Visual Studio Code Distilled

Evolved Code Editing for Windows, macOS, and Linux

—

Second Edition

—

Alessandro Del Sole

Apress®

Visual Studio Code Distilled

Evolved Code Editing for Windows, macOS, and Linux

Second Edition

Alessandro Del Sole

Apress®

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

Alessandro Del Sole
Cremona, Italy

ISBN-13 (pbk): 978-1-4842-6900-8 ISBN-13 (electronic): 978-1-4842-6901-5
https://doi.org/10.1007/978-1-4842-6901-5

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

Cover image designed by Freepik (https://www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media LLC, 1 New York Plaza, Suite 4600, New York, NY 10004. Phone 1-800-SPRINGER, fax (201) 348-4505, email orders-ny@springer-sbm.com, or visit https://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 booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

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 https://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/9781484269008. For more detailed information, please visit https://www.apress.com/source-code.

Printed on acid-free paper

To my wife Angelica, you mean everything to me.

Table of Contents

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 everyone else at Apress for the opportunity and the great teamwork on this book.

Special thanks to the technical editor, Damien Foggon, who contributed to the quality and accuracy of the contents.

Special thanks to my wife 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 life cycle from coding to debugging to team collaboration.

With .NET 5 recently released and with .NET MAUI coming shortly, Visual Studio Code becomes even more important to support cross-platform development on multiple operating systems. 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 life cycle with a built-in debugger and integrated support for the popular Git version control engine.

With Visual Studio Code, you can work with individual code files or with folders containing projects or loose files. This chapter provides an introduction to Visual Studio Code, giving you information on when and why you should use it and details 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.

Visual Studio Code, a Cross-platform Development Tool

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 definitely a code-centric tool, which not only makes editing code files and folder-based project systems easier but also facilitates writing cross-platform web, mobile, and cloud applications over the most popular

A. Del Sole, *Visual Studio Code Distilled*, https://doi.org/10.1007/978-1-4842-6901-5_1

platforms, such as Node.js and .NET 5 (including earlier versions of .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. Visual Studio Code is therefore a complete development tool, rather than being a simple code editor. For a richer development experience, you will want to consider Microsoft Visual Studio 2019 on Windows and Visual Studio 2019 for Mac on macOS, but Visual Studio Code can be really helpful in many situations.

In this book, you'll learn how to use Visual Studio Code and how to get the most out of it; you'll discover how you can use it both as a powerful code editor and as a complete environment for end-to-end development. Except where necessary to differentiate operating systems, figures are based on Microsoft Windows 10, but typically there is no difference in the interface on Linux and macOS. Also, Visual Studio Code includes a number of color themes that style its layout. In this book, figures display the Light (Visual Studio) theme, so you might see different colors on your own screen if you choose a different color theme. Chapter 5 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, explore the features it offers, and discover 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 to make it easier to write code for web, mobile, and cloud platforms for any developers working on Windows, Linux, or macOS, providing independence from proprietary development environments.

For a better understanding of the nonproprietary nature of Visual Studio Code, 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 2019 would make you dependent on that specific integrated development environment (IDE). This also applies to the (free) Visual Studio 2019 Community edition. Conversely, though Visual Studio Code certainly is not intended to be a replacement for more powerful and complete environments, it 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, such as C# and JavaScript, 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 5) via extensibility
- Version control based on the popular Git version control system, 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, Visual Studio 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'll 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 best 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 command-line interface (CLI), a compiler, and a debugger. If you use a language

for which there is no extensive support (e.g., the open source Go programming language, https://golang.org), Visual Studio Code is not able to invoke a compiler. You can work around this by implementing task automation, discussed in Chapter 8, 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 working with the powerful graphical tools available in Microsoft Visual Studio 2019.
- Visual Studio Code 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 2019 or Microsoft Visual Studio 2019 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 elevate it above 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.

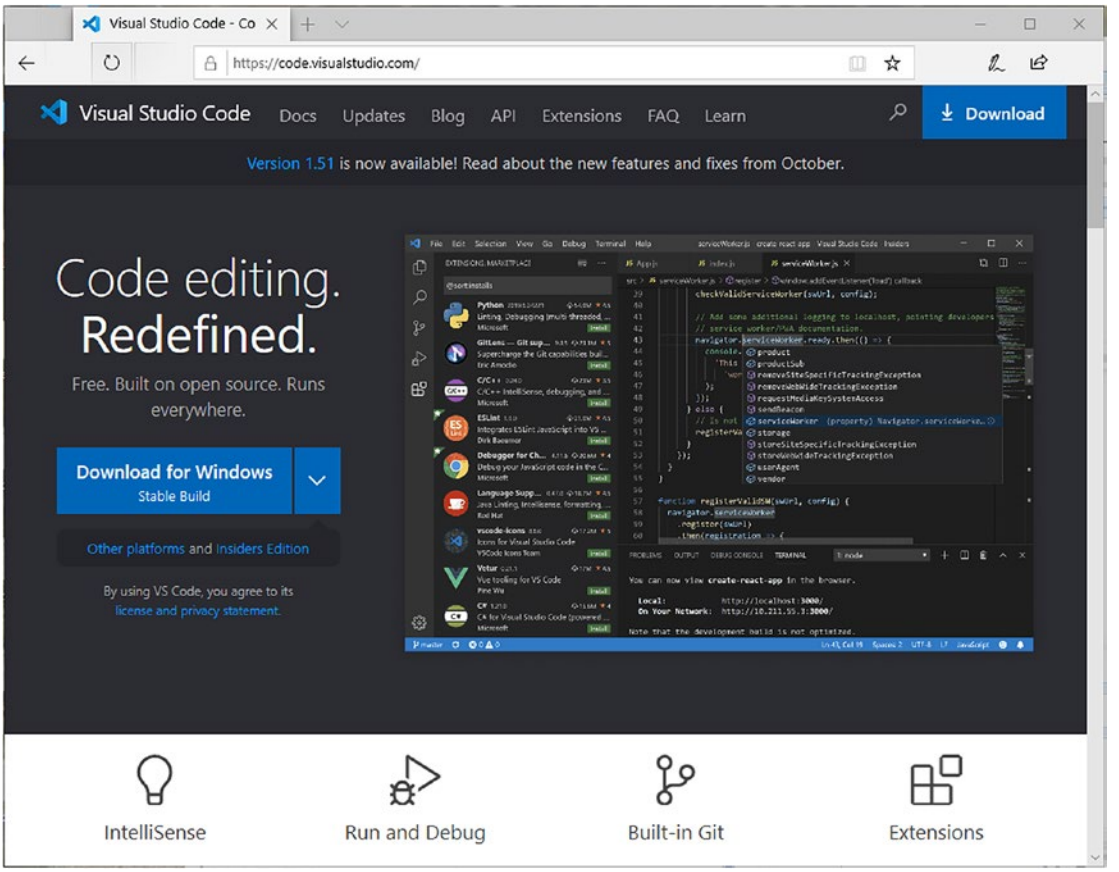

Figure 1-1. *The download page for Visual Studio Code*

Note Visual Studio Code can also run in Portable Mode, which means that you can create a self-containing folder that can be moved across environments. Since this is a very specific scenario, it isn't covered in this book; you can read the documentation (https://code.visualstudio.com/docs/editor/portable) to learn the steps required to generate Portable Mode.

In the following sections, you will learn tips for installing Visual Studio Code on the various supported systems.

Note The latest stable release of Visual Studio Code at the time of this writing is version 1.56, released in April 2021.

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-in 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 User Installer. If you instead wish to download the system-level installer, go to `https://code.visualstudio.com/download` and select the System Installer download that best fits your system configuration (32 or 64 bit, or ARM).

Once the download has been completed, launch the installer and simply follow the guided procedure that is typical of most 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 rightclicking 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\username\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.

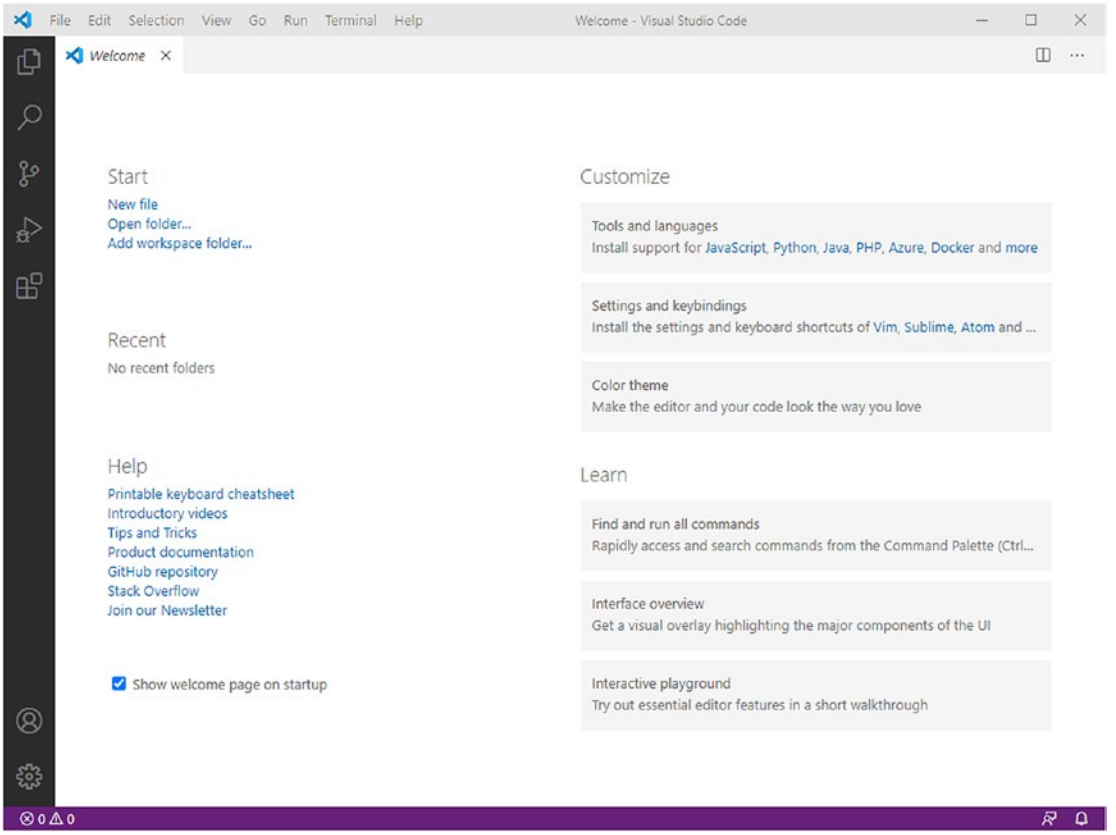

Figure 1-2. *Visual Studio Code running on Windows*

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.

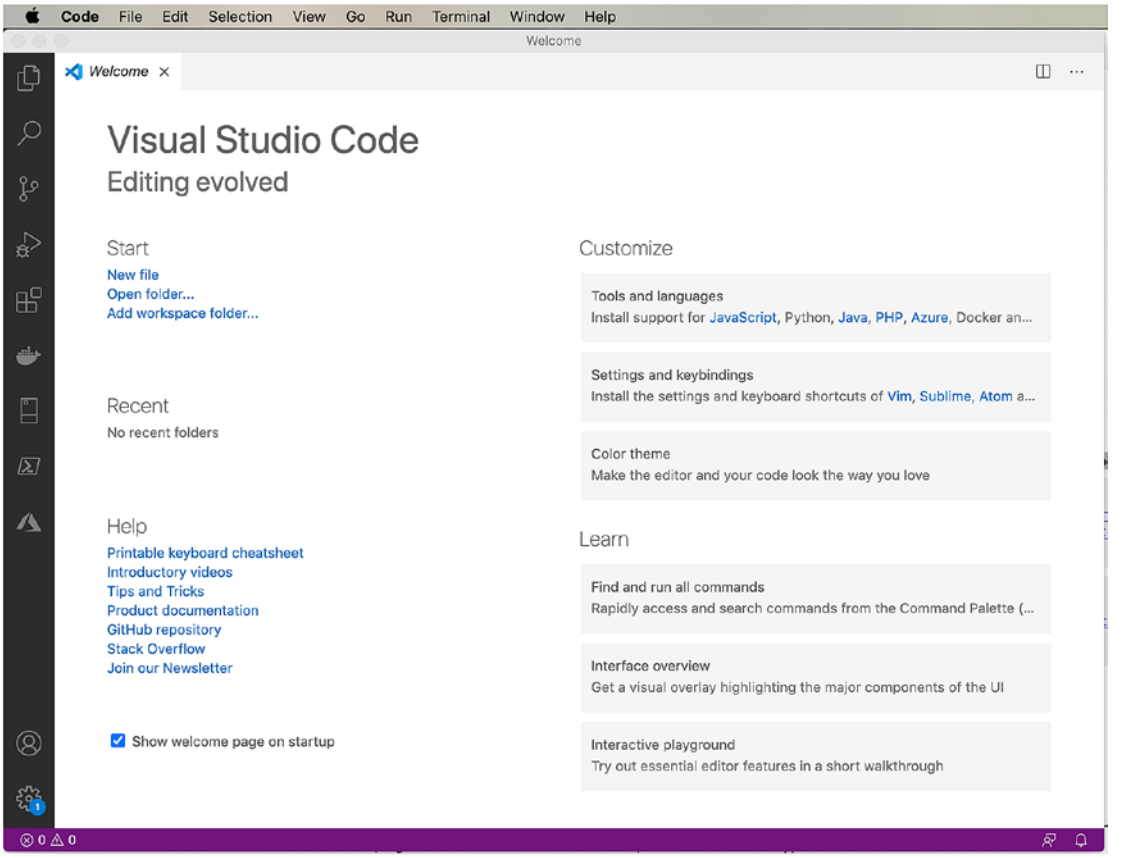

Figure 1-3. *Visual Studio Code running on macOS*

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 need the .deb installer. For the Red Hat Linux, Fedora, and SUSE distributions, you 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 Visual Studio Code is installed, simply 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.

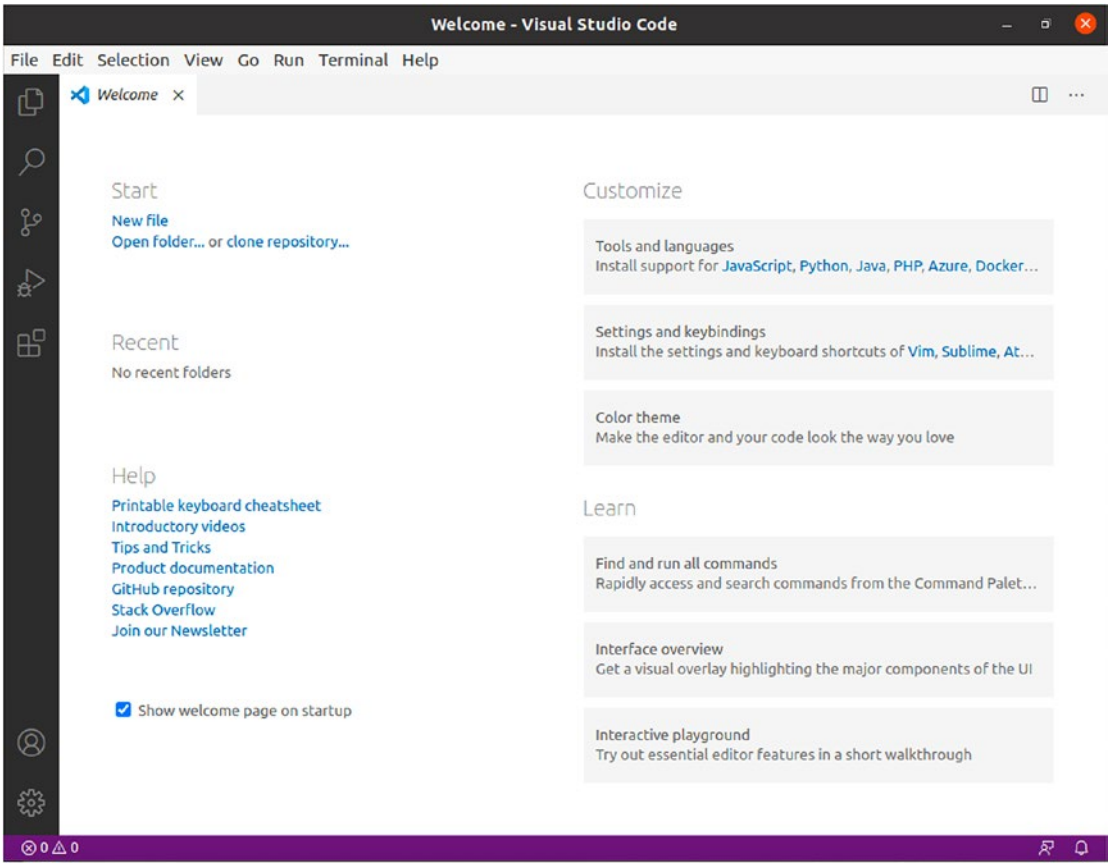

Figure 1-4. *Visual Studio Code running on Ubuntu*

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 (`https://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 pop-up message 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
```

You can also just type `display` and the command will be automatically listed in the command palette (see Figure 1-5).

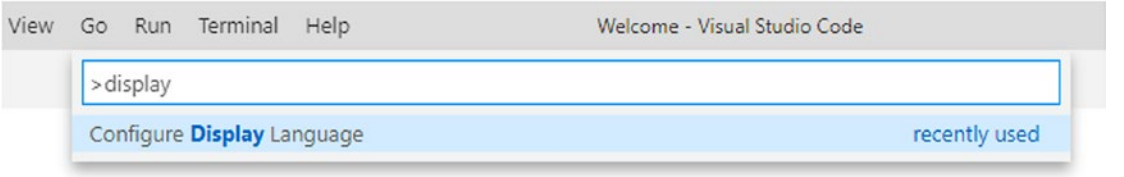

Figure 1-5. *Invoking the command to change the localization*

Note The Command Palette will be discussed thoroughly in Chapter 2.

When you click this command, the Command Palette displays two options:

- **en**, which allows for selecting American English as the culture. This is the default localization and is always available.
- **Install additional languages**, which allows for installing additional language packs built by Microsoft.

When you click **Install additional languages**, VS Code shows a list of available language packs, as you can see in Figure 1-6.

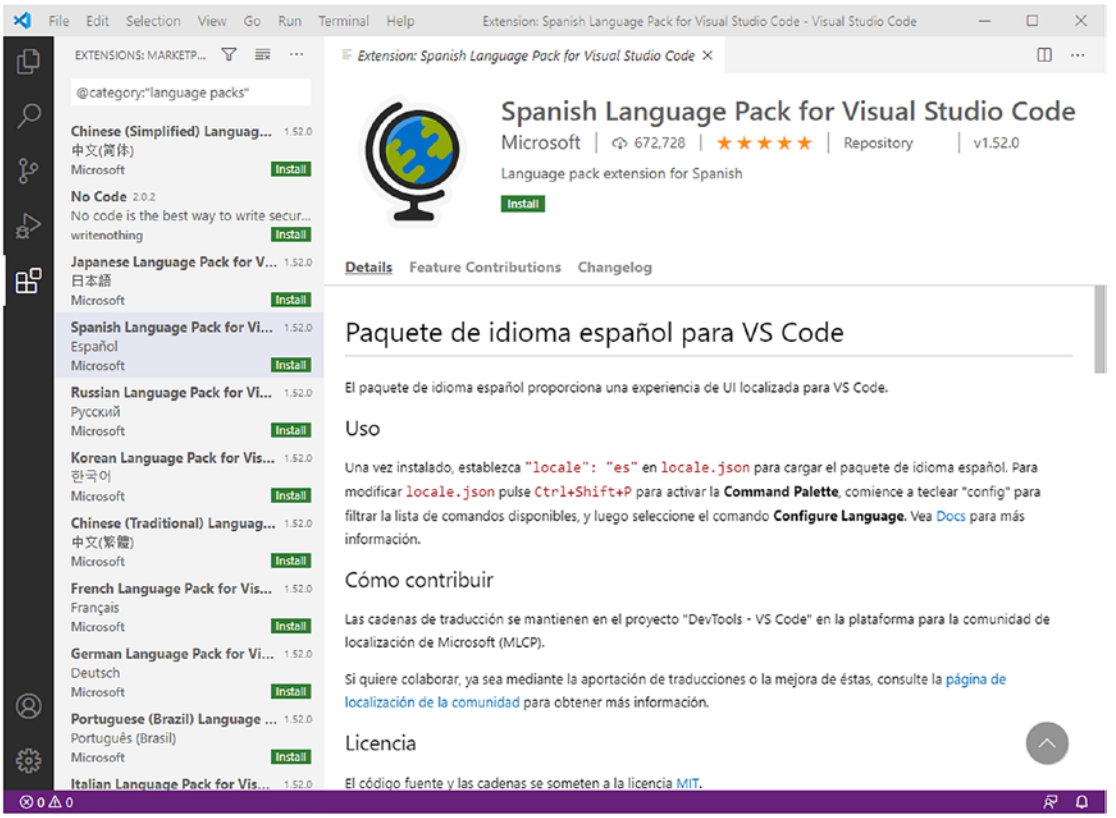

Figure 1-6. *Installing language packs*

Select the language pack to see a localized description, then click the **Install** button. Visual Studio Code's user interface will then be localized at restart, based on your selection.

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 of the **Application** settings group, disable the background updates option. Figure 1-7 shows an example based on Windows. (Obviously, on macOS and Linux, the **Enable Windows Background Updates** option is not available.)

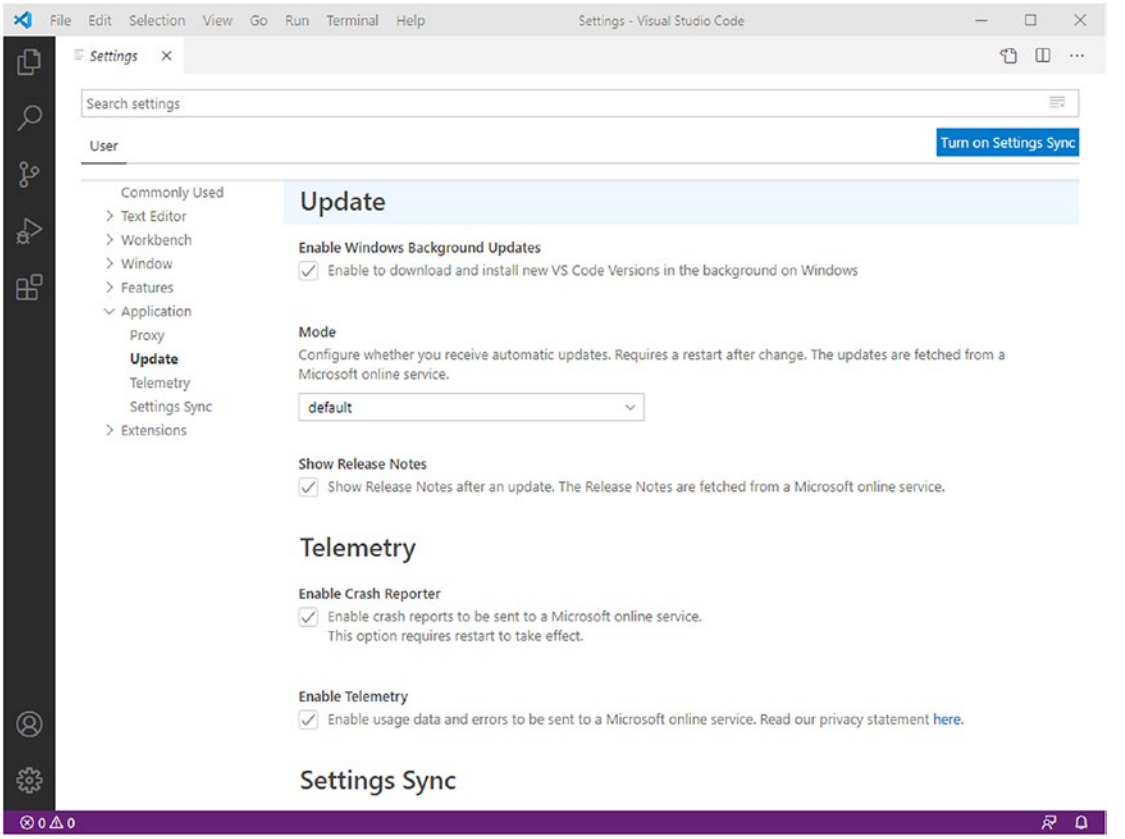

Figure 1-7. *Disabling automatic updates*

You follow the same steps to re-enable updates in the background. Whenever Visual Studio Code receives an update, you will receive a notification suggesting that you restart Code in order to apply changes. The first time you restart Visual Studio Code after an update, you will see the release notes for the version that was installed, as demonstrated in Figure 1-8.

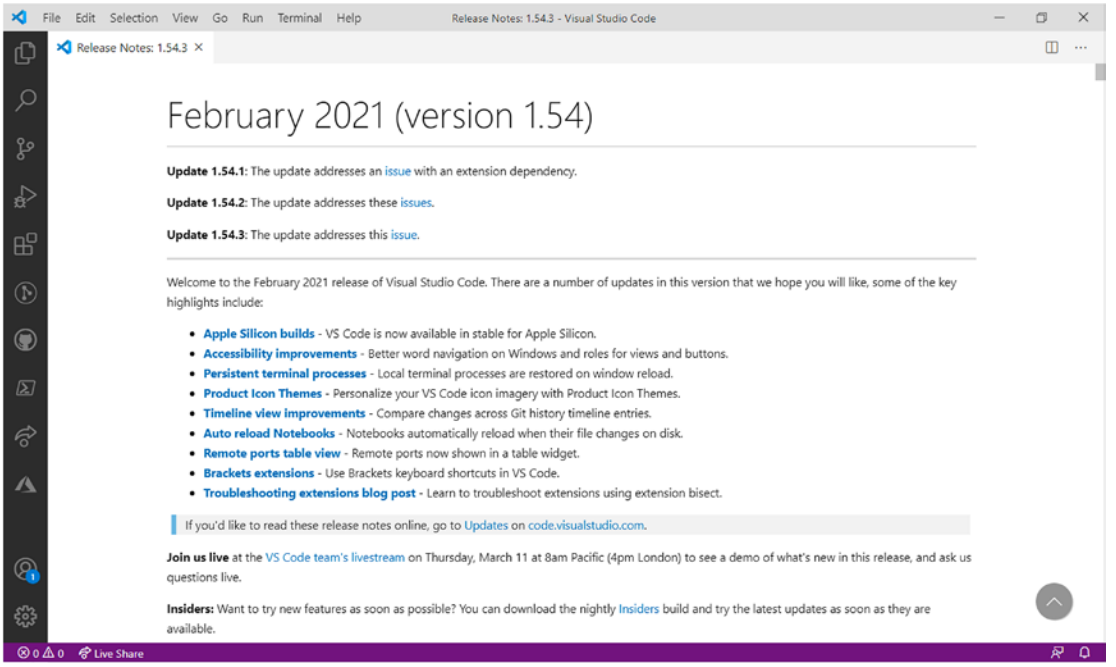

Figure 1-8. *VS Code release notes*

Release notes contain the list of new and updated features, as well as hyperlinks that will open the proper feature page in the documentation. You can recall release notes at any time from **Help ➤ Release Notes**.

Previewing Features with Insiders Builds

By default, the download page of the Visual Studio Code's website 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.

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 1-9).

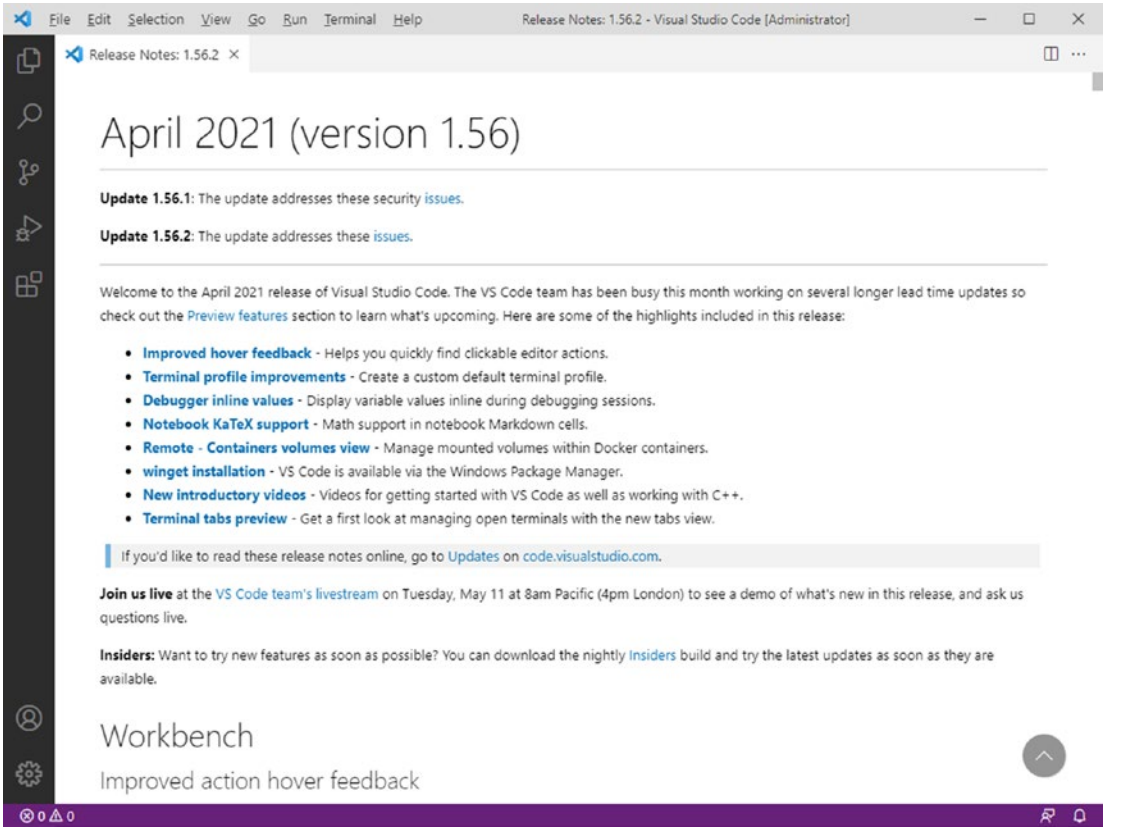

Figure 1-9. *Visual Studio Code Insiders builds*

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 builds, 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, you will see all the amazing code editing features that make Visual Studio Code a rich, powerful crossplatform 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 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, the Panels area, and the Side Bar. This chapter explains how the user interface is organized and how you can be most productive using 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 explore the features discussed in this chapter. Then, Chapter 4 discusses more thoroughly how you can work with individual files and multiple files, in one or more languages, concurrently.

A. Del Sole, *Visual Studio Code Distilled*, https://doi.org/10.1007/978-1-4842-6901-5_2

The Welcome Page

At startup, Visual Studio Code displays the Welcome page, as shown in Figure 2-1.

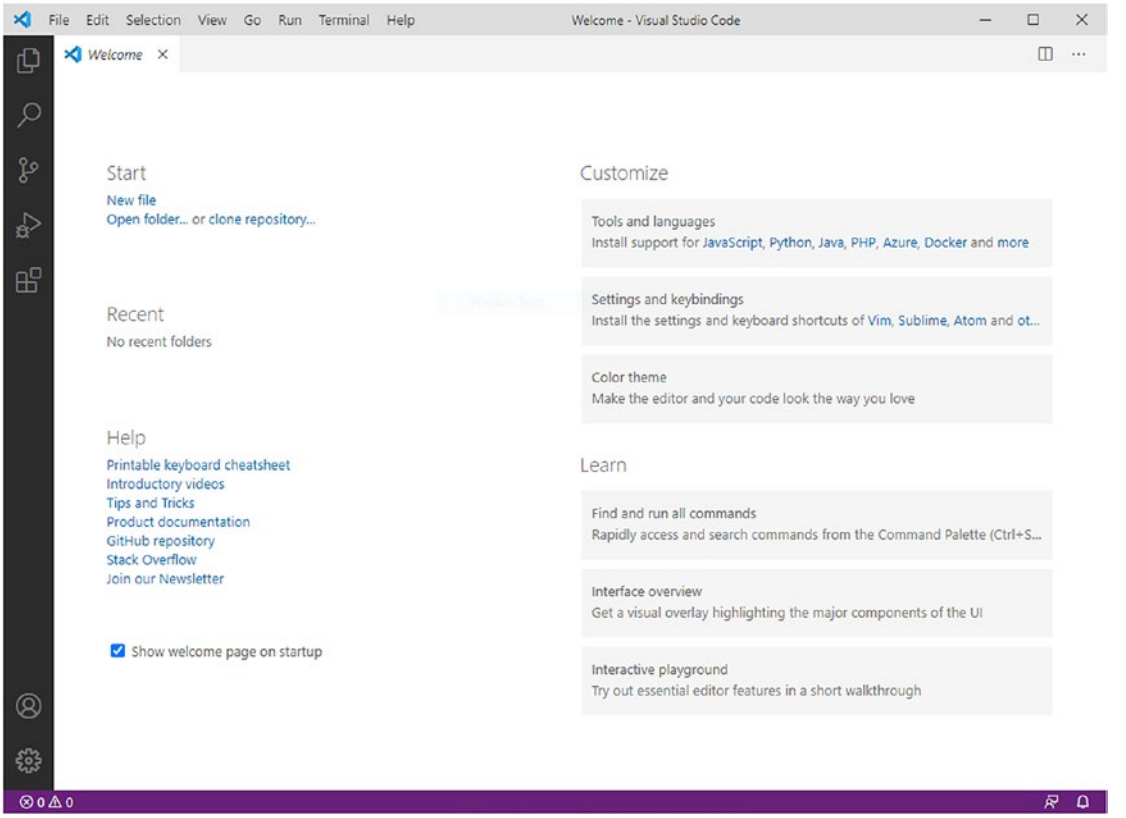

Figure 2-1. *The Welcome page*

On the left side of the page, under the **Start** group, you find shortcuts for creating and opening files and folders, and for cloning an existing Git repository. Under the **Recent** group is a list of recently opened files and folders that you can 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 of the Welcome page, under the **Customize** group, you can find shortcuts to customize Visual Studio Code by installing extensions, changing keyboard shortcuts, and changing color themes. Under the **Learn** group are additional shortcuts to learning resources about commands and the user interface.

Most of the features highlighted in the Welcome page are discussed throughout this book. By default, the Welcome page is set to show up every time you launch Code. To change this default behavior, remove the check mark from the **Show welcome page on startup** check box. To re-enable the Welcome page on startup, click **Help ➤ Welcome** and add the check mark back.

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 or edit multiple files side by side concurrently. Figure 2-2 shows an example of the latter.

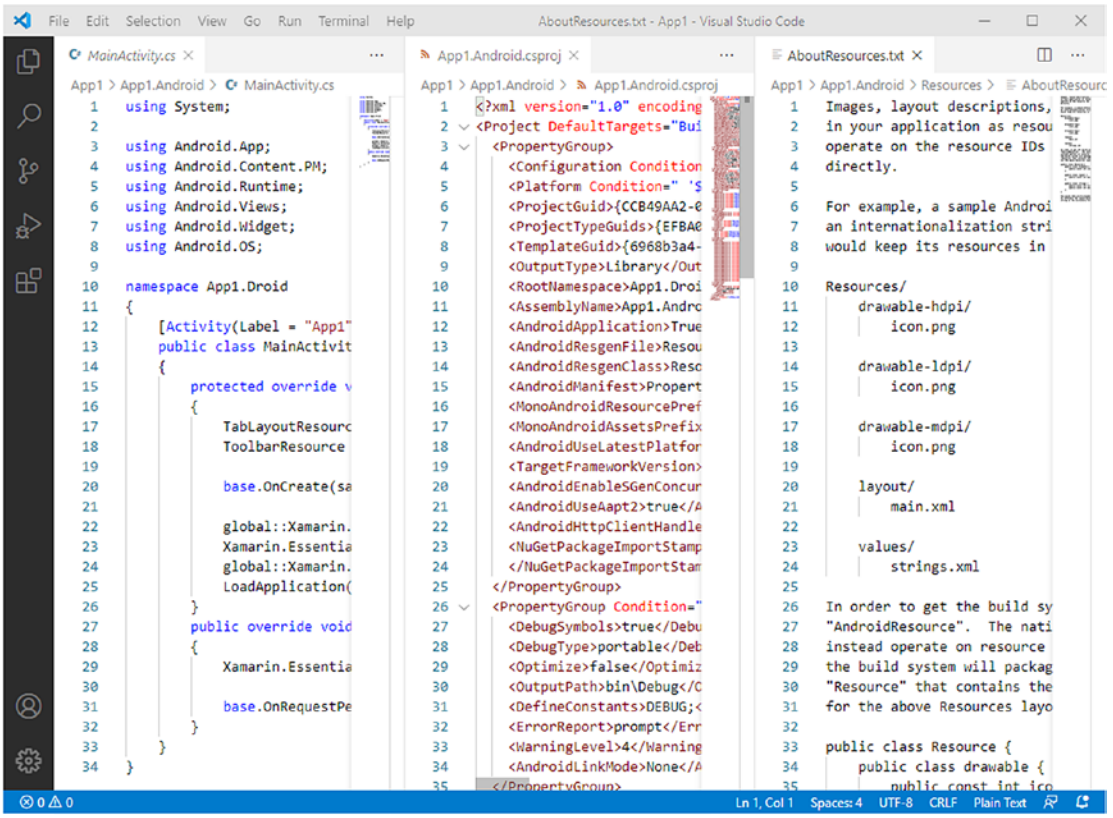

Figure 2-2. *The code editor and multiple file views*

To do this, you have a couple options:

- Right-click a file name in the Explorer bar and then select **Open to the Side**.
- Ctrl-click a file name in the Explorer bar. This is discussed in the section "The Side Bar" later in this chapter.
- 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. Open editors can also be organized into groups. To accomplish this, you can drag and drop the title of an open editor close to another one and they will be both grouped in the same space and the Explorer bar will show the list of groups. You can quickly switch between editors by pressing Ctrl + 1, 2, and 3. Keep in mind this works with up to nine editor windows. 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

You can reorder and resize editor windows based on your preferences. To reorder an editor, click the editor's header (which is where you see the file name) and move the editor to a different position. Resizing an editor can instead be accomplished by clicking the mouse left button when the pointer is on the editor's border, until it 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. You can reset the original zoom factor with Appearance ➤ Reset Zoom.

Note In Visual Studio Code, the zoom is actually an accessibility feature. As an implication, when you zoom the code editor, everything else 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. This is only visible when VS Code is connected to a Git repository.
- Errors and warnings detected in the source code.
- The cursor position expressed in line and column.
- Tab size, 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 programming or markup language for the open file. By clicking the current language name, you can change the language from a drop-down list that pops 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 enables you to switch between projects.
- The feedback button, which enables you to share 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 purple when you open a single file, blue when you open a folder, and orange when Visual Studio Code is in debugging mode. Additionally, third-party extensions might use the Status Bar to display their own information.

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, Run and Debug, Extensions, Accounts, 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 the Side Bar.

The Side Bar

The Side Bar is one of the most important tools in Visual Studio Code, and one of the tools you will interact more with. It is composed of five tools, each enabled by the corresponding icon, described in the following subsections.

The Explorer Bar

The Explorer bar is enabled by clicking the first icon from the top of the side bar 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.

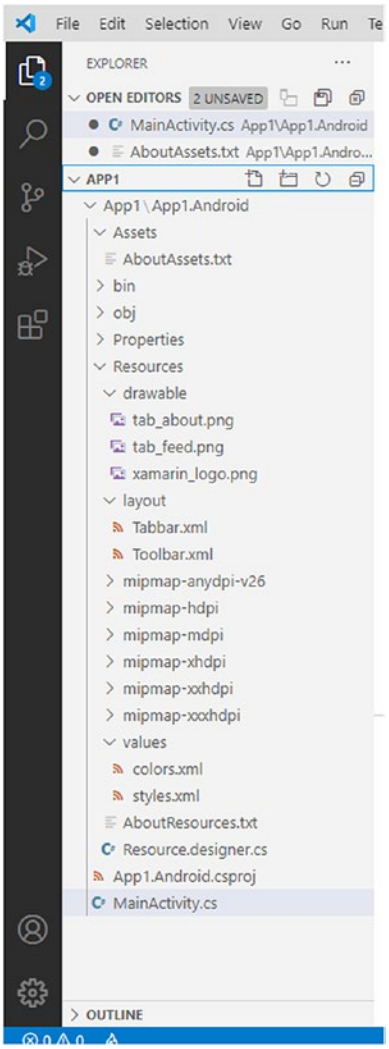

Figure 2-5. *The Explorer bar*

Note You must hover your cursor over a folder name (APP1 in Figure 2-5) to make the four buttons visible.

The subview that shows a folder structure provides four buttons (from left to right): **New File**, **New Folder**, **Refresh Explorer**, and **Collapse Folders in Explorer**, 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 Editors**. Right-clicking a folder or file name in Explorer provides a context menu that offers common commands (such as **Open to the Side**, referenced earlier in this chapter). A very interesting command is **Reveal in 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 implicit. Figures 2-6 and 2-7 show the OUTLINE based on a TypeScript file and based on an HTML file, respectively.

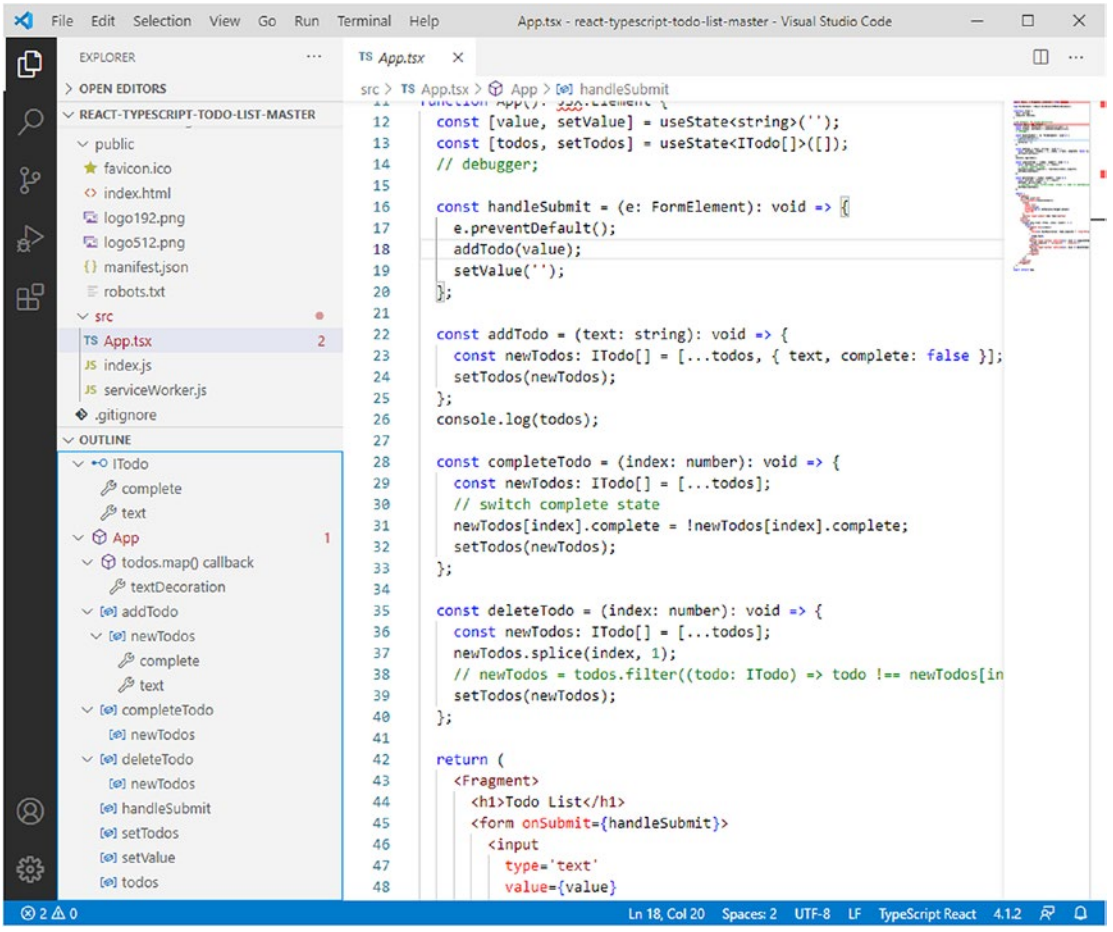

Figure 2-6. *The Outline view on a TypeScript file*

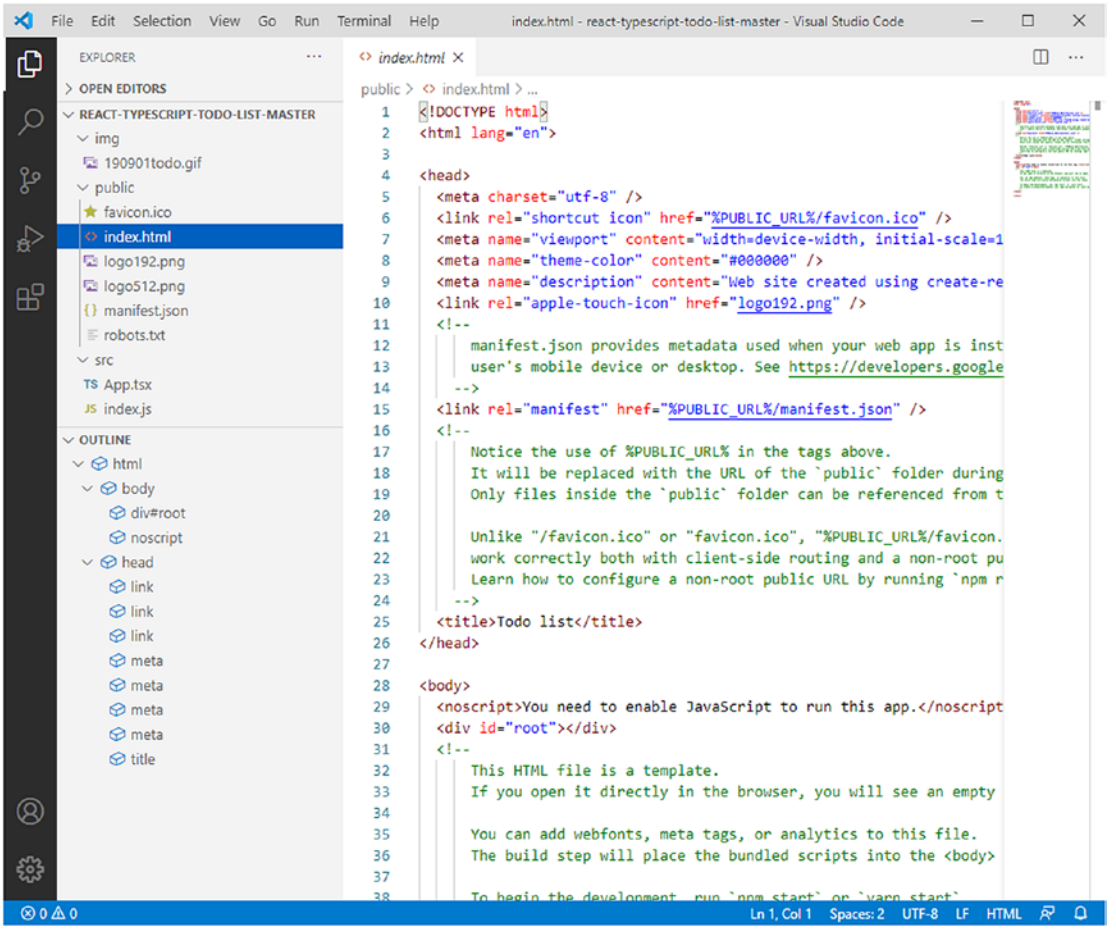

Figure 2-7. *The Outline view on an HTML file*

You can expand types and members defined in a markup file 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. 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. Currently, the Outline view is only available to languages such as JavaScript, TypeScript, HTML, Markdown, and JSON. Support for additional languages might be available when installing the appropriate extensions.

The Search Tool

The Search tool, enabled by clicking 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), which you enable by clicking the **...** button located under Replace. In the example, search is performed only within .cs files.

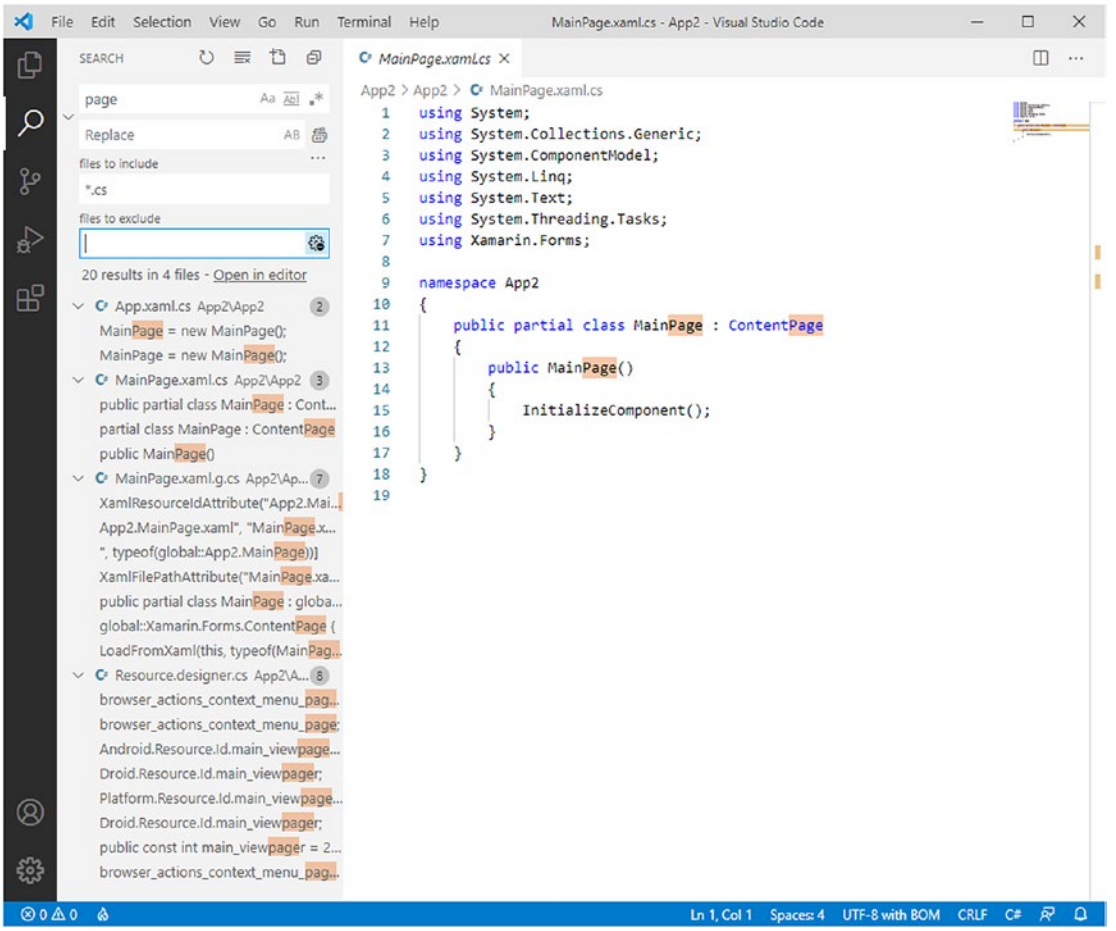

Figure 2-8. *The Search tool*

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 located in the toolbar close to the SEARCH header. 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 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, 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 of the side bar (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. Modified files are listed under the **Changes** group. Three buttons are available for each listed file: **Open File**, **Discard Changes**, and **Stage Changes**. In Git, as you will learn in Chapter 7, the concept of staging changes means keeping changes separate from the main code branch so that a developer can evaluate whether to commit the changes or discard them. Clicking a file name enables a split view that shows the differences between the modified code and the original code; this topic will also be more thoroughly discussed in Chapter 7.

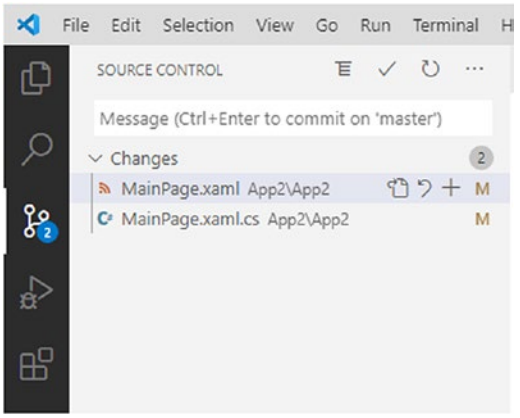

Figure 2-9. *The Git bar*

The Git bar also provides a pop-up menu that contains the list of supported Git commands in Visual Studio Code organized into submenus, such as Commit, Push, Pull, and several more you will discover later in the book. Click the **...** button in the top-right corner of the Git bar to open the menu.

The Run and Debug Bar

Visual Studio Code is not only a simple code editor, but 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 describes in more detail this important part of Visual Studio Code, but for now note that you can access the debugging tools by clicking the fourth icon from the top of the side bar. This opens the Run and Debug bar, shown in Figure 2-10.

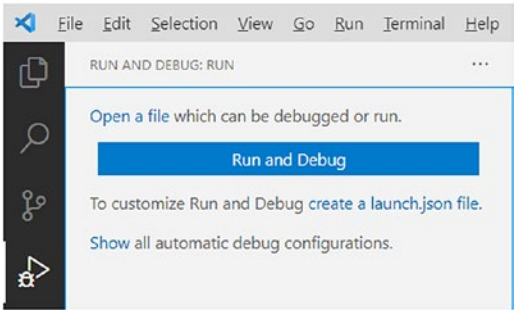

Figure 2-10. *The Run and Debug bar*

In Chapter 9 you will see how to configure the debugging tools and how powerful they are in Visual Studio Code. You will also see how easy it is to install 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, but Figure 2-11 provides an example of how the Extensions bar appears.

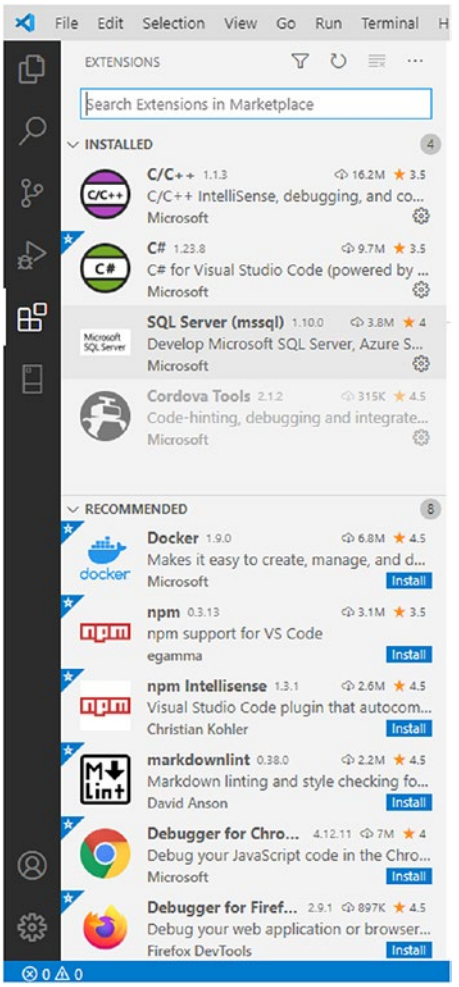

Figure 2-11. *The Extensions bar*

You not only can search online for extensions, but also see the list of installed extensions as well as disabled and recommended extensions.

The Accounts Button

One of the biggest benefits of Visual Studio Code is that you can customize it in many ways by arranging the development environment in whichever configuration is most convenient for you. This includes extensions, keyboard shortcuts, general settings, and much more.

If you run VS Code on multiple machines, it would be very useful if you could re-create your environment automatically on all the machines, without the need to set your preferences manually on each machine. Fortunately, this is possible using the **Accounts** button on the Side Bar.

With this tool, you can sign in with a Microsoft or GitHub account and your settings will be synchronized across all the VS Code installations to which you have signed in with the same account. Following is a list of settings that can be synchronized:

- General settings
- Keyboard shortcuts
- Extensions
- User-defined code snippets
- State of the user interface

You enable settings synchronization by clicking the **Accounts** button and then **Turn on Settings Sync**. At this point Code shows a list of settings that you can sync across machines, selecting all of them by default, as shown in Figure 2-12.

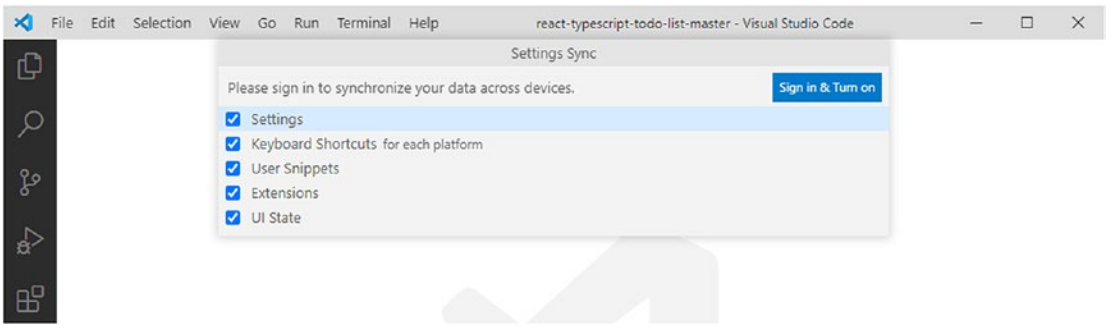

Figure 2-12. *The selection of settings to synchronize*

Select the settings you want to sync, then click **Sign in & Turn on**. At this point you will be asked to specify which kind of account you want to use, such as Microsoft or GitHub. Obviously, you need to use the same account on all the other Code installations. A browser window opens in which you enter your credentials, and you will quickly get a confirmation message when sign-in is completed.

Note On Windows, the Firewall might prompt you with a warning saying that VS Code is trying to open a resource on the Web. If this happens, you can safely allow this action.

At this point Visual Studio Code starts synchronizing all the selected settings, which might take a while. Behind the scenes, settings synchronization is based on two files, settings.json and extensions.json, which VS Code needs to merge from different installations. If it encounters problems in merging these files automatically, VS Code gives you an option to manually merge settings with the same merging tool used with Git. This is a very useful feature and it will save you a lot of time in getting the same comfortable environment across machines.

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 pop-up menu with a list of commands that represent shortcuts for customizing Visual Studio Code (and that will be discussed more thoroughly in Chapter 5). Among others, a command in the menu enables you to manually search for product updates.

Navigating Between Files

Other than clicking the tab of an editor, Visual Studio Code provides two ways of navigating between files. The quickest way is to press 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 currently open files and select one for editing, as shown in Figure 2-13.

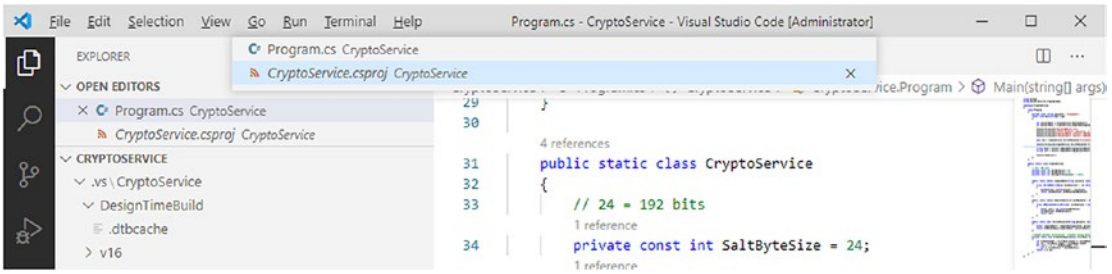

Figure 2-13. *Navigating between active files*

The Command Palette

Together with the code editor and the Activity Bar and Side Bar, the Command Palette is another very important tool in Visual Studio Code, which enables you to access Visual Studio Code built-in commands and also commands added by extensions via the keyboard. You can open the Command Palette, shown in Figure 2-14 with **View ➤ Command Palette** or via the Ctrl+Shift+P keyboard shortcut (⌘+P on macOS).

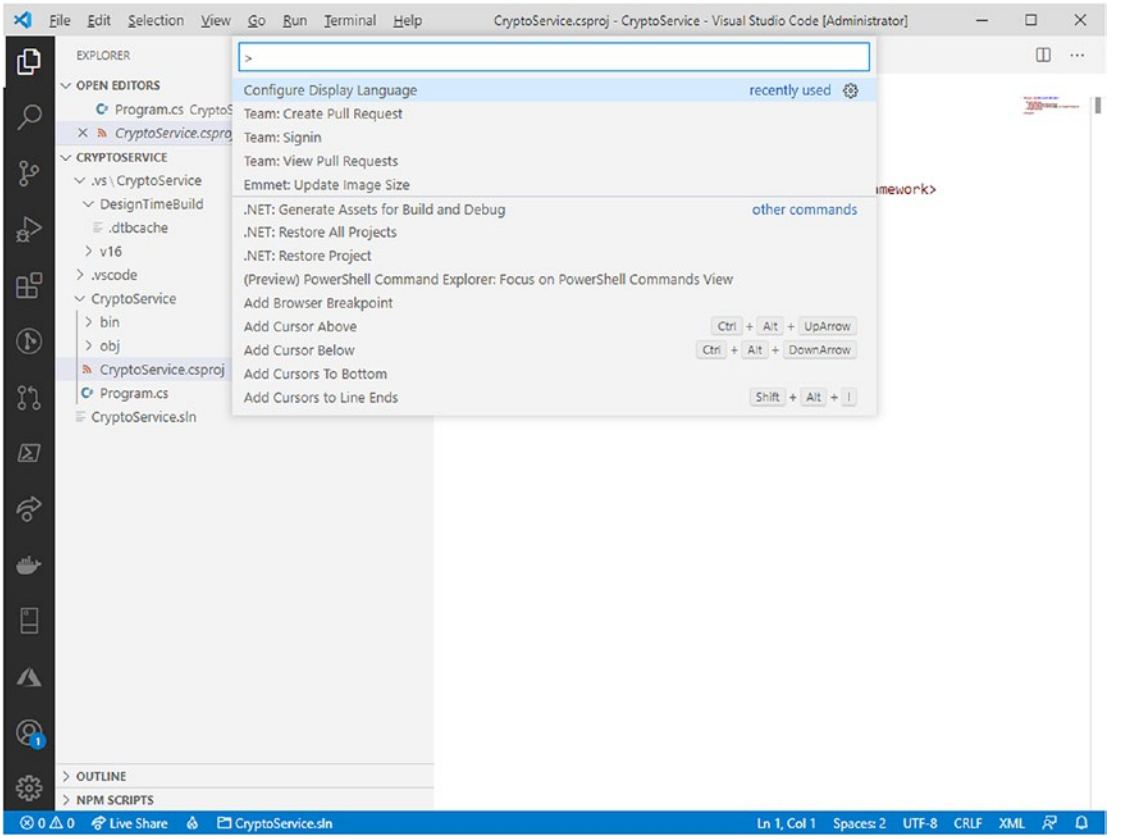

Figure 2-14. *The Command Palette*

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 enables you to install extensions as well as restore 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 related to extensions, debugging, and Git, will be discussed in the following chapters, so it is important that you get started with the Command Palette at this point.

The Panels Area

Visual Studio Code very often needs to display not only 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 composed 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 the panels represent becomes available (such as the debugger sending information about symbols in the source code). Additionally, by default Panels area appears at the bottom of the VS Code's user interface, but you can move it to the side of the workspace by right-clicking a panel and then selecting **Move Panel Right** or **Move Panel Left**, or restore the original position with **Move Panel to Bottom**. In addition, you can now drag and drop panels in a different position using the mouse. Let's now discuss each panel in more detail.

The Problems Panel

With languages that have built-in enhanced editing support, such as TypeScript (`https://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 2-15).

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 2-15 shows an example based on some C# code that contains an unused variable (warning) and a syntax error.

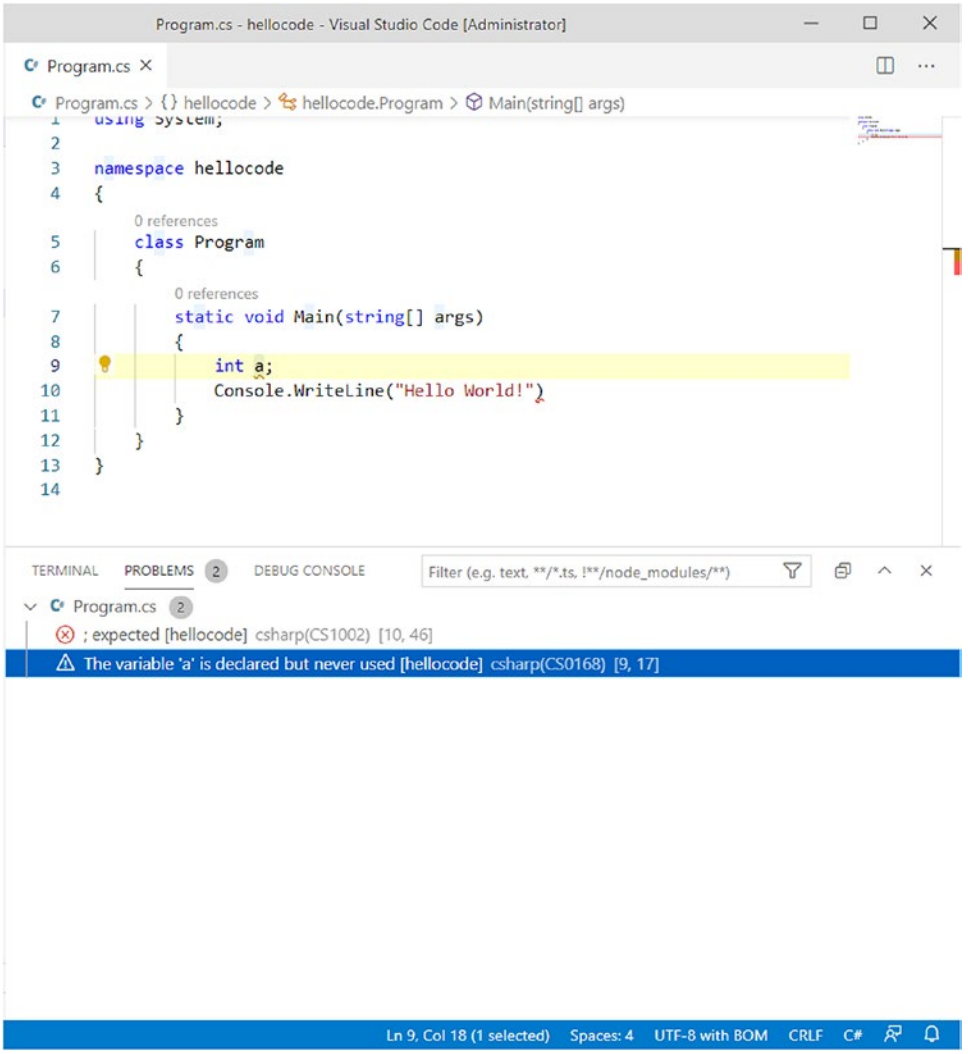

Figure 2-15. *The Problems panel*

If you have multiple files open, the Problems panel groups 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 2-16 shows an example based on the output of .NET's NuGet package manager.

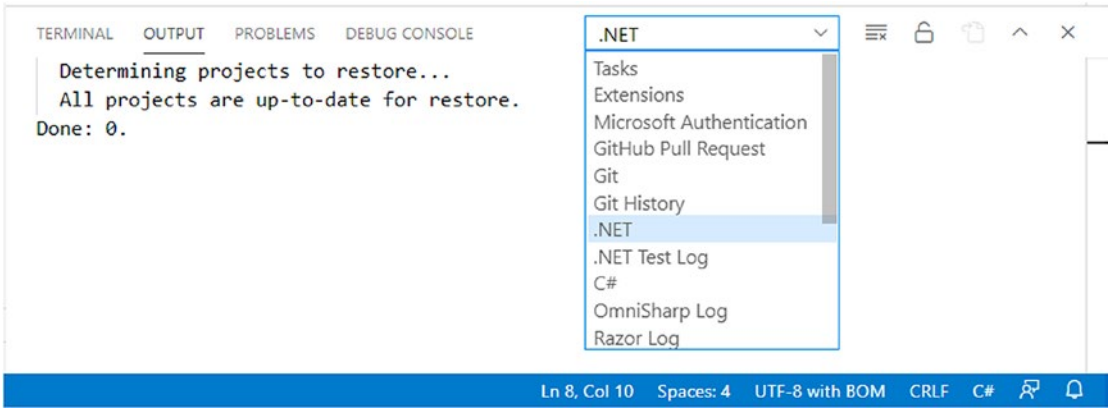

Figure 2-16. *The Output panel*

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 2-17 shows an example based on the execution of a simple C# application.

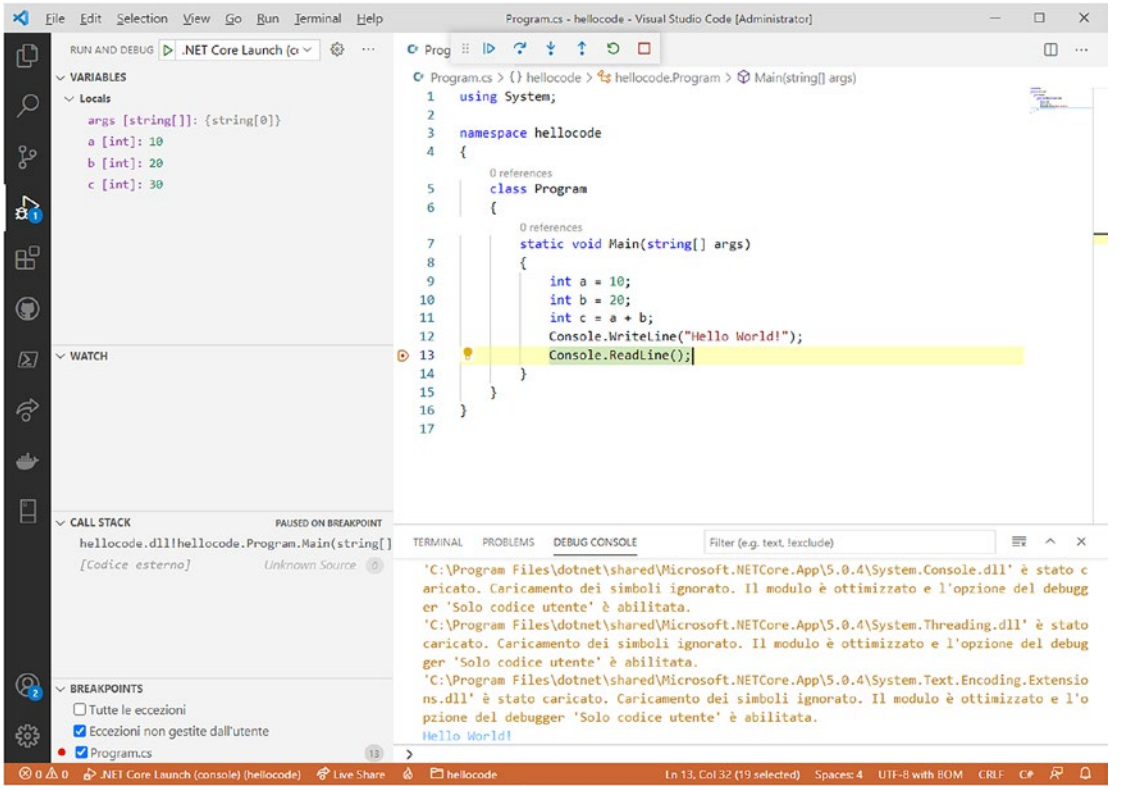

Figure 2-17. *The Debug Console panel*

The Debug Console not only shows information about code execution, debug symbols, and any other information a debugger needs to display, but also acts as an interactive console where you can evaluate expressions. Figure 2-17 shows that 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, 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 2-18 shows an example based on Windows.

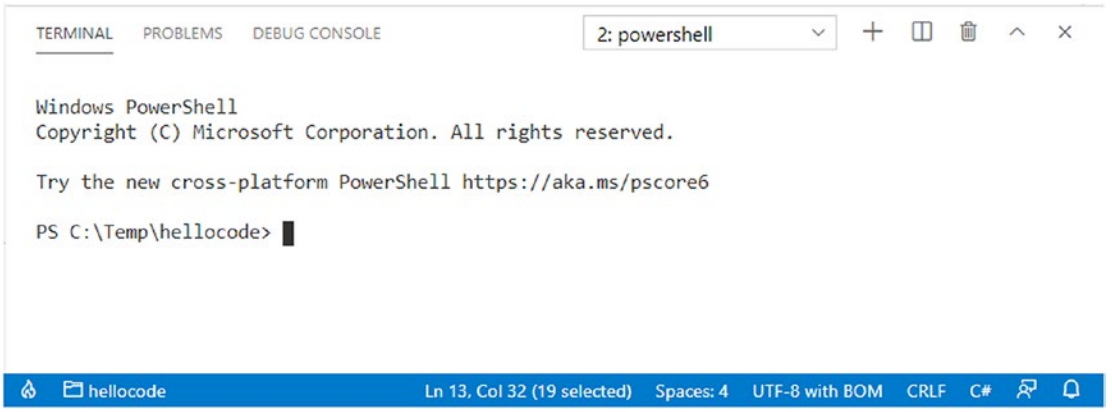

Figure 2-18. *The Terminal panel*

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, you can select a different tool by clicking the drop-down menu on the panel's toolbar and then clicking **Select Default Shell**. At this point you will be able to select, from the Command Palette, from 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 2-19.

Figure 2-19. *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 with frequently. 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, the Accounts button, 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, get messages from internal and external tools and debuggers, and execute commands and scripts via the Terminal.

Now that you have seen how the environment is organized, it is time to have some 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 codefocused 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#, JavaScript, 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.

A. Del Sole, *Visual Studio Code Distilled*, https://doi.org/10.1007/978-1-4842-6901-5_3

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 downloadable from the Visual Studio Marketplace. This is discussed in more detail in Chapter 6, but, in the meantime, you can have a look at the available languages out of the box. For the purposes of this book, an introduction to C# and C++ is provided for your convenience.

Working with C# and C++

The C# programming language deserves a more detailed introduction, 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 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 that you 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, but you can easily install the C# extension without further information by 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 easy installation steps just described for the C# extension by 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 tool. If you are familiar with Microsoft Visual Studio, 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.

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;
```

