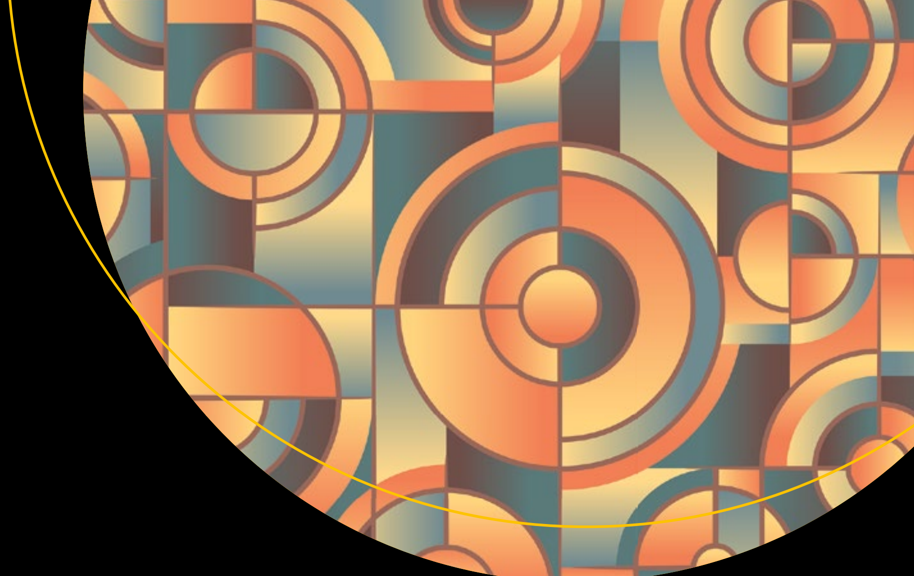


Getting Started with Visual Studio 2019

Learning and Implementing New Features

Dirk Strauss

Apress®

Getting Started with Visual Studio 2019

Learning and Implementing New Features

Dirk Strauss

Apress®

Getting Started with Visual Studio 2019: Learning and Implementing New Features

Dirk Strauss
Uitenhage, South Africa

ISBN-13 (pbk): 978-1-4842-5448-6 ISBN-13 (electronic): 978-1-4842-5449-3
https://doi.org/10.1007/978-1-4842-5449-3

Copyright © 2020 by Dirk Strauss

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

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

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

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

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Smriti Srivastava
Development Editor: Laura Berendson
Coordinating Editor: Shrikant Vishwakarma

Cover designed by eStudioCalamar

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

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

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

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

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-5448-6. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

To Adele, Tristan, and Irénéé.
My everything for you, always!

Table of Contents

About the Author ix

About the Technical Reviewer xi

Acknowledgments xiii

Introduction xv

Chapter 1: Getting to Know Visual Studio 2019 1

Installing Visual Studio 2

Visual Studio 2019 System Requirements 4

Using Workloads 6

Exploring the IDE 9

The Solution Explorer 9

Toolbox 19

The Code Editor 21

Navigating Code 26

Navigate Forward and Backward Commands 26

Navigation Bar 27

Find All References 28

Reference Highlighting 30

Go To Commands 31

Go To Definition 33

Peek Definition 34

Features and Productivity Tips..35
Track Active Item in Solution Explorer..36
Hidden Editor Context Menu..37
Open in File Explorer 38
Finding Keyboard Shortcut Mappings 39
Clipboard History...39
Go To Window 40
Navigate to Last Edit Location..41
Multi-caret Editing..41
Features in Visual Studio 2019 44
Visual Studio Search...44
Solution Filters 45
Visual Studio IntelliCode...51
Visual Studio Live Share..55

Chapter 2: Working with Visual Studio 2019......................................61

Visual Studio Project Types...62
Various Project Templates 67
Managing NuGet Packages...73
Using NuGet in Visual Studio 74
Hosting Your Own NuGet Feeds..81
Creating Project Templates 82
Creating and Using Code Snippets...87
Creating Code Snippets...90
Using Bookmarks and Code Shortcuts..95
Bookmarks 95
Code Shortcuts...98
Adding Custom Tokens 101

The Server Explorer 104
Running SQL Queries 111
Visual Studio Windows 116
C# Interactive 117
Code Metrics Results 118
Send Feedback 122

Chapter 3: Debugging Your Code 123

Working with Breakpoints 124
Setting a Breakpoint 124
Conditional Breakpoints and Actions 130
Manage Breakpoints with Labels 135
Exporting Breakpoints 138
Using DataTips 139
Visualizing Complex Data Types 140
Using the Watch Window 144
The DebuggerDisplay Attribute 144
Evaluate Functions Without Side Effects 147
Format Specifiers 151
Diagnostic Tools 152
CPU Usage 155
Memory Usage 157
The Events View 159
The Right Tool for the Right Project Type 160
Immediate Window 161
Attaching to a Running Process 163
Attach to a Remote Process 165
Reattaching to a Process 166

Remote Debugging 167
System Requirements 167
Download and Install Remote Tools..168
Running Remote Tools..169
Start Remote Debugging 170

Chapter 4: Unit Testing 175

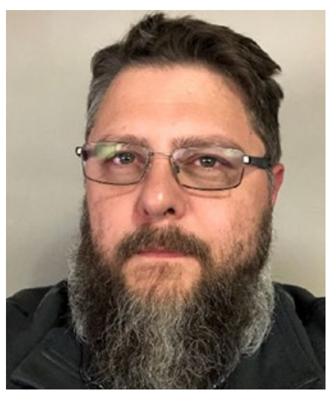
Creating and Running Unit Tests 175
Create and Run a Test Playlist 182
Testing Timeouts...184
Using Live Unit Tests 186
Using IntelliTest to Generate Unit Tests...191
Focus IntelliTest Code Exploration..202
How to Measure Code Coverage in Visual Studio 204

Chapter 5: Source Control...209

Create a GitHub Account..210
Create and Clone a Repository...214
Cloning a Repository...228
Create a Branch from Your Code...233
Creating and Handling Pull Requests..240
Working with Stashes...255

Index...261

About the Author



Dirk Strauss is a software developer from South Africa with over 13 years of programming experience. He has extensive experience in SYSPRO Customization, with C# and web development being his main focus. He studied at the Nelson Mandela University where he wrote software part-time to gain a better understanding of the technology. He remains passionate about writing code and imparting what he learns with others.

About the Technical Reviewer

James McCaffrey works for Microsoft Research in Redmond, Wash. James has a PhD in cognitive psychology and computational statistics from the University of Southern California, a BA in psychology, a BA in applied mathematics, and an MS in computer science. James worked on several key products including Azure and Bing. James is also the Senior Technical Editor for Microsoft *MSDN Magazine*, the most widely read technical journal in the world.

Acknowledgments

First off, I would like to thank my wife and children for their support while writing this book. I would not have been able to do it without you on my side.

I would also like to thank the team at Apress for their support of this book and for turning my vision into reality. It is a topic that I have wanted to write about for a very long time.

I want to thank James McCaffrey for his help and dedication during the review of this book. Technically reviewing a book such as this is not easy, and his feedback and suggestions are always appreciated and highly valued.

Last, but not least, I want to thank you for reading this book. Your passion to know more is what drives me to learn more, and impart what I learn. It's a symbiotic relationship that benefits us as we both grow and become better at what we do.

Introduction

Visual Studio 2019 is the next version of the stellar development tool we love to use. This book is for folks ready to get to know the IDE a little bit better. It aims to get you started on the road to exploring Visual Studio 2019, beyond what you are already comfortable with.

The book starts off with installing Visual Studio and adding workloads. Then you explore the IDE a bit more before having a look at the existing (and some new) features in Visual Studio. After that, a few productivity tips are thrown in for good measure.

Being able to effectively work with different project types and knowing when to use which are explored in a bit more detail in Chapter 2. We will also have a look at using NuGet packages and how to manage them. We then see how to make use of project templates and then explore using and creating code snippets. This chapter covers many of the basics that are essential to working with Visual Studio and include using bookmarks, code shortcuts, the Server Explorer, and other Visual Studio Windows.

In Chapter 3, we will take a closer look at debugging techniques such as using breakpoints, setting conditional breakpoints, breakpoint actions, and labels. We will see how to effectively use data tips as well as the DebuggerDisplay attribute. We then take a closer look at diagnostic tools and the Immediate Window. Finally, to close off the debugging chapter, we see how to attach to a running process and how to use remote debugging.

The next chapter will introduce you to creating and running unit tests. We will also see how to create live unit tests, how to use IntelliTest to generate unit tests, and how to measure code coverage in Visual Studio.

Finally, we look at working with Git and GitHub. We see how to create a GitHub account and what creating and cloning a repository involves.

You will learn how to commit changes in code to the repository and how to create a branch of your code when you need to work on a new feature in isolation. Then we will look at creating a pull request and how these pull requests are handled. Lastly, we have a look at the benefit of working with stashes.

If you need a nice reference book that deals exclusively with (and only with) Visual Studio, then have a look at what this book has to offer you. If you spend any time using Visual Studio or want to learn how working with Visual Studio 2019 can increase your productivity, then this book will make a perfect reference book for your office.

CHAPTER 1

Getting to Know Visual Studio 2019

Visual Studio is an amazing bit of software. If you have been using Visual Studio for a number of years, you will certainly agree that the IDE offers developers a host of tools and features to make them more productive. You will also be aware that it has grown a lot during the past couple of years and is an absolute powerhouse when it comes to providing tools to develop world-class software.

Initially released as Visual Studio 97 in February 1997, this was the first attempt at using a single development environment for multiple languages. The evolution of Visual Studio is detailed in Table 1-1.

Table 1-1. *The evolution of Visual Studio*

Release	Version	.NET Framework	Release Date
Visual Studio 2019	16.0	3.5–4.8	April 2, 2019
Visual Studio 2017	15.0	3.5–4.7	March 7, 2017
Visual Studio 2015	14.0	2.0–4.6	July 20, 2015
Visual Studio 2013	12.0	2.0–4.5.2	October 17, 2013
Visual Studio 2012	11.0	2.0–4.5.2	September 12, 2012
Visual Studio 2010	10.0	2.0–4.0	April 12, 2010

(*continued*)

© Dirk Strauss 2020
D. Strauss, *Getting Started with Visual Studio 2019*,
https://doi.org/10.1007/978-1-4842-5449-3_1

Table 1-1. *(continued)*

Release	Version	.NET Framework	Release Date
Visual Studio 2008	9.0	2.0, 3.0, 3.5	November 19, 2007
Visual Studio 2005	8.0	2.0, 3.0	November 7, 2005
Visual Studio .NET 2003	7.1	1.1	April 24, 2003
Visual Studio .NET 2002	7.0	1.0	February 13, 2002
Visual Studio 6.0	6.0	N/A	June 1998
Visual Studio 97	5.0	N/A	February 1997

There is so much to see and learn when it comes to Visual Studio. Therefore, in this chapter, we will start by having a look at the following:

- Installing Visual Studio
- What workloads are
- Exploring the IDE (integrated development environment)
- Existing and new features available in Visual Studio 2019
- Productivity tips

If you are using a macOS or a Windows machine, Visual Studio will happily run on both. Let us see where to find the Visual Studio Installer and get going.

Installing Visual Studio

At the time of this writing, Visual Studio 2019 is available for Windows machines as well as for macOS machines. You can download Visual Studio 2019 for Windows from `https://visualstudio.microsoft.com/vs/`, and if you are on macOS, you will need to head on over to `https://visualstudio.microsoft.com/vs/mac/` to download the installer.

Clicking the Download Visual Studio button, you will see a list drop-down with the options as displayed in Figure 1-1.

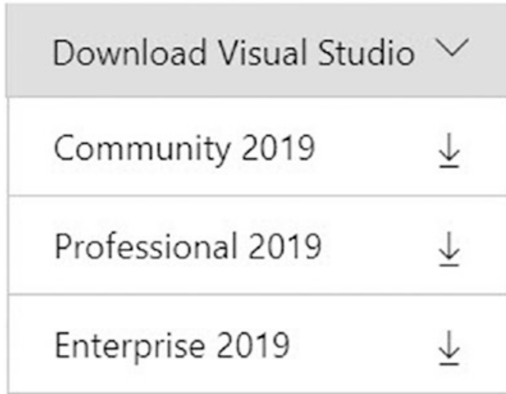


Figure 1-1. *Versions of Visual Studio available*

If you would like to compare the Visual Studio 2019 versions, you can have a look at `https://visualstudio.microsoft.com/vs/compare/` for a detailed comparison. The bottom line is that if you want Visual Studio 2019 for free, download Visual Studio Community 2019.

Visual Studio Community 2019 is aimed at students, open source, and individual developers. The paid tiers include Visual Studio Professional 2019 which is aimed toward small teams and Visual Studio Enterprise 2019 aimed at large development teams.

Microsoft specifies enterprise organizations as those having more than 250 PCs or more than $1 million US dollars in annual revenue.

Let us have a brief look at the recommended system requirements for installing Visual Studio on your machine. For a comprehensive list, browse to `https://docs.microsoft.com/en-us/visualstudio/releases/2019/system-requirements` and have a read through that.

Visual Studio 2019 System Requirements

The system requirements for installing Visual Studio 2019 might differ from those of previous versions of Visual Studio. Refer to the documentation on `https://docs.microsoft.com` to review the system requirements for previous versions of Visual Studio.

Visual Studio Enterprise 2019, Visual Studio Professional 2019, Visual Studio Community 2019, and Visual Studio Team Foundation Server Office Integration 2019 all support the following minimum system requirements.

Operating Systems

The following Windows operating systems (64-bit recommended) are supported:

- Windows 10 version 1703 or higher
- Windows Server 2019 – Standard and Datacenter
- Windows Server 2016 – Standard and Datacenter
- Windows 8.1 with update 2919355
- Windows Server 2012 R2 with update 2919355
- Windows 7 SP1 with latest Windows updates

Hardware

There is obviously a line here that developers generally don't like to cross when it comes to the minimum hardware specs. Many developers I know will geek out on system RAM and favor SSDs over HDDs. Nevertheless, here are the minimum recommended requirements:

- 1.8 GHz or faster processor (quad-core or better recommended).
- 2 GB of RAM (8 GB of RAM recommended).

- 2.5 GB of RAM minimum if running on a Virtual Machine.
- Between 800 MB and 210 GB of available hard disk space (depending on installed features, 20–50 GB of free space is typically required).
- For improved performance, install Windows and Visual Studio on an SSD.
- Minimum display resolution of 720p (1280x720) but works best at WXGA (1366x768) or higher.

Supported Languages

Visual Studio and the Visual Studio Installer is available in 14 languages as follows:

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

- Spanish
- Turkish

Additional Notes

There are several additional requirements to take note of that I will briefly list here. There are however other requirements that might be of importance to your unique development environment. For a full list, refer to the system requirements at the following link: `https://docs.microsoft.com/en-us/visualstudio/releases/2019/system-requirements`

- Administrator rights are required to install Visual Studio.
- .NET Framework 4.5 is required to run the Visual Studio Installer and install Visual Studio.
- Visual Studio requires .NET Framework 4.7.2 and is installed during setup.

Using Workloads

After Visual Studio has been installed, you can customize the installation by selecting feature sets, also known as workloads. Think of workloads as a collection of individual features that belong together. This allows you to easily modify Visual Studio to include only what you need.

To launch the workloads screen, find the Visual Studio Installer as can be seen in Figure 1-2.

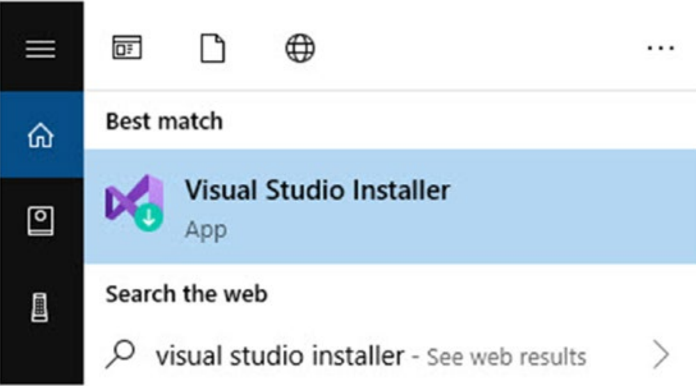


Figure 1-2. *Find the Visual Studio Installer*

Clicking the Visual Studio Installer will launch the installer from where you can modify your installation of Visual Studio as seen in Figure 1-3.

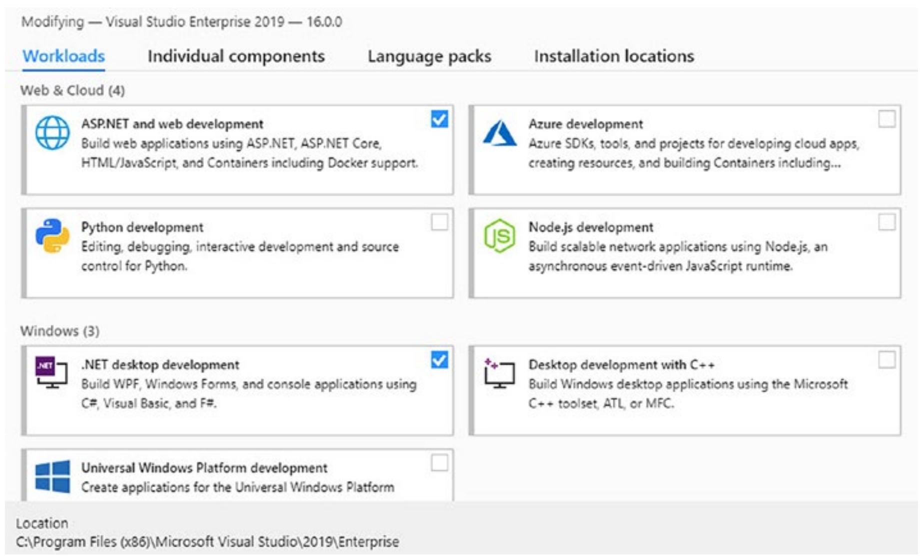


Figure 1-3. *Installing additional workloads*

If you want to start doing Python development, you can simply check the *Python development* workload and install that. As can be seen in Figure 1-4, this will update the installation details section and show you exactly what is being installed and how much additional space you will need to install the selected workload.

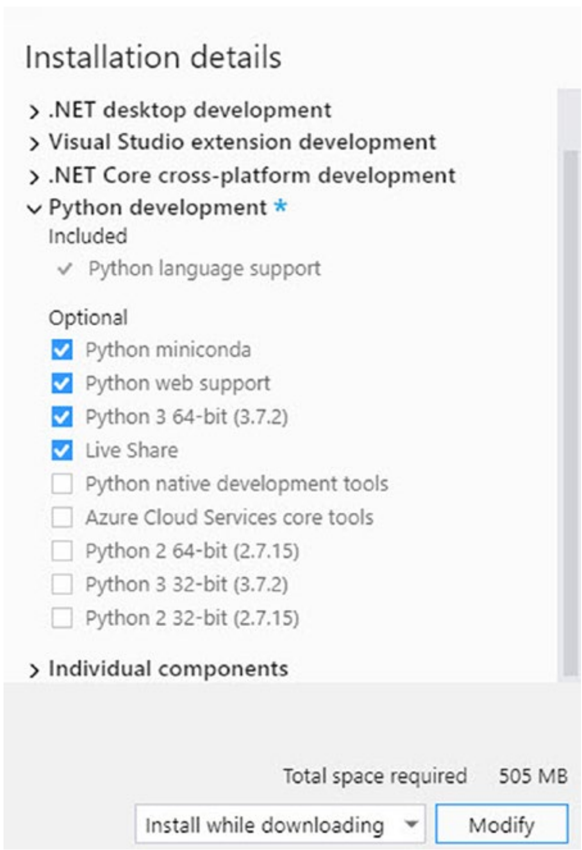


Figure 1-4. *Workload installation details*

The workloads also contain tabs, namely, Workloads, Individual components, Language packs, and Installation locations. If you needed to install an additional component such as Service Fabric Tools, you can do so by selecting the component on the Individual components tab.

When you have checked all the workloads and individual components you would like to install, you can choose to do the installation while downloading or to download everything before installing as can be seen in Figure 1-5.

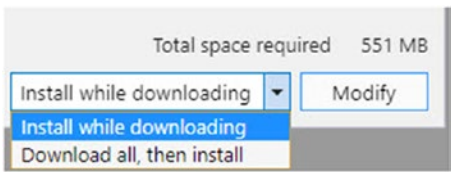


Figure 1-5. *Installation options*

This will modify your existing installation of Visual Studio 2019 and apply the changes you selected.

Exploring the IDE

The Visual Studio IDE is full of features and tools that help developers do what they need to do, efficiently and productively. Developers start off creating one or more projects that contain the logic for their code. These projects are contained in what we call a solution. Let's have a look at the Solution Explorer first.

The Solution Explorer

In Visual Studio, the notion of solutions and projects is used. A solution contains one or more projects. Each project contains code that runs the logic you need in order for your application to do what it does.

Consider the example of a Shipment Locator application as can be seen in Figure 1-6.

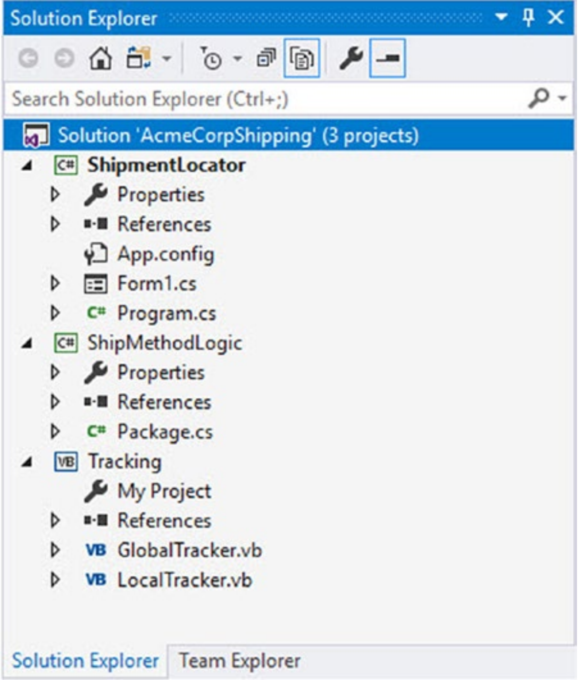


Figure 1-6. *The Shipment Locator Solution*

It is to this solution that you will add all the required projects in order to create your application. From the example in Figure 1-6, we can see that the solution contains three projects. The projects are as follows:

- ShipmentLocator – WinForms application – C#
- ShipMethodLogic – Class Library – C#
- Tracking – Class Library – VB.NET

Of particular interest, you will notice that you can have a solution that contains a mix of C# projects and VB.NET projects. You are therefore not limited by a particular language and can create applications containing a mix of .NET languages.

The reason that we can mix .NET languages in the same solution is due to something we call IL (Intermediate Language). IL is used by the .NET Framework to create machine-independent code from the source code used in your projects.

The WinForms application will contain the UI needed to track and trace shipments. In order for the WinForms application to be able to use the logic contained in the other two class libraries, we need to add what is called a reference to the other projects.

This is done by right-clicking the project that you want to add the reference to and selecting *Add Reference* from the context menu (Figure 1-7).

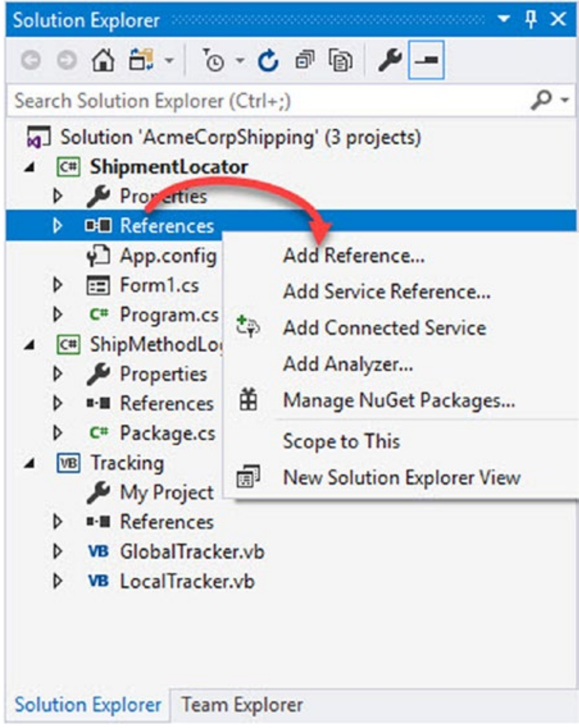


Figure 1-7. *Adding a project reference*

When you click the Add Reference menu item, you will be presented with the Reference Manager screen as seen in Figure 1-8.

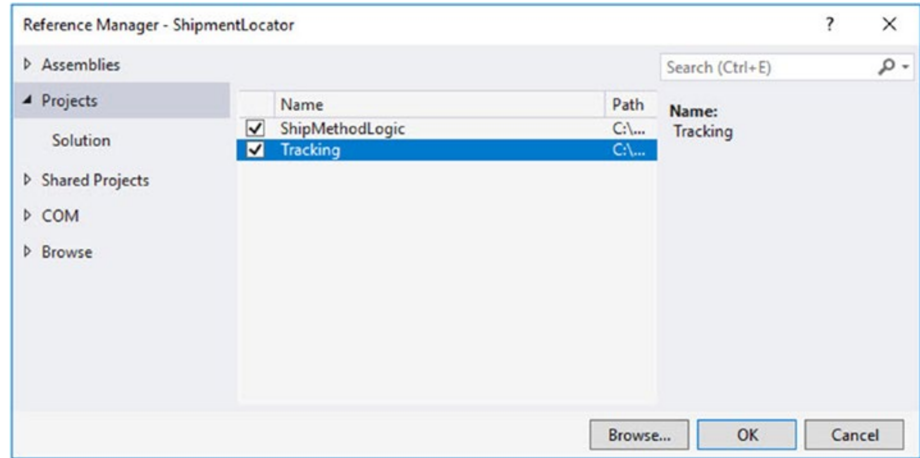


Figure 1-8. *The Reference Manager screen*

Under the Projects tab, you will find the other two Class Library projects in your solution. By checking each one and clicking the OK button, you will add a reference to the code in these projects.

If you had to expand the References section under the ShipmentLocator project, you will see that there are two references to our Class Library projects ShipMethodLogic and Tracking as can be seen in Figure 1-9.

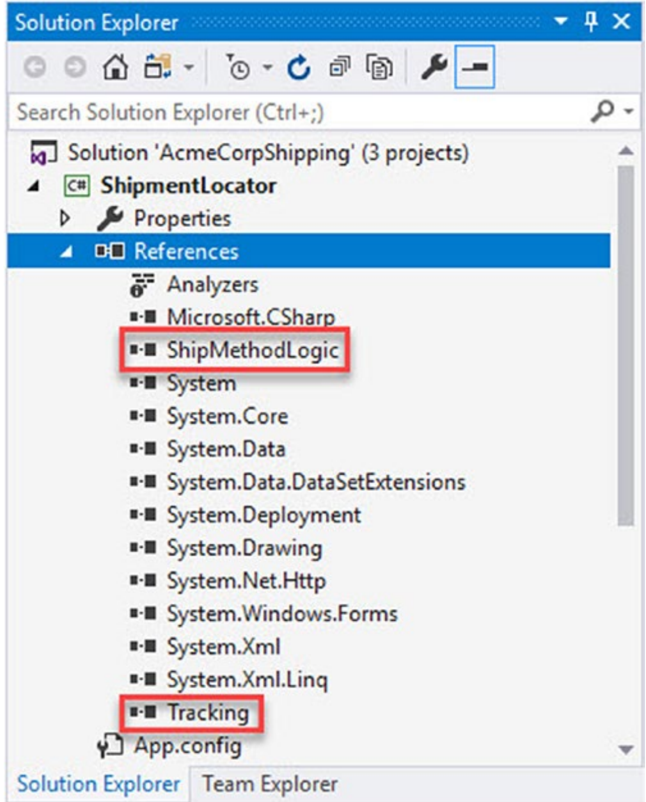


Figure 1-9. *Added references*

This will now make all the code you write in the ShipMethodLogic and Tracking projects, available to the ShipmentLocator project. Having a look at the toolbar on the Solution Explorer (Figure 1-10), you will notice that it contains several buttons.

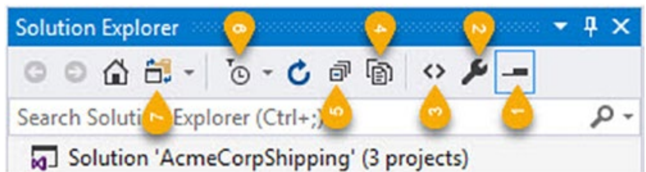


Figure 1-10. *The Solution Explorer toolbar*

The buttons contained here are displayed as needed. The View Code button, for example, will only show up in the toolbar when a file containing code is selected in the Solution Explorer. These buttons give you quick access to the following features as outlined in the following:

1. Preview Selected Items
2. Properties
3. View Code
4. Show All Files
5. Collapse All
6. Pending Changes Filter
7. Toggle between Solution and Folder views

I will not go through each one in detail, but of particular interest, you will notice that the Show All Files will display unnecessary files and folders such as the bin folder in your Solution Explorer. Go ahead and click the Show All Files button, and look at the Solution Explorer again.

By looking at Figure 1-11, you can see that it now displays the bin folder and the obj folder. These folders are not necessary for your code but are important to your solution.

The obj folder contains bits of files that will be combined to produce the final executable. The bin folder contains the binary files that are the executable code for the application you are writing.

Each obj and bin folder will contain a Debug and Release folder that simply matches the currently selected build configuration of your project.

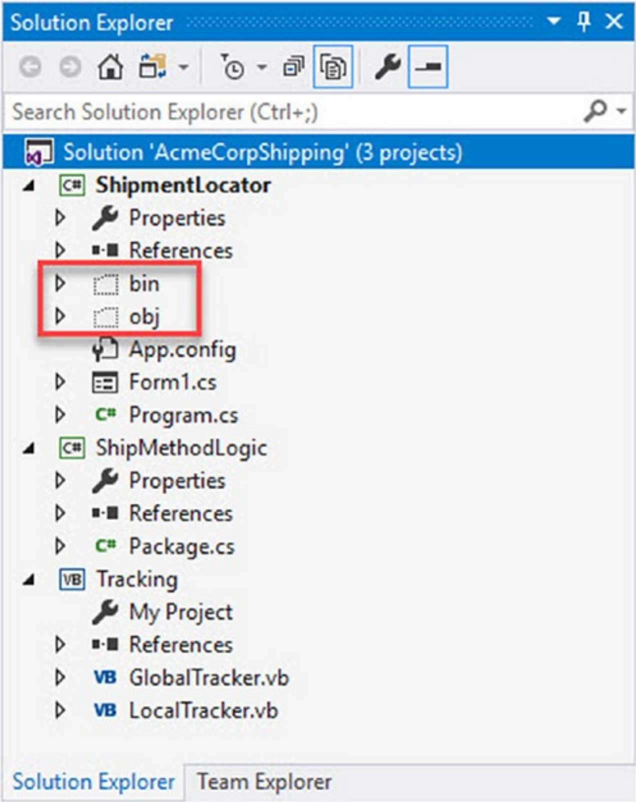


Figure 1-11. *Solution Explorer displaying all files*

You can now right-click the bin folder as seen in Figure 1-12 and click the Open Folder in File Explorer menu to quickly have a look at the contents of the folder.

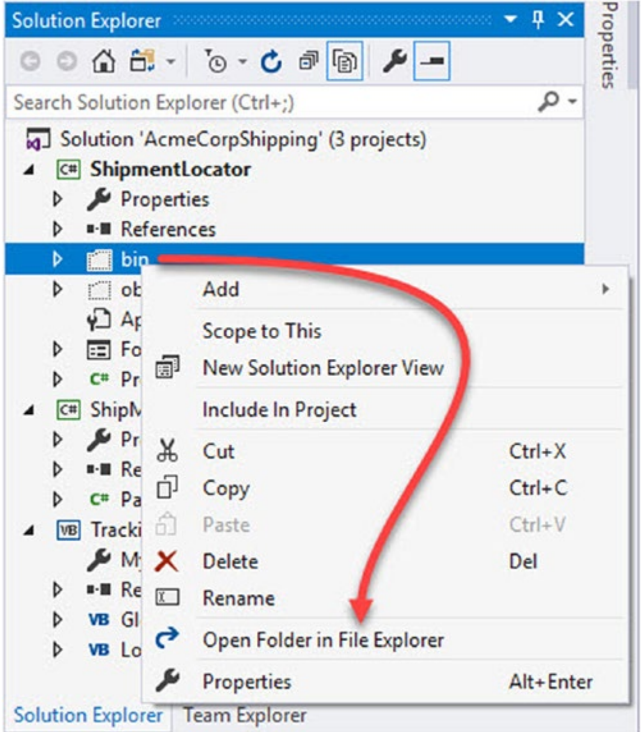


Figure 1-12. *Open Folder in File Explorer*

This is a nice shortcut for anyone needing to navigate to the location of the Visual Studio files in the solution.

If you open the bin folder and click the Debug folder contained in the bin folder, you will see the main exe as well as any referenced dll files in the project (Figure 1-13).

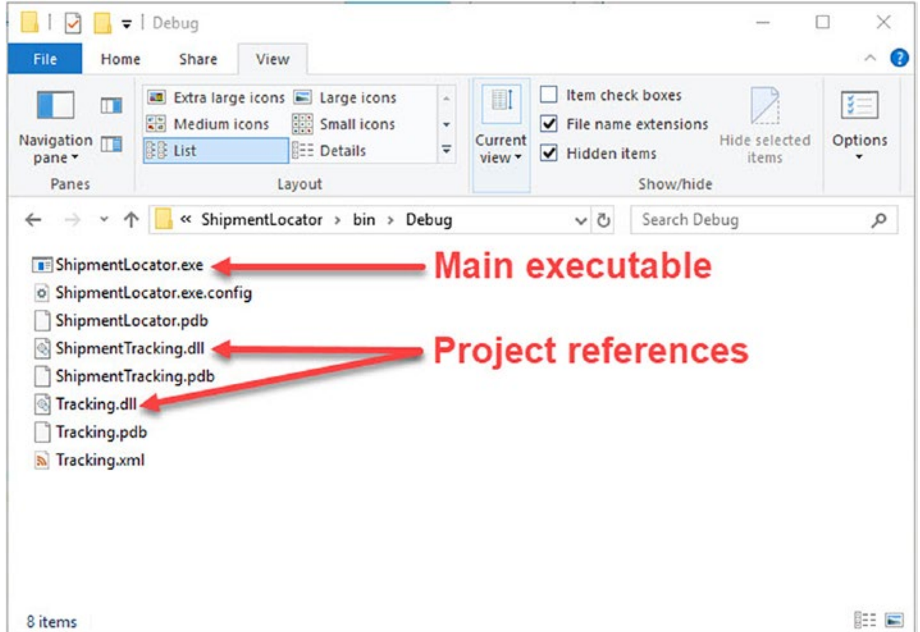


Figure 1-13. *The contents of the Debug folder*

These files will be updated each time you build or run your project. If this folder is blank, perform a build of your solution by pressing F6 or by right-clicking the solution and clicking Build Solution from the context menu as seen in Figure 1-14.

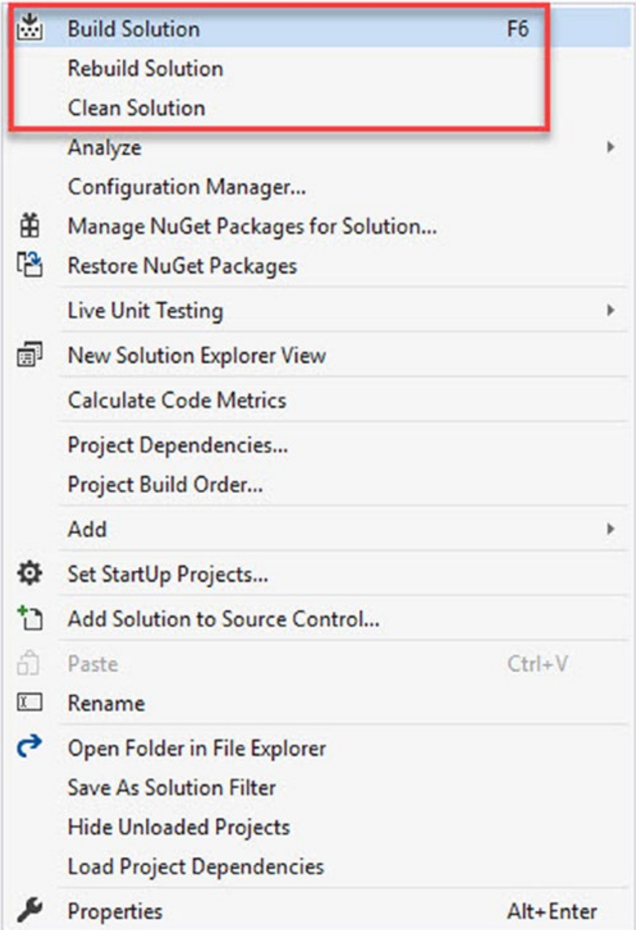


Figure 1-14. *Right-click solution options*

You might be wondering what the difference between Build Solution, Rebuild Solution, and Clean Solution is? The differences are as follows:

- Build Solution will do an incremental build of the solution of anything that has changed since the last build.
- Rebuild Solution will clean the solution and then rebuild the solution from scratch.

- Clean Solution will only clean the solution by removing any build artifacts left over by the previous builds.

If you are receiving funny build errors that do not seem to be errors in your code editor, try cleaning your solution and building it again.

Toolbox

When dealing with a UI file such as a web application or a WinForm application, you will notice that you have a Toolbox at your disposal (Figure 1-15).

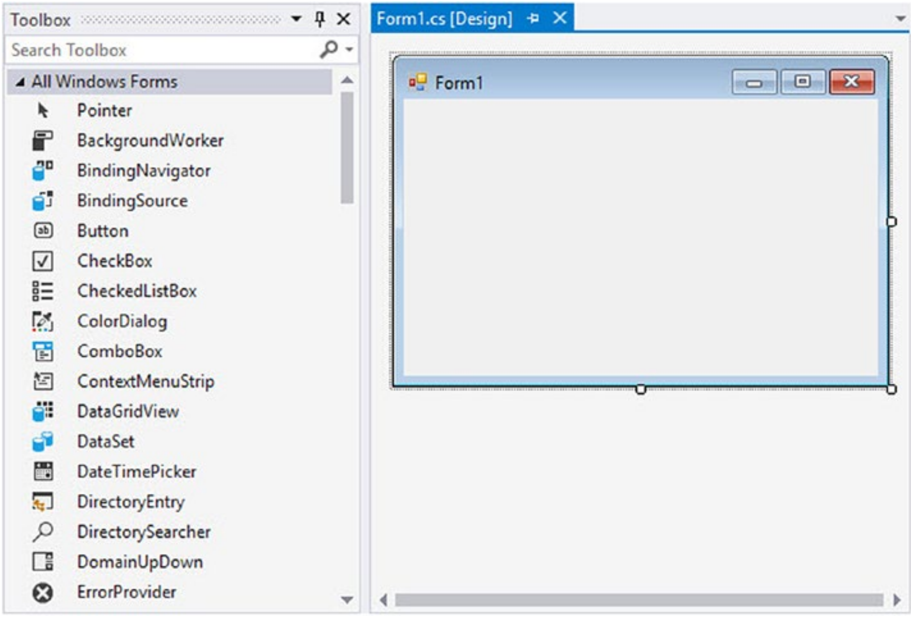


Figure 1-15. *The WinForms Toolbox in Visual Studio*

The Toolbox allows you to add controls to your application such as text boxes, buttons, drop-down lists, and so on. This allows developers to design the UI of the application by dragging and dropping the relevant controls on the design surface.

You can also open the Toolbox by clicking the View menu and selecting the Toolbox menu item. It is worth noting that for some project types, you will not see any items in the Toolbox.

If you do not like the default layout of the Toolbox, you can right-click the tab or on an individual item in the Toolbox and perform one of several actions from the context menu as seen in Figure 1-16.

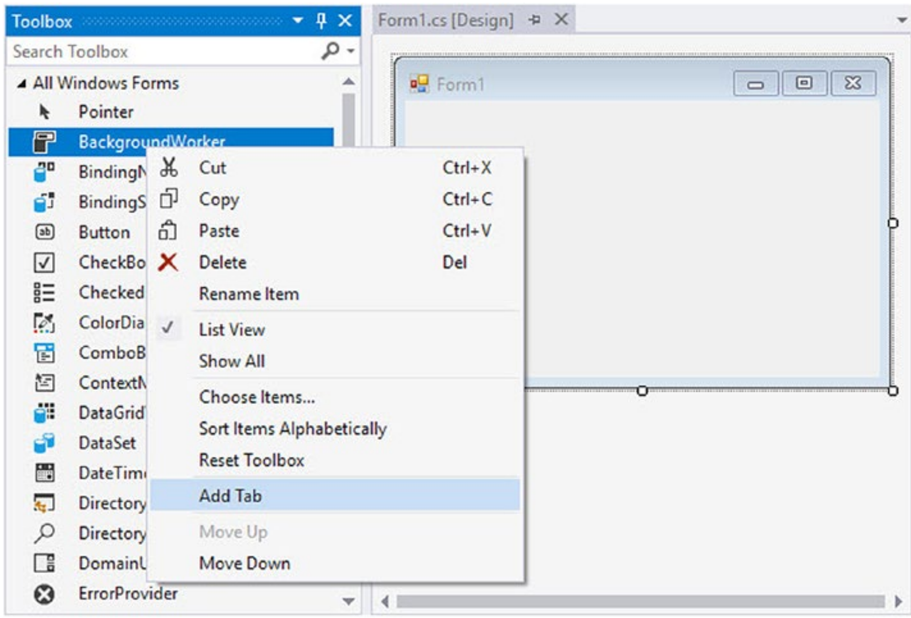


Figure 1-16. *Toolbox context menu*

The context menu allows you to do the following:

- Rename an item
- Choose additional items
- Remove items
- Move items up and down
- Sort items
- Add a new tab

If you have third-party controls installed such as DevExpress or Telerik, you will find the controls specific to the installed components under their own tab in the Toolbox.

The Code Editor

Let's add some basic UI components to our WinForm application as seen in Figure 1-17. To this code-behind, we will add some code to our project, just to get the ball rolling. All that this application will do is take a given waybill number and return some location data for it.

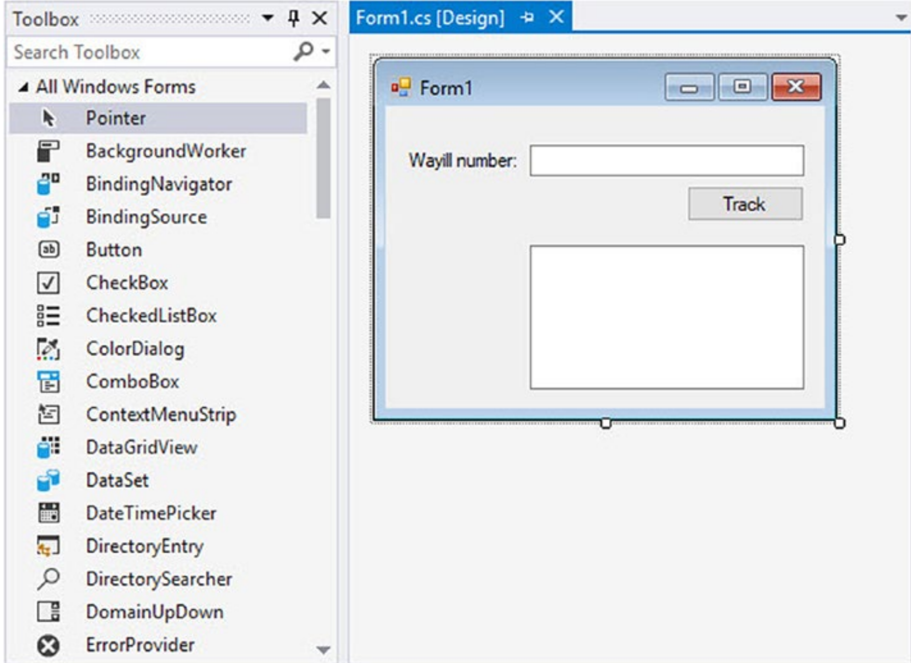


Figure 1-17. *The application design*

The location data will be hard-coded in a Location class that was added to the project.

After adding the UI elements to the designer, swing over to the code window for the main WinForm application called Form1.cs. Add the following code in Listing 1-1 to the code-behind.

You will notice after adding the code that Visual Studio starts to underline some of the added code as seen in Figure 1-18. This is because Visual Studio is making suggestions to improve the quality of your code.

Listing 1-1. The Code-Behind Form1.cs

```
private void BtnTrack_Click(object sender, EventArgs e)
{
    if (!(string.IsNullOrWhiteSpace(txtWaybill.Text)))
    {
        string waybillNum = txtWaybill.Text;
        if (WaybillValid())
        {
            Package package = new Package(waybillNum);
            Location packLoc = package.TrackPackage();
            if (packLoc != null)
            {
                txtLocationDetails.Text = $"Package location: " +
                    $"{packLoc.LocationName} with coordinates " +
                    $"Long: {packLoc.Long} and " +
                    $"Lat: {packLoc.Lat}";
            }
        }
        else
            MessageBox.Show("You have entered an invalid
            Waybill number");
    }
}

private bool WaybillValid()
{
    return txtWaybill.Text.ToLower().Contains("acme-");
}
```

The underlined code is code that Visual Studio is making suggestions for improvement on.

```
Form1.cs    Form1.cs [Design]
ShipmentLocator    ShipmentLocator.Form1    Form1_Load(object sender, EventArg

          1 reference
26        private void BtnTrack_Click(object sender, EventArgs e)
27        {
28            if (!(string.IsNullOrWhiteSpace(txtWaybill.Text)))
29            {
30                string waybillNum = txtWaybill.Text;
31                if (WaybillValid())
32                {
33                    Package package = new Package(waybillNum);
34                    Location packLoc = package.TrackPackage();
35                    if (packLoc != null)
36                    {
37                        txtLocationDetails.Text = $"Package location: " +
38                            $"{packLoc.LocationName} with coordinates " +
39                            $"Long: {packLoc.Long} and " +
40                            $"Lat: {packLoc.Lat}";
41                    }
42                }
43                else
44                    MessageBox.Show("You have entered an invalid Waybill number");
45            }
46        }
47
          1 reference
48        private bool WaybillValid()
49        {
50            return txtWaybill.Text.ToLower().Contains("acme-");
51        }
52    }
53 }

100 %    0    4
```

Figure 1-18. *Visual Studio code improvement suggestions*

To view the details of the suggestion, hover your mouse over one of the underlined lines of code. Visual Studio will now display the details of the suggested change as seen in Figure 1-19.

```
string waybillNum = txtWaybill.Text;
if (WaybillValid())
{
    Package package = new Package(waybillNum);
                                                  ge();
      class ShipmentTracking.Package
      use 'var' instead of explicit type          ge location: " +
      Name can be simplified.                      coordinates " +
      Show potential fixes (Alt+Enter or Ctrl+.)
    }
}
else
    MessageBox.Show("You have entered an invalid Waybill number");
```

Figure 1-19. *Code change suggestion*

Here we can see that Visual Studio is suggesting the use of the *var* keyword. At the bottom of the code editor, you will also see that Visual Studio displays the count of errors and warnings as seen in Figure 1-20.

Figure 1-20. *Errors and warnings*

You are able to navigate between the warnings and errors by clicking the left and right arrows. You can also perform a code cleanup by clicking the little brush icon or by holding down Ctrl+K, Ctrl+E.

After cleaning up the code and adding the code suggestions that Visual Studio suggested, the code looks somewhat different as can be seen in Figure 1-21.

```
private void BtnTrack_Click(object sender, EventArgs e)
{
    if (!(string.IsNullOrWhiteSpace(txtWaybill.Text)))
    {
        var waybillNum = txtWaybill.Text;
        if (WaybillValid())
        {
            var package = new Package(waybillNum);
            var packLoc = package.TrackPackage();
            if (packLoc != null)
            {
                txtLocationDetails.Text = $"Package location: " +
                    $"{packLoc.LocationName} with coordinates " +
                    $"Long: {packLoc.Long} and " +
                    $"Lat: {packLoc.Lat}";
            }
        }
        else
        {
            _ = MessageBox.Show("You have entered an invalid Waybill number");
        }
    }
}

private bool WaybillValid() => txtWaybill.Text.ToLower().Contains("acme-");
}
}
```

Figure 1-21. *Code suggestions applied*

With all the code suggestions applied, Visual Studio displays a clean bill of health in the status indicator at the bottom of the code editor.

Navigating Code

Visual Studio provides several features allowing developers to navigate code throughout the solution. Knowing how to use these navigation features will save you a lot of time.

Navigate Forward and Backward Commands

If you look at the toolbar in Visual Studio, you will see the Navigate Forward (Ctrl+Shift+-) and Navigate Backward (Ctrl+-) buttons. These allow developers to return to the last 20 locations that the developer was at.

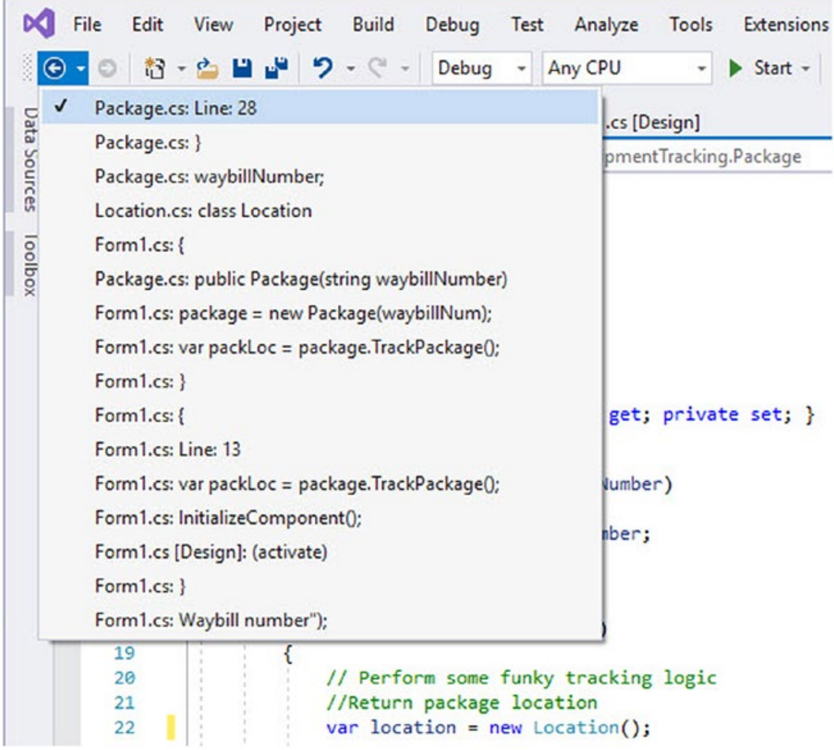


Figure 1-22. *Navigate forward and backward*

You can also find these commands from the View menu under Navigate Backward and Navigate Forward.

Navigation Bar

The navigation bar in Visual Studio as seen in Figure 1-23 provides drop-down boxes that allow you to navigate the code in the code base. You can choose a type or member to jump directly to it in the code editor.

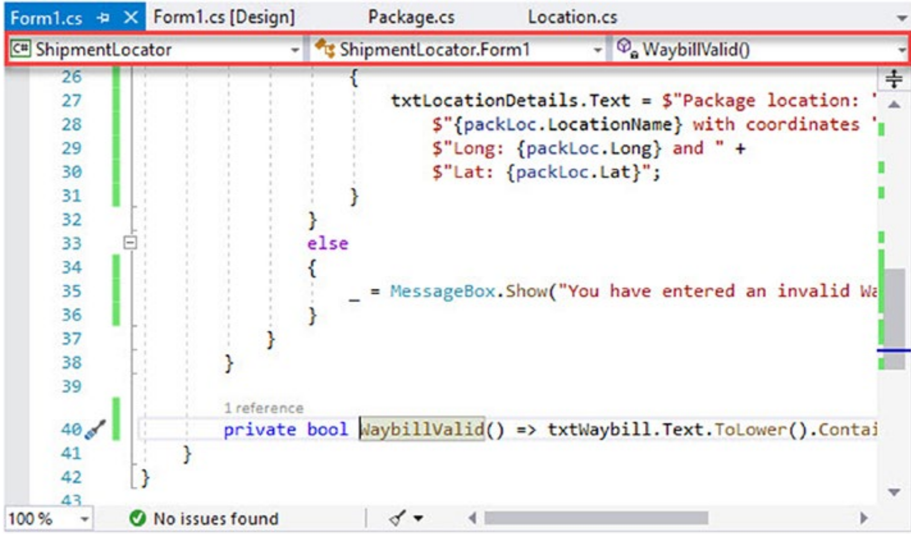


Figure 1-23. *Visual Studio navigation bar*

It is useful to take note that members defined outside the current code file will be displayed but will be disabled and appear gray. You can cycle through the drop-down boxes in the navigation bar by pressing the tab key.

