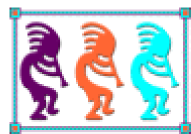


Practical Version Control: Using Git and GitHub in Your Daily Work

Doug Hennig
Stonefield Software Inc.
Email: doug@doughennig.com
Corporate Web sites: stonefieldquery.com
stonefieldsoftware.com
Personal Web site : DougHennig.com
Blog: DougHennig.BlogSpot.com
Bluesky: @doughennig.bsky.social

Git is the most popular version control system and GitHub is the most popular remote repository site. Yet many VFP developers still do not use Git in their day-to-day practice or aren't completely comfortable working with it. This session discusses how to get started with Git and GitHub, what tools I use in my daily work, and what workflow I use, whether as a solo developer or on a team.

Introduction

Version control, also known as source control, is essential for modern development. It's really the only way to work efficiently in a multi-developer environment, but is incredibly useful even for single developer projects. I'll admit I was slow to adopt it, but since I started, I can't imagine working on a project without it. I even use version control for non-development tasks, such as sharing Southwest Fox and Virtual Fox Fest files with Rick Schummer and Tamar Granor. If you haven't implemented version control yet, hopefully this document will convince you to.

You may have heard of, or worked with, older version control systems like Visual Source Safe (VSS). With VSS, there was a shared database containing the repository of files. This shared database often caused performance issues. In addition, you had to check out and check in files. Checked in files were marked as read-only so you couldn't edit them. Checking them out marked them as read-write for you, which allowed you to edit them, but locked others out, which caused concurrency issues.

Git is a distributed version control system (DVCS). The local repository is stored in a folder named .git. There's no checking in or out: you just make changes to files, commit them (sort of like checking in), and deal with conflicts if they occur when merging changes from other repositories. Each developer has their own local repository where they work from.

GitHub is a platform for remote repositories. It provides a common location to store the merged changes from all developers. Developers create a local repository by cloning a remote repository, retrieve changes others make by pulling from it, and share their changes by pushing to it.

Git is command-line based, but in my mind, that's an inefficient way to work. There are graphical tools available, including TortoiseGit and SourceTree, that make it easier to work with.

Getting started

This section provides details on installing Git and several related tools I recommend. Even if you've already installed Git and some or all of these tools, you might find the configuration details discussed here useful.

Install Git

Download and install the Windows version of Git (either 32-bit or 64-bit; since we aren't using Git as a COM object from VFP, the 64-bit version is fine) from https://git-scm.com/downloads/win. You can leave the default settings for installation options except for a few places I recommend:

- If you use GUI tools for Git, you won't likely ever see Git display an editor, but if it does, you don't want the VIM editor that comes with Git because it's like a 70's era UNIX editor. Instead, in the *Choosing the default editor used by Git* step (**Figure 1**), choose Notepad or Notepad++ if you have it.

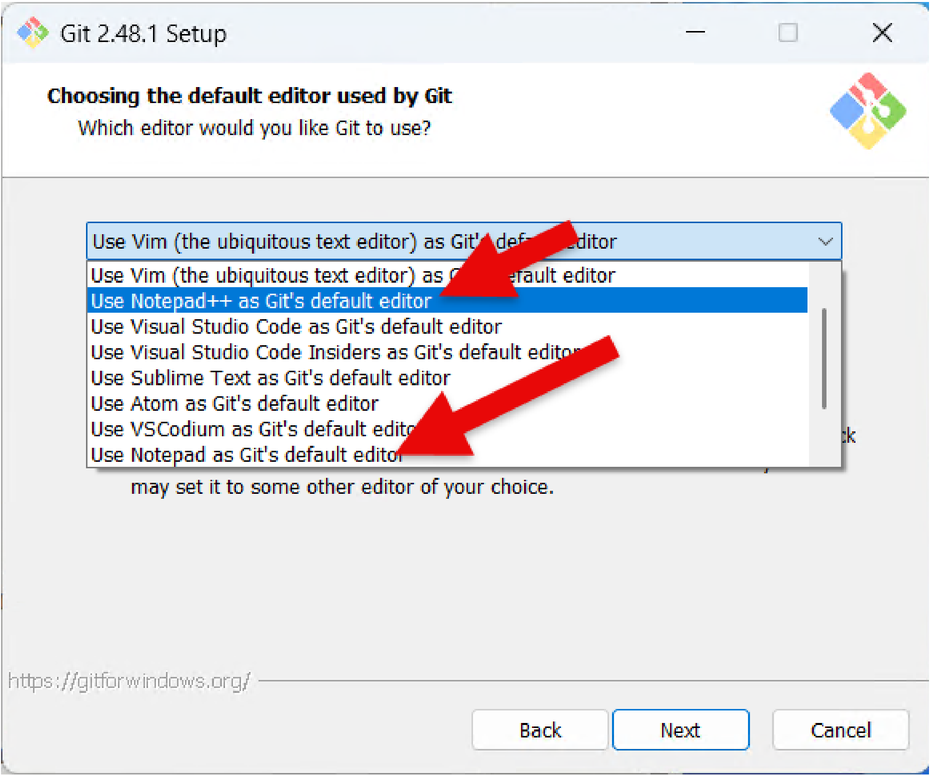


Figure 1. Select with Notepad++ (if you use it) or Notepad as the editor, since the VIM editor that comes with Git is like a 70's era UNIX editor.

- Modern repositories use "main" as the main branch name rather than the traditional "master" so choose that setting in the *Adjusting the name of the initial branch in new repositories* step (**Figure 2**).

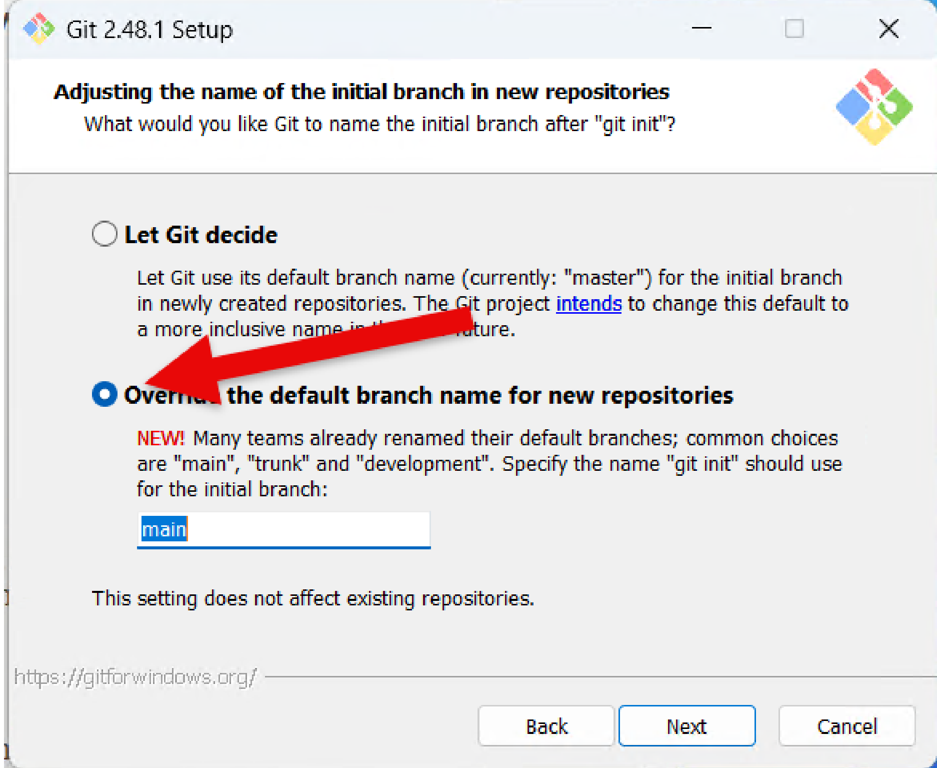


Figure 2. Modern repositories use "main" as the main branch name rather than the traditional "master".

- We don't want Git altering line endings when text files are committed, so choose the *Checkout as-is, commit as-is* setting in the *Configure the line ending conversions* step (**Figure 3**).

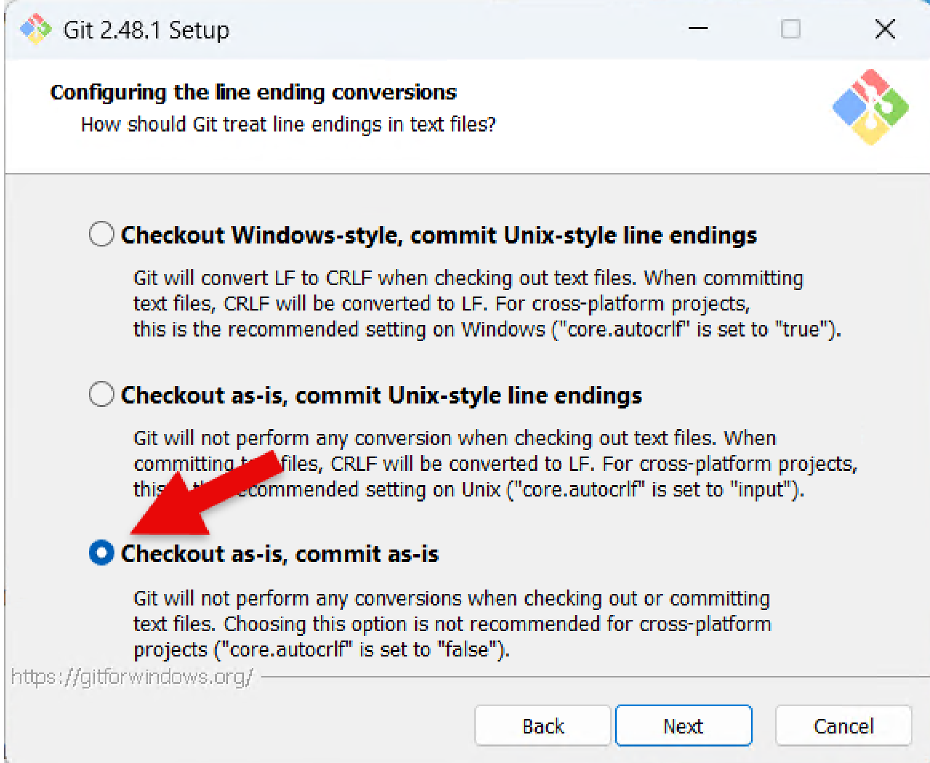


Figure 3. We don't want Git altering line endings when text files are committed.

- You may also wish to use the Windows Console rather than the default MinTTY console when using Git Bash (**Figure 4**). (I don't discuss Git Bash in this document as I'm focusing on GUI rather than command-line tools.)

