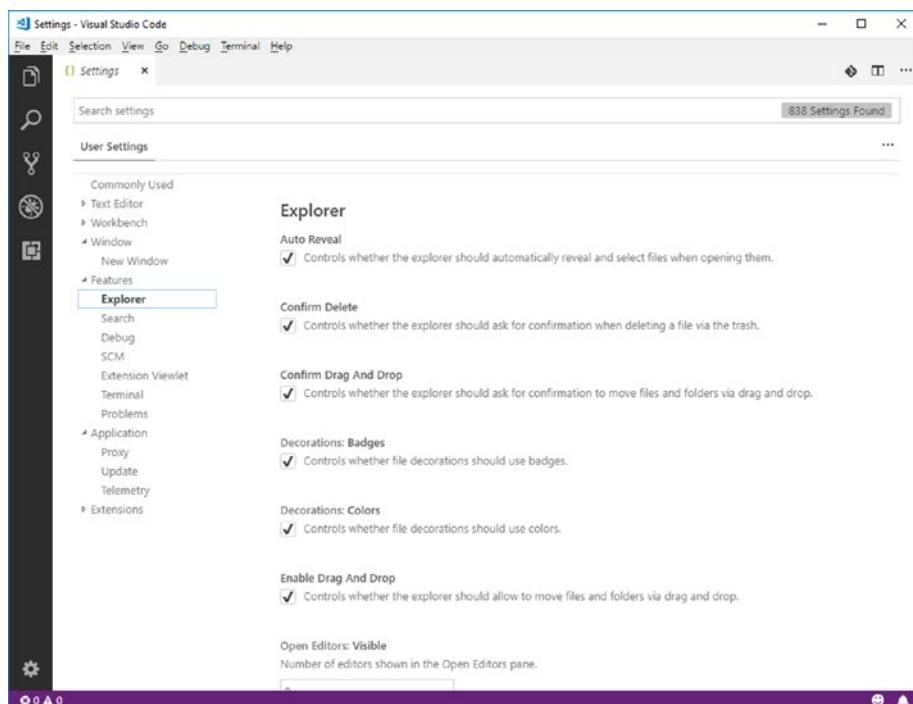


Figure 5-4. Changing user settings

Similarly, you could change settings and preferences for the text editor, the whole application, and extension settings. In fact, extensions that allow for customizing preferences store their settings in the same place as VS Code does, so that you have a unique settings editor. There are hundreds of settings and the number varies depending on your configuration and installed extensions, so it's not possible to list all settings here. For more details about available settings, visit the official documentation (<https://code.visualstudio.com/docs/getstarted/settings>).

Note Remember that changes will be applied only after you save your edits with **File ▶ Save**.

Behind the Scenes: The `settings.json` File

Behind the scenes, VS Code (and extensions) stores settings inside a file called `settings.json`. In this file, each key/value pair represents a specific setting and its value. Before Visual Studio Code 1.27.1, settings could only be edited manually by modifying the `settings.json` file, and only in version 1.27.1, the editor discussed in the previous section was introduced.

It is important to understand how this file works, so click the ... button below the number of settings displayed in the search bar, and then click **Open `settings.json`**. Figure 5-5 shows how the editor appears at this point.

CHAPTER 5 CUSTOMIZING VISUAL STUDIO CODE

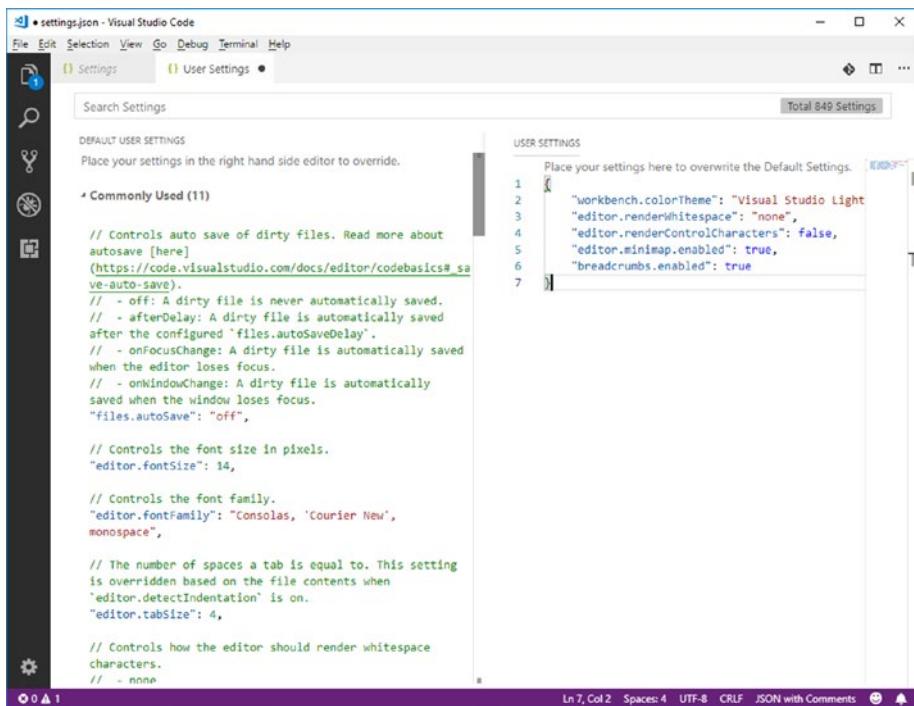


Figure 5-5. Working with the settings.json file

As you can see, the editor view is split in two areas: the **DEFAULT USER SETTINGS** on the left and the **USER SETTINGS** on the right.

Default user settings relate to Visual Studio Code's environment and tools but also to supported languages and to installed extensions' behavior (if any). The default settings view provides detailed comment for each available setting expressed with JSON format so that you can easily understand what setting a particular line applies to. You can easily provide custom settings by overriding one or more default settings, writing inside settings.json. Figure 5-5 shows an example where you can see how to change the theme, how to control white characters, how to control characters and breadcrumbs in the code editor, and how to enable the Minimap mode. Also, you will see how IntelliSense helps you choose

among available settings as you type. It is worth mentioning that you can search for settings by typing a search key in the **Search Settings** box. The settings editor will highlight occurrences of the search key as you type.

IntelliSense also allows you to get more information about a given settings by clicking the information icon, which shows hints about the setting with a convenient tooltip exactly as you would expect after learning about IntelliSense's features in Chapter 3, “Language Support and Code Editing Features.” When done, do not forget to save settings.json otherwise your changes will be lost.

A Real-World Example: Working with Proxies

If you work for an enterprise, the network might probably be behind a proxy server. In this case, you or the system administrator might need to configure Visual Studio Code to work with the proxy. If you do not, you will not be able to download packages, extensions, and product updates. Visual Studio Code should automatically detect proxies and ask for your credentials, but this does not always happen, so you might need some manual steps.

The first thing to do is making sure that the following sites are in the allowed applications list of the firewall:

- vscode-update.azurewebsites.net
- vscode.blob.core.windows.net
- marketplace.visualstudio.com
- *.gallerycdn.vsassets.io
- rink.hockeyapp.net
- vscode.search.windows.net
- raw.githubusercontent.com
- vsmarketplacebadge.apphb.com

CHAPTER 5 CUSTOMIZING VISUAL STUDIO CODE

The next step is configuring Code to work with the proxy. Actually, if the `http_proxy` and `https_proxy` environment variables have been defined at the system level, Visual Studio Code will use their values. If these variables have not been set, you must provide the proxy address in the user settings. In the settings editor, locate **Proxy** under the **Application** category. Then, as you can see in Figure 5-6, enter the proxy address in the **Proxy** text box.

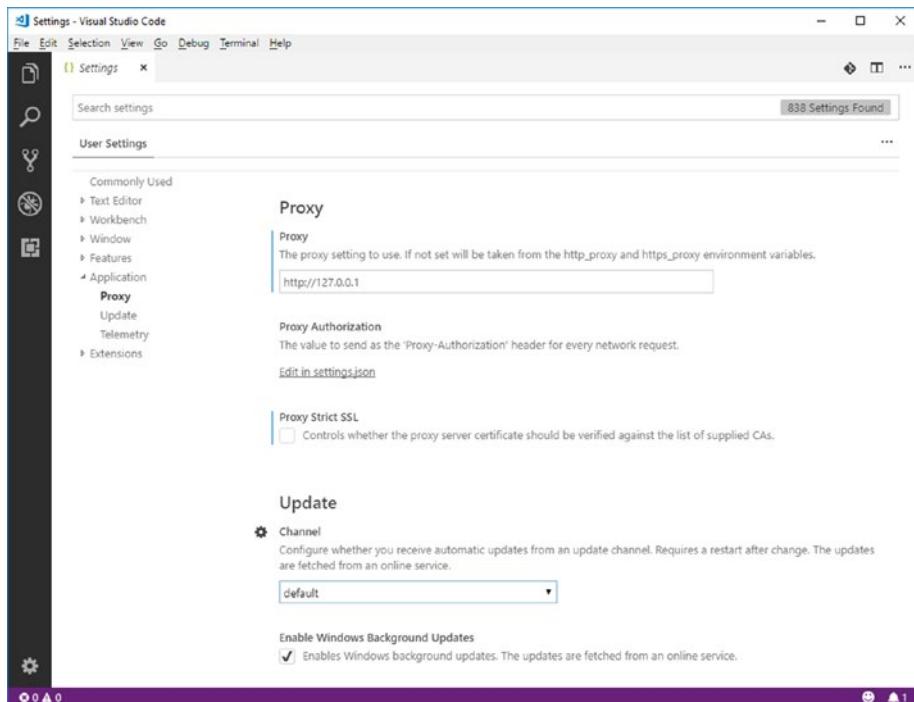


Figure 5-6. Configuring VS Code to work behind a proxy server

If your proxy also requires an authorization header, this must be specified in the `settings.json` file, so you have to click the **Edit in settings.json** hyperlink and then enter the value supplied by your network administrator as the value for `http.proxyAuthorization` key. Also, flag the **Proxy Strict SSL** option if the certificate should be verified against the list of supplied certification authorities.

Save your changes and try to see if Visual Studio Code is able to download extensions, packages and libraries required by some languages, and product updates. If you still encounter network issues, you should ask your network administrator to help you configure the proxy settings.

Note Some protection programs such as Symantec Endpoint Protection block some Visual Studio Code installation (and update) files because they are recognized as Cryptolocker virus instances. Obviously, these are false positives, but you might want to talk to your network administrator to review the protection rules for Visual Studio Code.

Understanding Workspace Settings

Differently from user settings, which globally apply to VS Code's environment, workspace settings only apply to the current folder. As an implication, you first need to open an existing folder in order to customize workspace settings.

Next you still select **File > Preferences > Settings**. At this point the settings editor will show two tabs, one for user settings and one for workspace settings, as demonstrated in Figure 5-7.

CHAPTER 5 CUSTOMIZING VISUAL STUDIO CODE

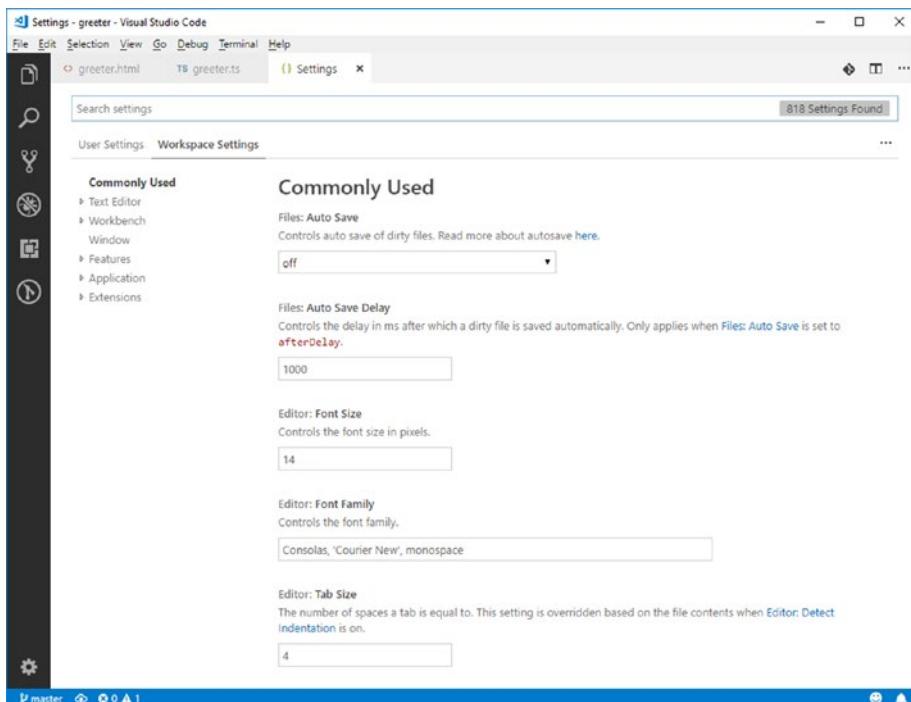


Figure 5-7. Customizing workspace settings

You customize workspace settings exactly as you do with user settings, so you have not only a second view in the settings editor but also another settings.json file where you can specify your preferences. The settings.json for workspace settings is saved under the .vscode subfolder that Visual Studio Code creates inside the opened folder, restricting settings availability to the current folder only.

Customizing Key Bindings

In the VS Code terminology, key bindings represent shortcuts you use to invoke commands and actions from the keyboard instead of using the mouse. Visual Studio Code includes a huge number of keyboard shortcuts that you can override with custom values. This is particularly useful if you used to work with other development tools and you want to get the same

keyboard shortcuts in Visual Studio Code. You will see how to accomplish this by downloading ready-to-use key binding extensions, but it's important for you to know how key bindings work.

Like user and workspace settings, key bindings are represented with JSON markup, and each is made of two elements: key, which stores one or more keys to be associated to an action, and command, which represents the action to invoke. In some cases, VS Code might offer the same shortcuts for different scenarios. This is the typical case of the escape key, which targets a number of actions depending on what you are working with, such as the code editor or a tool window. In order to identify the proper action, key binding settings support the **when** element, which specifies the proper action based on the context. You can quickly get the list of current key bindings by selecting **File > Preferences > Keyboard Shortcuts**. At this point, Visual Studio Code will display a nicely formatted list of commands and shortcuts, as you can see in Figure 5-8.

| Command | Keybinding | Source | When |
|------------------------------------|---------------------------|---------|---|
| Add Cursor Above | Ctrl + Alt + UpArrow | Default | editorTextFocus |
| Add Cursor Below | Ctrl + Alt + DownArrow | Default | editorTextFocus |
| Add Cursors to Line Ends | Shift + Alt + I | Default | editorTextFocus |
| Add Line Comment | Ctrl + K Ctrl + C | Default | editorTextFocus & !editor_READONLY |
| Add Selection To Next Find Match | Ctrl + D | Default | editorFocus |
| Back | Alt + LeftArrow | Default | inQuickOpen |
| Change All Occurrences | Ctrl + F2 | Default | editorTextFocus & !editor_READONLY |
| Change Language Mode | Ctrl + K M | Default | — |
| Clear | Ctrl + K | Default | terminalFocus |
| Clear Selection | Escape | Default | terminalFocus & terminalTextSelected & !terminalFindVisible |
| Close All Editor Groups | Ctrl + K Ctrl + Shift + W | Default | — |
| Close All Editors | Ctrl + K Ctrl + W | Default | — |
| Close Window | Ctrl + Shift + W | Default | — |
| Close Window | Ctrl + W | Default | !editorIsOpen & !multipleEditorGroups |
| Close Workspace | Ctrl + F | Default | — |
| Color Theme | Ctrl + K Ctrl + T | Default | — |
| Compare Active File with Clipboard | Ctrl + C | Default | — |
| Connect | Ctrl + Shift + C | Default | editorTextFocus & editorLangId == 'sql' |
| Continue | F5 | Default | inDebugMode |
| Copy | Ctrl + Insert | Default | textInputFocus |
| Copy | Ctrl + C | Default | textInputFocus |
| Copy Line Down | Shift + Alt + DownArrow | Default | editorTextFocus & !editor_READONLY |
| Copy Line Up | Shift + Alt + UpArrow | Default | editorTextFocus & !editor_READONLY |

Figure 5-8. The list of current key bindings

In order to customize key bindings, all you need to do is clicking the **keybindings.json** hyperlink under the search box and edit the keybindings.json file that Code generates for you. The code editor gets split into two views: on the left view, you can see the full list of default key bindings, whereas on the right view, you can override default shortcuts with custom ones (see Figure 5-7).

Note Remember that Visual Studio Code has (and allows for customizing) different default key bindings depending on what operating system it is running on.

You can quickly add a custom key binding by clicking the **Define Keybinding** button or use the shortcut suggested in the button text (which varies depending on your operating system). When you do this, a popup appears and asks you to specify the key binding, as shown in Figure 5-9.

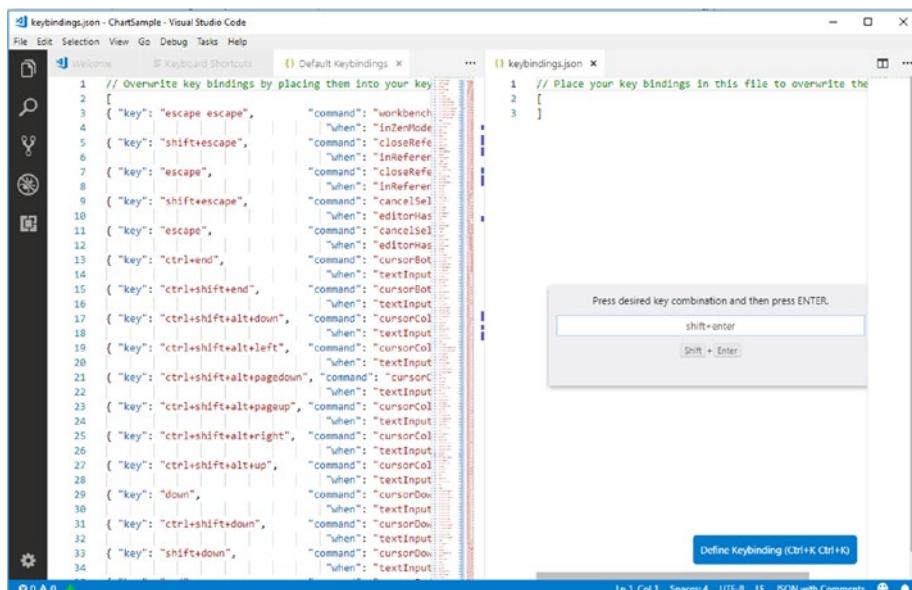
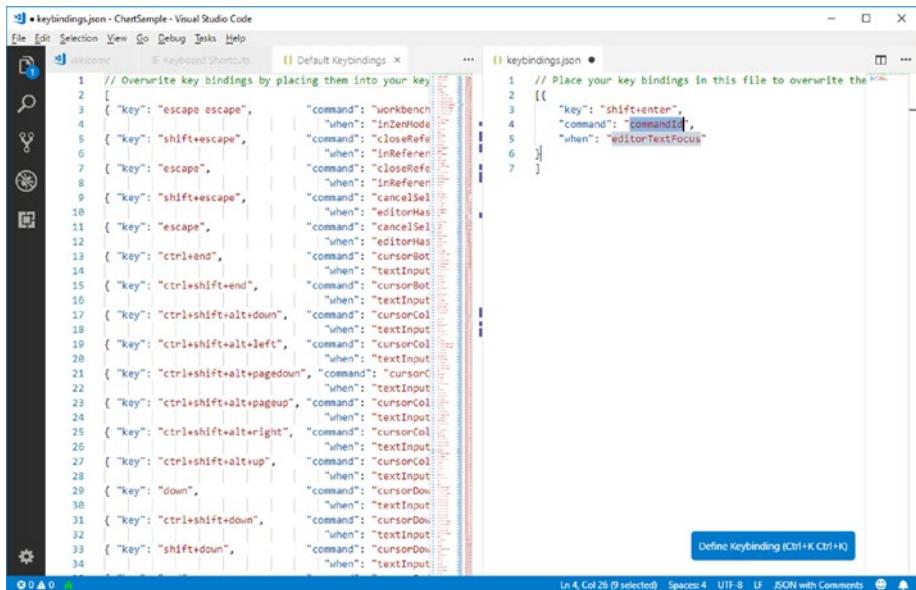


Figure 5-9. Adding a keyboard shortcut

When you press Enter, the JSON markup for the new keyboard shortcut is added, as shown in Figure 5-10.



```
// Place your key bindings in this file to overwrite the default key bindings.
[{"key": "shift+enter", "command": "commandId", "when": "editorTextFocus"}]
```

Figure 5-10. Editing the new keyboard shortcut

You will need to edit the command and when elements with the command you want to map and for which scenario. You can look at the original markup on the left to get them both. Actually, the when element is optional. Save your changes to the keybindings.json file to get your new keyboard shortcuts ready.

Summary

Visual Studio Code allows for several customizations that will help you feel at home especially if you used to work with other development tools or code editors. You can select a different color theme from a list, you can customize the environment settings globally or for a specific folder, and you can even create custom keyboard shortcuts.

But the very good news is that customizations can also be downloaded as extensions, as well as new languages, debuggers, and tools. Extensibility is discussed in the next chapter.

CHAPTER 6

Installing and Managing Extensions

Extensibility is one of the key features in Visual Studio Code, because you can add tools, languages, code snippets, debuggers, key bindings, and themes. Especially about languages, Visual Studio Code allows for extending the code editor with specific syntax support, which can also include IntelliSense, code snippets, and code refactoring.

This all means that Visual Studio Code has open support for any language and any tool on any platform, opening to infinite development scenarios. This chapter explains how to find and install extensions and how to manage extensions on your system.

Installing Extensions

You have two ways of browsing and installing extensions: from the Visual Studio Marketplace and from within Visual Studio Code. The Visual Studio Marketplace is a web site that contains extensions for the most popular Microsoft development tools and services, such as Visual Studio, Visual Studio Code, and Visual Studio Team Services. It is available at <https://marketplace.visualstudio.com>, and you will need to click the Visual Studio Code tab to see a list of extensions for Visual Studio Code. Figure 6-1 shows the Marketplace for Visual Studio Code.

CHAPTER 6 INSTALLING AND MANAGING EXTENSIONS

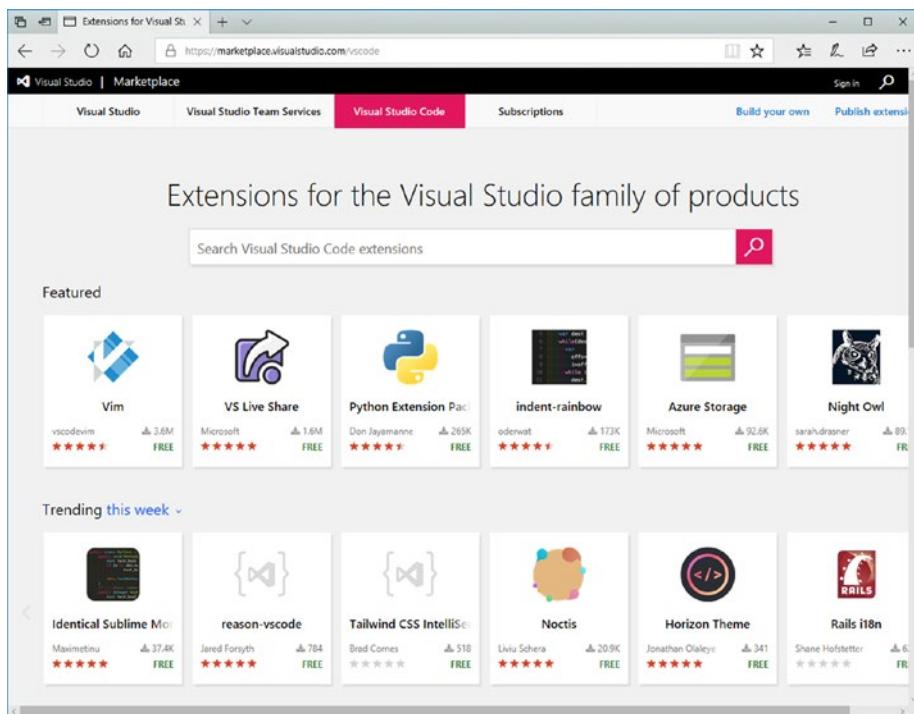


Figure 6-1. The Visual Studio Marketplace

You can search for extensions by typing in the search box, or you can use the groups below, such as Featured, Trending, Most Popular, and Recently Added. Once you have found an extension of your interest, click its name and you will see a detail page. Figure 6-2 shows an example based on the C# extension by Microsoft.

