Chapter 1 Code Editing Evolved for Windows, Linux, and OS X

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 by supporting the application development life cycle with a built-in debugger for Node.js and integrated support for the popular Git version control engine. With Code, you can work with individual code files or structured file systems based on folders. This chapter provides full guidance about the powerful editing features available and explains how Visual Studio Code manages code files in its workspace.

Tip: In this book I will use the Visual Studio Code, VS Code, and Code names interchangeably.

Why Visual Studio Code

Before you learn how to use Visual Studio Code, what features it offers and how it provides an improved code editing experience, it is very helpful to understand its purpose. Visual Studio Code is not a simple code editor; it is a powerful environment that puts writing code at its center. The main purpose of Visual Studio Code is not building binaries (such as .exe and .dll files); it's making it easier to write code for web, mobile, and cloud platforms for any developers working on different operating systems, such as Windows, Linux, and OS X, keeping you independent from proprietary development environments. For a better understanding, I'll go through an example: Think of ASP.NET Core 1.0, the new cross-platform, open-source technology able to run on Windows, Linux, and OS X that Microsoft produced to create portable web applications. Forcing you to build cross-platform, portable web apps with Microsoft Visual Studio 2015 would make you dependent on this IDE. You could argue that Visual Studio 2015 Community edition is free of charge, but it only runs on Windows, so what about Linux or Mac users? On the other hand, though it is certainly not intended to be a replacement for more powerful and complete environments such as its major brother, Visual Studio Code can run on a variety of operating systems and can manage different project types, as well as the most popular languages. To accomplish this, Visual Studio Code provides the following core features:

- Built-in support for coding with many languages, including those you typically use in cross-platform development scenarios, with advanced editing features and support for additional languages via extensibility.
- Built-in debugger for Node.js, with support for additional debuggers (such as Mono) via extensibility.
- Version control based on the popular Git engine, which provides an integrated experience for collaboration that supports code commits and branches, and 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 that is different from other developer tools, such as Microsoft Visual Studio. In fact, Visual Studio Code is a folder-based environment that makes it easy to work with code files that are not organized within projects, and offers a unified way to work with different languages. Starting from this assumption, Code offers an advanced editing experience with features that are common to any supported languages, plus some features that are available to specific languages. As you will 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. Now that you have a clearer idea of Code's goals, you are ready to learn the amazing editing features that put it on top of any other code editor.

When should I use Visual Studio Code?

As the product name implies, Visual Studio Code is a code-centric tool, with a primary focus on web, cross-platform code. That said, it does not provide all the features you need for full, more complex application development and application life-cycle management and is not intended to be the proper choice with some development platforms. If you have to make a choice, consider the following points:

- Visual Studio Code does not invoke compilers, so it does not produce binaries such as .exe or .dll files. You can certainly implement task automation, which will be discussed in Chapter 3, 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 scenarios but it can be very complicated with certain applications and development platforms, especially if you are used to working with the powerful graphical tools available in Microsoft Visual Studio.
- The built-in debugger is specific to web, cross-platform languages and it currently targets only Node.js. Plus, it is not as advanced as the Visual Studio debugger.
- 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 2015, which is the premier development environment for building, testing, deploying, and maintaining any kind of application. The Visual Studio 2015 Community edition is free and provides a full development environment for any development platform.

Language support

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

Table 1: Language Support

Available Languages Grouped by Editing Features	
Batch, C++, Clojure, CoffeeScript, Dockerfile, F#, Go, Jade, Java,	Common features (syntax coloring, bracket matching)

Available Languages Grouped by Editing Features	
Handlebars, Ini, Lua, Makefile, Objective-C, Perl, PowerShell, Python, R, Razor, Ruby, Rust, SQL, Visual Basic, XML	
Groovy, Markdown, PHP, Swift	Common features and code snippets
CSS, HTML, JSON, Less, Sass	Common features, code snippets, IntelliSense, outline
JavaScript	Common features, code snippets, IntelliSense, outline, parameter hints. The JavaScript language service is based on Salsa.
TypeScript, C#	Common features, code snippets, IntelliSense, outline, parameter hints, refactoring, find all references. C# is powered by Roslyn and OmniSharp and is optimized for cross-platform .NET development with DNX.

Starting from version 0.10.10, C# is no longer available out of the box; instead it is available as an extension. The reason why I included C# in Table 1 is for consistency with previous versions and because many of you might already have experienced C# editing features in the past. In order to have C# installed, follow these steps:

1. Open a C# code file (.cs) in Visual Studio Code.
2. When asked, accept the option to install the C# extension.
3. Restart Visual Studio Code when the installation is complete.

Visual Studio Code can be extended for additional languages with tools produced by the developer community. These extensions can be downloaded from the Visual Studio Code Marketplace. This is discussed in more detail in Chapter 5, "Customizing and Extending Visual Studio Code." In the meantime, you can have a look at the available languages.

Note: This book describes all the available features in Visual Studio Code based on common development and coding scenarios. However, Visual Studio Code also has support for very specific scenarios with C#, JSON, HTML, Markdown, TypeScript, and DockerFile, so do not miss the Languages page in the documentation.

Editing features

Visual Studio Code provides many of the features you would expect from a powerful code editor. This section describes what editing features make your coding experience amazing with this new tool. If you are familiar with Microsoft Visual Studio 2013 and higher, 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 of the editing features, enabling you to edit code faster. For this reason, I will also mention the keyboard shortcuts for every feature I describe. With regard to this, remember that the counterpart for the Windows' Ctrl key in Mac is Cmd.

Syntax coloring

Visual Studio Code provides the proper syntax coloring for the language your code files are written with. Figure 1 shows an example based on a C# code file.

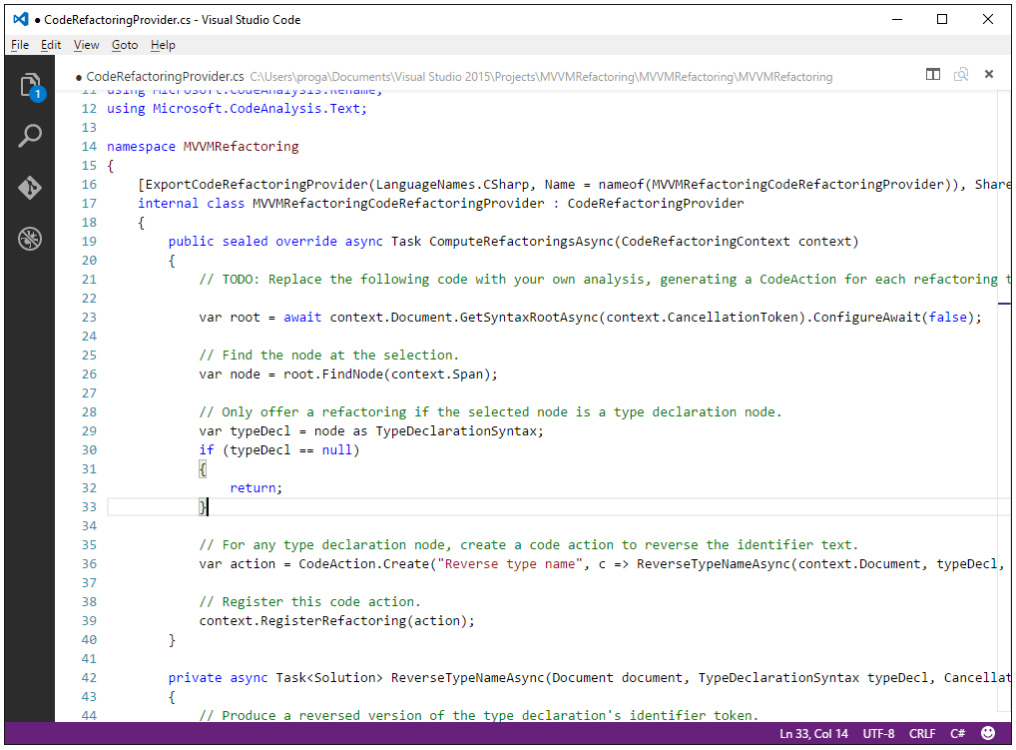


Figure 1: Syntax Highlighting

Syntax coloring is arguably the most important feature when editing a code file, and this is available to all supported languages. You can also define your own highlighting rules when adding custom languages (see Chapter 5).

Bracket matching

The code editor can highlight matching brackets, as shown in Figure 2.

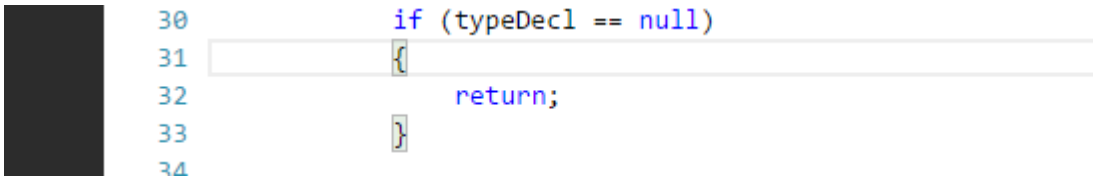


Figure 2: Matching brackets are highlighted.

This feature is extremely useful for identifying the beginning and end of long code blocks and is triggered once the cursor is near one of the brackets.

IntelliSense

IntelliSense is a feature that offers rich, advanced word completion via a convenient pop-up that appears as you type. In developer tools from Microsoft such as Visual Studio, IntelliSense has always been one of the most popular features because 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 shows IntelliSense in action.

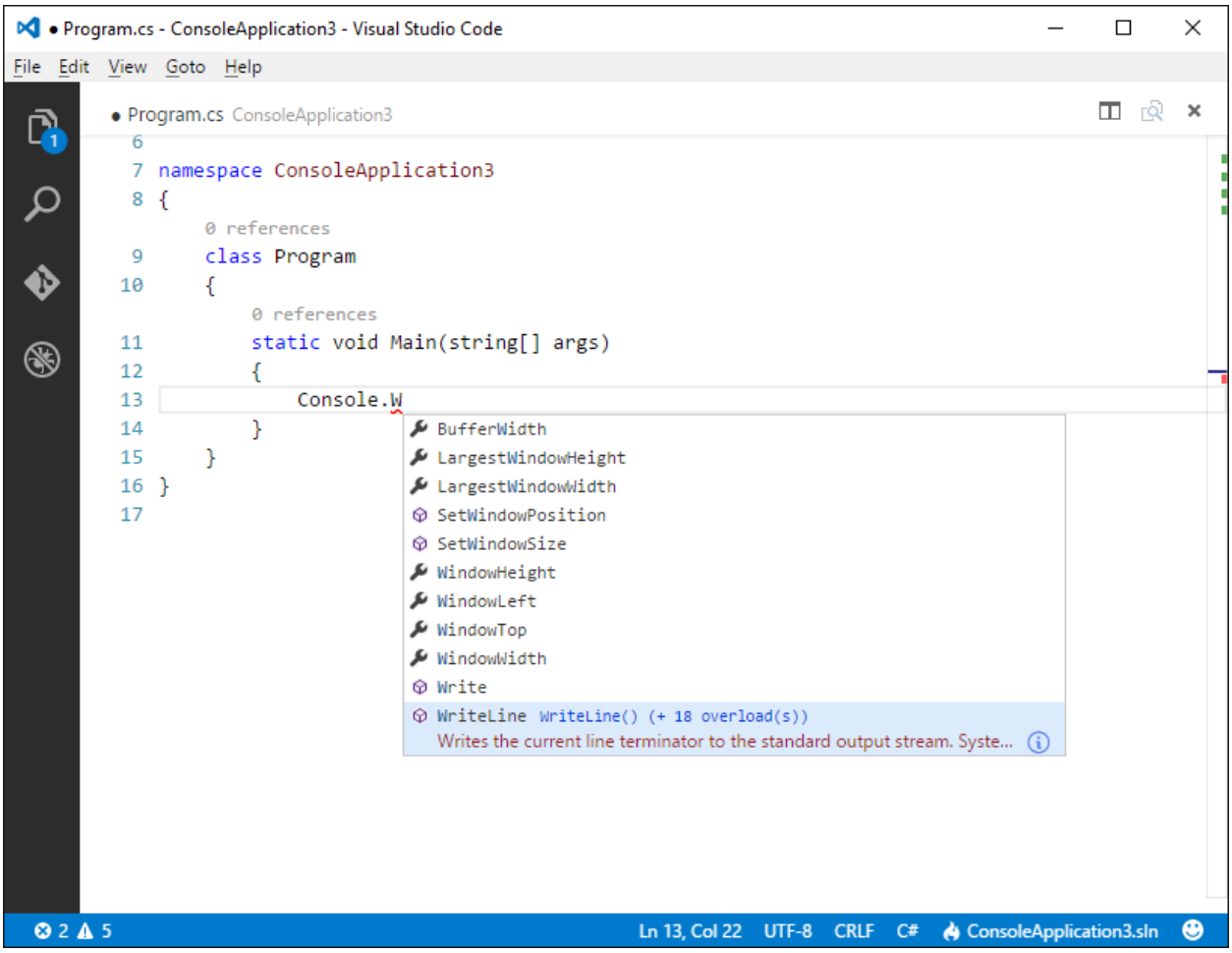


Figure 3: IntelliSense shows suggestions as you type and provides rich word completion.

As you can see in Figure 3, IntelliSense shows a list of available members for the given type--in this case **`Console`**--as you write. When you select a word from the completion list, Code shows the summary for the member documentation. Also, if you click the **i** symbol (which stands for Information), you can see the full member documentation (see Figure 4).

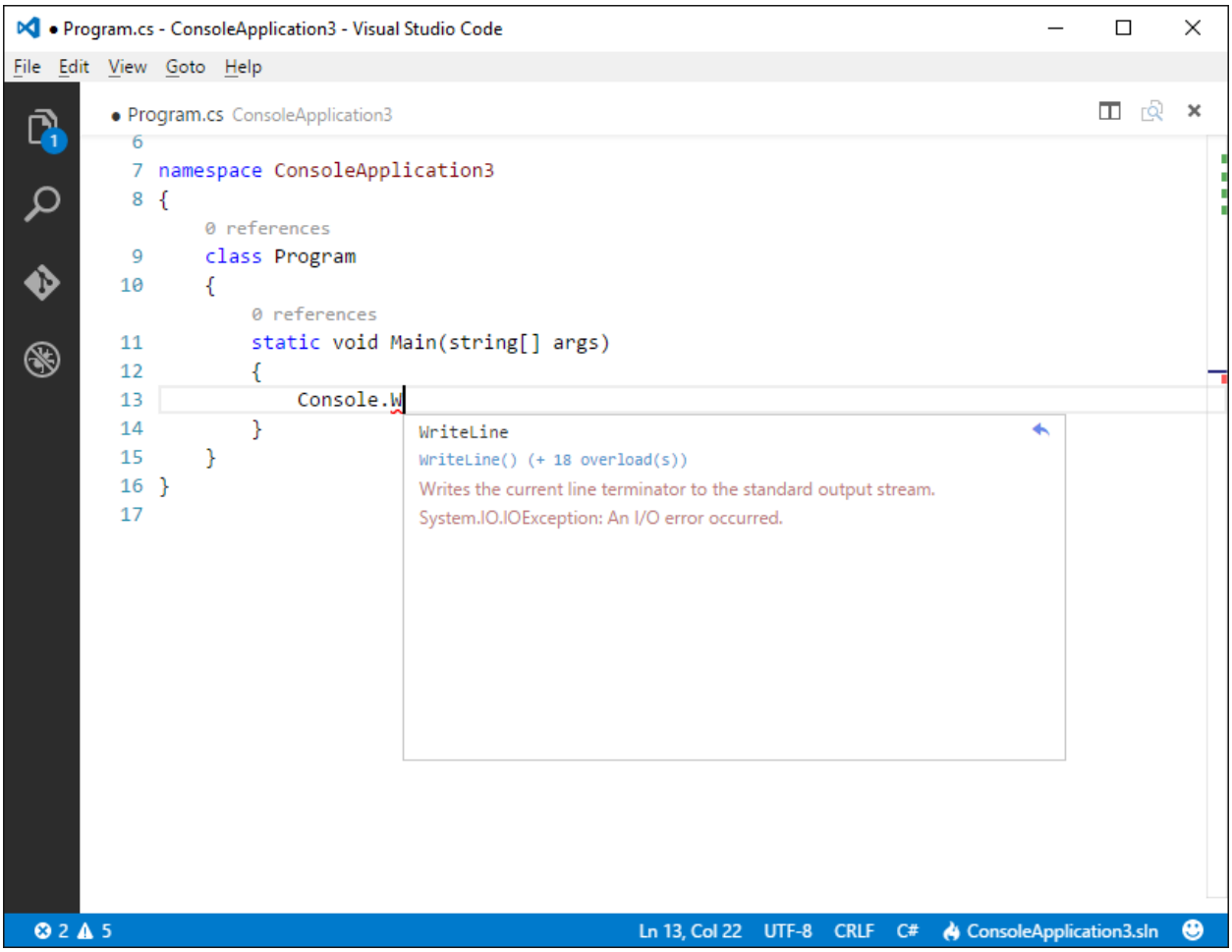


Figure 4: Getting a member's full documentation.

To quickly complete the word insertion, use either Tab or Enter. Not limited to this, IntelliSense in Visual Studio Code also supports suggestion filtering: Based on the CamelCase convention, you can type the uppercase letters of a member name to filter the suggestion list. For instance, if you are working against the **`System.Console`** type and you write **`cv`**, the suggestion list will show the **`CursorVisible`** property, as demonstrated in Figure 5.

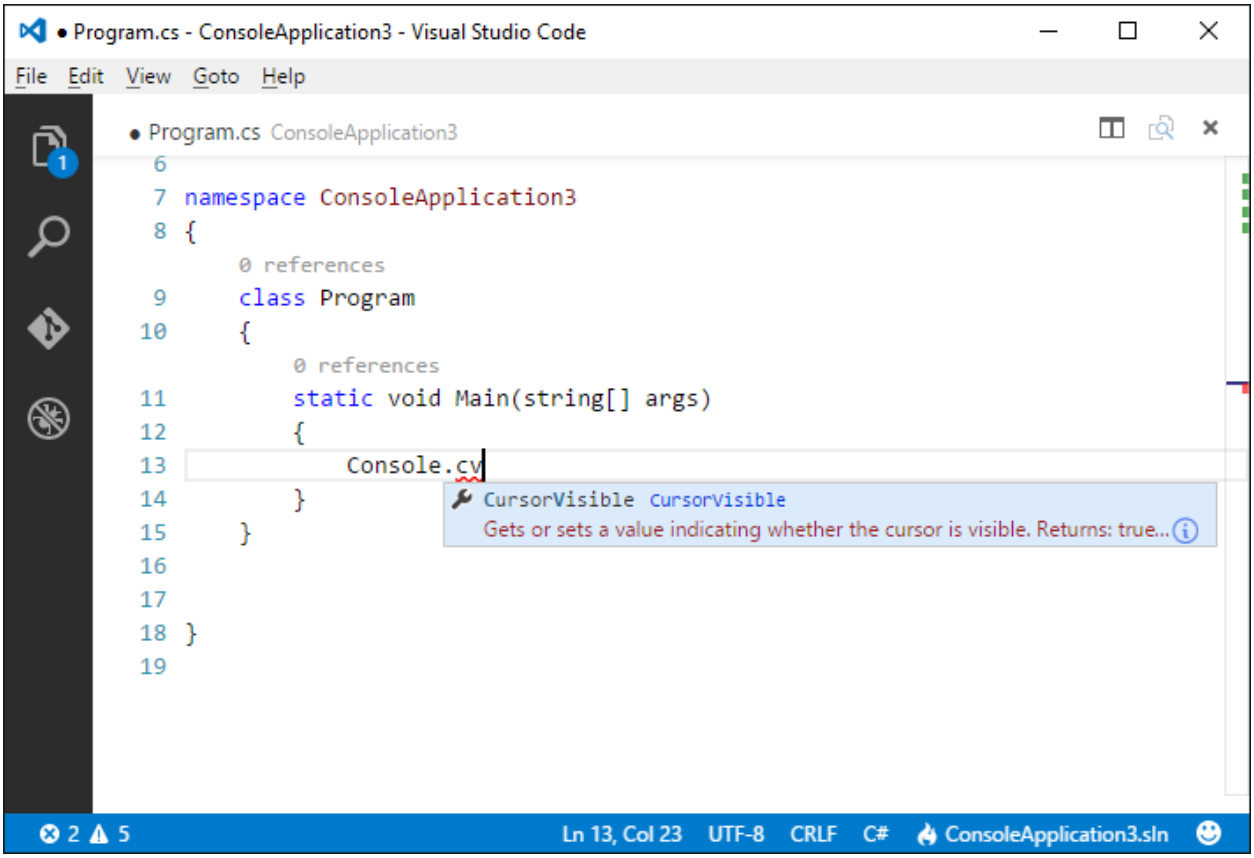


Figure 5: Filtering Suggestions

Out of the box, IntelliSense is available to the following languages: CSS, HTML, JavaScript, JSON, Less, Sass, TypeScript, and C#.

Parameter hints

When you code a function invocation, IntelliSense also shows a tooltip that describes each parameter if the documentation has been supplied (such as XML comments in C#). This feature is called **parameter hints** and you can see it in action in Figure 6.

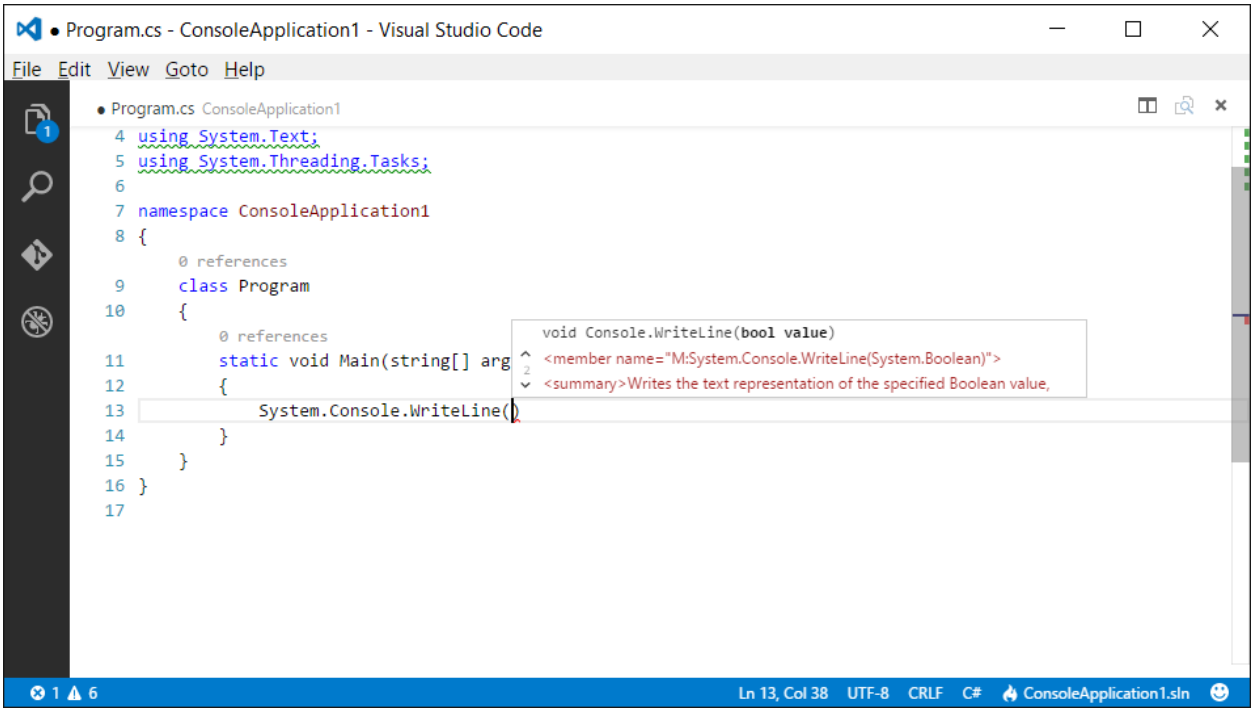


Figure 6: Visualizing Parameter Hints

Code block folding

The code editor allows folding delimited code blocks. Just hover the pointer over line numbers and the **-** symbol will appear near the start of a code block. Just click it to fold the code block. You will see the **+** symbol at this point, which you can then click to unfold the code block.

Go To Definition

With supported languages, Visual Studio Code provides an interesting feature called **Goto Definition**. You can hover over a symbol with the pointer and the symbol will appear as a hyperlink. Also, a tooltip will show the code that declares that symbol (see Figure 7).

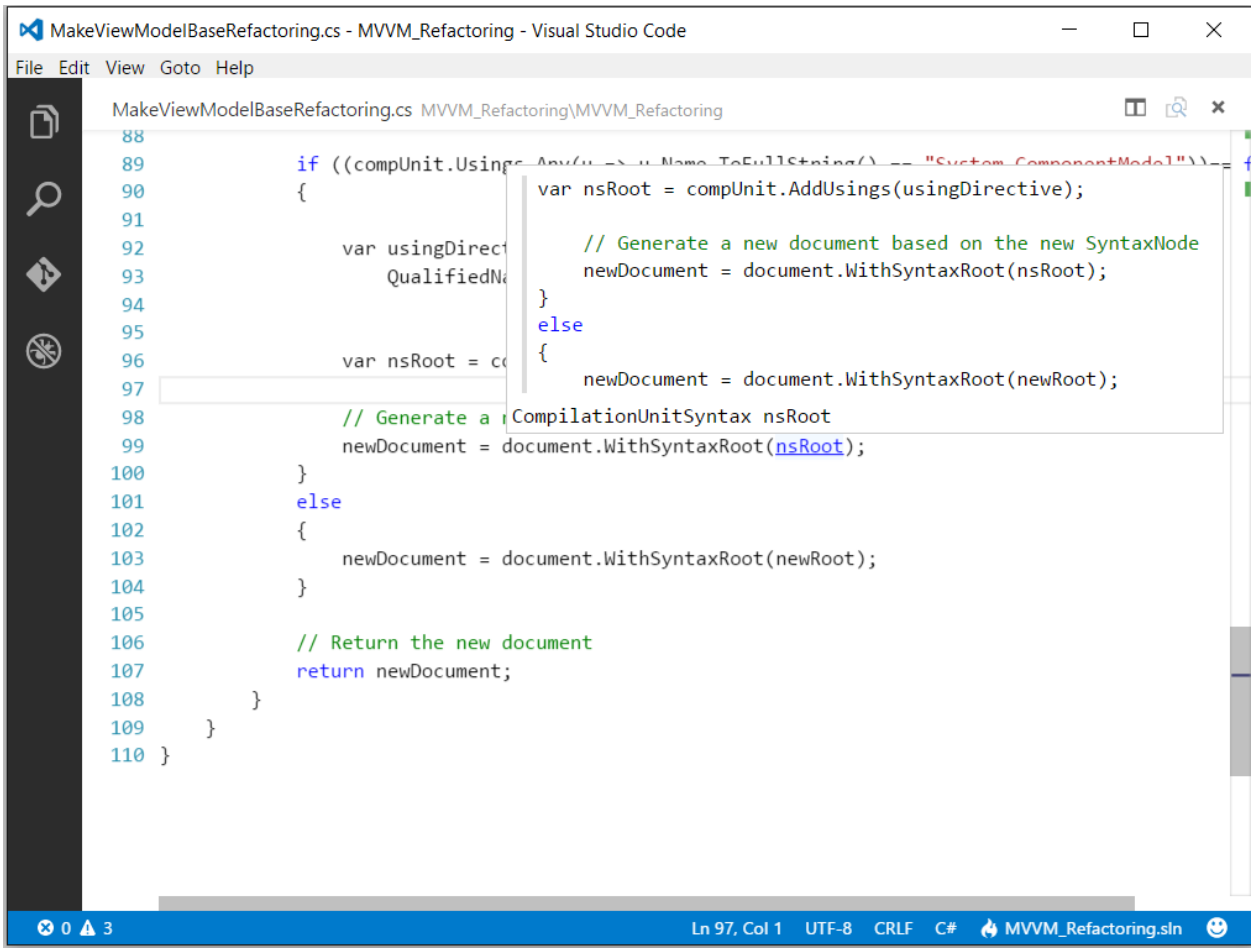


Figure 7: Using Go To Definition to discover a symbol's definition in code.

Also, you can hold Ctrl and click on the symbol and Code will open the definition directly in the file that contains the related code.

Tip: Alternatively, you can press F12 or right-click and select Go To Definition.

Tooltips

You can hover over types, variables, and type members, and Visual Studio Code will show a tooltip that contains the documentation for the selected object. This feature is available only if the documentation has been provided, such as XML comments for C#. Figure 8 shows an example where you can see a descriptive tooltip of a type called **`Document`**.

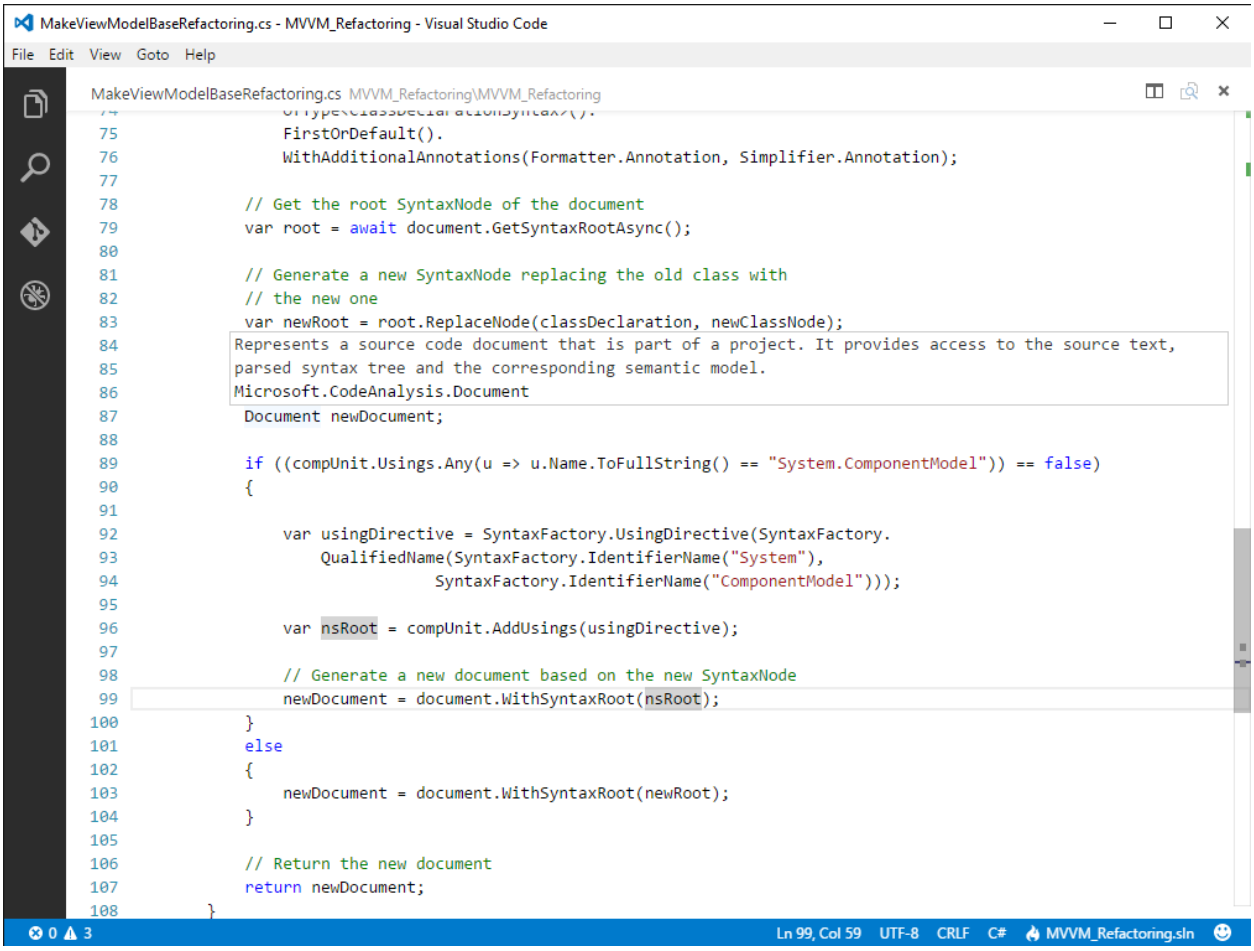


Figure 8: Getting type and member documentation with tooltips.

If you instead hover over a variable name, the tooltip will show the type for the variable.

Find All References

Find All References is a very useful feature that makes it easy to see how many times and where a member has been used across the code. For each member, the code editor shows the number of references found. If you click this number, the code editor displays a pop-up containing the code where the first occurrence is found, while the right side the pop-up shows the list of occurrences, as shown in Figure 8.

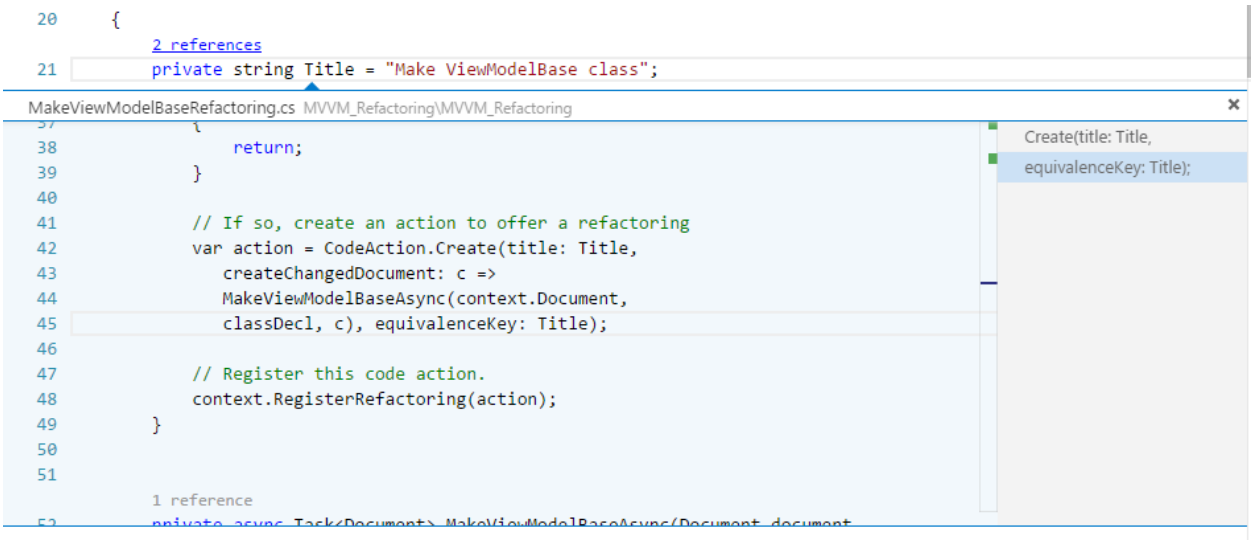


Figure 9: See where and how many times members have been used with Find All References

Just click on each occurrence in the list to see the code that has a reference to it. It is very important to note that this pop-up is interactive, which means that you can edit the code directly without having to open the containing code file separately. This allows you to keep your focus on the code, saving time. Also notice that the interactive pop-up shows the file name that contains the selected reference at the top.

Tip: You can also enable Find All References by pressing Shift+F12 or by right-clicking and then selecting Find All References.

Rename Symbol

Renaming a symbol is a common editing 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** command, a small interactive pop-up appears (see Figure 10). There you can write the new name without any dialogs, keeping your focus on the code.

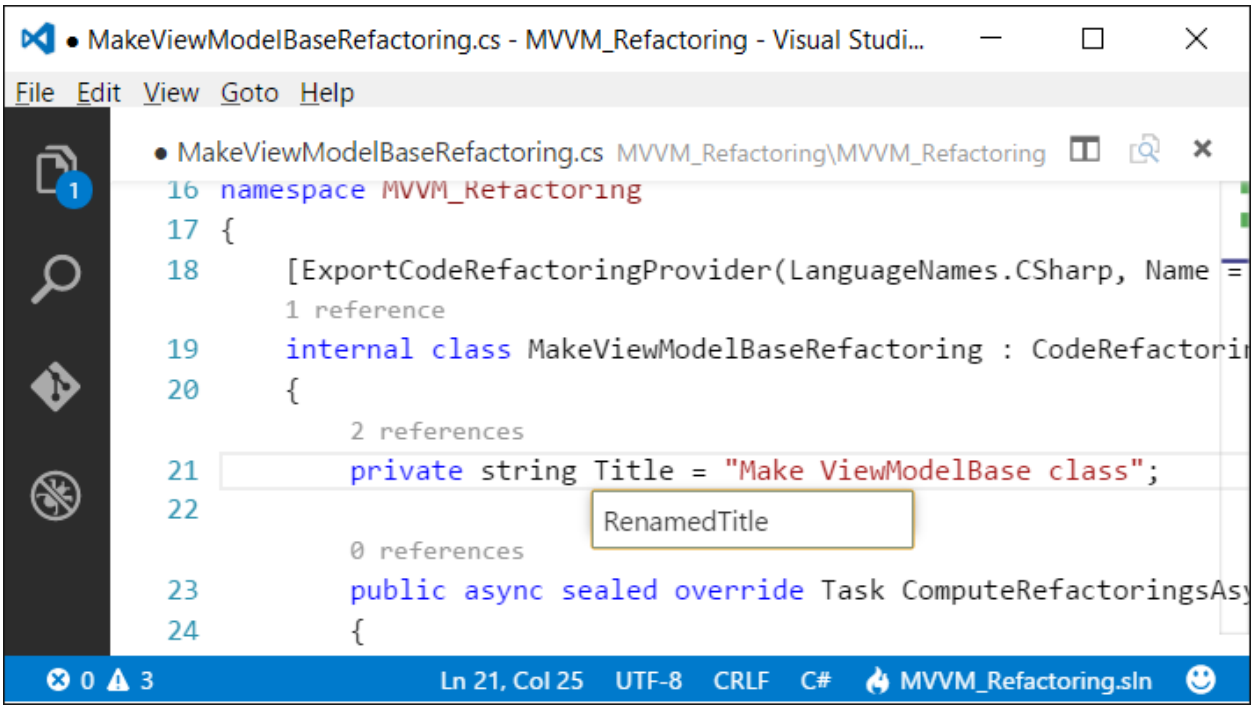


Figure 10: Renaming a Symbol In-Line

Additionally, you can rename all the occurrences of an identifier. Just right-click the identifier and then select **Change All Occurrences** (or press Ctrl+F2); all the occurrences will be highlighted and updated with the new name as you type (see Figure 11).

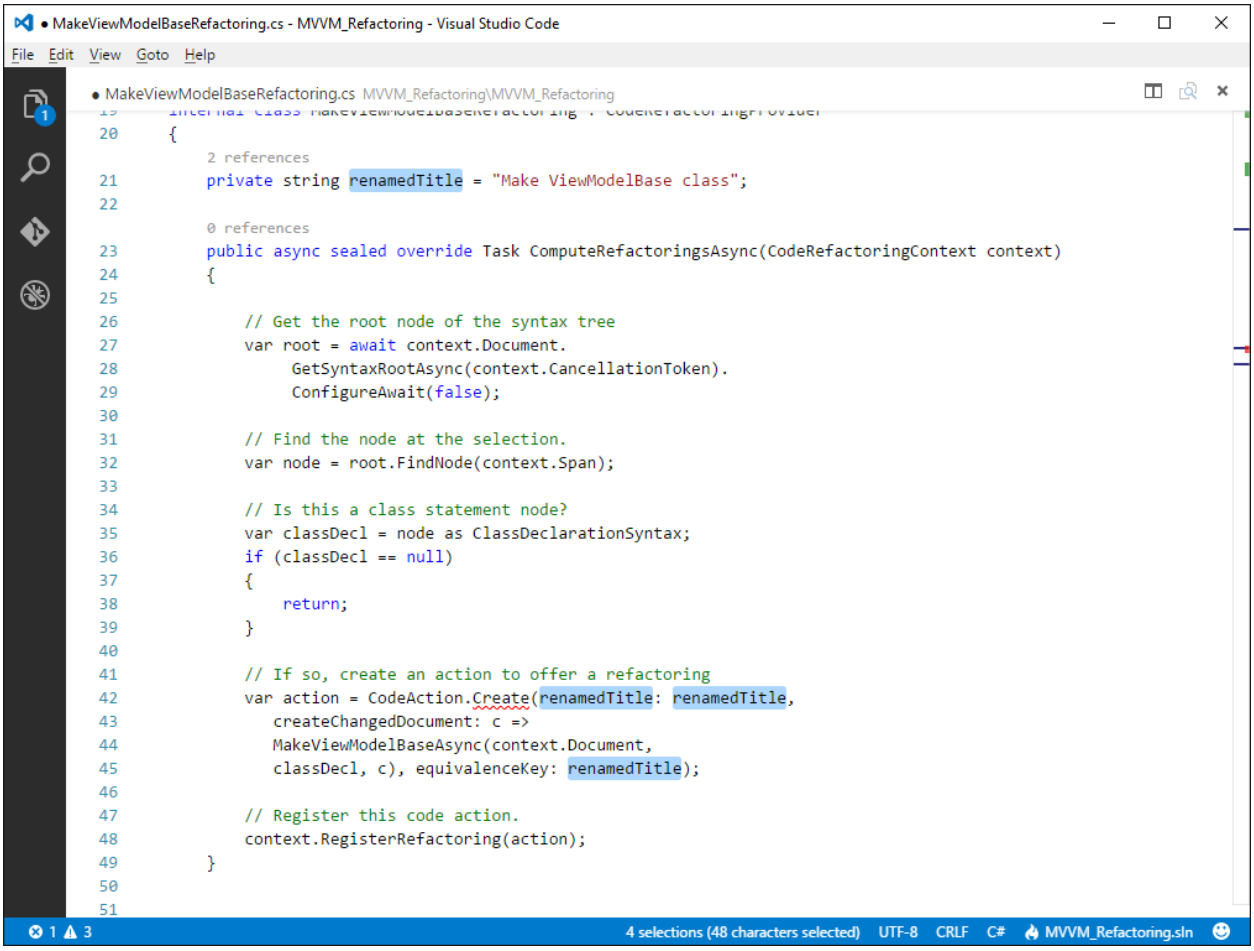


Figure 11: Changing All Occurrences of an Identifier

Peek Definition

Suppose you have dozens of code files and you 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 would also move your focus away from the original code. Visual Studio Code brilliantly solves this problem with a feature called **Peek Definition**. You can simply right-click a type name and then select **Peek Definition** (the keyboard shortcut is Alt+F12). An interactive pop-up window appears showing the code that defines the type, giving you not only an option to look at the code, but also the option to directly edit it. Figure 12 shows the peek window in action.

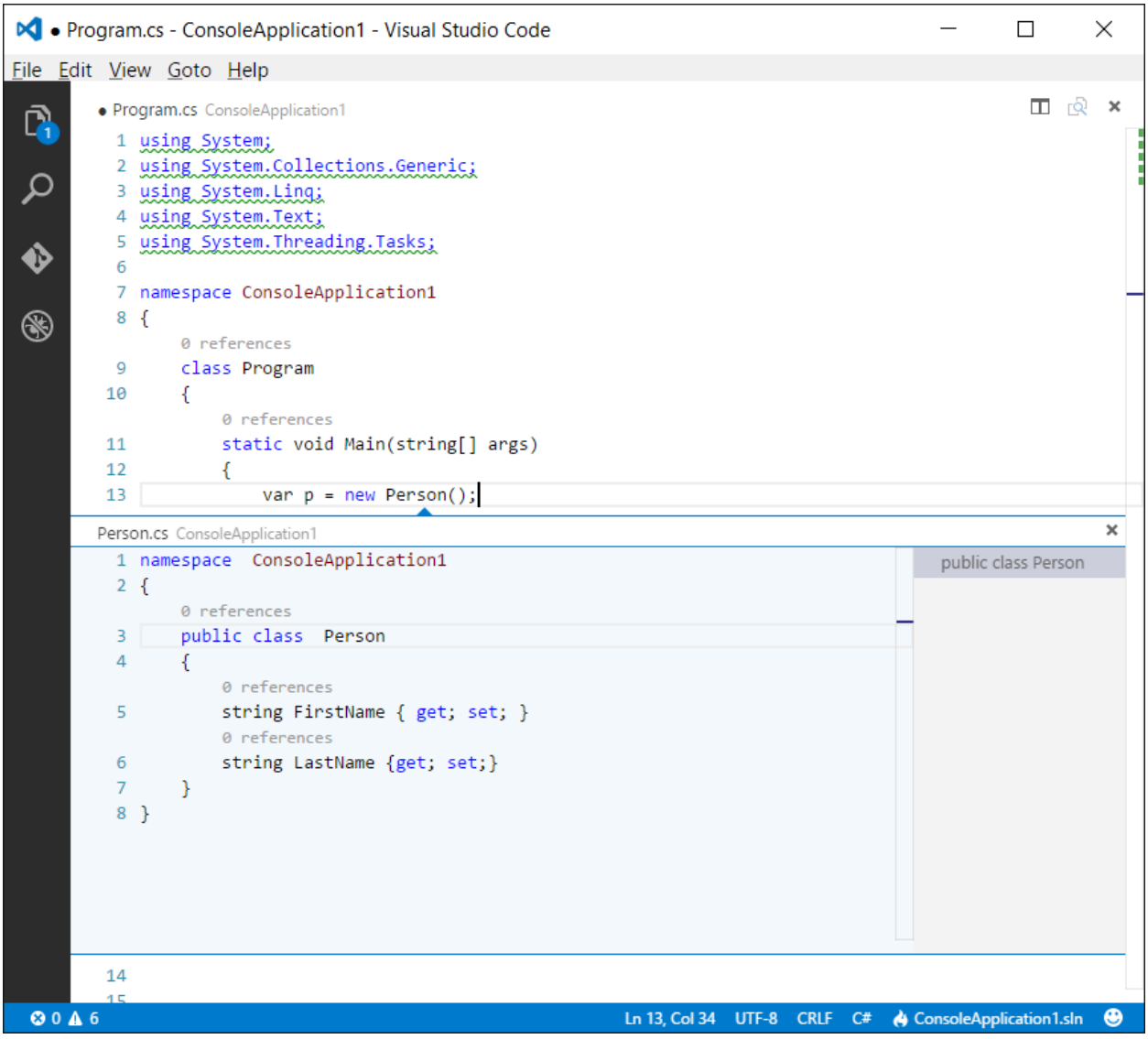


Figure 12: Investigating a Type Definition with Peek Definition

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

Fast Selection

You can press Ctrl+D to select a word or identifier at the right of the cursor.

Multi-Cursors

Visual Studio Code's editor supports multi-cursors. Each cursor operates independently and you can add secondary cursors by holding Alt and clicking at the desired position. You will see that secondary cursors are rendered thinner. The most typical situation in which you want to use multi-cursors is when you want to add (or replace) the same text in different positions of a code file. Figure 13 shows an example.

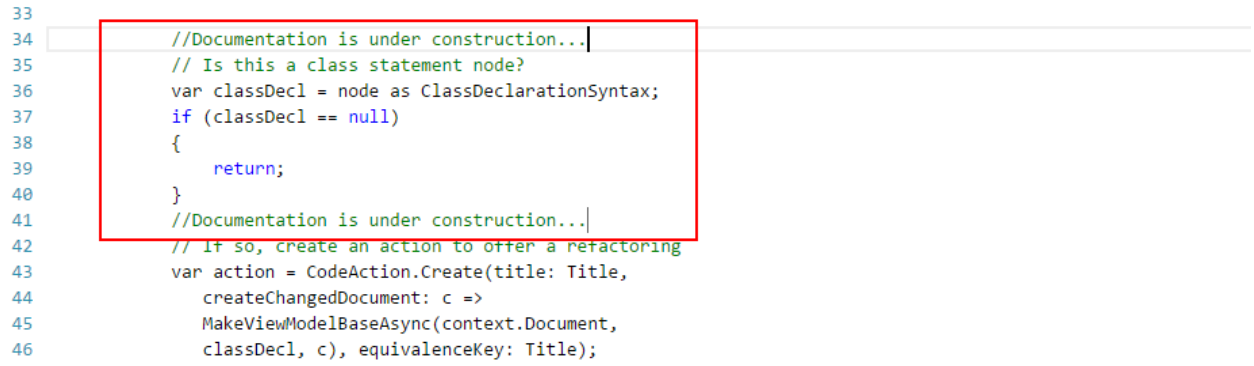


Figure 13: Multi-cursors can be used to add the same text in multiple positions.

The Change All Occurrences feature you saw before automatically uses multi-cursors too.

Goto Symbol

Another interesting and powerful feature is **Goto Symbol**. By pressing Ctrl+Shift+O, you will be able to browse symbols in the current code file. Also, if you type **:** in the search box, symbols will be grouped by category. Figure 14 shows how to browse symbols grouped by category.

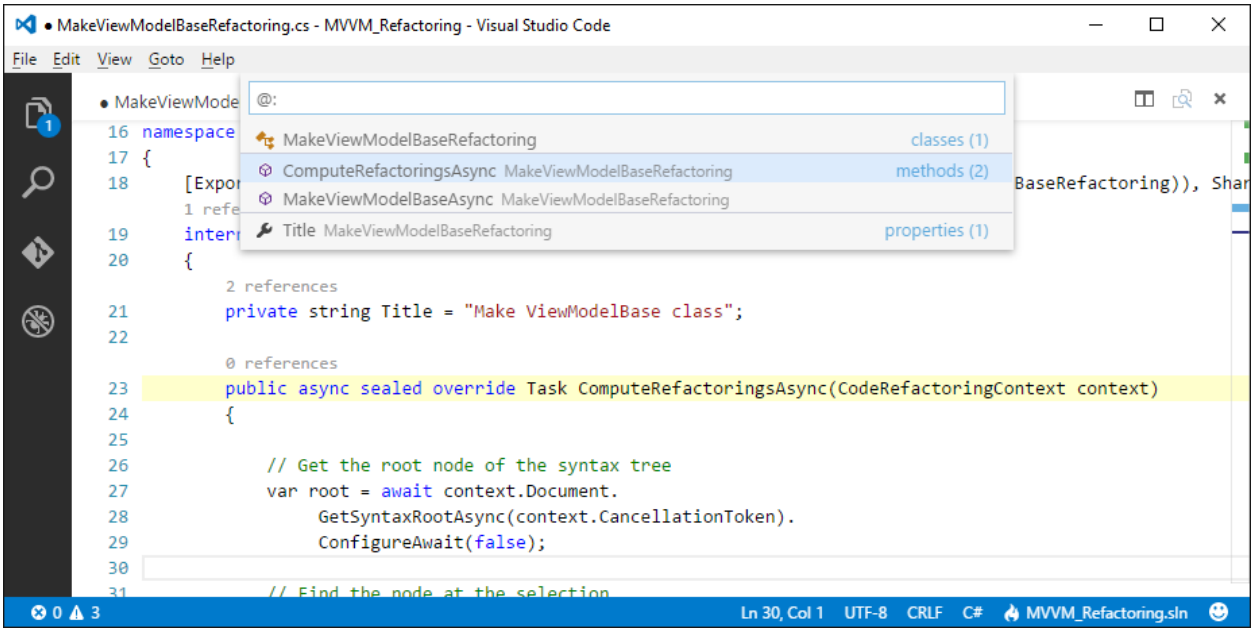


Figure 14: Browsing Symbols

Use the up and down arrow keys to highlight the corresponding symbol definition in the code file. In addition, the symbol list is automatically filtered based on what you type in the search box.

