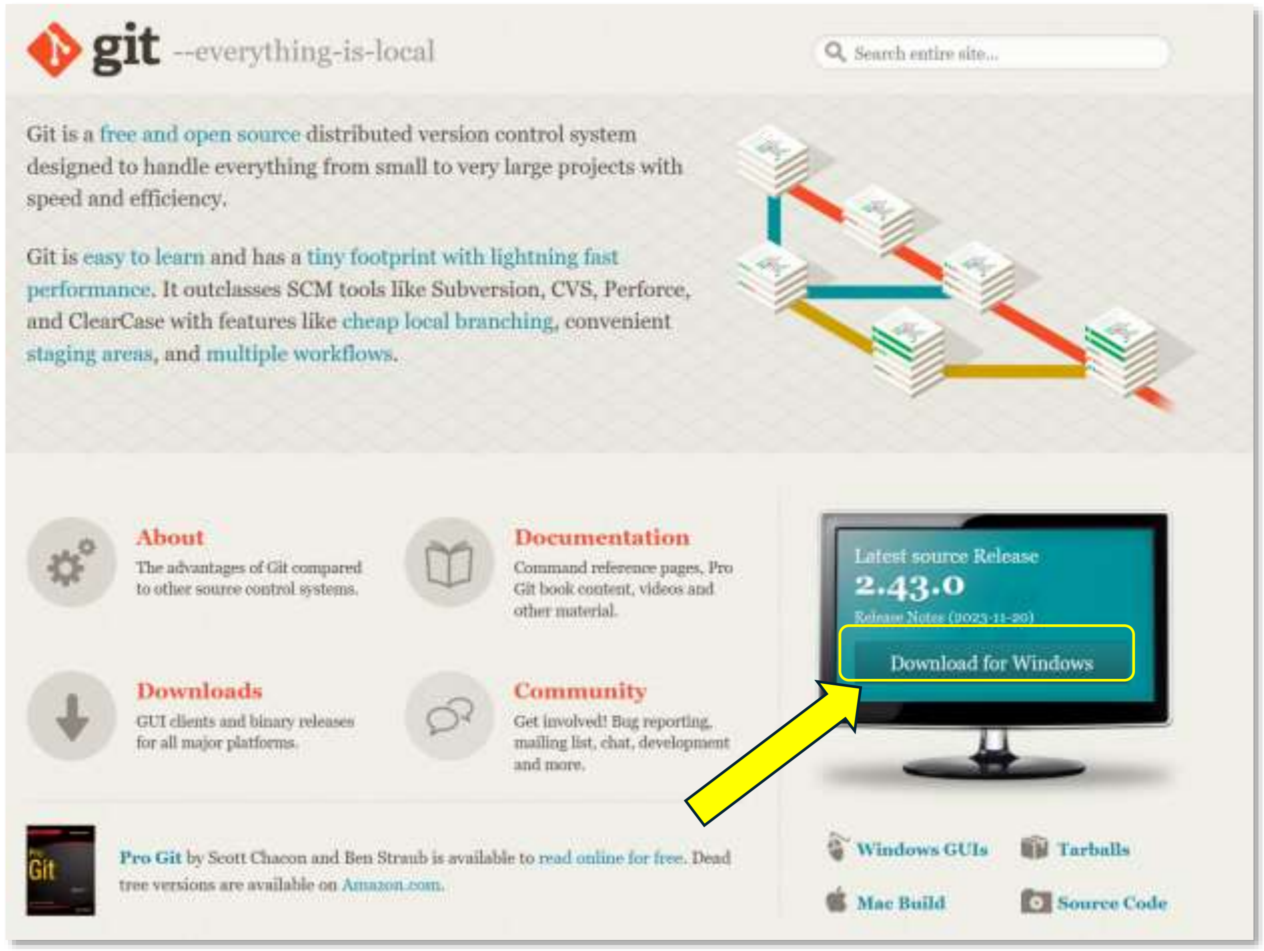
Git, GitHub, & Azure DevOps

Introduction and installation

This curriculum is focused on the tools available for source control which is a vital technology for all software developers. In this class, the tools being used are Git, GitHub, and Azure DevOps.

1.1. Installing Git

To install Git, navigate to http://git-scm.com.



git --everything-is-local

Search entire site...

Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency.

Git is easy to learn and has a tiny footprint with lightning fast performance. It outclasses SCM tools like Subversion, CVS, Perforce, and ClearCase with features like cheap local branching, convenient staging areas, and multiple workflows.

About
The advantages of Git compared to other source control systems.

Documentation
Command reference pages, Pro Git book content, videos and other material.

Downloads
GUI clients and binary releases for all major platforms.

