# Using Git For Source Control In Visual Studio 2019

The following is a list of sections in this document. Using Microsoft Word, you can use these as hyperlinks to navigate to any particular section. But **u**sing Apache Open Office, these hyperlinks do not work; instead, they merely serve as a table of contents. You can navigate to the start of any section via bookmarks; type F5 to bring up the Navigator; they double-click Bookmark1 for 1st section header, Bookmark 2, for 2nd section header, etc.

|  |  |
| --- | --- |
| [Create a Local Git Repository](#_Create_a_Local) | [Create a Repository for an Existing Solution](#_Create_a_Repository) |
| [Push to a Remote Repository](#_Push_to_a) | [Clone a Git Repository](#_Clone_a_Git) |
| [Commit and Push Changes](#_Commit_and_Push) | [Fetch and Pull Changes](#_Fetch_and_Pull) |
| [Resolve Merge Conflicts](#_Resolve_Merge_Conflicts) | [Create and Checkout a Branch](#_Create_and_Checkout) |
| [Merge a Branch Back into Master](#_Merge_a_Branch) | [Create and Push Tags](#_Create_and_Push) |
| [Continue to Work on a Branch](#_Continue_to_Work) | [Merge vs. Rebase in Visual Studio](#_Merge_vs._Rebase) |
| [Connect to an Azure Repo](#_Connect_to_an) | [Install the Pull Requests for Visual Studio Extension](#_Install_the_Pull) |
| [Review and Comment a Pull Request](#_Review_and_Comment) | [Push an Update for a Pull Request](#_Push_an_Update) |
| [Approve and Complete a Pull Request](#_Approve_and_Complete) | [View and Compare Commits](#_View_and_Compare) |
| [Browse the History of a File](#_Browse_the_History) | [Use CodeLens to View Changes](#_Use_CodeLens_to) |

## Create a Local Git Repository

* Open Visual Studio (without reference to any existing solution).
* Click the link Continue without code (in the lower-left corner of the home page).
* When working with a newly installed instance of Visual Studio, you may need to install Git as a Source Control plug-in. To remedy this . . .
  + Use the pull-down menu **Tools > Options**. Visual Studio displays a pop-up dialog titled **Options**.
  + Type “Source Control” in the **Search Options** text box – upper-left corner. Visual Studio displays “Source Control Plug-in Selection” in the left panel of the dialog.
  + Click Source **Control**. Visual Studio displays the section titled **Plug-in Selection** in the right panel of the dialog.
  + If “Git” is not displayed in the **Current source control plug-in** combo box, change it to “Git”.
* Open the **Team Explorer** via the **View** pull-down menu.
* Click the “house” icon in the toolbar (top of page). Visual Studio takes you to the **home** page of **Team Explorer**.
* For 1st-time use, check the “Git Settings”. Click the **Settings** icon. If the list of settings is not shown, click the **Global Settings** link. Visual Studio displays the **Settings** page.
* Make changes to anything displayed that is not up-to-date; then click the **Update** command button.
* Navigate to the “Connect” page. You can either (1) click the green double-prong icon in the toolbar, or (2) click the down-arrow at the right side of the row immediately below the toolbar and continue with **Projects > Manage Connections**. Visual Studio displays the current list of Service Providers – followed by the list of Visual Studio Git Repositories (if any).
* You can click the **New** drop-down list, or you can repeat the previous step and click **Projects > New Repository** (instead of **Projects > Manage Connections**). In either case Visual Studio displays a path for the new repository and a **Create** command button.
* Type the name of the repository at the end of the path. (The Pluralsight instructor suggests the name “WiredBrainCoffee.ShopInfoTool”. But bearing in mind that the name of the repository will be the same name as that of the folder containing the Visual Studio solution, I chose to the name “UsingGitForSourceControlInVisualStudio2019”.) Next click the **Create** button. Visual Studio adds the new repository to the list of Git Repositories. Visual Studio has also initialized the Git repository, which you can verify via the file explorer.
* You can open this new repository by double-clicking it, or by **right-click > open**.
* Click the “house” icon in the toolbar to go back to the **Home** page. Note the name of the repository under the toolbar, and that Visual Studio provides a few repository details if you hover the mouse over the name.
* Click the **Changes** icon to see the brief history of this new repository. You will note that the newly created files **.gitattributes** and **.gitignore** are “staged”; this means that they have been “added” but not yet “committed”.
* Let us commit the 2 Git files. Type “add git files” into the “Enter a message” text box; then click **Commit Staged** in the pull-down list below the text box. Visual Studio displays a note confirming the **Commit**.
* Click the “house” icon in the toolbar to go back to the **Home** page.
* Click **New…** under **Solutions** at the bottom of the team-explorer display. Visual Studio displays the **Create a new project** page in the center pane.
* You may need to use the **Search** window at the top of the page to find a link to the “Console App (.Net Core)” template. (Be careful to select a template that indicates your preferred language – e.g. C#.) Select the link to the template, and click the **Next** command button. Visual Studio displays the **Configure your new project** page.
* You may prefer to use a wrapper folder for the solution, and a subfolder for the project. In this case unclick the **Place solution and project in the same directory** check-box (at the bottom of the dialog).
* Type appropriate names for the solution and project names; in my opinion I find it good practice to use the same name for the solution and wrapper folder. Following the Pluralsight instructor’s example, I typed “WiredBrainCoffee.ShopInfoTool” into the **Project name** text box. You will probably find that the Location is the same as the location for our repository; if not, correct it. Finally click the **Create** command button. Visual Studio initializes the console app with a **Program.cs** page and a Solution Explorer (whose name shows up at the bottom of the Team Explorer page).
* Change the Console.WriteLine() argument (in **Program.cs**) to “Wired Brain Coffee – Shop Info Tool!”. Click the floppy-disk icon (in the toolbar) to save the change.
* Click the **Changes** icon (in the Team Explorer Home page). Observe the “added” files are now Program.cs, the .csproj file for the new project, and the .sln file for the new solution. Type “Add project” into the “Enter a message” text box; then click the **Commit All** command button.
* Use the drop-down list (under the toolbar) to navigate to the **Branches** page. Right click either row – WiredBrainCoffee.ShopInfoTool (master) or master – and select **View History…** command button. Visual Studio displays the history log in the center pane. You can drill down by clicking any row of the history log, and Visual Studio displays the files that were committed at that point in time.

## Create a Repository for an Existing Solution

Remove the **.git** folder, **.gitattributes**, and **.gitignore** from the solution that was created in the previous section. The remaining files comprise an ordinary Visual Studio solution/project that is no longer under Git revision control. We can use this as an example of how to create a repository for an existing solution.

* Open Visual Studio for the existing solution.
* Right click the solution name in the name in the Solution Explorer. Visual Studio brings up a pop-up menu.
* Click the menu item **Add** Solution **to Source Control…** . Visual Studio brings back the **.git** folder, **.gitattributes**, and **.gitignore**.
* A very easy alternative is to click **Add to Source Control** at the right edge of the status bar; then click **git** when it pops up. (The status bar is located at the bottom of Visual Studio’s screen.) (You can try this alternative by removing – again - the **.git** folder, **.gitattributes**, and **.gitignore**.)
* After adding the solution to source control, observe the **git** status displayed at the right edge of the status bar.
* Notice, also, that after either of the methods of adding the solution to source control, Visual Studio has automatically **commit**ted all of the appropriate files. (Refer to the end of section [Create a Local Git Repository](#_Create_a_Local) for notes on how to see what files have been committed.)

## Push to a Remote Repository

The purpose of this section is to back up the Git repository to a remote location (e,g, another server or the cloud). Navigate back to our Visual Studio demo example. Then – from the Team Explorer/Home, click the **Sync** icon. There you should see the options **Push to Azure DevOps Services** and **Push to Remote Repository**. The Pluralsight instructor chose the 2nd of these two options.

The instructor’s demo suggested using an existing Azure DevOps account – with an existing repository. At the outset I did not have an existing Azure DevOps account; but I did have a Microsoft account, which I used to create the DevOps account.

Navigate to https://dev.azure.com. This web page will invite the new Azure DevOps user to open a new account. You will need the user-id and password of a Microsoft account. You will also need to supply the name of your 1st DevOps repository. The name that I chose was **UsingGitForSourceControlInVisualStudio2019**.

* Under **Push to Remote Repository** click the **Publish Git Repo** command button. Visual Studio displays (1) a new text box with the instruction **Enter the URL of an empty Git repo** and (2) a new command button labeled **Publish**.
* We need to go to Azure DevOps to collect the URL. Use an internet browser to navigate to the new DevOps repository. On the left edge of the home screen for this repository, you will see a vertical row of named icons. Click on the icon titled **Repos**. Azure DevOps displays a screen titled “*RemoteRepositoryName* is empty. Add some code!”
* Under **Clone to your computer** the right edge of the 1st line has an icon depicted as 2 overlapping sheets; when you hover over this icon Azure DevOps displays **Copy Clone URL to Clipboard**. Click this icon. Azure DevOps inserts a string into the clipboard.
* Return to Visual Studio, and paste the string into the text box labeled **Enter the URL of an empty Git repo.** Then click the command button labeled **Publish**. In order to complete this operation, I had to supply the user-ID and password of my Microsoft account. Visual Studio displayed output that confirmed that the new remote repository had received some data.