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

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

The **Edit** menu also provides a command to work with code snippets, **Emmet: Expand Abbreviation**. This command is the menu representation of keyboard shortcuts offered by the code editor to add a code snippet. 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 provides 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.

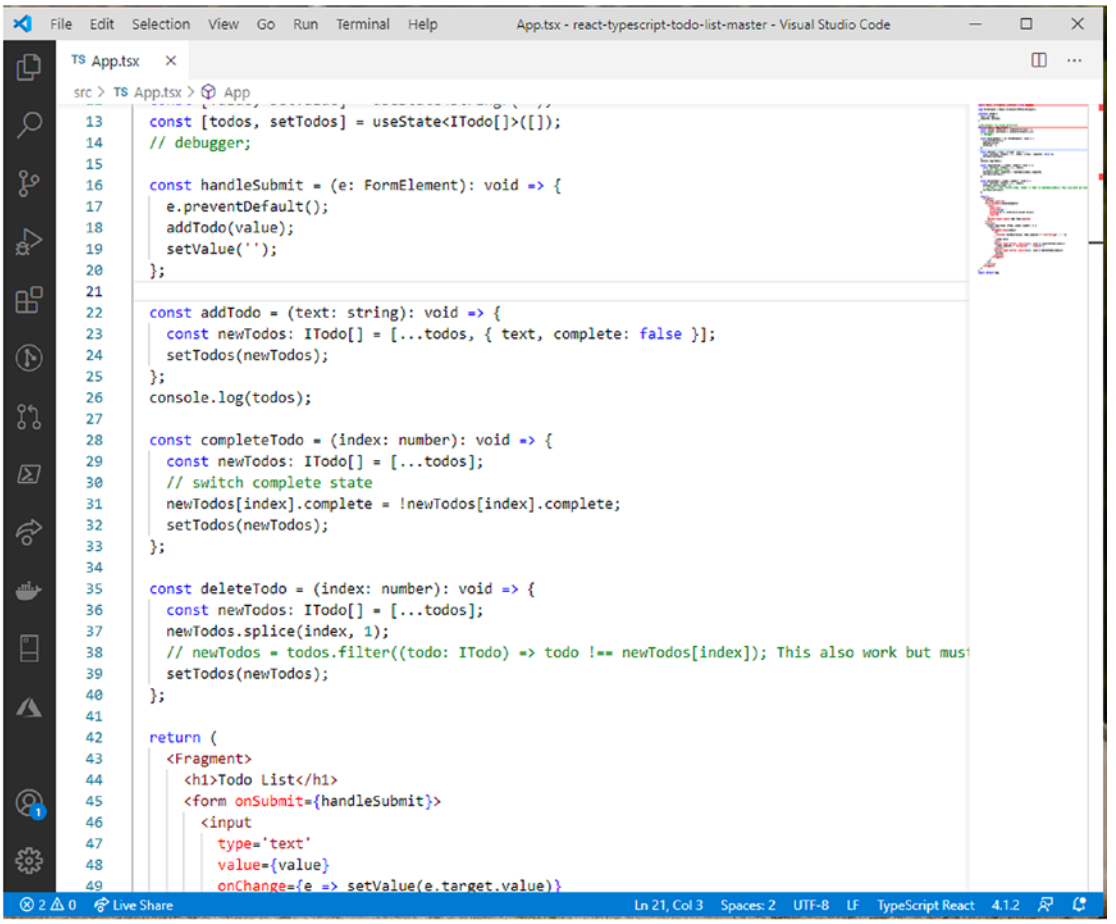

Figure 3-1. *Syntax colorization*

Syntax colorization is available for other languages via extensibility. If you need to work with a language that is not included with Visual Studio Code 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 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.

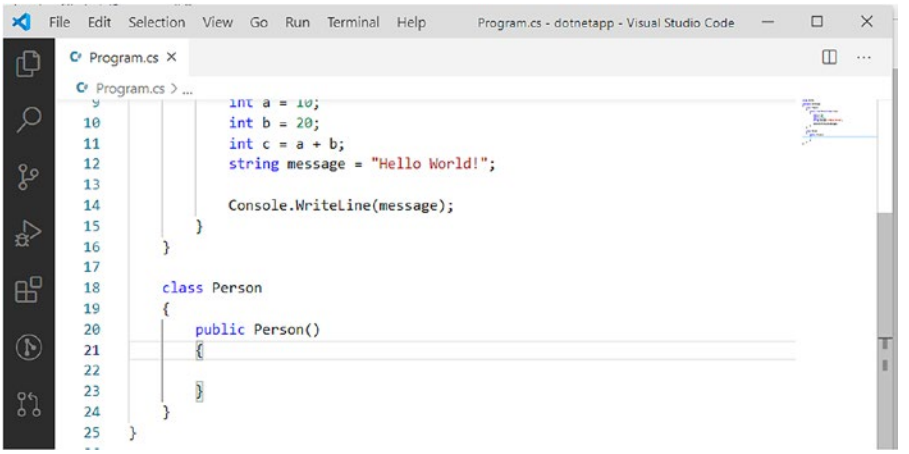

Figure 3-2. *Delimiter matching*

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 quickly select a word or identifier at the right of the cursor. You can also quickly select all the text within the delimiters of a code block by pressing Shift+Alt+Arrow Right, and you can quickly deselect the same code block by pressing Shift+Alt+Arrow Left.

Code Block Folding

The code editor allows folding delimited code blocks. Just hover your cursor over line numbers and a symbol representing a down arrow 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.

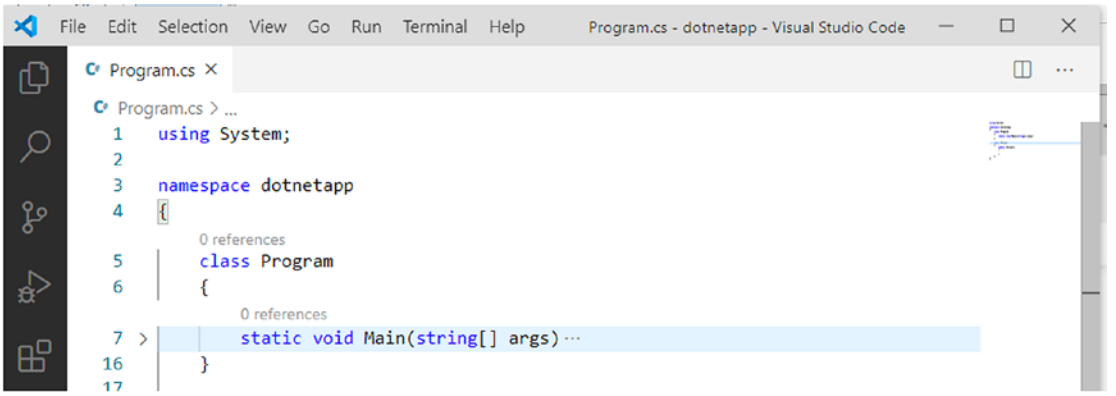

Figure 3-3. *Code block folding*

Note If code block folding is not enabled in the code editor, open VS Code's Settings, then in the **Text Editor** group enable both the **Folding** and **Folding Highlight** options.

Multicursors

The code editor supports multicursors. Each cursor operates independently, and you can add secondary cursors by pressing Alt+Click at the desired position. 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 Table 3-1 in the "Language Support" section to review which 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*

Notice that if the code snippet contains variable names or identifiers, these might be highlighted to suggest that you give them a different name (like for the `error` identifier in Figure 3-5). When you rename a highlighted identifier, all occurrences are 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, Figure 3-6 shows how the code editor suggests terminating a statement with the `Class` keyword in a Visual Basic file, based on what the developer is typing.

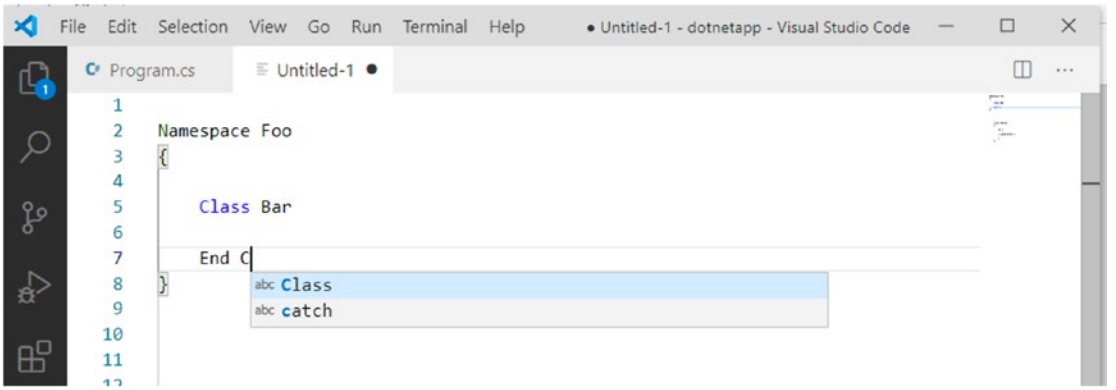

Figure 3-6. *Completing a statement with word completion*

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.

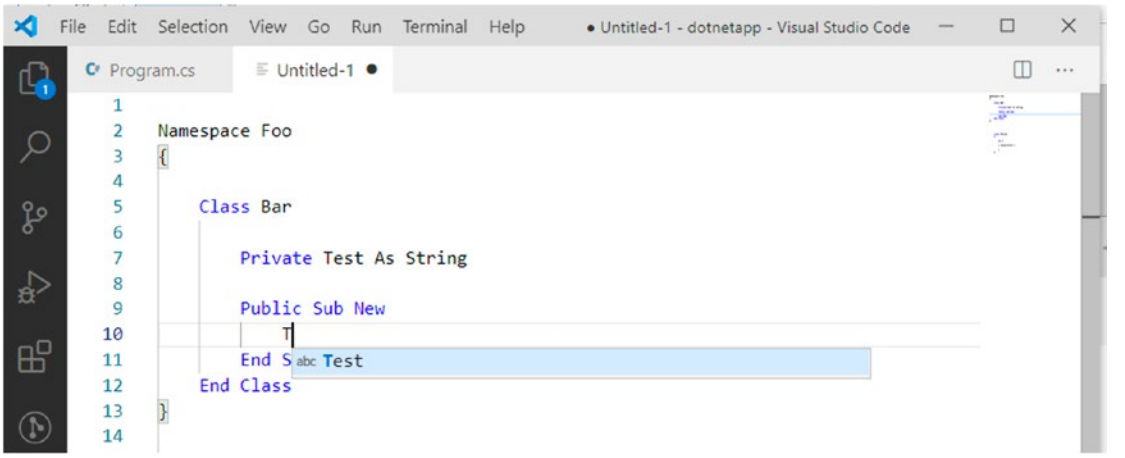

Figure 3-7. *The code editor can suggest identifiers declared in the code*

Minimap Mode

Sometimes it is difficult to find 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.

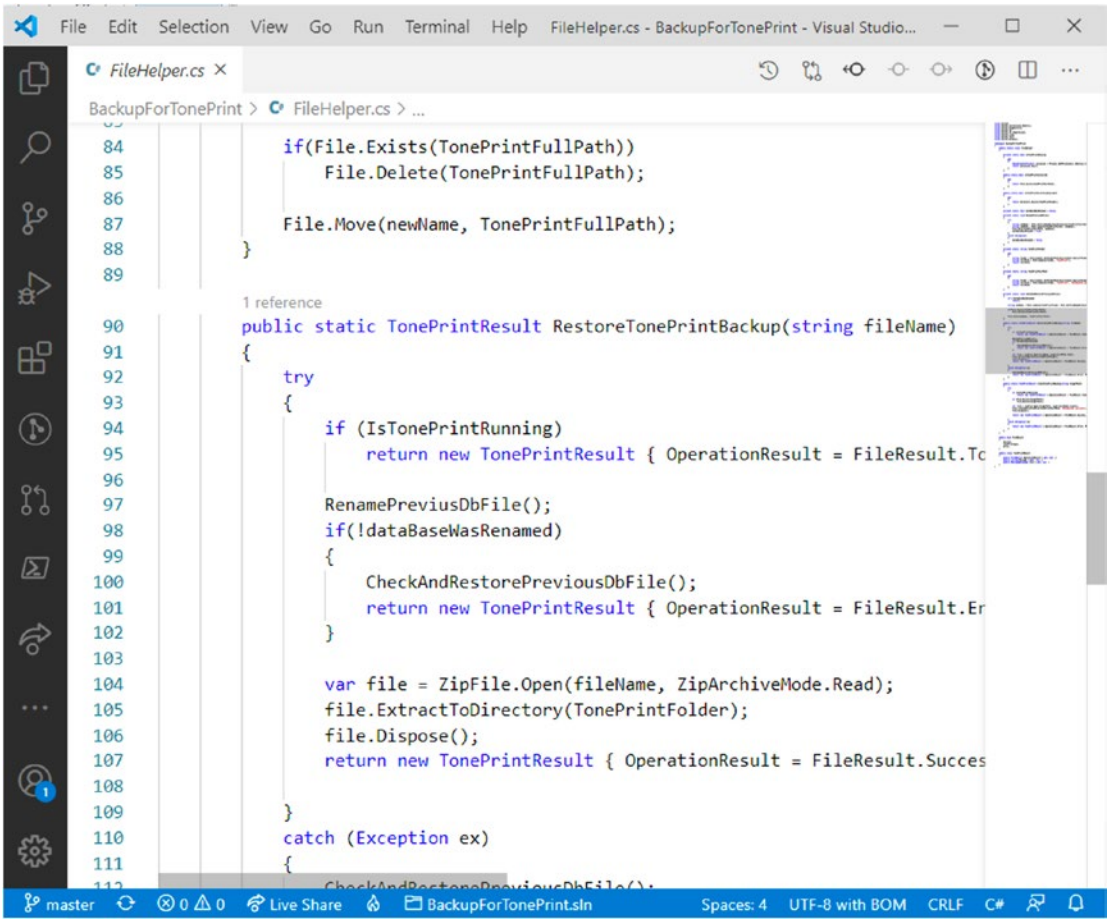

Figure 3-8. *The Minimap allows for previewing source code on the scrollbar*

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 ➤ Show Minimap** command.

Whitespace Rendering and Breadcrumbs

A very common feature with text editors is the option to show light dots instead of white spaces. In Visual Studio Code, this is possible for white spaces within indentations. To accomplish this, you select **View ➤ Render Whitespace**. Figure 3-9 shows an example of how white spaces for indentations are replaced with dots. For this figure, the Solarized Light color theme has been used for better visualization on the paper.

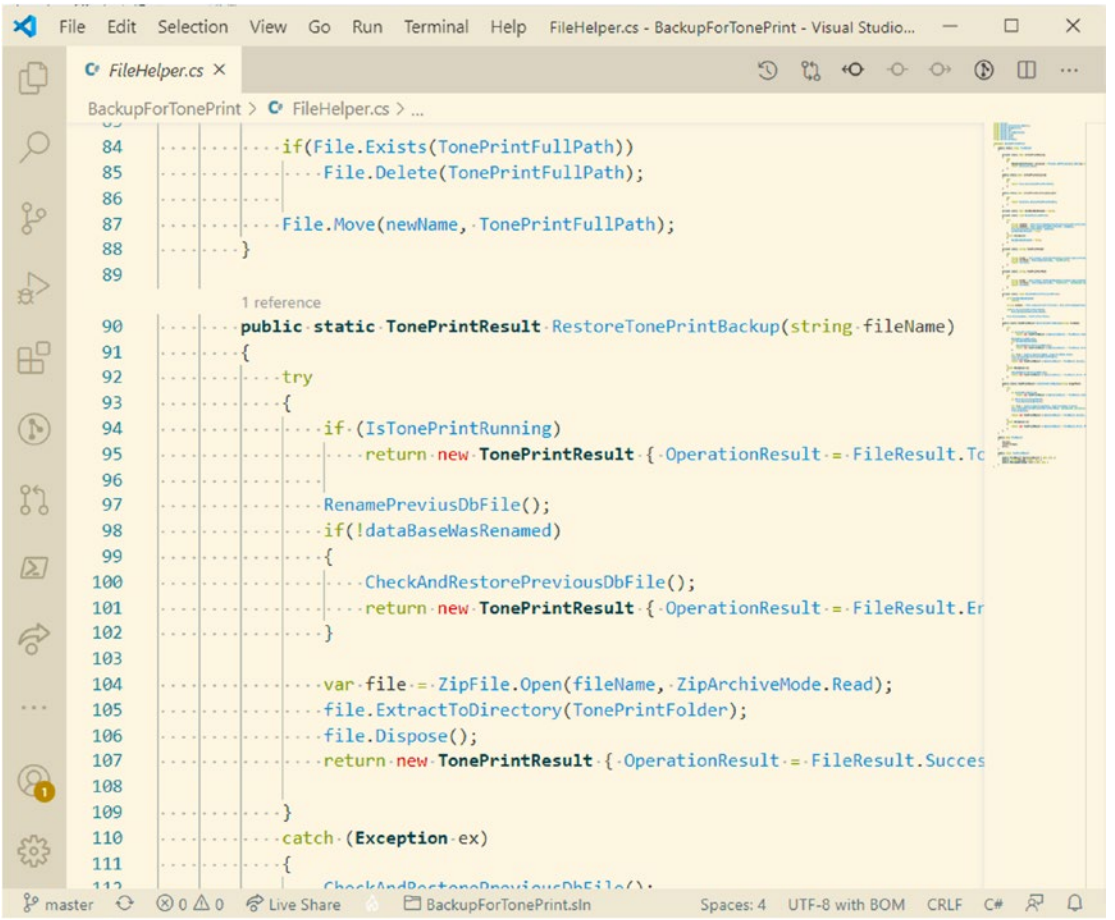

Figure 3-9. *Rendering indentation spaces with dots*

Simply use 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, this command shows the list of types and members defined in the current code file at the top of the editor, which you can expand to see their members, as shown in Figure 3-10.

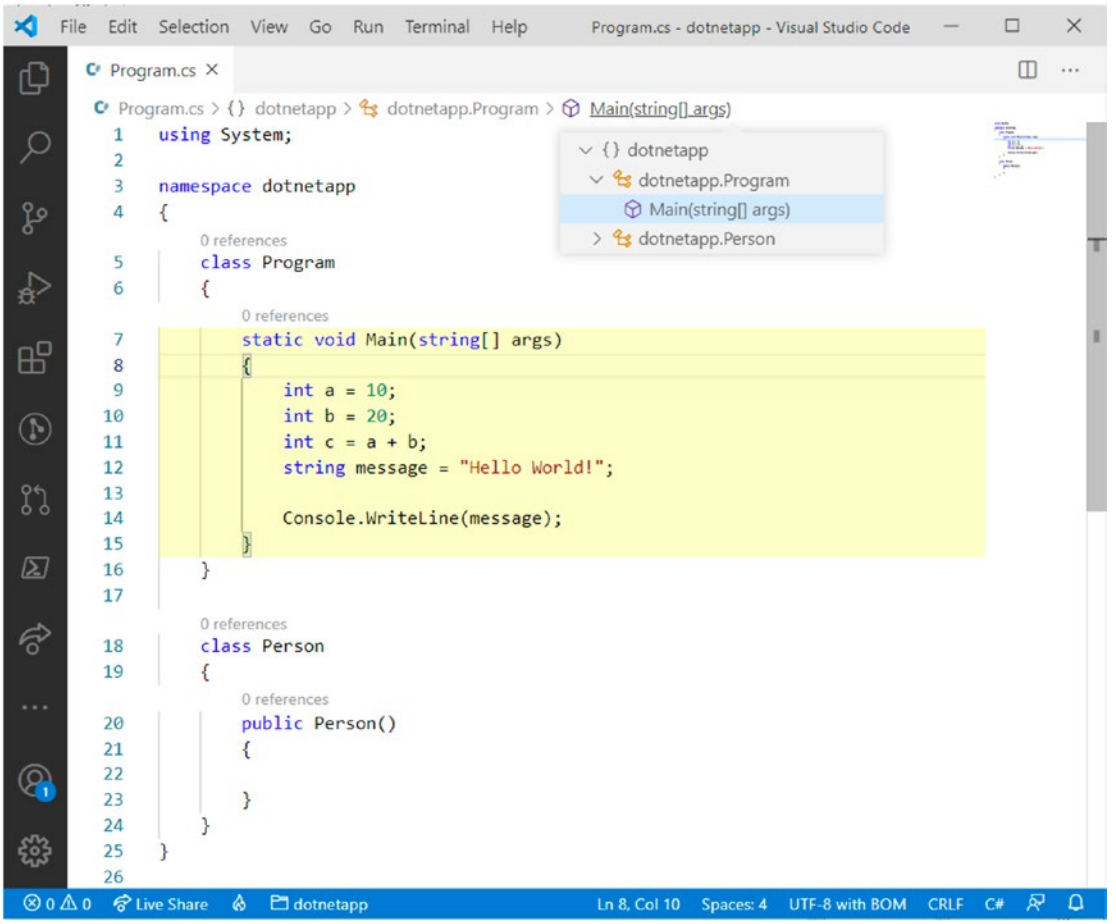

Figure 3-10. *Navigating between types and members with breadcrumbs*

Clicking a type or member name moves the cursor to its definition and highlights the related code block, 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 what 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.

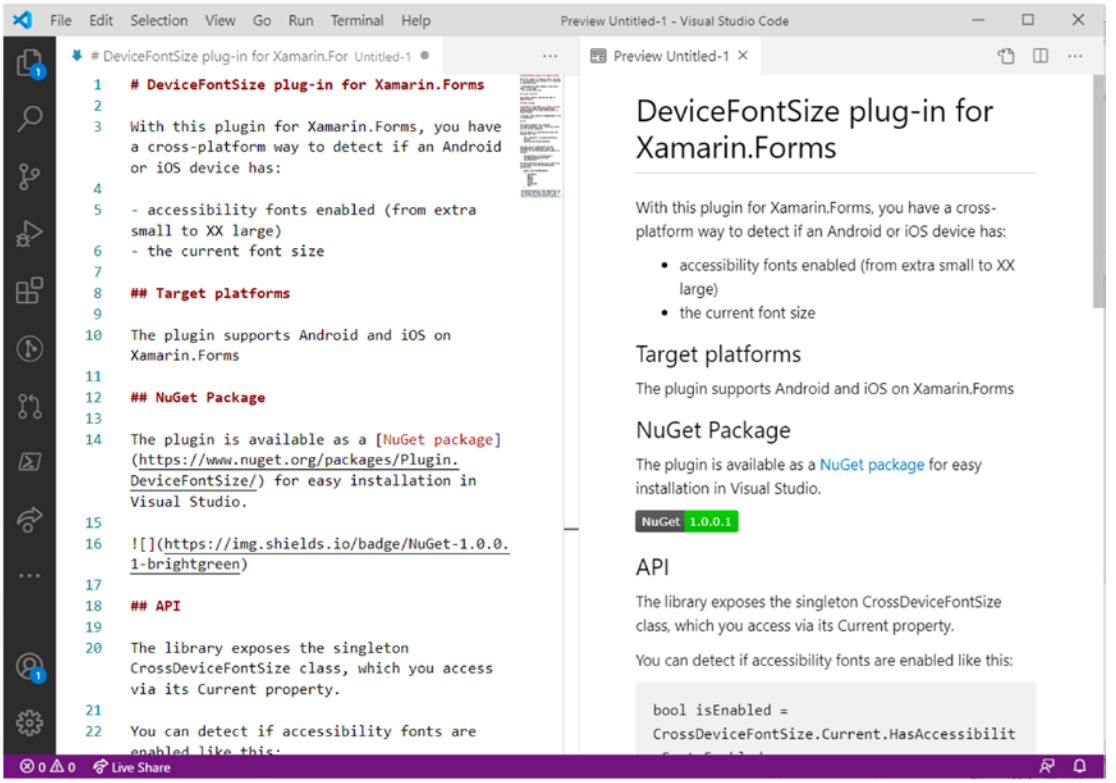

Figure 3-11. *Integrated Markdown preview*

This feature is very useful because it allows you 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 for 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, with the appropriate extensions installed, to languages like C#, C++, and Python.

Working with IntelliSense

IntelliSense provides rich, advanced word completion via a convenient pop-up list 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.

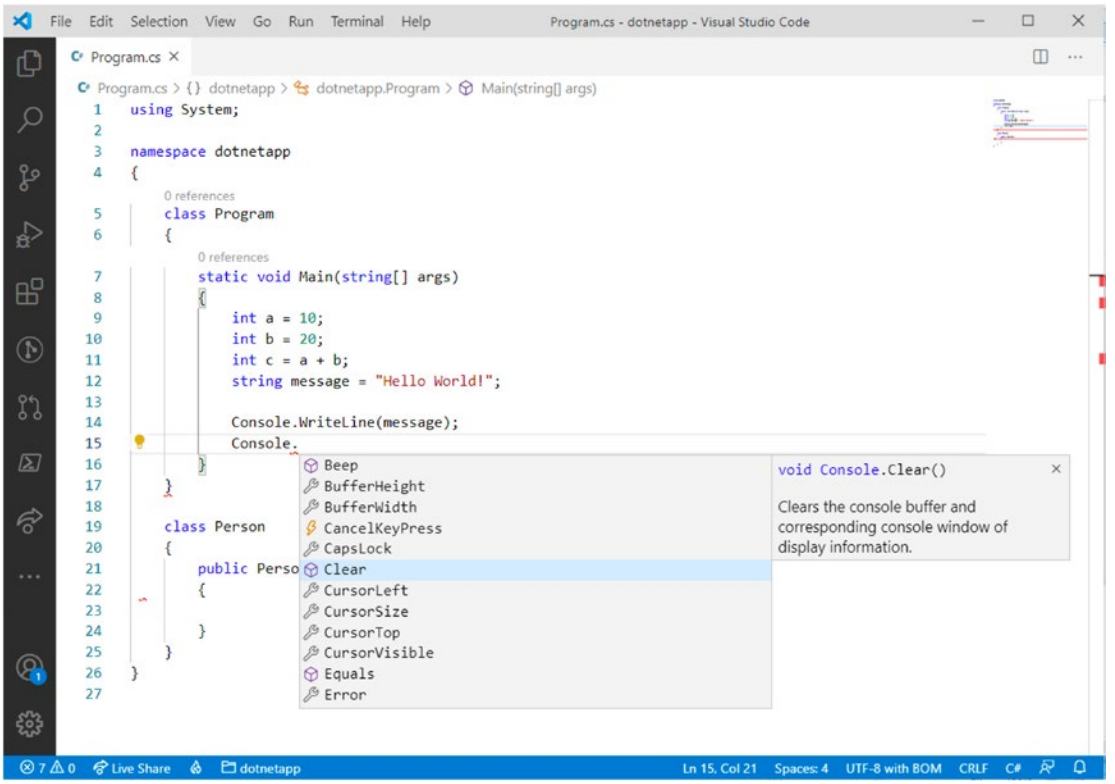

Figure 3-12. *IntelliSense showing suggestions as you type and advanced word completion*

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 scroll the list with the keyboard and stop on a word from the completion list, Visual Studio Code shows the member documentation. The little arrow at the right of the dialog can be used to turn the documentation off.

Note The documentation for a type or member is available only 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 enables IntelliSense to display it in a tooltip, like in Figure 3-12.

Press either Tab or Enter to complete the word insertion, or simply click. 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 type `cv`, the suggestion list will show the `CursorVisible` property, as demonstrated in Figure 3-13.

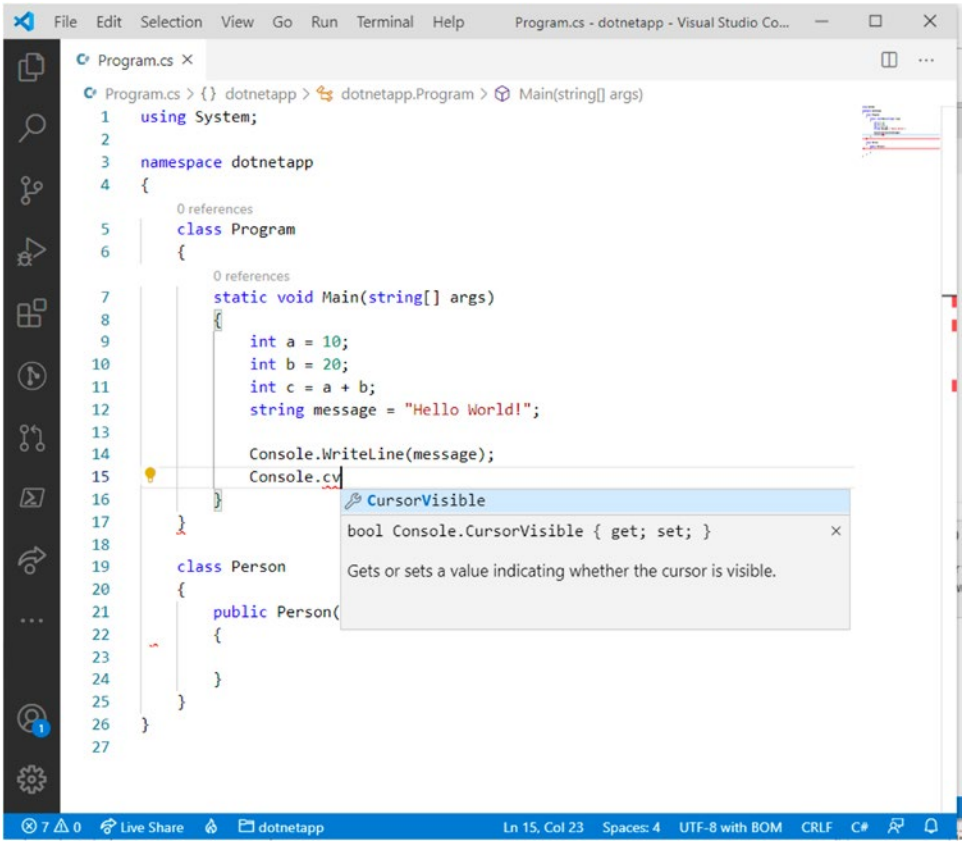

Figure 3-13. *Suggestion filtering in IntelliSense*

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.

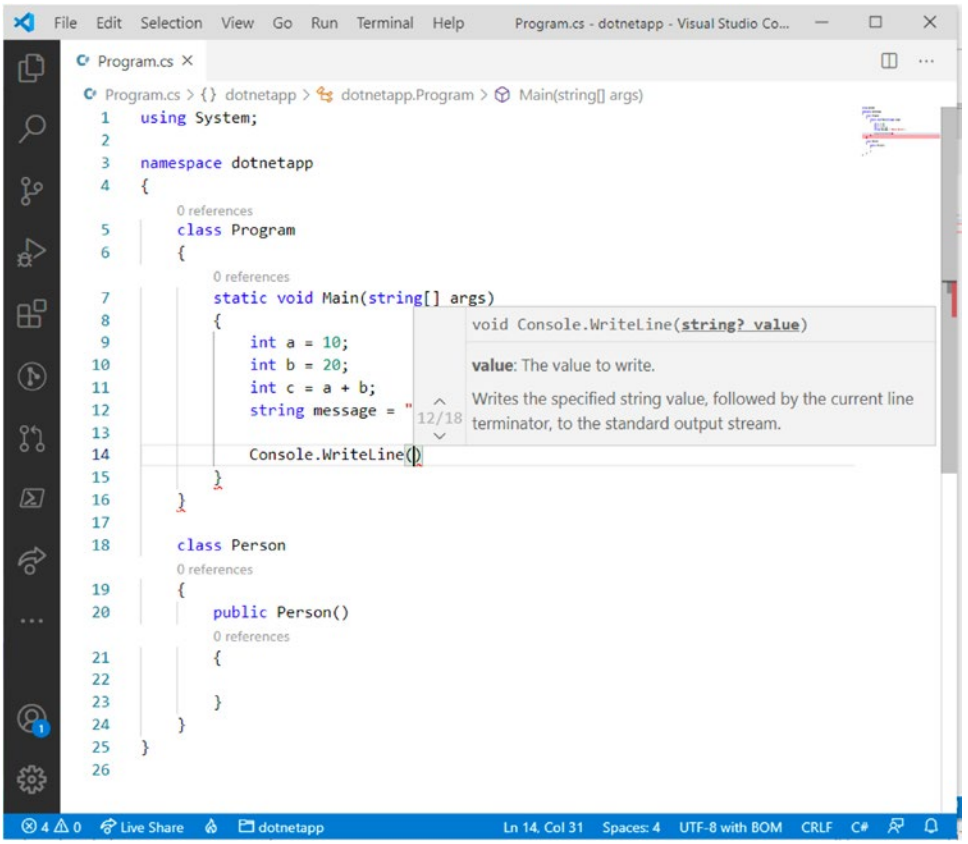

Figure 3-14. *IntelliSense showing parameter hints*

For languages such as C# and TypeScript or, more generally, 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 your cursor over types, variables, and type members, Visual Studio Code shows a tooltip that contains the documentation for the selected object. Figure 3-15 provides an example.

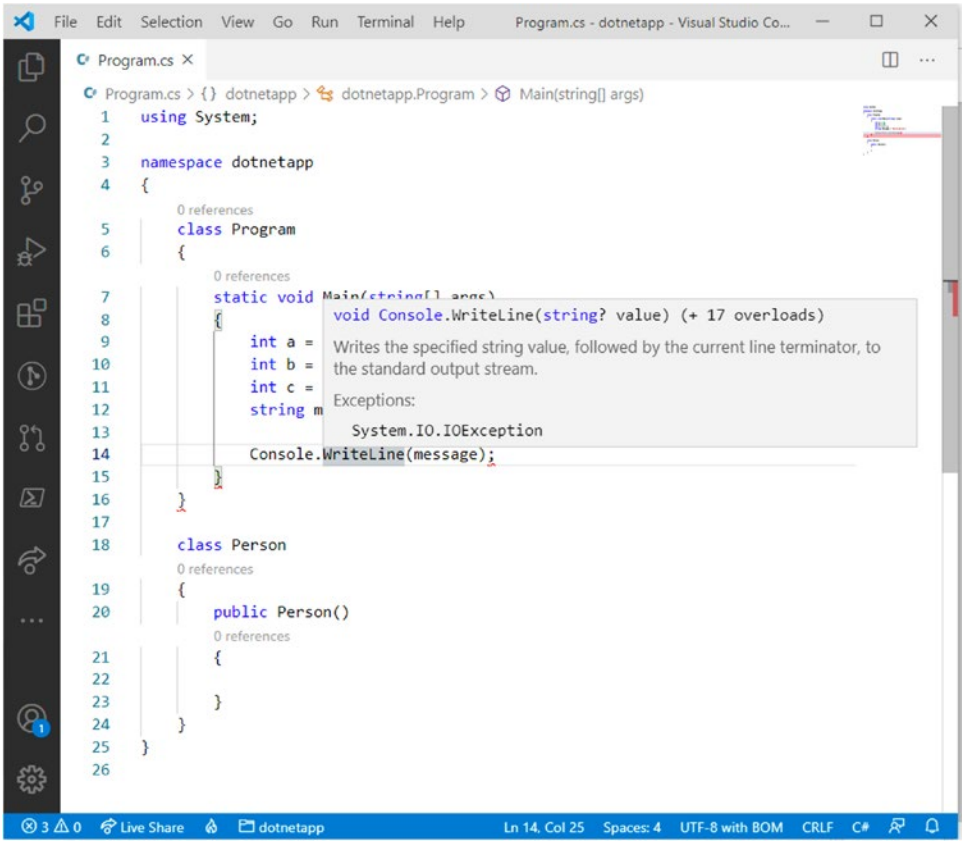

Figure 3-15. *Tooltips provide quick, inline documentation*

Like parameter hints, this feature is available only if the documentation has been implemented

---

Note If you hover your cursor over a variable name, the tooltip shows only the type for the variable.

Go to Definition and Peek Definition

Visual Studio Code provides another interesting feature called *Go to Definition*. If you hover your cursor over a symbol and press Ctrl (or ⌘ on macOS), the symbol appears as a hyperlink; also, a tooltip shows the code that declares that symbol. If you click the type name while pressing Ctrl, 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.

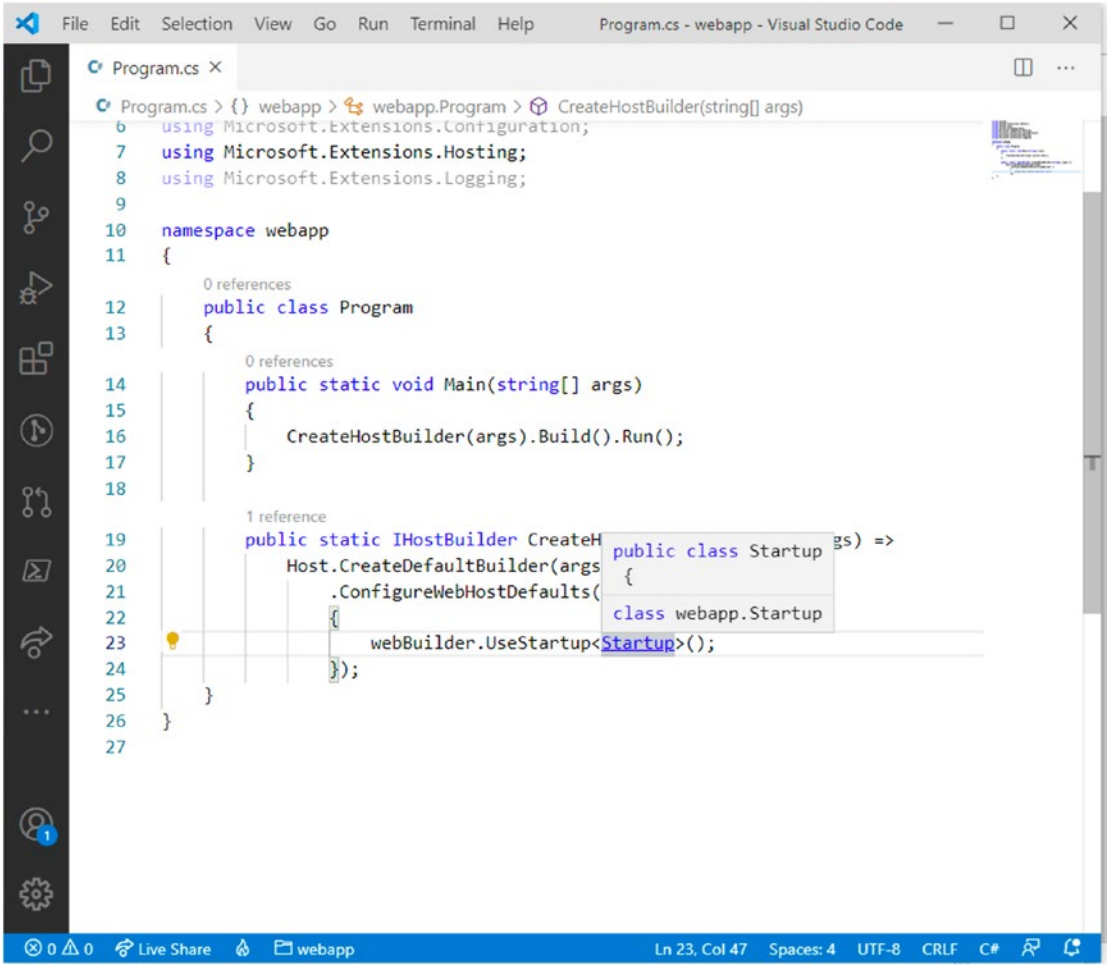

Figure 3-16. *Ctrl + hovering over a type enables Go to Definition*

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 and any NuGet package that includes the type definition information, not just your code.

Now suppose that you have dozens of code files and want to see or edit the definition of a type you are currently using. With other editors, you would search among the code files, which not only can be annoying but also moves 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 ➤ Peek Definition** (the keyboard shortcut is Alt+F12); an interactive pop-up 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 3-17 shows the Peek Definition window in action. You can press Esc to quickly close the Peek Definition window as an alternative to clicking the Close button.

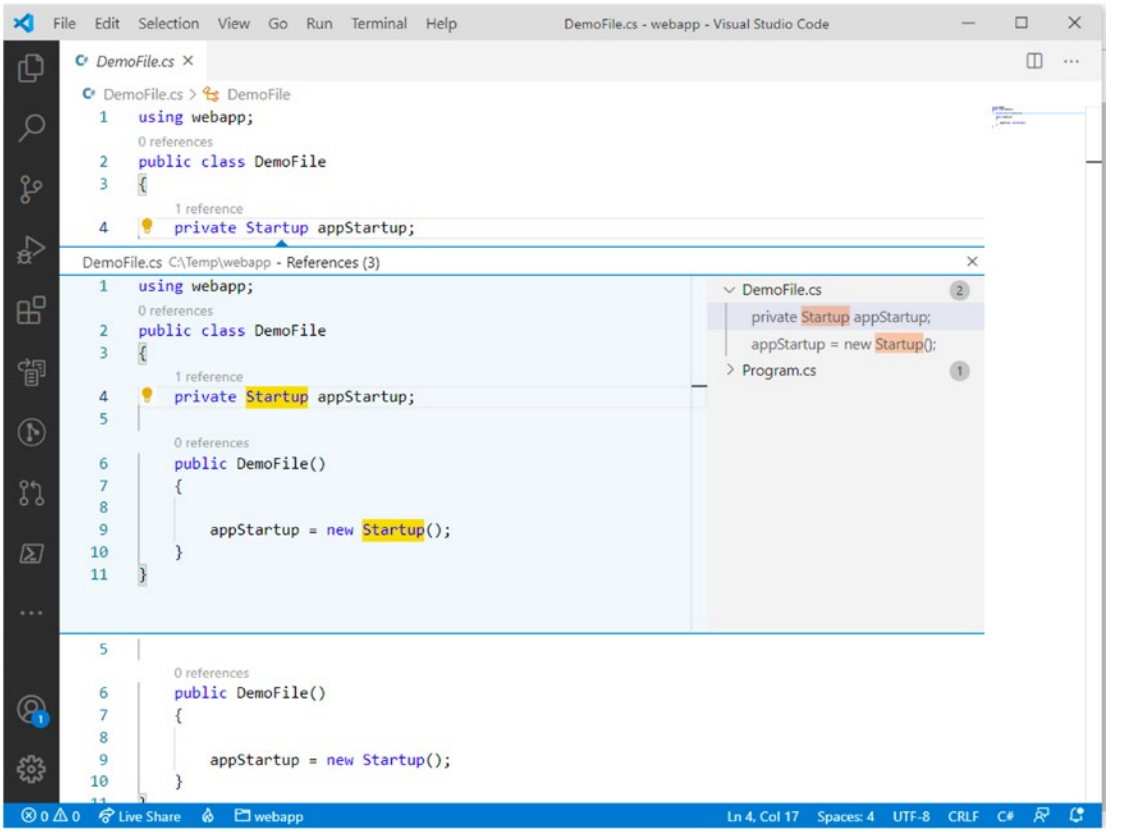

Figure 3-17. *Working on a type defined in another file with Peek Definition*

As you can see, the Peek Definition 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.

Go to Implementation and Peek Implementations

Sometimes you might need to understand how many times and where an interface or an abstract class has been implemented.

Though you can accomplish this by finding a type's references (see the next section), Visual Studio Code now offers more convenient ways that work similarly to Go to Definition and Peek Definition, respectively called Go to Implementation and Peek Implementations. You can right-click an interface or abstract class definition and then select **Go to Implementation** or **Peek ➤ Peek Implementations**. Both actions bring up

an interactive, nested editor that shows the list of implementations of the selected type on the right, and the code for the first occurrence of the implementation, as you can see in Figure 3-18.

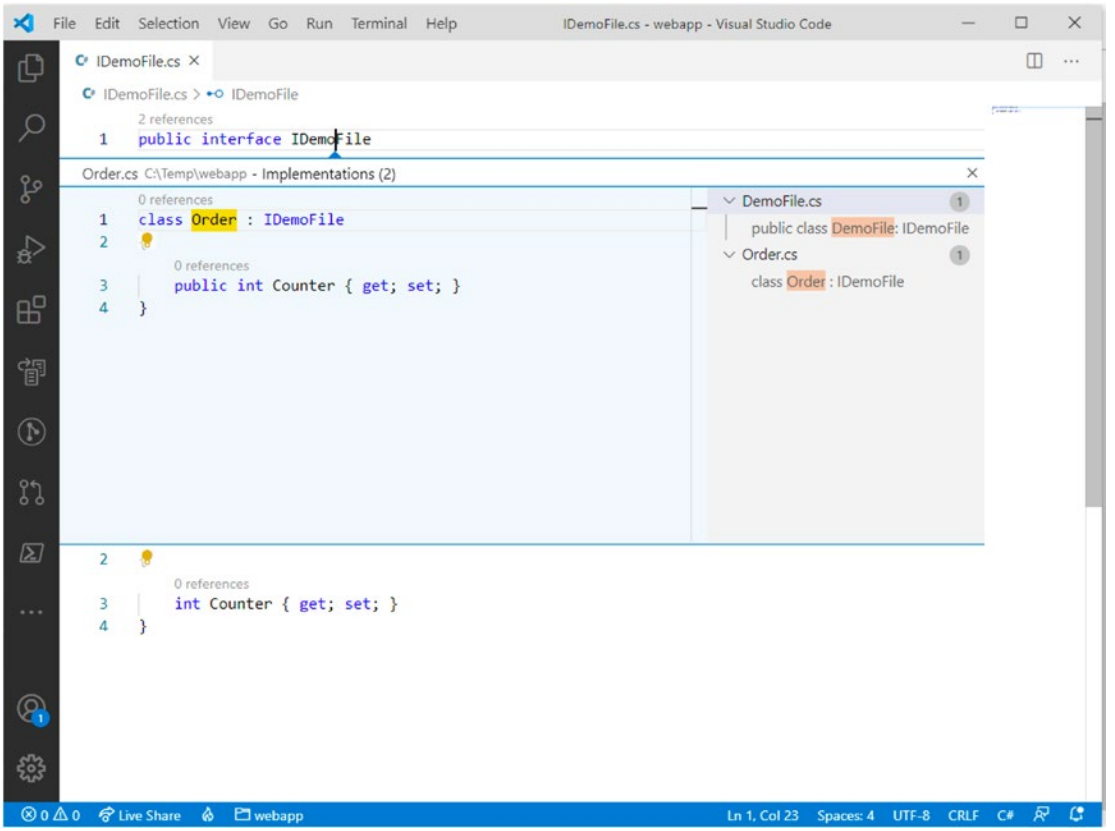

Figure 3-18. *Navigating among type implementations*

The difference between the two actions is the following: with Go to Implementation, when you click an implementation in the list, VS Code opens a new editor window pointing to the file that contains the implementation; with Peek Implementations, when you click an implementation in the list, it is displayed in an interactive pop-up window similarly to how Peek Definition works.

Finding References

You will often need to know where types or members have been used across your code, and Visual Studio Code provides two nice tools to retrieve references.

The first tool is called Find All References, which you might already be familiar with if you have experience with Visual Studio on Windows. There are different options to run this tool: you can right-click a type or member name and then select **Find All References** or you can press Shift+Alt+F12 (Option+Shift+F12 on macOS). Figure 3-19 shows an example based on finding all references of a type called `Startup`.

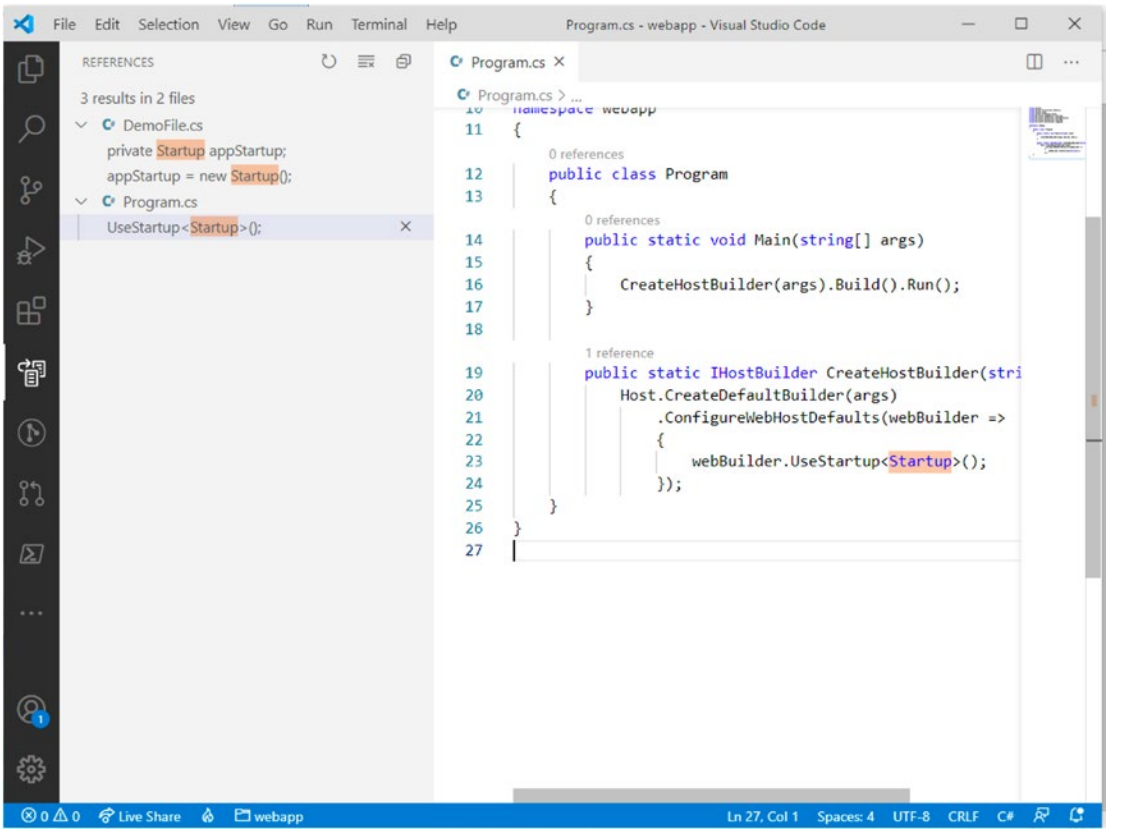

Figure 3-19. *Finding all references of types and members*

The References panel opens on the left side of the screen and shows a list of references grouped by code file, together with the total number of references and of code files involved. It also adds a new entry to the Side Bar that is disabled once you close the References panel. The occurrences are highlighted; when you click one of them, an editor opens on the file that contains the selected occurrence, which will be highlighted inside the code.