CHAPTER 6 INSTALLING AND MANAGING EXTENSIONS

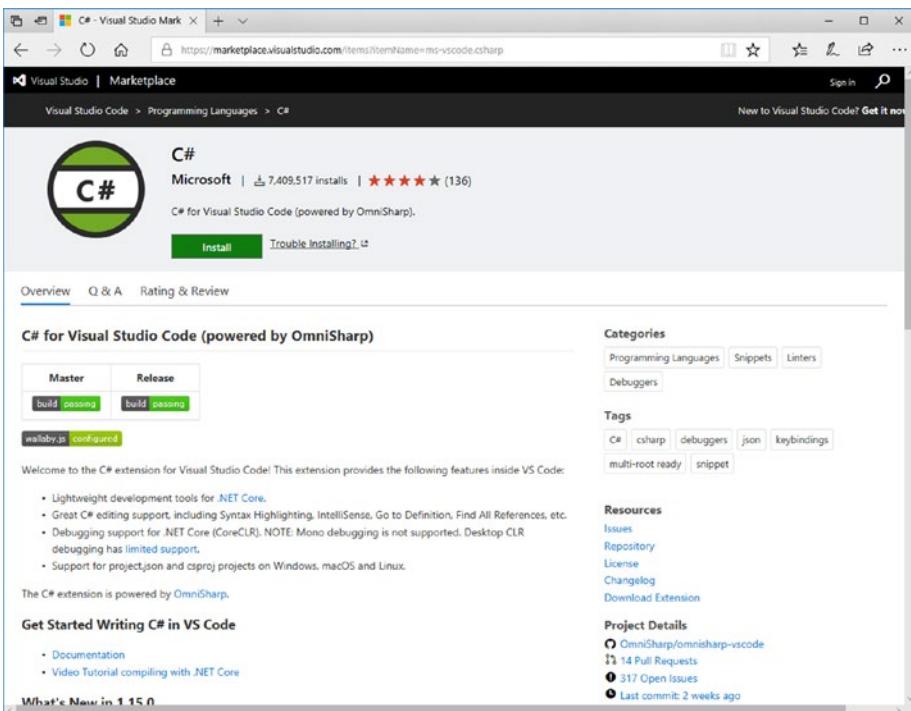


Figure 6-2. Detail page for an extension

An extension's page provides a detailed description and guidance about using the extension, often providing links to additional documentation, resources, and to the source code (if open source). I strongly recommend you to read the detail page to get information about what the extension includes, especially with extensions that add language support, because it is important to know if there is only support for a new syntax or also for IntelliSense, code snippets, and debugging.

If you click the **Install** button, the download link will be opened by Visual Studio Code for easy installation. You can also download the offline installer of the extension for later reuse. To do so, click the **Download Extension** hyperlink under the **Resources** group, on the right of the page. In this way you will be able to download a .vsix installer file that you can then launch manually.

Note If you have experience with the Microsoft Visual Studio development environment, you probably know that VSIX is the format used by Microsoft for extension installer files. However, the VSIX format for Visual Studio Code is not the same. Extensions for Visual Studio Code are packaged with a tool called **vsce** and cannot work with Visual Studio 2017 on Windows and with Visual Studio for Mac.

The second way of installing extensions is from within Visual Studio Code. You can open the Extensions bar and search for an extension, then you can click a specific extension to get the details, as shown in Figure 6-3.

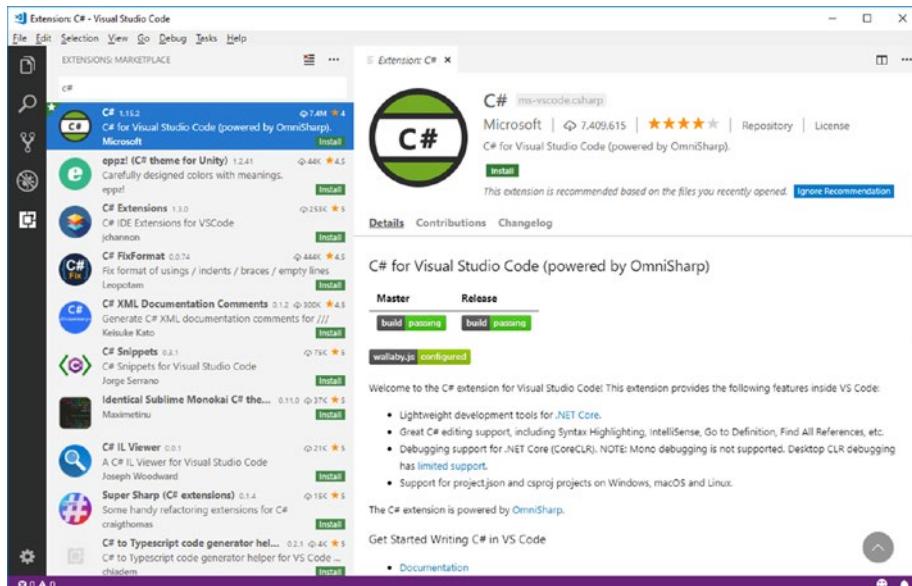


Figure 6-3. Installing extensions from within Visual Studio Code

You can click the **Install** button when ready. You will need to click the **Reload** button (that appears once the installation completes) to enable the extension in VS Code. You can also filter the search results; for instance, if you type category:linters in the search box, Visual Studio Code will list

all the extensions that provide linting support with syntax colorization to specific languages. You can use the same category names you see in the Visual Studio Marketplace.

As an alternative, you can use the Command Palette to download (and manage) extensions. You can open the Command Palette, type in ext, and a list of self-explanatory commands related to extension management will appear. You will typically prefer working with extensions from the Command Palette when you do not want to lose focus on the active editor window, otherwise using the Extensions bar's user interface is definitely easier.

Note Many extensions, especially extensions that provide full language support such as C# and C/C++, rely on additional tools like debuggers and libraries. These additional tools are usually downloaded the first time you use the extension. For example, in the case of the C# extension, required tools and libraries are downloaded the first time you create or open a C# file. Also, newly downloaded extensions might need some initial configuration. In this case, a popup will appear explaining what you need to do to get started.

Extension Recommendations

Visual Studio Code can provide suggestions about recommended extensions based on your activity. When you open the Extensions bar, you will see a group called **RECOMMENDED EXTENSIONS**, under the list of installed extensions.

The list of recommended extensions varies on your activity and might be empty the first times you work with Visual Studio Code. Not limited to this, Visual Studio Code can suggest extensions based on the file you open. For example, suppose you open a code file written with the Go language but you do not have installed any Go extension yet. Visual Studio Code has

CHAPTER 6 INSTALLING AND MANAGING EXTENSIONS

built-in support for the Go language syntax, so the editor provides syntax colorization and basic word completion, but you might want to work with a richer editing experience that includes code snippets, code navigation, and rich IntelliSense support. In this case, Code will suggest that an extension is available to help you work with Go files and will offer to install it, as represented in Figure 6-4.

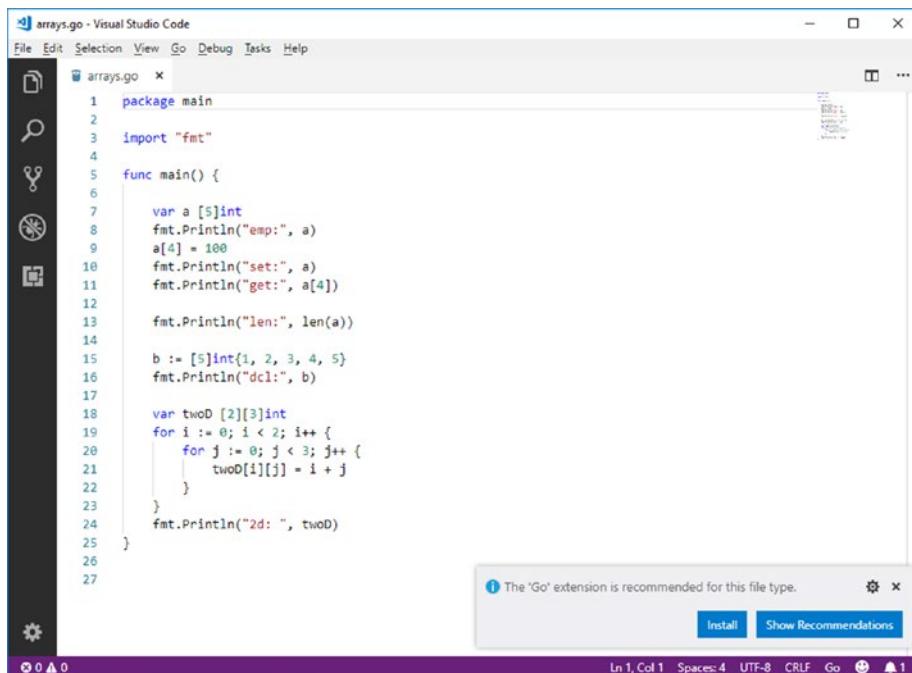


Figure 6-4. Extension recommendations based on the current file

You can click **Install** and Visual Studio Code will automatically install the extension that it thinks to be the most appropriate, or you can click **Show Recommendations** to see a list of possible extensions. In both cases, the Extensions bar will be opened and you will see the list of available recommended extensions, but when you click **Install**, the proposed extension will be already installing.

Useful Extensions

The Visual Studio Marketplace contains tons of useful extensions, but there is a set that I personally recommend after using Visual Studio Code for a long time in my daily job. Table 6-1 summarizes a list of useful extensions, with the description.

Table 6-1. Recommended Extensions for Visual Studio Code

| Name | Description | Type |
|---------------------------|---|-----------------------------|
| C# | C# full language support | Language, debugger, editing |
| C/C++ | C and C++ full language support | Language, debugger, editing |
| Python | Python full language support | Language, debugger, editing |
| Language Support for Java | Java full language support | Language, editing |
| Microsoft SQL Server | SQL Server support | Language, editing, tools |
| Debugger for Chrome | JavaScript debugging with the Chrome browser | Debugger |
| Debugger for Java | Java debugging support | Debugger |
| Debugger for Edge | JavaScript debugging with the Edge browser | Debugger |
| Cordova Tools | Mobile development with Apache Cordova | Editing, tools |
| Node Debug | Debug support for Node.js | Debugger |
| Visual Studio Keymap | Keyboard shortcuts based on Microsoft Visual Studio | Key binding |

(continued)

Table 6-1. (continued)

| Name | Description | Type |
|-----------------------------|---|--------------------------|
| Atom Keymap | Keyboard shortcuts based on Atom | Key binding |
| Notepad++ Keymap | Keyboard shortcuts based on Notepad++ | Key binding |
| Docker | Language support for Dockerfile | Language, editing, tools |
| vscode-icons | Colored icons for the Explorer bar | Tools |
| GitLens | Extend Git integrated features for Visual Studio Code | Tools |
| PowerShell | PowerShell scripting support | Language, editing, tools |
| Visual Studio Team Services | Integrated Git support for the Visual Studio Team Services platform | Tools |

As you work with Visual Studio Code on your projects and on the operating system of your choice, you will be able to find and fine-tune extensions that will help you be more productive.

Managing Extensions

The Extensions bar allows you to quickly manage extensions. It shows the list of installed extensions, as shown in Figure 6-5. Then, for each extension, the button with the gear icon opens a popup menu that contains commands for disabling or uninstalling an extension.

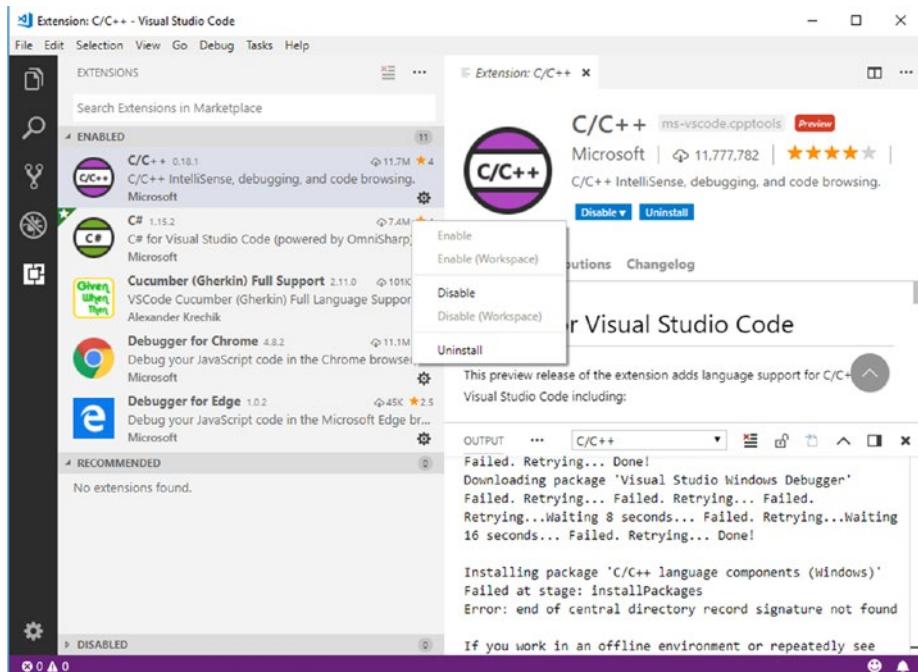


Figure 6-5. Shortcuts for extension management

You can also click an extension name, and the detail page will show the **Disable** and **Uninstall** buttons. Notice that every time you disable or uninstall an extension, you will need to click a button called **Reload** (that appears when the extension has been disabled or uninstalled) to refresh the development environment. It is worth mentioning that you can change the default view of the Extensions bar (displaying the list of installed extensions) by clicking the ... button at the top of the **EXTENSIONS** group. When you click this button, a popup menu appears showing different options, such as viewing popular extensions, as well as commands for searching extension updates and installing extensions from .vsix files.

Note Shortcuts for extension management are also available in the Command Palette.

Configuring Extensions

Visual Studio Code has some options that allow you to control the global behavior of extensions. You can see these options in the user settings, under the **Extensions** group, as shown in Figure 6-6.

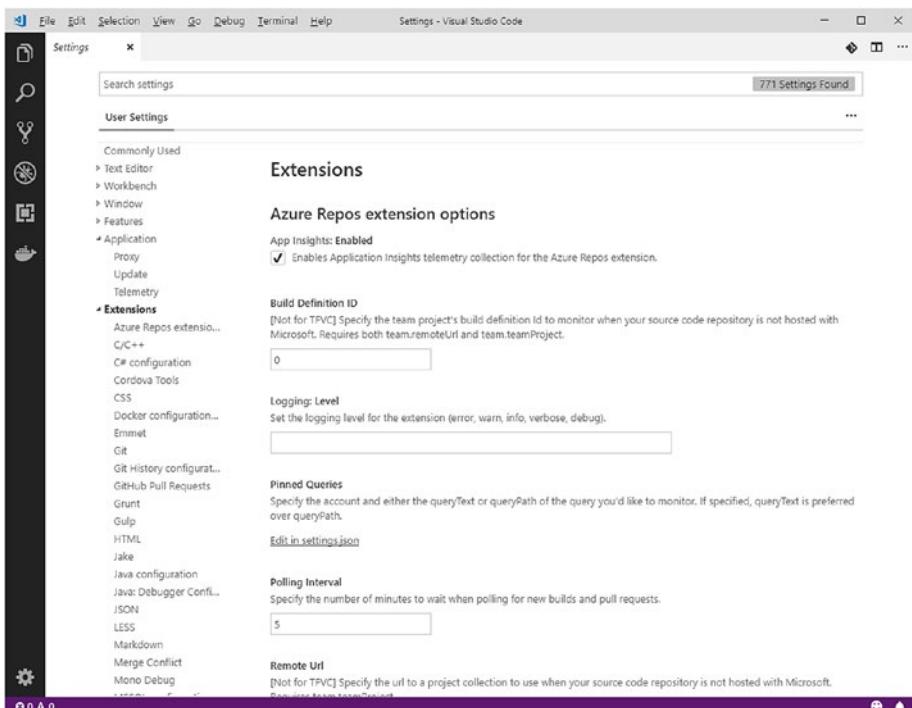


Figure 6-6. Customizing options about extension management

There are detailed comments that explain what each option is about. Additionally, each extension might allow for customizing its own behavior in the user settings. For instance, suppose you have the C# extension installed. If you look in the user settings, you will find a group called C# Configuration. If you expand this group, you will see the full list of options about the C# extension, which include options for code editing and for tools the extensions add. Figure 6-7 shows these options.

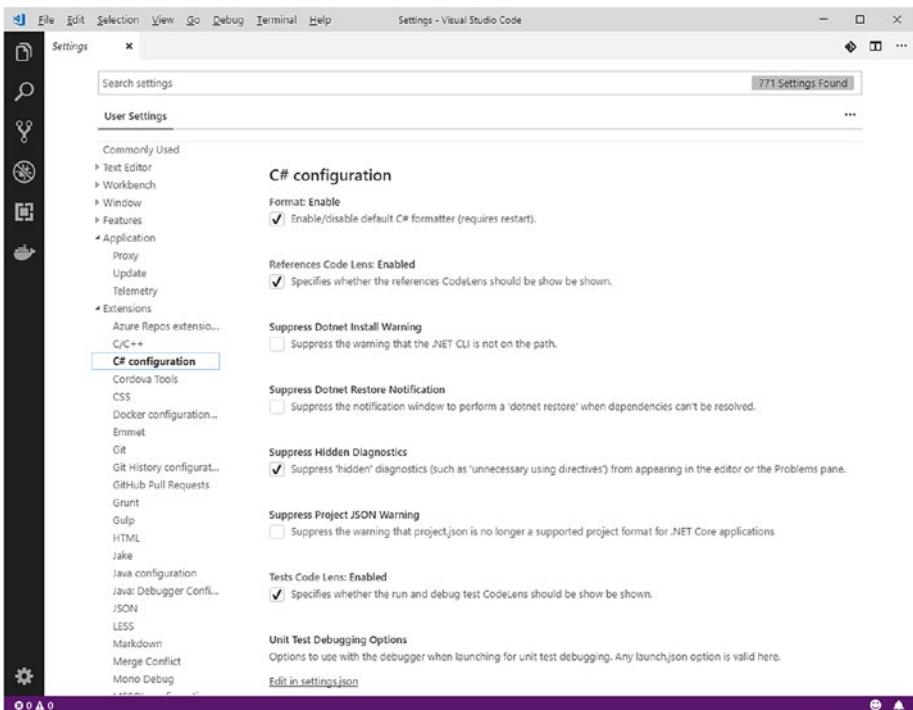


Figure 6-7. Customizing extension options

Normally, extension authors provide detailed comments that explain what an option is about so that it is easier for you to fine-tune an extension behavior, such as in the case of the C# extension.

Hints About Extension Authoring

You can build extensions for Visual Studio Code and share them through the Marketplace. You can basically build any type of supported extension, such as language support, editing features, themes, code snippets, debugger adapters, and key bindings. You will also need to register as a publisher on the Visual Studio Marketplace, which requires you to have a Microsoft account.

Extensions are usually written with TypeScript and, for most of them, you can use an extension generator such as the Yeoman tool on Node.js. As you can imagine, extension authoring is a complex task, and it is out of scope in a book from the Distilled series. If you are interested in extension authoring, you can walk through the official documentation (<https://code.visualstudio.com/docs/extensions/overview>) which provides examples and guidance for many scenarios.

Summary

Extensibility is a key feature in Visual Studio Code, because it allows adding power to the development environment. Extensions can add new languages (with or without rich editing support), debuggers, keyboard shortcuts, themes, code snippets, and tools. You can install extensions from the Visual Studio Marketplace or from within Visual Studio Code, through the Extensions bar or the Command Palette.

Visual Studio Code can also provide extension recommendations based on the context, for example, when you open a file written in a language for which there is no built-in support. Visual Studio Code makes it also simple to manage extensions with shortcuts to disable and uninstall extensions but also with configuring extensions' behavior via the user settings file. In the next chapter, you will see how to leverage extensions to add features to Visual Studio Code to another core feature that makes it a step forward compared to its competitors: version control with Git.

CHAPTER 7

Source Control with Git

Writing software often involves collaboration. This is true if you are part of a development team but also if you are involved in open source projects, or if you are an individual developer who has interactions with customers. Microsoft strongly supports both collaboration and open source, so Visual Studio Code provides an integrated source control system based on Git and that can be extended to other providers.

This chapter describes all the integrated tools for collaboration over source code from within Visual Studio Code that are available out of the box, but also how to use extensions that you will find very useful in the real life to better review your code and to push your work to Visual Studio Team Services.

Source Control in Visual Studio Code

Visual Studio Code supports different source control providers via extensibility, but it offers integrated support for Git. Git (<https://git-scm.com/>) is a very popular distributed, cross-platform version control engine that makes collaboration easier for small and large projects. One of the reasons for its popularity is that Git is open source, and therefore it has always been loved by large open source communities.

Visual Studio Code works with any Git repository, such as GitHub or Visual Studio Team Services, and provides an integrated way to manage your code commits.

Notice that this chapter is not a guide to Git; rather it is a place to learn how Visual Studio Code works with it, so for further information, visit the Git official page. Also, remember that Visual Studio Code requires the Git engine to be installed locally, so make sure it is available on your machine or download it from <https://git-scm.com/downloads>. In order to demonstrate how Git version control works with Visual Studio Code, I will use a small TypeScript project called Greeter, available in the TypeScript Samples repository from Microsoft (<https://github.com/Microsoft/TypeScriptSamples>). You can download the repository on your system and extract the Greeter subfolder on your disk. Obviously, you are totally free to use another example or another project of your choice, regardless of the language. At this point, open the project in Visual Studio Code to start collaborating over the source code.

Downloading Other Source Control Providers

As I mentioned earlier, VS Code supports additional source control managers, also referred to as SCM, via extensibility. You can open the Extensions bar and type `SCM providers` in the search box in order to find third-party extensions that target other source control engines. Figure 7-1 shows an example, where you can see how an extension that adds support for the Subversion engine (<https://subversion.apache.org>) has been selected.

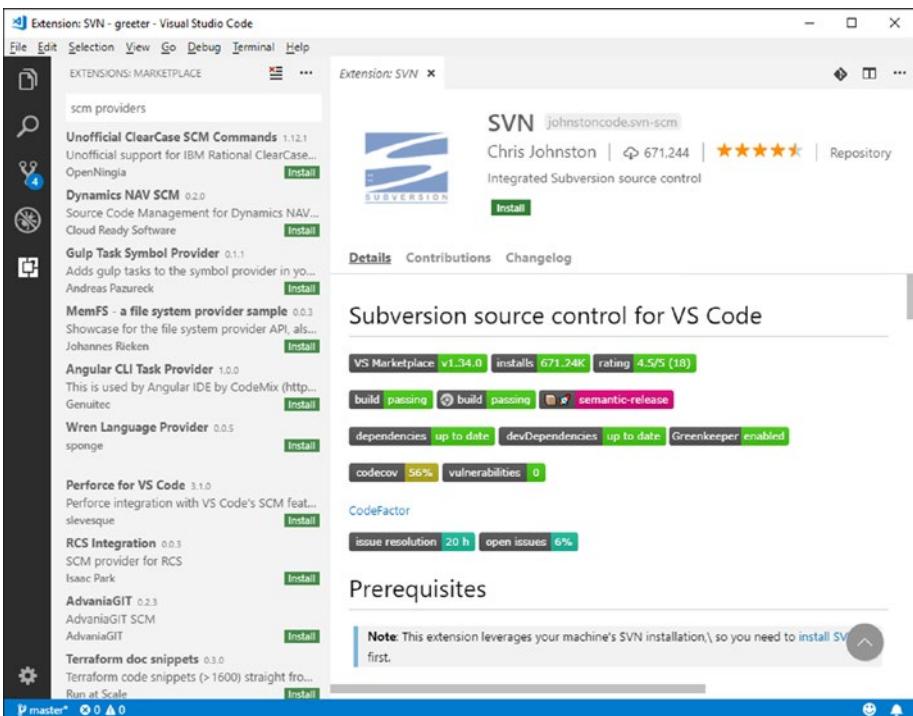


Figure 7-1. Installing additional source control providers

Because VS Code provides in-the-box support only for Git, other source control providers will not be discussed in this chapter. If you wish to install SCM extensions, make sure you refer to the documentation provided by the producer.