Go back to Azure Devops – under the **Repos** section (as described above). The Contents tab displays the list of files that have been committed; the **History** tab displays the list of commits.

Go back to Visual Studio – Team Explorer/Branches. **Remotes/origin** has been added. Under Team Explorer/Settings there is now a section (near the bottom) titled **Remotes**.

## Clone a Git Repository

For this section we will need to pretend to be a 2nd team member. I created a new Windows 7 user. More recently I have purchased a 2nd computer, which runs on Windows 10.

* Navigate to https://dev.azure.com. This will automatically start Azure Devops in the existing organization.
* Click **Organization Settings** (at bottom left of screen).
* Click **Users** – in pane at left. Azure DevOps pops up a **Users** dialog.
* Click the **Add users** command button. Azure DevOps pops up a **Add new users** dialog.
* Select **Basic** in the **Access** level pull-down list. Type the users Microsoft-account email address into the text box under **Users.** Select the project you want the new user to share (e.g. UsingGitForSourceControlInVisualStudio2019). Finally, click the **Add** command button Azure Devops displays a row in the **Users** dialog as a confirmation that the user has been added.
* Devops also responded to the previous step by sending an invitation e-mail to the new user.

Log into Windows 7 or Windows 10 as the new user.

* Create a blank folder into which you plan to hold the cloned repository. I created a folder titled **OtherUserRepos** under sncole\source (as a brother to **GitRepositories**). Secondly I created a subfolder titled **UsingGitForSourceControlInVisualStudio2019**.
* Open Visual Studio. Under **Get Started** you will see a link titled **Clone a Repository**. Click it. Visual Studio pops up the **Clone a Repository** page.
* For the **Local path** text box, I browsed to C:\Users\sncole\source\OtherUserRepos\UsingGitForSourceControlInVisualStudio2019.
* The Pluralsight instructor shows us a 1st approach, which involves clicking **Azure DevOps** below **Browse a repository**. We won’t be using this approach, because the 2nd approach will be similar to what we would do with other types of foreign repositories (e.g. GitHub).
* To get a URL for the 2nd approach, start by logging into Devops. You should land in the home screen.
* To go to the files in the checked in repository (in my example its name is “UsingGitForSourceControlInVisualStudio2019”) hover the mouse over the repository name to cause the dots below to show links. Click the link that looks like a branch. A slower (but easier-to-remember way is to (1) click the repository name, (2) then click **Repos > Files** – starting from the menu in the left pane.
* There are two buttons in the upper-left corner of the **Files** page. Click the **Clone** button.
* Devops pops up a dialog titled **Clone** Repository**.**
* Copy the HTTPSURL into the clipboard.
* Go back to Visual Studio. Paste the URL from the clipboard into the **Repository location** text box.
* Click the **Clone** button in the lower-left corner of the screen.

There appeared to be a problem with the results of the clone when examining the cloned repository at C:\Users\sncole\source\OtherUserRepos\UsingGitForSourceControlInVisualStudio2019; the **.git** folder appeared to be missing, and the names of **.gitattributes** and **.gitignore** were blank. But these problems were not due to Devops nor Visual Studio. Instead, it was simply the fact that the newly created user required some file-visibility settings. Use Google with the keywords **windows show full file name** and **windows show hidden files** to resolve the problem.

Next we are going to clone a repository from GitHub.

* In Visual Studio, select **continue without code**. Visual Studio shows an empty Solution Explorer, and a Team Explorer with a reminder of recent activity.
* In the Team Explorer go to the **Manage** **Connections** page; use the branch-shaped icon immediately to the right of the **Home** icon – in the Team Explorer toolbar. Visual Studio displays the **Connect** page.
* Expand Local **Git Repositories** (if necessary) to expose 4 links - **New**, **Add**, **Clone**, and **View Options**.
* Click **Clone**. Visual Studio expands the **Connect** dialog – with a text box that you can use to supply the URL of a repository.
* Click the **…** button to the right of the text box that will contain the repository URL. Visual Studio opens a file browser.
* Use the browser to find (or create) a folder where the cloned repository will reside. This folder must be empty.
* Open a web browser, and log into GitHub (assuming that you have previously pushed git repositories there; I have done so). Select an organization that has one or more git repositories. The list of such repositories should be in a vertical array in the left-hand pane.
* Click the name of a repository to clone. When the toolbar button **<>Code** is selected, GitHub displays the history of git activity for this repository in the center pane. Above the center pane are one or more buttons, one of which is labeled **Code** (surrounded on both sides by down arrows).
* Click the **Code** button that is above the center pane. GitHub pops up a dialog titled **Clone**.
* Select **HTTPS** on the line immediately below the title. GitHub shows un underscore beneath **HTTPS.**
* Click the URL in the textbox below **HTTPS**. GitHub highlights the text to indicate that it is selected.
* Click the button to the right of the URL text box. This instructs GitHub to copy the highlighted text into the clipboard.
* Return focus to Visual Studio. Paste the URL from the clipboard to the text box in the expanded **Connect** dialog. Finally click the **Clone** button at the bottom of the expanded **Connect** dialog. Visual Studio clones the solution into the indicated local folder. This completes cloning from GitHub.

## Commit and Push Changes

* In Visual Studio reopen the solution UsingGitForSourceControlInVisualStudio2019. Add a new class library to the solution. The type of the class library is “.NET Standard – C#, Windows”. (To find the template for this class library, use the **Search** box in the **Add a new project** dialog.) Supply the name of the project – WiredBrainCoffee.DataAccess. The location should be, by default, the same folder where the project WiredBrainCoffee.ShopInfoTool resides.
* Use the Solution Explorer to delete the **Class1.cs** file.
* Add a new folder in the WiredBrainCoffee.DataAccess project. Rename the new folder “Model”.
* Add a new class under the “Model” folder, whose name is “CoffeeShop”. Visual Studio should have created the file CoffeeShop.cs under the “Model” folder.
* Edit CoffeeShop.cs; insert “public” in front of “class CoffeeShop”.
* Add 2 properties within the body of class CoffeeShop

public string Location { get; set; }

public int BeansInStockInKg { get; set; }

* Put the mouse cursor on a **using** statement; type <ctrl>. Visual Studio invites you to let it remove the unnecessary **using** statements. Type the **Enter** key to say “yes”.
* Add new class in the WiredBrainCoffee.DataAccess project, whose name is “CoffeeShopDataProvider”.
* Make this class **public**.
* Add the following to the body of CoffeeShopDataProvider

public IEnumerable<CoffeeShop> LoadCoffeeShops()

{

yield return new CoffeeShop { Location = "Frankfurt", BeansInStockInKg = 107 };

yield return new CoffeeShop { Location = "Freiburg", BeansInStockInKg = 23 };

yield return new CoffeeShop { Location = "Munich", BeansInStockInKg = 56 };

}

* Put the mouse cursor on a using statement; type <ctrl>. Visual Studio invites you to let it remove the unnecessary **using** statements. Type the **Enter** key to say “yes”.
* Next edit program.cs. We need a using statement “ using WiredBrainCoffee.DataAccess.Model; “. We are also adding code immediately after the “Console.WriteLine” statement.

Console.WriteLine("Write 'help' to list available commands");

var coffeeShopDataProvider = new CoffeeShopDataProvider();

while (true)

{

var line = Console.ReadLine();

var coffeeShops = coffeeShopDataProvider.LoadCoffeeShops();

if (string.Equals("help", line, StringComparison.OrdinalIgnoreCase))

{

Console.WriteLine("> Available coffee shop commands:");

foreach(var coffeeShop in coffeeShops)

{

Console.WriteLine($"> " + coffeeShop.Location);

}

}

}

* Build and run the application. Note that the “help” command works; the application lists the locations of the available coffee shops.

**This note is not pertinent to Git; instead, it pertains to “yield return” in LoadCoffeeShops().** It is instructive to test the “help” command and to follow the code in Visual Studio’s debugger starting at “var coffeeShops = coffeeShopDataProvider.LoadCoffeeShops()”. Use F11 (Step Into) repeatedly. Notice that execution control does not jump into LoadCoffeeShops() when executing the “var coffeeShops = . . . “ instruction. But when stepping through the **foreach** loop (3 lines later) execution does jump into LoadCoffeeShops() to fetch each CoffeeShop element in coffeeShops. This illustrates that **yield return** is an efficient way to defer execution until the result is needed.

Now let us “commit” and “push” our changes (git terminology).