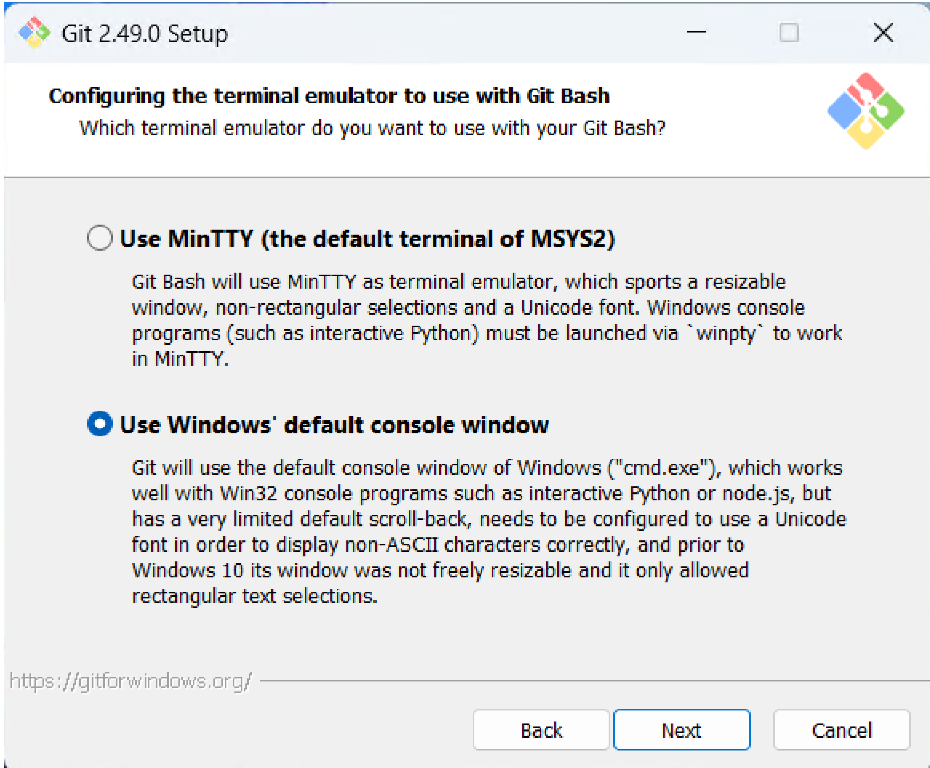


Figure 4. You may wish to use the Windows Console for Git Bash.

By default, Git should be case-insensitive for filenames on Windows systems but you can ensure that's the case by typing the following in a command window:

```
git config --global core.ignorecase true
```

Install TortoiseGit

TortoiseGit provides several features I use a lot, including Git-specific items in the File Explorer shortcut menu and icon overlays showing the status of files (**Figure 5**). It also provides GUI dialogs for things like commit, pull, and push so you don't have to use command line statements.

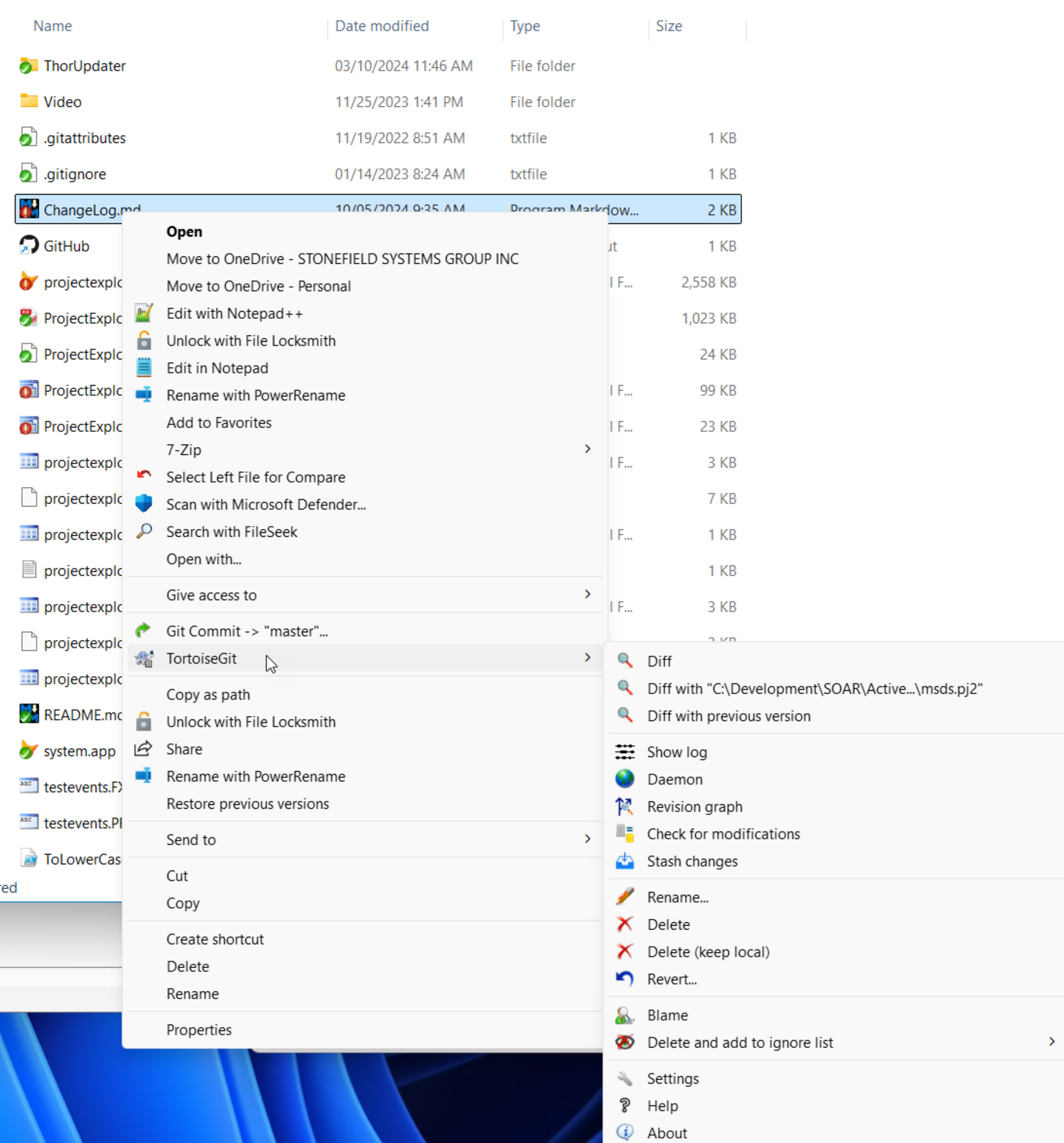


Figure 5. TortoiseGit provides Git-specific items and file overlays in the File Explorer shortcut menu.

Download and install the Windows version of TortoiseGit (choose the same bit version as Git) from https://tortoisegit.org/download. Again, you can leave the default settings but leave *Run First Start Wizard* turned on at the end. In the First Start Wizard that opens after installation is complete, leave the default settings but fill in the *Configure User Information* settings (**Figure 6**): this information is used to identify who made changes.

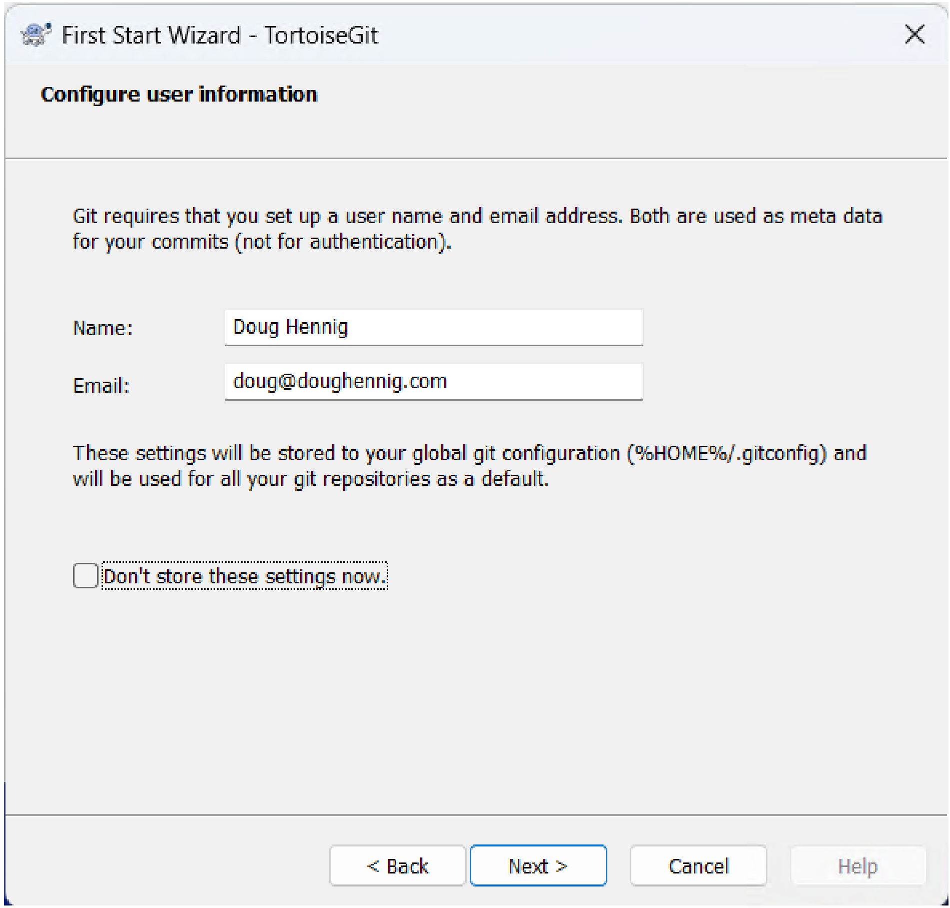


Figure 6. The Configure User Information settings are used to identify who made changes.

After installation, you'll want to configure how TortoiseGit uses icon overlays to display the status of files. In File Explorer, right-click any folder, choose *TortoiseGit*, *Settings*, and select Icon Overlays in the Settings dialog (**Figure 7**). Select ShellExtended. (Click Help to see information about the four choices.)