Open Symbol By Name

The C# and TypeScript languages support opening a symbol by name, regardless of the code file that contains the symbol. Simply press Ctrl+T and start typing the symbol name at the # prompt, as shown in Figure 15.

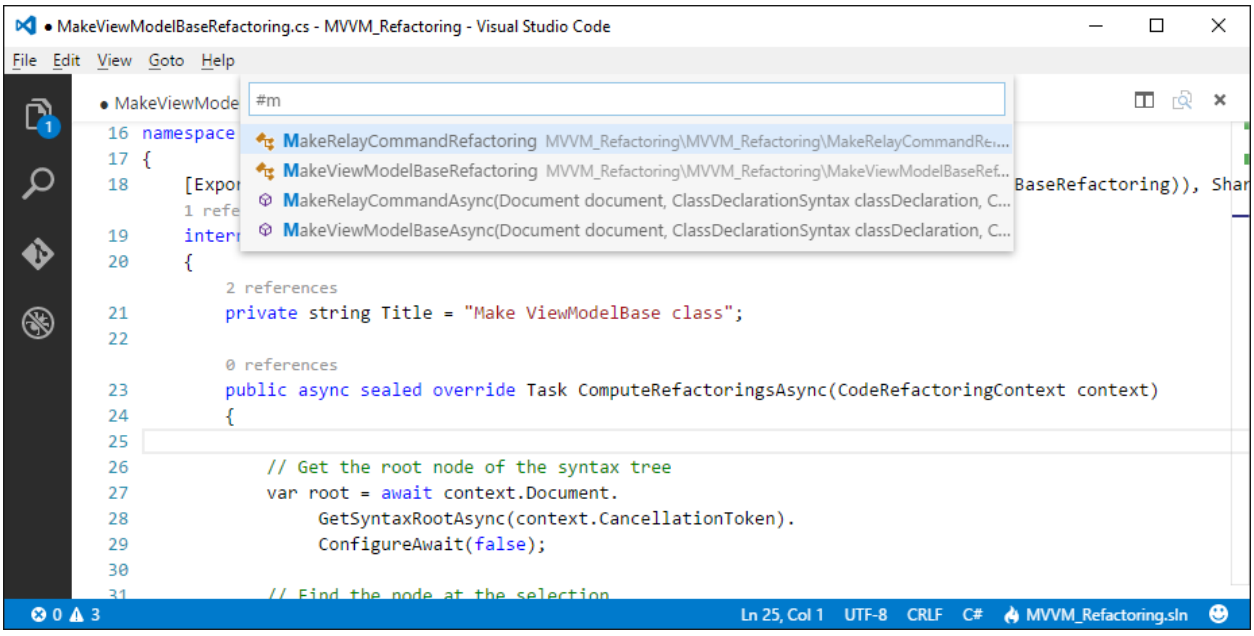


Figure 15: Browsing Symbols by Name

If you click the name of a symbol that is defined inside a different code file, this will be opened inside a new editor window.

Shrinking and Expanding Text Selection

You can easily expand text selection by pressing Shift+Alt+Right Arrow, and shrink text selection by pressing Shift+Alt+Left Arrow within enclosing delimiters of a code block.

Code Issues and Refactoring

With C# and TypeScript, 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.

Note: This feature is fully available to C# and TypeScript however there is some basic support for JavaScript and CSS. This section describes the full functionality based on some C# code.

According to the severity of a code issue, Visual Studio Code underlines the pieces of code that need your attention with squiggles. Green squiggles mean a warning; red squiggles mean an error that must be fixed. If you hover over the line or symbol with squiggles, you will get a tooltip that describes the issue. Figure 16 shows two code issues, one with green squiggles for an unnecessary **`using`** directive, and one with red squiggles for a symbol that does not exist. The figure also shows the tooltip for the code issue with the higher severity level.

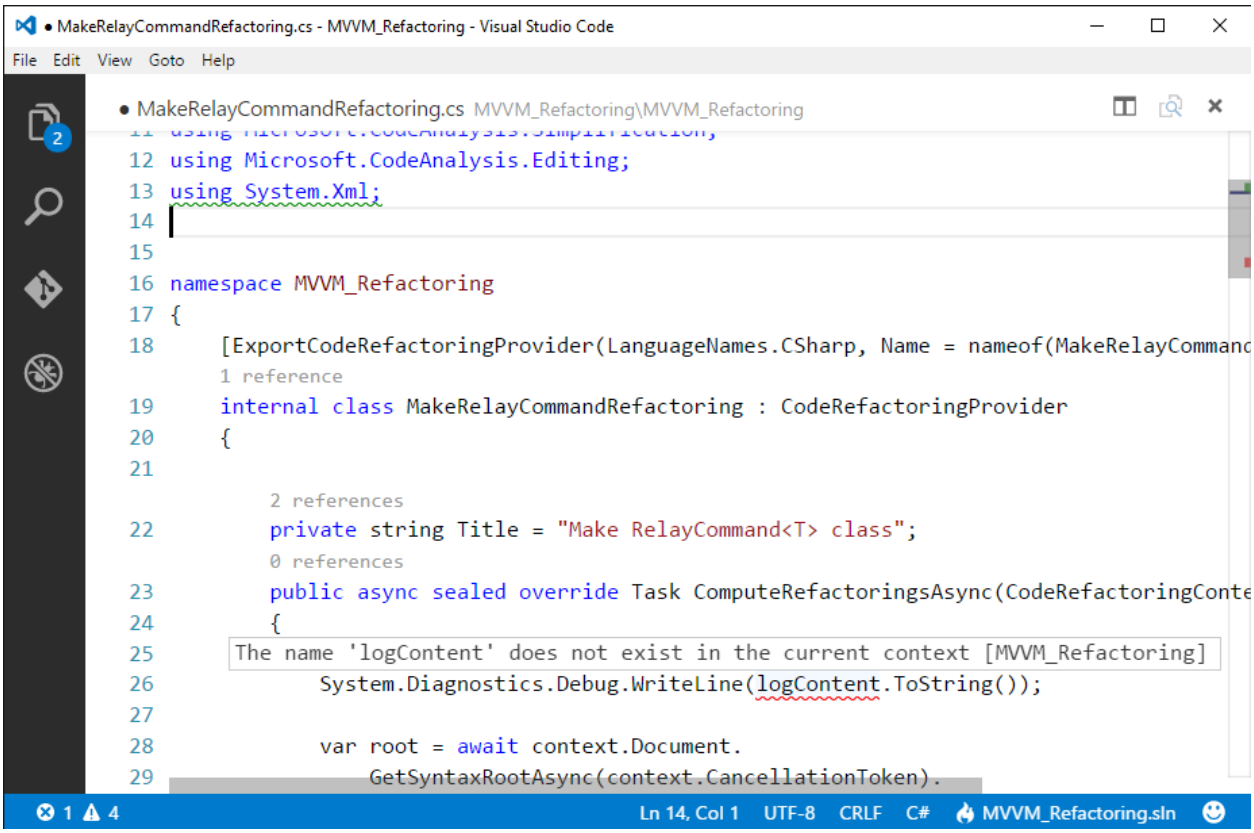


Figure 16: Visual Studio Code detects issues as you type.

Of course, Visual Studio Code also provides an integrated, easy way to fix code issues via the so-called Light Bulb shown in Figure 17 under the unnecessary **`using`** directive.

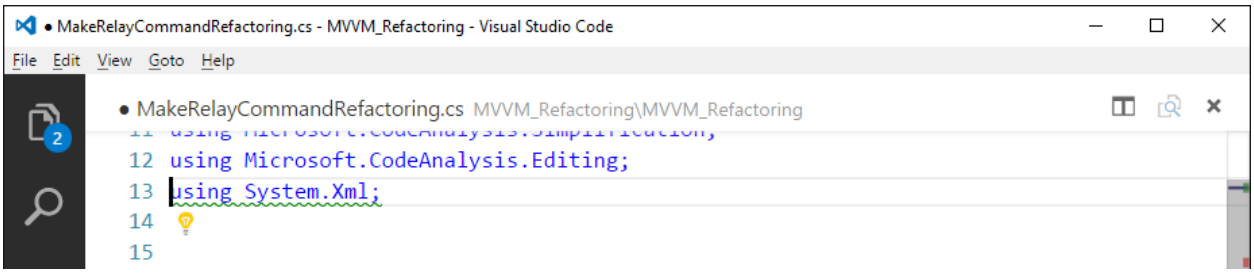


Figure 17: The Light Bulb provides a shortcut for resolving code issues.

When you click the Light Bulb, Visual Studio Code shows possible code fixes for the current context. In this case, it allows you to remove unnecessary directives and to sort directives, as shown in Figure 18.

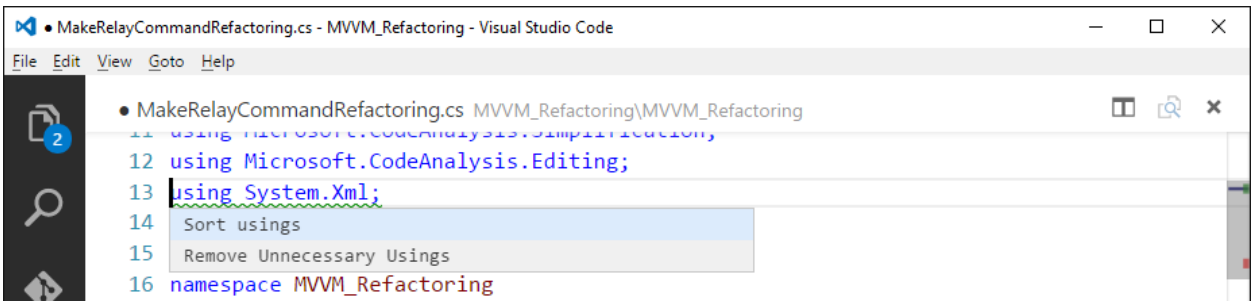


Figure 18: Fixing code issues with the Light Bulb.

Just click the desired fix and it will be applied to the code issue. Actually, this tool is much more powerful. Consider the symbol called **`logContent`** that does not exist in the code file and is underlined as a code issue as expected (see Figure 16). If you click this symbol and then open the Light Bulb, the code editor will show a list of possible proper fixes for the current context, such as creating a field, creating a property, creating a local variable, and more, as shown in Figure 19.

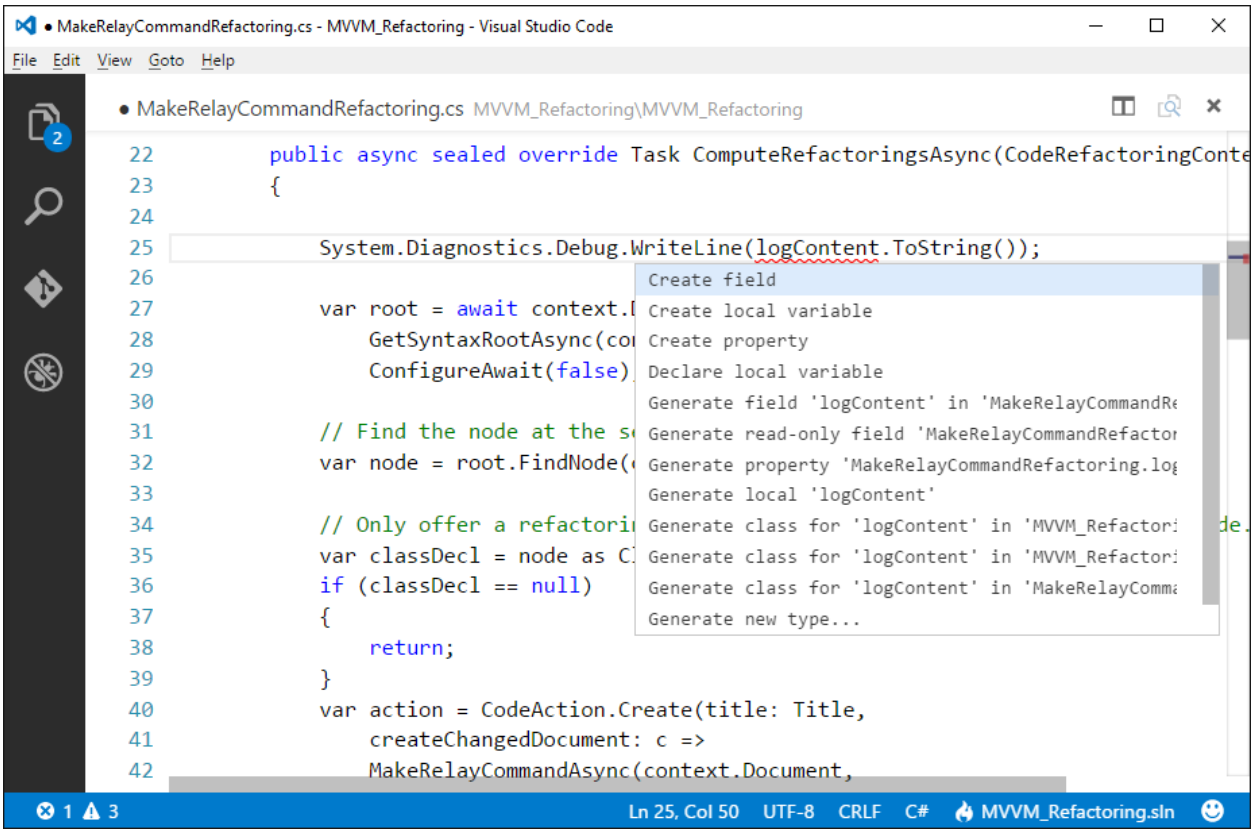


Figure 19: The Light Bulb provides proper code fixes based on the context.

For instance, if you select the **Create property** option, Visual Studio Code generates a property stub for you, as shown in Figure 20.

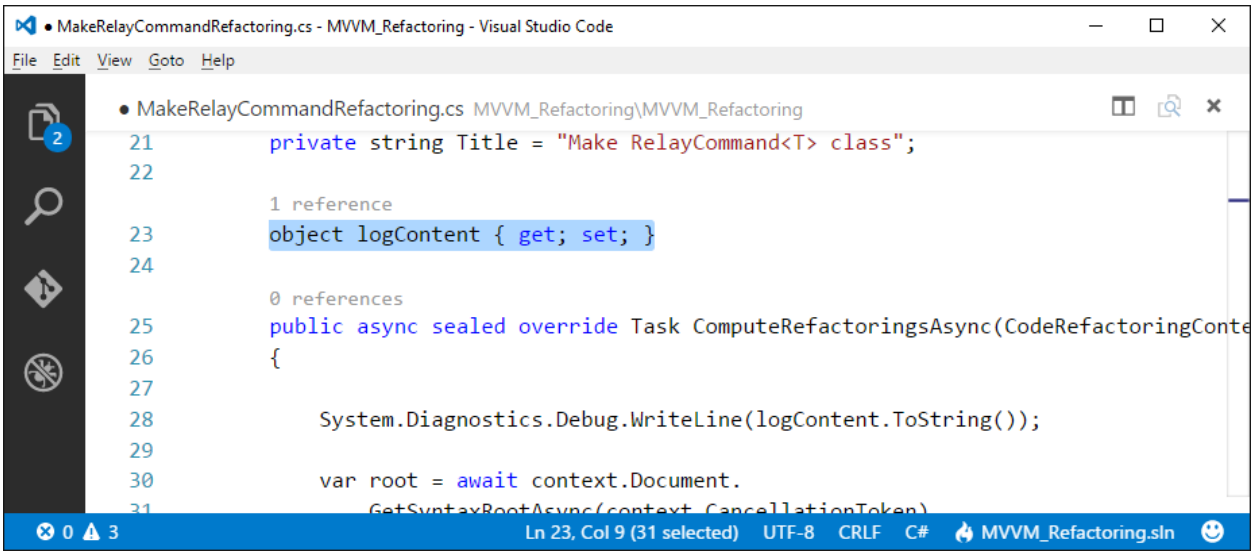


Figure 20: The property has been generated as expected.

There's much more power than this. Consider Figure 21, where you see a class that needs to implement the **`IDisposable`** interface. As you can see, there is a code issue because the code editor cannot find the definition for this interface, so it provides possible fixes.

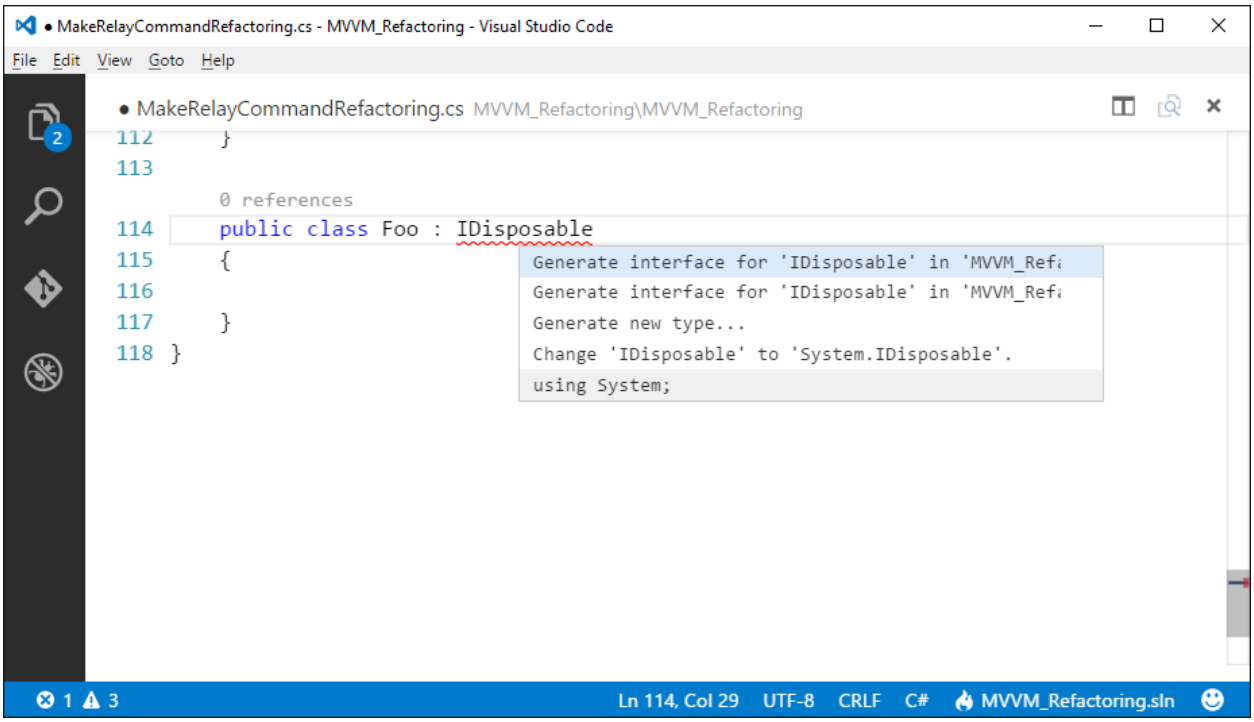


Figure 21: Getting Possible Fixes for a Missing Interface Implementation

We know this interface is from the **`System`** namespace, so the proper fix here is the **`using System;`** option. When selected, Visual Studio Code adds a **`using System;`** directive to the code file. However, this is not enough to solve the code issue because the class actually lacks the interface implementation. If you click the Light Bulb again, you will see how the code editor offers to implement the interface for you based on four different choices, as demonstrated in Figure 22.

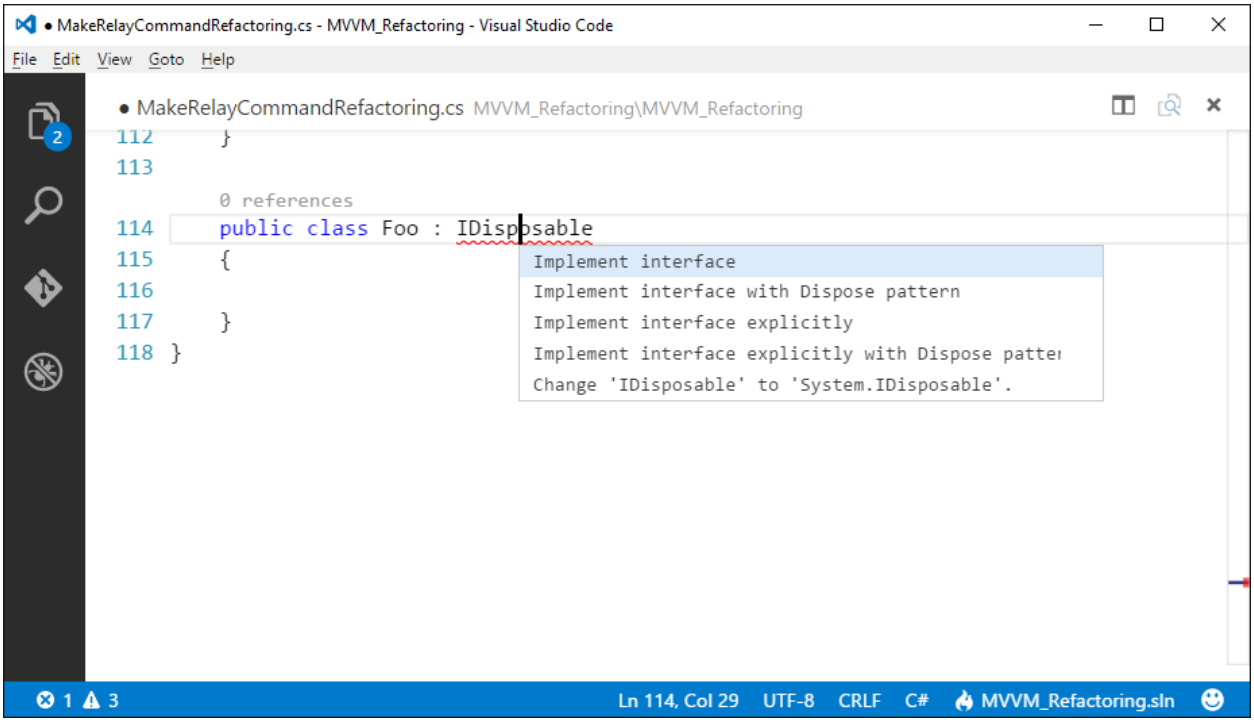


Figure 22: Contextualized Suggestions for the Interface Implementation

If you select the **Implement interface with Dispose pattern**, you will see how Visual Studio Code supplies all the plumbing code for a correct interface implementation, as shown in Figure 23.

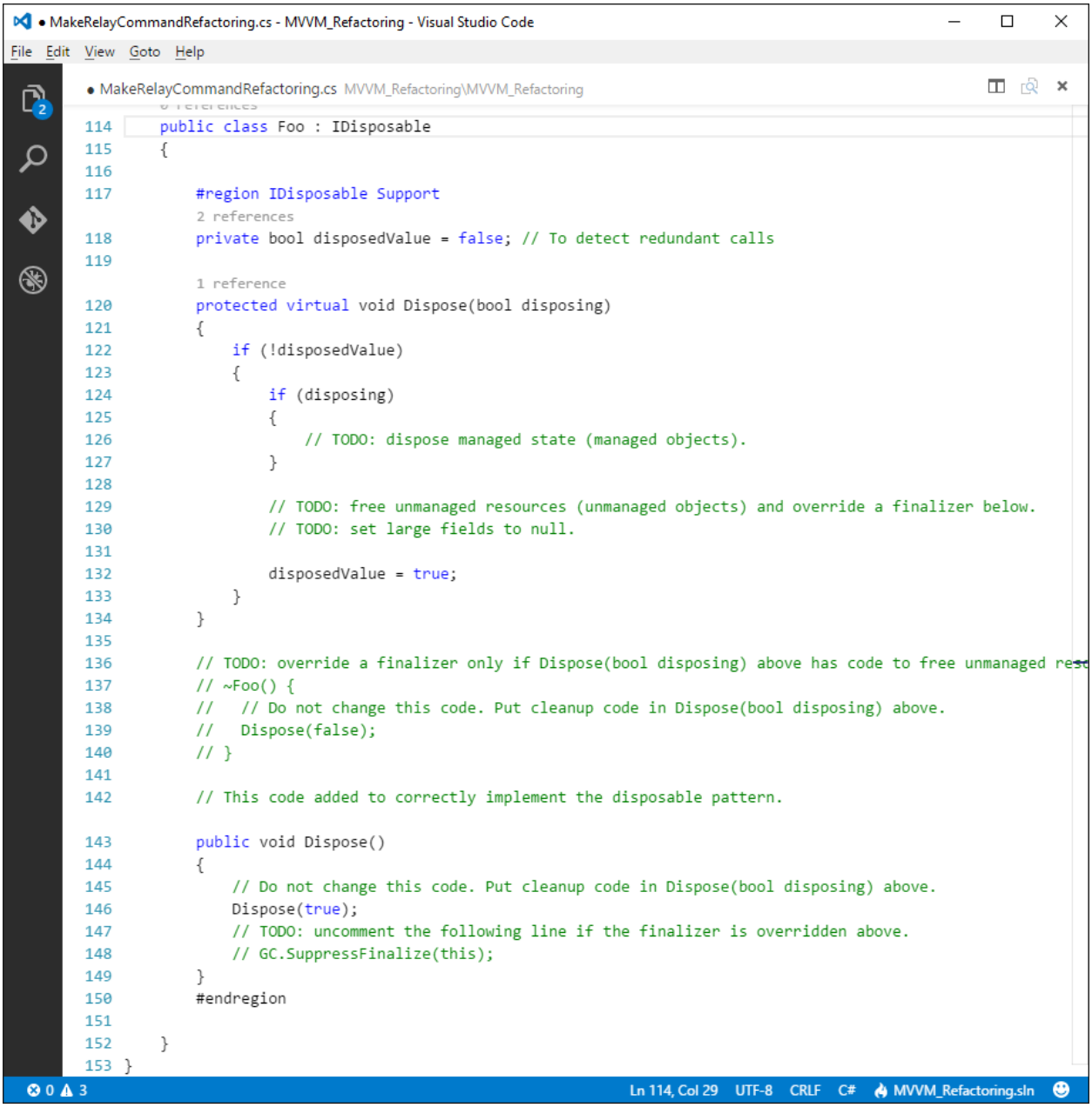


Figure 23: Visual Studio Code applies the code fix, adding all the plumbing code.

If you chose one of the other code fixes, you would get a similar result but with different implementation. Though it is not possible to show examples for all the code fixes that Code can apply, what you have to keep in mind is that suggestions and code fixes are based on the context for the code issue, which is a very powerful feature that makes Visual Studio Code a unique editor. Once you know how to work with the Light Bulb, you will see on your own possible fixes depending on the code issues you encounter. Visual Studio Code also provides an alternative way to see all the code issues it finds. Select **View** > **Errors and Warnings** (Ctrl+Shift+M) or click the **Errors and Warnings** symbols at the lower left corner. This opens a list of code issues (see Figure 24) that you can also filter by typing into the search box. Then you can simply click a code issue and move to it immediately.

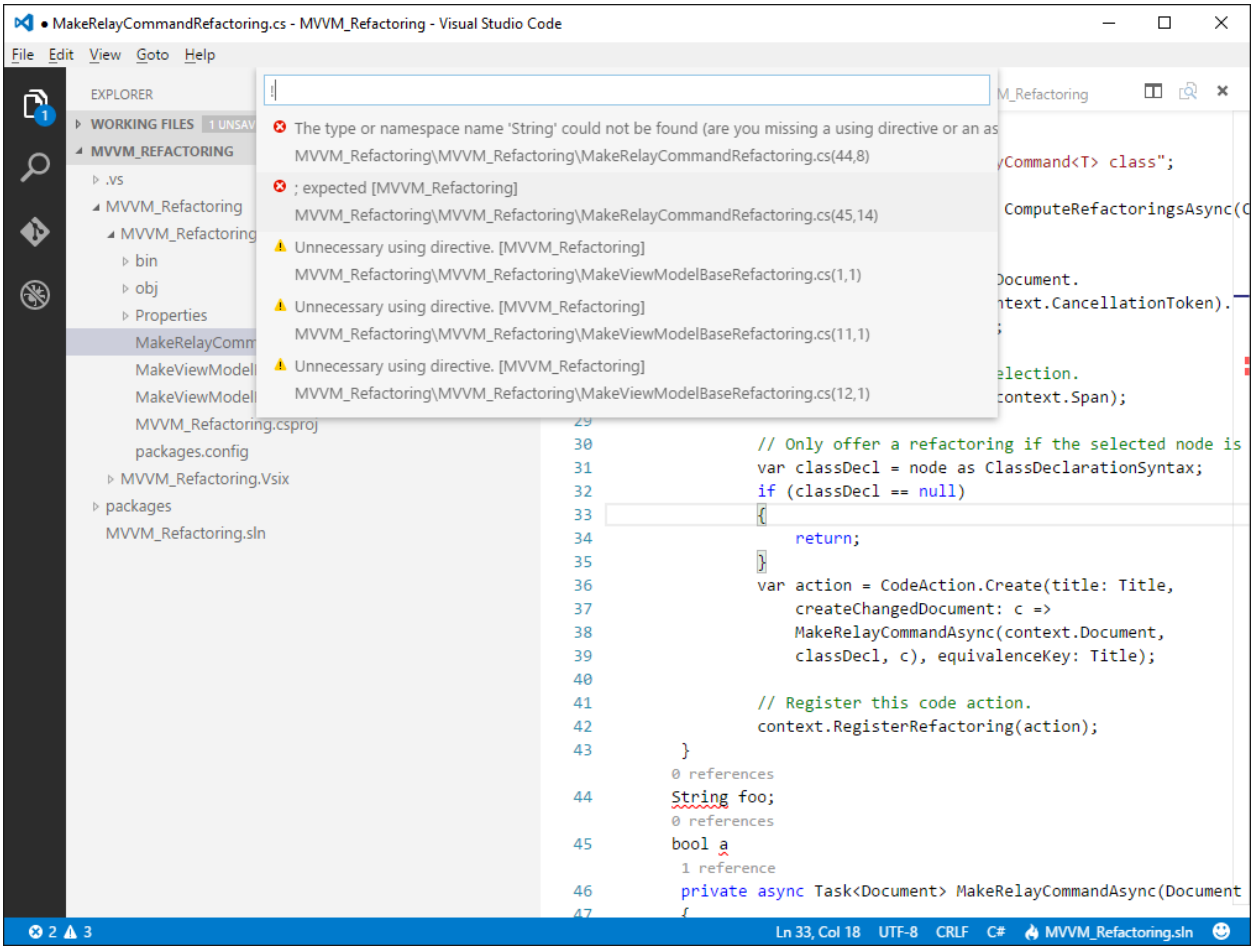


Figure 24: Showing the List of Code Issues

Note: Behind the scenes, this feature is based on the .NET Compiler Platform, also known as Roslyn. Roslyn provides open source C# and Visual Basic compilers with rich code analysis APIs. Visual Studio Code does not work with compilers like Visual Studio does, but it calls a special language service that performs code analysis as you type. Visual Studio 2015 heavily uses the Roslyn APIs to empower the code editor with even more advanced features, such as live preview. If you want to discover more about Roslyn with .NET, consider reading Roslyn Succinctly.

Preview for Markdown

Markdown is a popular markup language used to write documents for the web. If you have ever worked on open source projects hosted on the popular GitHub platform, you've probably written some Markdown documents. Visual Studio Code has integrated support for editing Markdown files (.md), as well as live previews. Figure 25 shows the editing of a Markdown document.

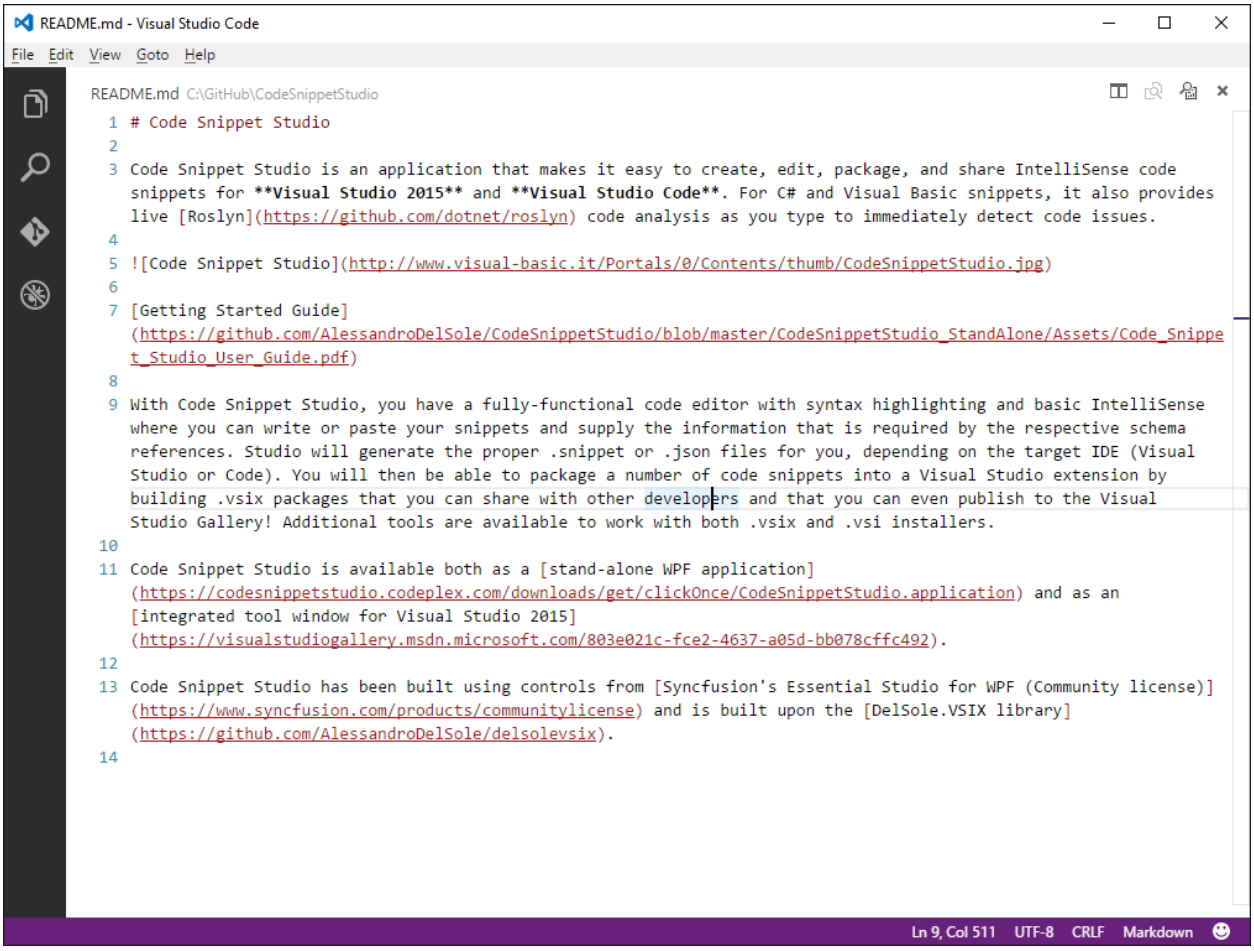


Figure 25: Writing a Markdown document.

If you press Ctrl+Shift+V, Visual Studio Code shows a live preview of how the document will be rendered, as shown in Figure 26.

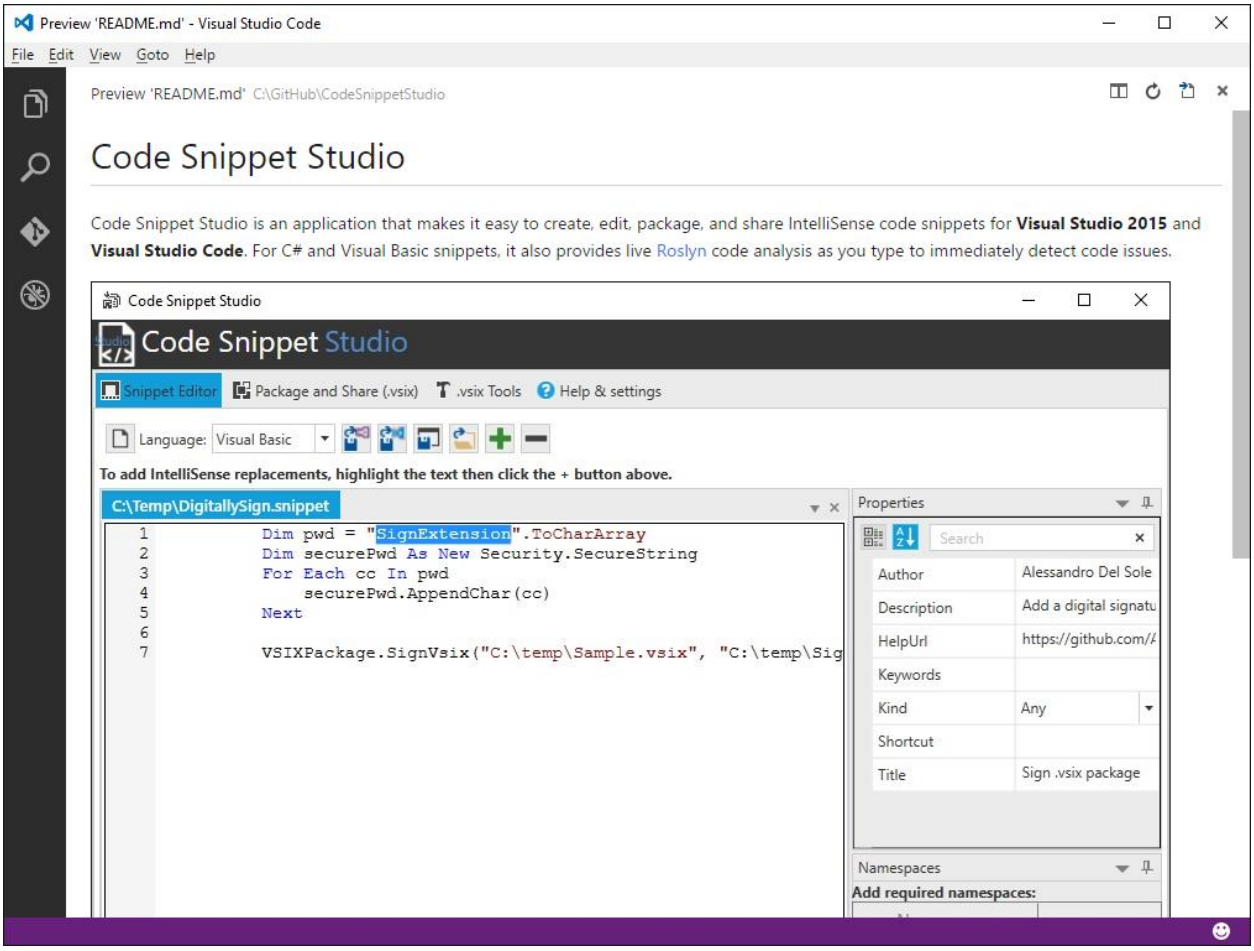


Figure 26: Markdown Live Preview

This is a very useful feature that gives you an immediate view of what the document will look like.

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. Table 1 lists what languages support code snippets natively. For instance, in C# you can easily add a property definition by using the **prop** code snippet, as shown in Figure 27.

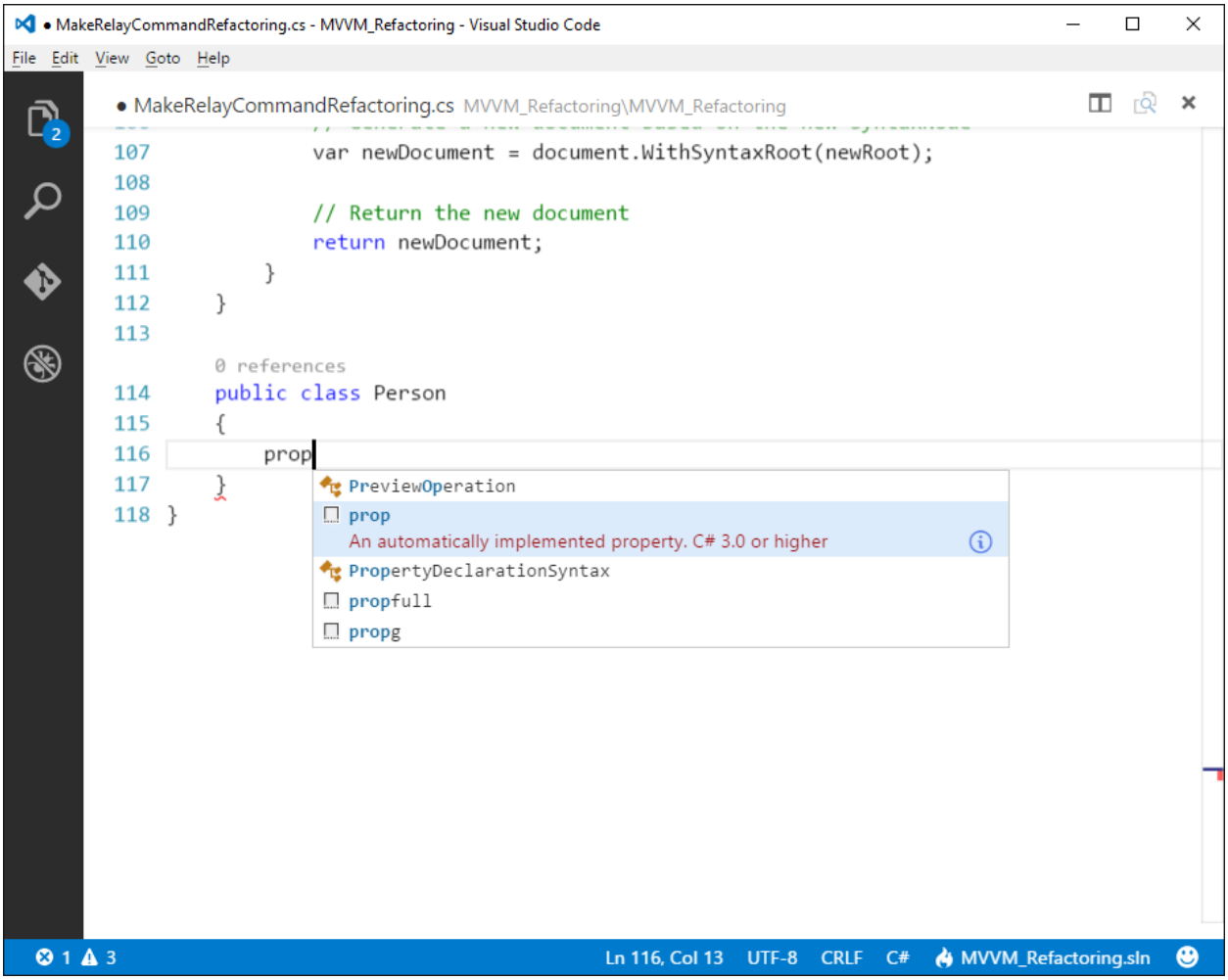


Figure 27: Inserting a Code Snippet Using Emmet Abbreviations

Code snippets are available in the IntelliSense pop-up as you type. They're recognizable by the small white sheet icon. Pressing Tab over the snippet in the previous figure produces the result shown in Figure 28.

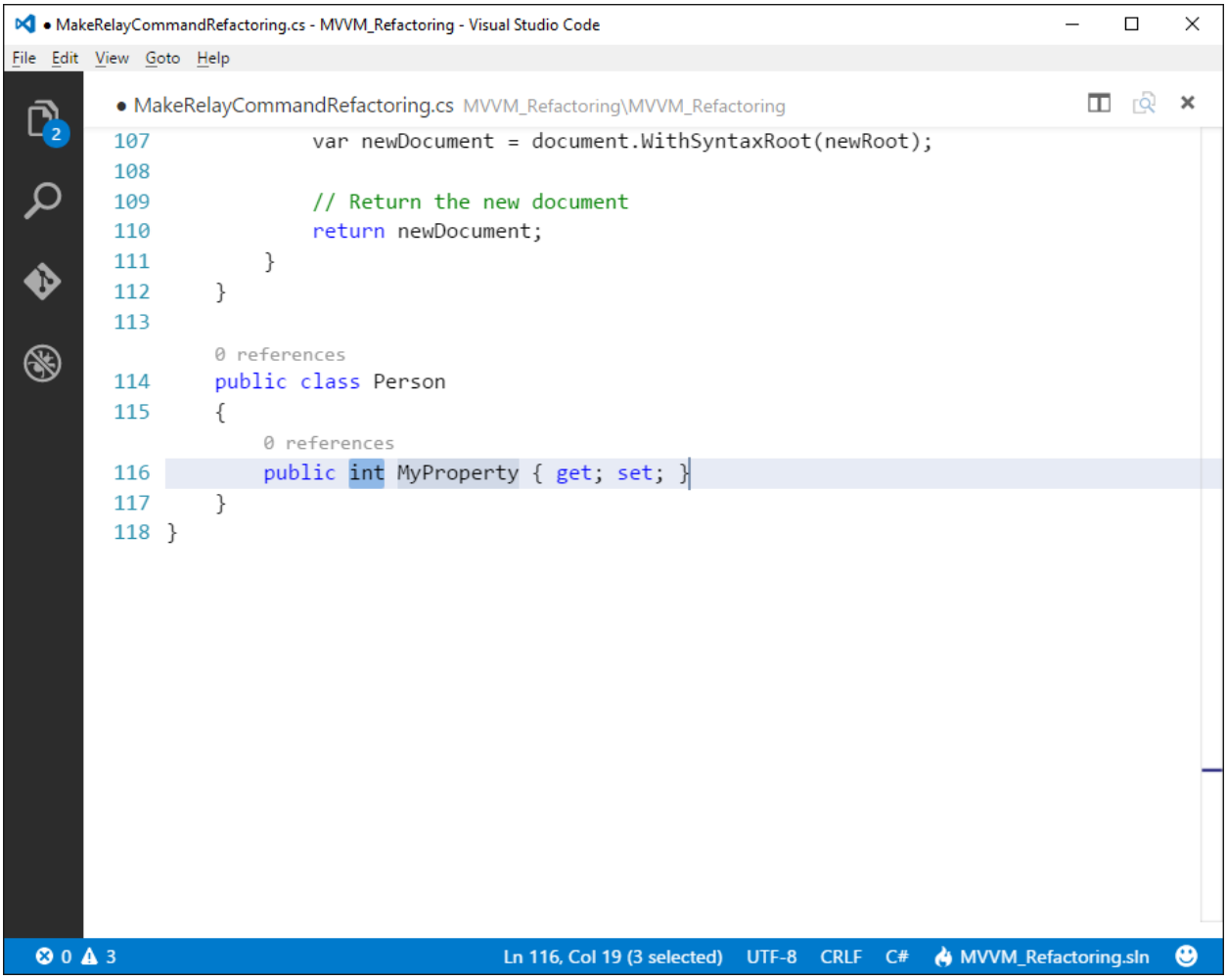


Figure 28: A Property Added via a Code Snippet

As another example, with JavaScript code files you can use the **define** snippet (see Figure 29) to add a module definition. This code snippet produces the result shown in Figure 30.