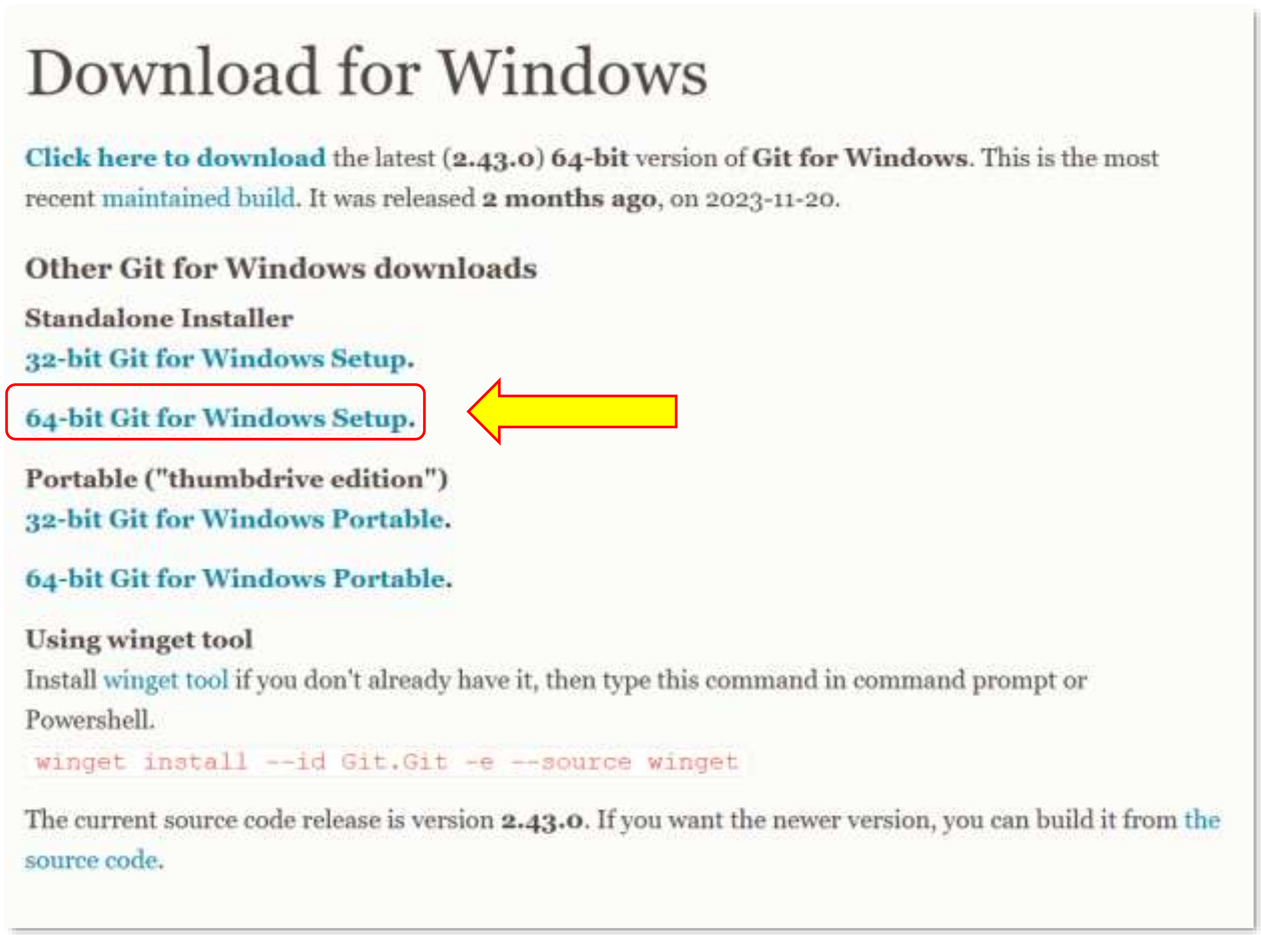
Community
Get involved! Bug reporting, mailing list, chat, development and more.

Latest source Release
2.43.0
Release Notes (2023-11-20)
Download for Windows

Pro Git by Scott Chacon and Ben Straub is available to read online for free. Dead tree versions are available on Amazon.com.

Windows GUIs Tarballs
Mac Build Source Code

The green screen monitor should be displaying the current version of Git and your operating system and a button to download to your disk. Click that button. It will display a page to select the version of Git to download.



Download for Windows

Click here to download the latest (2.43.0) 64-bit version of Git for Windows. This is the most recent maintained build. It was released 2 months ago, on 2023-11-20.

Other Git for Windows downloads

Standalone Installer

32-bit Git for Windows Setup.

64-bit Git for Windows Setup.

Portable ("thumbdrive edition")

32-bit Git for Windows Portable.

64-bit Git for Windows Portable.

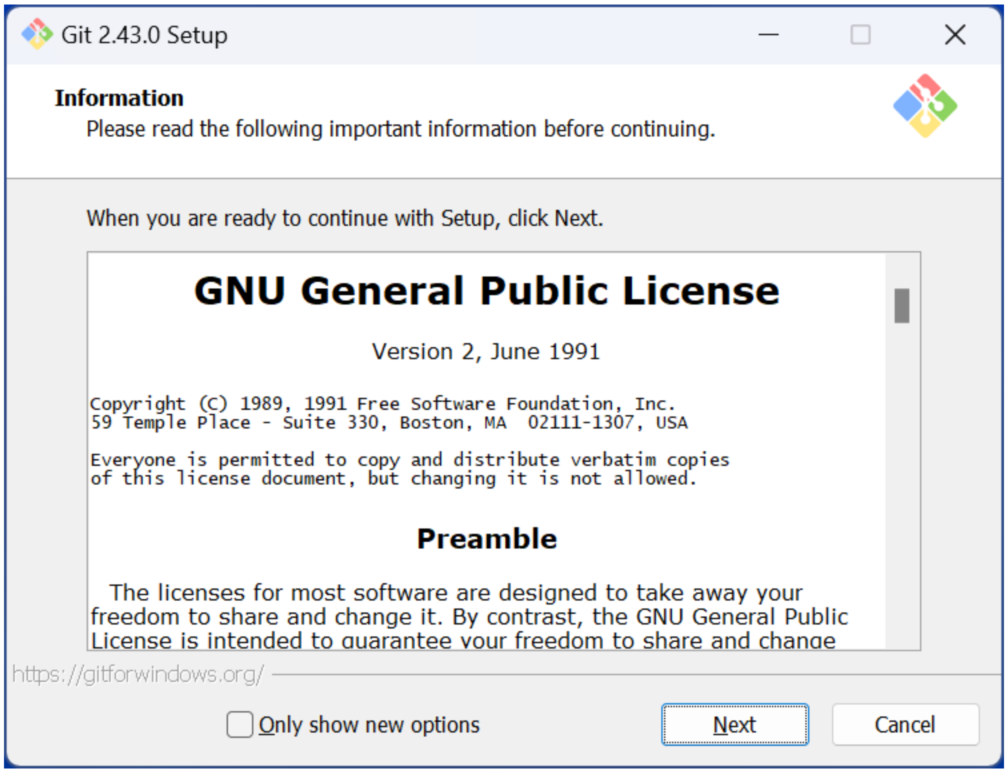
Using winget tool

Install winget tool if you don't already have it, then type this command in command prompt or Powershell.

```
winget install --id Git.Git -e --source winget
```

The current source code release is version 2.43.0. If you want the newer version, you can build it from the source code.