* Navigate to Team Explorer – the **Home** tab. Click the **Changes** icon. Visual Studio provides an overview of the changes. This display has 2 options – (1) the tree-view option is hierarchical; (2) the list-view option indicates (explicitly) which files are either new or changed; the added files are indicated by the suffix **[added]**. You can toggle between the two views by clicking the **…** icon to the right of **+** on the line that contains **Changes**; the pop-up menu contains either **Switch to List View** or **Switch to Tree View**.
* You can also see the changes in the Solution Explorer: changed files are preceded by a red check mark. Added files are indicated by a green “+” as a prefix.
* If you right click any red-checked (changed) item in the Solution Explorer, the pop-up menu contains 5 options that are pertinent to git:

Undo…

Commit…

View History…

Compare with Unmodified…

Blame (Annotate)

* Clicking **Commit…** is equivalent to clicking the **Changes** icon in Team Explorer (inviting you to write a comment and to commit the added or changed items).
* Try right-clicking a file on the **Changes** dialog that has been changed (not added) – for example program.cs. Visual Studio displays a pop-up menu that contains the option **Compare with Unmodified…** (which we saw above in the pop-up menu when right-clicking the **Solution** row).
* Continue by clicking **Compare with Unmodified…** Visual Studio displays a split-screen comparison of the file – before and after the changes.
* From the **Changes** dialog, you could commit the changes for all of the files: type some text (e.g. “Implement help command”) in the box labeled “Enter a message” (near the top of the dialog);then click the down arrow to the right of **Commit All**. Visual Studio provides several flavors of “committing”.

Commit All

Commit All and Push

Commit All and Sync

then click **Commit All**.

* But suppose you wanted to commit some of the files separately from the other files. For example, you might want to commit the new files in the new project (WiredBrainCoffee.DataAccess). Right-click the folder “WiredBrainCoffee.DataAccess”, and click **Stage** in the pop-up menu. Visual Studio displays the result: 3 of the files have been staged, and there are 3 remaining files to deal with. Let us also stage the solution file: right-click the file with .sln extension, and click **Stage** in the pop-up menu. The updated status is now 4 files staged and 2 files remaining files to deal with. Next, type some text (e.g. “Add data access project to solution”) in the box labeled “Enter a message”. Finally, click the down arrow to the right of **Commit Staged**, and click **Commit Staged**. Visual Studio displays a brief confirmation near the top of the **Changes** dialog.
* For the remaining (changed) files, enter text (e.g. “Implement help command” in the box labeled “Enter a message”, and click **Commit All** to commit the remaining files.
* Go to the **Branches** page (use down arrow to right of **Changes** to display the menu, and click **Branches**).
* Right-click **master** – near the bottom of the page. Then click **View History…** in the pop-up menu. Visual studio displays the history in the center pane, and our 2 recent commits are listed at the top.
* Next, we want to push these commits to the remote project. Go to the **Synchronization**  page (use down to right of **Branches** – in Team Explorer, and click **Sync**.) Visual Studio displays an indication of 2 **Outgoing Commits**.
* It is good practice – before pushing outgoing commits – to check for incoming commits. Click **Fetch** below **Incoming Commits**. The status – that there are no incoming commits – has not changed. Therefore, click the **Push** command. Visual Studio displays confirmation that the “push” was successful.

We will look at the remote repository (to see our changes) in the next section, where we go back to acting as the other team member.

## Fetch and Pull Changes

* Assume the identity of the 2nd team member – either a different user in Windows 7 or a user in Windows 10. Open the Visual Studio example that we have been using; the solution name is “UsingGitForSourceControlInVisualStudio2019”.
* Open the Team Explorer.
* Go to the **Synchronization** page (click the **sync** command from the **Home** tab).
* Click **Fetch** under **Incoming Commits**. In this example the 1st team member has made two commits, and Visual Studio shows them under **Incoming Commits**.
* Click **Pull** under **Incoming Commits**. Visual Studio retrieves the new and changed files. **Pull** does a “fetch” and a “merge”.
* Prove that the updated application works by running it; type the “help” command (as above) to verify that it lists the locations of the coffee shops.
* The 2nd team member notices an inconsistency in the text displayed to the user: the invitation says “Write 'help' to list available commands”; but the 1st line written in response to ‘help’ is “> Available coffee shop commands:”. To make them consistent the 2nd team member inserts “coffee shop” in front of “commands” in the invitation.
* Navigate to the **Changes** dialog in Team Explorer. Right-click the changed file (Program.cs) shown in the **Changes** dialog, and select **Compare with Unmodified…**. This lets you compare the previous revision with the newer revision.
* Now let’s commit this change. Type “Adjust help message.” into the “Enter a message” text box; click the drop-down list identified by **Commit All**, and **Commit All and Push**. (If you have not yet saved the change to the local computer, Visual Studio will prompt you to save the change.) Finally Visual studio performs the commit-and-push action, and displays a confirmation note near the top of the **Synchronization** dialog.

We will look at the remote repository (to see our change) in the next section – from the vantage of the 1st team member.

## Resolve Merge Conflicts

We are now acting as the 1st team member. As usual, we navigate to our example in Visual Studio. The 1st team member is contemplating a 2nd command, “quit”. First he expands the invitation to read “Write 'help' to list available commands, write ‘quit’ to exit application”. Obviously, this statement is inconsistent with the change just made by the 2nd team member, but the 1st team member is probably not aware of the 2nd team member’s change. When these 2 versions are merged, this statement will present a conflict that requires human intervention to resolve it.

* To implement the ‘quit’ command insert the following immediately after “var line = Console.ReadLine();” (near the top of the “while” loop.

if ( string.Equals ( “quit”, line, StringComparison.OrdinalIgnoreCase ) )

{

break;

}

* Run the application to prove that the change works.
* Now let’s commit this change. Navigate to the **Changes** page in the Team Explorer. Type “Implement quit command.” into the “Enter a message” text box; invoke **Commit All** from its dropdown list.
* Move to the Synchronization page (the **sync** button).
* Let’s see what happens if we forget to check for incoming commits; click the **Push** command. Visual Studio displays an error message in a pop-up window. Similar information is displayed in the **Output** window after you select **Source Control – Git** from the **Show output from:** dropdown list. Notice also the failure note at the top of the **Synchronization** page.
* In the Synchronization page, under **Incoming Commits**, click the **Fetch** command to discover changes that were evidently made by another team member. Visual Studio researches the remote repository, finds the commit from the other team member, and displays the information under **Incoming Commits**.
* In the Synchronization page, under **Incoming Commits**, click the **Pull** command to merge this commit with our code. Visual Studio begins the merge but encounters a conflict. Refer to the instructions displayed in the **Synchronization** page. The conflict is also illustrated in the source code the file that has the conflict.
* Visual Studio has more tools that could be quite useful in analyzing a merge conflict. Click **Conflicts:** in the **Synchronization** page to navigate to the **Resolve Conflicts** page. It shows that this (single) conflict resides in the Program.cs file.
  + Click the file name (Program.cs); Visual Studio displays 3 comparison options and 3 action options.
  + You could keep the local file (changes made by the 1st team member) by clicking **Keep Local**; the **Diff** command to the left of **Keep Local** provides a side-by-side comparison of the local changes.
  + You could keep the remote file (changes made by the 2nd team member) by clicking **Take Remote**; the **Diff** command to the left of **Take Remote** provides a side-by-side comparison of the remote changes.
  + You could merge the conflicting changes by click the **Merge** button; the **Compare Files** command below the **Merge** button provides a side-by-side comparison of the files changed locally and remotely.
* The 1st team member has decided that he likes the change made by the 2nd team member, and he also wants to retain his changes. Click the **Merge** button on the **Resolve Conflicts** page. What we see on the top half of the screen are the two conflicting changes. On the bottom half we see the partially resolved merge; specifically the string comparison with “quit” has already been included in the merged result. The arrow keys in the upper-left corner - at the top of the 3-way split screen - can be used to navigate forward and backward when the file contains more than one conflict. In this (simplified) instance there is only one conflict – denoted by dashed-line borders.
* The easiest way to retain both changes is to click the check box adjacent to the local version of the row in conflict. Visual Studio copies this local change into the merged result below. To capture the remote rewording, we can copy the new words “coffee shop “ into the clipboard and paste them into the merged result. One also has the option of typing new text into the merged result.
* When we are satisfied with the merged result, click the **Accept Merge** button (to the left of the arrow keys in upper-left corner). Visual Studio updates the text in the **Resolve Conflicts** page of Team Explorer.
* Click Commit **Merge** – at the top of the **Resolve Conflicts** page. Visual Studio reverts to the **Changes** page.
* Right-click Program.cs, and select **Compare with Unmodified…** to review the merge-operation changes.
* The Pluralsight instructor did not suggest this, but in my opinion it would also be a good idea to run the application to confirm that the merged text executes as you would wish.
* Type “Merge help-instruction change – from team member 2.” into the “Enter a message” text box; then click **Commit Staged**.
* Navigate to the **Synchronization** page. Visual Studio now displays two commits that are ready to pushed to the remote repository.
* Click **Push** to move the changes to the remote repository.
* Navigate to the **Branches** page of Team Explorer. Then right-click on the **master** branch, and click **View History** in the pop-up menu. Visual Studio displays the history in the center pane.
* Click the 3-vertical-dot icon in the upper-left corner to display history graphically – with divergence between the 2 team members, and with convergence in the merge process.