There is also another tool called Go to References (Shift+F12), which works inside the active editor window. You enable Go to References either by right-clicking the object name and then selecting **Go to References** or by clicking the number of references at the top of the member definition (see Figure 3-19). You can use the first option anywhere in the code, whereas you can use the second option only when the type or member definition is focused in the code editor.

The user interface for Go to References is the same as for Find All References. Visual Studio Code also provides another useful tool to find type and member references, called Peek References. You can enable this tool by right-clicking an object name and then selecting **Peek ➤ Peek References**. As the name implies, Peek References displays all the references in the active editor, inside an interactive panel similar to what you saw previously with Peek Definition. Figure 3-20 shows an example, again based on finding all references of a type called `Startup`.

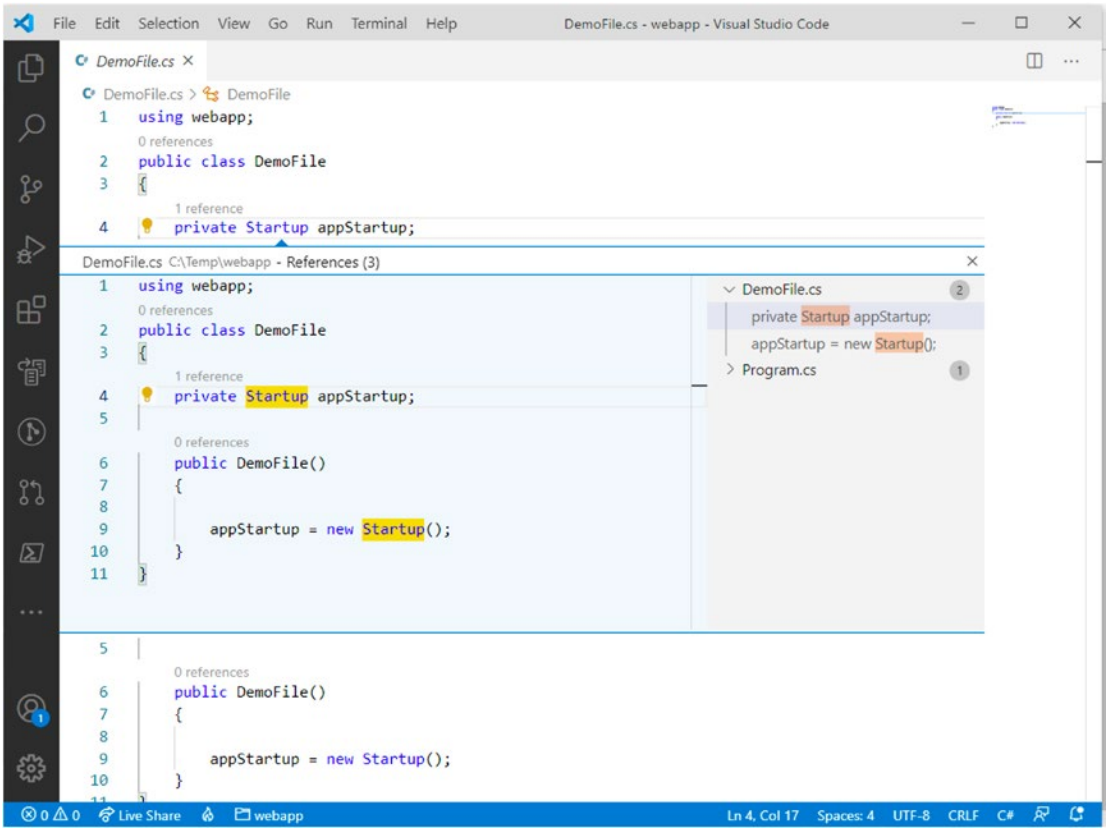

Figure 3-20. *Finding references with Peek References*

If you click an occurrence in the list on the right, the code editor opens a pop-up window containing the code where that occurrence has been found. It is very important to note that this pop-up window is interactive, which means that you can edit the code directly without the need to open the containing code file separately. This enables you to keep your focus on the code, saving time. Also, notice that the interactive pop-up window shows, at the top, the file name that contains the selected reference.

Similar to Find All References is Find All Implementations, which makes it easy to find implementations of an interface or abstract class. Figure 3-21 shows an example where an interface called `IPerson` is implemented by two classes, `Person` and `Employee`. Find All Implementations shows in a tree view all the implementations of the interface and highlights the class definition in the code editor.

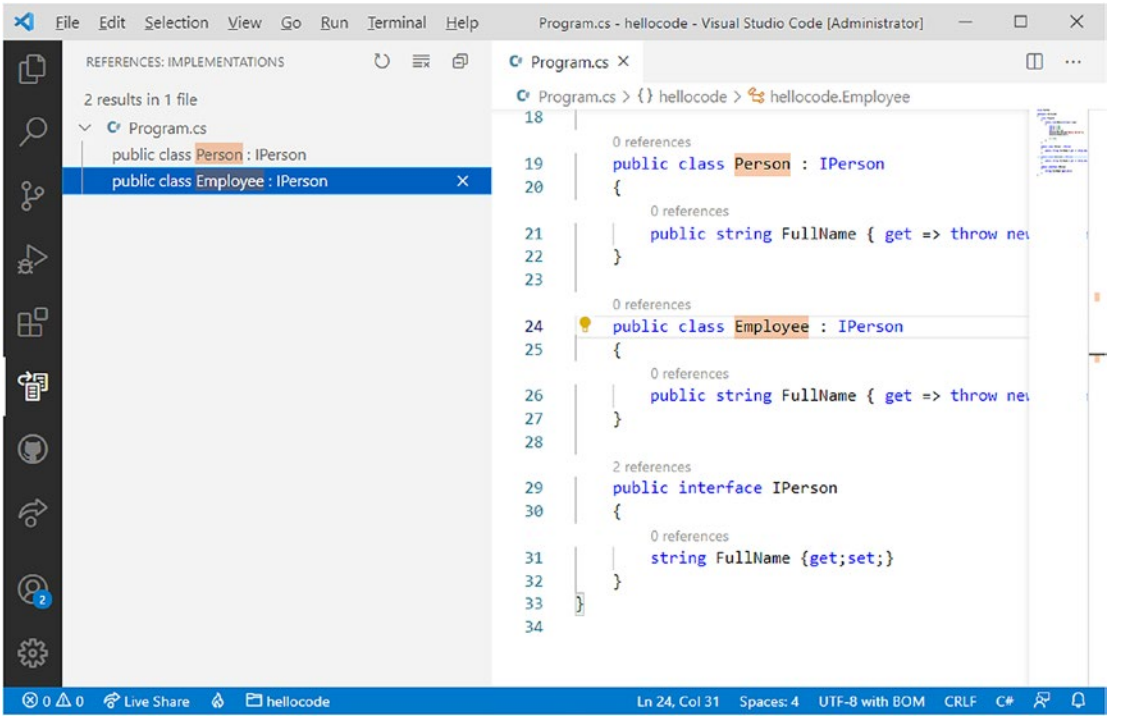

Figure 3-21. *Finding all type implementations*

Renaming Symbols and Identifiers

Renaming a symbol is a frequent task, 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 pop-up box appears. There you can write the new name without any dialogs, keeping your focus on the code. Figure 3-22 shows an example based on a symbol called app.

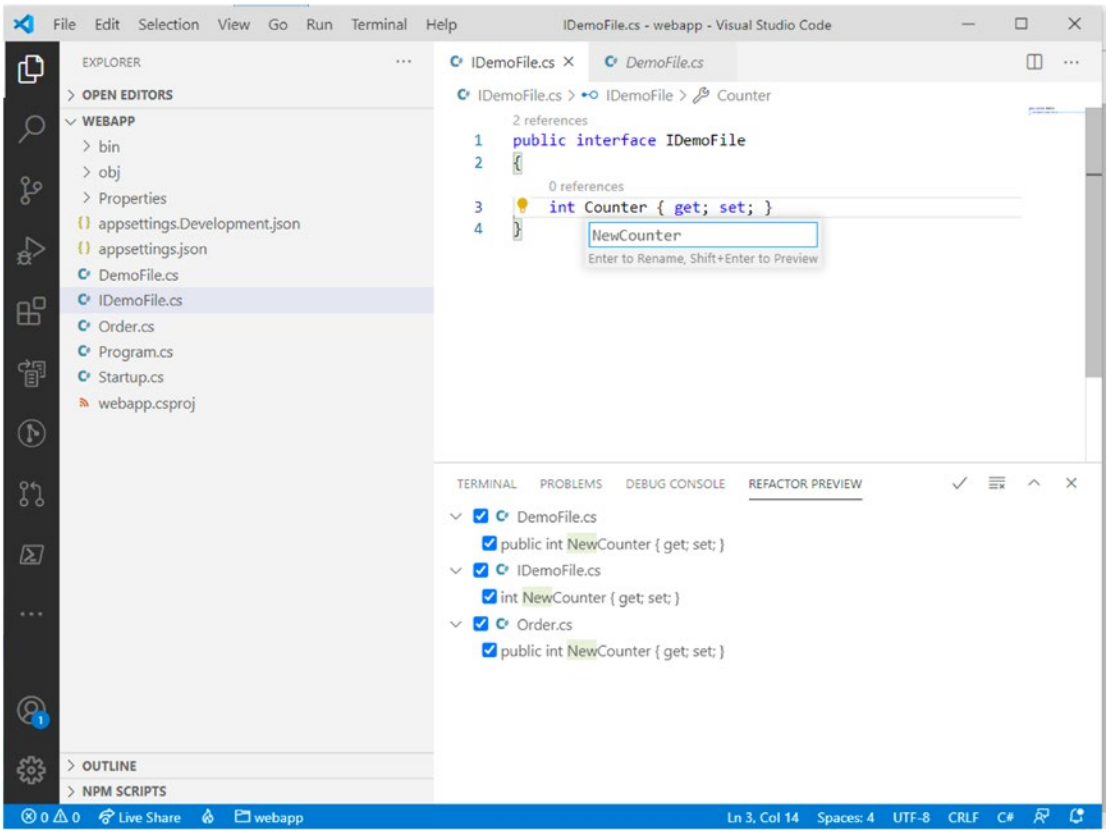

Figure 3-22. *Renaming symbols*

If you press Shift+Enter before renaming, Visual Studio Code shows a preview of how symbols will be renamed (see the REFACTOR PREVIEW tab at the bottom of Figure 3-22). Toolbar buttons in the tab enable you to accept changes (Apply Refactoring button) and reject changes (Discard Refactoring button).

By pressing Enter, 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 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 get a tooltip that describes the issue. Figure 3-23 shows two code issues, one with green squiggles (an unused local variable) and one with red squiggles (a symbol that does not exist).

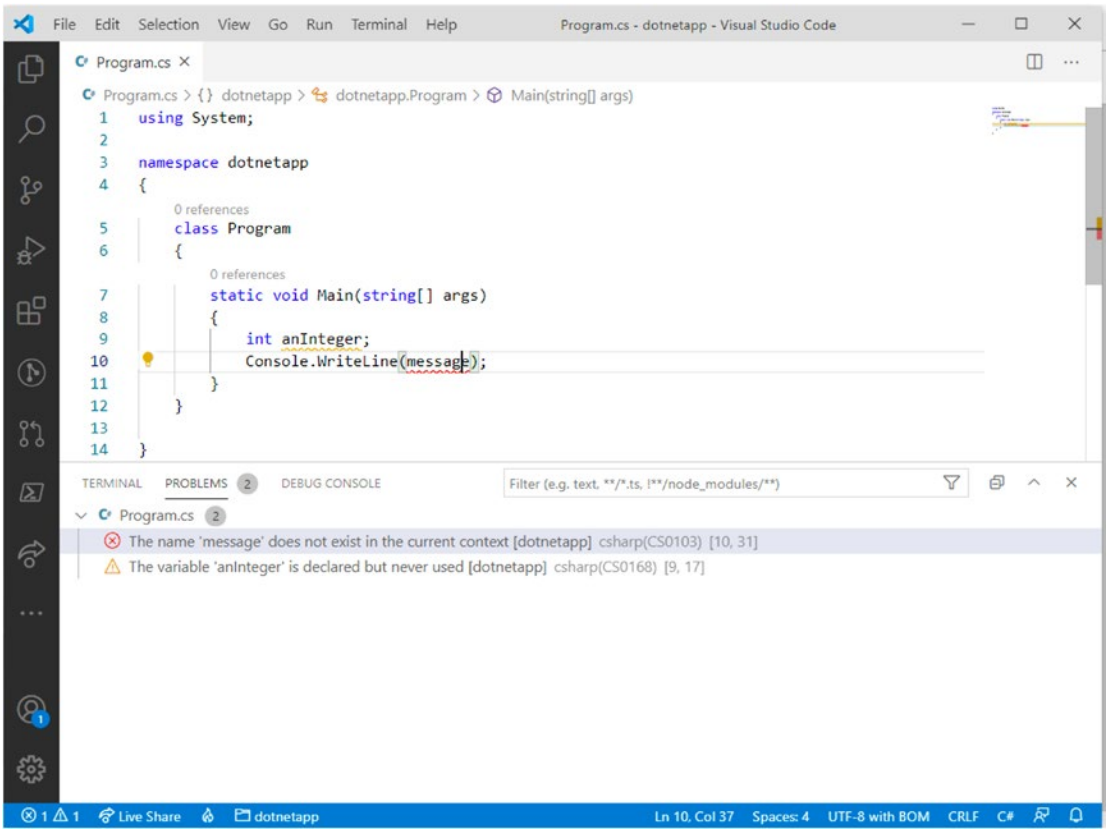

Figure 3-23. *Code issue detection as you type*

Code issues are detected as you type and they are also listed in the Problems panel. Look again at Figure 3-23 and note the icon with the shape of a light bulb. This icon is a shortcut for a tool called Light Bulb. When you click the icon, Visual Studio Code shows possible code fixes for the current context. For example, Figure 3-24 shows the suggestions that the Light Bulb provides to fix the missing symbol underlined with red squiggles.

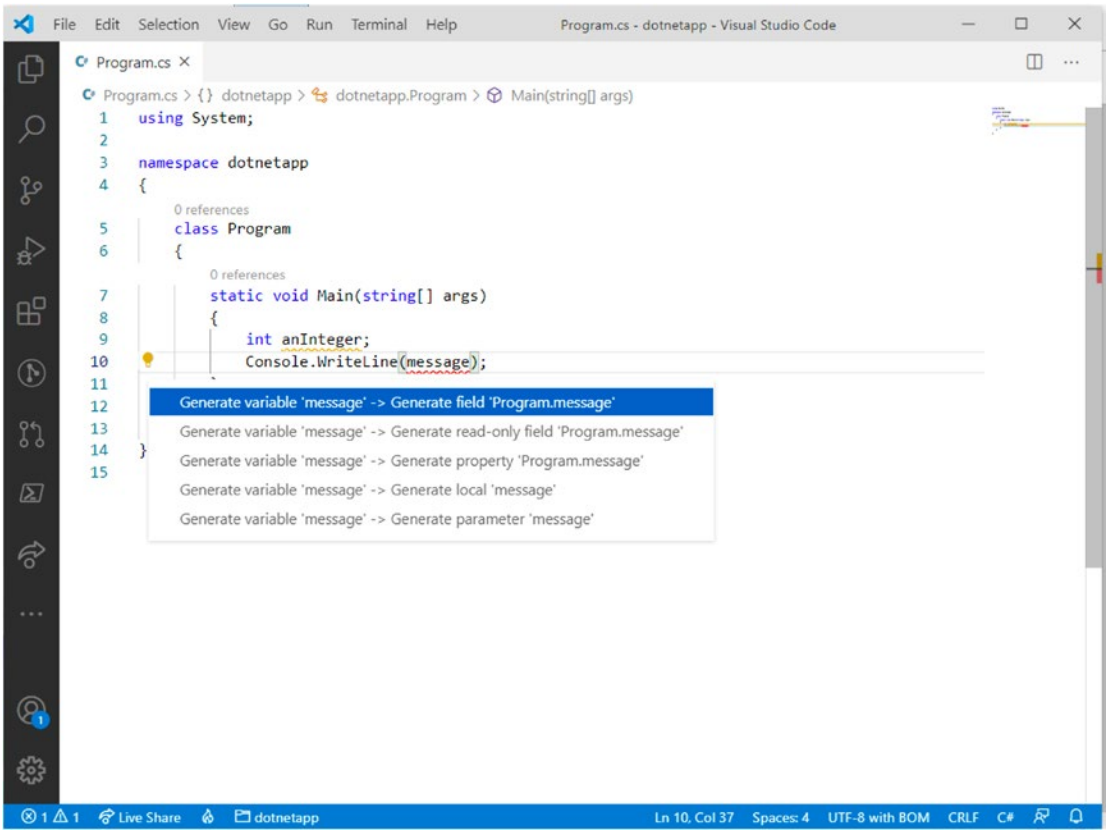

Figure 3-24. *Potential fixes suggested by the Light Bulb*

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

```
private static bool welcomeMessage;
```

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 does not have enough information to infer a different type so it will generate one based on the type parameter accepted by the first overload of the method, which is `bool` for `WriteLine`. However, when the code contains some information that Visual Studio Code could use to understand the proper type, it generates properties, fields, local variables, and parameters of the expected type. With the Light Bulb, it is also easier to generate

types on the fly. Figure 3-25 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, including the option of a nested class.

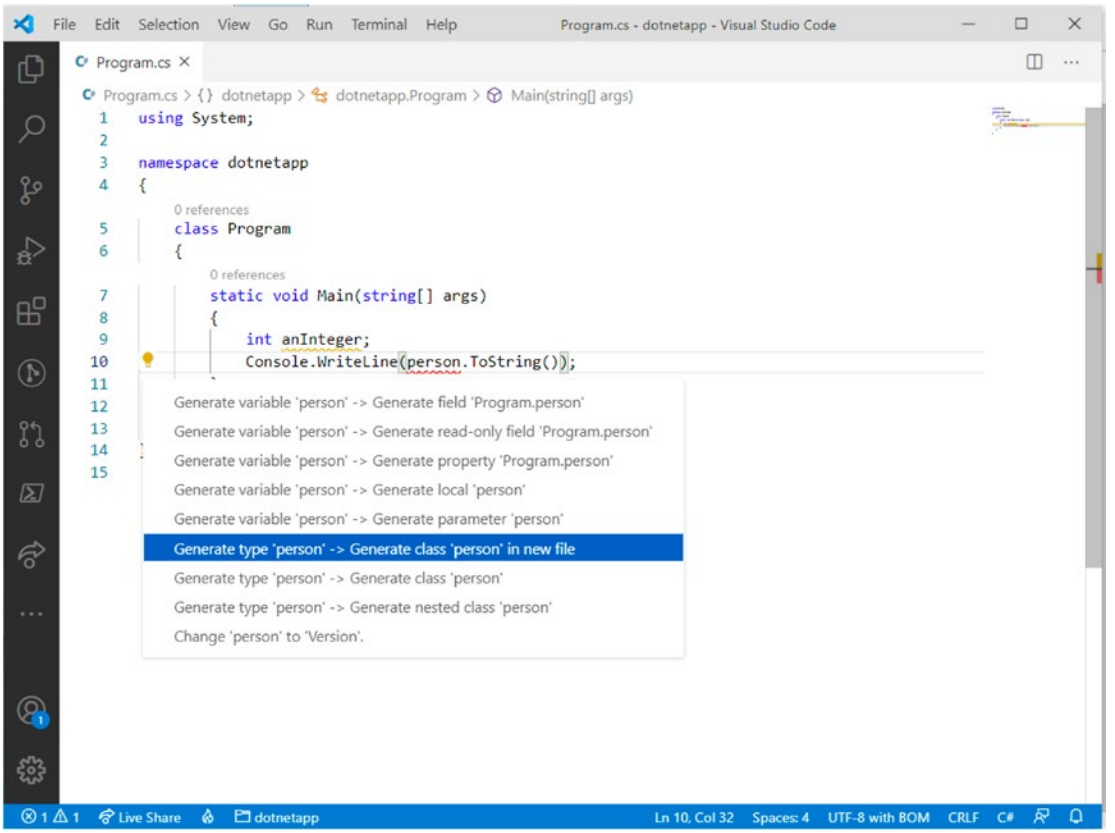

Figure 3-25. *Generating types on the fly*

The Light Bulb also 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 3-26.

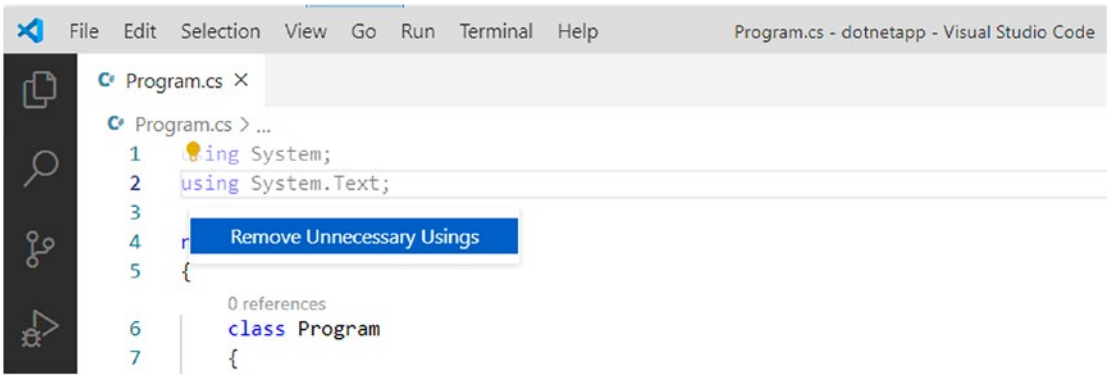

Figure 3-26. *Code refactoring made easy*

Actually, the Light Bulb tool offers even more power. Suppose you want to create a class that implements the `IDisposable` interface. As you can see in Figure 3-27, 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.

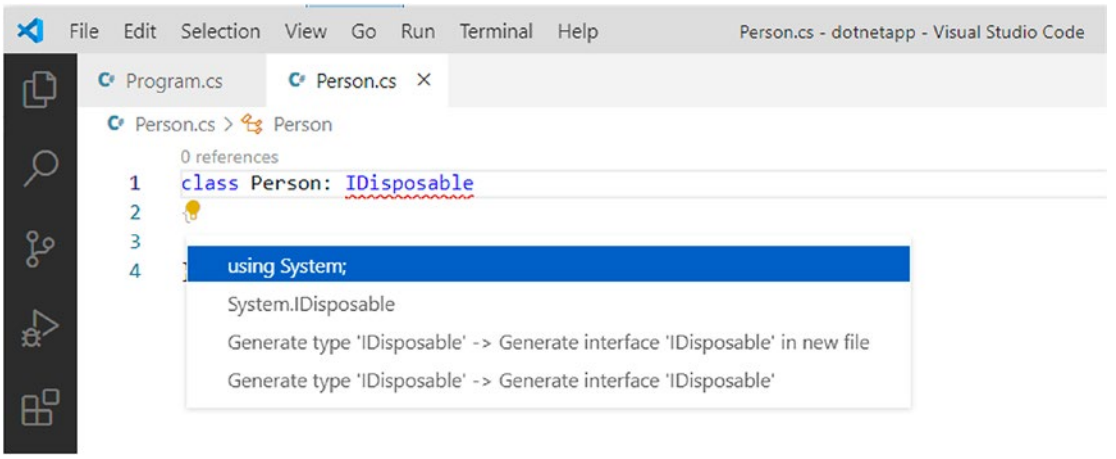

Figure 3-27. *Adding missing directives*

At this point, `IDisposable` is still underlined with a red squiggle because the code is not implementing the interface yet. When a code issue is detected on the usage of a type, you can hover your cursor over the underlined code and see an informational tooltip, as demonstrated in Figure 3-28.

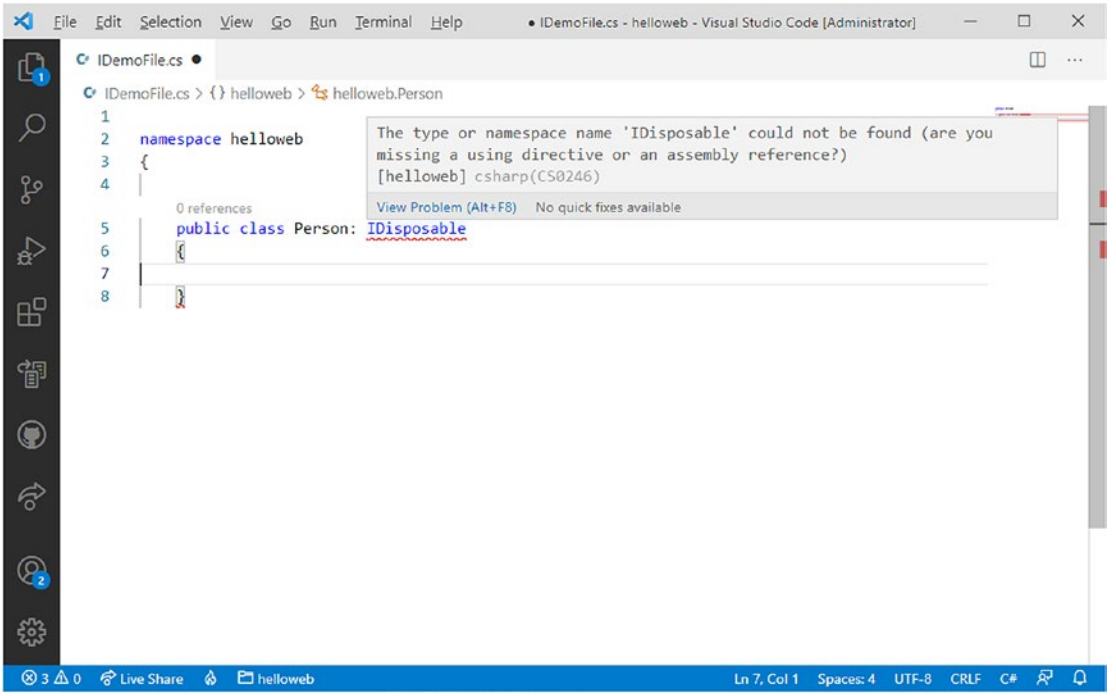

Figure 3-28. *Informational tooltips about code issues*

Tooltips disappear when you move the cursor off the issue, but you can click **Peek Problem** and dock the error description inside a red box that stays in the code editor. If you still have the Light Bulb enabled, 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 3-29).

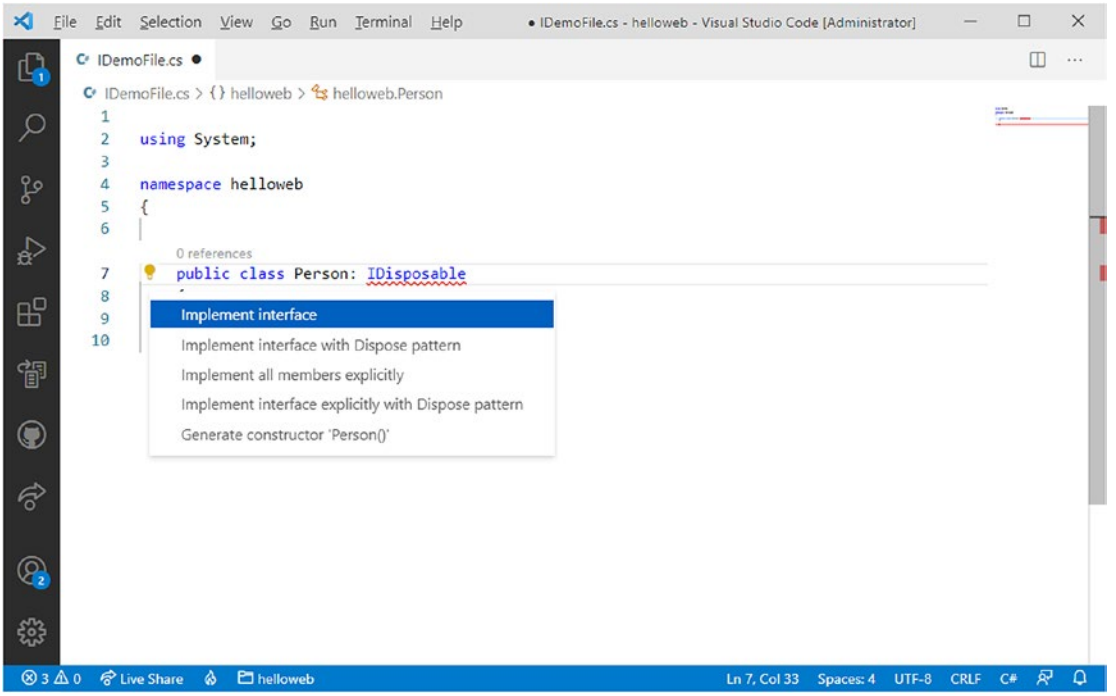

Figure 3-29. *The Light Bulb provides suggestions based on the current context*

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

```
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(disposing: true);
        GC.SuppressFinalize(this);
    }
    #endregion
}
```

You would get a similar result, but with different implementation, if you choose one of the other possible code fixes. Though it is not possible to show examples for all the code fixes that Visual Studio 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 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 in Chapter 4.

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 independence from proprietary project systems as well as its 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 to organize a structured view of the list of files in the folder:

- Tsconfig.json
- Jsconfig.json
- Package.json
- Project.json
- .sln Visual Studio solutions for and .csproj project files for .NET with the C# extension installed

A. Del Sole, *Visual Studio Code Distilled*, https://doi.org/10.1007/978-1-4842-6901-5_4

If VS 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 to work 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. In addition, it checks if an extension is available on the Marketplace for the selected language and, if so, offers to install it to improve the editing experience. 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 pop-up box shows the list of open files; by pressing Ctrl+Tab, you can browse files and cycle through the files in the list, and when you release the keys, the selected file becomes the active editing window.

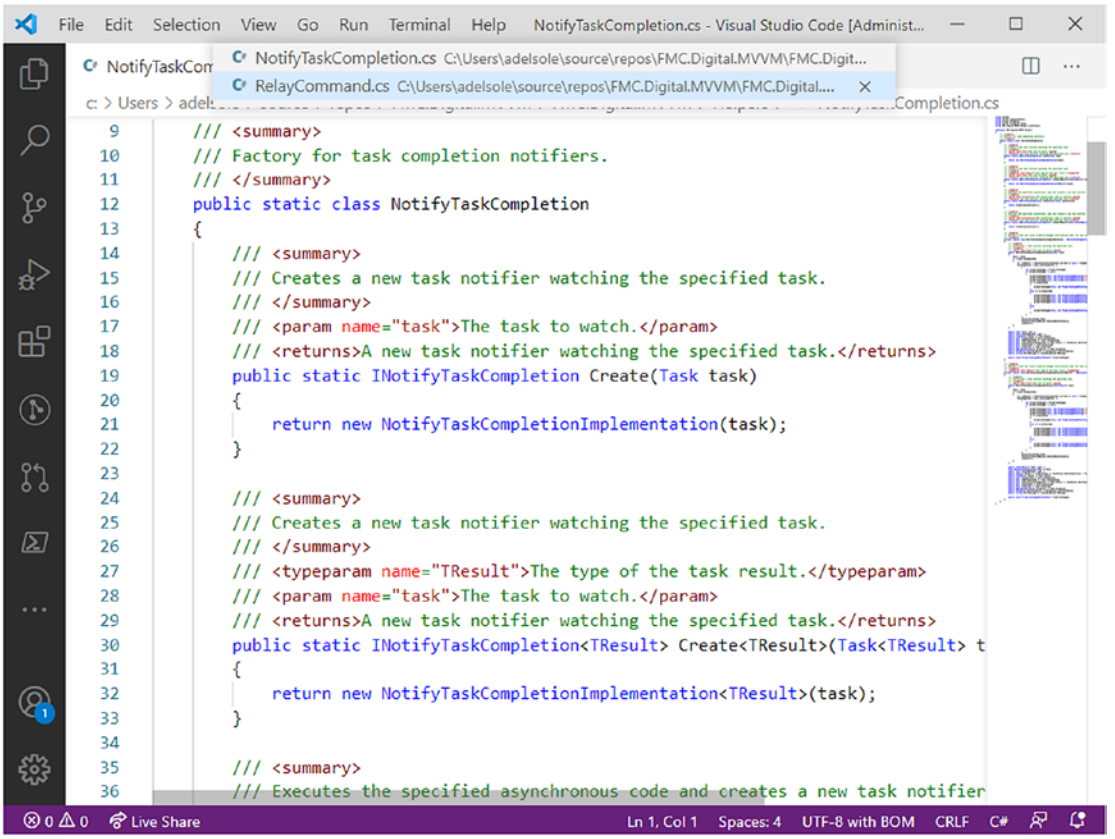

Figure 4-1. *Quickly navigating between open editors*

You can close an editor simply by clicking the **Close** button in the upper-right corner of each tab, or by using **File ➤ Close Editor**. You can also quickly close all open editors with the **Close All** command in the top-right options, under the **...** shortcut.

Note In 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 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 on 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. 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 from which you can select the new language for the current file. You can also start typing a language name to filter the list.

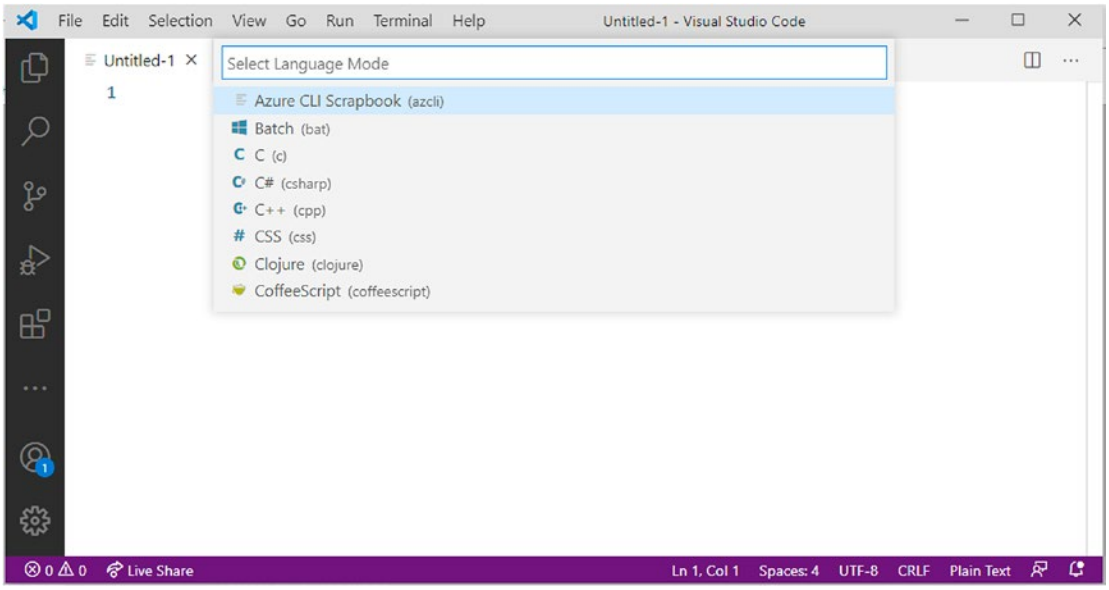

Figure 4-2. *Selecting the language for a new file*

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 are first asked to select an action between **Reopen with Encoding** and **Save with Encoding**. Click the first option to 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).

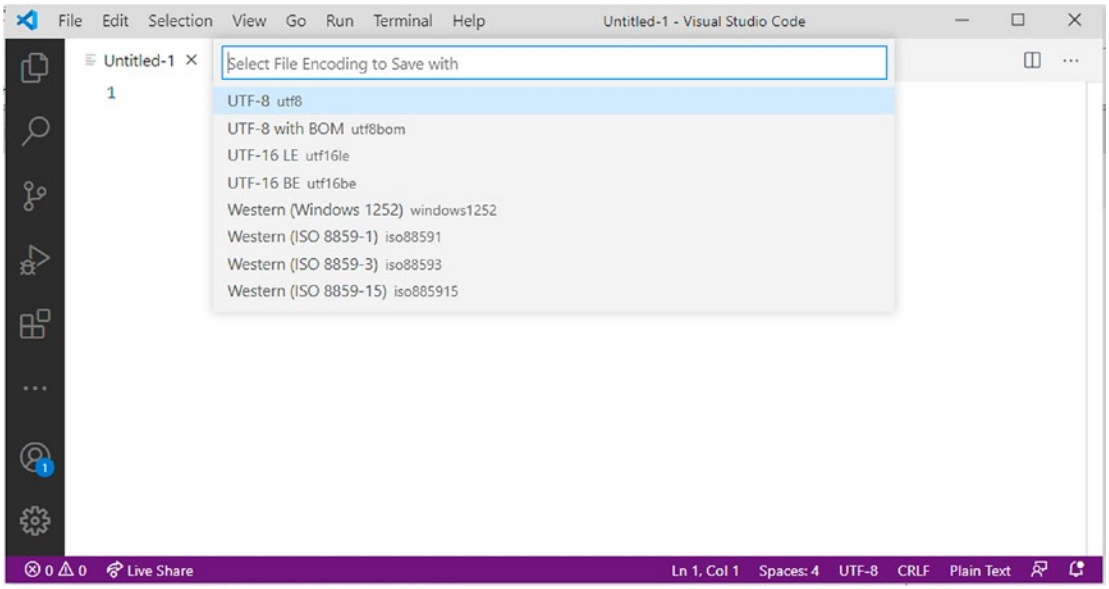

Figure 4-3. *Selecting the file encoding*

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. On Windows, the default sequence is CRLF, while on macOS and

Linux it is LF. 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 opens a search box in which you can type the line number you want to go to, and the line of code is immediately highlighted as you type (see Figure 4-4). When you press Enter, the cursor is moved to the start of the selected line.

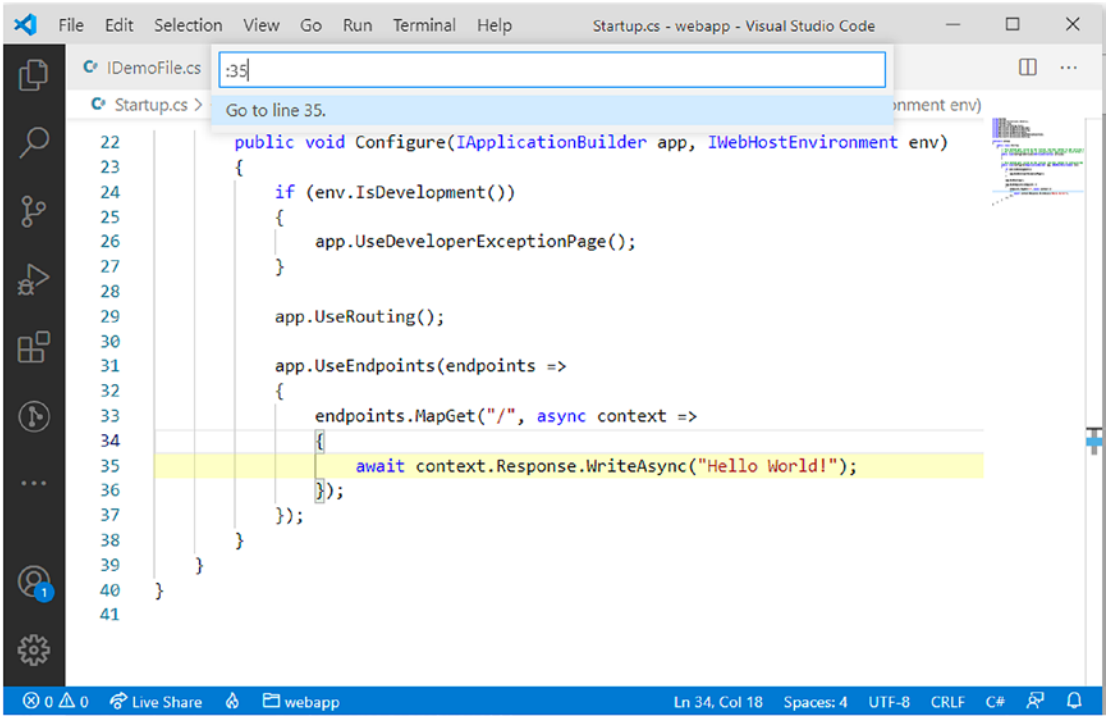

Figure 4-4. *Quickly moving to a specific line of code with* ***Go to Line***

Working with Folders and Projects

Unlike 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 the following:

- *MSBuild solution files (.sln)*: In this case, VS 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 VS Code needs the Microsoft C# extension installed to properly treat solution files. Note 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.
- *tsconfig.json files*: If found, VS Code knows these represent 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, VS Code knows these represent 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 VS Code automatically resolves the project type based on the folder's content.
- *project.json files*: If found, VS Code treats the folder as a .NET Core project.

Note Opening a .sln, .csproj, 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 none 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 explore how to work with folders and supported projects in Visual Studio Code, with corresponding examples.

Opening a Folder

You open a folder via **File ➤ Open Folder** or via the **Open Folder** shortcut on 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 also provides 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 creates a structured view in the Explorer bar, where it shows all files and subfolders that belong to the main folder. Figure 4-5 shows an example based on a TypeScript project.

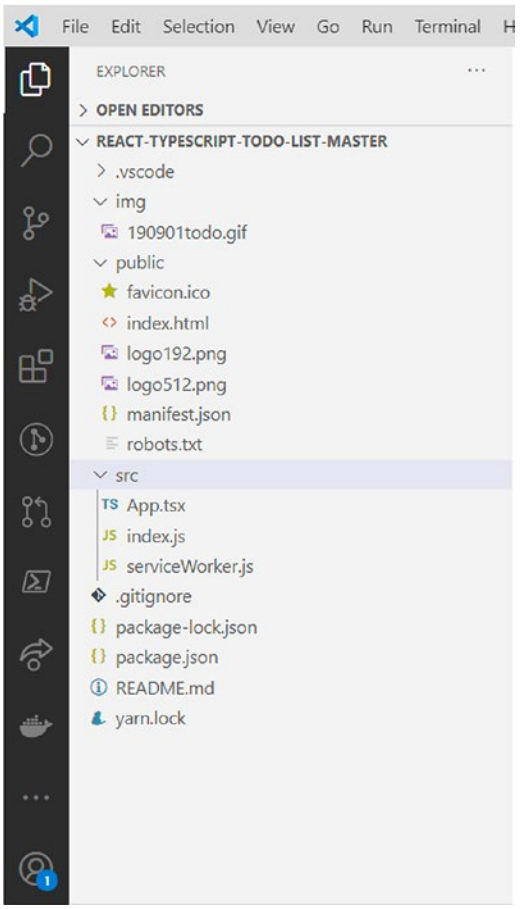

Figure 4-5. *The structured view of files and folders in Explorer*

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 Solutions

When you open a folder that contains a .NET solution based on the MSBuild project system (.sln file) or a C# project (.csproj 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.

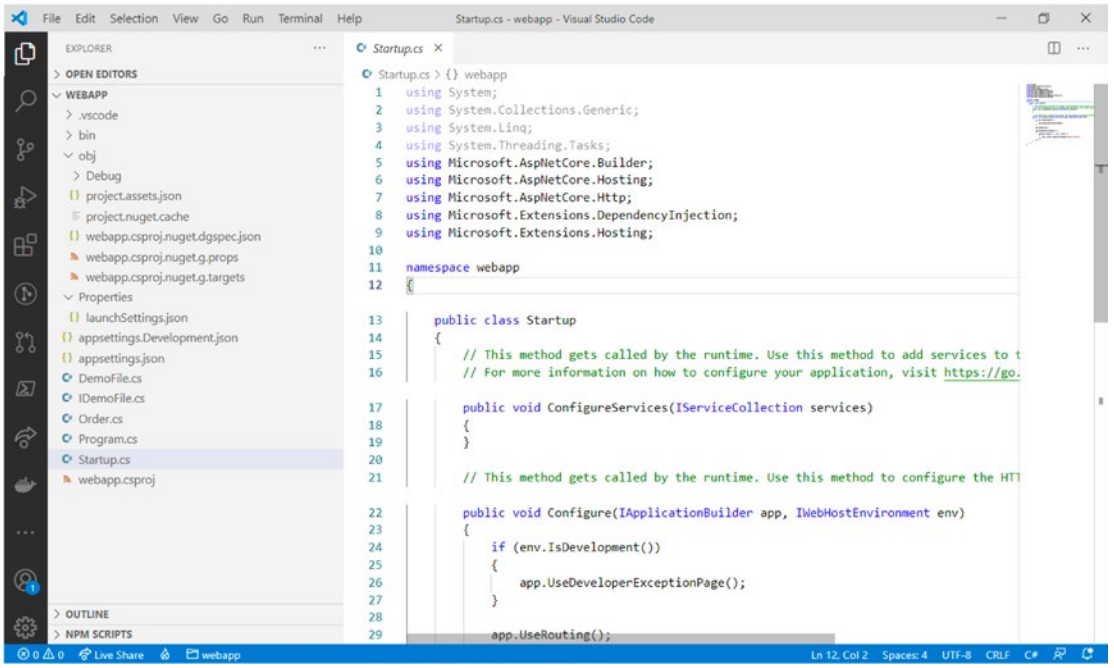

Figure 4-6. *A .NET solution opened in Visual Studio Code*

Notice how the root level in Explorer is the project name. You can browse folders, browse code files, and edit anything that Visual Studio Code can properly recognize. It is worth mentioning that VS Code can certainly open any MSBuild solution, not only .NET Core solutions, but it is only able to run and debug .NET Core applications, not .NET Framework solutions. For instance, the most recent version of .NET Core allows creating Windows Presentation Foundation (WPF) and Windows Forms projects; Visual Studio Code and the C# extension support opening this type of solutions as well as running and debugging code. WPF and Windows Forms projects created for the .NET Framework can still be opened in VS Code, and you will still benefit from the structured folder view in the Explorer bar and the 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.

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.

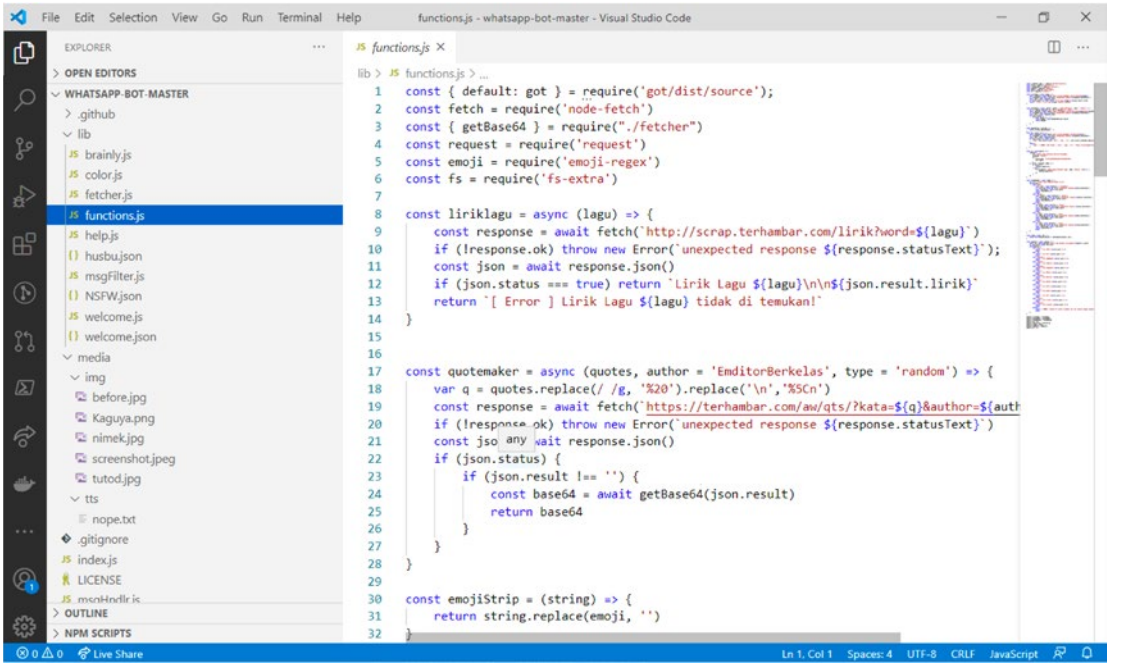

Figure 4-7. *A JavaScript project opened in Visual Studio Code*

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

Opening Loose Folders

Visual Studio Code supports opening folders that contain unrelated, loose assortments of files. VS 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.

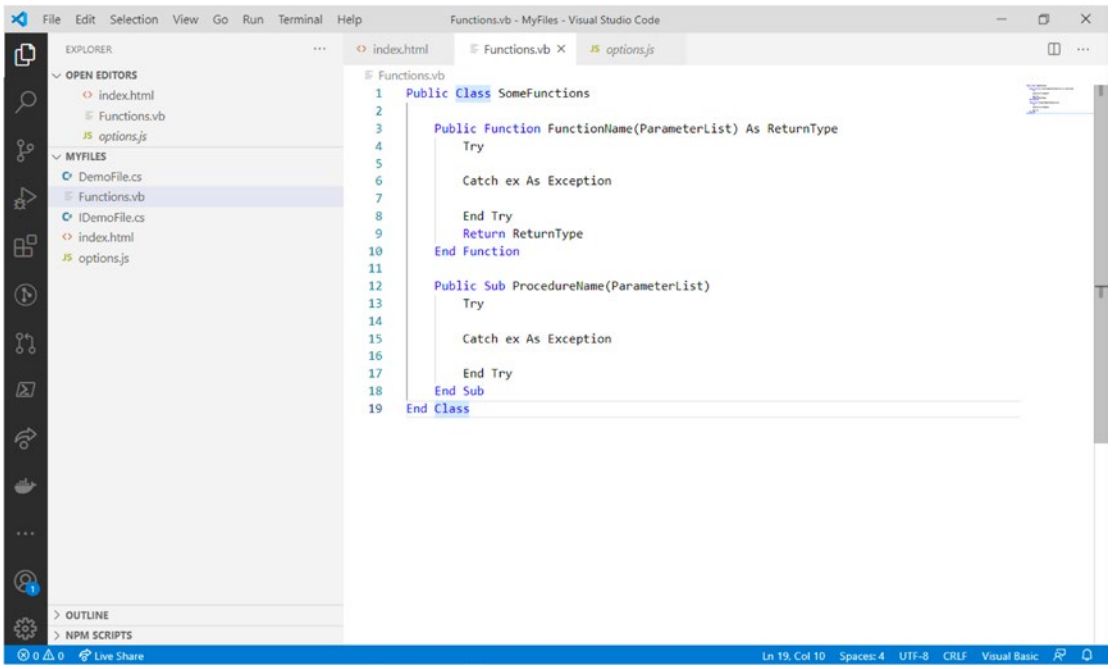

Figure 4-8. *A folder containing a loose assortment of files*

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 the concept of a *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, that includes a .NET Core Web API project, a JavaScript project, and a loose folder.

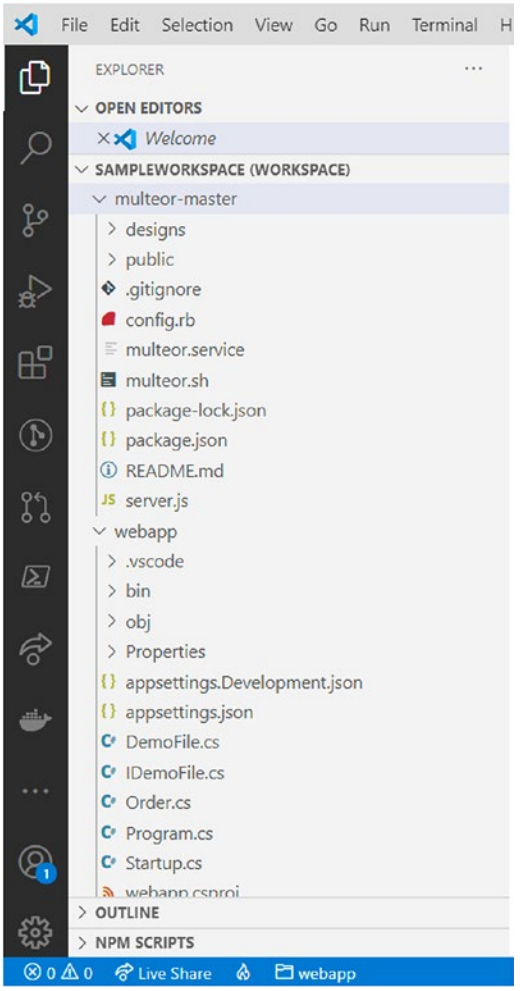

Figure 4-9. *A workspace can group multiple projects and folders into one logical container*

The multeor-master folder contains the files for 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 sections, I will explain in more detail how to create and open workspaces and what is the structure of a workspace file.

Creating Workspaces

You can create a workspace regardless of whether you already have a folder open. If you do already have a folder open, select **File ➤ Save Workspace As** and 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 the .code-workspace extension, the structure of which will be explained shortly.

The workspace name is simply the file name without the .codeworkspace extension and is 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**. Added folders are displayed in the Explorer bar under the workspace root.

If you do not have any folders already open, you can start either with **File ➤ Save Workspace As** or with **File ➤ Add Folder to Workspace**. With the first option, you basically create an empty workspace with a name, and then you add folders as described in the preceding text. With the second option, you instead create an empty, untitled workspace starting from an existing folder. In this case, in fact, the Explorer bar shows **UNTITLED (WORKSPACE)** as the new workspace name. When you save the workspace as described in the preceding text, the Explorer bar shows 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 group their content under the workspace root and let you work as 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 simply results in viewing the file content, not opening the workspace. Similarly, opening a folder that contains a .code-workspace file results in opening only 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 a .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 how the workspace file of the example shown in Figure 4-9 looks on my machine, and will vary on your computer:

```
{
    "folders": [
        {
            "path": ".\MyFiles"
        },
        {
            "path": "C:\\Source\\webapp"
        },
        {
            "path": "C:\\Source\\multeor-master"
        }
    ]
}
```

Notice that the full pathname of a folder is provided only if the folder is not in the same location of the workspace file. In this case, the .code-workspace file, the webapp folder, and the multeor-master folders are all in the same location; instead, the MyFiles folder is located under a different folder. If you want to see for yourself the structure of a workspace file, you can open it in 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.

Visual Studio Code also supports a number of project systems such as .NET Core, TypeScript, and JavaScript, and it allows for creating and managing workspaces. Workspaces are logical containers of folders that make it easy to have multiple projects and folders under the same visual root. VS Code is not only a very powerful code editor but also a very flexible environment that 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. Examples of popular customizations are color themes and key bindings. Table 5-1 summarizes customizations and extensions in VS Code.

A. Del Sole, *Visual Studio Code Distilled*, https://doi.org/10.1007/978-1-4842-6901-5_5

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 type 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 VS 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 to customize Visual Studio Code by 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. A brief introduction to color themes was given at the beginning of Chapter 1, but now you will get more details.

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 is shown in the Command Palette, as you can see in Figure 5-1.

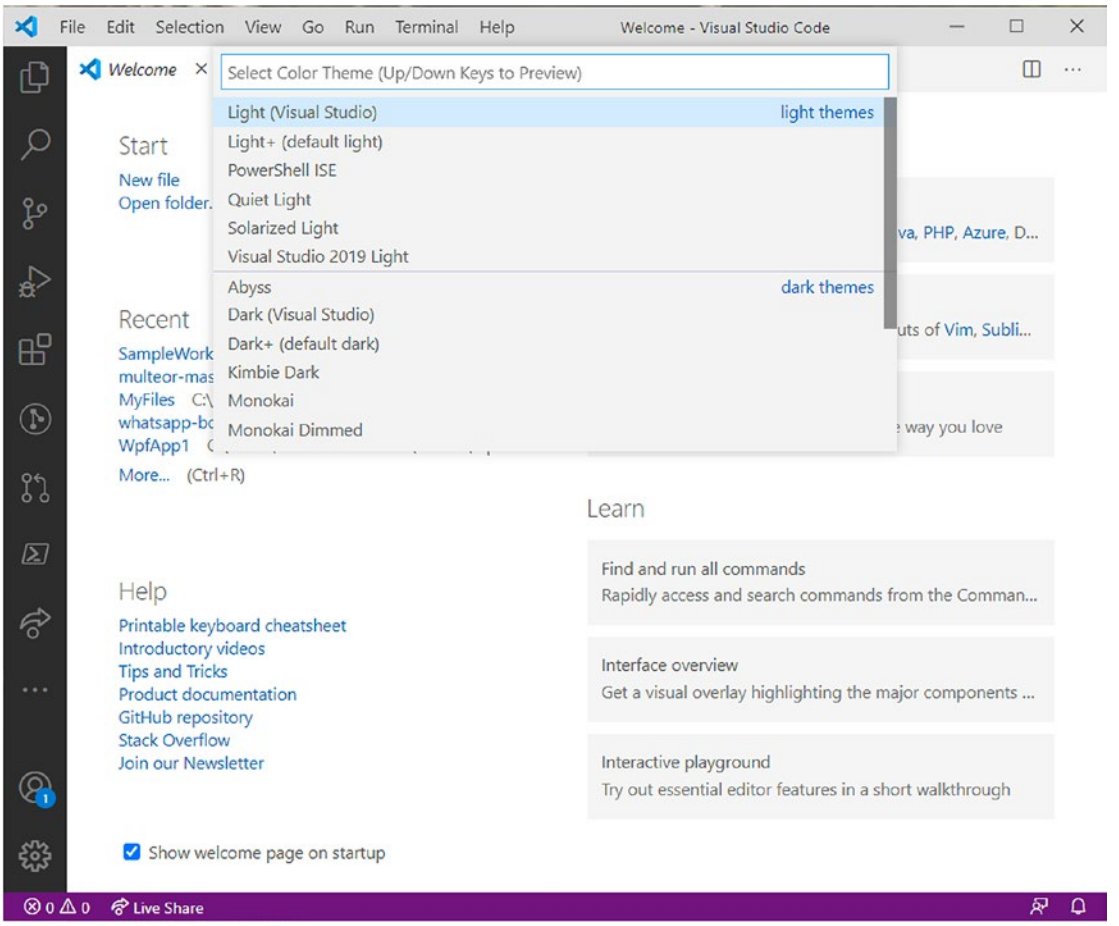

Figure 5-1. *Selecting a theme*

Themes are divided into light themes, dark themes, and high-contrast themes. Once you select a different color theme, it is 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; try out the other themes to find one that suits you.

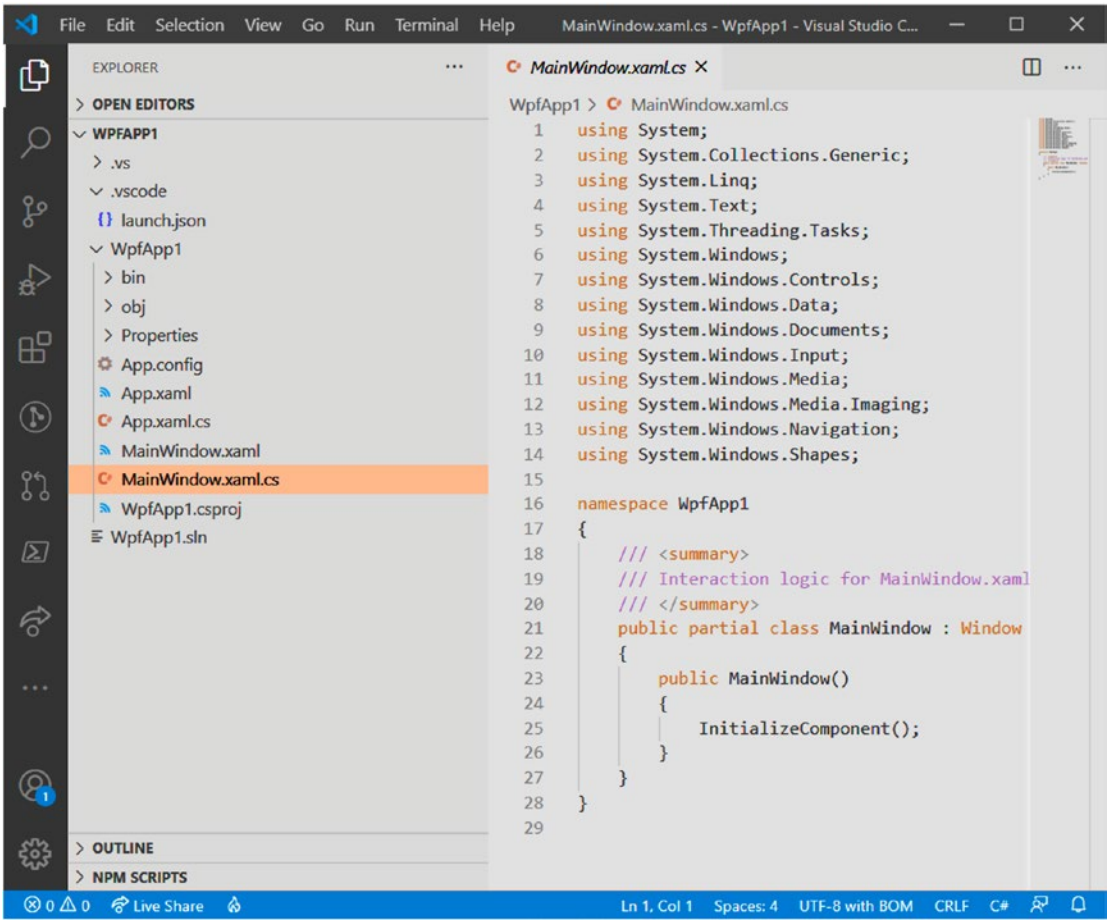

Figure 5-2. *The Dark (Visual Studio) theme applied to Visual Studio Code*

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 exception. 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. The following subsections cover both user setting 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**. When you do this, the settings editor appears, as represented in Figure 5-3.