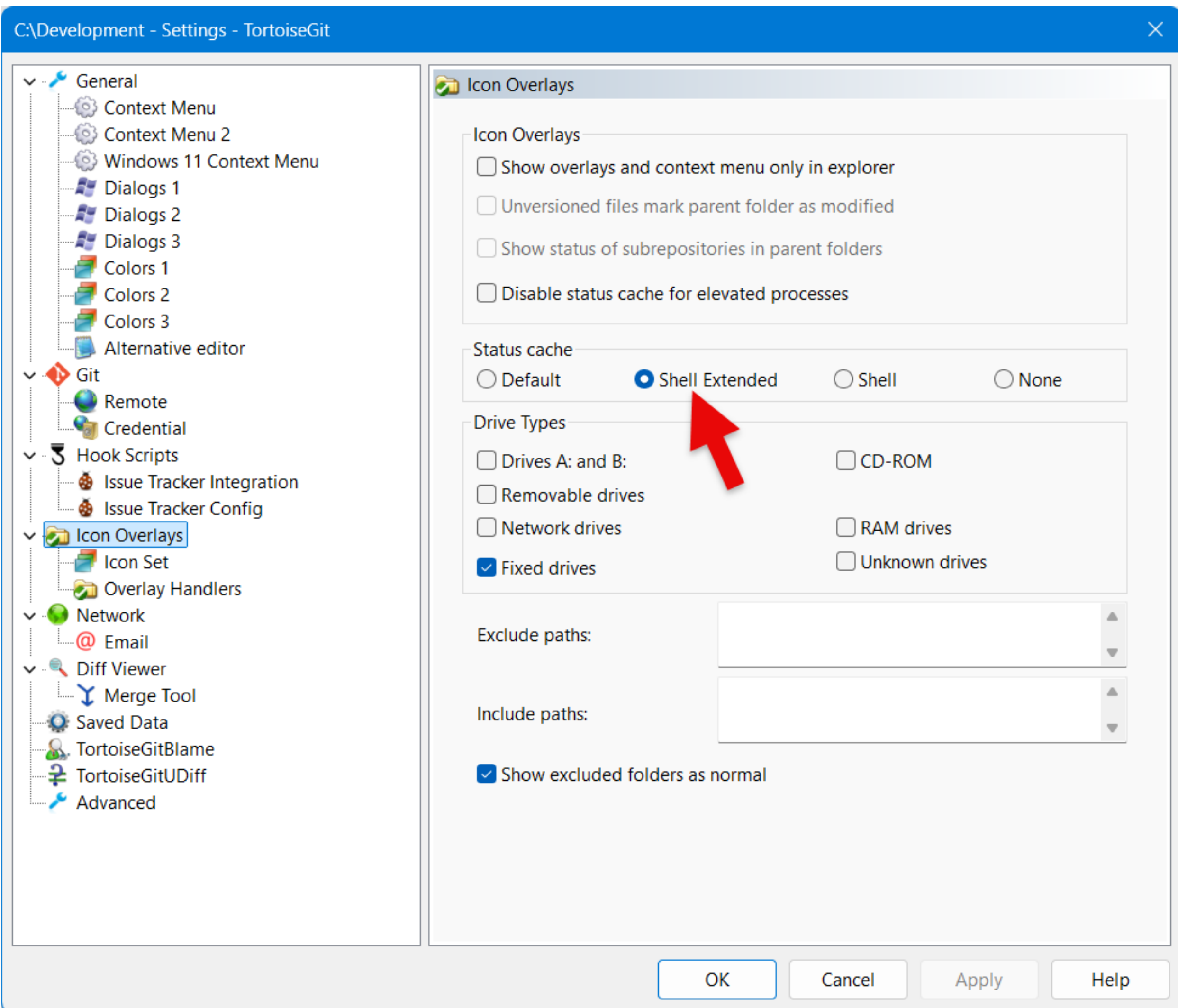


Figure 7. Choose ShellExtended for the Status cache setting.

Even if you made Git case-insensitive, by default TortoiseGit is still case-sensitive when it comes to icon overlays. Choose Advanced, find the OverlaysCaseSensitive setting, and change its value to false **(Figure 8).**

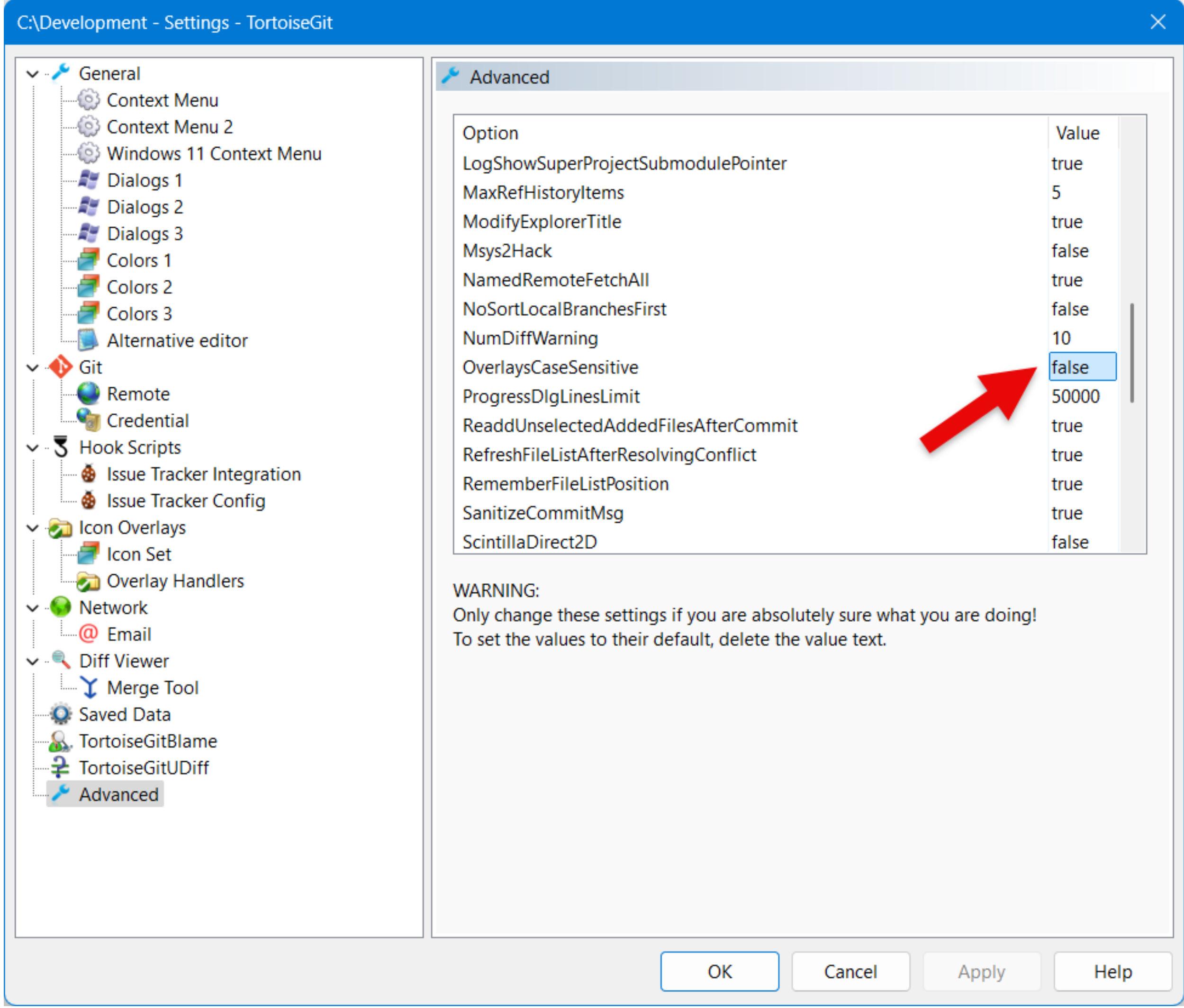


Figure 8. Make TortoiseGit case-insensitive for icon overlays.

If you wish, you can select which set of icons are used for icon overlays. Select Icon Set in the Settings dialog and choose the desired set from the dropdown list (**Figure 9**).

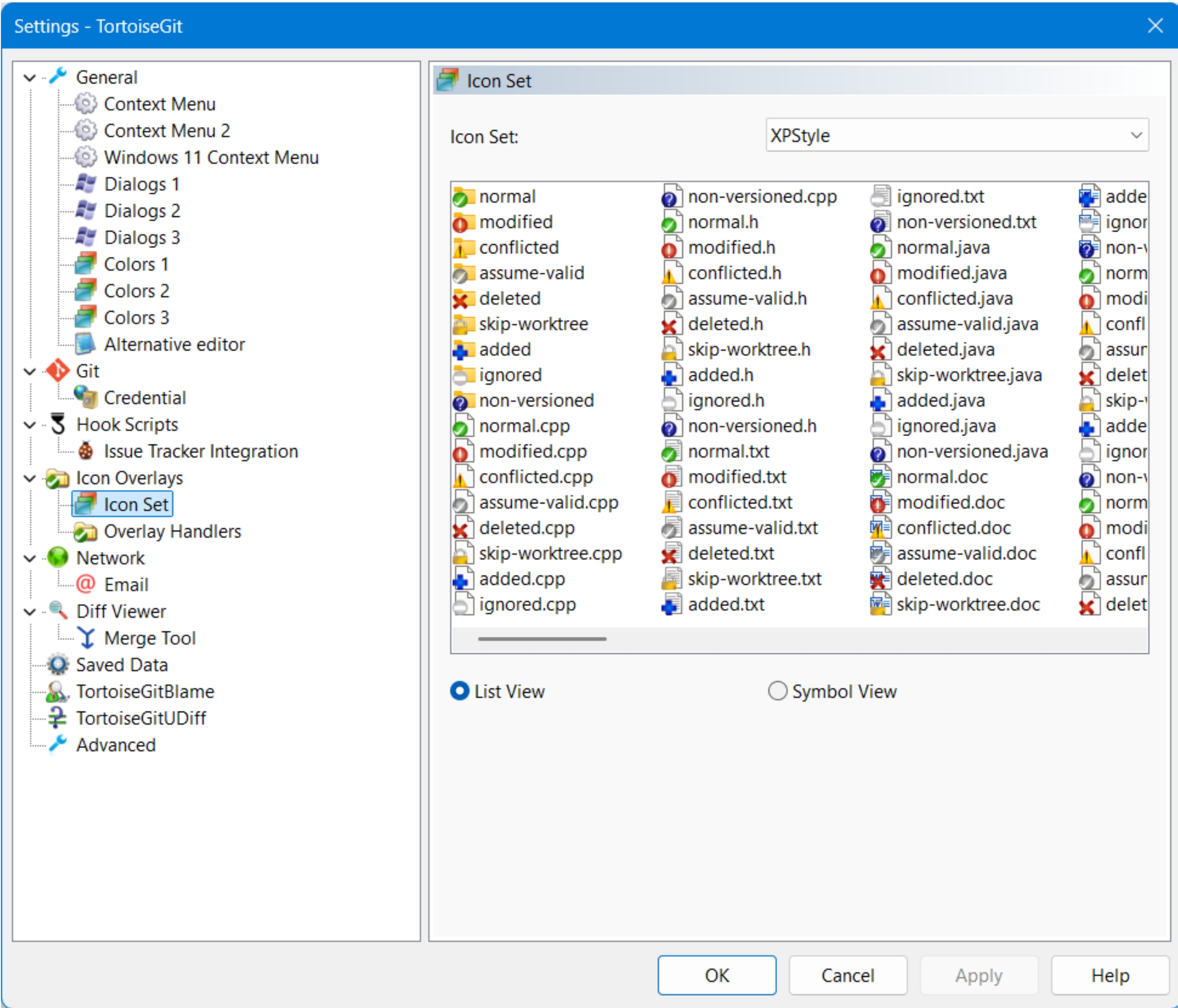


Figure 9. You can select which icon set to use for icon overlays.

Not all icon overlays may appear in File Explorer. For example, on my system, unversioned files (files that aren't included in a repository) didn't show any icon overlay rather than showing as a blue question mark. The problem is that, as discussed at https://answers.microsoft.com/en-us/windows/forum/all/onedrive-steals-icon-overlays/50a41e06-6daa-4006-af24-7abc8a72e1a4, Windows supports a limited number of overlay "slots" and some other applications, including OneDrive and Dropbox, may use some of the slots. In my case, it was OneDrive stealing some of the slots TortoiseGit uses. Since I don't care about OneDrive icon overlays, I selected Overlay Handlers in the Settings dialog (**Figure 10**), clicked *Start registry editor*, and deleted the OneDrive settings in the Registry Editor. Note that you must restart your system for these changes to take effect. Although it hasn't happened to me (yet), some users report that OneDrive may recreate the Registry entries, forcing you to delete them again.