Each drop-down also has its own individual function. The left drop-down will allow you to navigate to another project that the current file belongs to. To change the focus to another class or type, use the middle drop-down to select it. To navigate to a specific procedure or another member in a particular class, select it from the right drop-down.

Find All References

Visual Studio allows you to find all the references for a particular element in your code editor. You can do this by selecting the code element and pressing *Shift+F12* or by right-clicking and selecting *Find All References* from the context menu.

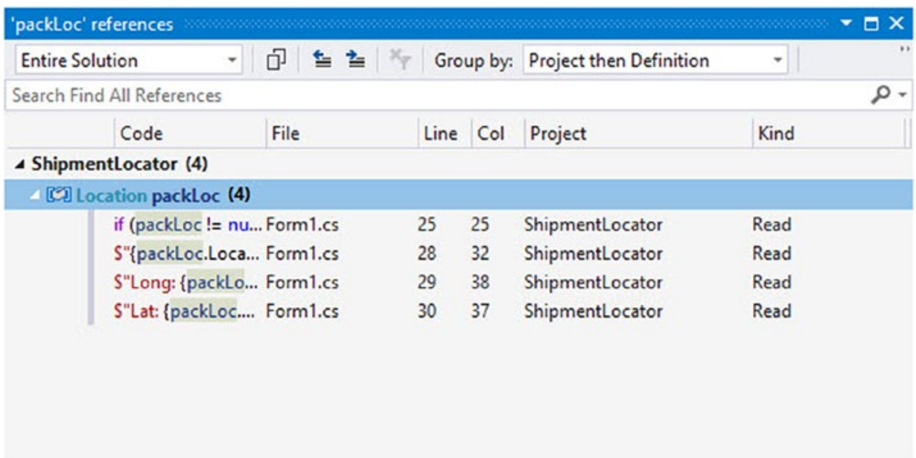


Figure 1-24. *Find All References results*

The find results are displayed in a tool window as seen in Figure 1-24. The toolbar for the find results tool window as seen in Figure 1-25 is also really helpful.

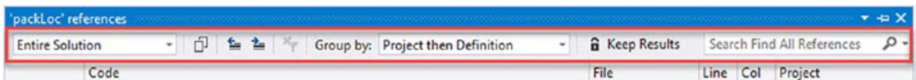


Figure 1-25. *References window toolbar*

From here you can do the following:

- Change the search scope
- Copy the selected referenced item
- Navigate forward or backward in the list
- Clear any applicable search filters (filters are added by hovering over a column in the results window and clicking the filter icon that is displayed)
- Change the grouping of the returned results

- Keep the search results (new searches are opened in a new tool window)
- Search the returned results by entering text in the Search Find All References text box

Hovering your mouse of a returned search result will pop up a preview screen of the code. To navigate to a search result, press the Enter key on a reference or double-click it.

Reference Highlighting

Visual Studio makes it really easy to see selected items in the code editor. If you click a variable, for example, you will see all the occurrences of that variable highlighted in the code editor as seen in Figure 1-26.

Figure 1-26. *Default reference highlighting*

But did you know that you can change the color of the highlight from the Options in Visual Studio? Go to *Tools, Options, Environment, Fonts and Colors, Highlighted Reference* as seen in Figure 1-27.

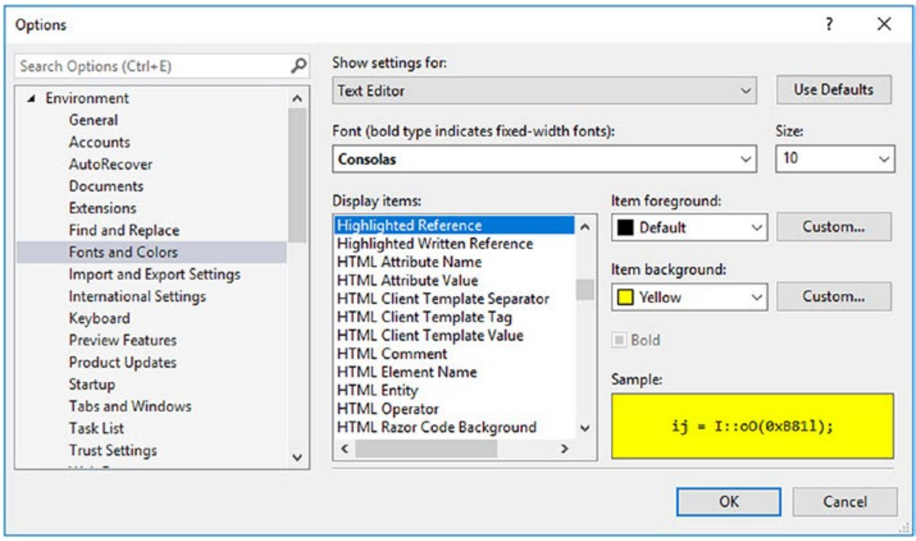


Figure 1-27. *Change the Highlighted Reference color*

Change the color to yellow and click OK. All the references to the variable you just selected will now be highlighted in yellow.

Go To Commands

I'll admit that these are probably the commands that I use the least in Visual Studio. All with the exception of Ctrl+G. Go ahead and open Visual Studio, and type Ctrl+G.

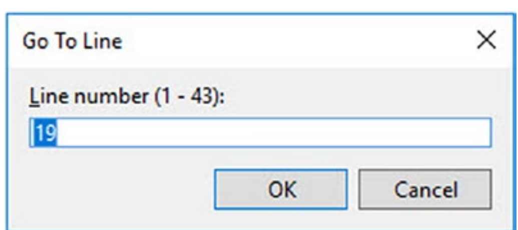


Figure 1-28. *Go To Line*

As can be seen in Figure 1-28, a window pops up that allows you to jump to a specific line of code. This is incredibly useful when trying to explain something to another developer not sitting in the same room as you.

The list of Go To commands are as follows:

- Ctrl+G – Go To Line which allows you to move to the specified line number in the currently active document
- Ctrl+T or Ctrl+, – Go To All which allows you to move to the specified line, type, file, member, or symbol
- Ctrl+1, Ctrl+F – Go To File that allows you to move to a specified file in the solution
- Ctrl+1, Ctrl+R – Go To Recent File allows you to move to a recently visited file in the solution
- Ctrl+1, Ctrl+T – Go To Type allows you to move to a specific type in the solution
- Ctrl+1, Ctrl+M – Go To Member allows you to move to the specific member in the solution
- Ctrl+1, Ctrl+S – Go To Symbol allows you to move to the specific symbol in the solution
- Alt+PgDn – Go To Next Issue in File
- Alt+PgUp – Go To Previous Issue in File
- Ctrl+Shift+Backspace – Go To Last Edit Location

While typing Ctrl+1 might seem slightly finicky, you soon get used to it, and the commands start to feel more natural. Ctrl+Shift+Backspace is another command that I find very useful. This is especially true when editing large code files.

Go To Definition

Go To Definition allows you to jump to the definition of the selected element. Go to the example project for this chapter, and find the click event for the Track button.

Just a reminder that the code for this book can be downloaded from GitHub.

In there you will see that we are working with a class called Package that creates a new package we would like to track.

```
1 reference
private void BtnTrack_Click(object sender, EventArgs e)
{
    if (!(string.IsNullOrWhiteSpace(txtWaybill.Text)))
    {
        var waybillNum = txtWaybill.Text;
        if (WaybillValid())
        {
            var package = new Package(waybillNum);
            var packLoc = package.TrackPackage();
            if (packLoc != null)
            {
                txtLocationDetails.Text = $"Package location: " +
                    $"{packLoc.LocationName} with coordinates " +
                    $"Long: {packLoc.Long} and " +
                    $"Lat: {packLoc.Lat}";
            }
        }
```


Figure 1-29. *Go To Definition*

Place your cursor on Package, and hit F12 to jump to the class definition. You can also hold down the Ctrl button and hover over the class name. You will notice that Package becomes a link you can click on. Lastly, if you have your feet up and you only have your mouse to navigate with (the other hand is holding a cup of coffee), you can right-click and select *Go To Definition* from the context menu.

Peek Definition

Where Go To Definition navigates to the particular definition in question, Peek Definition simply displays the definition of the selected element in a pop-up. Place your cursor on `Package` and right-click. From the context menu, select Peek Definition.

```
16  private void BtnTrack_Click(object sender, EventArgs e)
17  {
18      if (!(string.IsNullOrWhiteSpace(txtWaybill.Text)))
19      {
20          var waybillNum = txtWaybill.Text;
21          if (WaybillValid())
22          {
23              var package = new Package(waybillNum);
                                                        Package.cs
    12
    13      public Package(string waybillNumber)
    14      {
    15          WaybillNumber = waybillNumber;
    16      }
    17
    18      public Location TrackPackage()
    19      {
    20          // Perform some funky tracking logic
    21          //Return package location
    22          var location = new Location();
    23
    24          return location;
    25      }
    26  }
    27 }
24              var packLoc = package.TrackPackage();
25              if (packLoc != null)
26              {
```

Figure 1-30. *Peek Definition pop-up*

As can be seen in Figure 1-30, the pop-up window displays the code for the `Package` class. You can navigate through the code displayed in this pop-up as you would any other code window. You can even use Peek Definition or Go To Definition inside this pop-up.

In the pop-up window, right-click `Location`, and select Peek Definition from the context menu. The second Peek Definition will start a breadcrumb path as seen in Figure 1-31.

```
16        private void BtnTrack_Click(object sender, EventArgs e)
17        {
18            if (!(string.IsNullOrWhiteSpace(txtWaybill.Text)))
19            {
20                var waybillNum = txtWaybill.Text;
21                if (WaybillValid())
22                {
23                    var package = new Package(waybillNum);
                                                                  Location.cs
 8    {
 9        public class Location                                   Location.cs
10        {
11            public string Long { get; set; } = "080º 37' 49.5\" W";
12            public string Lat { get; set; } = "05º 13' 01.3\" S";
13            public string LocationName { get; set; } = "Secret Batcave";
14        }
15    }
16    breadcrumb
24                    var packLoc = package.TrackPackage();
25                    if (packLoc != null)
26                    {
```

Figure 1-31. *Breadcrumb path*

You can now navigate using the circles and arrows that appear above the Peek Definition pop-up window. The arrows only appear when you hover your mouse over the circles, but this definitely makes it much easier to move between the code windows.

Features and Productivity Tips

Visual Studio is full of existing productivity tips that have been around for years and that some developers do not know about. In this section, we will be looking at some of those.

Track Active Item in Solution Explorer

In Visual Studio, this option is not on by default. As you change the code file you are working in, the file isn't highlighted in the Solution Explorer. You will know this is the case when you see the following arrows in the toolbar of the Solution Explorer as seen in Figure 1-32.

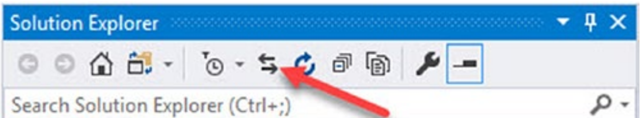


Figure 1-32. *Track Active Item in Solution Explorer*

Clicking these arrows will highlight the file that you are currently editing in the Solution Explorer. I find this extremely useful, so I let Visual Studio permanently track the current file. To set this, click Tools, Options, Projects and Solutions, General, and check Track Active Item in Solution Explorer.

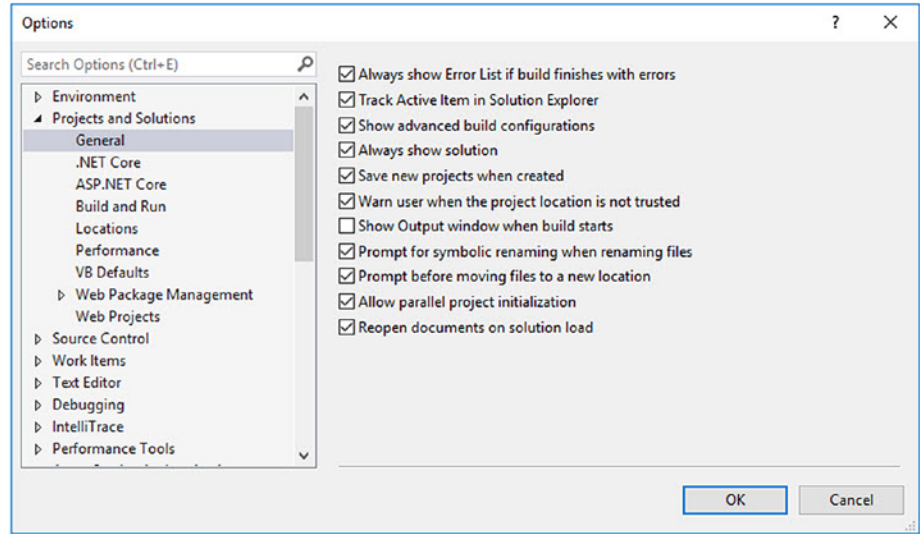


Figure 1-33. *Track Active Item Setting*

With this setting enabled, the arrows are not displayed in the toolbar of the Solution Explorer.

Hidden Editor Context Menu

When you are in a code file, you can access a variety of menu items by right-clicking and selecting the menu items in the context menu. But did you know that you can hold down Alt+` to bring up a special context menu (that is different from right-click)? It has more editor commands in it as seen in Figure 1-34.

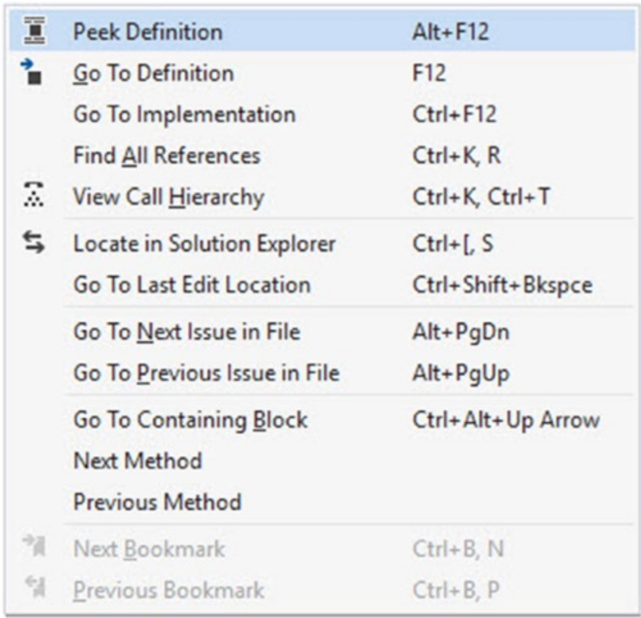


Figure 1-34. *Special context menu*

This gives you a little more control over navigating through errors, methods, etc., in your current code file.

Open in File Explorer

Sometimes you need to quickly get to the actual Visual Studio files of your solution. This might be to go and copy a file from the bin folder or to open a file in another text editor. To do this, you don't even need to leave Visual Studio. As seen in Figure 1-35, you can right-click the solution and click Open Folder in File Explorer.

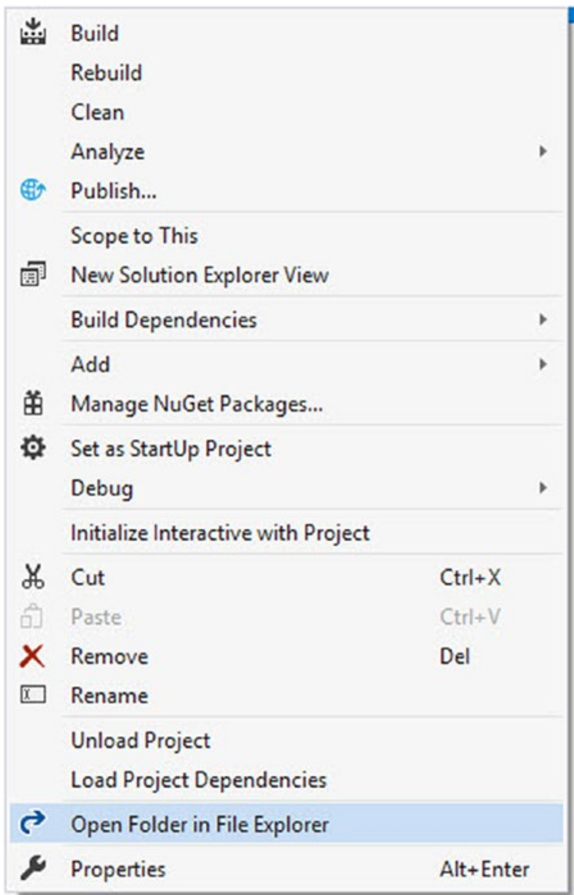


Figure 1-35. *Open Folder in File Explorer*

This will open a new File Explorer window where your Visual Studio solution is located.

Finding Keyboard Shortcut Mappings

Sometimes when you use a keyboard shortcut, and nothing happens, you might be using it in the wrong context. To see what keyboard shortcuts are mapped to, head on over to Tools, Options, Environment, Keyboard as seen in Figure 1-36.

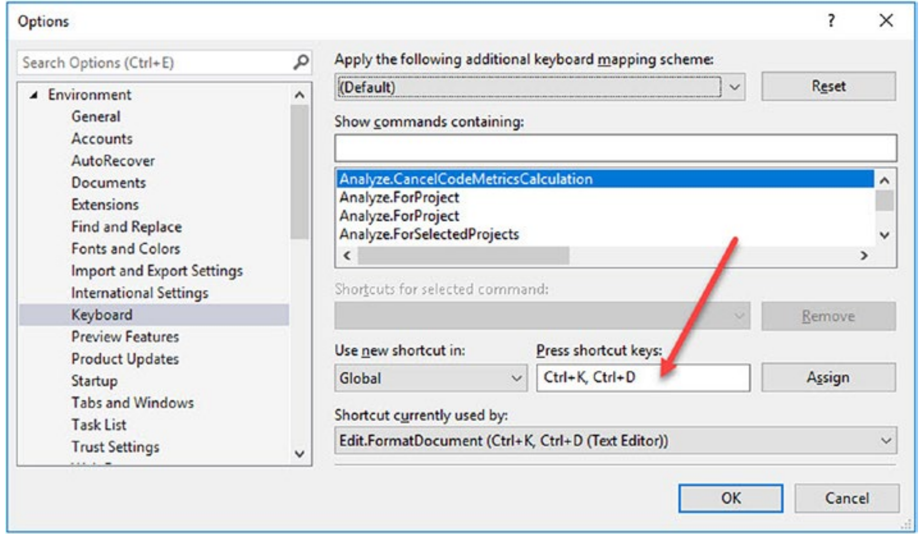


Figure 1-36. *Find keyboard shortcut mappings*

Press the shortcut keys, and Visual Studio will show you what the shortcut is currently used for. This is also very useful for assigning new keyboard shortcuts to check that the keyboard shortcut you have in mind is not already bound to another command.

Clipboard History

Visual Studio allows you to access your clipboard history. This is very useful if you have to copy and paste several items repeatedly.

Instead of going back and forth between copy and paste, simply hold down Ctrl+Shift+V to bring up the clipboard history as seen in Figure 1-37.

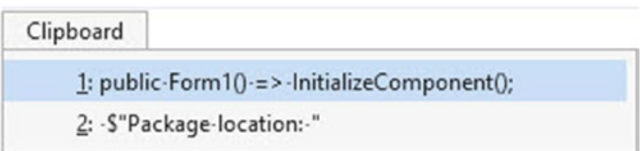


Figure 1-37. *Clipboard history*

Now you can just select the copied text that you want to paste and carry on with editing your code. The copied item also remains in the clipboard history after pasting.

Go To Window

So this could actually have gone under the Navigating Code section, but I wanted to add it here because it made more sense to discuss it as a productivity tip.

Hold down Ctrl+T and you will see the Go To window pop up. Now type a question mark, and see the options available to you as seen in Figure 1-38.

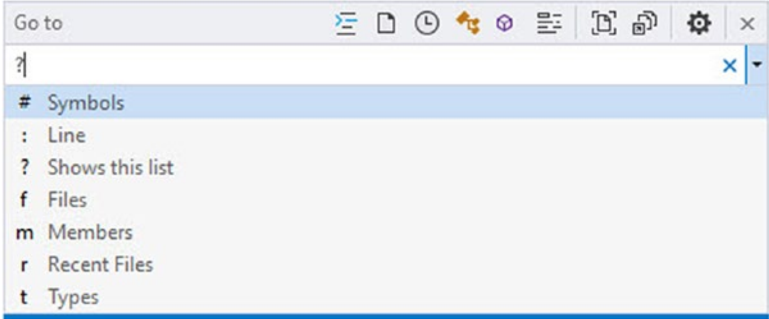


Figure 1-38. *Go To Window*

You can view the recent files by typing in an *r* instead of a question mark. Also nice to note is the ability to jump to a specific line of code. You will remember earlier in the chapter that we discussed the Go To commands and Ctrl+G in particular. Here, you can do the same thing by typing in : followed by the line number.

Navigate to Last Edit Location

Earlier on in this chapter, we discussed the Navigate Backward and Navigate Forward commands. This is great, but if you want to navigate to the last place you made an edit in the code file, hold down Ctrl+Shift+Backspace. This will jump to the last place that you made an edit in one of your code files.

Multi-caret Editing

This is a feature that I absolutely love using. Consider the following SQL Create Table Statement:

Listing 1-2. Create SQL Table Statement

```
CREATE TABLE [dbo].[menu](
        [itemName] [varchar](50) NOT NULL,
        [category] [varchar](50) NOT NULL,
        [description] [varchar](50) NOT NULL,
 CONSTRAINT [PK_menu] PRIMARY KEY CLUSTERED
```

This is a rather small table, but sometimes we have very large tables that we need to work with. I want to create a simple C# class for this table and need to create some C# properties. Why type out everything when you can copy, paste, and edit all at once.

Paste the column names into a C# class file, and then hold down Ctrl+Alt, and click in front of the first square bracket of each column as can be seen in Figure 1-39.

```
public string itemName] [varchar] (50) NOT NULL,
public string category] [varchar] (50) NOT NULL,
public string description] [varchar] (50) NOT NULL,
```

Multi-caret

Figure 1-39. *Multi-caret selection*

The cursor is placed at each line at the position you placed it. Now start typing the property definition. All the lines are edited. After typing `public string`, hit delete to remove the first square bracket.

I now want to add the `{ get; set; }` portion of my property. I can either do the same multi-caret selection or I can also select one or more characters and hold down Shift+Alt+. to select matching selections as seen in Figure 1-40.

```
public string itemName] [varchar] (50) NOT NULL,
public string category] [varchar] (50) NOT NULL,
public string description] [varchar] (50) NOT NULL,
```

Select matching

Figure 1-40. *Selecting matching selections*

This now allows me to easily select exactly all the lines I want to edit at the same time and allows me to paste the `{ get; set; }` needed for my properties. I now end up with the completed code as seen in Figure 1-41.

```
0 references
public string itemName { get; set; } //varchar] (50) NOT NULL,
0 references
public string category { get; set; } //varchar] (50) NOT NULL,
0 references
public string description { get; set; } //varchar] (50) NOT NULL,
```

Figure 1-41. *Completed code properties*

Being able to easily select code or place a caret in several places on the same line or across lines allows developers to be really flexible when editing code. Speaking about placing the caret on several places on the same line, it is, therefore, possible to do the following as seen in Figure 1-42.

```
0 references
static void Main(string[] args)
{
    var a = "The dog is lazy but the dog is awake";
    var b = "The dog is lazy but the dog is awake";
    var c = "The dog is lazy but the dog is awake";
}
```

Figure 1-42. *Multi-caret selection on the same line*

We can now edit everything at once (even if we have selected multiple places on the same line) as seen in Figure 1-43.

```
0 references
static void Main(string[] args)
{
    var a = "The cat is lazy but the cat is awake";
    var b = "The cat is lazy but the cat is awake";
    var c = "The cat is lazy but the cat is awake";
}
```

Figure 1-43. *Multi-caret editing on the same line*

Holding down Ctrl+Z will also work to undo everything at once. If you want to insert carets at all matching selections, you can select some text and hold down Shift+Alt+; to select everything that matches your current selection as seen in Figure 1-44.

```
0 references
public string itemName { get; set; } //varchar] (50) NOT NULL,
0 references
public string category { get; set; } //varchar] (50) NOT NULL,
0 references
public string description { get; set; } //varchar] (50) NOT NULL,

0 references
static void Main(string[] args)
{
    var a = "The cat is lazy but the cat is awake";
    var b = "The cat is lazy but the cat is awake";
    var c = "The cat is lazy but the cat is awake";
}
```

Figure 1-44. *Insert carets at all matching selections*

I selected the text “cat” and held down Shift+Alt+; and Visual Studio selected everything that matches. As you can see, it also selected the category property, which I don’t want to be selected. In this instance, Shift+Alt+. will allow me to be more specific in my selection.

If you find yourself forgetting the keyboard shortcuts, you can find them under the Edit menu. Click Edit, Multiple Carets to see the keyboard shortcuts.

Features in Visual Studio 2019

Visual Studio 2019 comes packed with a few very nice productivity features. A lot of thought has been put into making Visual Studio easy to navigate and to find things in Visual Studio 2019. The first feature I want to have a look at is Visual Studio Search.

Visual Studio Search

I think that we can all agree that more speed equals improved productivity. The faster I can access a menu item, and the less time I have to spend looking for something, the more my productivity increases. This is where Visual Studio Search comes in.

If you hold down Ctrl+Q you will jump to the search bar where you can immediately start typing as seen in Figure 1-45.

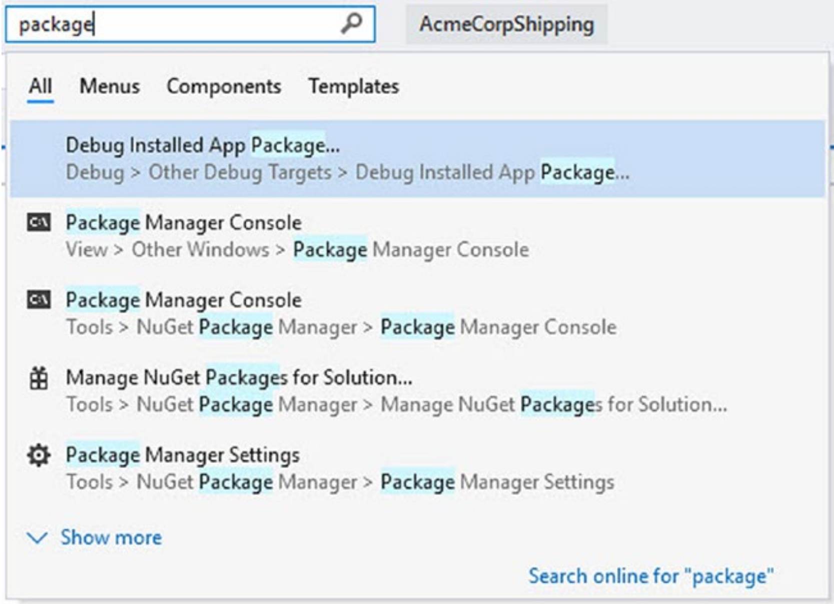


Figure 1-45. *Visual Studio Search*

Visual Studio will perform the required search and display the results to you that you can further filter by clicking the Menus, Components, or Templates tabs. Visual Studio performs a fuzzy search, which means that even if you misspell a word, chances are that Visual Studio will know what you intended to type and return the correct results for you.

Solution Filters

Sometimes we have to work on Solutions that contain a lot of projects. More often than not, you as a developer will not be working on every project in that specific solution. This is where Solution Filters come in. They allow you to only load the projects that you care about or are actively working on. Consider the currently loaded solution as seen in Figure 1-46.

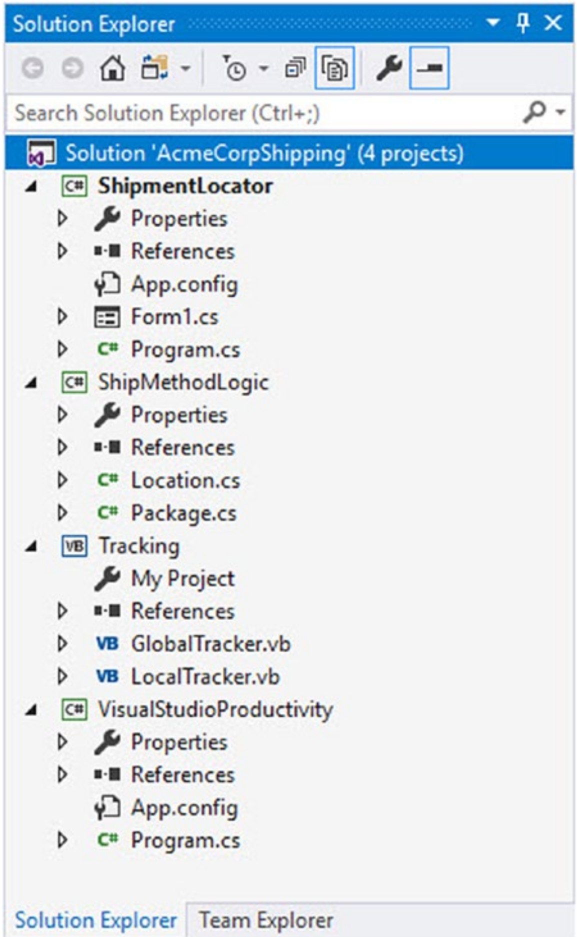


Figure 1-46. *AcmeCorpShipping Solution Unfiltered*

You can see that all the projects are loaded in this solution. If we only work on the ShipmentLocator and ShipMethodLogic projects, we can create a Solution Filter to only load those projects. Right-click the projects that you don't work on, and click Unload Project from the context menu. Your solution will now look as in Figure 1-47.

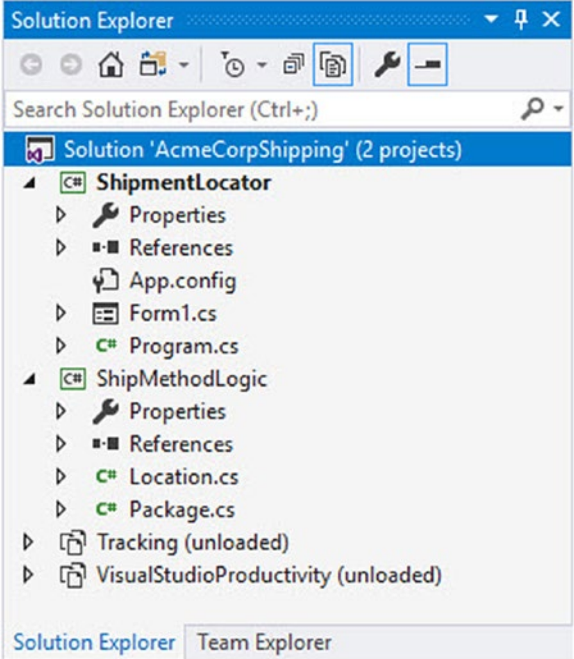


Figure 1-47. *AcmeCorpShipping Solution with unloaded projects*

Now, with the projects unloaded that you do not work on, right-click the solution and select Save As Solution Filter as seen in Figure 1-48.

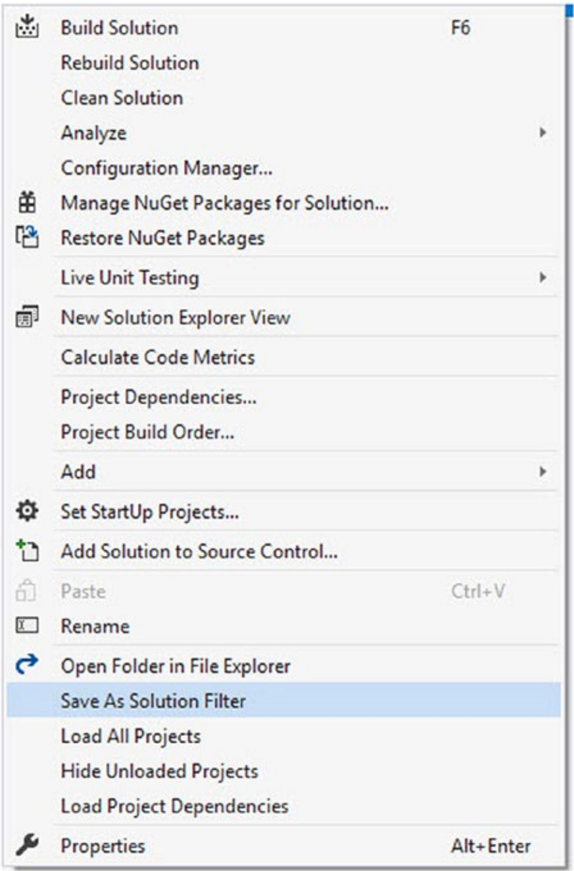


Figure 1-48. *Save As Solution Filter*

Visual Studio will now create a new type of solution file with an .slnf file extension as seen in Figure 1-49.

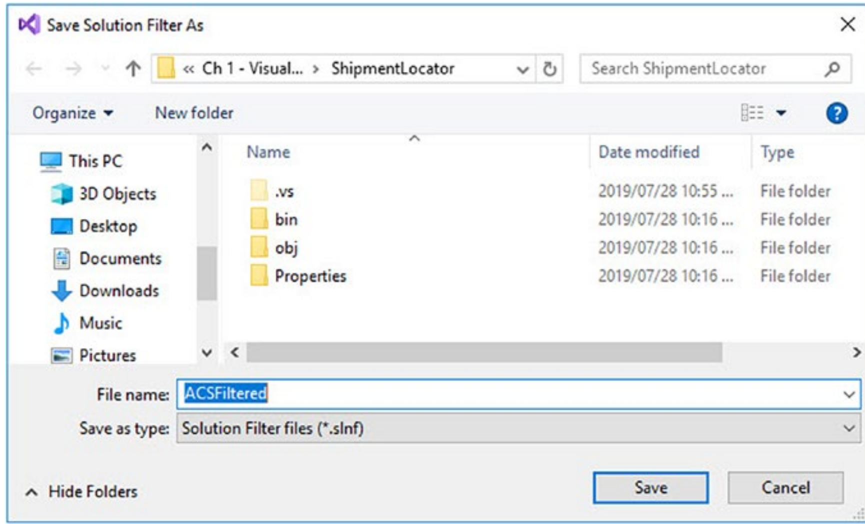


Figure 1-49. *Save as Solution Filter file*

When you open your project using the Solution Filter, you will see as in Figure 1-50 that only the projects you selected to keep will be loaded.

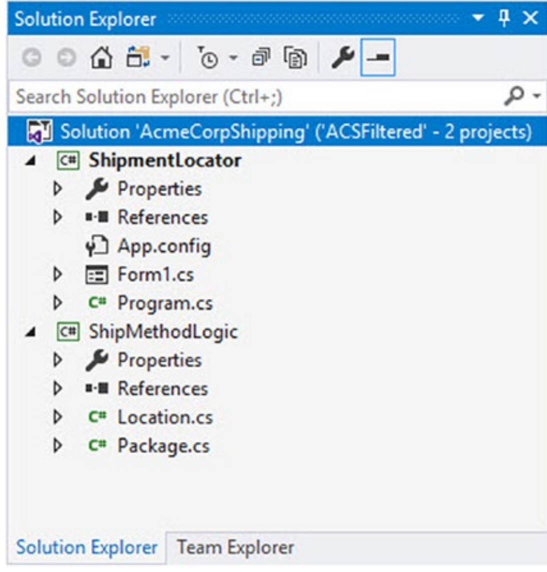


Figure 1-50. *Filtered solution*

Now with the filtered solution, if you click the solution, you will see that you can Load App Projects, Show Unloaded Projects, or Load Project Dependencies from the context menu as seen in Figure 1-51.

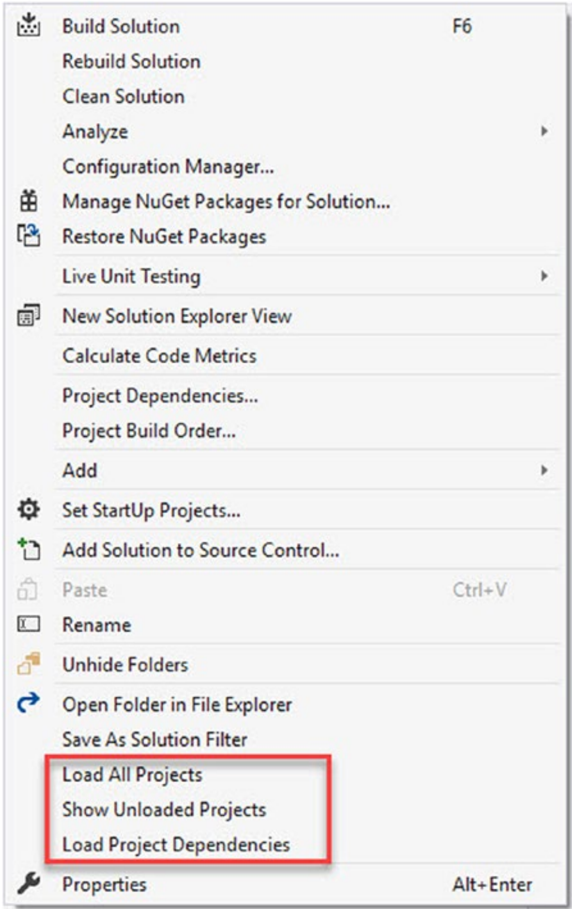


Figure 1-51. *Filtered solution context menu*

You still have full control of the filtered solution from the context menu and can easily load the full solution as needed.

Visual Studio IntelliCode

Visual Studio IntelliCode is a really nice feature that has been added to Visual Studio. Microsoft calls it AI-assisted development because it uses machine learning to figure out what you are most likely to use next and putting that suggestion at the top of your completion list. These are usually displayed as starred recommendations.

Without IntelliCode, when you dot on an object such as our `package` object in the AcmeCorpShipping solution, you will see the usual completion list that Visual Studio pops up as seen in Figure 1-52.

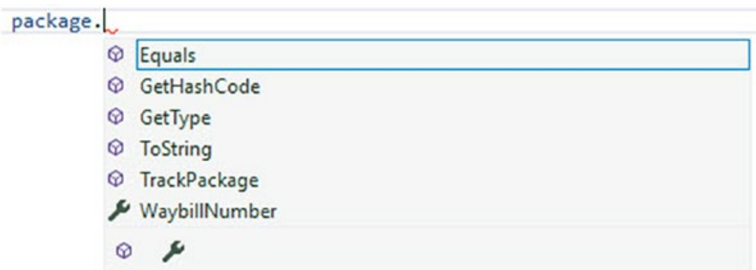


Figure 1-52. *Visual Studio completion list*

In Visual Studio 2019, hit Ctrl+Q and type IntelliCode to search for IntelliCode. Select IntelliCode Model Management from the search results. You can also go to View, Other Windows and click IntelliCode Model Management. You will then see the Visual Studio IntelliCode window pop-up inside Visual Studio as seen in Figure 1-53.

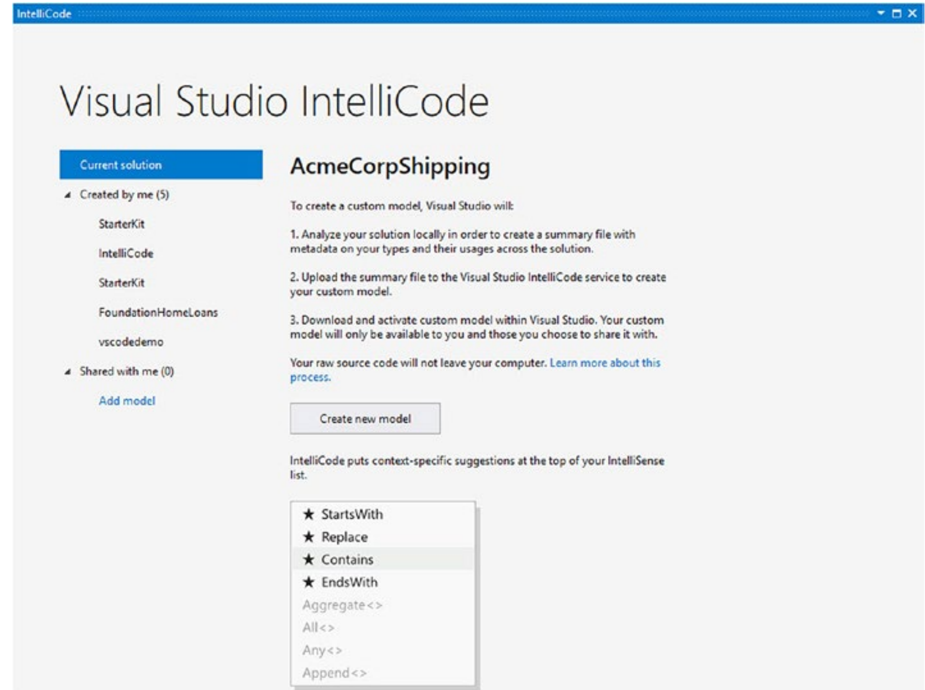


Figure 1-53. *Visual Studio IntelliCode*

Click Create new model to allow Visual Studio IntelliCode to analyze your code and create the model. When Visual Studio has completed the analysis and created the model, you will be able to share the model with other developers (which is great for teams), delete the model, or retrain the model as seen in Figure 1-54.

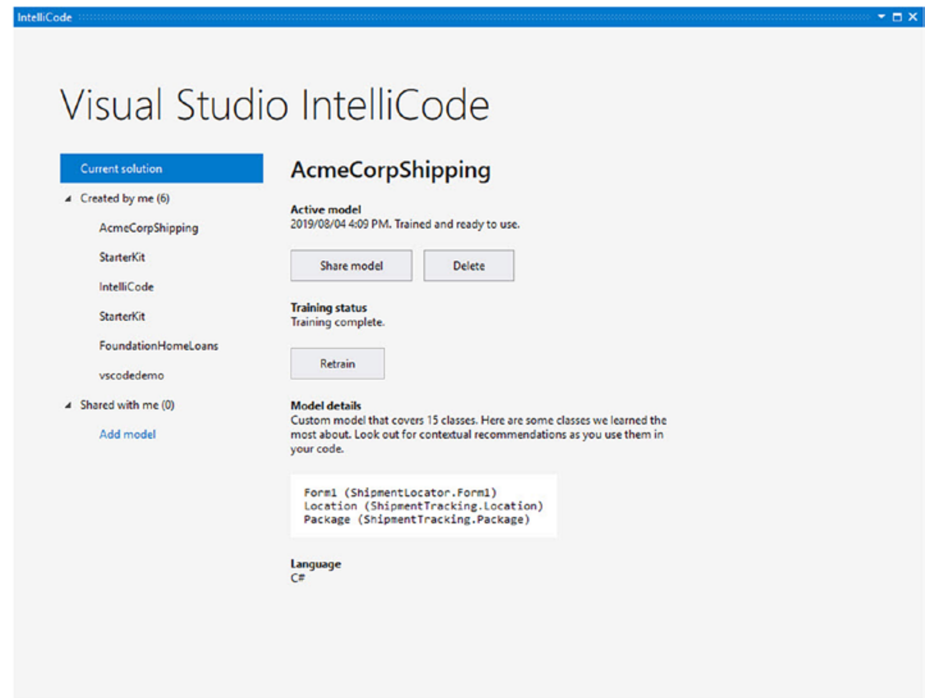


***Figure 1-54.** Visual Studio IntelliCode model completed*

This time around, if you dot on the package in the AcmeCorpShipping solution, you will see the starred recommendations from IntelliCode as seen in Figure 1-55.

```
package.
★ TrackPackage
Equals
GetHashCode
GetType
ToString
TrackPackage
WaybillNumber
```

***Figure 1-55.** IntelliCode starred recommendations*

Compare this to the Visual Studio completion list in Figure 1-52. You can see that IntelliCode has identified the `TrackPackage` method as the most likely method that you will want to use.

IntelliCode uses open source GitHub projects with 100 stars or more to distill the wisdom of the community in order to generate recommendations for your code. To demonstrate how clever Visual Studio IntelliCode is, have a look at Figure 1-56.

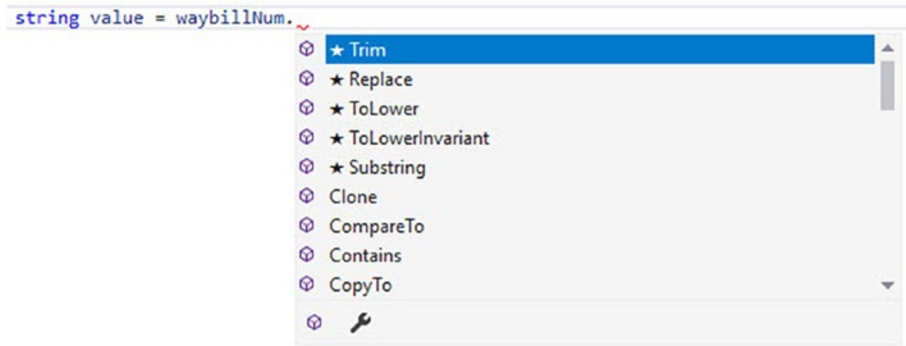


Figure 1-56. *IntelliCode acting on a string*

IntelliCode knows the most likely things I would want to do with a string variable and places those methods at the top of the suggestion list. The suggestions look quite a bit different when I am working with a string array as seen in Figure 1-57.

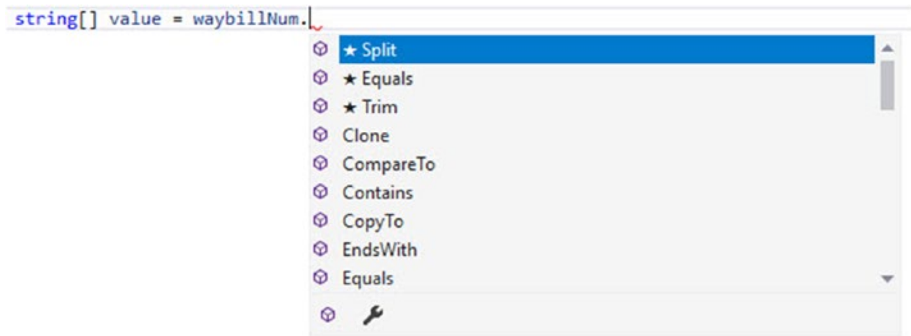


Figure 1-57. *IntelliCode acting on a string array*

This means that IntelliCode takes the current context into account when suggesting methods in the completion list. If you would like to view the model generated by IntelliCode, you can head on over to `%TEMP%\Visual Studio IntelliCode`. In one of the created folders, you will find a subfolder called "UsageOutput." Look for a JSON file in the "UseageOutput" folder. This is where the contents of the extracted data is stored for your model.

It is important to note that Microsoft does not receive any of your code. IntelliCode only uploads data and information about your code to Microsoft's servers. All your code remains on your computer.

IntelliCode is a productivity feature that will definitely benefit developers on a day-to-day basis.

Visual Studio Live Share

During my years of writing code, I have often had the need to explain some portion of logic or feature of the code I am working with, to another developer. This usually involves them having to get a copy of the code base from source control and us having to direct each other over a Skype call and quote line numbers in order to collaborate properly.

To find out more about Visual Studio Live Share, or to download a copy for Visual Studio Code or Visual Studio 2017, go to `https://visualstudio.microsoft.com/services/live-share/`.

Visual Studio Live Share is included by default in Visual Studio 2019. Visual Studio Live Share does not require developers to be all "set up" in order to assist each other or to collaborate on projects. This means that a

developer running Visual Studio Code on a Linux machine can collaborate with another developer running Visual Studio 2019 on a Windows 10 machine.

To start a Visual Studio Live Share session, you need to click the Live Share icon in the top right corner of Visual Studio 2019 as can be seen in Figure 1-58.

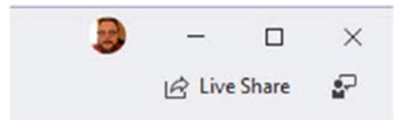


Figure 1-58. *Visual Studio Live Share Icon*

When you click the icon, Visual Studio starts up Live Share, and the progress is indicated as can be seen in Figure 1-59.

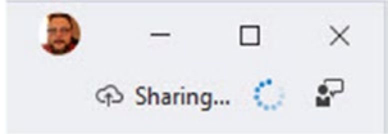


Figure 1-59. *Visual Studio Live Share In Progress*

When the sharing link has been generated, Visual Studio will display a notification as seen in Figure 1-60.

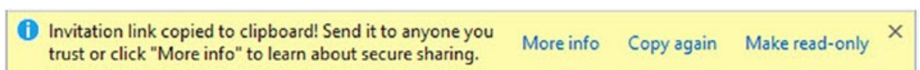


Figure 1-60. *Visual Studio Live Share Link Generated*

It is copied to the clipboard by default, but you can copy it again, make it read-only, or learn more about secure sharing. When you share the link with a fellow developer, they simply have to paste the link into a browser to start the collaboration. Figure 1-61 shows the browser after pasting the link.

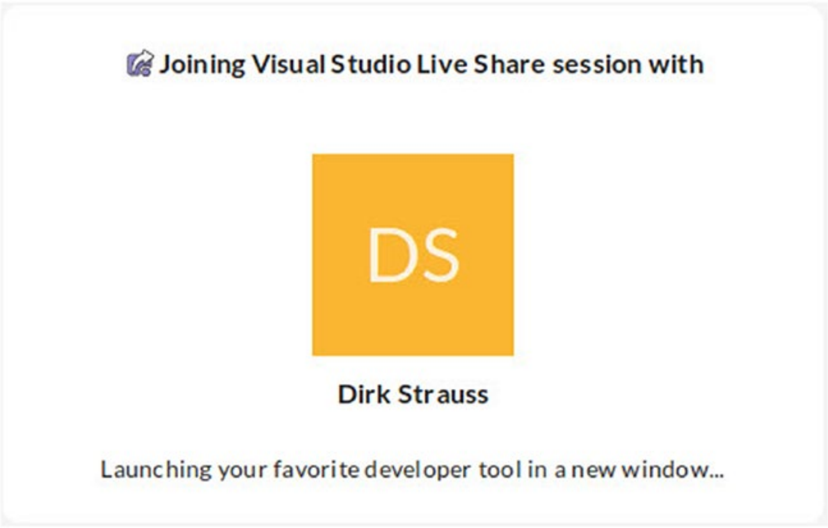


Figure 1-61. *Starting Visual Studio Live Share session*

In this example, I am sharing the link with a developer that is running Visual Studio Code on Linux Mint. Linux then pops up a Launch Application notification as seen in Figure 1-62.

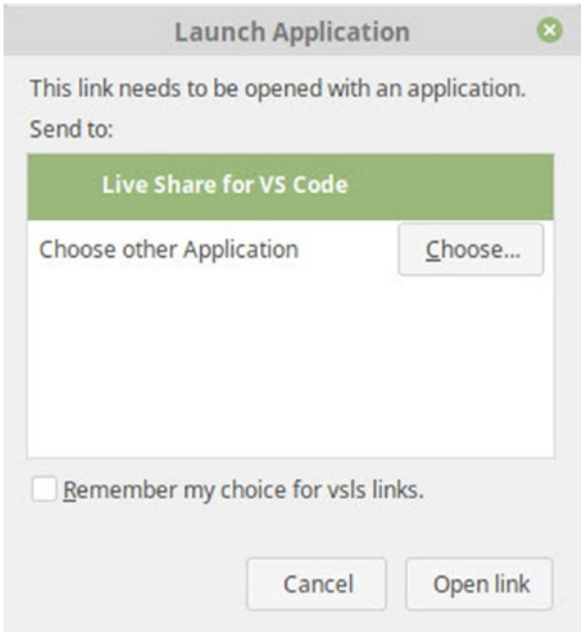


Figure 1-62. *Launch Application Notification on Linux*

Visual Studio Code is already installed on the Linux machine; therefore, the Launch Application notification offers that as the default choice for opening vsls links. When you click the Open Link button, Visual Studio Code launches, and your Live Share session is initiated as seen in Figure 1-63. Visual Studio Code then has a copy of the code that I am sharing on my machine.

Figure 1-63. *Visual Studio Live Share session*

As Jason navigates his way around the code, I can see this in my code via a marker that pops up momentarily displaying his name as seen in Figure 1-64.

Figure 1-64. *I can see Jason's current position in the code*

Over in Visual Studio Code, Jason can see where I am via a similar marker that momentarily pops up my name as seen in Figure 1-65.

```
var package = new Package(waybillNum);
var packLoc = package.TrackPackage();
if (packLoc != null)
{
Dirk Strauss ationDetails.Text = $"Package location: " +
        $"{packLoc.LocationName} with coordinates " +
        $"Long: {packLoc.Long} and " +
        $"Lat: {packLoc.Lat}";
}
```

***Figure 1-65.** Jason can see my current position in the code*

This allows us to know what the other is doing and where we are working at any given time. In Visual Studio 2019, I now also have a new Live Share tab displayed as seen in Figure 1-66.