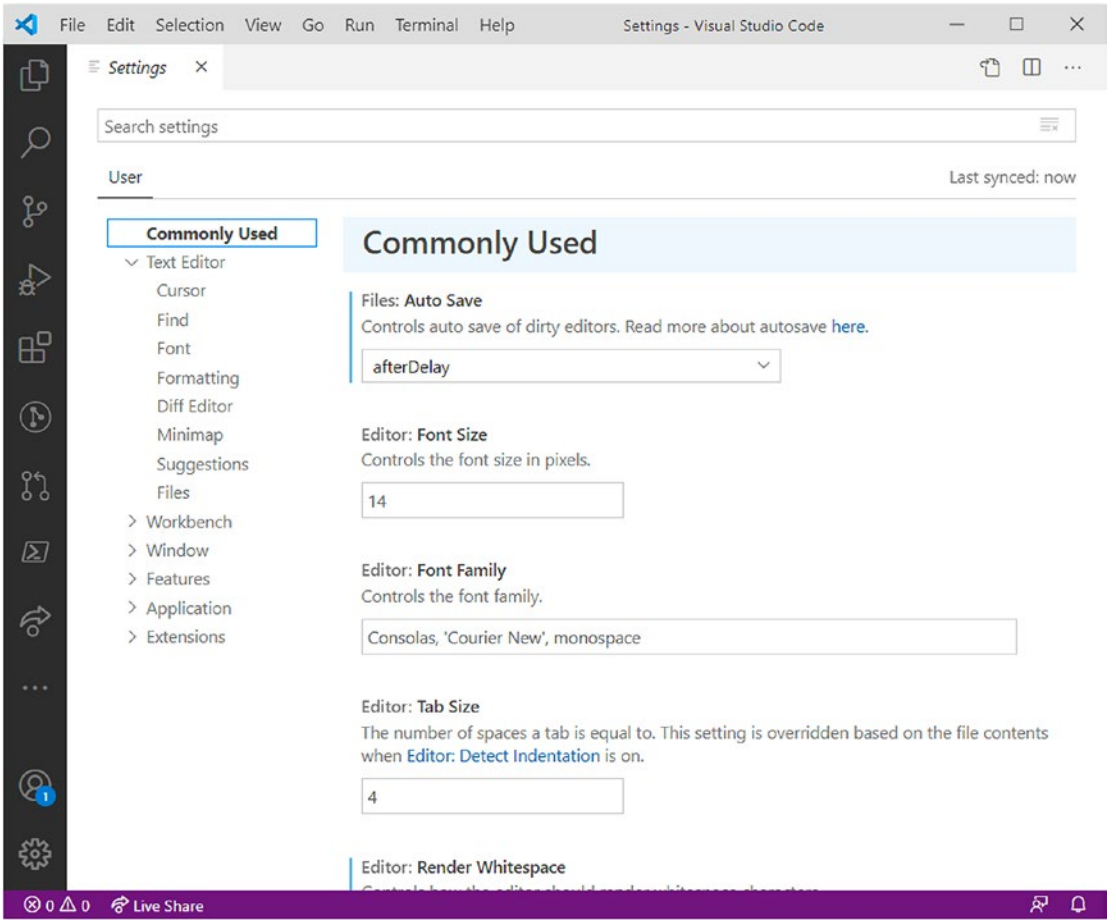

Figure 5-3. *Working with user settings*

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 settings categories manually, or you can just scroll the list of settings, and the related category is automatically highlighted as you scroll. For instance, you could control the behavior of the Explorer bar by locating and selecting **Explorer** under the **Features** category, and there you could change the current settings, as shown in Figure 5-4.

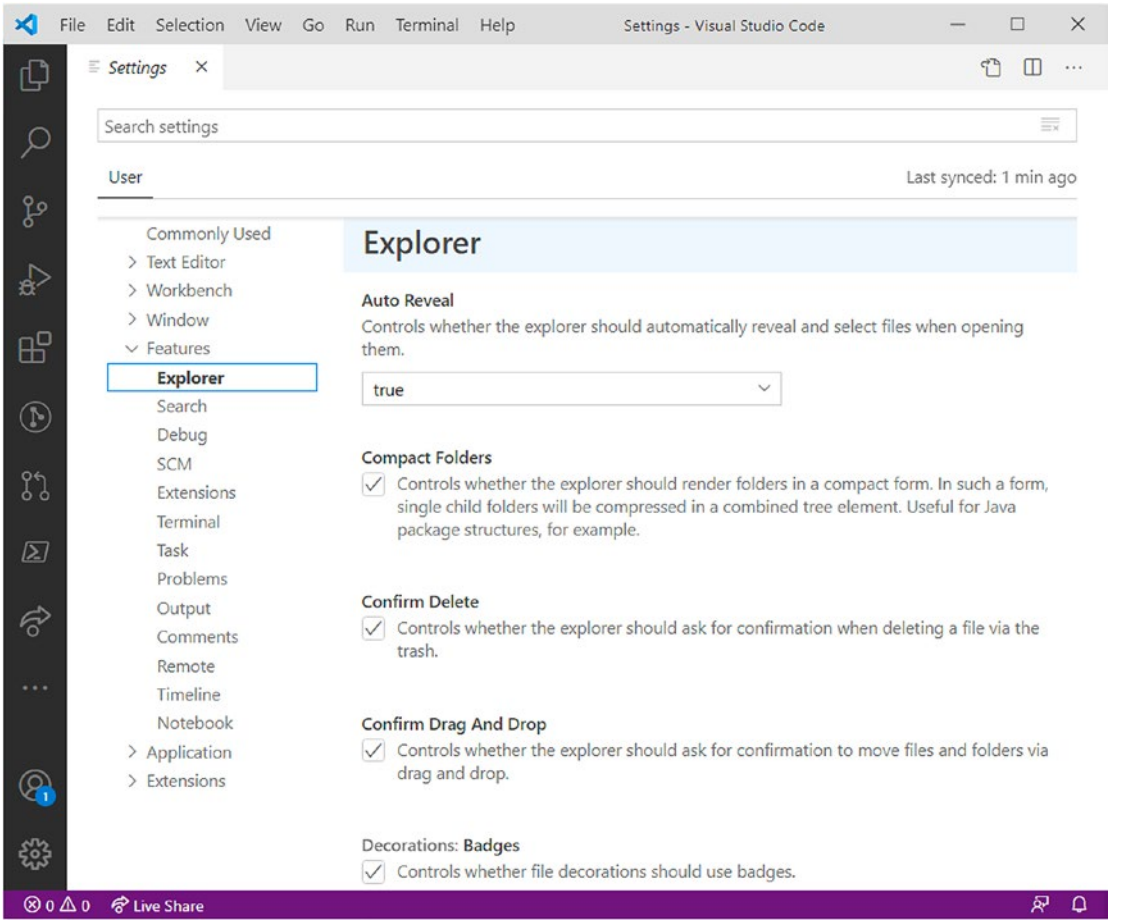

Figure 5-4. *Changing user settings*

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).

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.

It is important to understand how this file works, so click the **Open Settings (JSON)** button located above the search bar and represented by a sheet icon with a plus symbol overlayed (the first from left to right). Figure 5-5 shows how the editor appears at this point.

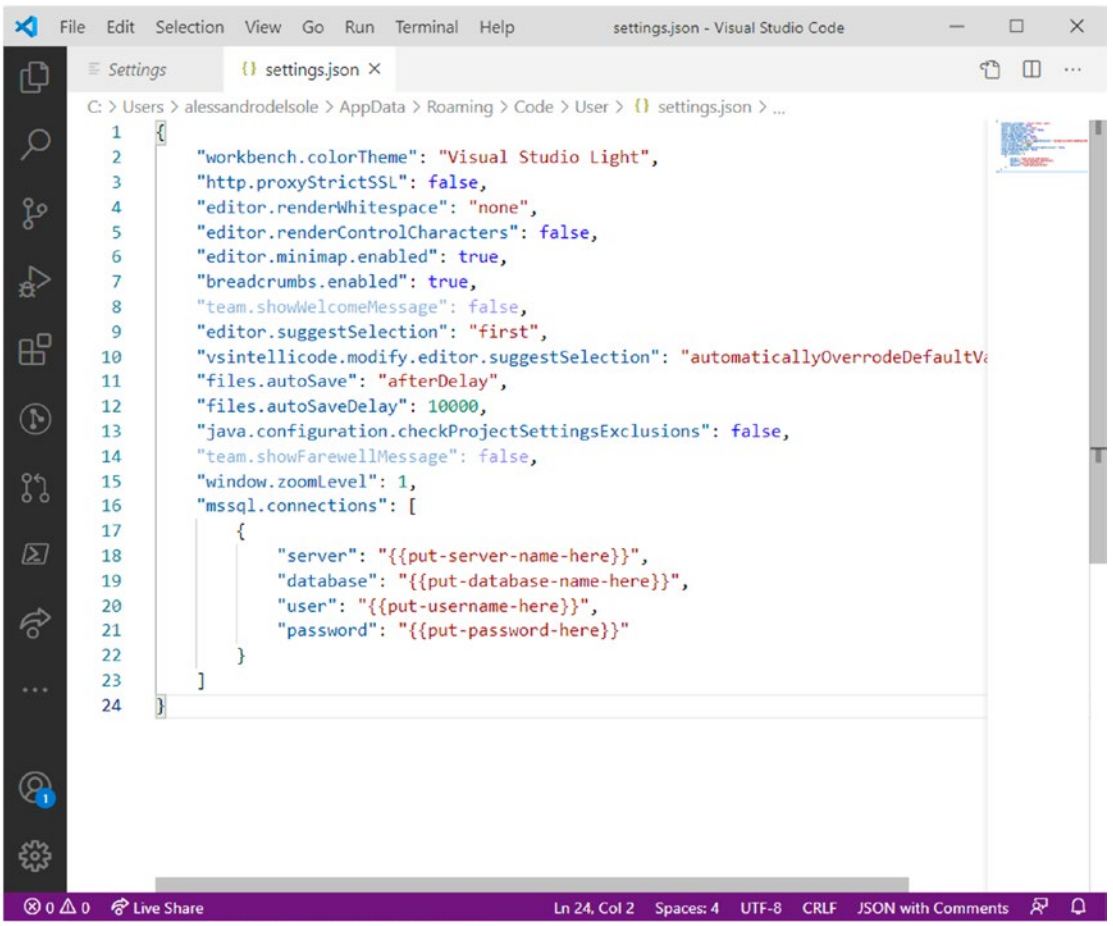

Figure 5-5. *Working with the settings.json file*

As you can see, the editor for settings.json allows you to define custom settings by overriding one or more default settings. It is worth mentioning that changes you do in this file are at the user or workspace level only, and do not affect general settings of VS Code. Figure 5-5 shows an example of 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. The code editor also reports errors, such as missing commas or curly braces, as you would expect when editing a JSON file. In Figure 5-5 you can also see that it is possible to customize settings for an extension: I have the Microsoft SQL Server extension installed on my machine, and settings.json allows for specifying the extension settings such as the server address and credentials. Every time you modify a setting in the user interface, the related JSON is updated in settings.json.

IntelliSense also allows you to get more information about a given settings by clicking the rollover, which shows hints about the setting with a convenient tooltip, exactly as you would expect after learning about IntelliSense's features in Chapter 3. When you are 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 probably is 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 to take some manual steps.

The first thing to do is make sure that the sites described in Table 5-2 are in the allowed applications list of the firewall.

Table 5-2. *Sites to Be Allowed by a Firewall*

URL	Description
update.code.visualstudio.com	Visual Studio Code download and update server
code.visualstudio.com	Visual Studio Code documentation
go.microsoft.com	Microsoft link forwarding service
vscode.blob.core.windows.net	Blob storage for Visual Studio Code
marketplace.visualstudio.com	Visual Studio Marketplace
*.gallery.vsassets.io	Visual Studio Marketplace
*.gallerycdn.vsassets.io	Visual Studio Marketplace
rink.hockeyapp.net	Crash reporting service
bingsettingssearch.trafficmanager.net	In-product settings search
vscode.search.windows.net	In-product settings search
raw.githubusercontent.com	GitHub repository raw file access
vsmarketplacebadge.apphb.com	Visual Studio Marketplace badge service
az764295.vo.msecnd.net	Content Delivery Network (CDN) for Visual Studio Code downloads
download.visualstudio.microsoft.com	Visual Studio download service, which includes dependencies for extensions such as C# and C++

The next step is to configure VS Code to work with the proxy. Actually, if the `http_proxy` and `https_proxy` environment variables have been defined at the system level, VS Code uses 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.

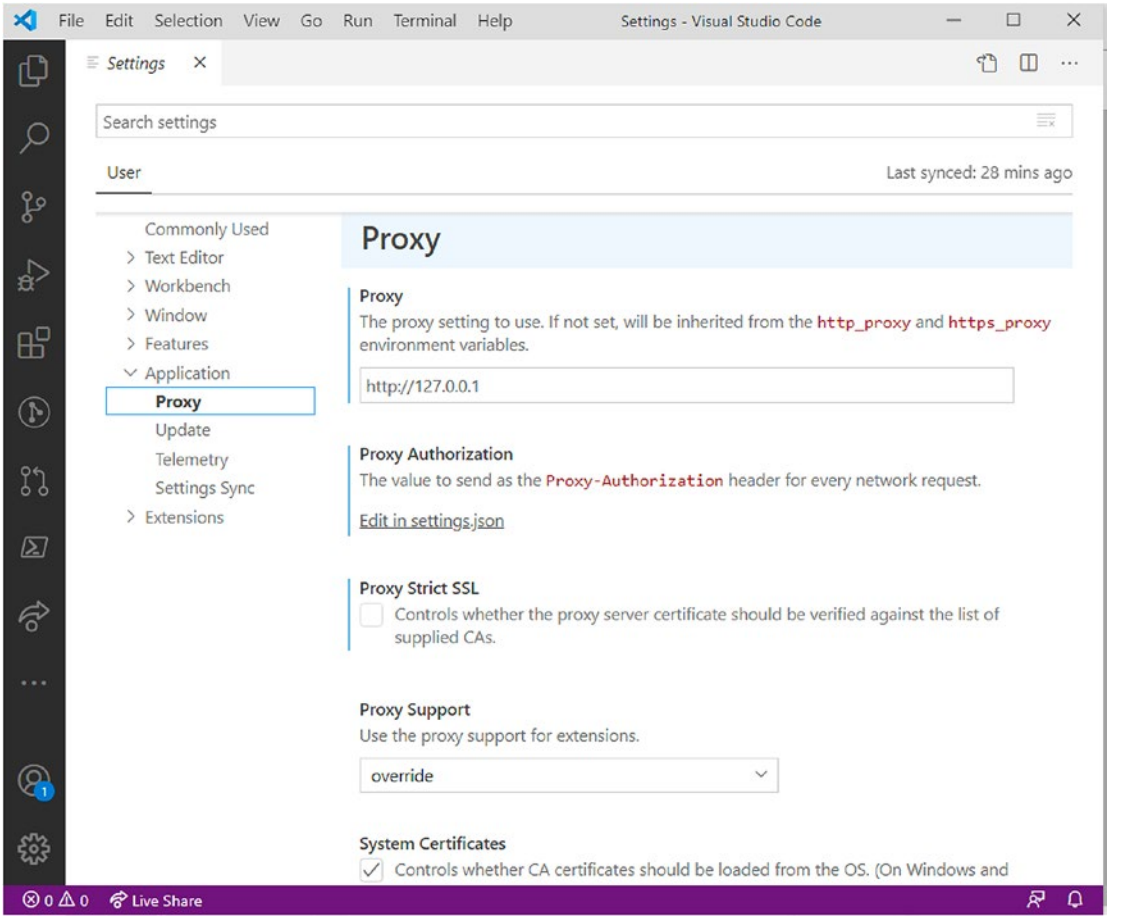

Figure 5-6. *Configuring VS Code to work behind a proxy server*

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 the `http.proxyAuthorization` key. Also, check the **Proxy Strict SSL** checkbox if the certificate should be verified against the list of supplied certification authorities.

Save your changes and check 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.

Privacy Settings: Telemetry

By default, Visual Studio Code anonymously collects and sends to Microsoft information about usage, errors, and crashes. You can disable one or more of these telemetry settings by scrolling the user settings to the **Telemetry** group, located under the **Application** category (see Figure 5-7).

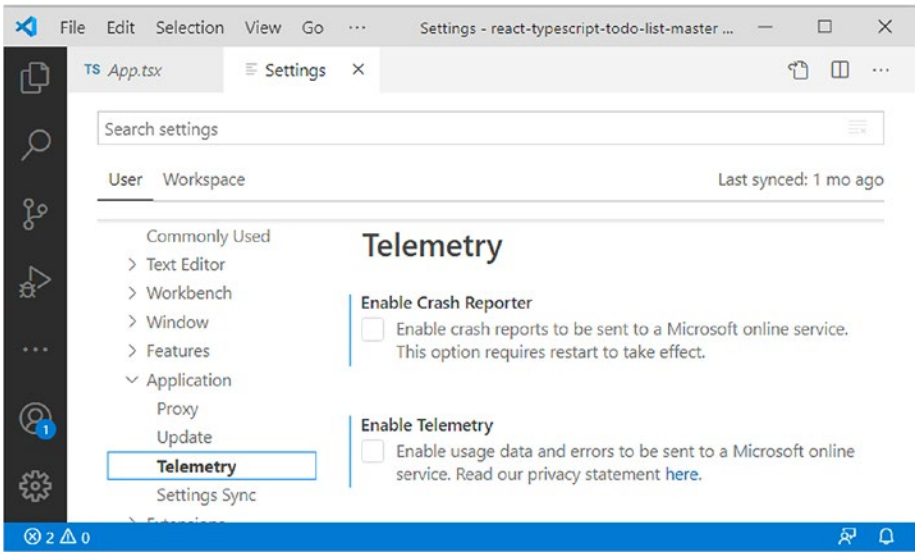

Figure 5-7. *Managing telemetry in Visual Studio Code*

The **Enable Crash Reporter** option allows sending crash reports to Microsoft, while the **Enable Telemetry** allows sending usage data and errors. A shortcut to the privacy policy is also available, and I recommend that you read it before enabling one or both the options.

Synchronization Settings

In Chapter 1 you learned that Visual Studio Code allows for synchronizing settings across different installations. You have full control over items that can be synchronized through the **Settings Sync** group under the **Application** category.

You can decide which extension will be synchronized and which not, you can exclude specific settings from synchronization, and you can disable or re-enable keybinding synchronization. Apart from the latter, which is managed via a simple check box, you need to make your changes in the settings.json file. The **Ignored Extensions** and **Ignored Settings** hyperlinks enable you to edit specific blocks of settings about extensions and general settings, respectively. As mentioned previously, IntelliSense will help adding the available settings. Figure 5-8 shows an example, but keep in mind that available settings may vary on your machine, especially depending on the extensions you have installed.

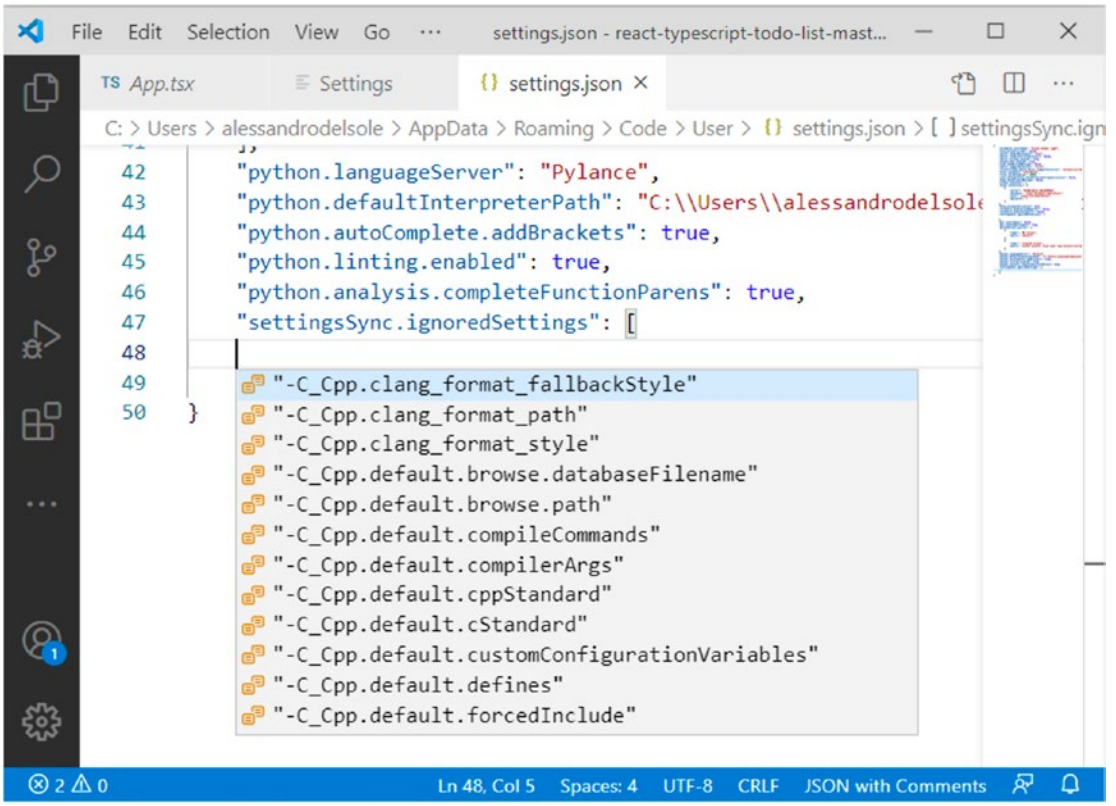

Figure 5-8. *IntelliSense helps manage synchronization settings*

Understanding Workspace Settings

Differently from user settings, which globally apply to VS Code's environment, workspace settings apply to the current workspace and folders in the workspace. As an implication, you first need to open an existing workspace, or add an existing folder to a new workspace, to customize workspace settings.

Next you still select **File ➤ Preferences ➤ Settings**. At this point the settings editor shows three tabs: one for user settings, one for workspace settings, and one for individual folders within the workspace, as demonstrated in Figure 5-9.

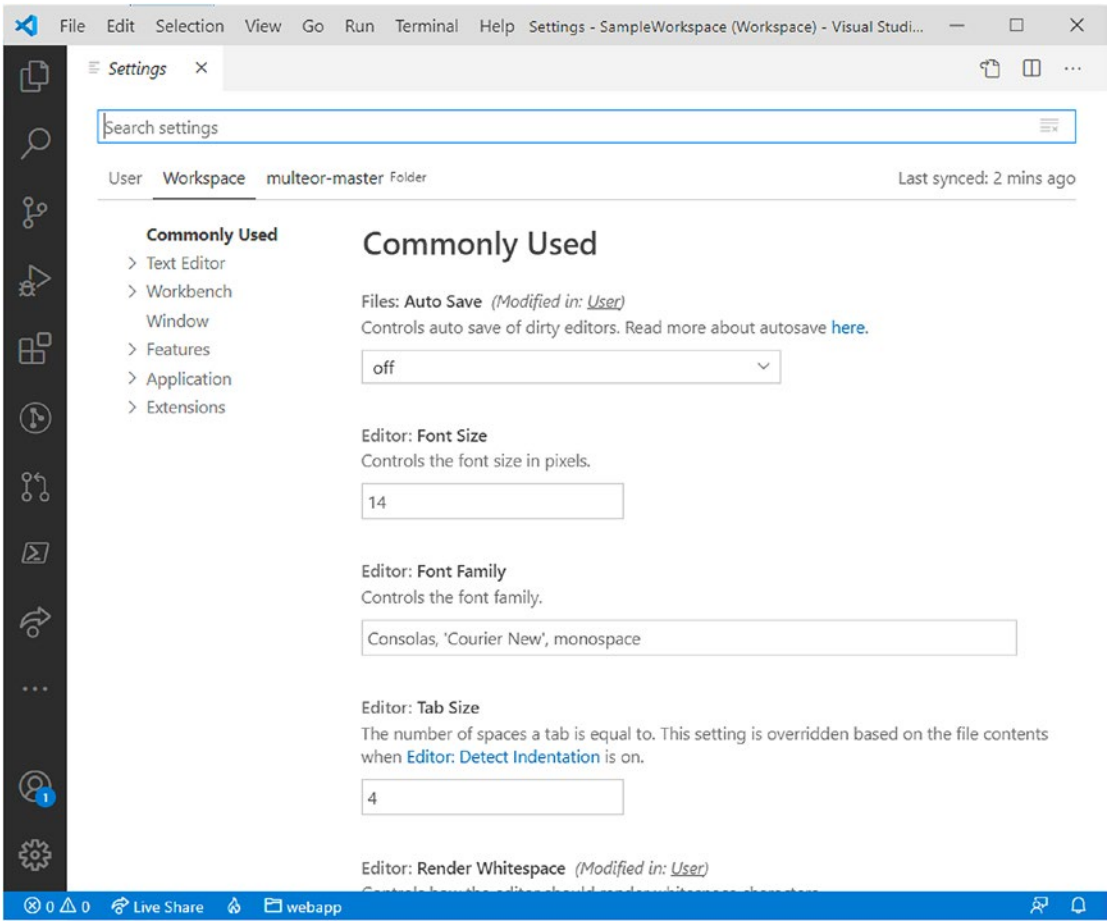

Figure 5-9. *Customizing workspace settings*

You customize workspace and folders settings exactly as you do with user settings, so you have not only a second view in the settings editor but also two other JSON files where you can specify your preferences. More specifically, workspace settings are stored in the .code-workspace file (you can see this in the Explorer), while folder settings are stored in the settings.json file. The .code-workspace file is saved under the workspace folder, while settings.json is saved under the .vscode subfolder that Visual Studio Code creates inside the opened folder, restricting settings availability to the current folder only.

Customizing Keyboard Shortcuts

Visual Studio Code includes a huge number of keyboard shortcuts that you can override with custom values. This is particularly useful if you are used to working with other development tools and you want to have the same keyboard shortcuts in Visual Studio Code.

Note In the next chapter you will learn how to download ready-to-use keyboard shortcuts that will save you a lot of time, but it's first important for you to know how they actually work.

Like user and workspace settings, keyboard shortcuts 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 Esc key, which targets a number of actions depending on what you are working with, such as the code editor or a tool window. To identify the proper action, keyboard shortcut settings support the **when** element, which specifies the proper action based on the context. You can quickly get the list of current keyboard shortcuts by selecting **File ➤ Preferences ➤ Keyboard Shortcuts**. At this point, Visual Studio Code displays a nicely formatted list of commands and shortcuts, as you can see in Figure 5-10.

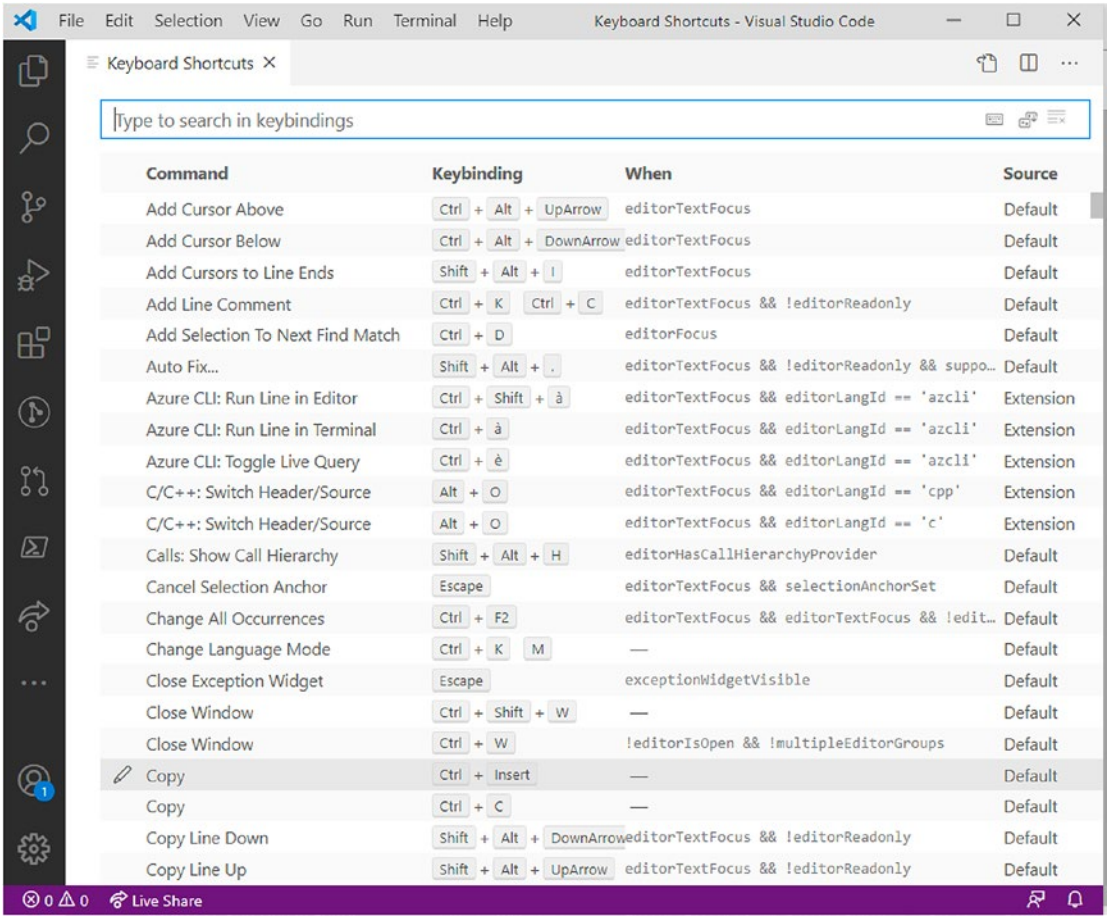

Figure 5-10. *The list of current keyboard shortcuts*

To customize keyboard shortcuts, all you need to do is click the **Open Keyboard Shortcuts** button, represented by a sheet icon with a plus symbol overlayed, located at the top-right corner of the window. This opens the keybindings.json file, where you can override default shortcuts with custom ones (see Figure 5-11).

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

You can quickly add a custom keyboard shortcut by clicking the **Define Keybinding** button or by using the shortcut suggested in the button text (which varies depending on your operating system). When you do this, a pop-up box appears and asks you to specify the keyboard shortcut, as shown in Figure 5-11.

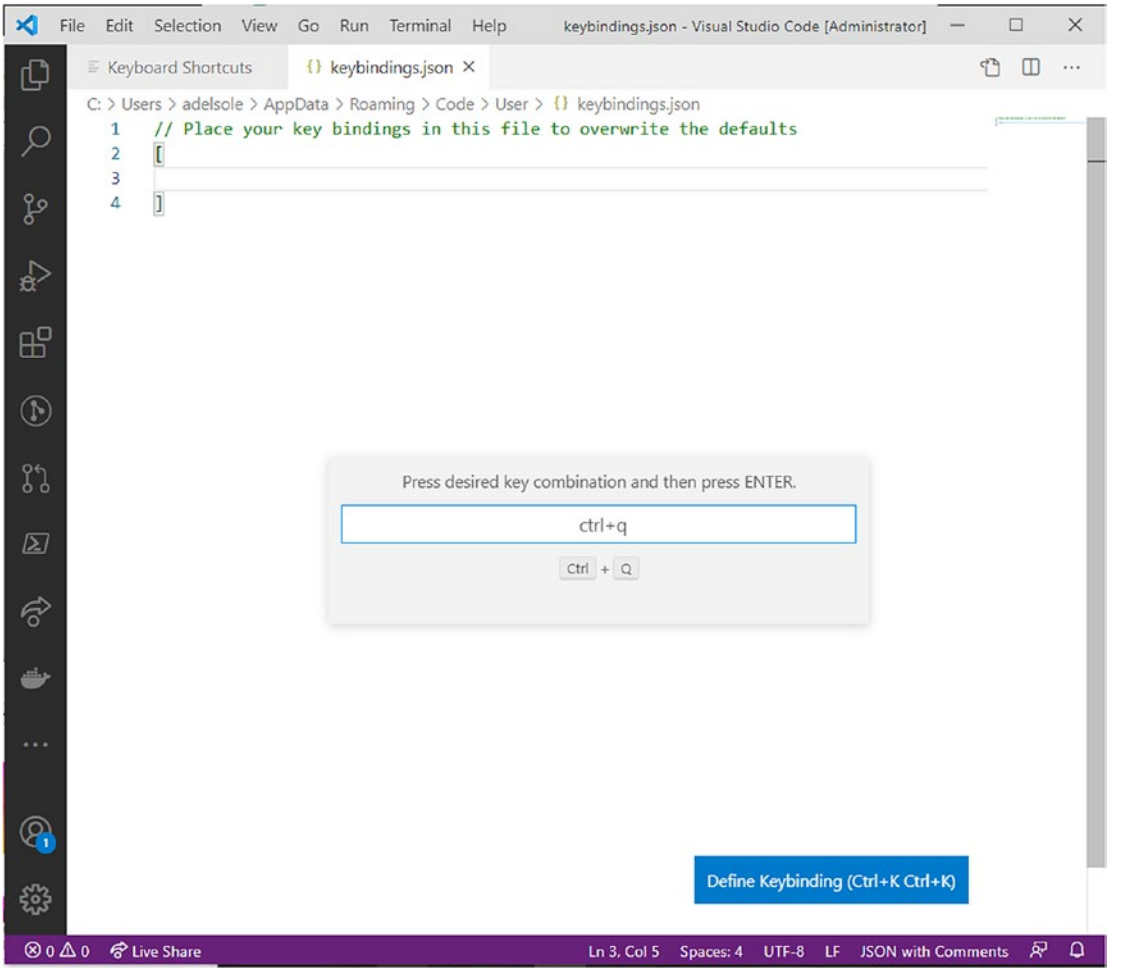

Figure 5-11. *Adding a keyboard shortcut*

When you press Enter, the JSON markup for the new keyboard shortcut is added, as shown in Figure 5-12.

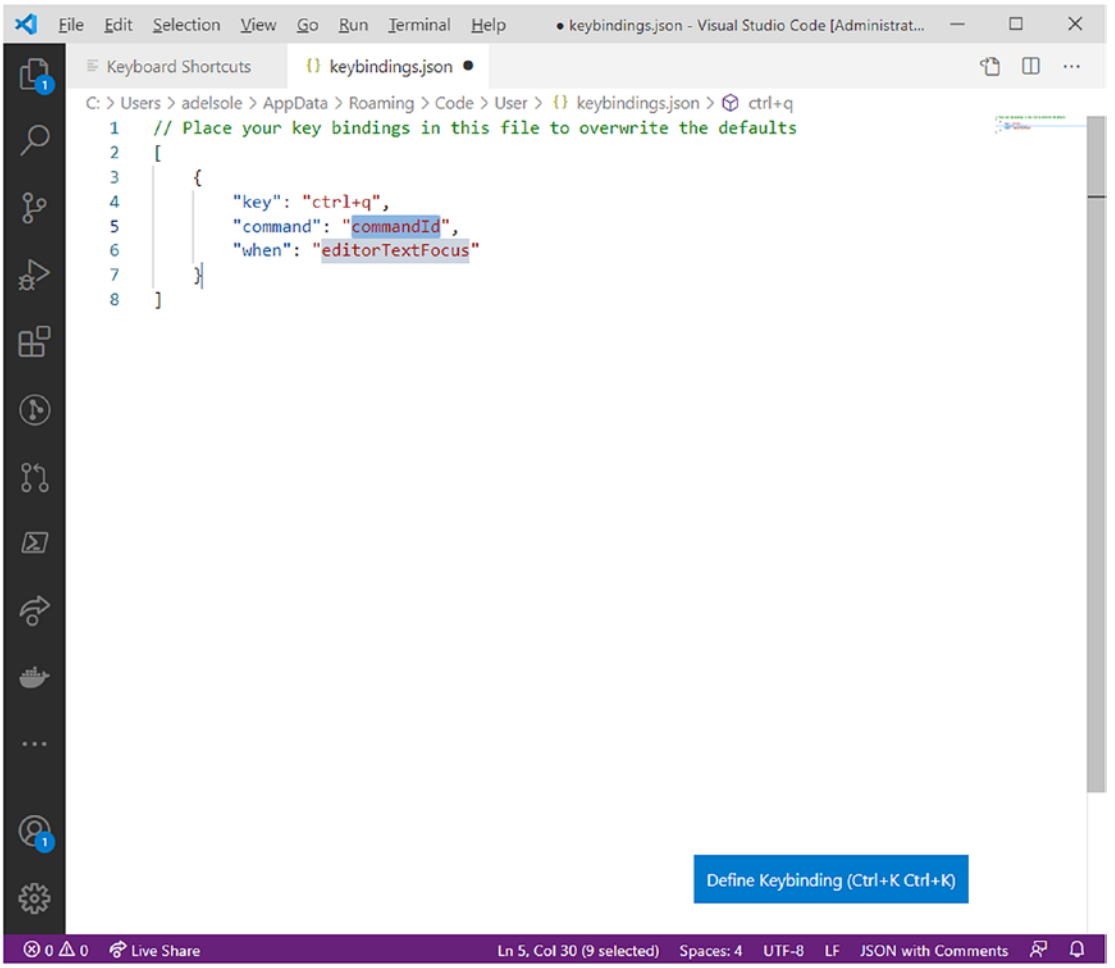

Figure 5-12. *Editing the new keyboard shortcut*

You need to edit the `command` and `when` elements with the command you want to map and for which scenario. Additionally, when editing keybindings.json manually, you need to supply the markup for both the old shortcut and the new one. For example, suppose you want to replace the Alt+O shortcut for the C/C++ extension (Switch: Header/Source) with Shift+Alt+O. The markup you would need to write looks like the following:

```
{
    "key": "shift+alt+o",
    "command": "C_Cpp.SwitchHeaderSource",
    "when": "editorTextFocus && editorLangId == 'cpp'"
},
```

```
{
    "key": "alt+o",
    "command": "-C_Cpp.SwitchHeaderSource",
    "when": "editorTextFocus && editorLangId == 'cpp'"
}
```

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 enables you to make several customizations that will help you feel at home, especially if you are used to working 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. Extensibility is especially beneficial in the area of languages, because Visual Studio Code enables you to extend 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 the possibilities 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 website that contains extensions for the most popular Microsoft development tools and services, such as Visual Studio, Visual Studio Code, and Azure DevOps. It is available at `https://marketplace.visualstudio.com`, and you need to click the Visual Studio Code tab to see a list of extensions for Visual Studio Code. Figure 61 shows the Marketplace for Visual Studio Code.

A. Del Sole, *Visual Studio Code Distilled*, https://doi.org/10.1007/978-1-4842-6901-5_6

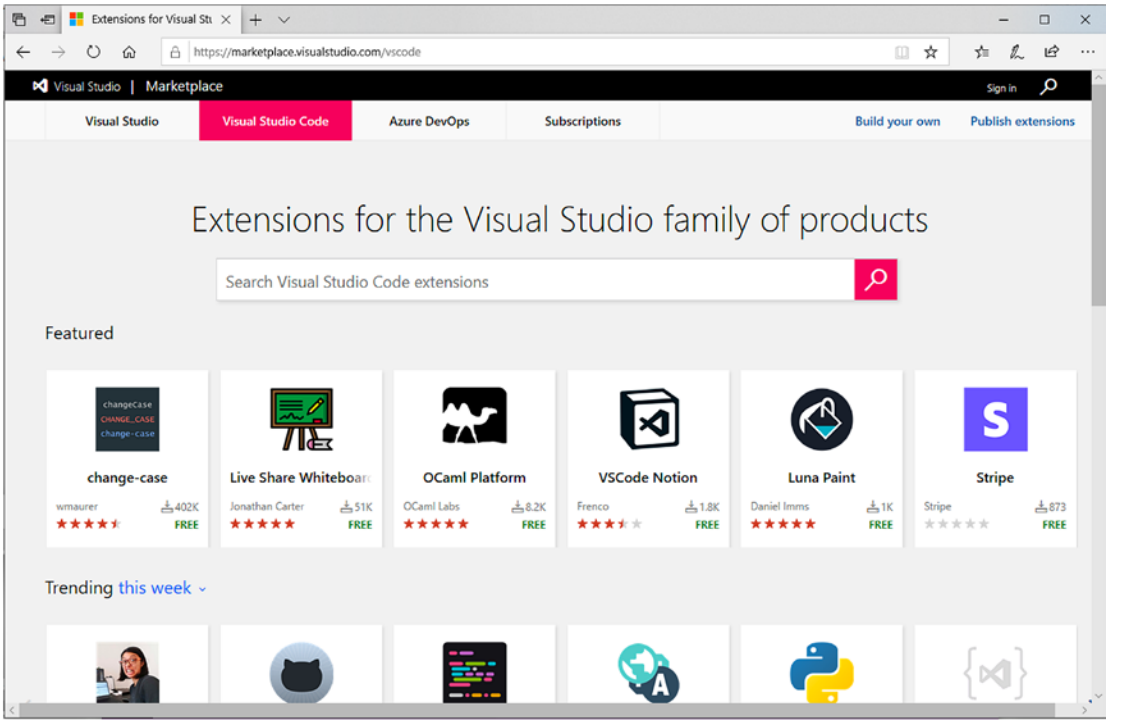

Figure 6-1. *The Visual Studio Marketplace*

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

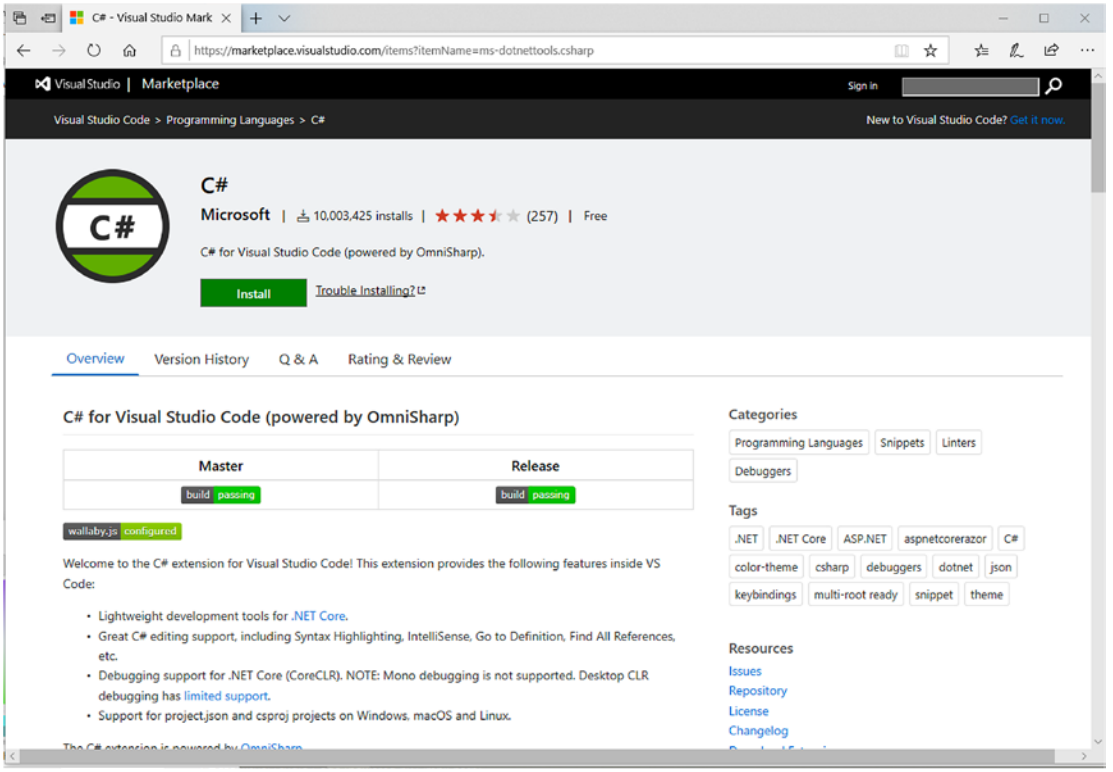

Figure 6-2. *Detail page for an extension*

An extension's page provides a detailed description and guidance about using the extension, often providing links to additional documentation, resources, and the source code (if open source). I strongly recommend that you 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 support only for a new syntax or also for IntelliSense, code snippets, and debugging.

If you click the **Install** button, your browser will ask your confirmation to open the download link with Visual Studio Code. When this starts, the extension will automatically be installed. 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 2019 on Windows or 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 and then click a specific extension to get the details, as shown in Figure 6-3.

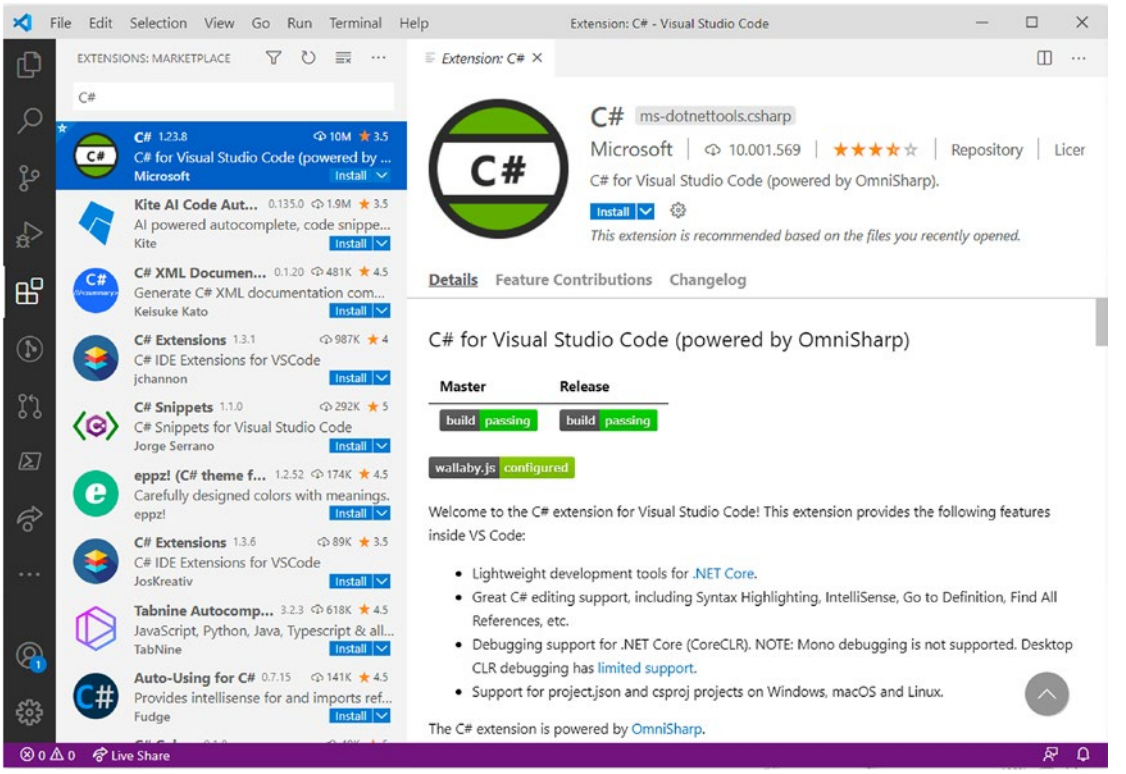

Figure 6-3. *Installing extensions from within Visual Studio Code*

You can click the **Install** button when ready. You 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. 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. These include libraries to support .NET Core debugging and tools to improve the editing experience via IntelliSense and live static analysis. Also, newly downloaded extensions might need some initial configuration. In this case, a pop-up box 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**, under the list of installed extensions.

The list of recommended extensions varies on your activity and might be empty the first time you work with Visual Studio Code. As one option, 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 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, VS 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.

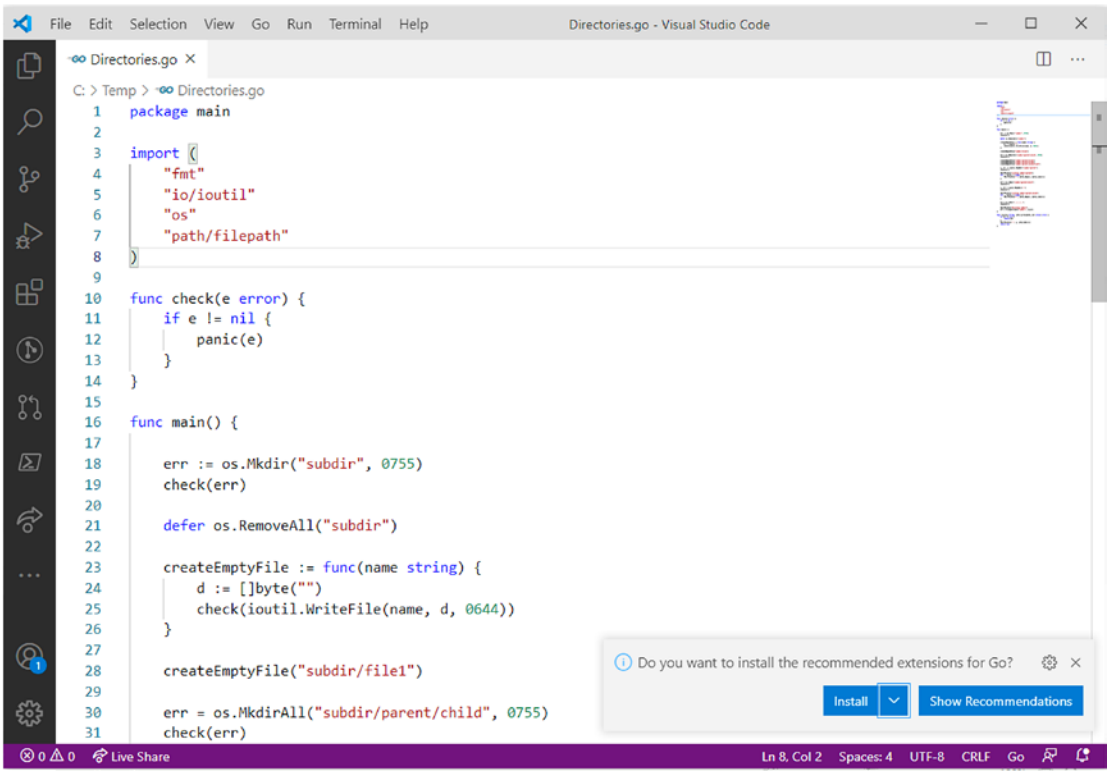

Figure 6-4. *Extension recommendations based on the current file*

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 open 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 this set of useful extensions.

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
SQL Server (mssql)	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 Microsoft 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
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
Live Share	Extension for collaborative, real-time development that shares your instance of VS Code with other developers	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 pop-up menu that contains commands for disabling or uninstalling an extension.

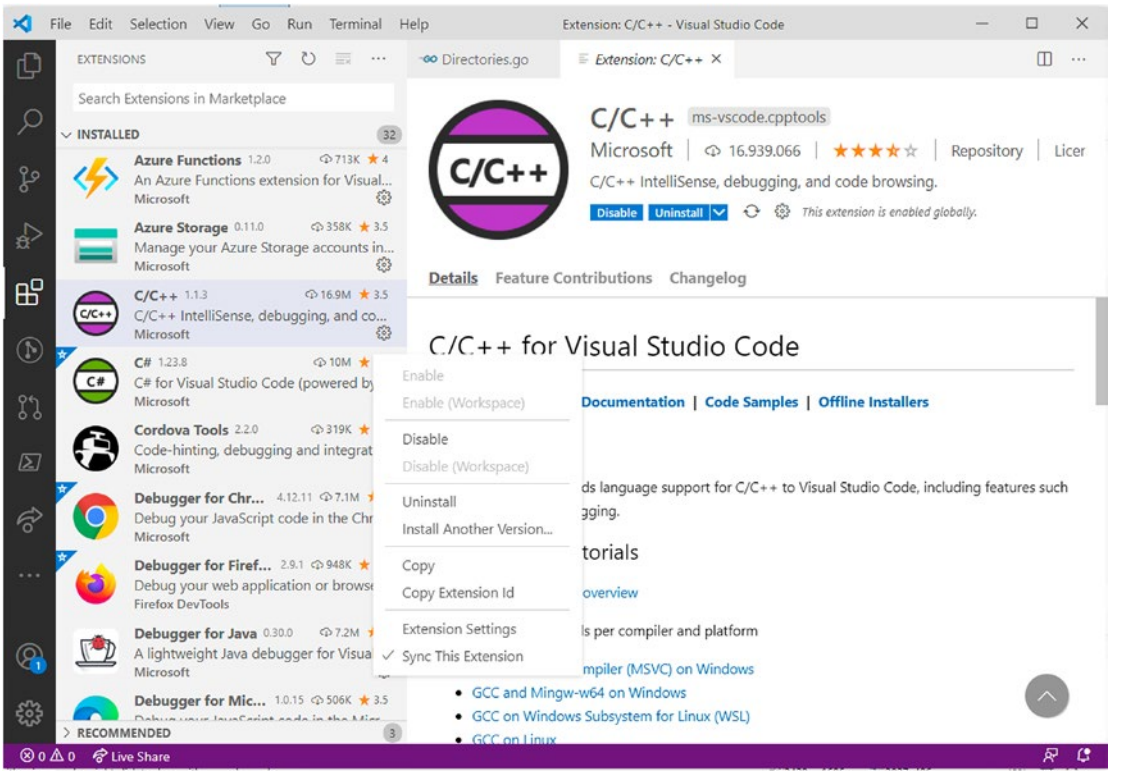

Figure 6-5. *Shortcuts for extension management*

You can also click an extension name, and the detail page will show the **Disable** and **Uninstall** buttons. Notice that when you disable or uninstall an extension, in most cases 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 and selecting the **Views** submenu. You then can choose among different options, such as viewing popular extensions, checking for 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 (which is based on the list of extensions installed on my machine and likely differs from yours).

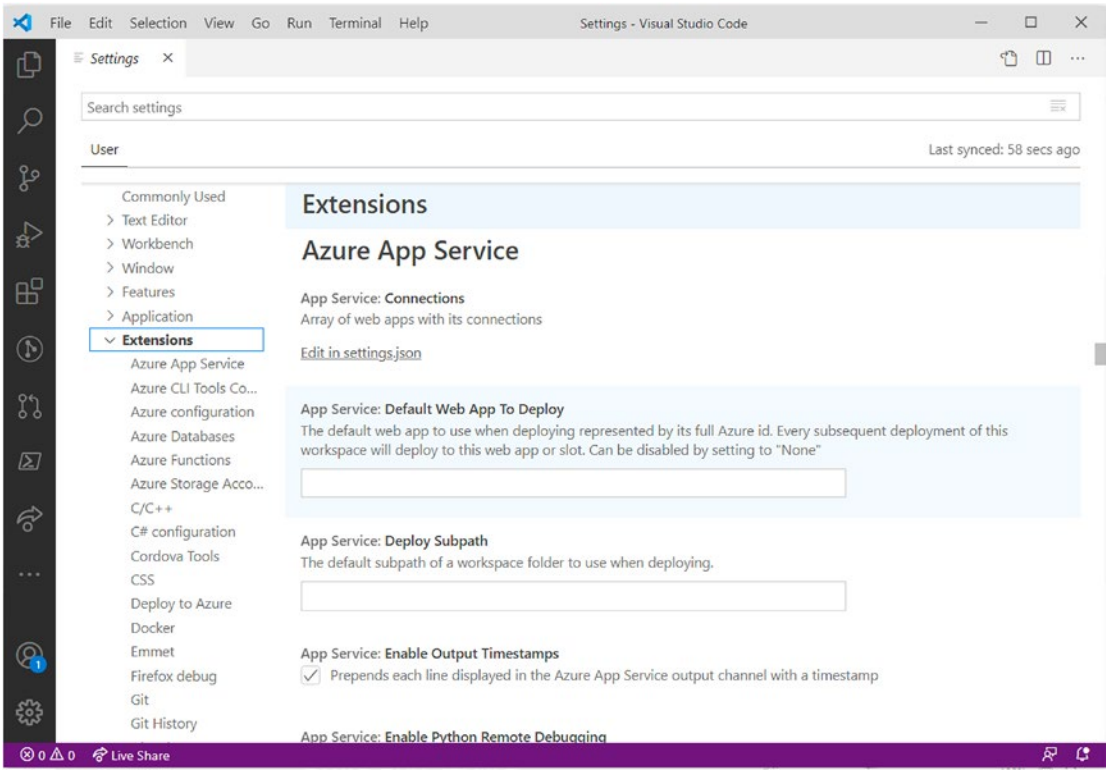

Figure 6-6. *Customizing options about extension management*

There are detailed comments that explain what each option is about. Each extension allows for customizing its own behavior in the user settings and edits can also be done in the well-known settings.json file. 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.

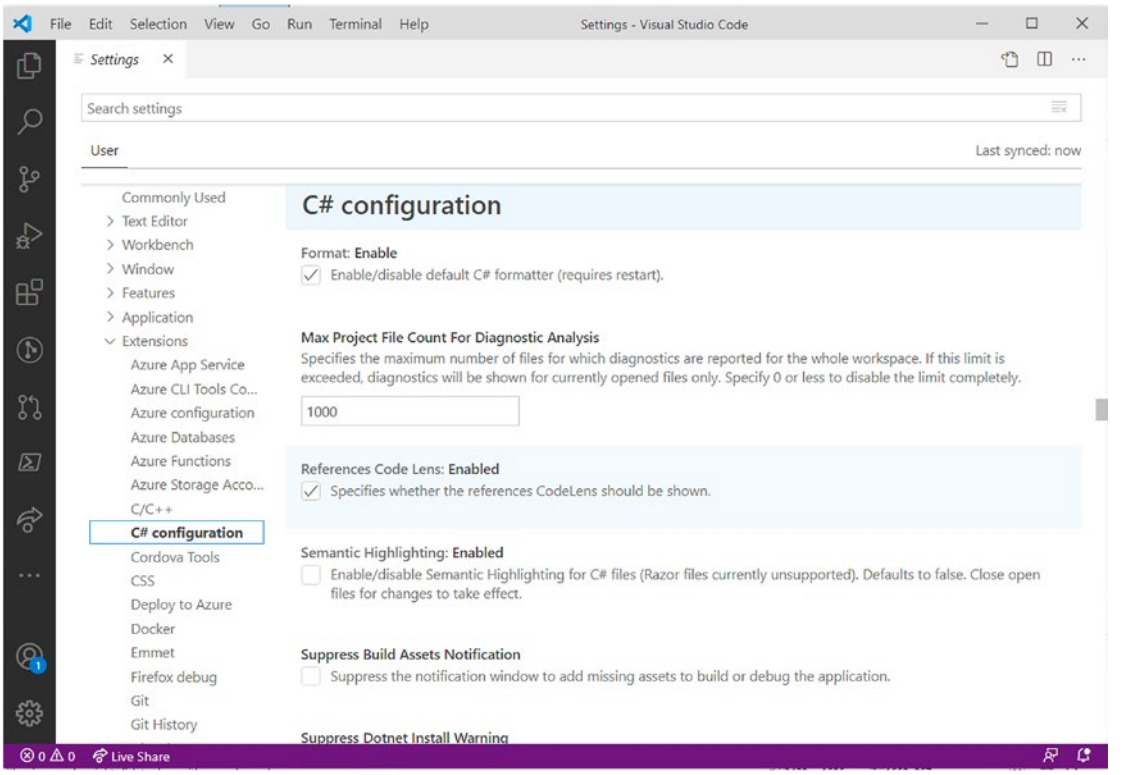

Figure 6-7. *Customizing extension options*

If you want to instead edit extension settings in the settings.json file, IntelliSense will simplify your work by showing setting names and a tooltip with the setting description when you scroll the list. Figure 6-8 shows an example where IntelliSense is showing some settings for the C# extension, identified with the `csharp` literal.