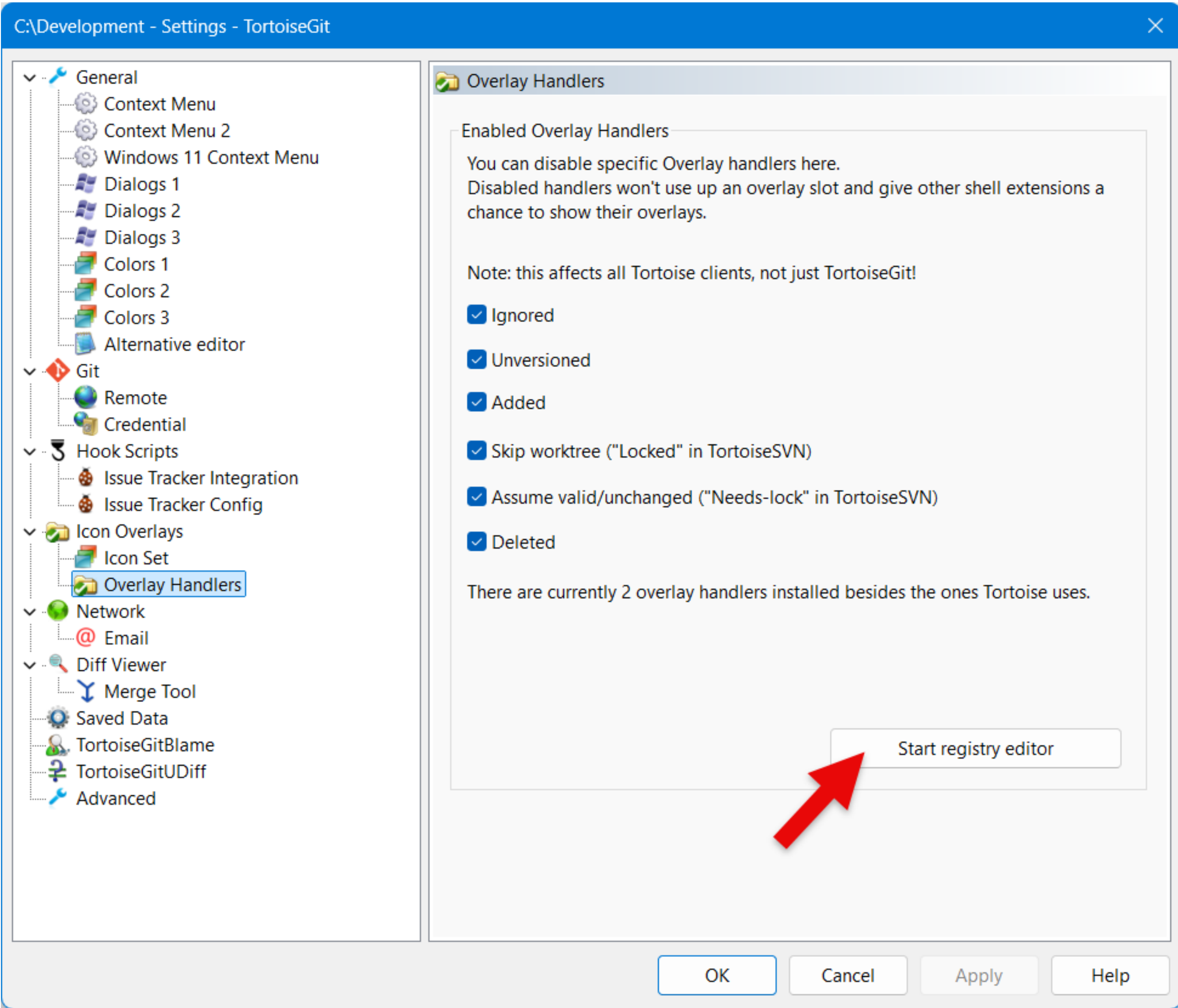


Figure 10. Use the Registry Editor to delete unwanted icon overlay handlers such as OneDrive.

You can also turn off certain icon overlay types if you wish in the Enable Overlay Handlers section.

Install Beyond Compare

This is optional, but Beyond Compare is much better and easier to use than the diff tool that comes with Git. It also allows you to compare files and folder, a feature I use all the time. It's also inexpensive: $35 per user for the Standard Edition and $70 for the Pro Edition as of this writing. I highly recommend it. You can purchase and download it from https://www.scootersoftware.com.

To configure it for use with Git, open a command window and execute the following commands (substitute "C:/Program Files/Beyond Compare 5" for the folder where it was installed; note the forward slash rather than a backslash):

```
git config --global diff.tool bc
```

```
git config --global difftool.bc.path "C:/Program Files/Beyond Compare 5/bcomp.exe"
git config --global merge.tool bc
git config --global mergetool.bc.path "C:/Program Files/Beyond Compare 5/bcomp.exe"
git config --global difftool.prompt false
git config --global mergetool.keepBackup false
```

To configure it for use with TortoiseGit, do the following:

- Right-click a folder and choose *TortoiseGit*, *Settings*
- Select Diff Viewer in the panel at the left; see **Figure 11**
- Change the radio buttons from TortoiseGitMerge to External
- In the textbox below that, enter:

  ```
  "C:\Program Files\Beyond Compare 5\BComp.exe" %base %mine /title1=%bname
  /title2=%yname /leftreadonly
  ```

- Select Merge Tool in the panel at the left
- Change the radio buttons from TortoiseGitMerge to External
- In the textbox below that, enter:

  ```
  "C:\Program Files\Beyond Compare 5\BComp.exe" %mine %theirs %base %merged
  /title1=%yname /title2=%tname /title3=%bname /title4=%mname
  ```

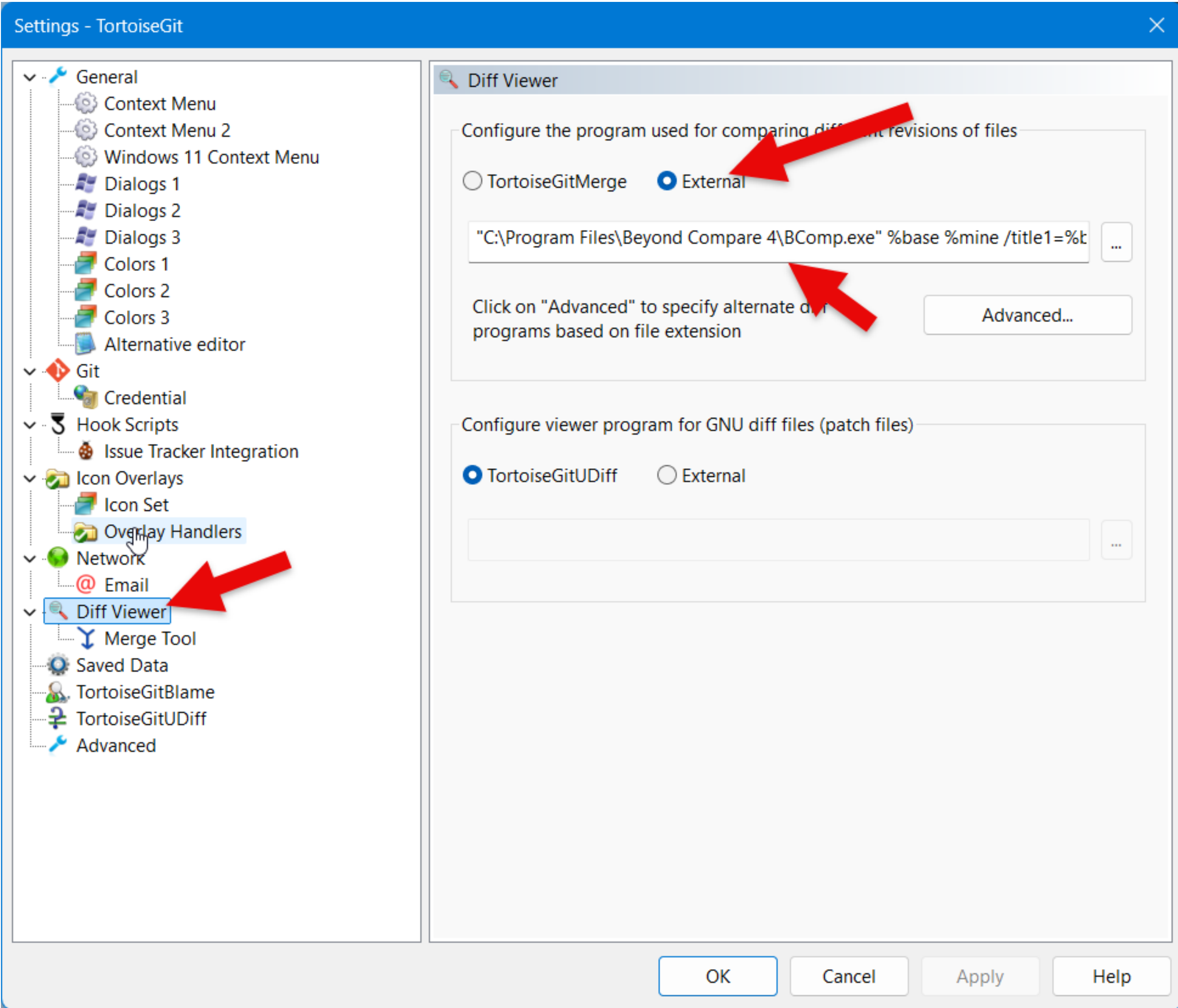


Figure 11. Edit the TortoiseGit Diff Viewer settings to use Beyond Compare.

Frank Perez created a handy add-on for Beyond Compare that knows how to handle VFP binary files so you can compare them directly without having to convert to text equivalents (which is discussed later). You can download it from http://pfsolutions-mi.com/Product/VFP2Text.

Install SourceTree

SourceTree (**Figure 12**) is a free GUI tool for Git. Although TortoiseGit is handy for simple common tasks like commits thanks to its File Explorer integration, SourceTree has a much better UI, especially for more complex tasks such as reviewing changesets. You can download SourceTree from https://www.sourcetreeapp.com.