Because the architecture of virtually all current computers are 64 bits, select that option to download and install. The file is downloaded to your Downloads folder. The file name will start with Git and end with .exe. Execute the file to start the installation.



Information

Please read the following important information before continuing.

When you are ready to continue with Setup, click Next.

GNU General Public License

Version 2, June 1991

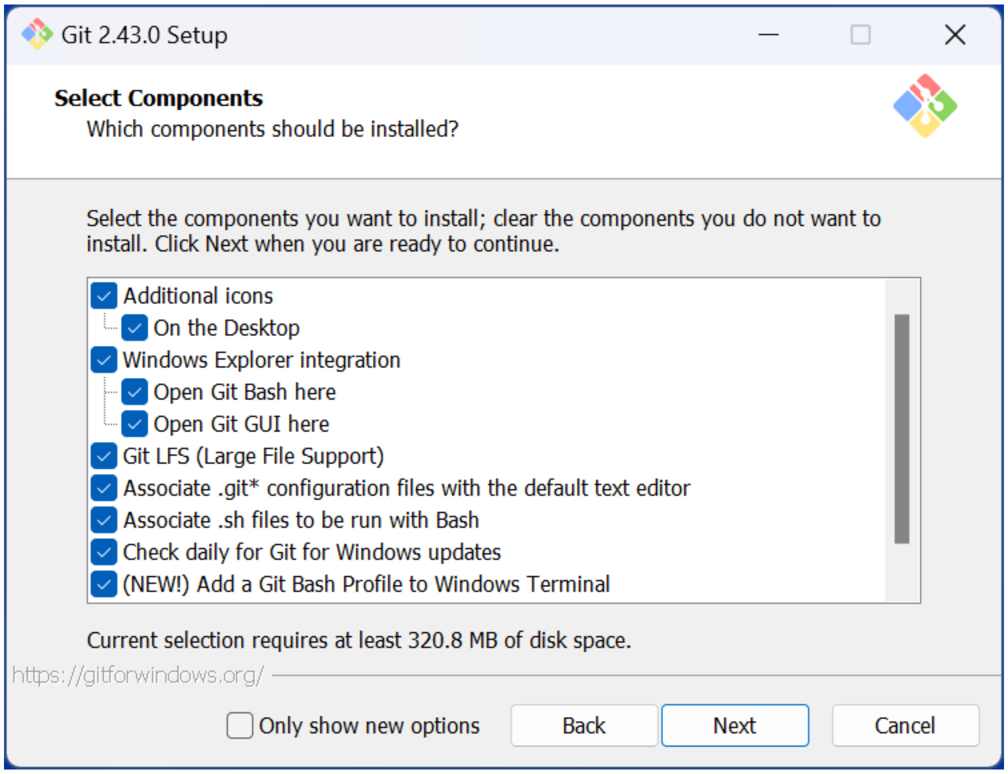
```
Copyright (C) 1989, 1991 Free Software Foundation, Inc.
59 Temple Place - Suite 330, Boston, MA  02111-1307, USA

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.
```

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change

This dialog displays the Git license. For the initial installation, do not check the "Open show new options"

Click Next.



Select Components

Which components should be installed?

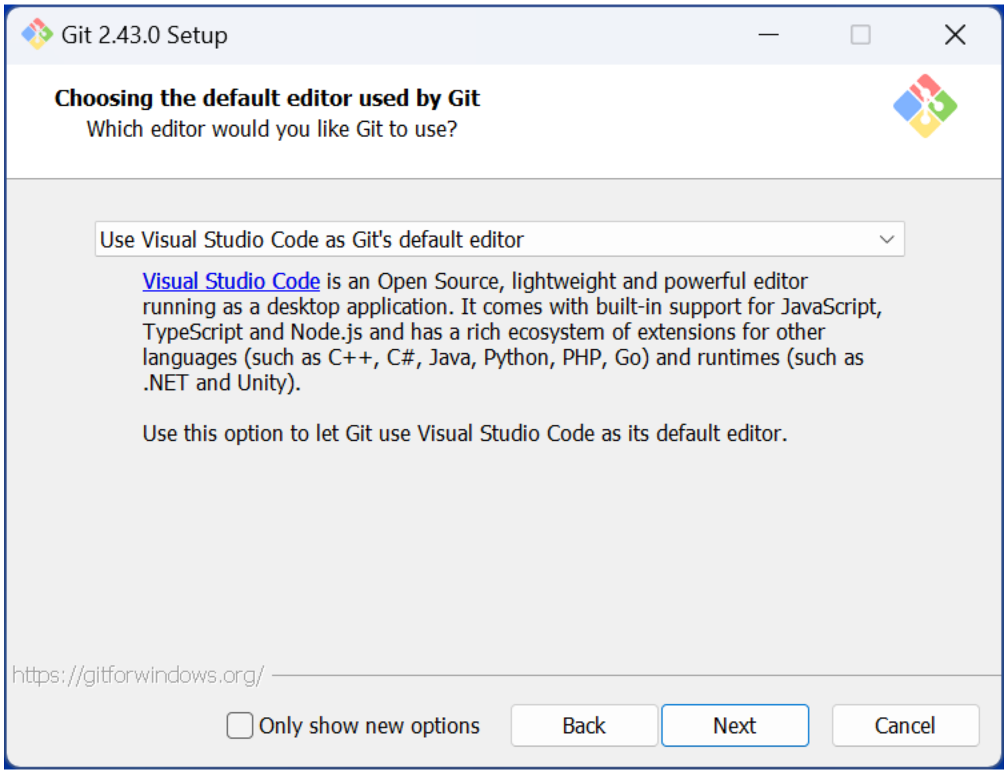
Select the components you want to install; clear the components you do not want to install. Click Next when you are ready to continue.