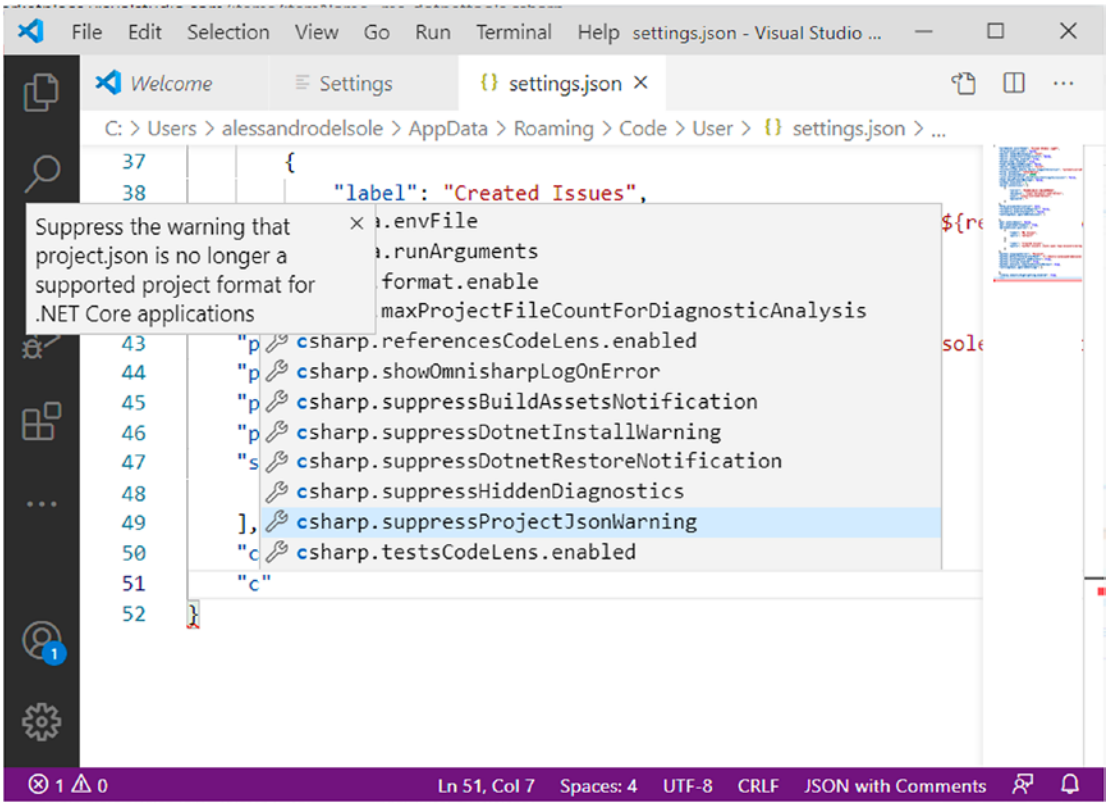

Figure 6-8. *Customizing extension options in settings.json*

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 Visual Studio 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 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/api), which provides examples and guidance for many scenarios.

Summary

Extensibility is a key feature in Visual Studio Code, because it allows you to add 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, such as when you open a file written in a language for which there is no built-in support. Visual Studio Code makes also makes managin extensions simple, with shortcuts to disable and uninstall extensions and the capability to configure 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 whether you are part of a development team, are involved in open source projects, or 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 that is based on Git and can be extended to other providers.

This chapter describes not only 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 on the job to better review your code and to push your work to Azure DevOps. Notice that the source control and version control terms are used interchangeably.

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 Azure DevOps, and provides an integrated way to manage your code commits.

Note 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`. 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`).

A. Del Sole, *Visual Studio Code Distilled*, https://doi.org/10.1007/978-1-4842-6901-5_7

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, but to follow along with the examples in this chapter, you'll need Greeter. 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 to find third-party extensions that target other source control engines. Figure 7-1 shows an example of selecting an extension that adds support for the Subversion engine (`https://subversion.apache.org`).

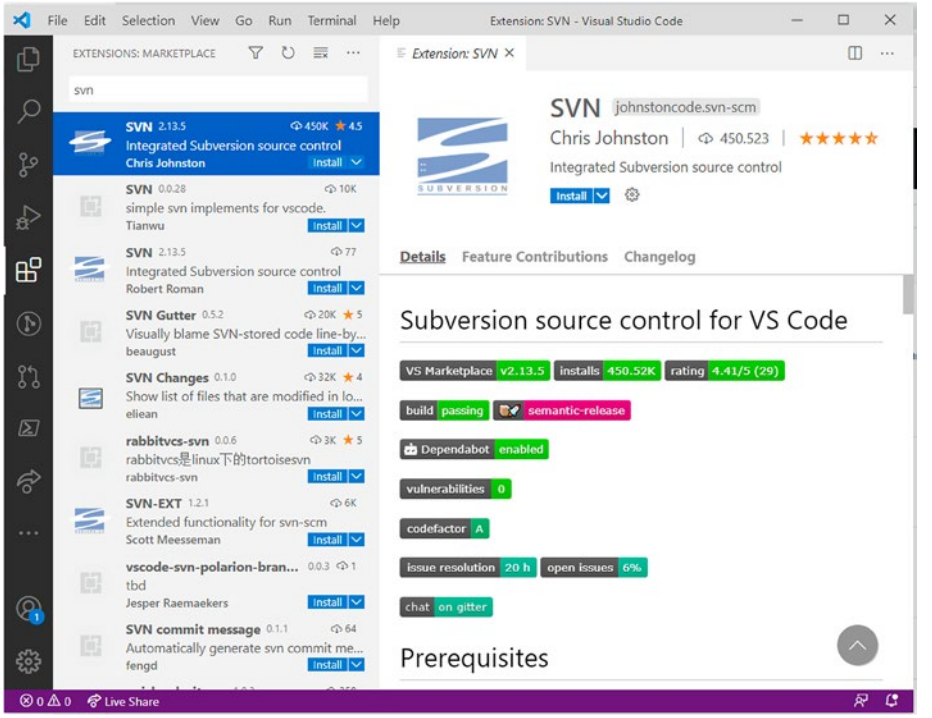

Figure 7-1. *Installing additional source control providers*

Because VS Code provides in-the-box support only for Git, other source control providers are not 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 supports both a local repository 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*. Although this term is not used in this book, you will encounter it often, especially when searching for information about open source projects.

Initializing a Local Git Repository

As a starting point for the following examples, open the Greeter project downloaded previously. The first thing you need to do is create 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.

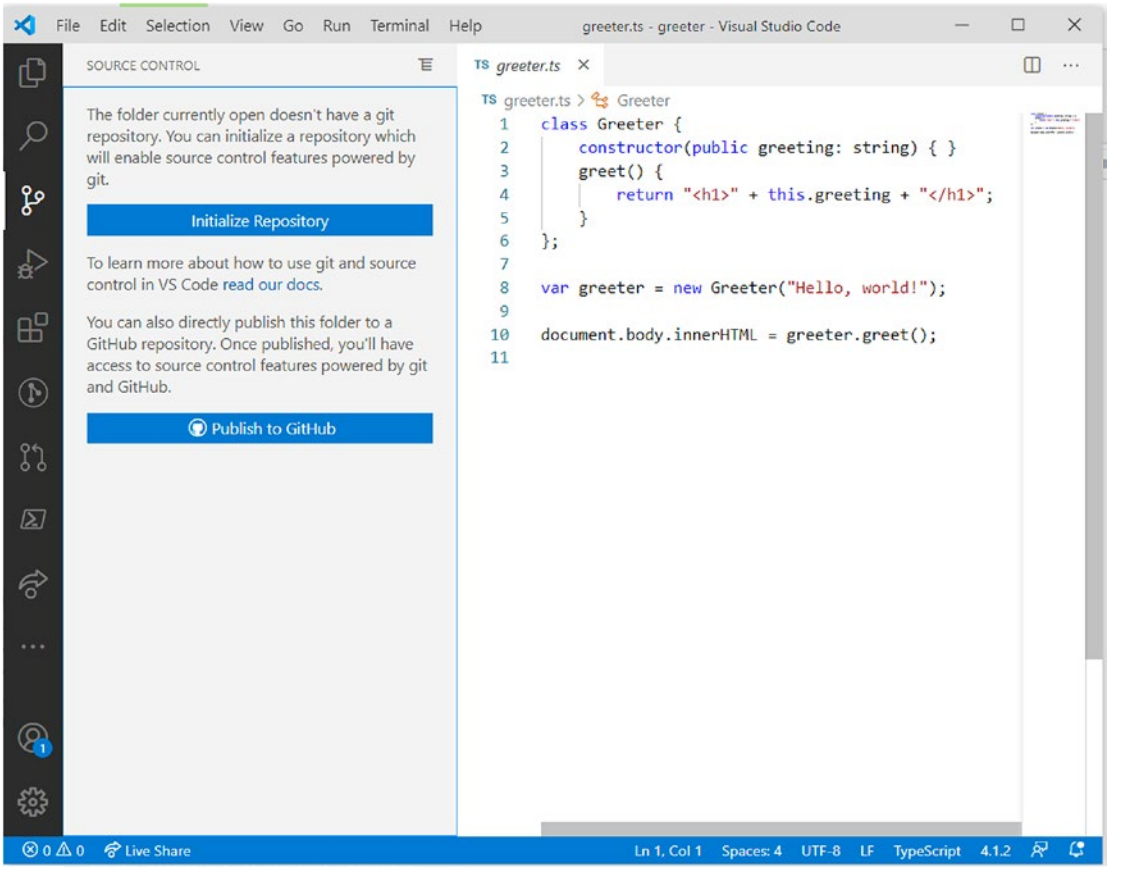

Figure 7-2. *Ready to initialize a local Git repository*

Clicking the **Publish to GitHub** button would allow you to initialize a local repository and publish to GitHub at the same time, but because it is important to understand how the flow works and how to properly authorize VS Code to GitHub, the steps here are split into creating a local repository first and then publishing to the remote one. Click the **Initialize Repository** button at the top (see Figure 7-2). Visual Studio Code will initialize the local repository and show the list of files that now are under version control but not committed yet (see Figure 7-3).

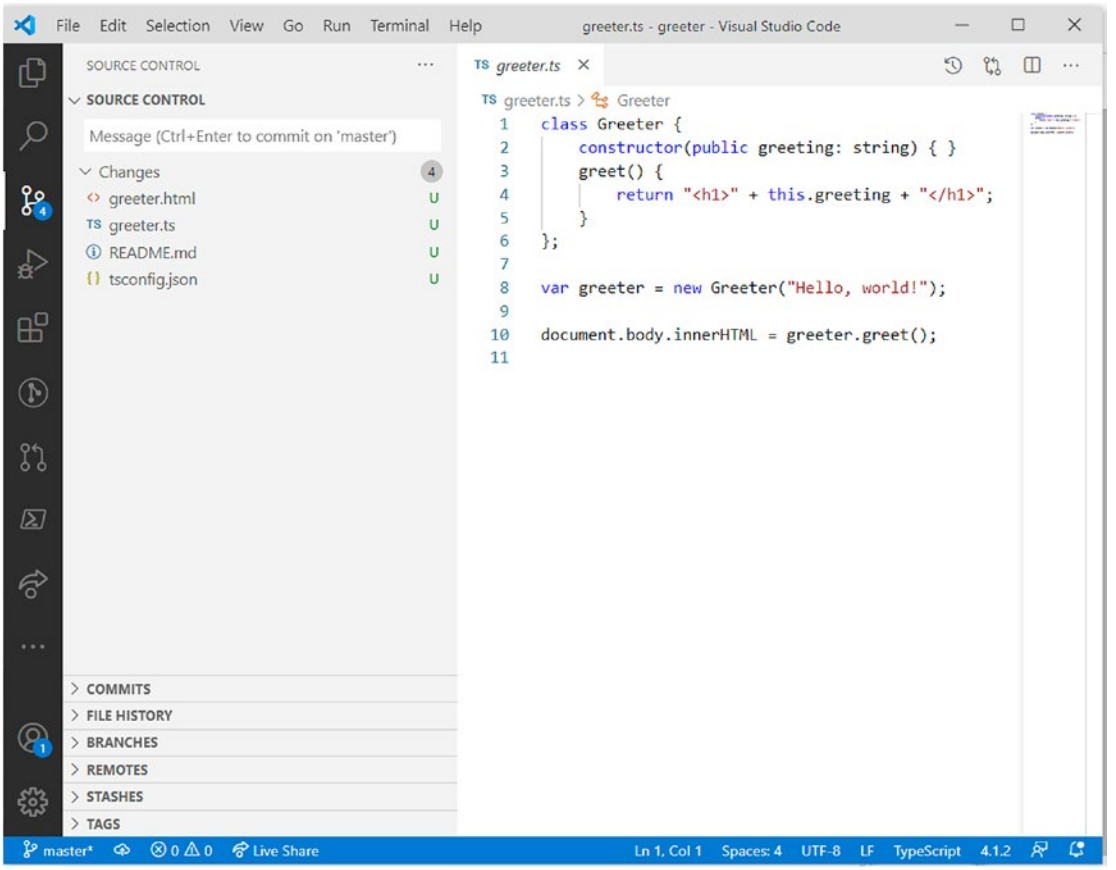

Figure 7-3. *Files are under version control but not committed yet*

Notice how the Git icon shows the number of pending changes. This is an important indicator that you will always see anytime you have pending, uncommitted changes. Write a commit description and then press Ctrl+Enter. You will see a warning message saying that there are no staged files at the moment, and you will be offered to stage and commit directly all files. Staging will be discussed in the next section, so for now click **Yes**. 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 VS Code. The next section explains how to accomplish this.

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 Azure DevOps. This section shows you how to create a remote repository on GitHub. I chose GitHub not only because of the popularity of the platform but also because Visual Studio Code includes a built-in extension called GitHub that is expressly designed to simplify the workflow against GitHub itself. This requires you to have an existing GitHub account or to create one for free at `https://github.com/join`. Visual Studio Code makes it very easy to publish repositories to GitHub with a single mouse click, but VS Code first needs to be authorized by the GitHub engine, so there are some preliminary steps to do just once.

Note GitHub no longer supports Microsoft browsers such as Edge and Internet Explorer. Though you can open the website with both, some actions will not be available. I recommend opening GitHub with a browser such as Chrome or Firefox.

On the Status Bar, click the **Publish to GitHub** button, identified by an icon representing a cloud with an arrow and located to the right of the **master** branch name. Figure 7-4 shows this button inside the green box.

Figure 7-4. *The Publish to GitHub button*

An alert will inform you that VS Code wants to access GitHub and, after you click OK to accept, it will open the default browser pointing to a GitHub page where it will be possible to authorize VS Code. Click **Authorize**, then enter your GitHub credentials and accept the access requirements that the extension requires. Next, GitHub generates an authorization token that is specific for Visual Studio Code and that looks like the one generated on my machine, visible in Figure 7-5.

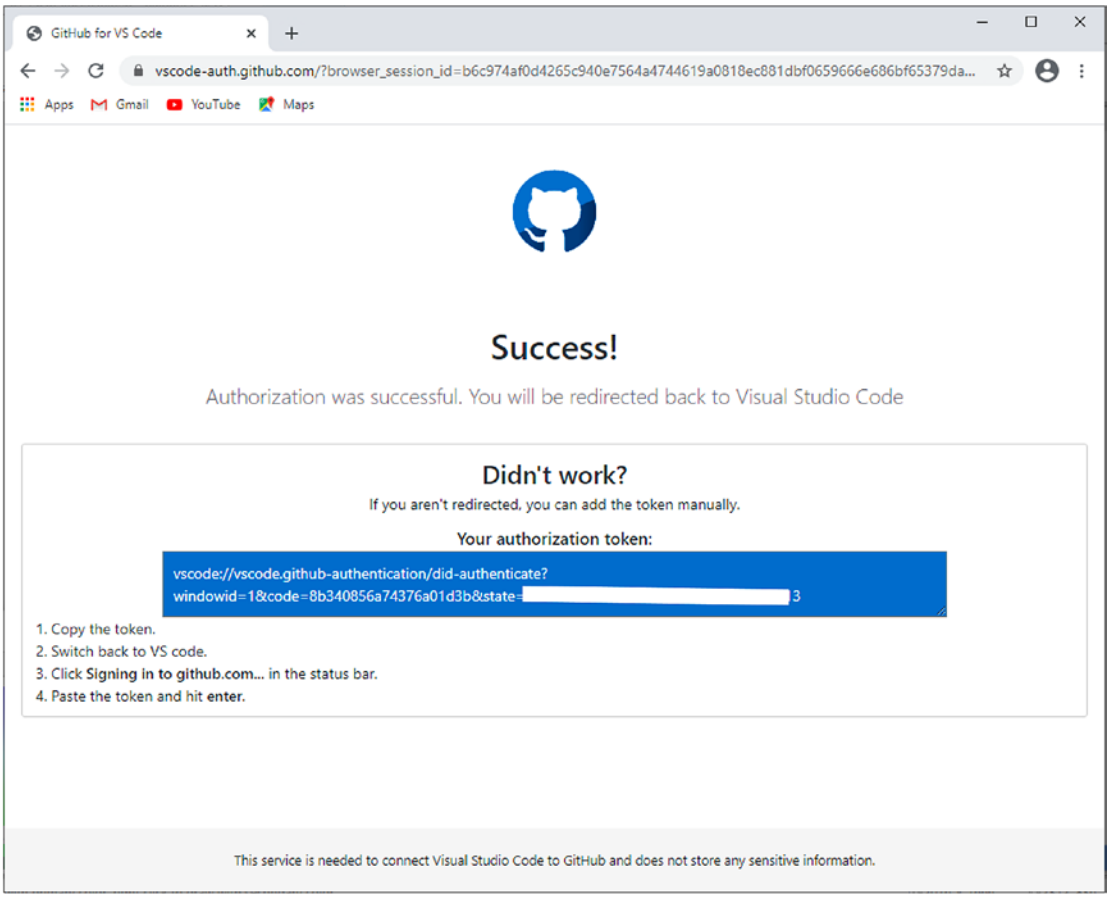

Figure 7-5. *An authorization token generated for Visual Studio Code*

Your browser will ask your permission to open an URL with Visual Studio Code. Allow this, so that Visual Studio Code will be able to complete the authentication process automatically. (This is an improvement over previous versions, which required entering the token manually.) At this point VS Code is enabled to access GitHub. As I mentioned previously, the steps required to authorize Visual Studio Code need to be done only once. Note that you will not get confirmation that the authorization has completed…it is a silent process.

At this point you need to click again the **Publish to GitHub** button on the Status Bar. VS Code shows a text box containing the repository name; by default, this is based on the current folder name, but you can write a different name. It also provides two options to publish the repository to GitHub based on the folder name, as you can see in Figure 7-6; one option is to publish to a private repository, and the other option is to publish to a public repository.

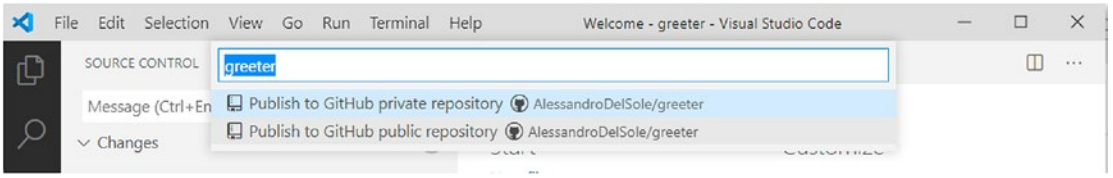

Figure 7-6. *Available options to publish the repository remotely*

For the current example, the public option will be used, but you are free to choose whichever option you prefer. When publishing is completed, you will get a confirmation message and an option to open the GitHub repository in the browser.

Note If you work with platforms different from GitHub, you can easily associate a remote repository by clicking the **…** button located in the upper-right corner of the Source Control bar and then selecting **Remote ➤ Add Remote**. This is explained in practice in the section "Working with Azure DevOps and Team Foundation Server" toward the end of this chapter.

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 VS Code, handling file changes is very straightforward. In Figure 7-7 you can see how the number of pending 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). Note that these markers are also visible in the Explorer bar.

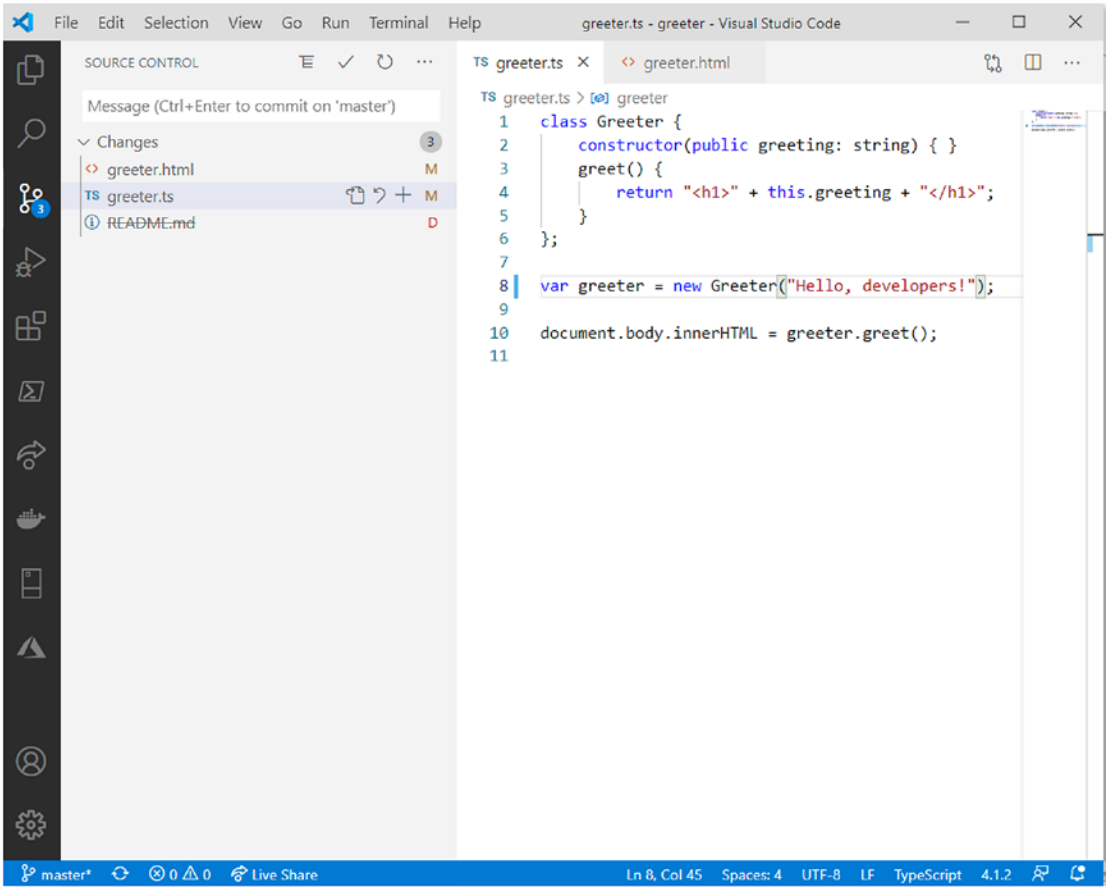

Figure 7-7. *Identifying the number of pending changes*

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 7-8 shows an example.

The left side shows the old version and the right side shows the new one. The line highlighted in red represents code that has been removed, whereas the line highlighted in green represents new code. Specific changes inside the lines of code are represented with darker shades of red and green, as you can see for the words `world` and `developers` in Figure 7-8. This is a very important tool when working with any version control engine.

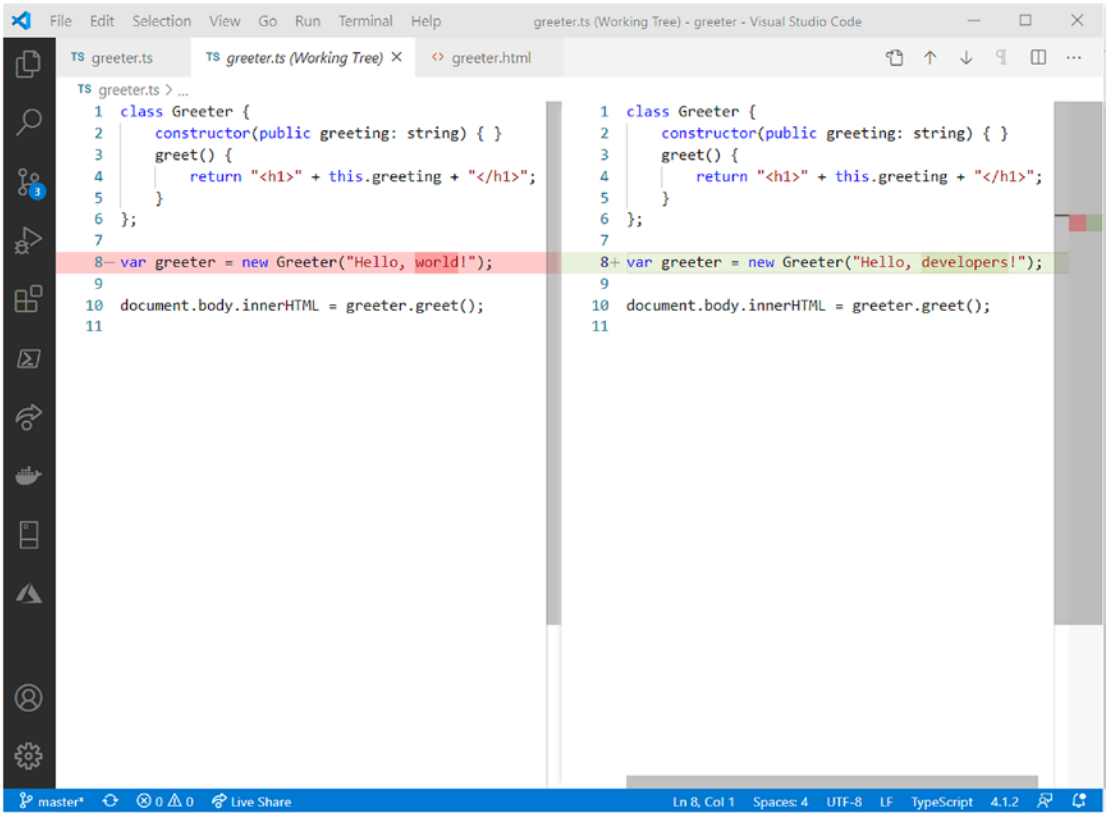

Figure 7-8. *Comparing file versions with the Diff tool*

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 selecting **Stage All Changes** or clicking the plus icon on the bar. Visual Studio Code organizes staged files into a logical container, as you can see in Figure 7-9. Similarly, you can unstage files by clicking the – symbol.

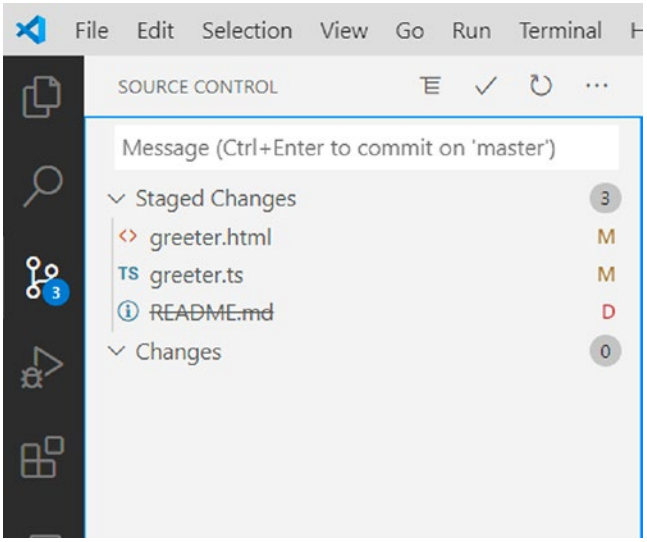

Figure 7-9. *The view of staged and unstaged changes*

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 gets committed to the repository.

Managing Commits

The **...** button provides access to additional actions, such as **Commit**, **Sync**, **Pull**, **Stash,** and **Pull (Rebase)**. Figure 7-10 shows the full list of builtin Git synchronization commands available in VS Code. Notice that some of them are grouped into submenus, such as **Pull, Push** that you can see in Figure 7-10.

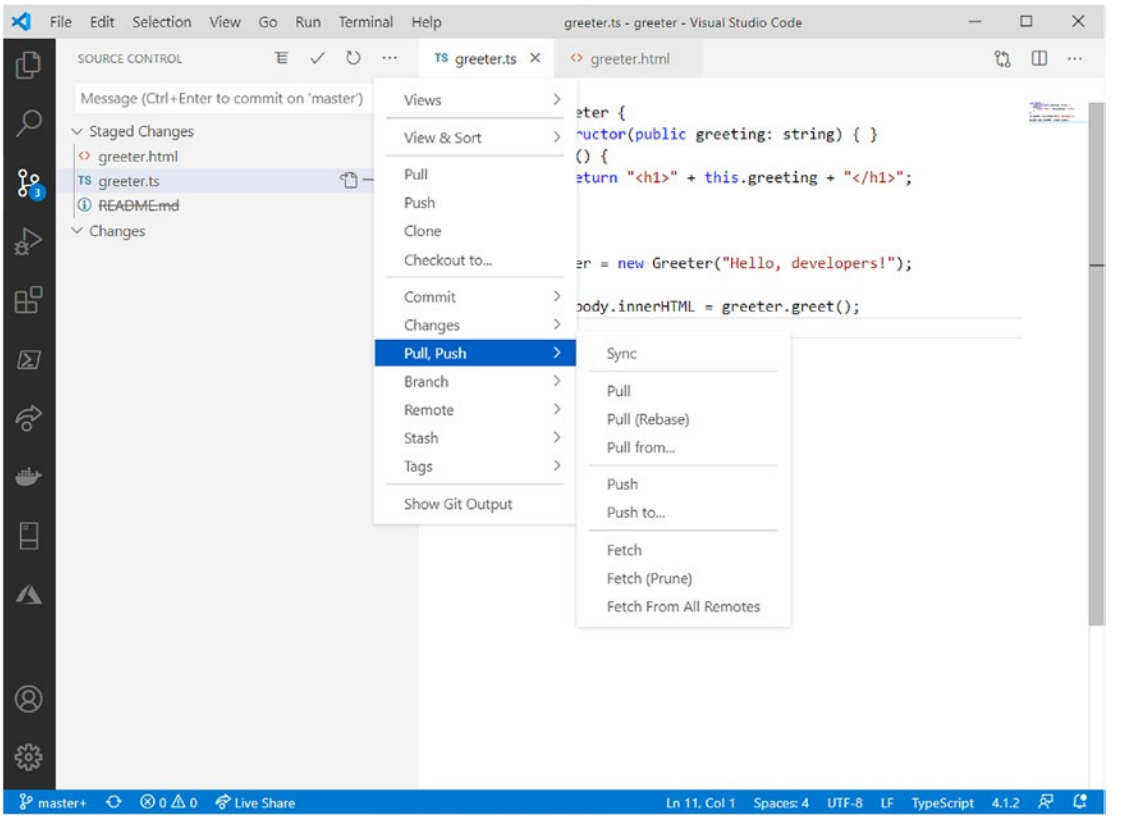

Figure 7-10. *Shortcuts to commit and synchronize changes*

When you are satisfied with your work on the source code, you can select the **Commit ➤ Commit All** command to commit your changes. Remember that this action commits files to the local repository. Also, before you commit, you might want to check staged and nonstaged changes so that the code is committed without missing any files. You have to use the **Push** command 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 **Commit ➤ Undo Last Commit** command. **Pull** and **Pull (Rebase)**, both in the **Pull, Push** submenu, allow you to merge a branch into another branch; **Pull** 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 local branch. The **Sync** command in the same submenu 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, under the **Stash** submenu, you can retake the latest version of your unfinished work or 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 7-11 shows an example.

Figure 7-11. *Messages from the Git command line are shown in the Output panel*

You will need to select **Git** from the drop-down menu 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 pop-up menu shown in Figure 7-10.

Working with the Git Command-Line Interface

The Command Palette has support for specific Git commands that you can type as if you were in a command-line terminal. Figure 7-12 shows a partial list of available Git commands, displayed by typing `Git` in the Command Palette. The full list of commands is quite long and cannot be totally included in Figure 7-12, but you can type `Git` on your own computer and scroll the list to see all available commands.

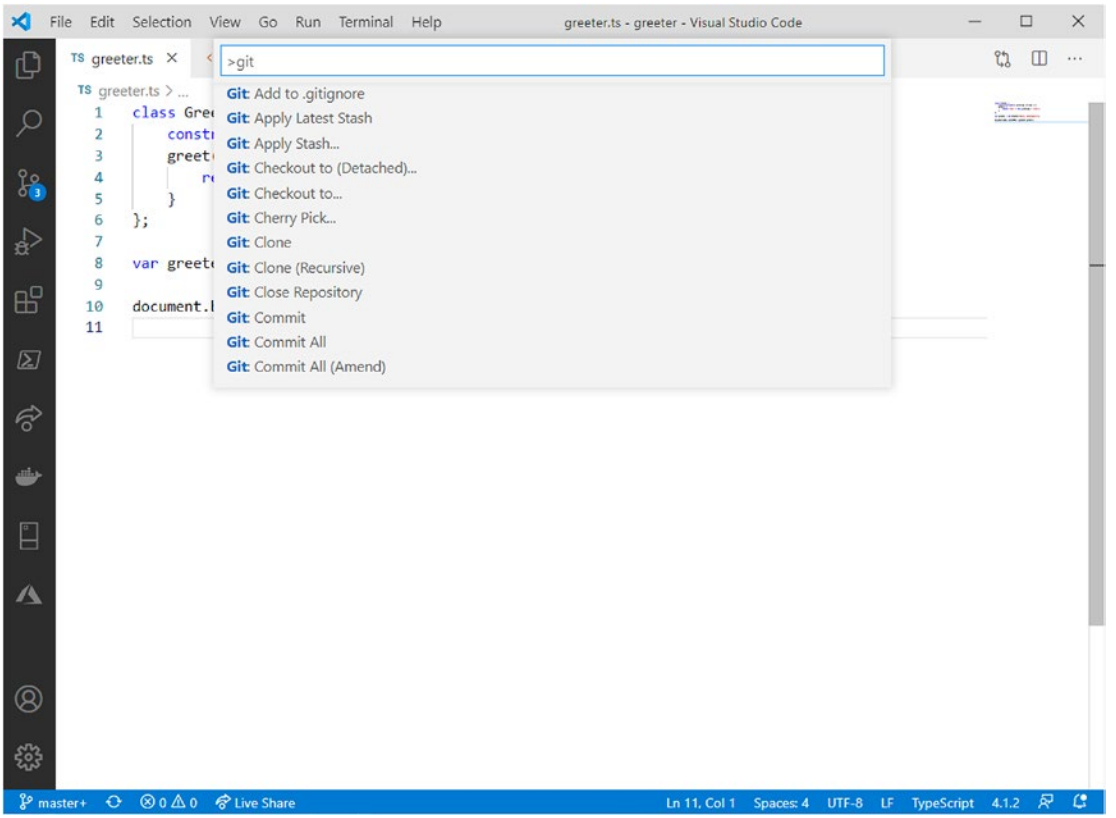

Figure 7-12. *Supported Git commands in the Command Palette*

It is worth mentioning that the list of commands is also grouped by most recently used and all commands.

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 life cycle, 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**.

Note There have been recent changes in GitHub, so if you first create a remote repository on this platform directly, the main branch is no longer called **master**, but instead is called **main**. This change is specific to GitHub, so if you create a Git repository either locally or on other platforms, you still get the **master** branch.

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 is to create a branch from the Command Palette by typing `Git branch`, selecting the **Git: Create Branch** option, and specifying a new branch name, such as **development**. This creates a new branch locally, based on **master**. The second option is to click the current branch name in the Status Bar (**master** in this case) and then click the **Create new branch** command (see Figure 7-13). Enter the new branch name, and then press Enter.

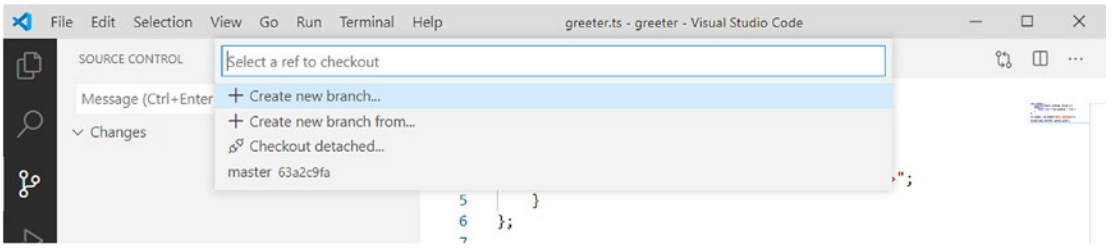

Figure 7-13. *Creating a branch*

In addition, you can use the **Create new branch from** command to create a new branch from a branch that is not the active one. 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 7-14).

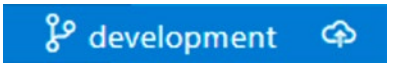

Figure 7-14. *The new branch is set as active and ready to be published*

Switching to a Different Branch

Switching to a different branch is very easy. Simply click the name of the active branch in the Status Bar, and VS Code displays the list of branches, as shown in Figure 7-15. If the repository already has a remote branch, it will also be visible in the list.

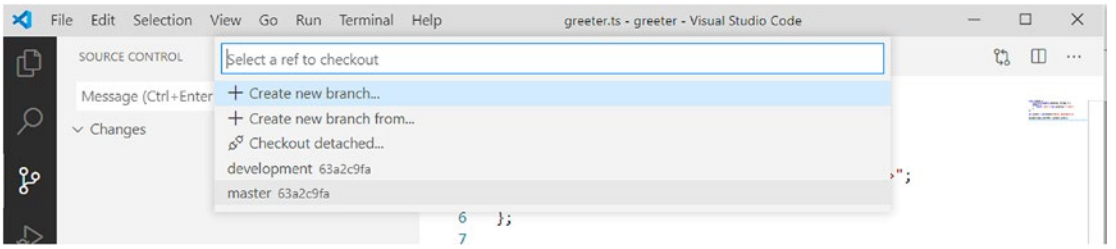

Figure 7-15. *Selecting a different branch*

Click the desired branch, and VS Code checks it out and sets 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 (which normally happens via pull requests, described later in this chapter). You can merge from a branch into another one via the Command Palette, using the **Git: Merge Branch** command. VS Code shows the list of branches, and you need to select the branch you want to merge from into the current branch (see Figure 7-16).

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.

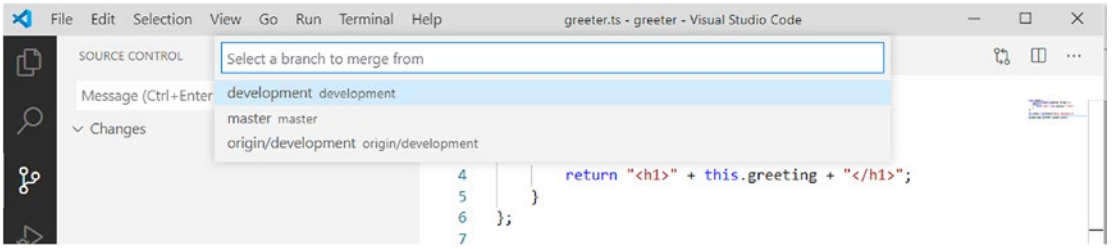

Figure 7-16. *Merging from a branch*

In the example, some changes were made and pushed to the development branch, then the master branch has been selected as the active one and changes from development will be merged into master.

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

Resolving Merge Conflicts

When you merge branches in which the same code files were modified, Visual Studio Code leverages the Git tooling to combine the different edits into one code inside the target files. However, sometimes VS Code is not able to automatically combine the edits, in which case it raises a *merge conflict*. If this happens, VS Code shows an editor where it highlights the code on which a conflict exists, displaying the current version and the incoming version with different colors, as you can see in Figure 7-17, which shows an example of one conflict due to edits on the same line of code in different branches.

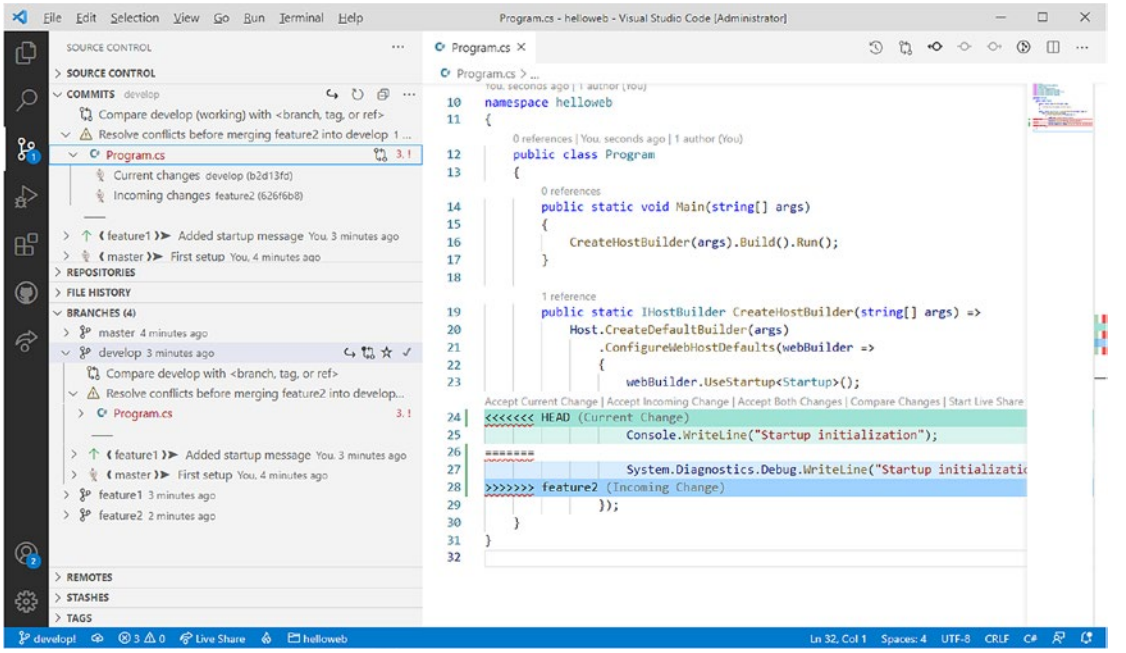

Figure 7-17. *Resolving merge conflicts*

Conflicts are also visible in the COMMITS panel of the Side Bar, and must be resolved before merging can be completed. As you can see in Figure 7-17, the code editor provides inline shortcuts to quickly resolve the conflict:

- *Accept Current Change*: Keeps the existing code and rejects the incoming change.
- *Accept Incoming Change*: Overwrites the existing code with the incoming edits.
- *Accept Both Changes*: Keeps both the existing and incoming code. Incoming code is appended to the existing code.
- *Compare Changes*: With several conflicts, allows for deciding which of the existing code or incoming code should be merged.
- *Start Live Share*: Only available with the Live Share extension installed, allows starting a live sharing session to ask for help from other developers.

What the right choice is only depends on your preference. Visual Studio Code gives you an integrated and user-friendly way to quickly solve merge conflicts without dealing with complex Git commands.

Hints About Rebasing Branches

Among the available commands for Git in Visual Studio Code, you will find one called Rebase. In Git, rebasing still allows you to include the changes made by a branch in another branch, but rebasing and merging accomplish this task differently.

More specifically, rebasing does not create overlaps between branches but rather appends code changes to the end of the target branch, which means that the history of the code is easier to understand, even if there is a need to frequently incorporate the commits of one branch into the other.

Rebasing therefore offers the possibility of accessing a more linear history, because, unlike merging, it allows you to not incorporate unnecessary commits into the target branch.

However, rebasing should be used with care. For example, if another team member is working on the same branch, it is preferable to avoid rebasing because this might lead to the duplication of the branch instead of merging changes.

Deleting Branches

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 7-16, 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. Also, remember that you can delete remote branches only if you created them.

Adding Power to the Git Tooling with Extensions

The integrated tools for Git cover all the needs that you, as a developer, may 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 enables you to view 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. After you have installed the extension, you can right-click a file inside the folder view of Explorer bar and select **Git: View File History**.

Figure 7-18 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. Local branches are highlighted in green and remote branches in red.

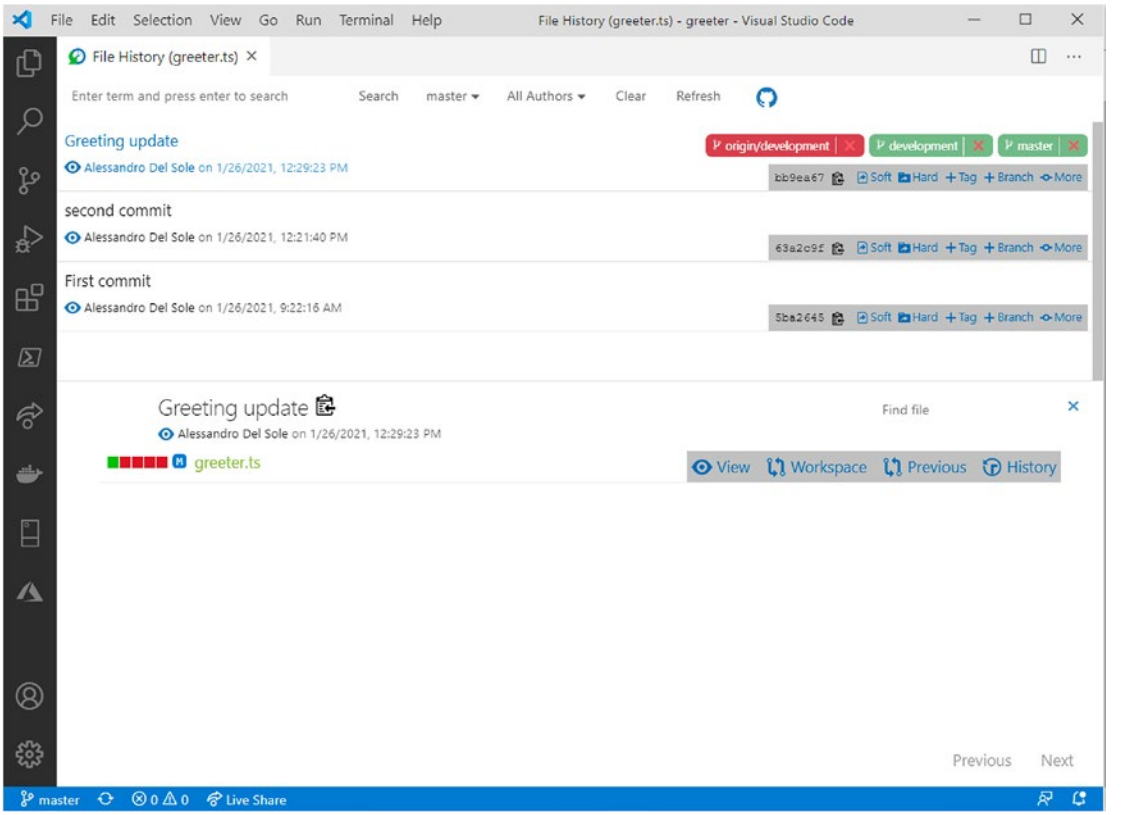

Figure 7-18. *Viewing the history of commits with Git History*

Note If the commit author has associated a picture to the Git credentials, Git History shows the picture near the author name.

If you click the **More** shortcut at the right of each commit, a menu appears showing a number of very useful commands that make it easier to work with commits (see Figure 7-19).

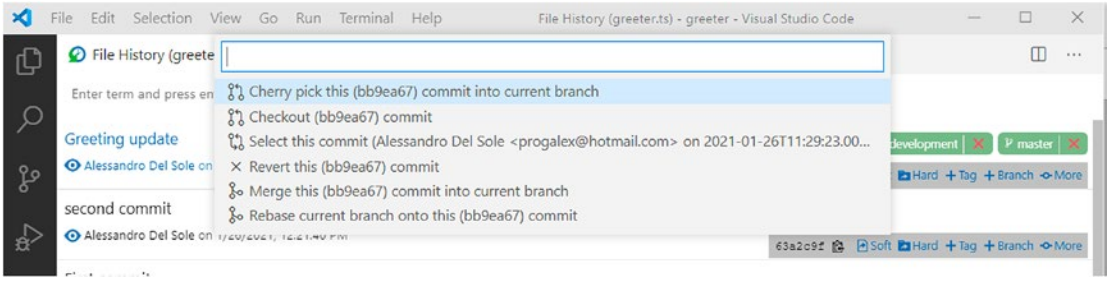

Figure 7-19. *Git History provides commands that make it easier to work with commits*

At the bottom of the view, you will see the list of files involved in the selected commit. If you click a file name, you also get 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, because 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 extension that will boost your productivity is GitLens. At first usage, GitLens requires you to be authorized by GitHub, so VS Code will invite you to follow the same steps you did when creating your first remote repository. GitLens adds to VS Code many features and commands related to Git. For example, GitLens extends the Source Control bar (see Figure 7-20) with a number of useful Git groups.

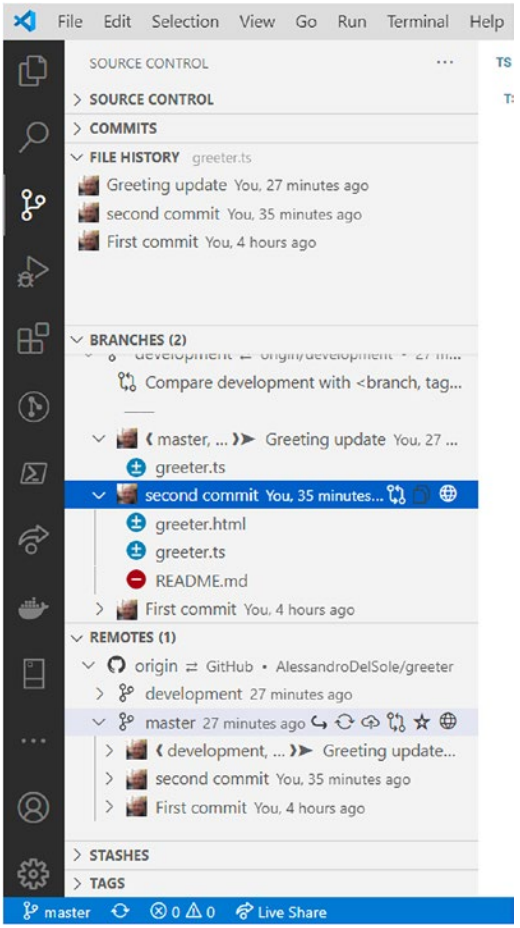

Figure 7-20. *The Source Control bar extended by GitLens*

The GitLens extension adds several areas to the Source Control bar. The **BRANCHES** and **REMOTES** areas show the list of local and remote branches, respectively, and, for each branch, GitLens displays the list of commits. Each commit can be expanded to see the commit message, the list of files involved in the commit, and an icon that represents the operation made on the file. The **STASHES** area shows stashed changes with a similar structure (if any). The **FILE HISTORY** area shows the list of commits for a file (this requires an open editor). 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 shows a list of commands, as shown in Figure 7-21.

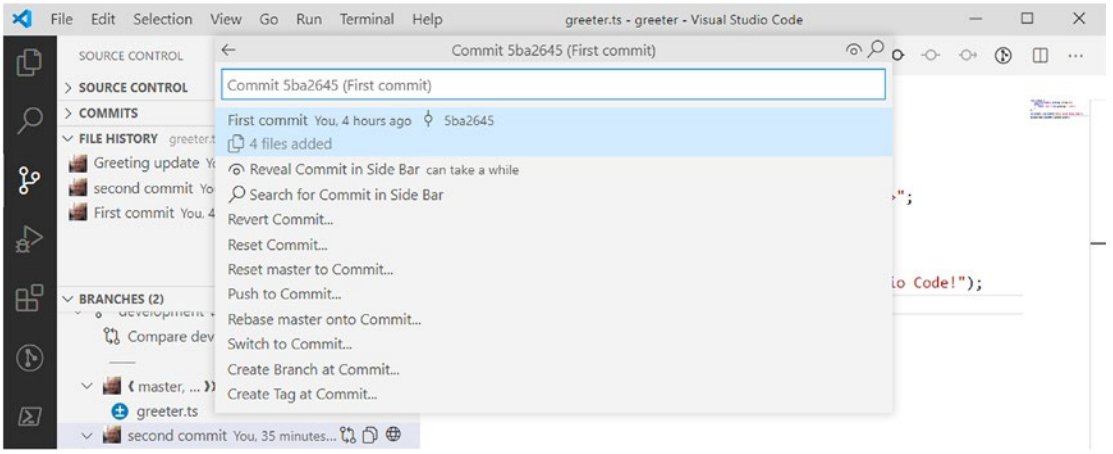

Figure 7-21. *GitLens commands*

These commands allow you to not only open the commit in your remote repository but also open the commit revisions. Additionally, you can copy 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 7-22 shows an example where GitLens highlights that a code change to the `Greeter` class was made 4 hours earlier by the author.

Note If you hover your cursor over the GitLens, you will see some information such as author, code differences, and commit number inside an interactive pop-up box.

Figure 7-22. *GitLens adds summary information about a code snippet.*

If you click at the left side of the divider, you get to the menu shown in Figure 7-21. If you instead click the author name, VS Code shows a pop-up box that contains the list of commits made by the selected author, and if you hover over a commit name, you see the full commit details (see Figure 7-23).

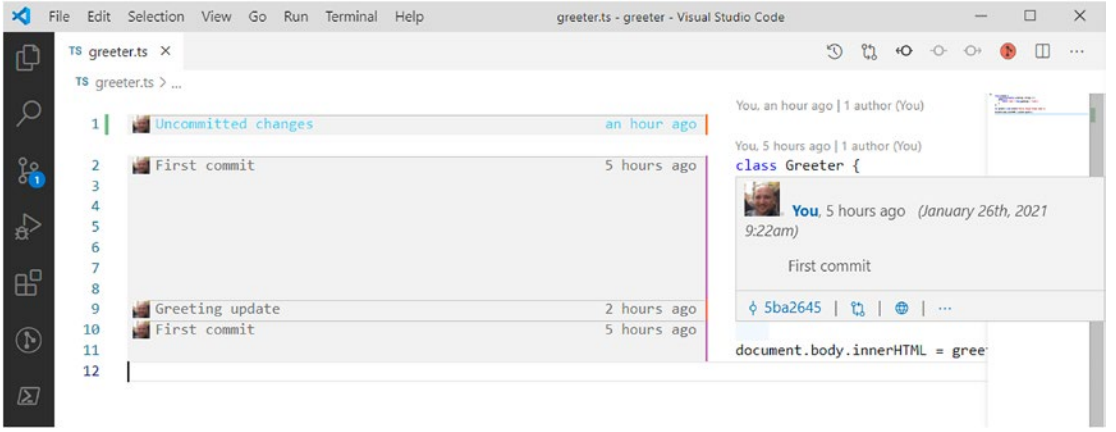

Figure 7-23. *GitLens showing information about a commit*

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 selfexplanatory.

Note All the preceding commands described are also available via shortcuts that you can find in the upper-right corner of the code editor bar (see Figure 7-23).

GitHub Pull Requests and Issues

Pull requests in Git make it easier to perform code reviews, while issues enable you to keep track of feedback from other developers. With pull requests, your code is not automatically merged into a branch until someone else on the team reviews the code and accepts it. If you use GitHub for your repositories, an extension called **GitHub Pull Requests and Issues** is available to introduce support for pull requests in Visual Studio Code. When you first install the extension (and reload the environment), you are asked to sign into GitHub. To accomplish this, you can either click **Settings** in the Side Bar and then click **Sign in to use GitHub Pull Requests and Issues**, or click the **Sign in** button in the GitHub bar. Simply follow the same steps you did to authorize GitLens.

After you provide your GitHub credentials and open a folder that is associated to a remote repository hosted on GitHub, you will be able to leverage the GITHUB bar, which you enable by clicking the GitHub icon on the Side Bar. An example of the GITHUB view is provided in Figure 7-24.

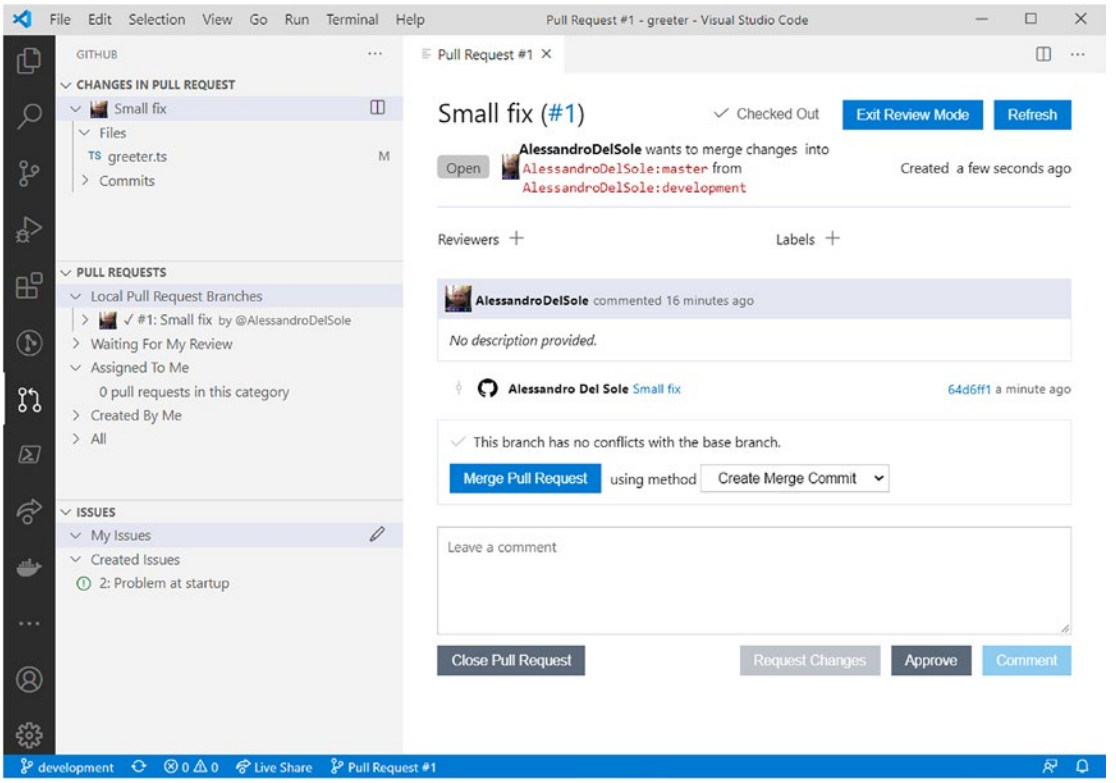

Figure 7-24. *The GitHub Pull Requests view*

The extension supports both viewing and submitting pull requests, regardless of their source, which can be VS Code, GitHub, or another development environment connected to the same repository. When pull requests are available, you see them listed in the view. If you select a pull request, a new editor window appears showing all the pull request details, and you have the option of adding comments and then closing, rejecting, or approving the pull request (see Figure 7-24).

You can also work on the pull request locally by clicking the **Checkout** button, which displays it under the Local Pull Request Branches node in the tree view.

You can create issues from within Visual Studio Code by using the + button, after which you can edit and then save them so that they are associated to the remote repository. Viewing issues happens inside the browser, so when you click the globe icon at the right side of an issue, the default web browser opens the GitHub page for the issue.

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://dev.azure.com) and Team Foundation Server are the complete solutions from Microsoft to manage the entire application life cycle, 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, and the good news is that you do not need any extensions. I will use Azure DevOps so that you do not need to have an on-premises installation of Team Foundation Server. Also, I will reuse the Greeter project described in the previous sections. If you want to do the same, you can simply delete the local .git folder located under the project folder.

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.

Creating a Team Project

From the home page, click the **New Project** button. As you can see in Figure 7-25, you need to supply a team project name, a source control engine, and a work item process.

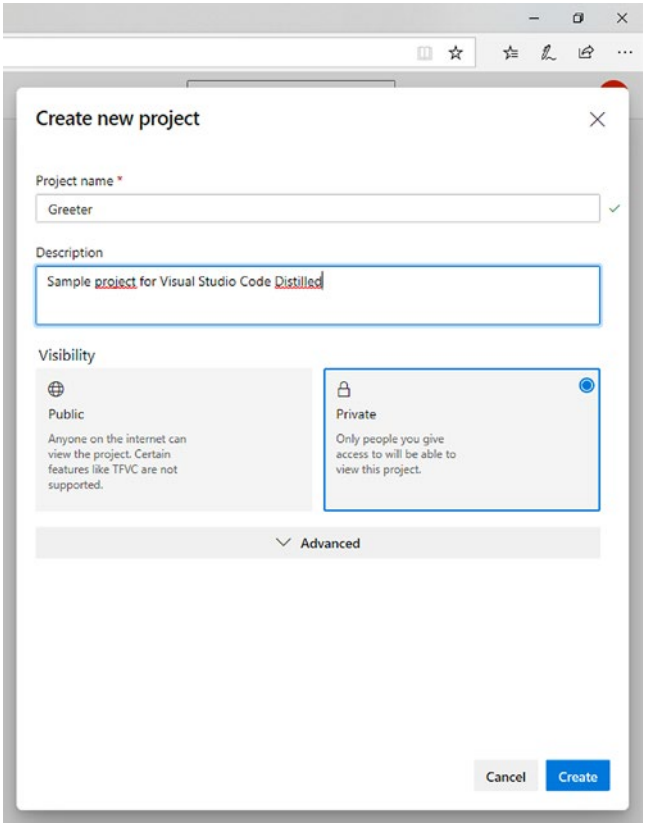

Figure 7-25. *Creating a team project in Azure DevOps*

Enter a project name and click **Create**. After a few seconds, your new team project will be ready. At this point, the Azure DevOps site shows a page with all the information about your new team project. Now click **Repos** on the left side of the screen so that you can see all the information about the new Git repository (see Figure 7-26). Notice that the new repository is created with the same name as the new project. Copy the repository URL into the clipboard, as it will be necessary very shortly.