Copyright 2025, Doug Hennig

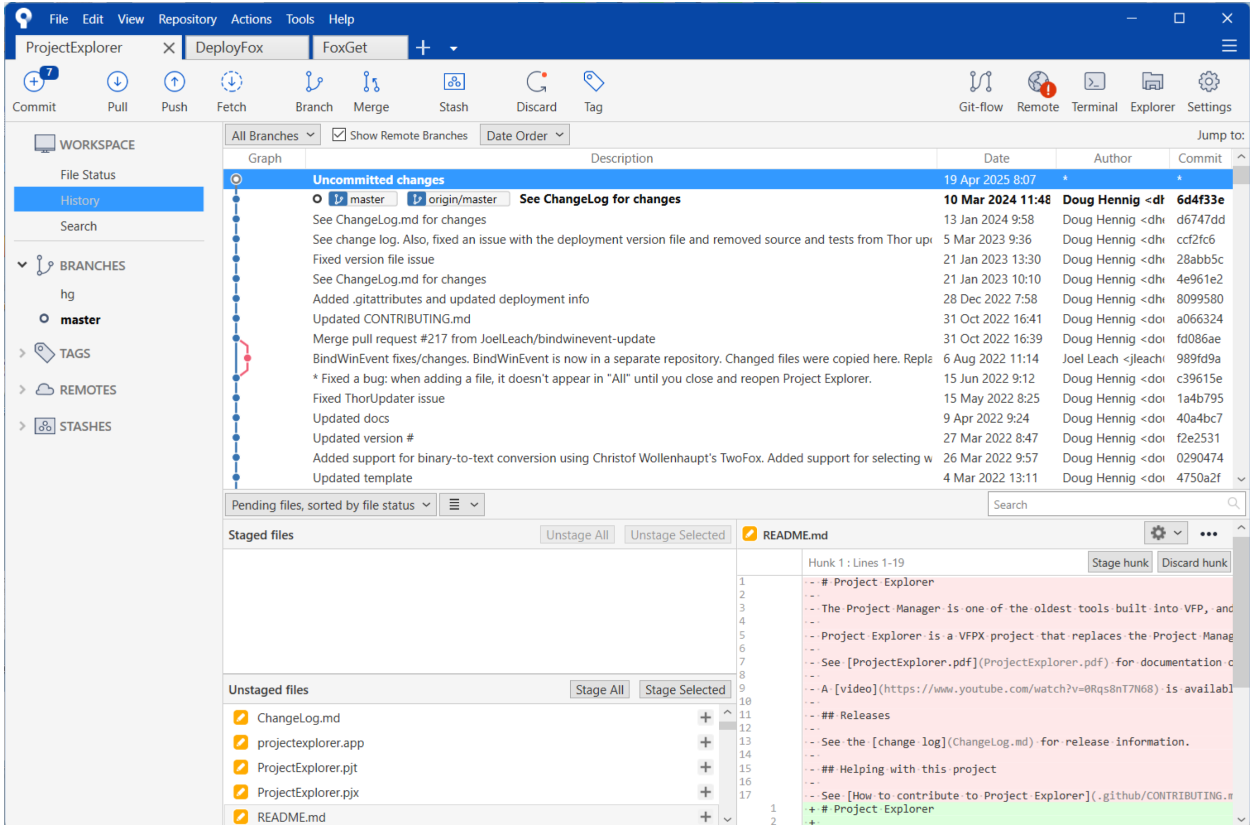


Figure 12. SourceTree is a free GUI tool for Git.

Install Thor

Git relies on comparing files so it can determine what changes were made to them. However, it cannot compare binary files and a lot of VFP source code is in binary files like SCX, VCX, and FRX files. What we need is the text equivalent of the binary files, which Git can compare, so we need a tool to create those equivalents. Also, as we'll see later, sometimes you need to go the other way: recreate a binary file from its text equivalent.

VFP comes with such a tool named SCCText, but it isn't very good. Fortunately, there are several community-provided replacements, the most popular of which is FoxBin2PRG by Fernando Bozzo. It generates text files with the last character of the file extension changed to "2" (for example, SC2, VC2, and FR2), and can regenerate the binary file from the text equivalent.

Before installing FoxBin2PRG, I recommend installing Thor if you haven't already. Thor is a tool manager that makes installing and running VFP utilities and tools much easier. It's a VFPX tool available at https://github.com/VFPX/Thor. Installation instructions are at https://github.com/VFPX/Thor/blob/master/Docs/Thor_install.md. It's important to follow all the instructions closely.

Copyright 2025, Doug Hennig

Install FoxBin2PRG

After installing Thor, run VFP, choose *Check for Updates* from the Thor menu, turn on Thor Repository, PEM Editor, Dynamic Forms (as the instructions mentioned in the previous step note, these are required), and FoxBin2PRG. If you haven't already done so, you may as well also select GoFish, a much better search utility than the Code References tool that comes with VFP. Click Install to install these tools.

After installation, FoxBin2PRG is available from the Thor Tools, Applications menu in VFP. It can also be automated with a batch file or PowerShell script. If you want to use FoxBin2PRG from the File Explorer shortcut menu, which I do a lot, see the *Use with MS Windows SendTo* section of https://github.com/fdbozzo/foxbin2prg for instructions.

You may wish to tweak how FoxBin2PRG works. To do so, edit FoxBin2prg.cfg in the Thor\Tools\Components\FoxBin2Prg subdirectory of where Thor is installed. My config file has the following changes from the defaults (note that you have to remove the leading "*" on lines you want to change):

- ShowProgressBar: 2 so it only displays when processing multiple files at a time.
- ExtraBackupLevels: 0 so no backups are made.
- HomeDir: 0 so the home directory for a project isn't saved in PJ2 files.
- DBF_Conversion_Support: 8 so it includes the data, not just the structure, in DB2 files.
- BodyDevInfo: 2 to include the BodyDev column in PJ2 files. This is only necessary if you use Project Explorer, which uses BodyDev to store metadata.

Install Markdown Monster

Markdown (.md files) is the standard format for readable files on GitHub. Markdown is similar to HTML but much simpler syntax. For example, it uses characters like # to indicate a heading rather than an <h> tag. I recommend using Markdown for project documentation. In addition to being smaller files than Microsoft Word DOCX and PDF files, they have the benefit of being text files so they can be compared, changes merged, etc.

Although Markdown files are just text files and you can edit them using Notepad, I recommend using Markdown Monster from Rick Strahl. It displays both the text and rendered content, has a built-in table editor, and many functions that make editing Markdown much easier. Purchase ($79 for s single user as of this writing), download, and install it from https://markdownmonster.west-wind.com/. Note that this is only needed for any team member that will edit Markdown files.

Creating a local repository

"What Is a Git Repository? A Beginner's Guide to Understanding Git" (https://www.getint.io/blog/what-is-a-git-repository) says "A Git Repository is the heart of any project managed with Git. It's where all the files, their version history, and metadata about changes are stored. Think of it as a dedicated container for your project, tracking

every modification so you can collaborate, experiment, and recover old versions effortlessly."

There are two types of repositories:

- A local repository exists on each developer's computer.
- A remote repository exists on a server or cloud-based platform like GitHub. It acts like the central location of the source code.

Syncing changes between repositories involves pulling and pushing. Pulling is retrieving changes from the remote repository to the local and pushing is sending changes from the local to the remote.

A repository can only contain files in a certain folder and below, so if your source code spans multiple folders, you have to decide what repository structure to use.

For example, take the folder structure shown in **Figure 13**. App1 and App2 contain the source code for two applications and Common contains files used in all applications (libraries, utilities, commercial tools, VFPX projects, etc.). Should you put the repository under Development, meaning there's one repository for all applications? That might seem cumbersome but the alternative is to create separate repositories for App1, App2, and Common, and know that the App1 and App2 repositories don't contain all the source code for the applications. It also means that when you refresh your local repository with changes made by someone else, you have to pull from at least two repositories (Common and whatever app you're working on).

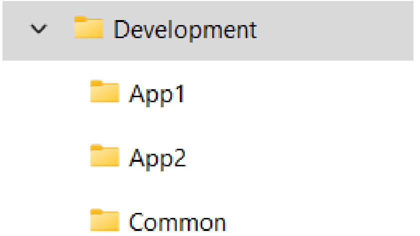


Figure 13. You have to decide where to put the repository when your source code spans multiple folders.

An alternate development structure is to not have common files in a separate folder but in subdirectories of each application folder. That means the common files are duplicated, but disk space is cheap, it prevents versioning issues (breaking some applications when you upgrade to the latest version of a library), and it allows you to have separate repositories for each application if desired.

In this case, we're going to create the repository in Development so it includes App1, App2, and Common. In File Explorer, right-click Development, choose *Git Create repository here*, and click OK in the dialog that appears. Notice a new folder named .git appears. You can take a peek at the contents if you wish, but don't touch anything. It's best to consider it to be a black box and leave it alone.

Now it's time to add files to the repository.

What files belong in the repository

Right-click Development and choose *TortoiseGit*, *Add*. Notice that all files in all subdirectories appear in the dialog that appears. Do they all need to go into the repository?

Here's what belongs and doesn't belong:

- Only include what's needed for another developer. Fo example, exclude ZIP, TBK, BAK, log files, and so on.
- Exclude generated files such as FXP, MPR, and MPX.
- Exclude data (DBF, FPT, CDX, DBC, DCX, and DCT) unless they're part of the source code such as metadata.
- Generate and include the text equivalents of VFP binary files since binary files can't be compared. We'll discuss doing that using FoxBin2PRG.
- Documentation such as Markdown files.
- Anything else considered to be source code or integral to the application.

Should VFP binary files be included? Some developers say yes, others say no. Theoretically, you only need the text equivalents since FoxBin2PRG can generate the binary files from them. However, what if FoxBin2PRG fails to correctly generate a binary? I haven't seen that happen but are you willing to trust that it'll work 100% of the time?

The downside of including binaries in the repository is that VFP changes the Timestamp field in the binaries even when nothing has changed, such as when building the project with *Recompile all files* turned on. FoxBin2PRG won't generate a new text equivalent when only Timestamp is different, so the only text equivalents that have changed are the ones where the source code was changed. However, Git still sees the binaries as changed. It's just something you have to live with.

There are several ways you can generate text equivalents:

- If you use Project Explorer, a replacement for the VFP Project Manager available at https://github.com/DougHennig/ProjectExplorer, it can be configured to automatically call FoxBin2PRG to generate the text equivalent after you're done editing the file.
- If you set up File Explorer Send To items, you can right-click one or multiple binary files and choose *Send To*, *FoxBin2Prg – Binary to Text*.
- With the project open in VFP, choose Thor Tools, Applications, FoxBin2Prg, Projects, Convert all binary files to text files (I assigned Alt-1 as the hotkey for that item in Thor so it's a quick keypress)

- Create a VFP program, batch file, or PowerShell script to call FoxBin2PRG.exe, passing the necessary parameters to generate text equivalents for one or more projects, folders, or whatever you wish.

VFP sometimes changes the case of binary files when you save changes.

- If you MODIFY FORM test, VFP uppercases the extension of the SCX and SCT files when you save.
- If you MODIFY FORM test.scx, it uses that extension for the SCX but uppercases the SCT.
- If you MODIFY FORM Test, it uses that name for the SCX and SCT.

These weird casing rules can cause issues with Git because it was originally created for Linux, where file name case is significant. As I discussed in the sections on installing Git and TortoiseGit, you can configure them to not be case-sensitive. Note that FoxBix2PRG automatically changes file extensions to lower case when it creates the text equivalents of binary files.

For each folder you want included in the repository:

- Right-click the folder and choose *TortoiseGit*, *Add*
- Check the files you want added (omit FXP, BAK, TBK, MPR, MPX, etc.)
- Click OK

Once you've added all the files, right-click the parent folder and choose *Git Commit -> Main*, enter a commit message (the traditional message for a first commit is "Initial Commit"), and click Commit.

Create text equivalents for the VFP binary files using whatever mechanism you wish, then right-click the folder, choose *TortoiseGit*, *Add*, and click OK to add them to the repository. Then commit the additions.

Notice the Commit dialog has a section titled Not Versioned Files displaying the FXP files. You probably don't want to see those files, neither here nor in other TortoiseGit dialogs. We can tell Git to ignore those files by right-clicking one of them, choosing *TortoiseGit*, *Add to ignore list*, **.FXP*. In the dialog that appears (**Figure 14**), choose *Ignore items(s) recursively* and click OK.