- [✓] Additional icons
  - [✓] On the Desktop
- [✓] Windows Explorer integration
  - [✓] Open Git Bash here
  - [✓] Open Git GUI here
- [✓] Git LFS (Large File Support)
- [✓] Associate .git* configuration files with the default text editor
- [✓] Associate .sh files to be run with Bash
- [✓] Check daily for Git for Windows updates
- [✓] (NEW!) Add a Git Bash Profile to Windows Terminal

Current selection requires at least 320.8 MB of disk space.

This dialog displays the components that are selected to be installed. You should check all the boxes.

Click Next.



Git 2.43.0 Setup

Choosing the default editor used by Git

Which editor would you like Git to use?

Use Visual Studio Code as Git's default editor

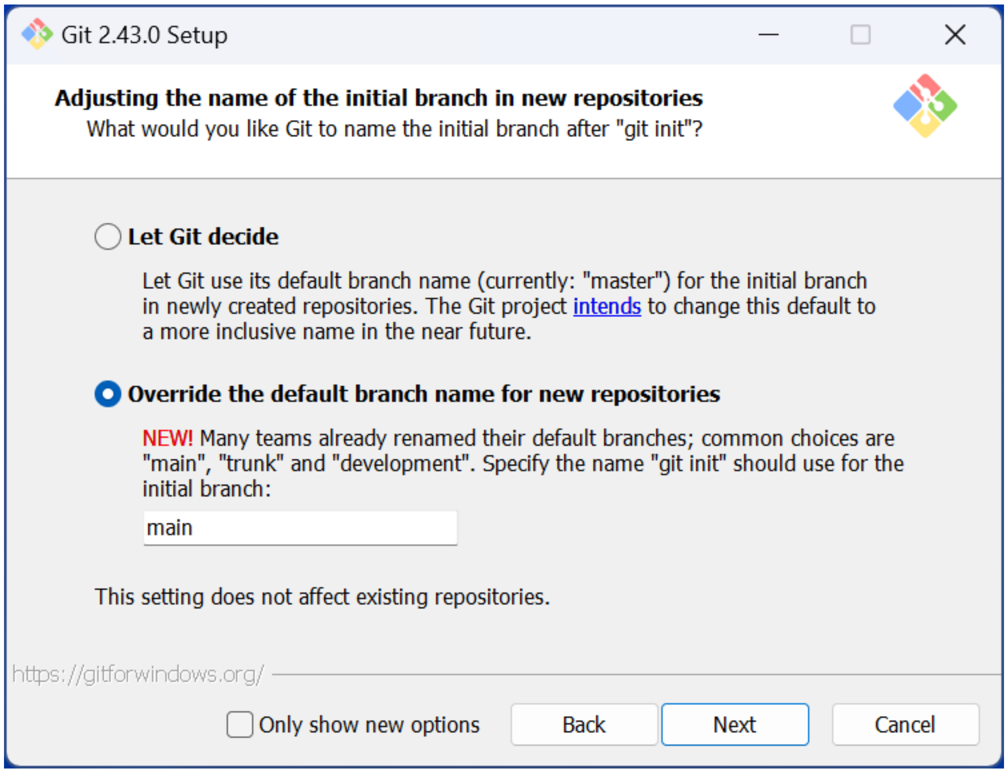
Visual Studio Code is an Open Source, lightweight and powerful editor running as a desktop application. It comes with built-in support for JavaScript, TypeScript and Node.js and has a rich ecosystem of extensions for other languages (such as C++, C#, Java, Python, PHP, Go) and runtimes (such as .NET and Unity).

Use this option to let Git use Visual Studio Code as its default editor.

https://gitforwindows.org/

There are times when Git needs to open a text editor. This dialog allows you to select which text editor Git should open. If you have Visual Studio Code install, that is a good choice as displayed in the image. If no text editors have been installed, you can select notepad. You can change this option if you installed a good programmers editor in the future.

Click Next.



Adjusting the name of the initial branch in new repositories

What would you like Git to name the initial branch after "git init"?

( ) **Let Git decide**

Let Git use its default branch name (currently: "master") for the initial branch in newly created repositories. The Git project intends to change this default to a more inclusive name in the near future.