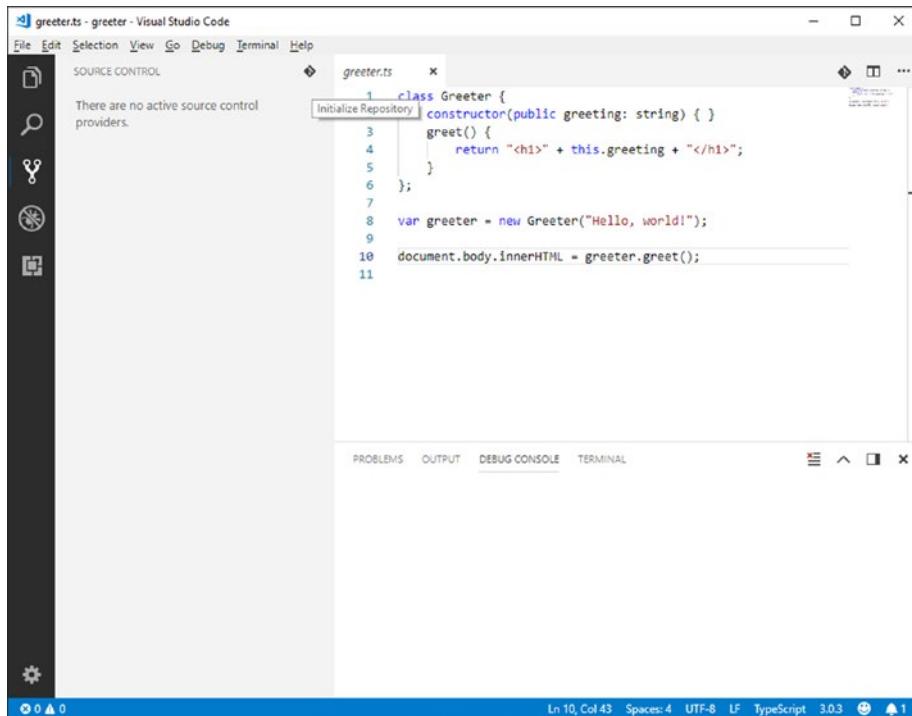
Managing Repositories

With Git, version control requires both a local and a remote repository to work. This section explains how to create both, supplying information that you will not find in the documentation especially for remote repositories.

Note A very popular abbreviation for repository is *repo*. This will not be used in this book, but you will find it very often especially when searching for information about open source projects.

Initializing a Local Git Repository

The first thing you need to do is creating a local repository for the current project. This is accomplished by opening the Git tool from the side bar, as shown in Figure 7-2.



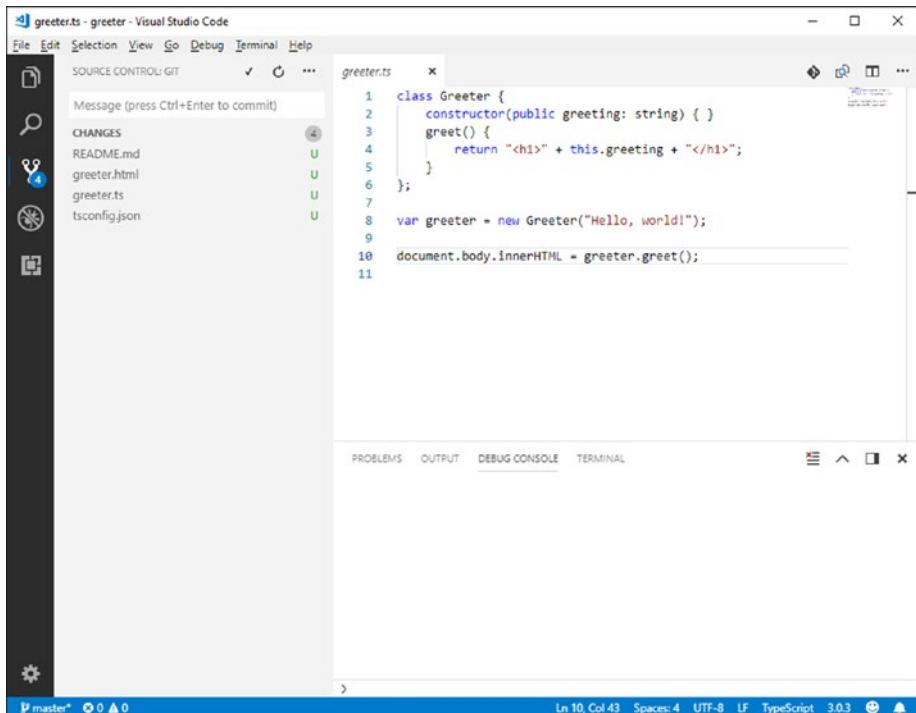
The screenshot shows the Visual Studio Code interface. The title bar reads "greeter.ts - greeter - Visual Studio Code". The menu bar includes File, Edit, Selection, View, Go, Debug, Terminal, and Help. The left sidebar has icons for File Explorer, Search, and Source Control. The Source Control icon is highlighted, and a tooltip says "There are no active source control providers." The main editor area contains the following TypeScript code:

```
1 class Greeter {
2     constructor(public greeting: string) { }
3     greet() {
4         return "<h1>" + this.greeting + "</h1>";
5     }
6 }
7
8 var greeter = new Greeter("Hello, world!");
9
10 document.body.innerHTML = greeter.greet();
11
```

The bottom status bar shows "Ln 10, Col 43 Spaces:4 UTF-8 LF TypeScript 3.0.3" and a file count of "1".

Figure 7-2. Ready to initialize a local Git repository

Click the **Initialize repository** button at the top (see Figure 7-2). Visual Studio Code will initialize the local repository and will show the list of files that now are under version control but not committed yet (see Figure 7-3).



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

- Title Bar:** greeter.ts - greeter - Visual Studio Code
- File Menu:** File Edit Selection View Go Debug Terminal Help
- SOURCE CONTROL: GIT:** Message (press Ctrl+Enter to commit)
- Changes List:** CHANGES
README.md (U)
greeter.html (U)
greeter.ts (U)
tsconfig.json (U)
- Editor:** greeter.ts (Content shown below)
- Bottom Status Bar:** master* 0 0 0 0 Ln 10, Col 43 Spaces: 4 UTF-8 LF TypeScript 3.0.3

```
1 class Greeter {  
2     constructor(public greeting: string) {}  
3     greet() {  
4         return "<h1>" + this.greeting + "</h1>";  
5     }  
6 };  
7  
8 var greeter = new Greeter("Hello, world!");  
9  
10 document.body.innerHTML = greeter.greet();  
11
```

Figure 7-3. Files are under version control but not committed yet

Notice how the Git icon shows the number of pending changes. This is an important indicator that you will always see any time you have pending, uncommitted changes. Write a commit description and then press **Ctrl+Enter**. At this point, files are committed to the local repository, and the list of pending changes will be cleaned. Now there is a problem: you need a remote repository, but the official documentation does not describe how to associate one to Code. Let's see how to accomplish this too.

Creating a Remote Repository

Visual Studio Code works with any Git repository. There are plenty of platforms that use Git as the version control engine, but probably the most popular platforms are GitHub, Atlassian Bitbucket, and Microsoft Visual Studio Team Services. In this section, you will see how to create a remote repository on GitHub. This requires you to have an existing GitHub account, otherwise you can create one for free at <https://github.com/join>. Once signed in, create a new repository. Figure 7-4 shows an example for a new repository called GreeterDistilled.

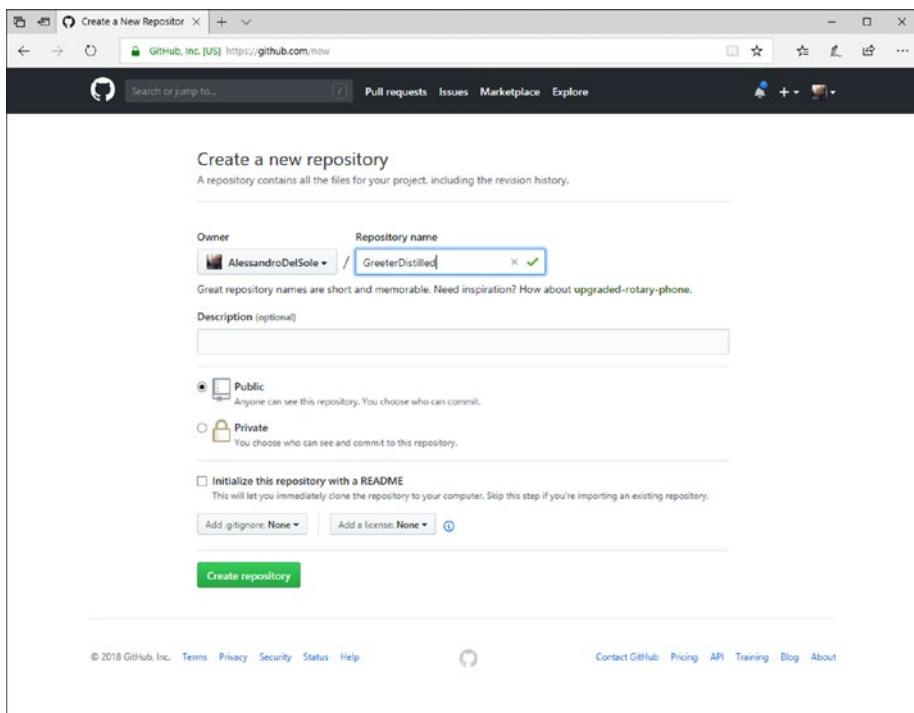


Figure 7-4. Creating a remote repository

Once the repository is ready, GitHub provides fundamental information you need to associate the remote repository with the local one. Figure 7-5 shows the remote address for the Git version control engine and the command lines you have to type to perform the association.

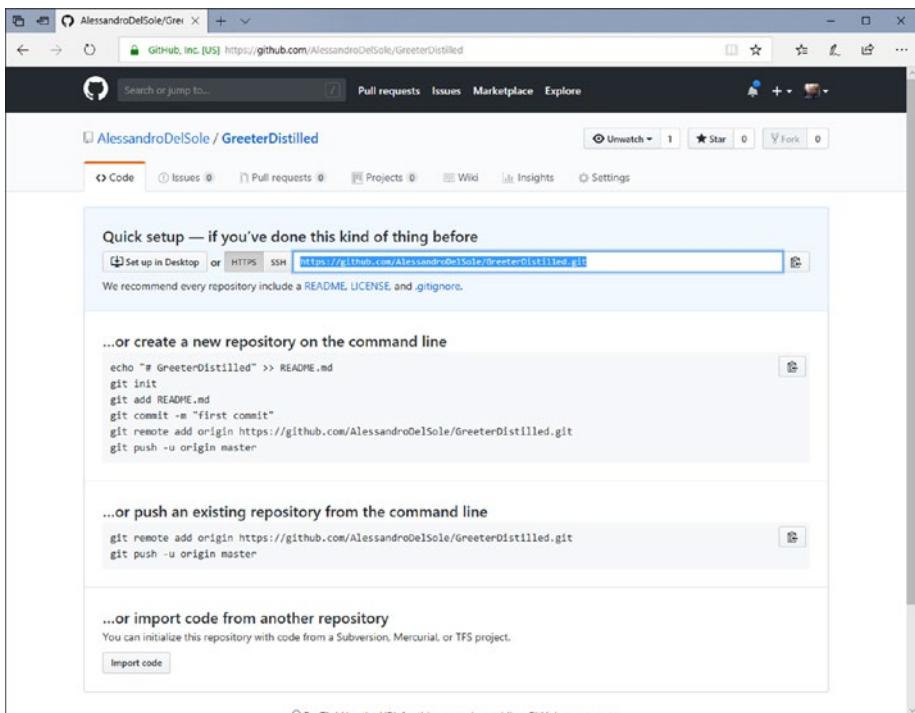


Figure 7-5. The information you need to push the local repository

The next step is associating the local repository to the remote one by typing some Git commands. This can be accomplished directly within VS Code, through the Terminal (**Terminal** ▶ **New Terminal**). When the Terminal is ready, type the following command:

```
> git remote add origin https://github.com/YourAccount/
YourRepository.git
```

where YourAccount represents your GitHub account and YourRepository represents the name of your repository, such as GreeterDistilled in the current example. This command associates the local repository with the remote repository. The next command you have to type is the following:

```
> git push -u origin master
```

This command makes a first synchronization between the local and remote repositories, uploading files to a default branch called master. Now you really have everything you need and you can start discovering the Git integration that Visual Studio Code offers.

Handling File Changes

Git locally tracks changes on your code files, and the Git icon in VS Code shows the number of files with pending changes. This number is actually updated only after you save your files. In Visual Studio Code, handling file changes is very straightforward. In Figure 7-6 you can see how the number of changes is highlighted in the Git icon but also how files that have changes are marked with a brown M (where M stands for Modified), whereas deleted files are marked with a red D (where D stands for Deleted).

The screenshot shows the Visual Studio Code interface with the title bar "greeter.ts - greeter - Visual Studio Code". The menu bar includes File, Edit, Selection, View, Go, Debug, Terminal, Help. The left sidebar has icons for file operations like Open, Save, Find, and Diff. The main area shows a code editor with "greeter.ts" open, containing TypeScript code for a Greeter class. Below the code editor is a status bar with "Ln 10, Col 28 Spaces: 4 UTT-8 LF TypeScript 3.0.3". On the far left, the Source Control panel is visible, titled "CHANGES". It lists three files: "README.md" (D), "greeter.html" (M), and "TS greeter.ts" (M). A message at the top of the panel says "Message (press Ctrl+Enter to commit)". The bottom right corner of the screen shows the page number "133".

Figure 7-6. Getting the number of pending changes

By clicking a file in the list, you can see the differences between the current and previous versions of the file with the **Diff** tool. Figure 7-7 shows an example.

```

greeter.ts (Working Tree) - greeter - Visual Studio Code
File Edit Selection View Go Debug Terminal Help
greeter.html greeter.ts (Working Tree) x greeter.ts
1 class Greeter {
2   constructor(public greeting: string) {
3     greet() {
4       return "<h1>" + this.greeting + "</h1>";
5     }
6   };
7
8 -var greeter = new Greeter("Hello, world!");
9
10 document.body.innerHTML = greeter.greet();
11
1 class Greeter {
2   constructor(public greeting: string) {
3     greet() {
4       return "<h1>" + this.greeting + "</h1>";
5     }
6   };
7
8+var greeter = new Greeter("Hello, Code!");
9
10 document.body.innerHTML = greeter.greet();
11

```

master* C 0 0 ▲ 0 Ln 8, Col 1 Spaces: 4 UTF-8 LF TypeScript 3.0.3

Figure 7-7. Comparing file versions with the Diff tool

On the left side, you have the old version, while the new one is on the right. The line highlighted in red represents code that has been removed, whereas the line highlighted in green represents new code. This is a very important tool when working with any version control engine.

Staging Changes

You can promote files for staging, which means marking them as ready for the next commit. This is actually not mandatory, as you can commit directly, but it is useful to have a visual representation of your changes. You can stage a file by simply clicking the + symbol near its name, or you can stage all files by right-clicking the **CHANGES** title and then select **Stage All Changes**. Visual Studio Code organizes staged files into a logical container, as you can see in Figure 7-8. Similarly, you can unstage files by clicking the - symbol.

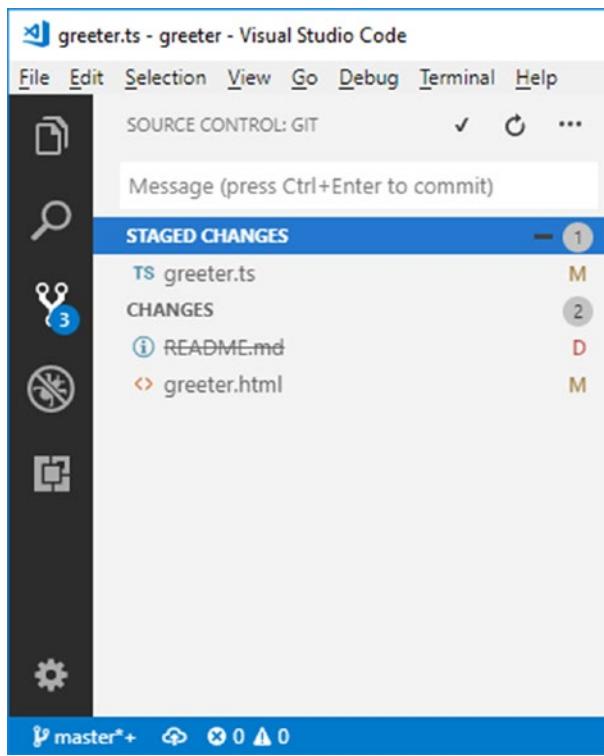


Figure 7-8. The view of staged and unstaged changes

The workflow based on staging is very convenient, because if you no longer want to commit a file, you can simply unstage it before the code goes to the server.

Managing Commits

The ... button provides access to additional actions, such as **Commit**, **Sync**, **Pull**, **Stash**, and **Pull (Rebase)**. Figure 7-9 shows the full list of built-in Git synchronization commands available in VS Code.

CHAPTER 7 SOURCE CONTROL WITH GIT

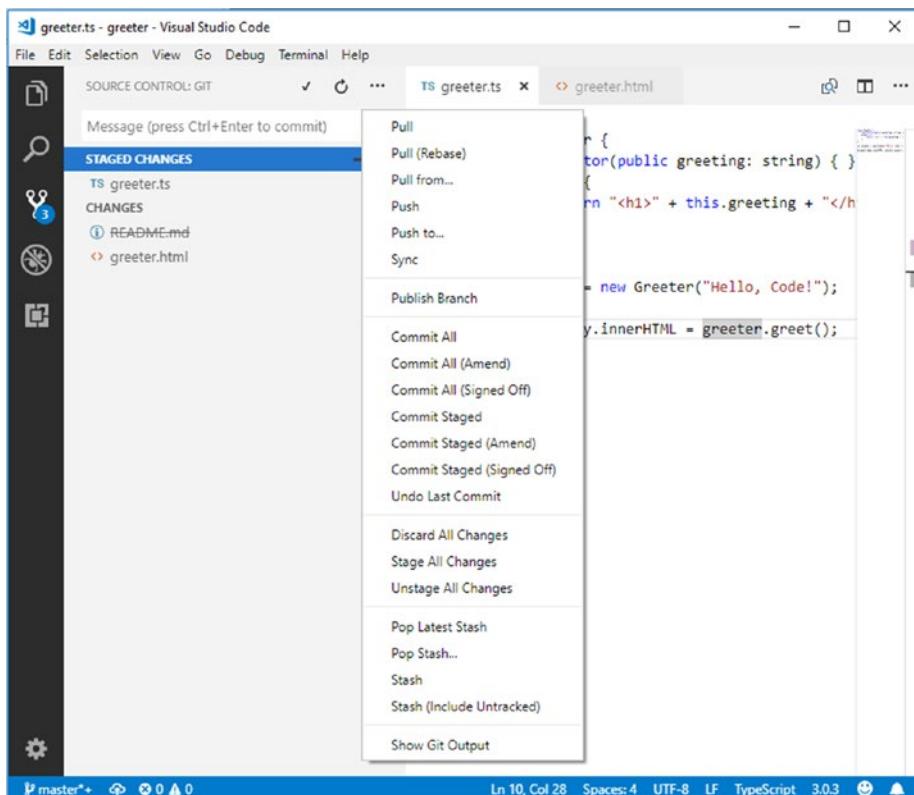
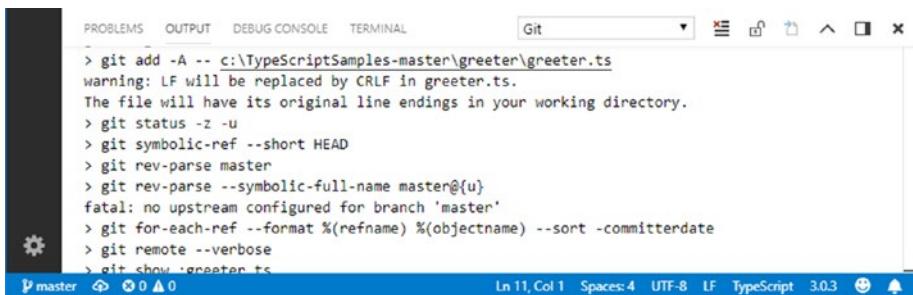


Figure 7-9. Shortcuts to commit and synchronize changes

When you are satisfied with your work on the source code, you can select the **Commit All** command to commit your changes. Remember that this action commits files to the local repository. You have to use the **Push** command in order to send changes to the remote repository. You also have an option to undo the last commit and revert to the previous version with the **Undo Last Commit** command. **Pull** and **Pull (Rebase)** allow to merge a branch into another branch; Pull actually is nondestructive and merges the history of the two branches, while Pull (Rebase) rewrites the project history by creating new commits for each commit in the original branch. The **Sync** command performs a **Pull** first and then a **Push** operation, so that both the local and remote repositories are synchronized. There is

also a command called Stash, which allows for storing modified tracked changes and staged changes in a cache, so that you can switch to another branch while having unfinished work on the current branch. Then, with the **Pop Latest Stash** and **Pop Stash** commands, you can retake the latest version of your unfinished work and a specific version of the unfinished work, respectively. Every time you work with Git commands, such as Commit and Push, Visual Studio Code redirects the output of the Git command line to the Output panel. Figure 7-10 shows an example.



The screenshot shows the Visual Studio Code interface with the Output panel open. The tab bar at the top includes PROBLEMS, OUTPUT, DEBUG CONSOLE, and TERMINAL. The OUTPUT tab is selected, showing a dropdown menu with 'Git' highlighted. The main area of the Output panel displays the following Git command history:

```

> git add -A -- c:\TypeScriptSamples-master\greeter\greeter.ts
warning: LF will be replaced by CRLF in greeter.ts.
The file will have its original line endings in your working directory.
> git status -z -u
> git symbolic-ref --short HEAD
> git rev-parse master
> git rev-parse --symbolic-full-name master@{u}
fatal: no upstream configured for branch 'master'
> git for-each-ref --format %(refname) %(objectname) --sort -committerdate
> git remote --verbose
> git show :greeter.ts

```

The status bar at the bottom indicates the current file is 'greeter.ts', with 'Ln 11, Col 1', 'Spaces: 4', 'UTF-8', 'LF', 'TypeScript 3.0.3', and other icons.

Figure 7-10. Messages from the Git command line are shown in the Output panel

You will need to select **Git** from the dropdown in the Output panel in order to see the Git output. You can also open the Output panel using the **Show Git Output** command from the popup menu as you can see in Figure 7-9.

Working with the Git Command Line Interface

The Command Palette has support for specific Git commands which you can type as if you were in a command line terminal. Figure 7-11 shows the list of available Git commands, which you can see by typing **Git** in the Command Palette. The list is quite long and cannot be totally included in Figure 7-11, but you can scroll it to see all available commands.

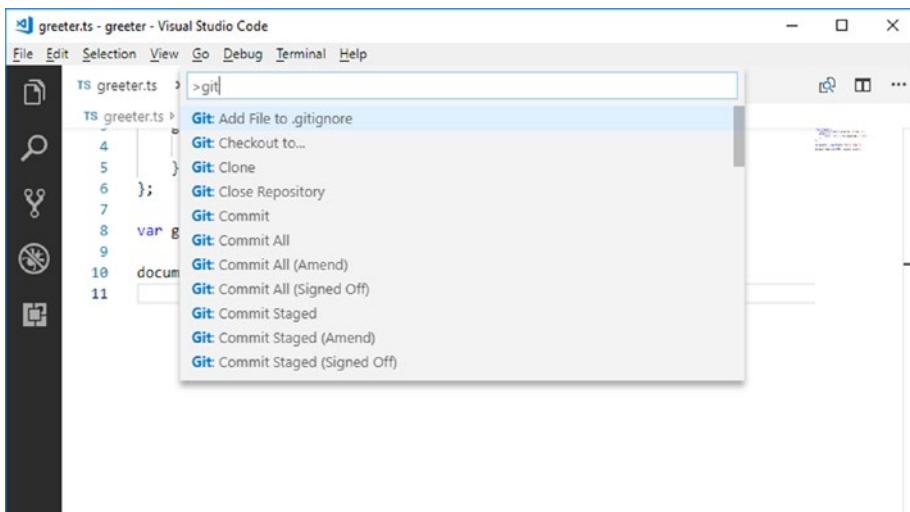


Figure 7-11. Supported Git commands in the Command Palette

For instance, you can use `Git Sync` to synchronize the local and remote repositories, or you can use `Git Push` to send pending changes to the remote repository. A common scenario in which you use Git commands is with branches.