## Create and Checkout a Branch

In this section we are going to proceed with our implementation of the Wired Brain Coffee application. But instead of making changes as we did earlier with the “quit” command, we will create a new branch; and we introduce the enhanced code in this new branch. This will cause less interference with other team members who may want to fix bugs or fine tune the work that has already been done on the “help” and “quit” commands (in the **Master** branch).

* Open the Visual Studio solution for Wired Brain Coffee.
* Go to the Team Explorer, Home Tab, and click the **Branches** icon. Next click the **New Branch** command in the tool bar. Visual Studio presents the **Branches/New Branch** dialog.
* What we need to specify at this point is (1) the name of the new branch – into the **Enter a branch name** text box under the tool bar and (2) another branch that we want to branch from – the dropdown list below the new branch name textbox. Type “feature/coffeeShopCommands” into the **Enter a branch name** text box. We don’t really have a choice with regard to the selection of the other branch; **master** is the only existing branch until now.
* At this stage we defer checking out the branch while creating it; therefore, uncheck the **Checkout branch** check box.
* Click the **Create branch** command button. Visual Studio creates this new branch “coffeeShopCommands” in a team-explorer folder titled “feature”.
* Notice – at the right edge of the status bar (at the bottom of the Visual Studio screen) that the currently checked out branch is still **master**. To start work on the new branch either double-click the branch named coffeeShopCommands (in the Team Explorer **Branches** page), or right-click coffeeShopCommands, and select **Checkout** from the pop-up menu. Notice that after checking out coffeeShopCommands, the name of the current branch changes in the status bar.
* Open the Program.cs file. Add new code after block

if ( string.Equals(“help”, line, StringComparison.OrdinalIgnoreCase))

{

. . .

}

The new code is . . .

else

{

var foundCoffeeShops = coffeeShops.Where ( x => x.Location.StartsWith

( line, StringComparison.OrdinalIgnoreCase ) ).ToList();

if ( foundCoffeeShops.Count == 0 )

{

Console.WriteLine ( $”> Command ‘{line}’ not found” ) ;

}

else if ( foundCoffeeShops.Count == 1 )

{

var coffeeShop = foundCoffeeShops.Single();

Console.WriteLine ( $”> Location: {coffeeShop.Location}” ) ;

Console.WriteLine ( $”> Beans in stock: {coffeeShop.BeansInStockInKg} kg” ) ;

}

else

{

Console.WriteLine ( $”> Multiple matching coffee shop commands found:” ) ;

foreach ( var coffeeType in foundCoffeeShops )

{

Console.WriteLine ( $”> { coffeeType.Location}” ) ;

}

}

}

The **.Where** method of coffeeShops (at the beginning of the added code) has a wavy red underscore, which denotes some kind of error. Placing the mouse cursor on **.Where**, and typing <ctrl>. allows us to discover that a **using** clause that refers to **System.Linq** is needed. (Type the **Enter** key to accept the addition of this **using** clause.)

* Test the application.
  + Type “help” to list the valid commands
  + Type “Thomas” to see the error message when the command is not found.
  + Type “munich” to see what happens if a single command matches the text that was typed.
  + Type “fr” to see the error message when multiple commands match the text that was typed.
  + Type “frank” to see what happens if a single command matches the abbreviated text that was typed.
  + Type “quit” to exit.
* Go to Team Explorer to commit the changes. Navigate to the **Changes** page. Visual Studio displays the fact that there is one change.
* Type “Handle coffee shop commands.” Into the “Enter a message” text box. This time, we select **Commit All and Push** from the **Commit All** dropdown list. Visual Studio displays an indication of success with regard to the “feature/coffeeShopCommands” branch.
* Examine the Team Explorer **Branches** page. Click (to expand) **remotes/origin** node, and click (to expand) the “feature” folder. Visual Studio displays “coffeeShopCommands” to show that this branch is now in the remote repository.

## Merge a Branch Back into Master

We are now acting as the 2nd team member. As usual, we navigate to our example in Visual Studio. This team member will look for any changes made by the 1st team member, and will then decide what to do with them (if any).

* Navigate to Team Explorer, **Branches** page. The **master** branch is highlighted, which indicates that this is the branch that is currently checked out.
* Switch to the **Synchronization** page (**Sync** option in the pull-down list). Click **Fetch** under **Incoming Commits** – to discover whether the 1st team member has committed any changes. Visual Studio displays 2 commits – implement “quit” command and merge help-instruction change.
* Check to see what the 1st team member did with regard to merge-help-instruction change. Double-click its commit notification. Visual Studio displays the **Commit Details** dialog, and the bottom part of this dialog indicates that he changed the program.cs file.
* Right-click program.cs, and select Compare **with Previous…** Visual Studio displays the comparison by showing the previous/current side-by-side, and by highlighting the rows that have changed. If this 2nd team member is alert, she will also observe some unrecognized code with regard to the “quit” command.
* Switch back to the Synchronization page, and double-click the commit with regard to the “quit” command. As before, Visual Studio displays the **Commit Details** dialog with program.cs noted as the changed file.
* Right-click program.cs, and select **Compare with Previous…** Visual Studio displays the comparison as before, but this time it highlights the changes pertaining to the “quit” command.
* From the **Synchronization** page, click **Pull** under **Incoming Commits** – to bring this code into the local copy of this application. After pulling the code Visual Studio displays a confirmation that the local repository has now been updated.
* The next step is to check for branches. Navigate to Team Explorer, **Branches** page. Click (to expand) **remotes/origin** node, and click (to expand) the “feature” folder. Visual Studio displays “coffeeShopCommands” to show that this branch has been added.
* Right-click “coffeeShopCommands”, and then click **Checkout** to get a local copy of this branch. Visual Studio highlights “coffeeShopCommands” – under the 1st instance of “feature” in the tree in this dialog.
* The 2nd team member does not know – at this point – what changes were made in this branch. To get this detail, from the **Branches** page, right-click the branch name “coffeeShopCommands”, and click **View History** in the pop-up menu. Visual Studio displays the history in the center pane. Notice that the top line (most recent commit) contains the comment “Handle coffee shop commands”.
* Right click the top line in the history, and click **View Commit Details**. Visual Studio displays the details of this commit in the Team Explorer view – in a page titled **Commit *xxxxxxxx***, where ***xxxxxxxx***is the code for the commit. At the bottom of this dialog Visual Studio displays the source files that were changed; in this case the file is Program.cs.
* Use Solution Explorer to look at Program.cs. The 2nd team member observes that it now contains an else clause to implement other commands. It is convenient to toggle between the “coffeeShopCommands” and “master” branches by using the button at (or near) the right edge of the status bar (at the bottom of the screen); this button shows a double arrow, a branch name, and an upward-point triangle. Click it to induce a pop-up menu, and select the name of the branch that you want to select.

This 2nd team member could choose to work with the 1st team member to complete work on the “coffeeShopCommands” branch. But she has a more pressing duty – to introduce a 3rd branch – pertaining to the inventory of paper cups.

* Navigate to Team Explorer, **Branches** page. Click the **New Branch** button in the toolbar. Visual Studio expands the dialog to add a text box for the name of the new branch, a drop-down list to select the name of the branch to branch from, a check-out check box, and two command buttons.
* Type “feature/paperCupsInStock” into the **Enter a branch name** text box. Select “master” as the branch that this new branch will branch from. Set the **Checkout branch** check box to checked. (We intend to start work immediately in the new branch.) Click the **Create Branch** command button. Visual Studio creates the new branch, displays it in the “feature” folder. (We observe that “paperCupsInStock” is checked out – highlighted in the “feature” folder, and displayed in the status bar at the bottom of the screen.)
* We have no interest in dealing with the “coffeeShopCommands” at the present time. To simplify our work environment, right-click “coffeeShopCommands”, and click **Delete** in the pop-up menu. (It remains available in the remote depository, and we can verify that by expanding nodes under “remotes/origin”.)
* Navigate to the Solution Explorer, WireBrainCoffee.DataAccess/Model/Coffee.cs. This gives us access to the CoffeeShop class.
* Add a 3rd property:

public int PaperCupsInStock { get; set; }

* Update the CoffeeShopDataProvider class to supply values for the PaperCupsInStock property – by adding the clause

, PaperCupsInStock = *n*

for each of the 3 locations: for Frankfurt substitute 350 for *n*; for Freiburg substitute 250 for *n*; for Munich substitute 427 for *n*.