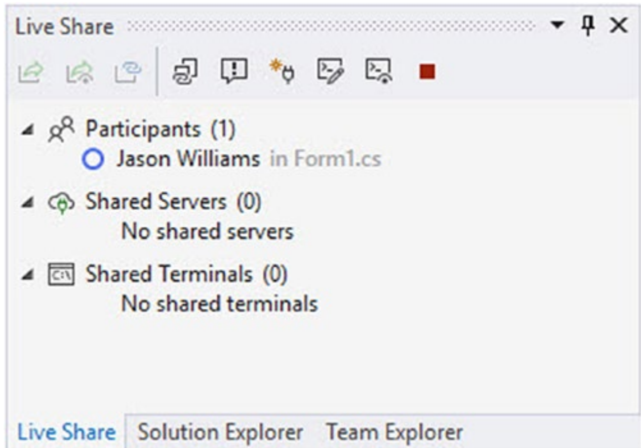


***Figure 1-66.** Live Share tab in Visual Studio 2019*

From there, I can end the Live Share session, share the terminal, manage shared servers, focus participants, or copy the sharing link again. At any time, I am in total control of what I share. It is also important to note that my code lives on my machine. It is not saved on the participant's machine.

CHAPTER 2

Working with Visual Studio 2019

If you have been working with previous versions of Visual Studio, you will find that Visual Studio 2019 definitely does not break the mold. What I mean to say is that Visual Studio 2019 feels much the same as previous versions, and that's a good thing.

While there are new features and enhancements in Visual Studio 2019, developers will find it really easy to work with from the start. If, however, you are new to Visual Studio, there are a few topics that deserve a closer look. This is what Chapter 2 is all about. We will be taking a look at the following:

- Visual Studio project types and when to use them
- Managing NuGet packages
- Creating project templates
- Creating and using code snippets
- Using bookmarks and code shortcuts
- The Server Explorer window
- Visual Studio Windows

© Dirk Strauss 2020
D. Strauss, *Getting Started with Visual Studio 2019*,
https://doi.org/10.1007/978-1-4842-5449-3_2

Chapter 2 is actually an extension of Chapter 1 in many respects. Things that didn't make it into Chapter 1 are being discussed in Chapter 2. I do believe, however, that these are essential to working with Visual Studio and will benefit developers in their day-to-day coding.

Visual Studio Project Types

Visual Studio 2019 allows developers to create a new project in a couple of ways. The most obvious is when you start Visual Studio 2019. You will be presented with the Start screen as seen in Figure 2-1.

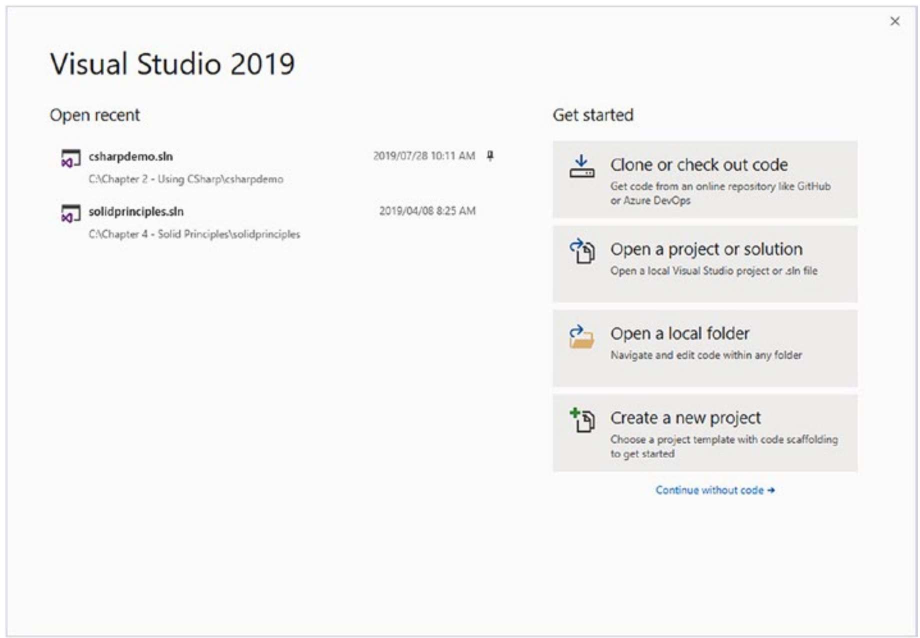


Figure 2-1. *Visual Studio 2019 Start screen*

Here you will see recent projects that you can pin to the Start screen to always keep them available. If you right-click any of the recent projects, you will see a context menu pop up as shown in Figure 2-2.

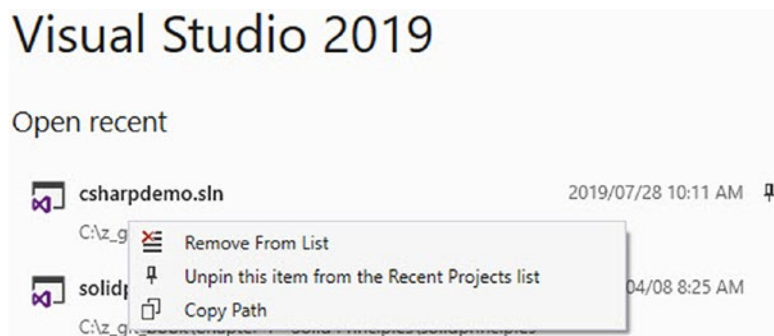


Figure 2-2. *Context menu options on recent projects*

You can remove the project from the list, pin or unpin it from the list, or copy the path to the project. Visual Studio 2019 has also made it quite easy to get to where you need to when working with projects. As seen in Figure 2-3, developers have a few options available to them when they are ready to start working with code.

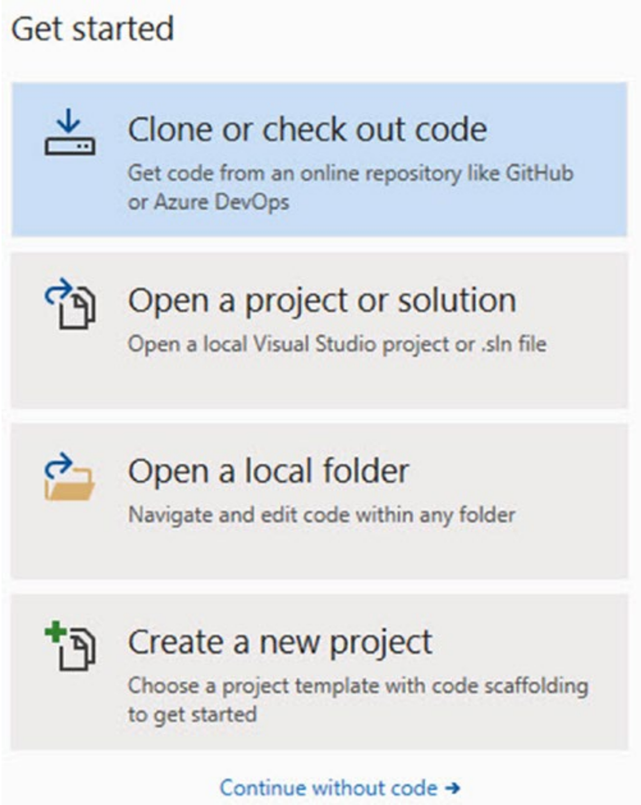


Figure 2-3. *Get started section*

Here you can start by grabbing code from GitHub or Azure DevOps, open a local Visual Studio project, open a local folder to edit code files, create a new project, or continue without code.

If Visual Studio is already open, you can create a new project from the menu bar by clicking the New Project button as seen in Figure 2-4. You can also hold down Ctrl+Shift+N.

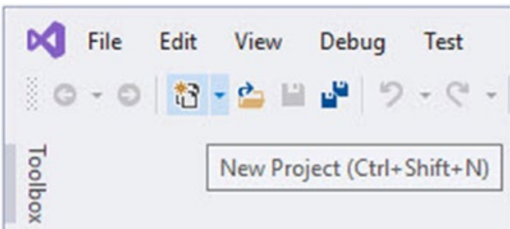


Figure 2-4. *New Project Toolbar button*

The Create a new project screen is displayed as seen in Figure 2-5, and you have a whole new experience here too when it comes to finding the project type that you want to create.

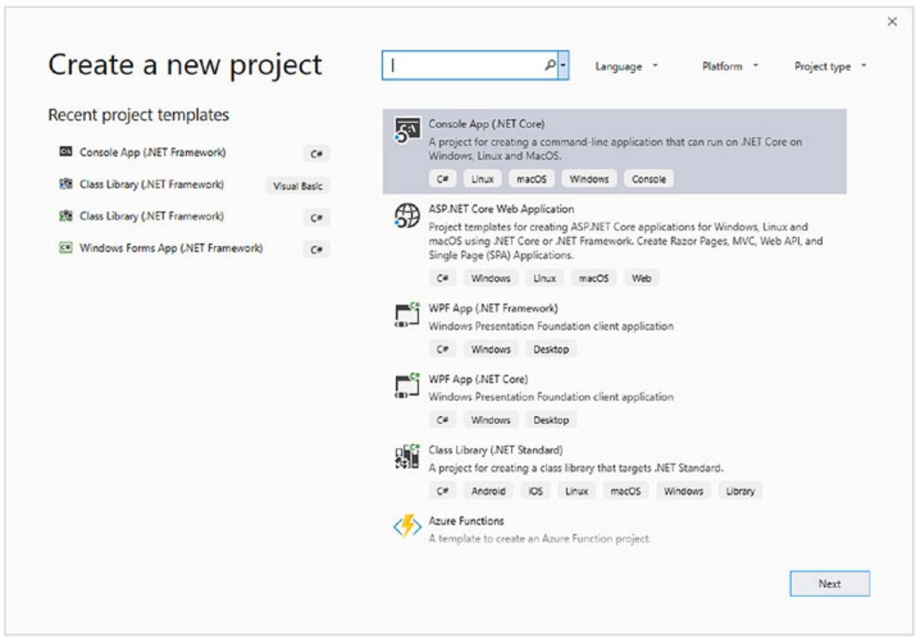


Figure 2-5. *Create a new project*

You will see recent project templates displayed which is great should you need to get up and running with similar projects as what you have created before. You can also search for and filter project templates by language, platform, or project type as seen in Figure 2-6.

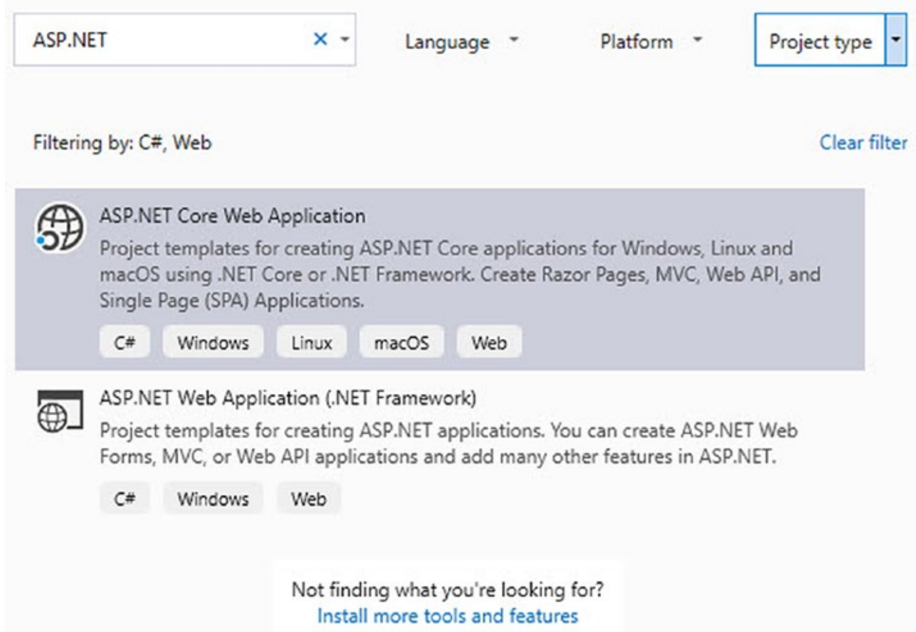


Figure 2-6. *Filter project templates*

This allows you to quickly find what you are looking for.

Take note, though, that if you do not find what you are looking for, you may need to install a workload. To do this, click Install more tools and features. Refer to Chapter 1 to see how to use workloads in Visual Studio.

There are several project templates that you can choose from. Let's see which ones there are and what project is suitable for specific situations.

Various Project Templates

Visual Studio 2019 has a whole host of project templates to choose from. I would even go as far as to say that it's now even easier to find the template you need to use due to the filters in the Create a new project window. Let's have a look at a few of these project templates net.

Console Applications

I remember that the first time that I wrote a single line of code was in a Console Application. This is a great template to use when you don't need a UI for your application. The Console Application project template running on the .NET Framework can be seen in Figure 2-7.

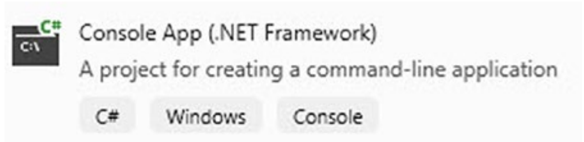


Figure 2-7. *Console App (.NET Framework)*

You will notice that this application is suited for running on Windows machines. But what if you need to run the Console Application across platforms such as Windows, Linux, and macOS? This is where .NET Core comes into play.

The Console Application project template running on .NET Core can be seen in Figure 2-8.

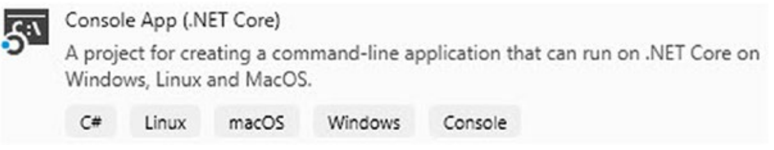


Figure 2-8. *Console App (.NET Core)*

A few years ago (long before .NET Core was ever a thing), I needed to create an application that could be triggered on a schedule. The application's executable would then be passed one of the several parameters that the application used to determine which database to connect to.

The application had to run without any user intervention to perform some sort of maintenance task. Due to the fact that no user intervention was needed, a Console Application best suited the use case. Be aware that a Console Application can accept user input, but for my purposes with this application, it was not necessary.

Windows Forms Application

In contrast to the Console Application, the Windows Forms application template is used when you need to create an app that has a UI. The project template (like the Console Application) can run on the .NET Framework or on .NET Core as seen in Figure 2-9.

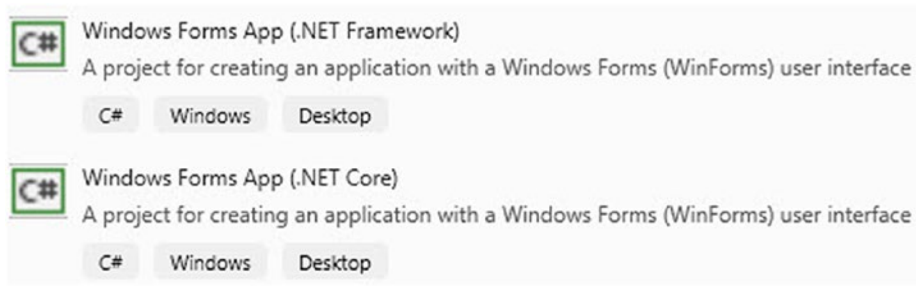


Figure 2-9. *Windows Forms App (.NET Core and .NET Framework)*

It is important to note that Windows Forms apps built on .NET Core will still only be able to run on Windows and is not able to run on Linux or macOS.

This means that WPF and Windows Forms apps built on .NET Core will only run on Windows. There are no plans to make these application types cross-platform.

Why then use .NET Core for Windows Forms applications? Well, .NET Core applications are very, very fast. So it is well worth thinking about using .NET Core for Windows Forms applications.

It is possible to port Windows Forms applications to .NET Core 3.0. You will need to run a tool called the .NET Portability Analyzer to check if your application uses any APIs not currently supported in .NET Core. If it does, you will need to refactor your code to avoid those unsupported dependencies.

For a detailed step-by-step, refer to the Microsoft Developer channel on YouTube, and look for the video "How to Port Desktop Applications to .NET Core 3.0". At the time of writing this book, the URL to this video was `www.youtube.com/watch?v=upVQEUc_KwU`.

Scott Hunter and Olia Gavrysh discuss this topic at the 7:56 minute mark.

Windows Service

If you ever need to create a Windows application that continually runs in the background, performing some specific task, your best choice would be to use a Windows Service template as seen in Figure 2-10.

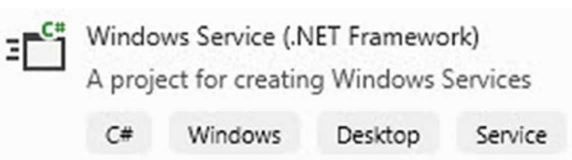


Figure 2-10. *Windows Service (.NET Framework)*

Imagine that the application needs to monitor specific activity (be that in a database or file system) and then write messages to an event log. A Windows Service is perfectly suited for this purpose.

Windows Services have an `OnStart` method that allows you to define what needs to happen when the service starts. By definition, Windows Services are long-running applications that need to poll or monitor the system it runs on. To enable the polling functionality, you will need to use a Timer component.

A common mistake is to use a Windows Forms Timer for a Windows Service. You must ensure that you use the timer in the `System.Timers.Timer` namespace instead.

The `System.Timers` timer (Figure 2-11) that you add to the Windows Service will raise an `Elapsed` event at specific intervals.

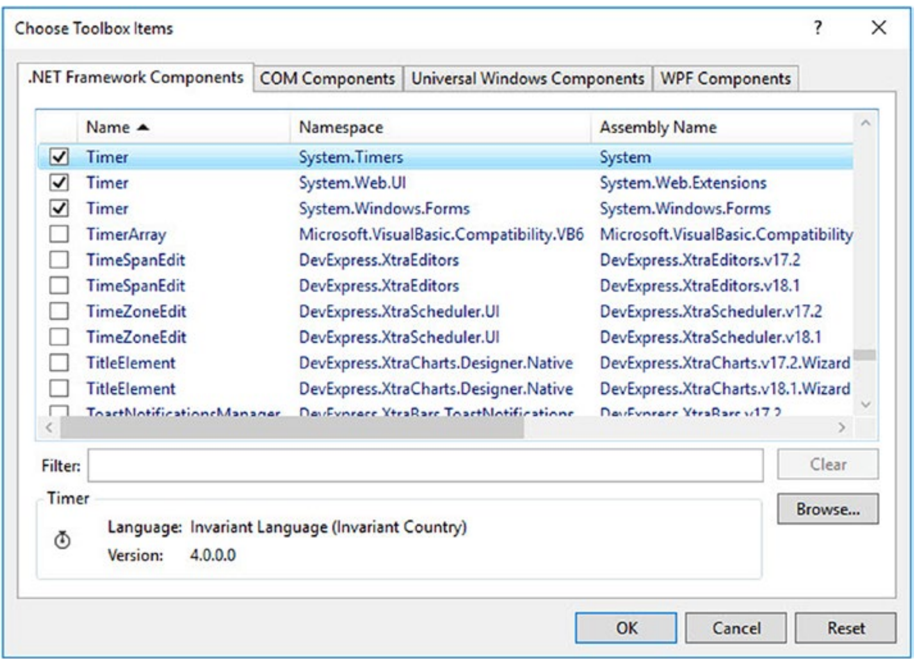


***Figure 2-11.** Various Timer Namespaces*

It is in this `Elapsed` event that you can write the code that your service needs to run in order to do what it needs to do.

Web Applications

If you need to create applications that are web-based, you will definitely be creating an ASP.NET Web Application. If you have a look at the project templates, you will notice that you can create an ASP.NET Web Application that runs on .NET Core as seen in Figure 2-12.

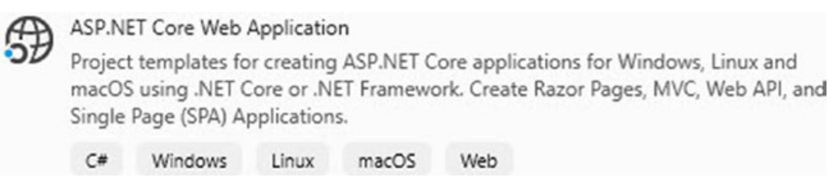


Figure 2-12. *ASP.NET Core Web Application template*

If you do not need to create an ASP.NET Core Web Application, you can create a web application that runs on the .NET Framework as seen in Figure 2-13.

Figure 2-13. *ASP.NET Web Application template*

As can be seen in the template description, these templates will allow you to create a regular Web Forms application, an MVC application or Web API application. If you need to run your application in a browser, then create an ASP.NET Web Forms or ASP.NET MVC application.

A Web API, on the other hand (sometimes also referred to as Web Services), is an application programming interface (API) that allows communication between various clients such as browsers, mobile devices,

etc., and other software components such as a database. It can be used as a stand-alone application or as part of an ASP.NET Web Forms or MVC application.

Class Library

The last project template we will be looking at is the Class Library. It is worth noting that the Class Library will create a DLL that you can reuse in your applications. This is the purpose of a Class Library project.

There are many more project templates available in Visual Studio and are dependant on the workloads you have installed.

As you see in Figure 2-14, the Class Library can be based on .NET Core, .NET Framework, or .NET Standard.

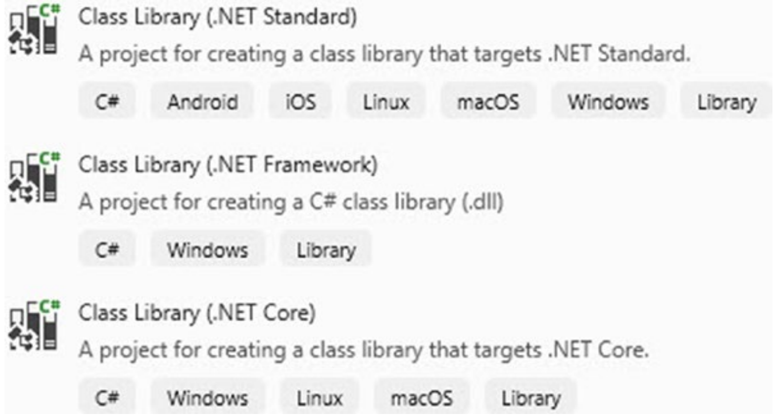


Figure 2-14. *Class Library Projects*

You might be wondering what the differences are between the various project templates. You will find a hint when you look at the tags. The Class Library running the .NET Framework will create a DLL that will only work on Windows machines.

The Class Library running .NET Core will create a library that will run on Windows, Linux, and macOS (it's therefore cross-platform).

The Class Library running on .NET Standard will create a library that is guaranteed to run on all of the platforms supported in Visual Studio. The .NET Standard is a specification of all the APIs that work on all of the platforms. Therefore, if you create a Class Library on .NET Standard, it is guaranteed to run on desktop, mobile, Web, etc.

For more information regarding .NET Standard, have a look at the following article on Microsoft Docs: https://docs.microsoft.com/en-us/dotnet/standard/net-standard

There are many more project templates to choose from, and the project templates you see will depend on the workloads that you have installed. Explore some of the different workloads available to you, and see what project templates are available to you after installing a particular workload.

Managing NuGet Packages

As a software developer, being able to reuse code is essential to any modern development effort. In fact, being able to share code is the cornerstone of a healthy development community. There are many developers that create extremely useful code libraries that can add functionality to your particular application.

This is where NuGet becomes an essential tool for developers to create, share, and consume useful code. As a developer, you can package a DLL along with other required content needed for the DLL to function correctly, into a NuGet package.

Essentially, NuGet is just a ZIP file with a .nupkg extension that contains the DLLs you have created for distribution. Included inside this package is a manifest file that contains additional information such as the version number of the NuGet package.

Packages uploaded to nuget.org are public and available to all developers that use NuGet in their projects. Developers can, however, create NuGet packages that are exclusive to a particular organization and not available publicly. We will have a look at hosting your own NuGet feeds later on. For now, let's have a look at how to use NuGet in your own Visual Studio project.

Using NuGet in Visual Studio

Developers can access NuGet right from within Visual Studio, but you can also browse `www.nuget.org` to find packages to use in your applications. In the following example, we will be using NuGet from within Visual Studio to add functionality to our ShipmentLocator application.

I have added a login form to the ShipmentLocator application as seen in Figure 2-15. What I want to do is encrypt the password typed in by the user and compare that to the encrypted password in the database.

As a rule, you should never be able to decrypt a password. If you can decrypt a password, then so can others that have more malicious intentions. After user registration, the encrypted password is stored in a database. Login requests are then encrypted and compared with the encrypted password in the database. If it's a match, they are successfully authenticated.

This is a very basic login screen but serves the purpose of illustrating how to use NuGet in your projects.

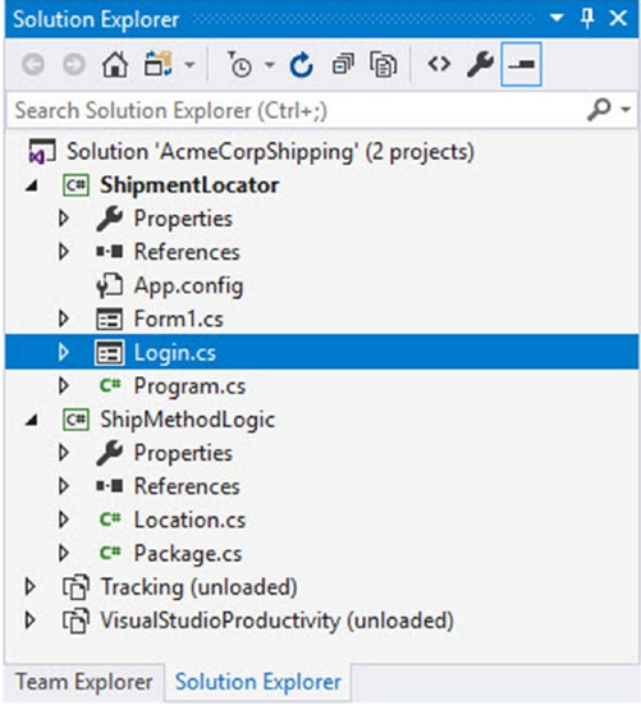


Figure 2-15. *Login form added*

You can definitely roll your own solution when it comes to encryption. Another route to take is to check NuGet to see if there are any solutions available that you can use.

To add a NuGet package to your project, right-click the project in the Solution Explorer, and click Manage NuGet Packages from the context menu as seen in Figure 2-16.

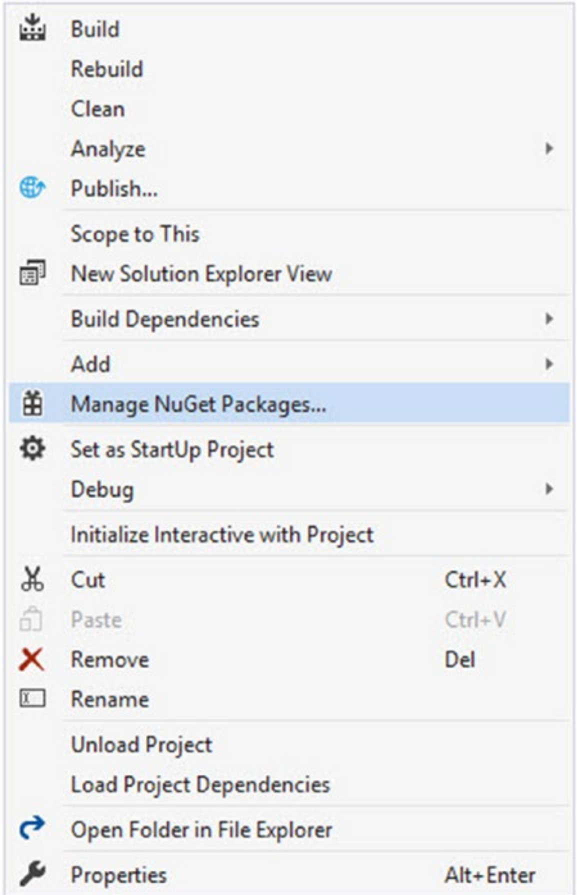


Figure 2-16. *Manage NuGet Packages*

From the NuGet Package Manager screen that is displayed, you can search for a NuGet package based on keywords you enter. As can be seen in Figure 2-17, I will be using a NuGet package called EncryptValidate that I created that provides encryption functionality.

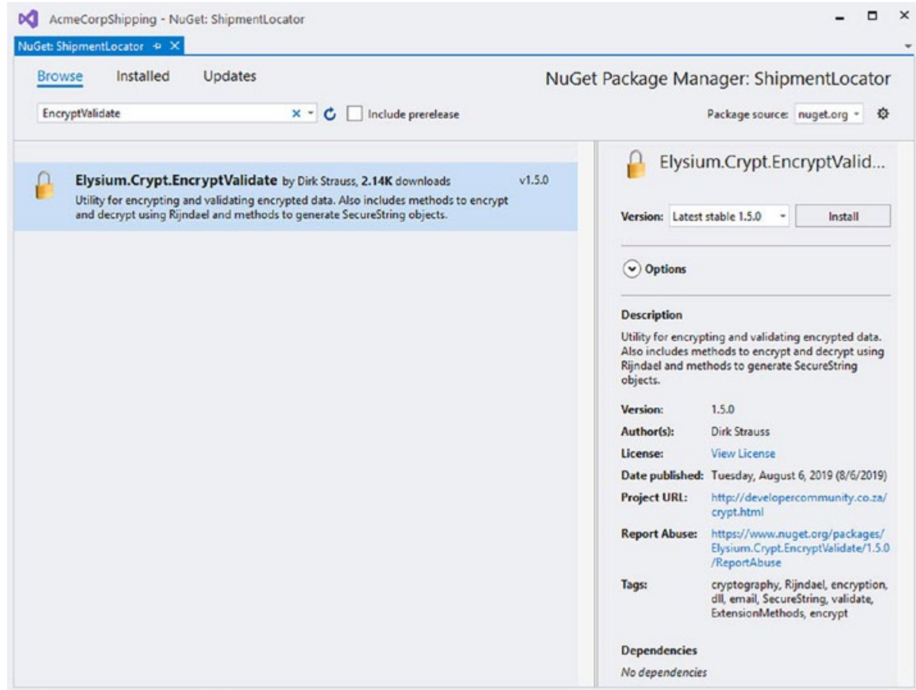


Figure 2-17. *NuGet Package Manager*

The NuGet Package Manager screen provides a lot of information about the package you are going to install. The current version number is displayed, license information, project URL, as well as the author and download count for the particular package.

The NuGet Package Manager also makes it easy for you to install previous versions of the NuGet package (Figure 2-18) if you find that the latest package does not work properly with your code.

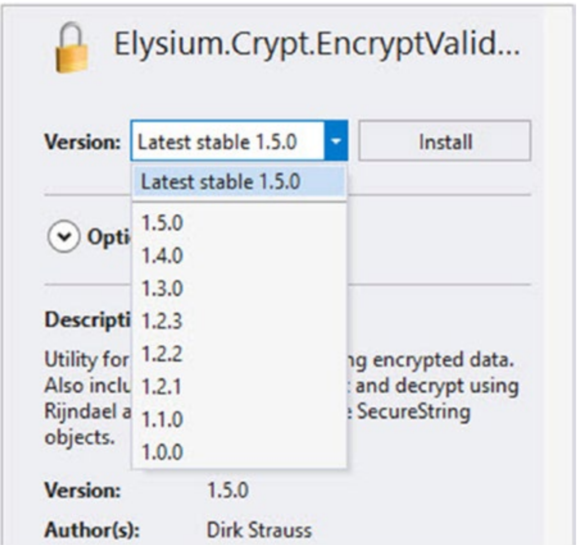


Figure 2-18. *Installing previous versions*

This allows you to easily roll back to a previous version should you need to. After installing the package, the NuGet Package Manager will indicate that this package has been installed as seen in Figure 2-19.

Figure 2-19. *NuGet Package Installed*

Figure 2-20 shows the NuGet package in the Visual Studio references.

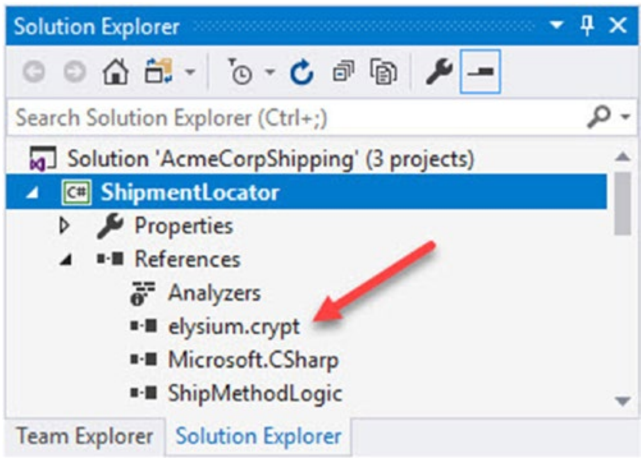


Figure 2-20. *References*

With NuGet, everything you need to use the package is added to your project. Now that we have added the EncryptValidate package to our project, let's start adding some code.

Listing 2-1. ValidateLogin method

```
private bool ValidateLogin()
{
    var blnLogin = false;
    try
    {
        var password = txtPassword.Text;

        // This encrypted password would be read from a
        database
        var storedEncrPassw = ReadEncryptedValueFromDatabase;

        if (ValidateEncryptedData(password, storedEncrPassw))
        {
            blnLogin = true;
        }
    }
    catch (Exception ex)
    {
        _ = MessageBox.Show(ex.Message);
    }
    return blnLogin;
}
```

The encrypted password is stored in the database. It is read in and stored in the `storedEncrPassw` variable. The clear-text password and the stored encrypted password are then validated. If validation succeeds, the user is logged in.

Remember, the code for this project is available on GitHub.

By adding a single NuGet package, we have added functionality to encrypt passwords, validate encrypted passwords, and encrypt and decrypt text using Rijndael, converting text to a `SecureString` object, reading the value from the `SecureString` object, and more. All this functionality has been added without having to write the logic ourselves.

This is the power that NuGet provides. It is definitely something you as a developer need to consider using if you do not already do so.

Hosting Your Own NuGet Feeds

Sometimes, you might need to create and share packages that are only available to a limited audience. Think of the developers inside your organization. Perhaps the company you work with does not allow the sharing of code with a public audience. Perhaps the code you want to share is really specific to your organization and not suitable for a public audience. Whatever the situation, NuGet supports private feeds in the following ways:

- Local feed – On a network file share
- NuGet.Server – On a local HTTP server
- NuGet gallery – Hosted on an Internet server using the NuGet Gallery project, you can manage users and features to allow searching and exploring available packages similar to nuget.org.

There are also other NuGet hosting solutions that do support the creation of remote private feeds. Some of these are

- Azure Artifacts
- MyGet – `https://myget.org/`
- ProGet – `https://inedo.com/proget`
- TeamCity – `www.jetbrains.com/teamcity/`

For a full list of NuGet hosting products and for more information on creating your own NuGet feeds, have a look at the following link on Microsoft Docs: `https://docs.microsoft.com/en-us/nuget/hosting-packages/overview`

Creating Project Templates

Sometimes developers create class libraries and code that they need to use over and over again across various new projects. What developers end up doing is copy and paste code into new class libraries. There is, however, an easier way to create projects that reuse code that you have previously written.

Enter Visual Studio project templates. These templates allow developers to speed up their development by including previously written code in new projects. Let's assume that we have created a project called ProjectUtilities (as seen in Figure 2-21) that contains various helper methods.

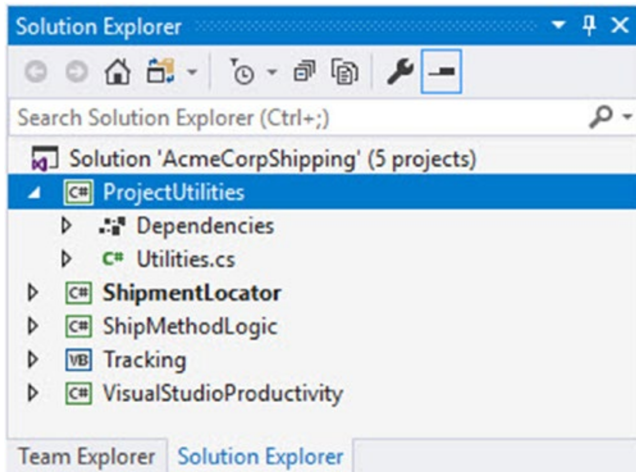


Figure 2-21. *ProjectUtilities Project*

This Class Library is something that we will need to use over and over in various projects. So I have decided to create a project template from it.

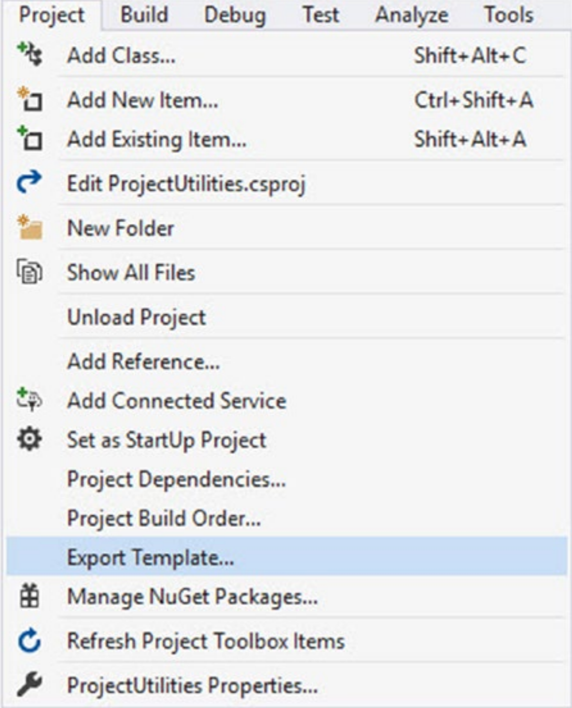


Figure 2-22. *Export Template*

From the Project menu, click Export Template as seen in Figure 2-22.

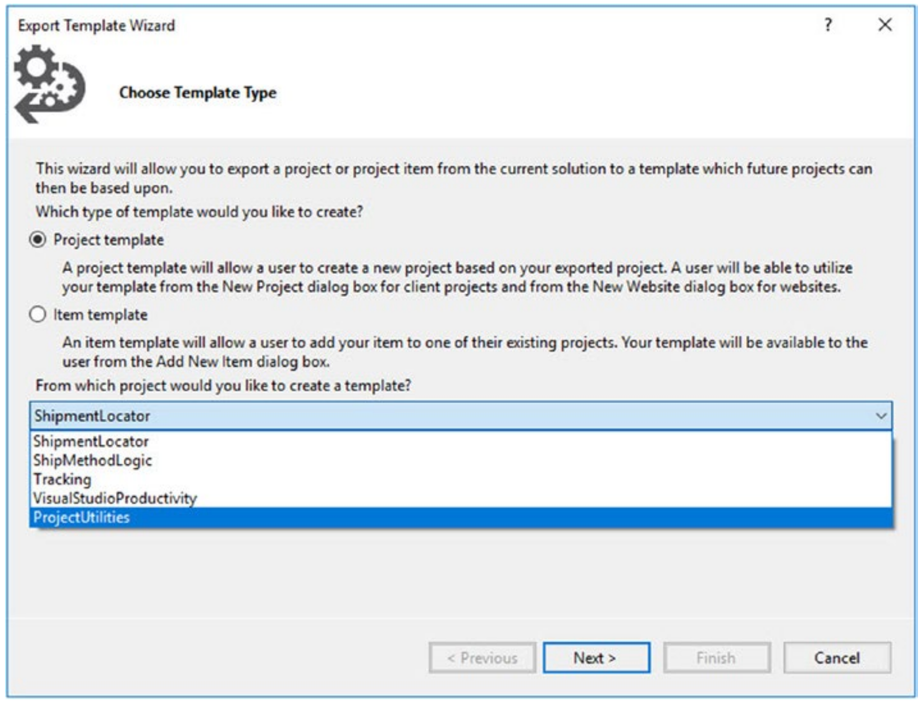


Figure 2-23. *Export Template Wizard*

The Export Template Wizard is displayed as seen in Figure 2-23. This allows you to specify which template you need to create. The options are to create a Project template or to create an Item template. A Project template is what we are after in this example, but you can also create an Item template. This will allow you to add the code via the Add New Item dialog box in Visual Studio.

For this example, however, we keep the Project template option selected and select the project to export from the drop-down list. This drop-down lists all the projects in my Visual Studio solution. Select the ProjectUtilities project, and click the Next button.

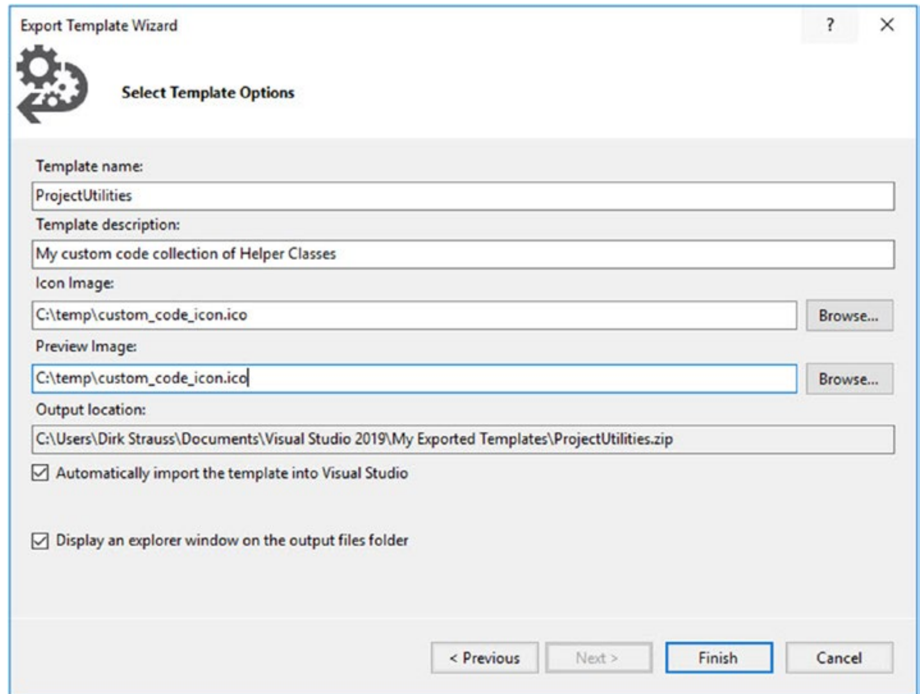


Figure 2-24. *Add Template Options*

The next window displayed is where one can enter various template options (Figure 2-24). Here I can give the template a suitable name and description, specify the icon and preview images, and select to import the template into Visual Studio automatically. Click Finish to create the new project template.

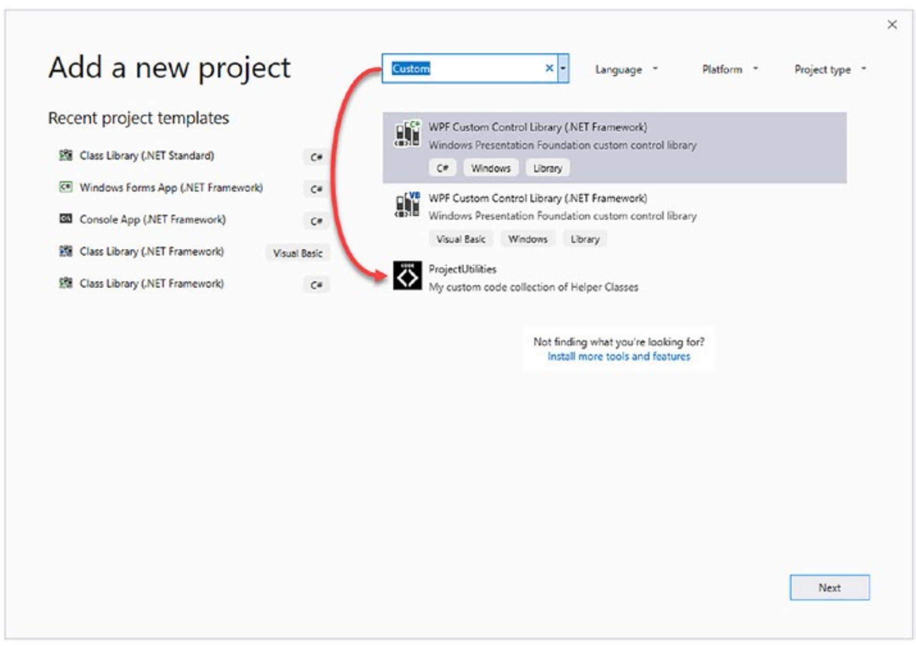


Figure 2-25. *Add a new project*

The next time I create a new project in Visual Studio, I can search for my Custom project template (Figure 2-25) and have it available for me to select.

Creating and Using Code Snippets

Code snippets in Visual Studio are small blocks of reusable code that you can insert into your code file by using a shortcut and tabbing twice or by using the right-click menu.

As an example (Figure 2-26), open a C# code file in Visual Studio and type the word `try` and hit the tab key twice.

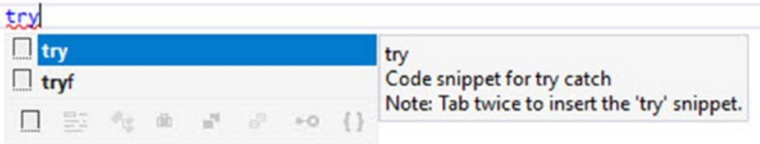


Figure 2-26. *Inserting a try code snippet*

This will insert a try-catch into your code file and allow you to enter the specific exception type being handled as seen highlighted in Figure 2-27.

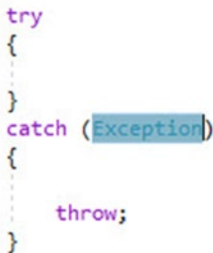


Figure 2-27. *The inserted try-catch block*

If you want to see all the available code snippets, you can open the Code Snippets Manager (Figure 2-28) by going to the Tools menu and clicking Code Snippets Manager.

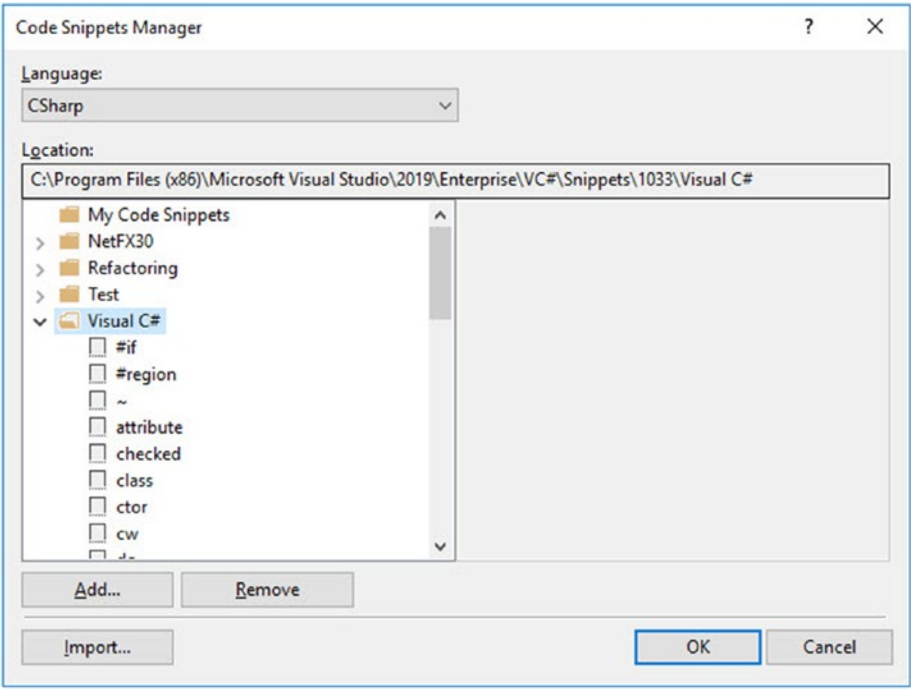


Figure 2-28. *Code Snippets Manager*

You can also hold down Ctrl+K, Ctrl+B to open the Code Snippets Manager window. Clicking each code snippet will display the description, shortcut, snippet type (expansion or surrounds with), and author. While some shortcuts are obvious (do, else, enum, for, and so on), others are not and might take some getting used to remembering to enter the shortcut and tabbing twice to insert the snippet.

If you can't remember the shortcut, you can invoke the snippets by hitting Ctrl+K, Ctrl+X (as seen in Figure 2-29) while inside the code file you are editing. This will display a menu in place that will allow you to search for and select the specific code snippet you want to use.

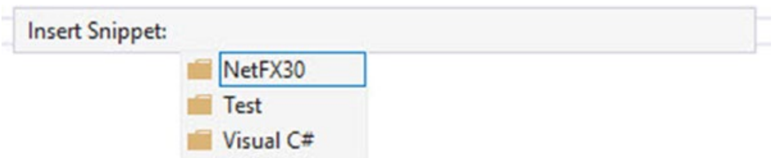


***Figure 2-29.** Ctrl+K, Ctrl+X to invoke a code snippet*

You can also right-click and select Snippets and then Insert Snippet from the context menu. The last way that you can insert a code snippet is via the menu bar by going to Edit, IntelliSense and clicking Insert Snippet. Visual Studio also allows developers to create their own code snippets. Let's have a look at that process next.

Creating Code Snippets

If there is one thing I wish, is that there was a nice interface baked into Visual Studio for creating and adding code snippets. Perhaps one day, but for now we have to do it the old-fashioned way.

This is through the use of an XML file. The basic snippet template XML looks as in Listing 2-2.

Listing 2-2. Basic Snippet Template

```
<?xml version="1.0" encoding="utf-8"?>
<CodeSnippets xmlns="http://schemas.microsoft.com/
VisualStudio/2005/CodeSnippet">
    <CodeSnippet Format="1.0.0">
        <Header>
            <Title></Title>
        </Header>
```

```
        <Snippet>
            <Code Language="">
                <![CDATA[]]>
            </Code>
        </Snippet>
    </CodeSnippet>
</CodeSnippets>
```

Let's assume that we have created a Custom project template that includes a logging class in our helper classes.

Refer to the previous section regarding Creating Project Templates.

This logging class will always be added to all new projects going forward, and I have to include it in the catch block of every try. The code for the logging class is basically as in Listing 2-3.

Listing 2-3. Basic Logging Class

```
public static class Logger
{
    public static void Log(string message)
    {
        // Perform some sort of logging
    }
}
```

Inside my code, I would like to be able to automatically add the code to log the error every time I insert a try-catch. The code snippet (Listing 2-4) file I create must, therefore, import the namespace as well as expand or surround the required code. Replacement parameters have also been defined in the snippet file for the `Exception` type by surrounding the word to replace (namely, expression) with the `$` characters.

Listing 2-4. Custom Try-Catch Snippet

```
<?xml version="1.0" encoding="utf-8"?>
<CodeSnippets xmlns="http://schemas.microsoft.com/
VisualStudio/2005/CodeSnippet">
    <CodeSnippet Format="1.0.0">
        <Header>
            <Title>Try Catch Log</Title>
            <Author>Dirk Strauss</Author>
            <Description>Creates a try catch that includes
            logging.</Description>
            <Shortcut>tryl</Shortcut>
            <SnippetTypes>
                <SnippetType>Expansion</SnippetType>
                <SnippetType>SurroundsWith</SnippetType>
            </SnippetTypes>
        </Header>
        <Snippet>
            <Declarations>
              <Literal>
                 <ID>expression</ID>
                 <ToolTip>Exception type</ToolTip>
                 <Function>SimpleTypeName(global::System.
                 Exception)</Function>
              </Literal>
            </Declarations>
             <Code Language="CSharp">
                <![CDATA[
                try
                {
                    $selected$
                }
```

```
                    catch ($expression$ ex)
                    {
                        Logger.Log(ex.Message);
                        $end$
                              throw;
                    }
                    ]]>
                </Code>
                <Imports>
                    <Import>
                        <Namespace>ProjectUtilities</Namespace>
                    </Import>
                </Imports>
            </Snippet>
        </CodeSnippet>
</CodeSnippets>
```

It is also worth noting that the code snippet might be XML, but the file extension must be .snippet for Visual Studio to be able to import it. If you refer back to Figure 2-28, you will notice an Import button on the Code Snippets Manager screen.

Click that button; browse for and import your newly created code snippet for the custom try-catch. You will notice that I have defined the shortcut as `tryl` for try-catch log.

This time, if you type the `tryl` shortcut into your code window, you will see that the description and title of the custom try-catch is displayed as seen in Figure 2-30.