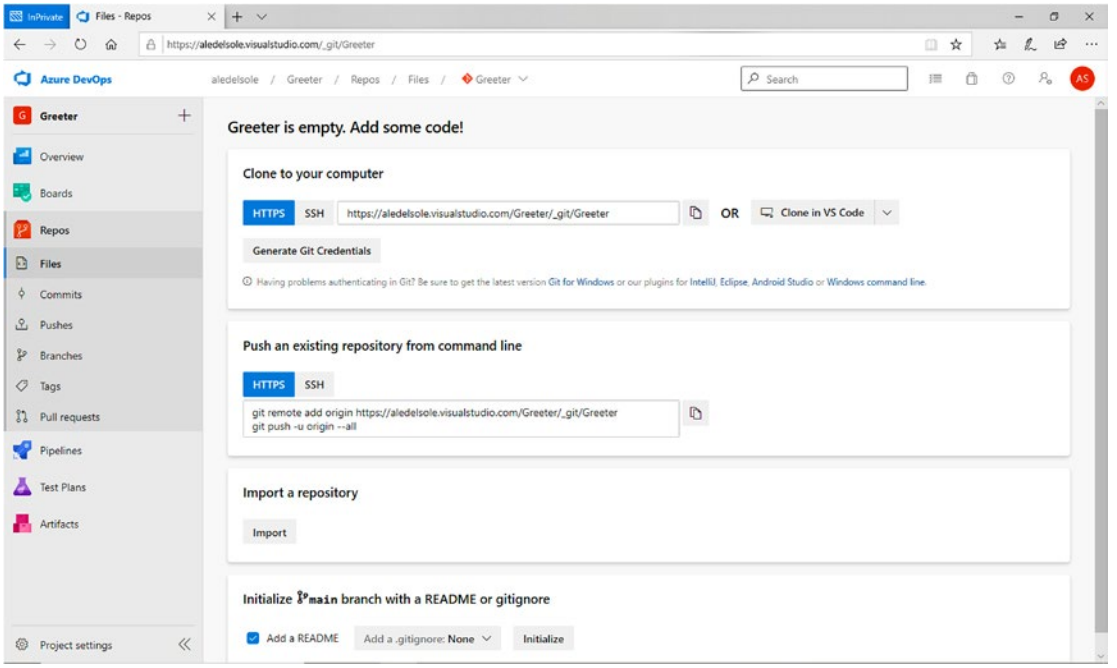

Figure 7-26. *Information about a Git repository on Azure DevOps*

Now that a remote repository is set up, you have several options to associate it to Visual Studio Code. You could clone the repository to the local machine, or you could even use the Git CLI. However, the simplest yet most effective option is to use the VS Code tools you have seen in the first part of this chapter, as described next.

Connecting Visual Studio Code to a Remote Repository

Go back to Visual Studio Code. The first thing to do is initialize a local Git repository (see the "Initializing a Local Git Repository" section earlier in the chapter for a refresher). Once you have a local repository set up, you can connect it to the remote Azure DevOps repository with little effort.

In the Source Control bar, click the **...** button, then **Remote ➤ Add Remote**. You first need to specify the name of the remote repository (which is the one you specified in Azure), then you will have the option to enter the URL of the remote repository you created, so paste the URL and press Enter (see Figure 7-27).

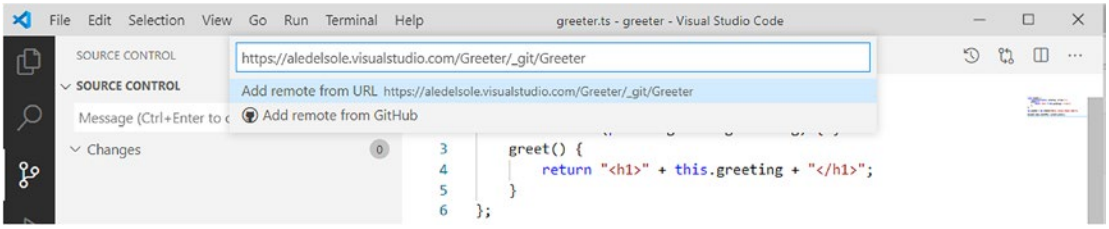

Figure 7-27. *Specifying an Azure DevOps remote repository*

You are also asked to provide a name, which is used as a project identifier. Enter a name of your choice, with no blank spaces, then press Enter. At this point Visual Studio Code links the local repository to the remote one, but note that you do not get any confirmation message of the operation completion, only indicators running on the Status Bar.

The very last step is to push the branch to the remote repository, using any of the options described in the first part of this chapter; however, you need to take care about the main branch. As previously mentioned, due to recent changes in Azure DevOps that reflect what GitHub also does, when you create a repository on Azure DevOps, the main branch is now named `main` rather than `master`. The problem is that VS Code still creates a `master` branch. So basically you need to push the `master` branch from VS Code and then create a pull request to merge `master` into `main` so that you will be able to work with the new branch.

Note All these steps are necessary if you connect existing code to a remote repository. If you start from creating a remote repository for a new project, you can clone the repository in VS Code so that you start with the main branch directly.

Once changes are pushed, they are visible in the Repos view of the Azure DevOps project (see Figure 7-28).

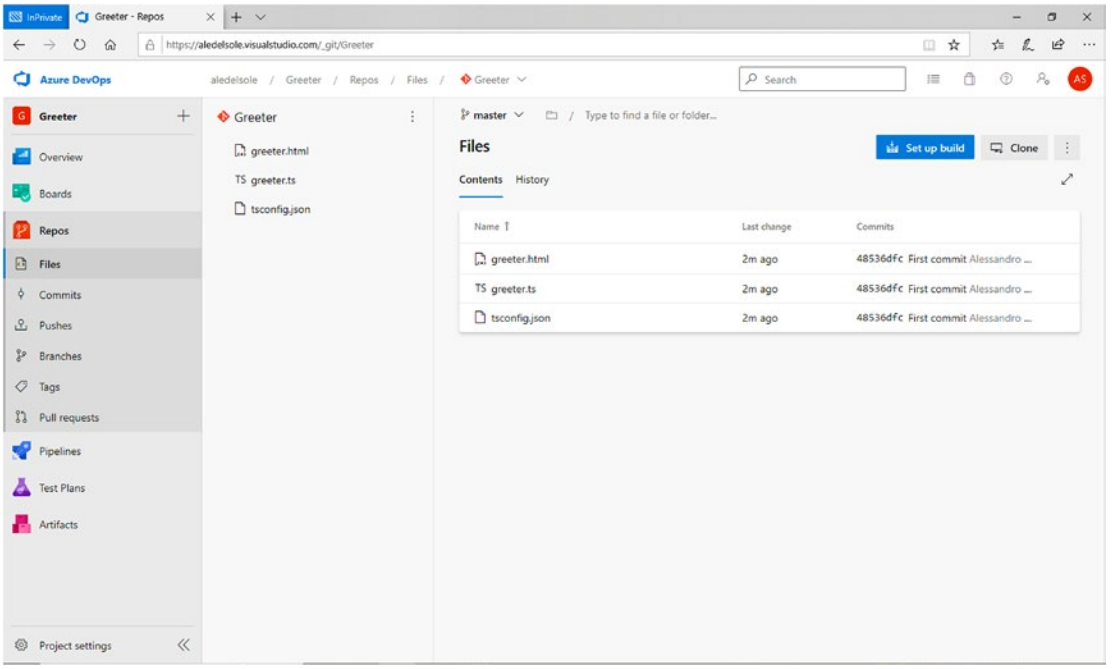

Figure 7-28. *The source code has been pushed to Azure DevOps*

Now that your code has been pushed remotely, other developers will be able to collaborate on the project. The key point is how easy it has been to set up a connection between a local Git repository and a remote Azure DevOps one, all from within Visual Studio Code.

Summary

Writing software involves collaboration, whether you are part of a development team, involved in open source projects, or are an individual developer who has interactions with customers. In this chapter you have explored 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. You have also 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 and Issues, that will boost your productivity by adding important features that every developer needs when it comes to team collaboration. Finally, you learned how easy it is to link a local repository to a remote Git repository hosted on Azure DevOps, the premiere cloud solution from Microsoft to manage the whole application life cycle. Behind the scenes, Visual Studio Code invokes the Git commands to execute operations over your source code, and it is preconfigured to work with this external tool.

However, Visual Studio Code 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 about in the next chapter.

CHAPTER 8

Automating Tasks

As described in previous chapter, Visual Studio Code is more than a simple code editor because it enables you to execute operations such as compilingand testing code by running external tools. In this chapter you will learn how VS Code can execute external programs via tasks, by both existing tasks and customized tasks. To run the examples provided in this chapter, you need the following software:

- Node.js, a free and open source JavaScript runtime based upon Chrome's JavaScript engine, which you can download 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 helps 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 in this chapter about MSBuild tasks on Windows, you instead need Microsoft Visual Studio 2019. The Community edition is available for free at https://visualstudio.microsoft.com.

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 life cycle, such as compilation, debugging, and testing.

A. Del Sole, *Visual Studio Code Distilled*, https://doi.org/10.1007/978-1-4842-6901-5_8

In Visual Studio Code terminology, integrating with an external program within the flow of the application life cycle is a *task*. Running a task means not only 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, such as the integrated Terminal.

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 is not always necessary, and VS Code does all the work for you. If VS Code 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, generates a tasks.json file. If VS Code is able to detect the type of project or source code inside the folder, it also prefills the tasks.json content with the proper information; otherwise, you 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 recompiles 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 in specific situations.

Running and Managing Tasks

The first approach to understanding tasks in practice is to run 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`).

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

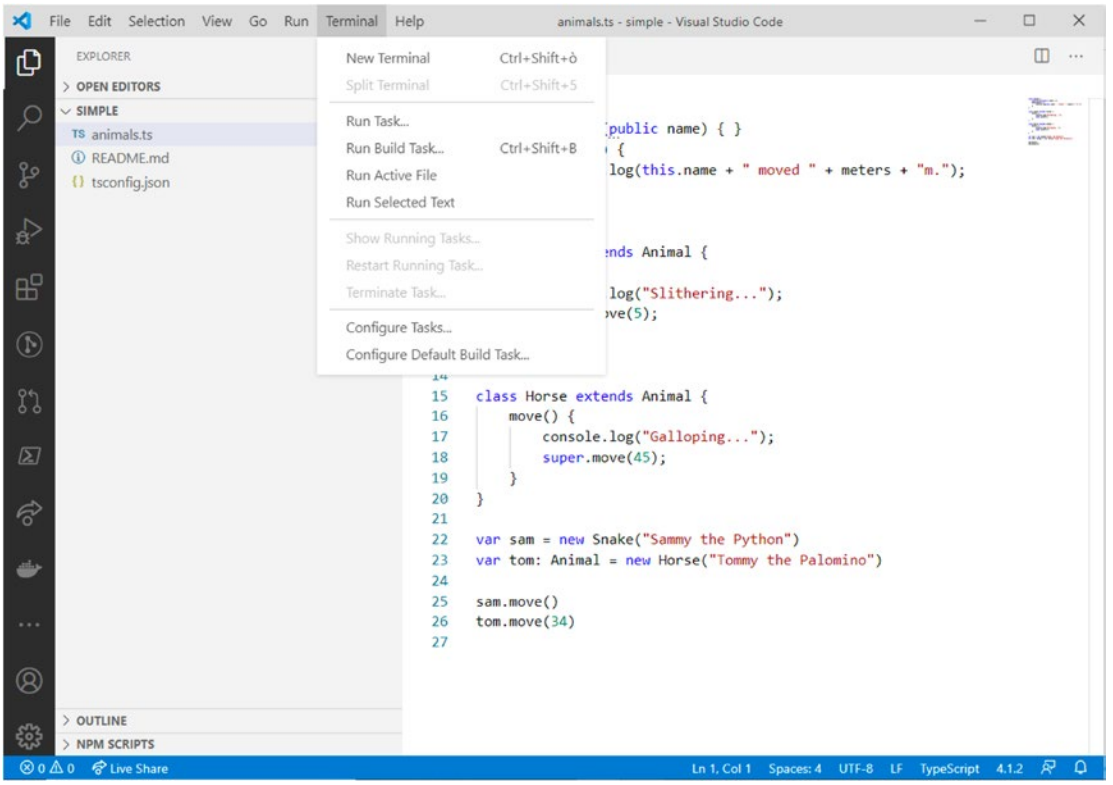

Figure 8-1. *Commands for running and managing tasks in the Terminal menu*

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 Tasks	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 selection of the task to use as the build task

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

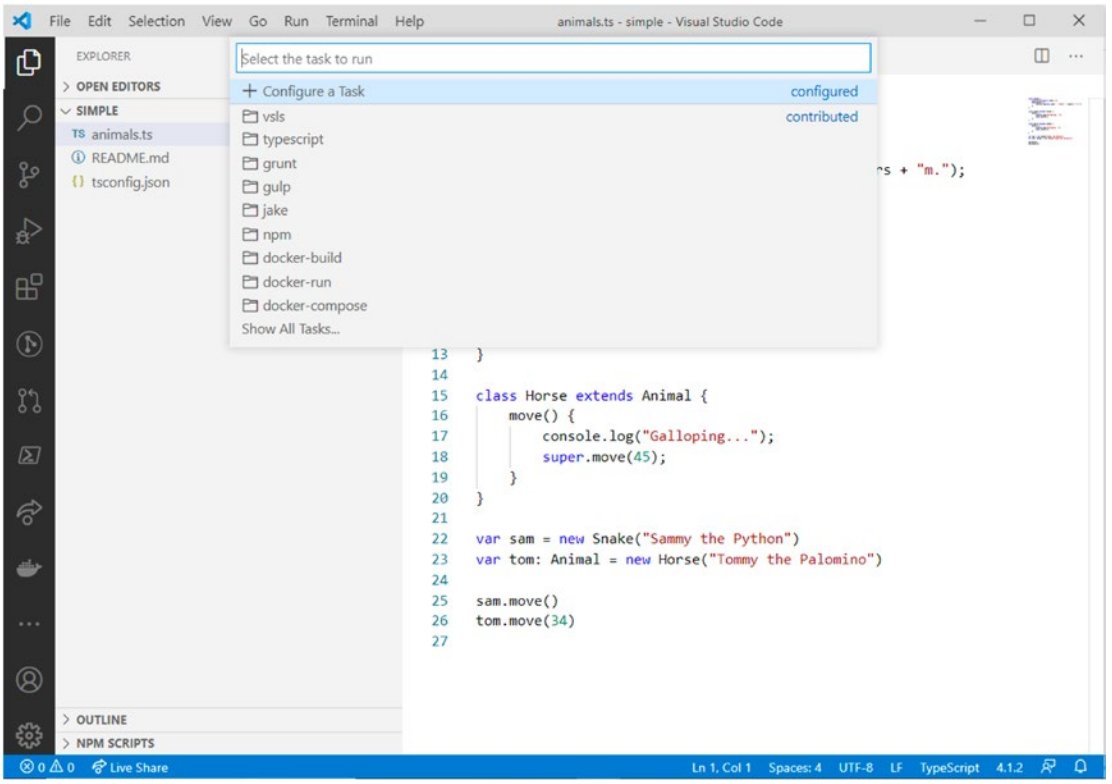

Figure 8-2. *Selecting task categories from the Command Palette*

From here you can pick up a group of available tasks by category. In this case, you need to select the **typescript** category. At this point the Command Palette displays the list of available tasks for that category, as you can see in Figure 8-3.

Figure 8-3. *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 launches the tsc compiler and compiles the TypeScript code into JavaScript code, as shown in Figure 8-4.

Note In the case of TypeScript, the build task compiles TypeScript code into JavaScript code. In the case of other languages, the build task generates binaries from the source code. More generally, a build task produces 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.

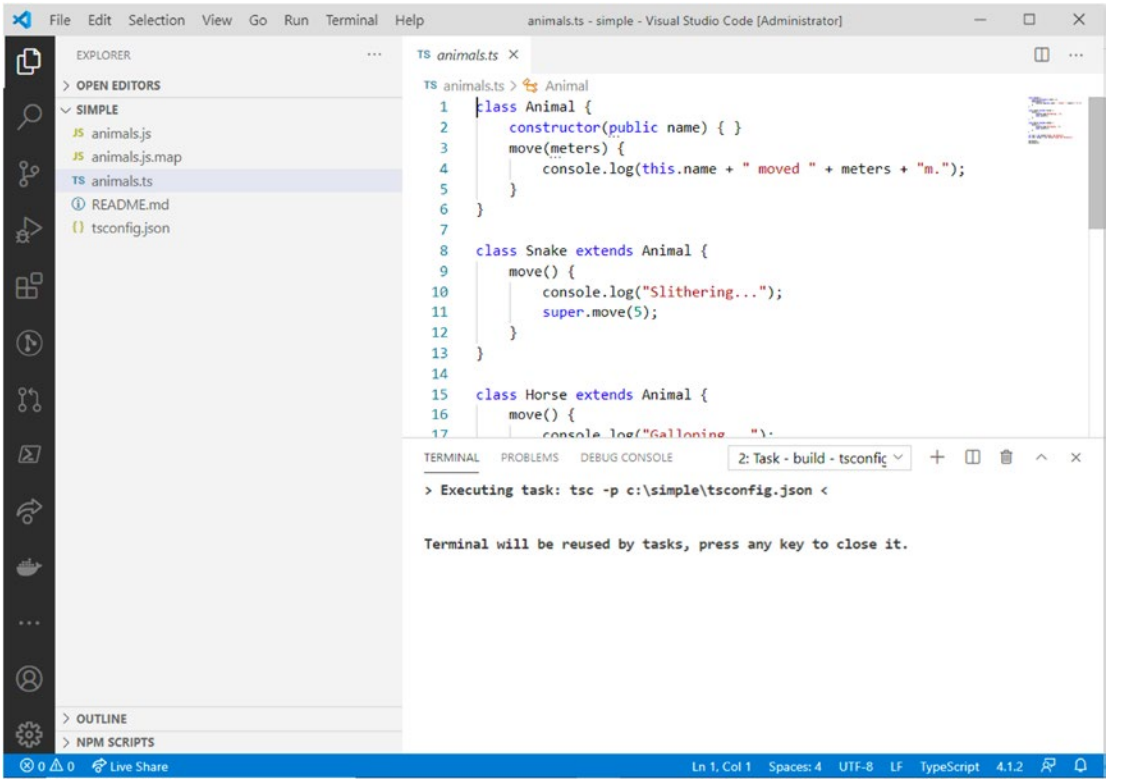

Figure 8-4. *Executing a build task*

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.

Note If the Terminal shows an error message saying that a .ps1 file could not be loaded because running scripts is disabled on the systems, try to first restart VS Code as an administrator and to repeat the steps. If this does not solve the issue, you need to enable script execution on your machine. You can do this on your own if you are the computer administrator; otherwise you need to ask the administrator of your network. You can find more detailed information on how to enable script execution depending on your environment and on how to enable specific privileges at https:/go.microsoft.com/fwlink/?LinkID=135170.

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 8-5, describing the error and the line of code that caused it.

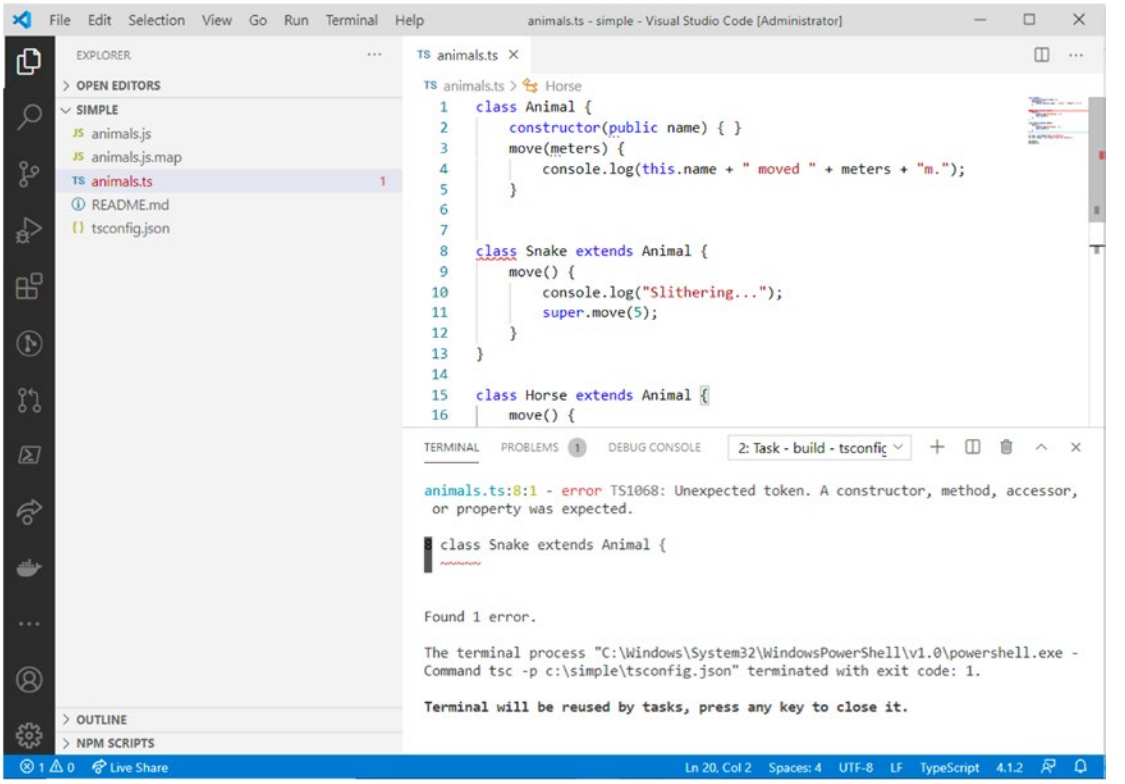

Figure 8-5. *Visual Studio Code shows the output of the external tool in a convenient way*

In the real world, this error probably would not 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, because 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 generates a new tasks.json file under the .vscode folder, and it then opens this file in a new editor window. The content and structure of tasks.json will be discussed in the upcoming "Configuring Tasks" section, so for now let's focus on the new default build task. Select **Terminal ➤ Run Build Task**, or use the keyboard shortcut, and you will see how the default build task will be executed, without the need to specify 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 8-6 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`.

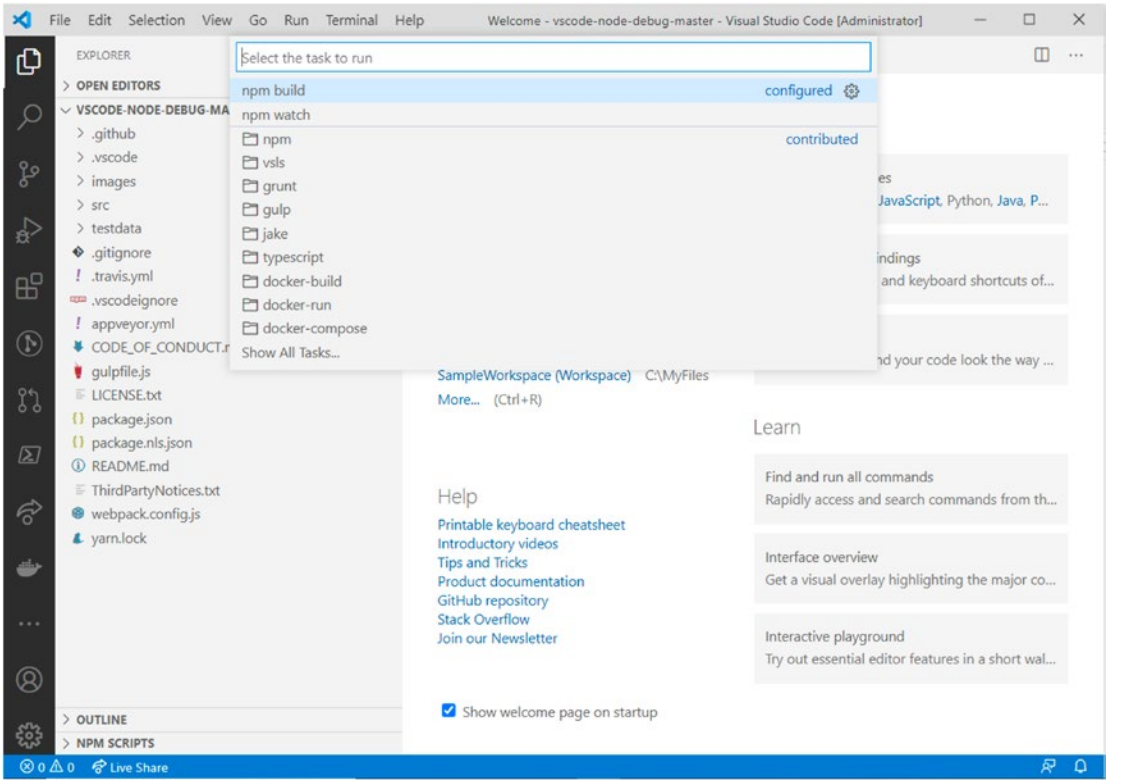

Figure 8-6. *Auto-detected tasks*

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, such as the **npm build** and **npm watch** tasks. You can then open the **npm** category to view the full list of preconfigured tasks that can run against npm.

Auto-detected tasks are very useful because they allow you 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 present 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.

To complete both the examples, you 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 provides a free command-line compiler. Free Pascal is available for Windows, macOS, Linux, and other systems, and you can download it from `https://www.freepascal.org`.
- On Windows only, download the latest version of the .NET Framework (4.8 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 C:\FPC\VersionNumber on Windows and /FPC/VersionNumber on macOS and Linux), 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 8-7 shows how Visual Studio Code appears with Pascal source files currently opened.

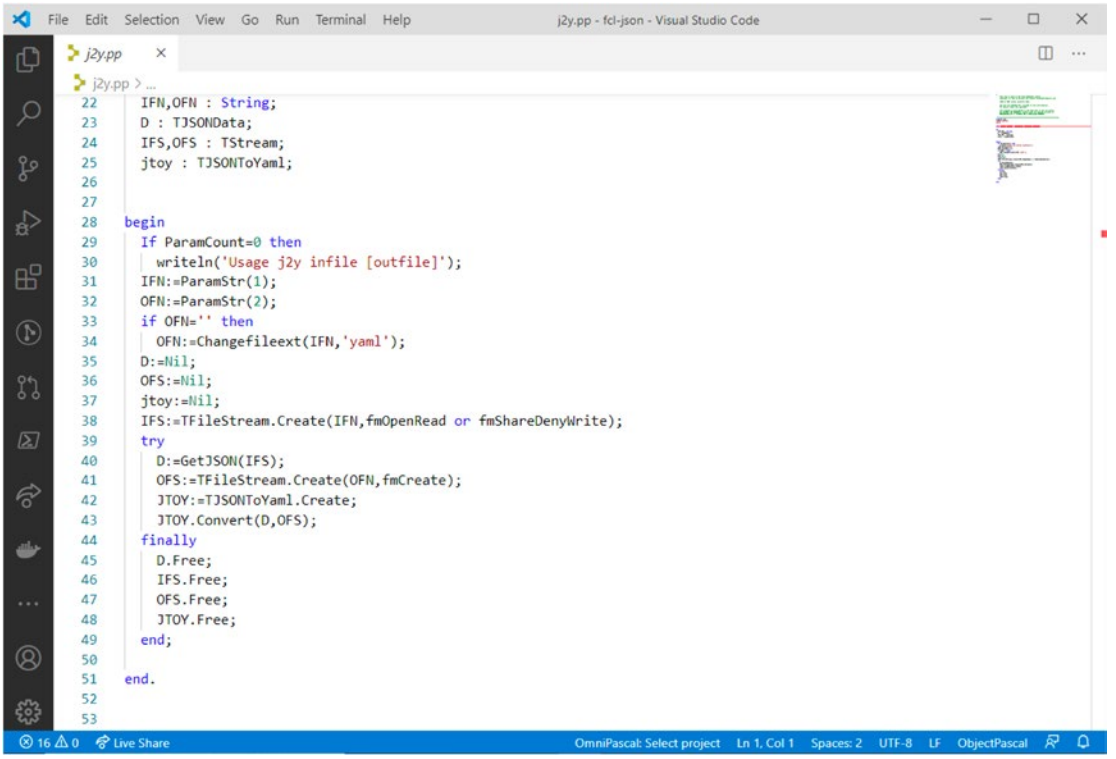

Figure 8-7. *Editing Pascal source code*

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. You can accomplish this 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 Tasks**. When the Command Palette appears asking for a task to configure, select **Create tasks.json file from template** (see Figure 8-8). There is no existing task to configure at this particular point, so the only thing you can do is create a new tasks.json file.

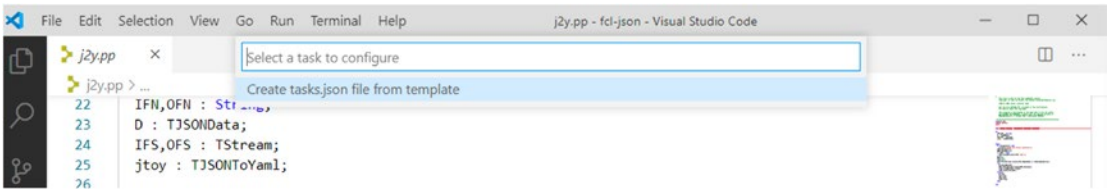

Figure 8-8. *Creating a new task from scratch*

2. The Command Palette shows the list of available task templates: **MSBuild**, **maven**, **.NET Core**, and **Others** (see Figure 8-9). Select **Others** to create a new task that is independent from other systems.

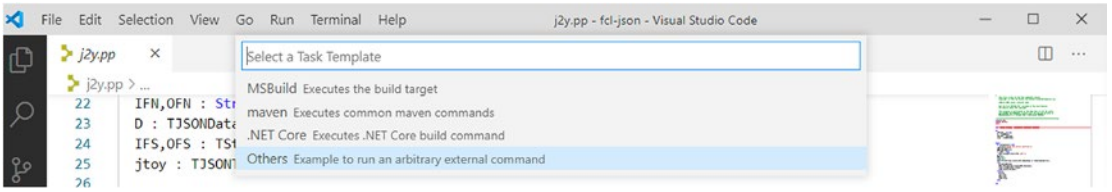

Figure 8-9. *Selecting a task template*

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 task, 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 you to specify command-line arguments for the program you invoke. The list of supported JSON properties is available in Table 8-2 in the upcoming "Understanding tasks.json Properties" section, but if you are impatient, you can quickly look at the table and then return here.

Now suppose you want to create a build task, which, by convention, is the type of task you use to compile source code. You can accomplish this 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, which 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 claims to take care of registering the program's path, but remember to take 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. Variables are discussed in the section "Understanding Substitution Variables" and summarized in Table 8-3 later in this chapter. Notice how IntelliSense helps you find the appropriate properties in tasks.json, as shown in Figure 8-10.

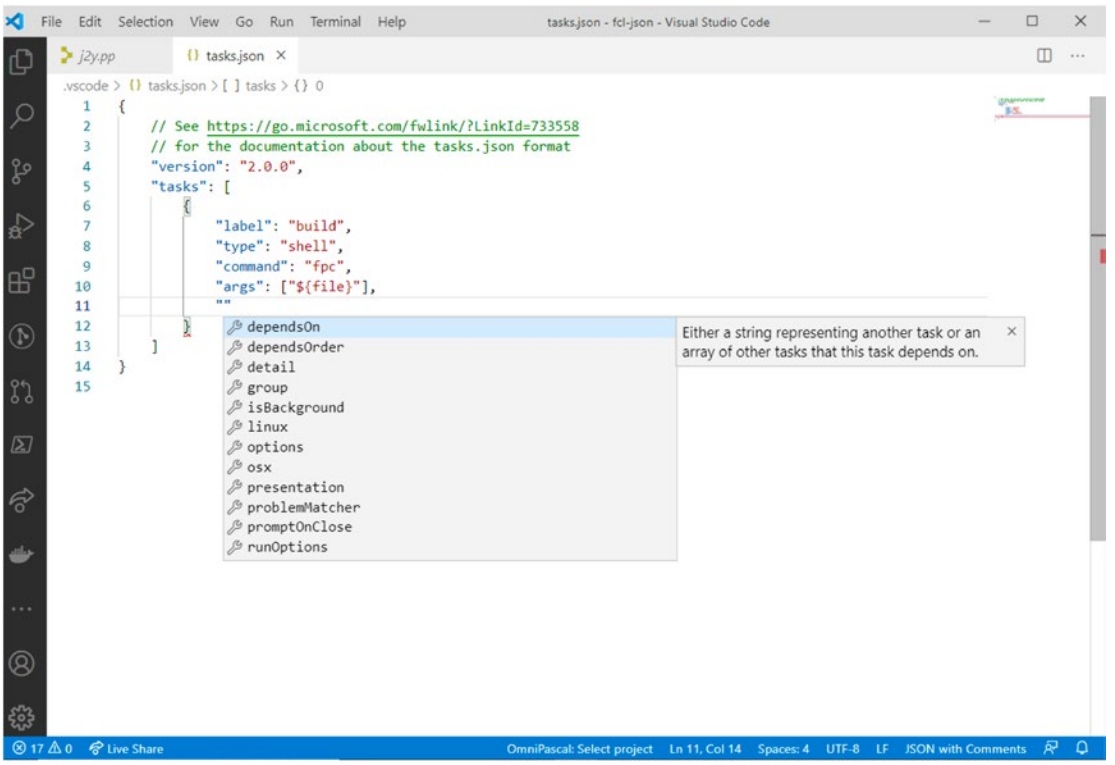

Figure 8-10. *IntelliSense helps defining task properties*

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 8-11).

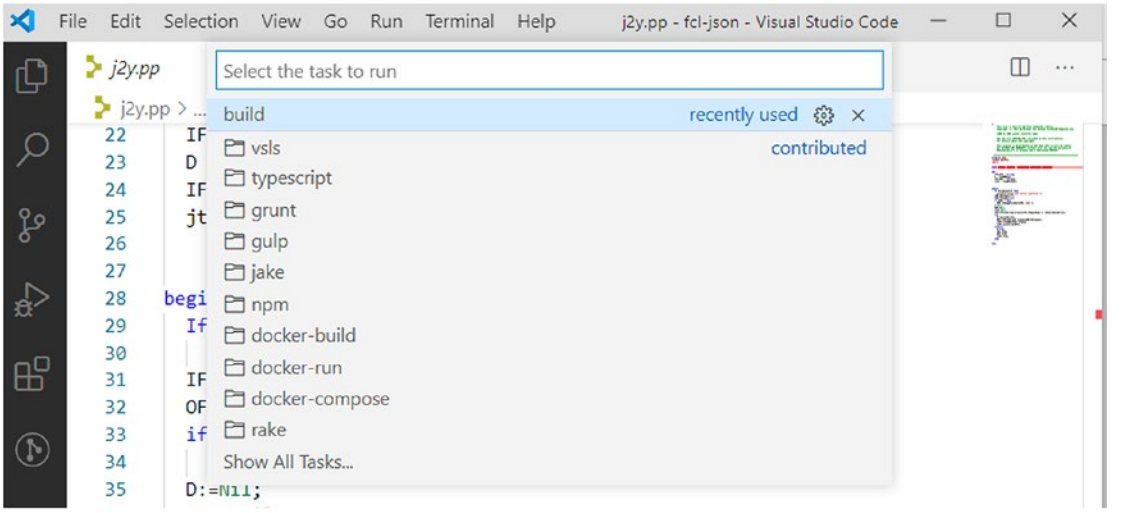

Figure 8-11. *Selecting the new task*

At this point, VS Code asks what would you like to do to detect any problems encountered during the execution of the external program so that it can display them 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 the section "Understanding Problem Matchers" later in this chapter. For now, select **Continue without scanning the task output** (see Figure 8-12).

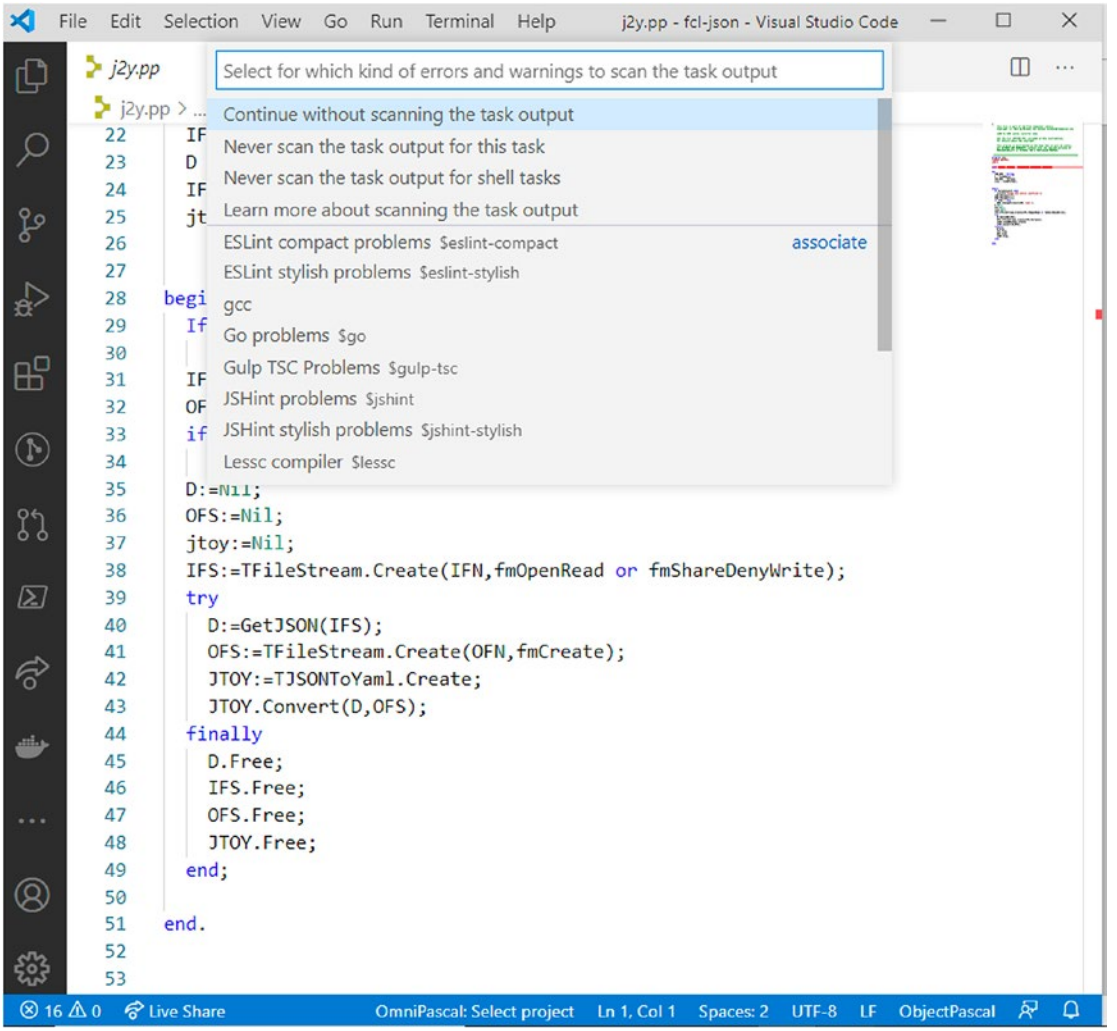

Figure 8-12. *Selecting a problem matcher*

The Free Pascal compiler is executed in the Terminal panel, where you also see the program output, as demonstrated in Figure 8-13.

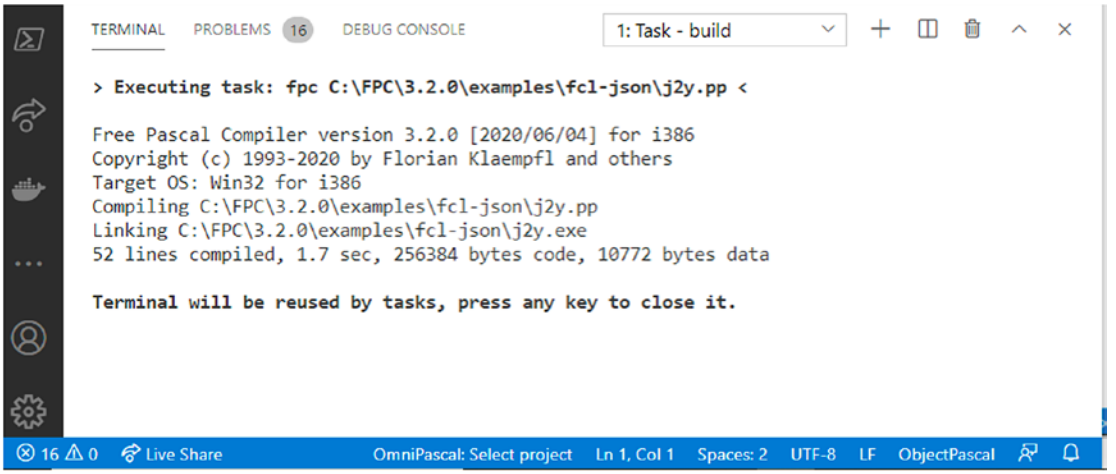

Figure 8-13. *Executing the Free Pascal compiler*

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

The tasks.json file can define multiple tasks. As introduced earlier in this chapter, among 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",
    "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 see both tasks in the Command Palette, as demonstrated in Figure 8-14.

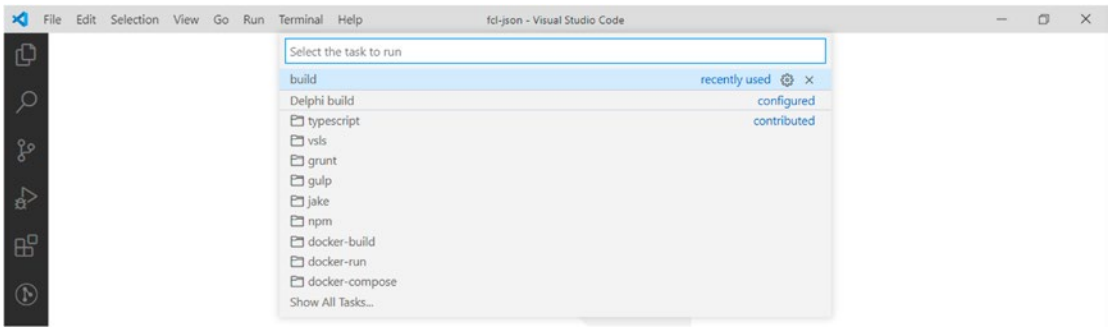

Figure 8-14. *All defined tasks are displayed in the Command Palette*

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 to configure a

default build task for Pascal files. As you learned in the "The Default Build Task" section previously, you can easily accomplish this with the following steps:

1. Select **Terminal ➤ Configure Default Build Task**.
2. In the Command Palette, select the **build** task defined previously by adding an `isDefault` parameter (as you will see shortly in code).
3. With a Pascal source file active, select **Terminal ➤ Run Build Task**, or press the keyboard shortcut for your system.

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

Understanding tasks.json Properties

There are a number of properties available to customize a task. Table 8-2 provides a summary of common properties that you can 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 `shell` and `process`. With `shell`, the command is interpreted as a shell command (such as bash, cmd, or PowerShell). With `process`, 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.
`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.

(*continued*)

Table 8-2. (*continued*)

Property Name	Description
`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",
            "command": "fpc",
            "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 8-15).

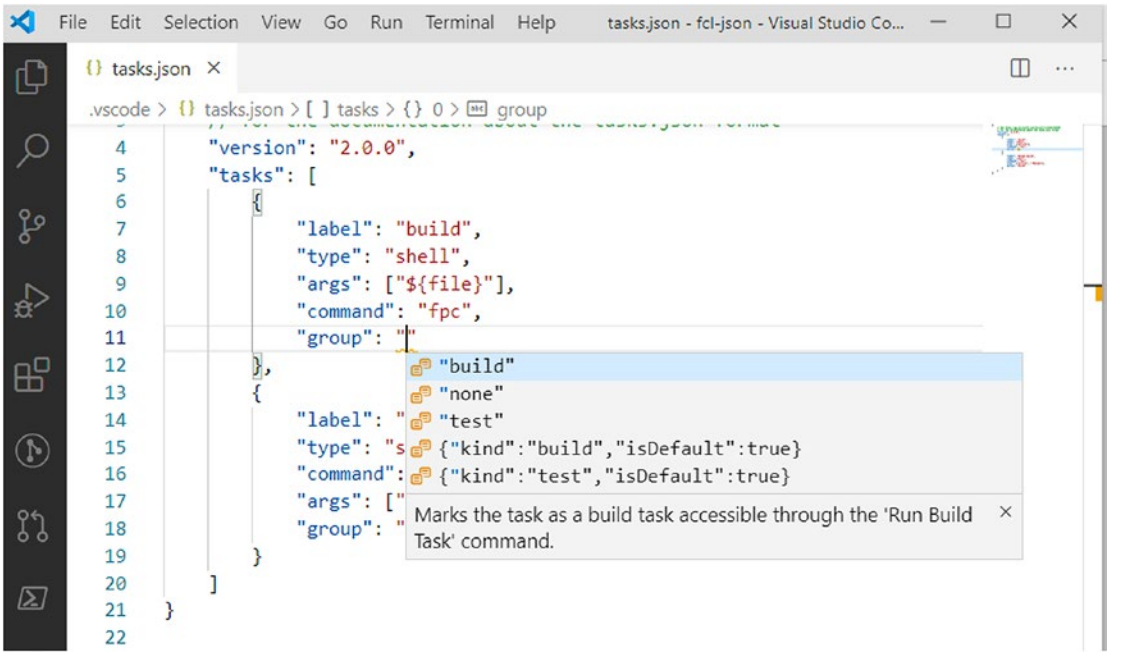

Figure 8-15. *IntelliSense helping with groups*

You can also specify additional values for 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 Tab, 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 focus when the task is running.
- `panel` can be `shared`, `dedicated`, or `new` and 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; the schema is available at https://code.visualstudio.com/docs/editor/tasks-appendix.

Understanding Substitution Variables

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
`${workspaceFolder}`	Represents the path of the currently opened folder.
`${workspaceFolderBasename}`	Represents the name of the currently opened folder without any slashes.
`${file}`	The path to the active code file.
`${relativeFile}`	The active code file relative to `${workspaceFolder}`.
`${fileBasename}`	The active code file's base name, without path and leading slash.
`${fileBasenameNoExtension}`	The active code file's base name without the extension.
`${fileDirname}`	The path of the directory that contains the active code file.
`${fileExtname}`	The file extension of the active code file.
`${cwd}`	The current working directory of the task.
`${lineNumber}`	The currently selected line number in the active file.
`${selectedText}`	The currently selected text in the active file.
`${env.VARIABLENAME}`	References an environment variable, such as `{$env.PATH}`.

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 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 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.

Reusing Existing Task Templates

In the previous example about compiling Pascal source code, you saw 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 commands 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 that shown in Figure 8-9. 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",
// Do not generate summary otherwise it leads to duplicate errors in
   Problems panel
                "/consoleloggerparameters:NoSummary"
            ],
            "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 desktop projects built upon the .NET Framework), 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 support 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"
        },
        {
            "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 that 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",
            "type": "shell",
            "args": [
                "build",
                // Ask dotnet build to generate full paths for file names.
                "/property:GenerateFullPaths=true",
                // Do not generate summary otherwise it leads to duplicate
                   errors in Problems panel
                "/consoleloggerparameters:NoSummary"
            ],
            "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 an 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.

Note Starting with .NET Core 3, it is possible to build desktop apps with C# and Visual Studio Code will be able to debug and run them without any additional configuration. However, desktop apps have been built for decades with Windows Presentation Foundation and Windows Forms upon the full .NET Framework. Because Visual Studio Code has no direct support for .NET Framework, you will need to customize the tasks configuration as explained in this section.

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 2019 create a blank WPF project with Visual Basic as the language. There's no need to write code, as the focus is 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 demonstrate.

Select **Terminal ➤ Configure Tasks**. Select the **Create template from task.json** option first, then 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` (this is what I have on my machine) with the name of the Visual Studio edition you have on your machine, for example:

```
"command": "C:\\Program Files (x86)\\Microsoft Visual Studio\\2019\\
Enterprise\\MSBuild\\Current\\Bin\\MSBuild"
```

Also, change the value of the `reveal` property from `silent` to `always` for demonstration purposes, so that you can see the output of MSBuild in the Terminal panel. Now select **Terminal ➤ Run Task** and select the preconfigured build task, and MSBuild will be started and the solution will be built, as you can see in Figure 8-16.

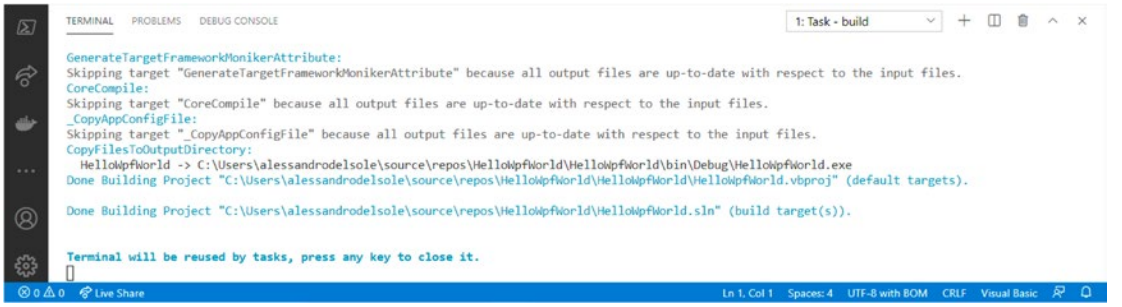

Figure 8-16. *Compiling a WPF project written in Visual Basic with the MSBuild task*

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 delve into problem matchers in a bit more detail.

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#_processingtaskoutput-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(.*)",

// The first match group matches the file name which is relative.
        "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, whose value 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 that you read the official documentation available at https://code.visualstudio.com/docs/editor/tasks#_defining-a-problem-matcher.

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, you can run a batch program (.bat) in Windows or a shell script file (.sh) on macOS by simply clicking **Terminal ➤ Run Active File**.

The file name is passed to the current terminal program on your system (PowerShell on Windows or the bash shell on Linux and macOS) so that the operating system tries 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 executes the file. The output is 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 life cycle 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 can 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 two chapters, first with C# and then with Python.

CHAPTER 9

Building and Debugging Applications: .NET 5 and Other Platforms

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 first provides a general overview of application development, and then it explains how to build .NET 5 projects supported in Visual Studio Code and how to use all the built-in, powerful debugging features. Even if you do not plan to use C# with Visual Studio Code, I recommend that you read this chapter because most of the concepts are applicable to other languages as well, especially TypeScript, JavaScript, and Python.

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. This section explains how to build projects based on the new .NET 5, but you can similarly create projects with the command-line interface offered by other platforms.

I also recommend that you create a dedicated folder on disk for the following examples. With the help of the file manager tool on your system (Windows Explorer on Windows, Finder on macOS, and Nautilus on Linux distributions such as 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.

A. Del Sole, *Visual Studio Code Distilled*, https://doi.org/10.1007/978-1-4842-6901-5_9

Note The following topics are discussed in the context of .NET 5, but Visual Studio Code supports all .NET Core versions up to 3.1. All explanations and examples therefore apply to .NET Core as well.

Introducing .NET 5

.NET 5 is the new major release of the Microsoft .NET technology. After releasing .NET Core a few years ago, Microsoft has had in mind the vision of a complete unification between .NET Framework and .NET Core, working on a single, cross-platform API that could bring the great power of .NET to any developer on any system.

As you might know, .NET Core is a cross-platform, open source, modular runtime 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. Its bigger brother, the .NET Framework, also includes the ability to create desktop applications, such as Windows Forms and Windows Presentation Foundation, but the .NET Framework's biggest limitation is that it only runs on Windows.

So .NET 5 can be considered as an update for both .NET Core and .NET Framework; with it, Microsoft brings together the two technologies and offers a unified development platform that has the flexibility and portability of .NET Core, plus the full power of .NET Framework. .NET 5 also includes C# 9 and F# 5, but it does not support mobile development with C# and F#, which is planned for .NET 6 with the inclusion of Xamarin. Additionally, at this writing, with .NET 5 you can only create desktop apps on Windows.

There are several ways to get .NET 5. As a developer working with Visual Studio Code, the easiest way is to download the latest release from the official website (`https://dotnet.microsoft.com`). This website enables you to select the installation package that matches your operating system. For the following explanations and examples, I'm assuming you have downloaded and installed .NET 5 on your machine.

Creating .NET 5 Projects

.NET 5 ships with a rich command-line interface that provides many options to create different kinds of projects and individual files. You can create projects and files from the command line by using the `dotnet` tool, more specifically by invoking the `dotnet new` command. For example, if you want to create a Console application with C#, you would enter the following command:

```
> dotnet new console
```

By default, the `dotnet` tool assumes you want to use C# unless you explicitly specify a different language. For example, the following command enables you to create a Console application with Visual Basic:

```
> dotnet new console -lang VB
```

Table 9-1 provides a comprehensive list and description of all the available templates.

Table 9-1. *Available .NET Project and File Templates*

Template Name	Short Name	Language
Console Application	console	C#, F#, VB
Class Library	classlib	C#, F#, VB
WPF Application	wpf	C#, VB
WPF Class Library	wpflib	C#, VB
WPF Custom Control Library	wpfcustomcontrollib	C#, VB
WPF User Control Library	wpfusercontrollib	C#, VB
Windows Forms (WinForms) Application	winforms	C#, VB
Worker Service	worker	C#
Unit Test Project	mstest	C#, F#, VB
NUnit 3 Test Project	nunit	C#, F#, VB

(*continued*)

Table 9-1. (*continued*)

Template Name	Short Name	Language
NUnit 3 Test Item	nunit-test	C#, F#, VB
xUnit Test Project	xunit	C#, F#, VB
Razor Component	razorcomponent	C#
Razor Page	page	C#
MVC ViewImports	viewimports	C#
MVC ViewStart	viewstart	C#
Blazor Server App	blazorserver	C#
Blazor WebAssembly App	blazorwasm	C#
ASP.NET Core Empty	web	C#, F#
ASP.NET Core Web App (Model-View-Controller)	mvc	C#, F#
ASP.NET Core Web App	webapp, razor	C#
ASP.NET Core with Angular	angular	C#
ASP.NET Core with React.js	react	C#
ASP.NET Core with React.js and Redux	reactredux	C#
Razor Class Library	razorclasslib	C#
ASP.NET Core Web API	webapi	C#, F#
ASP.NET Core gRPC Service	grpc	C#
dotnet gitignore file	gitignore	
global.json file	globaljson	
NuGet Config	nugetconfig	
Dotnet local tool manifest file	tool-manifest	
Web Config	webconfig	
Solution File	sln	
Protocol Buffer File	proto	

Note All Windows Forms and WPF templates are available to Visual Basic only with .NET 5. For C# and F# they have already been available since .NET Core 3.1. However, the majority of the templates described in Table 9-1 have been available since previous versions.

In this section I will show an example based on C# and an ASP.NET Core web application built upon the Model-View-Controller (MVC) pattern. 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`, which is also commonly used on Windows as a shortcut for `chdir`.

Next, type the following command to build a new .NET 5 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 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 5 will automatically restore NuGet packages for the solution. You could also do this manually by typing the following command:

```
> dotnet restore
```

If you were to type `dotnet run`, the development server would start running and then you would need to open your browser and launch the application manually. 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 recognizes 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 to run 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 5, 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 **Run ➤ Run Without Debugging**. Visual Studio Code first asks you to specify an environment, so select .NET Core, then it starts the default build task. For Web applications, VS Code starts an instance of the development server, but in order to run the application you need to manually open the browser and enter the Web address you see in the Terminal panel.

Note The first time you run some code, VS Code might show a pop-up message saying that required assets are needed to enable building and debugging. Accept the offer and VS Code will do the rest.

Figure 9-1 shows the web application built previously.

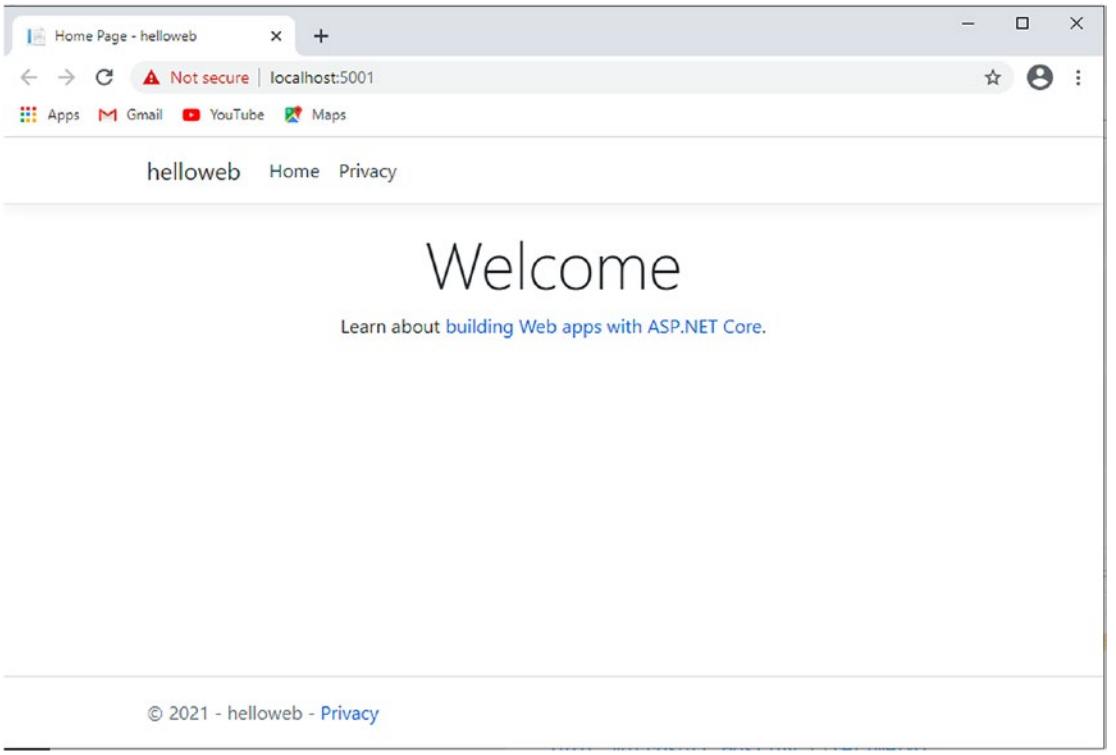

Figure 9-1. *The .NET web application running*

Note Your browser might show a warning saying that the website is not secure. Because the local development environment is currently being used, you can ignore the warning and proceed to display the web page. Also, some browsers might ask to add a security exception for the current site, which you might want to accept to avoid the warning every time.

ASP.NET web applications use an open source development server called Kestrel (`https://github.com/aspnet/AspNetCore`), which allows for independence from proprietary systems. By default, Kestrel listens for the application on port 5001, which means your application can be reached at `http://localhost:5001`. You can change the default port setting in a file called launch.json, which is discussed more thoroughly in the later section "Configuring the Debugger."

With the preceding simple steps, you have been able to create and run a .NET 5 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 5 is not the only platform you will use with VS Code. Depending on the platform, you will use specific command-line tools to build a new project. In the next chapter you will learn how to work with Python projects, but providing some context in this chapter is worthwhile as well. For example, with Node.js you can quickly create JavaScript projects based on the Express.js framework (`https://expressjs.com`).

Express is a minimal and flexible Node.js web application framework that provides a robust set of features to develop web and mobile applications. It facilitates the rapid development of Node-based web applications and includes features such as setting up middleware to respond to HTTP requests, defining a routing table used to perform different actions based on HTTP methods and URL, and dynamically rendering HTML pages based on passing arguments to templates. An easy way to start creating apps with Express is to use the Express application generator (`https://expressjs.com/en/starter/generator.html`), which you install with the following command:

```
> npm install -g express-generator
```

Next, you can generate a JavaScript project with the following command:

```
> express expressexample
```

Note that npm requires using all lowercase letters. You can then type `code .` to open the new project in Visual Studio Code. Figure 9-2 shows a JavaScript project created with the Express JavaScript framework inside Visual Studio Code.