[✓ **Override the default branch name for new repositories**]

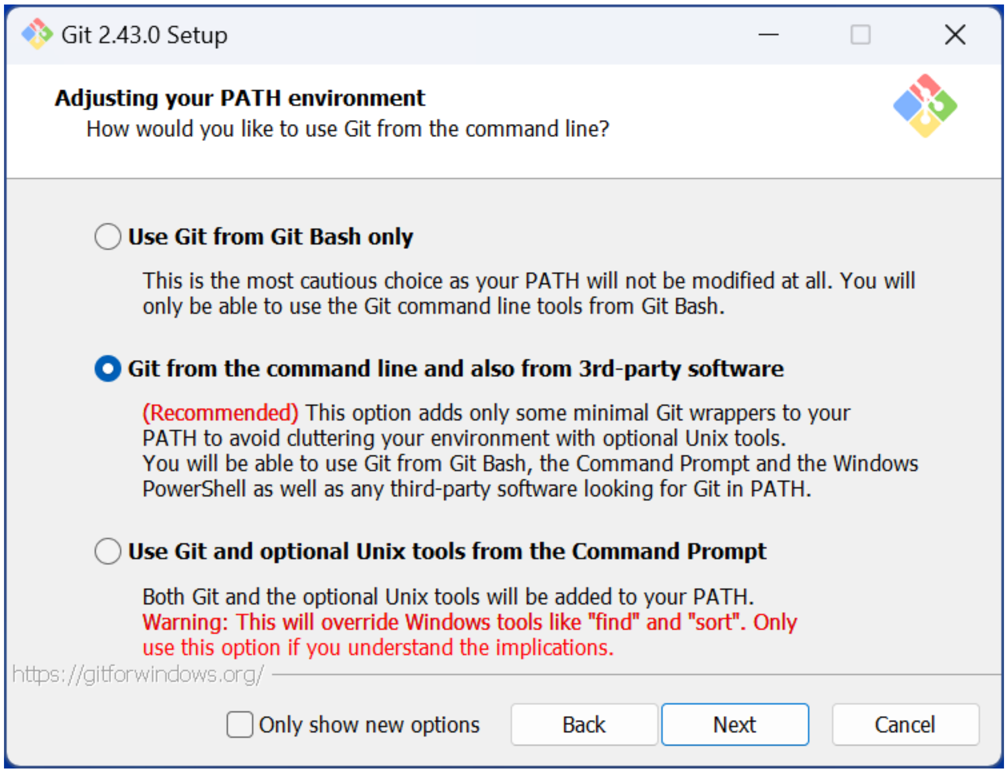
NEW! Many teams already renamed their default branches; common choices are "main", "trunk" and "development". Specify the name "git init" should use for the initial branch:

main

This setting does not affect existing repositories.

This dialog defines the name of the initial branch that is created whenever a new repository is created. Choosing “Let Git decide” means the branch name will be “master”. Most developers are choosing a different name so clicking the “Override the default branch name for new repositories” and type “main” in the text box as shown above.

Click Next.



Adjusting your PATH environment

How would you like to use Git from the command line?

( ) **Use Git from Git Bash only**

This is the most cautious choice as your PATH will not be modified at all. You will only be able to use the Git command line tools from Git Bash.

[✓] **Git from the command line and also from 3rd-party software**

(Recommended) This option adds only some minimal Git wrappers to your PATH to avoid cluttering your environment with optional Unix tools. You will be able to use Git from Git Bash, the Command Prompt and the Windows PowerShell as well as any third-party software looking for Git in PATH.

( ) **Use Git and optional Unix tools from the Command Prompt**

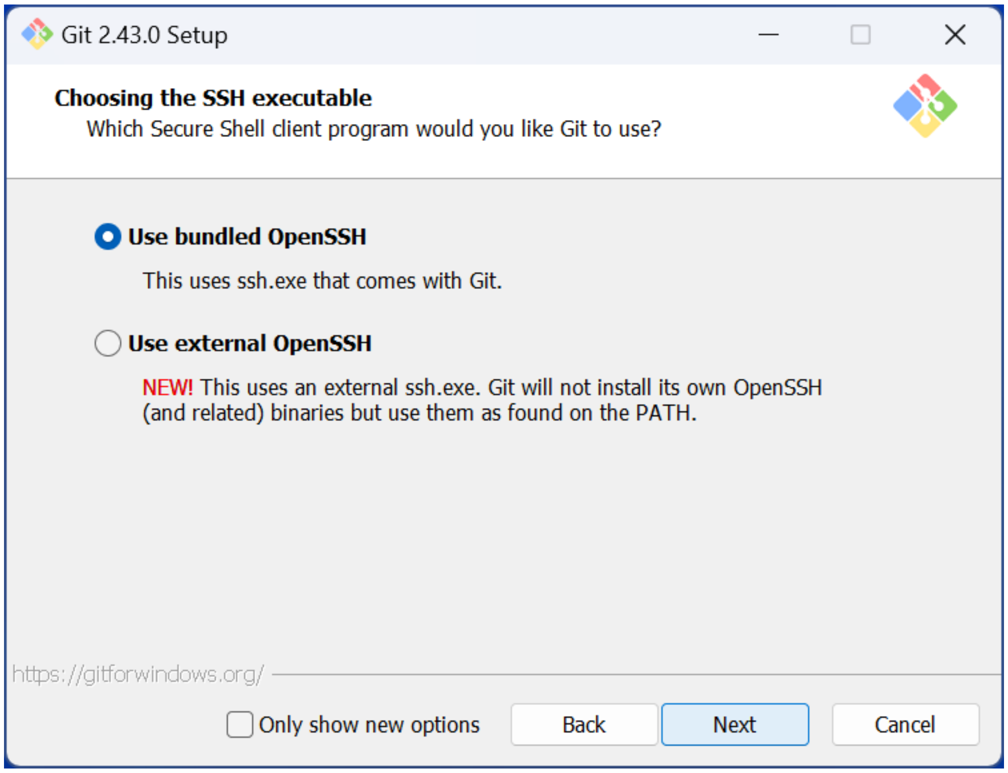
Both Git and the optional Unix tools will be added to your PATH. Warning: This will override Windows tools like "find" and "sort". Only use this option if you understand the implications.

https://gitforwindows.org/

[ ] Only show new options

This dialog defines which command interfaces (i.e. cmd, bash) you can use to enter Git commands. The recommended options is the best choice because it allows using both cmd and bash.