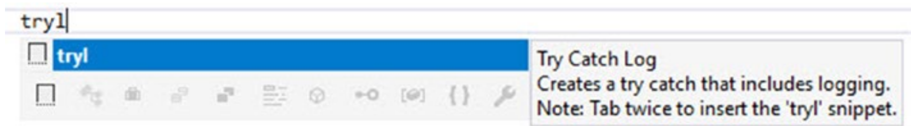


Figure 2-30. *Custom try-catch to include logging*

When you hit the tab key twice, the custom code snippet is inserted, and the required namespace, ProjectUtilities, which we created earlier as a project template, is imported along with the code snippet. This can be seen in Figure 2-31.

```
1   using ProjectUtilities;
2   using System;
3
4   namespace ShipmentTracking
5   {
        2 references
6       public class Package
7       {
            1 reference
8           public string WaybillNumber { get; private set; }
9
            1 reference
10          public Package(string waybillNumber)...
14
            1 reference
15          public Location TrackPackage()
16          {
17              // Perform some funky tracking logic
18              //Return package location
19              var location = new Location();
20
21              try
22              {
23
24              }
25              catch (Exception ex)
26              {
27                  Logger.Log(ex.Message);
28
29                  throw;
30              }
31
```

Figure 2-31. *Added try-catch including namespace*

This new code snippet is now available in all your future projects. Your C# code snippets live in the Documents folder in `C:\Users\[USERNAME]\Documents\Visual Studio 2019\Code Snippets\Visual C#\My Code Snippets`.

The code snippet schema reference is available on Microsoft Docs at the following link: https://docs.microsoft.com/en-us/visualstudio/ide/code-snippets-schema-reference?view=vs-2019

More often than not, you will be creating your own code snippets based off of an existing code snippet. This allows you to reuse functionality you know is working in the existing snippet and include it in your own.

Code snippets are definitely a very powerful productivity feature in Visual Studio.

Using Bookmarks and Code Shortcuts

At some point in your career, you will most likely be working on a very large code base. Do this for a while, and you will get bogged down with remembering where a specific bit of code is or where you need to go to get to a specific portion of logic.

Visual Studio can assist developers in bookmarking certain sections of code as well as adding shortcuts to other areas of code. Let's have a look at what bookmarks and shortcuts are and when to use which.

Bookmarks

Let's say that you are busy finishing up for the day, but just before you check in your code, you notice that there is some code that doesn't look quite right. It is a method that has a single return statement and you know that you can use an expression body for methods.

You really don't have the time to play around further because you have already passed the point that you needed to leave for home. So in order to not forget to have a closer look at this tomorrow, you decide to bookmark the method.

Place your cursor at the line of code you want to return to, and hold down Ctrl+K, Ctrl+K and Visual Studio will add a bookmark as seen in Figure 2-32.

```
        1 reference
48      private bool WaybillValid()
49      {
50          return txtWaybill.Text.ToLower().Contains("acme-");
51      }
52  }
53 }
54
```

Figure 2-32. Bookmark in Visual Studio

The bookmark is added to the side of the code editor and is indicated by a single black bookmark icon.

In order to see all the bookmarks in your project, you can hold down Ctrl+K, Ctrl+W or go to the View menu item and select Other Windows and then click Bookmark window.

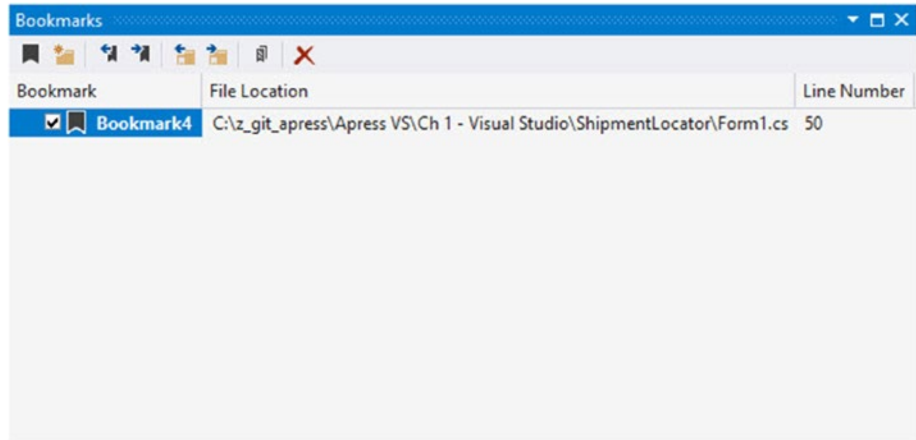


Figure 2-33. Bookmarks window

The Bookmarks window is displayed as seen in Figure 2-33. From the toolbar in the Bookmarks window, you can group bookmarks in folders, navigate between bookmarks, navigate between bookmarks in the current folder, toggle a bookmark on the currently selected line in code, disable all bookmarks, and delete bookmarks.

There is however another feature not so obvious by looking at this Bookmarks window, and that is the ability to rename bookmarks. To rename a bookmark, click a selected bookmark, and you will see that the name (in this case "Bookmark4") becomes editable.

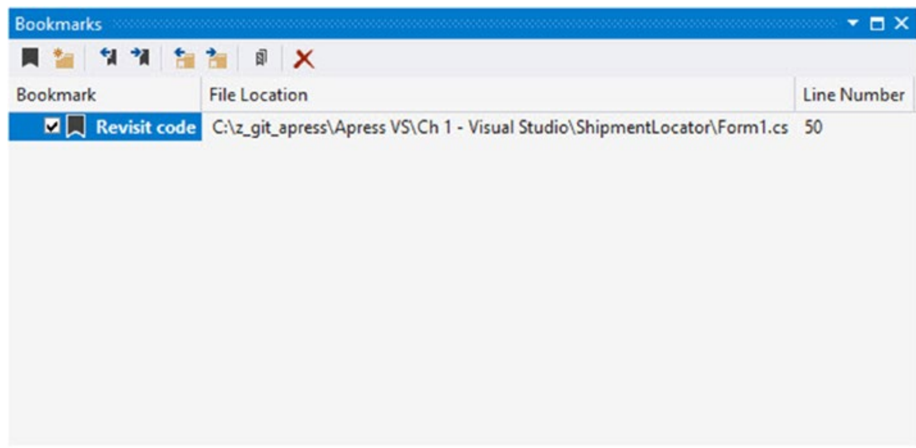


Figure 2-34. *Renamed bookmark*

Now you can rename your bookmark to something more relevant to what you need to remember as seen in Figure 2-34. Go ahead and add some more bookmarks to other random areas of code. Your Bookmarks window will end up looking rather full.

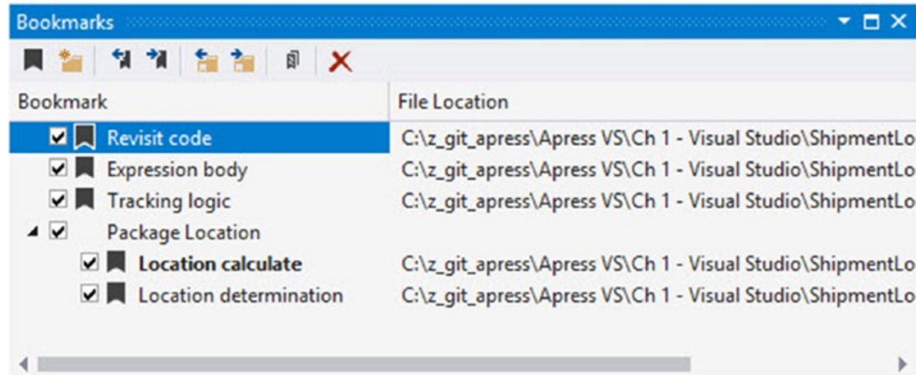


Figure 2-35. *Bookmarks collection*

Now click the delete button on the Bookmarks toolbar. The bookmark is deleted without any confirmation from the user.

This is something I can sort of understand. Imagine how irritating it would be having to confirm every delete, especially when you want to remove only a subsection of bookmarks from your collection.

It is for this reason that I use bookmarks only as a short-term solution to remind me to go and perform some action in code or to refactor something I think needs refactoring.

For me, a bookmark is something I will come back to within the next day or so. Something I don't want to put off doing. It is, therefore, a temporary placeholder to something I need to revisit.

But what if I wanted to go and add a more permanent pointer to some logic in code? This is where code shortcuts come in. Let's have a look at this next.

Code Shortcuts

The ability to add code shortcuts in Visual Studio is more helpful when you need to quickly jump to a certain section of code regularly. To add a shortcut to a specific section of code in Visual Studio, you need to place your cursor on the line of code you need to revisit and type Ctrl+K, Ctrl+H.

```
34
      2 references
35    public static class Logger
36    {
          2 references
37        public static void Log(string message)
38        {
39            // Perform some sort of logging
40        }
41    }
42 }
43
```

Figure 2-36. *Code Shortcut Added Indicator*

Visual Studio will then add the shortcut as seen in Figure 2-36. To view all the shortcuts added to your project, hold down Ctrl+\, Ctrl+T or go to the View menu, and select Task List to open the Task List window as seen in Figure 2-37.

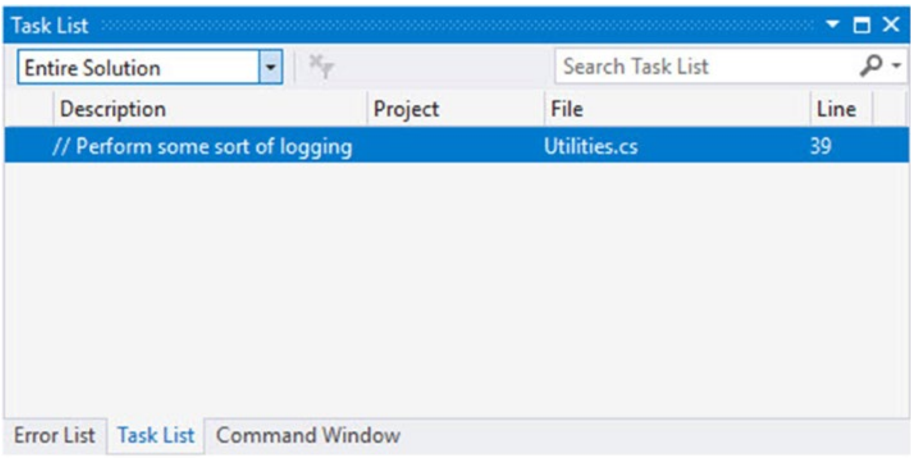


Figure 2-37. *The Task List*

In some ways, I prefer the Task List more than using bookmarks because I can quickly add items to revisit by adding `//TODO:` in my code. With your Task List open, go to any place in your code, and add the following comment.

Listing 2-5. TODO Comment

```
// TODO: Remember to do something here
```

Now have a look at your Task List. You will notice that the TODO comment has been added to your Task List as seen in Figure 2-38.

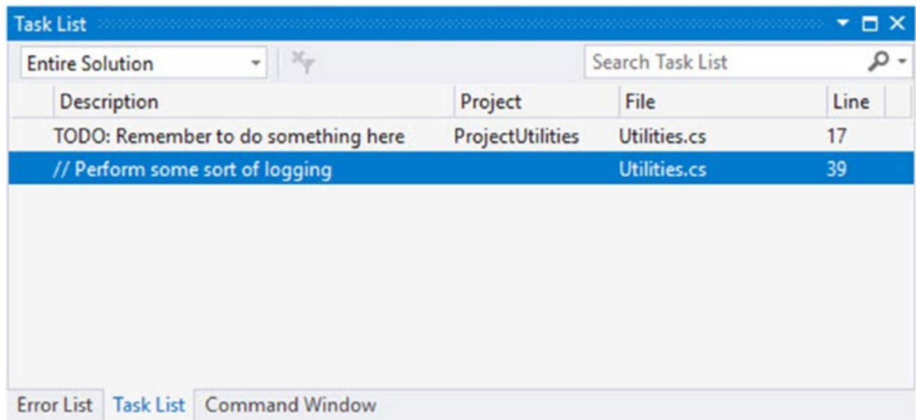


Figure 2-38. *TODO comments in the Task List*

This is a nice and quick method for adding reminders to your code so that you can easily refer to them and navigate to them by double-clicking the item in the Task List. You can, therefore, use the Task List to take you directly to the predefined location in code.

In Visual Studio, TODO is what we call a predefined token. Therefore, a comment in your code that uses a predefined token will appear in your Task List. The tokenized comment is made up of the following:

- The comment marker, which is //
- The predefined token (TODO in our example)
- The rest of the comment

The code in Listing 2-5 is, therefore, a valid comment using a token and will appear in the Task List. Visual Studio includes the following default tokens:

- HACK
- TODO
- UNDONE
- UnresolvedMergeConflict

These are by no means case sensitive and will appear in your Task List if following the form in Listing 2-5. You can also add your own custom tokens. Let's see how to do that next.

Adding Custom Tokens

I like the idea of `TODO` to add items to my Task List, but I would also like to add a custom token to add an entry in my Task List that is a nice to have feature. Something that is less restrictive than a `TODO`, because that implies that this action must be completed.

I do not want to have a bunch of `TODO` entries for items that are simply nice to have features. For this reason, I want to add a custom token called `NOTE` that is simply a reminder to look at something, if and when I have the time.

To add the custom token, go to the Tools menu, and click Options. Under Environment, select Task List as seen in Figure 2-39.

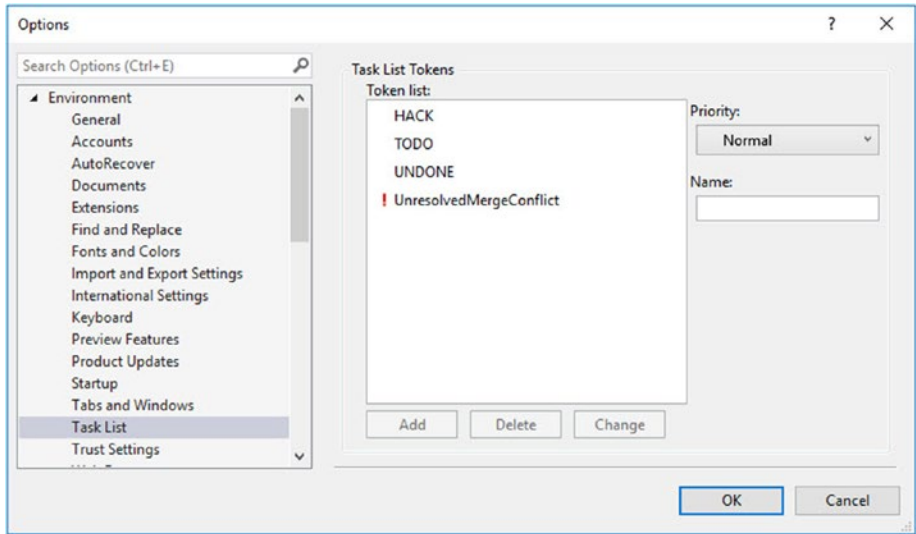


Figure 2-39. *Add Custom Tokens*

In the Name text box, add the word NOTE and set the priority to Low. Then click the Add button. The custom token `NOTE` is added as can be seen in Figure 2-40.

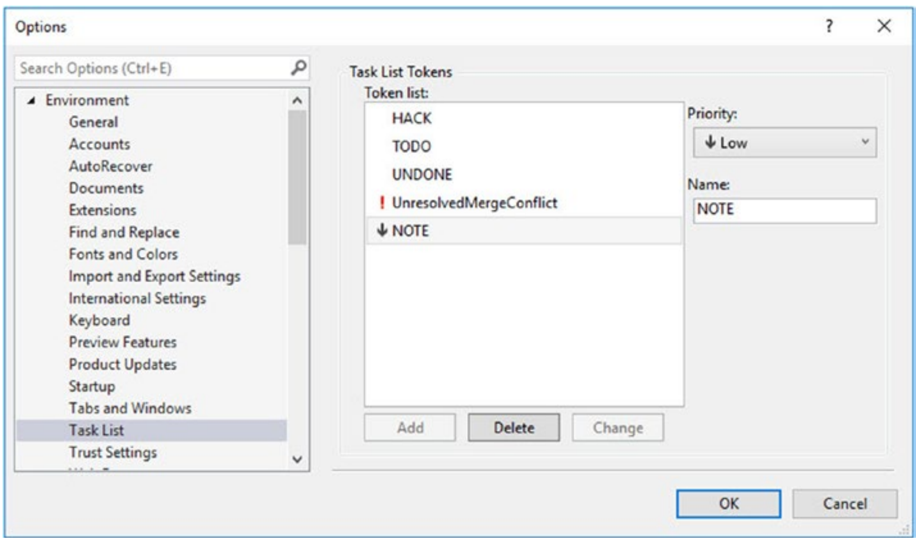


Figure 2-40. *The Custom Token Added*

Adding a `NOTE` to your code will pop up in your Task List as a low priority task as seen in Figure 2-41.

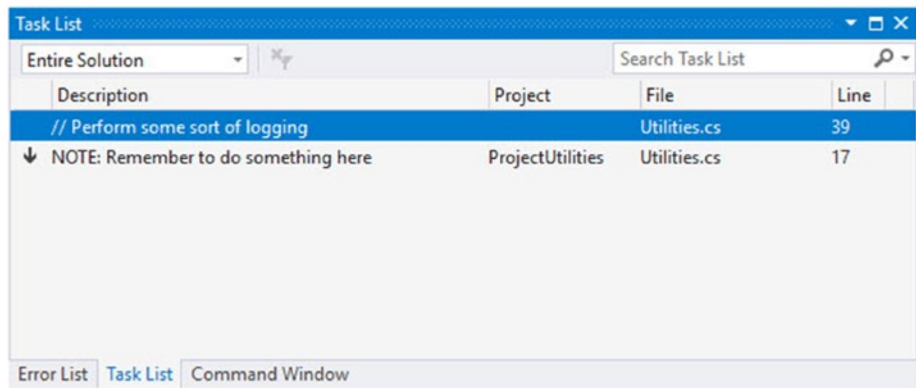


Figure 2-41. *Adding a NOTE token comment*

Being able to add custom tokens in Visual Studio, as well as applying a priority to each, allows you to be very specific with comments that contain tokens. This way you can greatly increase the ease and efficiency of navigating through a large code base.

The Server Explorer

As the name suggests, the Server Explorer provides a quick and easy way of accessing servers. You can use it to test connections and view SQL Server databases or any databases that have the ADO.NET provider installed.

You can access the Server Explorer by holding down Ctrl+Alt+S or by going to the View menu and clicking Server Explorer.

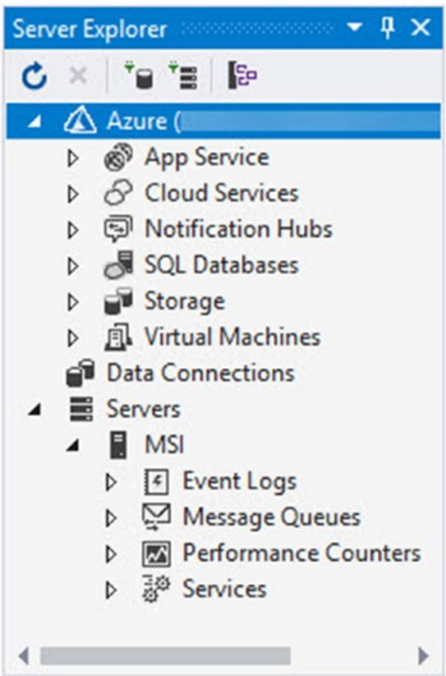


Figure 2-42. *Server Explorer*

As seen in Figure 2-42, the Server Explorer offers access to Event Logs, Message Queues, Performance Counters, and Services on my local machine (MSI). It also provides access to my Azure subscriptions.

I have a local instance of SQL Server installed, so now I can connect to this instance right from within Visual Studio by clicking Connect to Database.

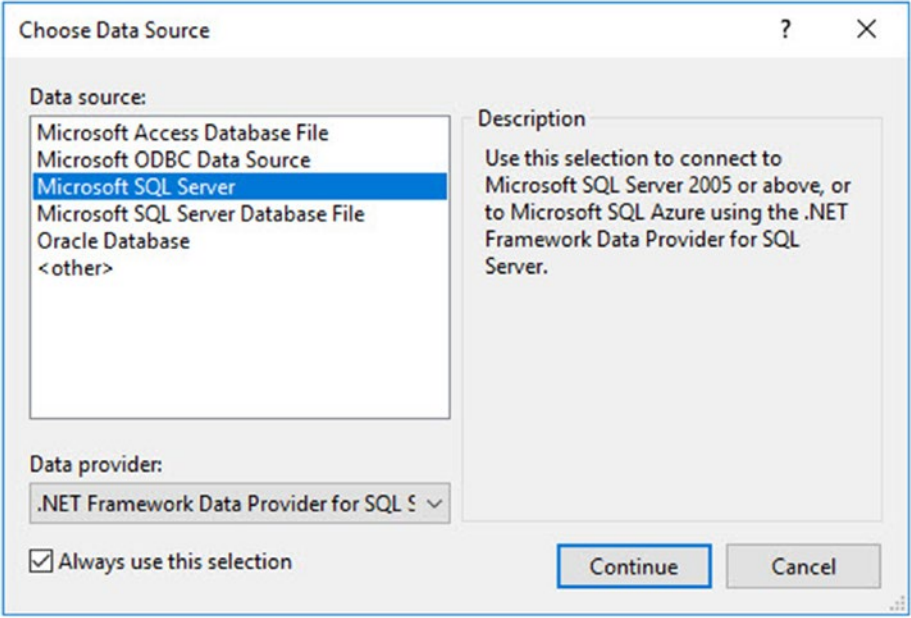


Figure 2-43. *Choose Data Source*

This will display a window allowing you to choose a data source as seen in Figure 2-43. You can connect to various types of data sources, but we are only interested in Microsoft SQL Server for now. Select that from the list and click Continue.

The next window (Figure 2-44) allows you to define your connection to the database. Here you need to specify the server name, the authentication type, and if SQL Server Authentication is selected, provide the username and password.

This will then allow you to select a database from the list to connect to. To check if the connection settings are correct, you can click the Test Connection button.

Figure 2-44. *Add Connection*

After adding the database to your Server Explorer, you will see the instance added to your list from where you can expand the various nodes to view Tables, Views, Stored Procedures, etc., as seen in Figure 2-45.

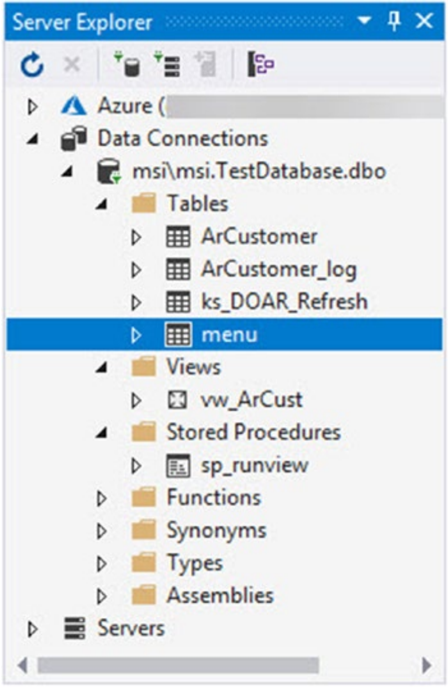


Figure 2-45. *Database Added to Server Explorer*

By double-clicking a table, Visual Studio will display the table designer for you along with a create table T-SQL statement as seen in Figure 2-46.

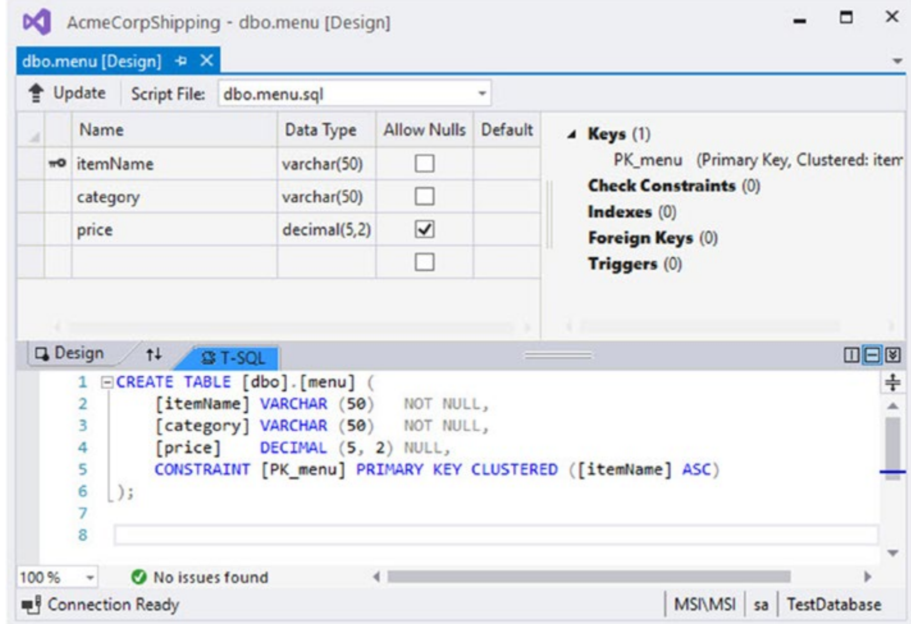


Figure 2-46. *Table Designer*

From this window, you can easily update the table. The create table statement in Figure 2-46 is listed in Listing 2-6.

Listing 2-6. Create Table Statement

```
CREATE TABLE [dbo].[menu] (
    [itemName] VARCHAR (50)   NOT NULL,
    [category] VARCHAR (50)   NOT NULL,
    [price]    DECIMAL (5, 2) NULL,
    CONSTRAINT [PK_menu] PRIMARY KEY CLUSTERED ([itemName] ASC)
);
```

We can now modify the menu table by altering the T-SQL statement as follows (Listing 2-7).

Listing 2-7. Modified Create Table Statement

```
CREATE TABLE [dbo].[menu] (
    [itemName] VARCHAR (50)   NOT NULL,
    [category] VARCHAR (50)   NOT NULL,
    [price]    DECIMAL (5, 2) NULL,
    [priceCategory]    VARCHAR (5) NULL,
    CONSTRAINT [PK_menu] PRIMARY KEY CLUSTERED ([itemName] ASC)
);
```

I want to add a price category field to the table. When I modify the create table statement, I see the changes reflected in the table designer as seen in Figure 2-47.

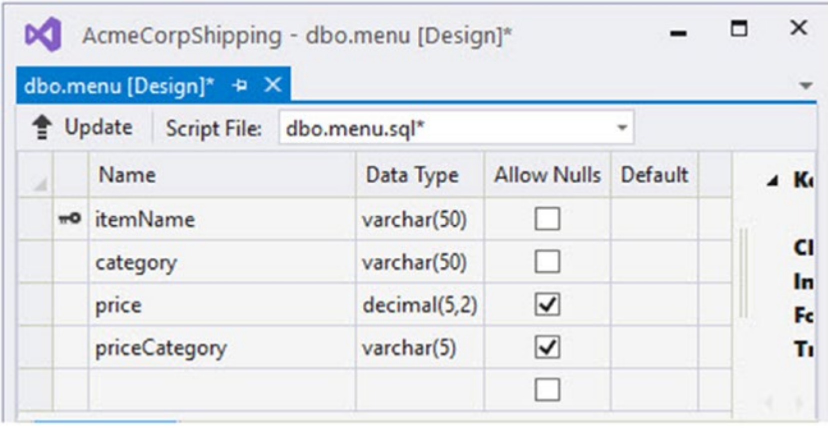


Figure 2-47. *Table Design Updated*

The changes have not been applied to my table yet. For this to update the table, I need to click the Update button.

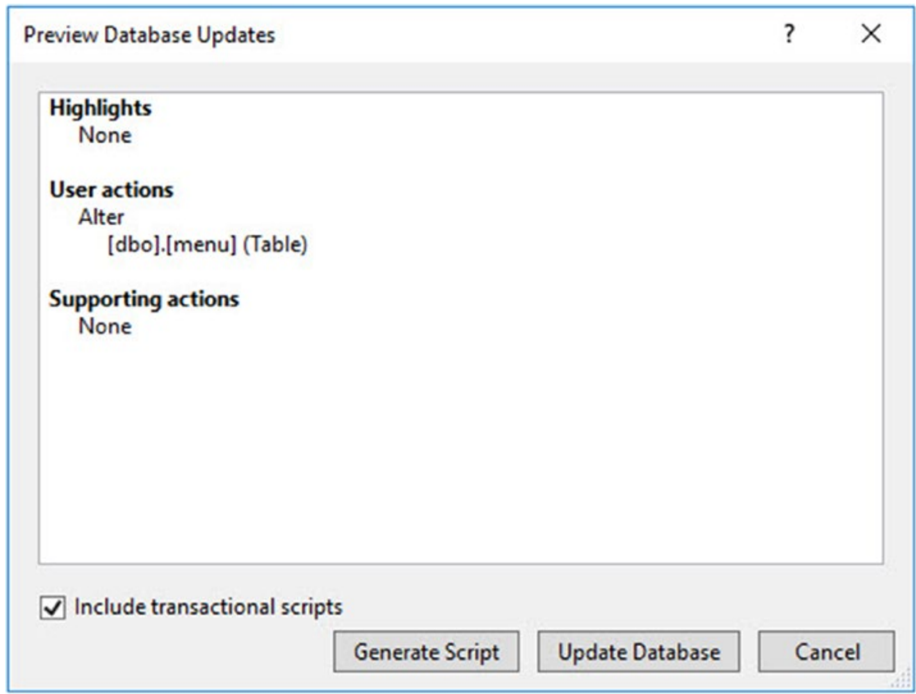


Figure 2-48. *Preview Database Updates*

This will now allow me to preview the database updates about to be applied as seen in Figure 2-48. If you do not want to let Visual Studio update the table, you can have it generate the script by clicking the Generate Script button. Alternatively, you can go ahead and click the Update Database button.

This will then start the process of updating the database table with the changes you made.

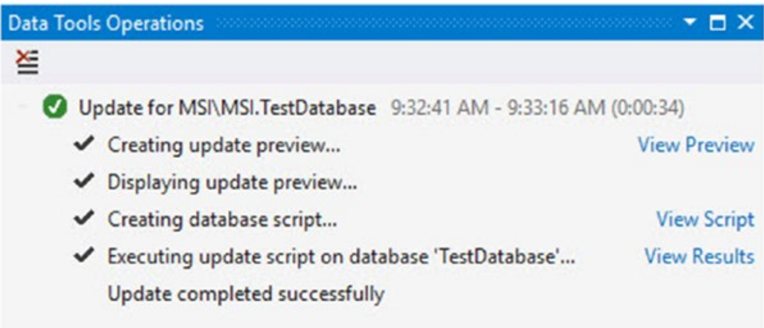


Figure 2-49. *Data Tools Operations*

After the update is complete, you can see the results in the Data Tools Operations window as seen in Figure 2-49. From here, you can view the script as well as view the results.

Running SQL Queries

The Server Explorer also allows developers to run SQL queries, right from within Visual Studio. Go ahead and right-click a table (Figure 2-50), and click New Query from the context menu.

Take note that the context menu changes depending on what item you have right-clicked in the Server Explorer. When right-clicking a table, you will see items related to a SQL table. When right-clicking a View, you will see items specific to the View such as Show Results and Open View Definition. When right-clicking a Stored Procedure, the context menu will display the Execute command.

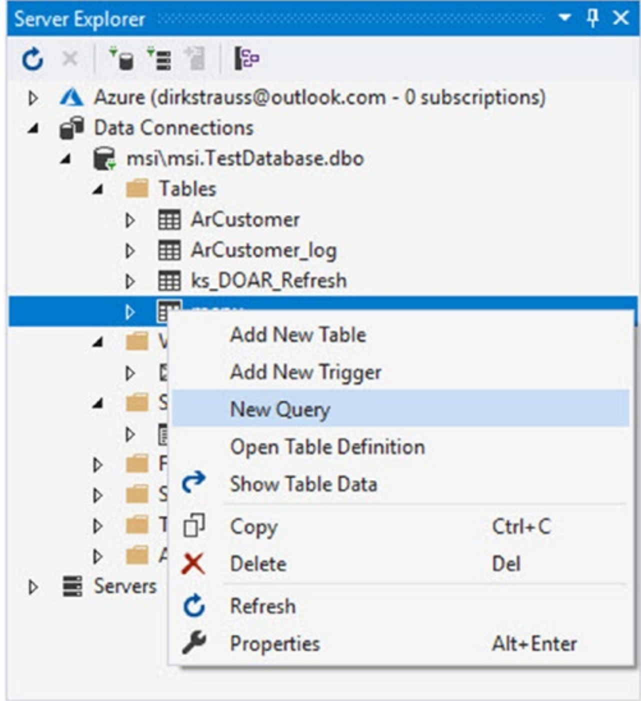


Figure 2-50. *Run a SQL Query*

Copy the SQL query in Listing 2-8. You will obviously have had to create the table first using the `CREATE` statement in Listing 2-7.

Listing 2-8. SQL Select Statement

```
SELECT
        itemName
        , category
        , price
        , priceCategory
FROM menu
```

When you have pasted the SQL statement (Figure 2-51), execute it by clicking the run button, by holding down Ctrl+Shift+E or executing it with the debugger by holding down Alt+F5.

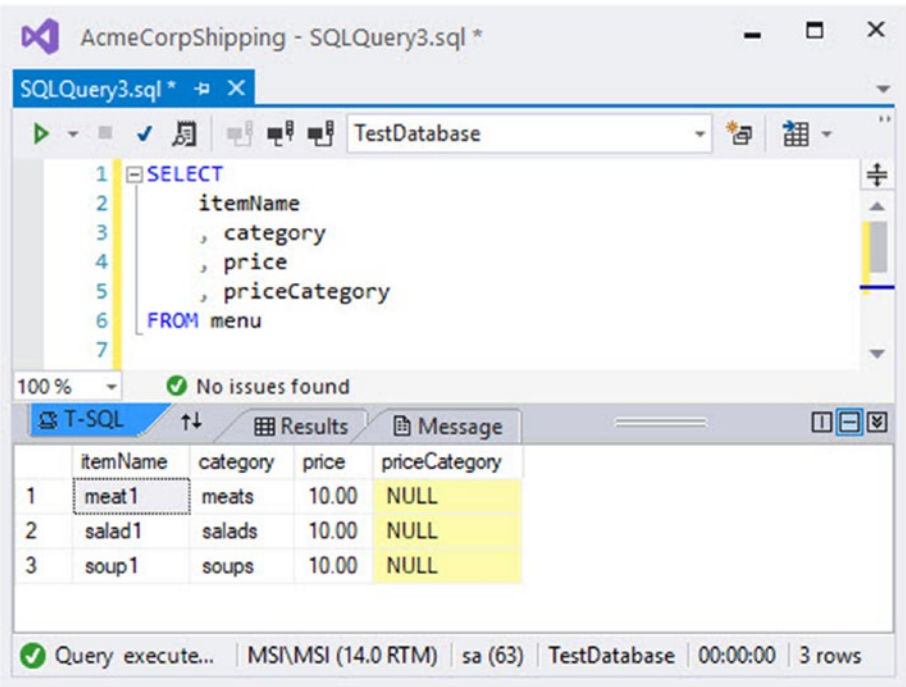


Figure 2-51. *Running a Select Statement*

If you are used to pressing F5 in SQL Server Management Studio, you might find yourself starting the Visual Studio debugger instead of running the query. I simply find it easier to click the run button and avoid my muscle memory faux pas.

Adding an additional item to the table is easily done by running the `INSERT` statement in Listing 2-9.

Listing 2-9. Insert Statement

```
INSERT INTO [dbo].[menu]
        ([itemName],[category],[price],[priceCategory])
        VALUES
        ('bread','breads',2.50,'baker')
```

If we run the SELECT statement again, you will see that the entry has been added to the table as seen in Figure 2-52.

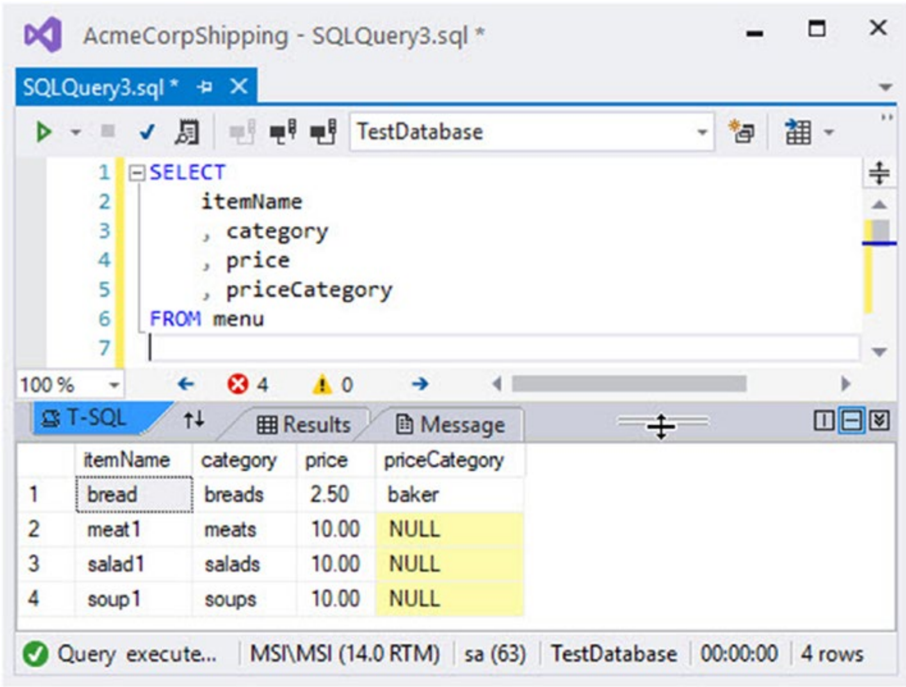


Figure 2-52. *New Item Inserted*

From the results displayed in Figure 2-52, we can see that by adding the priceCategory column, we have a few NULL fields in the menu table. Let's change that by running the SQL statement in Listing 2-10.

Listing 2-10. SQL Update Statement

```
UPDATE menu
SET priceCategory = 'DELI'
WHERE category IN ('meats', 'salads', 'soups')
```

When we look at the table data after the UPDATE statement (Figure 2-53), you will see that the table has been updated to display the correct priceCategory values for the items contained in the table.

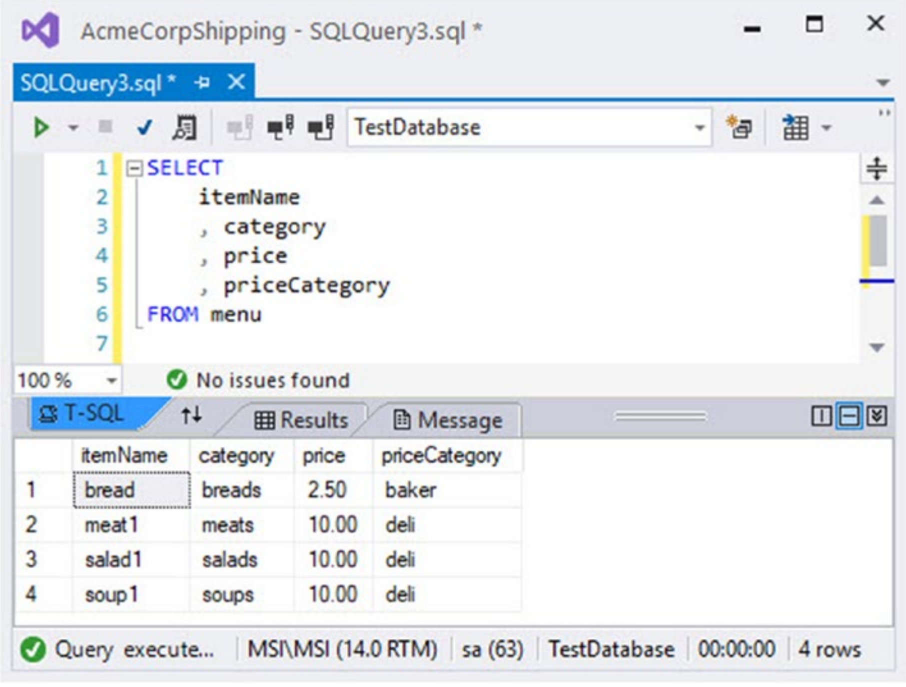


Figure 2-53. *Table Updated*

While running SQL statements isn't mind-blowing, it is very convenient being able to do all this without ever leaving Visual Studio. The Server Explorer definitely offers much more functionality than illustrated in this chapter. Dig around it a bit more, and see what the Server Explorer can do for your productivity.

Visual Studio Windows

I have often maintained that developers get stuck in a rut when it comes to working with Visual Studio. They tend to stick to what they know and keep on doing things that way until the cows come home.

This isn't necessarily a bad thing, but developers might miss out on some of the awesome tools and features available to them that Visual Studio provides right out of the box. In this section, I want to briefly discuss two of the items found under the View, Other Windows menu as seen in Figure 2-54.

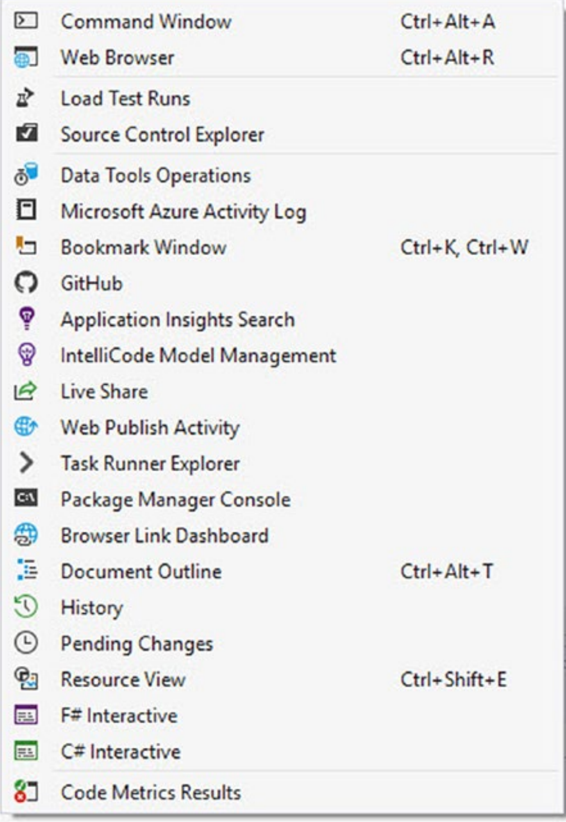


Figure 2-54. *Other Windows in Visual Studio*

Here are too many windows to discuss all in this chapter, but I will touch on two that I find very useful.

C# Interactive

How often have you wanted to test a small bit of code just to see if it works correctly? Well, with C# Interactive, you can do just that without having to debug your entire solution. Found toward the bottom of the View, Other Windows menu, C# Interactive is almost hidden. But gems usually are and you're going to love using it if you don't already.

Click C# Interactive, and paste the following code in Listing 2-11. After pasting the code into C# Interactive, hit the Enter key to run the code.

Listing 2-11. Running a LINQ Query

```
var numList = new List<int>() { 153, 114, 116, 213, 619, 18,
176, 317, 212, 510 };
var numResults = numList.Where(x => x > 315);
foreach(var num in numResults)
{
   Console.WriteLine(num);
}
```

The results are immediately displayed below the code you pasted. Your C# Interactive window should now look as in Figure 2-55.

```
C# Interactive (64-bit)
> var numList = new List<int>() { 153, 114, 116, 213, 619, 18, 176, 317, 212, 510 };
. var numResults = numList.Where(x => x > 315);
. foreach (var num in numResults)
. {
.     Console.WriteLine(num);
. }
619
317
510
> |
100 %
```

Figure 2-55. *C# Interactive Code Results*

C# Interactive is what we refer to as a REPL (Read-Eval-Print Loop). Being able to input expressions that are evaluated and results returned makes on the spot debugging possible in Visual Studio.

C# Interactive supports IntelliSense, so you get the same kind of editor experience as in Visual Studio. For a list of available keyboard shortcuts, REPL commands, and Script directives that C# Interactive supports, just type in `#help` and press the Enter key.

Code Metrics Results

The project that we have been using in this chapter is really not complex at all. It is really just to illustrate the concepts in this book. If you take a more complex project, one of the projects you have worked on at work, this next screen will look a lot different.

Code Metric Results (Figure 2-56) is a set of measurements that allow developers to gain a better insight into the code that they are producing.

Code Metrics Results

Filter: None Min:

Hierarchy	Maintainability...	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Code
ProjectUtilities (Debug)	97	6	1	2	41
ShipmentLocator (Debug)	76	33	7	39	437
ShipMethodLogic (Debug)	92	11	1	3	43
Tracking (Debug)	95	23	3	23	111
VisualStudioProductivity (I	96	9	1	2	39

Error List Task List Code Metrics Results

Figure 2-56. *Code Metrics Results*

The image in Figure 2-56 are the results for the ShipmentLocator application we have been using throughout this chapter. It's not really complex at all, so the metrics might seem all fine.

Please note that when you first open the Code Metrics Results screen, it will be blank. You need to click the Calculate Code Metrics for Solution button in the top left corner of the window.

Looking at the same screen (Figure 2-57) for a more complex project (one of my old legacy projects), the metrics are quite different.

Code Metrics Results

Filter: None Min:

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Code
Authentication (Debu	76	723	4	147	5 150
Common (Debug)	86	933	4	158	7 033
Comms (Debug)	78	1 537	4	177	10 756
Core (Debug)	77	25 186	7	902	155 980
Core.Data (Debug)	77	19 932	4	692	142 517
Crypto (Debug)	80	31	1	18	320
SysEnvironment (Deb	100	1	1	0	13
Service (Release)	72	427	4	67	2 266
Business (Debug)	82	3 987	1	42	19 292
Helpers (Debug)	82	146	2	88	1 437
Lib (Debug)	76	892	1	207	7 148
Security (Debug)	85	77	1	18	486
TestApp (Debug)	78	1 645	4	173	11 653

Figure 2-57. *Code Metrics Results on a large project*

Each metric in the window refers to a specific software measurement that was performed. These are the code metrics that Visual Studio calculates:

- Maintainability Index
- Cyclomatic Complexity
- Depth of Inheritance
- Class Coupling
- Lines of Code

The advantage of these metrics makes it possible for developers to understand what portions of code need to be worked on or more rigorously tested. It also allows developers to identify potential risks in their software. Ratings in this window are also color coded so that developers can quickly identify trouble spots.

Maintainability Index

This will be a value between 0 and 100 and represents how easy it is to maintain the code. The higher the value, the more maintainable your code is.

Cyclomatic Complexity

This metric measures the structure of your code and how complex it is. It uses the number of code paths it finds that flow through the program to calculate this score. A higher number indicates a complex control flow and is, therefore, harder to test and maintain. The numbers displayed in Figure 2-56 and Figure 2-57 are totaled for each project in the solution. Here it makes sense to expand the hierarchy and drill down to the individual methods to see where the problem areas lie.

Depth of Inheritance

As the name suggests, this metric measures the number of classes that inherit from each other. This goes all the way down to the base class. This means that a high number indicates a deep inheritance which is bad. This is because any changes to a base class have the potential to result in breaking changes further up in the derived classes. Here you will be wanting to see a lower score.

Class Coupling

Class Coupling basically measures how many classes a single class uses. Here, a high number is bad, and a low number is good. Class Coupling has been shown to accurately predict software failures. With a high coupling score, the maintenance and reusability of the class become really difficult because it depends on too many other types.

Lines of Code

The lines of code here are based on the count of the IL code. So this isn't a true count of lines of code in the source file. Nevertheless, you will probably agree with me that a high count indicates that a lot is happening. Expanding the projects and viewing the code counts for individual methods will allow you to see which methods are trying to do too much. A high line count will indicate a method that is harder to maintain. Try to refactor these methods and simplify them.

Send Feedback

The Visual Studio team definitely takes feedback seriously. So much so that it drives much of what they do to improve Visual Studio.

If you are experiencing a problem in Visual Studio 2019, click the feedback button as seen in Figure 2-58.

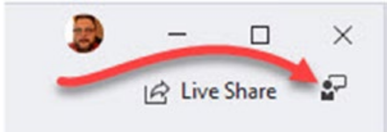


Figure 2-58. *Send Feedback button*

You can now report a problem or suggest a feature right from inside Visual Studio 2019.

As a developer, we should take the time to report issues we come across. This can be anything from crashes to slow performance or something else unexpected.

CHAPTER 3

Debugging Your Code

Debugging code is probably one of the most essential tasks that a developer performs. Being able to run your application and pause the execution of code midway is a lifesaver. But there is a lot more to debugging than just setting breakpoints and viewing results.

In this chapter, we will be discussing the options available to you as a developer that needs to effectively debug their code. We will be looking at

- Using breakpoints, conditional breakpoints, breakpoint actions, and labels and exporting breakpoints
- Using data tips
- The DebuggerDisplay attribute
- Diagnostic tools and Immediate Window
- Attaching to a running process
- Remote Debugging

Visual Studio gives developers all the required tools in order to effectively debug the code you are experiencing problems with. Without being able to debug your code, it will be virtually impossible to resolve any issues you might be experiencing.

Not being able to effectively debug your application (not knowing how to effectively use the tools you have) is just as bad as not having the tools to debug with in the first place.

© Dirk Strauss 2020
D. Strauss, *Getting Started with Visual Studio 2019*,
https://doi.org/10.1007/978-1-4842-5449-3_3

Working with Breakpoints

If you are familiar with debugging in Visual Studio, this chapter might seem like old hat for you. Stick around, there might be sections discussed here that you didn't know about.

If you are new to Visual Studio, the concept of debugging in Visual Studio is when you run your application with the debugger attached. Debugging allows you to step through the code and view the values stored in variables. More importantly, you can see how those values change.

Setting a Breakpoint

The most basic task of debugging is setting a breakpoint. Breakpoints mark the lines of code that you want Visual Studio to pause at, allowing you to take a closer look at what the code is doing at that particular point in time. To place a breakpoint in code, click the margin to the left of the line of code you want to inspect as seen in Figure 3-1.

```
36  try
37  {
38      var password = txtPassword.Text;
39
40      // This encrypted password would be read
41      // from a database based on the username
42      var storedEncrPassw = ReadEncryptedValueFromDatabase;
43
44      if (ValidateEncryptedData(password, storedEncrPassw))
45      {
46          blnLogin = true;
47      }
48  }
```

Figure 3-1. *Setting a breakpoint*

This line of code is contained in the `ValidateLogin()` method. The method is called when the user clicks the login button. Press F5 or click Debug, Start Debugging to run your application. You can also just click the Start button as shown in Figure 3-2.

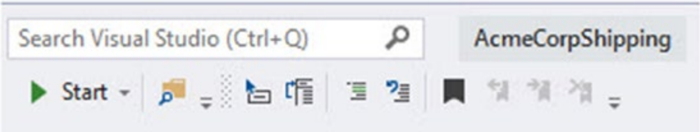


Figure 3-2. *The Start button*

After you start debugging, and a breakpoint is hit, the debug toolbar in Visual Studio changes as seen in Figure 3-3.

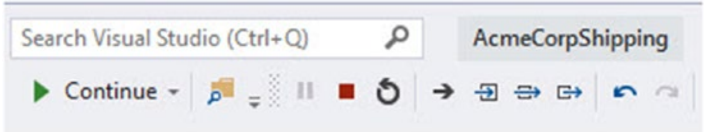


Figure 3-3. *Debug Toolbar when breakpoint hit*

The Start button now changes to display Continue. Remember, at this point, your code execution is paused in Visual Studio at the breakpoint you set earlier.

In order to step through your code, you can click the step buttons as displayed in Figure 3-4.

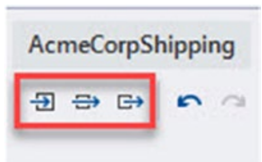


Figure 3-4. *Step Buttons*

From left to right, these buttons are as follows:

- Step Into (F11)
- Step Over (F10)
- Step Out (Shift+F11)

When you step into a method, you jump to the point in the editor where that method's code is. If you do not want to step into the method, you can click the Step Over button or press F10 to carry on with the next line of code. If you are inside a method and want to step out and continue debugging the calling code, click the Step Out button or press Shift+F11.

Step into Specific

Imagine that we need a method that generates a waybill number based on specific business rules. Then, when the application starts, the text box field is auto-populated with the generated waybill number.

The code used to generate the random waybill functionality is listed in Listing 3-1.

Listing 3-1. Waybill Generation Code

```
private string GenerateWaybill(string partA, int rndNum) =>
$"{partA}-{rndNum}-{DateTime.Now.Year}-{DateTime.Now.Month}";

private string WBPartA() => "acme-";

private int WBPartB(int min, int max)
{
    var rngCrypto = new RNGCryptoServiceProvider();
    var bf = new byte[4];

    rngCrypto.GetBytes(bf);
    var result = BitConverter.ToInt32(bf, 0);

    return new Random(result).Next(min, max);
}
```

In the form load of the tracking application, we then make a call to the `GenerateWaybill()` method and pass it the other two methods `WBPartA()` and `WBPartB()` as parameters as seen in Listing 3-2.

Listing 3-2. Form Load

```
private void Form1_Load(object sender, EventArgs e)
{
    var frmLogin = new Login();
    _ = frmLogin.ShowDialog();

    txtWaybill.Text = GenerateWaybill(WBPartA(),
    WBPartB(100,2000));
}
```

If you had placed a breakpoint on the line of code that contains the `GenerateWaybill()` method and step into the methods by pressing F11, you would first step into method `WBPartA()`, then into method `WBPartB()` and lastly into the `GenerateWaybill()` method.