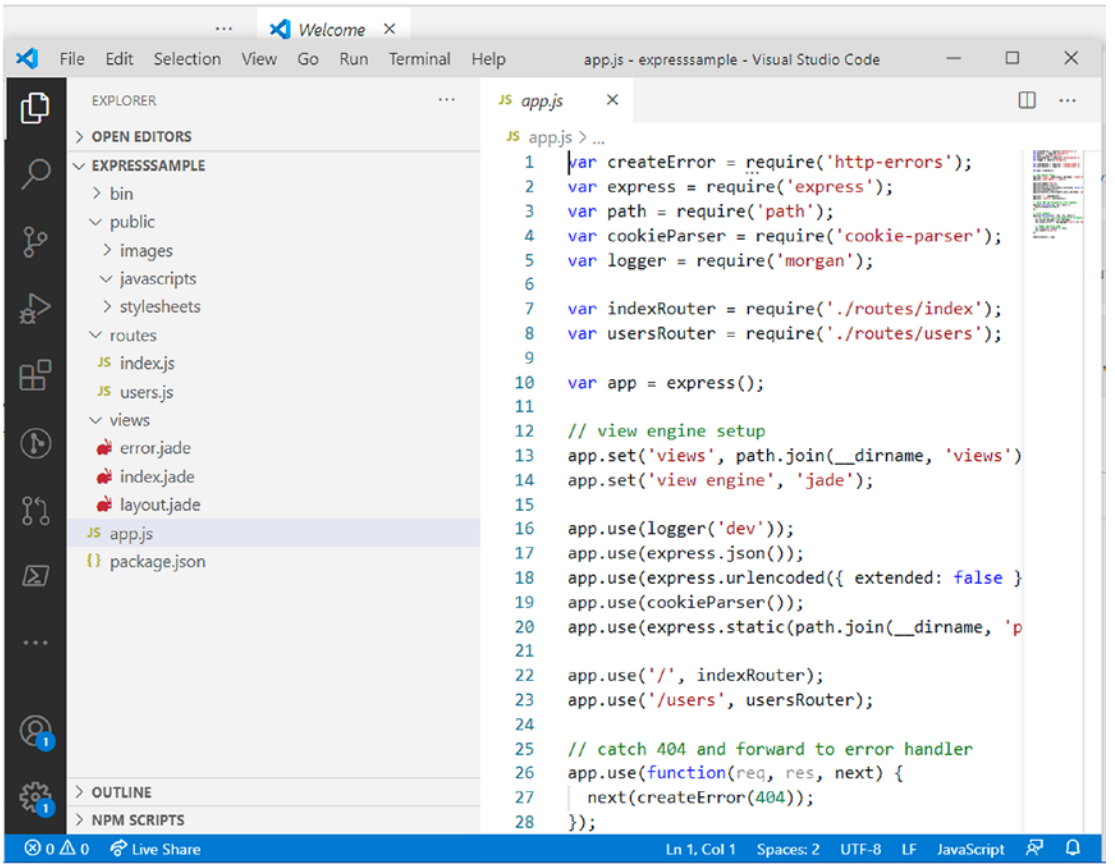

Figure 9-2. *A JavaScript project created with the Express JavaScript framework in VS Code*

You follow a similar process with other command-line tools that allow for generating projects, such as the Yeoman generator (`https://yeoman.io/`), 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:

```
> npm install -g cordova
```

Then you can easily build a Cordova project with the following command:

```
> 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 adds 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 also need some additional specific tools for Cordova, depending on what system you intend to target. For iOS, you 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 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 code debugging capability of Visual Studio Code is one of its most powerful features and probably the one that makes it a notch above 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 5 installed, the C# extension for Visual Studio Code detects the availability of a compatible debugger and takes care of attaching it to VS Code.

We will consider the scenario of using C# and .NET Core as the example of how debugging works, so reopen the HelloWeb folder that you 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 5.

The Run view provides a way to interact with the debugger. Figure 9-3 shows how it appears at this point.

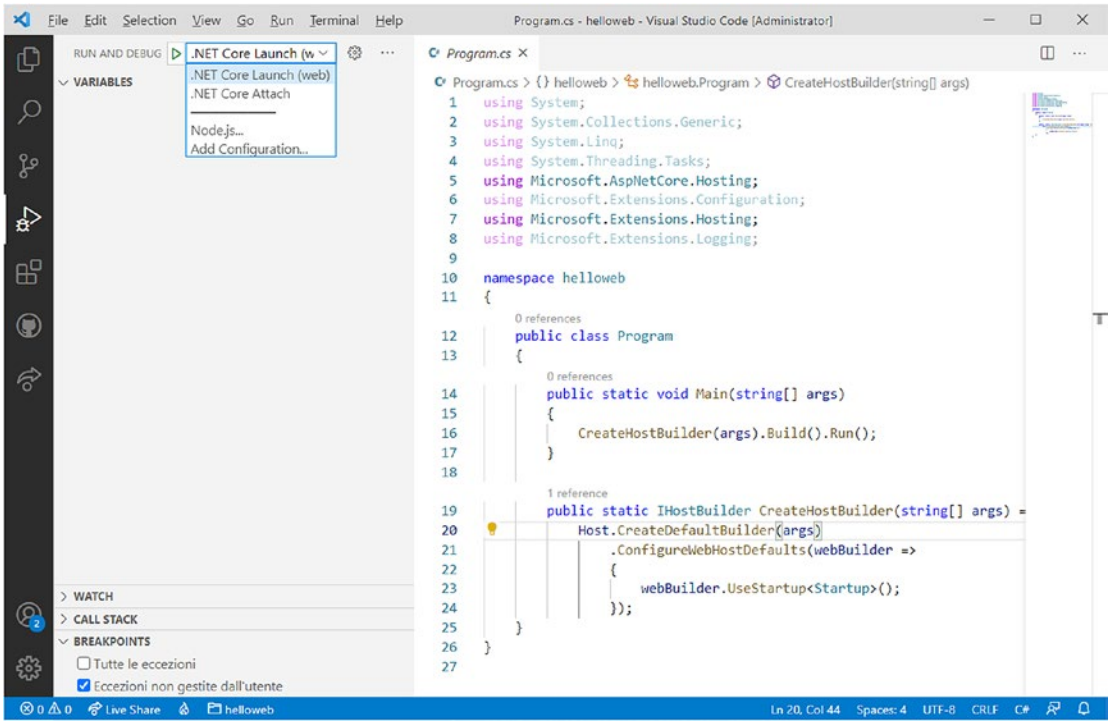

Figure 9-3. *Run view*

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

- The **Start Debugging** button, represented with the play icon (the white and green arrow). Clicking this button starts the application with an instance of the debugger attached.
- The configuration drop-down 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).
- A submenu represented by the **...** button that contains the list of available and selected views, plus the **Debug Console** command, which opens the Debug Console panel where you see the output messages from the debugger.

After this quick overview, you are ready to 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 5, where an extension takes care of everything, default configurations are provided. Figure 9-3 shows the 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 used to attach the debugger to another running .NET 5 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 in a special file called launch.json. Visual Studio Code stores this file in the .vscode subfolder (along with 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:

```
{
   "version": "0.2.0",
   "configurations": [
       {
           // 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
            "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": "${workspaceFolder}/bin/Debug/net5.0/HelloWeb.dll",
            "args": [],
            "cwd": "${workspaceFolder}",
            "stopAtEntry": false,
            // Enable launching a web browser when ASP.NET Core starts.
               For more information: https://aka.ms/VSCode-CS-LaunchJson-
               WebBrowser
            "serverReadyAction": {
                "action": "openExternally",
                "pattern": "\\\\bNow listening on:\\\\s+(https?://\\\\S+)"
            },
            "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.
- `env`, which represents the environment. In the case of .NET 5, 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 are not 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** button located at the bottom-right corner of the code editor when launch.json is the active file, 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 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 of by pressing F9 on the line of your interest. For instance, place a breakpoint on line 18 of the Startup.cs file, as shown in Figure 9-4.

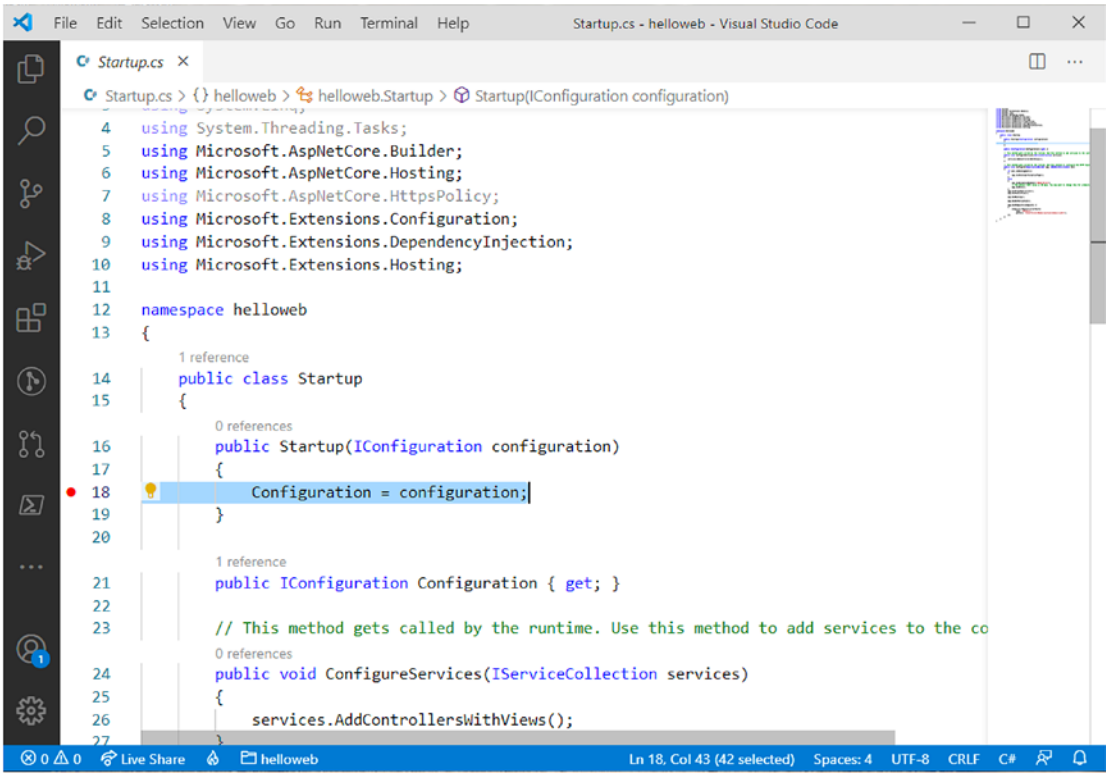

Figure 9-4. *Adding breakpoints*

You can remove a breakpoint by simply clicking it again, or you can manage breakpoints in the **Breakpoints** area of the Run view (see Figure 9-5).

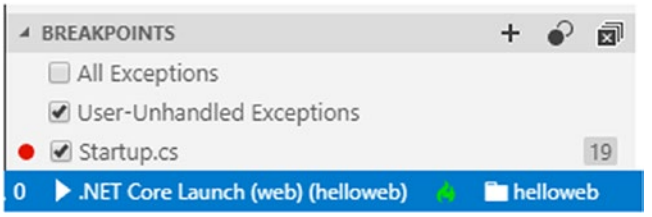

Figure 9-5. *Managing breakpoints*

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 userunhandled 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 can see in action all the debugging tools and make decisions when breakpoints are hit. In the Run view, make sure the **.NET Core Launch (web)** configuration is selected, then click the **Start** button or press **F5**. Visual Studio Code launches 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 9-6 shows VS Code hitting a breakpoint and all the debugging instrumentation. The line of code highlighted in yellow is the line that will be executed as the next one.

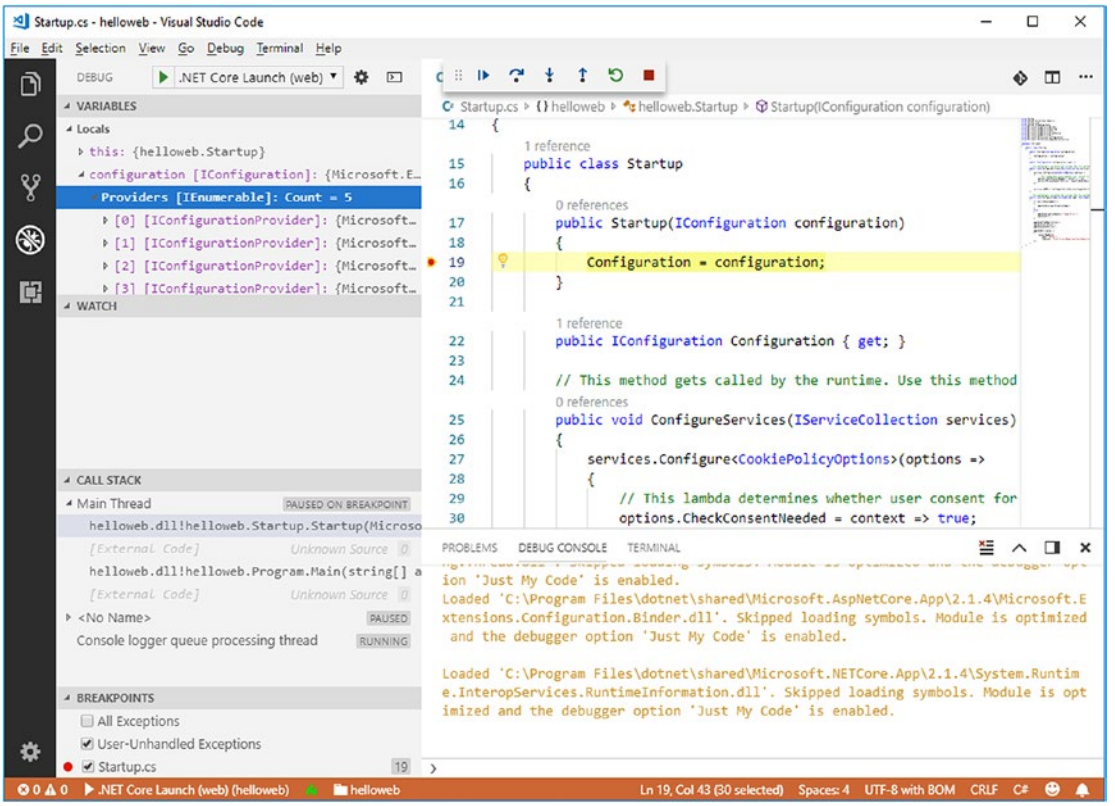

Figure 9-6. *The debugging tools available when a breakpoint is hit*

Notice that the status bar becomes orange while debugging and 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. This panel includes a sublist called Locals, which displays the list of the variables that are currently in scope. Each can be further expanded to see their details.
- **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 takes 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 9-6) called Debug action pane, which is composed 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. Statements within method bodies are also executed one at a time.
- **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 **Run** menu, together with their keyboard shortcuts. For example, if you click the **Step Over** button, the highlighted line runs and the execution advances one line (see Figure 9-7). If you hover your cursor over a variable name in the code editor, a convenient pop-up box enables you to easily investigate values and property values (depending on the type of the variable), as demonstrated in Figure 9-7, which shows a pop-up box that includes 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 Run and Debug bar.

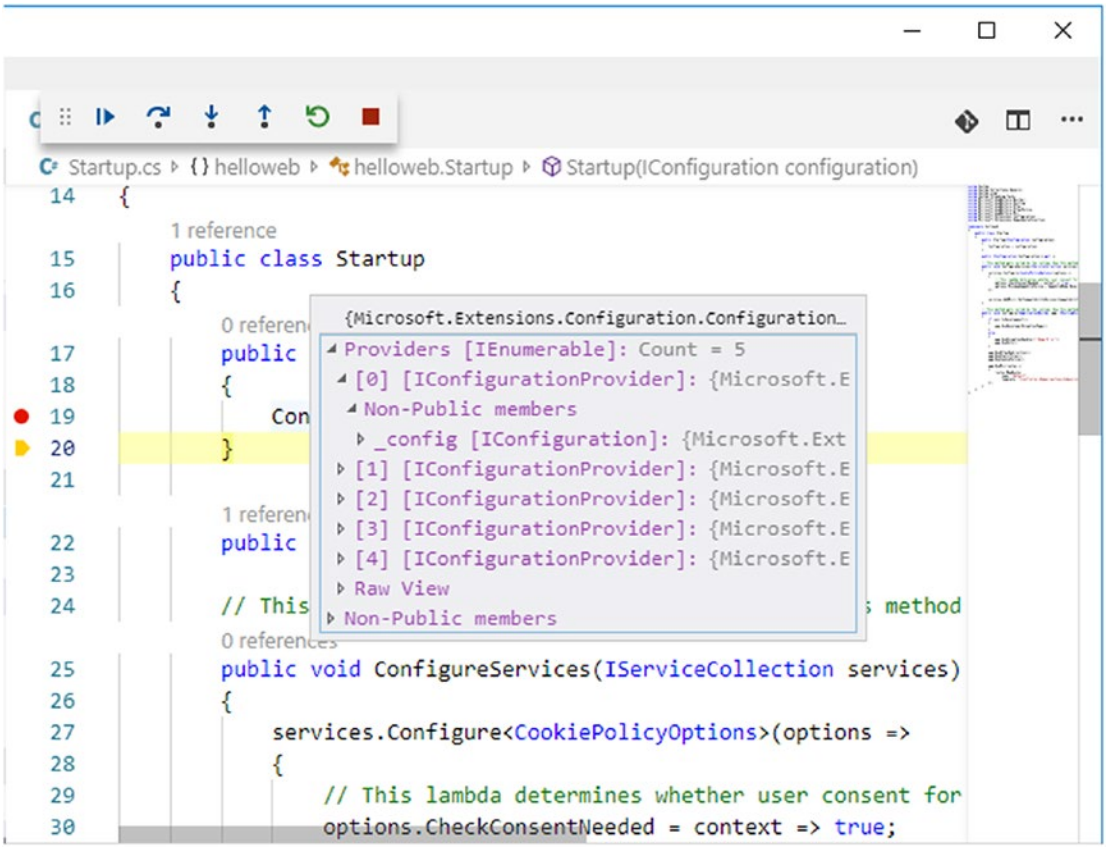

Figure 9-7. *Investigating property values at debugging time*

Evaluating Expressions

You have an option to use the **Watch** tool to evaluate expressions. 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 returns `true` or `false` depending on whether or not the object has an instance. Figure 9-8 shows an example.

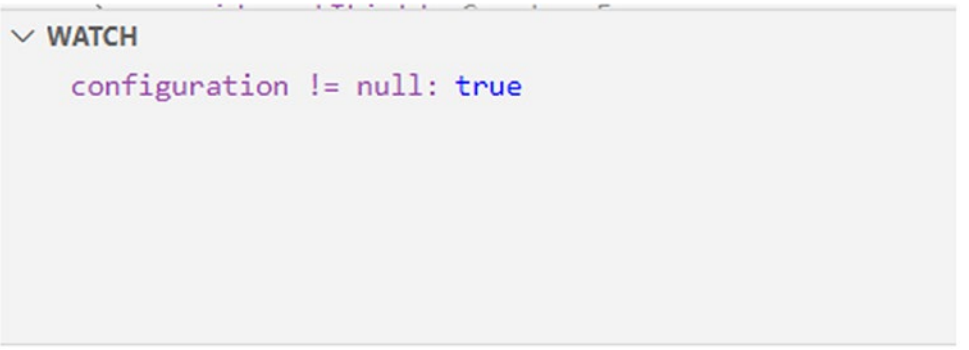

Figure 9-8. *Evaluating expressions*

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 opens the containing file, highlighting the method call (see Figure 9-9).

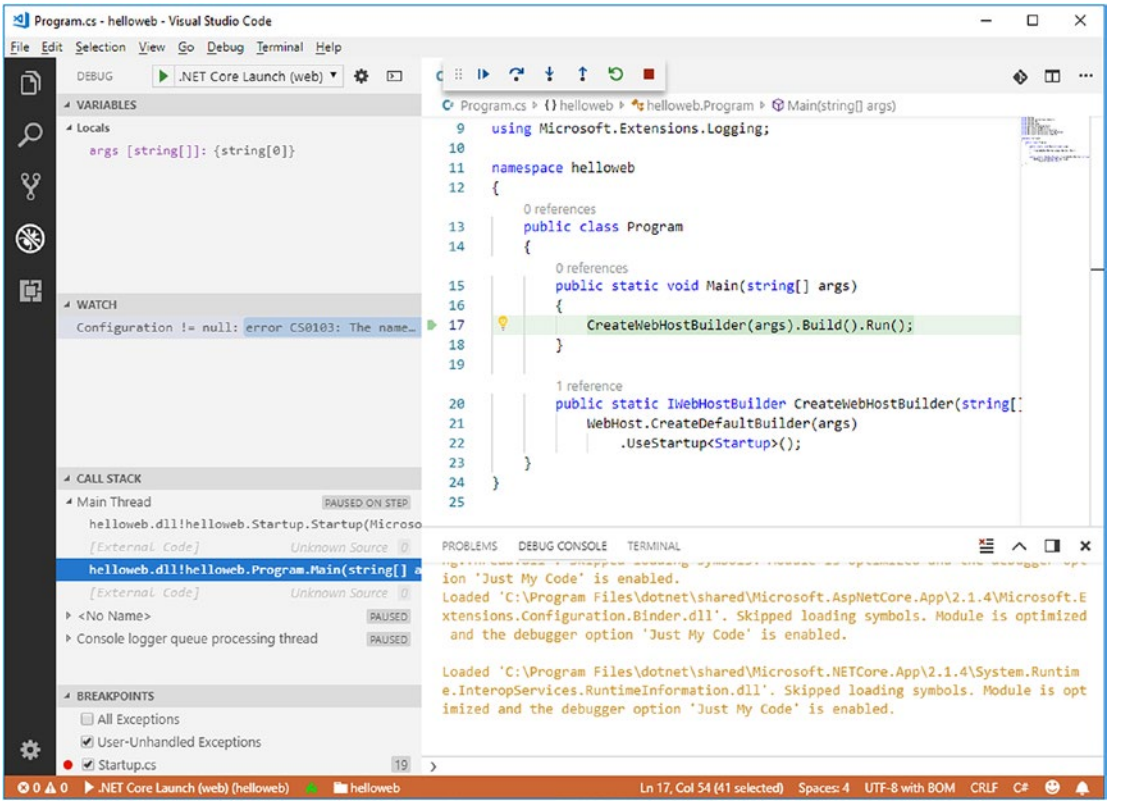

Figure 9-9. *Walking through method calls*

As you walk through method calls, the Locals subview of the VARIABLES panel also updates to show variables that are in the current scope. The code editor can highlight method calls only if the method is part of the source code, so it does not allow further control over the methods marked as [External Code] in the CALL STACK (see Figure 9-9), 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 9-10 shows an example that evaluates if the `configuration` variable is not null.

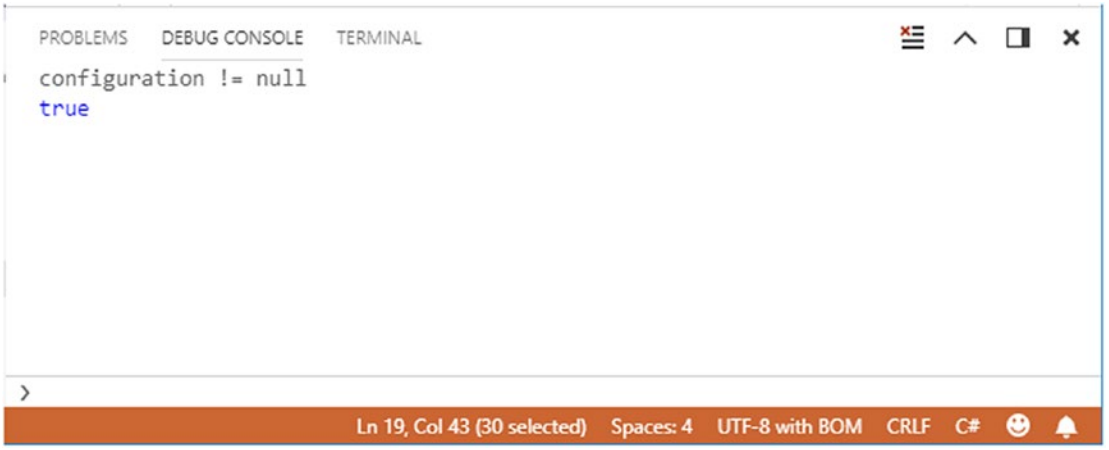

Figure 9-10. *Evaluating expressions in the Debug Console panel*

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 5 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.

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.

CHAPTER 10

Building Applications with Python

Python is a very popular and powerful programming language that can be used to develop applications of any kind, and it is especially useful to build data science and data analysis applications.

Python is an interpreted, object-oriented programming language that can be learned by developers of any experience. This chapter describes how Visual Studio Code supports building and debugging Python code, including specific code editing features. Obviously, the chapter's focus is not the Python language but rather how Python can find used with VS Code.

Chapter Prerequisites

In this chapter, I provide examples of running and debugging Python code. Following along with these examples requires that you install the following components before you continue reading:

- The Python interpreter with its tools, which you can download from the Python official site (`https://www.python.org/downloads`). The download page automatically detects your operating system and offers the appropriate download package for Windows, macOS, and Linux distributions.
- The Python extension for Visual Studio Code provided by Microsoft, which you can install via the Extensions panel. There are several extensions for Python in the Marketplace, but I recommend that you download the official one, shown in Figure 10-1, because it dramatically improves the development experience with a debugger and additional coding tools.

A. Del Sole, *Visual Studio Code Distilled*, https://doi.org/10.1007/978-1-4842-6901-5_10

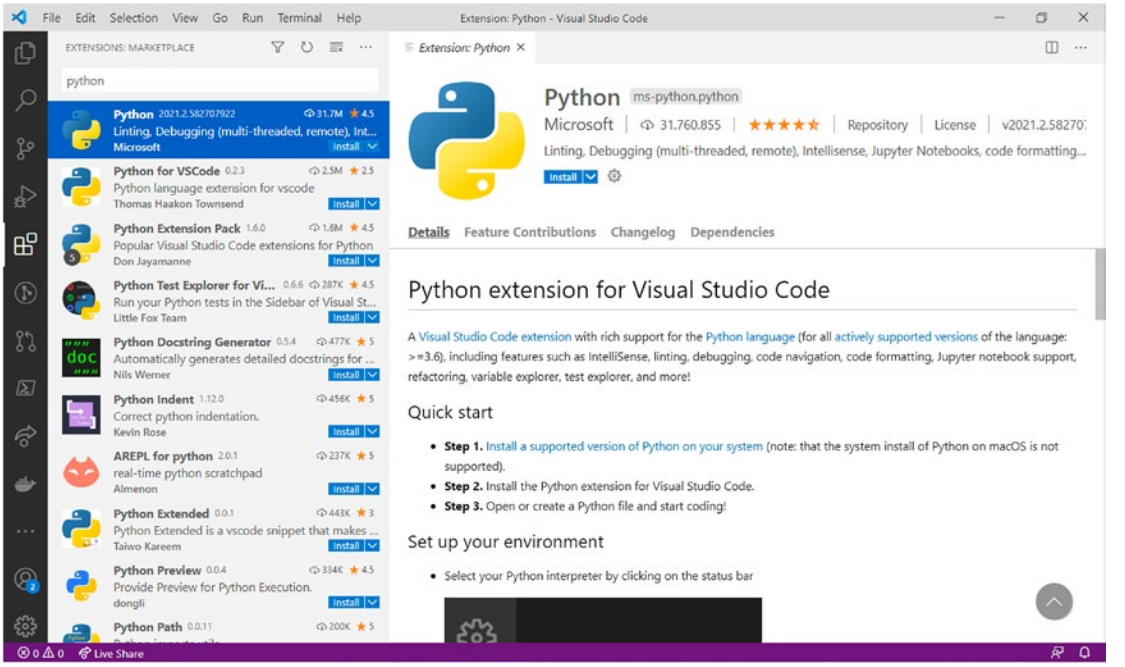

Figure 10-1. *The official Python extension from Microsoft*

Note This chapter walks through a simple code example, but in the real world you might want to build more complex applications, in which case you need additional components. For instance, building data science applications requires Anaconda (`https://www.anaconda.com`), a distribution that includes Python and the R programming languages, plus a set of libraries specific for data science. If you instead need to do web development, you might want to consider Django (`https://www.djangoproject.com`), a web framework built with Python.

If you haven't already created a dedicated folder on disk for the code examples (mine is called VSCode), as suggested in the previous chapters, I recommend doing so for this chapter.

Now that you have all the minimum required tools installed, you are ready to start coding and debugging with Python in Visual Studio Code.

Creating Python Applications

Previously in the book you learned that Visual Studio Code is independent from proprietary project systems and platforms and, consequently, does not offer any built-in options to create projects, and this is also true for the Python programming language.

What you can do with Visual Studio Code is open existing Python files and projects, or create new code files from within the development environment. As an example, let's consider a simple battleships game available in one code file at `pythonfiddle.com/battleships-game-in-python/`. In Visual Studio Code, create a new file and then select Python as the language from the well-known drop-down menu located in the bottom-right corner. The source code in its current state will not work with the latest versions of the Python interpreter, because it is missing parentheses enclosing parameters of the `print` function and some string-to-integer conversions. The modified and working code for Python is listed here for your convenience:

```
import random

board = []

for x in range(0,5):
  board.append(["O"] * 5)

def print_board(board):
  for row in board:
    print (" ".join(row))

print ("Let's play Battleship!")
print_board(board)

def random_row(board):
  return random.randint(0,len(board)-1)

def random_col(board):
  return random.randint(0,len(board[0])-1)

ship_row = random_row(board)
ship_col = random_col(board)
print (ship_row)
print (ship_col)
```

```
for turn in range(4):
        guess_row =  int(input("Guess Row:"))
        guess_col = int(input("Guess Col:"))

        if guess_row == ship_row and guess_col == ship_col:
          print ("Congratulations! You sunk my battleship!")
          break
        else:
            if turn == 3:
                    board[guess_row][guess_col] = "X"
                    print_board(board)
                    print ("Game Over")
                    print ("My ship was here:
                    [" + str(ship_row) + "][" + str(ship_col)
                    + "]")
            else:
                    if (guess_row < 0 or guess_row > 4) or
                       (guess_col < 0 or guess_col > 4):
                        print ("Oops, that's not even in the ocean.")
                    elif(board[guess_row][guess_col] == "X"):
                        print
                              ("You guessed that one already.")
                    else:
                        print ("You missed my battleship!")
                        board[guess_row][guess_col] = "X"
                    print (turn + 1)
                    print_board(board)
```

Save the file as BattleshipsGame.py. This is a simplified implementation of the battleships game and is mostly for learning purposes, but it is enough to understand how Visual Studio Code can support Python development. You will immediately notice powerful editing features as you type the source code, such as (but not limited to) IntelliSense and parameter hinting, but before highlighting Python-specific editing features, I will walk you through running and debugging Python code.

Running Python Code

Visual Studio Code automatically attempts to retrieve an appropriate Python interpreter on your machine when you assign this language to a code file or open an existing file. Sometimes VS Code might not be able to do this even if you previously installed a Python interpreter successfully, in which case you receive a warning similar to the one shown in Figure 10-2.

Figure 10-2. *Visual Studio Code could not find a Python interpreter*

Clicking the **Select Python Interpreter** button in the warning card or the same-named item at the bottom-left corner of the Status Bar enables you to pick your favorite version of the Python interpreter (see Figure 10-3).

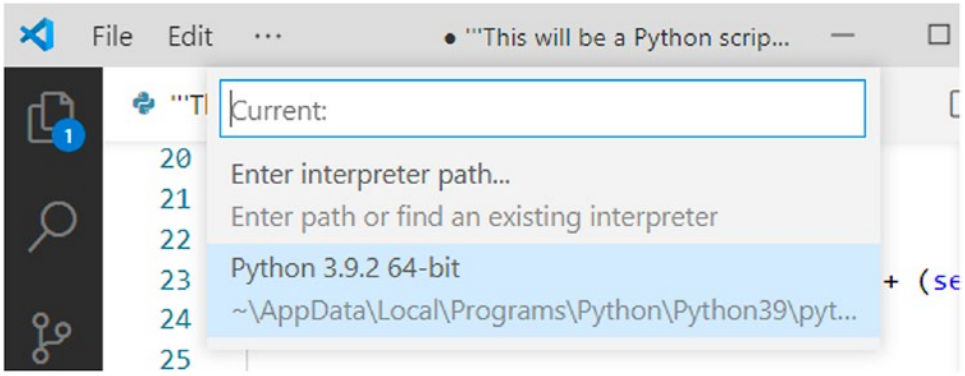

Figure 10-3. *Selecting a version of the Python interpreter*

This is a very nice option in case you need to select a specific version and not necessarily the most recent one. Once you have selected a Python interpreter, the name appears on the Status Bar, replacing the Select Python Interpreter button, and you can either run or debug your code. Let's start with running code, which you can do by

selecting **Run ➤ Run Without Debugging**. The Python runtime builds the code file and, if no error is found, the output of the code is displayed in an instance of the Terminal panel. Figure 10-4 shows an example based on the sample game provided previously.

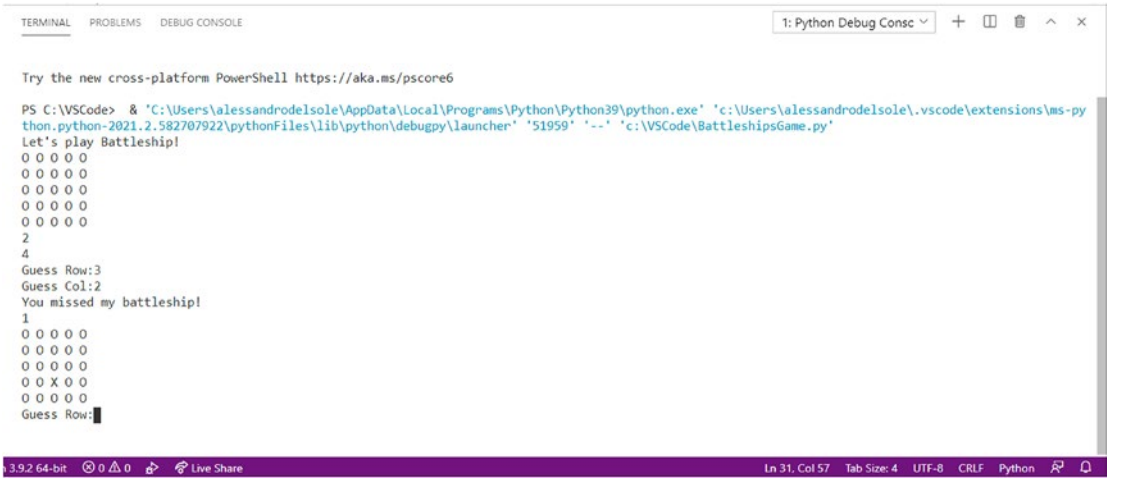

Figure 10-4. *Output of Python code in the Terminal*

The Terminal allows user input, so you will be able to enter the values for the battleships. Behind the scenes, Visual Studio Code invokes a tool called Launcher, which is installed together with the Python interpreter and makes it possible to run Python code from the command line.

Note In more specific development scenarios based on the Anaconda libraries, such as data science, Visual Studio Code is able to display additional tool windows and show charts and calculation results inside the development environment. More details are available in the official Data Science Tutorial (`code.visualstudio.com/docs/python/data-science-tutorial`).

For the next example, make sure you add a breakpoint at line 30 (as described in Chapter 9). This is to demonstrate how debugging tools for Python work. You start debugging Python code by pressing F5, by clicking the **Run and Debug** button in the Run panel, or by selecting **Run ➤ Start Debugging**. At this point Visual Studio Code asks you what file or program you want to debug, as shown in Figure 10-5.

Figure 10-5. *Selecting the debugging target*

You can select any one of the configurations, which are provided by the Python extension for VS Code, described in Table 10-1.

Table 10-1. *Debug Configurations for Python*

Configuration Name	VS Code Description	Description
Python File	Debug the currently active Python file	Starts debugging the currently active Python file, where "active" means the file in the active editor.
Module	Debug a Python module by invoking it with `-m`	A Python module can be considered as a code library, comparable to namespaces in a C# library. Debugging with the `-m` switch enables VS Code to also debug a module.
Remote Attach	Attach to a remote debug server	Allows connecting VS Code to a remote debug service.
Attach using Process ID	Attach to a local process	Allows connecting the debugger to a process that is already running. You need to retrieve the process ID (e.g., on Windows you can do so via the Task Manager).

(*continued*)

Table 10-1. (*continued*)

Configuration Name	VS Code Description	Description
Django	Launch and debug a Django web application	Django is a high-level Python web framework that enables rapid development of secure and maintainable websites. With this option, you can debug a Django project in VS Code.
FastAPI	Launch and debug a FastAPI web application	FastAPI is a modern web framework for building APIs with Python (requires version 3.6 or higher). With this configuration, you can use VS Code to debug a FastAPI project.
Flask	Launch and debug a Flask web application	Flask is another framework that allows building web applications with Python. With this configuration, VS Code makes it possible to debug Flask projects.
Pyramid	Launch and debug a Pyramid web application	Pyramid is a framework for Python that allows for creating web applications based on the Model-View-Controller (MVC) pattern. With this configuration, you can debug a Pyramid project in VS Code.

For the current example, select the first option, **Python File**, which allows for debugging the current code file. The application starts in the integrated Terminal and VS Code's Status Bar becomes orange, which indicates that the application is in debug mode. In the Terminal you will be able to enter the values for the battleships game, and then, because you previously set a breakpoint, the execution will break at line 30. This will enable all the toolboxes in the Run panel as well as data tips in the code editor (see Figure 10-6).

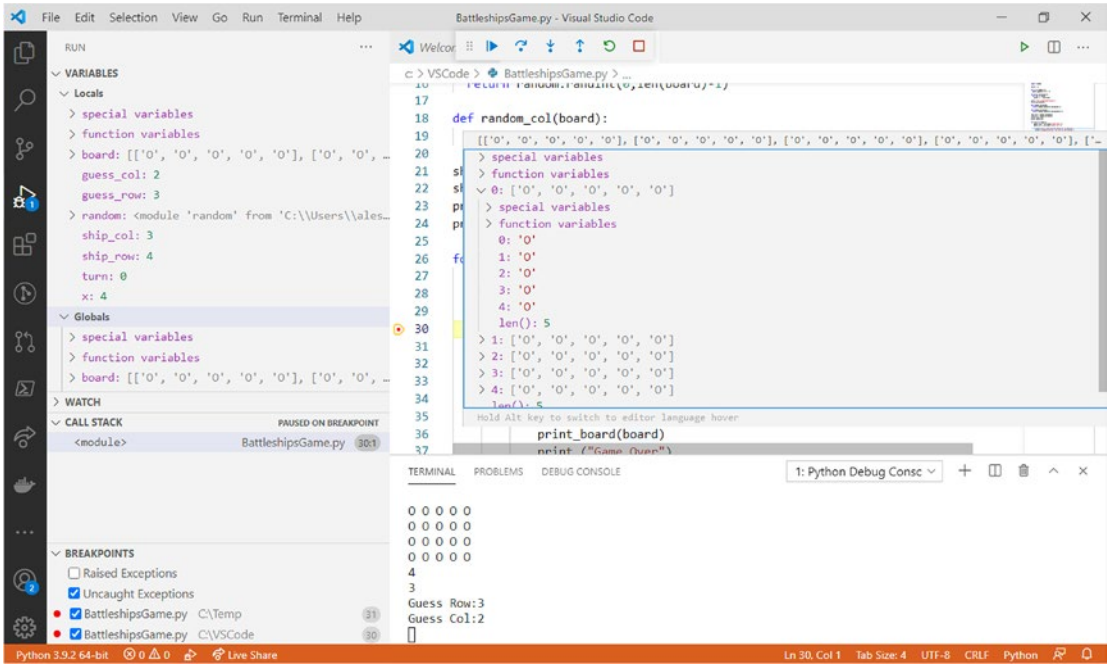

Figure 10-6. *The application in debug mode and debugging tools enabled*

If you hover your cursor over a variable name in the code editor, you will be able to see its current value. For instance, if you hover over the `guess_col` variable, you will see that it contains the integer value you entered during the execution. However, Python debugging tools offer more: if you hover over a complex type like the `board` variable, which is a `list` of arrays, you will see how a sophisticated data tip shows values for each array in the list. You can expand the **Special variables** and **Function variables** groups to get more information about runtime functions.

The values you see through data tips are also visible in the **Locals** group of the VARIABLES tool in the Run panel. Debugging tools for Python are also able to catch runtime exceptions and to display appropriate information to solve them. To understand how this works, you can intentionally introduce a runtime exception in the current sample file. Consider line 27, which looks like the following:

```
guess_row = int(input("Guess Row:"))
```

Change the line as follows:

```
guess_row = input("Guess Row:")
```

This particular line will still work, because it still waits for the user to enter something from the keyboard; the difference from the original line is simply that the input, of type `str`, is not converted into an `int`. However, while comparisons with the equality operator will succeed, comparisons made with the < and > operators at line 40 will fail, because this line attempts to compare the user input, which is now a string, with an integer value, and such a comparison is not supported, so a runtime exception will happen. Figure 10-7 shows how Visual Studio Code breaks the application execution when it encounters a runtime exception.

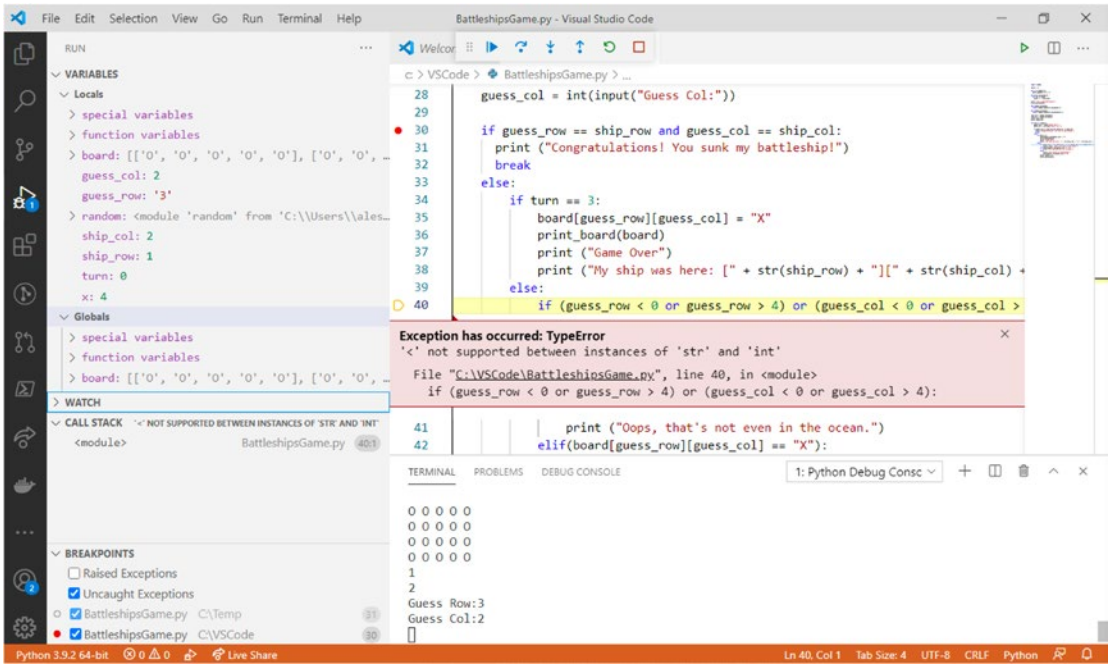

Figure 10-7. *Debugging runtime exceptions in Python*

More specifically, the exception information is displayed in a different-colored tooltip that is displayed right below the line of code that caused the error. In this tooltip, you can see the exception type (`TypeError` in this case), the number and content of the line of code, and the full error message. Actually, the tooltip also displays the name of the file that caused the exception in the form of a hyperlink. This is very useful when the exception was raised by a different file in the execution hierarchy, enabling you to quickly go to the problem by clicking the file name.

As you have seen, debugging Python code in Visual Studio Code is a rich experience, but actually the Python extension offers even more functionality, such as a dedicated language service and additional features, discussed in the next section.

Code Editing Features for Python

The Python extension for Visual Studio Code brings most of the powerful code editing features described in Chapter 3 to Python files. This section walks you through the evolved code editing features, describing how these can improve your productivity.

Enhanced Word Completion with IntelliSense

Probably the most productive code editing feature with any language is an enhanced word completion engine, and VS Code brings IntelliSense to Python. IntelliSense shows up as you type and displays documentation tooltips about the type or member currently selected in the IntelliSense pop-up box. Figure 10-8 shows an example based on the `bin` function, where you can also see how syntax colorization is available in the tooltip to provide easier understanding of the method usage.

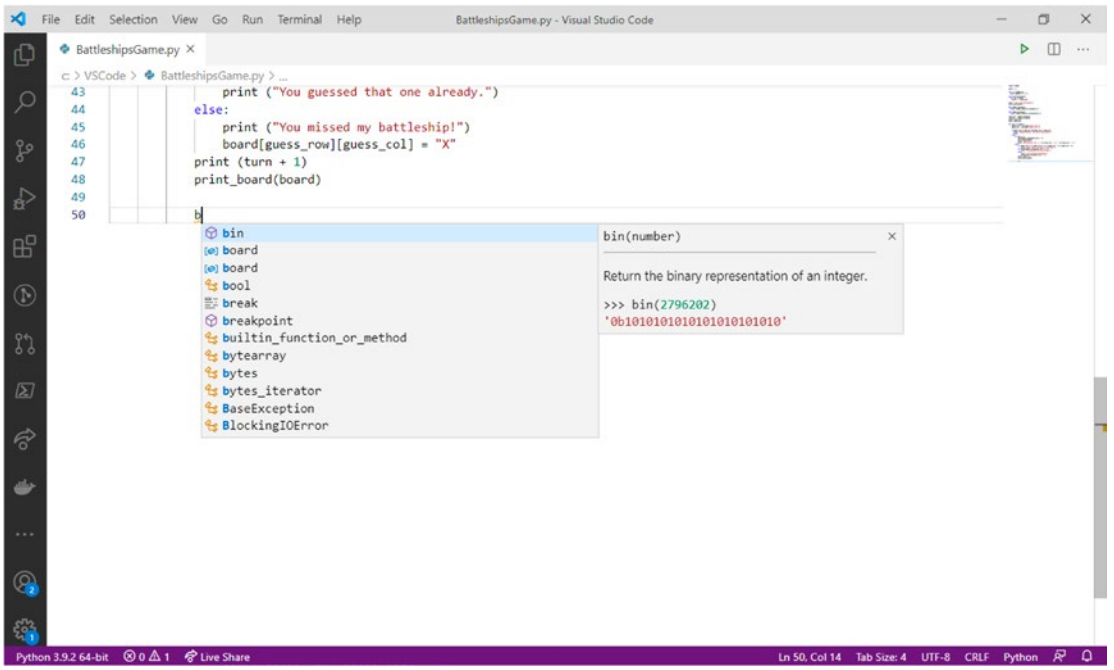

Figure 10-8. *IntelliSense in action with Python*

Understanding Function Parameters With Parameter Hints

Connected to IntelliSense is Parameter Hints. When you type the name of a function, you get suggestions on how to provide parameters, as demonstrated in Figure 10-9, which is based on the `pow` function.

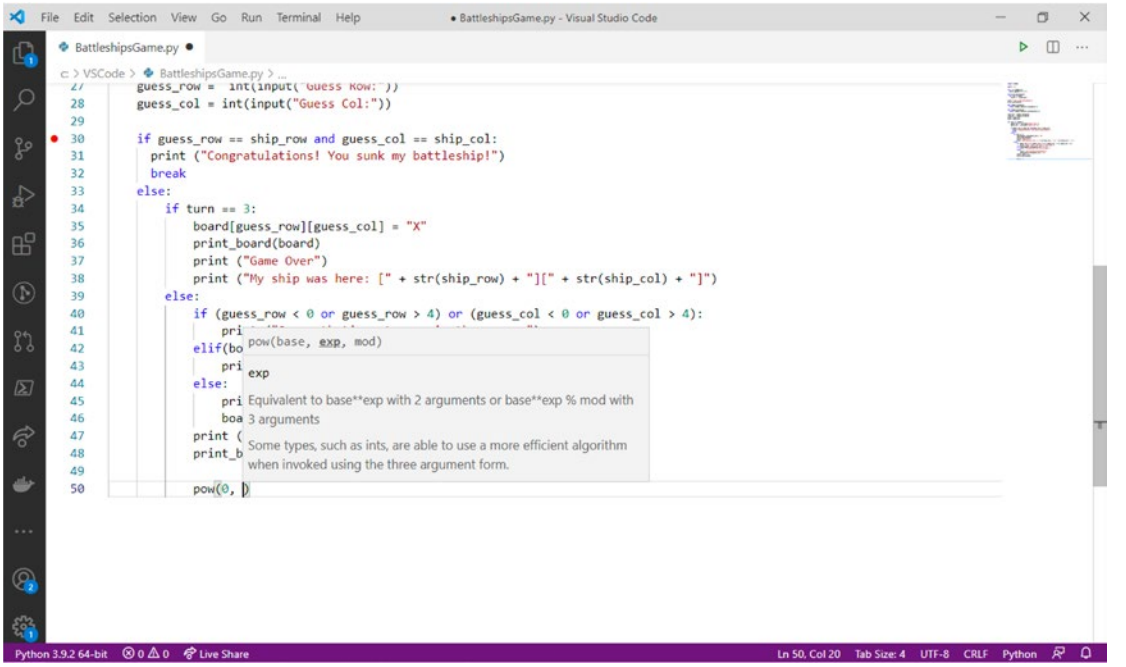

Figure 10-9. *Parameter Hints explains how to provide function parameters*

As you can see, the parameter you are currently supplying is highlighted in bold and underlined, while a description of the parameter itself is provided as the text content of the tooltip.

Quickly Retrieving Type Definitions

Among the code editor productivity features, Go to Definition and Peek Definition (see Chapter 3) are certainly very useful and popular, and these are also available to Python code files. To understand how they work in Python, right-click the `board` parameter of the `print_board` statement in the last line of the code file.

If you click **Go to Definition**, the cursor moves to the place where the `board` variable is declared. If you instead select **Peek** and then **Peek Definition**, the definition is shown inside an interactive pop-up box, where you can make your edits directly (see Figure 10-10).

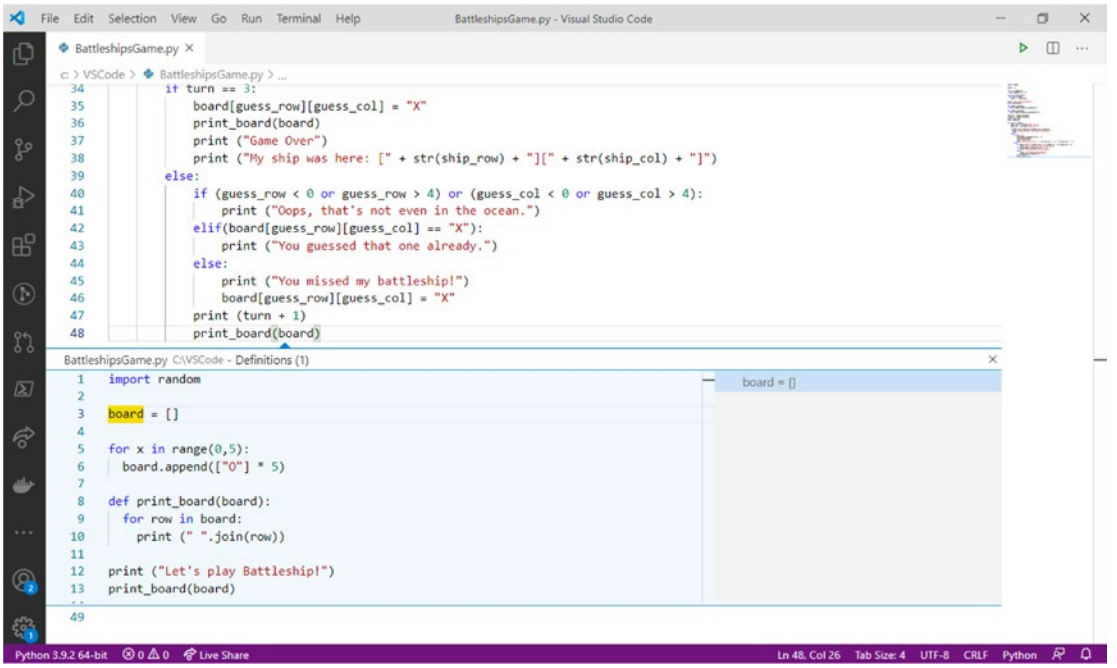

Figure 10-10. *Peeking type definitions*

Finding References

As explained in Chapter 3 and exactly like for other languages such as C#, you can quickly search for all references of a given type, member, or variable in Python. Simply right-click the object of your choice in the code editor and select **Find All References**. For instance, you can do this with the `board` variable in the sample code file and you will see where it was used across the code via the already well-known interactive editor, which highlights occurrences and shows a list of references on the right side of the panel. Figure 10-11 demonstrates this.

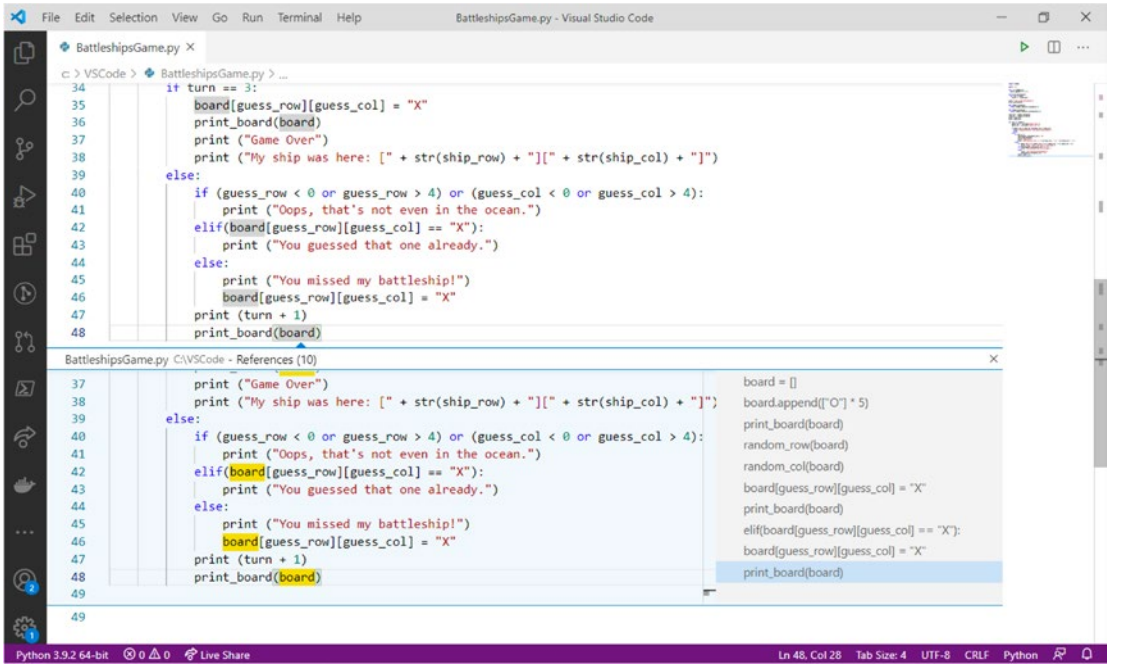

Figure 10-11. *Finding object references*

Note The Find All References user interface is basically an extended version of Peek Definition. The latter shows an individual reference of an object, which represents the place where it was defined. Find All References shows instead all the type or member references.

Renaming Symbols

With the Python extension, renaming symbols is an easy task. You can just right-click a symbol, select **Rename Symbol** (or press F2), and provide the new name, and all the occurrences in the source code will be renamed accordingly. When typing the new name, you can also press Shift+Enter and see a preview of all the occurrences that will be renamed.

Figure 10-12 shows an example based on the `board` variable, with the preview enabled.

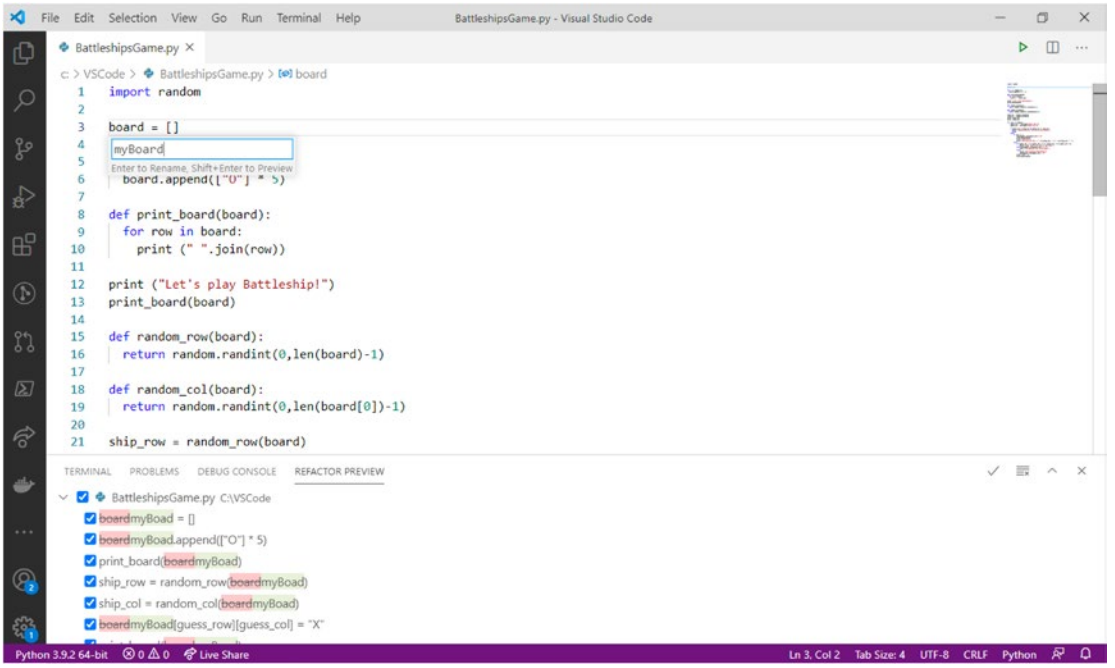

Figure 10-12. *Renaming symbols*

If you enabled the Refactor Preview panel, you need to click the tick icon in order to accept your changes. If you instead entered a new name without looking at the preview, simply press Enter and all the occurrences of (including references to) the symbol will be renamed.

Finding Code Issues with Linters

Linters highlight syntactical and stylistic problems in your code. Just as an example, linters highlight missing brackets or parentheses in a code block or highlight the usage of an undefined variable, underlining the code with squiggles. Linting is not enabled by default, but you can quickly do this via the Command Palette. You can type **Python Select Linter** directly, or just **Python** and then pick the appropriate command. Figure 10-13 shows how to enable linting with the list of commands filtered as I was typing.

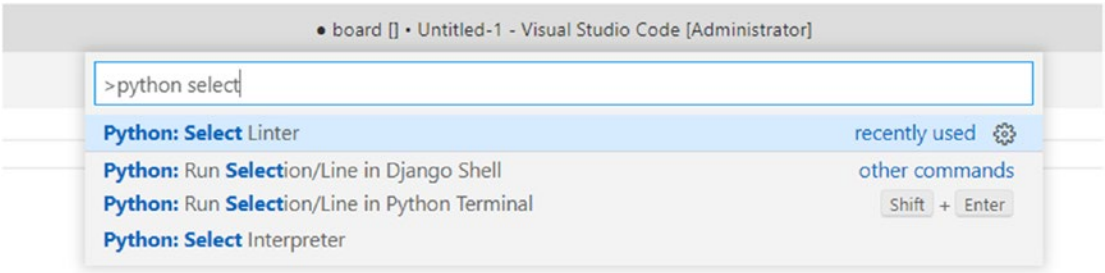

Figure 10-13. *Enabling Python linters*

When you select this command, the Command Palette also displays a list of available linters for Python. This is actually up to your choice, but I would suggest to use **pylint**, which is the official Microsoft linter provided via the Python extension. When the linter is enabled, the code editor displays squiggles under code that has issues, and these code issues are also detailed in the Problems panel, as shown in Figure 10-14.

Note If you have experience with C# in Visual Studio Code, you might expect the same behavior of live code analysis as you type, but, with Python, linters are able to show squiggles under code that has issues only after saving a code file or by explicitly invoking the linter from the Command Palette. An enhancement to this is provided by the Pylance extension, described shortly.

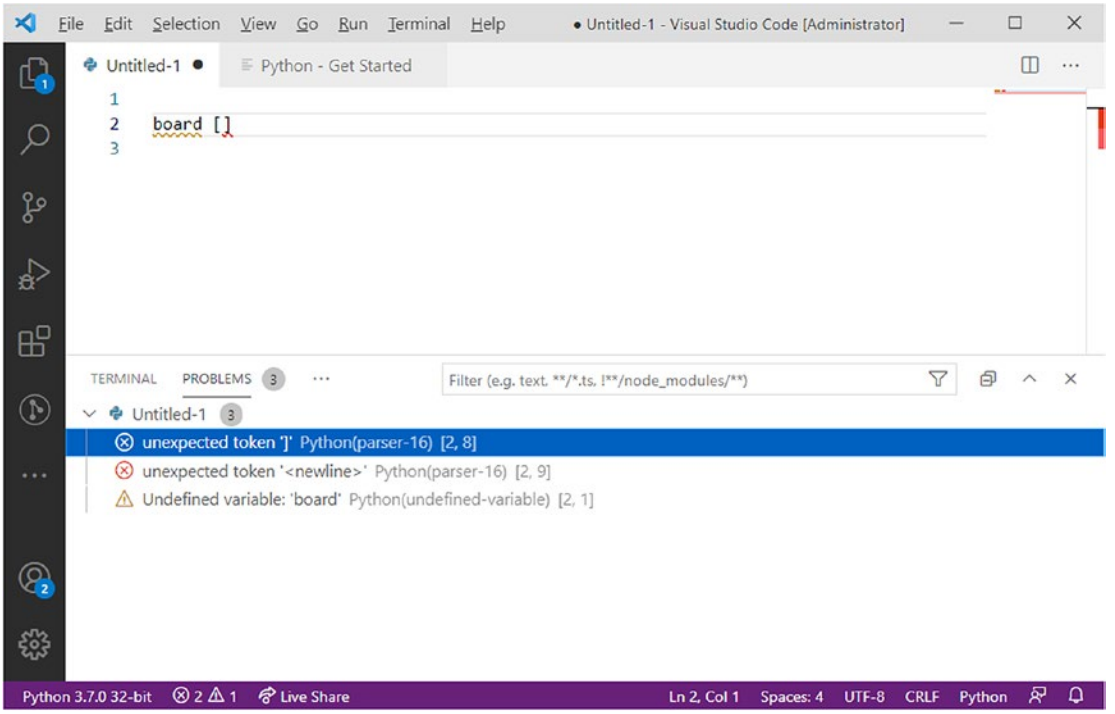

Figure 10-14. *Linters highlight code issues in the editor and in the Problems panel*

Note Linters, as well as the other editing features, can be further customized with the Settings user interface and via the Settings.json file. Because the goal of this book is to provide guidance on the most effective ways to get productive quickly, I am showing the fastest configuration options available with a few mouse clicks. If you want to dig deep into setting customizations, bookmark the related documentation at `https://code.visualstudio.com/docs/python/linting`, where you will also find more details about the pylint linter and summary information about the other linters listed in the Command Palette.

Advanced Code Editing with Pylance

Without a doubt, the Python extension for Visual Studio Code tremendously improves developer productivity and the coding experience, but Microsoft is doing even more. In fact, Microsoft is offering a new extension called Pylance, currently in preview, which

introduces code refactorings, IntelliCode (an evolved code completion engine powered by artificial intelligence), and other improvements.

When you open (or create) a Python code file, Visual Studio Code shows a pop-up box that offers to install Pylance, as shown in Figure 10-15. As an alternative, you can download the Pylance extension from the Extensions tool directly (see Figure 10-16).