Click Next.



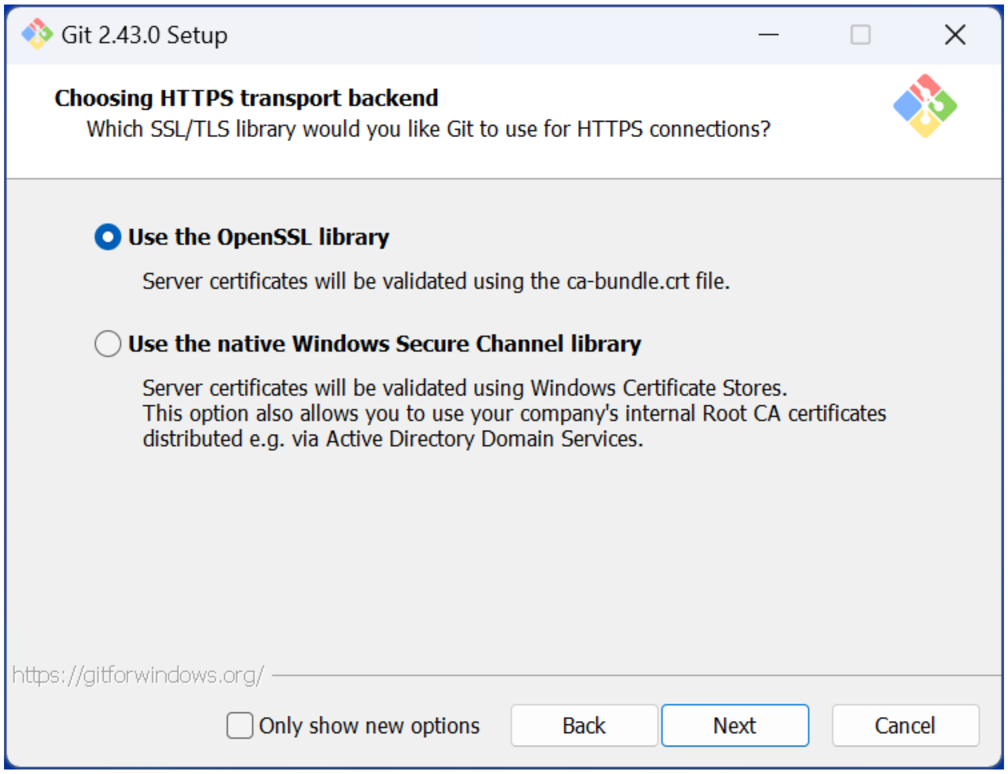
Choosing the SSH executable
Which Secure Shell client program would you like Git to use?

[✓ Use bundled OpenSSH]
This uses ssh.exe that comes with Git.

Use external OpenSSH
NEW! This uses an external ssh.exe. Git will not install its own OpenSSH (and related) binaries but use them as found on the PATH.

This dialog defines whether to use the SSH included with Git or to use a version of SSH that is already installed on your system. You should typically select "Use bundled OpenSSH".

Click Next.



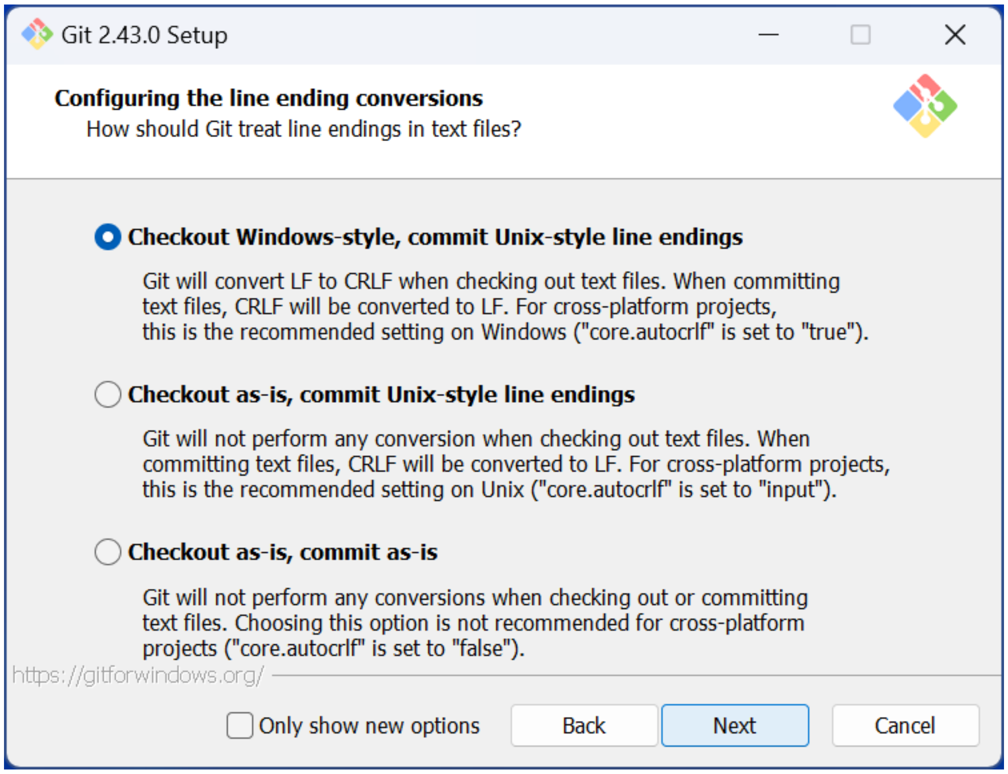
Choosing HTTPS transport backend
Which SSL/TLS library would you like Git to use for HTTPS connections?