* Commit the changes. Navigate to Team Explorer, **Changes** page. Type the commit message (in the **Enter a message** text box) – Add PaperCupsInStockproperty.
* Double-check the top line of this page to verify that we are on the paperCupsInStock branch. Click the **Commit All** dropdown list, and then the **Commit All and Push** entry in the list. Visual Studio asks you to confirm changes to CoffeeShop.cs and to CoffeeShopDataProvider.cs. Respond by clicking the **Save** command button. Visual Studio displays a confirmation message.
* The 2nd team member is done (for now) with the new feature, and she is ready to merge the new branch back with the “master” branch. Before continuing with the merge it is good practice to see if there is any recent commit to the “master” branch (from another team member). Navigate to the **Branches** page. Double-click the **master** branch node (to make it the currently selected branch). Right-click **master**, and click **Pull** from the pop-up menu. Visual Studio confirms that the local repository is up-to-date.
* Double-click the “paperCupsInStock” to make it (once again) the selected branch. Click **Merge** in the tool bar. Visual Studio probably displays “feature/paperCupsInStock” in the **Into current branch:** text box; change it to “master” by double-clicking the master-branch node in the branches tree in the lower part of the page. Select “feature/paperCupsInStock” in the **Merge from branch:** pull-down list. Uncheck the **Commit changes after merging** check box. Click the **Merge** command button. Visual Studio displays an indication that the merge operation is in progress.
* The merge operation does not complete until you perform the commit operation – you must navigate to the **Changes** page. Note the file changes to be committed – in the **Staged Changes** section. If you want to double-check, you can right-click any of the files to compare the unmodified and modified versions of the changes.
* Type “Merge feature/paperCupsInStock” into the **Enter a message** text box (near the top of the page). Select the **Commit Staged** dropdown list, and click **Commit Staged and Push**. Visual Studio displays notification that the commit and push were successful.
* Now that the recent branch has been merged with the master branch, we no longer need it. Navigate to the **Branches** page. If we expand the tree – at the nodes remotes/origin and “feature”, we see “paperCupsInStock” in two places. The instance closer to the top represents the instance in the local repository. Right-click this instance, and then click **Delete** to remove it. The instance closer to the bottom represents the instance in the remote repository. Right-click this instance, and then click **Delete Branch from Remote** to remove it; respond **Yes** to the “Are you sure . . .” pop-up dialog.

## Create and Push Tags

We continue acting as the 2nd team member. As usual, we navigate to our example in Visual Studio.

* Click the button at (or near) the right edge of the status bar (at the bottom of the screen); this button shows a double arrow, a branch name, and an upward-point triangle. Click it to induce a pop-up menu, and select **View History**.
* Navigate to the **Home** page of Team Explorer. Click on the **Tags** link to navigate to the **Tags** page. Click **New Tag** to start the process of adding a new tag. Visual Studio expands the **Tags** page – adding text boxes for the tag name and tag message, adding a check box to associate the tag with the current node of the current branch, and adding command buttons.
* Type “v0.5” for the tag name, and “Version 0.5 with paper cups in stock” for the tag message. Check the **Create tag against tip of current branch** check box. Click the **Create Tag** command button. Visual Studio confirms success by showing the tag name and tag message.
* Navigate back to the history list, and click the refresh button – upper-left corner. The tag name should show in the history, but as of Oct, 2020, the tag name no longer shows; the show-tags button in the toolbar above the history (near or at the right edge of the toolbar) appears to do nothing. This bug is being fixed at this time.
* It is also possible to create a tag directly in history view. For example, one might want a tag for the node at the end of the merge of the “quit” command and the improvement made by team member 2 (i.e. where the commit message is “Merge help-instruction change – from team member 2”). Right-click anywhere on the text for that node, and click **Create Tag** in the pop-up menu. Visual studio opens the **Commit Details** page of Team Explorer with the invitation to create a new tag.
* Type “v0.4” for the tag name and “Version 0.4 with quit command” for the tag message. Again the refresh button fails to display the tag in the history – due to the bug whose fix is in progress.
* Now we are ready to push these tags to the remote repository. Navigate to the **Tags** page of Team Explorer. Visual Studio displays the list of tags that have been committed. One can right-click a tag to get a pop-up menu and push tags individually. Instead, we want to push all of the tags to the remote repository; therefore, we click **Push All** in the toolbar. Visual Studio display a notification that the pushes were successful.

These tags can be used to fix a bug that was introduced earlier. One can click an early tag, create a branch starting at that tag, and fix the bug.

## Continue to Work on a Branch

We are now acting as the 1st team member. As usual, we navigate to our example in Visual Studio. We will continue to work on the “coffeeShopCommands” branch.

* Navigate to the **Branches** page in Team Explorer. It shows that we are on the “coffeeShopCommands” branch (also evident from the name at the right end of the status bar at the bottom).
* The code for this branch is in Program.cs, and this code is beginning to present a management burden. We need to refactor. Select all of the code in the else block (at the end of Program.cs), and type <ctrl>x to move it to the clipboard.

*It is awkward to* ***cut*** *text into the clipboard, and to defer* ***past****ing it somewhere else until we manufacture the destination module. It is easily likely that the clipboard text will be accidentally overwritten before the time arrives to* ***paste*** *it. An easy remedy is to comment-out the text instead of* ***cut****ting it. Then when the time comes to move the text, one can cut/paste it and then remove the commenting.*

In place of the deleted code type

var commandHandler = new CoffeeShopCommandHandler ( coffeeShops, line );

(CoffeeShopCommandHandler is not yet defined; coffeeShops is an instance of our data model; ‘line’ is the text that the user entered.)

* With the mouse hovering over “CoffeeShopCommandHandler”, click <ctrl>., and select the option of creating the CoffeeShopCommandHandler class in a new file.
* After the line in Program.cs that defines commandHandler, type

commandHandler.HandleCommand();

Type <ctrl>. and accept the invitation to implement HandleCommand() as a method in the CoffeeShopCommandHandler class.

* Navigate via Solution Explorer to the CoffeeShopCommandHandler class. This contains skeletons for the constructor and the HandleCommand() method. Change the access of HandleCommand() from “internal” to “public”.
* Paste the statements from the clipboard into the body of HandleCommand() – replacing the”NotImplementedException()” statement that was produced by default. As we did earlier apply <ctrl>. to “.Where” to induce the “using System.Linq;” statement. (We can also remove “using System.Linq;” from Program.cs.)
* Similarly we are refactoring the code for the “help” command, which appears in the block immediately before the last “else” statement in Program.cs. <ctrl>X (i.e. **cut)** the code in this block, or (preferably) defer **cut** by commenting out the code temporarily.
* Replace the omitted code with

var commandHandler = new HelpCommandHandler ( coffeeShops);

* With the mouse hovering over “HelpCommandHandler”, click <ctrl>., and select the option of creating the HelpCommandHandler class in a new file.
* Immediately after the inserted line above type

commandHandler.HandleCommand();

Type <ctrl>. and accept the invitation to implement HandleCommand() as a method in the HelpCommandHandler class.

* Navigate via Solution Explorer to the HelpCommandHandler class. This contains skeletons for the constructor and the HandleCommand() method. Change the access of HandleCommand() from “internal” to “public”.
* Paste the statements from the clipboard into the body of HandleCommand() – replacing the”NotImplementedException()” statement that was produced by default.
* Replace

$"> " + coffeeShop.Location

with

$"> {coffeeShop.Location}"

* More improvements are possible. Notice that we have two similar classes, both of which contain the method HandleCommand(). This suggests an interface. Put the mouse cursor on the class declaration HelpCommandHandler, and click <ctrl>. Visual Studio displays a pop-up menu.
* Click Extract **Interface…** Visual Studio displays a pop-up dialog titled **Extract Interface**.
* Change the **New interface name** to ICommandHandler. The default for the file name (**New file name:** ICommandHandler.cs) is what we want, and check the check box adjacent to HandleCommand(). Click the **OK** command button. Visual Studio creates the iCommandHandler interface, and it has changed the HelpCommandHandler to show that it implements the interface.
* Type “: ICommandHandler” at the end of the class declaration CoffeeShopCommandHandler, so that this class similarly implements the interface.
* Now in Program.cs we can factor out the statement

commandHandler.HandleCommand();

There are 2 instances. Delete one of them, and move the other to the 1st statement after the **else** block, but do this after we declare commandHander before the last **if** statement using the interface:

ICommandHandler commandHandler;

and remove “var” from the commandHandler assignment statements in the **if/else** block.

* There is another simplification. We can replace the **if/else** statements with code that initializes commandHandler at the end of the statement where it is declared:

ICommandHandler commandHandler =

string.Equals("help", line, StringComparison.OrdinalIgnoreCase) ?

new HelpCommandHandler(coffeeShops) :

new CoffeeShopCommandHandler(coffeeShops, line);

Unfortunately Visual Studio finds fault with this code. The error message is CS0173: Type of conditional epression cannot be determined because there is no implicit conversion between ‘WiredBrainCoffee.ShopInfoTool.HelpCommandHandler’ and ‘WiredBrainCoffee.ShopInfoTool.CoffeeShopCommandHandler’. The Pluralsight instructor found a work-around. (1) append “as ICommandHandler” after HelpCommandHandler(coffeeShops), and (2) change “ICommandHandler” – at the beginning of the statement to “var”.