Did you know that you can choose which method to step into? When the breakpoint is hit, hold down Alt+Shift+F11 and Visual Studio will pop up a menu for you to choose from as seen in Figure 3-5.

Figure 3-5. *Step Into Specific*

Simply select the method you want to step into and off you go.

Run to Click

When you start debugging and you hit a breakpoint, you can jump around quickly within the code by clicking the Run to Click button. While in the debugger, hover your mouse over a line of code as seen in Figure 3-6, and click the Run to Click button that pops up.

```
1 reference | Dirk Strauss, Less than 5 minutes ago | 1 author, 1 change
private void BtnTrack_Click(object sender, EventArgs e)
{
    if (!(string.IsNullOrWhiteSpace(txtWaybill.Text)))
    {
        var waybillNum = txtWaybill.Text;
        if (WaybillValid())
        {
            var package = new Package(waybillNum);
            var packLoc = package.TrackPackage();
            if (packLoc != null)
            Run execution to here
                    tails.Text = $"Package location: " +
                        $"{packLoc.LocationName} with coordinates " +
                        $"Long: {packLoc.Long} and " +
                        $"Lat: {packLoc.Lat}";
            }
        }
        else
        {
            _ = MessageBox.Show("You have entered an invalid Waybill number");
        }
    }

}
```

Hover mouse and click

Figure 3-6. *Run to Click*

This will advance the debugger to the line of code where you clicked, allowing you to continue stepping through the code from the new location. Quite handy if you do not want to be pressing F10 a gazillion times.

Run to Cursor

Run to Cursor works in a similar fashion to Run to Click. The difference being that with Run to Cursor you are not debugging. With the debugger stopped, you can right-click a line of code and click Run to Cursor from the context menu as seen in Figure 3-7.

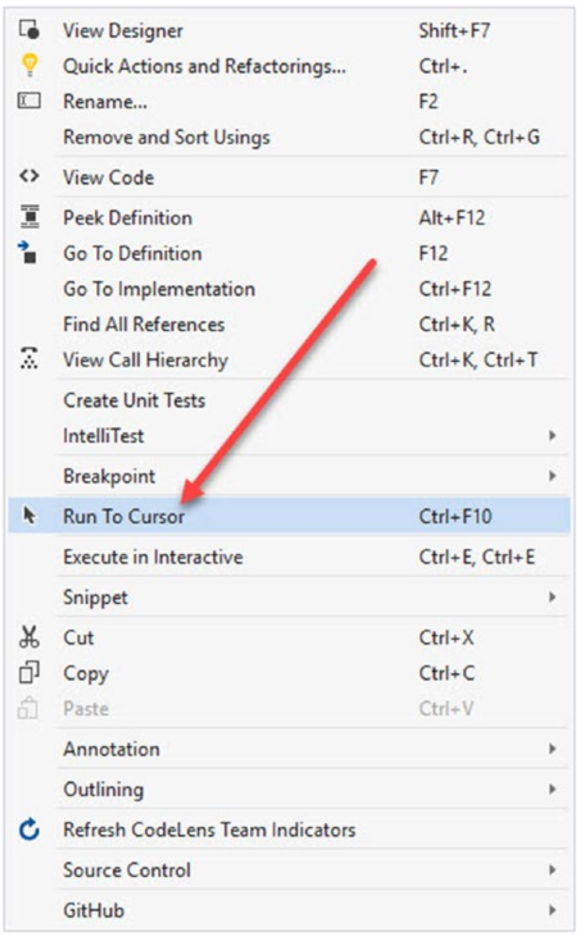


Figure 3-7. *Run to Cursor*

Doing this will start the debugger and set a temporary breakpoint on the line you right-clicked on. This is useful for quickly setting a breakpoint and starting the debugger at the same time. When you reach the breakpoint, you can continue debugging as normal.

Be aware though that you will be hitting any other breakpoints set before the temporary breakpoint first. So you will need to keep on pressing F5 until you reach the line of code you set the temporary breakpoint on.

Conditional Breakpoints and Actions

Sometimes you need to use a condition to catch a bug. Let's say that you are in a for loop, and the bug seems to be data related. The erroneous data only seems to enter the loop after several hundred iterations. If you set a regular breakpoint, you will be pressing F10 until your keyboard stops working.

This is a perfect use case for using conditional breakpoints. You can now tell the debugger to break when a specific condition is true. To set a conditional breakpoint, right-click the breakpoint, and click Conditions from the context menu as seen in Figure 3-8.

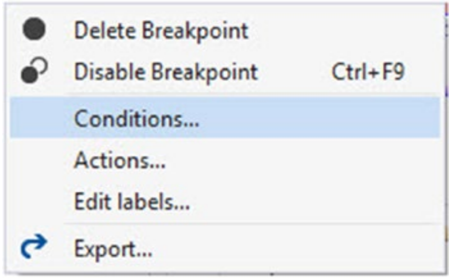


Figure 3-8. *Breakpoint context menu*

You can now select a conditional expression and select to break if this condition is true or when changed as seen in Figure 3-9.

We will discuss Actions shortly.

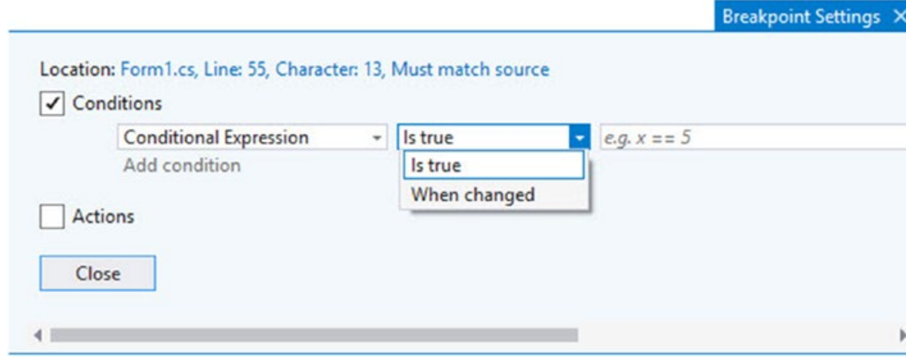


Figure 3-9. *Conditional Expression*

You can also select to break when the Hit Count is equal to, a multiple of, or greater or equal to a value you set as seen in Figure 3-10.

Breakpoint Settings
Location: Form1.cs, Line: 55, Character: 13, Must match source
Conditions
Hit Count
=
e.g. 5
Add condition
=
Is a multiple of
>=
Actions
Close

Figure 3-10. *Hit Count Condition*

The last condition you can set on a conditional breakpoint is a Filter as seen in Figure 3-11.

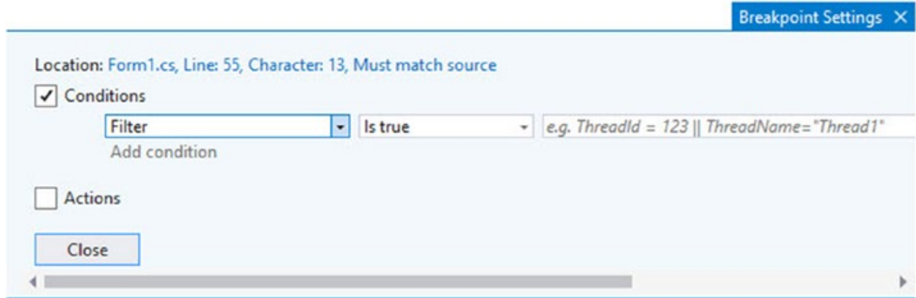


Figure 3-11. *Filter Condition*

You will have noticed the Actions checkbox from the Breakpoint Settings. You will also see the Actions menu on the context menu in Figure 3-8. Here you can add an expression to log to the Output Window using specific keywords that are accessed using the $ symbol.

The special keywords are as follows:

- $ADDRESS – Current Instruction
- $CALLER – Previous function name
- $CALLSTACK – Call stack
- $FILEPOS – The current file and line position
- $FUNCTION – Current function name
- $PID – Process ID
- $PNAME – Process name
- $TICK – Milliseconds elapsed since the system was started, up to 49.7 days
- $TID – Thread ID
- $TNAME – Thread name

You can now use these special keywords to write an entry to the Output Window. You can include the value of a variable by placing it between curly braces (think of Interpolated Strings). Listing 3-3 shows an example of an expression that uses the `$FUNCTION` keyword.

Listing 3-3. Action Expression

```
The value of the counter = {iCount} in $FUNCTION
```

Placing this Breakpoint Action in the constructor of the `Login()` form of the `ShipmentLocator` application will be indicated by a diamond instead of a circle as seen in Figure 3-12.

Figure 3-12. *Breakpoint Action*

When you run your application, you will see the expression output in the Output Window as seen in Figure 3-13.

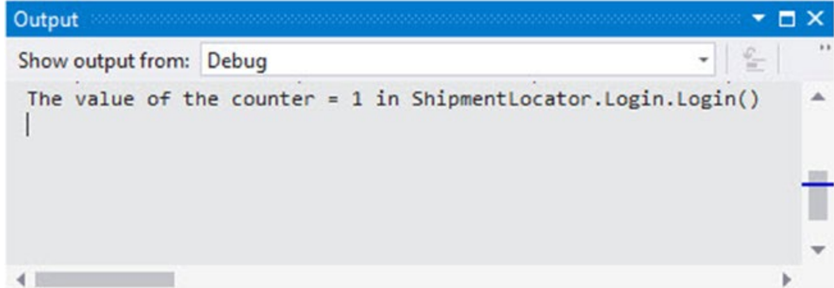


Figure 3-13. *Action Expression in Output Window*

This is great for debugging because if you don't select a condition, the Action will be displayed in the Output Window without hitting the breakpoint and pausing the code. The Breakpoint Action can be seen in Figure 3-14.

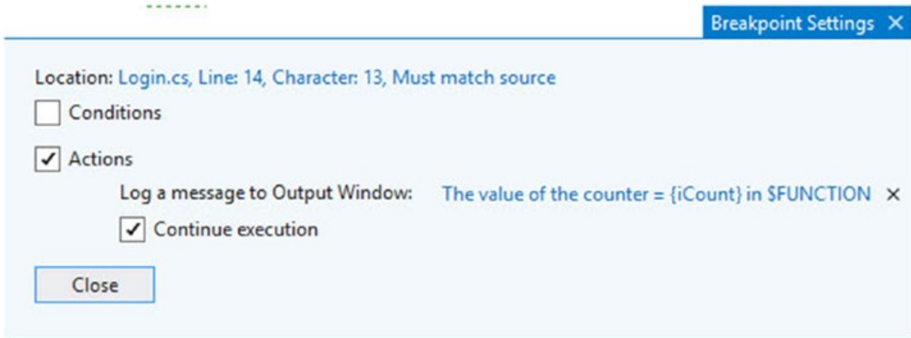


Figure 3-14. *The Breakpoint Action*

If you want to pause the code execution, then you need to uncheck the Continue execution checkbox.

Manage Breakpoints with Labels

As you continue debugging your application, you will be setting many breakpoints throughout the code. Different developers have different ways of debugging. Personally, I add and remove breakpoints as needed, but some developers might end up with a lot of set breakpoints as seen in Figure 3-15.

```
11
   1 reference | Dirk Strauss, 18 hours ago | 1 author, 1 change
12 private void Form1_Load(object sender, EventArgs e)
13 {
14     var frmLogin = new Login();
15     _ = frmLogin.ShowDialog();
16
17     txtWaybill.Text = GenerateWaybill(WBPartA(), WBPartB(100,2000));
18 }
19
   1 reference | Dirk Strauss, 18 hours ago | 1 author, 1 change
20 private void BtnTrack_Click(object sender, EventArgs e)
21 {
22     if (!(string.IsNullOrWhiteSpace(txtWaybill.Text)))
23     {
24         var waybillNum = txtWaybill.Text;
25         if (WaybillValid())
26         {
27             var package = new Package(waybillNum);
28             var packLoc = package.TrackPackage();
29             if (packLoc != null)
30             {
31                 txtLocationDetails.Text = $"Package location: " +
32                     $"{packLoc.LocationName} with coordinates " +
33                     $"Long: {packLoc.Long} and " +
34                     $"Lat: {packLoc.Lat}";
35             }
36         }
```

Figure 3-15. *Many Breakpoints Set*

This is where the Breakpoints window comes in handy. Think of it as mission control for managing complex debugging sessions. This is especially helpful in large solutions where you might have many breakpoints set at various code files throughout your solution.

The Breakpoints window allows developers to manage the breakpoints that they have set by allowing them to search, sort, filter, enable, disable, and delete breakpoints. The Breakpoints window also allows developers to specify conditional breakpoints and actions.

To open the Breakpoints window, click the Debug menu, Windows, and then Breakpoints. You can also press Ctrl+D, Ctrl+B. The Breakpoints window will now be displayed as seen in Figure 3-16.

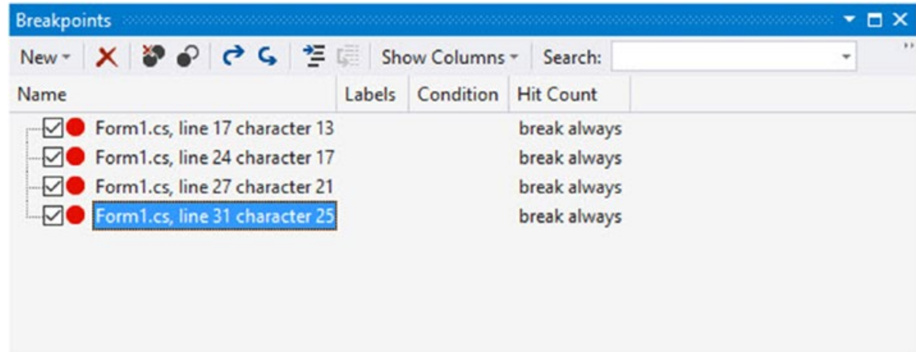


Figure 3-16. *Breakpoints window*

Compare the line numbers of the breakpoints listed in Figure 3-16 with the breakpoints displayed in Figure 3-15. You will see that this accurately reflects the breakpoints displayed in the Breakpoints window.

The only problem with this window is that it doesn't help you much in the way of managing your breakpoints. At the moment, the only information displayed in the Breakpoints window is the class name and the line number.

This is where breakpoint labels are very beneficial. To set a breakpoint label, right-click a breakpoint, and click Edit labels from the context menu as seen in Figure 3-17.

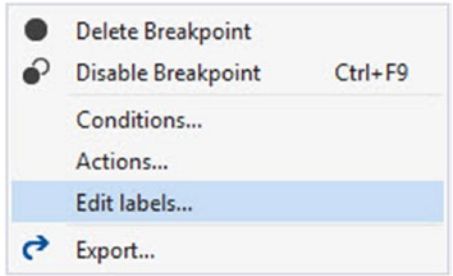


Figure 3-17. *Edit Breakpoint Labels*

The Edit breakpoint labels window is then displayed as seen in Figure 3-18.

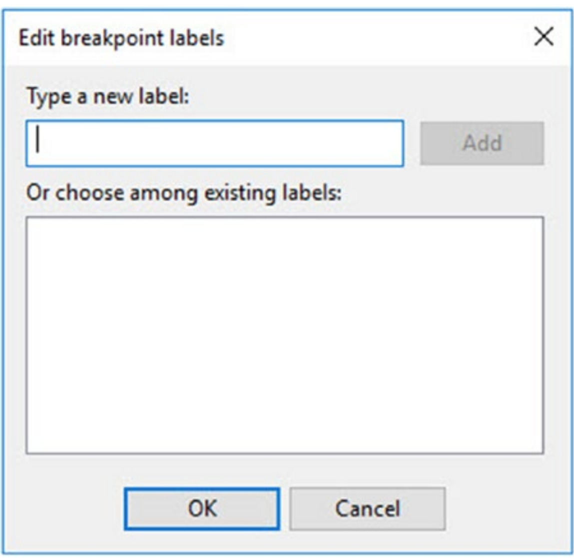


Figure 3-18. *Add a new breakpoint label*

You can type in a new label or choose from any of the existing labels available. If you swing back to the Breakpoints window, you will see that these labels are displayed, making the identification and management of your breakpoints much easier.

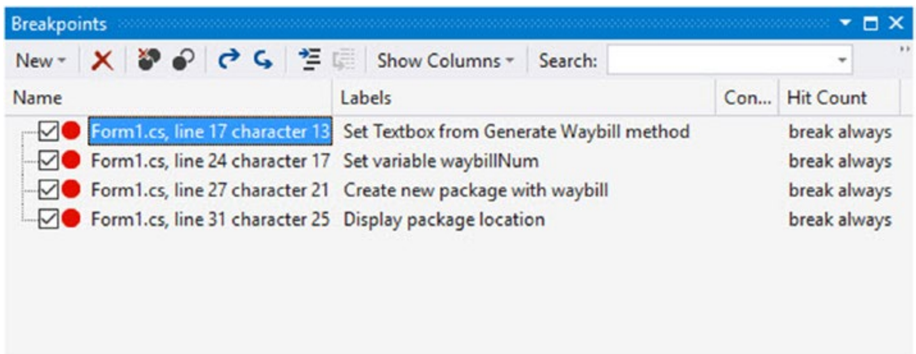


Figure 3-19. *Breakpoints window with Labels Set*

You are in a better position now with the breakpoint labels set to manage your breakpoints more effectively.

Exporting Breakpoints

If you would like to save the current state and location of the breakpoints you have set, Visual Studio allows you to export and import breakpoints. This will create an XML file with the exported breakpoints that you can then share with a colleague.

I foresee the use of Visual Studio Live Share replacing the need to share breakpoints with a colleague just for the sake of aiding in debugging an application. There are, however, other situations I can see exporting breakpoints as being beneficial.

To export your breakpoints, you can right-click a breakpoint and click Export from the context menu, or you can click the export button in the Breakpoints window. You can also import breakpoints from the Breakpoints window by clicking the export or import button as highlighted in Figure 3-20.

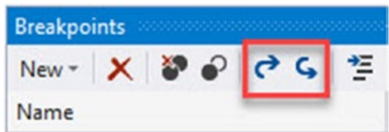


Figure 3-20. *Import or Export Breakpoints*

I'm not too convinced that the icons used on the import and export buttons are indicative of importing and exporting something, but that is just my personal opinion.

Using DataTips

DataTips in Visual Studio allows developers to view information about variables during a debug session. You can only view DataTips in break mode, and DataTips only work with variables that are currently in scope.

This means that before you are able to see a DataTip, you are going to have to debug your code. Place a breakpoint somewhere in your code and start debugging. When you hit the breakpoint that you have set, you can hover your mouse cursor over a variable. The DataTip now appears showing the name of the variable and the value it currently holds. You can also pin this DataTip as seen in Figure 3-21.

Figure 3-21. *Debugger DataTip*

When you pin a DataTip, a pin will appear in the gutter next to the line number. You can now move this DataTip around to another position on the screen. If you look below the pin icon on the DataTip, you will see a "double down-arrow" icon. If you click this, you are able to add a comment to your DataTip as seen in Figure 3-22.

```
var waybillNum = txtWaybill.Text;
if (WaybillValid())
{
    var package = new Package(waybillNum);
```

waybillNum "acme--1456-2019-8"

This is the generated Waybill number

Figure 3-22. *DataTip Comment*

DataTips also allow you to edit the value of the variable, as long as the value isn't a read-only value. To do this, simply select the value in the DataTip and enter a new value. Then press the Enter key to save the new value.

Visualizing Complex Data Types

DataTips also allows you to visualize complex data in a more meaningful way. To illustrate this, we will need to write a little bit of code. We are going to create a class, then create a list of that class, and then create a data table from that list that we will view in the DataTip. I have just created a small Console Application. Start off by creating the class in Listing 3-4.

Listing 3-4. The Subject Class

```
public class Subject
{
    public int SubjectCode { get; set; }
    public string SubjectDescription { get; set; }
}
```

We are going to create a list of the Subject class. Before we do this, however, we need to write the code that is going to create a DataTable of the values in List<Subject>. This code is illustrated in Listing 3-5.

Listing 3-5. Convert List to DataTable

```
static DataTable ConvertListToDataTable<T>(List<T> list)
{
    var table = new DataTable();
    var properties = typeof(T).GetProperties();

    foreach (var prop in properties)
    {
        _ = table.Columns.Add(prop.Name);
    }

    foreach (var item in list)
    {
        var row = table.NewRow();
        foreach (var property in properties)
        {
            var name = property.Name;
            var value = property.GetValue(item, null);

            row[name] = value;
        }
        table.Rows.Add(row);
    }

    return table;
}
```

In the `Main` method, add the code in Listing 3-6. We will then place a breakpoint on the call to the `ConvertListToDataTable()` method and step over that so that we can inspect the `table` variable's DataTip.

Listing 3-6. Create the List<Subject> and the DataTable

```
static void Main(string[] args)
{
    var lstSubjects = new List<Subject>();
    for (var i = 0; i <= 5; i++)
    {
        var sub = new Subject();
        sub.SubjectCode = i;
        sub.SubjectDescription = $"Subject-{i}";
        lstSubjects.Add(sub);
    }

    var table = ConvertListToDataTable<Subject>(lstSubjects);
}
```

When you hover over the `table` variable, you will see that the DataTip displays a magnifying glass icon as seen in Figure 3-23.

Figure 3-23. *The table variable DataTip*

If you click the magnifying glass icon, you will see the contents of the `table` variable displayed in a nice graphical way as seen in Figure 3-24.

DataTable Visualizer

SubjectCode	SubjectDescription
0	Subject-0
1	Subject-1
2	Subject-2
3	Subject-3
4	Subject-4
5	Subject-5

Close

Figure 3-24. *DataTable Visualizer*

The magnifying glass icon tells us that one or more visualizers are available for the particular variable. In this example, the DataTable Visualizer.

Bonus Tip

If you are feeling adventurous, pin the DataTip that is displayed when hovering over the `table` variable, and right-click the pinned DataTip. You can now copy the value, copy the expression, add a new expression, and remove the expression previously added. Go ahead and add the following expression in Listing 3-7.

Listing 3-7. Add a DataTip Expression

```
table.Rows.Count
```

This is great if you forgot to add a variable watch or just want to see some additional info regarding the variable in the DataTip.

Using the Watch Window

The Watch window allows us to keep track of the value of one or more variables and also allows us to see how these variable values change as one steps through the code.

You can easily add a variable to the Watch window by right-clicking the variable and selecting Add Watch from the context menu. Doing this with the `table` variable in the previous section will add it to the Watch 1 window as illustrated in Figure 3-25.

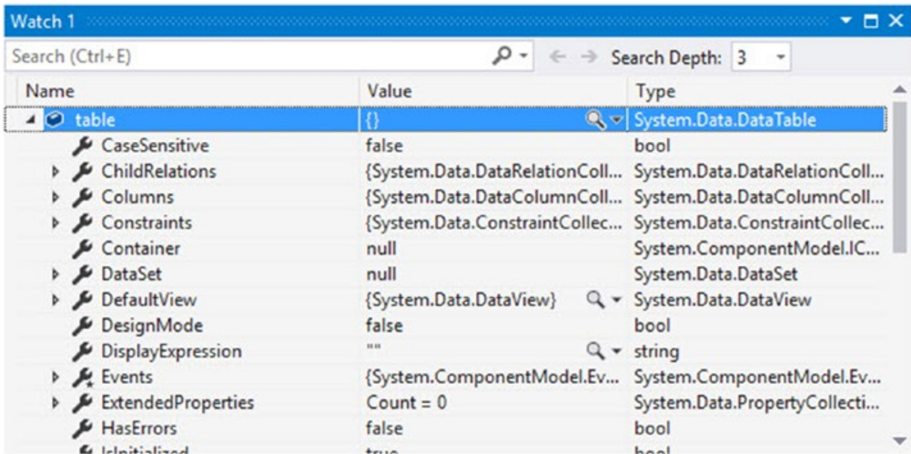


Figure 3-25. *The Watch 1 window*

Here you can open the visualizer by clicking the magnifying glass icon or expand the `table` variable to view the other properties of the object. I use the Watch window often as it is a really convenient way to keep track of several variables at once.

The DebuggerDisplay Attribute

In the previous section, we discussed how to add a variable to the Watch window in Visual Studio. We saw that we can view the value of a variable or variables easily from this single window.

Using the same code we wrote in the previous section, add the variable called `lstSubjects` to the Watch window, and expand the variable. You will see the values of the `lstSubjects` variable listed as `{VisualStudioDebugging.Subject}` in the Value column as seen in Figure 3-26.

Watch 1

Search (Ctrl+E) Search Depth: 3

Name	Value	Type
lstSubjects	Count = 6	System.Collections.Generic.List<VisualStudioDebug...
[0]	{VisualStudioDebugging.Subject}	VisualStudioDebugging.Subject
[1]	{VisualStudioDebugging.Subject}	VisualStudioDebugging.Subject
[2]	{VisualStudioDebugging.Subject}	VisualStudioDebugging.Subject
[3]	{VisualStudioDebugging.Subject}	VisualStudioDebugging.Subject
[4]	{VisualStudioDebugging.Subject}	VisualStudioDebugging.Subject
[5]	{VisualStudioDebugging.Subject}	VisualStudioDebugging.Subject
Raw View		

Figure 3-26. *The lstSubjects Variable Values*

To view the values of each item in the list, we need to expand the list item (Figure 3-27) and inspect the values.

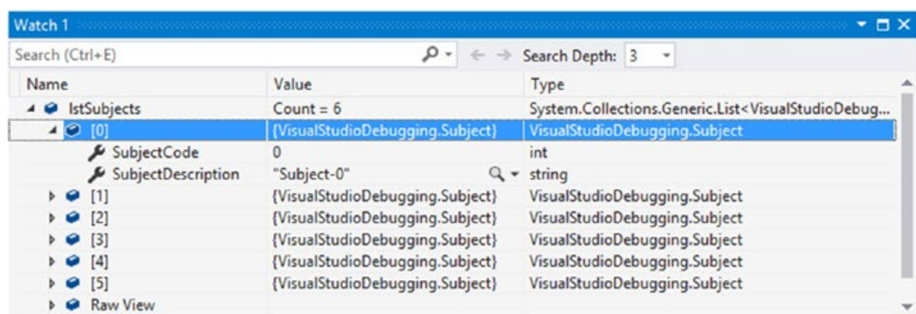


Figure 3-27. *View list items values*

This will quickly become rather tedious, especially when you are dealing with a rather large list and you are looking for a specific value.

This is where the DebuggerDisplay attribute comes into play. We are going to modify the Subject class.

Ensure that you add the statement using System.Diagnostics to your code file.

Modify your Subject class as in Listing 3-8.

Listing 3-8. Modified Subject Class

```
[DebuggerDisplay("Code: {SubjectCode, nq}, Subject:
{SubjectDescription, nq}")]
public class Subject
{
    public int SubjectCode { get; set; }
    public string SubjectDescription { get; set; }
}
```

Start debugging your code again, and have a look at your Watch window after adding the DebuggerDisplay attribute. Your item values are more readable as seen in Figure 3-28.

Watch 1

Search (Ctrl+E) Search Depth: 3

Name	Value	Type
lstSubjects	Count = 6	System.Collections.Generic.List<VisualStudioDebug...
[0]	Code: 0, Subject: Subject-0	VisualStudioDebugging.Subject
[1]	Code: 1, Subject: Subject-1	VisualStudioDebugging.Subject
[2]	Code: 2, Subject: Subject-2	VisualStudioDebugging.Subject
[3]	Code: 3, Subject: Subject-3	VisualStudioDebugging.Subject
[4]	Code: 4, Subject: Subject-4	VisualStudioDebugging.Subject
[5]	Code: 5, Subject: Subject-5	VisualStudioDebugging.Subject
Raw View		

Figure 3-28. *The lstSubjects Variable Values with DebuggerDisplay*

The use of "nq" in the DebuggerDisplay attribute will remove the quotes when the final value is displayed. The "nq" means "no quotes."

Evaluate Functions Without Side Effects

While debugging an application, we probably do not want the state of the application to change because of an expression we are evaluating. It is, unfortunately, a fact that evaluating some expressions might cause side effects.

To illustrate this, we will need to write some more code. We will be creating a class called Student that contains a List of Subject as seen in Listing 3-9.

Listing 3-9. The Student Class

```
public class Student
{
    private List<Subject> _subjectList;
    public Student() { }
    public Student(List<Subject> subjects) => _subjectList =
    subjects;
    public bool HasSubjects() => _subjectList != null;

    public List<Subject> StudentSubjects
    {
        get
        {
            if (_subjectList == null)
            {
                _subjectList = new List<Subject>();
            }
```

```
                return _subjectList;
            }
        }
}
```

In this class, we have a HasSubjects() method that simply returns a Boolean indicating if the Student class contains a list of subjects. We also have a property called StudentSubjects that returns the list of subjects. If the list of subjects is null, it creates a new instance of List<Subject>.

It is here that the side effect is caused. If the HasSubjects() method returns false, calling the StudentSubjects property will change the value of HasSubjects().

This is better illustrated in the following screenshots. Create a new instance of Student, and place a breakpoint right after that line of code (Figure 3-29).

Figure 3-29. *Place Breakpoint after Student*

If we now use the Watch window to look at the value returned by the HasSubjects() method, we will see that it returns false (Figure 3-30).

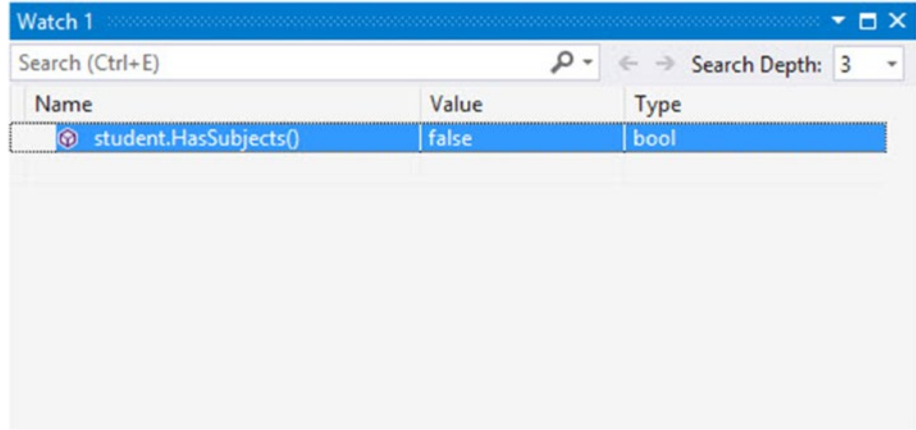


Figure 3-30. *HasSubjects() method returns false*

When we call the `StudentSubjects` property, we see this side effect come into play in Figure 3-31. As soon as this property is called, the value of the `HasSubjects()` method changes.

This means that the state of our `Student` class has changed because of an expression that we ran in the Watch window.

This can cause all sorts of issues further down the debugging path, and sometimes the change might be so subtle that you don't even notice it. You could end up chasing "bugs" that never really were bugs to begin with.

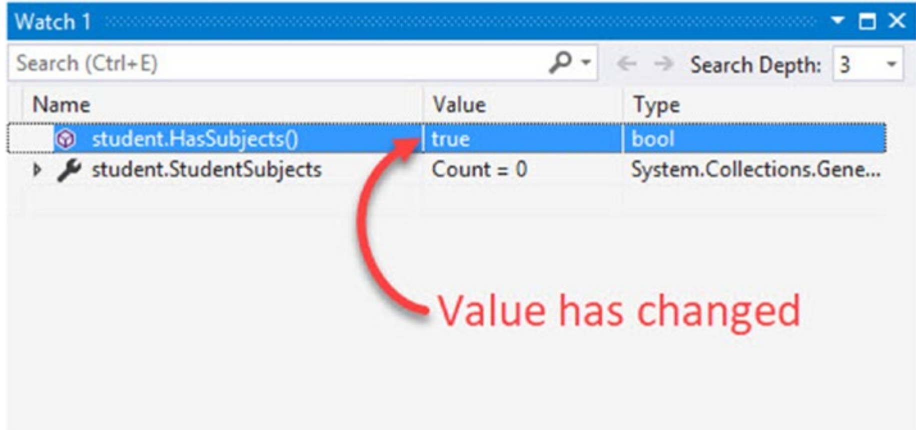


Figure 3-31. *HasSubjects() method value has changed*

To prevent any side effects from an expression just add, `nse` to the end of the expression as seen in Figure 3-32.

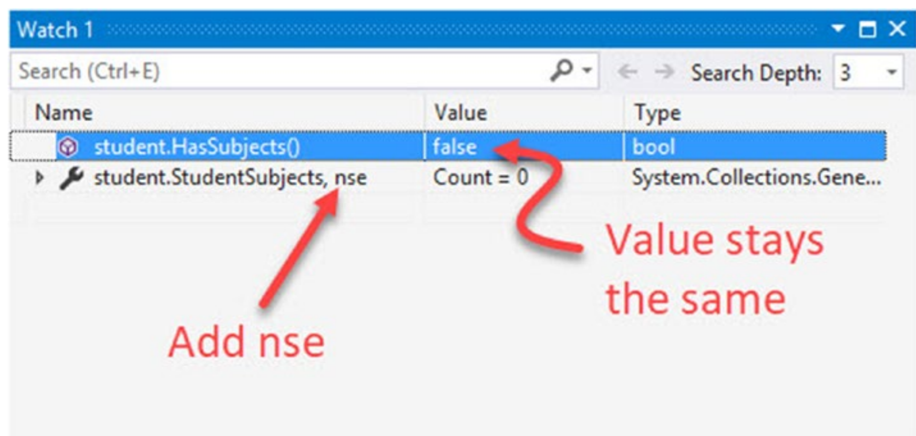


Figure 3-32. *Adding nse to Expression to Evaluate*

This time, the value of the `HasSubjects()` method remains the same which means that the state of your class remains unchanged. As you have probably guessed by now, the `nse` added after the expression stands for "No Side Effects."

Format Specifiers

Format specifiers allow you to control the format in which a value is displayed in the Watch window. Format specifiers can also be used in the Immediate and Command window. Using a format specifier is as easy as entering the variable expression and typing a comma followed by the format specifier you want to use. The following are the C# format specifiers for the Visual Studio debugger:

ac

Force evaluation of an expression decimal integer

d

Decimal integer

dynamic

Displays the specified object using a Dynamic View

h

Hexadecimal integer

nq

String with no quotes

nse

Evaluates expressions without side effects where “nse” means “No Side Effects”

hidden

Displays all public and nonpublic members

raw

Displays item as it appears in the raw node. Valid on proxy objects only

results

Used with a variable that implements IEnumerable or IEnumerable<T>. Displays only members that contain the query result

You will recall that we used the "nq" format specifier with the DebuggerDisplay attribute discussed in a previous section.

Diagnostic Tools

Visual Studio gives developers access to performance measurement and profiling tools. The performance of your application should, therefore, be high on your priority list. An application that suffers from significant performance issues is as good as broken (especially from an end user's perspective).

Visual Studio Diagnostic Tools might be enabled by default. If not, enable Diagnostic Tools by going to the Tools menu and clicking Options, Debugging, and then General. Ensure that Enable Diagnostic Tools while debugging is checked as seen in Figure 3-33.

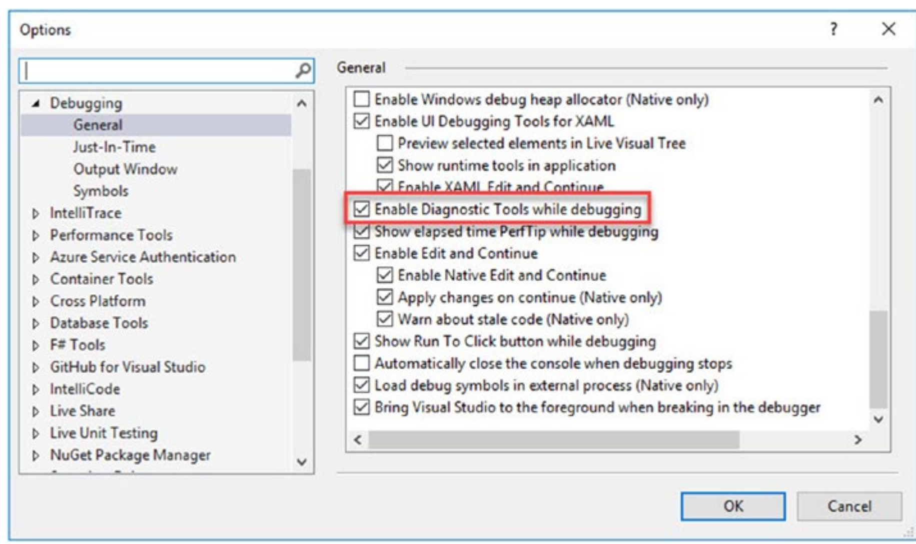


Figure 3-33. *Enable Diagnostic Tools*

This will ensure that the Diagnostic Tools window opens automatically when you start debugging. When we start debugging our ShipmentLocator Windows Forms application, the Diagnostic Tools window will be displayed as seen in Figure 3-34.

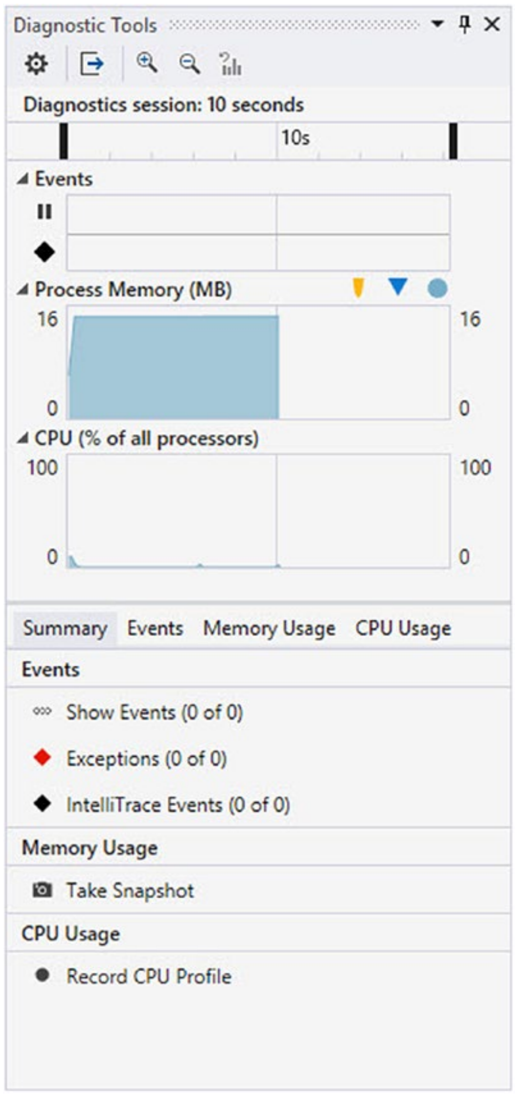


Figure 3-34. *Diagnostic Tools*

With our Windows Forms application, you can use Diagnostic Tools to monitor memory or CPU usage as seen in Figure 3-35.

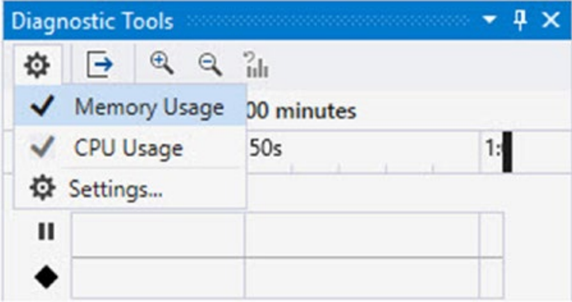


Figure 3-35. *Select what to analyze*

As you debug your application, you can see the CPU usage, memory usage, and other performance-related information.

CPU Usage

A great place to start your performance analysis is the CPU Usage tab. Place two breakpoints in your `Form1_Load` at the start and end of the function (Figure 3-36), and start debugging.

```
1 reference | Dirk Strauss, 5 days ago | 1 author, 1 change
private void Form1_Load(object sender, EventArgs e)
{
    var frmLogin = new Login();
    _ = frmLogin.ShowDialog();

    txtWaybill.Text = GenerateWaybill(WBPartA(), WBPartB(100,2000));
    this.Text = $"Shipment Locator - {DateTime.Now.Year}";
}
```

Figure 3-36. *Setting breakpoints to analyze CPU Usage*

When the debugger reaches the second breakpoint, you will have detailed profiling data for the region of code you analyzed as seen in Figure 3-37.

Summary Events Memory Usage CPU Usage

Open details... Search

Function Name	Total CPU [unit,...
ShipmentLocator.exe (PID: 46152)	425 (100.00%)
ShipmentLocator.Program::Main	298 (70.12%)
[External Call] System.Windows.Forms.Application.Run(Syste...	168 (39.53%)
ShipmentLocator.Form1::.ctor	129 (30.35%)
[External Call] System.Resources.ResourceManager.GetObjec...	70 (16.47%)
[External Call] System.Windows.Forms.Form..ctor()$##60021D8	34 (8.00%)
ShipmentLocator.Form1::InitializeComponent	13 (3.06%)
[External Call] System.Windows.Forms.TextBox..ctor()$##600...	11 (2.59%)
ShipmentLocator.Form1::Form1_Load	9 (2.12%)
[External Call] System.Windows.Forms.Control.set_Location(S...	7 (1.65%)
ShipmentLocator.Login::BtnLogin_Click	7 (1.65%)
[External Call] System.Windows.Forms.Form+ControlCollecti...	6 (1.41%)
ShipmentLocator.Login::.ctor	4 (0.94%)
[Broken]	3 (0.71%)
[External Call] System.DateTime.get_Now()$##600061C	2 (0.47%)
ShipmentLocator.Form1::GenerateWaybill	2 (0.47%)
[External Call] System.Windows.Forms.Application.EnableVis...	1 (0.24%)
[External Call] System.Windows.Forms.Form.Close()$##60022...	1 (0.24%)

Figure 3-37. *CPU Usage Analysis Results*

If any of the functions in the CPU Usage pane seem to be problematic, double-click the function to view a more detailed three-pane view of the analysis. As seen in Figure 3-38, the left pane will contain the calling function, the middle will contain the selected function, and any called functions will be displayed in the right pane.

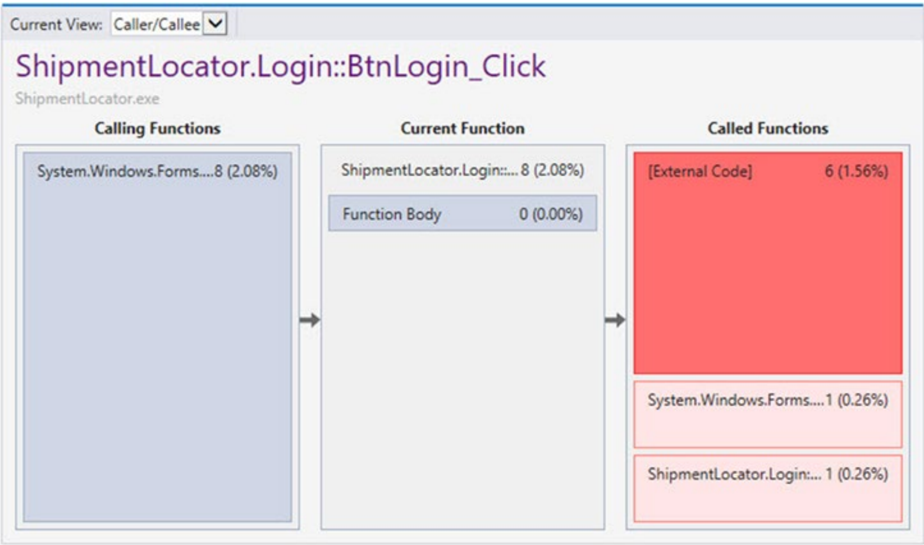


Figure 3-38. *Butterfly View of BtnLogin_Click*

In the Current Function pane, you will see the Function Body section which details the time spent in the function body. Because this excludes the calling and called functions, you get a better understanding of the function you are evaluating and can determine if it is the performance bottleneck or not.

Memory Usage

Visual Studio Diagnostic Tools allows developers to see what the change in memory usage is. This is done by taking snapshots. When you start debugging, place a breakpoint on a method you suspect is causing a memory issue. Then you step over the method and place another breakpoint. An increase is indicated with a red up arrow as seen in Figure 3-39.

Figure 3-39. Memory Usage Snapshots

This is often the best way to analyze memory issues. Two snapshots will give you a nice diff and allow you to see exactly what has changed.

Figure 3-40. Comparing Snapshots

You can also compare two snapshots by clicking one of the links in the Memory Usage snapshots (Figure 3-39) and viewing the comparison in the snapshot window that opens up (Figure 3-40). By selecting a snapshot in the Compare to drop-down list, you are able to see what has changed.

The Events View

As you step through your application, the Events view will show you the different events that happen during your debug session. This can be setting a breakpoint or stepping through code. It also shows you the duration of the event as seen in Figure 3-41.

Summary Events Memory Usage CPU Usage

Filter | Filter Events

Event	Time	Duration	Thread
Breakpoint: Main, Program.cs line 17	0.10s	107ms	[41200] Main Thread
Breakpoint: Main, Program.cs line 25	0.10s	1ms	[41200] Main Thread
Step: Main, Program.cs line 27	0.11s	9ms	[41200] Main Thread

Figure 3-41. *The Events View*

This means that as you step through your code, the Events tab will display the time the code took to run from the previous step operation to the next. You can also see the same events displayed as PerfTips in the Visual Studio Code Editor as seen in Figure 3-42.

```
24
25            var table = ConvertListToDataTable<Subject>(lstSubjects);
26
27            var student = new Student(); ≤25ms elapsed
28            if (student.HasSubjects())
29            {
30
31            }
32
33
```

```
Click to open the Diagnostic Tools window.

Up to 25ms elapsed during the previous step.
This value is an estimate and includes debug overhead.
```

Figure 3-42. *PerfTips in the Code Editor*

IntelliTrace events are available in this tab if you have Visual Studio Enterprise.

For a comparison of the Visual Studio 2019 Editions, head on over to `https://visualstudio.microsoft.com/vs/compare/` and see what each edition has to offer.

Perftips allows developers to quickly identify potential issues in your code.

The Right Tool for the Right Project Type

The following table shows which tool Visual Studio offers and the project types that can make use of these tools.

Table 3-1. *Performance Tools for Project Types*

Performance Tool	Windows Desktop	UWP	ASP.NET/ASP.NET Core
.CPU Usage	Yes	Yes	Yes
Memory Usage	Yes	Yes	Yes
GPU Usage	Yes	Yes	No
Application Timeline	Yes	Yes	No
PerfTips	Yes	Yes for XAML	Yes
Performance Explorer	Yes	No	Yes
IntelliTrace	VS Enterprise only	VS Enterprise only	VS Enterprise only
Network Usage	No	Yes	No
HTML UI Responsiveness	No	Yes for HTML	No
JavaScript Memory	No	Yes for HTML	No

Immediate Window

The Immediate Window in Visual Studio allows you to debug and evaluate expressions, execute statements, and print the values of variables. If you don't see the Immediate Window, go to the Debug menu, and select Windows, and click Immediate or hold down Ctrl+D, Ctrl+I.

Let's place a breakpoint in one of our `for` loops as seen in Figure 3-43.

```
15        {
16            var lstSubjects = new List<Subject>();
17            for (var i = 0; i <= 5; i++)
18            {
19                var sub = new Subject();
20                sub.SubjectCode = i;
21                sub.SubjectDescription = $"Subject-{i}";
22                lstSubjects.Add(sub);
23            }
```

Figure 3-43. *Breakpoint hit*

Opening up the Immediate Window and typing in `sub.SubjectDescription` will display its value as seen in Figure 3-44. You can also use `? sub.SubjectDescription` to view the value of the variable.

```
Immediate Window
sub.SubjectDescription
"Subject-0"
```

Figure 3-44. *Immediate Window*

If you had entered `sub.SubjectDescription = "Math"` you would be updating the value from “Subject-0” to “Math” as seen in Figure 3-45.

Figure 3-45. *Variable value changed*

You can also execute a function at design time (i.e., while not debugging) using the Immediate Window. Add the code in Listing 3-10 to your project.

Listing 3-10. DisplayMessage Function

```
static string DisplayMessage()
{
    return "Hello World";
}
```

In the Immediate Window, type `?DisplayMessage()` and press Enter. The Immediate Window will run the function and return the result as seen in Figure 3-46.

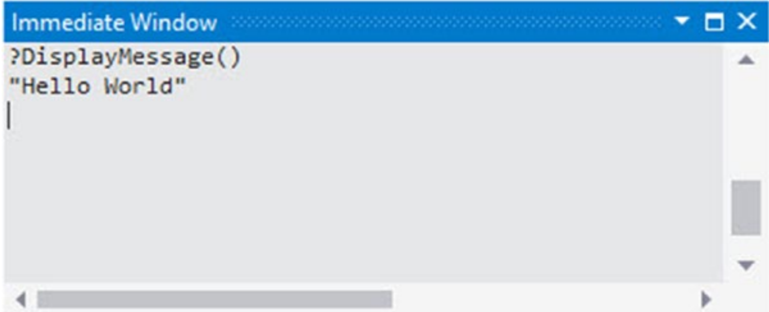


Figure 3-46. *Execute the DisplayMessage Function*

Any breakpoints contained in the function will break the execution at the breakpoint. Use the debugger to examine the program state.

Attaching to a Running Process

Attaching to a process allows the Visual Studio Debugger to attach to a running process on the local machine or a remote computer. With the process you want to debug already running, select Debug and click Attach to Process as seen in Figure 3-47. You can also hold down Ctrl+Alt+P to open the Attach to Process window.

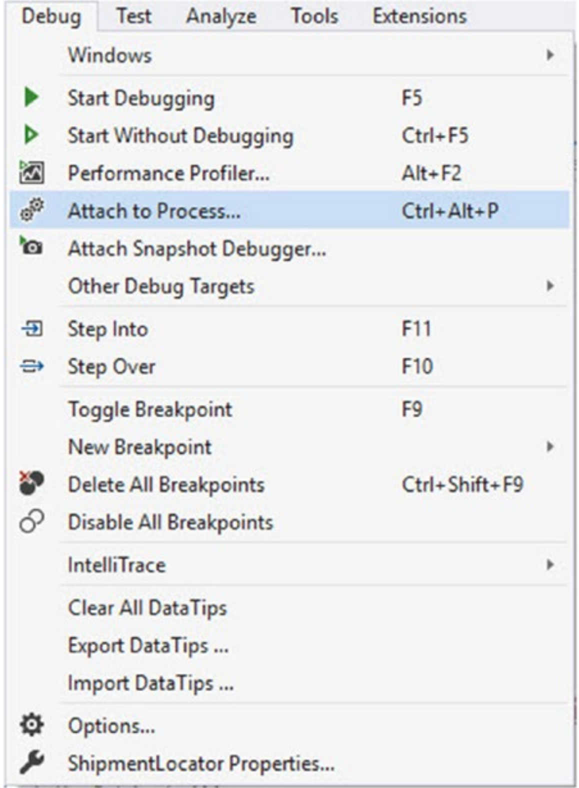


Figure 3-47. *Attach to Process*

The Attach to Process window is then displayed (Figure 3-48). The connection type must be set to Default, and the connection target must be set to your local machine name.

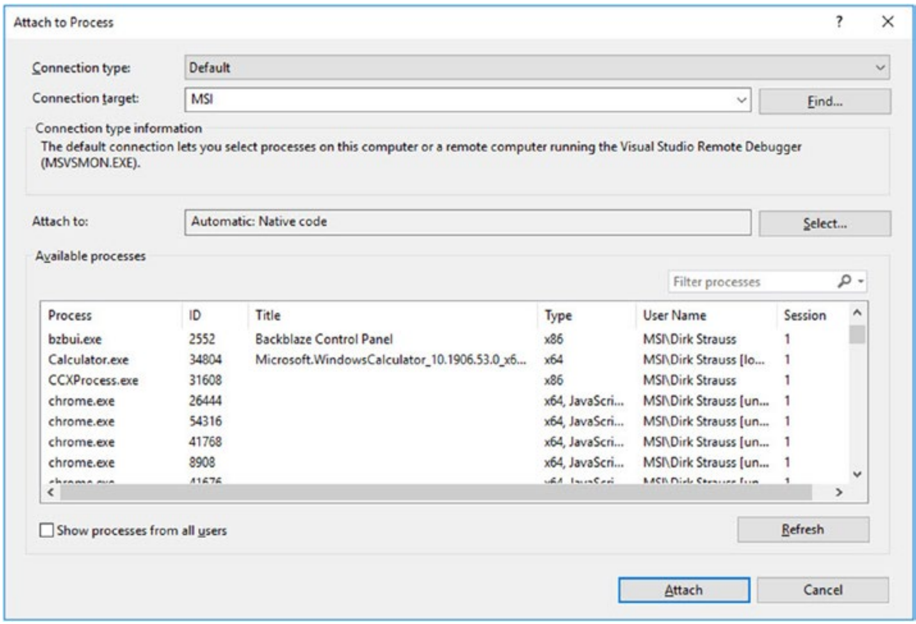


Figure 3-48. *Attach to Process*

The available processes list allows you to select the process you want to attach to. You can quickly find the process you want by typing the name in the filter process text box.