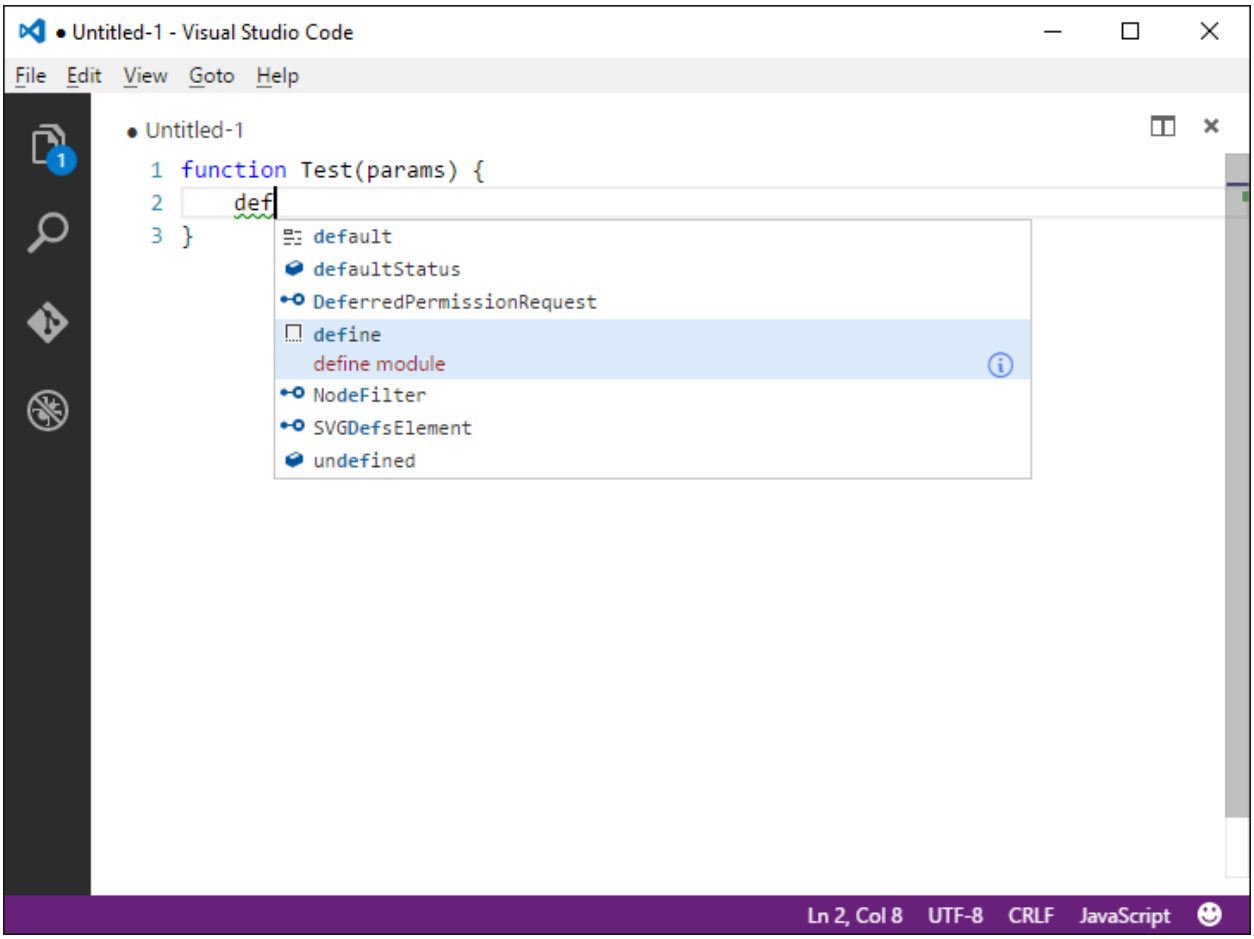


Figure 29: Adding a JavaScript Code Snippet

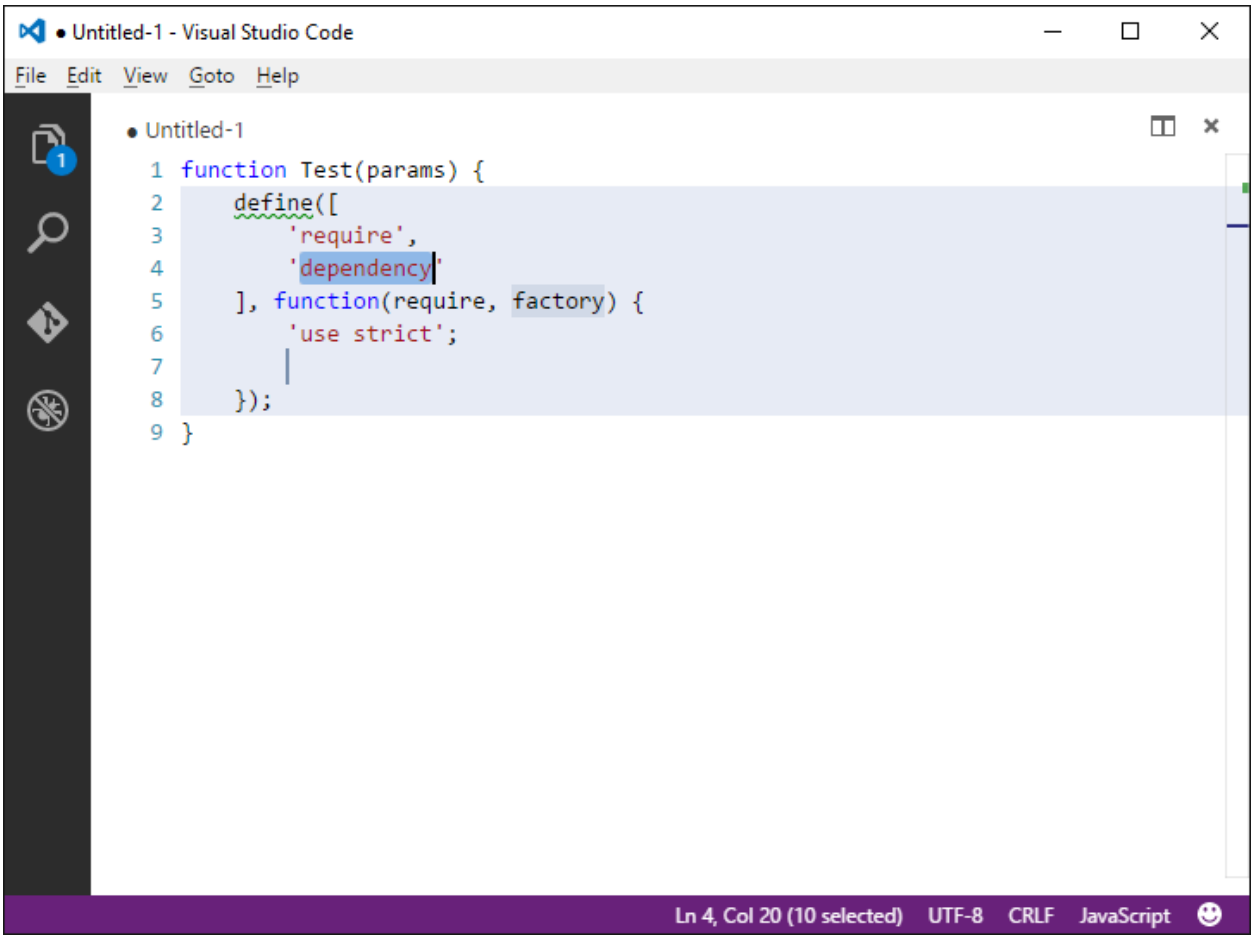


Figure 30: A JavaScript Module Definition Added via a Code Snippet

Visual Studio Code is not limited to built-in code snippets. You can download code snippets produced by the developer community for many languages, and you can even create and share your own snippets. This is discussed in Chapter 5, "Customizing and Extending Visual Studio Code".

Working with files and folders

Visual Studio Code is based on files and folders. This means that you can open one or more code files individually, 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:

- package.json
- project.json
- tsconfig.json
- .sln or .xproj Visual Studio solution and project files for ASP.NET Core

If Code finds one of these files, it is able to organize the file structure into a convenient editing experience and 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. The sections that follow describe how to work with single files and folders in Visual Studio Code, plus how Code manages projects.

Working with single code 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**, or by pressing Ctrl+O. On Windows, you can also right-click a file name in File Explorer and select **Open with Code** if you enabled that option when you installed VS Code. Visual Studio Code automatically detects the language for the code files and enables the proper editing features. Of course, you can certainly open more files and easily switch between files by pressing Ctrl+Tab. As you can see in Figure 31, a convenient pop-up shows the list of open files; by pressing Ctrl+Tab you will be able to browse files, and when you release the keys the selected file will become the active editing window.

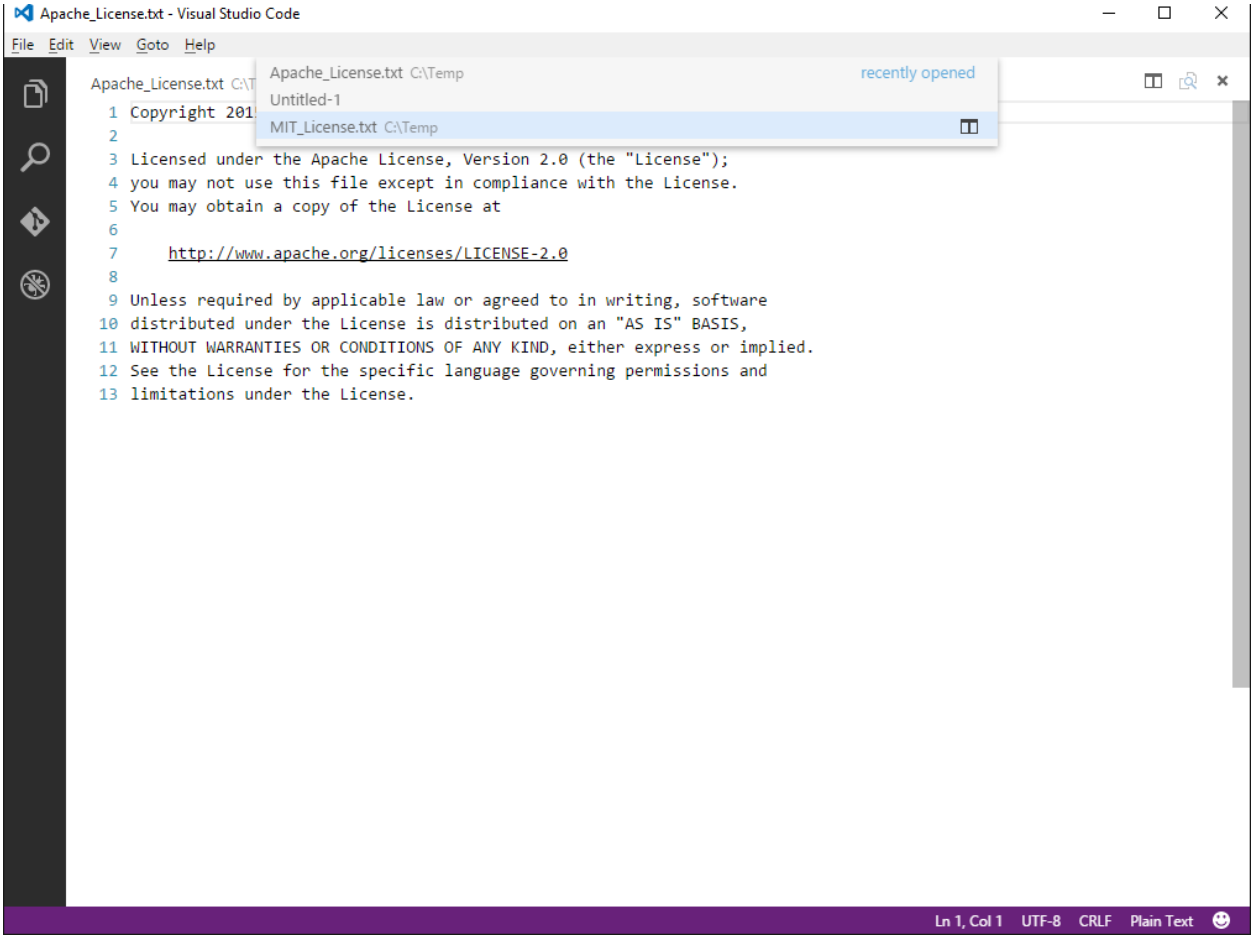


Figure 31: Switching between Code Files

Tip: Visual Studio Code also features the Explorer, which provides a structured, organized view of currently opened files. This is discussed in Chapter 2, "The Workspace and the User Interface."

An editor can be closed by using the **Close** button at the upper right corner, or by using the **Close All Files** command in the **File** menu.

Creating a new code file and language selection

You create a new file by clicking **File** > **New File**, or by pressing **Ctrl**+**N**. 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 lower right corner of VS Code, near the smile symbol. In this case, you will see Plain Text as the current mode, so click it. As you can see in Figure 32, you will be presented with a list of supported languages where you can select the new language for the current file. You can also start typing a language name to filter the list.

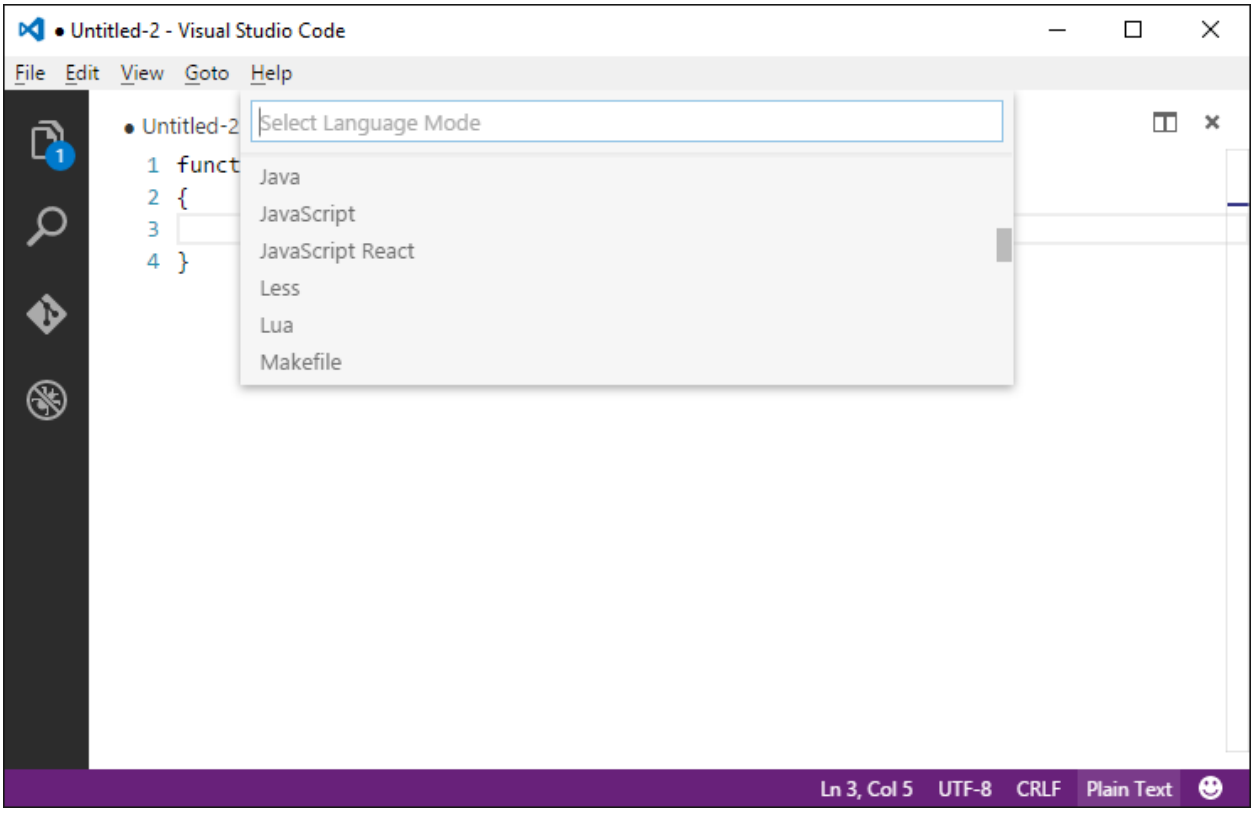


Figure 32: Switching between Code Files

When you select a new language, the Select Language Mode pop-up is updated with the selected language and the editor enables the proper features it, as shown in Figure 33 where the syntax colorization and IntelliSense is based on selecting JavaScript.

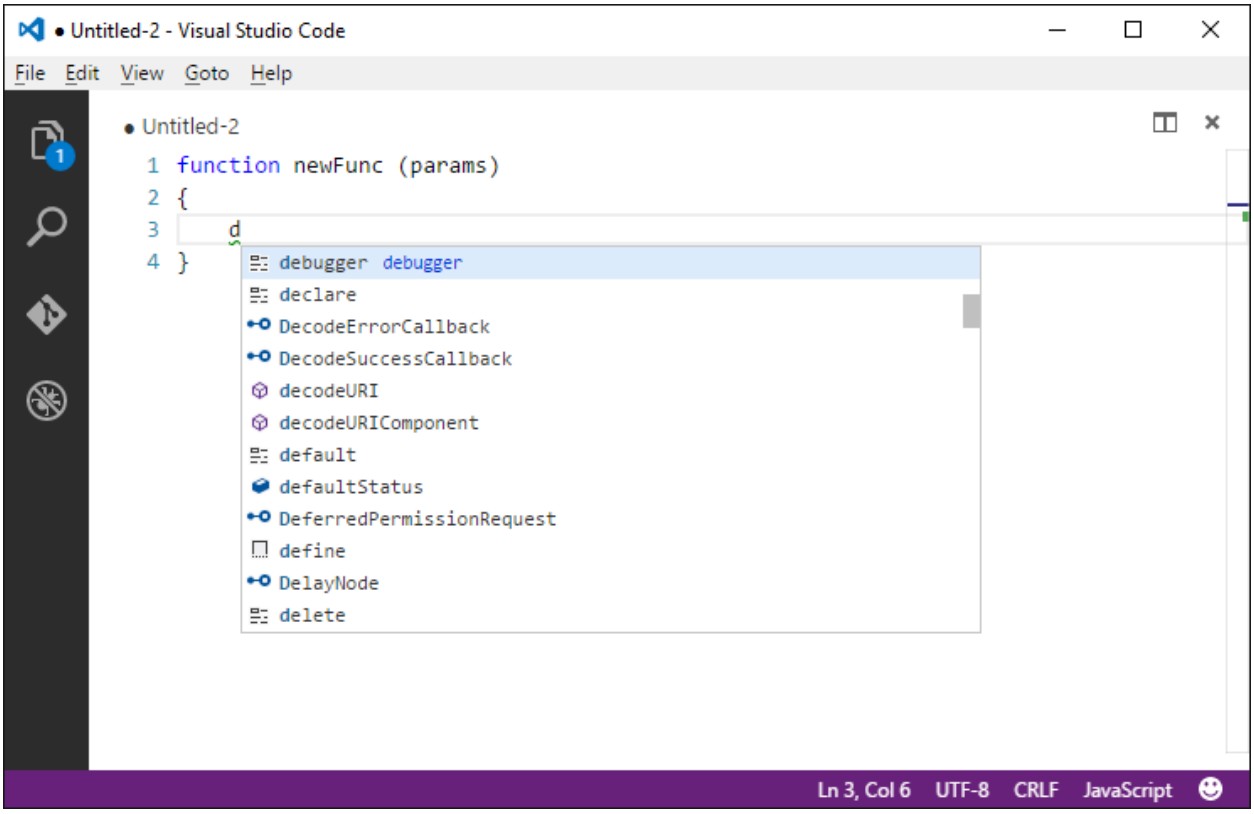


Figure 33: Changing the language for a code file enables the proper features.

Of course, you can change the current language for any existing files, 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 bottom right, represented in the previous figures as UTF-8, the current encoding. You will be presented with a list of supported encodings with a search box where you can start typing encoding names to filter the list (see Figure 34).

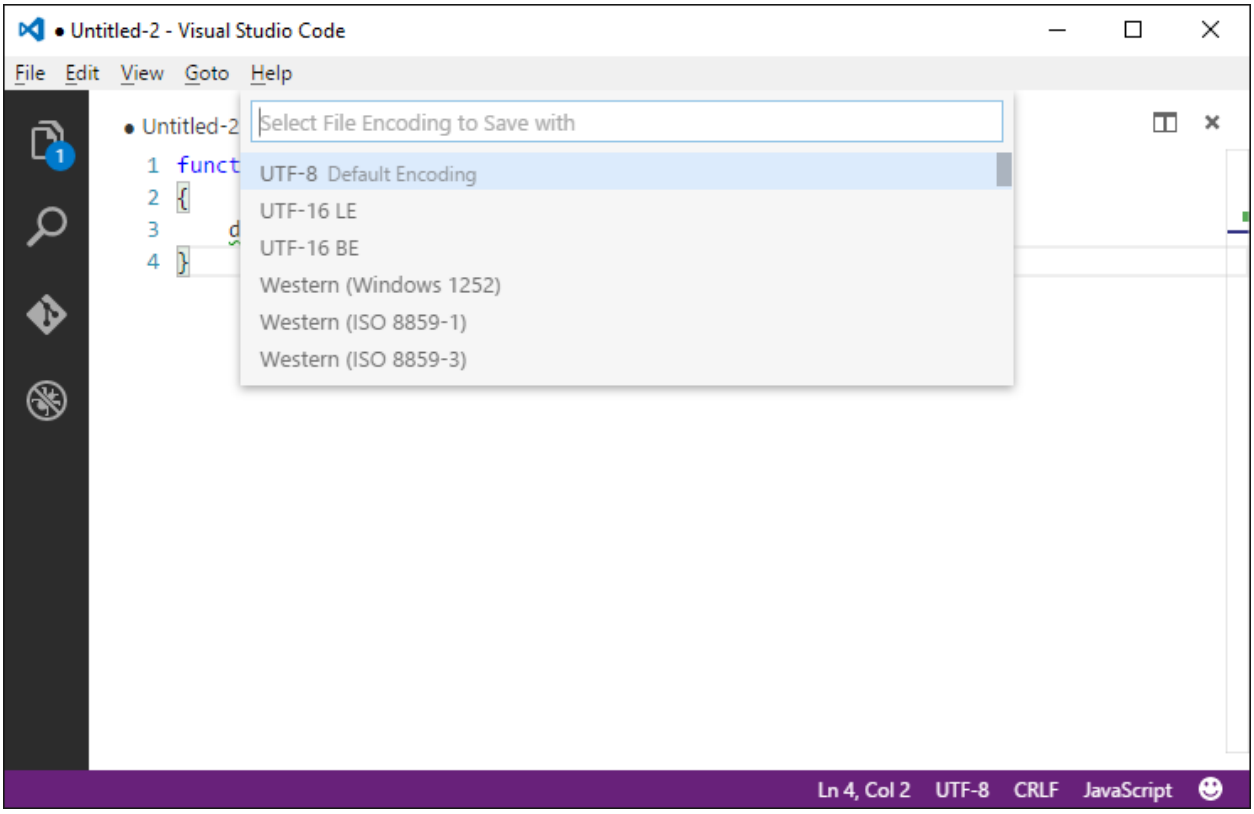


Figure 34: Selecting a Different Encoding

Similarly, you can change the line terminator by clicking the **Select End of Line Sequence** item, represented in previous figures by CLRF in the lower right corner of the editor. Visual Studio Code supports CRLF (carriage return and line feed) and LF (line feed); the default selection is CRLF. You can also move instantly to a line of code by clicking the **Go to Line** item, represented by the line and column number in the status bar. This will open a search box where you can type the line number you want to go to and the line of code will be immediately highlighted as you type. You can also navigate to a line by pressing **Ctrl**+**G** (see Figure 35).

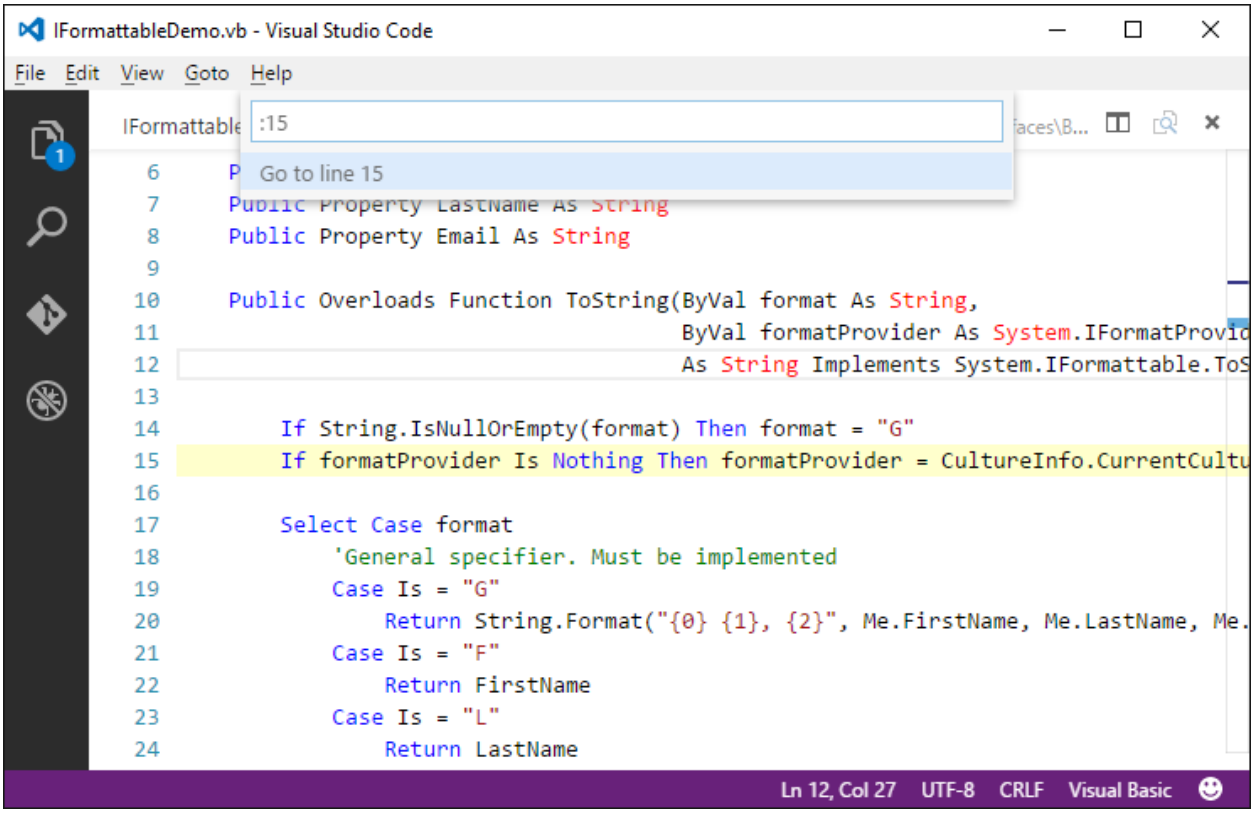


Figure 35: Go to Line in Action

Working with folders and projects

Compared to 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, Code first searches for:

- **project.json** files. If found, Code treats the folder as a DNX project written in C#. DNX stands for .NET Execution Environment and is a runtime with an SDK built on top of .NET Core for building portable, cross-platform ASP.NET Core solutions.
- **MSBuild solution files (.sln)**. If found, Visual Studio Code knows this is a solution built for Microsoft Visual Studio, so it scans the referenced projects (*.csproj and *.vbproj files) and organizes files and subfolders in the proper way.
- **tsconfig.json** files. If found, Visual Studio Code knows this represents the root of a TypeScript project, so it scans for the referenced files and provides the proper file and folder representation.
- **jsconfig.json** files. If found, Visual Studio Code knows this represents the root of a JavaScript project. So, similarly to TypeScript, it scans for the referenced files and provides the proper file and folder representation.
- **package.json** files. These are typically included with JavaScript projects and DNX projects, so Visual Studio Code automatically resolves the project type based on the folder's content.

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

If none of the supported projects are found, Visual Studio Code loads all the code files in the folder as a loose assortment, organizing them into a virtual folder for easy navigation. Now let's discover how Visual Studio Code allows you to work with folders and supported projects.

Opening a folder

Generally speaking, you open a folder with Code via **File** > **Open Folder**. On Windows, if you selected the option during installation, you can right-click a folder name in File Explorer and then select **Open with Code**. Whatever folder you open, Code organizes files and subfolders into a structured view represented in the Explorer side bar. Figure 36 shows an example.

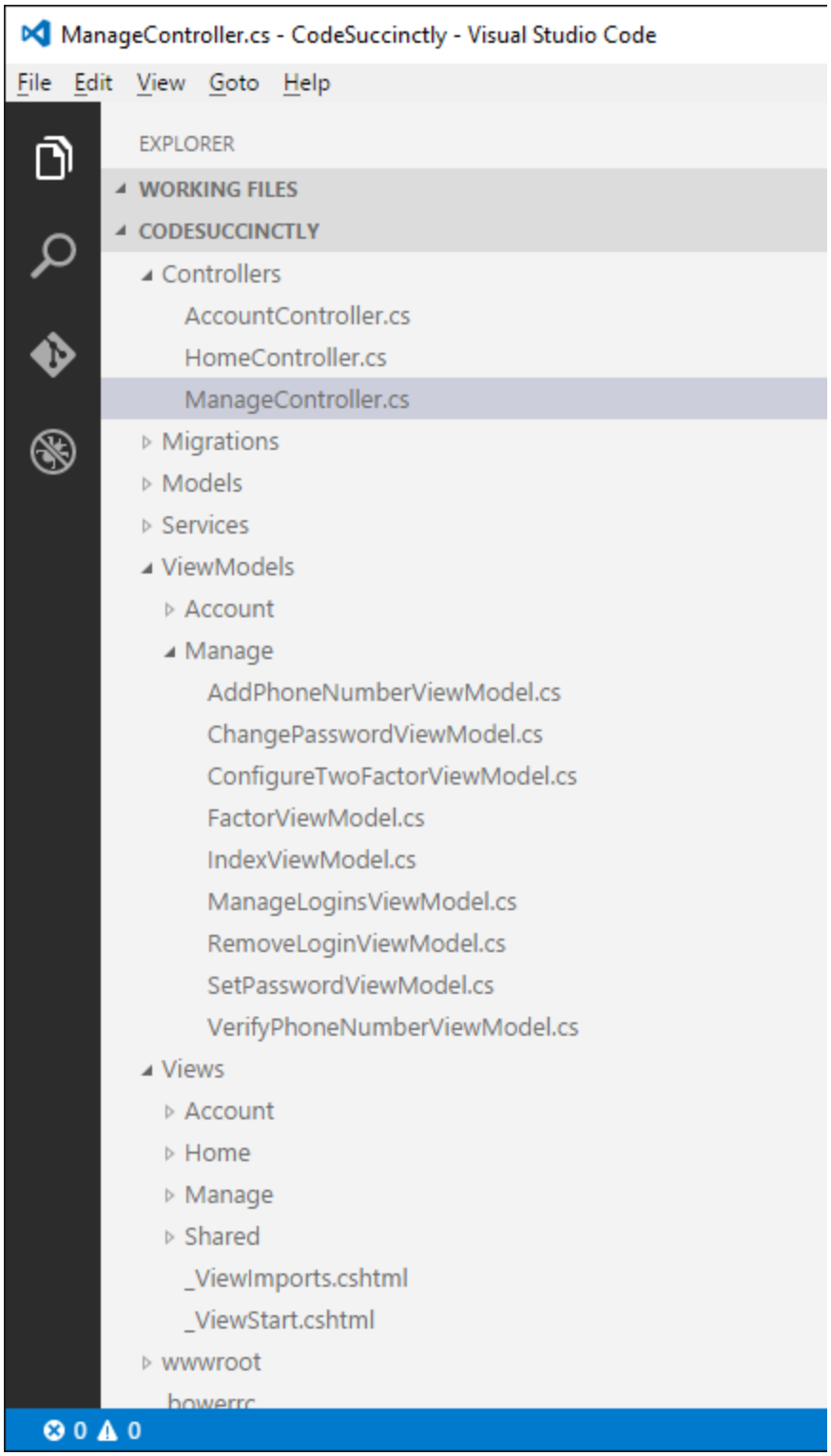


Figure 36: Organizing Folders and Files within Explorer

The root container is the folder name. Files and subfolders are nested in it, and you can expand each subfolder to browse every file it contains. Just click a file to open an editor window of it. The Explorer tool is much more than just file browsing, but let's save this for Chapter 2, where I will discuss Code's user interface deeply. For now, focus on the visual representation it provides for any folders.

Working with DNX projects (ASP.NET Core)

Visual Studio Code has deep support for DNX projects targeting ASP.NET Core with C#. When you open a folder that contains a DNX project, Visual Studio Code organizes all the code files into the Explorer bar and enables all the available editing features for C#. Figure 37 shows an example.

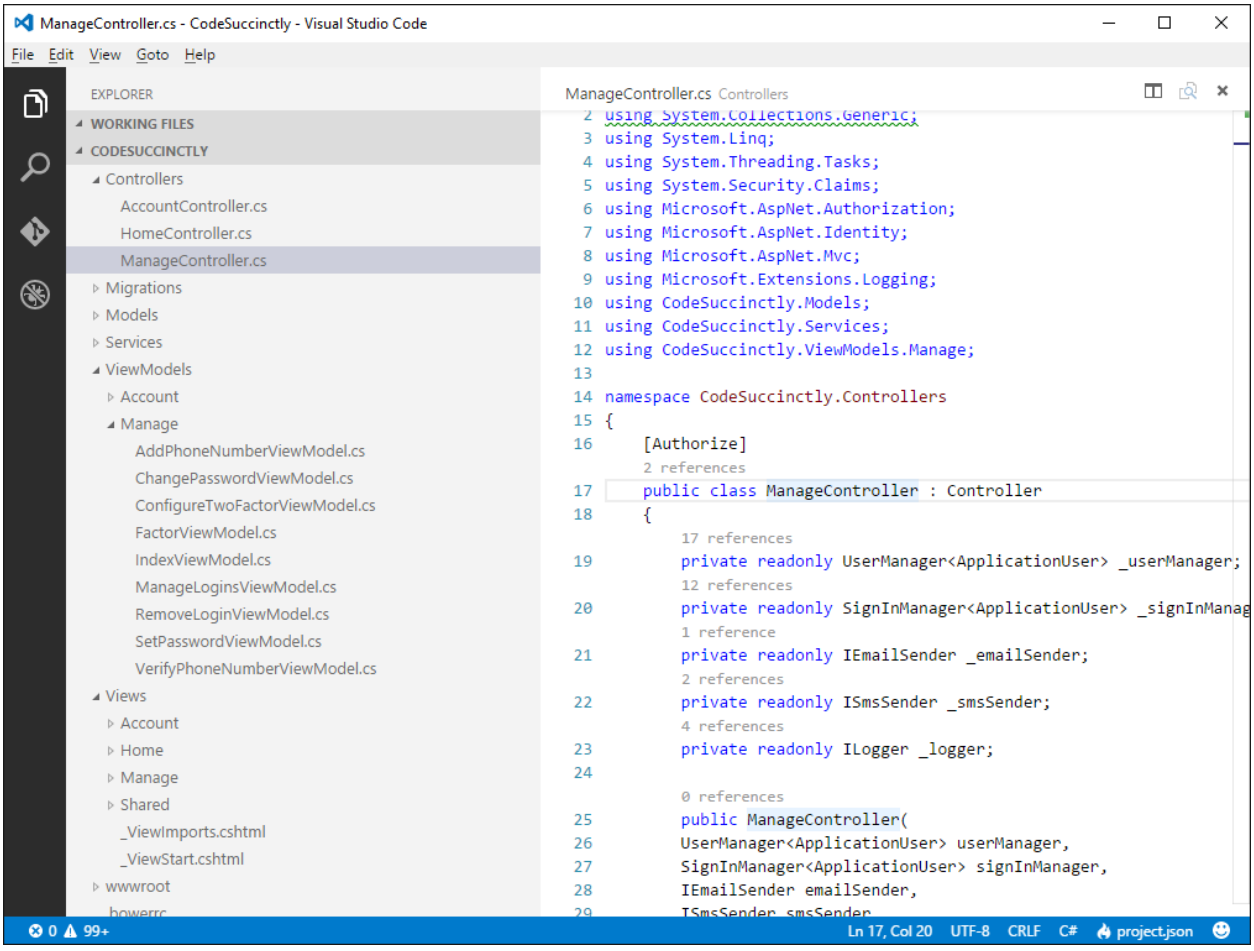


Figure 37: Opening a DNX Project with Visual Studio Code

Notice how the root level in Explorer is the project name. You can browse folders, code files, and edit anything that Visual Studio Code can properly recognize. You can also take a look at the project.json file, which contains project information, the list of dependencies, scripts, and other information. Figure 38 shows an excerpt.

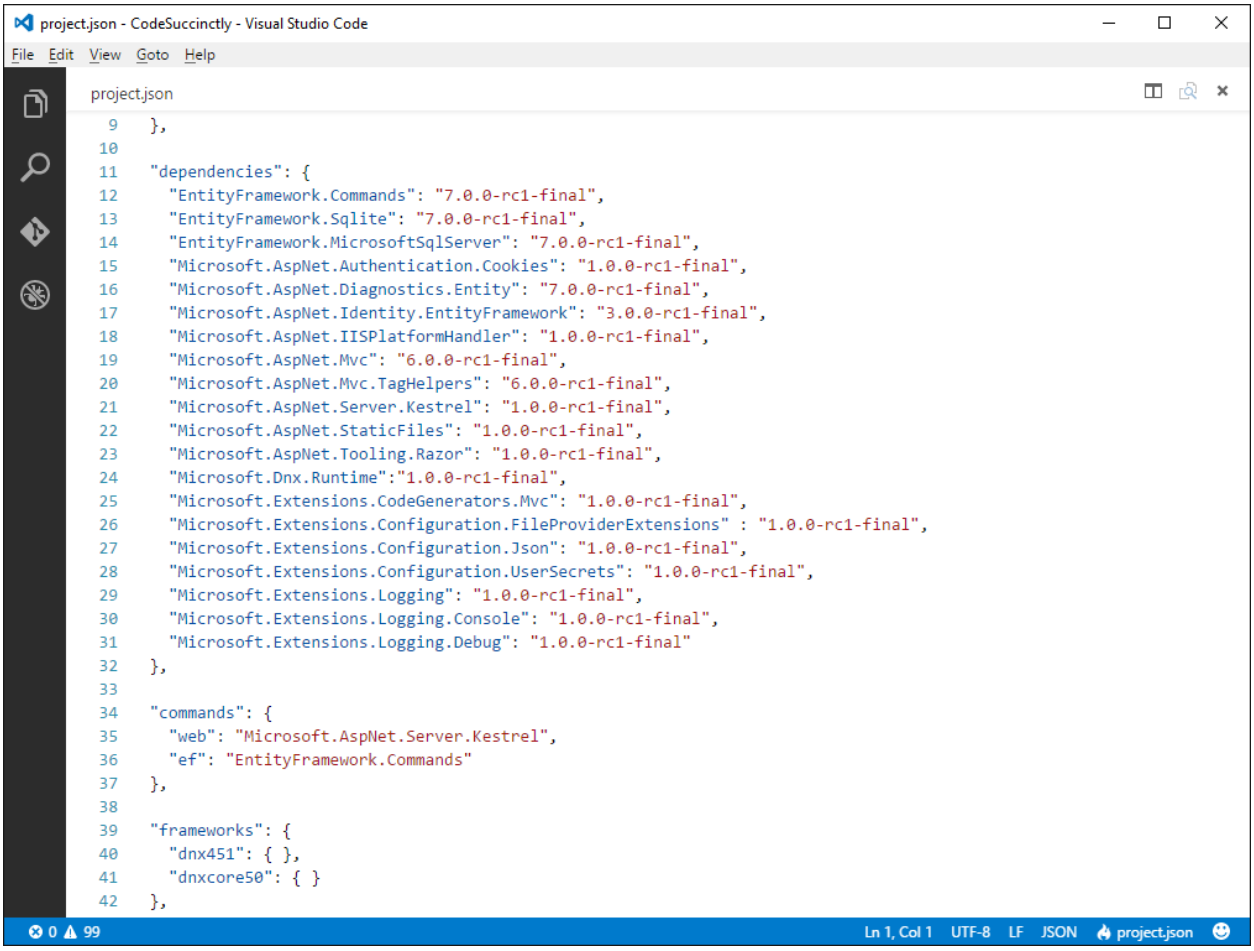


Figure 38: Examining a project.json file

Note: With DNX projects, you will be required to restore NuGet packages using the `dnu restore` command from the Command Palette. This is described in more detail in Chapter 4, "In Practice."

Working with JavaScript projects

Similarly to DNX, 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 39.

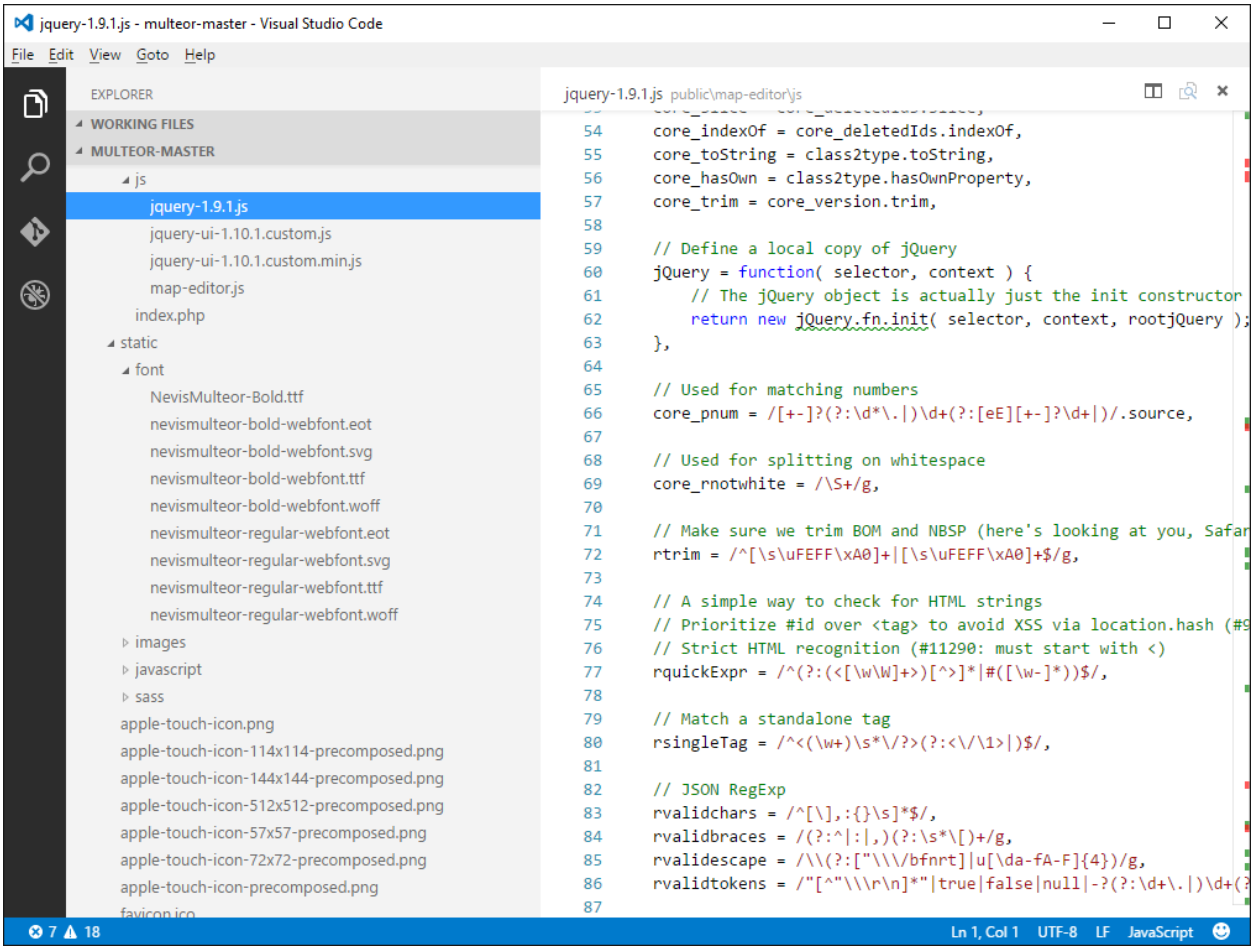


Figure 39: Opening a JavaScript Project

TypeScript projects behavior in Visual Studio Code is the same as for JavaScript.

Working with MSBuild solutions

You might also need to open Microsoft Visual Studio solutions (.sln) with Visual Studio Code. If this is the case, do not open the .sln file directly. This would result in an editor window showing the XML content of the file. Instead, open the folder that contains the solution file. As you can expect, Visual Studio Code organizes the project files and folders within Explorer, and it enables all the editing features available for the selected files. Figure 40 shows an example.

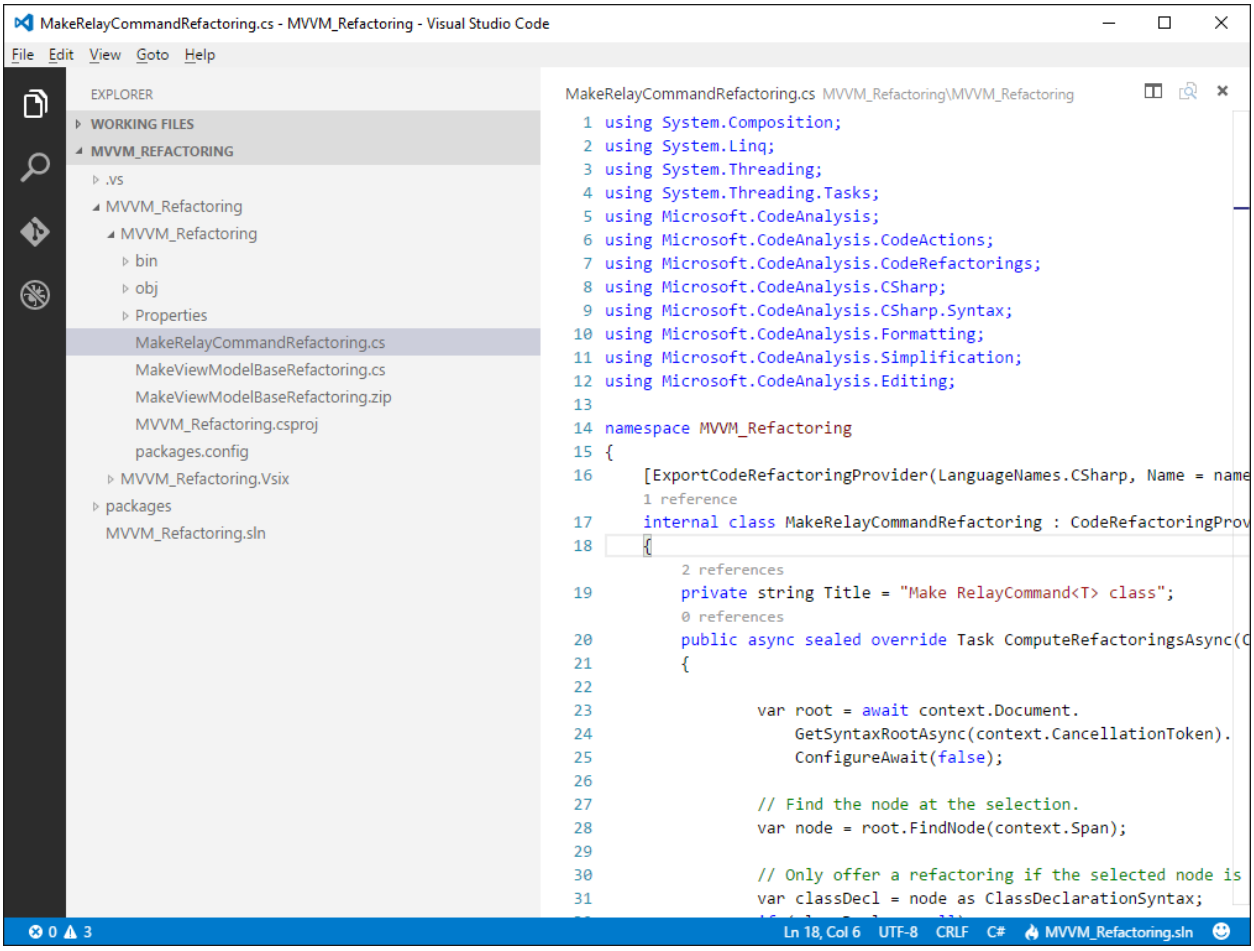


Figure 40: Opening a Visual Studio Solution

It is worth noting that features such as Find all References and code issue detection are enabled if Visual Studio Code detects NuGet packages inside the solution.

Tip: In Chapter 3, "Git Version Control and Task Automation," you will also learn how to launch the MSBuild engine from VS Code and compile a DNX project written with C#.

Working with loose folders

Visual Studio Code certainly allows opening folders that contain unrelated, loose files. Visual Studio Code creates a logical root based on the folder name, showing files and subfolders. Figure 41 shows an example based on a sample folder called MyFiles.

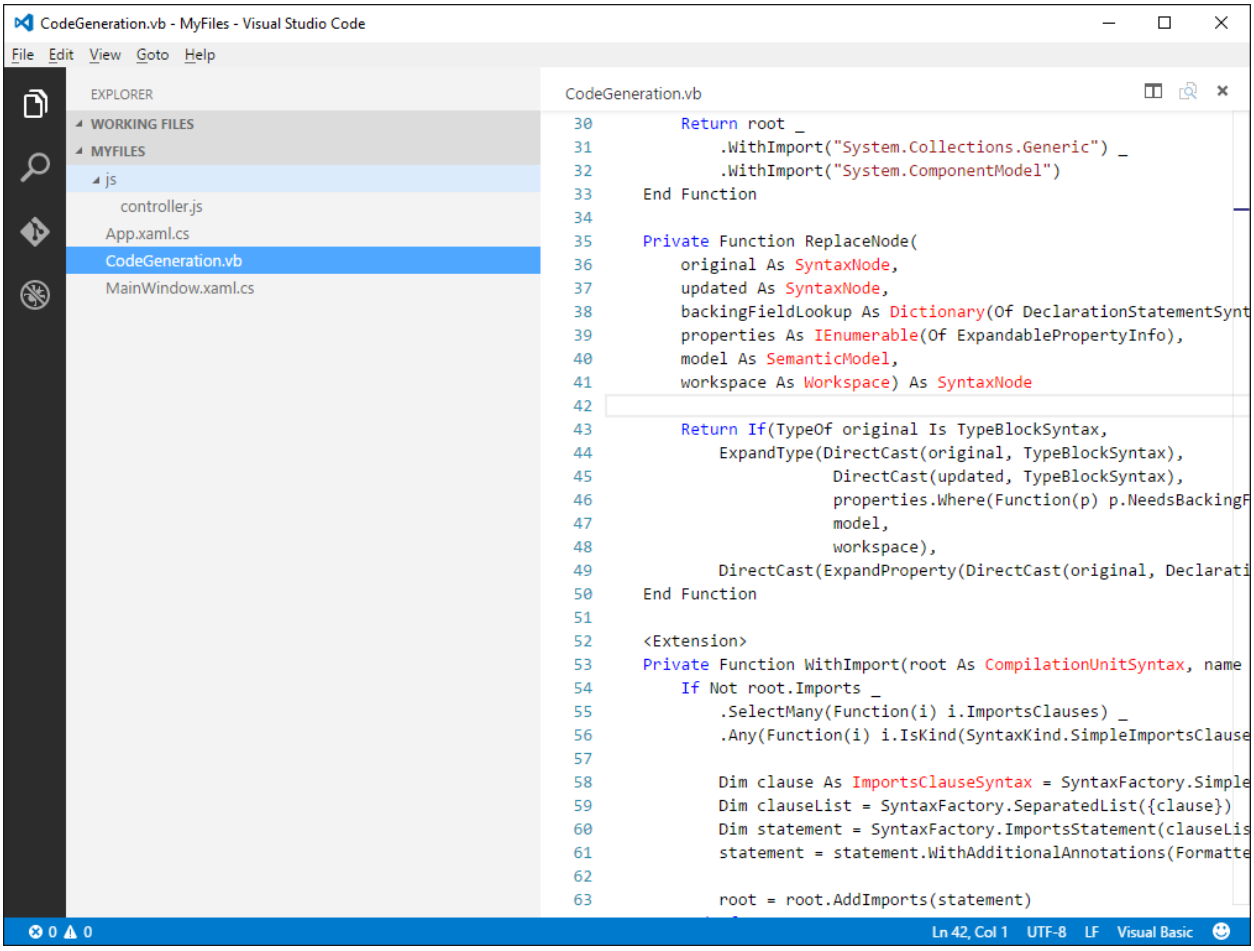


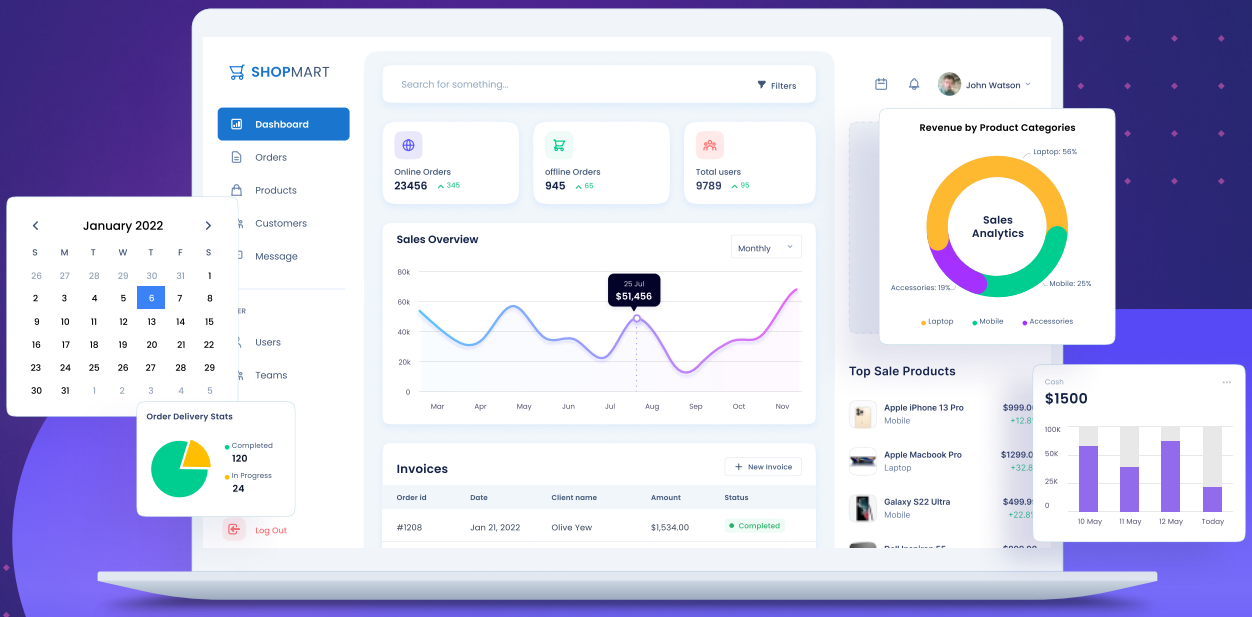
Figure 41: Opening a Folder Containing Loose Files

Chapter summary

Visual Studio Code is an open source, cross-platform, multi-language development environment that puts writing code at its center. Code supports a variety of languages and offers an enhanced editing experience with common features such as syntax colorization, and more advanced capabilities such as IntelliSense, code issue detection, and code browsing. In order to support different project systems, Visual Studio Code is file and folder based, which means that it supports working with loose code files and with organized, structured folders containing project and code files. After this long overview of the available editing features, the next logical step is discussing the workspace and the user interface.