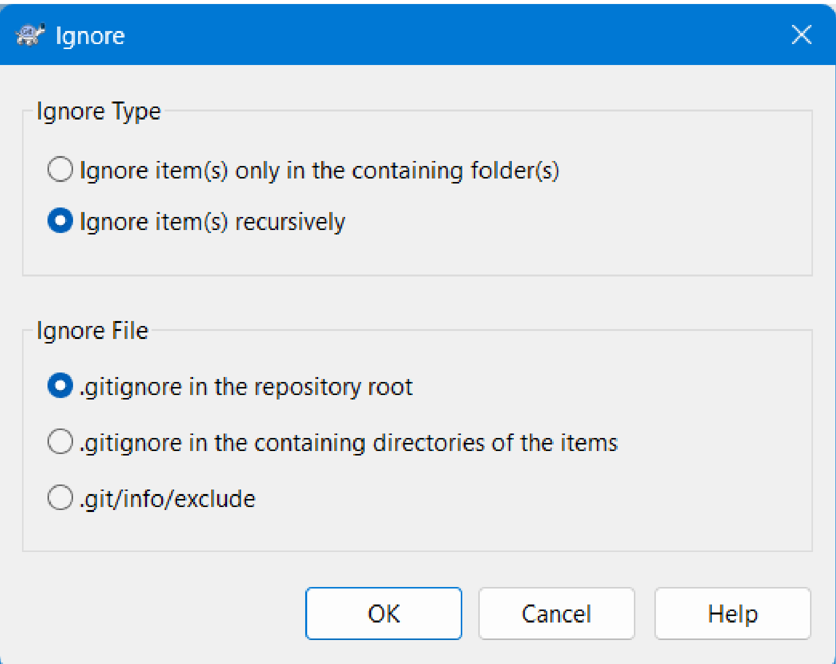


Figure 14. You can tell Git to ignore certain files and file types.

This creates a text file named .gitignore in the repository root. You can edit that file with Notepad or any other text editor and manually add additional files or file types, such as TXT, ZIP, BAK, and so on.

Christof Wollenhaupt has some example .gitignore files at https://github.com/cwollenhaupt/foxGitIgnore. Christof's team .gitignore file excludes VFP binary files from the repository. The sample files accompanying this document includes a modified version that removes those exclusions.

Be sure to add .gitignore to the repository.

Creating a remote repository

Now it's time to create the remote repository so you can share the source code with other developers.

- Navigate your browser to https://GitHub.com and log in (create an account if you haven't already).
- Create a new repository by clicking New in the panel at the left (**Figure 15**).

Copyright 2025, Doug Hennig

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere?
Import a repository.

Required fields are marked with an asterisk (*).

Owner *
DougHennig
Repository name *

Great repository names are short and memorable. Need inspiration? How about upgraded-doodle ?

Description (optional)

Public
Anyone on the internet can see this repository. You choose who can commit.

Private
You choose who can see and commit to this repository.

Initialize this repository with:

Add a README file
This is where you can write a long description for your project. Learn more about READMEs.

Add .gitignore
.gitignore template: None
Choose which files not to track from a list of templates. Learn more about ignoring files.

Choose a license
License: None
A license tells others what they can and can't do with your code. Learn more about licenses.

You are creating a public repository in your personal account.

Create repository

Figure 15. Create a remote repository at GitHub.com.

- Specify a repository name and make it Private (unless you want to share it with the world).
- Click Settings, Collaborators, Add People, enter the GitHub username for one of your team members, and click Add to Repository.

- Click Code and click the Copy button in the Quick Setup section to copy the URL for the remote repository to your clipboard (**Figure 16**).

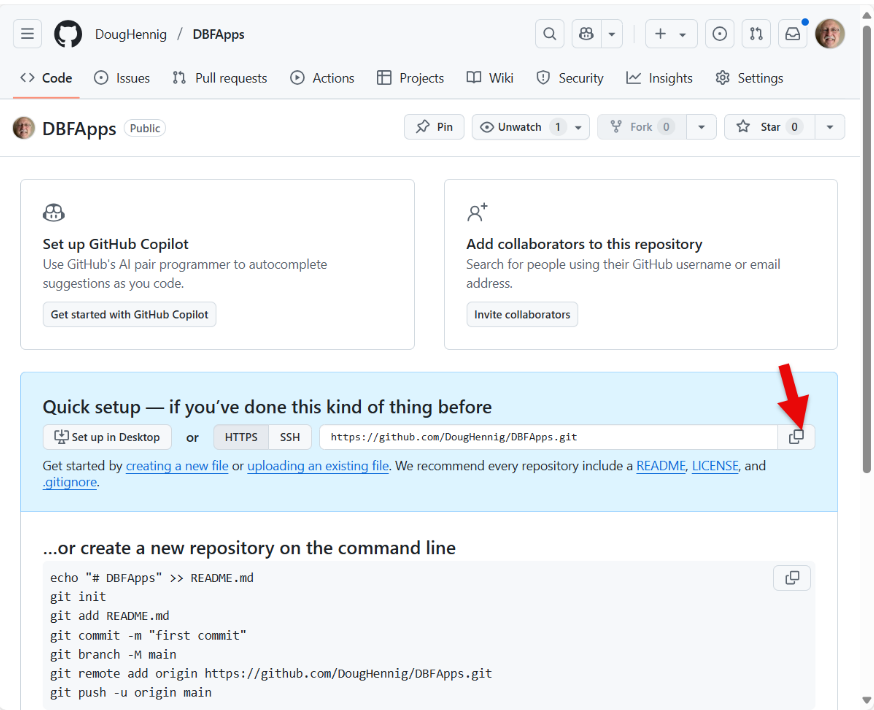
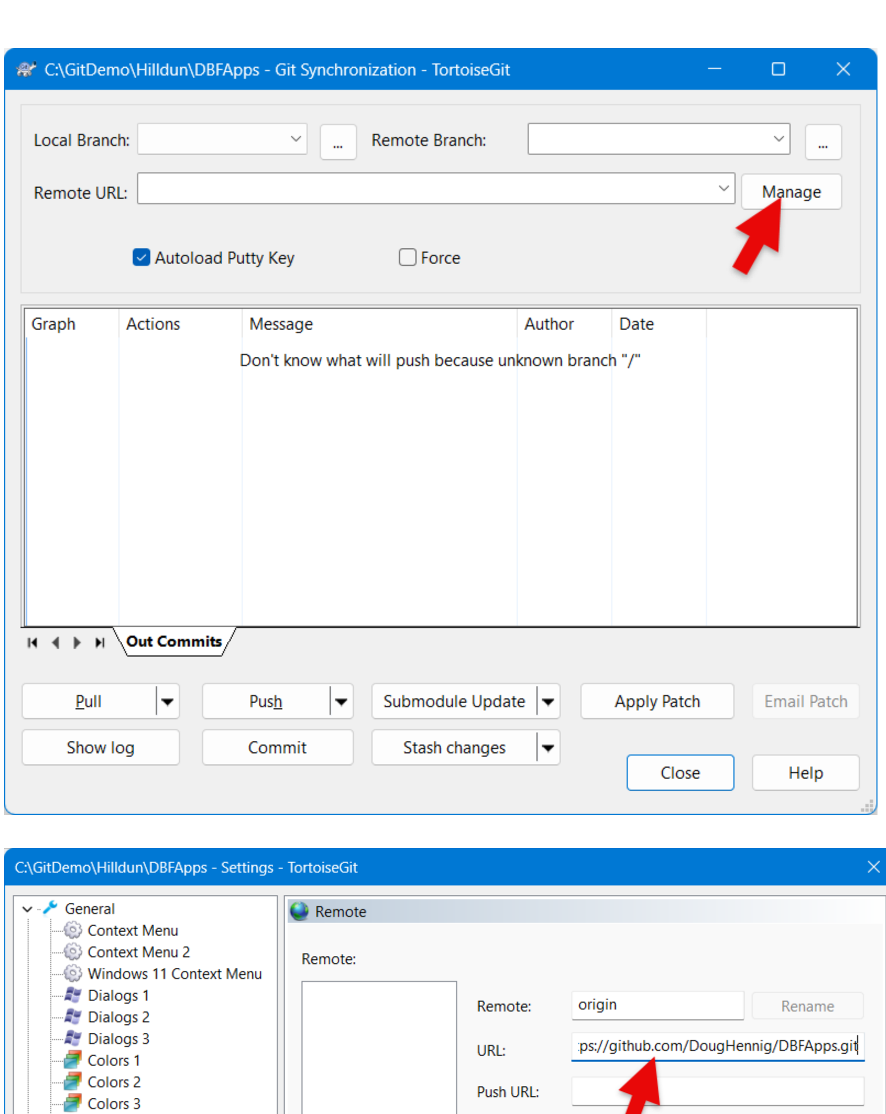


Figure 16. Copy the URL for the remote repository.

- Right-click the parent folder for the local repository and choose *Git Sync*. Click Manage, paste into the URL textbox, click OK, and click Push to push the local repository to the remote one. (**Figure 17**).



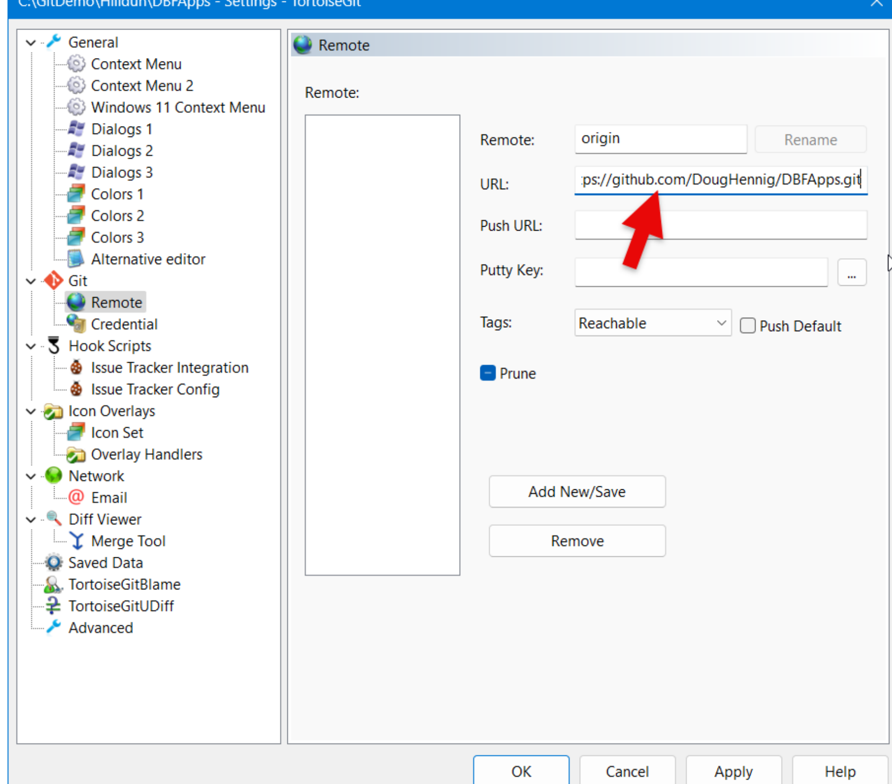


Figure 17. Paste the URL into the URL textbox.

- Refresh your browser; the source code should appear.
- Create a new text file named README.md in the parent folder for the local repository. Edit it and add whatever text you wish, such as a description of the project, who's working on it, and so on.
- Right-click README.md and choose *TortoiseGit*, *Add*. Click OK in the dialog that appears.
- Right-click the folder, choose *Git Commit -> Main*, enter a comment, and click Commit.
- Right-click the folder, choose *Git Sync*, click Pull (nothing has changed so nothing will be pulled, but get in the habit of always pulling before pushing), then click Push.
- Refresh your browser; the content of README.md displays.