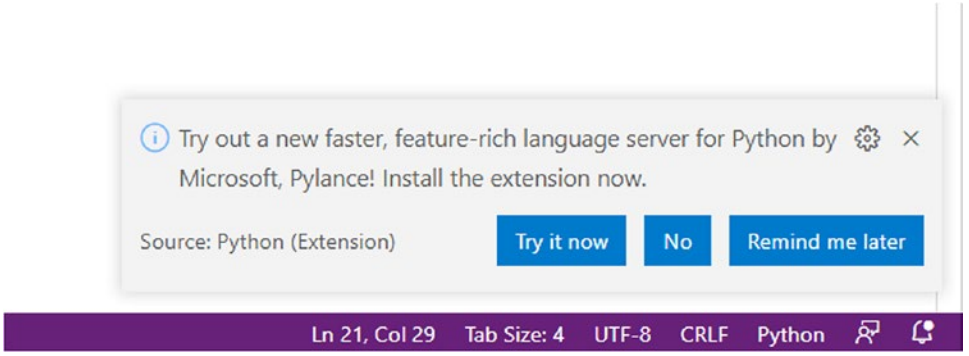

Figure 10-15. *Visual Studio Code offering to install the Pylance extension*

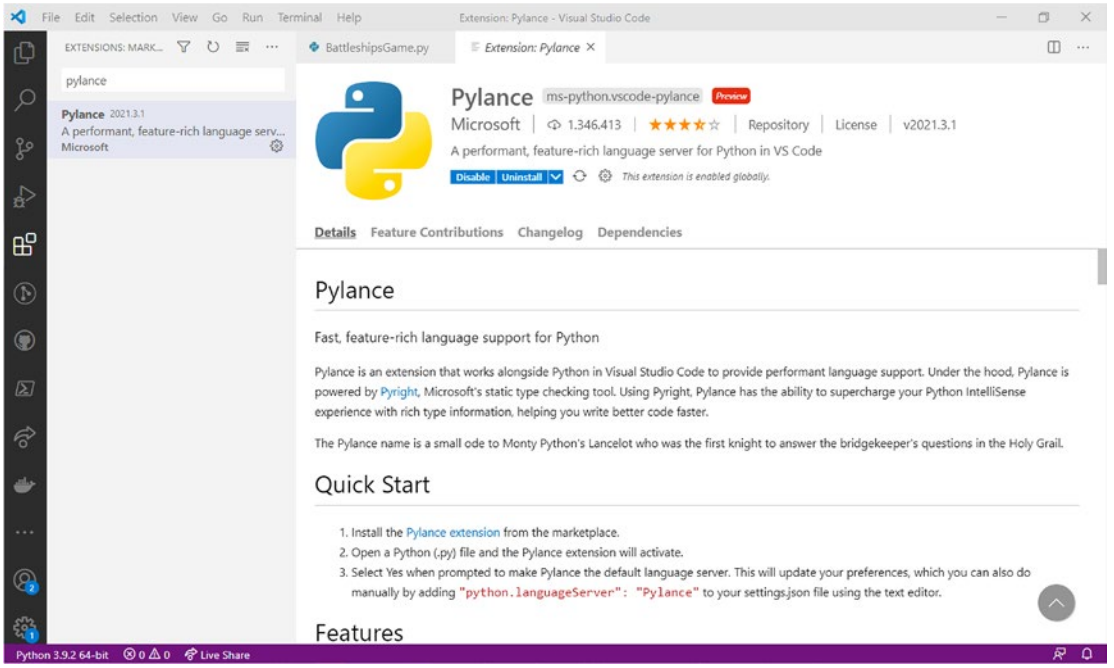

Figure 10-16. *The Pylance extension details*

Once Pylance has been installed, IntelliSense will be powered by IntelliCode. This tool learns from your code and from your patterns and offers an improved editing experience based on your coding styles, enabling IntelliSense to provide even better suggestions based on the coding context.

Pylance is not limited to offering an improved IntelliSense engine, but it makes it easier to write better code with new code refactorings and live code analysis. For instance, Pylance enables linters to show error squiggles as you type. As another example, whereas the Python extension, by default, only allows sorting `import` directives, Pylance introduces new refactorings: **Extract method**, **Extract variable** and automatic addition of the required `import` directives when adding code via IntelliSense or code snippets. For a better understanding of how this works, select the code block from line 5 to line 13 of the sample file, as shown in Figure 10-17. You will see a light bulb icon appear, which means that there are some suggestions to refactor the selected code block.

Figure 10-17. *Enabling suggestions for code fixes*

If you hover your cursor over the light bulb icon, you will see a tooltip saying **Show fixes**. Click it to see available suggestions for the current context; in this case there is one suggestion, **Extract method**. Click this suggestion and VS Code will extract a new method for the selected block, adding the related method call. This is demonstrated in Figure 10-18.

Figure 10-18. *Extracting a method*

You need to manually rename the new method, because Pylance provides a default name and does not enter in rename mode. Similarly, the code fix called **Extract variable** enables you to extract a variable from a code block, and it is available through the light bulb icon only if the context of the code allows for extracting variables. The light bulb icon is not the only shortcut to retrieve code fixes for a code block; you can also select a code block, right-click, and then select **Refactor** from the context menu.

Managing Pylance Settings

As I mentioned previously, at this writing Pylance is in a preview state, but you can have a look at what Microsoft is working on by enabling the Insiders Channel for the extension updates. You can do so in the VS Code's Settings (see Figure 10-19) by changing to **daily** the value for the **Pylance: Insiders Channel** option.

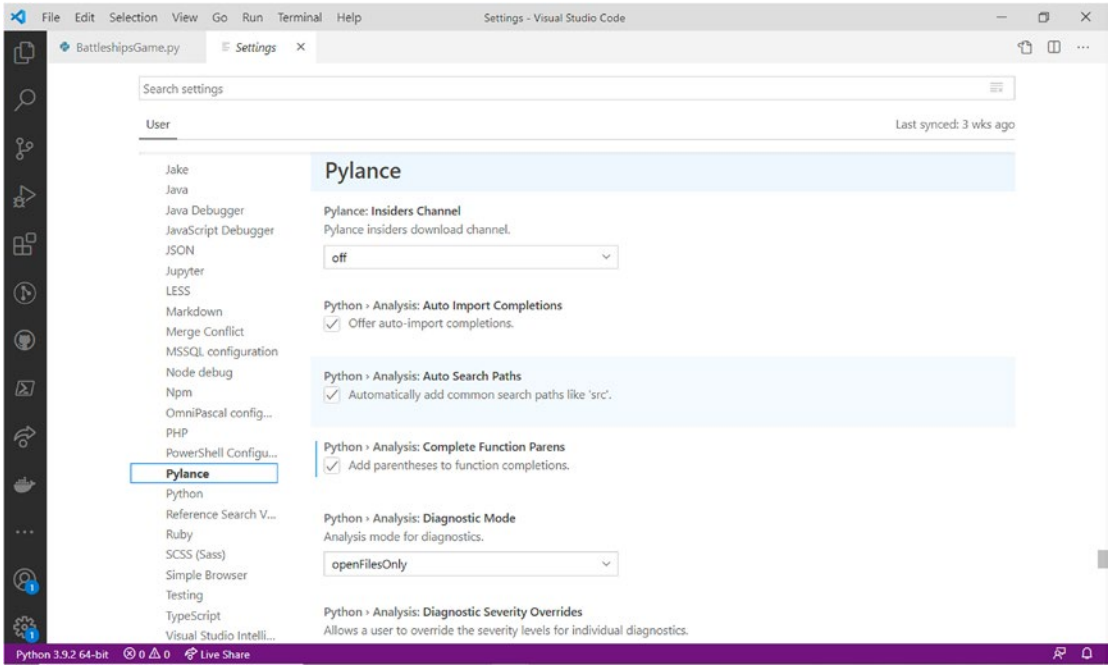

Figure 10-19. *Changing Pylance settings*

It is reasonable to expect more additions and improvements to Pylance once it reaches a production milestone.

Running Python Scripts

Python is also an interpreted language, so it allows for running arbitrary code without the need of a backing build process. Visual Studio Code supports Python as an interpreter, providing an option to write and run code via an REPL (read-eval-print-loop) interactive console, available within the Terminal.

You enable the Python REPL in the Command Palette by selecting the **Python: Start REPL command** (see Figure 10-20).

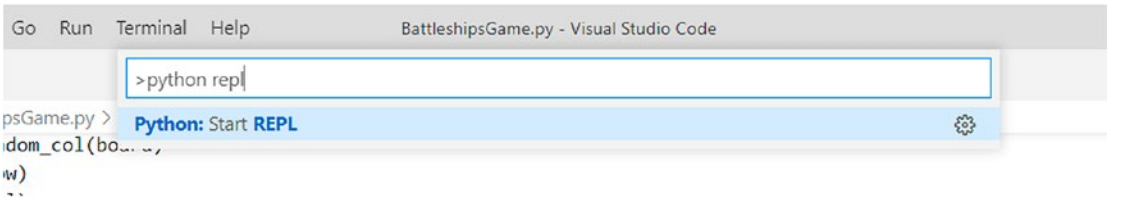

Figure 10-20. *Enabling the Python REPL console*

At this point the Terminal appears and loads the Python REPL, where you will be able to write and run arbitrary code. Figure 10-21 shows an example based on declaring a variable and printing its content onscreen.

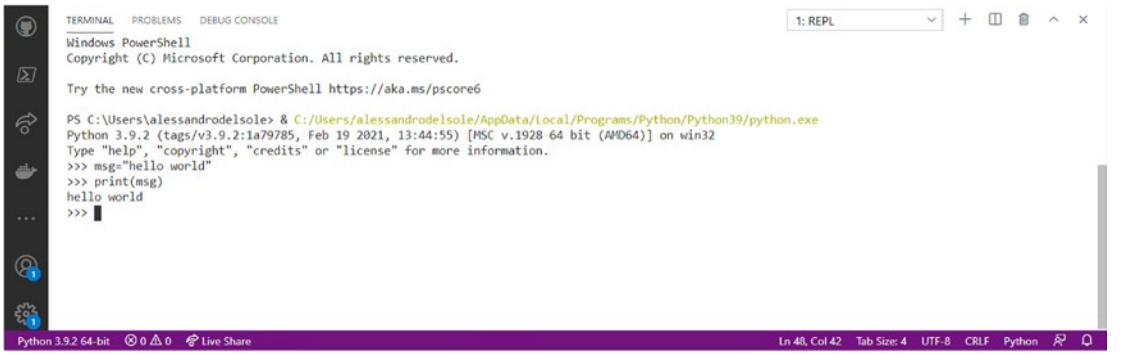

Figure 10-21. *Running arbitrary code in the Python REPL console*

This is another important tool for Python developers, because it is a very common way to use this language and certainly a way that leverages one of the most powerful of its characteristics.

Summary

Python is a very popular and powerful programming language which is fully supported by Visual Studio Code. It offers full support for evolved code editing, debugging, and even for advanced development with data science tools and libraries.

Visual Studio Code enhances support for Python with the official Python extension, which makes working with Python very similar to working with other languages and platforms, so you can apply existing skills and knowledge if you are approaching Python for the first time but have existing experience with C# or Node.js.

Microsoft is also investing in a new extension called Pylance, which provides an improved IntelliSense experience with IntelliCode and additional code refactorings. An interactive REPL for interpreted code completes the integrated tooling for Python.

Once again, Visual Studio Code demonstrates how versatile it is, providing a perfect environment for Python and its most popular flavors.

CHAPTER 11

Deploying Applications to Azure

Microsoft Azure is Microsoft's premiere cloud solution that offers many services, from hosting web applications and SQL databases to remote virtual machines, artificial intelligence services, and many more.

With Visual Studio Code, it is easy to deploy your code to Azure through a number of extensions that support multiple environments, such as Node.js and .NET, and that offer an integrated experience so that you can work directly within your development environment. Many extensions for Azure development are available, each targeting different scenarios, but it would require an entire book to describe them all, so in this chapter I will cover two of the most popular extensions: Azure App Services, which supports publishing web applications, and Azure Functions, which enables you to work with serverless apps directly from Visual Studio Code.

Note This chapter requires an active Microsoft Azure subscription to complete the examples. If you do not have one, you can get a free trial at `https://azure.microsoft.com/en-us/free`.

Introducing Azure Extensions

Visual Studio Code supports developing with the most popular and powerful Azure services. Support is integrated in the development environment with specific extensions available in the Visual Studio Marketplace. Table 11-1 lists and describes common extensions for Azure development.

A. Del Sole, *Visual Studio Code Distilled*, https://doi.org/10.1007/978-1-4842-6901-5_11

Table 11-1. *Common Extensions for Azure Development*

Extension	Description
Azure Account	Allows signing into one or more Azure subscriptions.
Azure App Service	Provides integrated support to deploy web applications to the cloud.
Azure CLI Tools	Installs all the command-line tools required to work with all the Azure services.
Azure Databases	Allows for creating, browsing, and managing SQL Azure, MongoDB, Cosmos DB, PostgreSQL, and DocumentDb databases directly within VS Code via an integrated browser.
Azure Functions	Provides integrated support for writing, testing and deploying Azure Functions.
Azure Machine Learning	Formerly called Visual Studio Code for AI Tools, allows for creating, building, training, and deploying machine learning models based on your Azure subscriptions.
Azure Resource Manager	Allows managing Azure resource groups in VS Code.
Azure Storage	Allows connecting to blobs, tables, files, and queue storage in your Azure subscriptions. It also allows uploading folders directly from within VS Code.
Deploy to Azure	Allows for setting up continuous integration and continuous deployment pipelines for Azure DevOps code repositories.
Docker	Allows for publishing containerized applications from Visual Studio Code, with improved code editing features for Docker and YAML files.
Kubernetes	Provides integrated support to deploy Docker containers to Kubernetes, an open source system for automating deployment, scaling, and management of containerized applications, supported by Azure.

I recommend that you bookmark the official documentation, available at `https://code.visualstudio.com/docs/azure/extensions`, for further details and examples. Noteworthy is that Visual Studio Code can support Docker and Kubernetes for containerized applications, which is something very important for many developers.

Deploying Web Applications

Deploying web applications to Azure with Visual Studio Code is very easy. You can retake the **helloweb** sample applications created with C# and .NET Core in Chapter 9, but it's worth remembering that publishing to Azure is not limited to these technologies, but is also possible for Node.js.

Note Visual Studio Code, the Microsoft Azure platform, and Azure extensions for VS code continuously evolve. New releases might introduce changes to what is described in this chapter.

Installing Extensions

The first thing you need to do is install the Azure App Service extension from the Marketplace. This extension also needs the Azure Account and the Azure Resources extensions, but these are installed together with the App Service, so you do not need to take any additional steps.

The Azure Account extension is actually required to enable developers to log into their Azure account from within Visual Studio Code and to select which subscription to use. The Azure Resources extension is used to manage resources groups, which are the places where your cloud services are organized. Figure 11-1 shows the Azure App Service extension in the Extensions panel.

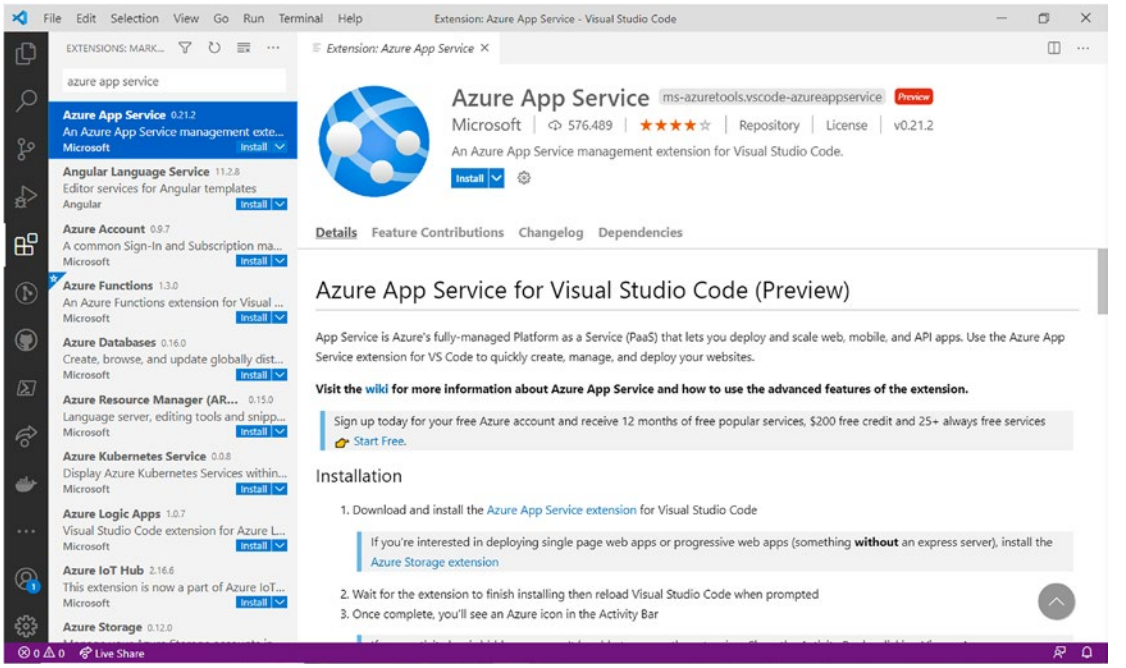

Figure 11-1. *The Azure App Service extension from Microsoft*

Signing into Azure Subscriptions

Once the Azure App Service extension has been installed, along with the Azure Account and Azure Resource Groups extensions, you need to sign in before you can use any service.

To accomplish this, you can use the **Azure: Sign In** command from the Command Palette or the **Sign in to Azure** shortcut in the App Service node of the Azure side bar. Either action opens an instance of your default browser pointing to the Microsoft Account login service. Simply enter your credentials, sign in, and close the browser window once you are logged in. Now in Visual Studio Code you can open the Azure extension and see the list of services associated to your subscription. Figure 11-2 shows an example based on my subscription.

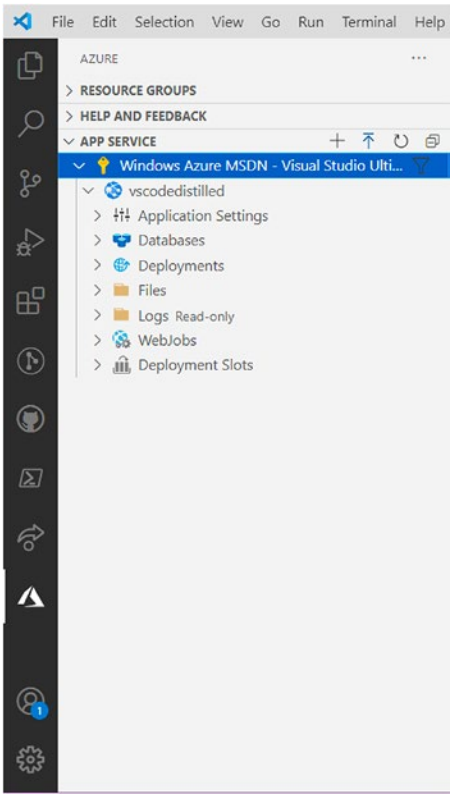

Figure 11-2. *The Azure services view*

Note The Microsoft Azure offering is very extensive and spans a plethora of services, so I recommend that you look at the official website (`https://azure.microsoft.com/en-us/free`) for detailed information. In addition, do not forget to enter the management portal (`https://portal.azure.com`), which gives you access to the full tools and options to create and manage your services and resources.

The hierarchical view displays resource groups and the services they contain, and it also supports multiple subscriptions.

You can quickly interact with each service by expanding its group and accessing the available options by right-clicking its name.

Publishing Web Applications

Visual Studio Code makes the process of publishing web apps to Azure very easy. The goal of this section is to demonstrate how quick and easy it is to publish a web application to Azure. Assuming you have opened the helloweb sample project, in the Azure view, right-click the name of your subscription and select **Create New Web App**.

A three-step wizard guides you through the creation of the application. The first step asks you to supply a unique name for your new web application in the Command Palette, as shown in Figure 11-3.

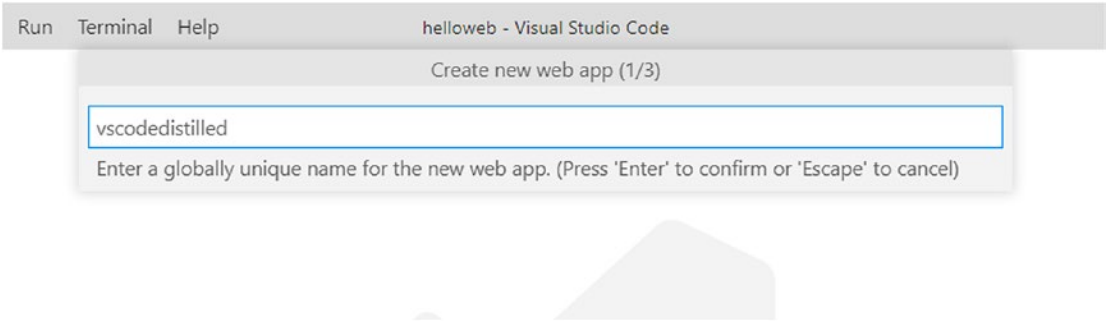

Figure 11-3. *Specifying a name for the web application*

Because the name you specify will be combined with the azurewebsites.net domain and represents the web address of your applications, if the name is already taken, a validation message appears, inviting you to choose a different name. You might want to specify a name that is different from `vscodedistilled`, which is the name I use for the examples in this chapter.

The next step is to specify the target environment for your web application; this is necessary because the Azure extension cannot detect which technology your app is based on. Figure 11-4 shows the list of available options.

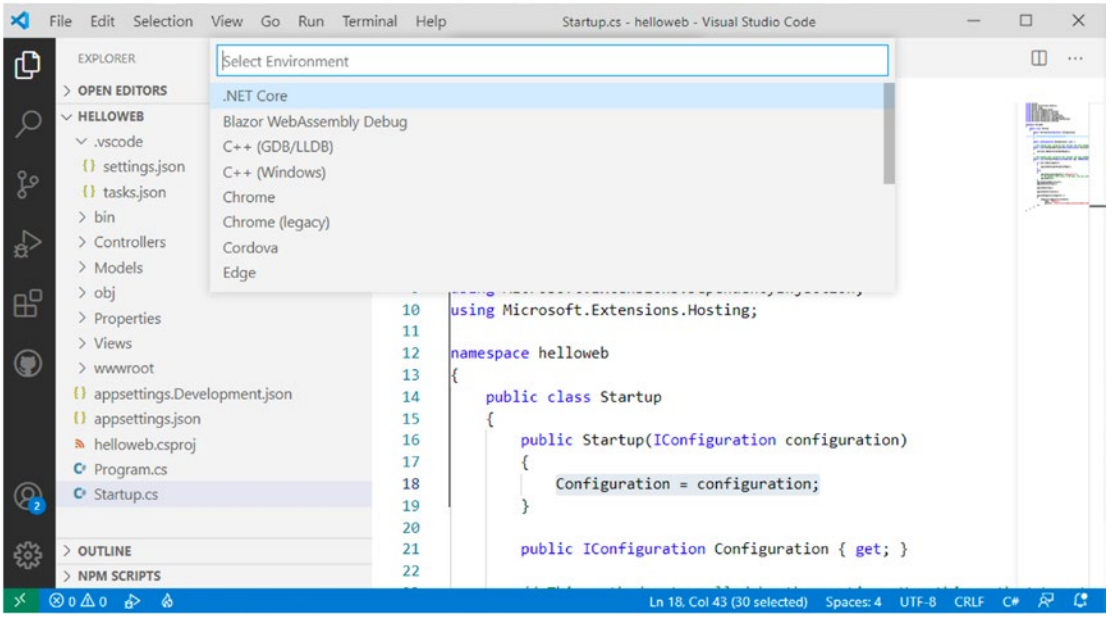

Figure 11-4. *Specifying a target platform*

Because the sample application was written on .NET 5, select this as the target platform. The last step of the wizard asks you to specify a pricing tier. I suggest using the Free tier, as shown in Figure 11-5.

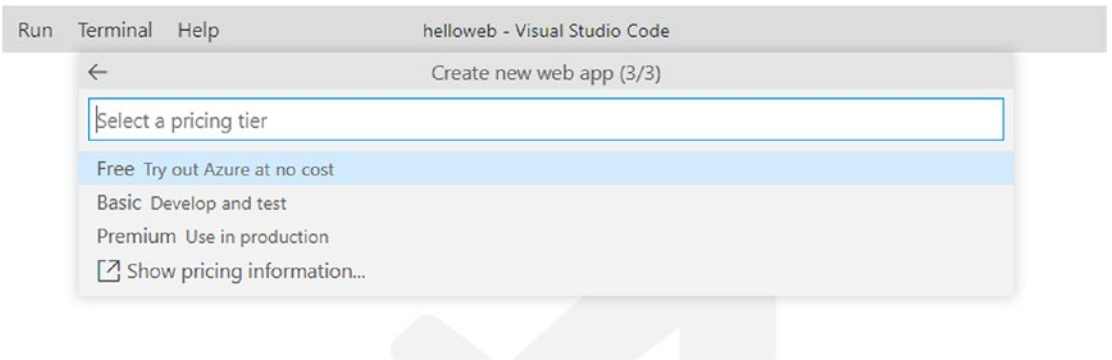

Figure 11-5. *Specifying a pricing tier*

After you complete these three easy steps, Visual Studio Code first builds the project in Release mode (and the result will be visible in the Terminal) and then starts creating the necessary resources inside your Azure subscription, and you will be able to see the progress in a pop-up box that appears in the bottom-right corner of the environment.

When everything is ready, a pop-up message asks if you want to enable automatic deployment. Click **Always Deploy** so that the application will be published.

When deployment is completed, the browser automatically launches the newly published application (see Figure 11-6). If this does not happen, you can right-click the application name in the APP SERVICE view of the Azure side bar and select **Browse Website**, then click the **Open** button in the dialog that informs you about the fact that an external program is being launched.

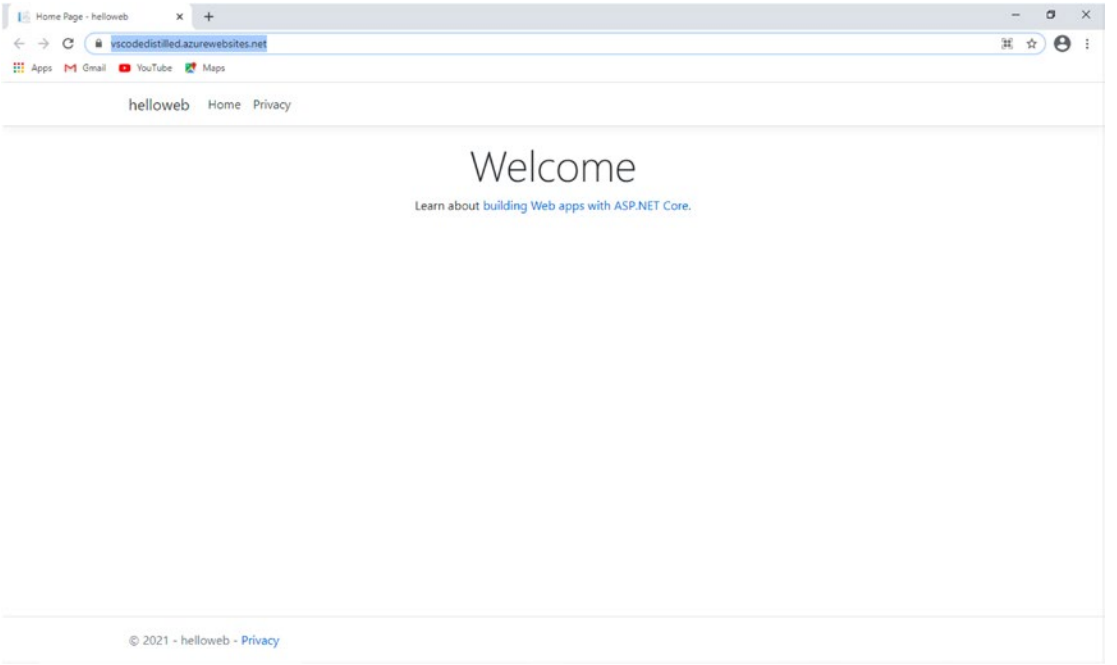

Figure 11-6. *The web application running in the cloud*

You need no additional steps. Your application is up and running in the browser, hosted in your Azure subscription. You can further manage your Azure services and resources, both within Visual Studio Code and in the Azure portal (`https://portal.azure.com`). Though managing resources in the Azure portal is a bigger topic and is out of the scope of this chapter, Figure 11-7 shows the management page for the sample web application, where you can see the full list of available settings on the left side and information on the deployment, the data center, and statistics in the main view.

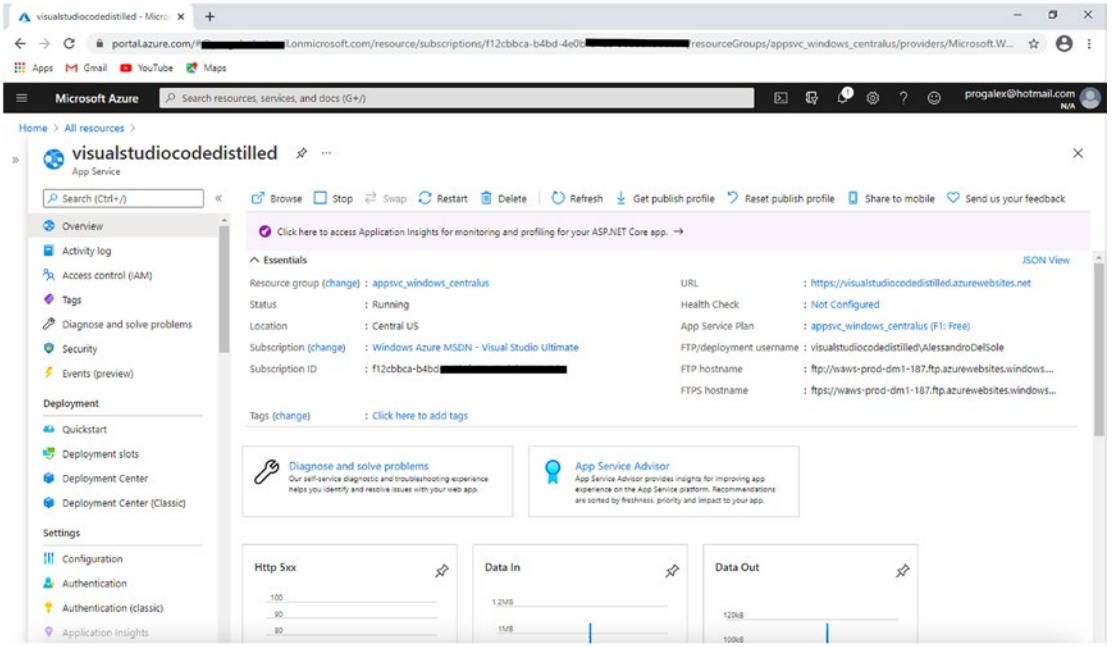

Figure 11-7. *Managing app services in the Azure portal*

Creating and Deploying Azure Functions

Put succinctly, Azure Functions (`https://docs.microsoft.com/en-us/azure/azure-functions`) is a service that allows for running code on-demand in the cloud, and it is considered part of the growing trend of *serverless computing*. The biggest benefit of using Azure Functions is that functions are triggered only when invoked, which not only reduces the usage of cloud resources but also reduces maintenance and infrastructure needs, thereby providing more cost saving.

Configuring Visual Studio Code

Azure supports writing Functions in several languages, such as C#, Python, Java, JavaScript, and Rust. Usually, tools are available for different development environments to write Azure Functions, such as Visual Studio 2019, and Visual Studio Code is no exception.

The first thing you need to develop Azure Functions with VS Code is Azure Functions Core Tools. This set of command-line tools is required to run the tasks necessary to develop, debug, and publish functions. On Windows, you have two ways to install these tools:

download the installer for Windows from the official website (https://bit.ly/3f1lHxR) or use the following command that leverages npm on Node.js and that you can run from a Terminal window in VS Code or from a developer command prompt:

```
> npm i -g azure-functions-core-tools@3 --unsafe-perm true
```

I recommend using the latter command-line method to install the tools, because Visual Studio Code might not recognize that the tools were installed via the installer package.

On macOS, you need to run the following commands:

```
> brew tap azure/functions
> brew install azure-functions-core-tools@3
```

On the latest version of Ubuntu, the required commands are the following:

```
> wget
-q https://packages.microsoft.com/config/ubuntu/20.04/packages-microsoft-
prod.deb
> sudo dpkg -i packages-microsoft-prod.deb
```

The installation commands vary depending on the Linux distribution, so you can locate the appropriate commands at https://github.com/Azure/azure-functions-core-tools#linux.

Once you have installed Azure Functions Core Tools, you need to install the Azure Functions extension for Visual Studio Code (see Figure 11-8).

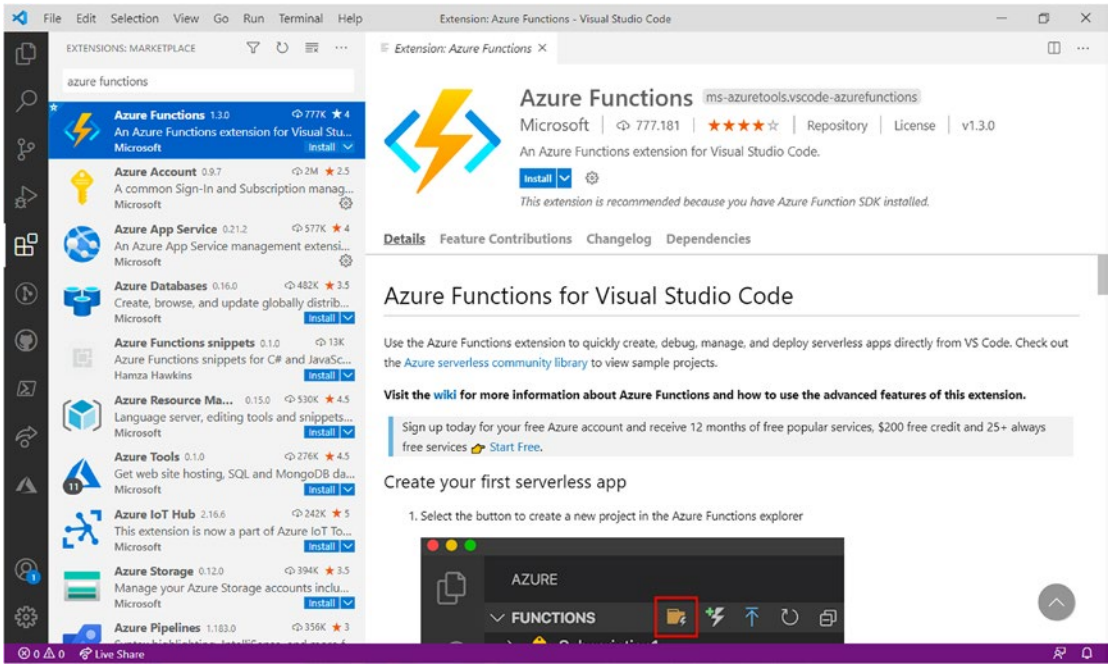

Figure 11-8. *The Azure Functions extension for VS Code*

The Azure Functions extension also needs the Azure Account one, which you already installed previously.

Creating Azure Functions

With the Azure Functions extension installed, VS Code simplifies the way you can create Azure Functions projects. For the current example about deploying Azure Functions, I will show how to create a function stub using the built-in templates, but you can certainly use existing Azure Functions projects created with other environments or sample projects.

If you are starting with new code, you first need to have (or create) a new folder on disk where the new projects will be created. For the next example, I have created a folder on disk called C:\AzureFunctionsDistilled.

When you have the folder ready, in Visual Studio Code enable the Command Palette and search for the command called **Azure Functions: Create New Project** (see Figure 11-9).

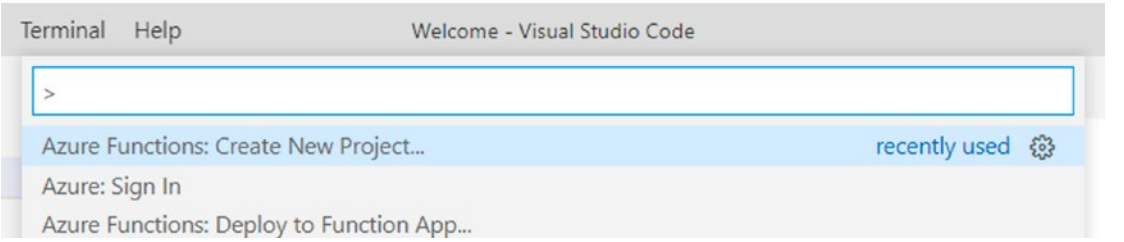

Figure 11-9. *Creating a new Azure Functions project*

Note There are two additional commands available to create Functions: **Create Function App in Azure** and **Create Function App in Azure (Advanced)**. Both commands allow to create a project that is automatically provisioned in your Azure subscription, together with a local project for development and debugging. In this book, I'm not using these commands in order to better highlight the different phases of development and debugging, and then deployment.

When you click this command, an eight-step wizard starts. First, you are asked to select a target folder on disk, so pick the one you created previously. Then you are asked to select a language. For the sake of consistency with the previous examples, I have selected C#, but you are free to use a different one. In the third step, you are asked to specify a runtime platform. If you selected C#, the wizard shows .NET versions and you can select the latest.

Note The wizard identifies .NET 5 as .NET 5 (Isolated). Understanding what this means requires taking a step back into the previous versions of Azure Functions. Previously, Azure Functions only supported a tightly integrated mode for .NET functions, which run as a class library in the same process as the host. Though this mode provides deep integration between the host process and the functions, this integration also requires a tighter coupling between the host process and the .NET function. For example, .NET functions running in-process are required to run on the same version of .NET as the Functions runtime. To enable you to run outside these constraints, you can now choose to run in an isolated process. .NET 5 (Isolated) then means that support for running functions out-of-process is now allowed.

If you selected another language, the list of target platforms will change depending on your language of choice.

In the fourth step, you have the option to select a project template (see Figure 11-10).

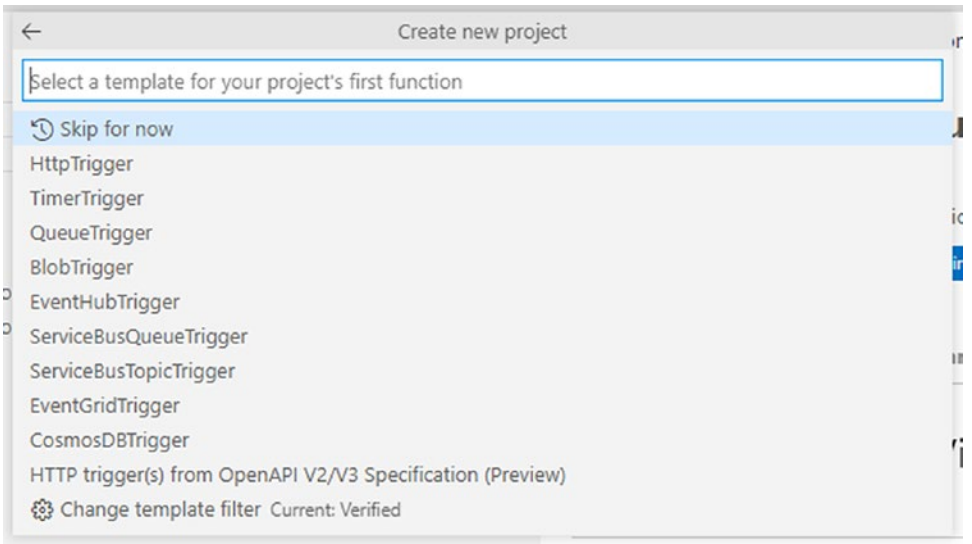

Figure 11-10. *Selecting an Azure Functions project template*

The project template you select here is not really relevant for the current example, whose goal is not to go into the details of Azure Functions development but rather to show how quick and easy building and deploying functions is. I selected the HttpTrigger template, which generates simple code that defines a function that is triggered on Azure when an HTTP/HTTPS request is intercepted, sending a response back.

In the fifth and sixth steps, you first enter a name for the new project (or leave the default project name, like AzureFunctionsDistilled in the current example) and then enter a namespace that will be used in the code. The namespace should be in the form `CompanyName.Function`; for example, my namespace is `AlessandroDelSole.AzureFunctionsDistilled`.

In the seventh step of the wizard, you specify a security access level: **Anonymous**, **Functions**, or **Admin**. Table 11-2 provides a short description of each authorization level.

Table 11-2. *Azure Functions Authorization Levels*

Level	Description
Anonymous	No authorization required; all HTTP requests pass.
Function	Function authorization level is based on security keys generated in the Azure portal. Host keys (at the application level) and function keys (at the function level) can work as security keys in the Function level.
Admin	Similar to the Function level, but only works with host keys (at the app level).

For the current example, you can just select the **Anonymous** level. In the final step of the wizard, you decide where to open the new project: **Current Window** (current instance of Visual Studio Code), **New Window** (new instance of Visual Studio Code), or **Add to WorkSpace** (the new project is added to an existing folder to create a workspace). Select **Current Window** and, after a few seconds, the new project will be available and you will be ready to edit the code depending on your needs (see Figure 11-11).

Note The function name defined by the `FunctionName` attribute must always be lowercase, otherwise the runtime will throw an exception. In the current example, make sure to change from `FunctionName("AzureFunctions Distilled")` to `FunctionName("azurefunctionsdistilled")`.

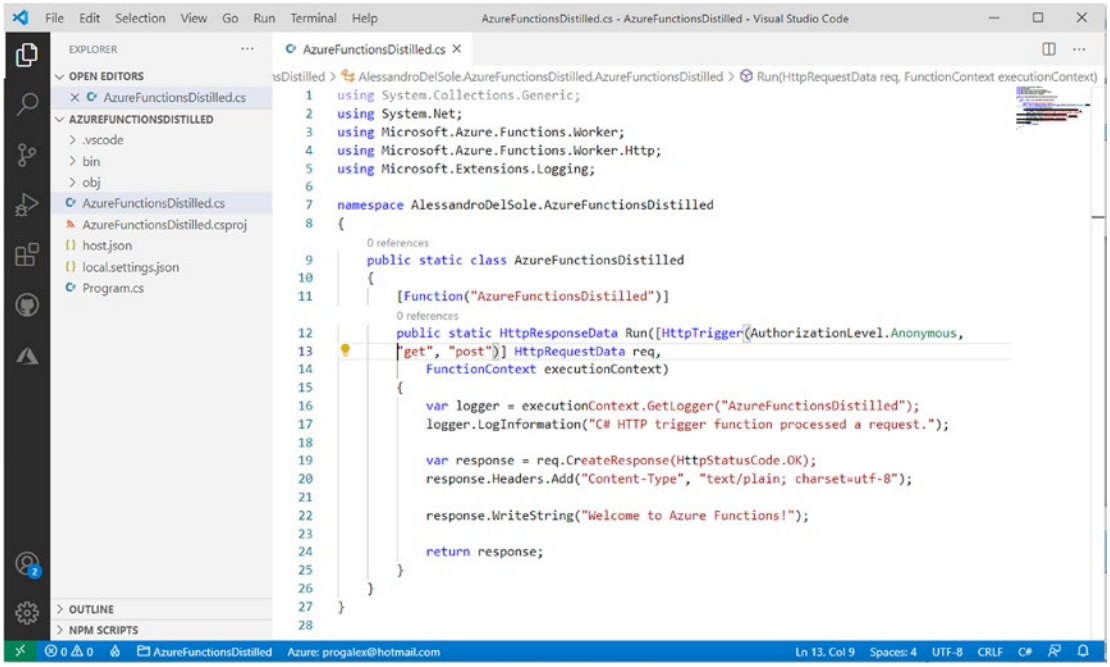

Figure 11-11. *Editing the Azure Functions project in VS Code*

You are now working fully locally, which is a good opportunity to debug your code on a development environment before promoting the code to the Azure, remote environment. Press F5 to start debugging, exactly as you would do with any C# project, and after a few seconds the Terminal will show not only the compiler output but also a local URL that you can use to test the code (see Figure 11-12).

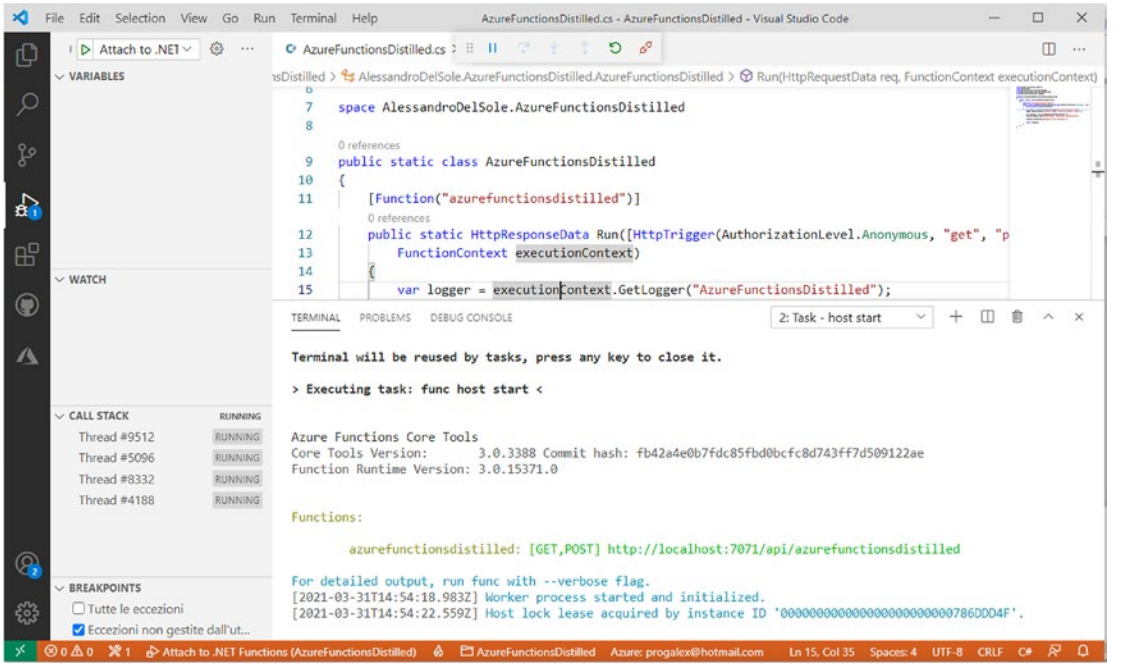

Figure 11-12. *Debugging an Azure Function*

The URL shown in the Terminal is the following: `http://localhost:7071/api/azurefunctionsdistilled`. 7071 is the port of the local development server, while `azurefunctionsdistilled` is the name (all lowercase) of the function defined in the code, and both will vary depending on the projects you create. You can paste the aforementioned URL into the address bar of your browser, and then press Enter. Figure 11-13 shows the function running in the browser and listening for `HTTP GET` and `POST` calls.

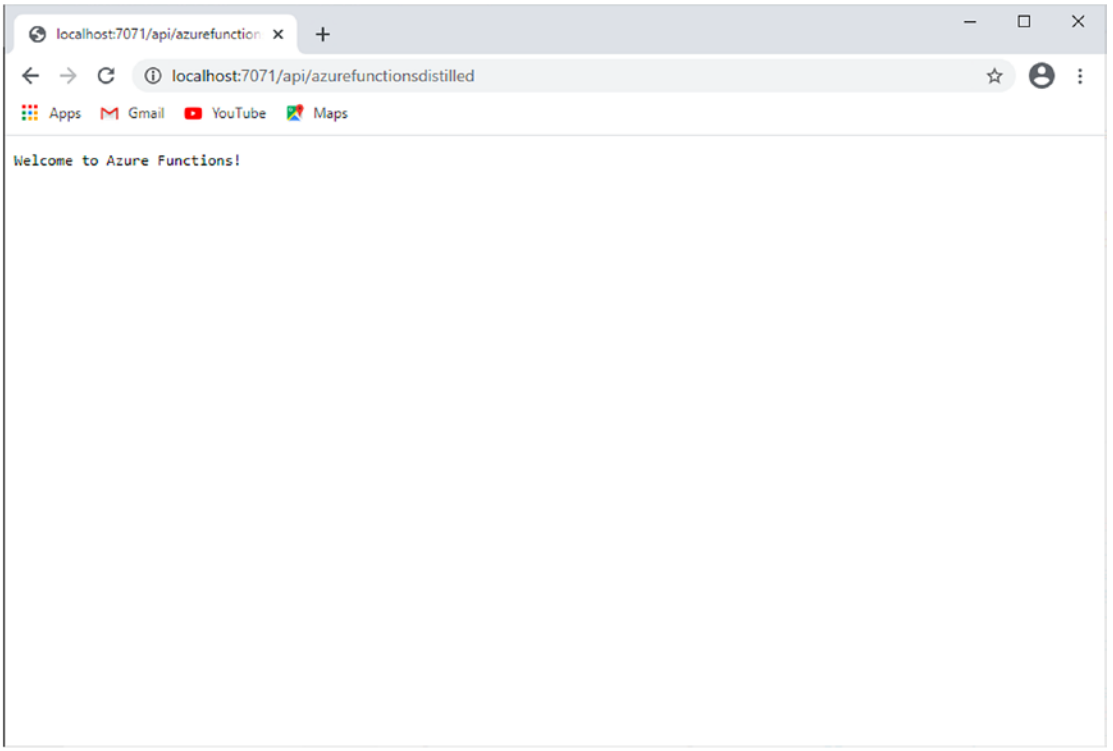

Figure 11-13. *Running an Azure Function locally*

Assuming that you have done all your local development, debugging, and testing, you can publish the Azure Function to the cloud, as described next.

Deploying Azure Functions

Deploying Azure Functions to your subscription in Visual Studio Code is an easy task. In the FUNCTIONS area of the Azure panel, you can click the **Deploy to Function App** button, highlighted in Figure 11-14, or you can right-click the subscription name in the FUNCTIONS view and then select the same command.

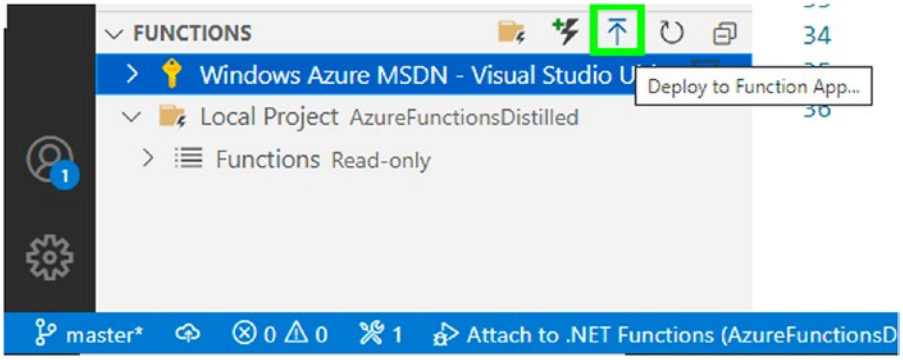

Figure 11-14. *Initiating the deploy process with the Deploy to Function App button*

Once you click this button, the Command Palette shows a quick wizard consisting of three steps. In the first step, specify if you want to create a new Azure Function app with default settings or with advanced settings (see Figure 11-15).

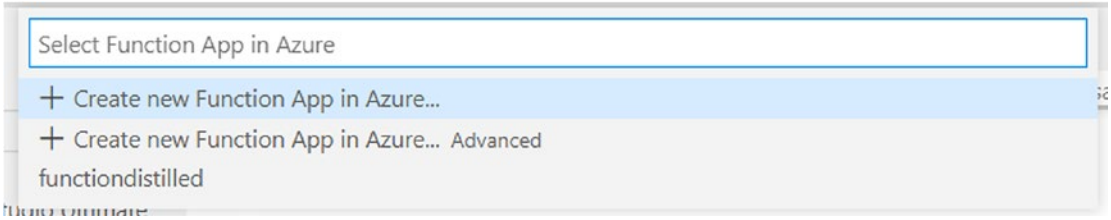

Figure 11-15. *Choosing default or advanced settings to create a new Function app*

Select the first (default) option and then press Enter. You are asked again to specify a unique name (for the current example it is `azurefunctionsdistilled`) and then to specify the target platform, and available options depends on the technology you used to build the app. Select the same platform you selected when creating the project.

Note You might see the (non-LTS) phrase close to a .NET version in the Command Palette. At this writing, it is .NET 5 (non-LTS). This phrase means that the identified version of .NET is not supported to long term (LTS stands for Long Term Support). The reason is that Microsoft plans to release .NET 6 by the end of 2021, and that will provide extensive support for this new version once it ships.

In the last step of the wizard, you need to specify a data center location (see Figure 11-16).

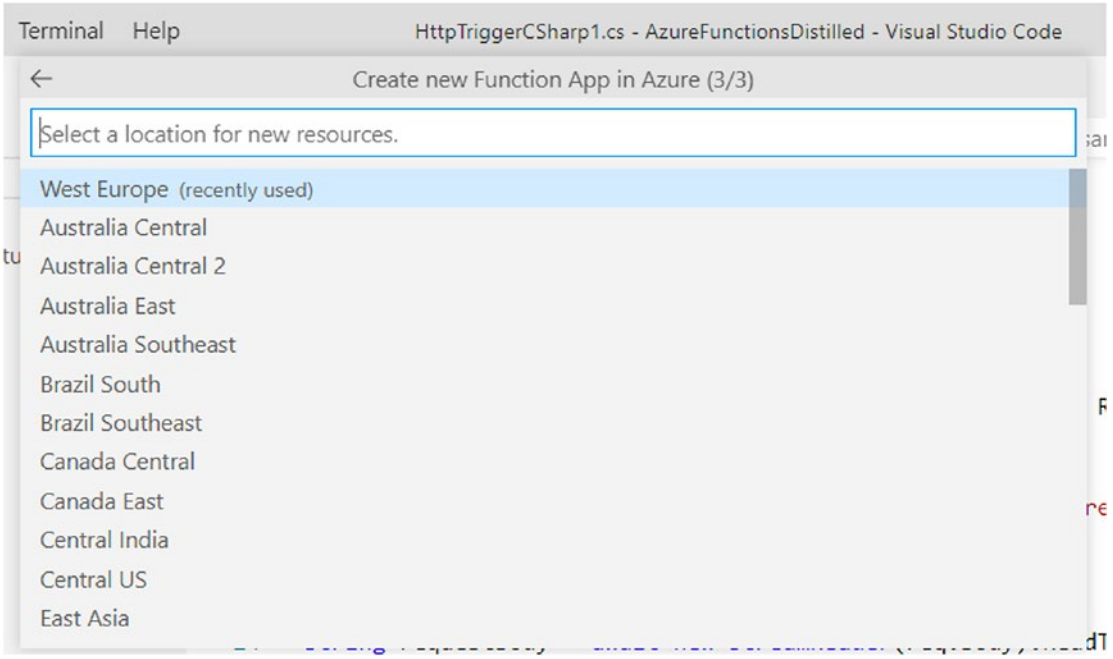

Figure 11-16. *Selecting a location for the data center*

If you have experience with Azure development, you know that this is a crucial choice, because the location you select has an impact on the costs charged to your subscription. At least for this example related to development purposes, make sure that you select the data center that is closest to your location (in my case it is West Europe), which translates to less latency and less bandwidth required and corresponding cost savings, especially if your subscription does not have a spending limit enabled.

Note Not all Azure regions and data centers offer the same services. For real-world scenarios, you might want to look at the official documentation about choosing the appropriate Azure region based on your location, needs, and requested services (https://azure.microsoft.com/en-us/global-infrastructure/geographies).

At this point, Visual Studio Code first builds the project in Release mode and then starts publishing the function to Azure. You can follow the progress in the Terminal and with the pop-up box that shows the name of the currently running task (see Figure 11-17).

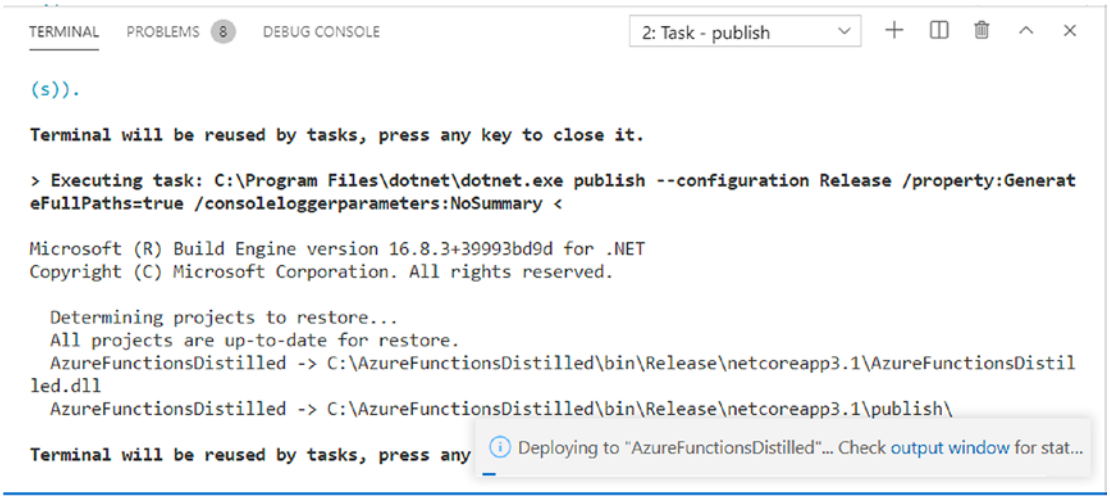

Figure 11-17. *Publication of the Azure Function is in progress*

After the last step, the function will be up and running in the cloud, which you can easily verify by opening the function's URL in the browser, as shown in Figure 11-18. Remember that the function's URL is made by the unique name you supplied when creating the project, followed by the azurewebsites.net domain name and by the /api/<functionname> part. In the case of an Azure Function, you can add the query string required to trigger the function itself. In Figure 11-18 you can see how the same query string used locally has also been supplied to the remote URL.

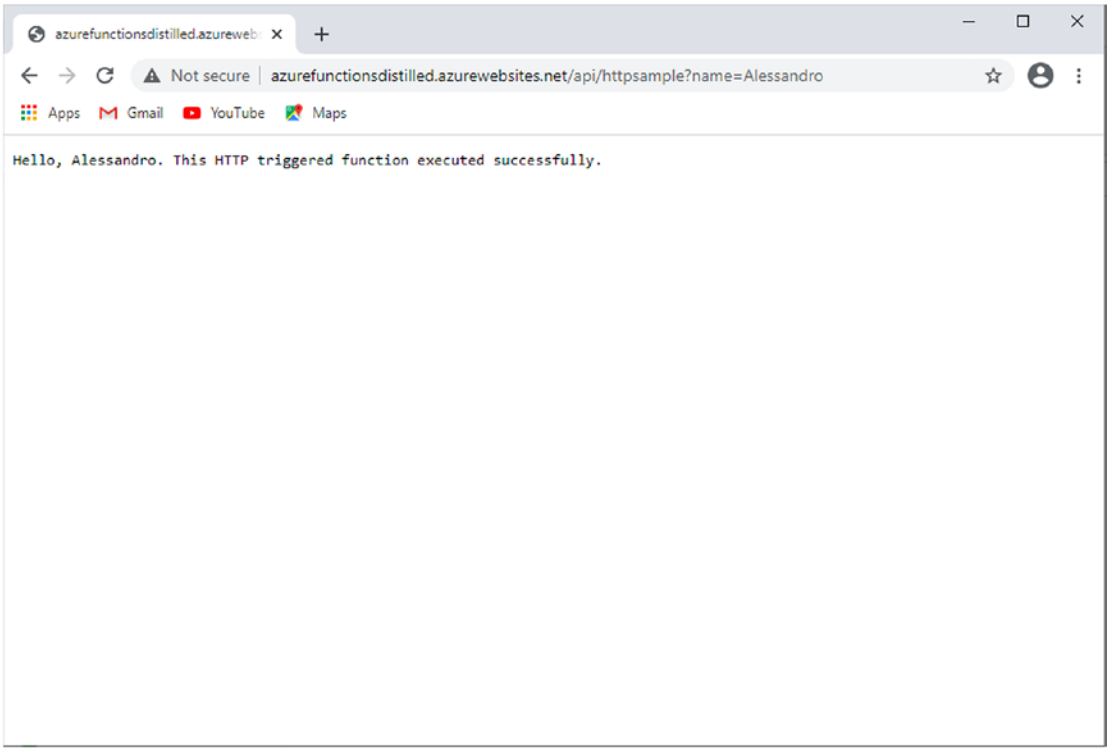

Figure 11-18. *The Azure Function is running in the cloud*

As you have seen, Visual Studio Code makes it very simple not only to deploy an Azure Function, but also to create a project and interact with the Azure subscription directly from within the environment, which improves overall productivity.

Note To avoid unexpected charges or consumption of your Azure credit, I recommend deleting all the resources that you no longer use, such as the sample applications created in this chapter. In VS Code you can quickly delete apps and functions by right-clicking on their name in the APP SERVICE and FUNCTIONS panels (respectively) of the Azure side bar and then selecting the appropriate Delete command. Additional resources can be deleted in the Azure portal.

Summary

Once again, Visual Studio Code demonstrates its power and versatility even with cloud development based on Microsoft Azure. With the Azure extensions, you have direct access to your subscriptions directly from within the environment.

With specialized extensions, such as Azure App Service and Azure Functions, you can create, configure, and deploy your web applications and functions with limited effort and a few mouse clicks, reducing the need to manage resources in the Azure portal only to situations in which you need custom configurations. In addition, multiple languages and environments are supported, including .NET Core, Java, Python, and Node.js, extending the cloud development possibilities to a larger number of companies and developers.

Index

A. Del Sole, *Visual Studio Code Distilled*, https://doi.org/10.1007/978-1-4842-6901-5

F

G

H

I, J

K

L

M

N, O

P, Q

R

S

T, U

V

W, X, Y, Z