THE WORLD'S BEST UI COMPONENT SUITE FOR BUILDING

POWERFUL APPS



GET YOUR FREE .NET AND JAVASCRIPT UI COMPONENTS

syncfusion.com/communitylicense

4.6 out of 5 stars

- 1,700+ components for mobile, web, and desktop platforms
- Support within 24 hours on all business days
- Uncompromising quality
- Hassle-free licensing
- 28000+ customers
- 20+ years in business

Trusted by the world's leading companies

Syncfusion

Chapter 2 The Workspace and the User Interface

The user interface and layout in Visual Studio Code are optimized to maximize the space for code editing. In addition, it provides easy shortcuts to quickly access all the additional tools you need in a given context, offering an awesome experience. More specifically, the user interface is divided into four areas: The code editor, the status bar, the view bar, and the side bar. This chapter explains how the user interface is composed and how you can maximize your productivity with it.

Tip: Most of the tools described in this chapter have buttons with just icons and no text. However, buttons have names that you can see by hovering over them with the pointer.

The code editor

The code editor is definitely the area where you spend most of your time. You can edit one file at a time as you saw in Chapter 1, but you can also open up to three code files side-by-side. Figure 42 shows an example.

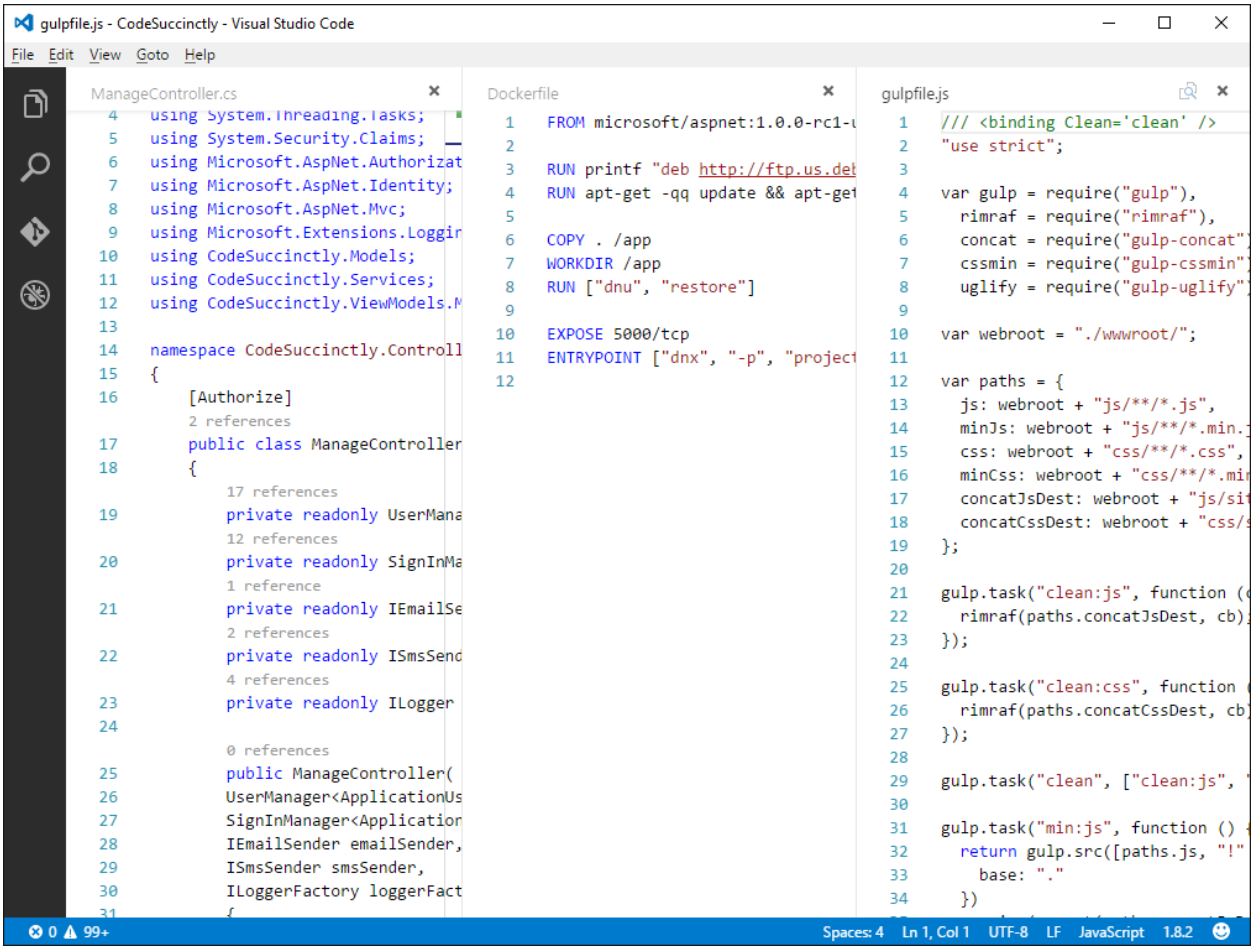


Figure 42: The code editor allows opening up to three code files.

To do this, you have a couple options:

- Right-click a file name in the Explorer bar and then select **Open to Side**.
- Hold Ctrl and click on a file name in Explorer.
- Press Ctrl+\ to split the editor in two.

Notice that if you already have three files open and you want to open another file, the editor that is active will display that file. You can quickly switch between editors by pressing Ctrl+1, Ctrl+2, and Ctrl+3.

Resizing and reordering editors

Editors can be resized and reordered. Resizing an editor can be accomplished by clicking the left mouse button over an editor's border when the pointer appears as a pair of left and right arrows. Reordering editors is done by clicking an editor's header, where the file name is displayed, and moving it to a different position.

Zooming editors

You can easily zoom in and out of the active editor by pressing **Ctrl**+**Plus Sign** and **Ctrl**+**Hyphen** respectively. Alternatively, you can select **View** > **Zoom in** and **View** > **Zoom out**. Notice that this is an accessibility feature, so when you zoom the code editor, the view bar and side bar will also be zoomed.

The status bar

The status bar contains information about the current file or folder and allows you to perform some quick actions. Figure 43 shows an example of how the status bar appears.

Figure 43: The status bar

From left to right, you find the following information:

- Git version control information and options, such as the current branch and the Synchronize Changes button.
- Errors and warnings. You already saw this in the Code Issues and Refactoring section in Chapter 1.
- Indentation. This is the Spaces: 4 item in Figure 43. You can click this to change the indentation size and to convert indentation to tabs or spaces.
- The cursor's position expressed in line and column numbers.
- The current file encoding.
- The current line terminator.
- The language for the open file.
- The project name if you open a folder that contains a supported project system. It is worth noting that, in case the folder contains multiple project files, clicking this item will allow switching between projects.
- The feedback button, which allows sharing your feedback about Visual Studio Code on Twitter.

The status bar is blue if you open a folder or violet if you open a single file.

The view bar

The view bar is placed at the left side of the user interface and provides quick access to four tools. Figure 44 shows the view bar.

Figure 44: The View Bar

The view bar can be thought of as a collapsed container for the side bar, and it provides shortcuts to the following tools: Explorer, Search, Git, and Debug, which are described in the next sections.

The side bar

The side bar is the most important companion for the code editor. It is composed of the four tools described in the next paragraphs. Each is enabled by clicking its corresponding icon.

The Explorer bar

You already encountered the Explorer bar in Chapter 1. It provides a structured, organized view of the folder or files you are working with. The Working Files subview contains the list of active files, including open files that are not part of a project, folder, or files that have been modified. These are instead shown in a subview whose name is the folder or project name. Figure 45 provides an enhanced example of the Explorer.

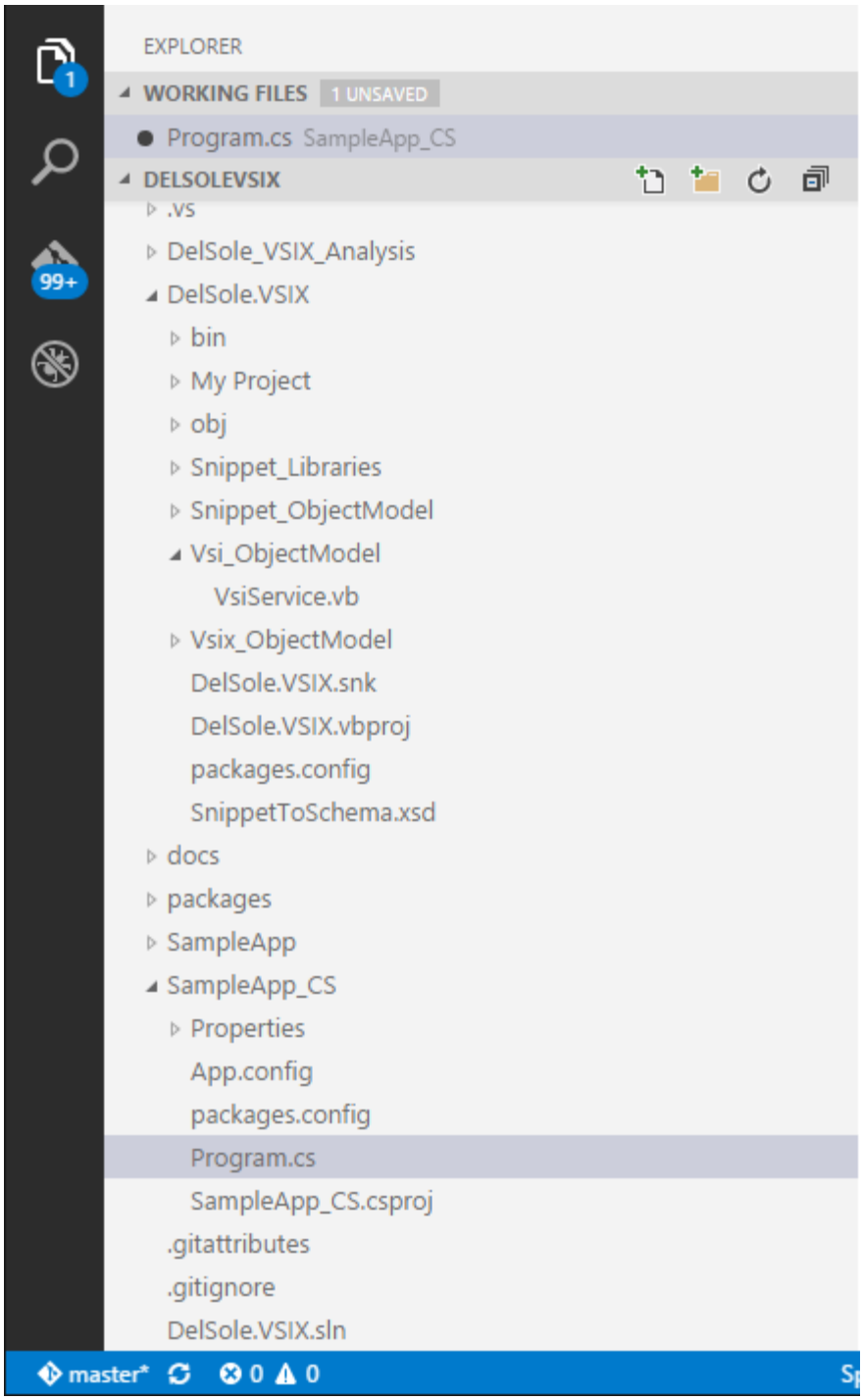


Figure 45: The Explorer Bar

The subview that shows a folder structure provides four buttons from left to right: New File, New Folder, Refresh, and Collapse. The function of each is self-explanatory. The Working Files subview has two buttons: Save All and Close All Files. Right-clicking a folder or file name in Explorer provides a context menu that offers common commands, such as the Open to the Side option mentioned at the beginning of this chapter. A very interesting command is Reveal to Explorer (or Reveal to Finder on Mac and Open Containing Folder on Linux), which opens the containing folder for the selected item. Notice that the Explorer icon in the view bar reports the number of modified files.

Searching across files

The Search tool allows searching 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 46 shows the Search tool in action with advanced options expanded (the Files to include and Files to exclude text boxes).

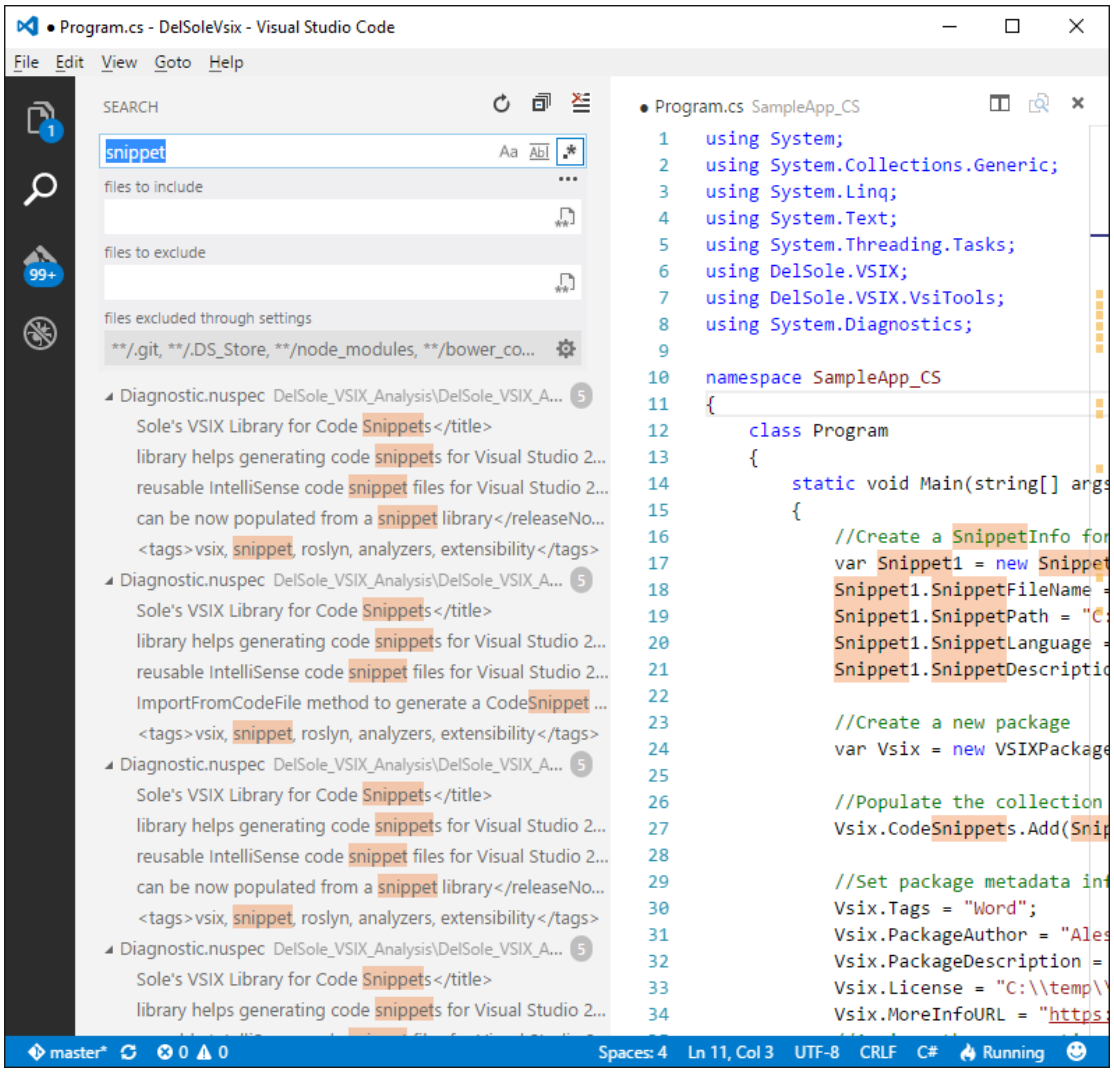


Figure 46: Searching Across Files

Search results are offered in a hierarchical view that groups all the files that contain the specified search key, and shows an excerpt of the line of code that contains it. Occurrences are highlighted in both the list of files and in the code editor. When you are done, you can clean up search results by clicking the **Clear Search Results** button.

Git and Debug

The side bar also provides access to two special and important tools in Visual Studio Code: Git integration for version control and the integrated debugger. Because both are described inside dedicated chapters later, I will not cover them here. However, for the sake of completeness, Figure 47 provides a glimpse of Git that also shows file differences, and Figure 48 shows an empty Debug area.

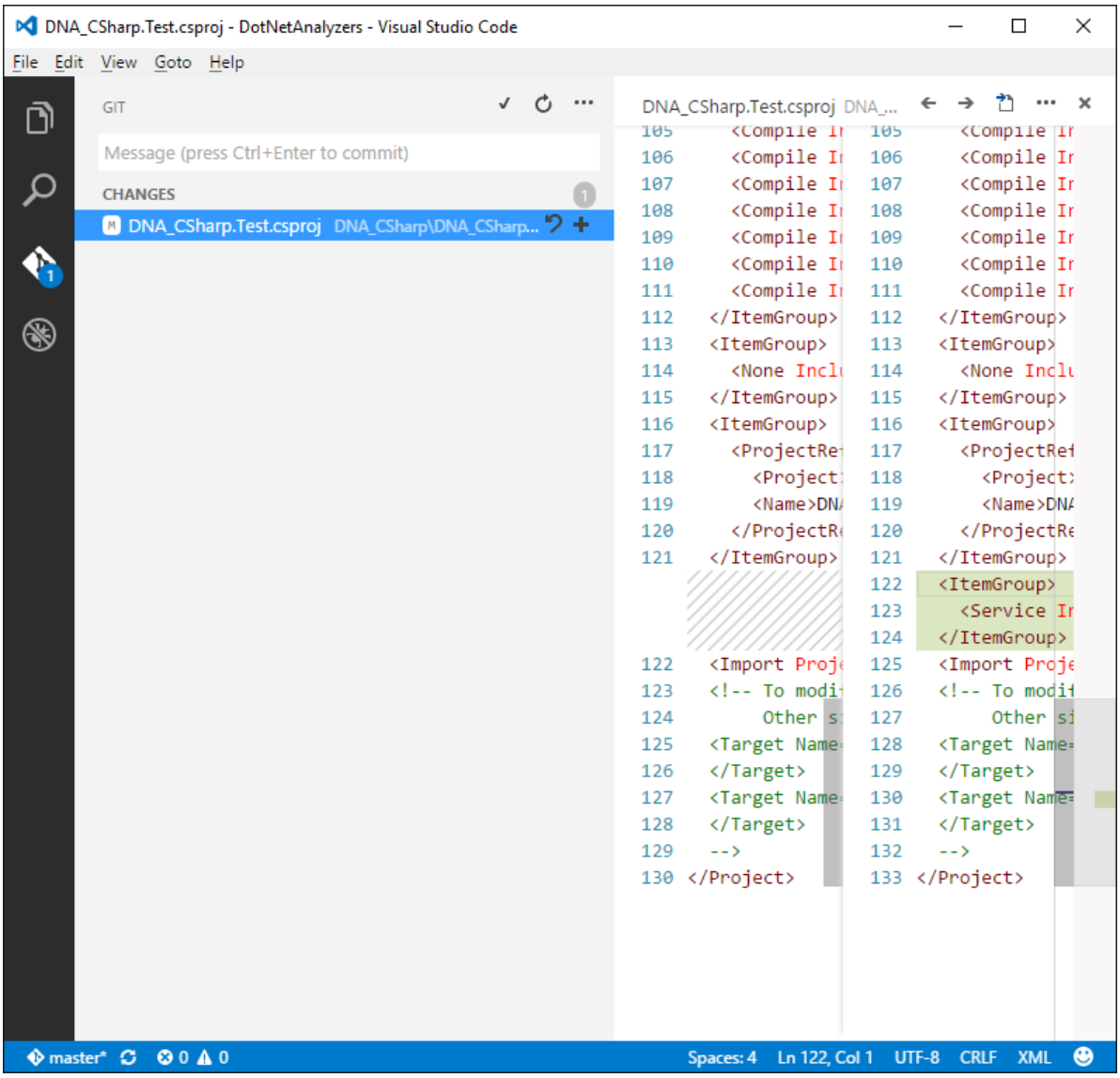


Figure 47: Git and the File Diff Tool

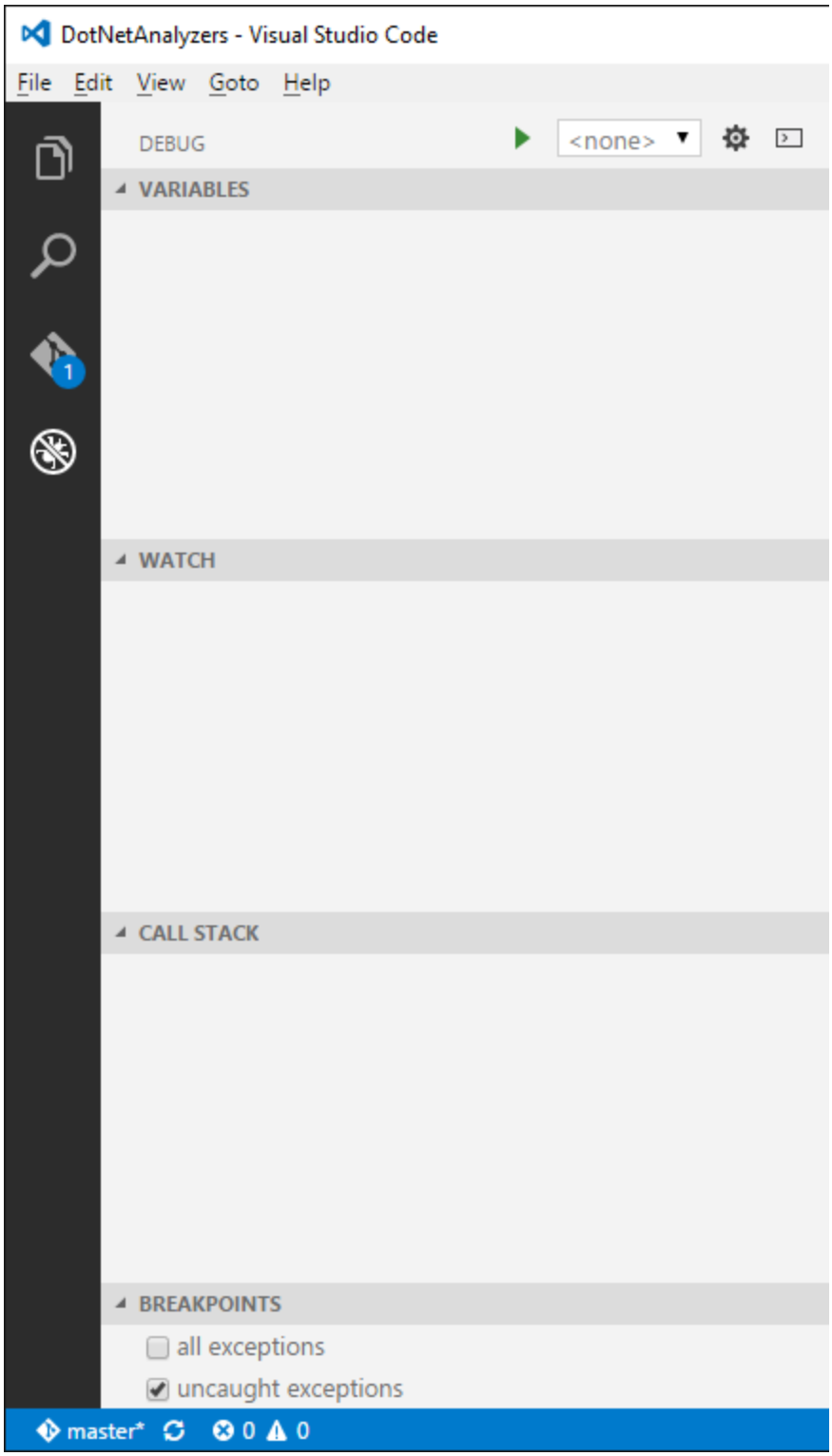


Figure 48: The Debug Area

Quick file navigation

Visual Studio Code provides two ways of navigating between files. If you press Ctrl+Tab, you will be able to browse the list of files that have been opened since Code was launched and you will be able to select one for editing, as shown in Figure 49.

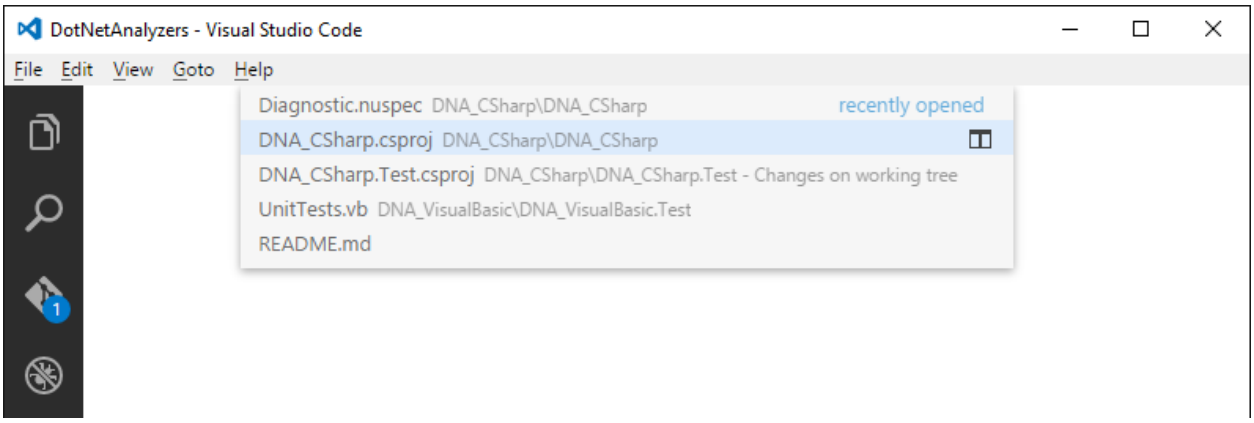


Figure 49: Browsing the List of Recently Opened Files

When you release the Tab key, the selected file is opened for editing. The second way is to press Alt+Left Arrow Key or Alt+Right Arrow Key to switch between active files.

The Command Palette

The Command Palette is an important tool that allows executing special commands. You enable the Command Palette by clicking **View** > **Command Palette** or by pressing either F1 or Ctrl+Shift+P. Figure 50 shows the Command Palette.

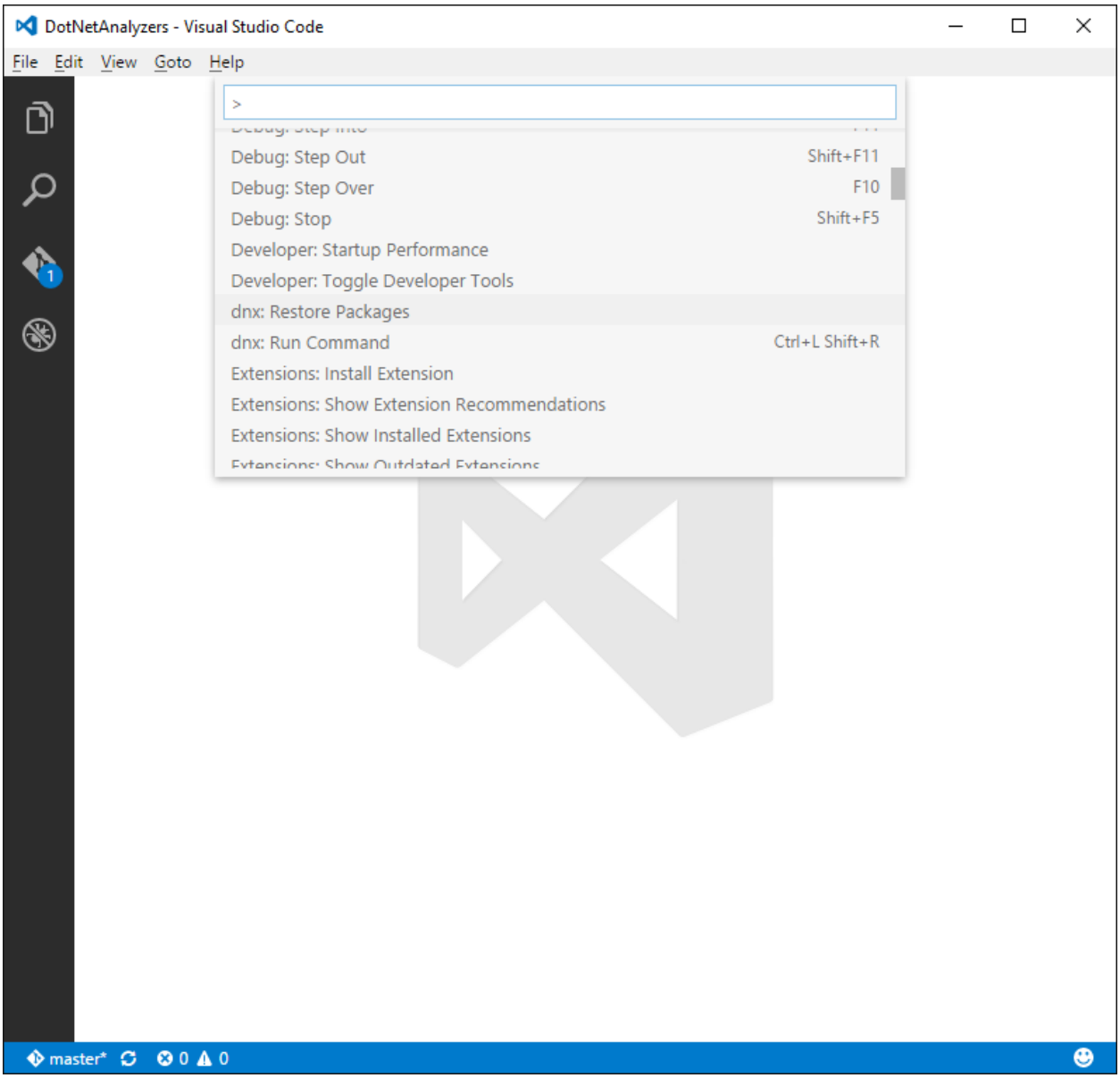


Figure 50: The Command Palette

The Command Palette is basically an easy way to access all of Visual Studio Code's functionalities via the keyboard. This is not just related to menu commands or user interface instrumentation, but also other actions that are not accessible elsewhere. For instance, the Command Palette allows installing extensions as well as restoring NuGet packages over the current project or folder. You can move up and down the list to see all the available commands, and you can type characters to filter the list. You will notice many commands map actions available within menus. Other commands, such as extension, debug, and Git commands, will be discussed in the next chapters so it is important that you get started with the Command Palette at this point.

Chapter summary

The user interface in Visual Studio Code is optimized to maximize the available space for editing. The workspace is divided into four areas. The code editor is where you write code and where you can open up to three files concurrently. The status bar reports summary information about the file or folder. The view bar is a collapsed container for the side bar that offers four important tools: Explorer, Search, Git, and Debug. Explorer provides an organized view of files and folders, while Search allows searching across files with advanced options. Git and Debug will be discussed in the next chapters. Visual Studio Code offers another important tool, the Command Palette, where you can browse the list of available commands, including those you execute against projects, folders, and files directly.

Chapter 3 Git Version Control and Task Automation

More often than not, writing software involves collaboration. For example, you might be part of a development team at work, or you might be involved in open source, community projects. Microsoft strongly supports both collaboration and open source, so Visual Studio Code perfectly integrates with Git. Also, in many cases you will need to launch external tools, which Visual Studio Code supports via Tasks. This chapter describes both topics.

Note: Because of the open source, cross-platform nature of Visual Studio Code, in this chapter I'll be providing examples based on independent application development frameworks. For instance, there's no Microsoft Visual Studio here, though you can certainly use anything you learn here with a Visual Studio solution opened with VS Code.

Version control with Git

Git is a very popular, open-source distributed version control engine that makes collaboration easier for small and large projects. Visual Studio Code works with any Git repository, such as GitHub or Visual Studio Team Services, and provides an integrated way to manage your code commits. Notice that this chapter is not a guide to Git, but rather an explanation of 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 if you haven't, download and install Git for your operating system. In order to demonstrate how Git version control works with Visual Studio Code, I will create a small Apache Cordova project. As you might know, Apache Cordova is a framework you can use to create cross-platform mobile applications based on JavaScript and HTML5. Of course, all of the steps and techniques described in this chapter apply to any project in any language Visual Studio Code supports. Make sure you have Node.js installed on your machine before going on.

Installing the Cordova command-line interface

In order to use the Apache Cordova tools, you first need to install the Cordova command-line interface (CLI). Of course, you can skip this step if you have installed the Apache Cordova tools for Visual Studio 2015. To accomplish the installation, and assuming you have Node.js installed, open a command prompt and type the following:

```
npm install -g Cordova
```

Installing the CLI won't take very long.

Creating a sample project

Tip: There is an extension for VS Code called Cordova Tools which provides additional features such as specific debuggers and commands. This is not used in this example, but it is something you might want to download if working with Cordova is of primary importance for you.

Creating a Cordova project from the command line is very easy. This is the simple command you need to create a Cordova project:

```
> cordova create CodeSuccinctly
```

Where **`CodeSuccinctly`** is the name of your project. Notice that the tool will create a folder named CodeSuccinctly where it will place the project files. Remember this when it's time to choose the project location. The new Cordova project at this point contains only an application skeleton with a few files, but that's enough for our purposes. When ready, open the project with Visual Studio Code, which means opening the newly created folder. Figure 51 shows what Code looks like at this point.

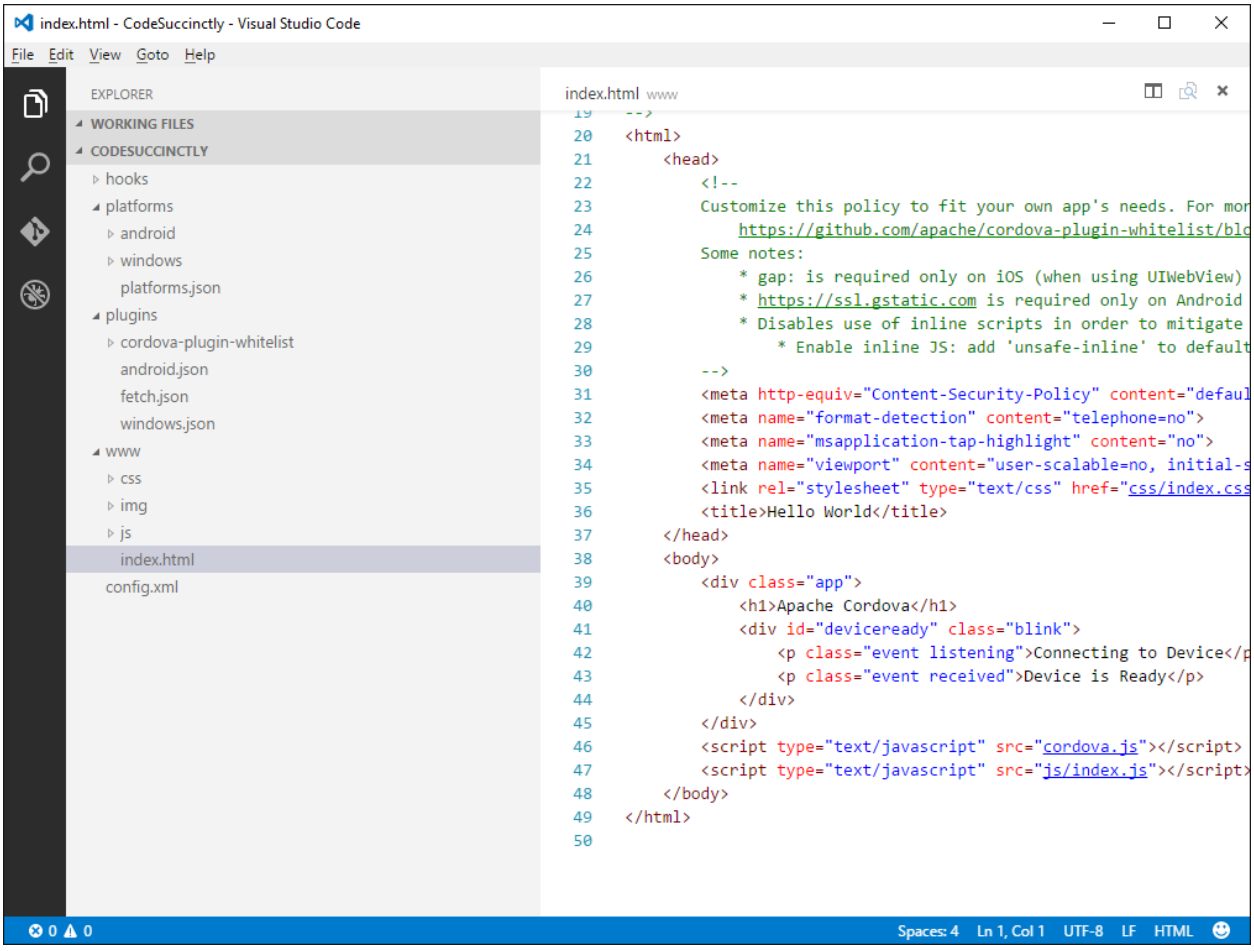


Figure 51: The Cordova Project Opened with Code

Now you are ready to start working with Git.

Initializing a local Git repository

If you are familiar with Git, you know that version control works with both a local and a remote repository. 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 52.

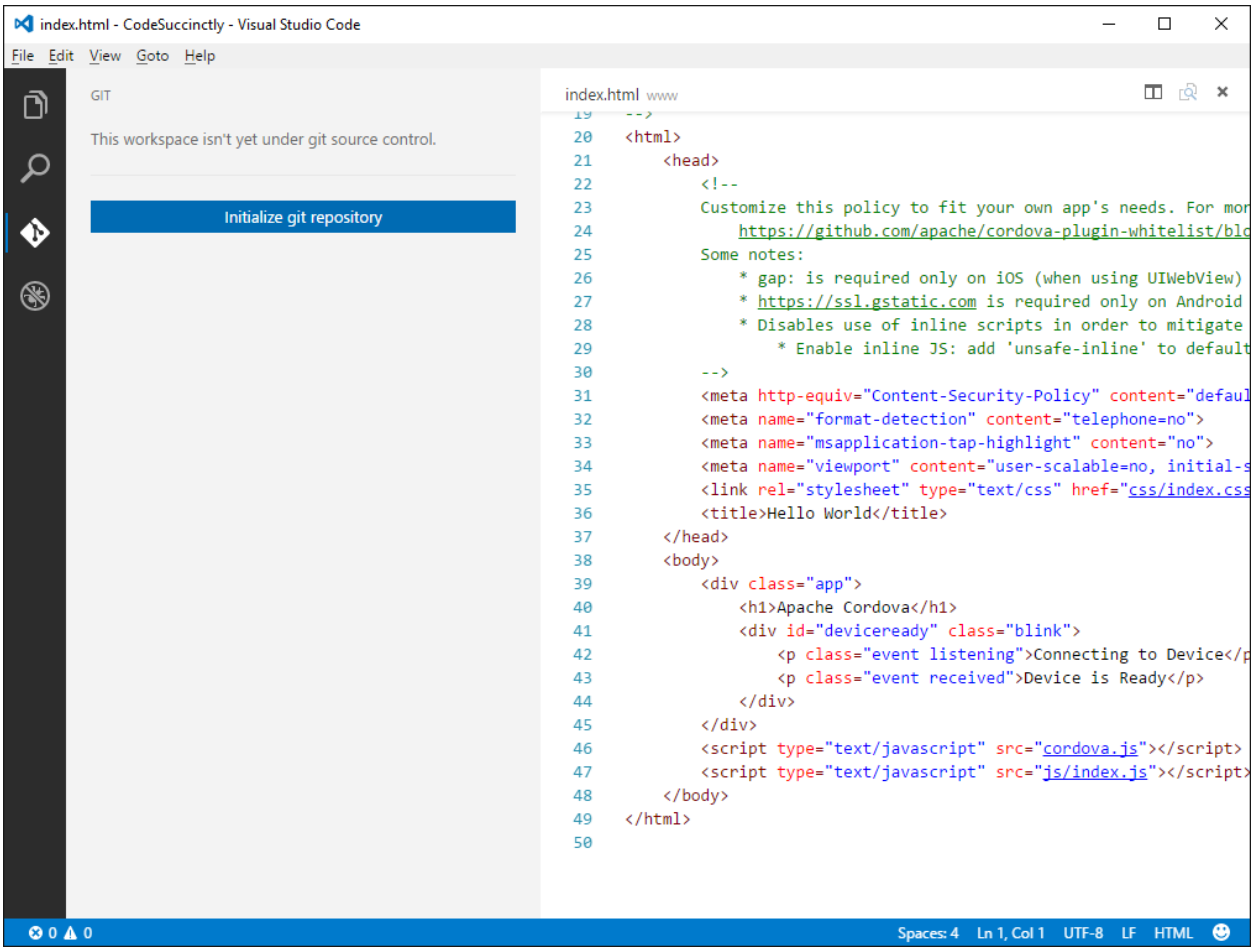


Figure 52: Ready to Initialize a Local Git Repository

Click **Initialize git repository**. Visual Studio Code will initialize the local repository and show the list of files that the version control is tracking but has not yet committed (see Figure 53).

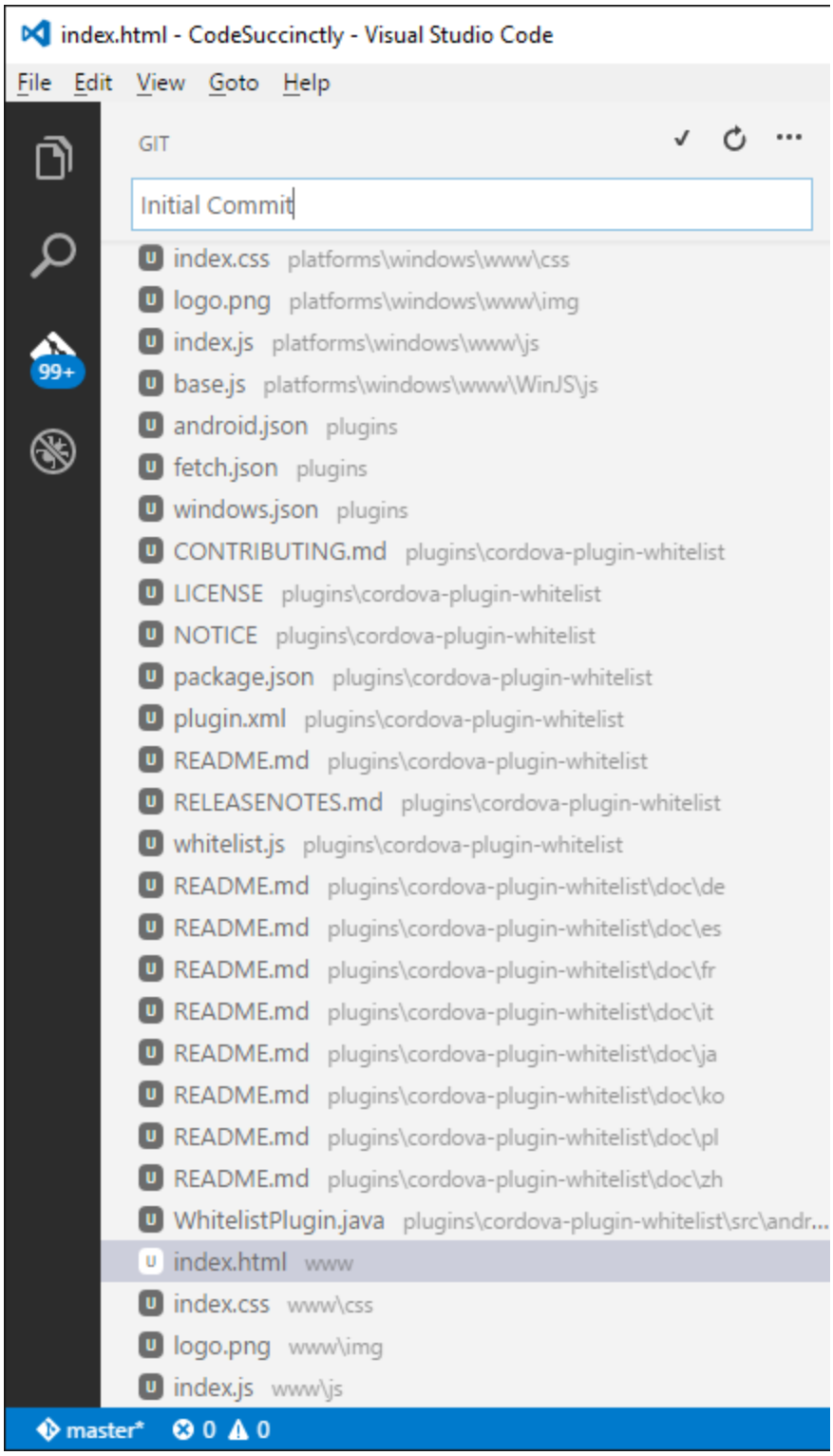


Figure 53: Files Ready for the First Commit

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

Creating a remote repository

Visual Studio Code can work 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 and Microsoft Visual Studio Team Services. For the sake of platform-independence, I will show how to create a GitHub remote repository and associate this to a local project. Of course, I assume you have an existing GitHub account. If not, you can create a new account and a new repository for free. Figure 54 shows an example.