You can, for example, attach to the w3wp.exe process to debug a web application running on IIS. To debug a C#, VB.NET, or C++ application on the local machine, you can use the Attach to Process by selecting the <appname>.exe from the available processes list (where <appname> is the name of your application).

Attach to a Remote Process

To debug a process running on a remote computer, select Debug and click Attach to Process menu, or hold down Ctrl+Alt+P to open the Attach to Process window. This time, select the remote computer name in the

Connection target by selecting it from the drop-down list or typing the name in the connection target text box and pressing Enter.

If you are unable to connect to the remote computer using the computer name, use the IP and port address.

Remote Debugger Port Assignments

The port assignments for the Visual Studio Remote Debugger are as follows:

- Visual Studio 2019: 4024
- Visual Studio 2017: 4022
- Visual Studio 2015: 4020
- Visual Studio 2013: 4018
- Visual Studio 2012: 4016

The port assigned to the Remote Debugger is incremented by 2 for each release of Visual Studio.

Reattaching to a Process

Starting with Visual Studio 2017, you can quickly reattach to a process you previously attached to. To do this, you can click the Debug menu and select Reattach to Process or hold down Shift+Alt+P. The debugger will try to attach to the last process you attached to my matching the previous process ID to the list of running processes. If that fails, it tries to attach to a process by matching the name. If neither is successful, the Attach to Process window is displayed and lets you select the correct process.

Remote Debugging

Sometimes you need to debug an application that has already been deployed to a different computer. Visual Studio allows you to do this via Remote Debugging. To start, you need to download and install remote tools for Visual Studio 2019 on the remote computer.

Remote Tools for Visual Studio 2019 enables app deployment, remote debugging, testing, profiling, and unit testing on computers that don't have Visual Studio 2019 installed.

System Requirements

The supported operating systems for the remote computer must be one of the following:

- Windows 10
- Windows 8 or 8.1
- Windows 7 SP 1
- Windows Server 2016
- Windows Server 2012 or Windows Server 2012 R2
- Windows Server 2008 SP 2, Windows Server 2008 R2 Service Pack 1

The supported hardware configurations to enable remote debugging are detailed in the following list:

- 1.6 GHz or faster processor
- 1 GB of RAM (1.5 GB if running on a VM)
- 1 GB of available hard disk space

- 5400-RPM hard drive
- DirectX 9-capable video card running at 1024 x 768 or higher display resolution

The remote computer and your local machine (the machine containing Visual Studio) must both be connected over a network, workgroup, or homegroup. The two machines can also be connected directly via an Ethernet cable.

Take note that trying to debug two computers connected through a proxy is not supported.

It is also not recommended to debug via a dial-up connection (do those still exist?), or over the Internet across geographical locations. The high latency or low bandwidth will make debugging unacceptably slow.

Download and Install Remote Tools

Connect to the remote machine, and download and install the correct version of the remote tools required for your version of Visual Studio. The link to download the remote tools compatible with all versions of Visual Studio 2019 is `https://visualstudio.microsoft.com/downloads#remote-tools-for-visual-studio-2019`.

If you are using Visual Studio 2017, for example, download the latest update of remote tools for Visual Studio 2017.

Also, be sure to download the remote tools with the same architecture as the remote computer. This means that even if your app is a 32-bit application, and your remote computer is running a 64-bit operating system, download the 64-bit version of the remote tools.

Install the remote tools, and click Install after agreeing to the license terms and conditions (Figure 3-49).

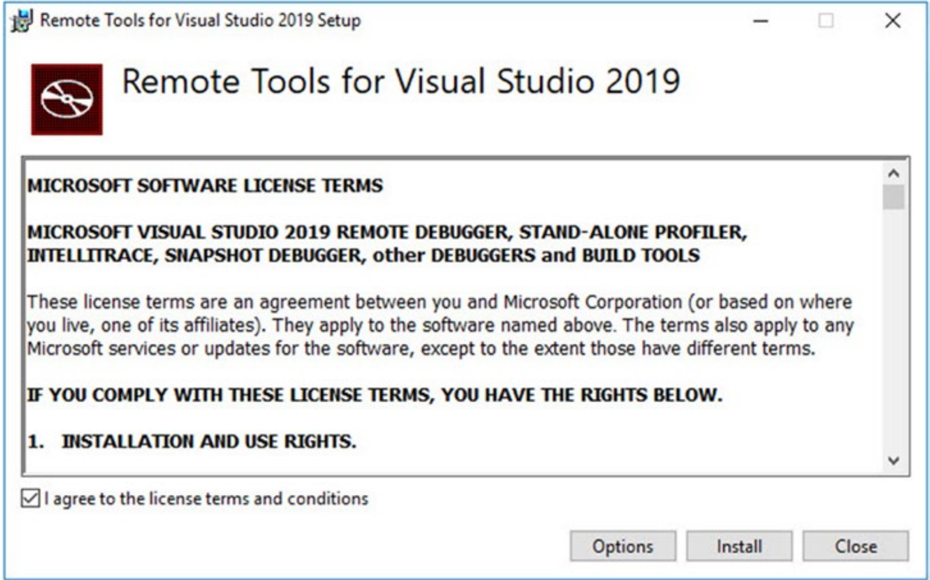


Figure 3-49. *Remote tools for Visual Studio 2019*

Running Remote Tools

After the installation has completed on the remote machine, run the Remote Tools application as Administrator if you can. To do this, right-click the Remote Debugger app, and click Run as Administrator.

At this point, you might be presented with a Remote Debugging Configuration dialog box. I did not encounter this window, but if you do, it possibly means that there is a configuration issue that you need to resolve. The Remote Debugging Configuration dialog box will prompt you to correct configuration errors it picks up.

When the configuration issues have been resolved, click the Configure remote debugging button in the dialog box to open the Remote Debugger window as seen in Figure 3-50.

Visual Studio 2019 Remote Debugger (Administrator)
File Tools Help
Date and Time Description
8/25/2019 2:06:30 AM Msvsmon started a new server named 'DESKTOP-H1MDEFE:4024'. Waiting for new connections.

Figure 3-50. *Visual Studio 2019 Remote Debugger*

You are now ready to start remote debugging your application.

Start Remote Debugging

The great thing about the Remote Debugger on the remote computer is that it tells you the server name to connect to. In Figure 3-50, you can see that the server is named `DESKTOP-H1MDEFE:4024` where `4024` is the port assignment for Visual Studio 2019. Make a note of this server name and port number.

In your application, set a breakpoint somewhere in the code such as in a button click event handler. Now right-click the project in the Solution Explorer, and click Properties. The project properties page opens up as seen in Figure 3-51.

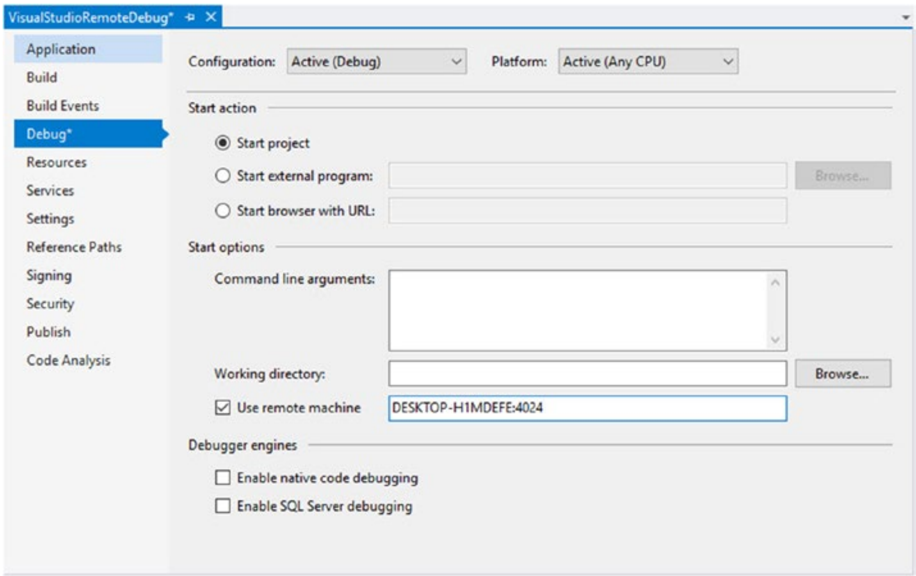


Figure 3-51. *Project properties page*

Now, perform the following steps to remotely debug your application:

1. Click the Debug tab, and check the Use remote machine checkbox, and enter the remote machine name and port noted earlier. In our example, this is `DESKTOP-H1MDEFE:4024`.

2. Make sure that you leave the Working directory text box empty and do not check Enable native code debugging.

3. When all this is done, save the properties and build your project.

4. You now need to create a folder on the remote computer that is exactly the same path as the Debug folder on your local machine (the Visual Studio machine). For example, the path to the project

Debug folder on my local machine is `<source path>ShipmentLocatorApp\VisualStudioRemoteDebug\bin\Debug`. Create this exact same path on the remote machine.

5. Copy the executable that was just created by the build you performed in step 3 to the newly created Debug folder on the remote computer.

Be aware that any changes to your code or rebuilds to your project will require you to repeat step 5.

---

6. Ensure that the Remote Debugger is running on the remote computer. The description should state that it is waiting for new connections.

7. On your local machine, start debugging your application and if prompted to enter the credentials for the remote machine to log on. Once logged on, you will see that the Remote Debugger on the remote computer displays that the remote debug session is now active (Figure 3-52).

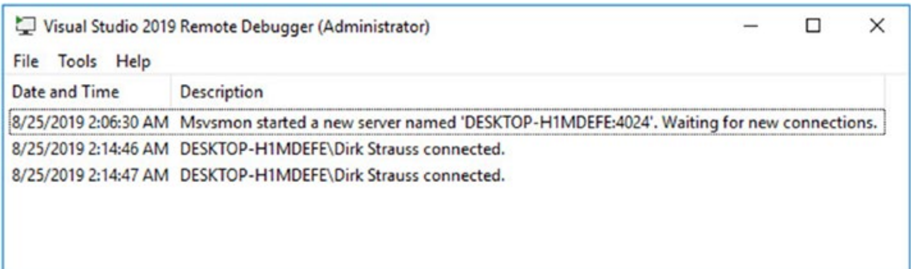


Figure 3-52. *Remote Debug Session Connected*

8. After a few seconds, you will see your application's main window displayed on the remote machine (Figure 3-53). Yep, breakfast is the most important meal of the day.

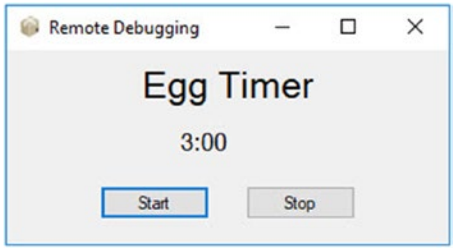


Figure 3-53. *Application main screen*

9. On the remote machine, take whatever action is needed to hit the breakpoint you set earlier. I simply set a breakpoint behind the Start button click event handler. When you hit the breakpoint, it will be hit on your local machine (Visual Studio machine).

If you need any project resources to debug your application, you will have to include these in your project. The easiest way is to create a project folder in Visual Studio and then add the files to that folder. For each resource you add to the folder, ensure that you set the Copy to Output Directory property to Copy always.

CHAPTER 4

Unit Testing

Many developers will have strong opinions on unit testing. If you are considering using unit tests in your code, then start by understanding why unit tests are useful and sometimes necessary.

Breaking down your code's functionality into smaller, testable units of behavior allows you to create and run unit tests. Unit tests will increase the likelihood that your code will continue to work as expected, even though you have made changes to the source code. In this chapter, we will have a look at

- Creating and running unit tests
- Using live unit tests
- Using IntelliTest to generate unit tests
- How to measure Code Coverage in Visual Studio

Unit tests allow you to maintain the health of your code and find errors quickly, before shipping your application to your customers. To introduce you to unit testing, we will start off with a very basic example of creating a unit test.

Creating and Running Unit Tests

Assume that you have a method that calculates the temperature in Fahrenheit for a given temperature in Celsius. The code that we want to create a unit test for will look as in Listing 4-1.

© Dirk Strauss 2020
D. Strauss, *Getting Started with Visual Studio 2019*,
https://doi.org/10.1007/978-1-4842-5449-3_4

Listing 4-1. Convert Celsius to Fahrenheit

```
public static class ConversionHelpers
{
    private const double F_MULTIPLIER = 1.8;
    private const int F_ADDITION = 32;
    public static double ToFahrenheit(double celsius)
    {
        return celsius * F_MULTIPLIER + F_ADDITION;
    }
}
```

We have constant values for the multiplier and addition to the conversion formula. This means that we can easily write a test to check that the conversion is an expected result.

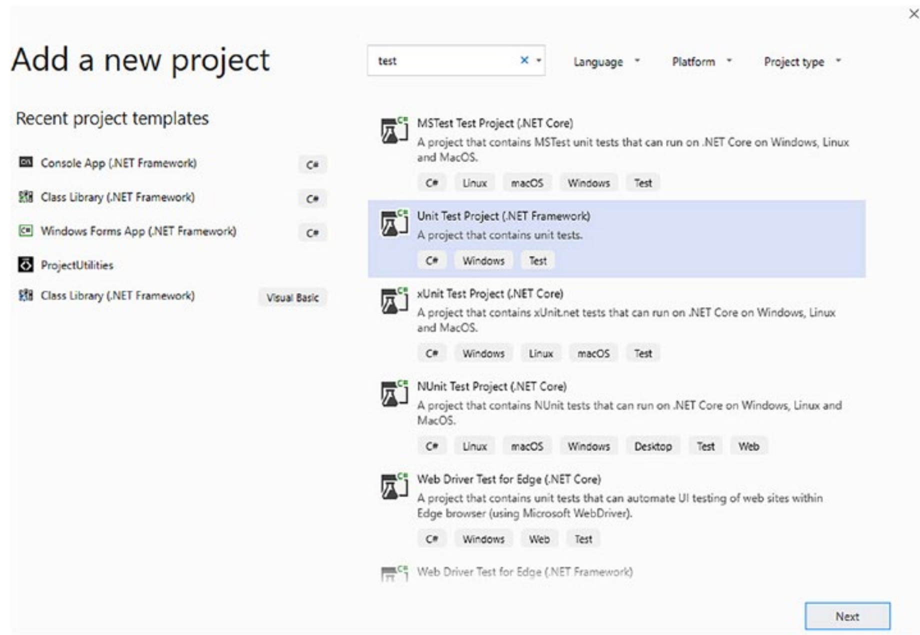


Figure 4-1. *Add a new Unit Test project*

Start off by adding a new Unit Test project to your solution. You will see (Figure 4-1) that you have the option to add a Unit Test project template for the test framework you prefer to use.

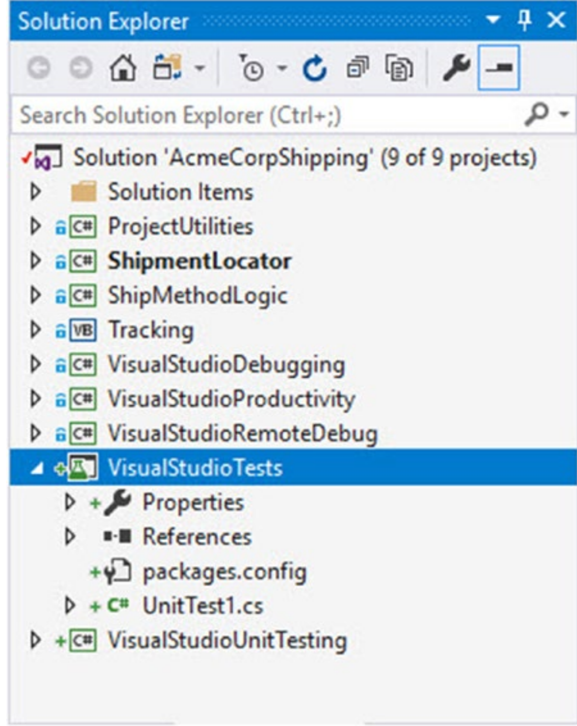


Figure 4-2. *Unit Test project added to the solution*

Once you have added your Unit Test project to your solution, it will appear in the solution with a different icon indicating that it is a Unit Test project (Figure 4-2).

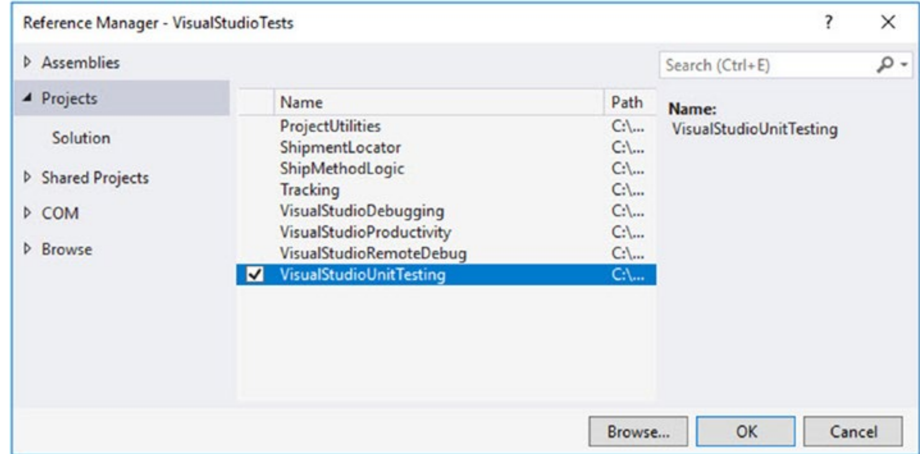


Figure 4-3. *Reference class to test*

To effectively test the class that contains the method that converts Celcius to Fahrenheit, we need to reference that class in our Unit Test project. Right-click the Unit Test project and add a reference to the project containing the class we need to test (Figure 4-3).

When the reference has been added to your test project, create the test as seen in Listing 4-2.

Listing 4-2. Unit Test for Fahrenheit

```
[TestClass]
public class ConversionHelperTests
{
    [TestMethod]
    public void Test_Fahrenheit_Calc()
    {
        // arrange - setup
        var celsius = -7.0;
        var expectedFahrenheit = 19.4;

        // act - test
        var result = ConversionHelpers.ToFahrenheit(celsius);
```

```
        // assert - check
        Assert.AreEqual(expectedFahrenheit, result);
    }
}
```

When you look at the code in Listing 4-2, you will notice that we do three things in a given test. These are

- Arrange – Where we set up the test
- Act – Where we test the code to get a result
- Assert – Where we check the actual result against the expected result

From the Test menu, select Windows and then Test Explorer, or hold down Ctrl E, T. In Test Explorer, click the green play button to run the test and see the test results displayed (Figure 4-4).

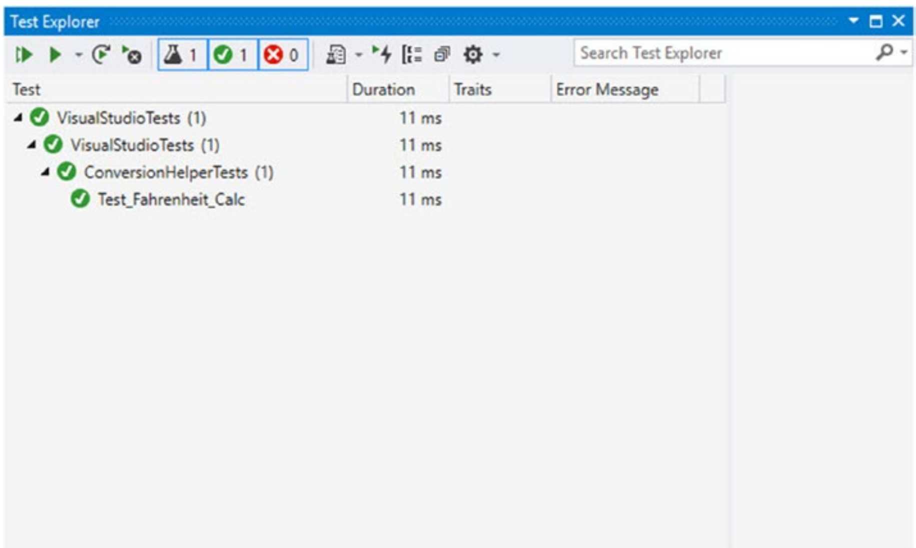


Figure 4-4. *Running your Unit Test*

From the results displayed in the Test Explorer, you can easily see which tests failed and which have passed. From our rather simple test in Listing 4-2, you can see that the test passed easily and that the result we expected was indeed the actual result of the test. Note that our test compares two type double values for exact equality. The Assert.AreEqual method has an overload that accepts an error tolerance parameter.

To see what happens when a test fails, modify the Integer value for the constant `F_ADDITION` variable as seen in Listing 4-3.

Listing 4-3. Modify the Fahrenheit Constant

```
private const double F_MULTIPLIER = 1.8;
private const int F_ADDITION = 33;
public static double ToFahrenheit(double celsius)
{
    return celsius * F_MULTIPLIER + F_ADDITION;
}
```

Running the tests again after the change will result in a failed test as seen in Figure 4-5. The change we made was a small change, but it's easy to miss this if we work in a team and on a big code base.

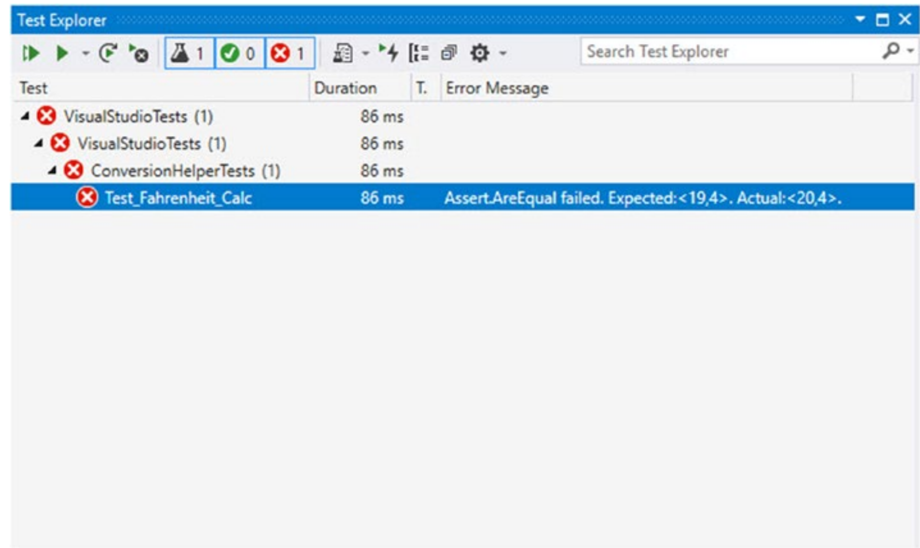


Figure 4-5. *Failed test results for Fahrenheit calculation*

What the Unit Test does is to keep an eye on the quality of the code as it changes throughout development. This is especially important when working in a team. It will allow other developers to see if any code changes they have made has broken some intended functionality in the code.

In Visual Studio 2019, you can also run the tests by right-clicking the test project and selecting Run Tests from the context menu.

The Test Explorer offers a lot of functionality, and you can see this from looking at the labels on the image in Figure 4-6.

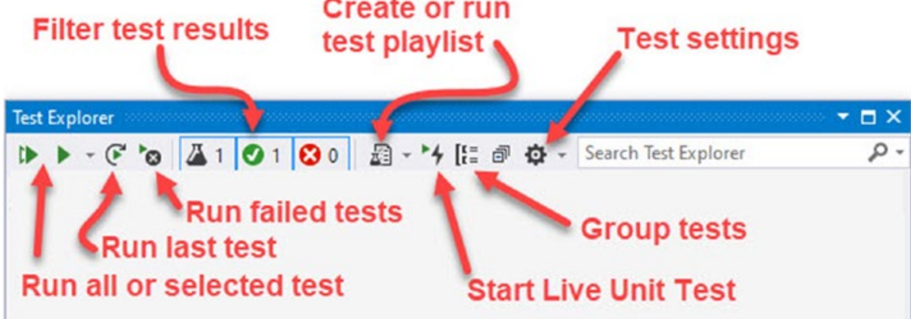


Figure 4-6. *Test Explorer Menu*

From the Test Explorer, you can

- Run all tests or just the last test
- Only run failed tests (great if you have many tests in your project)
- Filter the test results
- Group tests
- Start Live unit Testing (more on this later)
- Create and run a test playlist
- Modify test settings

Let's have a look at creating a test playlist.

Create and Run a Test Playlist

If your project contains many tests, and you want to run those tests as a group, you can create a playlist. To create a playlist, select the tests that you want to group from the Test Explorer, and right-click them. From the context menu that pops up, select Add to Playlist, New Playlist as seen in Figure 4-7.

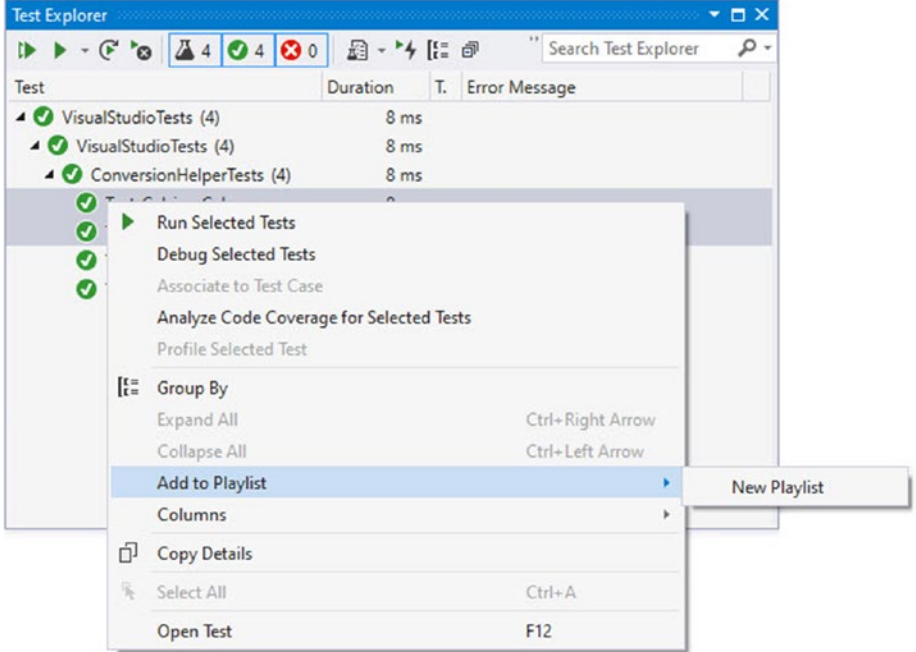


Figure 4-7. *Create a Playlist*

This will open a new Test Explorer window where you can run the tests and save the tests you selected under a new playlist name. This will create a .playlist file for you.

I created a new playlist called Temperature_Tests.playlist from the Celsius and Fahrenheit temperature conversion tests. The playlist file it creates is simply an XML file that in my example looks as in Listing 4-4.

Listing 4-4. Temperature_Tests.playlist file contents

```
<Playlist Version="1.0">
<Add Test="VisualStudioTests.ConversionHelperTests.Test_
Fahrenheit_Calc" />
<Add Test="VisualStudioTests.ConversionHelperTests.Test_
Celsius_Calc" />
</Playlist>
```

To open and run a playlist again, click the Create or run test playlist button and select the playlist file you want to run.

Testing Timeouts

The speed of your code is also very important. If you are using the MSTest framework, you can set a timeout attribute to set a timeout after which a test should fail. This is really convenient because as you write code for a specific method, you can immediately identify if the code you are adding to a method is causing a potential bottleneck. Consider the `Test_Fahrenheit_Calc` test we created earlier.

Listing 4-5. Adding a timeout attribute

```
[TestMethod]
[Timeout(2000)]
public void Test_Fahrenheit_Calc()
{
    // arrange - setup
    var celsius = -7.0;
    var expectedFahrenheit = 19.4;

    // act - test
    var result = ConversionHelpers.ToFahrenheit(celsius);

    // assert - check
    Assert.AreEqual(expectedFahrenheit, result);
}
```

As seen in Listing 4-5, I have added a timeout of 2000 milliseconds. If you run your tests now, it will pass because the calculation it performs is all it does. To see the timeout attribute in action, swing back to the ToFahrenheit method in the ConversionHelpers class and modify it by sleeping the thread for 2.5 seconds as seen in Listing 4-6.

Listing 4-6. Sleeping the Thread

```
public static double ToFahrenheit(double celsius)
{
    Thread.Sleep(2500);
    return celsius * F_MULTIPLIER + F_ADDITION;
}
```

Run your tests again and see that this time, your test has failed because it has exceeded the specified timeout value set by the `Timeout` attribute (Figure 4-8).

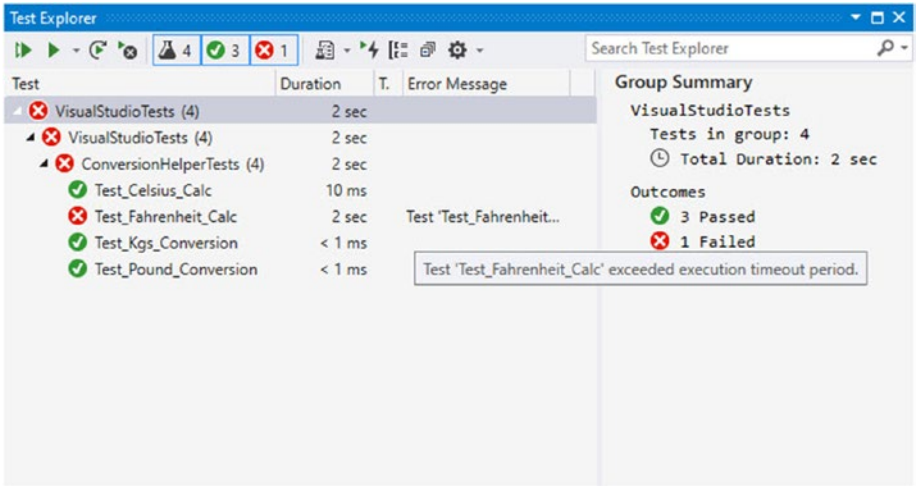


Figure 4-8. *Test timeout exceeded*

Identifying critical methods in your code and setting a specific timeout on that method will allow developers to catch issues early on when tests start exceeding the timeout set. You can then go back and immediately refactor the code that was recently changed in order to improve the execution time.

Using Live Unit Tests

First introduced in Visual Studio 2017, Live Unit Testing runs your unit tests automatically as you make changes to your code. You can then see the results of your unit tests in real time.

Live Unit Testing is only available in Visual Studio Enterprise edition for C# and Visual Basic projects targeting the .NET Framework or .NET Core. For a full comparison between the editions of Visual Studio, refer to the following link: `https://visualstudio.microsoft.com/vs/compare/`.

The benefits of Live Unit Testing are

- You will immediately see failing tests allowing you to easily identify breaking code changes
- Indicates Code Coverage allowing you to see what code is not covered by any unit tests

Live Unit Testing persists the data of the status of the tests it ran. It then uses the persisted data to dynamically run your tests as your code changes. Live Unit Testing supports the following test frameworks:

- xUnit.net – Minimum version xunit 1.9.2
- NUnit – Minimum version NUnit version 3.5.0
- MSTest – Minimum version MSTest.TestFramework 1.0.5-preview

Before you can start using Live Unit Testing, you need to configure Live Unit Testing from Tools, Options and selecting Live Unit Testing in the left pane (Figure 4-9).

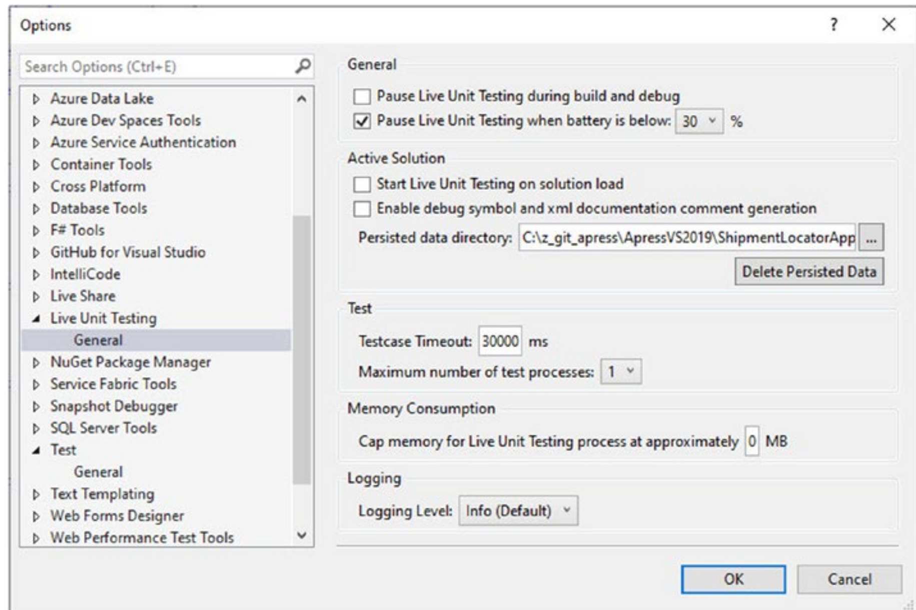


Figure 4-9. *Configure Live Unit Testing*

Once you have configured the Live Unit Testing options, you can enable it from Test, Live Unit Testing, Start. To see the Live Unit Testing window, click the Live Unit Testing button as seen in Figure 4-6.

The Live Unit Testing window is displayed as seen in Figure 4-10.

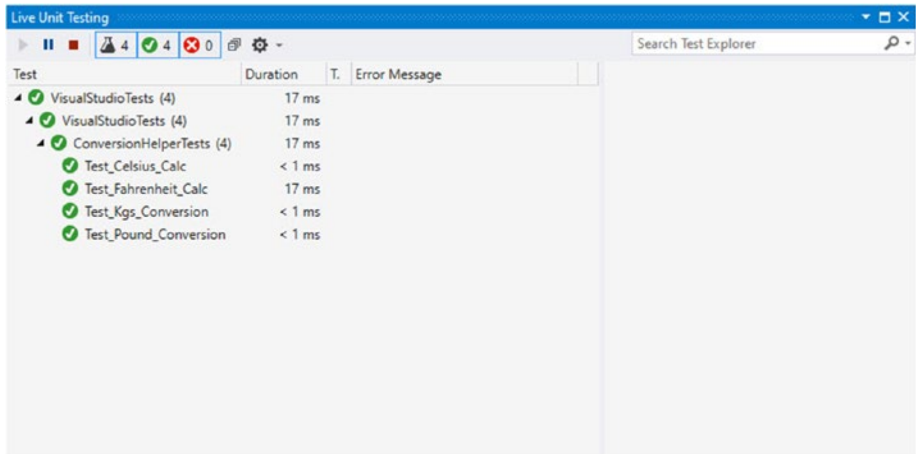


Figure 4-10. *Live Unit Testing window*

Make some breaking changes to your code and save the file. You will see that the Live Unit Testing window is updated to display the failing tests as seen in Figure 4-11.

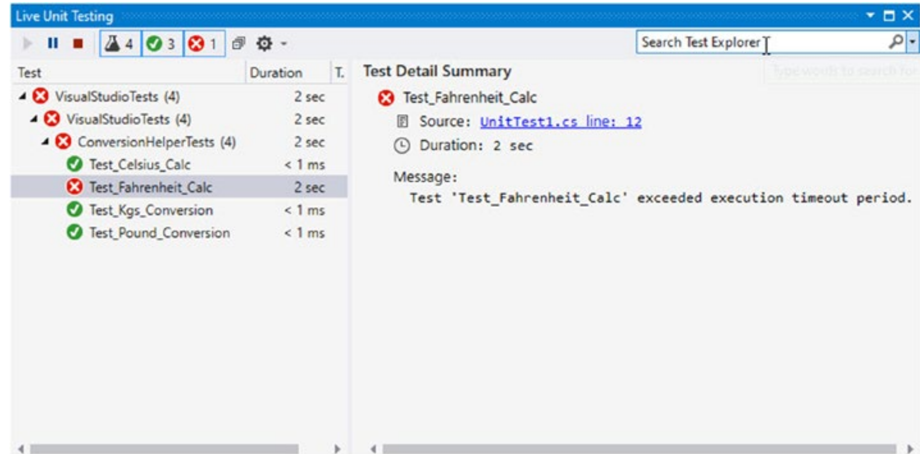


Figure 4-11. *Live Unit Testing results updated*

Live Unit Testing gives you a good insight into the stability of the code you write, as you write the code. Let's go a little further. Add the following class to your project under test (Listing 4-7):

Listing 4-7. Container Class implementing ICloneable

```
public class Container : ICloneable
{
    public string ContainerNumber { get; set; }
    public string ShipNumber { get; set; }
    public double Weight { get; set; }

    public object Clone() => throw new
    NotImplementedException();
}
```

Don't add any implementation to the `Clone` method. Swing back to the test project and add a Unit Test for the `Container` class as in Listing 4-8.

Listing 4-8. Unit Test for Container Class

```
[TestMethod]
public void Test_Container()
{
    var containerA = new Container();
    var containerB = containerA.Clone();

    var result = (containerA == containerB);

    Assert.IsFalse(result);
}
```

Start Live Unit Testing, and you will notice that your test fails as seen in Figure 4-12.

```
[TestMethod]
| 0 references | 0 changes | 0 authors, 0 changes
public void Test_Container()
{
    var containerA = new Container();
    var containerB = containerA.Clone();

    var result = containerA == containerB;

    Assert.IsFalse(result);
}
```

Figure 4-12. *Live Unit Test Results Failed*

Have a look at the `Container` class, and you will notice that Live Unit Testing has also updated the code file with the faulting method (Figure 4-13).

```
1 reference | 0 changes | 0 authors, 0 changes
public class Container : ICloneable
{
    0 references | 0 changes | 0 authors, 0 changes
    public string ContainerNumber { get; set; }
    0 references | 0 changes | 0 authors, 0 changes
    public string ShipNumber { get; set; }
    0 references | 0 changes | 0 authors, 0 changes
    public double Weight { get; set; }

    1 reference | 0/1 passing | 0 changes | 0 authors, 0 changes
    public object Clone() => throw new NotImplementedException();
}
```

Figure 4-13. *Container Class Live Unit Test results*

As soon as you add implementation to the `Clone` method, your Live Unit Test results are updated as seen in Figure 4-14.

```
1 reference | 0 changes | 0 authors, 0 changes
public class Container : ICloneable
{
    0 references | 0 changes | 0 authors, 0 changes
    public string ContainerNumber { get; set; }
    0 references | 0 changes | 0 authors, 0 changes
    public string ShipNumber { get; set; }
    0 references | 0 changes | 0 authors, 0 changes
    public double Weight { get; set; }

    1 reference | 0/1 passing | 0 changes | 0 authors, 0 changes
    public object Clone()
    {
        return this.MemberwiseClone();
    }
}
```

Figure 4-14. *Implementing the Clone method*

With Live Unit Testing, areas of code indicated by a dash are not covered by any tests. A green tick indicates that the code is covered by a passing test. A red X indicates that the code is covered by a failing test.

Using IntelliTest to Generate Unit Tests

IntelliTest helps developers generate and get started using Unit Tests. This saves a lot of time writing tests and increases code quality.

IntelliTest is only available in Visual Studio Enterprise edition.

The default behavior of IntelliTest is to go through the code and try to create a test that gives you maximum Code Coverage. To illustrate how IntelliTest works, I will create a simple class that calculates shipping costs as seen in Listing 4-9.

Listing 4-9. Calculate ShippingCost Method

```
public class Calculate
{
    public enum ShippingType { Overnight = 0, Priority = 1,
    Standard = 2 }

    private const double VOLUME_FACTOR = 0.75;

    public double ShippingCost(double length, double width,
    double height, ShippingType type)
    {
        var volume = length * width * height;
        var cost = volume * VOLUME_FACTOR;

        switch (type)
        {
            case ShippingType.Overnight:
                cost = cost * 2.25;
                break;
            case ShippingType.Priority:
                cost = cost * 1.75;
                break;
            case ShippingType.Standard:
                cost = cost * 1.05;
                break;
            default:
                break;
        }

        return cost;
    }
}
```

To run IntelliTest against the `ShippingCost` method, right-click the method, and click IntelliTest, Run IntelliTest from the context menu. The results will be displayed in the IntelliTest window that pops up as seen in Figure 4-15. You can also see the details of the generated Unit Test in the Details pane.

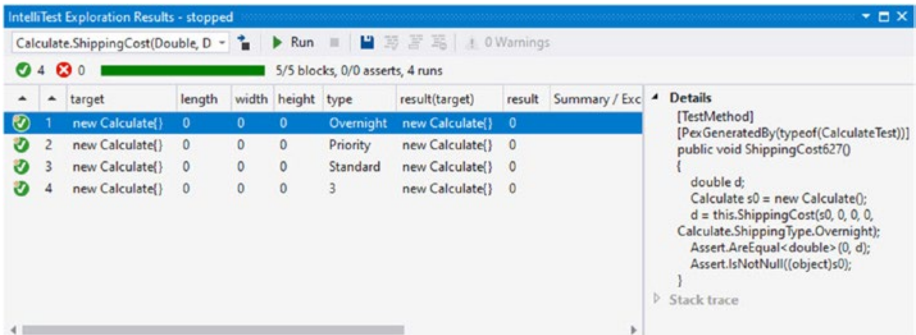


Figure 4-15. *IntelliTest Results*

IntelliTest has taken each parameter going to the method and generated a parameter value for it. In this example, all the tests succeeded, but there is still a problem. This is evident from the result value which is always zero.

We can never allow a parcel to be shipped with a zero shipping cost, no matter how small it is. What becomes clear here is that we need to implement minimum dimensions. We, therefore, need to modify the `Calculate` class as in Listing 4-10.

Listing 4-10. Modified Calculate Class

```
public class Calculate
{
    public enum ShippingType { Overnight = 0, Priority = 1,
    Standard = 2 }

    private const double VOLUME_FACTOR = 0.75;
    private const double MIN_WIDTH = 1.5;
```

```
    private const double MIN_LENGTH = 2.5;
    private const double MIN_HEIGHT = 0.5;

    public double ShippingCost(double length, double width,
    double height, ShippingType type)
    {
        if (length <= 0.0) length = MIN_LENGTH;
        if (width <= 0.0) width = MIN_WIDTH;
        if (height <= 0.0) height = MIN_HEIGHT;
        var volume = length * width * height;
        var cost = volume * VOLUME_FACTOR;

        switch (type)
        {
            case ShippingType.Overnight:
                cost = cost * 2.25;
                break;
            case ShippingType.Priority:
                cost = cost * 1.75;
                break;
            case ShippingType.Standard:
                cost = cost * 1.05;
                break;
            default:
                break;
        }

        return cost;
    }
}
```

Running IntelliTest again yields a completely different set of results as seen in Figure 4-16.

Figure 4-16. *IntelliTest Results on modified class*

This time you can see that no matter what the value of the parcel dimensions are, we will always have a result returned for the shipping costs. To create the Unit Tests generated by IntelliTest, click the Save button in the IntelliTest window.

This will create a new Unit Test project for you in your solution as seen in Figure 4-17.

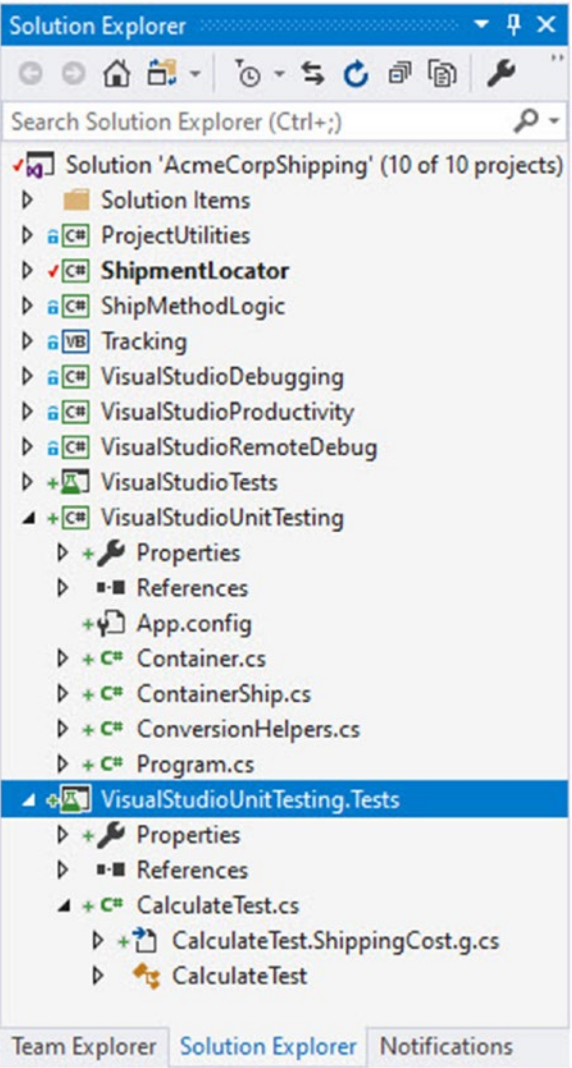


Figure 4-17. *Generated Unit Tests*

You can now run the generated Unit Tests as you normally would with Test Explorer. As you continue coding and adding more logic to the `Calculate` class, you can regenerate the Unit Tests by running IntelliTest

again. IntelliTest will then crawl through your code again and generate new Unit Tests for you to match the logic of your code at that time.

The underlying engine that IntelliTest uses to crawl through your code and generate the Unit Tests is Pex. Pex is a Microsoft Research project that was never productized or supported until IntelliTest started using it.

For a moment, I want you to think back to the code in Listing 4-10. Remember how we modified the code to include constant values to cater for Intellitest setting the default parameter values to zero? Imagine for a minute that we will never receive a zero as a parameter and that this check is built into the calling code. We can tell IntelliTest to assume values for these parameters.

Have a look at Figure 4-17, and locate the `CalculateTest` partial class generated for you by IntelliTest. The code generated for you is in Listing 4-11.

Listing 4-11. Generated CalculateTest Partial Class

```
[TestClass]
[PexClass(typeof(Calculate))]
[PexAllowedExceptionFromTypeUnderTest(typeof(ArgumentException),
AcceptExceptionSubtypes = true)]
[PexAllowedExceptionFromTypeUnderTest(typeof(InvalidOperation
Exception))]
public partial class CalculateTest
{

    [PexMethod]
    public double ShippingCost(
        [PexAssumeUnderTest]Calculate target,
```

```
        double length,
        double width,
        double height,
        Calculate.ShippingType type
    )
    {
double result = target.ShippingCost(length, width, height,
type);
        return result;
        // TODO: add assertions to method CalculateTest.
        ShippingCost(Calculate, Double, Double, Double,
        ShippingType)
    }
}
```

We are now going to tell the Pex engine that we want to assume certain values for the parameters. We do this by using `PexAssume`.

PexAssume is a static helper class containing a set of methods to express preconditions in parameterized Unit Tests.

Modify the code in the `CalculateTest` partial class' `ShippingCost` method by adding `PexAssume.IsTrue` as a precondition for each parameter as illustrated in Listing 4-12.

Listing 4-12. Modified CalculateTest Partial Class

```
[PexMethod]
public double ShippingCost(
    [PexAssumeUnderTest]Calculate target,
    double length,
```

```
    double width,
    double height,
    Calculate.ShippingType type
)
{
    PexAssume.IsTrue(length > 0);
    PexAssume.IsTrue(width > 0);
    PexAssume.IsTrue(height > 0);
    double result = target.ShippingCost(length, width, height,
    type);
    return result;
    // TODO: add assertions to method CalculateTest.
    ShippingCost(Calculate, Double, Double, Double,
    ShippingType)
}
```

By doing this, I can now modify my Calculate class to remove the constant values ensuring that the length, width, and height parameters are greater than zero. The Calculate class will now look as in Listing 4-13.

Listing 4-13. Modified Calculate Class

```
public class Calculate
{
    public enum ShippingType { Overnight = 0, Priority = 1,
    Standard = 2 }

    private const double VOLUME_FACTOR = 0.75;

    public double ShippingCost(double length, double width,
    double height, ShippingType type)
    {
        var volume = length * width * height;
        var cost = volume * VOLUME_FACTOR;
```

```
        switch (type)
        {
            case ShippingType.Overnight:
                cost = cost * 2.25;
                break;
            case ShippingType.Priority:
                cost = cost * 1.75;
                break;
            case ShippingType.Standard:
                cost = cost * 1.05;
                break;
            default:
                break;
        }

        return cost;
    }
}
```

Run IntelliTest again, and see that the parameter values passed through are never zero (Figure 4-18).

Figure 4-18. *IntelliTest Results with PexAssume*

You can modify the `CalculateTest` partial class by adding assertions to the `ShippingCost` method. When you expand CalculateTest in the Solution Explorer (Figure 4-19), you will see several `ShippingCost` Test methods listed.

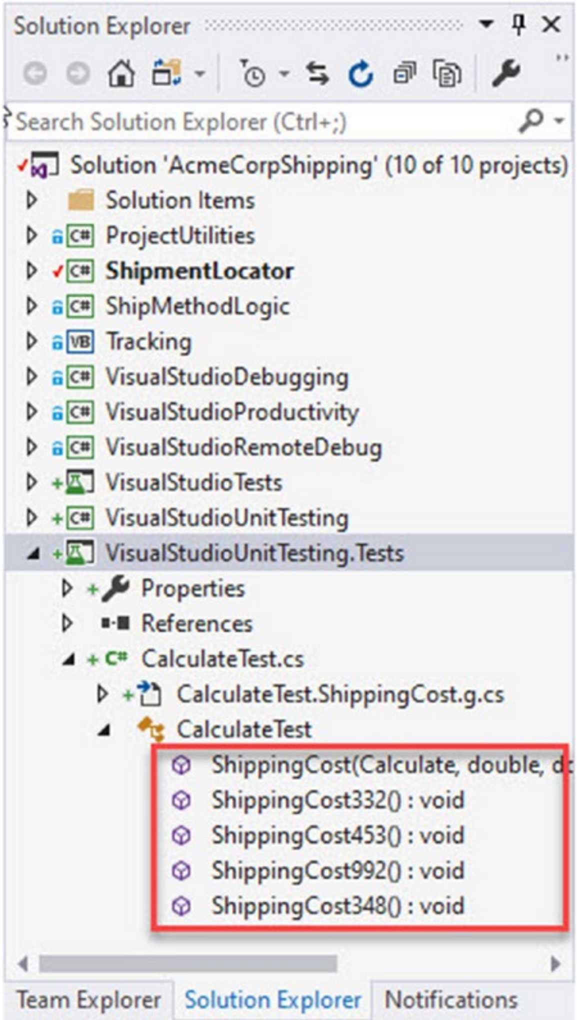


Figure 4-19. *ShippingCost Generated Tests*

These correspond to the IntelliTest results as seen in Figure 4-18. Do not modify these code files, as your changes will be lost when IntelliTest is run again and it regenerates those tests.

Focus IntelliTest Code Exploration

Sometimes IntelliTest needs a bit of help focusing code exploration. This can happen if you have an Interface as a parameter to a method and more than one class implements that Interface. Consider the code in Listing 4-14.

Listing 4-14. Focusing Code Exploration

```
public class ShipFreight
{
    public void CalculateFreightCosts(IShippable box)
    {

    }
}

class Crate : IShippable
{
    public bool CustomsCleared { get; }
}

class Package : IShippable
{
    public bool CustomsCleared { get; }
}

public interface IShippable
{
    bool CustomsCleared { get; }
}
```

If you ran IntelliTest on the `CalculateFreightCosts` method, then you will receive the following warnings as can be seen in Figure 4-20.

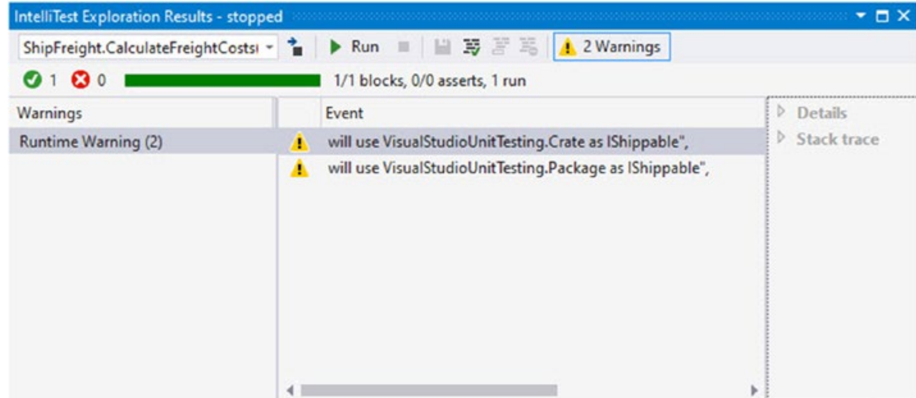


Figure 4-20. *Focus Code Exploration*

You can tell IntelliTest which class to use to test the interface. Assume that I want to use the Package class to test the Interface. Now, just select the second warning and click the Fix button on the menu as seen in Figure 4-21.