* After testing the application it is time to commit the changes. Navigate to the **Changes** page of Team Explorer. Visual Studio displays the names of the new or changed files. Type “Introduce command handlers.” into the **Enter a message** text box. Click **Commit All and Push** from the **Commit All** dropdown list. Visual Studio displays a confirmation that the push was successful.
* To find out if another team member has made any changes, navigate to the **Branches** page. Double-click **master** to set current branch to the master branch. (Note that “master” is displayed near the bottom right corner of the page.)
* Navigate to the Synchronization page. Click the **Fetch** command. Visual Studio displays the fact that team member 2 has made 2 commits.
* Click the **Pull** command to bring these changes from the remote repository to our local master branch. Visual Studio confirms that the local repository is up-to-date in the master branch.
* But these changes made by team-member 2 have not been merged into the feature/coffeeShopCommands branch. Nor have the changes in feature/coffeeShopCommands been merged into the master branch. The next section will deal with this question.

## Merge vs. Rebase in Visual Studio

Let us review where we stand. Create the following matrix by looking at each source module (1) from the perspective of the master branch, and (2) from the perspective of the feature/coffeeShopCommands. Recall that you can change the perspective by using the pop-up menu obtained by clicking the currently selected branch near the lower-right corner of the screen.

|  |  |  |
| --- | --- | --- |
|  | **master branch** | **feature/coffeeShopCommands** |
| Model/CoffeeShop | all properties: Location, BeansInStockInKg, and PaperCupsInStock | two properties: Location and BeansInStockInKg |
| CoffeeShopDataProvider | returns values for all properties | returns values only for Location and BeansInStockInKg |
| CoffeeShopCommandHandler | missing | available |
| HelpCommandHandler | missing | available |
| ICommandHandler | missing | available |
| Program | does not implement city-name commands; help-command implementation is imbedded in Program.cs | refers to CommandHandler modules for implementation of help command and of city-name commands |

We have two choices.

1. We can merge the master branch into the feature/coffeeShopCommands branch. This would make the feature/coffeeShopCommands the most up-to-date branch. To make the master branch equally up-to-date, we would next merge the updated feature/coffeeShopCommands back into the master branch. It looks unnecessarily complicated because we will have used two merges to bring everything up-to-date.
2. We can use a “rebase” instead; we would rebase the feature/coffeeShopCommands onto the master branch. The rebase requires (a) updating the feature/coffeeShopCommands with the latest commits from the master branch, and (b) imposing the changes in feature/coffeeShopCommands on top of the results of (a). Conflicts – if any – would have to be resolved. This combination would represent the updated status of the feature/coffeeShopCommands branch. (The master branch has not yet been changed.) At this point the history of feature/coffeeShopCommands contains the commits done by team member 2. Finally do a fast-forward merge that fast-forwards the state of the master branch to the same point as that of the feature/coffeeShopCommands branch. You might feel that it would be safe to delete the feature/coffeeShopCommands branch at this point; it would work functionally, but the history of the feature/coffeeShopCommands branch. To retain this history, do an ordinary merge of the master branch – instead of a fast-forward merge, and we can keep the feature branch for the purpose of record retention.

* We resume acting as the 1st team member. We are in Visual Studio with the current branch set to feature/coffeeShopCommands. Go to the **Branches** page of Team Explorer.
* Approach (1) noted above starts with a merge of the master branch into the feature/coffeeShopCommands branch. We’ll start this way to demonstrate how to do it, but we’ll abort at the last moment, and use approach (2).
* Click the Merge pull-down list. We want to merge from the master branch to the current branch (to incorporate the 2nd team member’s changes). Select “master” from the **Merge from branch** pull-down list. (**Into current branch** is already set correctly.) Uncheck the check box **Commit changes after merging**. This allows us to examine the changes made by the other team member. Click the **Merge** command button. Visual Studio confirms that the merge is in progress, and that it is waiting for us to commit.
* Navigate to the **Changes** page. Visual Studio displays the names of the changed files – CoffeeShop.cs and CoffeeShopDataProvider.cs. Double-click each of these in succession to induce Visual Studio display the changes side-by-side.
* We approve of the changes. Type “Merge latest changes from master” into the **Enter a message** text box. We are at the moment of approach (1). Instead of clicking one of the **Comment Staged** options, we click the **Abort** button below. Also, erase the text from the **Enter a message** text box. Visual Studio has undone the staging for the merge.
* Navigate back to the **Branches** page. Starting approach (2), we click the **Rebase/down-arrow** command button. We are doing the **Rebase** from our current (feature) branch onto the “master” branch. Select “master” from the **Onto branch:** pull-down list. Click the **Rebase** command button.
* Navigate to the **Changes** page. There are no changes, because there are no conflicts to be resolved. If there had been conflicts, we would have been notified in the same as with a merge; and we would have had to resolve them in the same was as with a merge.
* Return to the Branches page. Right-click coffeeShopCommands, and select **View History** from the pull-down menu. Examine the history in the center pane.
* Next - to push the recent-history commits to the remote repository – navigate to the **Synchronization** page – with coffeeShopCommands as our current branch. Visual Studio shows us 2 incoming commits and 4 outgoing commits. The commits labeled “Introduce command handlers” and “Handle coffee shop commands” appear to be redundant. The explanation from the Pluralsight instructor is that we had pushed the 2 commits to the remote repository before we performed the **Rebase**. In the remote repository those 2 commits took on a new identity. (Indeed, we can examine those commits, and verify that they have different identifier codes.) When we push the outgoing commits to the remote repository, we will need to overwrite the two commits that are already present in the remote repository. To drive the point home, let’s try to push to the remote repository, and look at Visual Studio’s remarks.
* Click the **Push** button. Visual Studio displays an error message that tells us that **Push** failed. We need to perform a “force” **Push**. But “force” **Push** is not available until we update the settings.
* Navigate to the **Settings** page in the Team Explorer. Click **Global Settings**. Visual Studio displays a dialog titled **Git Settings**. Check the check-box titled **Enable push –force**. Click the **Update** command button.
* Return the Synchronization page. The **force Push** that we are at the verge of performing is dangerous. Other team members may be in the process of working on this coffeeShopCommands branch, and they need to be warned. If they have already pulled the 2 commits, followed by our action of overwriting the commits, ugly side effects may occur.
* Click the **Push** button. We get the same error message from Visual Studio, except that now we are offered the **Force Push** option. Click that button. Visual Studio displays a confirmation message.
* Go back to the **Settings** page, uncheck the **Enable push –force** check box, and click the **Update** command button.
* Return to the Branches page, and update the display of history for the coffeeShopCommandsbranch. It looks very clean now.

## Connect to an Azure Repo

We continue acting as the 1st team member. We work with our example in Visual Studio with the current branch set to coffeeShopCommands.

* Navigate to Team Explorer. Click the icon with 2 dots at the top to go to the **Connect** page. There is a section of this page with the heading **Azure DevOps**. At the bottom of this section there is a button titled **Connect…** that allows conection to an Azure Repo. Click it. Visual Studio displays a pop-up dialog titled **Connect to a Project**. This dialog shows one or more organizations and projects (repositories) that the user (SNCole) has created.
* In particular one of the projects listed in the **Connect to a Project** dialog will be named the same as one of the names in the list at the bottom of the **Connect** page. Select that repository name. Visual studio expands the **Connect to a Project** dialog to show the path of that repository on the local computer, and it exposes a **Connect** button. Click the **Connect** button. Visual Studio changes the page of Team Explorer back to **Home**, and it confirms that we are connected to Azure DevOps.
* Click the icon with 2 dots at the top to return the **Connect** page. Notice that our project is displayed as part of the Azure DevOps remote depository and also selected in the list of local repositories.
* Navigate back to the **Home** page. Notice that there are a few additional icons listed: **Work Items**, **Builds**, and **Pull Requests**. Click **Pull Requests**. Visual Studio displays the **Pull Requests** page, which shows the pull requests that I have made and the pull requests that have been made to me.
* Click the New **Pull Request** button (near the top of the **Pull Requests** page). In response Visual Studio popped up a new web-browser page. (This new web page titled **New pull request** asked for particulars – source and destination branches “master” was shown as the default for the destination branch. I selected feature/coffeeShopCommands for the source branch. The web page then displayed a subtitle “feature/coffeeShopCommands into master”. Below this subtitle were (1) a text box where one enters the **Title** of the pull request, (2) a text box displaying the description of the pull request; the default of the description are the descriptions of the two recent commits. (3) other items to be supplied.
* Instead of continuing with the web browser, the Pluralsight instructor suggested that we use a facility in Visual Studio that eliminates the need to navigate to a web page. To do this we need to install the “pull request for Visual Studio extension”, which is described in the next section.

## Install the Pull Requests for Visual Studio Extension

This course has been dealing primarily with Azure DevOps. For details and downloads pertaining to GitHub refer to <https://visualstudio.github.com>.

We are now downloading the pull-request extension for Azure DevOps. Do the following for all computers that are using Visual Studio with Git.