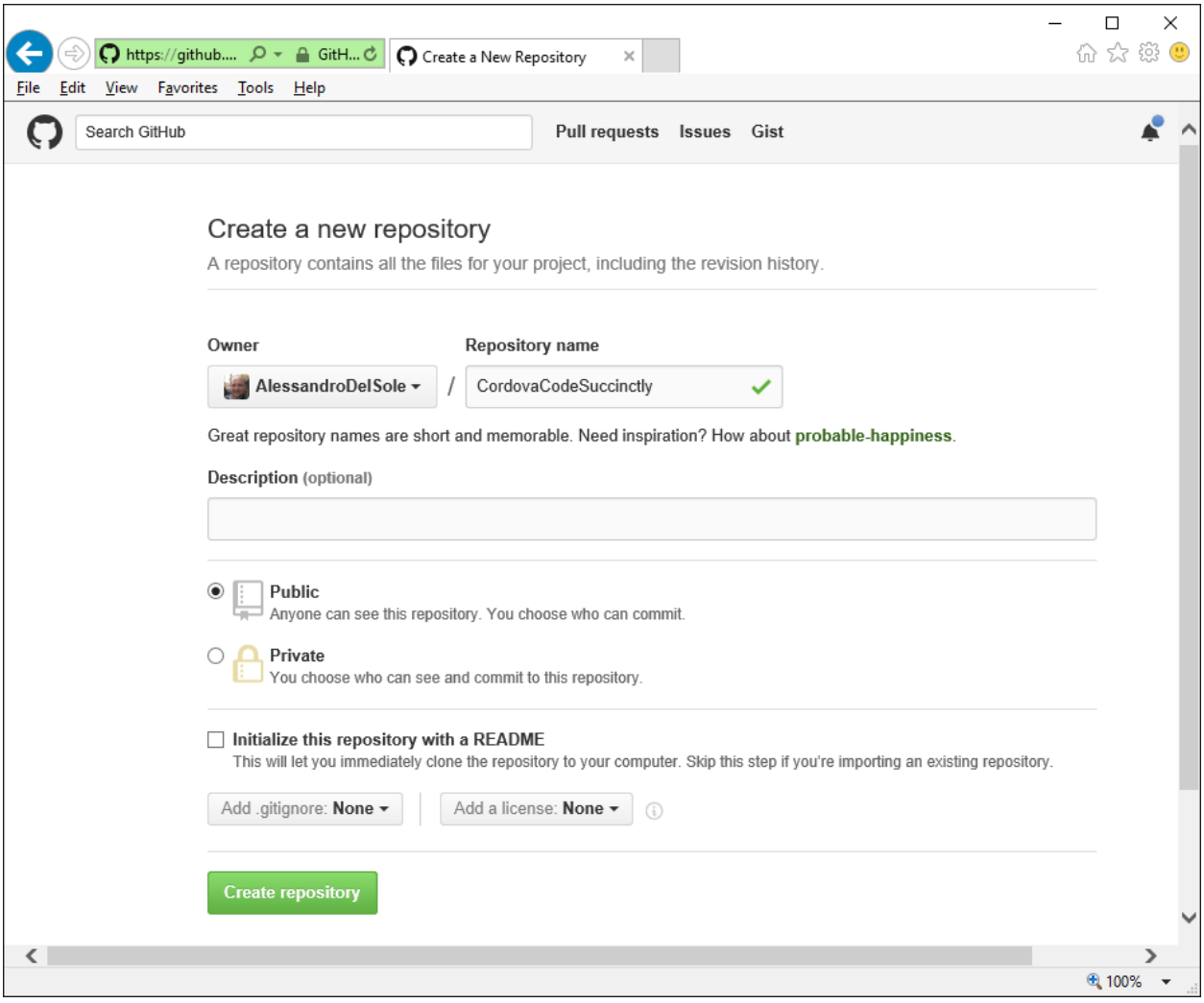


Figure 54: Creating a New Remote Repository

Once the repository is created, GitHub provides fundamental information you need to associate the remote repository with the local one. Figure 55 shows the remote address for the Git version control engine and the commands you have to type to perform the association.

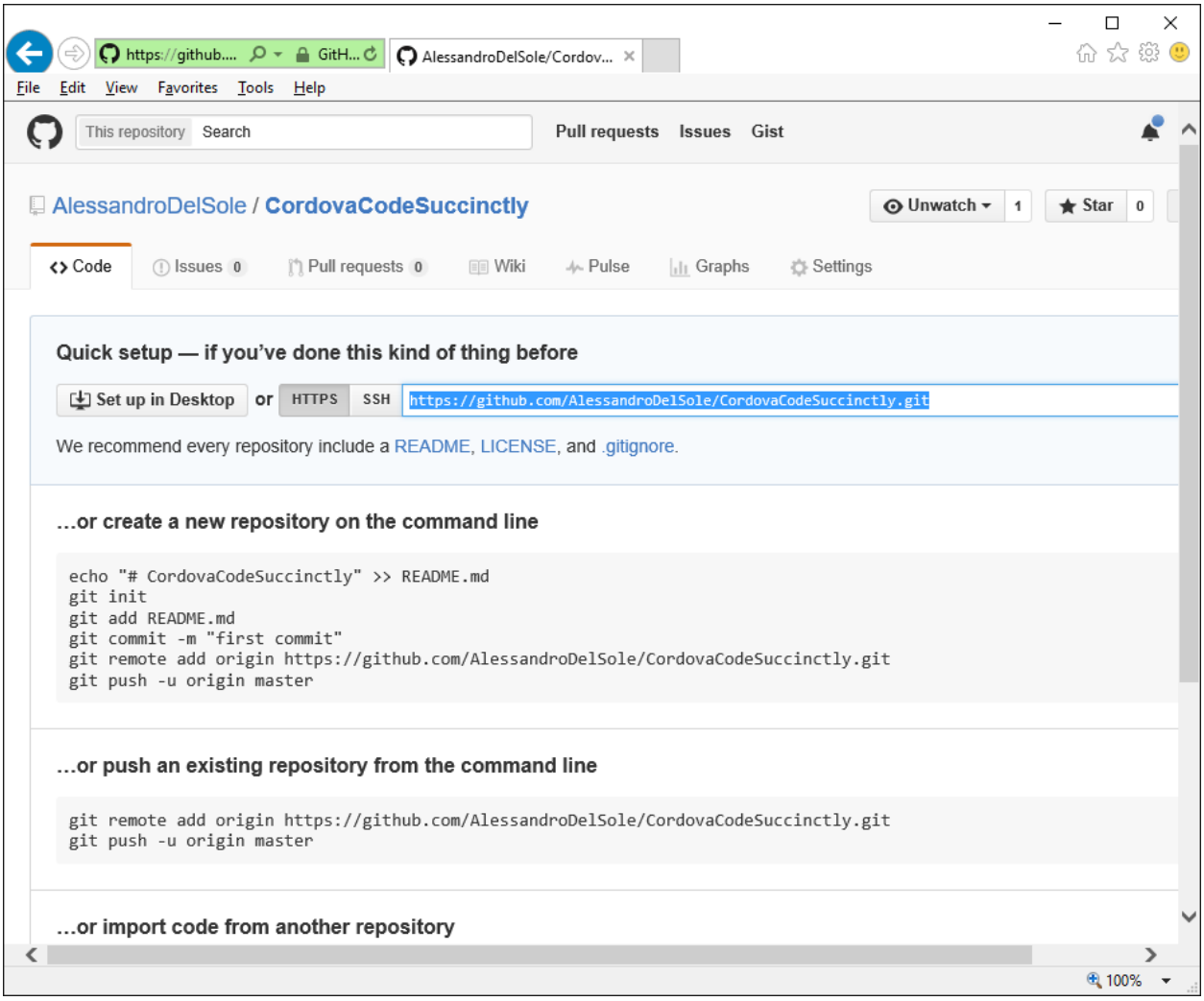


Figure 55: Getting the Necessary Information to Associate the Remote Repository to the Local One

The next step is to open the command prompt and navigate to the folder that contains your Cordova project. Next, write the following two lines:

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

> git push -u origin master
```

Where **`YourAccount`** and **`YourRepository`** represent your GitHub user name and the repository name respectively. The first line tells Git that the remote origin for the current folder (which contains the local repository) is the specified URL. The second line pushes the content of the local repository into the branch called master. Depending on your Internet connection, it will take up to a couple minutes to complete the upload operation. Now you really have everything you need and you can start discovering the Git integration that Visual Studio Code offers.

Managing file changes

Git monitors the code in your local repositories and detects any changes. The Git icon shows the number of files with pending changes. In Figure 56 you can see an example based on two edited files.

Tip: The number over the icon is actually updated only when you save your files locally.

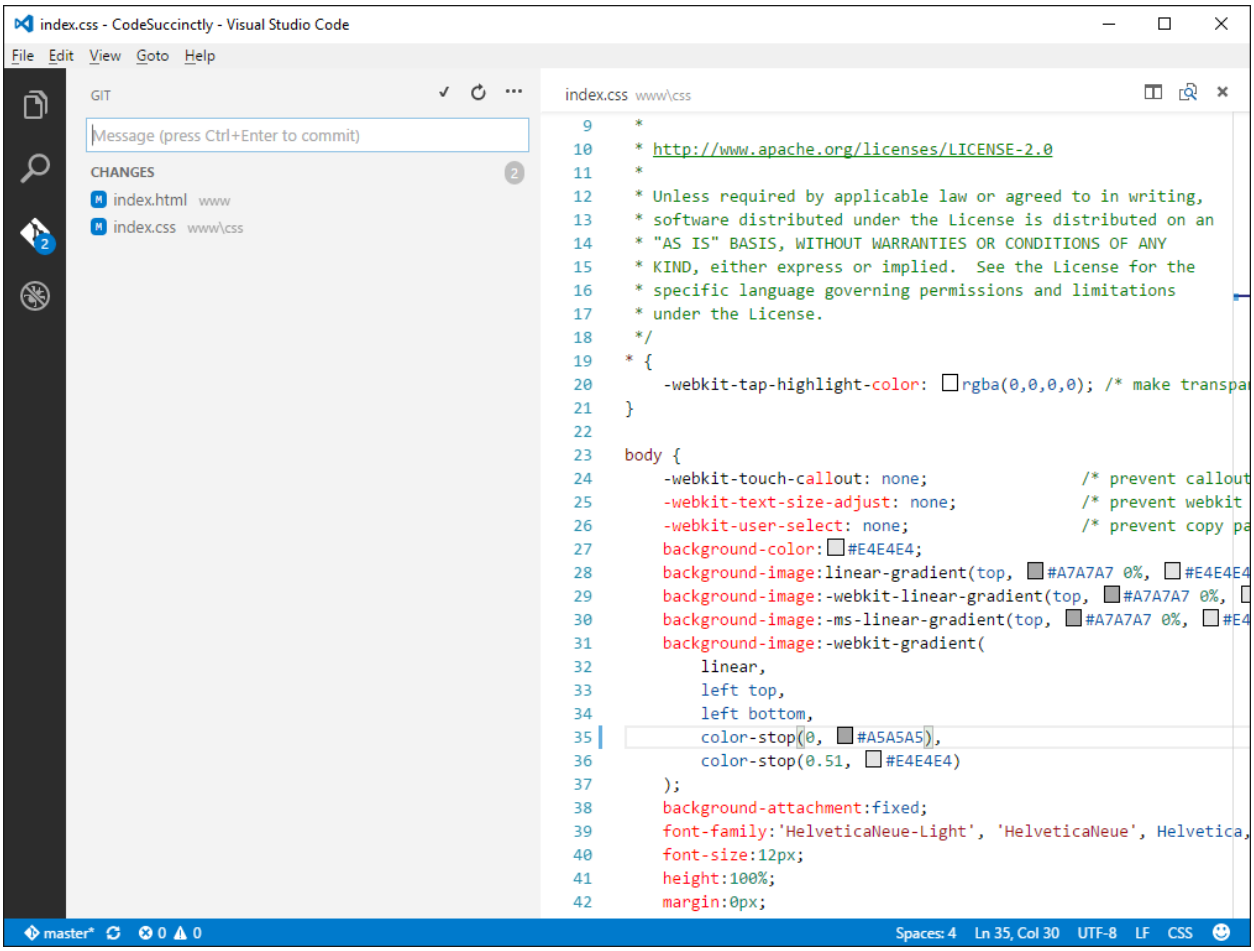


Figure 56: Getting the List of Files with Pending Changes

By clicking a file in the list, you can see the differences between the new and the old file versions, as shown in Figure 57. This tool is called **Diff**.

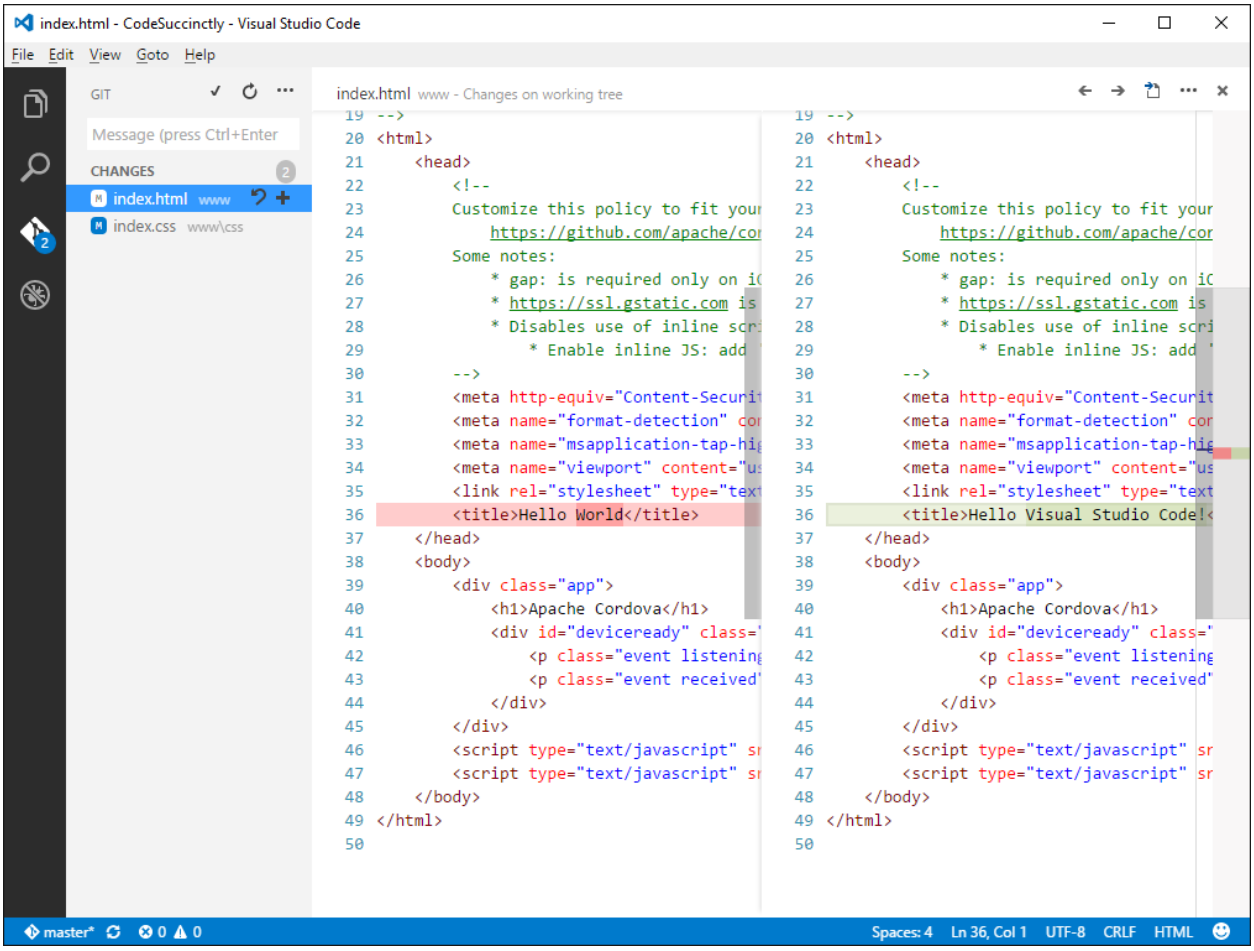


Figure 57: The Diff tool allows you to compare changes between versions.

On the left side you have the old version, while the new one is on the right. This is a very important tool when working with any version control engine. You can also 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 clicking the **+** symbol near its name. Visual Studio Code organizes staged files into a logical container, as you can see in Figure 58. Similarly, you can unstage files by clicking the **–** symbol.

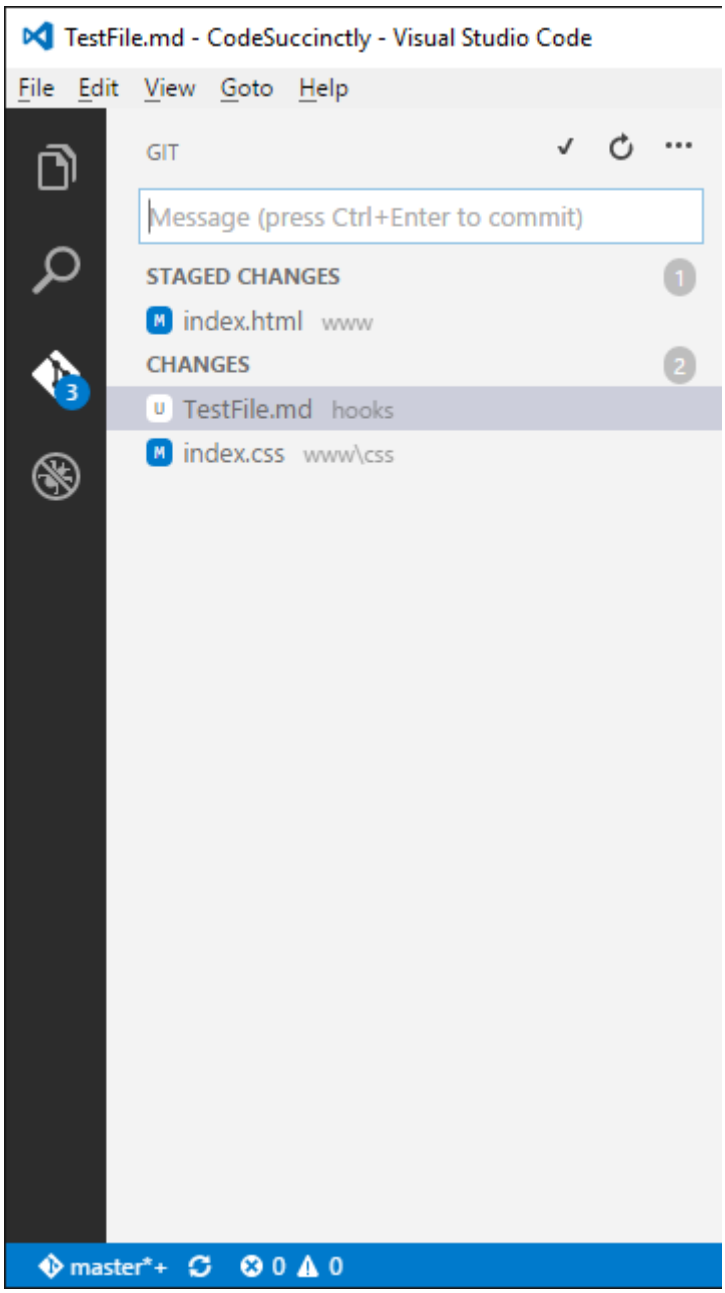


Figure 58: Staged changes are grouped separately.

Managing commits

When you are ready, you can click the **…** button and access additional actions, such as Commit, Sync, Pull, and Pull (Rebase). Figure 59 shows how you can commit all files, and also shows how files marked for deletion are represented with a red icon.

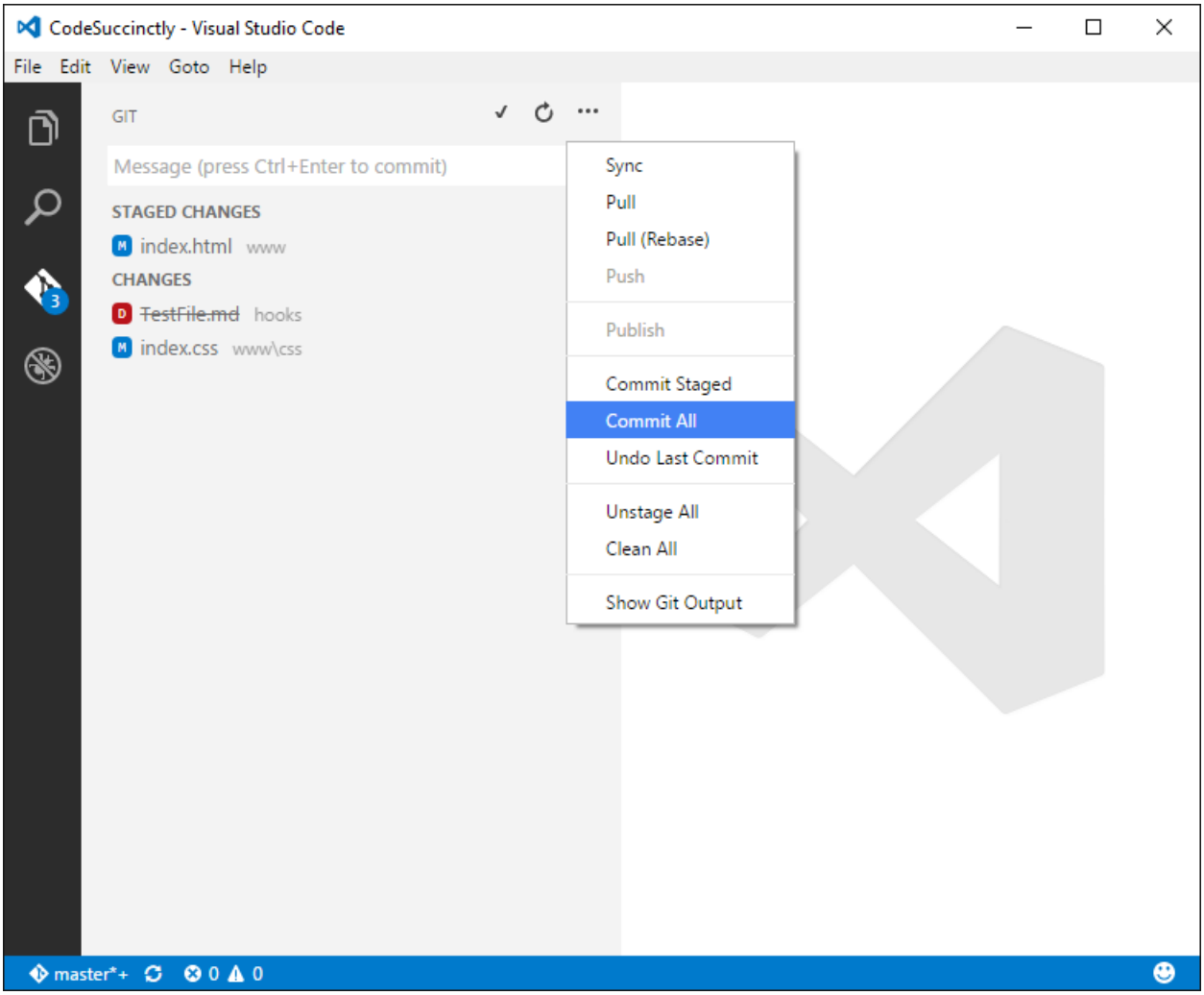


Figure 59: Accessing Git Commands

When ready, click **Commit All**. Remember that this action commits files to the local repository. You then have to **Sync** in order to synchronize changes with the remote repository. You also have an option to undo the last commit and revert to the previous version with the **Undo Last Commit** command. Pull and Pull (Rebase) allow you to merge a branch into another branch. Pull actually is non-destructive and merges the history of the two branches, while Pull (Rebase) rewrites the project history by creating new commits for each commit in the original branch.

Git commands

The Command Palette has support for specific Git commands which you can type as if you were in a command shell. Figure 60 shows the list of available Git commands.

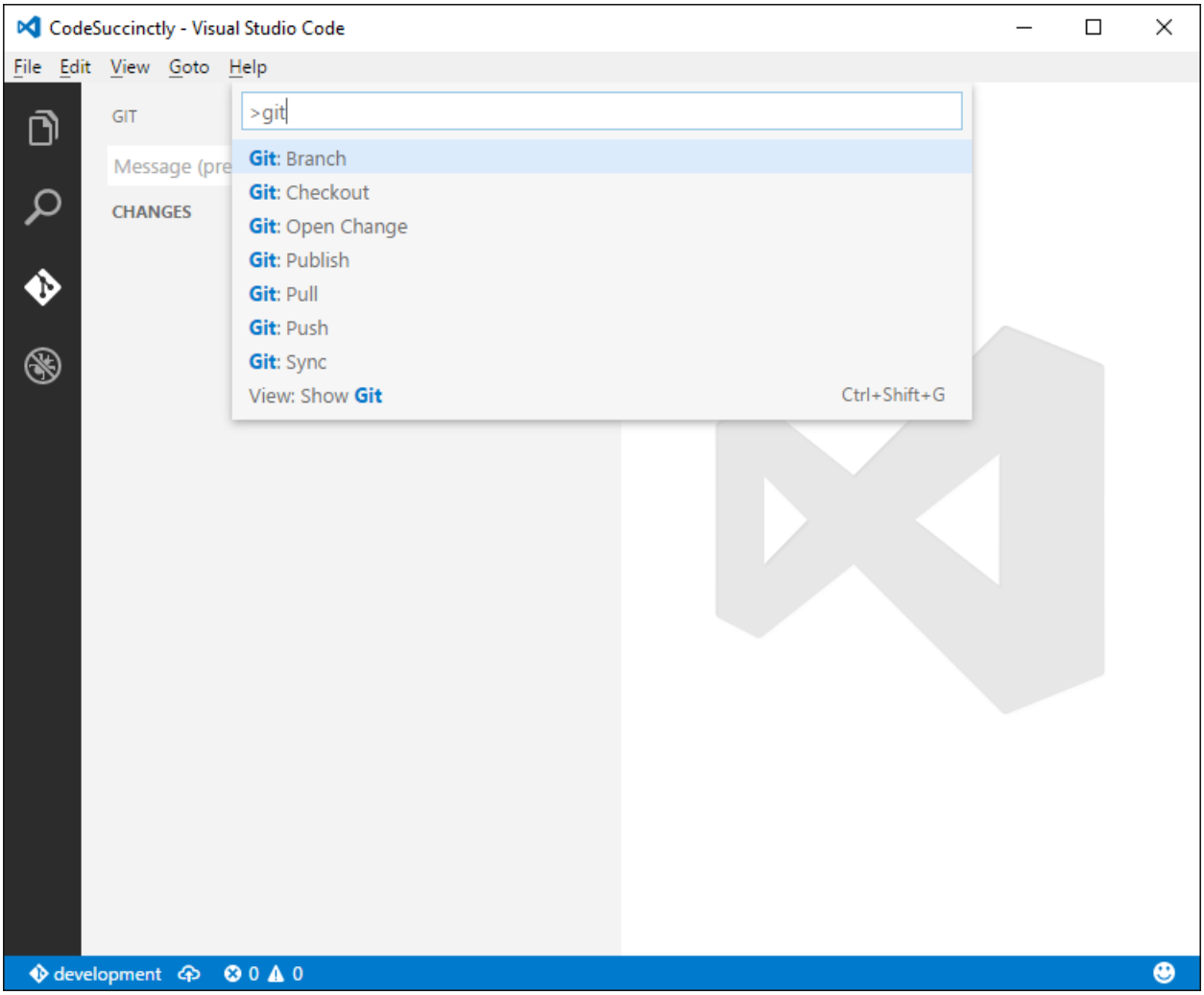


Figure 60: Supported Git Commands in the Command Palette

For instance, you can use **Git Sync** to synchronize the local and remote repository, 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.

Working with branches

I'll use an example to explain what a branch is. Say you have a project that, at a certain point in its life cycle, goes to production. Now you need to continue the development of your project, but you do not want to do it over the code you have written so far. You can create two histories by using a branch. When you create a repository, you also get a default branch called **master**. Continuing with the example, the master branch could contain the code that has gone to production and now you can create a new branch, one called **development**, based on master but different from it. In the Command Palette, you can type **Git branch** and then you will be asked to specify a new branch name. This will create a new branch locally, based on master. As you can see in Figure 61, the status bar shows the new branch as the active branch. When you are ready, you can publish the new branch to the remote repository with the **Publish Branch** button, represented by the cloud icon.

Figure 61: The New Branch and the Publish Branch Button

With this tool, you do not need to create and manage a branch in your Git repository provider.

Automating tasks

I have mentioned several times that Visual Studio Code is more than just a code editor; it is an end-to-end development environment. After reading the book this far, I expect you have a better understanding of what I mean. But there is more. Any respectable development environment needs to run external tasks, such as programs, compilers, or general purpose tools. Of course, this is the case with Code too. Visual Studio Code allows running external programs **Tasks**. When you run a Task, Visual Studio Code will display the Output window where the output of external programs is redirected to. This window will be covered shortly.

Note: What you will see in this section is a practical implementation of tasks. Of course, Visual Studio Code supports many, many types of tasks and more complex operations. That said, be sure to check out the official documentation about specific and complex task configurations.

Understanding the tasks.json file

When you want to run an external program, you need to configure a new task. A task is a set of instructions and properties represented with the JSON notation. An important thing to keep in mind is that tasks are only available for projects and folders, so you cannot run a task against a single code file. In order to configure a task, you first need to open a project or folder, and then you open the Command Palette and type **Configure Task Runner**. Visual Studio Code adds a subfolder called **.vscode** and generates a new file called **tasks.json**, which contains one or more tasks that will be executed against the project or folder. The auto-generated task.json contains a pre-configured task (see Code Listing 1) and a number of additional tasks enclosed within comments, for specific scenarios.

Code Listing 1

```
// A task runner that calls the TypeScript compiler (tsc) and
// compiles a HelloWorld.ts program.
{
      "version": "0.1.0",

      // The command is tsc. Assumes that tsc has been installed using
npm install -g typescript.
      "command": "tsc",

      // The command is a shell script.
      "isShellCommand": true,

      // Show the output window only if unrecognized errors occur.
      "showOutput": "silent",

      // args is the HelloWorld program to compile.
      "args": ["HelloWorld.ts"],

      // Use the standard tsc problem matcher to find compile problems
      // in the output.
      "problemMatcher": "$tsc"
}
```

The preconfigured default task starts the TypeScript compiler against a file called HelloWorld.ts. I'm going to show you how to implement your custom tasks shortly, but before that you need to understand what a task is made of. In its most basic form, a task is made of the following JSON elements:

- **`version`**. This represents the VS Code minimum version and should be left unchanged.
- **`command`**. This is the external program you want to run. Remember that you will need to include the full path for the program if an environment PATH variable has not been configured for it.
- **`isShellCommand`**. This represents whether a command is a shell script or not (you will leave this **`true`** in most cases).
- **`showOutput`**. This specifies when the Output window must show the output captured from the external program. Supported values are **`always`**, **`never`**, and **`silent`**. They will show the output window always, never, or only when unrecognized errors are detected.
- **`args`**: One or more arguments to be passed to the **`command`**. Typical arguments are file names and command-line switches.

Optionally, you can specify a **problem matcher**, which is a way to make Visual Studio Code understand and process styles of errors and warnings produced by external programs. For instance, the **`$tsc`** problem matcher (TypeScript) assumes that file names in the output are relative to the opened folder; the **`$mscompile`** (C# and Visual Basic) and the **`$lessCompile`** (Less) problem matcher assumes that file names are represented as an absolute path. Code supports many problem matchers out of the box, but you can define your own matchers for programs that are not supported directly. Also, you can configure multiple tasks and you will see an example based on Batch files later in this chapter. Discovering custom problem matchers is left to you as an exercise (look at the Visual Studio Code docs). Here, you will learn how to configure practical tasks such as running a Python program, executing a Batch file, and compiling a Visual Studio solution with MSBuild.

Tip: Tasks you define will not be executed until you save task.json, so do not forget this. Also, remember to close the task.json editor if your task runs against a different, active code file (or at least switch the view to the code file).

Predefined common variables

Visual Studio Code supports a number of predefined variables that you use instead of regular strings and are useful for representing file and folder names when passing these to a command. The following is a list of supported variables (which you can also see in the comments at the beginning of task.json):

- **`${workSpaceRoot}`**. The root folder.
- **`${file}`**. The active code file.
- **`${fileBaseName}`**. The active code file's base name.
- **`${fileDirname}`**. The name 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 spawned process.
- **`${env.VARIABLENAME}`**: References the specified environment variable, such as **`{$env.PATH}`**.

Using variables is very common when you run a task that works at the project or folder level or against file names that you either cannot predict or that you do not want to hardcode. You will see a couple examples in the next sections.

Sample 1: Running a Python program

Note: This section assumes you have already installed Python. If not, you can download it from the official website.

Python is a very popular programming language, and Visual Studio Code supports Python syntax out of the box. However, if you want to run a Python program you should launch the interpreter separately. Fortunately, you can configure a task and run Python from within Code. Because tasks run against projects and folders, the first thing you have to do is set up a folder on disk where you place your Python code files. The next thing to do is open this folder with Visual Studio Code. Figure 62 shows a sample folder called PythonPrograms, which contains a code file called StringAssignmentsSample.py, opened in Visual Studio Code.

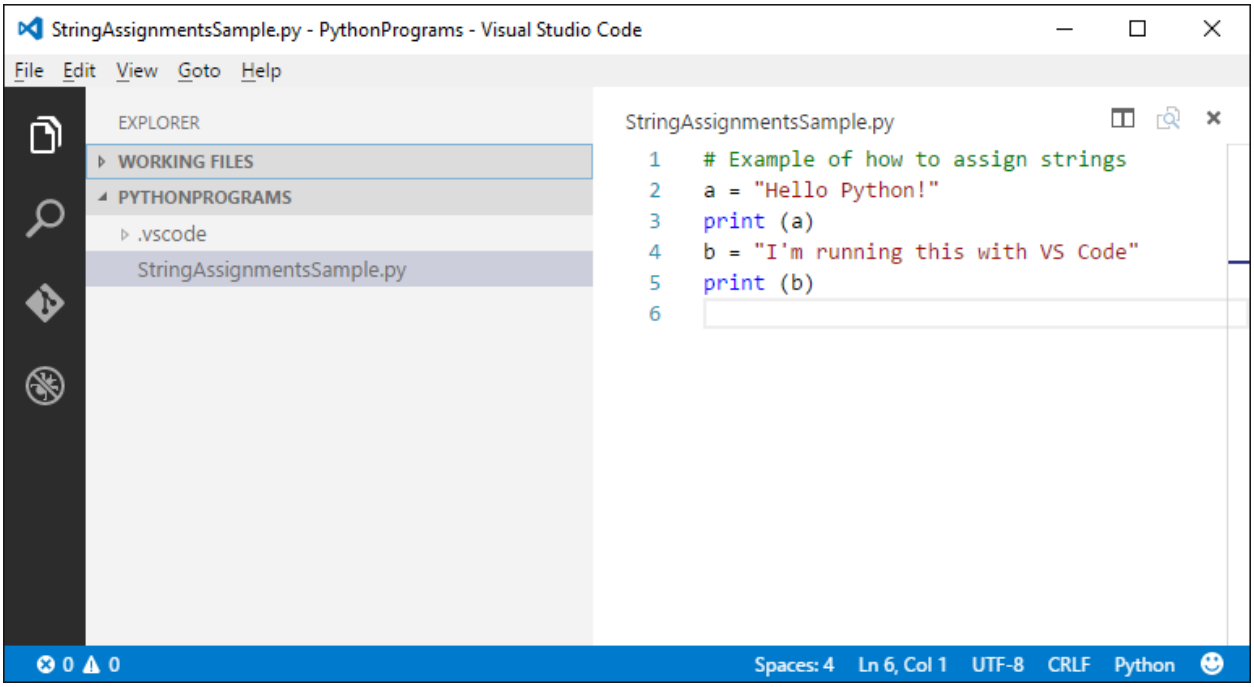


Figure 62: A Python Program Opened in Visual Studio Code

Press **Ctrl**+**Shift**+**P** or the **F1** key to open the Command Palette. Next, type **Configure Task Runner** or select the auto-fill suggestion item that appears while typing, as shown in Figure 63.

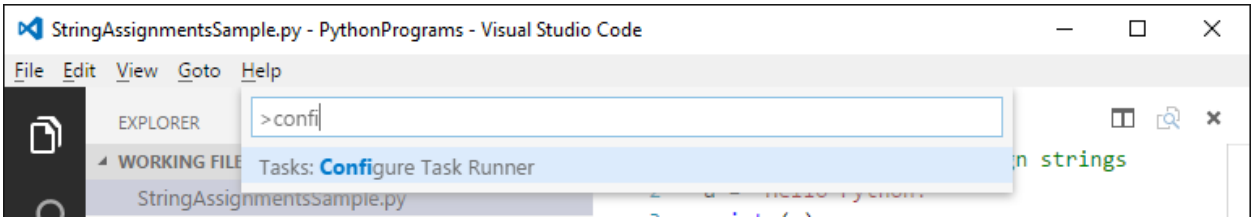


Figure 63: Invoking the Task Configuration Tool

At this point Visual Studio Code generates a new task.json file for the current folder. Either clear the content of the file or comment the pre-configured TypeScript task and write the new task shown in Code Listing 2.

Code Listing 2

```
// A task runner that calls the Python interpreter and
// runs the current program.
{
      "version": "0.1.0",

      // The command is Python.exe.
      "command": "C:\\Python34\\Python.exe",

      // The command is a shell script.
      "isShellCommand": true,

      // Always show the output window.
      "showOutput": "always",

      // args is the program to compile.
      "args": ["${file}"]
}
```

Code Listing 2 is very simple: It specifies the full path for the Python interpreter, because on my machine it is not registered with the PATH environment variable. Next, it specifies to always show the interpreter's output in the Output window, and the argument, which is a variable that represents the currently opened file that we want to run. Save task.json, close it, and then press **Ctrl**+**Shift**+**B** or type **Run Build Task** in the Command Palette. At this point, Visual Studio Code launches Python.exe against the active file and shows the result in the Output window, as shown in Figure 64.

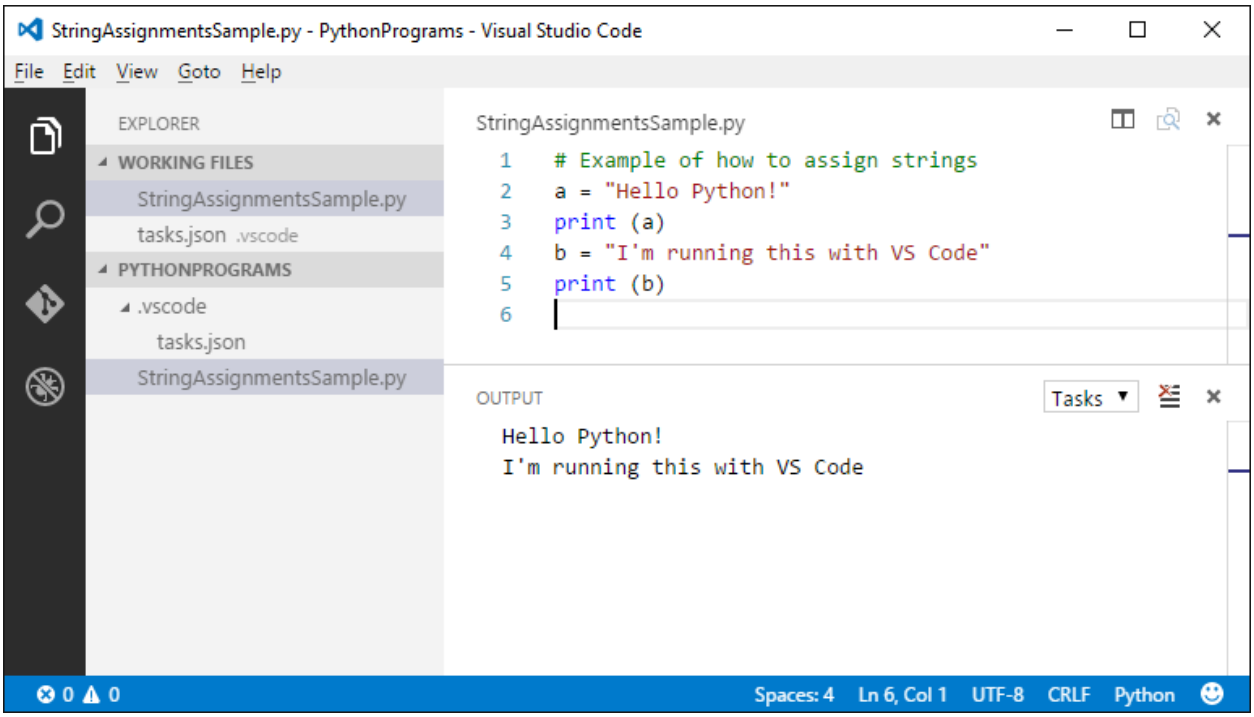


Figure 64: The Output window shows results from the external program.

As you can see, this is a very useful and powerful feature because it runs an external program and brings its output directly into the development environment.

Tip: The Output window is not interactive. For instance, if your program expects user input, the Output window is not able to capture it. Keep this in mind when you configure your tasks.

Sample 2: Executing Batch programs

With Batch files (.bat), you can execute one or more DOS commands supported by the operating system. Suppose you have a Batch file called CreateDir.bat, whose purpose is to create a folder on disk and redirect the output of the **DIR** command into a text file, which will be created in the newly created folder. Code Listing 3 shows the code for this.

Code Listing 3

```
@ECHO OFF

SET FOLDERNAME=C:\TEMP\TESTFOLDER
SET FILENAME=%FOLDERNAME%\TEXTFILE.TXT
MKDIR %FOLDERNAME%

ECHO %FOLDERNAME%  CREATED

REM REDIRECT OUTPUT TO A TEXT FILE
ECHO This is a text file created programmatically>>%FILENAME%

REM REDIRECT DIR LIST TO A TEXT FILE
DIR >>%FILENAME%

ECHO %FILENAME% CREATED
```

Now assume this file is stored in a folder that we open in Visual Studio Code. The goal is to run this Batch program from within Code itself. To do this, you again run the **Configure Task Runner** command, and edit tasks.json as shown Code Listing 4.

Code Listing 4

```
{
  "version": "0.1.0",
  "command": "${workspaceRoot}/CreateDir.bat",
  "isShellCommand": true,
  "showOutput": "always",
  "args": [ ]
}
```

In this case, your command is the Batch file whose execution is handled by the operating system that will treat it like an executable program. There are no arguments here because the Batch file does not expect any. Notice how the **`${workspaceRoot}`** variable is specified to tell Code that the file is in the root folder. If you save tasks.json and press **Ctrl**+**Shift**+**B**, Visual Studio Code will launch the Batch file and show the result in the Output window, as shown in Figure 65.

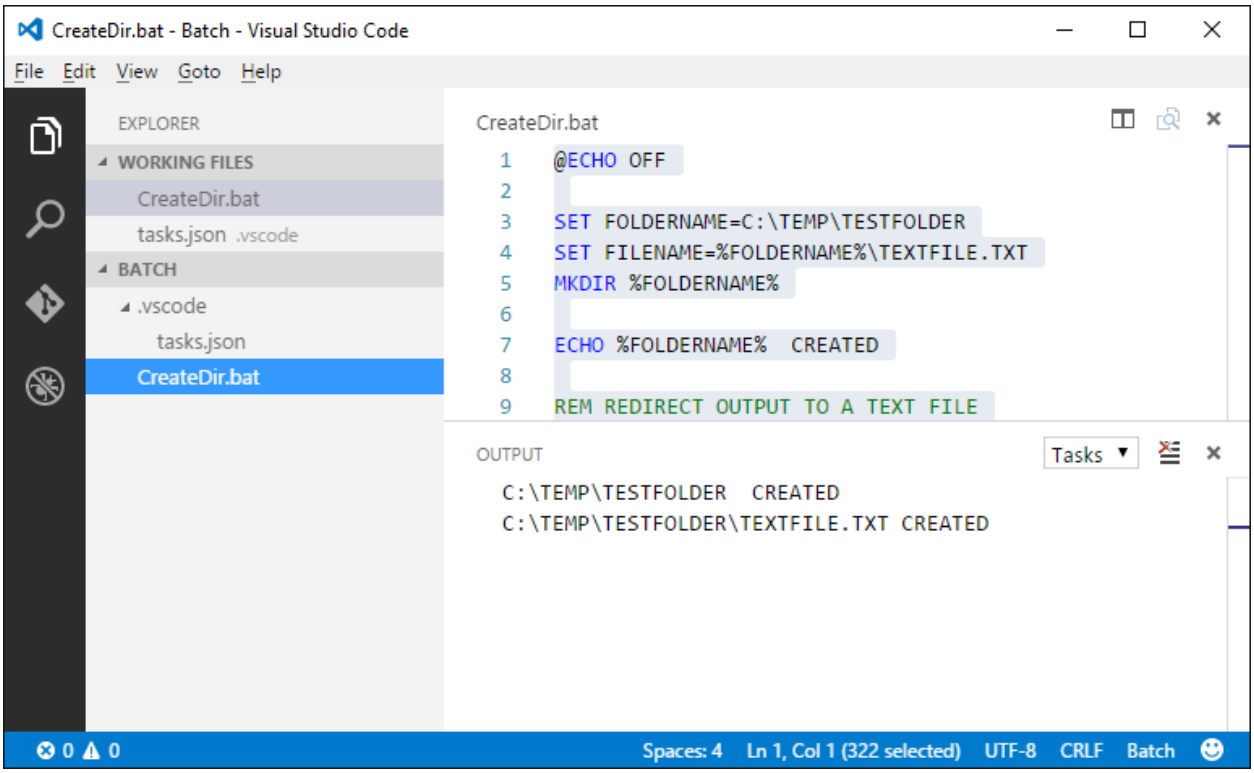


Figure 65: The Output window shows the result of executing a Batch file.

Sample 3: Compiling MSBuild solutions

Note: This example only works on Microsoft Windows.

If you are used to working with Microsoft Visual Studio, one of the things you would definitely want in Visual Studio Code is the ability to compile your solutions and projects. Though this should not become a requirement, it would definitely be nice to have, and actually you can get it in a few steps. In fact, 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 two Visual Basic projects, but of course all the steps apply to any .sln file and any supported languages. To enable MSBuild execution, you configure a task with the JSON code shown in Code Listing 5.

Code Listing 5

```
// A task runner that calls the MSBuild engine and
// compiles a .sln solution.
{
      "version": "0.1.0",

      // The command is msbuild.
      "command": "msbuild",

      // The command is a shell script.
      "isShellCommand": true,

      // Always show the output window for detailed information.
      "showOutput": "always",

      // args is the solution name.
      "args": ["WPCDemo.Sln"],

      // Use the standard problem matcher to find compile problems
      // in the output.
      "problemMatcher": "$msCompile"
}
```

It is worth noting that the command is **`msbuild`** and that the task is using the standard **`$msCompile`** problem matcher to find compile problems that are specific to Visual Basic and C#. Running this task by pressing **Ctrl**+**Shift**+**B** causes VS Code to run MSBuild.exe over the specified solution file, producing the sample output shown in Figure 66.

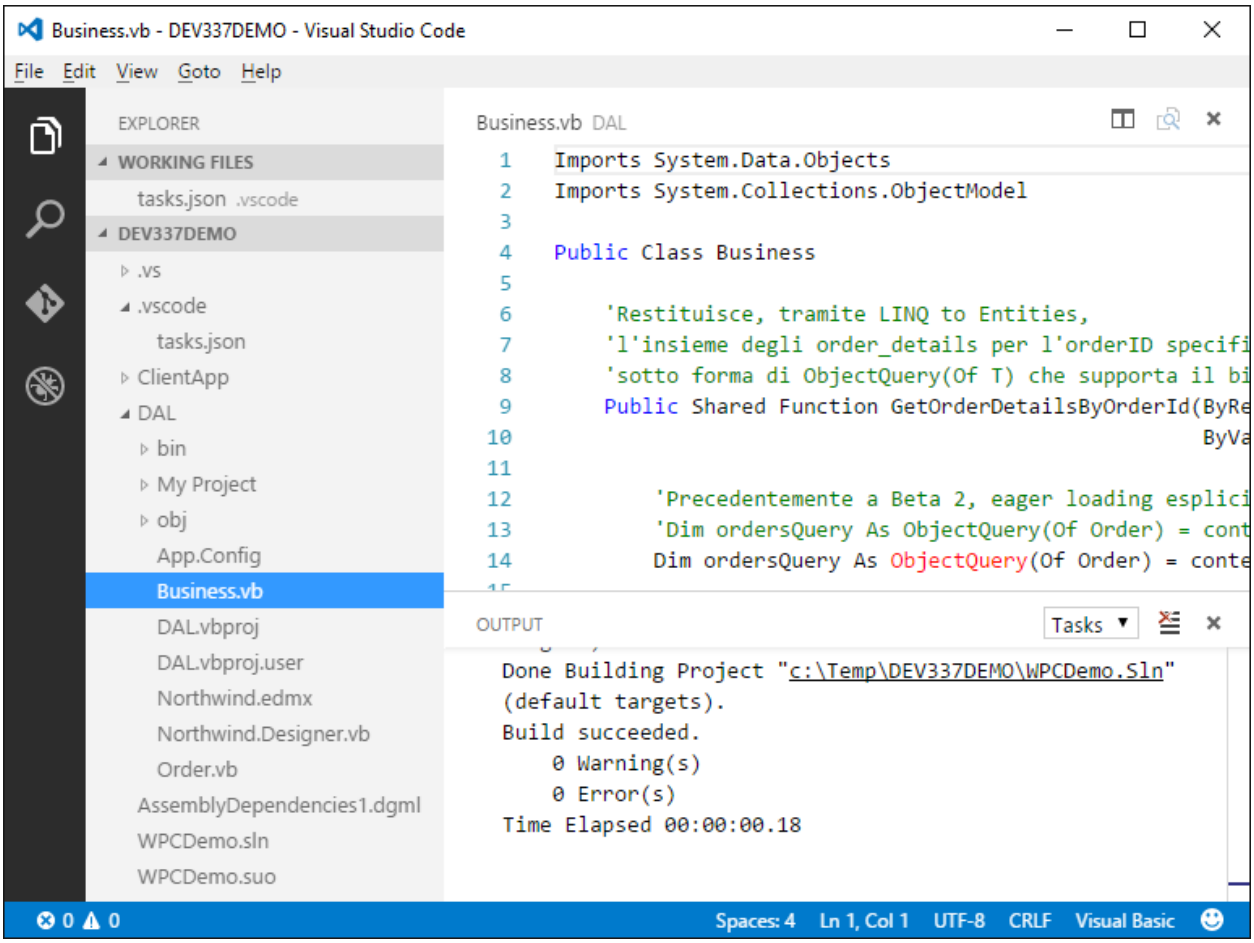


Figure 66: The Output window shows the result of compiling an MSBuild solution.

If you want to use the powerful editing features of Visual Studio Code and you want to be able to compile your solutions, this option is definitely a nice addition.

Understanding and configuring multiple tasks

So far you have seen how to configure one task. In practice, you might need to configure multiple tasks for a project or folder. Out of the box, Visual Studio Code supports two tasks with conventional names: build task (**Ctrl**+**Shift**+**B**) and test task (**Ctrl**+**Shift**+**T**). If you configure just one task, as you did in the previous examples, it is automatically considered a build task. If you want to differentiate build and test tasks, you need to specify this in tasks.json. Code Listing 6 shows an example based on the MSBuild task configured previously.

Code Listing 6