Creating and Managing Branches

For a better understanding of what a branch is, suppose you have a project that, at a certain point of its lifecycle, goes to production. You need to continue the development of your project, but you do not want to do it over the code you have written so far.

You can create two histories by using a branch. When you create a repository, you also get a default branch called **master**. Continuing with the example, the master branch could contain the code that has gone to production, and now you can create a new branch, such as **development**, based on master but different from it. In Visual Studio Code, you have different options to create a new branch. The first option allows creating a

branch from the Command Palette, where you can type `Git branch`. Then select the **Git: Create branch** option, and you will be asked to specify a new branch name, such as `development`. This will create a new branch locally, based on `master`. The second option is clicking the current branch name in the status bar (`master` in this case) and then click the Create new branch command (see Figure 7-12).

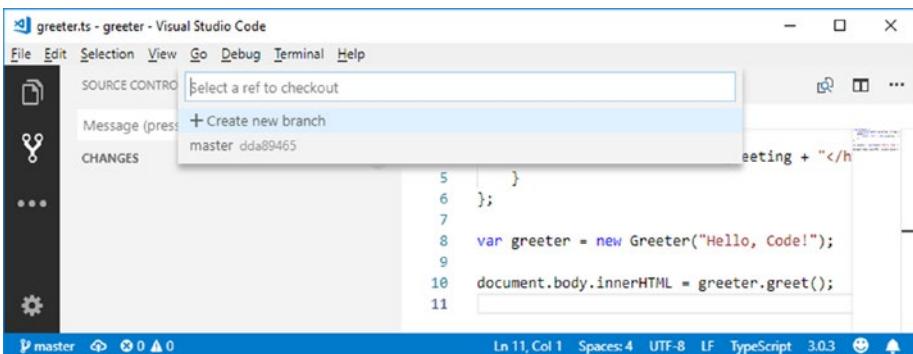


Figure 7-12. Creating a branch

You will need to enter the new branch name, and then press Enter. When a new branch is created, the status bar shows it as the active branch; when you are ready, you can publish the new branch to the remote repository with the **Publish Changes** button, represented by the cloud icon (see Figure 7-13).



Figure 7-13. The new branch is set as active and ready to be published

Switching to a Different Branch

Switching to a different branch is very easy. Simply click the name of the active branch, and VS Code will display the list of branches, as shown in Figure 7-14.

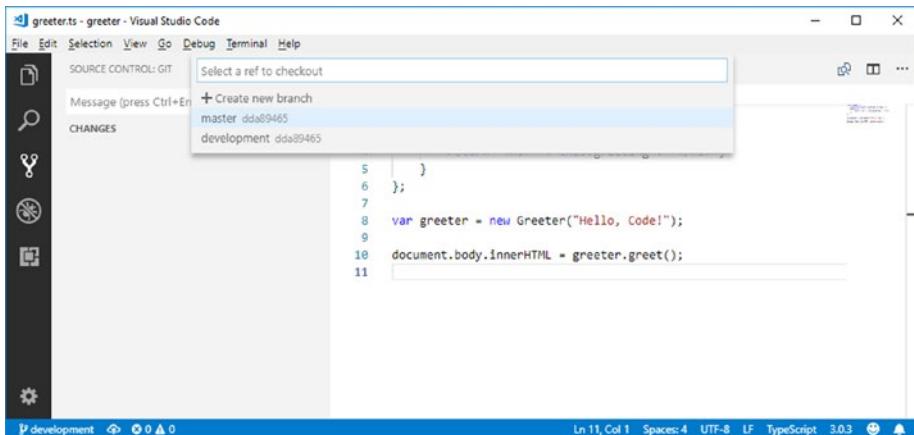


Figure 7-14. Selecting a different branch

Click the desired branch, and VS Code will check it out and set it as the active branch.

Merging from a Branch

Suppose you have completed and tested some work on the development branch and you want this work to be published to production. Because the production code is on the master branch, you must bring all the work from the development branch to the master branch. This is a merge operation. You can merge from a branch into another one via the Command Palette, using the **Git: Merge Branch** command. VS Code will show the list of branches, and you will need to select the branch you want to merge from into the current branch (see Figure 7-15).

Note Remember that the branch that receives the merge is the active branch, so make sure you have switched to the proper branch before starting a merge operation.

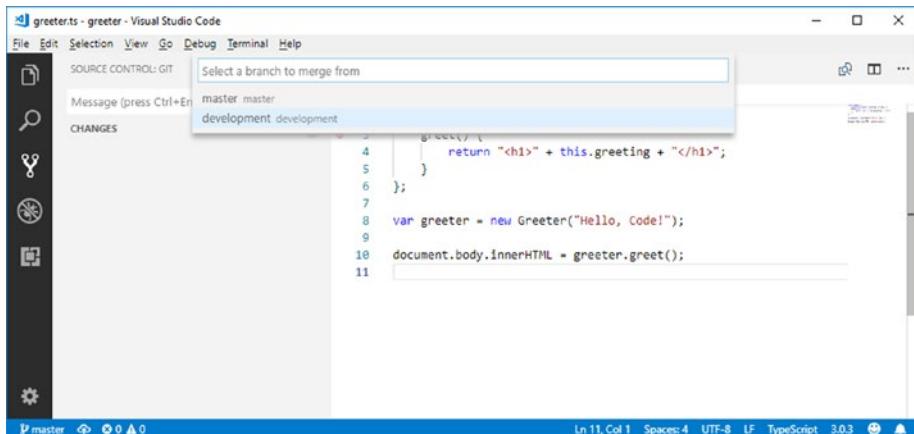


Figure 7-15. Merging from a branch

Once the merge operation is completed, remember to push your changes to the remote repository.

Deleting Branches

Deleting a branch is not very common, because branches help keep the history of the source code, but sometimes you might have branches that have been created only for testing some code and that are not really necessary in the application lifecycle management. In this case, in the Command Palette, you can use the **Git: Delete Branch** command.

With a user interface like what you see in Figure 7-15, VS Code shows the list of branches. Select the branch you want to delete and press Enter. Remember that the active branch cannot be deleted, and you first need to switch to a different branch.

Adding Power to the Git Tooling with Extensions

The integrated tools for Git cover all the needs that you, as a developer, can have when working with local and remote repositories to manage your source code, but there are extensions that provide additional power to the integrated tools.

This section describes the most useful free extensions that will improve your collaboration experience in Visual Studio Code.

Git History

Git History is a free extension that adds the option to get a very detailed view of the history of your source code, such as information and author about each commit and that can display how a file has gone through branches; plus it adds commands that make it easier to manage your code against Git. Assuming you have installed the extension, for example, you can right-click a file and select **Git: View File History**.

Figure 7-16 shows an example based on a file that has three commits. If available, the view shows the branches where the file has been included, comments and author for the commit, and the commit ID, and it allows for searching and filtering contents by branch and author.

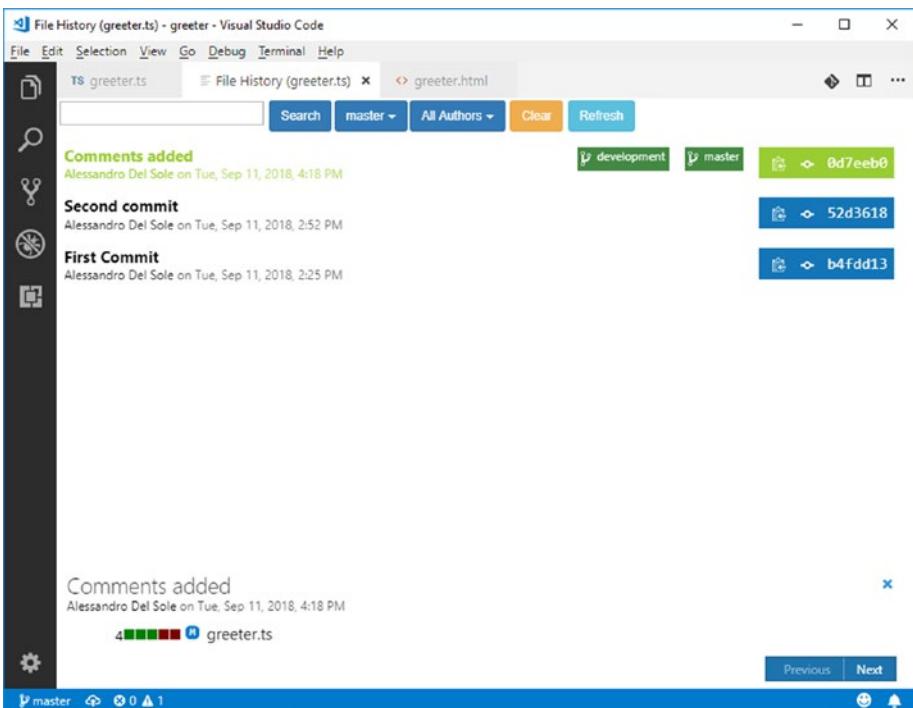


Figure 7-16. Viewing the history of commits with Git History

Note If the commit authors have associated a picture to the Git credentials, Git History will show the picture near the author name.

If you click the icon at the left of the commit ID, a menu will appear showing a number of very useful commands that will make it easier to work with commits (see Figure 7-17).

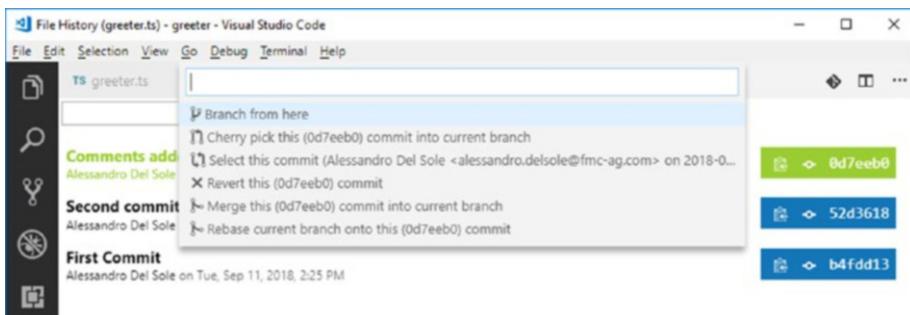


Figure 7-17. *Git History provides commands that make it easier to work with commits*

At the bottom of the view, you will see the list of files involved in the selected commit. If you click a file name, another menu will appear providing shortcuts to compare the file with the previous version and to view the history of that file. Git History is a very useful extension especially when your team works with the Agile methodologies, and for each task in the backlog, a new branch is created and then merged into one branch at the end of the sprint, making it easier to walk through the history of the work.

GitLens

Another extremely useful that will boost your productivity is called **GitLens**. GitLens adds many features and commands to Visual Studio Code about Git. For example, it adds a new bar called **GITLENS** (see Figure 7-18) that you enable by clicking the GitLens icon in the side bar (typically the last one, below the Extensions icon).

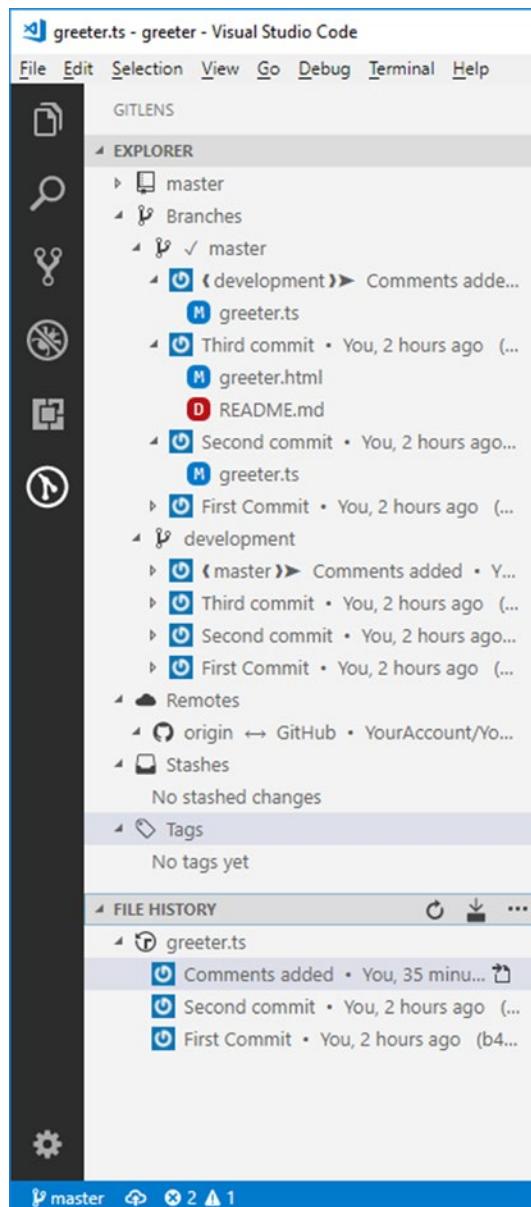


Figure 7-18. The GitLens bar with the Explorer and File History

The GitLens bar is divided into two areas: **EXPLORER** and **FILE HISTORY**. EXPLORER shows the list of both local and remote branches, and, for each branch, it displays the list of commits. For each commit, it displays the commit message, the list of files involved in the commit, and an icon that represents the operation made on the file, such as M for Modified and D for Deleted. Not limited to this, it also shows stashed changes (if any). The FILE HISTORY area shows the list of commits for a file, once you click it in the EXPLORER. For each commit, you can see the name, the author, and the time of last edit. The status bar in VS Code now provides, with GitLens, a field containing the current commit's author name and time of last edit. If you click this information, VS Code will show a list of commands as shown in Figure 7-19.

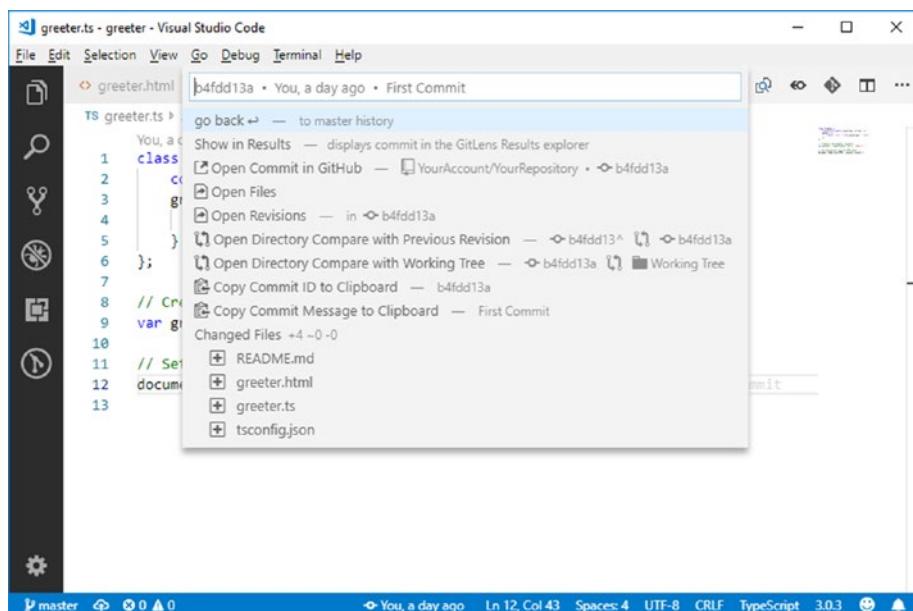
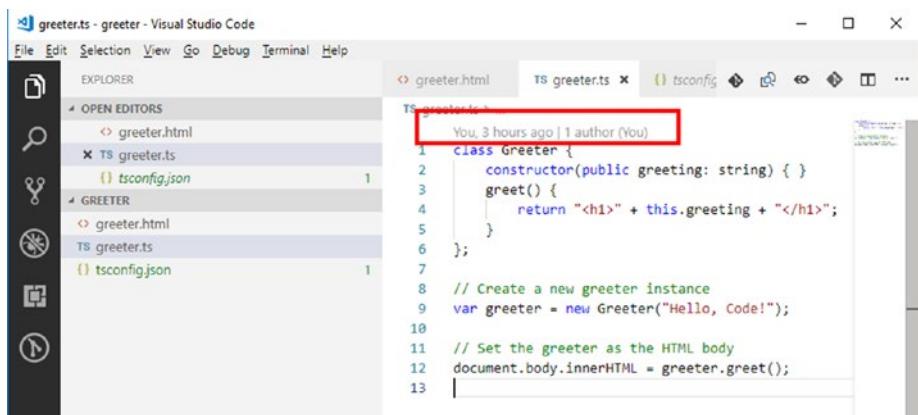


Figure 7-19. *GitLens commands*

These commands allow you to open the commit in your remote repository but also to open the commit revisions. Additionally, it allows copying the commit ID or message to the clipboard. You can also expand the file names below and see individual details for the current code commit. GitLens also adds summary information about edits made on a specific code snippet, right above the code snippet itself. Figure 7-20 shows an example.

A screenshot of Visual Studio Code showing the GitLens extension. The interface includes a dark-themed sidebar with icons for file operations like Open, Save, and Find. The main area has tabs for 'greeter.html', 'TS greeter.ts', and 'tsconfig.json'. The 'tsconfig.json' tab is active, displaying the following code:

```
1  class Greeter {  
2      constructor(public greeting: string) {}  
3      greet() {  
4          return "<h1>" + this.greeting + "</h1>";  
5      }  
6  };  
7  
8  // Create a new greeter instance  
9  var greeter = new Greeter("Hello, Code!");  
10  
11 // Set the greeter as the HTML body  
12 document.body.innerHTML = greeter.greet();  
13
```

A red box highlights the commit information for line 1: 'You, 3 hours ago | 1 author (You)'. The status bar at the bottom right shows '147 changes'.

Figure 7-20. GitLens adds summary information about a code snippet

If you click at the left side of the divider, you will get to the menu shown in Figure 7-19. If you instead click the author name, VS Code will show a popup that contains the list of commits made by the selected author and, if you hover over a commit name, you will see the full commit details (see Figure 7-21).

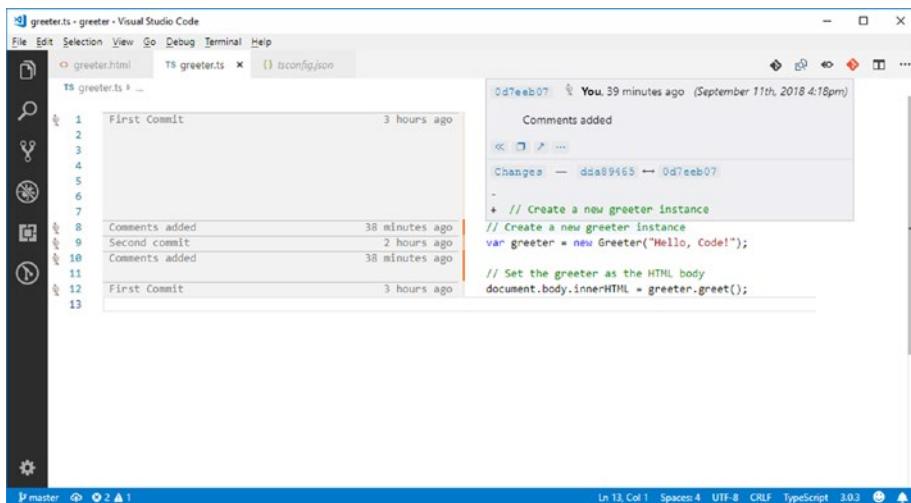


Figure 7-21. *GitLens shows information about a commit*

Other commands are available in the context menu when you right-click the code editor, such as Copy Commit ID To Clipboard, Copy Message To Clipboard, and Copy Remote File URL To Clipboard, all self-explanatory.

Note All the preceding commands described are also available via shortcuts that you can find on the upper right corner of the code editor bar (see Figure 7-21).

GitHub Pull Requests

Pull requests in Git make it easier to perform code reviews. With pull requests, your code is not automatically merged into a branch until someone else in the team reviews the code and accepts it. If you use GitHub for your repositories, an extension called **GitHub Pull Requests** is available to introduce support for pull requests in Visual Studio Code. When you first install the extension (and reload the environment), you will

be asked to sign into GitHub. After you provide your GitHub credentials and open a folder that is associated to a remote repository hosted on GitHub, you will see a new treeview called **GITHUB PULL REQUESTS** in the Git bar (see Figure 7-22).

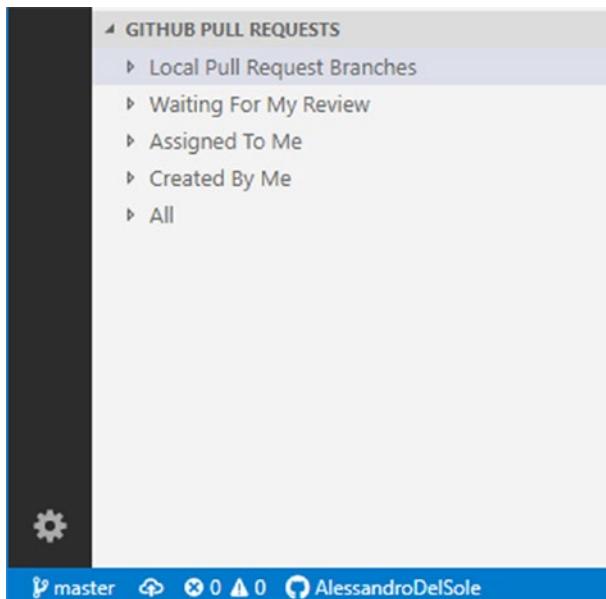


Figure 7-22. The GitHub Pull Requests view

Currently, the extension does not support submitting pull requests from VS Code, but you can manage existing pull requests submitted by other tools such as Microsoft Visual Studio or GitHub itself. When pull requests are available, you will see them listed in the view. If you select a pull request, a new editor window will appear showing all the pull request details, and you will have the option of add a comment and then close, reject, or approve the pull request (see Figure 7-23).

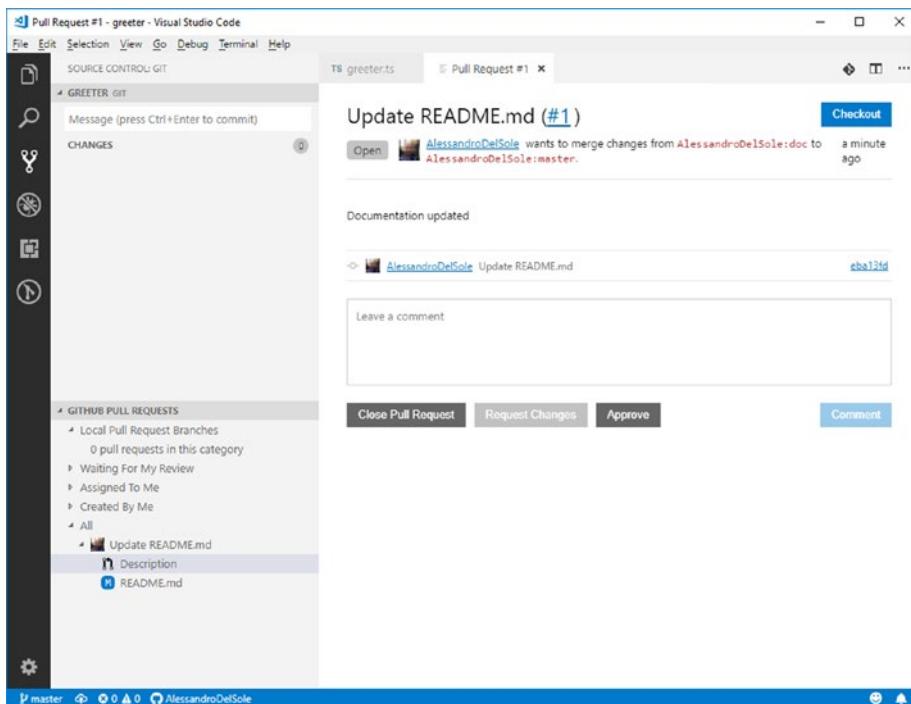


Figure 7-23. Managing a pull request from VS Code

You will also be able to work on the pull request locally if you click the **Checkout** button, and it will be displayed under the Local Pull Request Branches node in the treeview. This is a very useful extension especially if you work within Agile teams, but remember it only supports GitHub as the host.