* Invoke the Extensions pulldown menu in Visual Studio’s top line. Click **Manage Extensions**. Visual Studio exposes the **Manage Extensions** dialog.
* Expand Online in the dialog’s left panel. Visual Studio displays **Visual Studio Marketplace**. Type “pull request” into the search box – in the upper-left corner of the dialog and identified by a magnifying glass. Visual Studio displays – in the center pane a paragraph titled **Pull Requests for Visual Studio**  - provided by Microsoft DevLabs.
* Click anywhere in this **Pull Requests for Visual Studio** paragraph. Visual Studio highlights the paragraph and adds a **Download** command button.
* Click the Download command button. Visual Studio displays notification of the download operation, and it displays a note saying “Your changes will be scheduled.. The modifications will begin when all Microsoft Visual Studio windows are closed.”
* Close Visual Studio. Windows displays notifications from “VSIX Installer”. In particular the last notification contains “Pull Requests for Visual Studio” and “By clicking ‘Modify’, you agree with the above license terms (if any) and the installing of any prerequisites.”.
* Click the Modify command button. Windows completes the modification.

Create a Pull Request

We continue acting as the 1st team member. We work with our example in Visual Studio with the current branch set to coffeeShopCommands. Our objective is to create a pull request to merge the coffeeShopCommands branch into the **master** branch. When we compare the history of the 2 branches, we observe that coffeeShopCommands is 2 commits ahead of **master**.

* Navigate to the **Pull Requests** page in Team Explorer, and click the **New Pull Request** button in the page’s tool bar. Visual Studio displays the **New Pull Request** page. This time – as a result of our having installed the pull-request extension – Visual Studio is soliciting the details instead of a web page. (If the text on the top line of this page does not start with “From feature/ coffeeShopCommands”, the current branch is – incorrectly – not set to coffeeShopCommands. The end of this line will be “to master”, because – in our example – master is the only other branch.)
* Change the text in the **Title:** box to “Handle coffee shop commands”. Change the text in the **Description:** box to “Handle coffee shop commands, and introduce command handlers”. Type “sncole0” into the list of **Reviewers**. This is the name of the 2nd team member, to whom the pull request is directed.

***This explains what a pull request is; a suggestion to another team member to perform the merge.***

* The bottom of the **New Pull** Request page shows the source files that have been changed or added. If you click one of these , it will show the source code as it stands in the currently selected branch (coffeeShopCommands). You have the option of displaying how this source code has changed by clicking the button in Visual Studio’s tool bar whose icon looks like 2 side-by-side lists. This is an alternate-action button; click it to display the comparison, and click it again to turn off comparison.
* Click the **Create** command button – at the middle of the right side of the **New Pull Request page**.Visual Studio confirms that the pull request was successfully built; refer to the notification on the **Pull Requests** page under **Requested by Me**.

## Review and Comment a Pull Request

We resume acting as the 2nd team member. As usual, we navigate to our example in Visual Studio. Click the **Extensions** pull-down menu, and select **Manage Extensions**. Select **Installed** in the left pane, and sort by **Most Recent** (at the top of the center pane). Notice that **Pull Requests for Visual Studio** is installed.

* Navigate to Team Explorer. Select Manage **Connections**; Visual Studio displays the **Connect** page.
* Click the **Connect...** button to navigate to our team's repositories in Azure DevOps. Select the repository that we are working on. Visual studio displays a dialog titled **Connect to a Project** in the center pane. Confirm that the folder displayed at the bottom of the page is the folder for our solution.
* Click the Connect command button at the bottom of the dialog. Visual Studio updates the page in the Team Explorer, and it contains an icon for **Pull Requests**.
* Click the **Pu**l**l Requests** icon. Visual Studio switches to the **Pull Requests** page. Notice that this page shows – at the bottom of the page – the pull request assigned to the 2nd team member, and that this pull request was created by the 1st team member.
* Double-click the name of the pull request (at the bottom of the page). Visual Studio switches to a page titled **Pull Request Details**.
* The top section shows that the 1st team member wants to merge the coffeeShopCommands branch into the master branch. To see the changes that this merge would cause, click the **Open** link. In addition to displaying a description in the center pane, Visual Studio displays a list of the source files that have been added or changed. In this case the only changed file is Program.cs.
* Click Program.cs. Visual Studio displays the merge-base version (master branch) and the changed version (coffeeShopCommands) side-by-side in the center panel. As we observed (in a previous section of this course), a side-by-side icon is available in Visual Studio's tool bar, and you can click it to toggle between side-by-side and changed-version view.
* The 2nd user approves the “var Command-handler = . . .” statement. To express approval, the 2nd user right-clicks the statement; Visual Studio displays a pop-up menu. This menu allows the 2nd user to add a comment, or to add a “Like”.
* Select **Add a Like** from the menu. Visual Studio appends a thumbs-up emo-gee in the margin to the left of the statement.
* The 2nd user now clicks CoffeeShopCommandHandler.cs in Team Explorer. (Use the side-by-side icon in the toolbar – if necessary – to show this source file in changed-version view.)
* The 2nd user focuses on the 3 lines of code

var coffeeShop = foundCoffeeShops.Single();

Console.WriteLine($"> Location: {coffeeShop.Location}");

Console.WriteLine($"> Beans in stock: {coffeeShop.BeansInStockInKg} kg");

She wants to see code that would display the paper-cup inventory.

* Place the cursor at the end of the 1st of the 3 lines. Right-click to induce the pop-up menu. Click **Add Comment** in the pop-up menu. Visual Studio pops up a dialog which is connected (graphically) to the 1st of the 3 lines.
* Type “Hello “. Then click the @ in the pop-up dialog. Visual Studio pops up a 2nd dialog that lists team members that are involved in this pull request. Click an item that represents the 1st team member, e.g. his e-mail address. Next type “, You should include the paper cups in stock.”. Finally click the **Comment**  button in the pop-up dialog. Visual Studio displays the attached comment.
* Focus back on Team Explorer. Click the down arrow to the right of the **Approve** link on the **Pull Request Details** page. Click **Approve with suggestions** in the pull-down list. Notice that Visual Studio has understood the 2nd team member's selected action, and has imposed a check mark on the square that identifies this team member. Finally click the **Exit** link.

## Push an Update for a Pull Request

* We resume acting as the 1st team member. We are in Visual Studio with the current branch set to feature/coffeeShopCommands. The 1st team member has received notification that the recent pull request was approved – with suggestion – by the 2nd team member.
* Navigate to the **Pull Requests** page of Team Explorer. Visual Studio displays the recent pull request, and indicates “Approved with suggestions”. Double-click the name of the pull request. Visual Studio advances to the **Pull Request Details** page.
* Click the **Open** link above the **Description**. Visual Studio expands the options in the **Pull Request Details** page, and lists the pertinent source files. In the center pane – at the bottom – Visual Studio displays **Pull Request Comments**. The text of the comment was preceded by the name of the source file that the comment refers to.
* Click the name of the source file. Visual Studio displays the source file in the center pane. (Use the side-by-side icon in the toolbar – if necessary – to show this source file in changed-version view.) You can use the mouse to drag the comment window to a different position if it covers up code that you need to look at.
* The 1st team member likes the comment. Hover the mouse over the comment text. In response Visual Studio provides two icons: (1) suggesting “reply” and (2) a heart. Click the heart to express “I like your comment”; the heart fill color changes. Click the “reply” icon; Visual Basic provides a new pop-up dialog where one types the response to the comment.
* In the reply text box type “Hello “. Then click the @ in the pop-up dialog. Visual Studio pops up a 2nd dialog that lists team members that are involved in this pull request. Click an item that represents the 1st team member, e.g. his e-mail address. Next type “, that’s a great idea. I’ll add it in an update.”. Then click the **Comment** command button.
* Focus back on Team Explorer. Click the **Exit** link. Verify that we are still working on the coffeeShopCommands branch.
* Navigate to the Solution Explorer. Open the source file CoffeeShopCommandHandler.cs. Focus on the 3 lines of code

var coffeeShop = foundCoffeeShops.Single();

Console.WriteLine($"> Location: {coffeeShop.Location}");

Console.WriteLine($"> Beans in stock: {coffeeShop.BeansInStockInKg} kg");

* Append 1 line

Console.WriteLine($"> Paper cups in stock: {coffeeShop.PaperCupsInStock}");

* Test the application to verify that the change works.
* Navigate to Team Explorer, **Changes** page. Visual Studio shows that CoffeeShopCommandHandler.cs is the only change. Write click CoffeeShopCommandHandler.cs, and click **Compare with Unmodified…** to verify that the added line is the only change. Type the “commit” message, “Write paper cups in stock to console” into the **Enter a message** text box.
* Expand the **Commit All** drop-down list. Click **Commit All and Push** in this drop-down list. Visual Studio commits and pushes the change; it displays **Successfully pushed to origin/feature/** at the top of the **Synchronization** page.
* We already have a pull request in progress for this branch. Therefore, navigate to the **Pull Requests** page. Double-click the name of the pull request. Visual Studio advances to the **Pull Request Details** page.
* It looks slightly different from before. Now, near the bottom of the page, we have a pull-down list **Update 2 of 2**. Expand this list to see the descriptions of the two updates.
* Click the changed source file CoffeeShopCommandHandler.cs from the list at the bottom of the **Pull Request Details** page. Visual Studio displays this code (with earlier comments) in the center pane. (Use the side-by-side icon in the toolbar – if necessary – to show this source file in changed-version view.)
* Right-click the added line of code; select **Add Comment** from the pop-up menu. Visual Studio pops up a dialog where the team member can type comments.
* Using the “@” icon (as before) to include the name of the addressee, type “Hello @<*other team member*>, here are the paper cups in stock”; then click the **Comment** command button. Visual Studio displays this new comment with the source code.
* This completes the work of the 1st team member in complying with the suggestion from the other team member.