```
{
  "version": "0.1.0",
  "command": "msbuild",
  "isShellCommand": true,
  "showOutput": "always",
  "args": ["WPCDemo.sln"],
  "tasks": [
    {
      "taskName": "build",
      "isBuildCommand": true,
      "showOutput": "always"
    },
    {
      "taskName": "test",
      "showOutput": "never"
    }
  ]
}
```

The following is a list of key points about configuring multiple tasks:

- A JSON array called **`tasks`** is used to specify multiple tasks.
- Each task is represented by the **`taskName`** property.
- Because you no longer have a default task, the **`isBuildCommand`** property set as **`true`** indicates which of the tasks the build task is associated with Ctrl+Shift+B.

Code Listing 6 defines two tasks for the build and test tasks supported by Visual Studio Code. The build task is available by pressing **Ctrl**+**Shift**+**B**, and the test task is available by pressing **Ctrl**+**Shift**+**T**. Both are also available through the Command Palette, as you can see in Figure 67.

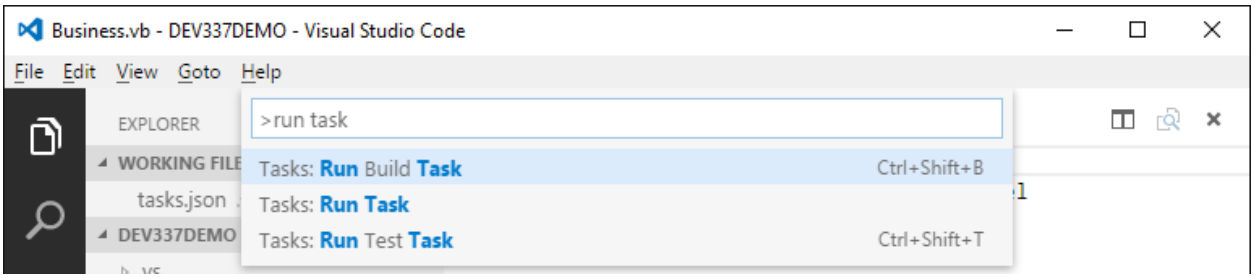


Figure 67: Invoking the build and test tasks from the Command Palette.

Not limited to this, you can also specify additional tasks with custom names. Code Listing 7 shows how to define a new task called onError, which is displayed in the Output window only when compile errors are detected. Finally, Figure 68 shows how you can invoke the custom task from the Command Palette.

Code Listing 7

```
{
  "version": "0.1.0",
  "command": "msbuild",
  "isShellCommand": true,
  "showOutput": "always",
  "args": ["WPCDemo.sln"],
  "tasks": [
    {
      "taskName": "build",
      "isBuildCommand": true,
      "showOutput": "always"
    },
    {
      "taskName": "test",
      "showOutput": "never"
    },
    {
      "taskName": "onError",
      "showOutput": "silent"
    },
  ]
}
```

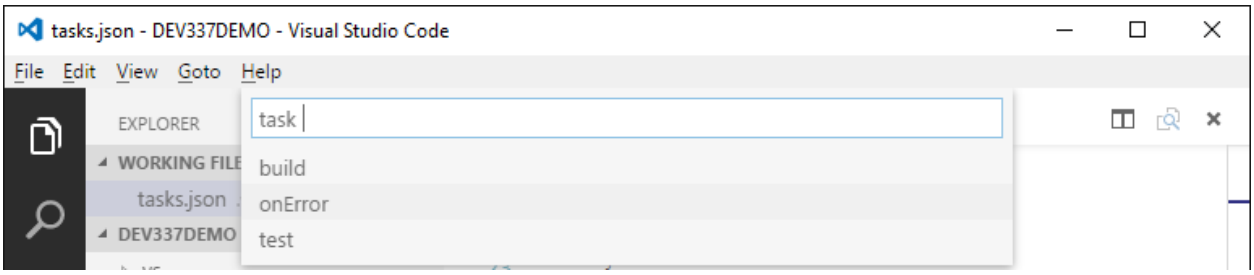


Figure 68: Invoking Custom Tasks from the Command Palette

By defining multiple tasks, you can run external programs over your project and folders targeting different configurations.

Chapter Summary

Visual Studio Code provides integrated support for source code version control based on the Git engine, offering tools that make it easy to commit and manage your changes, including branches. Code also allows you to define build, test, and custom tasks, enabling you to run external programs such as compilers over your projects and folders. These are key features that make Visual Studio Code different from a simple source code notepad. Now you have in your hands all the powerful features you need to start writing great applications with Code, which is the topic of the next chapter.

Chapter 4 Creating and Debugging Applications

Being an end-to-end development environment, Visual Studio Code offers opportunities that you will not find in other code editors. In fact, with Code you can create and debug a number of application types. This chapter will get you started writing web applications with Visual Studio Code, providing guidance on debugging as well where supported.

Chapter prerequisites

Before you start writing web applications with Visual Studio Code, you need to make sure you have installed the following prerequisites:

- Node.js, a JavaScript runtime that includes, among the others, the **npm** (Node Package Manager) command-line tool.
- DNX, the.NET Execution Environment required to build cross-platform web applications with ASP.NET Core 1.0 (formerly ASP.NET 5), the new development platform from Microsoft for the next generation of cross-platform web apps.

DNX integrates well with Node.js and provides additional command-line tools that you will use very often. The next step is installing Yeoman, a very popular command-line tool that helps you with scaffolding a variety of modern web projects. Installing Yeoman is very easy: Open a command prompt and type the following line, assuming you have already installed Node.js:

```
> npm install -g yo generator-aspnet gulp bower
```

This line will install the ASP.NET Core generator from Yeoman (also referred to as **yo**), plus **Gulp** and **Bower**.

Note: If you are new to cross-platform web development, you might wonder what Gulp and Bower are. In a few words, Gulp is a tool that simplifies the automation of repetitive tasks whereas Bower is a library and package manager for web apps. Click their links in the previous paragraph to get more information.

If you are wondering why you need these additional tools, the answer is very simple: They allow creating and managing applications in a way that is completely platform and environment independent. For instance, Yeoman generates ASP.NET Core applications based on project.json files, which is suitable for Visual Studio Code and many other development tools on different operating systems and platforms. This is different from creating ASP.NET Core apps with Microsoft Visual Studio, which generates proprietary solution files. Though the result is the same, and though Visual Studio is much more powerful, of course, you need Windows and Visual Studio itself. Another prerequisite is installing the Express application framework that you use to scaffold a Node.js application. The command you type in the command line is the following:

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

Now you have all the necessary tools and are ready to go.

Creating an ASP.NET Core web application

To create your first ASP.NET Core web application for Visual Studio Code, you need to run the Yeoman tool from the command line and type a few lines.

Tip: The previous sentence is true in part. You can actually launch Yeoman from the Command Palette in Visual Studio Code by typing** Yo**, pressing** Enter**, and then selecting the proper commands. However, you must first have knowledge of the commands themselves; this is why you should type them manually first.

Yeoman generates a new web application in a subdirectory of the current directory, so you must be sure of where you want to create the application. For instance, suppose you have a folder called C:\MyWebApps. Open a command prompt for this folder, and then type the following line:

```
> yo aspnet
```

This will launch the Yeoman ASP.NET 5 generator, which provides a number of possible choices as shown in see Figure 69.

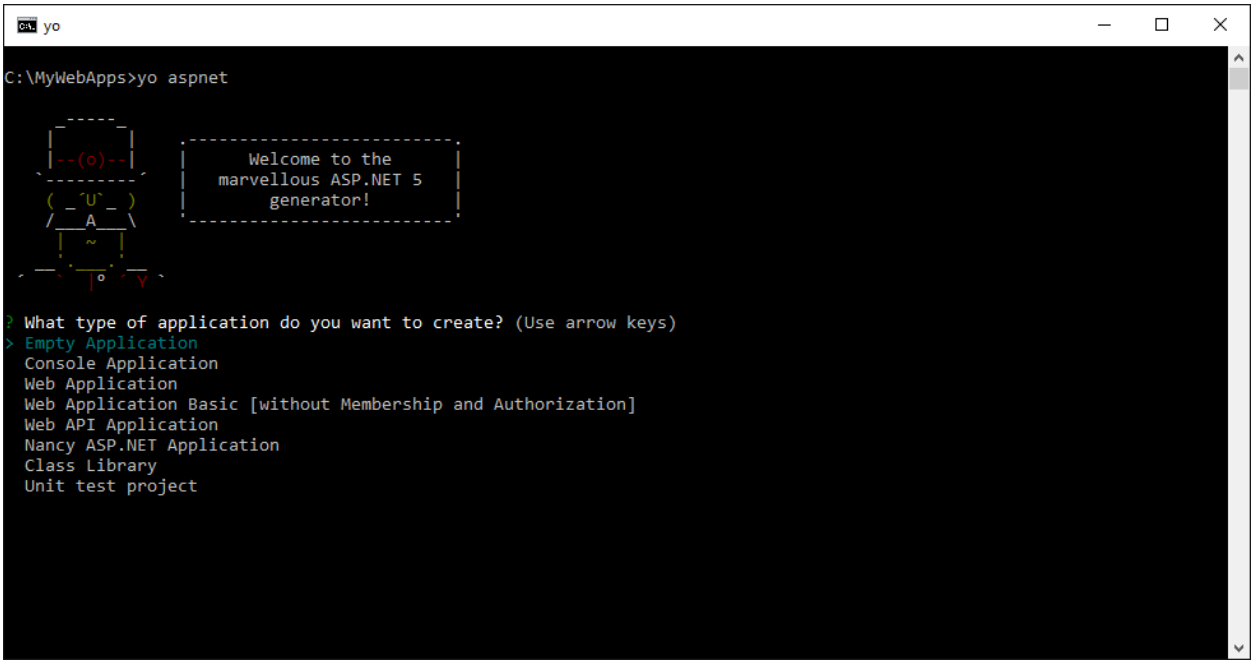


Figure 69: The Yeoman ASP.NET 5 Generator

Notice how you can create a variety of projects, including the popular Web API. Select the **Web Application** template and press **Enter**. You will be prompted to enter a name for the application. For consistency, type **CodeSuccinctlyWeb** and press **Enter**. Yeoman generates a subfolder called C:\MyWebApps\CodeSuccinctlyWeb and all the necessary files for scaffolding the application in this folder. You can see the generation progress in the Console window and, at completion, Yeoman suggests a sequence of commands to run (see Figure 70).

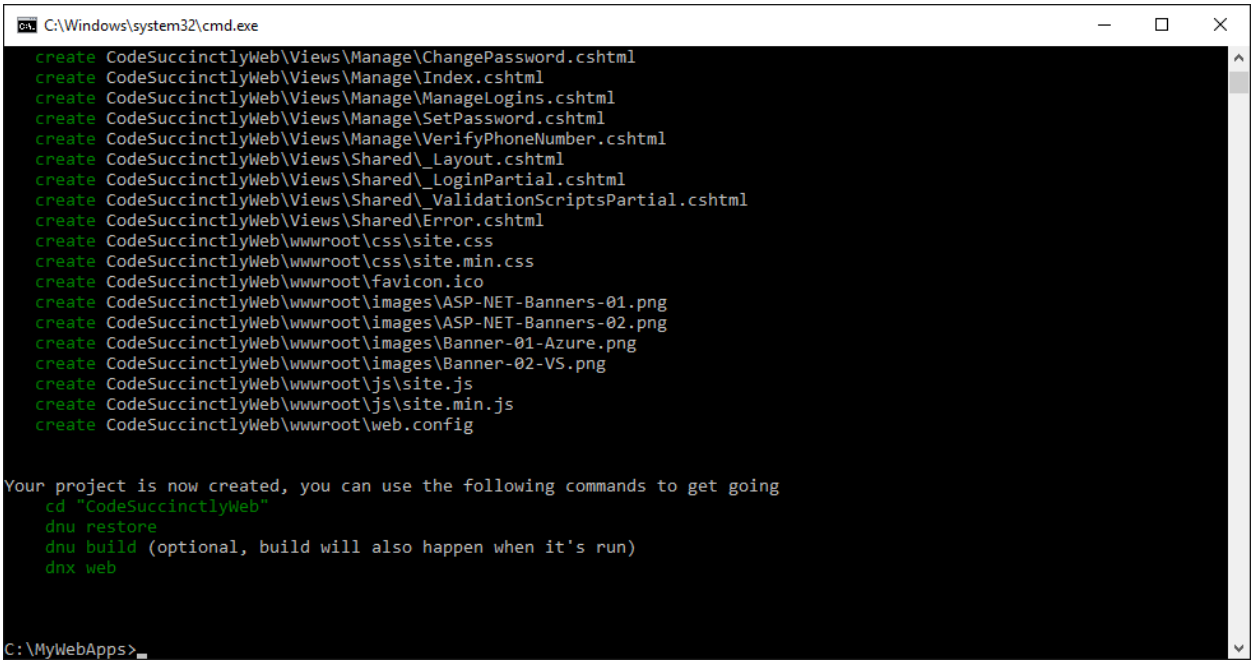


Figure 70: Progress Reporting and Command Suggestions

For now, you can skip running the suggested commands, as you will do this from within Visual Studio Code shortly.

Tip:** dnu **stands for .NET Utilities and allows executing specific actions against projects. Type** dnu –help **for further information

Now open Visual Studio Code, select **File** > **Open Folder**, and select the **CodeSuccinctlyWeb** folder. Visual Studio Code will find a project.json file and organize the files in a proper view.

Note: Explaining how an ASP.NET Core project is structured is beyond the scope of this book. Here we focus on how to work with ASP.NET Core projects in VS Code. For further details, refer to the official ASP.NET Core documentation.

Restoring NuGet packages

When Code opens an ASP.NET Core project, it analyzes the project.json file and checks if all the required NuGet packages are available. If they aren't, it shows a message and offers to restore NuGet packages for you (see Figure 71). Missing NuGet packages are probably the most common cause for many errors in the source code. In Figure 71 you can see a huge number of red squiggles regarding the fact that the code editor does not know how to parse that code unless the required packages are restored.

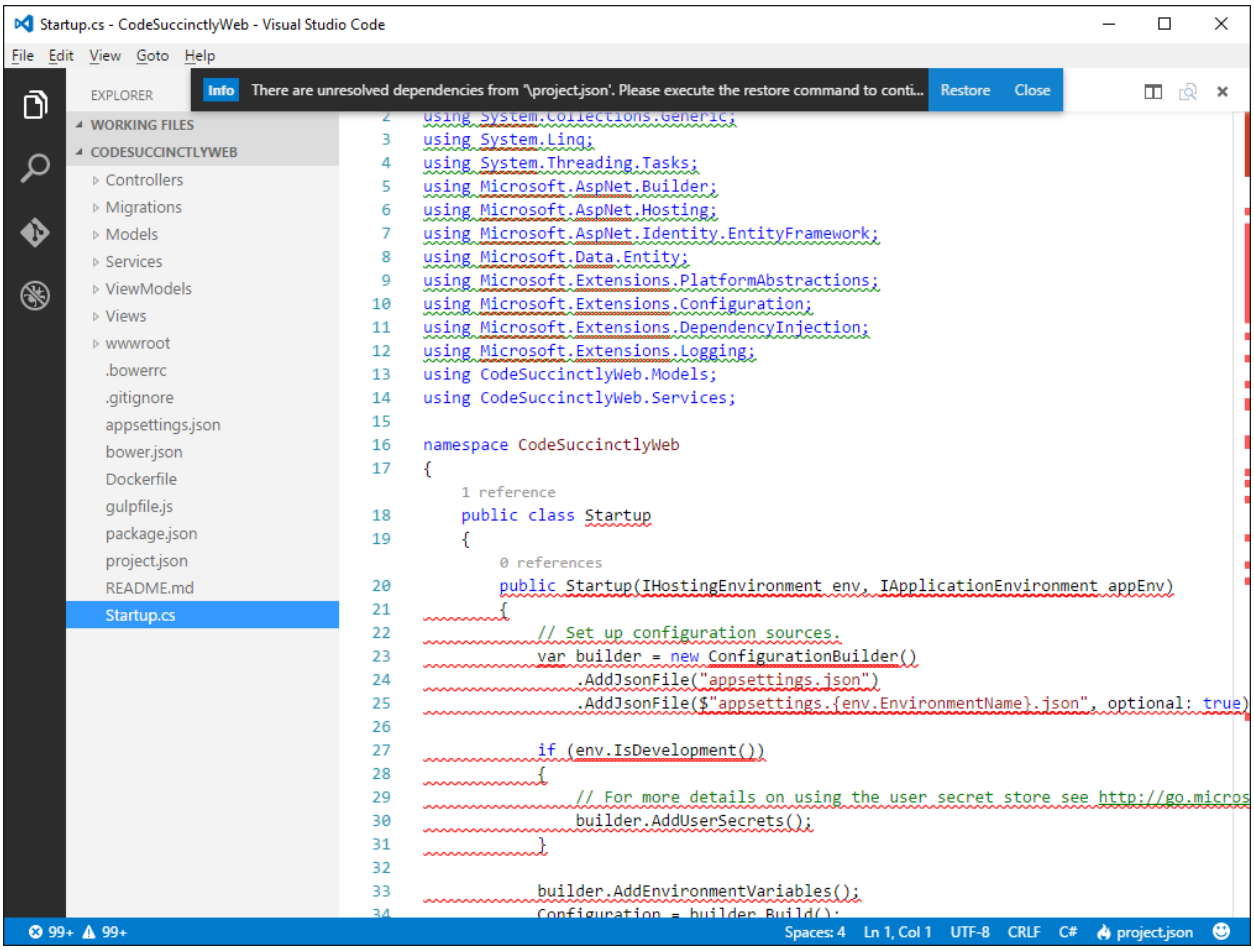


Figure 71: Code offers to restore missing NuGet packages.

You can manually request to restore NuGet packages in the Command Palette by first typing **`dnx restore packages`**, and then selecting the **`dnu restore`** command. If you have multiple projects in your folder, you will have an option to target a specified project. Once you restore all the NuGet packages, the red squiggles will go away.

Running the application

You have just built your first ASP.NET 5 application for Visual Studio Code and you are ready to see how to get it up and running. In the Command Palette, type **`dnx run command`**. This will display two available commands: **`dnx web`** and **`dnx ef`**. The first command builds and starts the web application, whereas the second command allows Entity Framework commands to be executed against a project. For now, execute **`dnx web`**. To deploy your applications, Visual Studio Code uses a new development server that replaces IIS Express, called **Kestrel**. When you execute the command, a console window appears providing information about the application execution, including errors and problems, during the entire life cycle. By default, Kestrel listens for the application on port 5000, which means your application can be reached at http://localhost:5000 and will appear as shown in Figure 72.

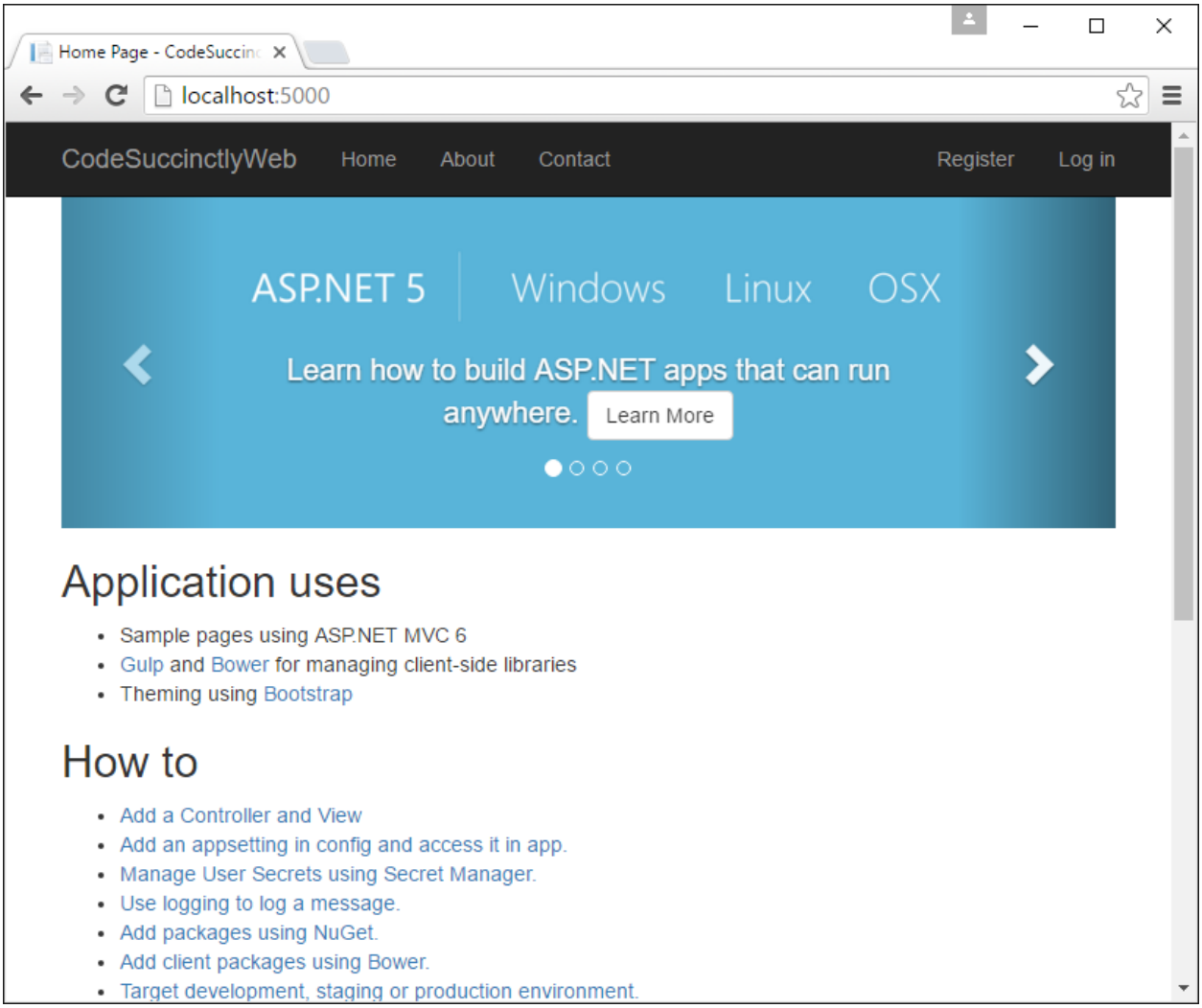


Figure 72: The sample web application is up and running.

Tip: You can change the default port (5000) in the file called Dockerfile.

Alternatives for publishing your ASP.NET 5 applications to different production server types are explained in the documentation. Microsoft has also produced a great tutorial that explains how to publish your ASP.NET 5 application to Microsoft Azure in a few steps.

A step forward: Implementing data access with Entity Framework 7

Note: If you typically work with Microsoft Visual Studio, you will certainly perform the steps described in this section in that IDE rather than in VS Code. However, here we demonstrate how many platforms and technologies from Microsoft can be used with cross-platform applications outside of Visual Studio.

Most applications, including web apps, need to work with data. You can find many tutorials about writing applications with VS Code, but very limited resources about implementing data access, so I want to guide you through this. As you might know, Entity Framework is the data access platform for modern applications from Microsoft, which has been recently open-sourced. Version 7 has some notable improvements, plus it works well with mobile and ASP.NET Core applications. You can easily implement data access in your ASP.NET Core application by using the **`dnx ef`** command, where **`ef`** stands for Entity Framework. Though the Command Palette allows invoking Entity Framework commands from within Code, a better approach is to use the command prompt. With Entity Framework, you can target many scenarios such as creating a data model based on an existing database, creating a data model and then scripting the database, or using the Code First approach to define a data model for both a new or existing database. For the current sample application, I will show you how to generate a data model based on the popular Northwind database and how to expose queries through a controller. Before you do anything else, in Visual Studio Code open the **project.json** file for the current project and, in the **dependencies** node, add a reference to the **EntityFramework.MicrosoftSqlServer.Design** library, with the same version number as the **EntityFramework.MicrosoftSqlServer** library. This is a fundamental step, otherwise scaffolding will fail immediately. When you save project.json, Visual Studio Code will detect the missing NuGet package and will offer to download it for you. Of course accept the offer, but if this does not happen, perform a **`dnu restore`** command manually. Now open a command prompt pointing to the **C:\MyWebApps\CodeSuccinctlyWeb** folder. When ready, type the following line:

```
> dnx ef
```

This will activate the Entity Framework command line for the current folder. In the console window, you will see a list of available commands, such as database, context, and migrations. You already have a database in this case, so the command you need to work with is **context**, which allows generating a **`DbContext`** class and will scaffold the data model. Depending on your database configuration, write the following command:

```
> dnx ef dbcontext scaffold
"Server=.\sqlexpress;Database=Northwind;Trusted_Connection=True;"
EntityFramework.MicrosoftSqlServer
```

Replace the server name with your instance of SQL Server if necessary. After a few seconds, you will see a completion message. If you take a look at the project in Visual Studio Code, you will see new files have been added to the project. More specifically, there is a class for each table in the database (e.g., Customers.cs, Orders.cs, Order_Details.cs), plus the context class defined inside the NorthwindContext.cs file. Suppose you want your web application to implement queries over data. The proper way to do this is with a Web API controller, which you can add using Yeoman. Because you have an existing and configured project, you can run Yeoman from the Command Palette. So, type **`yeoman aspnet`**. At this point, the Command Palette shows a list of items that can be added to an existing ASP.NET Core project. Select the **WebApiController** item and, when prompted, enter **CustomersController** as the controller name (see Figure 73).

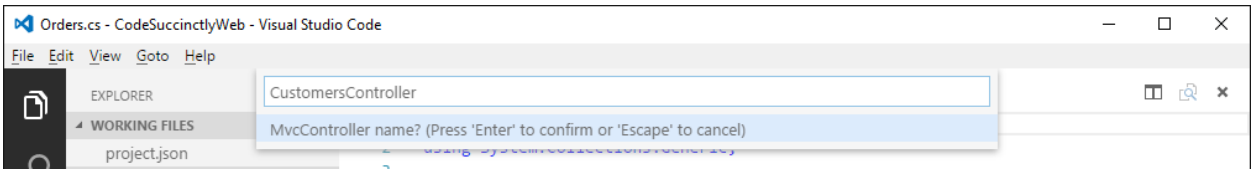


Figure 73: Adding a new Web API Controller

The new controller is added to the Controllers folder and the Output window shows information about the code generation. If you look at the error list, you will see an error stating **An anonymous type cannot have multiple properties with the same name**. This is a known issue and you can fix it by navigating to the code where the error is and remove the duplicate properties. Web API controllers accept requests through the common HTTP verbs, such as GET, POST, PUT, and DELETE. For this reason, the new controller has auto-generated methods matching some of the most common HTTP verbs. Imagine you want to return a filtered list of customers. To do this, rewrite the **`Get`** method as shown in Code Listing 8.

Code Listing 8

```
        NorthwindContext context=new NorthwindContext();
        // GET: api/values
        [HttpGet]
        public IEnumerable<string> Get()
        {
            var query = from cust in context.Customers
                        select cust.CompanyName;

            return query;
        }
```

Also, remember to add a **`using CodeSuccinctlyWeb`** directive. Before testing the application, you need a final edit. Open the **Web.config** file under the **wwwroot** folder and add the connection string declaration as follows:

```
  <connectionStrings>
    <add name="NorthwindContext"
         connectionString="Data Source=.\SqlExpress;Initial
Catalog=Northwind;Integrated Security=True"
         providerName="System.Data.SqlClient" />

  </connectionStrings>
```

In real-world scenarios, you might want to supply the connection string in your C# code instead of placing it in the configuration file, but it is not necessary in this case. Now run the application again with commands you learned previously and open a web browser to the http://localhost:5000 web address. By following the routing conventions of Web API, you can get the list of customers by typing the following address: http://localhost:5000/api/Customers. The result you get is shown in Figure 74.

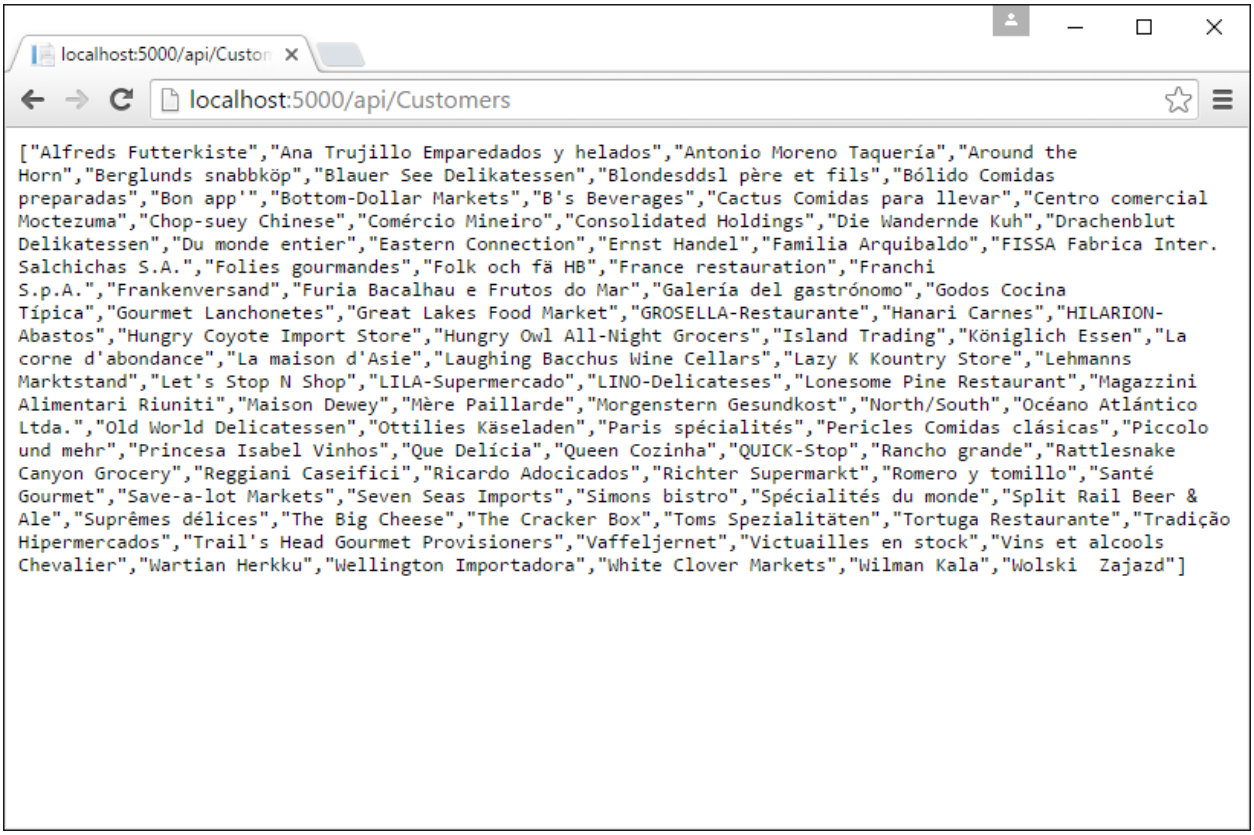


Figure 74: Querying Data Exposed by a Web API Controller

With a few steps, you have been able to implement data access in your ASP.NET Core application using Entity Framework. Additional steps might involve adding code migrations, CRUD operations, and proper views to present information with a user interface.

Debugging in Visual Studio Code

One of the most important tools in Visual Studio Code is the built-in debugger, which is another key feature that marks a significant difference compared to other code editing tools. At the time of this writing, Code has integrated support for Node.js (JavaScript) applications and experimental support for Mono with C# and F# via a special extension. Additional debuggers produced by the developer community can be found in the Visual Studio Marketplace.

To try out the debugging, the first thing you need is a Node.js application. If you do not have one, you can use the Express application framework, which makes it easy to scaffold a new Node.js app. Express works similarly to Yeoman, so it creates an application in a subfolder of the current directory. Open a command prompt pointing to a directory where you want to place your new application's folder, such as the sample C:\MyWebApps directory created previously. When ready, write the following command in the command prompt to create a sample application called SampleNodeJsApp:

```
> express SampleNodeJsApp
```

After a few seconds, the application will be ready. Notice that at completion, Express suggests you install the application dependencies (see Figure 75), so navigate to the subfolder and type **`npm install`**.

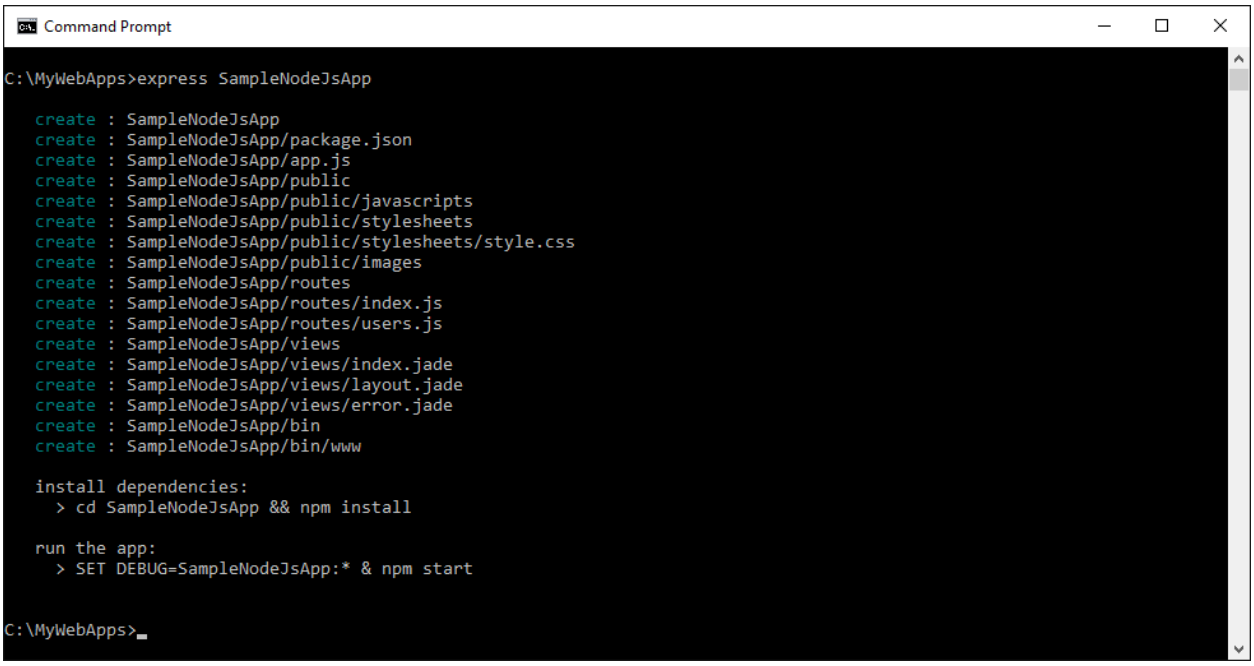


Figure 75: Creating a Node.js Application with Express

Testing the application is very easy, as you simply need to type **`npm start`**. By default, this will start the server listening to port 3000, so in your web browser you can reach the application at http://localhost:3000. Figure 76 shows the sample application running.

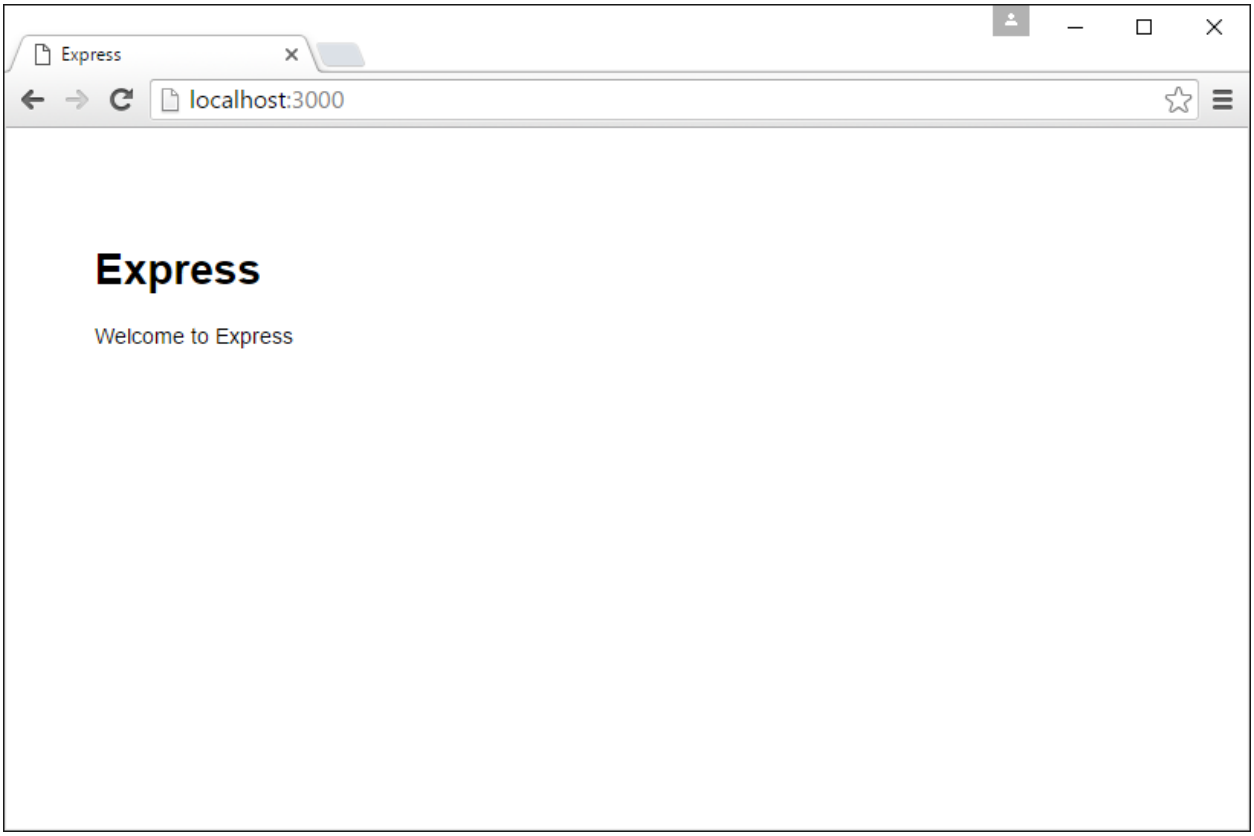


Figure 76: The Node.js application created with Express is running.

Behind the scenes, the project contains a package.json file which defines a script called **`start`** that runs **`node ./bin/www`**. This provides the startup mechanism.

Opening the Node.js application in Visual Studio Code

Start Visual Studio Code and open the **SampleNodeJsApp** folder. As expected, Code recognizes the application type, loads files, and provides the proper view. Open the **app.js** file and after the **`var app = express();`** line of code, add the following:

```
var welcomeText='Welcome to Node.js';

console.log(welcomeText);
```

This is a simple string variable declaration that will be useful for debugging purposes. Notice how powerful the IntelliSense is while you type.

Configuring the debugger

Before you can debug the application, you need to configure the debugger. First, open the **Debug** view. Second, click the gear icon that represents the **Open launch.json** command. VS Code will show a list of available debugging environments, depending on how many debuggers you have installed. Simply select **Node.js** and press **Enter**. At this point, Code creates a file called launch.json, which is where you configure the debugger. Visual Studio Code supports two debugging configurations: launch and attach. You use launch to debug the project that is currently opened in Code whereas you use attach to debug an already running application. When Code generates launch.json, it automatically provides default values for both configurations, as shown in Code Listing 9.

Code Listing 9

```
{
      "version": "0.2.0",
      "configurations": [
            {
                  "name": "Launch",
                  "type": "node",
                  "request": "launch",
                  "program": "${workspaceRoot}\\bin\\www",
                  "stopOnEntry": false,
                  "args": [],
                  "cwd": "${workspaceRoot}",
                  "preLaunchTask": null,
                  "runtimeExecutable": null,
                  "runtimeArgs": [
                        "--nolazy"
                  ],
                  "env": {
                        "NODE_ENV": "development"
                  },
                  "externalConsole": false,
                  "sourceMaps": false,
                  "outDir": null
            },
            {
                  "name": "Attach",
                  "type": "node",
                  "request": "attach",
                  "port": 5858,
                  "address": "localhost",
                  "restart": false,
                  "sourceMaps": false,
                  "outDir": null,
                  "localRoot": "${workspaceRoot}",
                  "remoteRoot": null
            }
```