Working with Azure DevOps and Team Foundation Server

Azure DevOps (<https://azure.microsoft.com/en-us/solutions/devops>), formerly Visual Studio Team Services, and Team Foundation Server are the complete solutions from Microsoft to manage the entire application lifecycle, from development to testing to continuous

integration and delivery. Azure DevOps is a cloud service, whereas Team Foundation Server works on premises. Among the many features, they both provide source control capabilities based on two engines: Git and the Microsoft Team Foundation Server engine.

In this section I will explain how to configure a Git repository that you can use for source control with Visual Studio Code. I will use Azure DevOps so that you do not need to have an on-premise installation of Team Foundation Server.

Before going on reading, you will need to install the **Visual Studio Team Services extension for Visual Studio Code**, following the steps you are already familiar with. This extension suits for both Azure DevOps and Team Foundation Server.

Note According to the extension documentation, you will also need the Team Foundation command line client installed. If you have Visual Studio 2017 on Windows or Visual Studio for Mac installed, you already have all you need. If not, or if you are on Linux, visit <https://bit.ly/2x99fEH> and search for the most appropriate installer based on your system.

You obviously need an account on Azure DevOps, which you can create by using a Microsoft account. If you do not have one, you can get a Microsoft account at www.outlook.com, and then you can get an account on Azure DevOps at <https://aka.ms/SignupAzureDevOps>. Follow all the instructions required to configure your account for the first time. When in the home page, click the **Create Project** button. As you can see in Figure 7-24, you will need to supply a team project name, a source control engine, and a work item process.

CHAPTER 7 SOURCE CONTROL WITH GIT

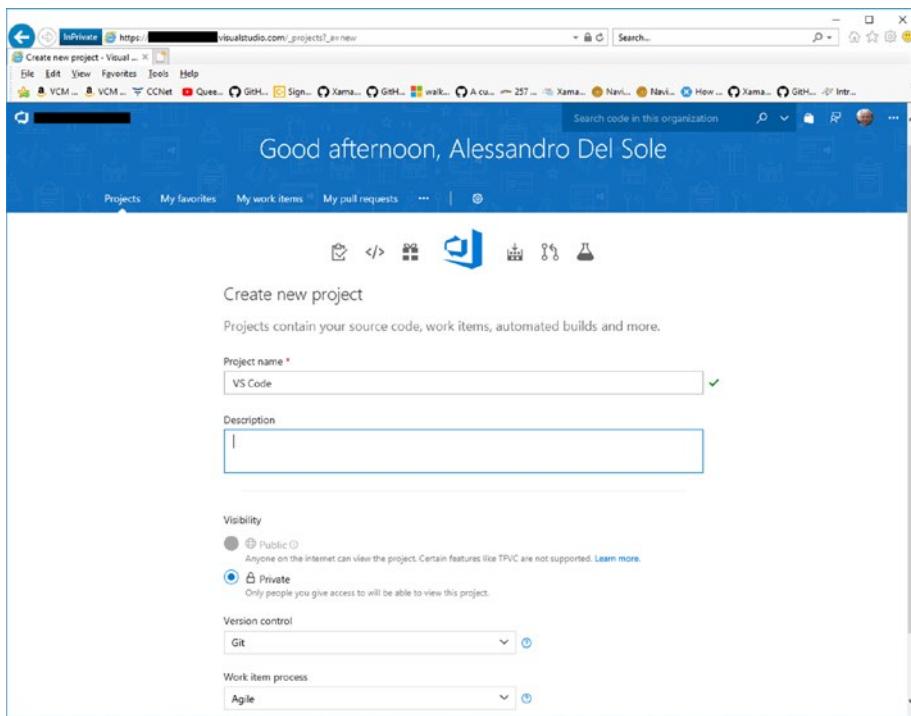


Figure 7-24. Creating a team project in Azure DevOps

Enter a project name of your choice, such as VS Code in the example, and make sure that Git is selected as the source control engine. Leave **Agile** as the choice for the work item process, and finally click **Create**. After a few seconds, your new team project will be ready. At this point, the Azure DevOps site will show a page with all the information about your new Git repository. Before cloning the repository on your local machine, a good idea is providing your Azure DevOps credentials to Visual Studio Code. To accomplish this, in VS Code open the Command Palette and type `> Team Signin`. The login popup will appear, so enter the Microsoft account credentials you used to set up your Azure DevOps workspace and wait for VS Code to authenticate. Now, if you go back to the web portal, you will simply need to click the **Clone in VS Code** button (see Figure 7-25).

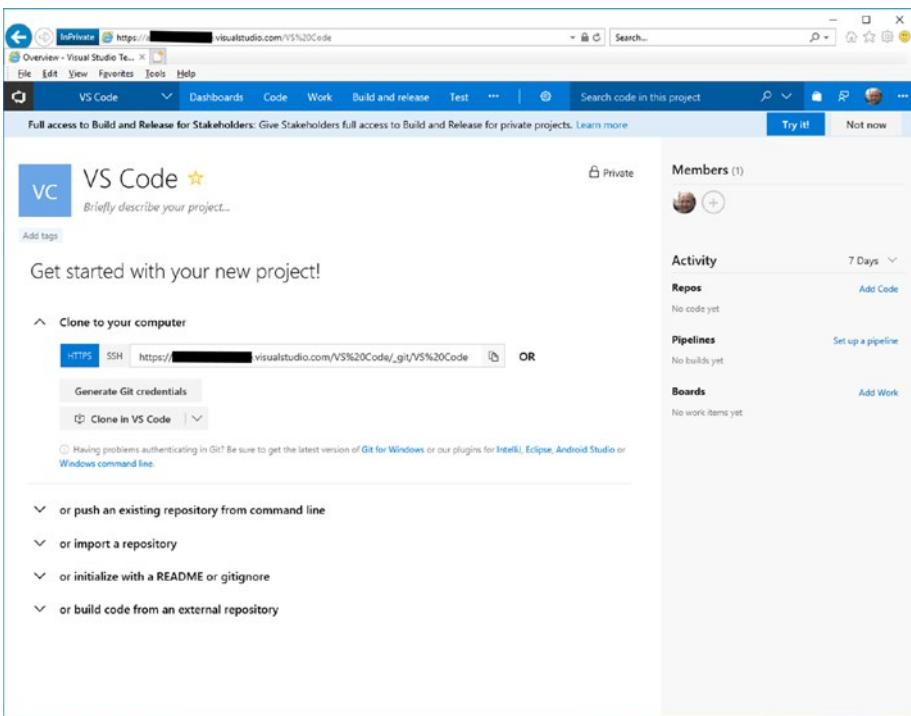


Figure 7-25. Cloning a repository from Azure DevOps

By clicking this button, you will be asked to grant VS Code the permission to open the remote repository. When permission is granted, VS Code will ask you to specify a local folder for cloning the remote repository. Once you have specified the folder, the cloning process will start, and after it completes, you will have both a remote and a local repository. The target folder will be opened in Visual Studio Code after cloning the repository, and you will now be able to use all the Git capabilities described in the previous sections. Moreover, the extension adds a few shortcuts to the status bar, as you can see in Figure 7-26.



Figure 7-26. The Team Services buttons in the status bar

More specifically, from left to right, after the branch name and the Synchronize Changes button, you can see the name of the team project, then a shortcut that allows for browsing pull requests on the web portal, a shortcut for opening build definitions on the web portal, a shortcut for viewing pinned work items, and a feedback button you can use to share your thoughts about the extension.

Though most of the operations that are not strictly related to Git, such as opening build definitions and work items, must be done in the web portal, both Azure DevOps and Team Foundation Server are very popular and widely used services among enterprises, so having an option to connect them to Visual Studio Code so easily will save you a lot of time.

Summary

Writing software involves collaboration. This is true if you are part of a development team but also if you are involved in open source projects, or if you are an individual developer who has interactions with customers. In this chapter you have seen how Visual Studio Code provides integrated tools to work with Git, the popular open source and cross-platform source control provider.

You have seen how to create a local repository with the Git bar and how to associate it to a remote repository with a couple of commands from the integrated terminal. Then you have seen how you can handle file changes, including commits, and how you can create and manage branches directly from within the environment. In addition, you were introduced to some useful extensions, such as Git History, Git Lens, and GitHub Pull Requests, that will boost your productivity by adding important features that every developer needs when it comes to team collaboration. Finally, you have seen how easy it is to open in VS Code a Git repository hosted on Azure

DevOps, the premiere cloud solution from Microsoft to manage the whole application lifecycle. Behind the scenes, Visual Studio Code invokes the Git command in order to execute operations over your source code, and it is preconfigured to work with this external tool.

However, it is not limited to work with a small set of predefined tools, rather it can be configured to work with basically any external program. This is what you will learn in the next chapter.

CHAPTER 8

Automating Tasks

When talking about Visual Studio Code, you will often hear that it is not a simple code editor. This is certainly true, and the reason is that it allows executing operations such as compiling and testing code by running external tools. In this chapter you will learn how Code can execute external programs via tasks, by both learning about existing tasks and configuring custom tasks. In order to run the examples provided in this chapter, you will need the following software:

- Node.js, a free and open source JavaScript runtime based upon Chrome's JavaScript engine that can be downloaded from <https://nodejs.org>
- The TypeScript compiler (tsc), which you install via the Node.js command line with the following command:
`> npm install -g typescript`

Using Node.js and TypeScript will help you to avoid dependencies on the operating system and proprietary development environments. Obviously, all the topics discussed in this chapter apply to other languages and platforms as well. For the last example about MSBuild tasks on Windows, you instead need Microsoft Visual Studio 2017. The Community edition is available for free at www.visualstudio.com.

Note If you have worked with the first Visual Studio Code releases, it is important for you to know that the way tasks are handled has changed. If this is your case, you might want to read the migration guide for tasks written with older versions to the last version:
https://code.visualstudio.com/docs/editor/tasks#_convert-from-010-to-200.

Understanding Tasks

At its core, Visual Studio Code is a code-centric tool, so it often requires executing external programs to complete operations that are part of the application lifecycle, such as compilation, debugging, and testing.

In the Visual Studio Code terminology, integrating with an external program within the flow of the application lifecycle is a *task*. Running a task not only means executing an external program but also getting the output of the external program and displaying it in the most convenient way inside the user interface.

Note Tasks are only available with folders, not individual code files.

A task is basically a set of instructions and properties represented with the JSON notation, stored in a special file called `tasks.json`. If VS Code is able to detect the type of project or source code inside the folder, a `tasks.json` file will not always be necessary, and VS Code will do all the job for you. If it cannot detect the type of project or source code, or if you are not satisfied with the default settings of a task, under the current folder, it generates a hidden subfolder called `.vscode`, and, inside this folder, it also generates a `tasks.json` file. If VS Code is able to detect the type of project or source code inside the folder, it will also prefill the `tasks.json` content with

the proper information, otherwise you will need to configure tasks.json manually. For a better understanding, I will explain tasks that VS Code can detect and that it configures on your behalf, and then I will discuss how to create and configure tasks manually.

Tasks Types

There is no limit to how many types of tasks could be available for a source code folder, but the most common are the following:

- *Build task*: A build task is configured to compile the source code, assets, metadata, and resources into a binary or executable file, such as libraries or programs.
- *Test task*: A test task is configured to run unit tests in the source code.
- *Watch task*: A watch task starts a compiler in the so-called watch mode. In this mode, a compiler always watches for changes to any unresolved files after the latest build and will recompile them at every save.

Visual Studio Code provides built-in shortcuts to execute a build task. When new tasks are added, VS Code updates itself to provide shortcuts for the new tasks. Additionally, you can differentiate tasks of the same type. For example, you can have a default build task and other custom build tasks that can be executed only with specific situations.

Running and Managing Tasks

The first approach to understanding tasks in practice is running existing, preconfigured tasks. For the sake of simplicity, start Visual Studio Code and open the project folder called **simple** from the collection of examples you downloaded previously from the TypeScript Samples repository on GitHub (<https://github.com/Microsoft/TypeScriptSamples>).

CHAPTER 8 AUTOMATING TASKS

Visual Studio Code detects it as a TypeScript project, and therefore it will preconfigure some tasks (in the next section, I will provide more details about task auto-detection). Now open the **Terminal** menu. As you can see, there are several commands related to tasks, as you can see in Figure 8-1.

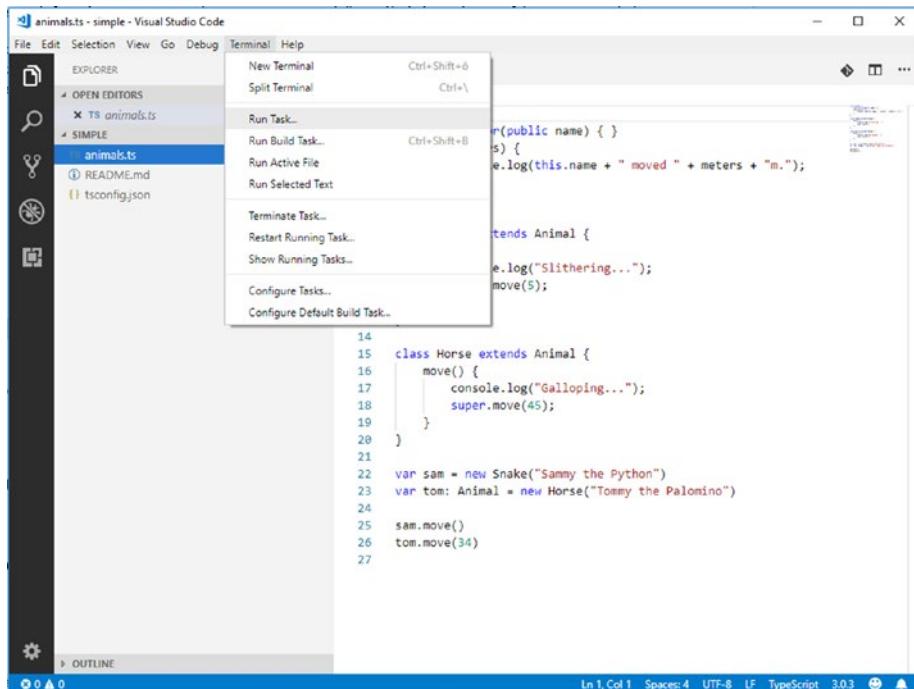


Figure 8-1. Commands for running and managing tasks in the Terminal menu

An explanation of each command is provided in Table 8-1.

Table 8-1. Commands for Task Execution and Management

| Command | Description |
|------------------------------|--|
| Run Task | Shows the list of available tasks in the Command Palette and runs the selected task. |
| Run Build Task | Runs the default, preconfigured build task (if any). |
| Terminate Task | Forces a task to be stopped. |
| Restart Running Task | Restarts the currently running task. |
| Show Running Task | Shows the output of the currently running task in the Terminal panel. |
| Configure Tasks | Shows the list of available tasks in the Command Palette and allows editing the selected task inside the tasks.json file editor. |
| Configure Default Build Task | Shows the list of available tasks in the Command Palette and allows selecting for the task that will be used as the build task. |

If you select **Run Task**, VS Code will open the Command Palette showing the list of available tasks, as represented in Figure 8-2.

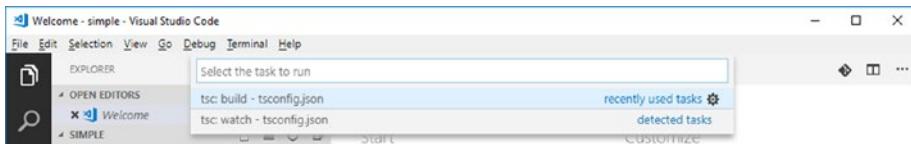
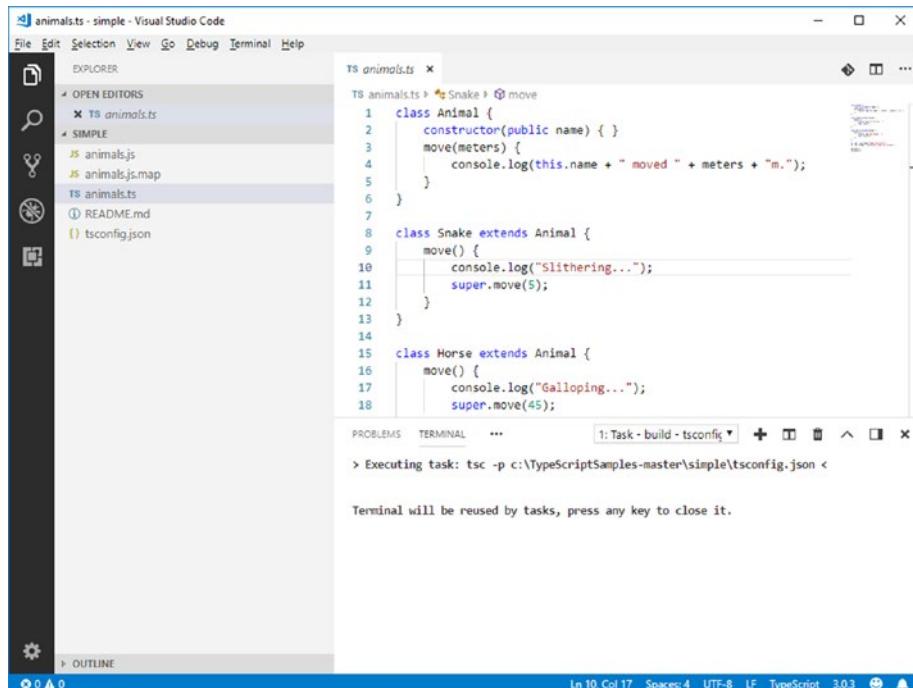


Figure 8-2. Running a task from the Command Palette

As you can see, there are two tasks: **tsc build** and **tsc watch**, both pointing to the tsconfig.json project file. This means that either task will run against the specified file. **tsc** is the name of the command line TypeScript compiler, whereas **build** and **watch** are two preconfigured

tasks whose description has been provided previously. If you select tsc build, Visual Studio Code will launch the tsc compiler and will compile the TypeScript code into JavaScript code, as shown in Figure 8-3.

Note In the case of TypeScript, the build task will compile TypeScript code into JavaScript code. In the case of other languages, the build task will generate binaries from the source code. More generally, a build task will produce the expected output from the compilation process depending on the language. Also, the list of available tasks varies depending on the type of project or folder you are working with. For example, for .NET Core projects, only a task called **build** is available.



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

- File Explorer:** Shows files in the 'SIMPLE' folder: animals.js, animals.ts, README.md, and tsconfig.json. 'animals.ts' is the active file.
- Editor:** Displays the contents of 'animals.ts'. The code defines three classes: Animal, Snake, and Horse, each with a move method that logs a message to the console.
- Terminal:** At the bottom, the terminal window shows the command being run: > Executing task: tsc -p c:\TypeScriptSamples-master\simple\tsconfig.json. It also displays the message: Terminal will be reused by tasks, press any key to close it.
- Status Bar:** Shows the current file is 'animals.ts', line 10, column 17, with 4 spaces, using UTF-8 encoding, and the TypeScript version is 3.0.3.

Figure 8-3. Executing a build task

The Terminal panel shows the progress and result of the task execution. In this case, the result of the task is also represented by the generation of a .js file and a .js.map file, now visible in the Explorer bar. You can stop and restart a task using the **Terminate Task** and **Restart Running Task** commands, respectively, both described in Table 8-1. Now suppose there is a critical error that prevents the build task from completing successfully. For demonstration purposes, remove a closing bracket from the code of the simple.ts file and run again the build task. At this point, Visual Studio Code will show the detailed log from the tsc tool in the Terminal panel, as shown in Figure 8-4, describing the error and the line of code that caused it.

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

- Title Bar:** animals.ts - Visual Studio Code
- File Menu:** File Edit Selection View Go Debug Terminal Help
- Explorer Bar:**
 - OPEN EDITORS: TS animals.ts
 - SIMPLE: JS animals.js, JS animals.js.map, TS animals.ts (highlighted)
 - README.md, tsconfig.json
- Editor Area:** A code editor window titled "TS animals.ts" containing the following TypeScript code:

```

1  class Animal {
2      constructor(public name) { }
3      move(meters) {
4          console.log(this.name + " moved " + meters + "m.");
5      }
6  }
7
8  class Snake extends Animal {
9      move() {
10         console.log("Slithering...");
11         super.move(5);
12     }
13
14
15  class Horse extends Animal {
16      move() {
17         console.log("Galloping...");
18         super.move(45);
    }
  
```
- Terminal Tab:** Task - build - tsconfig.json
- Terminal Output:**

```

> Executing task: tsc -p c:\TypeScriptSamples\master\simple\tsconfig.json <
animals.ts:15:1 - error TS1068: Unexpected token. A constructor, method, accessor, or property was expected.

  class Horse extends Animal {
  ^^^^^^

The terminal process terminated with exit code: 1
Terminal will be reused by tasks, press any key to close it.
  
```
- Status Bar:** Ln 13, Col 1 Spaces: 4 UTF-8 LF TypeScript 3.0.3

Figure 8-4. Visual Studio Code shows the output of the external tool in a convenient way

In the real world, this error would not probably happen because you have the Problems panel and red squiggles in the code editor that both highlight the error. But this is actually an example of how Visual Studio Code integrates with an external tool and shows its output directly in the Terminal panel, helping to solve the problem with the most detailed information possible.

The Default Build Task

Because building the source code is the most frequently used task, Visual Studio Code provides a built-in shortcut to run this task in the Terminal menu, called **Run Build Task** (Ctrl+Shift+B on Windows and ⌘+⌘+B on macOS). However, you first need to set a default build task, otherwise the Run Build Task command will behave like the Run Task command.

To accomplish this, select **Terminal > Configure Default Build Task**. When the Command Palette appears, select the task you want to be set as the default build task, in this case select **tsc build**. When you do this, Visual Studio Code is actually changing its default configuration and therefore will generate a new tasks.json file under the .vscode folder, and it will open this file in a new editor window. The content and structure of tasks.json file will be discussed shortly in this chapter, so for now let's focus on the new default build task. If you now select **Terminal > Run Build Task**, or use the keyboard shortcut, you will see how the default build task will be executed, without the need of specifying it every time from the Command Palette.

Auto-Detected Tasks

Visual Studio Code can auto-detect tasks for the following environments: Grunt, Gulp, Jake, and Node.js. Auto-detecting tasks means that Visual Studio Code can analyze a project built for one of the aforementioned platforms and generate the appropriate tasks without the need of creating custom ones. Figure 8-5 shows an example based on the Node debugger

extension for Visual Studio Code, whose source code is available at <https://github.com/Microsoft/vscode-node-debug>.