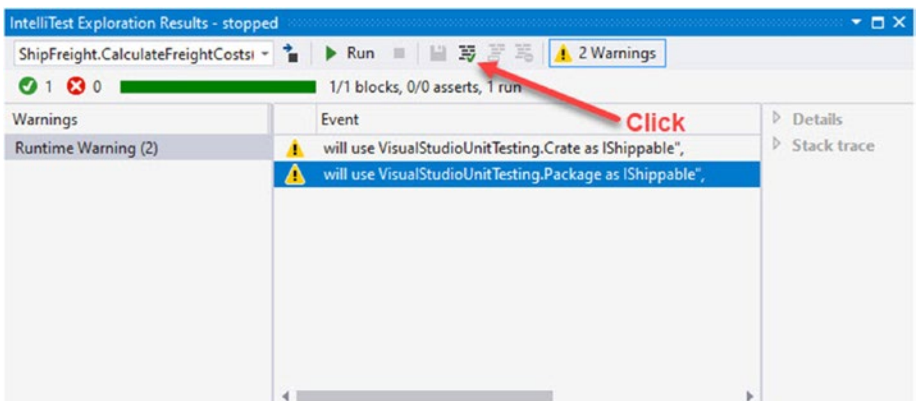


Figure 4-21. *Tell IntelliTest which class to use*

IntelliTest now updates the PexAssemblyInfo.cs file by adding `[assembly: PexUseType(typeof(Package))]` to the end of the file to tell IntelliTest which class to use. Running IntelliTest again results in no more warnings being displayed.

How to Measure Code Coverage in Visual Studio

Code Coverage indicates what portion of your code is covered by Unit Tests. To guard against bugs, it becomes obvious that the more code is covered by Unit Tests, the better tested it is.

IntelliTest is only available in Visual Studio Enterprise edition.

The Code Coverage feature in Visual Studio will give you a good idea of your current Code Coverage percentage. To run the Code Coverage analysis, open up Test Explorer, and click the drop-down next to the play button (Figure 4-22).

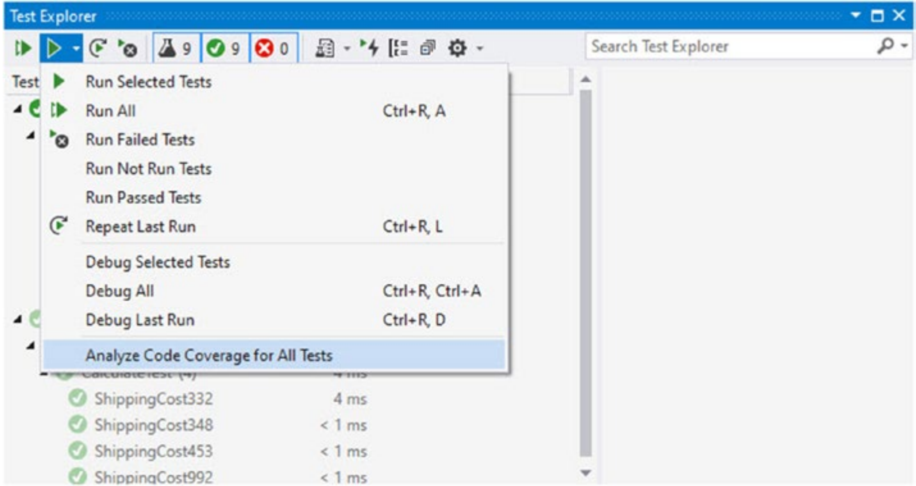


Figure 4-22. *Analyze Code Coverage*

Click Analyze Code Coverage for All Tests in the menu.

You can also go to the Test menu, click Windows, and click Test Explorer.

The Code Coverage Results are then displayed in a new window (Figure 4-23). You can access this window from the Test menu and then select Windows, Code Coverage Results or hold down Ctrl + E, C on the keyboard.

Code Coverage Results

Dirk Strauss_MSI 2019-09-16 07_45_50.cover

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
Dirk Strauss_MSI 2019-09-16 07_4...	81	56,64%	62	43,36%
visualstudiotests.dll	0	0,00%	18	100,00%
visualstudiounittesting.exe	81	81,82%	18	18,18%
visualstudiounittesting.tests.dll	0	0,00%	26	100,00%

Figure 4-23. *Code Coverage Results*

In the Code Coverage Results window, you can Export the results, Import Results, Merge Results, Show Code Coverage Color, or Remove the results.

Figure 4-24. *Toggle Code Coverage Coloring*

This will toggle colors in your code editor to highlight areas touched, partially touched, and not touched at all by tests. The colors used to highlight the code can also be changed. To do this, head on over to Tools, Options, Environment, Fonts and Colors (Figure 4-25).

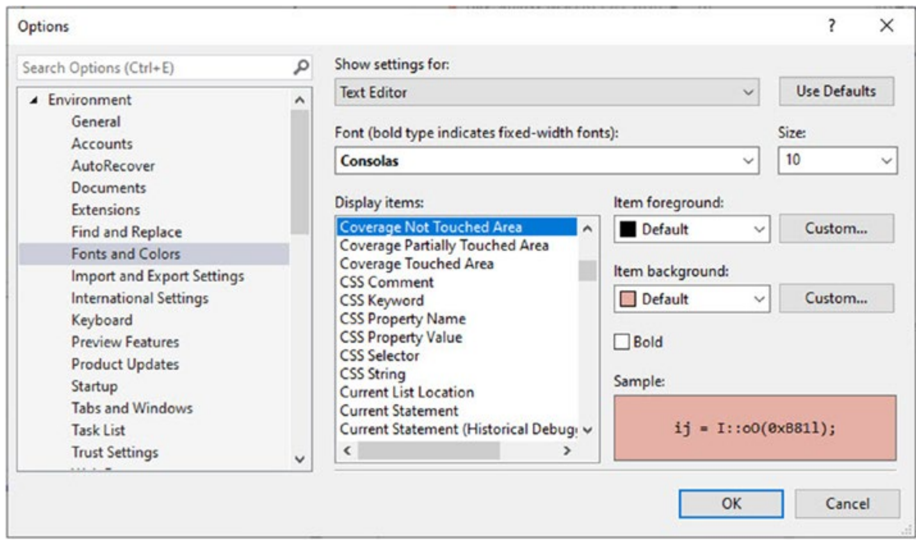


Figure 4-25. *Change Fonts and Colors*

This should give you a good understanding of how much code is covered by Unit Tests. Developers should typically aim for at least 80% Code Coverage. If the Code Coverage is low, then modify your code to include more tests. Once you are done modifying your code, run the Code Coverage tool again as the results are not automatically updated as you modify your code.

Code Coverage is typically measured in blocks. A block of code is a section of code that has exactly one entry point and one exit point. If you prefer to see the Code Coverage in terms of lines covered, you can change the results by choosing Add/Remove Columns in the results table header (Figure 4-26).

Code Coverage Results

Dirk Strauss_MSI 2019-09-16 07_45_50.coverage

Hierarchy	Not Covered (Lines)	Not Covered (% Lines)	Covered (Lines)	Covered (% Lines)
Dirk Strauss_MSI 2019-09-16 07_4...	68	43,31%	89	56,69%
visualstudiotests.dll	0	0,00%	30	100,00%
visualstudiounittesting.exe	68	70,83%	28	29,17%
visualstudiounittesting.tests.dll	0	0,00%	31	100,00%

Figure 4-26. *Code Coverage Expressed in Lines*

Code Coverage is a great tool to allow you to check if your code is sufficiently covered by Unit Tests. If you aim for 80% Code Coverage, you should be able to produce well-tested code. The 80% Code Coverage is not always attainable. This is especially true if the code base you're working on has a lot of generated code. In instances such as these, a lower percentage of code cover is acceptable.

CHAPTER 5

Source Control

If you have worked on projects in a team environment, or if you need a place to keep your own code safe, then you'll agree that using a source control solution is essential. It doesn't matter if it's a large enterprise solution or a small Pet project, Visual Studio makes it extremely easy for developers to use Git and GitHub.

Git is a tool that developers install locally on their machine. GitHub is an online service that stores code safely that has been pushed to it from computers using the Git tool.

In 2018, Microsoft acquired GitHub for $7.5 billion in Microsoft stock. This acquisition of GitHub brought about changes to their pricing tiers. Previously, developers could only create public repos on the free tier. In January 2019, however, GitHub announced that developers can now create unlimited private repositories on the free tier.

© Dirk Strauss 2020
D. Strauss, *Getting Started with Visual Studio 2019*,
https://doi.org/10.1007/978-1-4842-5449-3_5

This is really great, especially if you are working on a side project that you do not want to share with anyone just yet. In this chapter, we will be looking at using Git and GitHub inside Visual Studio 2019. We will see how to

- Create a GitHub account
- Create and clone a repository
- How to commit changes to a repository
- Create a branch from your code
- Creating and handling pull requests

These are all things that developers will do on a daily basis when working with Git and GitHub. While the process might change slightly if you use a different source control strategy, the concepts remain the same.

Create a GitHub Account

Let's start off with creating a GitHub account. Point your browser to `www.github.com`, and create an account by clicking the signup button.

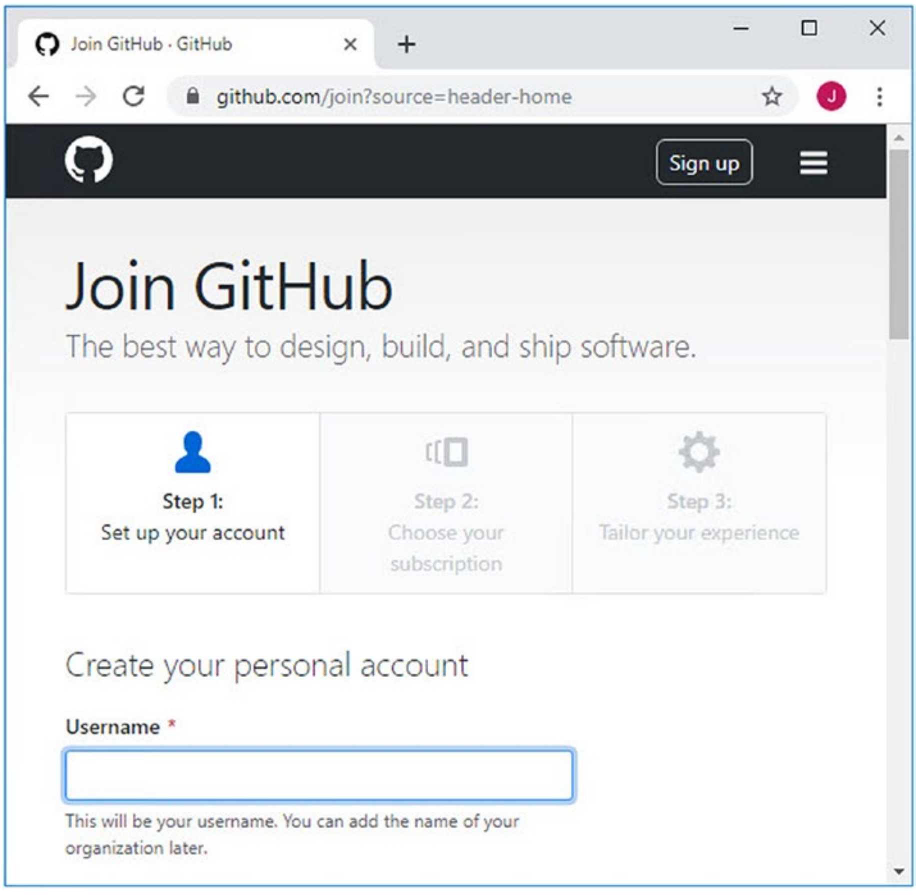


Figure 5-1. *Sign up for GitHub*

Enter a username (Figure 5-1), e-mail address, and password. After entering your details, you will need to select a subscription. GitHub offers two subscriptions, the free subscription that offers the basics suitable for most developers and a pro subscription. The free subscription is very generous and will appeal to most developers. The free subscription includes

- Unlimited public and private repositories
- Three collaborators for private repositories

- Issues and bug tracking
- Project management

The pro subscription, on the other hand, offers more but will be more suited to large teams of developers in an organization. The pro subscription includes

- Unlimited public and private repositories
- Unlimited collaborators
- Issues and bug tracking
- Project management
- Advanced tools and insights

This will allow a team of developers to work on a private repository for a nominal monthly subscription.

For more info on all GitHub's products, browse to the following URL `https://help.github.com/en/articles/githubs-products`.

After creating your account on GitHub, you will be sent an e-mail to verify your account with. Clicking the link in this e-mail will take you back to GitHub, to the Create a new repository page. If not, you can access your repositories from the menu under your profile image. This will take you to your repositories page from where you can create your first repository as seen in Figure 5-2.

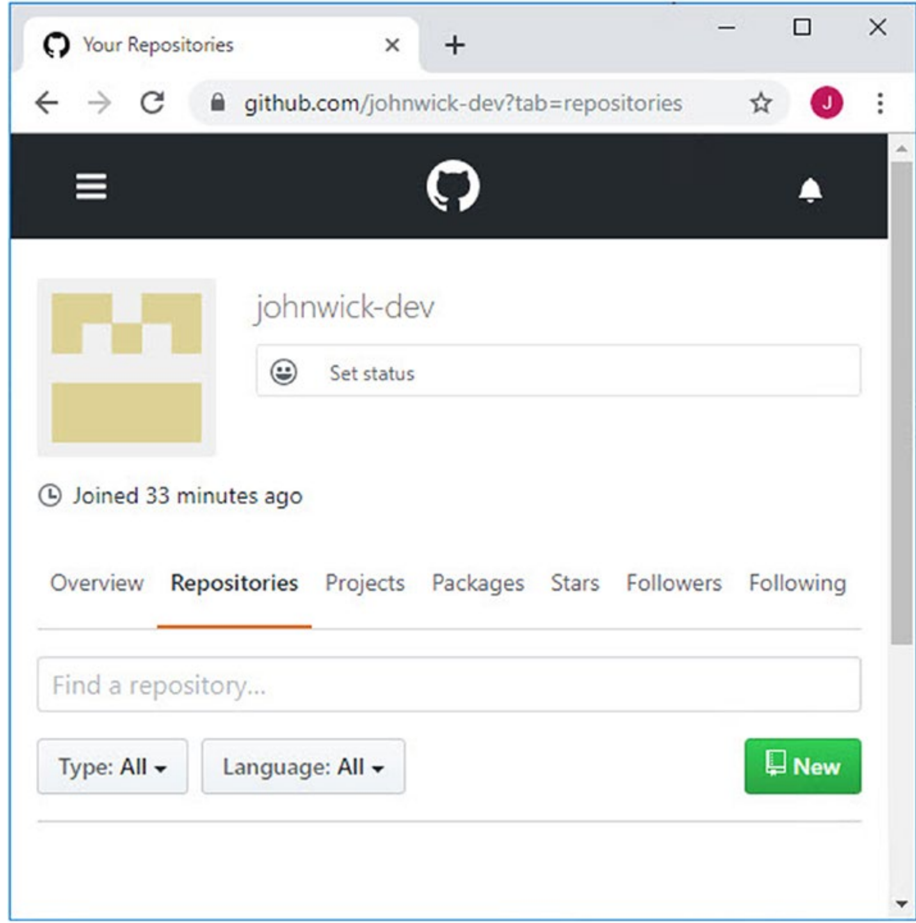


Figure 5-2. *Create a new repository*

The next logical step is to create a repository for your new project. Let's have a look at that in the next section.

Create and Clone a Repository

Before you start, you need to see if you have installed the GitHub extension for Visual Studio. To do this, you need to run the Visual Studio Installer. To find the Visual Studio Installer, search for it in Windows, or find it under the Windows start menu as seen in Figure 5-3.

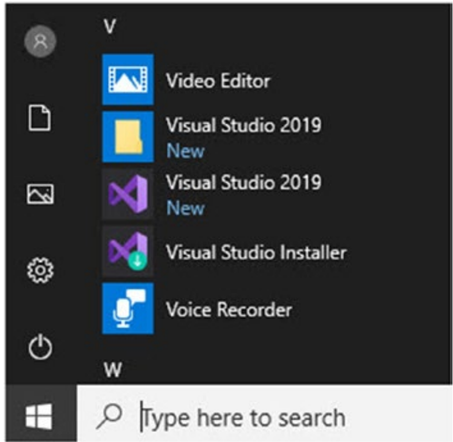


Figure 5-3. *Visual Studio Installer*

Sometimes, the Visual Studio Installer needs to update before running, so you need to give it a few minutes to do this depending on the speed of your Internet connection.

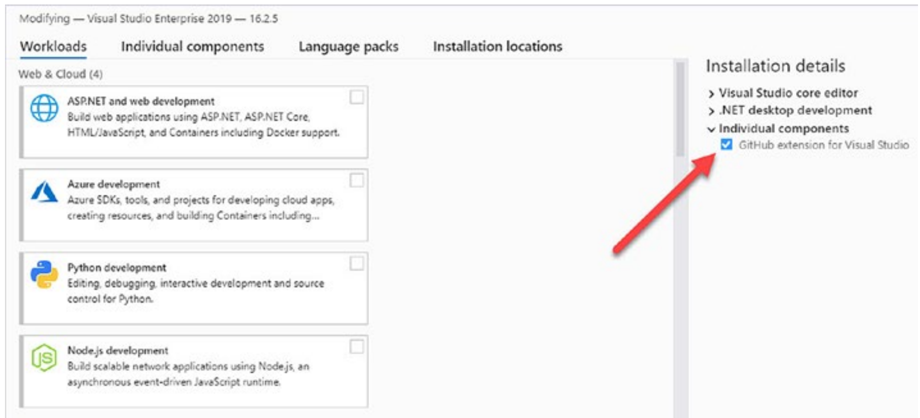


Figure 5-4. *GitHub extension for Visual Studio*

When the Visual Studio Installer has launched, you will see an option to modify your installation of Visual Studio. Click Modify and have a look at the installation details for Visual Studio. The GitHub extension for Visual Studio should be displayed under the Individual components. If you don't see the GitHub extension here, you will need to install it. To do this, change to the Individual components tab, and check the option to install the GitHub extension for Visual Studio under Code tools.

Once you have installed the GitHub extension, start Visual Studio and open up your solution. For this example, I have just created a small project called MyPetProject as seen in Figure 5-5.

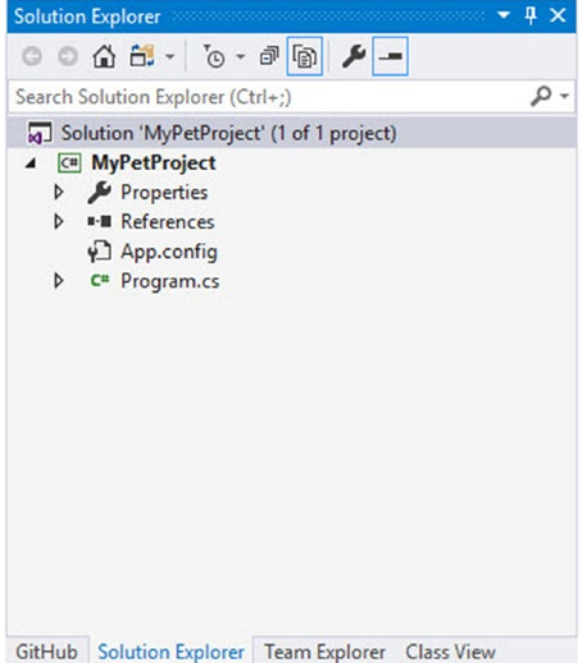


Figure 5-5. *MyPetProject solution in Visual Studio*

To get started adding this to a new repository, right-click the solution, and click Add Solution to Source Control as seen in Figure 5-6.

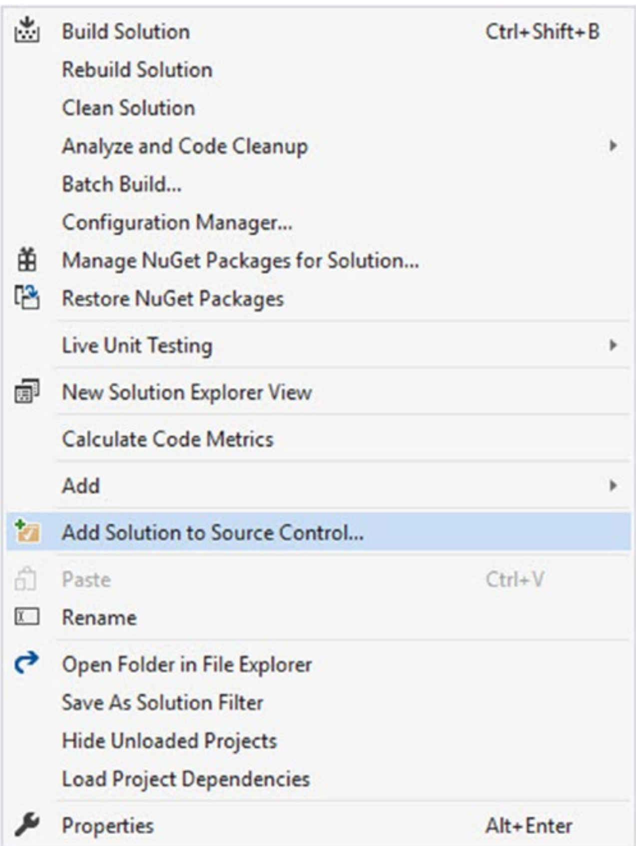


Figure 5-6. *Add Solution to Source Control*

This will then create a new Git repository for your solution. Open up the Output Window (Ctrl+Alt+O) from the View menu, and click Output. Here you will see that a new local Git repository has been created (Figure 5-7).

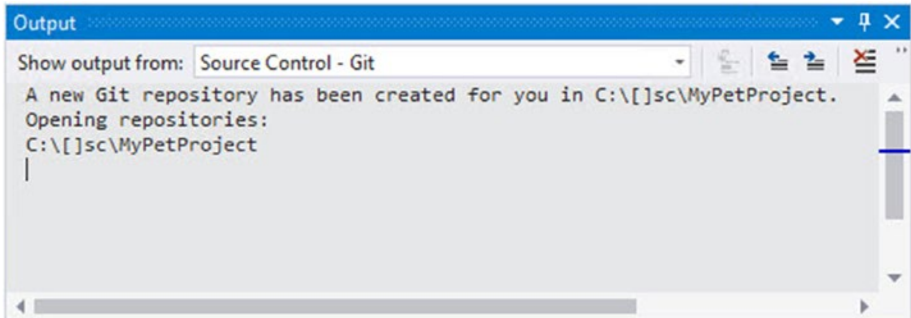


Figure 5-7. *New Git repository created*

It is important to remember that this project is now under source control using Git. Remember that we mentioned earlier that Git is the source control plumbing, the tool that developers install locally on their machines.

If you never want a backup of your code in the cloud, or never want to collaborate with other developers, you can just use Git. This is, however, a quite unlikely scenario. Especially now that GitHub allows you free private repositories.

It is, therefore, the logical next step to push your code to a GitHub repository. To do this, you need to open the GitHub tab as seen in Figure 5-8.

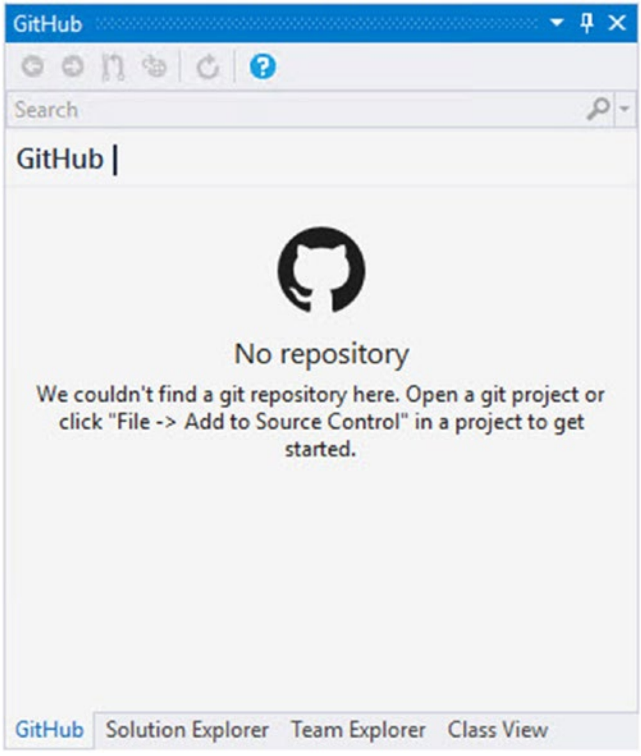


Figure 5-8. *The GitHub tab*

If you do not see the GitHub tab, click the View menu, then click Other Windows, and then click GitHub. The tab will tell you that the repository is not yet in GitHub. Change to the Team Explorer tab as seen in Figure 5-9. If you do not see the Team Explorer tab (Ctrl+\, Ctrl+M), go to the View menu, and click Team Explorer.

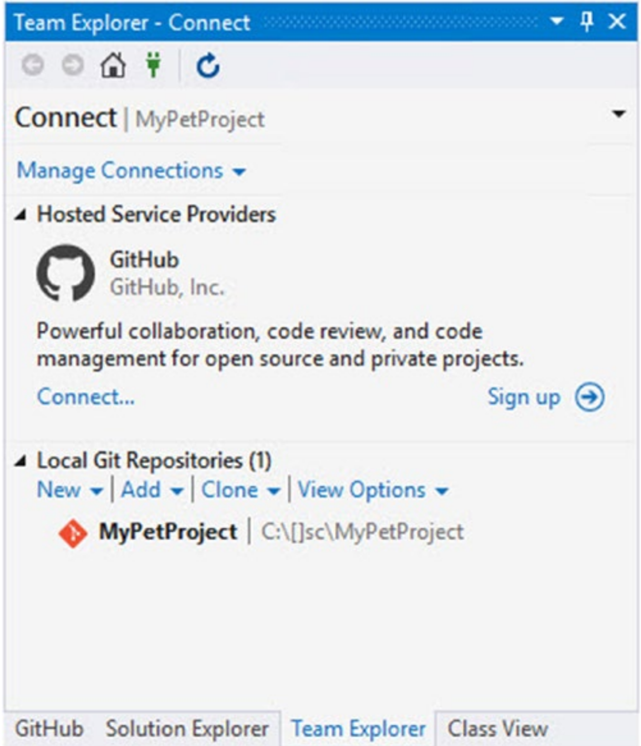


Figure 5-9. *Team Explorer*

With Team Explorer now open, you will see your local Git repository created earlier on. Under the Hosted Service Providers, you will see the option to sign up to GitHub or connect to GitHub. We have already created a GitHub account in the previous section, so here we will only be connecting to GitHub. Click the Connect link to open up the Connect to GitHub screen as seen in Figure 5-10.

Figure 5-10. *Connect to GitHub*

Once you have connected to GitHub, change to the Synchronization section of the Team Explorer window by clicking the Connect header and selecting Sync from the drop-down menu (Figure 5-11).

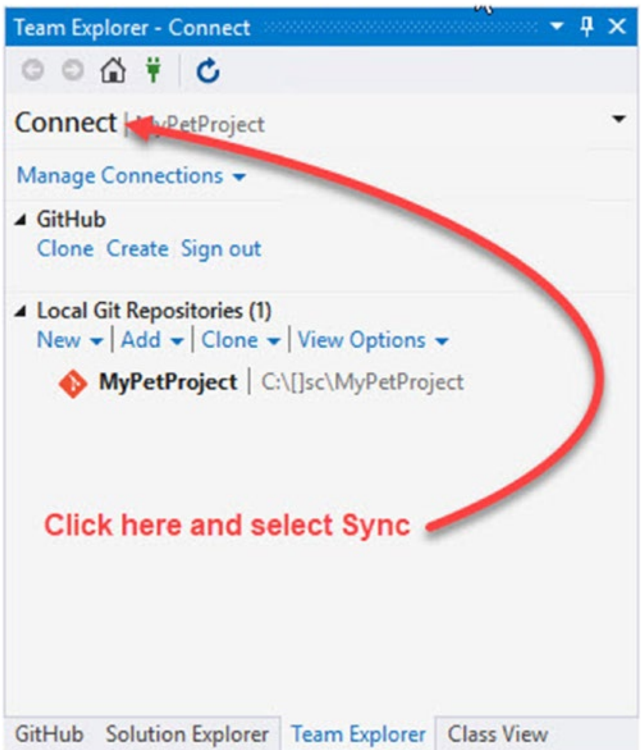


Figure 5-11. *Change to Sync*

The Synchronization screen is now displayed as seen in Figure 5-12.

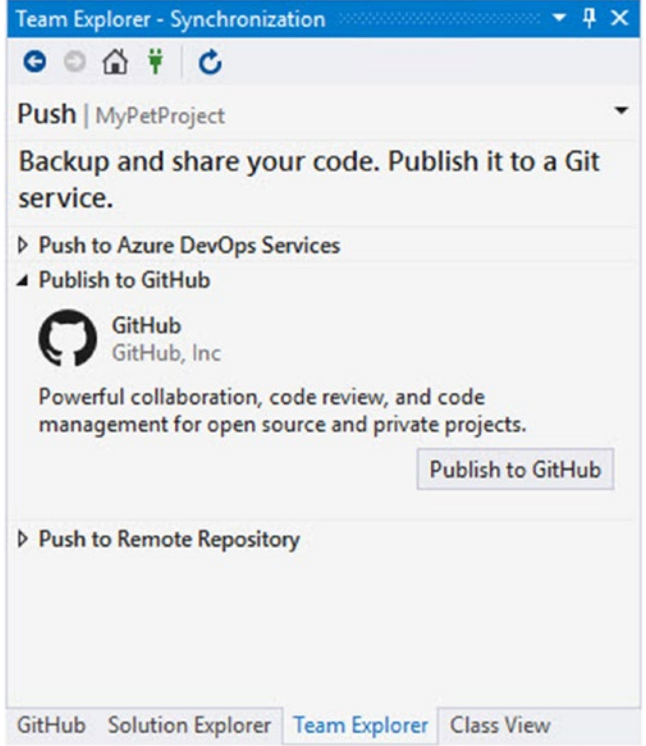


Figure 5-12. *Publish to GitHub*

From this screen, we will be publishing our code to GitHub.

If you do not see the option to publish to GitHub, then restart Visual Studio, and go to the Synchronization tab again. The option to publish to GitHub should be displayed after the restart.

To start the process, click the Publish to GitHub button (refer to Figure 5-12). This will then display the publish settings as seen in Figure 5-13.

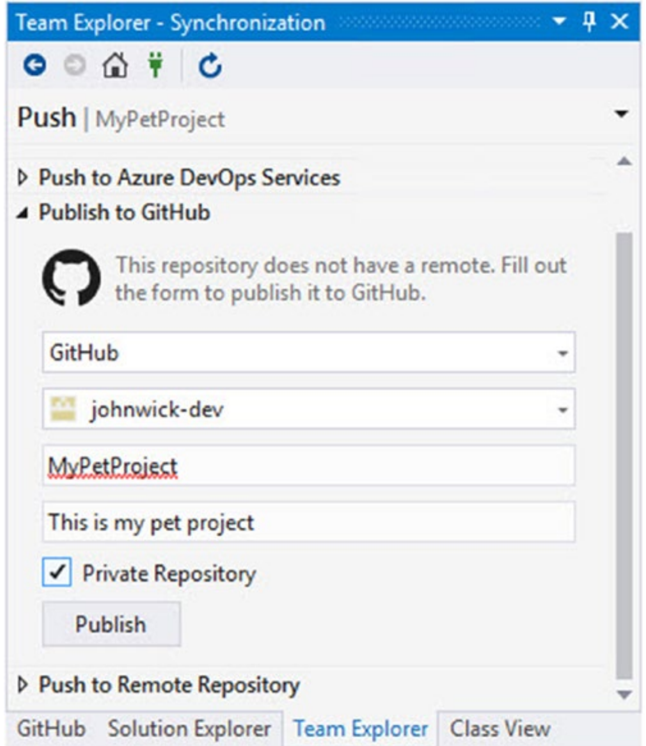


Figure 5-13. *Publish Settings*

Here we can select the account we want to use. In my case, it is the account that I created in the previous section and the one that I signed in to my GitHub account with earlier (refer to Figure 5-10).

The repository name should be pre-populated for you from your local repository. You can also add an optional description. Importantly, be sure to check the private repository checkbox if you want to keep your code private. When you are ready, click the Publish button.

Note that if you have two-factor authentication set up, you will be prompted at this point to provide the authentication code from your authenticator application on your mobile phone.

After the repository has been published to GitHub, you will see the new repository if you view your GitHub repositories online as seen in Figure 5-14.

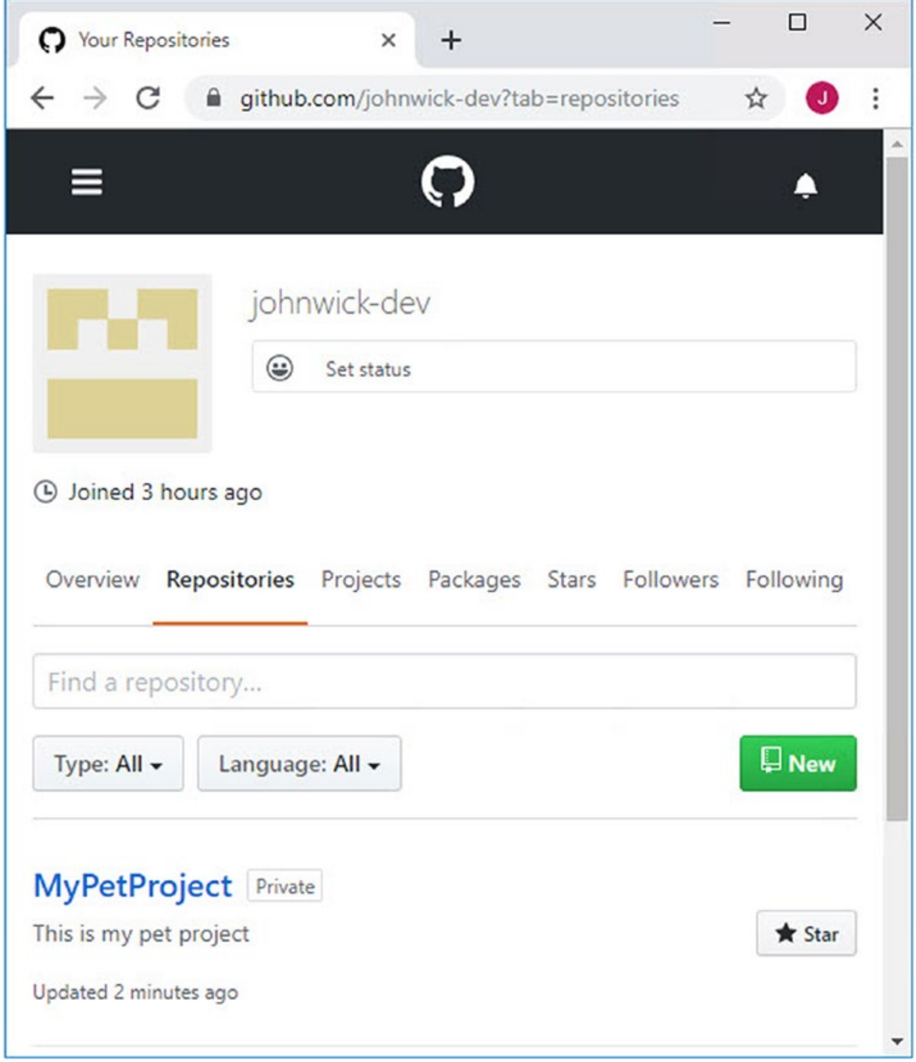


***Figure 5-14.** GitHub Repository Created Online*

I have some staged changes in my repo. Now I want to commit my code to my new GitHub repository. Switch to the Changes view in Team Explorer, and enter a commit message, and click the Commit Staged button (Figure 5-15).

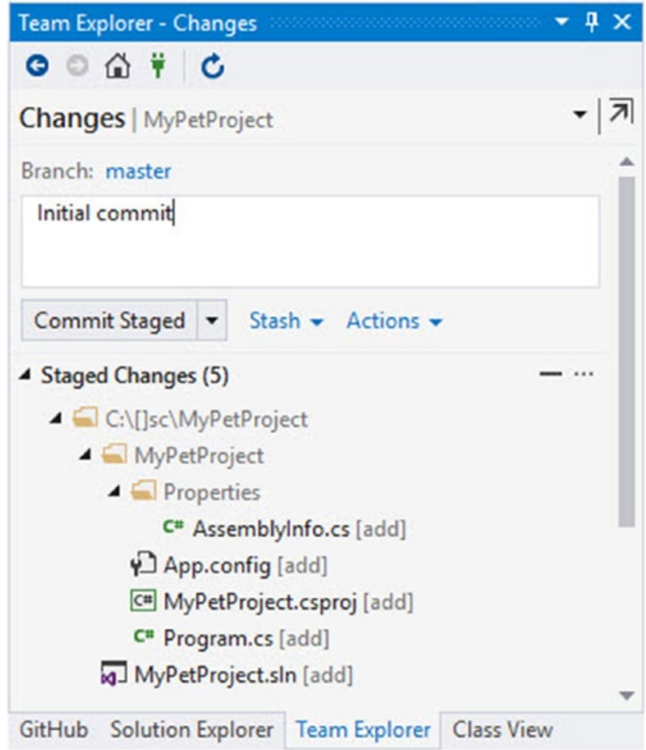


Figure 5-15. *Commit Staged Code*

This will commit the changes locally (Figure 5-16).

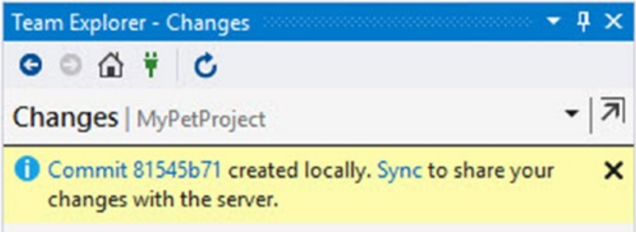


Figure 5-16. *Commit created Locally*

Remember, your commit is to your local Git repo. To push the changes to our GitHub repository, change to the Sync view. Do this by clicking the Changes header to view the drop-down menu, and select Sync as seen in Figure 5-17.

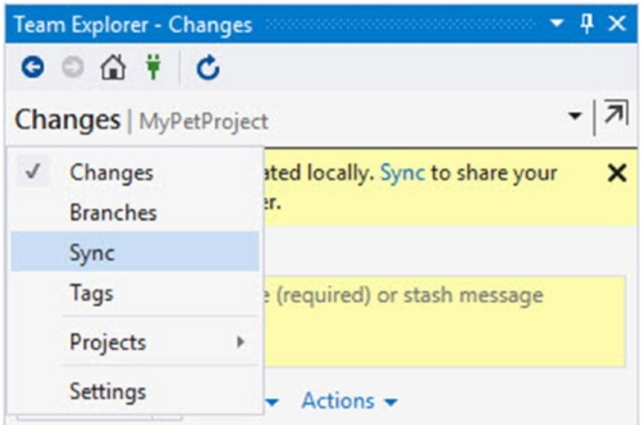


Figure 5-17. *Switch to Sync*

The Synchronization view is now displayed (Figure 5-18). Here you will see all the outgoing commits that are yet to be pushed to the server.

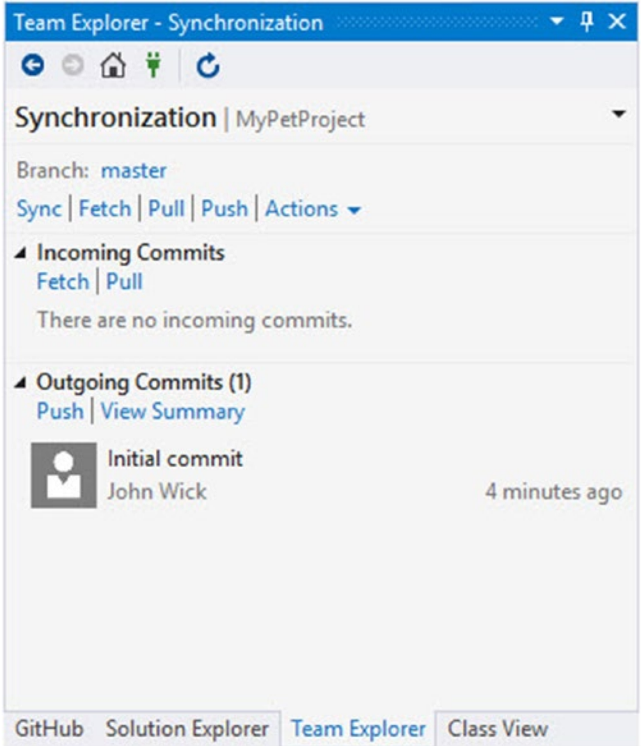


Figure 5-18. *Sync your changes with the server*

Clicking Push will send all the code to the GitHub repository. On GitHub, if you view the code for the repository, you will see the latest changes displayed there.

Cloning a Repository

What I want to do now is have another colleague of mine contribute to my project. Seeing as this is a private repository, I need to invite him to collaborate. In GitHub, go to the settings tab in your repository. Then click Collaborators as seen in Figure 5-19.

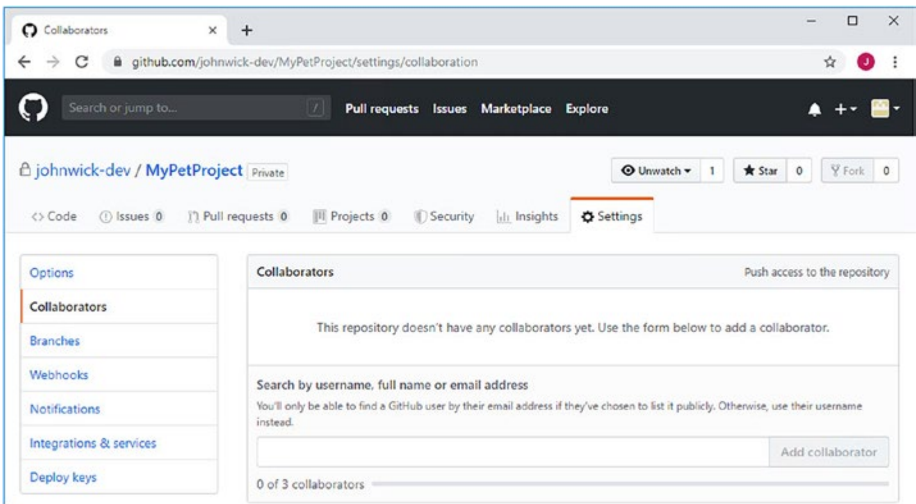


Figure 5-19. *Add Collaborators*

The free account can have three collaborators, so this will be using one of your allotted collaborators. In this example, John has invited me to collaborate on his Pet Project. I will now receive a notification in my inbox that John wants me to work on his project with him (Figure 5-20).

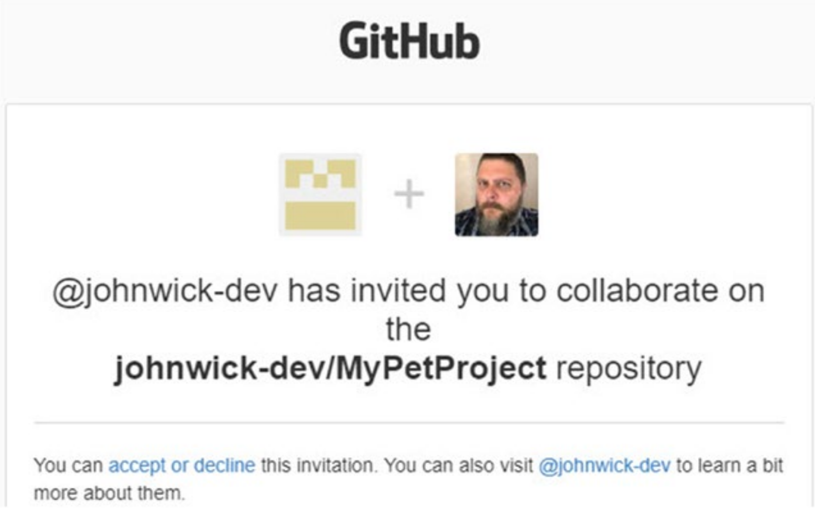


Figure 5-20. *Invitation to Collaborate*

Once I accept the invitation, I will have push access to the project. John will now see me as a collaborator under the collaborator's tab in GitHub. To start working on the code, I need to clone the repository to my local machine. Start Visual Studio, and then click the Clone or check out code option under the Get started section of the Visual Studio start screen (Figure 5-21).

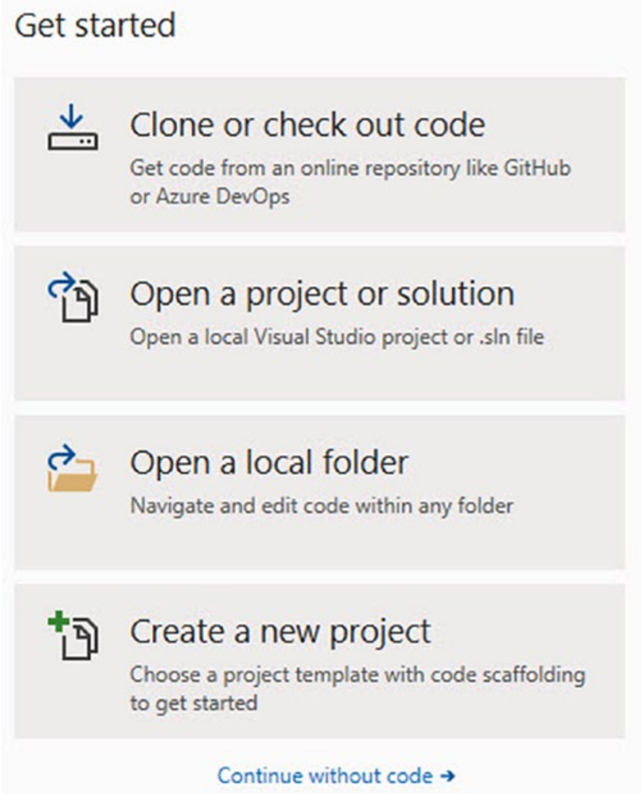


Figure 5-21. *Clone or check out code*

This will take you to the Clone or check out code screen (Figure 5-22).

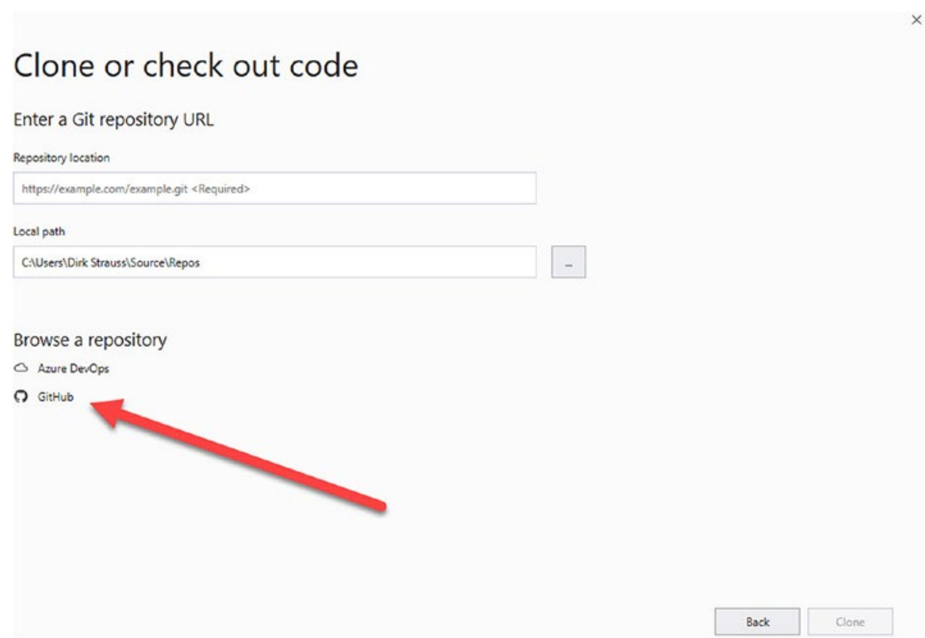


Figure 5-22. *Clone from GitHub*

From this screen, you can enter the repository location to get the code from, but because I am a collaborator on John's GitHub project, I can simply click the GitHub option (Figure 5-22).

This will display the Open from GitHub screen (Figure 5-23).

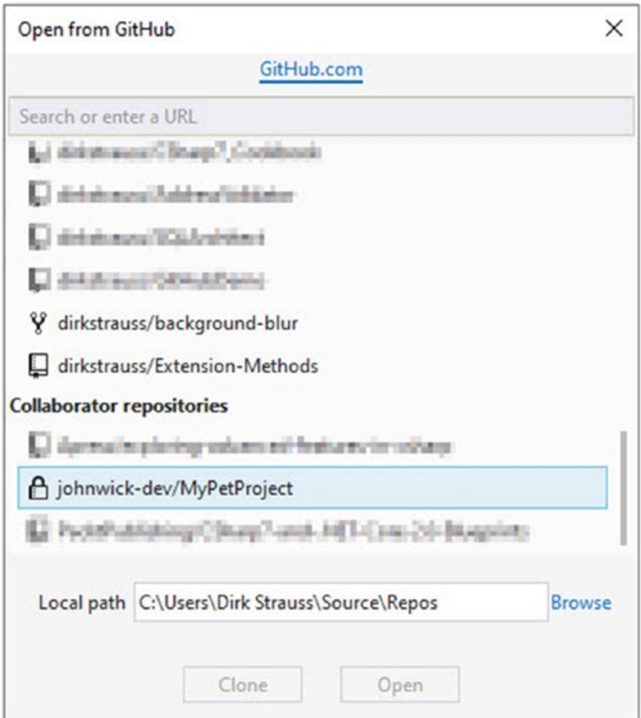


Figure 5-23. *Open from GitHub*

It is here that I will see the project that John invited me to under the Collaborator repositories. Select the project, ensure that the local path is correct, and click the Clone button. The Visual Studio project is then cloned to my local machine and displayed in my Team Explorer (Figure 5-24).

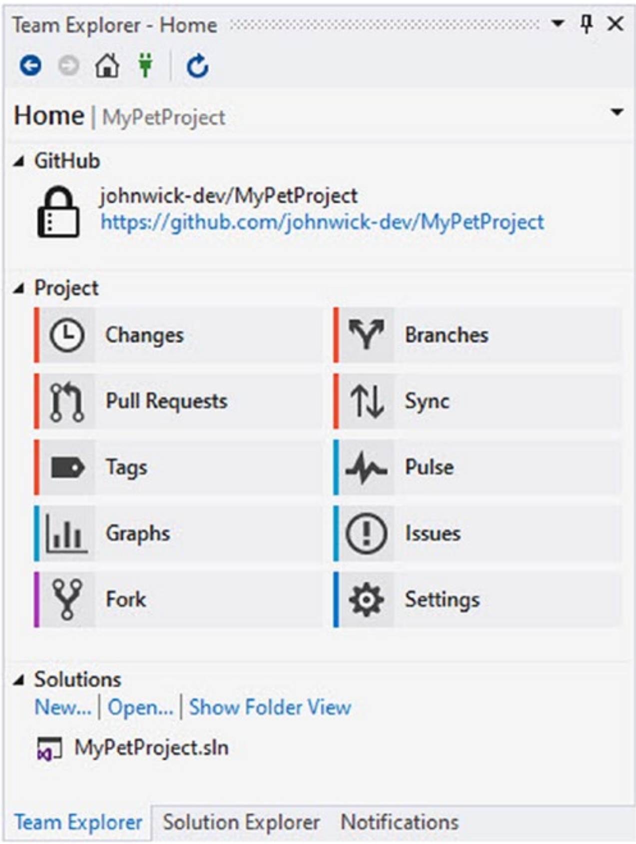


***Figure 5-24.** The cloned project in Team Explorer*

I can now collaborate with John on his Pet project and share my changes with him easily.

Create a Branch from Your Code

John has a new feature that needs to be added to his Pet project. It would be better for me to work on the changes to the project in an isolated manner. To do this, I can create a branch in Git. A branch allows me to make changes to the code without changing the code in the main

branch, also called the master branch. In Visual Studio, I can see that I am currently working on the master branch if I look at the bottom right status bar in Visual Studio (Figure 5-25).

Figure 5-25. *Working in the master branch*

To create a new branch, switch to the Branches view in Team Explorer (Figure 5-26).