Clone the remote repository

Copy URL for the remote repository from GitHub. On another machine (or to simulate it, in another folder on your machine), right-click the parent folder where you want the local repository to go, choose *Git Clone*, notice the URL is already filled in, and click OK.

Simple workflow

Let's do some steps that show a typical, simple workflow for making changes to the source code. In these steps, I'll refer to Machine1 as the machine/folder where the first local repository was created and Machine2 as the machine or folder when the remote repository was cloned.

- On Machine1, open App1\app1.pjx, open Test.scx, add a CommandButton, and save.
- Generate the text equivalent for Test.scx; for example, right-click it in File Explorer and choose *Send to*, *FoxBin2Prg – Binary to Text*.
- Right-click the folder, choose *Git Commit -> Main*, enter a comment, and choose Commit. Note the Commit button has other choices in a dropdown menu, including Commit & Push, but we'll only use Commit because we'll always pull before pushing.
- Right-click the folder, choose *Git Sync*, click Pull (nothing has changed yet), then click Push.
- On Machine2, open App1\app1.pjx. Notice that unless the directory structures are identical on the two machines, you'll get the message about changing the home directory; choose Yes. Edit Main.prg, make some change, and save it.
- Right-click the folder, choose *Git Commit -> Main*, enter a comment, and click Commit.
- Right-click the folder, choose *Git Sync*, click Pull, and notice that changes appear in the In Changelist tab. Right-click Test.sc2 and choose *Compare Revisions* and you'll see that a CommandButton was added. Click Push.

- Open App1\app1.pjx, open Test.scx, and note the CommandButton added on Machine1 exists. Add code to its Click event and save. Generate the text equivalent, commit, pull, and push.
- On Machine1, open Test.scx, add a second button, and save. Generate the text equivalent, commit, and pull. Notice you now have a conflict because different changes were made to Test.scx. However, there is no conflict for Test.sc2. That means the changes made weren't incompatible and Git automatically merged them together in the SC2 file (it couldn't do that for the binary files; hence the conflict). Select both Test.scx and sct, right-click, and choose Resolved. All this does is tell Git that we've handle the conflict and everything is fine, although it isn't really yet.
- Right-click Test.sc2 and choose Send To, FoxBin2Prg – Text to Binary to regenerate the form. Open the form and notice that the code in Command1.Click comes from Machine2 and the new button from Machine1. Exit VFP, commit, pull, and push.
- On Machine2, pull to sync with the changes from Machine1.
- Still on Machine2. open App1\app1.pjx, open Test.scx, and note Command2 added on Machine1 exists. Add code to its Click event and save. Generate the text equivalent, commit, pull, and push.
- On Machine1, open Test.scx, add different code to the Click event of Command2, and save. Generate the text equivalent, commit, and pull. Notice you now have a conflict in all three files (Test.scx, sct, and sc2) because incompatible changes were made. Right-click Test.sc2 and choose Edit Conflicts. Beyond Compare opens (**Figure 18**) and shows four panels: Machine1's change (left panel, labeled HEAD), the original code before either change was made (middle panel, labeled Base), Machine2's change (right panel, labeled MERGE_HEAD), and the resolved code (bottom panel).

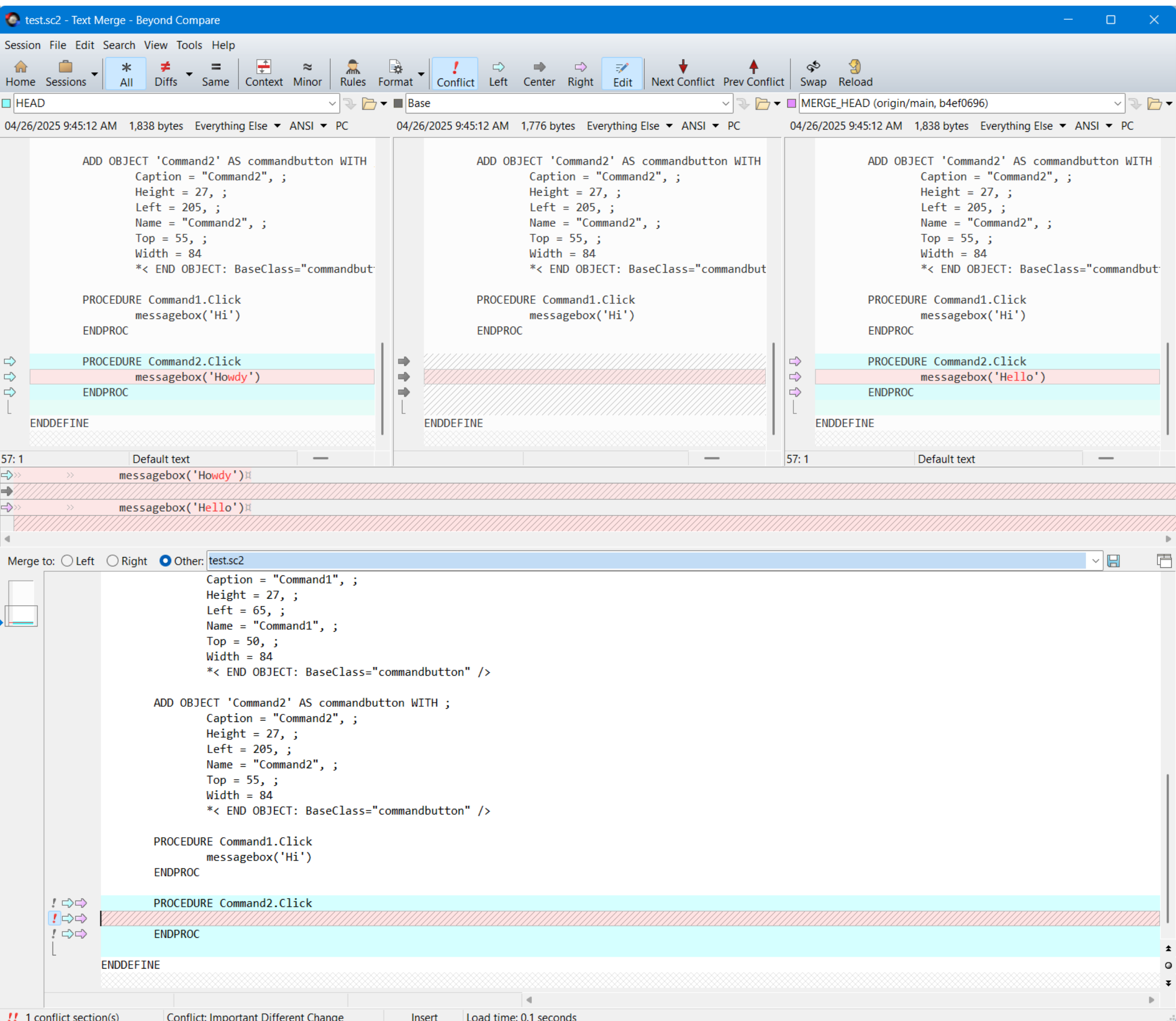


Figure 18. Beyond Compare shows the conflict between the changes.

- Since the changes are incompatible, we have to decide whose code to keep. In this case, let's take Machine1's. Click the Left button in the toolbar and notice the change appears in the merged code in the bottom panel. Close the window and choose Yes when asked to save. Right-click Test.sc2 in the Git Synchronization window and choose Resolved.
- For Test.scx, we can't merge the changes, but that's OK since we regenerate it from the updated SC2 file. So, select both the SCX and SCT files, right-click, and choose Resolved, then close the window.
- Right-click Test.sc2 and choose Send To, FoxBin2Prg – Text to Binary to regenerate the form. Commit (remove the lines starting with "#" from the automatically generated comment), pull, and push.
- Pull on Machine2. Right-click Test.sc2, choose *Tortoise Git*, *Diff with Previous Version*, and notice the differences.

Branching

"Branching Strategies in Git" (https://www.geeksforgeeks.org/branching-strategies-in-git) says "Branches are independent lines of work, stemming from the original codebase. Developers create separate branches for independently working on features so that changes from other developers don't interfere with an individual's line of work. Developers can easily pull changes from different branches and also merge their code with the main branch. This allows easier collaboration for developers working on one codebase."

I won't describe branching strategies here, as they're covered in detail in that article and many others, including Rick Borup's excellent "Team Git: Collaboration Made Easy" presentation at Southwest Fox 2023. We'll just look at the workflow involved. First, let's have Machine1 and Machine2 create new branches and commit changes to them.

- On Machine1, right-click the Development folder and choose Tortoise Git, Create Branch. Enter Feature1 for the branch name, turn on Switch to New Branch, and click OK. Note: if you forget to do this step before you start making changes you want to commit on a new branch, don't worry: when you commit, you can choose New Branch and enter the new branch name.
- On Machine2, do the same but use Feature2 for the branch name.
- On Machine1, open App1\app1.pjx, open Test.scx, add another button, save, and quit.
- Generate the text equivalent for Test.scx, commit (notice the menu item appears as Git Commit -> "Feature1"), and push (no need for a pull since the branch was just created).
- On GitHub, you'll see there are now two branches: main and Feature1.
- On Machine2, create a new program named Test.prg, quit, right-click it, and choose Tortoise Git, Add.
- Generate the text equivalent for App1.pjx, commit (notice the menu item appears as Git Commit -> "Feature2"), and push (no need for a pull since the branch was just created).
- On GitHub, you'll see there are now three branches.

Now let's merge the changes into main, which is the branch we'll release a new version of the application from.

- Because you always merge from another branch into the current one, on Machine1, right-click, choose Tortoise Git, Switch/Checkout, and select main from the dropdown list of branches.
- Right-click again, choose Tortoise Git, Merge, and select Feature1 from the dropdown list of branches.
- Repeat those two steps on Machine2, merging Feature2 into main.

- On Machine1, pull, then push. Do the same on Machine2. Pull again on Machine1 so it has Machine2's changes.
- You'll find that App1.pjx on Machine1 contains the Test.prg added on the Feature2 branch of Machine2 and Test.scx on Machine2 contains the new button added on the Feature1 branch of Machine1.

Now let's try something a little scary: let's lose some changes. On Machine1, switch to the Feature1 branch and notice that Test.prg no longer exists. Trust the repository! Switch back to main and it's back again. You can easily switch from one branch to another and Git will restore all files to the condition they were in on that branch.

Suppose rather than allowing developers to merge their own changes into main, you require that all changes are first reviewed by a manager. In that case, you'll use pull requests to request changes be merged from one branch into another.

One way to do that is to click the Compare & Pull Request button in GitHub (**Figure 19**). You can specify a title and a description for the pull request.