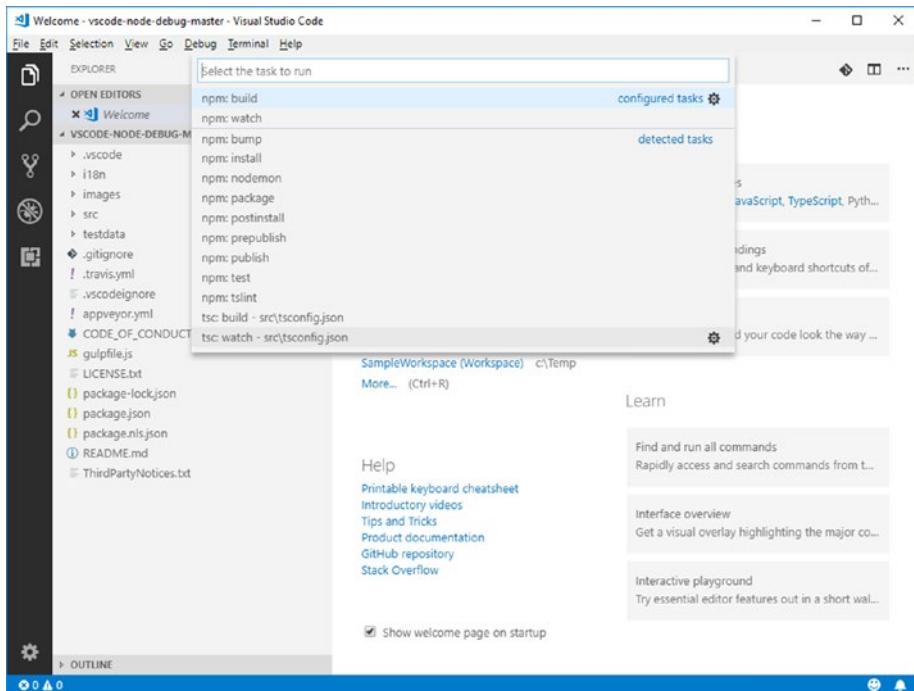


Figure 8-5. Auto-detected tasks

The source code of this extension is made of JavaScript and TypeScript files and is built upon the Node.js runtime. So Visual Studio Code has been able to detect a number of tasks that work well with this kind of project, including tasks to run **npm** (the command line tool for Node.js) and the **tsc** TypeScript compiler.

Auto-detected tasks are very useful because they allow to save a lot of time in terms of task automation. However, more often than not, you will have needs that are not satisfied by existing tasks, so you will need to make your own customizations.

Note In order to auto-detect tasks, behind the scenes VS Code requires that specific environments are installed. For example, VS Code can auto-detect tasks based on Node.js only if Node.js is installed; similarly, it can auto-detect tasks based on Gulp only if Gulp is installed and so on.

Configuring Tasks

When Visual Studio Code cannot auto-detect tasks for a folder, or when auto-detection does not satisfy your needs, you can create and configure custom tasks by editing the tasks.json file. In this section I will go through two examples that will help you understand how to configure your own tasks.

More specifically, I will explain how to compile Pascal source code files using the OmniPascal extension and the Free Pascal compiler, available to all operating systems, and how to build a Visual Studio solution based on the full .NET Framework on Windows by invoking the MSBuild.exe compiler.

In order to complete both the examples, you will need the following:

- The OmniPascal language extension for Visual Studio Code, which you can download via the Extensions panel. This extension is useful to enable Pascal syntax highlighting and code navigation, though you can still compile source files without it.
- The Free Pascal compiler, which includes all you need to develop applications using Pascal and that provides a free command line compiler. Free Pascal is available for Windows, macOS, Linux, and other systems, and it can be downloaded from www.freepascal.org.

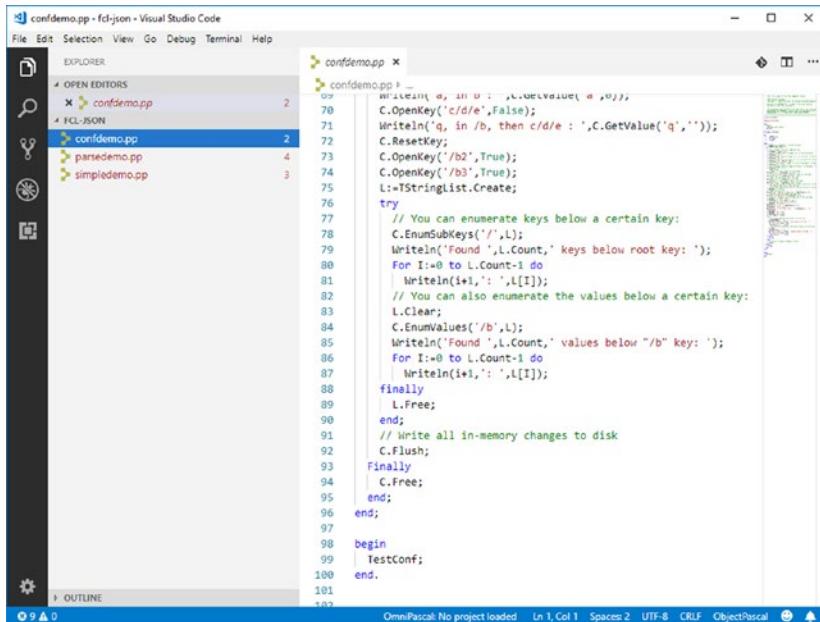
- On Windows only, download the latest version of the .NET Framework (4.7.2 at this writing), which includes the MSBuild.exe tool.

Let's start with an example based on the Pascal language.

First Example: Compiling Pascal Source Code

In this section, I will explain how to create a custom task that allows for compiling Pascal source code files by invoking the Free Pascal command line compiler from VS Code. Assuming you have downloaded and installed the required software as listed in the preceding text, locate the Free Pascal folder installation on disk (usually /FPC/version number), then open the **examples** folder. In Visual Studio Code, open any folder containing some Pascal source code. I will use one called fcl-json.

Figure 8-6 shows how Visual Studio Code appears with Pascal source files currently opened.



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

- Title Bar:** confdemo.pp - fcl-json - Visual Studio Code
- File Menu:** File Edit Selection View Go Debug Terminal Help
- Explorer Panel:** Shows 'OPEN EDITORS' with 'confdemo.pp' (2), 'fcl.json' (4), and 'confdemo.pp' (2) selected. It also lists 'parsetest.pas' (3) and 'simpleset.pas' (3).
- Editor Panel:** Displays the content of the 'confdemo.pp' file. The code is a Pascal program that interacts with a registry key. It includes sections for writing values, opening keys, reading values, and enumerating keys and values below a certain key. It uses the C and L variables and several writeln statements to output its progress.
- Bottom Status Bar:** OmniPascal No project loaded Ln 1, Col 1 Spaces 2 UTF-8 CR LF ObjectPascal

```

1  confdemo.pp - fcl-json - Visual Studio Code
2  File Edit Selection View Go Debug Terminal Help
3  EXPLORER
4  OPEN EDITORS
5  confdemo.pp 2
6  fcl.json 4
7  confdemo.pp 2
8  parsetest.pas 3
9  simpleset.pas 3
10
11  writeln( a, in b : _L.Values[ a, b ] );
12  C.OpenKey('c/d/e',False);
13  Writeln('q, in b, then c/d/e : ',C.GetValue('q',''));
14  C.ResetKey;
15  C.OpenKey('b/2',True);
16  C.OpenKey('b/3',True);
17  L:=TStringList.Create;
18  try
19    // You can enumerate keys below a certain key:
20    C.EnumSubKeys('/L');
21    Writeln('Found ',L.Count,' keys below root key: ');
22    For I:=0 To L.Count-1 do
23      Writeln(I+1,'.',L[I]);
24    // You can also enumerate the values below a certain key:
25    L.Clear;
26    C.EnumValues('/b/L');
27    Writeln('Found ',L.Count,' values below "/b" key: ');
28    For I:=0 To L.Count-1 do
29      Writeln(I+1,'.',L[I]);
30    finally
31      L.Free;
32    end;
33    // Write all in-memory changes to disk
34    C.Flush;
35    Finally
36      C.Free;
37    end;
38  end;
39  begin
40    testConf;
41  end.
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102

```

Figure 8-6. Editing Pascal source code

CHAPTER 8 AUTOMATING TASKS

The OmniPascal extension installed previously enables syntax colorization and the other common editing features. Now imagine you want to compile the source code into an executable binary by invoking the Free Pascal command line compiler. This can be accomplished by creating a custom task. Follow these steps to create a new tasks.json file and set up the custom task:

1. **Select Terminal ▶ Configure Task:** When the Command Palette appears asking for a task to configure, select **Create tasks.json from template** (see Figure 8-7). There is no existing task to configure at this particular point, so the only thing you can do is creating a new tasks.json file.
2. **The Command Palette will now show the list of available task templates:** **MSBuild**, **maven**, **.NET Core**, and **Others** (see Figure 8-8). Select **Others** to create a new task that is independent from other systems.



Figure 8-7. Creating a new task from scratch

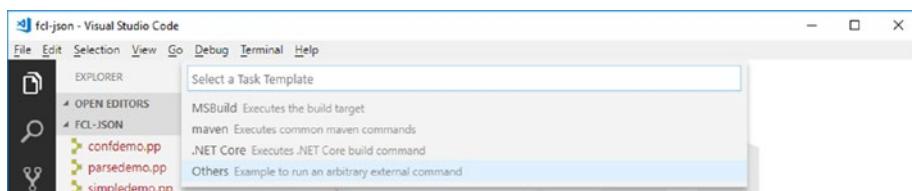


Figure 8-8. Selecting a task template

Visual Studio Code generates a subfolder called .vscode and, inside this folder, a new tasks.json file whose content at this point is the following:

```
{  
    // See https://go.microsoft.com/fwlink/?LinkId=733558  
    // for the documentation about the tasks.json format  
    "version": "2.0.0",  
    "tasks": [  
        {  
            "label": "echo",  
            "type": "shell",  
            "command": "echo Hello"  
        }  
    ]  
}
```

The core node of this JSON file is an array called `tasks`. It contains a list of tasks, and for each tasks, you can specify the text that VS Code will use to display it in the Command Palette (`label`), the type of task (`type`), and the external program that will be executed (`command`). An additional JSON property called `args` allows for specifying command line arguments for the program you invoke. The list of supported JSON properties is available in Table 8-2 and will be discussed later in this chapter, but if you are impatient, you can quickly look at the table and then get back here. Now suppose you want to create a build task which, by convention, is the type of task you use to compile source code. This can be accomplished by modifying `tasks.json` as follows:

```
{  
    // See https://go.microsoft.com/fwlink/?LinkId=733558  
    // for the documentation about the tasks.json format  
    "version": "2.0.0",  
    "tasks": [
```

```
{  
    "label": "build",  
    "type": "shell",  
    "command": "fpc",  
    "args": ["${file}"]  
}  
]  
}
```

The key points are the following:

- The `label` property value is now `build` so that the task is clearly provided as the build task.
- The `type` property value is `shell`, meaning it will be executed by the operating system's shell.
- The `command` property value is `fpc`, which is the file name of the Free Pascal compiler.
- The `args` property value is an array of command line arguments to be passed to the external program; in this case there is only one argument that is the active source file, represented by the `$(file)` variable.

Note As a general rule, an external program can be invoked without specifying its full path only if such a path has been registered in the operating system's environment variables, such as PATH on Windows. In the case of Free Pascal, the installer takes care of registering the program's path, but remember to have a look at the environment variables for other programs.

You could certainly specify the name of the file you want to compile, but using a variable is more flexible so that you can simply compile any file that is currently active in the code editor. In addition to the properties in tasks.json, variables are also discussed shortly and will be summarized in Table 8-3. Notice how IntelliSense helps you find the appropriate properties in tasks.json, as shown in Figure 8-9.

```

tasks.json - Visual Studio Code
File Edit Selection View Go Debug Terminal Help
EXPLORER OPEN EDITORS 1 UNSAVED
  • (1) tasks.json .vscode
  # FCL-JSON
  # .vscode tasks.json 2
  # confdemo.pp
  # parsedemo.pp
  # simpledemo.pp
tasks.json
1 {
2   // See https://go.microsoft.com/fwlink/?LinkId=733558
3   // for the documentation about the tasks.json format
4   "version": "2.0.0",
5   "tasks": [
6     {
7       "label": "build",
8       "type": "shell",
9       "command": "fpc"
10      ...
}
  
```

Arguments passed to the command when this task is invoked.

Intellisense tooltip content:

- args
- dependsOn
- group
- identifier
- isBackground
- linux
- options
- osx
- presentation
- problemMatcher
- promptOnClose
- windows

Figure 8-9. IntelliSense helps defining tasks properties

Save and close tasks.json, then open one of the Pascal source files. Now you can run the newly created build task. Select **Terminal > Run Task**, and from the Command Palette, select the **build** task (see Figure 8-10).

File Edit Selection View Go Debug Terminal Help

EXPLORER OPEN EDITORS build

Select the task to run

OmniPascal No project loaded Ln 10, Col 14 Spaces: 4 UTF-8 LF JSON with Comments

Figure 8-10. Selecting the new task

CHAPTER 8 AUTOMATING TASKS

At this point, VS Code will ask you what would you like to do to detect any problems encountered during the execution of the external program so that they can be displayed in the Problems panel. Detecting problems in the program's output is the job of a so-called problem matcher. This is a more complex topic and will be discussed in a dedicated section. For now, select **Continue without scanning the task output** (see Figure 8-11).

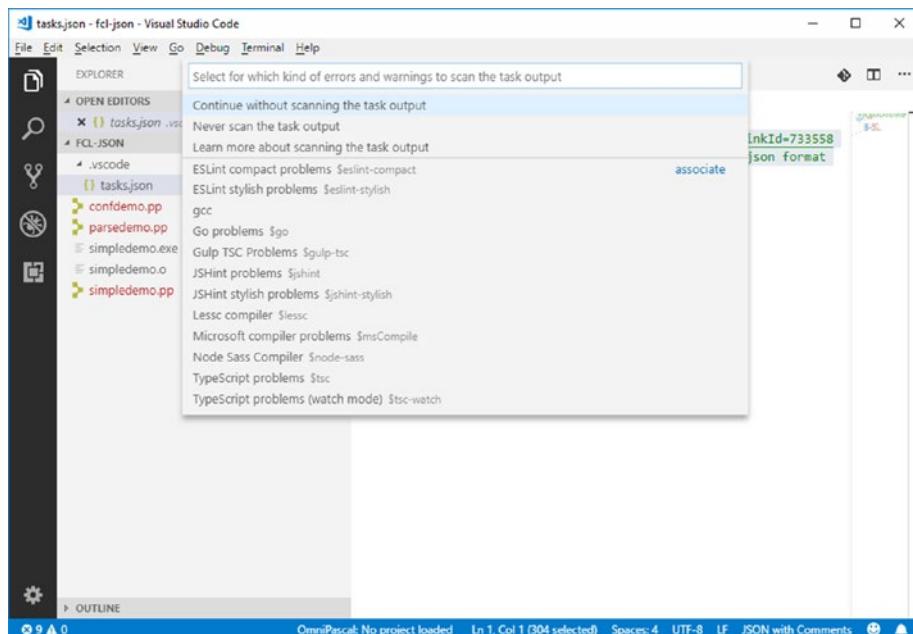


Figure 8-11. Selecting a problem matcher

The Free Pascal compiler will be executed in the Terminal panel, where you also see the program output as demonstrated in Figure 8-12.

The screenshot shows the OmniPascal IDE interface. At the top, there's a toolbar with icons for file operations like new, open, save, and close. Below the toolbar, a tab bar shows '1: Task - build'. The main area is a terminal window displaying the output of the Free Pascal compiler. The output shows the compiler version (3.0.4), copyright information (1993-2017 by Florian Klaempfl and others), target OS (Win32 for i386), compilation details (compiling and linking files to produce simpledemo.exe), and a message indicating the terminal will be reused. At the bottom of the terminal window, there's a status bar with the text 'OmniPascal: No project loaded' and various status indicators like 'Ln 87, Col 25', 'Spaces: 2', 'UTF-8', 'CRLF', 'ObjectPascal', and icons for smiley face and notifications.

```
> Executing task: fpc c:\FPC\3.0.4\examples\fcl-json\simpledo... <
Free Pascal Compiler version 3.0.4 [2017/10/06] for i386
Copyright (c) 1993-2017 by Florian Klaempfl and others
Target OS: Win32 for i386
Compiling c:\FPC\3.0.4\examples\fcl-json\simpledo...pp
Linking c:\FPC\3.0.4\examples\fcl-json\simpledo...exe
331 lines compiled, 0.4 sec, 234368 bytes code, 8916 bytes data

Terminal will be reused by tasks, press any key to close it.
```

Figure 8-12. Executing the Free Pascal compiler

If the execution succeeds, you will find a new binary file in the source code's folder. If it fails, the compiler's output displayed in the Terminal panel will help you understand what the problem was. Before moving to a second example, I will now explain more about default tasks, task templates, JSON properties in tasks.json, and variables.

Multiple Tasks and Default Build Tasks

Tasks.json can define multiple tasks. At the beginning of this chapter, I told you that, among the others, common tasks are build and test, but you might want to implement multiple tasks that are specific to your scenario. For example, suppose you want to use the Free Pascal compiler to build Delphi source code files.

The Free Pascal command line compiler provides the `-Mdelphi` option, which enables compilation based on the Delphi compatibility mode. You can therefore modify tasks.json as follows:

```
{
  // See https://go.microsoft.com/fwlink/?LinkId=733558
  // for the documentation about the tasks.json format
  "version": "2.0.0",
```

CHAPTER 8 AUTOMATING TASKS

```
"tasks": [
  {
    "label": "build",
    "type": "shell",
    "command": "fpc",
    "args": ["${file}"]
  },
  {
    "label": "Delphi build",
    "type": "shell",
    "command": "fpc",
    "args": ["${file}", "-Mdelphi"]
  }
]
```

As you can see, there is a new custom task called `Delphi build` in the tasks array which still invokes the Free Pascal compiler on the active file, but with the `-Mdelphi` option being passed as a command line argument. Now if you select **Terminal ▶ Run Task** again, you will see both tasks in the Command Palette, as demonstrated in Figure 8-13.

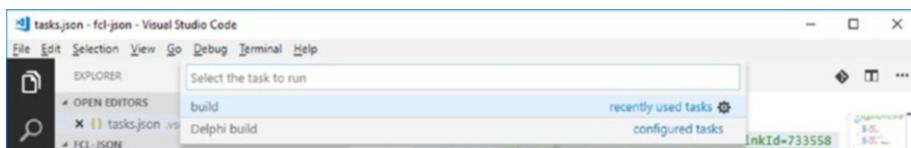


Figure 8-13. All defined tasks are displayed in the Command Palette

It is common to have multiple build tasks, and even multiple tasks of the same type, but in most cases, you will usually run the same task and keep other tasks for very specific situations. Related to the current example, you will usually build Pascal source files and sometimes build

Delphi source files, so a convenient choice is configuring a default build task for Pascal files. As you learned in the “The Default Build Task” section previously, this can be easily accomplished with the following steps:

1. Select **Terminal ▶ Configure Default Build Task**.
2. In the Command Palette, select the **build** task defined previously.
3. With a Pascal source file active, select **Terminal ▶ Run Build Task**, or press the keyboard shortcut for your system.

This command will automatically start the default build task, without the need of manually selecting a task every time.

Understanding tasks.json Properties and Substitution Variables

There are a number of properties available to customize a task. Table 8-2 provides a summary of common properties you use with custom tasks.

Table 8-2. Available Properties for Task Customization

| Property Name | Description |
|---------------|--|
| label | A string used to identify the task (e.g., in the Command Palette). |
| type | Represents the task type. For custom tasks, supported values are <code>shell</code> and <code>process</code> . With <code>shell</code> , the command is interpreted as a shell command (such as <code>bash</code> , <code>cmd</code> , or <code>PowerShell</code>). With <code>process</code> , the command is interpreted as a process to be executed. |
| command | The command or external program to be executed. |
| args | An array of command line arguments to be passed to the command. |

(continued)

Table 8-2. (continued)

| Property Name | Description |
|----------------------|---|
| windows | Allows specifying task properties that are specific to the Windows operating system. |
| osx | Allows specifying task properties that are specific to macOS. |
| linux | Allows specifying task properties that are specific to Linux and its distributions. |
| group | Allows for defining task groups and for specifying to which group a task belongs to. |
| presentation | Defines how Visual Studio Code handles the task output in the user interface (see the following example). |
| options | Allows for providing custom values about the cwd (current working directory), env (environment variables), and shell (default shell) options. |

The windows, osx, and linux properties will be discussed separately in the next section. The group property allows grouping tasks by category. For instance, if you consider the two multiple tasks created previously, they are both related to building code, so they might be grouped into a category called **build**. This is accomplished by modifying tasks.json as follows:

```
{
  // See https://go.microsoft.com/fwlink/?LinkId=733558
  // for the documentation about the tasks.json format
  "version": "2.0.0",
  "tasks": [
    {
      "label": "build",
      "type": "shell",
```

```

    "args": ["${file}"],
    "group": "build",
  },
  {
    "label": "Delphi build",
    "type": "shell",
    "command": "fpc",
    "args": ["${file}", "-Mdelphi"],
    "group": "build"
  }
]
}

```

Notice how IntelliSense shows the built-in supported values for the group property (see Figure 8-14).

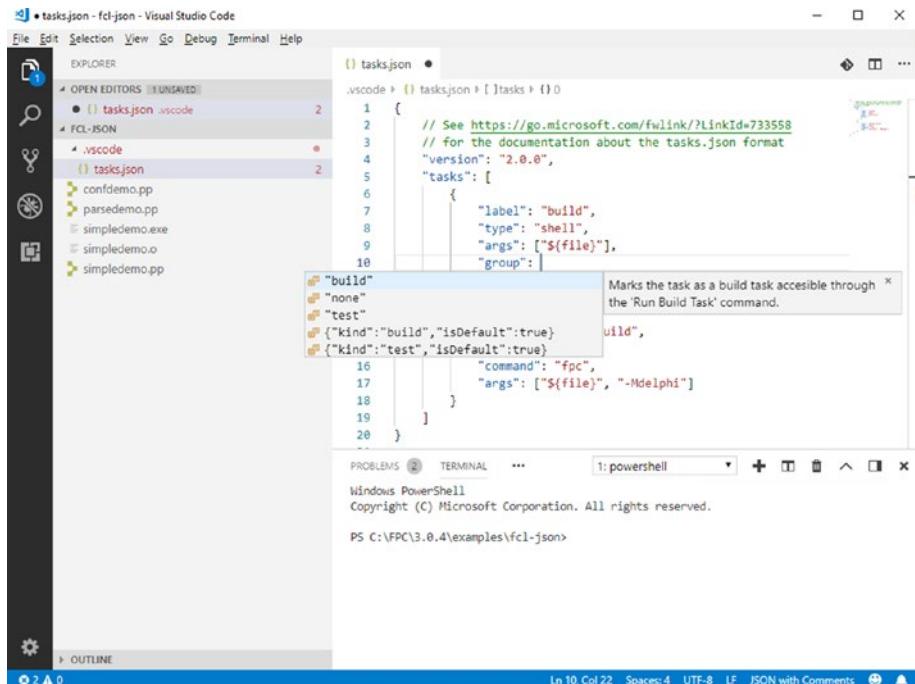


Figure 8-14. IntelliSense helping with groups

CHAPTER 8 AUTOMATING TASKS

Notice how you can also specify additional values to individual tasks in a group. For example, if you want to set a task as the default one in the group, you might change the JSON as follows:

```
"group": {  
    "kind": "build",  
    "isDefault": true  
}  
}
```

The `kind` property represents the group name and `isDefault` is self-explanatory. You can also customize the way VS Code handles the task output via the `presentation` property. When you type `presentation` and then press Enter, IntelliSense adds a number of key/value pairs with some default values, as follows:

```
"presentation": {  
    "echo": true,  
    "reveal": "always",  
    "focus": false,  
    "panel": "shared",  
    "showReuseMessage": true  
}
```

Following is the description of each key and its values:

- `echo` can be `true` or `false` and specifies whether the task output is actually written to the Terminal panel.
- `reveal` can be `always`, `never`, or `silent` and specifies whether the Terminal panel where the task is running should be always visible, never visible, or visible only when a problem matcher is not specified and some errors occur.

- focus can be true or false and specifies if the Terminal panel should get focused when the task is running.
- panel can be shared, dedicated, or new, and it specifies if the terminal instance is shared across tasks or if an instance must be dedicated to the current task or if a new instance should be created at every task run.
- showReuseMessage can be true or false and specifies whether a message should be displayed to inform that the Terminal panel will be reused by a task and that therefore it is possible to close it.

The values you see in the preceding snippet are the default values. In case of default values, a key can be omitted. For example, the following markup demonstrates how to create a new Terminal panel at every run without showing a reuse message:

```
"presentation": {  
    "panel": "new",  
    "showReuseMessage": false  
}
```

Other values can be omitted because we are okay with the default values seen in the preceding text.