## Approve and Complete a Pull Request

We resume acting as the 2nd team member. As usual, we navigate to our example in Visual Studio.

* Navigate to Team Explorer – Home screen. The notification near the top of the page gives assurance that this 2nd team member is still connected to the remote repository and is connected to our project.
* Navigate to the **Pull Requests** page. The pull request that we have been following is listed here.
* Double-click the name of the pull request (at the bottom of the page). Visual Studio switches to a page titled **Pull Request Details**.
* To continue drilling down, click the **Open** link above the **Description**. Visual Studio expands the options in the **Pull Request Details** page, and lists the pertinent source files. In the center pane – at the bottom – Visual Studio displays **Pull Request Comments** – from both team members.
* The 2nd team member knows – from the e-mail that she received – and from the comments in the center pane that the new comments pertain to CoffeeShopCommandHandler.cs.
* Click the name of the source file. Visual Studio displays the source file in the center pane. (Use the side-by-side icon in the toolbar – if necessary – to show this source file in changed-version view.) You can use the mouse to drag the comment windows to different positions if they covers up code that you need to look at.
* The 2nd team member examines the code and comments; she is now completely in favor of these updates. Focus back on Team Explorer. Click **Approve** to approve with no qualifications. Then click **Exit**.
* Navigate back to the **Pull Requests** page; you can use the back-arrow icon in the upper-left corner of the page to “go back”. Unfortunately the Visual Studio Extension does not contain the option to complete the pull request. We will have to use an internet browser to attain direct access to Azaure DevOps. Right-click the name of the pull request (at the bottom of the **Pull Requests** page; then select **Open in Browser**. Visual Studio opens the internet browser.
* The upper-left corner contains information that assures us that we are dealing with our pull request. It shows the name, the author, and the brief description of merging the feature branch into the master branch. In the right-hand pane – under **Reviewers –** we see that the 2nd team member approved this pull request. Other parts of the page show the code changes and comments.
* We complete the response to the pull request by clicking the **Complete** command button at the top of the right-hand pane. Azure DevOps displays a pop-up dialog titled **Complete pull request**.
* Click the **Complete merge** command button in the pop-up dialog. Azure DevOps confirms pull-request completion via a confirmation that it displays near the top of the center pane.
* We need to do a little clean-up with regard to the comments. Scroll down the center pane to the section where the comments are listed. There is a block for the 2nd comment; and shortly below that there is a block for the 1st series of comments. Each of these blocks has a **Resolve** command button, which we can use to dispose these comments (no longer needed). Click the two **Resolve** buttons.
* Focus back on Visual Studio – Team Explorer – **Pull Requests** page. In the toolbar at the top of the page we have a circular arrow icon, which conventionally means “refresh the page”.
* Click the “refresh page” icon. Visual Studio refreshes the page, and we can observe that the pull request is no longer shown.
* Navigate to the **Synchronization** page. If the current branch is not “master”, switch to “master” by using the double-arrow icon at the right edge of the Visual Studio status bar – near the bottom of the screen.
* Click the **Fetch** link. Visual Studio displays the incoming commits.
* Click the **Pull** link – *the one under* ***Incoming Commits*** - to pull these commits into the local repository.
* Everything seems to be up-to-date. Test the application by running it.

## View and Compare Commits

We resume acting as the 1st team member. We are in Visual Studio, Team Explorer, with the current branch set to **master**.

* Navigate to the **Brancbes** page. Right-click **Master**; Visual Studio displays a pop-up menu. Select **Pull** from this menu – to make sure we have the latest changes.
* Write-click **Master** again, and select **View History**. Visual Studio displays the commits. The most recent 4 of these are at the top of the list; the 3 commits for implementing the coffee shop commands, and the completed pull request. The “Author” of the completed pull-request appears to be incorrect; we know that the other team member completed the pull request. But hover the mouse over this commit; it shows that the **Author** is me, but that the **Committer** is the other team member.
* You can examine the details of a commit. Right-click a commit in the history, e.g. the 2nd-most-recent. Visual Studio displays a pop-up menu. Click **View Commit Details** in this pop-up menu. Visual Studio displays the details in the Team Explorer – on a page titled **Commit Details**. **Commit Details** shows the comment about the commit, and it lists the changed source files – in this case CoffeeShopCommandHandler.cs.
* You can compare two distinct commits. For example, press the <ctrl> keyboard key, and then click the most recent commit, and the commit at v0.5 (“merge feature/paperCupsInStock”). Visual Studio displays an indication that two commits are selected.
* Release the <ctrl> key, and right-click either of the two selected commits; then select **Compare Commits…** from the pop-up menu. Visual Studio displays a page titled **Compare Commits** in the Team Explorer. This page shows various details about the 2 commits – including the list of the source files that are different – at the bottom of the page.
* Double-click any source file in the list of changes. Visual Studio displays a side-by-side comparison in the center pane.
* Revert to the history view of the **master** branch – in the center pane. If you want to view only the commits that pertain to the **master** branch, click the **Show First Parent Only** icon – an icon showing something like a merge with 2 dark nodes and 1 white node. Visual Studio simplifies the view by hiding the commits that pertain to feature/coffeeShopCommands.

## Browse the History of a File

We continue acting as the 1st team member. We are in Visual Studio, Solution Explorer. This section describes tools that can be used to analyze the history of any single source file, for example Program.cs.

* Right-click the source file Program.cs. Visual Studio displays a pop-up menu. Select **Git > View History**. Visual Studio displays a list of commits with regard to this file in the center pane.
* Right click one of the commits in the list of commits. Visual Studio displays a pop-up menu with several useful options.
  + **Open** displays the source of this commit version in a new tab. (This could be used to select and copy code into the clipboard – especially if this code has been overwritten or deleted)
  + **View Commit Details** causes Visual Studio to display the **Commit Details** page of Team Explorer for this particular commit. In particular you are told of other source files that were added/changed at this commit.
  + **Compare with Previous…** displays the selected version and the preceding version side-by-side.
  + **Blame (Annotate)** is like **Open**, but with annotation; for each line of code, the annotation indicates the date when this line was introduced or most recently changed, and it provides the identity of the team member that introduced or changed the line of code. This feature is described in greater detail later in this section.
* Press the <ctrl> key, and select two of the commits in the list of commits. Next release the <ctrl> key, and right-click one of the two selected commits. Visual Studio displays a pop-up menu pertaining to the two selected commits. This menu has two useful compare options.
  + **Compare…** displays the two versions side-by-side.
  + **Compare Commits…** causes Visual Studio to display the **Commit Details** page of Team Explorer for these two commits. This includes the list of all source files that are different or that have been added between the two commits. If you double-click any of these files, Visual Studio will display the file (if it is merely added), or will display the two versions side-by-side.
* As was stated above the **Blame (Annotate)** option can be invoked for any revision level (commit). However, it is more likely that this feature would be used with the most recent revision level. Right-click the source file Program.cs in the Solution Explorer. Visual Studio displays a pop-up menu. Select **Git > Blame (Annotate)**. Visual Studio displays the source code of Program.cs – with annotation in the left margin.
* The annotation consists of the commit ID, the team member that introduced or changed the line of code, and the commit date. Click a commit ID - e.g. the commit ID adjacent to “using WiredBrainCoffee.DataAccess;” - to see the details of the commit. Visual Studio displays these details in the **Commit Details** page in the Team Explorer. The description of this commit is “Implement help command”.
* Note - at the bottom of the **Commit Details** page – the list of all source files that were involved in this commit. The other source file is WiredBrainCoffee.ShopInfoTool.csproj. Double-click this file name. Visual Studio displays the details of the changes to this .csproj file in a side-by-side view. Similarly double-click the other file name - Program.cs . Visual Studio displays the details of its changes – code to implement the “help” command.
* Revert back to the Blame-annotated view of Program.cs. Suppose we want to take backward steps through the history of this file. Right-click the annotation of the line that immediately precedes the “namespace” statement. Visual Studio displays a pop-up menu. Click **Annotate This Version**. Visual Studio displays the version when Program.cs was written for the 1st time.

## Use CodeLens to View Changes

There is another useful tool, Code Lens, that can be used to view the history of source files. Unfortunately in the (free) Community Edition of Visual Studio 2019, Code Lens is limited, and git commits are not included in the Code Lens feature. Since I am working with the Community Edition, I did not have the opportunity to shadow the Pluralsight instructor in using Code Lens to review history.