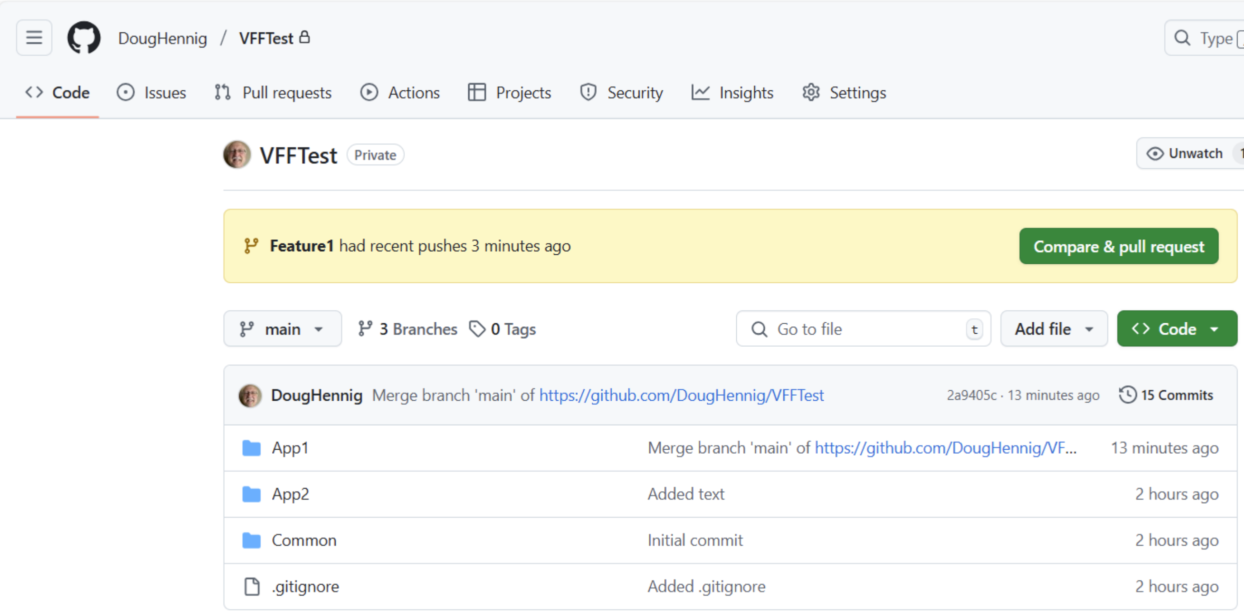


Figure 19. Create a pull request to request that changes in one branch be merged into another.

The pull request appears on the Pull Request tab in GitHub (**Figure 20**). The manager can view the commits and what changes were made to what files. They can add a comment and go back and forth with the developer until they're satisfied, then merge the pull request.

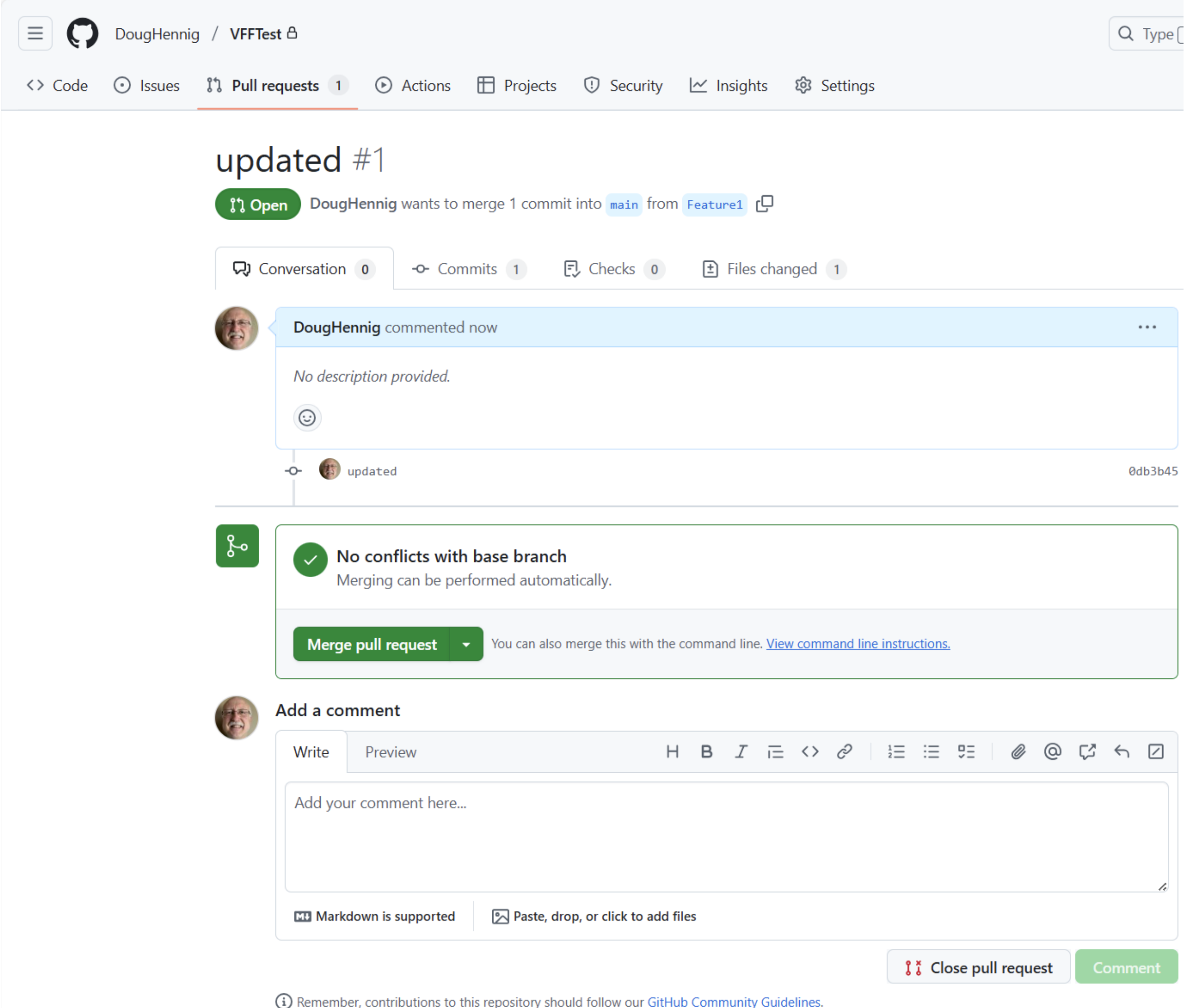


Figure 20. The manager can view, comment, and merge a pull request on the Pull Requests tab.

See https://www.geeksforgeeks.org/git-pull-request for more information about pull requests.

SourceTree

While I typically use the TortoiseGit functions in the File Explorer shortcut menu for simple things like commits, pulls, and pushes, other things are better done in better GUIs, such as SourceTree (Figure 12). For example, TortoiseGit's UI for looking at the history of commits or file changes is poor, while SourceTree's is easy to navigate. SourceTree also allows you to search for commit messages or file changes (**Figure 21**). You may find yourself using SourceTree for all Git-related tasks and forget about TortoiseGit altogether.

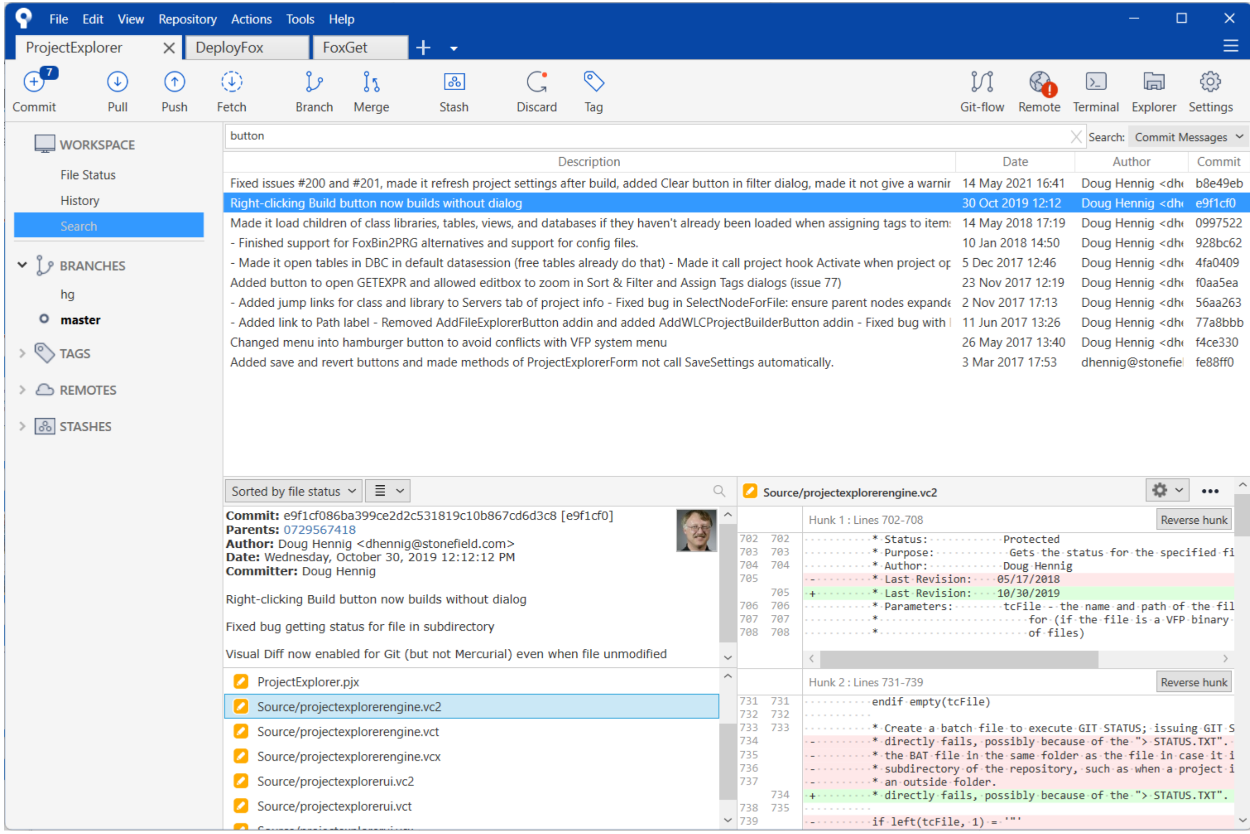


Figure 21. SourceTree allows you to search for commit messages or file changes.

Summary

Version control is essential for modern professional development. If you haven't implemented version control yet, hopefully this document demonstrated it's not that hard to get started and once you do, you'll never go back.

Biography

Doug Hennig is a partner with Stonefield Software Inc. He is the author of the award-winning Stonefield Query; the award-winning Stonefield Database Toolkit (SDT) (now open source); the MemberData Editor, Anchor Editor, and CursorAdapter and DataEnvironment builders that come with Microsoft Visual FoxPro; and the My namespace and updated Upsizing Wizard in Sedna. He also created several VFPX projects, including Project Explorer, OOP Menu, OOP Reports, and SFMail.

Doug is co-author of *VFPX: Open Source Treasure for the VFP Developer*, *Making Sense of Sedna and SP2*, *Visual FoxPro Best Practices For The Next Ten Years*, the *What's New in Visual FoxPro* series, and *Hacker's Guide to Visual FoxPro 7.0* (now open source). He was the technical editor of *Hacker's Guide to Visual FoxPro 6.0* and *The Fundamentals*. Doug wrote hundreds of articles in 20 years for *FoxRockX*, FoxTalk, FoxPro Advisor, Advisor Guide to Visual FoxPro, and CoDe magazines.

Doug spoke at every Microsoft FoxPro Developers Conference (DevCon) starting in 1997 and at user groups and developer conferences all over the world. He is one of the organizers of the Southwest Fox and Virtual Fox Fest conferences. He is one of the administrators for the VFPX VFP community extensions Web site. He was a Microsoft Most Valuable Professional (MVP) from 1996 through 2011. Doug was awarded the 2006 FoxPro Community Lifetime Achievement Award.