```
        ]
}
```

In most cases you will leave the default values unchanged, but you might need to tweak values for the attach configuration. The properties are self-explanatory and easy to understand, but you can always have a look at the official documentation for further details. Make sure you have saved launch.json before going on.

Managing breakpoints

Before starting the application in debugging mode, it is useful to place one or more breakpoints to discover the full debugging capabilities in VS Code. In order to place breakpoints, you click the white space near the line number. For instance, place a breakpoint on the **`welcomeText`** variable declaration you wrote previously, as shown in Figure 77.

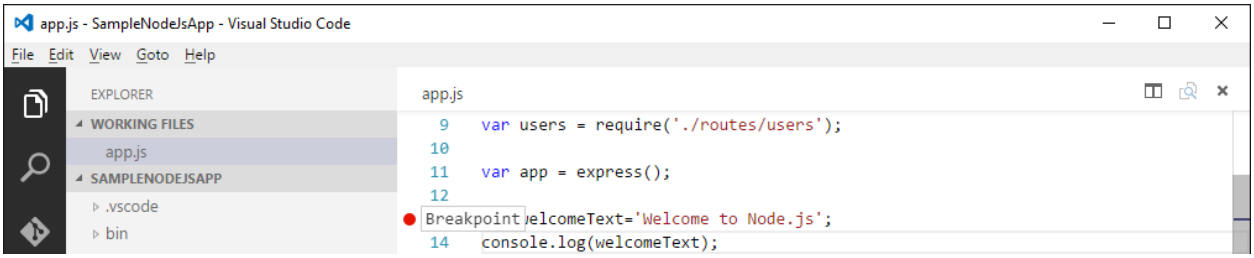


Figure 77: Placing Breakpoints in the Code Editor

You can remove a breakpoint by simply clicking it again, or you can manage breakpoints in the Breakpoints area of the Debug view (see Figure 78).

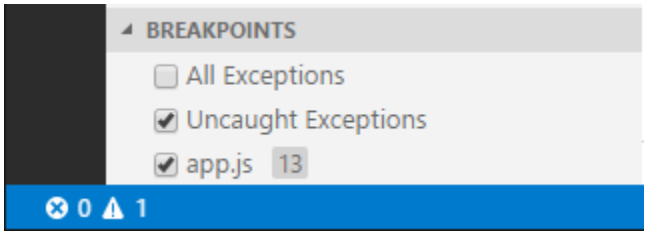


Figure 78: Managing Breakpoints

Debugging the application

In the Debug view, make sure the launch configuration is selected, then click the Start button or press F5. Visual Studio Code will launch the debugger and will break when it encounters an exception or a breakpoint, like in the current example. Figure 79 shows Code hitting a breakpoint and all the debugging instrumentation.

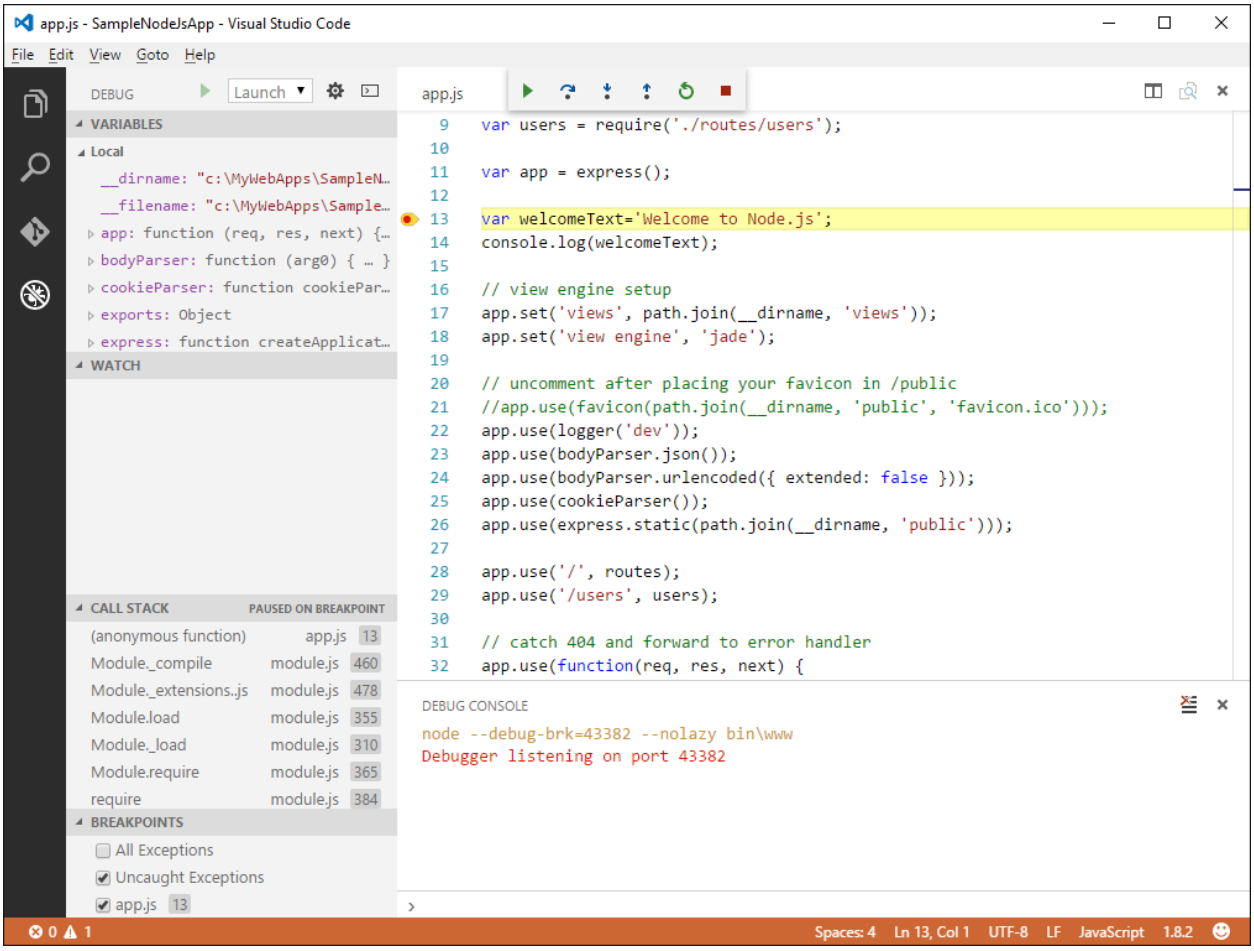


Figure 79: The debugger encounters a breakpoint and shows all the instrumentation.

Notice how the status bar turns orange while debugging and how the Output 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's control and that you can investigate by expanding each variable.
- **Watch**, where you can evaluate expressions.
- **Call Stack**, where you can see the stack of method calls. If you click a method call, the code editor will bring you to the code that is making that call.
- **Breakpoints**, where you can manage breakpoints.

At the top of the IDE, also notice the debugging toolbar (see Figure 80) called **Debug actions pane**, which holds the following commands from left to right:

- **Continue (F5)**, which allows continuing the application execution after breaking on a breakpoint or an exception.
- **Step Over (F10)**, which executes one statement at a time except for method calls, which are invoked without stepping into them.
- **Step Into (F11)**, which executes one statement at a time, including statements within method bodies.

- **Step Out (F12)**, which executes the remaining lines of a function starting from the current breakpoint.
- **Restart (Ctrl+Shift+F5)**, which restarts the application execution.
- **Stop (Shift+F5)**, which stops debugging.

Figure 80: The Debug Actions Pane

If you hover over a variable name in the code editor, a convenient pop-up makes it easy to investigate values and property values depending on the type of the variable. Figure 81 shows a pop-up showing information about the **app** variable. You can expand properties and see their values, and you can also investigate properties in the Variable area of the Debug side bar.

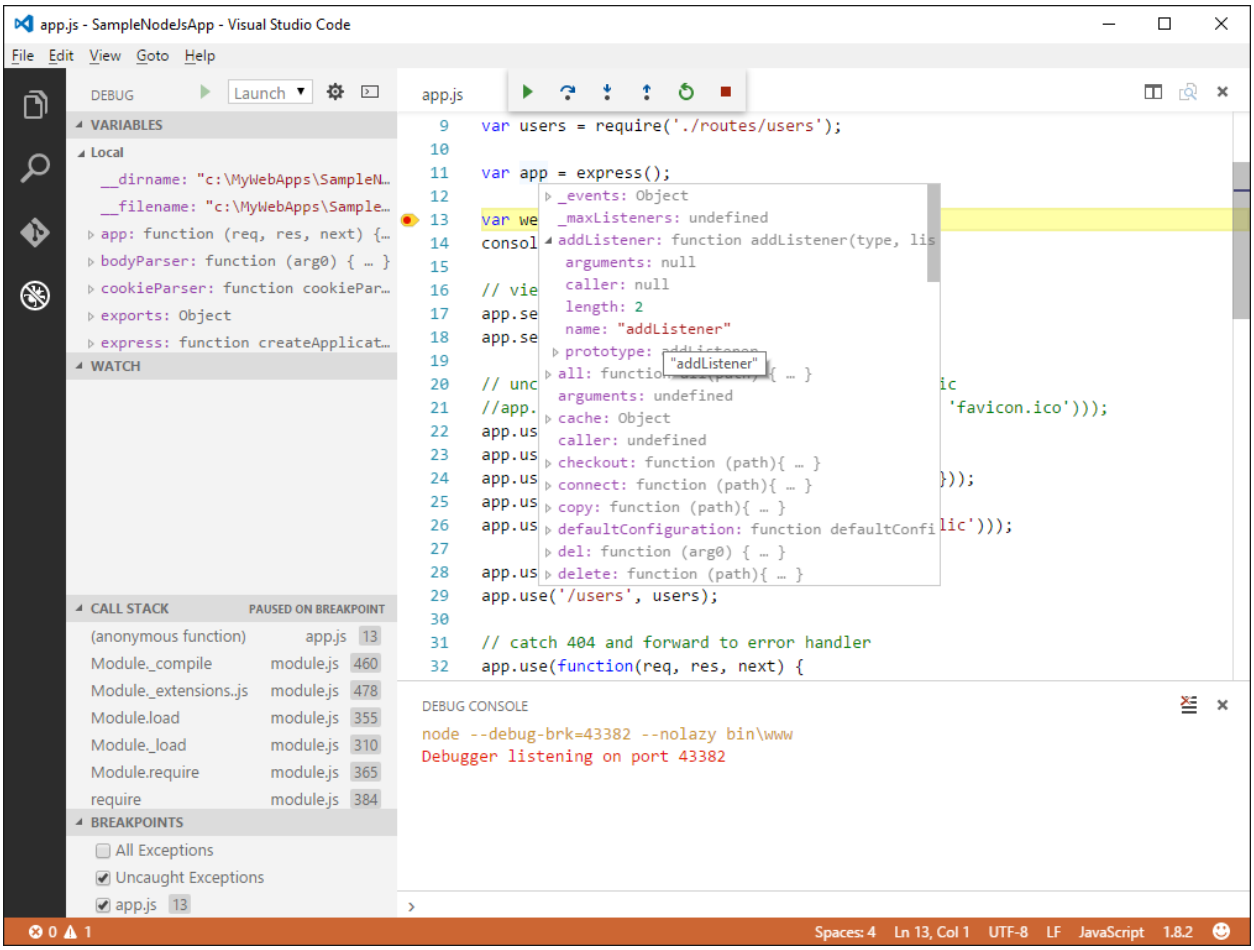


Figure 81: Analyzing and Investigating Property Values at Debugging Time

As another example, simply hover the **`welcomeText`** variable declared manually to see how a tooltip shows its current value (see Figure 82). In this case the value has been hardcoded, but if it was supplied programmatically, then this feature would make it really easy to understand if the actual value was the expected one.

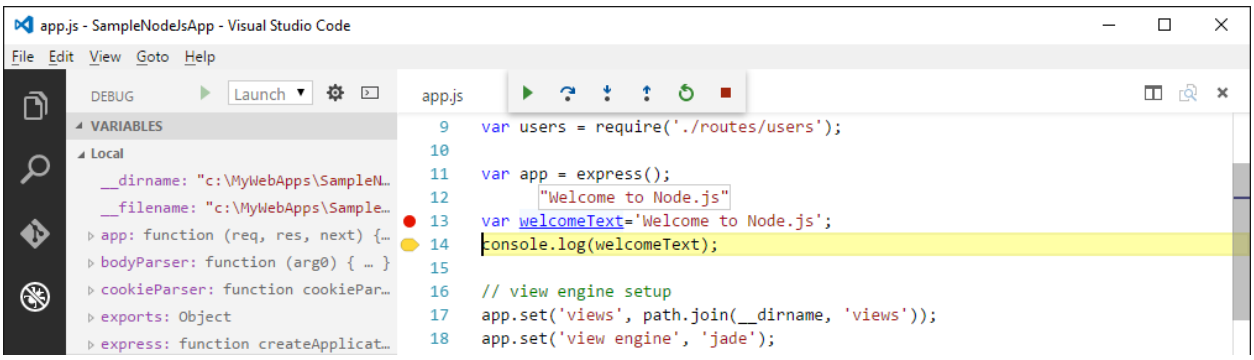


Figure 82: Analyzing a Local Variable

The debugger also offers the **Call Stack** feature, which allows stepping through the method calls stack. When you click a method call in the stack, the code editor will open the file containing the call, highlighting it as shown in Figure 83.

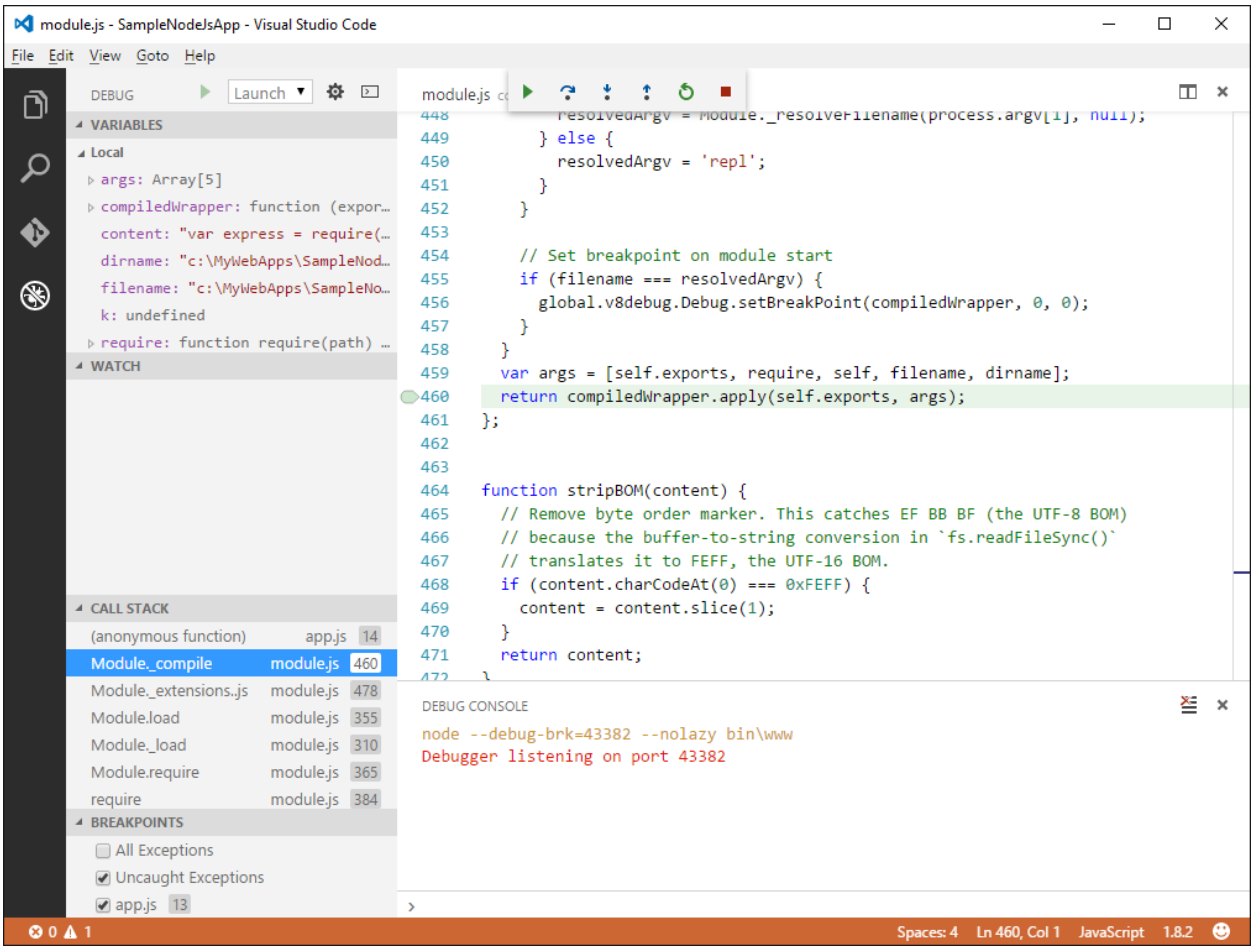


Figure 83: Analyzing Method Calls with Call Stack

Finally, you have an option to use the **Watch** tool to evaluate expressions. While debugging, right-click an object and then select **Debug: Add to Watch** (see Figure 84).

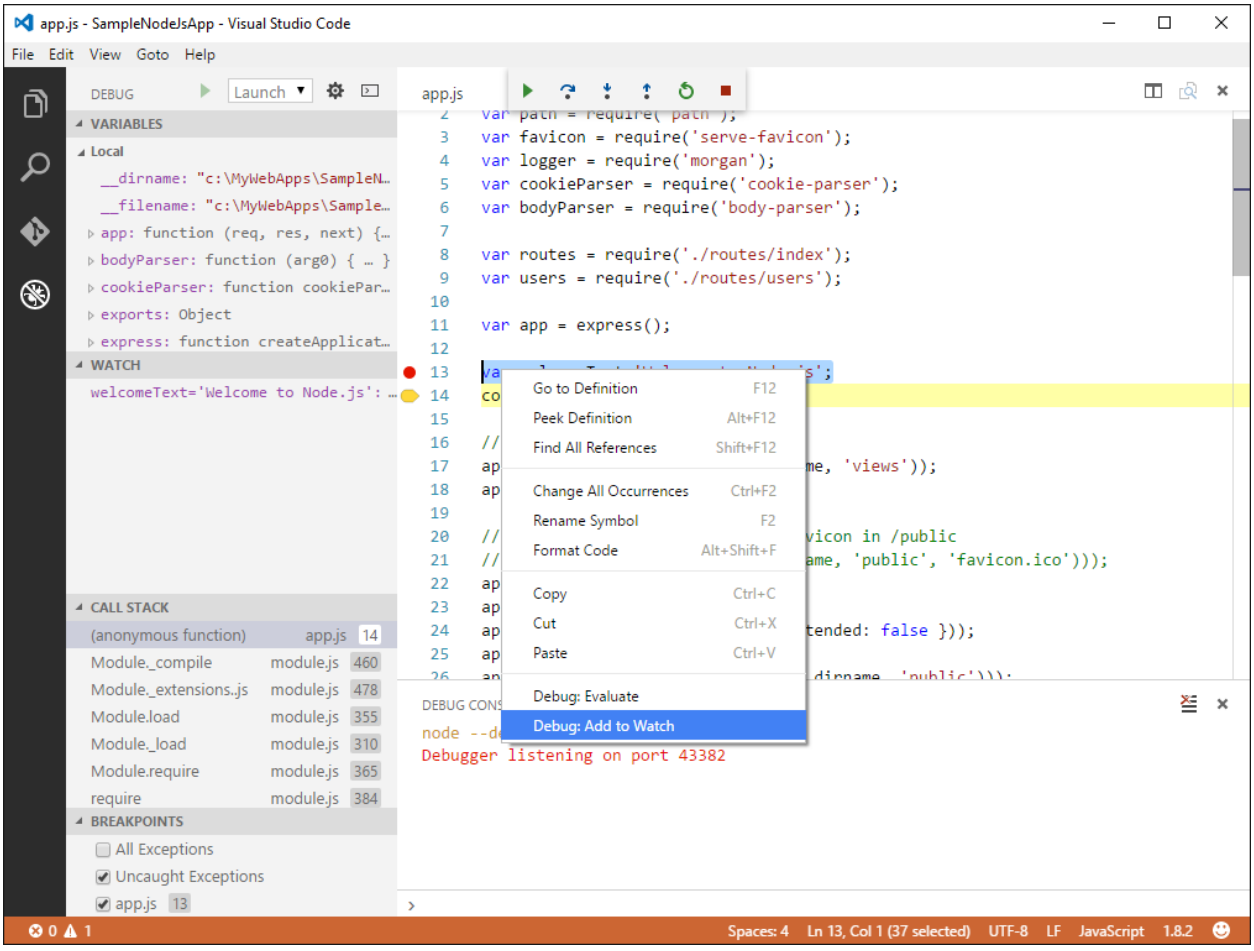


Figure 84: Using the Watch Debugging Tool

Another useful tool is the **Debug console**. You can open this by clicking the **Open Debug console** button to the right of the gear icon in the Debug view. When you open this tool, you can write expressions to be evaluated immediately. For instance, Figure 85 shows how to evaluate the **`welcomeText`** variable by typing its name. Code will show the variable's current value.

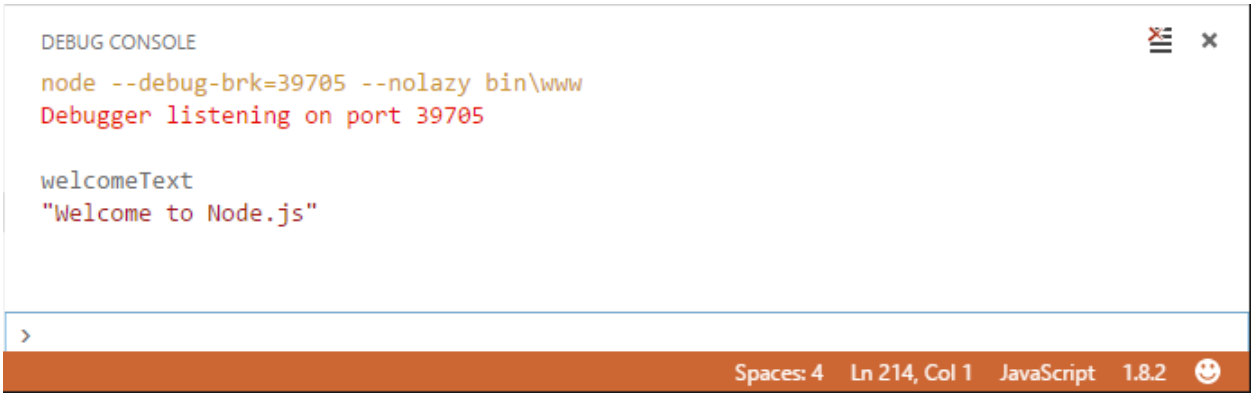


Figure 85: The Debug Console

It is worth mentioning that the Debug console also allows sending commands to the debugger, which is done by attaching the debugger to an already running application.

Hints about debugging Mono-based applications

Visual Studio Code has experimental support for debugging Mono-based C# and F# applications. At the time of this writing, this is only available for Linux and OS X, not Windows. Mono 3.1.2 or higher must be installed to use it. Also, you need to download the Mono debugger for Visual Studio Code from the Command Palette by typing the **`install extension`** command, and then selecting the **Mono Debug** extension. Once you have installed these prerequisites, you must invoke the compiler with the following line to enable an application for debugging:

```
> mcs -debug Program.cs
```

Or you can type the following command to debug an already running application:

```
> mono --debug --debugger-
agent=transport=dt_socket,server=y,address=127.0.0.1:55555 Program.exe
```

Additional information about Mono debugging as well as news about feature updates can be found in the official documentation.

Chapter summary

The power of Visual Studio Code as a development environment is apparent when it comes to writing real applications. With the help of specific generators, you can easily write ASP.NET 5 web applications using C# and the .NET Execution Environment (DNX). This chapter also described how easy it is to implement Entity Framework data access with a few commands. If you develop Node.js applications instead, you can also 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 experienced the most important and powerful features you need to write great cross-platform applications. Now it is time to discuss how to customize Visual Studio Code.

Chapter 5 Customizing and Extending Visual Studio Code

Being the great environment it is, Visual Studio Code can also be customized and extended in many ways. In fact, you can customize its appearance, the code editor, and keyboard shortcuts to make your editing experience extremely personalized. Also, you can download and install a number of extensions such as new languages with syntax colorization, debuggers, themes, linters, and code snippets. You can even create and share your own extensions and let other developers know how cool you are. This chapter explains how to customize and extend Visual Studio Code, including extensibility examples.

Note: In this chapter you learn how to customize and extend Visual Studio Code, and also the basics of creating and publishing your own extensions. However, explaining the core concepts of the extensibility model and describing every extensibility scenario is not possible here. For this reason, refer to the Overview document if you are interested in additional extensibility scenarios and the Approach to Extensibility document about the extensibility model.

Introducing the Visual Studio Code Marketplace

Visual Studio Code allows extending the environment with custom extensions. The developer community shares extensions to the **Visual Studio Code Marketplace**, also referred to as the **Extension Gallery**. To download extensions and customizations, open the Command Palette and type the following command:

```
> ext install
```

When you type this command, the list of extensions in the Gallery will be shown, and you will be able to see the extension types and filter the list as you type. This is the where you download new languages, code snippets, color themes, debuggers, complete extension bundles, and more. You can also browse the list of installed extensions by typing **`ext`**. From this list, you can also uninstall extensions with just a click. This is all you need to know for now about the Extension Gallery and related commands. I will go into more detail as I explain how to customize and extend Visual Studio Code.

Understanding customizations and extensions

Visual Studio Code can be personalized via customizations and extensions. The difference between them is that extensions add functionalities to a tool, change the behavior of existing functionalities, or provide new instrumentation. Supplying IntelliSense for a language that does not have it by default, adding commands to the status bar, and custom debuggers are examples of extensions. On the other hand, customizations are related to environment settings and do not add functionalities to a tool. Table 2 describes VS Code's extensibility points and summarizes customizations and extensions in VS Code.

Table 2: Visual Studio Code Extensibility

Feature	Description	Type
Color themes	Style the environment layout with different colors.	Customization
User and workspace settings	Specify your preferences about the editing experience.	Customization
Key bindings	Redefine key shortcuts in a way you feel more comfortable with.	Customization
Language grammar and syntax colorizers	Add support for additional languages with syntax colorizers and bring VS Code's evolved editing experience to your favorite languages.	Customization
Code snippets	Add TextMate and Sublime Text snippets and write repetitive code faster.	Customization
Debuggers	Add new debuggers for specific languages and platforms.	Extension
Language servers	Implement your validation logic for files opened in VS Code.	Extension
Activation	Load an extension when a specific file type is detected or when a command is selected in the Command Palette.	Extension

Feature	Description	Type
Editor	Work against the code editor's content, including text manipulation and selection.	Extension
Workspace	Enhance the status bar, working file list, and other tools.	Extension
Eventing	Interact with Code's life-cycle events such as open and close.	Extension
Evolved editing	Improve language support with IntelliSense, Peek, Go To Definition, and all the advanced, supported editing capabilities.	Extension

Except for color themes, key bindings, and user settings that you can customize via VS Code's specific options, customizations and extensions share the way they can be created. In fact, you typically create both (or at least you get started creating them) with Yeoman, the command-line generator you already met in Chapter 4. For this reason, after showing you how to install existing extensions, in this chapter I will describe how to create custom basic language support plus code snippets using Yeoman. I will also put you in the right direction to get started with extensions, but I will not cover the process of producing other types of extensions, which is something more complex and out of the scope of a book in the *Succinctly* series. What you really need to do before reading this chapter is install the Yeoman generator for VS Code, which can be done by typing the following in the command prompt:

```
> npm install –g yo code
```

Customizing Visual Studio Code

This section walks through supported customizations previously summarized in Table 2.

Choosing a theme

Visual Studio Code ships with a number of color themes you can pick to give the environment a different look and feel. You select a color theme by clicking **File** > **Preferences** > **Color Theme**. The list of available color themes will be shown in the Command Palette, as you can see in Figure 86.

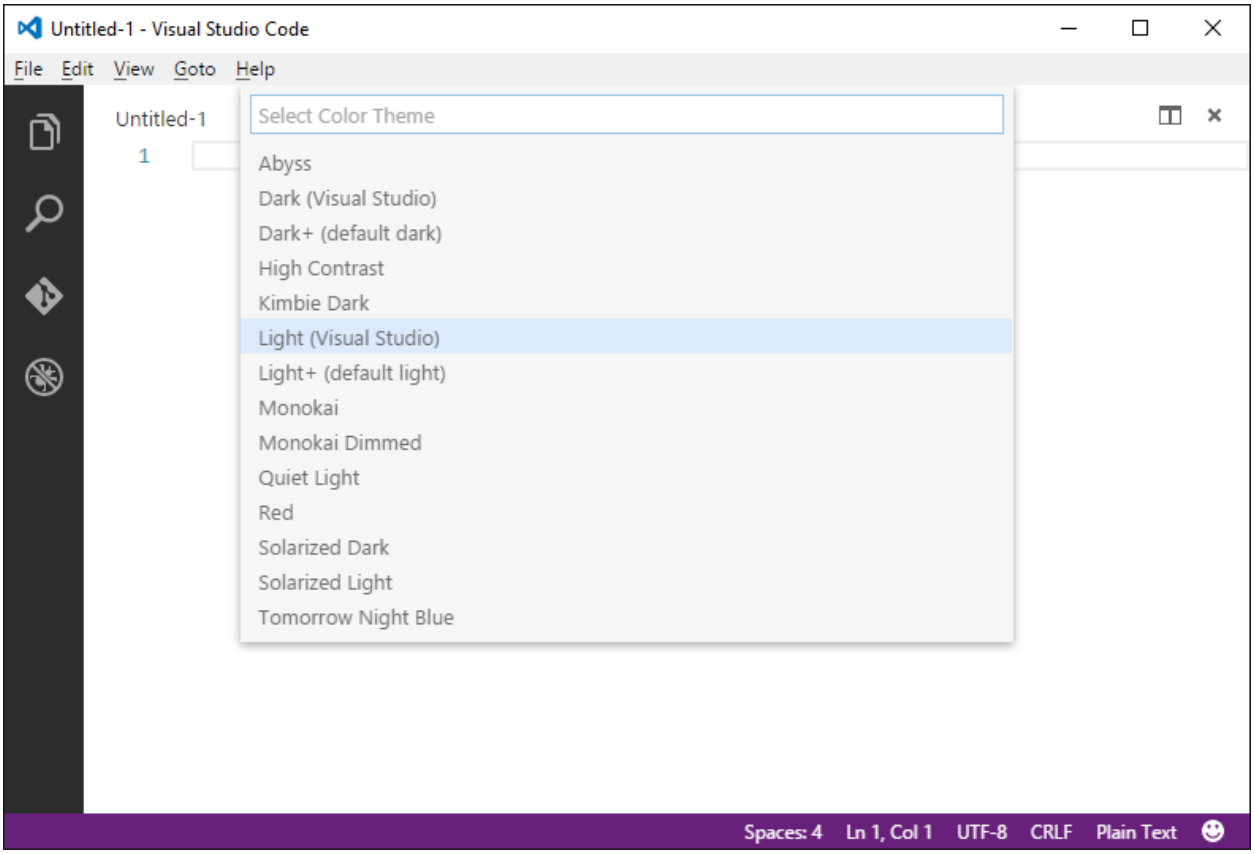


Figure 86: Selecting a Color Theme

Once you select a different color theme, it will be applied immediately. Also, you can get a preview of the theme as you scroll through the list with the keyboard. For instance, Figure 87 shows the Dark (Visual Studio) theme, which is a very popular choice that is similar to the Dark + default theme.

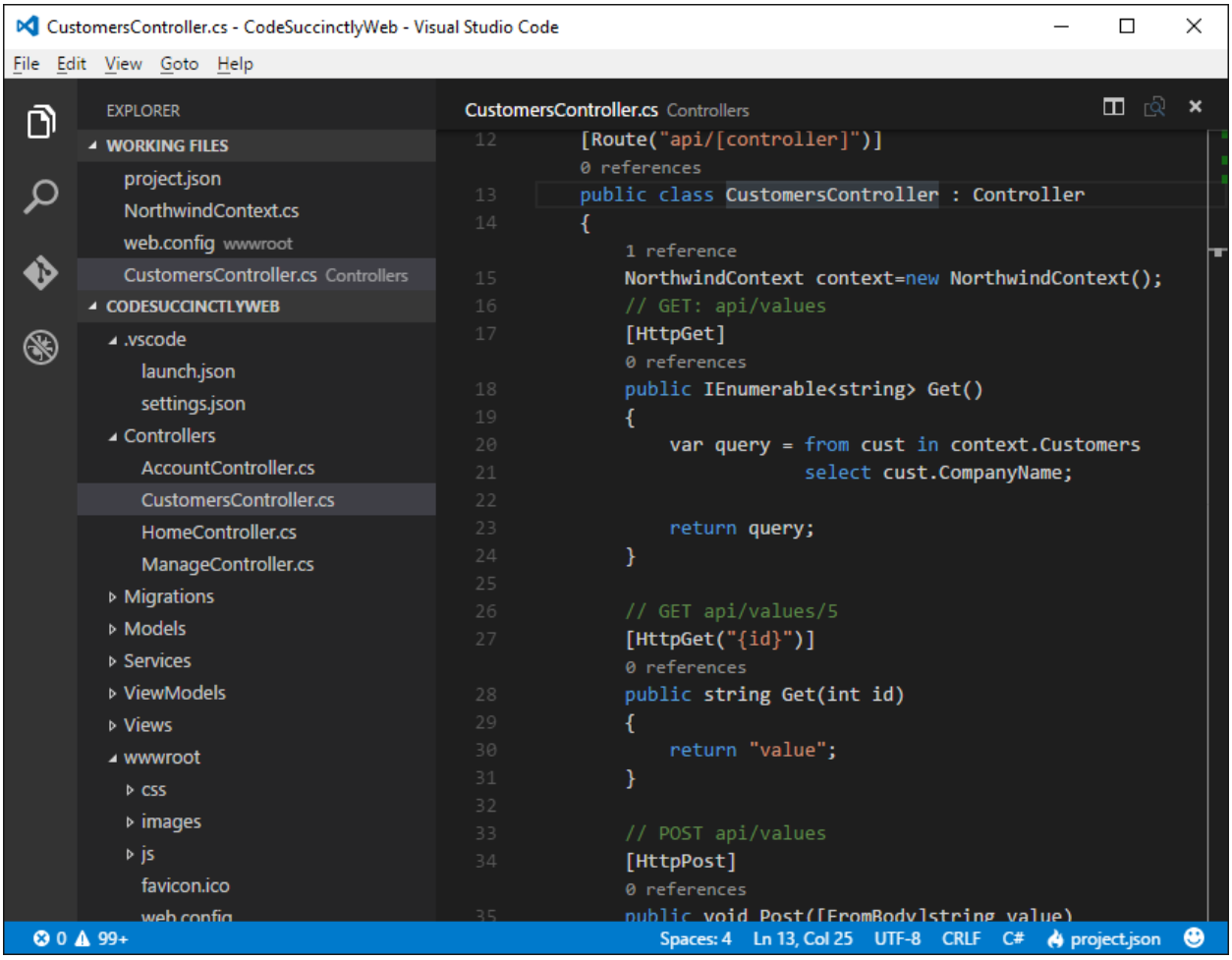


Figure 87: The Dark Theme

Trying the other available color themes is left to your curiosity.

Installing themes from the Extension Gallery

You can download and install additional themes produced by the developer community from the Extension Gallery. To do this, open the Command Palette by pressing **F1** and type **`install extension`**, then type **`theme`** to filter the list. At this point, you will see a list of available themes from the Gallery, as shown in Figure 88.

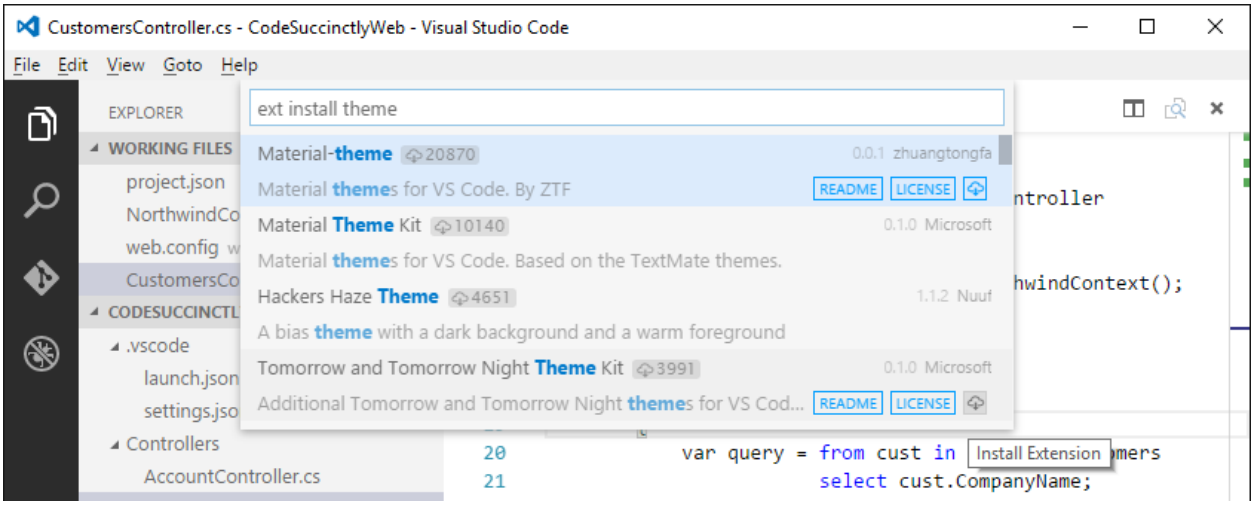


Figure 88: Selecting Themes from the Extension Gallery

If supplied by the theme producer, you can get information about a theme via the **Readme** and **License** files. Finally, click the **Install Extension** button, the last on the right, to install the selected theme. Notice that a VS Code restart will be required, and then you will be able to apply the new theme. You can also create custom themes using the Yeoman command-line tool that I will describe shortly.

Customizing the environment settings

In most applications, including other IDEs, you set environment settings and preferences via specific tools or windows. In Visual Studio Code, settings are instead specified writing JSON markup. There are two types of settings: user settings and workspace settings. User settings apply globally to the development environment, whereas workspace settings only apply to the current project or folder. Also, you can customize keyboard shortcuts in JSON format as well.

Customizing user settings

User settings globally apply to VS Code's development environment. Customizing user settings is accomplished by selecting **File** > **Preferences** > **User Settings**. This action splits the code editor into two views. In the left view, you can see the default JSON settings file; in the right view, you can see a new, empty JSON file called **settings.json** where you can override the default settings.

Tip: The settings.json file's location depends on your operating system and is visible at the top of the code editor.

Default settings relate to the code editor, file management, HTTP proxies, Git engine, linters, Markdown preview, and installed extensions' behavior (if any). The default settings view provides detailed comments for each available setting expressed with JSON format so that you can easily understand what setting a particular line applies to. For more details about available settings, visit the official documentation. You can easily provide custom settings by overriding one or more default settings, writing inside settings.json. Figure 89 shows an example where you can see how to change the default editor font, how to remove line numbers, and how to disable file auto-save. Also, you can see how IntelliSense will help you choose among available settings. It is worth mentioning that using IntelliSense to pick a setting will also add a default value that you will want to replace with a custom one.

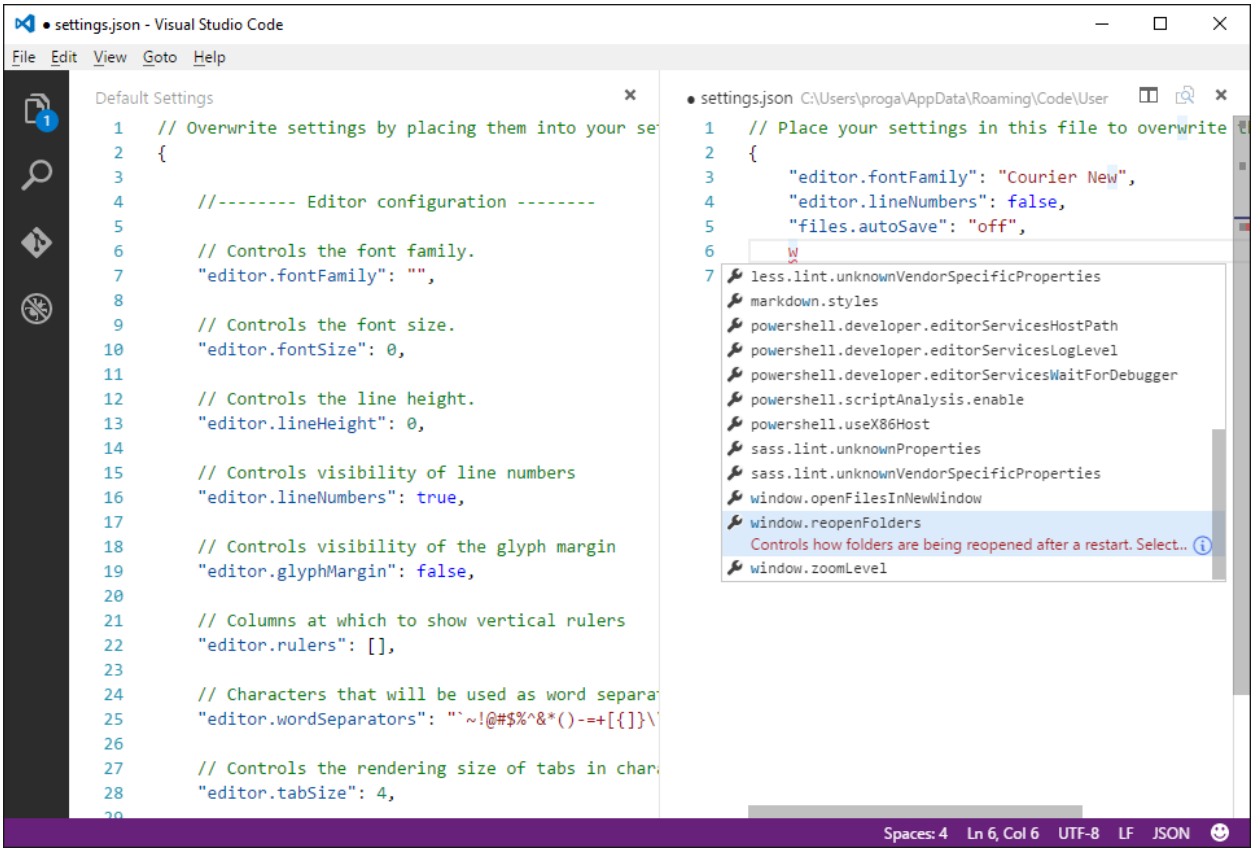


Figure 89: Supplying Custom User Settings

IntelliSense also shows hints about settings with a convenient tooltip under the selected setting. You can also expand the tooltip by clicking the information icon. When done, do not forget to save settings.json otherwise your changes will be lost.

Customizing workspace settings

Differently from user settings which globally apply to VS Code's environment, workspace settings only apply to the current folder. In order to customize workspace settings, you first need to open an existing folder. Next, select **File** > **Preferences** > **Workspace settings**. What you see and what you can do at this point are exactly the same as the user settings: You have a settings.json file where you can specify your preferences. The difference is that settings.json is saved in the .vscode subfolder, restricting the settings availability to the current folder only.

Customizing key bindings

In VS Code terminology, key bindings represent shortcuts you use to invoke commands and actions from the keyboard instead of using the mouse. Visual Studio Code includes a huge number of keyboard shortcuts (summarized in the official documentation) that you can override with custom values. Key bindings are represented with JSON markup and each is made of two elements: **`key`**, which stores one or more keys to be associated to an action, and **`command`**, which represents the action to invoke. In some cases, VS Code might offer the same shortcuts for different scenarios. This is the typical case of the Esc key, which targets a number of actions depending on what you are working with, such as the code editor or a tool window. In order to identify the proper action, key binding settings support the **`when`** element, which specifies the proper action based on the context. Customizing key bindings is very easy: All you need to do is select **File** > **Preferences** > **Keyboard Shortcuts**, and edit the **keybindings.json** file that Code generates for you. The code editor gets split into two views. In the left view you can see the full list of default key bindings, and in the right view you can override default shortcuts with custom ones. Remember that Visual Studio Code has different default key bindings and allows for customizing them depending on what operating systems it is running on.

Tip: Key bindings are only available globally, so you cannot supply custom keyboard shortcuts for a specific folder as you can with workspace settings. Also, keyboard shortcut defaults target the US keyboard layout. Luckily, Visual Studio Code highlights shortcuts that have different key bindings for your current keyboard layout, suggesting the keys you should use.

A cool feature in VS Code allows you to quickly add a custom key binding by pressing **Ctrl+K** twice. When you do this, a pop-up appears and asks you to specify the key binding, as shown in Figure 90.

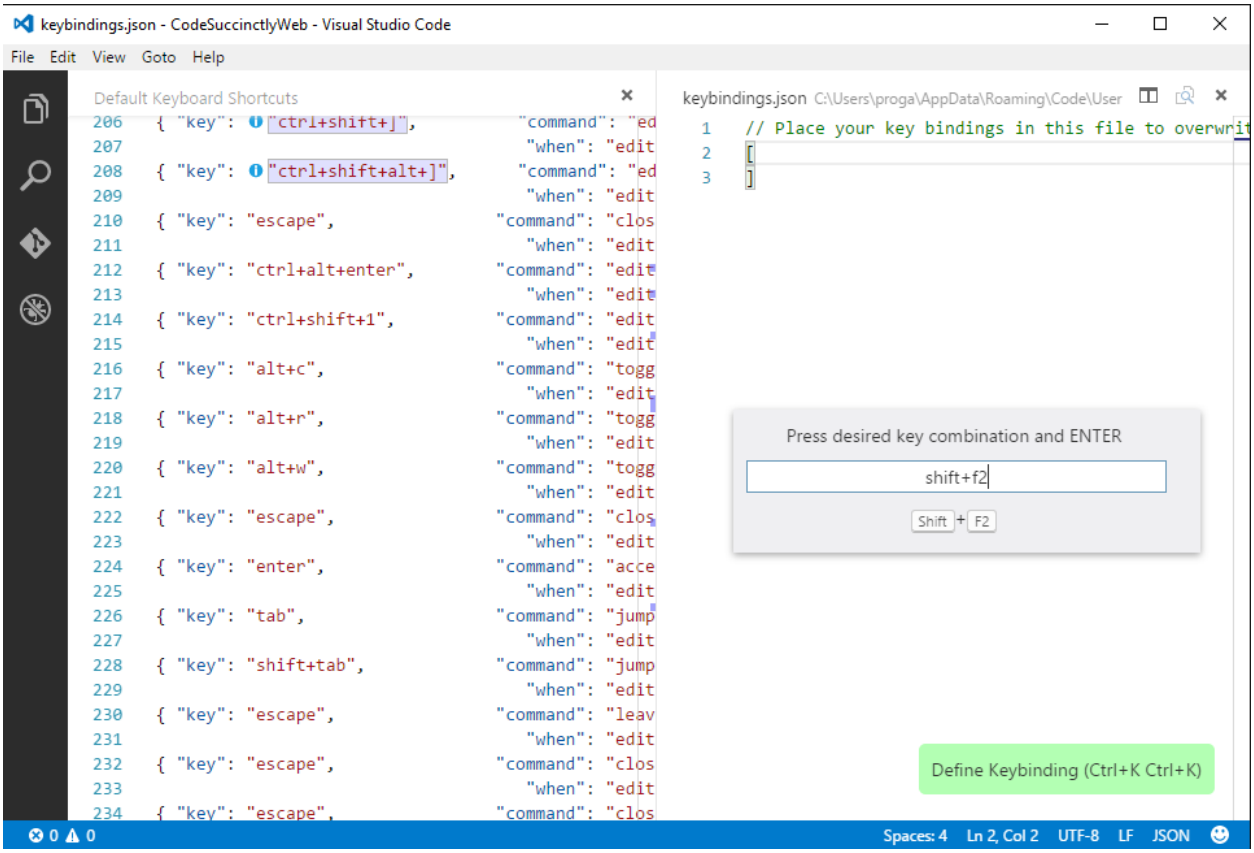


Figure 90: Entering a New Key Binding

Simply press the keys you want to bind to a new shortcut, and then press **Enter**. At this point, Visual Studio Code generates the proper JSON markup (see Figure 91), leaving you to specify the target command and optionally a **`when`** element to specify the action context.

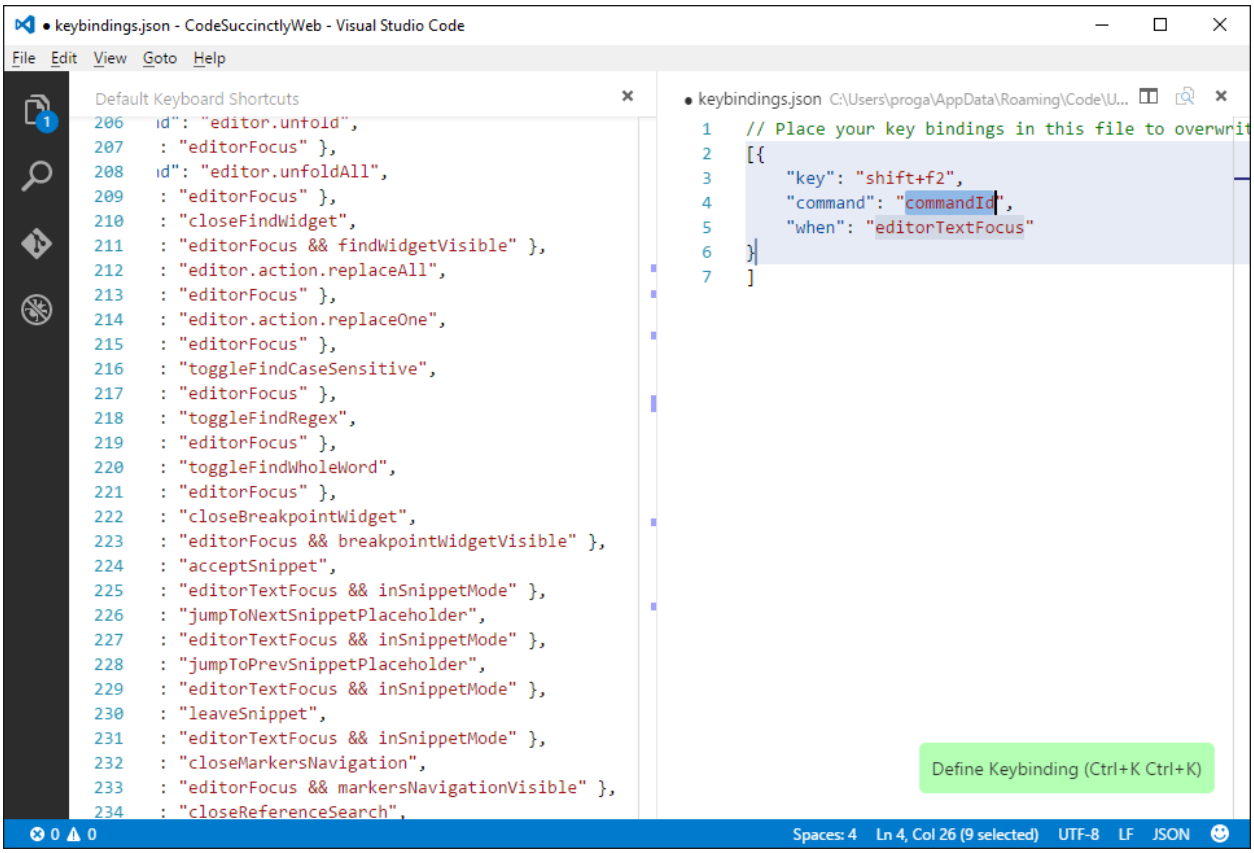


Figure 91: Specifying the New Key Binding's Settings

When you are done with your custom key bindings, do not forget to save **keybindings.json**.

Languages, syntax colorizers, and code snippets

Visual Studio Code can be customized with additional languages, syntax colorizers, and code snippets. You can get additional languages and snippets from the Extension Gallery. More often than not, you will also find bundles containing the language support and related code snippets, and possibly additional goodies such as tasks and debuggers. Installing new languages and code snippets is very easy. In the Command Palette, type **`ext install`** and you will be able to pick from the extension list the desired language and snippets, or bundles. For instance, Figure 92 shows a bundle called OmniPascal, which supports the Delphi programming language and includes code snippets, IntelliSense support, and Go To Definition support.

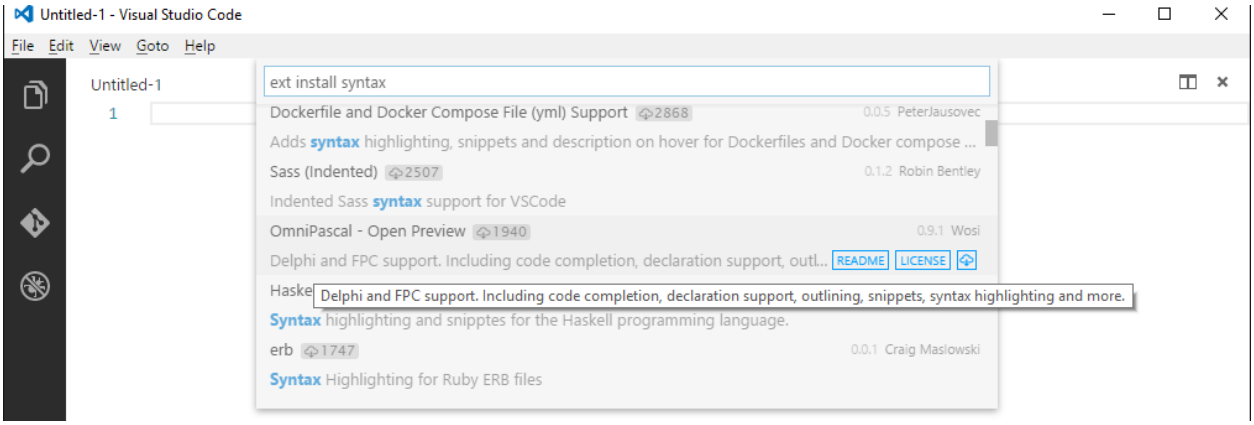


Figure 92: Choosing New Languages, Code Snippets, or Bundles

When you select an item, VS Code starts to install it. When finished, you will be prompted to restart the application. At this point, you will be able to use the new language and snippets as shown in Figure 93, which shows an example based on Delphi and the newly installed OmniPascal extension.

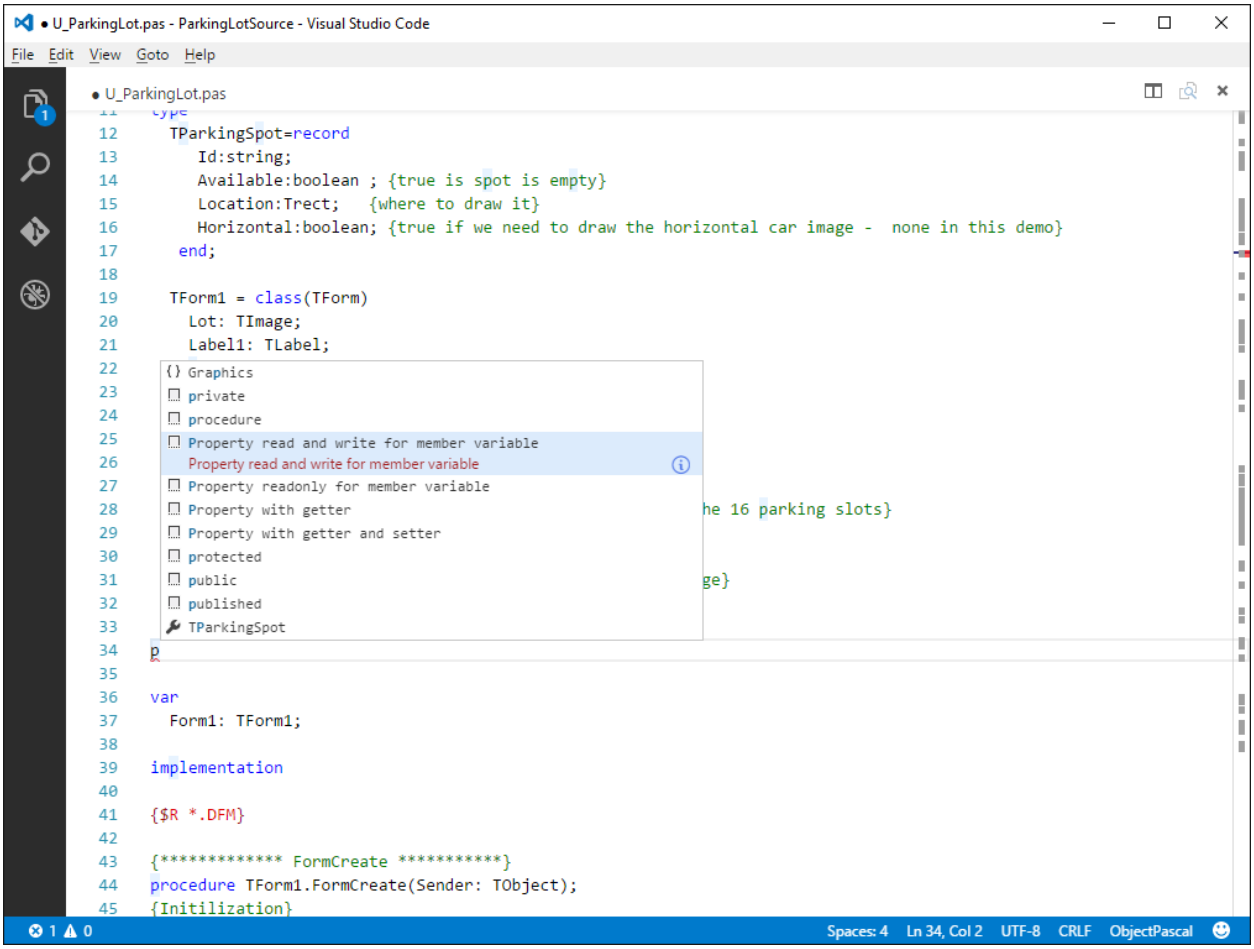


Figure 93: Writing Delphi Code with Syntax Colorization, Snippets, and IntelliSense Support

There is already rich support for important languages in the Extension Gallery, such as Python, Cobol, Fortran, and even PowerShell scripts. Among the others, you might want to check out the Cordova Tools for Visual Studio Code, which introduces a new debugger, commands, and IntelliSense for Cordova projects; the Python extension, which extends Python language support with IntelliSense, code snippets, a debugger, and enables features such as Go to Definition, Find all References, and Go to Symbol; and Visual Studio Team Services, which provides integration between VS Code and Visual Studio Team Services, introducing support for team projects, work items, and code management. These are really great examples of VS Code extensibility.

Create your first extension: Language with grammar, colorization, and snippets

As you know, Visual Studio Code supports a huge number of languages out of the box, plus you can install additional languages from the Visual Studio Code Marketplace. Not limited to this, you can add new languages to Visual Studio Code in a few steps. What you will do in the next example is add language support for the Ada programming language, including code snippets. Of course, all the steps described in this section apply to any other language.

Defining the language syntax

Visual Studio Code supports language syntaxes based on the TextMate .tmLanguage format. You can certainly define your own syntax, but the quickest way is starting with an existing one if the license agreement permits using the file. The first place you might want to search for an existing .tmLanguage syntax is GitHub. For instance, a well-done implementation of the Ada language syntax is available at https://raw.githubusercontent.com/mulander/ada.tmbundle/master/Syntaxes/Ada.tmLanguage.

Tip: Visual Studio Code needs the raw text of a language syntax definition. In the case of GitHub as the source, you need to open the .tmLanguage file in the raw view, which is accomplished by clicking the** Raw **button in the file view.

Once you have the syntax, you are ready to generate the language support for VS Code.

Generating new language support

Launch a command prompt and type **`yo code`**. This will launch the Yeoman Visual Studio Code Extension generator, which asks you what kind of extension you want to generate (see Figure 94).

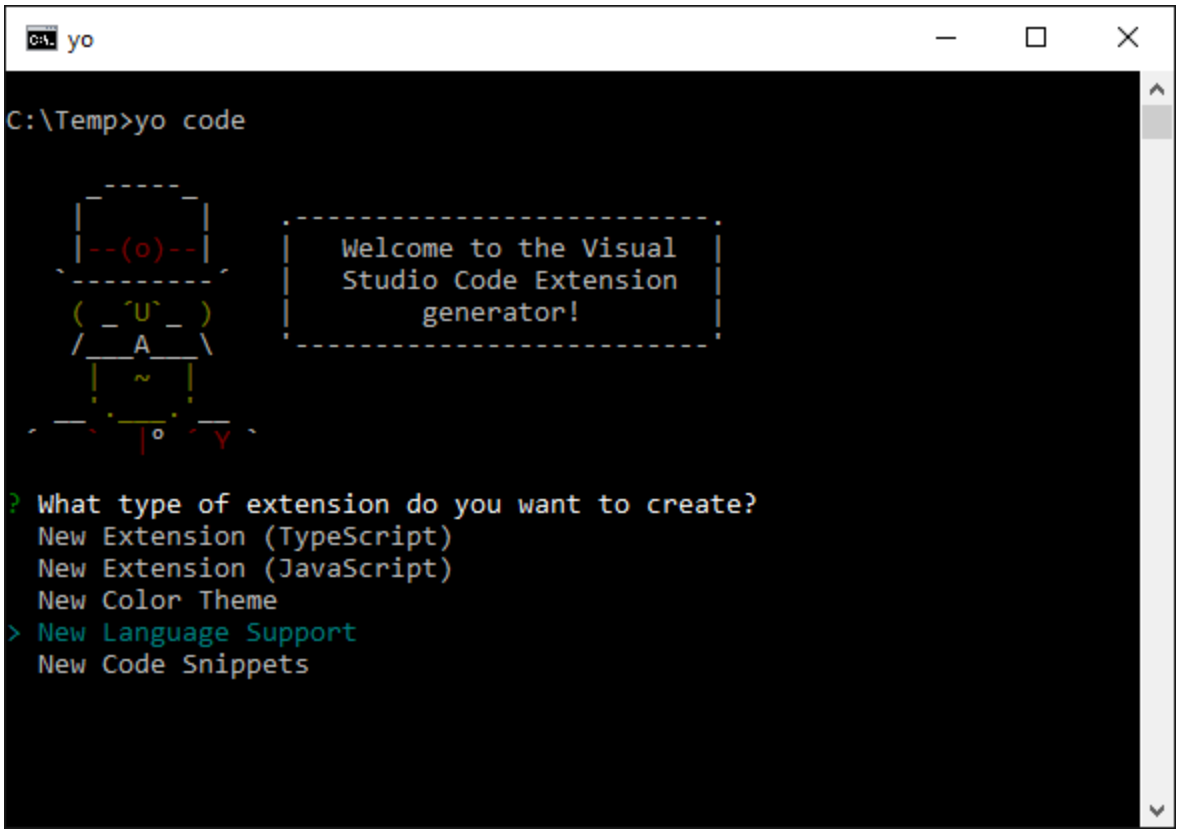


Figure 94: Yo Code's First Screen

As you can see, with this tool you can create extensions, color themes, new languages, and code snippets. Select **New Language Support** and press **Enter**. Next the tool will ask for several pieces of information in sequence. So, provide the following information (see Figure 95 for reference):

1. Enter the URL or path for the .tmLanguage syntax file, in this case https://raw.githubusercontent.com/mulander/ada.tmbundle/master/Syntaxes/Ada.tmLanguage.
2. Enter **ADA** as the name of the extension.
3. Accept the **ada** automatic suggestion as the extension identifier.
4. Provide an extension description. In this case I'm entering **ADA language and code snippets**.
5. Enter your publisher name. If you plan to publish extensions to the online Visual Studio Marketplace, keep note of the publisher name you are entering as you will need to register online (this will be described shortly).
6. Accept **ada** as the **languageId** option. Notice that this must always be lowercase.
7. Accept **Ada** as the detected name, which is what you will see when you select a language in the code editor's selector.
8. Accept the proposed file extensions that VS Code will associate to the current language—.adb and .ads in this case. Of course you can add or edit file extensions.

Your new language will be created in a few seconds. Figure 95 shows the full steps and the summary at the end of the operation.

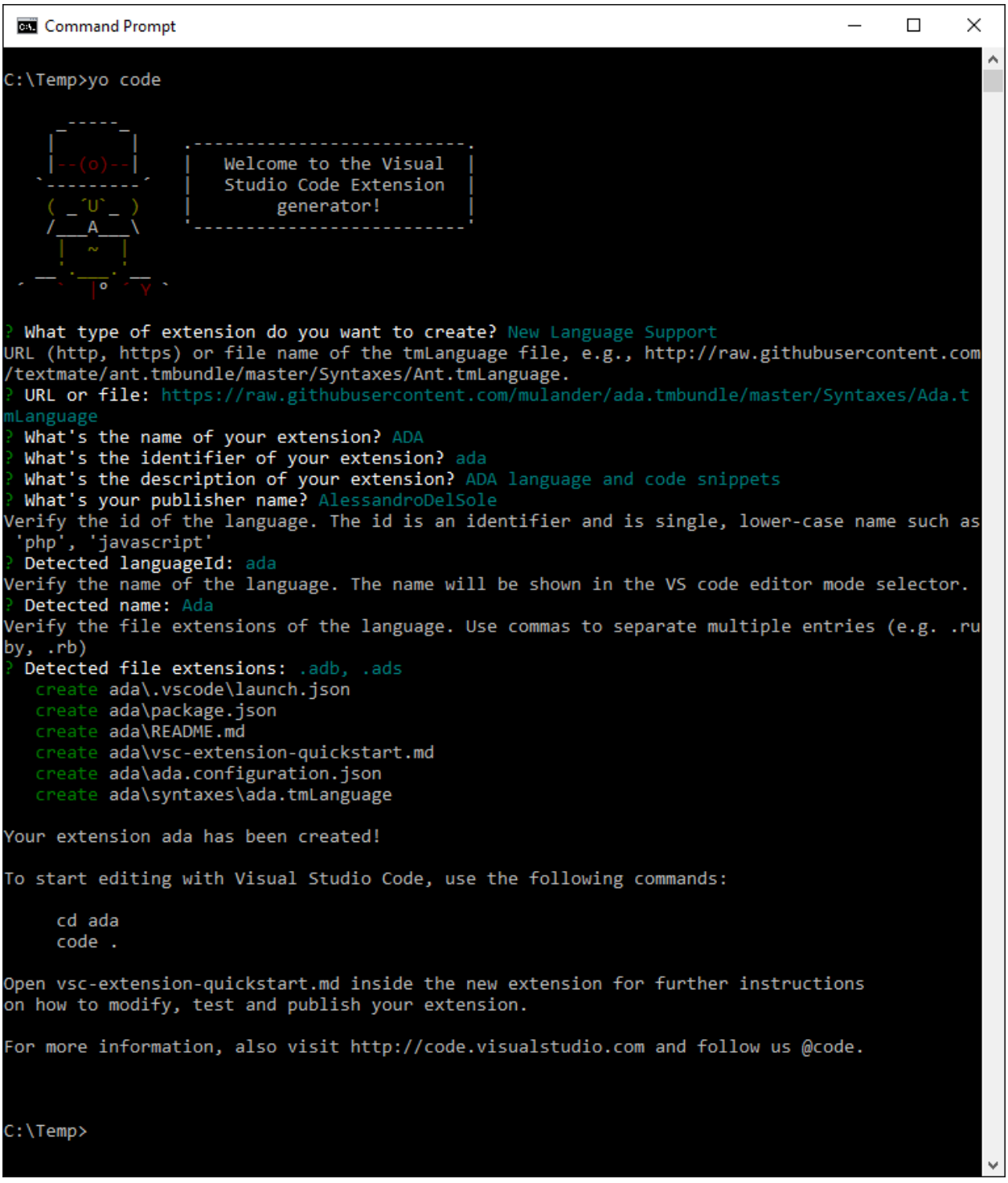


Figure 95: Generating the New Language with Yeoman

The new language will be created in a subfolder of the folder where you invoked the command prompt, which in my example is C:\Temp\Ada. Now, open the newly-created Ada folder with Visual Studio Code. As you can see in Figure 96, the Explorer shows the list of files that the new extension is made of. Among them, a README.md file is available and you should edit this file into a more personalized document. Figure 96 shows an example.

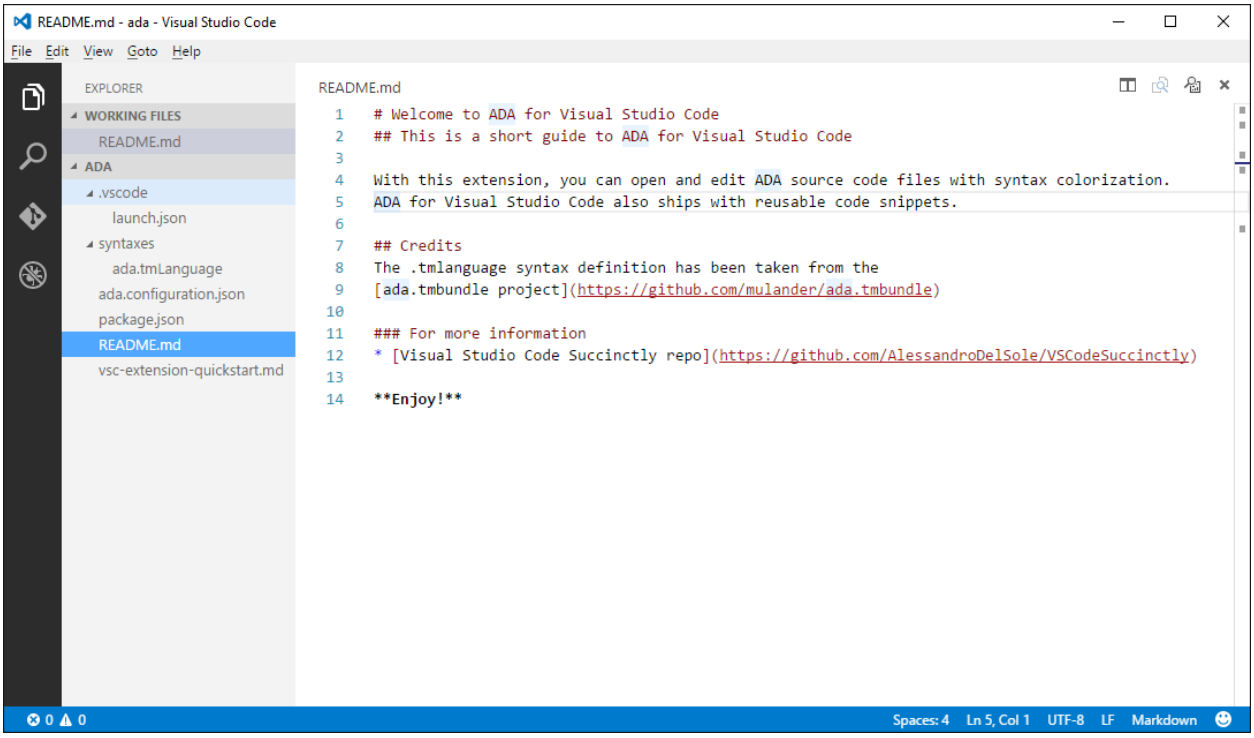


Figure 96: Editing README.md

Tip: When you use a .tmLanguage syntax file produced by other developers, though the license agreement typically allows using it with no restrictions, it is a good idea to give credit to the original work, as I'm doing in Figure 96.

Now open the **package.json** file, whose content is shown in Code Listing 10. Generally speaking, extensions and customizations have a manifest file called package.json that contains all the necessary information for Visual Studio Code to install and run the extension properly. As you can see, properties are self-explanatory and are the JSON counterpart of the information you entered when using Yeoman. It is worth mentioning the importance of **`categories`**, which are used to categorize an extension in the Visual Studio Marketplace. You can combine multiple categories, as you will see with snippets shortly. The **`contributes`** field stores the list of features the extension brings into Visual Studio Code--in this case, the language and its grammar. This is also the place to provide additional contributes, as you will do with code snippets in a moment.

Code Listing 10