[✓ Use the OpenSSL library]
Server certificates will be validated using the ca-bundle.crt file.

Use the native Windows Secure Channel library
Server certificates will be validated using Windows Certificate Stores.
This option also allows you to use your company's internal Root CA certificates distributed e.g. via Active Directory Domain Services.

This dialog will define which SSH support to use when issuing Git command over a HTTPS connection. Select “Use the OpenSSL library”.

Click Next.



Configuring the line ending conversions

How should Git treat line endings in text files?

[✓ Checkout Windows-style, commit Unix-style line endings]

Git will convert LF to CRLF when checking out text files. When committing text files, CRLF will be converted to LF. For cross-platform projects, this is the recommended setting on Windows ("core.autocrlf" is set to "true").

Checkout as-is, commit Unix-style line endings

Git will not perform any conversion when checking out text files. When committing text files, CRLF will be converted to LF. For cross-platform projects, this is the recommended setting on Unix ("core.autocrlf" is set to "input").

Checkout as-is, commit as-is

Git will not perform any conversions when checking out or committing text files. Choosing this option is not recommended for cross-platform projects ("core.autocrlf" is set to "false").

This dialog defines the control characters that will be on the end of text lines when stored in the repository.

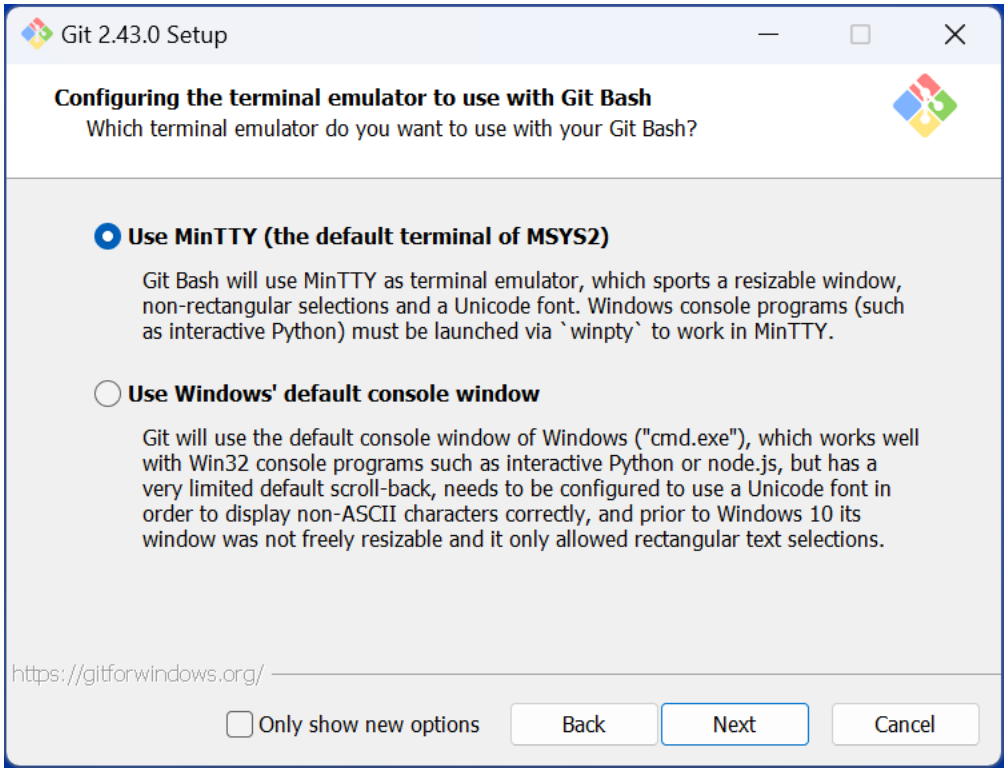
In Windows system, text lines end with two invisible control characters called the carriage return (CR) and line feed (LF). On Linux and MAC system, the text lines end with only a LF.

Selecting the “Checkout Windows-style, commit Unix-style line endings” will convert the LF endings to CRLF automatically when copying file from Git to the Windows disk so that all text tool can operating properly. When the file is copied back to Git, the CRLF will be converted back to LF. **This is the suggested option.**

Selecting the “Checkout as-is, commit Unix-style line endings” will not do any conversion when files are copied to the local disk. When the files are copied back to Git, any CRLF endings will be converted to LF.

Selecting the “Checkout as-is, commit as-is” will do not conversion when coping to disk or back to Git.

Click Next.



Configuring the terminal emulator to use with Git Bash
Which terminal emulator do you want to use with your Git Bash?

[✓] **Use MinTTY (the default terminal of MSYS2)**

Git Bash will use MinTTY as terminal emulator, which sports a resizable window, non-rectangular selections and a Unicode font. Windows console programs (such as interactive Python) must be launched via `winpty` to work in MinTTY.