Note The list of supported properties is much longer, but most of them are not of common use. If you want to get deeper knowledge about the full list of available properties, you can look at the tasks.json schema, which provides detailed comments about each property and that is available at <https://code.visualstudio.com/docs/editor/tasks-appendix>.

Visual Studio Code also offers several predefined variables that you can use instead of regular strings and that are useful to represent file and folder names when passing these to a command. Table 8-3 provides a summary of supported variables.

Table 8-3. Supported Substitution Variables

| Variable | Description |
|---|--|
| <code> \${workspaceFolder}</code> | Represents the path of the currently opened folder. |
| <code> \${workSpaceFolderBasename}</code> | Represents the path of the currently opened folder without any slashes. |
| <code> \${file}</code> | The active code file. |
| <code> \${relativeFile}</code> | The active code file relative to <code> \${workspaceFolder}</code> . |
| <code> \${fileBaseName}</code> | The active code file's base name. |
| <code> \${fileBaseNameNoExtension}</code> | The active code file's base name without the extension. |
| <code> \${fileDirname}</code> | The name of the directory that contains the active code file. |
| <code> \${fileExtname}</code> | The file extension of the active code file. |
| <code> \${cwd}</code> | The current working directory of the task. |
| <code> \${lineNumber}</code> | The currently selected line number in the active file. |
| <code> \${selectedText}</code> | The currently selected text in the active file. |
| <code> \${env.VARIABLENAME}</code> | References an environment variable, such as <code> \${env.PATH}</code> . |

Using variables is very common when you run a task that works at the project/folder level or against file names that you either cannot predict or that you do not want to hardcode. You can check the variables documentation for further details at <https://code.visualstudio.com/docs/editor/variables-reference>.

Operating System-Specific Properties

Sometimes you might need to provide task property values that are different based on the operating system. In Visual Studio Code, you can use the windows, osx, and linux properties to specify different values of a property, depending on the target.

For example, the following tasks.json implementation shows how to explicitly specify the path of an external tool for Windows and Linux (the directory names might not be the same on your machine):

```
{  
    // See https://go.microsoft.com/fwlink/?LinkId=733558  
    // for the documentation about the tasks.json format  
    "version": "2.0.0",  
    "tasks": [  
        {  
            "label": "build",  
            "type": "shell",  
            "args": ["${file}"],  
            "windows": {  
                "command": "C:\\Program Files\\FPC\\fpc.exe"  
            },  
            "linux": {  
                "command": "/usr/bin/fpc"  
            }  
        }  
    ]  
}
```

More specifically, you will need to move the property of your interest under the operating system property and provide the desired value. In the preceding code, the command property has been moved from the higher level down to the windows and linux property nodes. All supported properties can have different values, not only command.

Reusing Existing Task Templates

In the previous example about compiling Pascal source code, you have seen how to create a custom task from scratch. However, for some particular scenarios, you can leverage existing task templates, which consists of tasks.json files already preconfigured to work with specific command and settings.

The list of task templates may vary depending on the extensions you have installed, but assuming you have installed only the C# extension, your list should look like in Figure 8-8. The first template is called MSBuild and generates the following tasks.json file:

```
{  
    // See https://go.microsoft.com/fwlink/?LinkId=733558  
    // for the documentation about the tasks.json format  
    "version": "2.0.0",  
    "tasks": [  
        {  
            "label": "build",  
            "type": "shell",  
            "command": "msbuild",  
            "args": [  
                // Ask msbuild to generate full paths for file  
                // names.  
                "/property:GenerateFullPaths=true",  
                "/t:build"  
            ],  
            "group": "build",  
            "isDefault": true  
        }  
    ]  
}
```

```
"group": "build",
"presentation": {
    // Reveal the output only if unrecognized
    errors occur.
    "reveal": "silent"
},
// Use the standard MS compiler pattern to detect
errors, warnings and infos
"problemMatcher": "$msCompile"
}
]
}
```

This template is very useful if you want to work with Microsoft Visual Studio solutions inside VS Code, and a more specific example is coming in the next subsection. It is worth mentioning that this template has been included thinking about C# solutions (such as web applications and Xamarin projects), but MSBuild can build any kind of solution so it can be reused for different purposes.

The second template is called Maven and is tailored to work with the same-named build automation tool for Java. Such a template generates the following tasks.json file:

```
{
    // See https://go.microsoft.com/fwlink/?LinkId=733558
    // for the documentation about the tasks.json format
    "version": "2.0.0",
    "tasks": [
        {
            "label": "verify",
            "type": "shell",
            "command": "mvn -B verify",
            "group": "build"
        },
    ],
}
```

CHAPTER 8 AUTOMATING TASKS

```
{  
    "label": "test",  
    "type": "shell",  
    "command": "mvn -B test",  
    "group": "test"  
}  
]  
}
```

Obviously, Maven must be installed on your machine (you can find it at <https://maven.apache.org>). The third template is called .NET Core and, as the name implies, it generates a tasks.json file which is tailored to automate the build of .NET Core projects. The configuration looks like the following:

```
{  
    // See https://go.microsoft.com/fwlink/?LinkId=733558  
    // for the documentation about the tasks.json format  
    "version": "2.0.0",  
    "tasks": [  
        {  
            "label": "build",  
            "command": "dotnet build",  
            "type": "shell",  
            "group": "build",  
            "presentation": {  
                "reveal": "silent"  
            },  
            "problemMatcher": "$msCompile"  
        }  
    ]  
}
```

In this case, the command is not MSBuild; instead it is dotnet. These templates are useful for at least two reasons:

- They provide ready-to-use configurations for projects of the targeted type, where you might need only a few adjustments.
- They provide a complete task structure, where you only need to replace the command and target and optionally the presentation and the problem matcher.

You will now see an example based on the MSBuild task template.

Second Example: Building a MSBuild Solution (Windows Only)

MSBuild has been the Microsoft build engine since the very first release of the .NET Framework back in 2002. It is a very powerful tool, because it can build a Visual Studio solution with no effort. So, a very nice to have feature would be the possibility of compiling your solutions and projects inside Visual Studio Code.

You can configure a task to run MSBuild.exe, the build engine used by Visual Studio. In the next example, you will see how to compile an MSBuild solution made of a Visual Basic project based on Windows Presentation Foundation (WPF), but of course all the steps apply to any .sln file and to any supported languages. If you do not have one, in Visual Studio 2017 create a blank WPF project with Visual Basic as the language. There's no need of writing code, as I focus on the project type. Save the project, then open the project folder in VS Code.

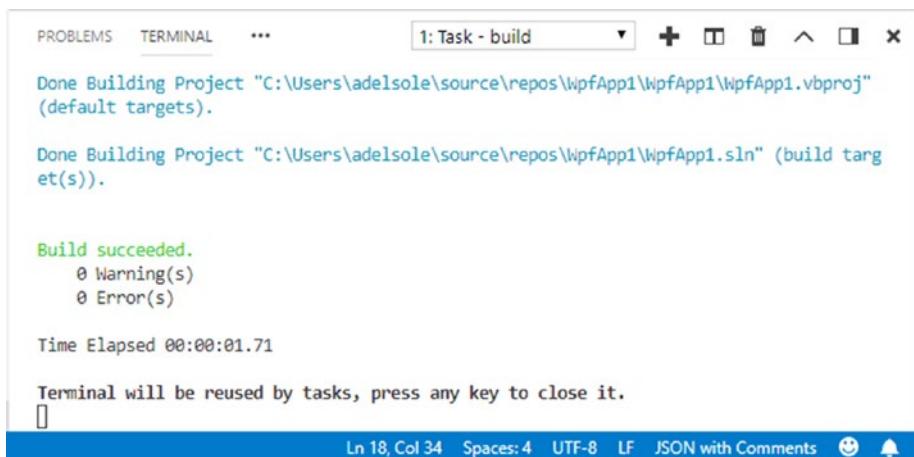
Before configuring a task, it is worth mentioning that, by default, the MSBuild path is not registered in the Windows' environment variables, so you have two possible alternatives:

- Add the MSBuild directory to the PATH environment variable via **Control Panel > System > Advanced system settings > Environment Variables**.
- Specify the full MSBuild pathname in tasks.json. This is the quickest option and the one I will use.

Select **Terminal > Configure Task**. Select the MSBuild template from the list of templates. When tasks.json has been created, change the value of the command property as follows, also replacing Enterprise with the name of the Visual Studio edition you have on your machine, for example:

```
"command": "C:\\Program Files (x86)\\Microsoft Visual Studio\\2017\\Enterprise\\MSBuild\\15.0\\Bin\\msbuild"
```

Also, change the value of the reveal property from never to always for demonstration purposes, so that you can see the output of MSBuild in the Terminal panel. Now if you select **Terminal > Run Task** and select the preconfigured build task, MSBuild will be started and the solution will be built, as you can see in Figure 8-15.



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

```
PROBLEMS TERMINAL ...
1: Task - build + □ ×
Done Building Project "C:\Users\adelsole\source\repos\WpfApp1\WpfApp1\WpfApp1.vbproj"
(default targets).

Done Building Project "C:\Users\adelsole\source\repos\WpfApp1\WpfApp1.sln" (build target(s)).

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

Time Elapsed 00:00:01.71

Terminal will be reused by tasks, press any key to close it.

```

The status bar at the bottom indicates: Ln 18, Col 34 Spaces: 4 UTF-8 LF JSON with Comments 😊 📲

Figure 8-15. Compiling a WPF project written in Visual Basic with the MSBuild task

The preconfigured MSBuild task uses the `$msCompile` problem matcher to detect problems related to C# and Visual Basic in the build output, so that they can be presented in a convenient way in the user interface. Let's spend some more words about problem matchers.

Understanding Problem Matchers

Problem matchers scan the task output text for known warning or error strings and report these inline in the editor and in the Problems panel. Visual Studio Code ships with a number of built-in problem matchers for TypeScript, JSHint, ESLint, Go, C# and Visual Basic, Lessc, and Node Sass (see https://code.visualstudio.com/docs/editor/tasks#_processing-task-output-with-problem-matchers).

Built-in problem matchers are extremely useful, because for the aforementioned environments, VS Code can present problems that occurred at build time in the Problems panel, but it can also highlight the line of code in the code editor that caused the problem.

You can also define custom problem matchers to scan the output of an external program. For instance, a problem matcher for scanning the Free Pascal compiler could look like the following:

```
"problemMatcher": {
    "owner": "external",
    "fileLocation": ["relative", "${workspaceRoot}"],
    "pattern": {
        "regexp": "(([A-Za-z]):\\\\\\(?:[^\\\\/:]*\\\\"
        "<|\\\\r\\\\n]+\\\\\\)*?)?[^\\\\\\\\s\\\\"
        "(?:\\\\\"<|\\\\r\\\\n)*)\\\\((\\\\d+)\\\\":
        "\\\\s.*fatal|error|warning|hint)\\\\"
        "s(.*)\\\\s(.*)",
```

```
"file": 1,  
// The second match group matches the line on which the  
problem occurred.  
"line": 2,  
// The third match group matches the column at which  
the problem occurred.  
"column": 3,  
// The fourth match group matches the problem's  
severity. Can be ignored. Then all problems are  
captured as errors.  
"severity": 4,  
// The fifth match group matches the message.  
"message": 5  
}
```

The owner property represents the language service, which is external in this case, but it could be, for example, cpp in the case of a C++ project. But the most important property is pattern, where you specify a regular expression (regexp) to match error strings sent by the external program. Also notice, with the help of comments, how matches are grouped by target. Building problem matchers can be tricky and it is out of the scope of this book, so I recommend you to read the official documentation available at https://code.visualstudio.com/docs/editor/tasks#_processing-task-output-with-problem-matchers.

Running Files with a Default Program

In case you are editing in VS Code a file whose type is associated with the operating system, you do not need to create custom tasks to run it. For example, a batch program (.bat) in Windows or a shell script file (.sh) on macOS can be run by simply clicking Terminal ➤ Run Active File.

The file name will be passed to the current terminal program on your system (PowerShell on Windows or the bash on Linux and macOS) so that the operating system will try to open the file with the program that is registered with the file extension, if any. In the case of a batch or shell script file, the operating system will execute the file. The output will be displayed in the Terminal panel.

Note Only the output of the operating system or of command line tools will be redirected to the Terminal panel. For instance, if you try to edit a .txt file and then select Terminal ➤ Run Active File, such a file will be opened inside the default text editor on your system, and there will be no additional interactions with the Terminal panel.

Summary

There are many features in Visual Studio Code that make it different from a simple code editor. Tasks are among these features. With tasks you can attach external programs to the application lifecycle and run tools like compilers. VS Code ships with task auto-detection for some environments, but it allows for creating custom tasks when you need to associate specific tools to a project or folder.

By working on the tasks.json file and with the help of IntelliSense, you will be able to include the execution of any external program in your folders. The execution of external programs like compilers is certainly useful, but it would not be so important if VS Code could not make a step forward: debugging code, which is discussed in the next chapter.

CHAPTER 9

Running and Debugging Code

Being an end-to-end development environment, Visual Studio Code offers opportunities that you will not find in other code editors. In fact, in Visual Studio Code, you can work with many project types and debug your code in several languages. This chapter explains how to scaffold projects supported in Visual Studio Code and how to use all the built-in, powerful debugging features.

Creating Applications

Visual Studio Code is independent from proprietary project systems and platforms and, consequently, it does not offer any built-in options to create projects. This means that you need to rely on the tools offered by each platform. In this section, I will explain how to scaffold projects based on .NET Core, but you can similarly create projects with the command line interface offered by other platforms.

It is also recommended to create a dedicated folder on disk for the next examples. With the help of the file manager tool on your system (Windows Explorer on Windows, Finder on macOS, and Nautilus on Ubuntu), create a folder called VSCode under the root folder, such as C:\VSCode or ~/Library/VSCode. In this folder, you will shortly create new applications.

Creating .NET Core Projects

.NET Core is the cross-platform, open-source, modular runtime from Microsoft to build applications using C#, F#, and Visual Basic that run on Windows, macOS, and Linux distributions. With .NET Core, you can create different kinds of applications such as web applications, Web API REST services, Console applications, and class libraries. Plans are to support desktop technologies as well.

.NET Core ships with a rich command line interface, which provides many options to create different kinds of applications. Discussing all supported project types is not possible here, so you can refer to the official documentation available at <https://dot.net>.

In this section I will show an example based on an ASP.NET Core web application built upon the model-view-controller (MVC) pattern. Creating a .NET Core application is accomplished via the command line. Open a command prompt or a terminal instance on the VSCode folder created previously, depending on your system.

Type the following command to create a new empty folder called HelloWeb:

```
> mkdir HelloWeb
```

Then, move into the new directory. On Windows and Linux, you can type

```
> chdir HelloWeb
```

On macOS, the command is instead cd. Next, type the following command to scaffold a new .NET Core web application using C#:

```
> dotnet new mvc
```

The mvc command line switch specifies that the new web application is based on the MVC pattern and the .NET Core SDK will generate all the plumbing code for some controllers and views. You could also use

the `web` switch and create an empty web application, but having some autogenerated pages will help with describing the debugging features. Once the project has been created, .NET Core will automatically restore NuGet packages for the solution. You could also do this manually by typing the following command:

```
> dotnet restore
```

If you typed `dotnet run`, the application would run in the default web browser. However, the goal is understanding how to run and debug the application in Visual Studio Code. So, open the project folder with VS Code. You can also type `code .` to open Visual Studio Code from the command line. Thanks to the C# extension, VS Code will recognize the presence of the `.csproj` project file, organizing files and folders and enabling all the powerful code editing features you learned previously.

The next step is running the application. As a general rule, in Visual Studio Code you have two options:

- Running the application with an instance of the debugger attached, where a debugger is available for the current project type. In the case of .NET Core, this ships with its own debugger that integrates with VS Code.
- Running the application without an instance of the debugger attached.

Let's start with the second option, and then the debugging features are described in detail in the next section. You can select **Debug ▶ Start Without Debugging**. Visual Studio Code will first start the default build task, and then it will start the application. Figure 9-1 shows the web application scaffolded previously.

CHAPTER 9 RUNNING AND DEBUGGING CODE

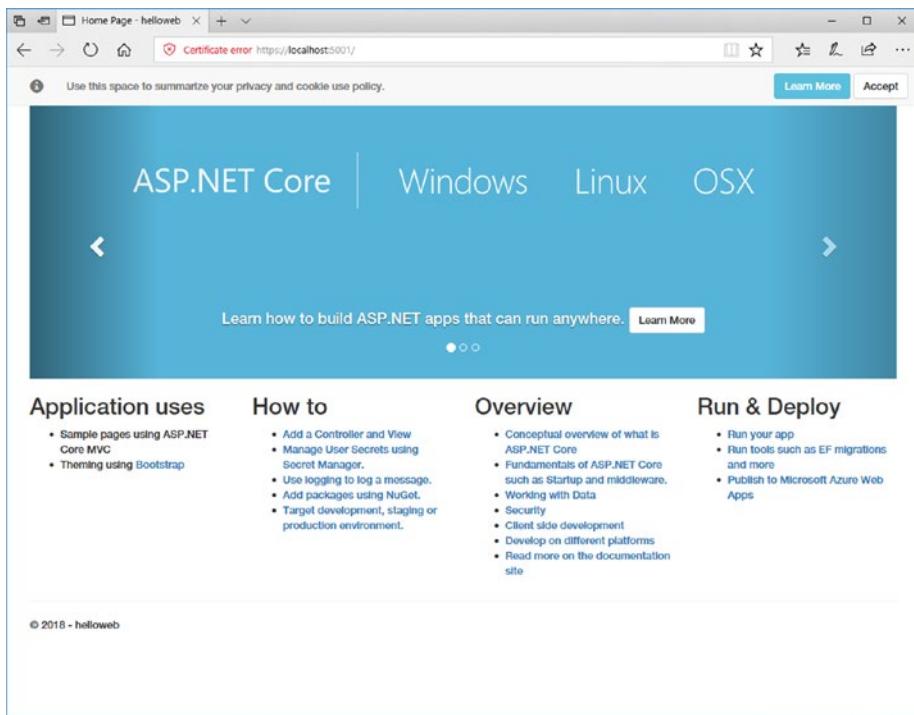


Figure 9-1. The .NET Core web application running

ASP.NET Core web applications use an open-source development server called Kestrel (<https://github.com/aspnet/KestrelHttpServer>), which allows for independency from proprietary systems. By default, Kestrel listens for the application on port 5001, which means your application can be reached at `http://localhost:5001`. The default port setting can be changed inside a file called `launch.json`, which I will discuss more thoroughly in the next paragraphs.

With simple steps, you have been able to create and run a .NET Core project in VS Code that you can certainly edit as you need with the powerful C# code editing features.

Creating Projects on Other Platforms

Obviously, .NET Core is not the only platform you will use with VS Code. Depending on the platform, you will use specific command line tools to scaffold a new project. For example, with Node.js you can use the Express generator which you install with the following command:

```
> npm install -g Express-generator
```

Next, you generate a project with the following line:

```
> Express ProjectName
```

You can then type `code .` to open the new project in Visual Studio Code. Similarly, you will do with other command line tools that allow for generating projects, such as the Yeoman generator, still available for Node.js, and that also allow for generating ASP.NET Core projects and VS Code extensions. For example, you could create mobile apps with the Apache Cordova framework (<https://cordova.apache.org>). Cordova is a JavaScript-based framework, and it works very well with Node.js. Apps you build with Cordova are based on JavaScript, HTML, and Cascading Style Sheets (CSS). First, you can install Cordova with the following command line:

```
> npm install -g Cordova
```

Then you can easily scaffold a Cordova project with the following line:

```
> cordova create MyCordovaProject
```

where `MyCordovaProject` is the name of the new project. Once you have a new or existing Cordova project, you can install the **Cordova Tools** extension for Visual Studio Code (<https://marketplace.visualstudio.com/items?itemName=vsmobile.cordova-tools>). This extension will add support for Cordova projects to the integrated debugger for Node.js, providing specific configurations to target Android and iOS devices, as well as simulators.

Note You will also need some additional specific tools for Cordova, depending on what system you intend to target. For iOS, you will need to install the tools described in the iOS Platform Guide from Apache Cordova (<https://cordova.apache.org/docs/en/latest/guide/platforms/ios/index.html>). For Android, you will need to install the tools described in the Android Platform Guide from Apache Cordova (<https://cordova.apache.org/docs/en/latest/guide/platforms/android/index.html>).

Debugging Your Code

The ability of debugging code is one of the most powerful features in Visual Studio Code and probably the one that makes it a step forward if compared to other code editors. Visual Studio Code ships with an integrated debugger for Node.js applications and can be extended with third-party debuggers. For instance, if you have .NET Core installed, the C# extension for Visual Studio Code detects the availability of a compatible debugger and takes care of attaching it to VS Code.

Let's consider C# and .NET Core as the example on how debugging works, so reopen the HelloWeb folder created previously.

Note All the features discussed in this chapter apply to all the supported debuggers (both built-in and via extensibility), so they are not specific to C# and .NET Core.

The **Debug** view provides a way to interact with the debugger. Figure 9-2 shows how it appears at this point.

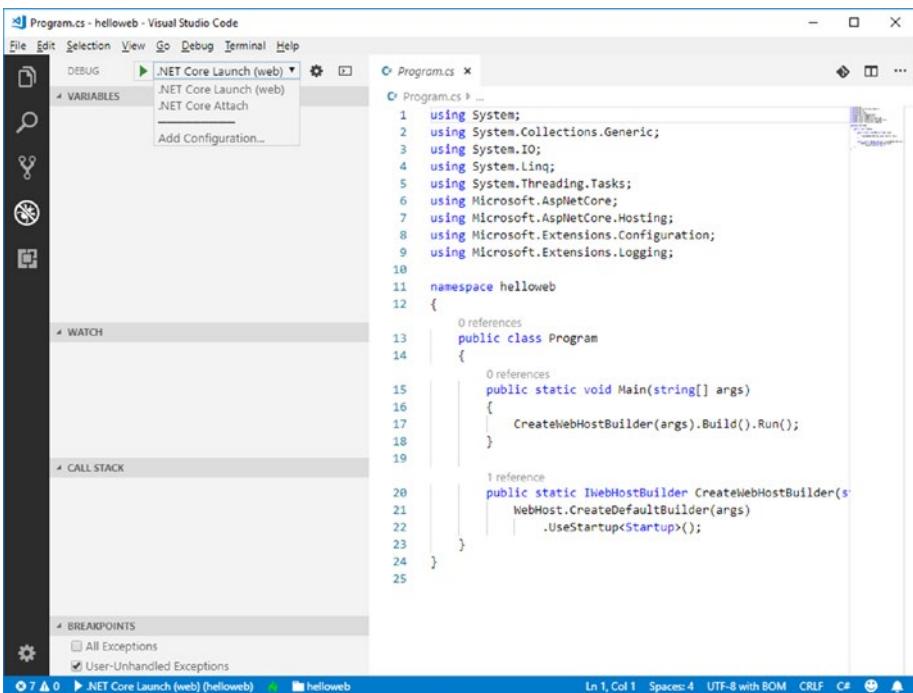


Figure 9-2. The Debug view

At the top of the view, you can see the **DEBUG** toolbar, which provides the following items:

- The **Start Debugging** button, represented with the green play icon. By clicking this button, you will start the application with an instance of the debugger attached.
 - The configuration dropdown box. Here you can select a debugger configuration for running the application.
 - The settings button, represented with the gear icon and whose tooltip says **Open launch.json** (details coming shortly).

- The **Debug Console** button, which opens the Debug Console panel where you see the output messages from the debugger.

After this quick overview, you will now learn about debugger configurations, and then you will walk through the debugging tools available in VS Code.

Configuring the Debugger

Before a debugger can inspect an application, it must be configured. For Node.js and for platforms like .NET Core, where an extension takes care of everything, default configurations are provided. If you take a look at Figure 9-2, you can see how there are two predefined configurations, **.NET Core Launch (web)** and **.NET Core Attach**.

The first configuration is used to run the application within the proper host, with an instance of the debugger attached. For an ASP.NET Core web application like in the current example, the host is the web browser. In the case of a Console application, the host would be the Windows' Console or the Terminal in macOS and Linux. The second configuration can be instead used to attach the debugger to another running .NET Core application.