```
{
    "name": "ada",
    "displayName": "ADA",
    "description": "ADA language and code snippets",
    "version": "0.0.1",
    "publisher": "AlessandroDelSole",
    "engines": {
        "vscode": "^0.10.10"
```

```
    },
    "categories": [
        "Languages"
    ],
    "contributes": {
        "languages": [{
            "id": "ada",
            "aliases": ["Ada", "ada"],
            "extensions": [".adb",".ads"],
            "configuration": "./ada.configuration.json"
        }],
        "grammars": [{
            "language": "ada",
            "scopeName": "source.ada",
            "path": "./syntaxes/ada.tmLanguage"
        }]
    }
}
```

Testing the extension

Now close VS Code and copy the **Ada** subfolder into **%USERPROFILE%\.vscode\extensions** (where USERPROFILE is the path for your user profile on disk, such as C:\Users\YourName). On Mac and Linux, this folder is located at **%HOME/.vscode/extensions**. This is the folder where VS Code's extensions are installed. If you now open Visual Studio Code, write some Ada code and select Ada as the current file language, you will see the proper syntax colorization, as shown in Figure 97.

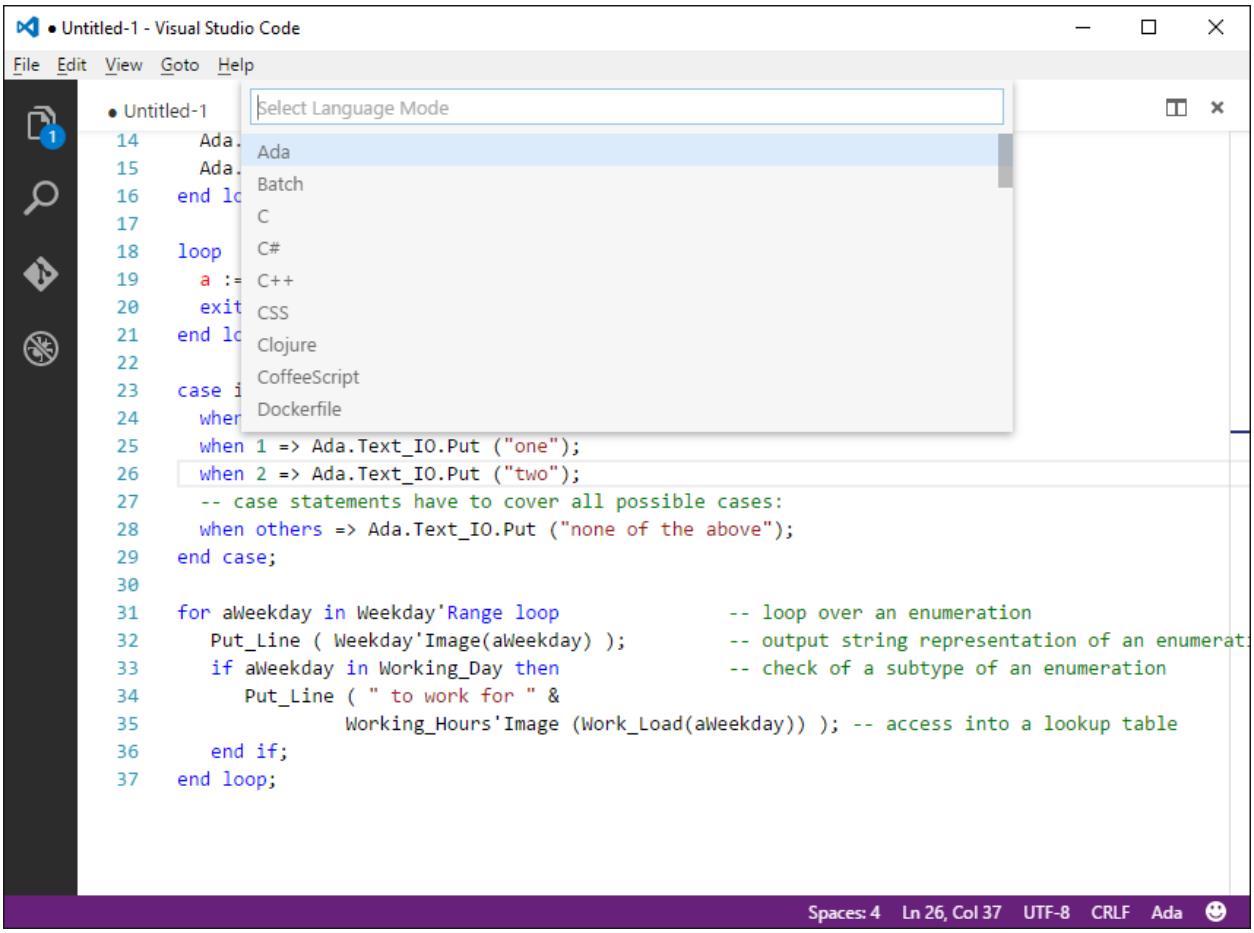


Figure 97: The custom language is properly recognized.

Also notice that VS Code will suggest the proper file extension when you first save a new file. Now you have a new language, but it would be a good idea to bundle some code snippets and make some customizations to the extension manifest.

Creating and packaging reusable code snippets

Visual Studio Code supports JSON as the format for code snippets. If you do not already have any written with JSON, you can generate code snippets using Yeoman starting from TextMate or Sublime Text code snippets. For instance, consider the following Ada code snippet that performs a simple **`for..loop`** iteration and prints the value of the **`i`** variable:

```
for i in 1 .. 10 loop
  Ada.Text_IO.Put (i);
end loop;
```

With either TextMate or Sublime Text, create a new snippet containing the previous code. Figure 98 shows how the code snippet looks in Sublime Text.

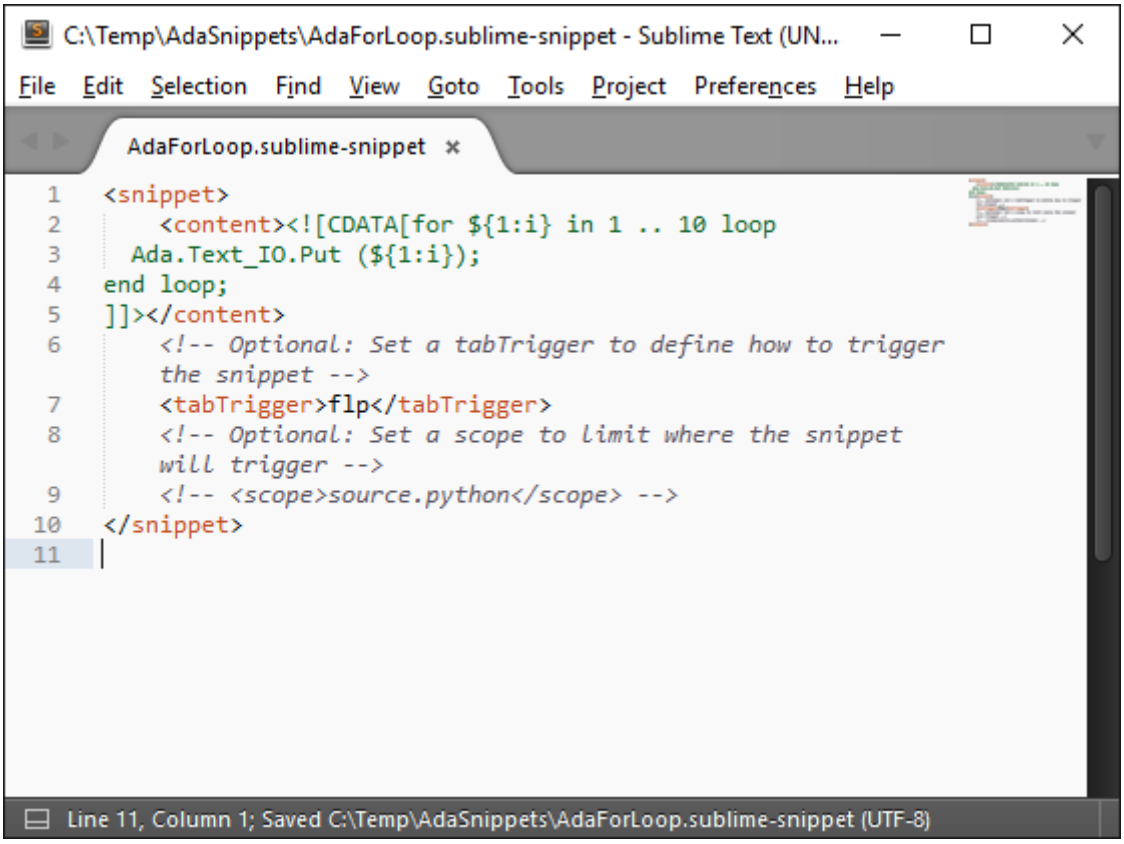


Figure 98: Preparing a Code Snippet with Sublime Text

TextMate and Sublime Text snippets are XML files. Basically you must enclose the code inside the **`CDATA`** section of the **`content`** node. Notice how you can use placeholders between brackets to define default values for a field. In this case the **`{1:i}`** placeholder means that the code editor will highlight the **`i`** variable so that the user understands it can be replaced with a different identifier. The number **`1`** will match any occurrence of the specified variable, but you could use **`2`**, **`3`**, and so on to suggest other default value replacements. You should also provide a keyboard shortcut that allows activating the code snippet in VS Code by pressing Tab, which is accomplished by providing the shortcut in the **`tabTrigger`** node. Let's leave off the **`scope`** node, which is not necessary at this point. Save the file as **AdaForLoop**. Now consider the following Ada code, which performs a simple **`loop..end loop`** loop:

```
loop
  a := a + 1;
  exit when a = 10;
end loop;
```

Figure 99 shows this snippet in Sublime Text.

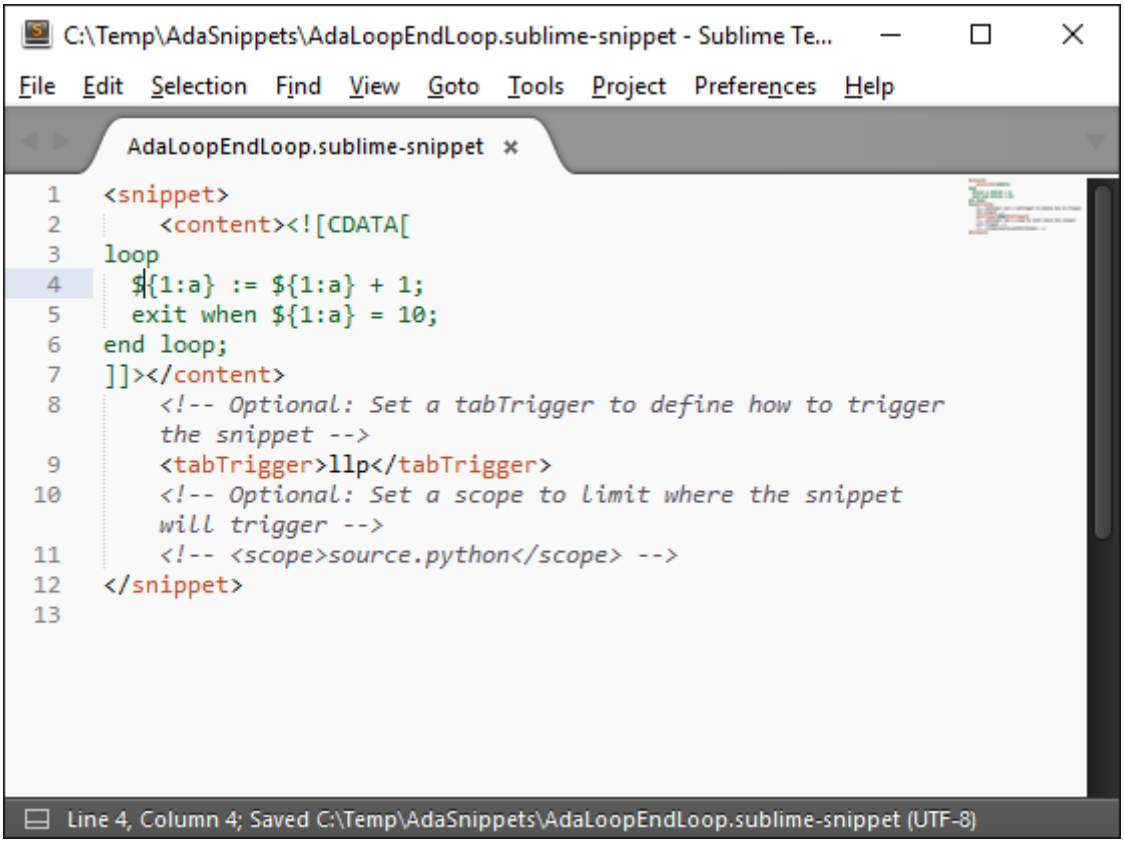


Figure 99: Preparing a Second Code Snippet with Sublime Text

Save the snippet as **AdaLoopEndLoop**. Now open a command prompt in the directory where you saved the code snippet files and type **`yo code`**. When the Yeoman tool starts, select **New Code Snippets** as the extension generation option. By using Figure 100 as a reference, provide in sequence:

1. The folder that contains TextMate or Sublime snippets. You can type **`.`** if you already opened a command prompt in the folder that contains the desired snippets.
2. The name of the extension, e.g., Ada Code Snippets.
3. The extension identifier. You can accept the auto-generated identifier.
4. An extension description.
5. Your publisher name, as you did with the language support generation.
6. The language **`id`** that your snippets are for. In this case, the language **`id`** is **ada**. In the case that you are packaging code snippets for other languages, you will need to supply the **`id`** for those languages (e.g., Markdown, Python).

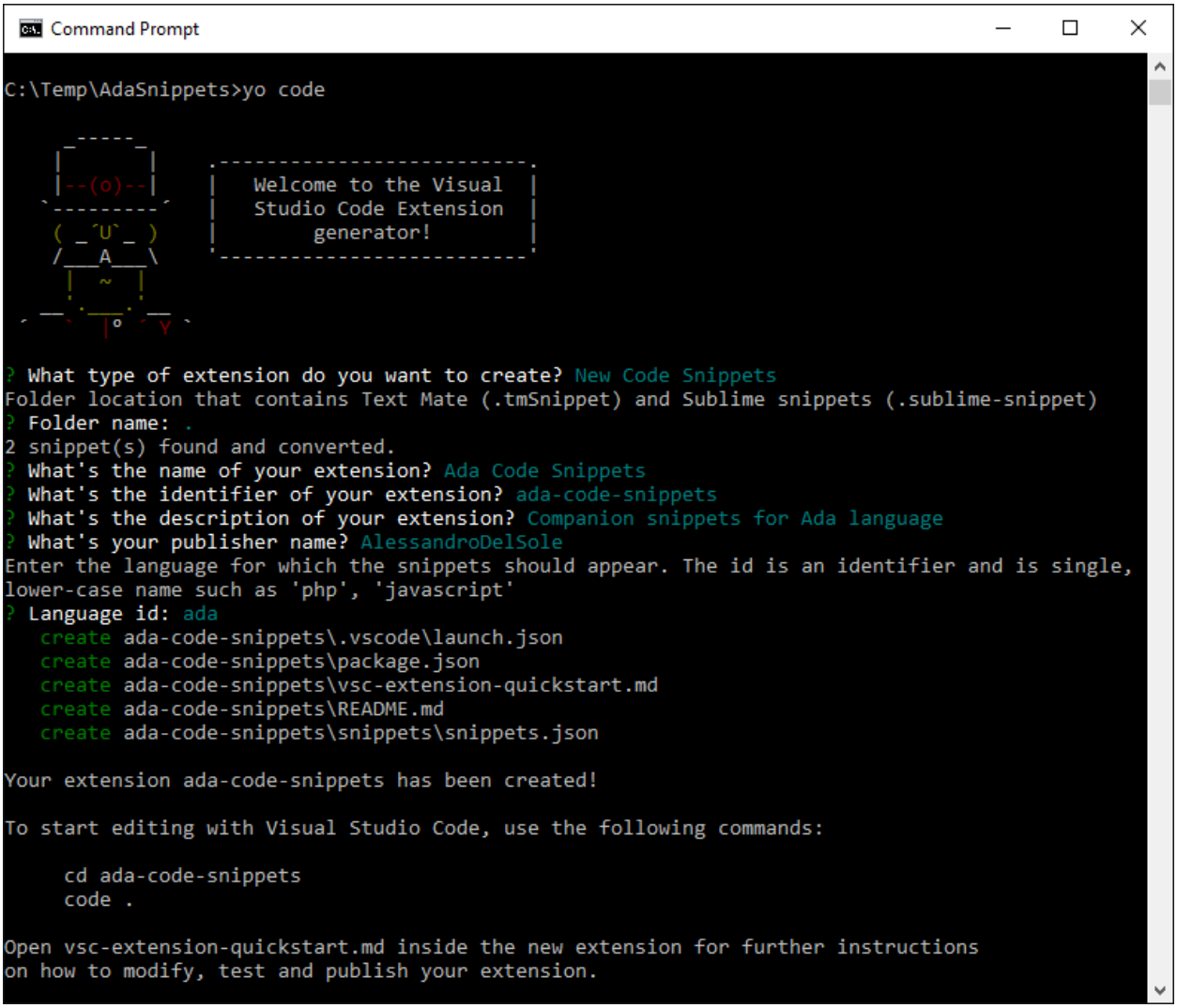


Figure 100: Generating a Code Snippet Extension

As for a new language, Yeoman generates a new folder containing the extension manifest (package.json), a README.md file that you are encouraged to edit before you distribute your extension, and a subfolder called **snippets**, which contains a file called **snippets.json**. This one contains all the code snippets you provided, converted to JSON format. You can certainly open the new folder with Visual Studio Code and investigate the extension structure, not just edit the README.md file. As it is, the code snippet extension is ready for distribution, so it is really easy to package and distribute code snippets for existing languages. However, the goal now is to distribute both the Ada language support and the JSON code snippets. First, copy the **snippets** subfolder into the folder that contains the Ada language extension (see the steps listed just before creating code snippets). Just to make sure you copy this folder into the proper place, it is at the same level of the **syntaxes** and **.vscode** folders. Second, you need to make an edit to the **package.json** file that was created for the Ada language support. More specifically, because you can combine multiple contributes in one extension, you will add a **`snippets`** property as shown in Code Listing 11. This **`snippets`** property can be simply copied and pasted from the package.json that was created for the code snippet extension.

Code Listing 11

```
{
    "name": "ada",
    "displayName": "ADA",
    "description": "ADA language and code snippets",
    "version": "0.0.1",
    "publisher": "AlessandroDelSole",
    "engines": {
        "vscode": "^0.10.10"
    },
    "categories": [
        "Languages", "Snippets"
    ],
    "contributes": {
        "languages": [{
            "id": "ada",
            "aliases": ["Ada", "ada"],
            "extensions": [".adb",".ads"],
            "configuration": "./ada.configuration.json"
        }],
        "grammars": [{
            "language": "ada",
            "scopeName": "source.ada",
            "path": "./syntaxes/ada.tmLanguage"
        }],
        "snippets": [{
                "language": "ada",
                "path": "./snippets/snippets.json"
        }]
    }
}
```

Notice how the **`categories`** property has been extended to included snippets as well. This will make the extension discoverable when users search for both languages and snippets. Save your edits, and run Visual Studio Code again. You already saw how the language support works, so now type the snippet shortcuts **`flp`** and then press **Tab**, or **`llp`** and then press **Tab**. Figure 101 shows how the **`loop..end loop`** code snippet has been inserted, highlighting placeholders for replacements.

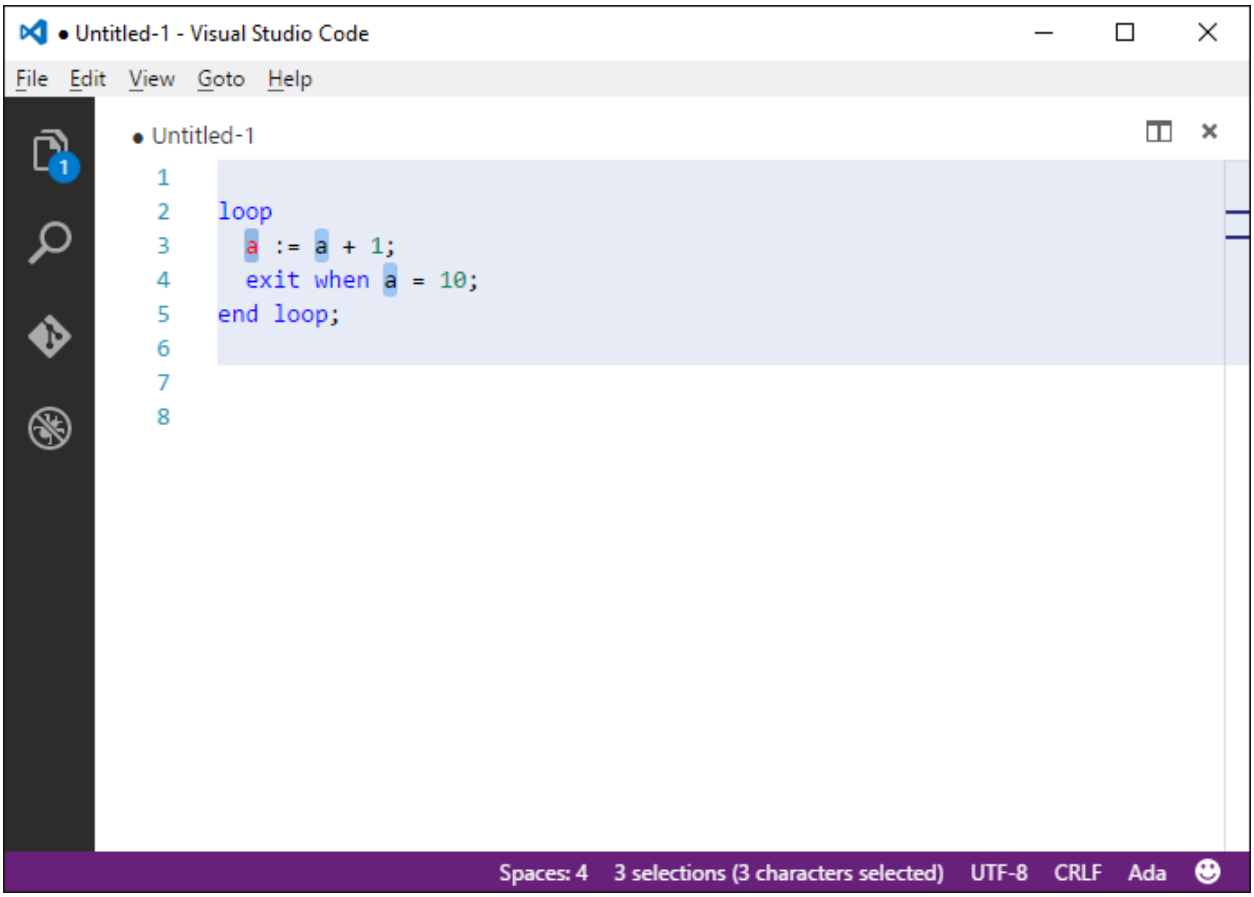


Figure 101: Support for Ada Code Snippets

So with relatively little effort you have been able to plug a new language with code snippets into Visual Studio Code. At this point, you are ready to share the result of your work with the rest of the world.

Become a registered publisher

Extending and customizing Visual Studio Code for your own usage is certainly helpful, but the real fun begins when you share your work with other developers. You can easily publish your extensions to the Visual Studio Code Marketplace so that other developers will be able to download them from within Code and will also be able to view summary information in the Marketplace portal. Before you start, you need to register as a publisher. The registration process passes through the Microsoft Visual Studio Team Services. Formerly known as Visual Studio Online, VS Team Services provides free services for up to five users on your team, such as version control, work management, build and test automation, and much more. The first thing you need to do is signing up for your free account. Once registered, you need a **Personal access token** which grants you permissions to publish your extensions to the Marketplace. To accomplish this, follow these steps:

1. Click your name in the upper-right corner of the main page, and then click **My Profile**.
2. Once you have accessed your profile administration, click the **Security** tab on the left side of the page (see Figure 102).

3. Click **Personal access tokens**, and then click **Add**.
4. Enter a name for your personal access token. As you can see in Figure 102, I am using **vsce**, which recalls the name of the tool you use to publish to the Marketplace and which you will use shortly, but you can use any name you like.
5. In the **Expires in** field, enter **1 year**.
6. In the **Accounts** field, select **All accessible accounts** to grant permission to any account associated to the current subscription.
7. In the **Authorized Scopes** group, select **All scopes**.
8. Click **Save** at the bottom of the page (not visible in Figure 102).

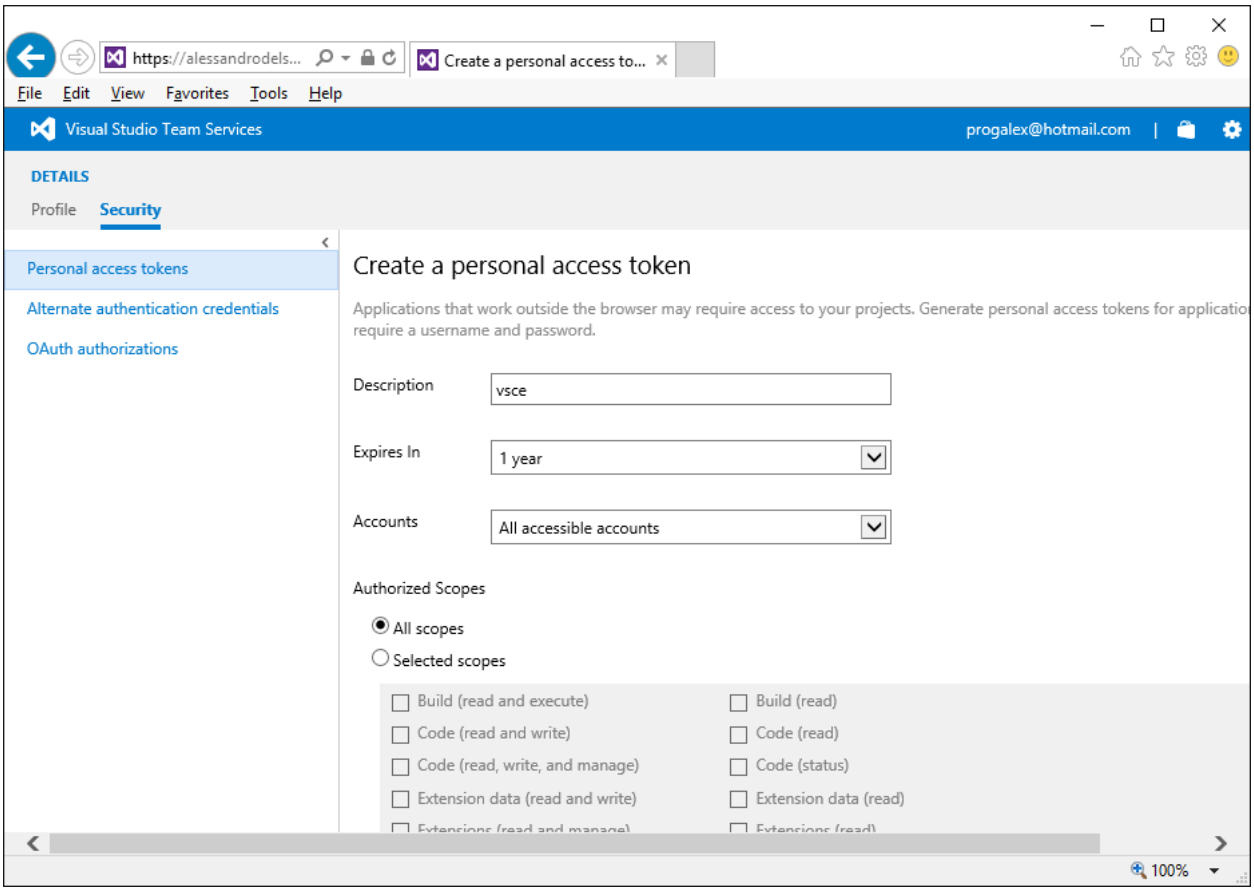


Figure 102: Creating a Personal Access Token

Note: Because the personal access token will not be stored anywhere, it is essential that you follow the portal's suggestion of saving it in a secure place, as you will need it to publish extensions online.

Customizing the extension manifest

The extension manifest, that is the package.json file, can be customized further to offer useful information to end users. The official documentation provides full details about the manifest customization, but here I will show common customizations. For instance, you can add an **icon** property to specify a 128 × 128 image that will be used to promote your extension in the gallery, specify a **license** property that indicates a license agreement for your extension, specify a **homepage** property that points to the home page of your extension on the web, specify a **bugs** property where you can specify a URL and email address that users might use to file bugs. Code Listing 12 shows an example of manifest customization including an icon, a license, a home page, and a bug submission page.

Code Listing 12

```
{
  "name": "ada",
  "displayName": "ADA",
  "description": "ADA language and code snippets",
  "version": "0.0.1",
  "publisher": "AlessandroDelSole",
  "icon": "images/MyIcon.jpg",
  "license": "SEE LICENSE IN LICENSE.md",
  "homepage":
"https://github.com/AlessandroDelSole/VSCodeSuccinctly/blob/AdaLanguage/REA
DME.md",
  "bugs": {
    "url": "https://github.com/AlessandroDelSole/VSCodeSuccinctly/issues",
    "email": "alessandro.delsole@visual-basic.it"
  },
  "engines": {
    "vscode": "^0.10.10"
  },
  "categories": [
    "Languages"
  ],
  "contributes": {
    "languages": [
      {
        "id": "ada",
        "aliases": [ "Ada", "ada" ],
        "extensions": [ ".adb", ".ads" ],
        "configuration": "./ada.configuration.json"
      }
    ],
    "grammars": [
      {
        "language": "ada",
        "scopeName": "source.ada",
        "path": "./syntaxes/ada.tmLanguage"
```

```
      }
    ],
    "snippets": [
      {
        "language": "ada",
        "path": "./snippets/snippets.json"
      }
    ]
  }
}
```

A couple of notes about the extension manifest:

- To provide an icon, create a subfolder inside the extension's folder and place a 128 × 128 image there.
- The **`license`** property value must always be **`SEE LICENSE IN LICENSE.md`**, so you need to add a **`LICENSE.md`** file in the root folder of your extension.

Do not forget to check out additional possibilities in the official documentation. Now you really have all you need to publish your work online.

Publishing and sharing to the Visual Studio Code Marketplace

In order to publish extensions to the Visual Studio Code Marketplace, you need a command-line tool called **`vsce`**. You can get it by typing the following in the command line:

```
> npm install -g vsce
```

Next, open a command prompt in the root folder of the extension you want to publish, in this case the Ada language extension. The first time you publish an extension, you first need to create a publisher identity associating vsce with your personal access token, which is accomplished with the following command:

```
> vsce create-publisher publisherName
```

Where **`publisherName`** is the unique identifier you want to use. When you press **Enter**, you will be prompted to enter a friendly, human readable name, and the personal access token you received previously. You will need to perform this step just once. The next step is simply typing **`vsce publish`** to publish your extension to the Visual Studio Code Marketplace. If everything succeeds, the extension will be available online in a few seconds. Figure 103 shows what the Ada extension looks like in the Marketplace portal.

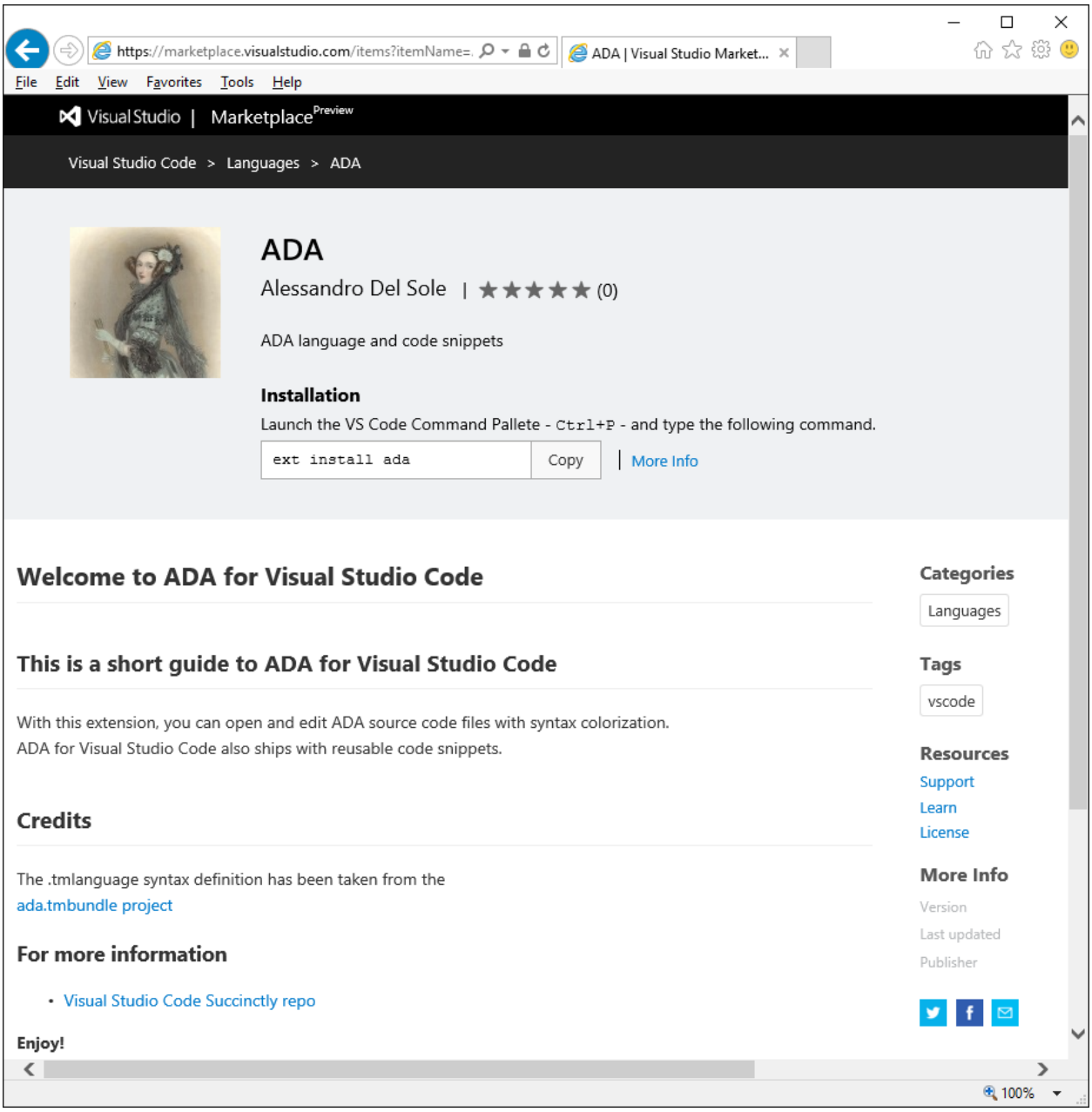


Figure 103: The published extension appears in the VS Code Marketplace.

Figure 104 shows instead how the extension appears in the Command Palette, where you can see how users can look at the license agreement and open the Readme file.

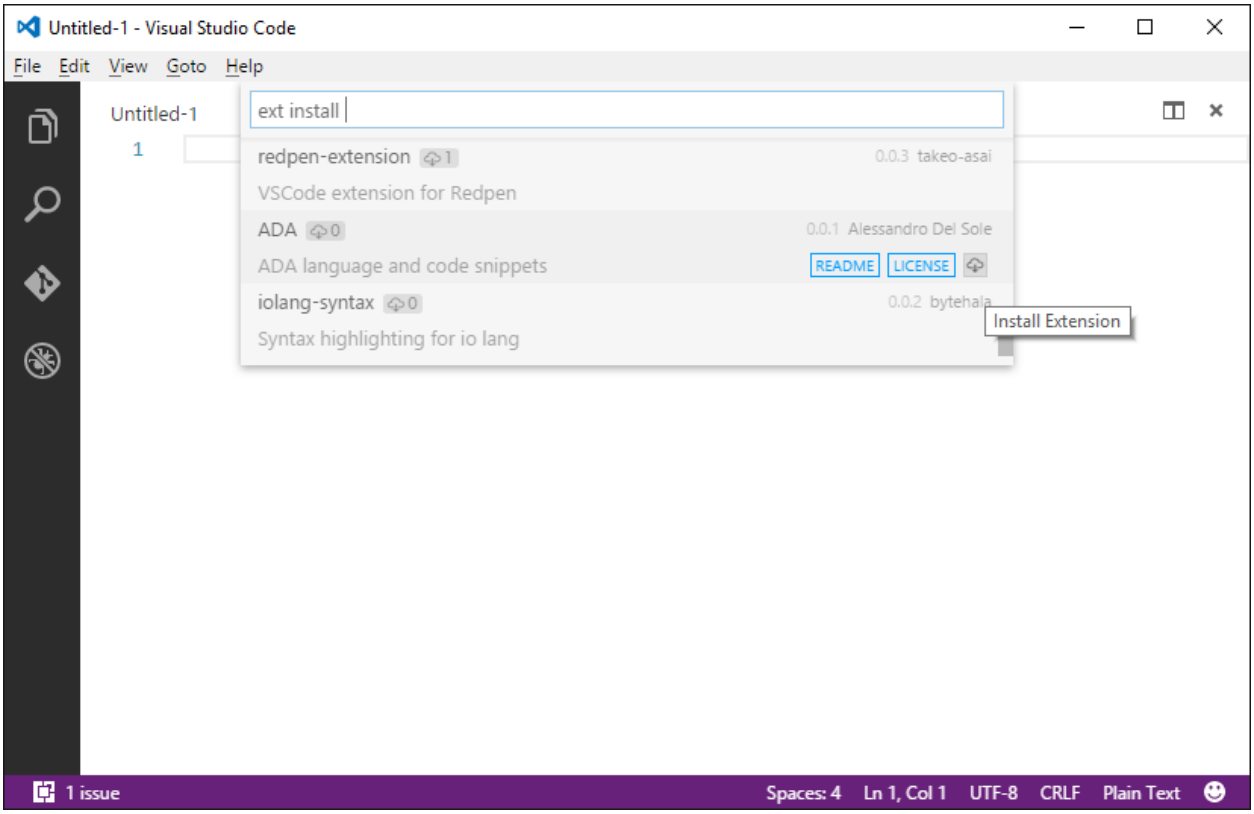


Figure 104: The extension appears in the Command Palette.

There is so much more in the Visual Studio Code extensibility, but in this chapter you have seen a number of steps, information, and resources that are common to any customization and extension you want to build.

Chapter Summary

Visual Studio Code can be customized and extended in many ways. You can customize VS Code's settings, such as the color theme, user and workspace settings, and key bindings. You can also download and install extensions from the Visual Studio Code Marketplace, and manage installed extensions, using the Command Palette. Not limited to this, you can write your own customizations and extensions using the Yeoman generator for Visual Studio Code, which enables you to get started with extensibility quickly, such as generating new language support, code snippets, color themes, and other kinds of extensions. Advanced extensibility has not been discussed in this chapter, but you can definitely have a look at the documentation.