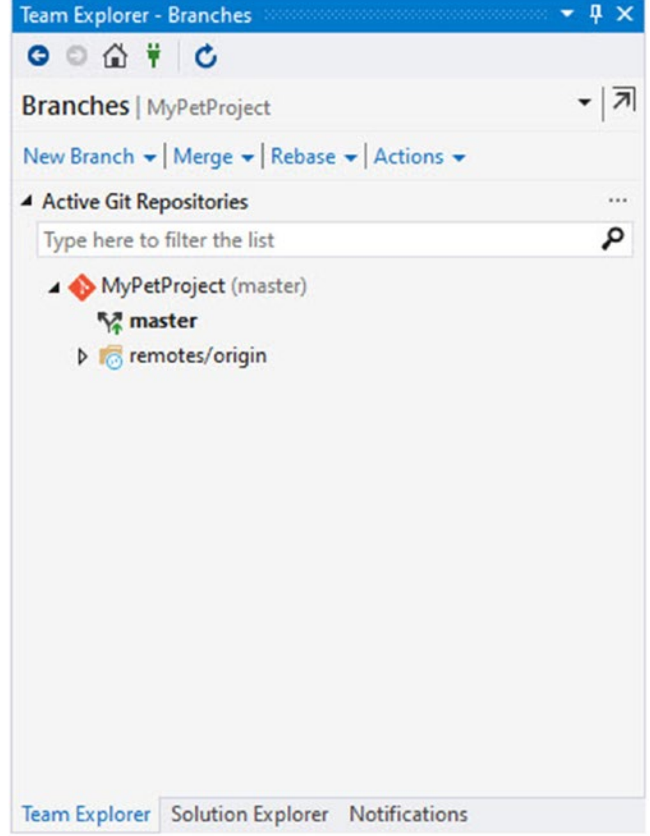


Figure 5-26. *Branches view*

I will now create a local branch in Visual Studio. To do this, click the New Branch link in the Branches view.

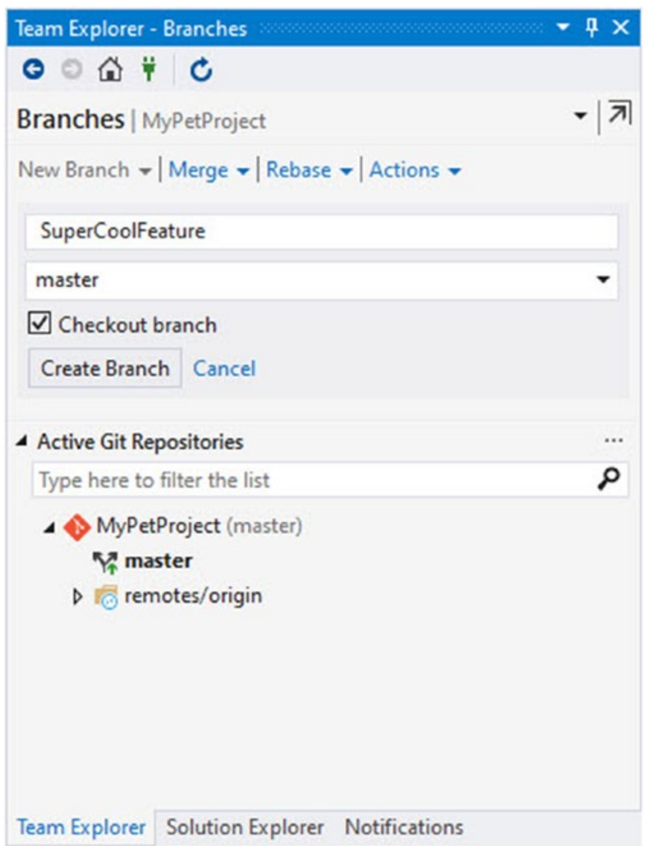


Figure 5-27. *Create a new Branch*

I can now give my new branch a suitable name (Figure 5-27) and tell it to create the branch from the master branch. I keep the Checkout branch selected to check out my new branch and click the Create Branch button.

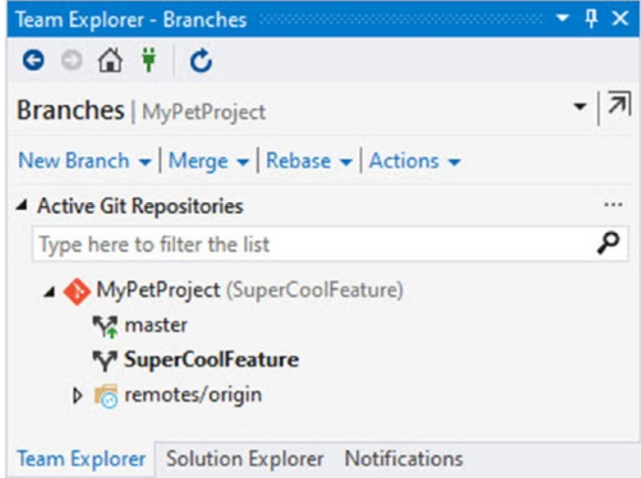


Figure 5-28. *Feature branch created*

As seen in Figure 5-28, my new local branch is created and checked out. When I look at the bottom right status bar in Visual Studio, I see that the new feature branch is checked out (Figure 5-29).

Figure 5-29. *Feature branch checked out*

This means that from now on, all changes made to the code will stay in this particular branch. Let's add some new code to the project.

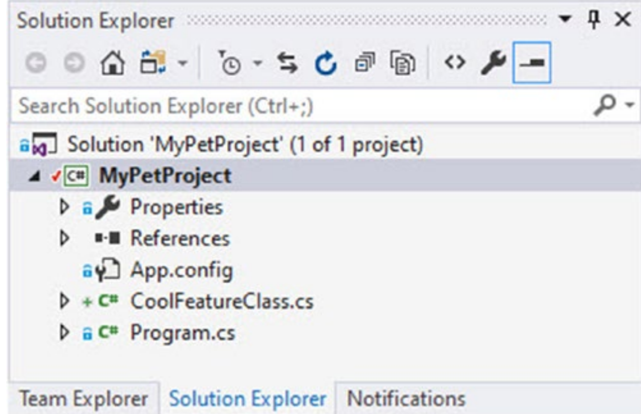


Figure 5-30. *New feature code added*

As seen in Figure 5-30, I have added a new class called `CoolFeatureClass` that contains the new code I added. I must now commit the changes to my branch. In Team Explorer, change to the Changes view. This will show me all the code that I have changed in my branch (Figure 5-31).

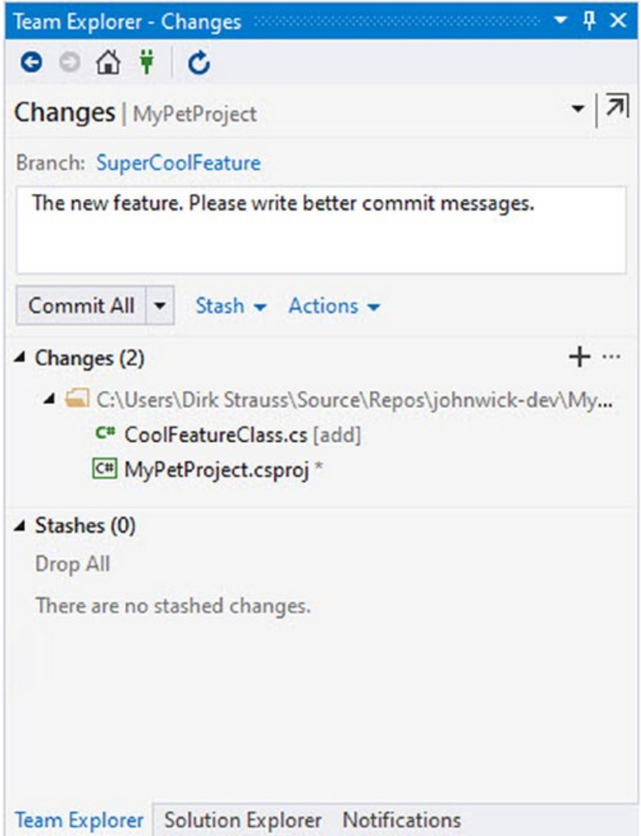


Figure 5-31. *Changes to feature branch*

You can see that the SuperCoolFeature branch is still selected. Under the changes section, you will see all the files that you have changed. Before you commit your code, you need to add a suitable commit message. Then I can click the drop-down next to the Commit All button and select Commit All and Push (Figure 5-32).

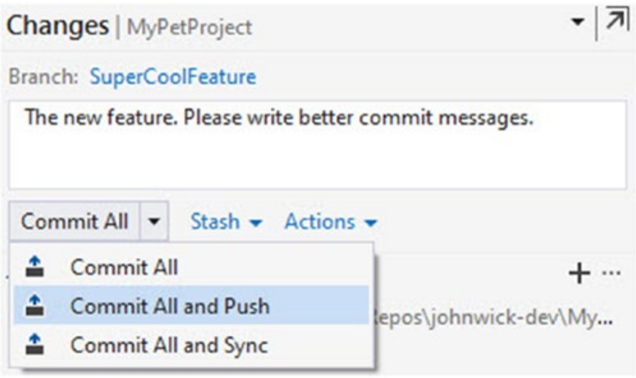


Figure 5-32. *Commit All and Push*

This will commit the changes to the local repo and then push them to the remote repo. If you switch to the Branches view in Team Explorer and expand the `remotes/origin` folder, you will see that your feature branch has been pushed to the server (Figure 5-33).

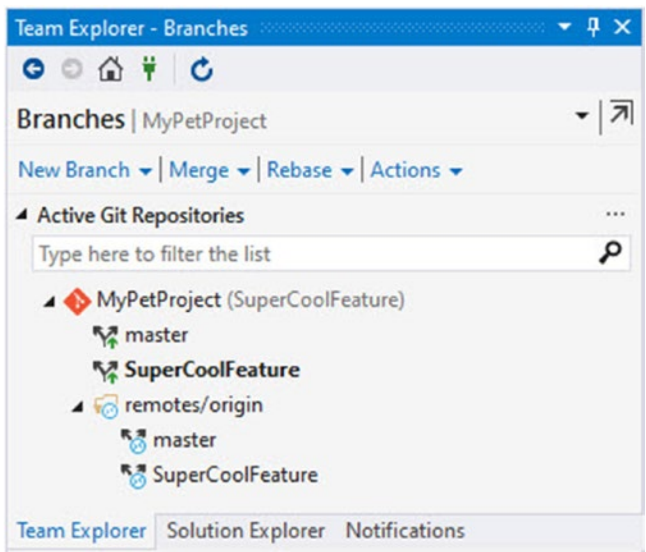


Figure 5-33. *Branch pushed to the server*

The code is now safely on the GitHub repo. How do I get my changes into the master branch? For this, we will be creating a pull request.

Creating and Handling Pull Requests

The term pull request might sound strange to some folks that aren't used to working with a source control system. The "pull" means to request that your code be pulled into the main working branch of the source code. Some developers also refer to a pull request as a merge request.

In Visual Studio, we can easily create a pull request. By doing this, we are telling the team that our code is ready to be peer-reviewed and, if it's good, merged into the main master branch.

You will remember that in the previous section, we created a branch and added all our new features to the branch. Then we committed those changes to Git (locally) before pushing them up to the server on GitHub.

To create a pull request, change to the GitHub tab. If you do not see the GitHub tab, click the View menu, then click Other Windows, and then click GitHub.

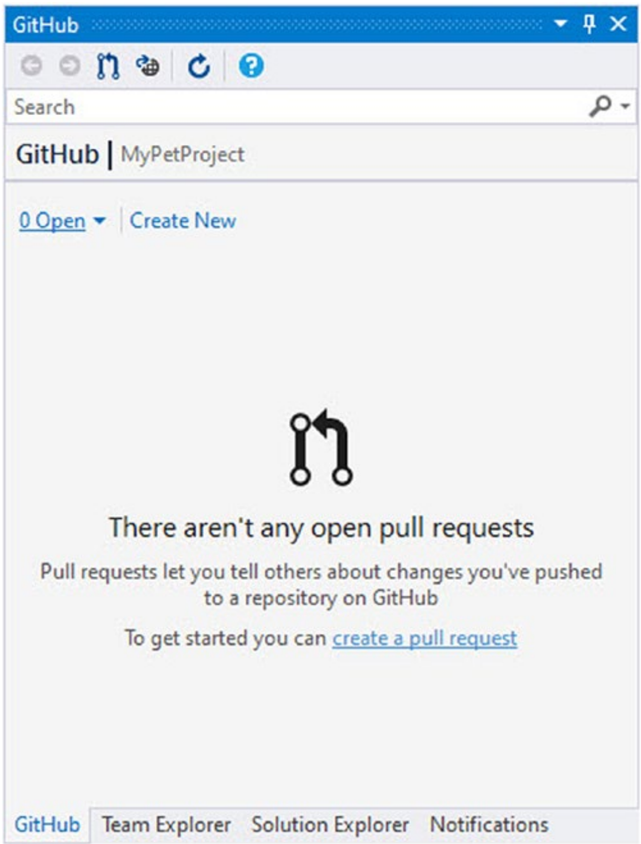


Figure 5-34. *Create a Pull Request*

You will see that (Figure 5-34) you can now create a pull request by clicking the "Create a pull request" link. Clicking the pull request link will bring you to a section where you can fill in your pull request details (Figure 5-35).

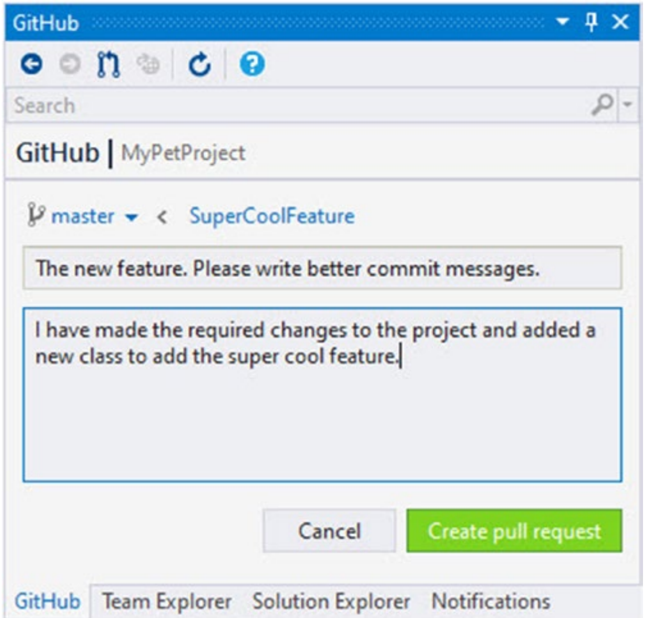


Figure 5-35. *Pull Request Details*

When you have added all the required details, you click the Create pull request button. This pull request will now go to John where he can review my code, add comments, and hopefully approve my changes.

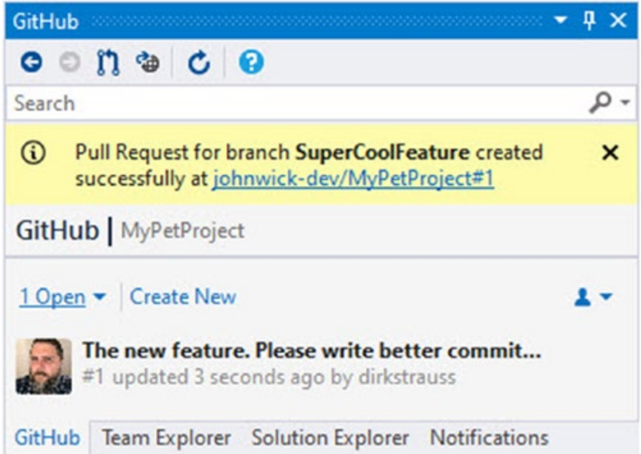


Figure 5-36. *Pull Request created*

When the pull request is successfully created, you will see the notification in Visual Studio.

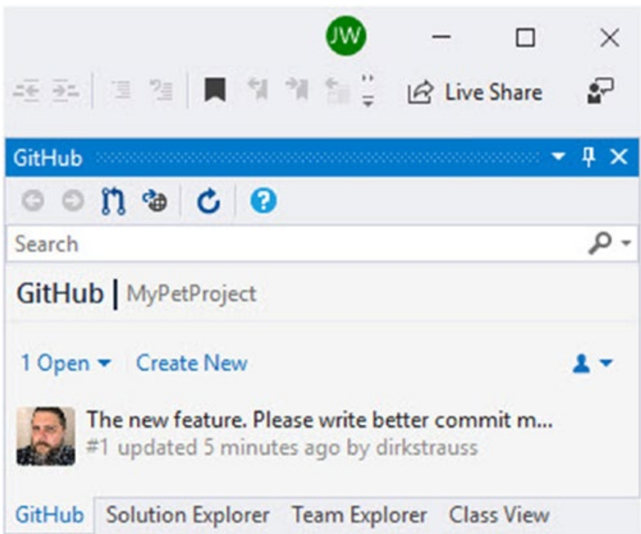


Figure 5-37. *New Pull Request Notification*

On the other side of the continent, John has just finished working on some code and sees my pull request in his GitHub tab in Visual Studio (Figure 5-37).

Seeing and approving pull requests in Visual Studio are brand new to 2019. If you are using an older version of Visual Studio, you will need to handle all pull requests in GitHub online.

John can now click the pull request that I created to view the details (Figure 5-38).

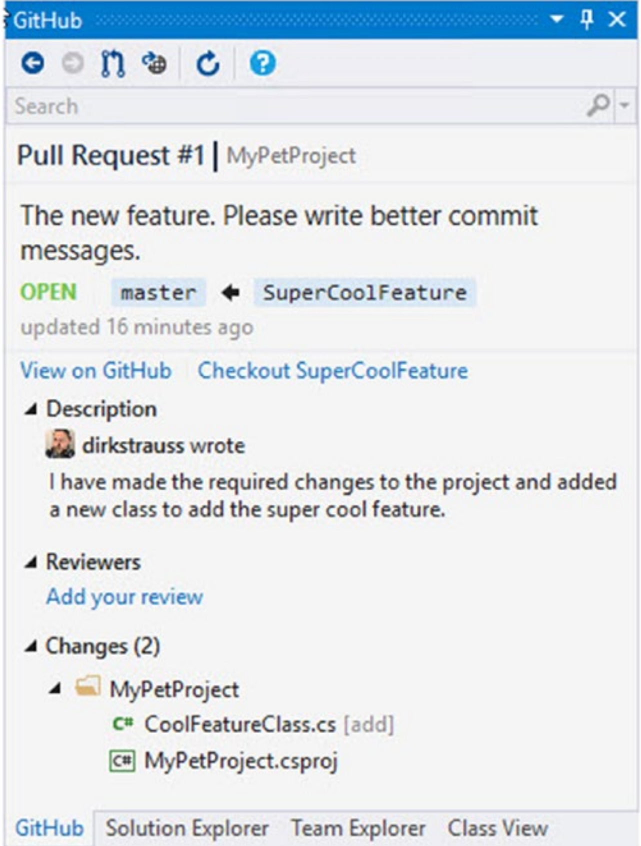


Figure 5-38. *View Pull Request Details*

In Figure 5-38, John can see that I only added a new class called `CoolFeatureClass`. He can also see that the .csproj file has changed.

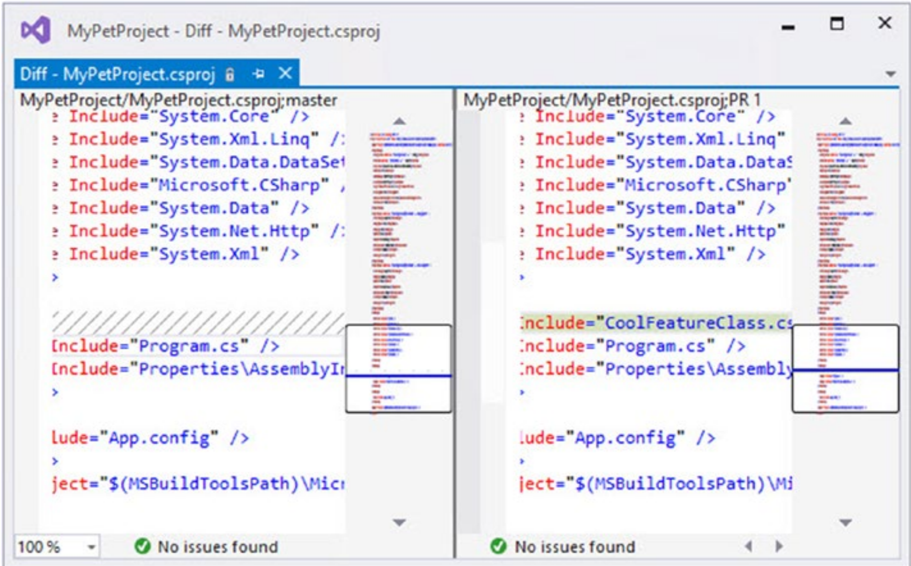


Figure 5-39. *View Differences*

Clicking the .csproj file will display the diff between the file in master and the file in the pull request (Figure 5-39). In fact, John can do this with every file that has been modified. This way, he can clearly see what code has changed and do a review of the code I have added.

Clicking the new class I added, John will not see a diff (because this is a new class), but he is still able to review the code.

```
Program.cs
MyPetProject/CoolFeatureClass.cs;PR 1
Miscellaneous Files

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MyPetProject
{
    public static class CoolFeatureClass
    {
        public static void FeatureAdded()
        {
            Console.WriteLine("This is a cool feature");
        }
    }
}
```

Figure 5-40. *Review code in Pull Request*

Hovering your mouse over the code, John will see a plus sign appear (Figure 5-40).

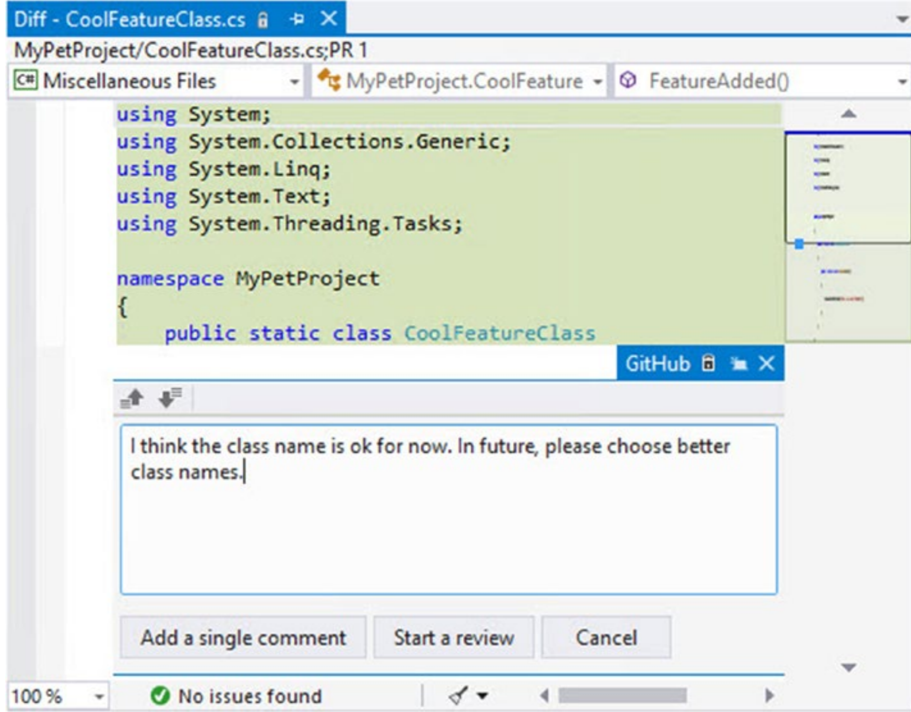


Figure 5-41. *Adding Comments to Pull Requests*

Clicking the plus sign will allow John to add a comment to the code I have added. Once the comments have been added, I can see these in the pull request in Visual Studio (Figure 5-42).

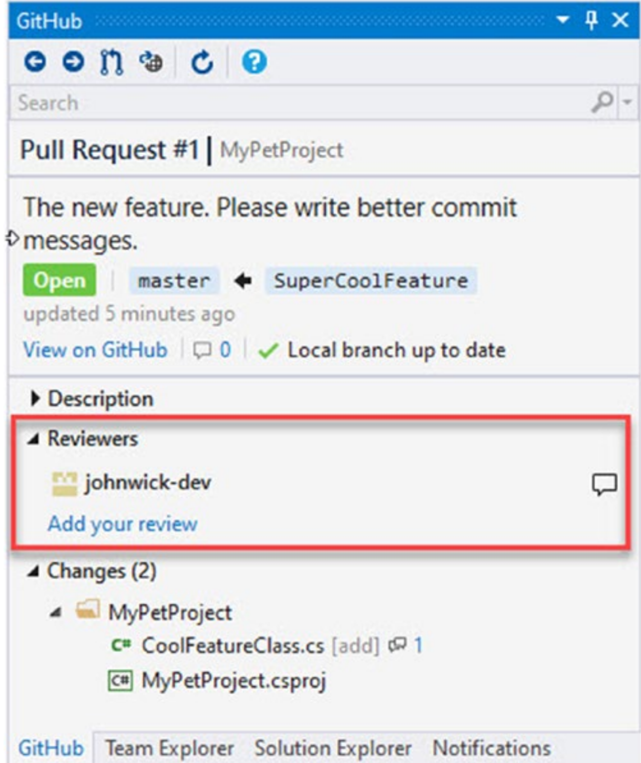


Figure 5-42. *View Pull Request Comments*

Under Reviewers in Visual Studio, I can see that John has added a comment. I can click the comment to view the details of it (Figure 5-43).

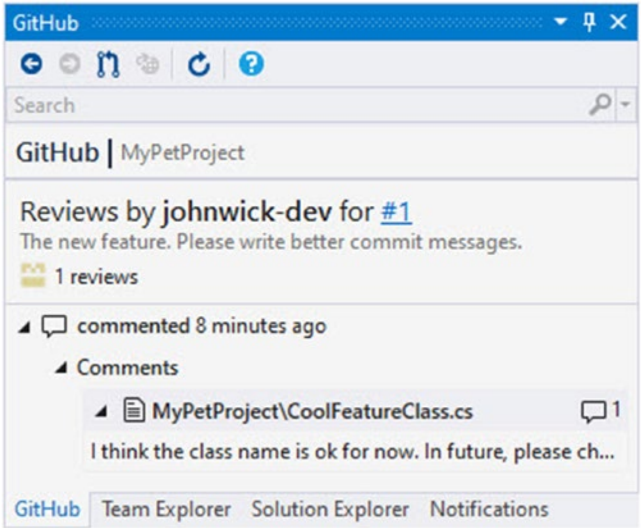


Figure 5-43. *View review comments*

This allows me to see the comments John added and take any action if needed. Referring back to Figure 5-38, John can now click the Add your review link under Reviewers and then approve the pull request (Figure 5-44).

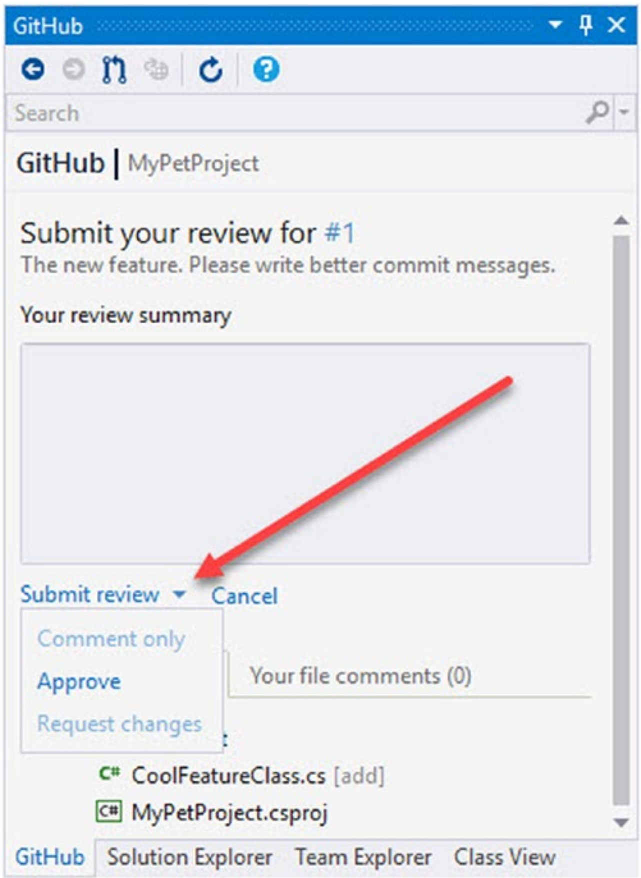


Figure 5-44. *Approving a Pull Request*

When John clicks the Approve link, the pull request is approved, and we can now merge the changes into the master branch. For this, we need to go to GitHub to do the merge.

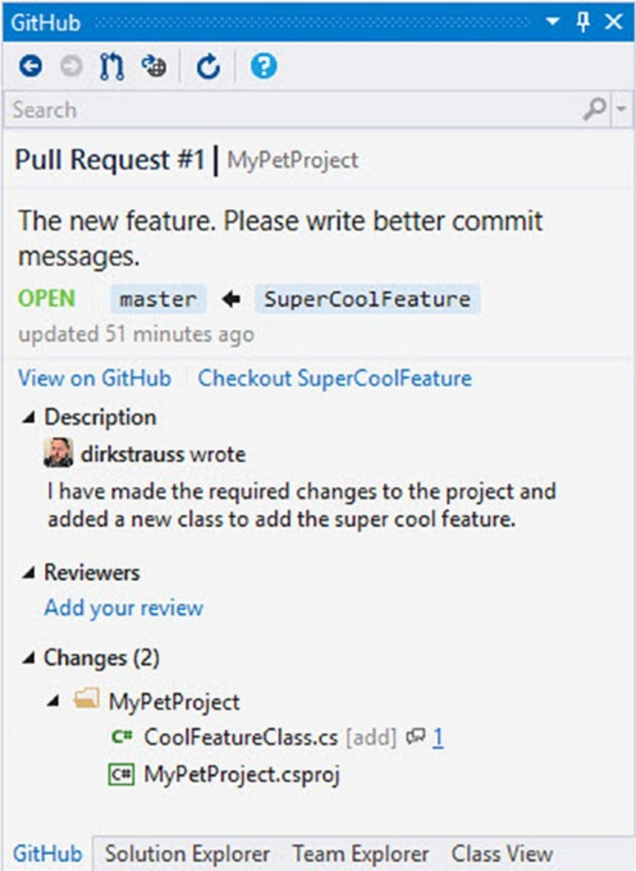


Figure 5-45. *Click View on GitHub to perform the Merge*

By clicking the View on GitHub link (Figure 5-45), John will be taken to GitHub to perform the merge. GitHub will open on the pull request page where he can see the comments that were added. From there he can merge the changes into the master branch (Figure 5-46).

Add more commits by pushing to the SuperCoolFeature branch on johnwick-dev/MyPetProject.
Changes approved
1 approving review Learn more.
Show all reviewers
1 approval
This branch has no conflicts with the base branch
Merging can be performed automatically.
Merge pull request
You can also open this in GitHub Desktop or view command line instructions.

Figure 5-46. *Merge Pull Request*

After the merge is complete in GitHub, I can safely delete my branch. When I go and refresh my GitHub tab, I will see that the Pull Request has been merged into the master branch.

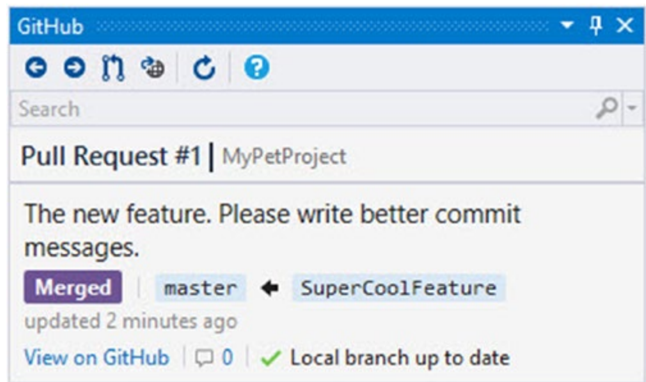


Figure 5-47. *Pull Request Merged into the master branch*

I can now switch to my master branch and pull the changes to get the new feature into my local master branch. Switch to your master branch by clicking the branch name in the bottom right toolbar of Visual Studio and selecting master from there.

You can also switch to the Branches view in Team Explorer and double click the local master branch to switch to it. Then, under the

Synchronization tab in Team Explorer, I will see all the incoming commits as seen in Figure 5-48.

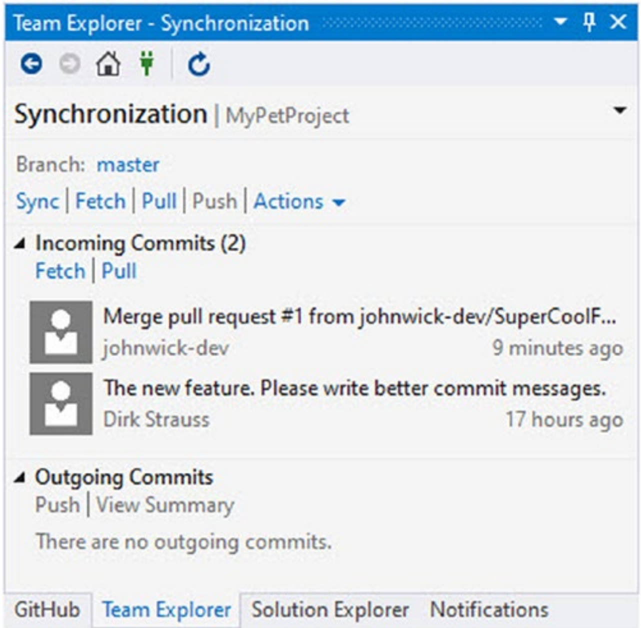


Figure 5-48. *View Incoming Commits*

Notice that the branch displayed is the master branch. This is because we switched to our local master branch in Git. The new feature was merged with the remote master branch on GitHub by John a few minutes ago. I need to pull those changes into my local master branch to get it up to date. To do this, I click the Pull link.

Fetch only downloads the changes from the remote repository (GitHub) but does not integrate the code into your local branch. Fetch just really shows you what changes there are that need to be merged into your local branch.

Pull is used to update your local branch with the latest changes on the remote repository. This merge might potentially result in merge conflicts that you need to resolve before continuing.

After the Pull has completed and the changes have been merged into my local master branch, my Solution Explorer will show the new class I added earlier to my feature branch, in my local master (Figure 5-49).

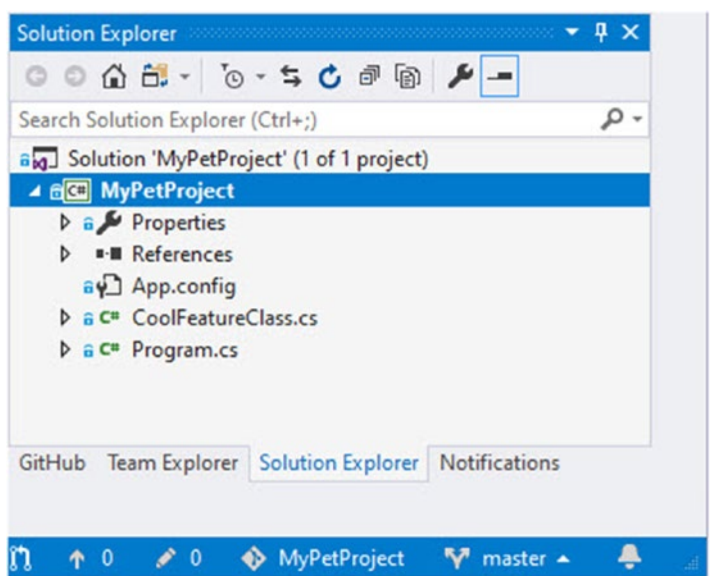


Figure 5-49. *Local master branch merged*

At this point, because the changes have been merged into the master branch, and my local master branch has been updated, I can safely delete the feature branch I created earlier.

Using pull requests allow developers to have a lot more control over the code that gets merged into the main working branch of the project. Using branches allow me to make changes to the code in an isolated manner without risking the stability of the master branch.

Working with Stashes

Sometimes you might be working on some changes, and you continue to make a whole range of changes without noticing that you are working on the wrong branch.

In Figure 5-50, you can see that we are currently working on the master branch. I should actually be making all my changes on the NewFeatures branch. This is a very easy mistake to make (perhaps not with the master branch), especially if you are working in several different branches in your code.

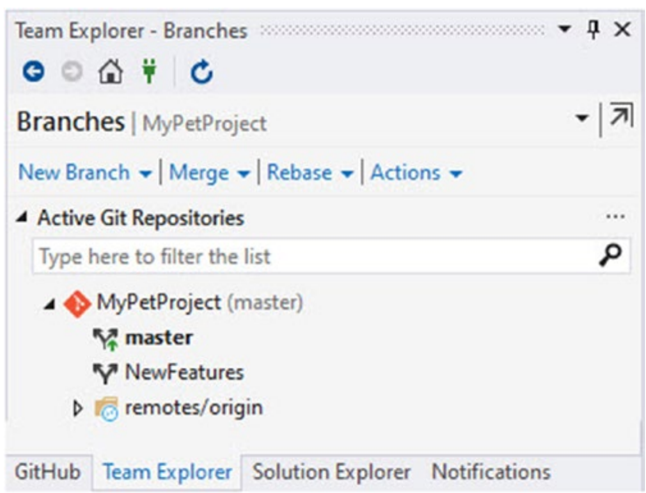


Figure 5-50. *Working in the master branch*

Switching to the changes tab in Team Explorer, I notice that I have made all my changes on the master branch (Figure 5-51) instead of on the correct NewFeatures branch.

Enter a world of pain, because I now need to backtrack everything I did and remove the code and then go and apply these to the correct branch. This is where stashes come in very handy.

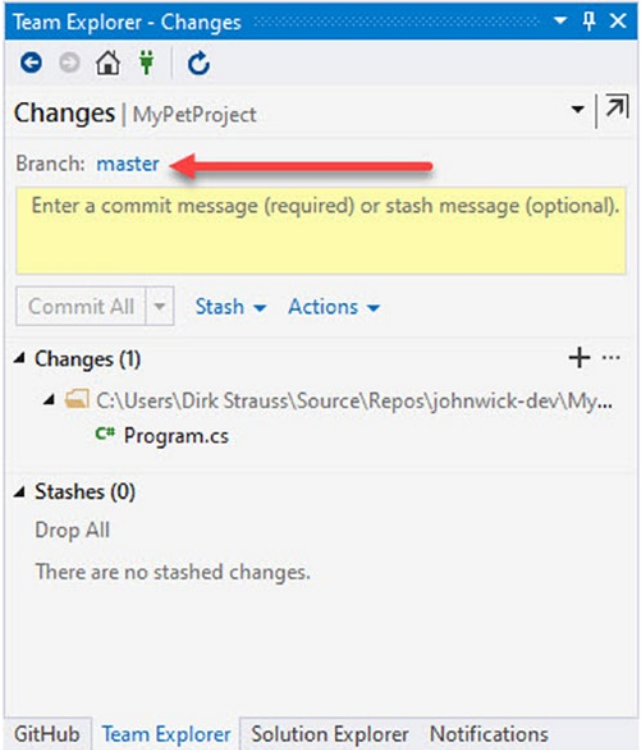


Figure 5-51. *Changes in incorrect branch*

Stashing takes all the changes I have made and puts them away locally (Figure 5-52). It then reverts all the changes I had made to the master branch. This means that I have my master branch back to the way it was before the changes were made.

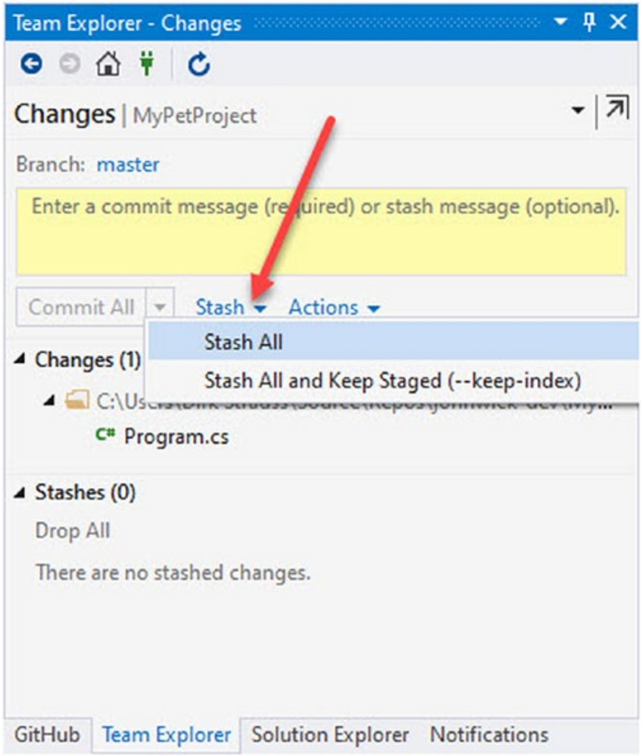


Figure 5-52. *Stash the changes on master*

When I stash my changes, they appear under the Stashes section (Figure 5-53).

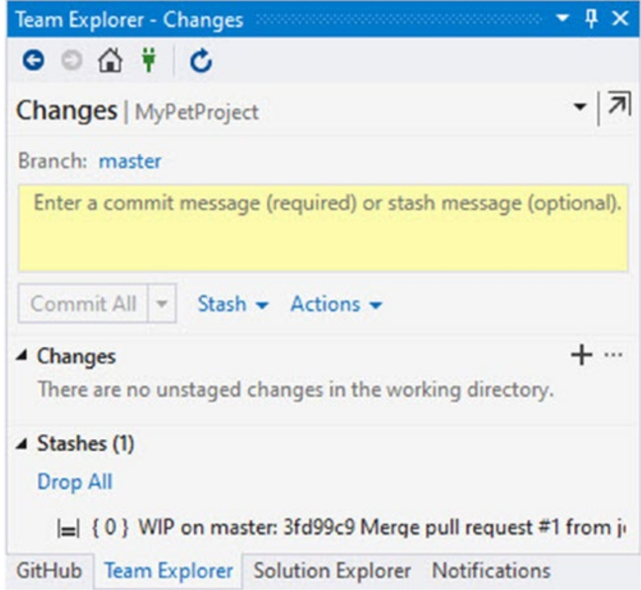


Figure 5-53. *Changes stashed*

I can then go and switch to the correct branch as seen in Figure 5-54.

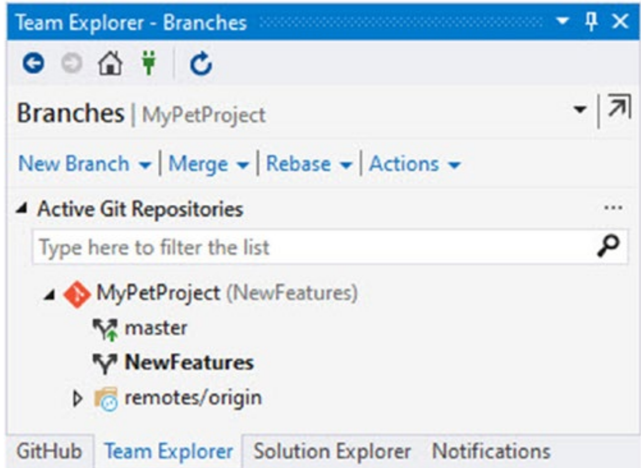


Figure 5-54. *Change to correct branch*

With my correct branch selected (Figure 5-55), I can view the changes, apply them, pop them, or drop the changes. The options are

- Apply – Apply the changes to the branch and keep the stash.
- Pop – Apply the changes to the branch and drop the stash.
- Drop – This will delete the stash without applying anything.

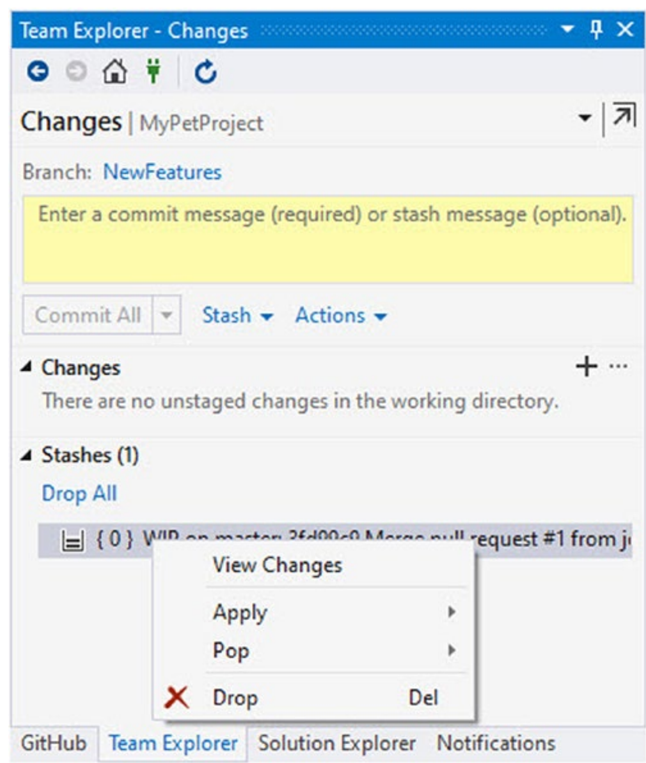


Figure 5-55. *Apply, Pop, or Drop the stash*

Stashing allows me to pause the changes I was working with and carry on with something else for a while. Another great example of using stashes is when I am working on a branch and I need to make a bug fix. I can stash my changes which will revert the code in my branch. Then I can make the bug fix and push that up to the server before popping my stash back to my branch. Stashing allows developers to be very flexible when working with code changes.

Index

A, B

Breakpoints, 124
- conditional and actions, 130
  - action expression, 133, 134
  - conditional expression, 131
  - context menu, 130
  - filter condition, 132
  - hit count condition, 131
  - keywords, 132
- cursor works, 129–131
- debug Toolbar, 125
- export option, 138
- GenerateWaybill() method, 126
- labels
  - edit labels, 136
  - new label window, 137
  - set breakpoints, 135
  - window, 136
- run to click button, 128
- setting, 124
- start button, 125
- step buttons, 125
- step into specific, 127
- ValidateLogin() method, 124
- Waybill generation code, 126

C

C# Interactive, 117, 118

D, E, F

Debugging code
- breakpoint (*see* Breakpoints)
- DataTips, 139
  - bonus tip, 143
  - comment, 140
  - ConvertListToDataTable() method, 141
  - DataTable visualizer, 143
  - source window, 139
  - visualize complex data, 140–143
  - watch window, 144
- DebuggerDisplay attribute, 144
  - item values, 146
  - lstSubjects variable values, 145
  - subject class modification, 146
  - view list items, 145
- diagnostic tools, 152
  - butterfly view, 157
  - CPU usage tab, 155–157

© Dirk Strauss 2020
D. Strauss, *Getting Started with Visual Studio 2019*,
https://doi.org/10.1007/978-1-4842-5449-3

Debugging code (*cont.*)
  enable option, 152
  events view, 159, 160
  memory usage, 157–159
  monitor memory, 154
  performance tools, 160
  tools window, 153, 154
 functions without side effects, 147
  adding nse value, 150
  format specifiers, 151
  HasSubjects() method, 149, 150
  screenshots, 148
  student class, 147
 immediate window
  breakpoint, 161
  DisplayMessage function, 162, 163
  variable value, 162
 points, 123
 remote tools
  application steps of, 171–173
  download and installation, 168, 169
  main screen, 173
  project properties page, 171
  running tools, 169, 170
  system requirements, 167, 168
 running process
  port assignments, 166
  process window, 163–165
  reattach to process, 166
  remote process, 165

G

Git and GitHub account
 pro subscription, 212
 repository creation, 213
 sign up, 211

H

Handling pull requests
 approve, 249
 comments, 247, 248
 creation, 241, 243
 details, 242, 244
 local master branch, 254
 master branch, 252
 merge action, 250, 252
 notification, 243
 review code, 246
 review comments, 249
 view differences, 245
 view incoming commits, 253

I, J, K

Integrated development environment (IDE), 9
 code editor, 21
  application design, 22
  code-behind, 22, 23
  code suggestions, 26
  errors and warnings, 25
  improvement suggestions, 24

solution explorer
add references, 13
context menu, 11
debug folder, 17
differences, 18
features, 14
file explorer, 16
obj and bin folder, 14, 15
project reference, 11
projects, 10
reference manager screen, 12
shipment locator application, 9
solution options, 18
toolbar, 13
WinForms application, 11
toolbox, 19–21
context menu, 20
WinForm application, 19
Intermediate Language (IL), 11

L, M

Live unit tests
benefits of, 186
Clone method, 190, 191
configuration, 186, 187
container class, 189
faulting method, 190
ICloneable class, 189
results, 190
results updated, 188
test frameworks, 186
window, 187, 188

N, O, P, Q, R

NuGet packages, 73
EncryptValidate, 76
hosting solutions, 82, 83
installation, 79
login form, 74, 75
manage packages, 75, 76
manager screen, 77
references, 79
ValidateLogin method, 80
versions, 78
ZIP file, 73

S, T

Source control solution, 209
branch code
changes feature, 238
commit all and push, 239
creation, 233
feature, 236, 237
new creation, 235
server, 239
team explorer, 234
clone repository, 214
add collaborators, 228
add solution, 217
changes local, 226, 227
check out code, 230
commit staged code, 226
connection, 221
GitHub, 231
GitHub extension, 215

Source control solution (*cont.*)
GitHub tab, 219
invitation, 229
MyPetProject solution, 216
online creation, 225
output window, 217, 218
publish settings, 223
synchronization screen, 222
synchronization view, 227, 228
Sync selection, 227
team explorer, 220, 233
window screen, 231
Windows start menu, 214
Git and GitHub, 210–213
pull requests (*see* Handling pull requests)
stashes, 255–260
Stashes, 255
apply/pop/drop, 259
correct branch, 258
incorrect branch, 256
master branch, 255
stash changes, 257, 258

U

Unit testing, 175
code coverage, 204
change fonts and colors, 206
feature, 204
lines, 207
results, 205
toggle coloring, 206
creation and run code
add project, 176
code conversion, 176
failed test results, 181
modification, 180
reference class to test, 178
results, 179
solution, 177
test explorer menu, 182
testing timeouts, 184, 185
test playlist, 182–184
test project, 178
IntelliTest, 191
calculate class, 199, 200
calculate class modification, 193, 195
CalculateTest partial class, 197, 198
focusing code exploration, 202–204
generation, 195, 196
PexAssume, 198, 200
results, 193
ShippingCost method, 192, 198
ShippingCost tests, 201
Live (*see* Live unit tests)

V, W, X, Y, Z

Visual Studio
bookmarks
collection, 98
icon, 95, 96

- renamed bookmark, 97
- window, 96

code metrics results
- class coupling, 121
- cyclomatic complexity, 121
- depth of inheritance, 121
- large project, 120
- lines of code, 122
- maintainability index, 121
- measurements of, 118

code shortcuts, 98
- indicator, 99
- task list, 99
- TODO comments, 100

code snippets
- insert, 87, 88
- logging, 93
- logging Class, 91
- namespace, 94
- tools menu, 88, 89
- try-catch block, 88
- try-catch snippet, 92–94
- XML file, 90

custom tokens, 101
- adding option, 101
- NOTE token comment, 103

evolution of, 1, 2

IDE (*see* Integrated development environment (IDE))

installation, 2, 3

IntelliCode, 51
- completion list, 51
- model completion, 53
- recommendations, 53
- string array, 54
- window pop-up note, 51

Live Share
- code current position, 60
- icon session, 55, 56
- Jason navigation, 59
- launch application notification, 58
- in progress, 56
- session, 57, 59
- sharing link, 56
- tab, 60

multi-caret editing
- code properties, 42
- insert option, 44
- line/across lines, 43
- matching selections, 42
- selection, 42
- SQL table statement, 41

navigate code, 26
- bar, 27, 28
- breadcrumb path, 35
- forward and backward commands, 26, 27
- Go To commands, 31, 32
- Go To definition, 33, 34
- highlighting references, 30, 31
- peek definition, 34–36
- references, 28–30

Visual Studio (*cont.*)
NuGet (*see* NuGet packages)
productivity and features
clipboard history, 39, 40
context menu, 37
edit location, 41
file explorer, 38, 39
Go To window, 40, 41
keyboard shortcut, 39, 40
multi-caret editing, 41–44
track active item, 36, 37
project types
class library, 72, 73
console applications, 67, 68
context menu options, 63
creation, 65
filter templates, 66
screen, 62
section, 64
service, 69–71
timer namespaces, 70
toolbar button, 65
web applications, 71, 72
windows forms application, 68, 69
search option, 44, 45
server explorer
connection, 106
database, 107
data source, 105
data tools operations, 111
preview database, 110
SQL queries, 111–115
statement creation, 108
table designer, 108
updated design, 109
view menu, 104
solution filters, 45
AcmeCorpShipping project, 47
context menu, 50
explorer, 45, 46
file name, 48, 49
project selection, 49
Save As option window, 47, 48
SQL queries
context menu, 111, 112
Insert statement, 114
select statement, 112, 113
table updated, 115
system requirements, 4
additional notes, 6
hardware, 4
languages, 5
operating systems, 4
templates
add options, 86
export template, 84
new project template, 86, 87
ProjectUtilities project, 82, 83
wizard, 85

versions of, 3
Windows menu, 116
  C# Interactive, 117, 118
  code metrics
    results, 118–122
  send feedback, 122
workloads
  details, 8
  installation of, 7
  options, 9
  Python development, 8
  screen, 6