Note Actually, there is a .NET Core Launch configuration that is different for each kind of application you create with .NET Core. For example, the configuration for Console applications is called **.NET Core Launch (Console)**. The concept to keep in mind is that a Launch configuration is provided to attach an instance of the debugger to the current project.

Debugger configurations are stored inside a special file called **launch.json**. Visual Studio Code stores this file inside the .vscode subfolder, exactly like for tasks.json. This special JSON file contains the markup that instructs Visual Studio Code about the output binary that must be debugged and about the application host. The content of launch.json for the current .NET Core sample looks like the following:

```
{  
    // Use IntelliSense to find out which attributes exist for  
    // C# debugging  
    // Use hover for the description of the existing attributes  
    // For further information visit  
    // https://github.com/OmniSharp/omnisharp-vscode/blob/  
    master/debugger-launchjson.md  
    "version": "0.2.0",  
    "configurations": [  
        {  
            "name": ".NET Core Launch (web)",  
            "type": "coreclr",  
            "request": "launch",  
            "preLaunchTask": "build",  
            // If you have changed target frameworks, make sure  
            // to update the program path.  
            "program": "${workspaceFolder}/bin/Debug/  
netcoreapp2.1/helloworld.dll",  
            "args": [],  
            "cwd": "${workspaceFolder}",  
            "stopAtEntry": false,  
            "internalConsoleOptions": "openOnSessionStart",  
            "launchBrowser": {  
                "enabled": true,  
                "args": "${auto-detect-url}",  
            }  
        }  
    ]  
}
```

```

    "windows": {
        "command": "cmd.exe",
        "args": "/C start ${auto-detect-url}"
    },
    "osx": {
        "command": "open"
    },
    "linux": {
        "command": "xdg-open"
    }
},
"env": {
    "ASPNETCORE_ENVIRONMENT": "Development"
},
"sourceFileMap": {
    "/Views": "${workspaceFolder}/Views"
}
},
{
    "name": ".NET Core Attach",
    "type": "coreclr",
    "request": "attach",
    "processId": "${command:pickProcess}"
}
,
]
}

```

As you can see, the syntax of this file is similar to the syntax of tasks.json. In this case you have an array called configurations. For each configuration in the array, the most important properties are

- name, which represents the configuration friendly name.

- `type`, which represents the type of runtime the debugger is running on.
- `request` (`launch` or `attach`), which determines whether the debugger is attached to the current project or to an external application.
- `preLaunchTask`, which contains any task to be executed before the debugging session starts. Usually, this property is assigned with the default build task.
- `program`, which represents the binary that will be the subject of the debugging session.
- `launchBrowser`, where operating system-specific properties contain the command that will be executed to start the application.
- `env`, which represents the environment. In the case of .NET Core, a value of `Development` instructs VS Code to run the Kestrel development server.

If you wanted to implement custom configurations, `launch.json` is the place where you would add them. Because these two configurations, and more generally default configurations, are enough for most of the common needs, custom configurations will not be covered in this book. The documentation provides additional details about this topic (https://code.visualstudio.com/docs/editor/debugging#_add-a-new-configuration).

Note If you click the **Add Configuration** command in the configuration dropdown box, you will be able to select from a built-in list of configurations that you can add to `launch.json`. This can be useful especially in those cases where VS Code should detect a project type and its configuration, but actually doesn't.

Managing Breakpoints

Before starting a debugging session, it is useful to place one or more breakpoints to discover the full debugging capabilities in VS Code. You place breakpoints by clicking the white space near the line number. For instance, place a breakpoint on line 19 of the Startup.cs file, as shown in Figure 9-3.

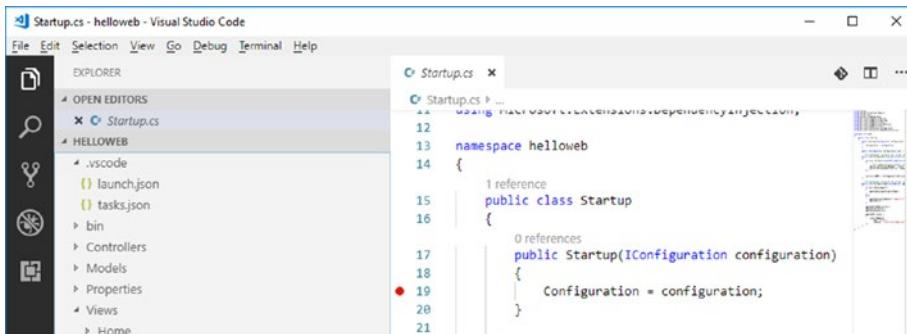


Figure 9-3. Adding breakpoints

You can remove a breakpoint by simply clicking it again, or you can manage breakpoints in the **Breakpoints** area of the Debug view (see Figure 9-4).

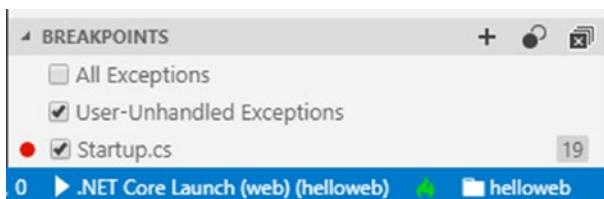


Figure 9-4. Managing breakpoints

Here you can see the list of files that contain any breakpoint and the line numbers. You can also cause the debugger to break on user-unhandled exceptions (default) and on all exceptions. You can click the **Add Function Breakpoint** (+) button. Instead of placing breakpoints

directly in source code, a debugger can support creating breakpoints by specifying a function name. This is useful in situations where source is not available but a function name is known.

Debugging an Application

Now it is time to start a debugging session, so that you will be able to see in action all the debugging tools and make decisions when breakpoints are hit. In the **Debug** view, make sure the **.NET Core Launch (web)** configuration is selected, then click the **Start** button or press **F5**. Visual Studio Code will launch the debugger, and it will display the output of the debugger in the **Debug Console** panel.

It will also break when it encounters an exception or a breakpoint, like in the current example. Figure 9-5 shows Code hitting a breakpoint and all the debugging instrumentation.

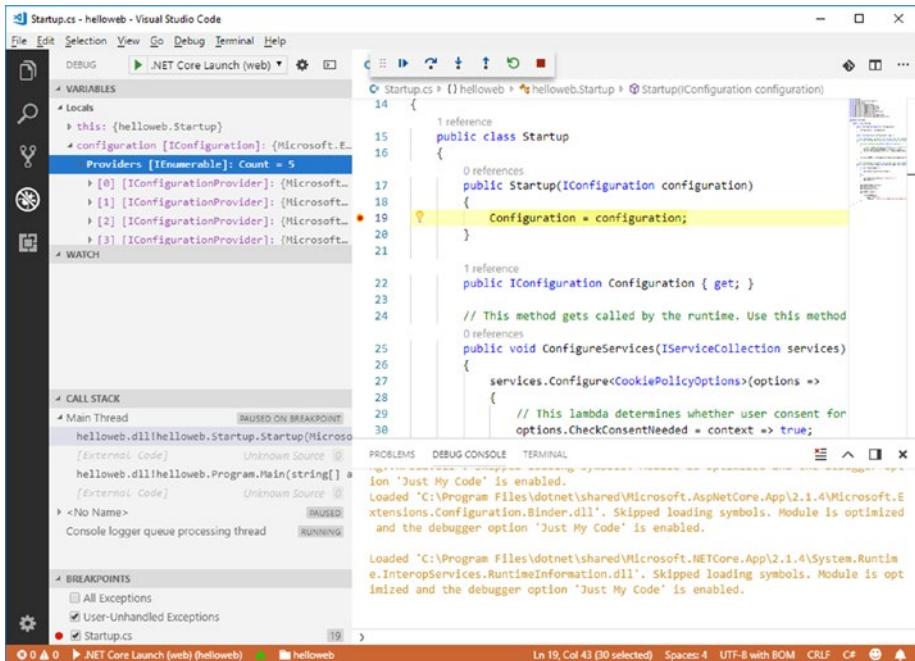


Figure 9-5. The debugging tools while a breakpoint is being hit

Notice how the status bar becomes orange while debugging and how the **Debug Console** window shows information about the debugging process. On the left side, the Debug view shows a number of tools:

- **VARIABLES**, which shows the list of variables that are currently under the debugger control and that you can investigate by expanding each variable.
- **WATCH**, a place where you can evaluate expressions.
- **CALL STACK**, where you can see the stack of method calls. If you click a method call, the code editor will bring you to the code that is making that call.
- **BREAKPOINTS**, where you can manage breakpoints.

At the top of the window, also notice the debugging toolbar (see Figure 9-5) called Debug action pane, made of the following commands (from left to right):

- **Continue**, which allows continuing the application execution after breaking on a breakpoint or an exception
- **Step Over**, which executes one statement at a time except for method calls, which are invoked without stepping into
- **Step Into**, which executes one statement at a time, including statements within method bodies
- **Step Out**, which executes the remaining lines of a function starting from the current breakpoint
- **Restart**, which you select to restart the application execution
- **Stop**, which you invoke to stop debugging

These commands are also available in the Debug menu, together with their keyboard shortcuts. If you hover a variable name in the code editor,

a convenient popup will make it easy to investigate values and property values (depending on the type of the variable), as shown in Figure 9-6 where you can see a popup showing information about the Configuration variable. You can expand properties and see their values, and you can also investigate properties in the **VARIABLES** area of the Debug side bar.

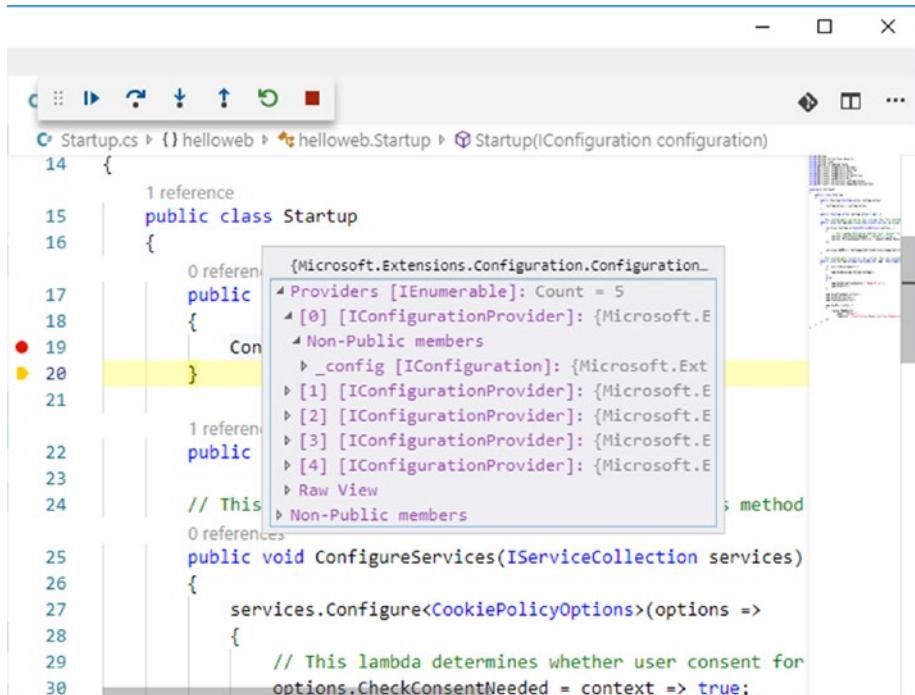


Figure 9-6. Investigating property values at debugging time

Evaluating Expressions

You have an option to use the **Watch** tool to evaluate expression. While debugging, click the **Add Expression** (+) button in the **Watch** box, then type the expression you want to evaluate. For instance, if you type `Configuration != null`, the Watch tool will return true or false depending if the object has an instance or not. Figure 9-7 shows an example.

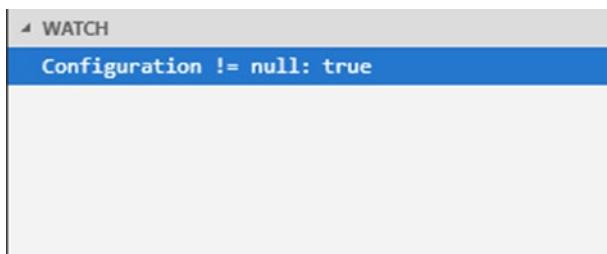


Figure 9-7. Evaluating expressions

The Call Stack

The debugger also offers the **Call Stack** feature, which allows stepping through the hierarchy of method calls. When you click a method call in the stack, the code editor will open the containing file, highlighting the method call (see Figure 9-8).

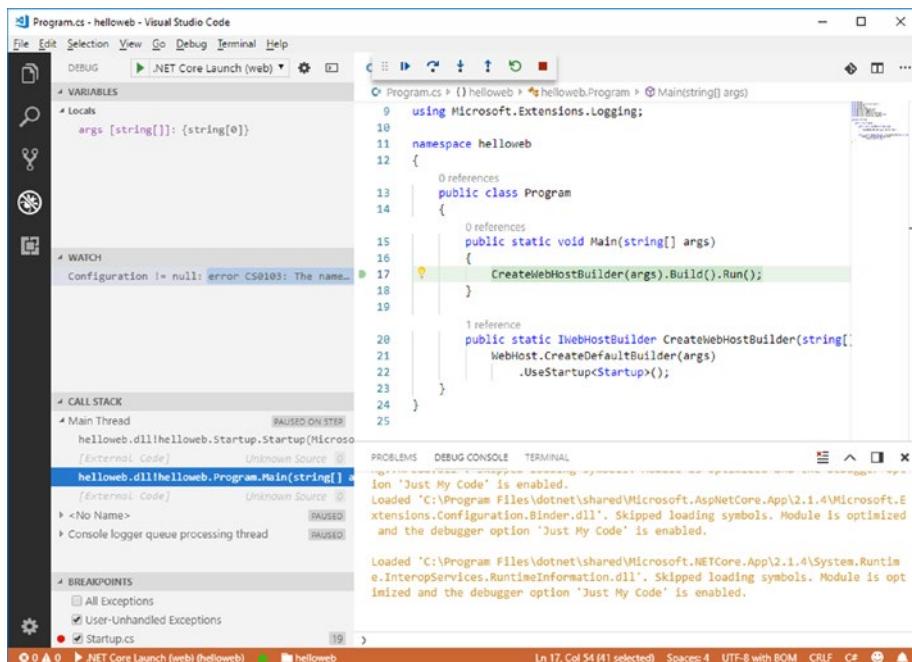


Figure 9-8. Walking through method calls

The code editor can highlight method calls only if it is part of the source code, but this feature is very useful especially when you encounter errors and you need to step back through the code.

The Debug Console Panel

The Debug Console is certainly the place where VS Code shows the debugger output but, as the name implies, it is also an interactive panel where you can evaluate expressions. You can type the expression near the `>` symbol and then press Enter.

Figure 9-9 shows an example that evaluates if the Configuration variable is not null.



A screenshot of the VS Code interface focusing on the Debug Console panel. The tab bar at the top shows 'PROBLEMS', 'DEBUG CONSOLE' (which is selected and highlighted in blue), and 'TERMINAL'. Below the tabs, the console window displays the following text:
configuration != null
true

Figure 9-9. Evaluating expressions in the Debug Console panel

Supporting Azure, Docker, and Artificial Intelligence

Microsoft has made many investments in the last couple of years to add to Visual Studio Code support for the most modern technologies and development scenarios. In fact, Microsoft has developed several extensions that allow for integrating with Microsoft Azure, Docker, and artificial intelligence services.

About Azure, you might want to consider the following extensions:

- Azure Functions, which allows for developing Azure functions in VS Code and publishing to Azure directly from the environment
- Azure App Service, which allows for deploying and scale web and mobile apps to Azure directly from VS Code
- Azure CLI Tools, which provides interaction with the Azure command line interface from Visual Studio Code

Obviously, you need an active Azure subscription to use these extensions. Not limited to this, Microsoft has developed a Docker extension, which not only brings syntax highlighting for Docker files but that also adds commands and support to create and publish containerized applications to Azure. As you can understand, Azure is at the core of Microsoft's business, and this includes artificial intelligence services available on the cloud. For this reason, Microsoft has also developed an extension called Visual Studio Code Tools for AI, which allows for building, testing, and deploying deep learning and other AI solutions. For developers using Python, this extension also makes it easier to consume AI services with this language. The official documentation provides detailed tutorials that help address these particular development scenarios, more specifically you can read

- Deploying Applications to Azure (<https://code.visualstudio.com/docs/azure/deployment>)
- Working with Docker (<https://code.visualstudio.com/docs/azure/docker>)
- Visual Studio Code Tools for AI (<https://github.com/Microsoft/vscode-tools-for-ai>)

Visual Studio Code, with its extensibility model and being independent from proprietary systems, can target an incredible number of development scenarios, from web to mobile to cloud.

Summary

The power of Visual Studio Code as a development environment comes out when you work with real applications. With the help of specific generators, you can easily generate .NET Core projects using C# or Node.js projects. This chapter described how you can leverage a powerful, built-in debugger that offers all the necessary tools you need to write great apps, such as breakpoints, variable investigation, call stack, and expression evaluators.

You finally saw how VS Code can target advanced scenarios such as deploying applications and functions to Azure, packaging Docker images, and consuming artificial intelligence services.

By completing this chapter, you have walked through all the most important and powerful features you need to know to write great cross-platform applications using Visual Studio Code.

Index

A, B

- Activity bar, 24–25
- Apache Cordova framework, 195
- Auto-detecting tasks, 164–165
- Azure DevOps, 150–154

C

- Code block folding, 51–52
- Code editing features
 - breadcrumbs, 57–58
 - built-in code snippets, 52–54
 - code block folding, 51–52
 - delimiter matching, 51
 - Markdown preview, 58–59
 - Minimap mode, 55–56
 - multicursors, 52
 - syntax colorization, 50
 - text manipulation and text selection, 48–49
 - whitespace rendering, 56–57
 - word completion, 54–55
- Code editor, 21–22
- Code refactoring, 78
- Code snippets, 53–54
- Color theme, 97
- Command Palette, 37

Customizations

- and extensions, 95–97
 - keyboard shortcuts
 - adding, 108
 - commands and actions, 106
 - keybindings.json file, 108
 - list of commands, 107
 - new, 109
 - theme selection
 - color themes, 97
 - Dark (Visual Studio), 98
 - user settings (*see* User settings)
 - workspace settings, 105–106
- Customize group, 20

D

- Dark (Visual Studio), 98
- Debug bar, 34–35
- Debug Console panel, 41
- Debugger
 - adding breakpoints, 202
 - Call Stack, 206–207
 - commands, 204
 - configuration
 - commands, 199–201
 - Debug Console panel, 207
 - Debug view, 196–197

INDEX

- Debugger (*cont.*)
 evaluate expression, 205–206
 .NET Core Launch (web), 198
 tools, 203–204
- Delimiter matching, 51
- E**
- Editor windows, 22
- Evolved code editing
- code issue detection
 - adding missing directives, 73
 - code refactoring, 68, 72
 - generating types, 71
 - IDisposable, 72, 73
 - interface with dispose pattern, 74–75
 - light bulb, 69, 72, 74
 - potential fixes, 70
 - find all references, 65
 - Go To Definition, 63–64
 - identifier, 68
 - inline documentation with Tooltips, 62–63
 - IntelliSense, 60–61
 - parameter hints, 62
 - peek definition, 66
 - renaming symbols, 67
- Explorer bar, 26–28
- Extensibility, 111
- Extensions
- authoring, 122
 - customizing options, 120–121
 - Git History, 142–144
- GitHub Pull Requests, 148–150
- GitLens, 144, 146–147
- installation, 111, 113–114
- recommendations, 115–116
- shortcuts, 119
- Visual Studio
- Marketplace, 111–112
- Extensions bar, 35–36
- F**
- Find All References
- feature, 65, 67
- Folders and projects
- extensibility, 84
 - files, 82–83
 - JavaScript project, opening, 87
 - loose assortments of files, 88
 - .NET Core solution, 86
 - opening folder, 84–85
 - structured view, 85
 - TypeScript projects, 88
- Free Pascal compiler, 166
- G**
- Git
- Command Palette, 137–138
 - file changes, 132–134
 - local repository, 128–129
 - manage commits, 135–137
 - remote repository, 130–131
 - staging changes, 134–135

Git bar, 32–33
 GitLens, 144
 Go To Definition, 63–64

H

Help group, 20
 https_proxy environment variables, 104

I

Individual files
 creation, 79–80
 editing window, 78
 encoding, 80–81
 Go to Line item, 81–82
 line terminator, 81
 IntelliSense, 60–61, 78, 103

J

jsconfig.json files, 83

K

Kestrel server, 194
 Keyboard shortcut, 45, 109

L

Language support, editing features, 46–47
 Learn group, 21
 Light Bulb, 69–70, 72–74, 76

M

Markdown syntax, 58
 Microsoft Azure, 208–209
 Minimap mode, 55–56
 Model-view-controller (MVC), 192
 MSBuild solution files (.sln), 82
 Multitor-master folder, 90
 Multicursors, 52

N

Navigating between files, 36
 .NET Core
 creation, 192
 MVC, 192
 running, 194
 solution, 86
 Node.js, 157

O

OmniPascal extension, 166
 Outline view, 28–30
 Output panel, 40

P, Q

package.json files, 83
 Panels area, 38
 Debug Console panel, 41
 output panel, 40
 problems panel, 38–39
 terminal panel, 42–43

INDEX

Parameter hints, [62](#)
Peek Definition, [66](#)
Problems panel, [38–39](#)
project.json files, [83](#)
Proxy Strict SSL, [104](#)

R

Recent group, [20](#)
Recommended
 extensions, [115, 117–118](#)

S

Search settings, [100](#)
Search tool
 Clear Search Results, [31](#)
 Replace All, [32](#)
 Replace text, [32](#)
Selection menu, [49](#)
Settings button, [36](#)
settings.json file, [101–103](#)
Side bar
 debug bar, [34–35](#)
 explorer bar, [26–28](#)
 extensions bar, [35–36](#)
 git bar, [32–33](#)
 outline view, [28–30](#)
 search tool, [31–32](#)
 settings button, [36](#)
Source control managers
 (SCM), [126](#)
Start group, [20](#)

Status bar, [23–24](#)
Syntax colorization, [50](#)

T

Task
 building, [164](#)
 commands, [161](#)
 compiling Pascal source
 code, [167–172](#)
 customization properties,
 [175–176, 178](#)
 default build task, [173–175](#)
 Free Pascal compiler, [173](#)
 JSON notation, [158–159](#)
 key and values, [178–179](#)
 MSBuild solution, [185–187](#)
 operating system-specific
 properties, [181–182](#)
 problem matchers, [172, 187–188](#)
 running, [159–160](#)
 templates, [182–183, 185](#)
 terminal panel, [163–164, 188](#)
 types, [159](#)
 variable, [180–181](#)
Terminal panel, [42–43](#)
Toggle Block Comment, [48](#)
Toggle Breadcrumbs, [57](#)
Toggle Line Comment, [48](#)
Toggle Render Whitespace, [56](#)
Tooltips, [63](#)
tsconfig.json files, [83](#)
TypeScript compiler (tsc), [157](#)

U

User interface, 19, 21, 37–38
 User settings
 changing, 100
 default, 102
 editor, 99, 101
 explorer, 100
 IntelliSense, 103
 Minimap mode, 102
 proxies, 103–104
 search, 100
 settings.json file, 101–103

V

Visual Studio Code, 77
 automating tasks, 4
 branch
 creation, 139
 deleting, 141
 merging, 140
 switching, 140
 browser, 5–6
 built-in debugger, 3
 built-in support, 3
 code-centric tool, 4
 color themes, 2
 cross-platform development
 tool, 2
 definition, 2
 download for macOS, 9

download for Windows, 7
 end-to-end development, 2
 features, 3–5
 insiders builds, 15–17
 installation
 macOS, 9–10
 Ubuntu, 10–11
 Windows, 6–9
 localization, 12–13
 needs, 2
 SCM providers, 125–127
 updation, 13–15
 user installer, 7
 version control, 4

Visual Studio Marketplace, 111–112

W, X

Windows Presentation Foundation
 (WPF), 86, 185, 186
 Word completion, 54–55
 Workspace
 creation, 91
 multiple projects and
 folders, 89–90
 opening, 92
 settings, 105–107
 structure, 92–93

Y, Z

Yeoman tool, 122