[ ] **Use Windows' default console window**

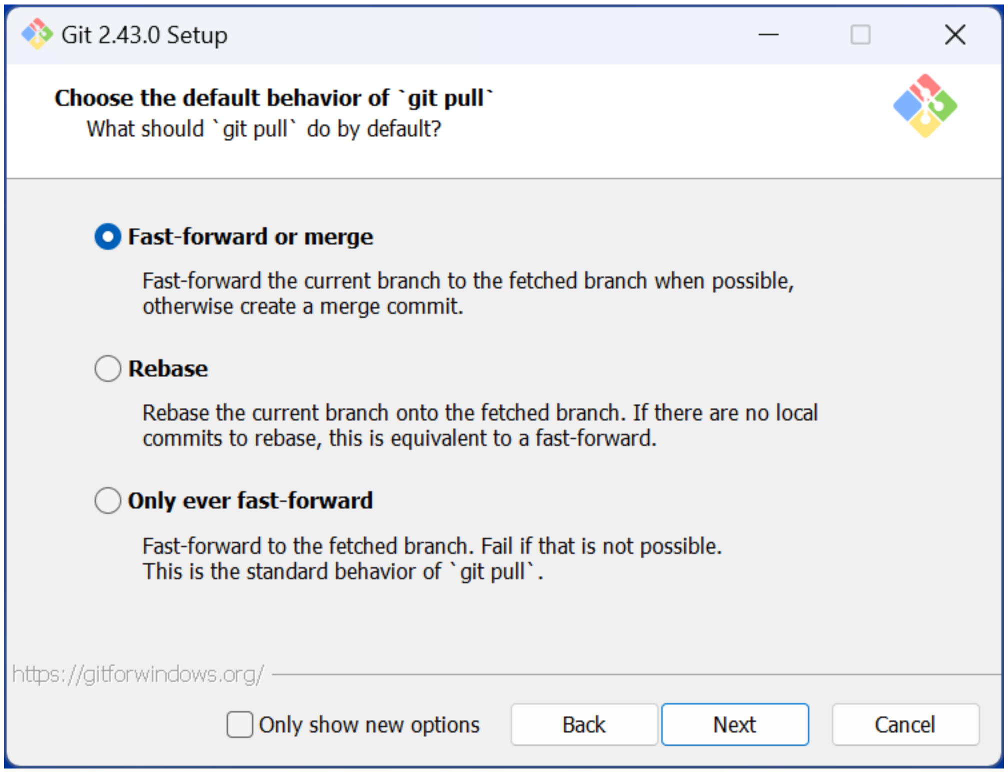
Git will use the default console window of Windows ("cmd.exe"), which works well with Win32 console programs such as interactive Python or node.js, but has a very limited default scroll-back, needs to be configured to use a Unicode font in order to display non-ASCII characters correctly, and prior to Windows 10 its window was not freely resizable and it only allowed rectangular text selections.

https://gitforwindows.org/

[ ] Only show new options

This dialog will give the option so use the bash shell application or the Window default cmd window. If the “Use MinTTY (the default terminal of MSYS2)” is used, the commands to navigate the disk system will be those similar to Linux & MAC. **This is the suggested option.** The “Use Windows’ default console window” will use standard Windows cli disk command.

Click Next



Choose the default behavior of `git pull`
What should `git pull` do by default?

● **Fast-forward or merge**
Fast-forward the current branch to the fetched branch when possible, otherwise create a merge commit.

○ **Rebase**
Rebase the current branch onto the fetched branch. If there are no local commits to rebase, this is equivalent to a fast-forward.

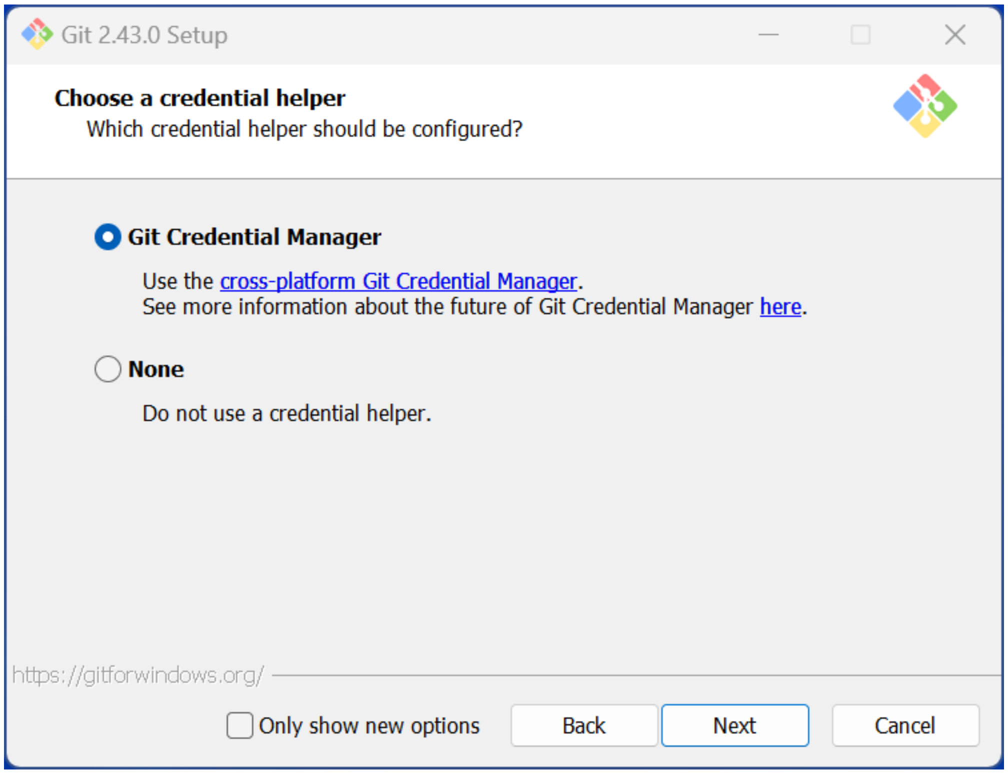
○ **Only ever fast-forward**
Fast-forward to the fetched branch. Fail if that is not possible. This is the standard behavior of `git pull`.

https://gitforwindows.org/

☐ Only show new options | Back | Next | Cancel

This dialog selects the default operating when doing a Git Pull which is copying code from the cloud repository to the local repository. Select “Fast-forward or merge”.

Click Next.



Choose a credential helper
Which credential helper should be configured?

● **Git Credential Manager**
Use the cross-platform Git Credential Manager.
See more information about the future of Git Credential Manager here.

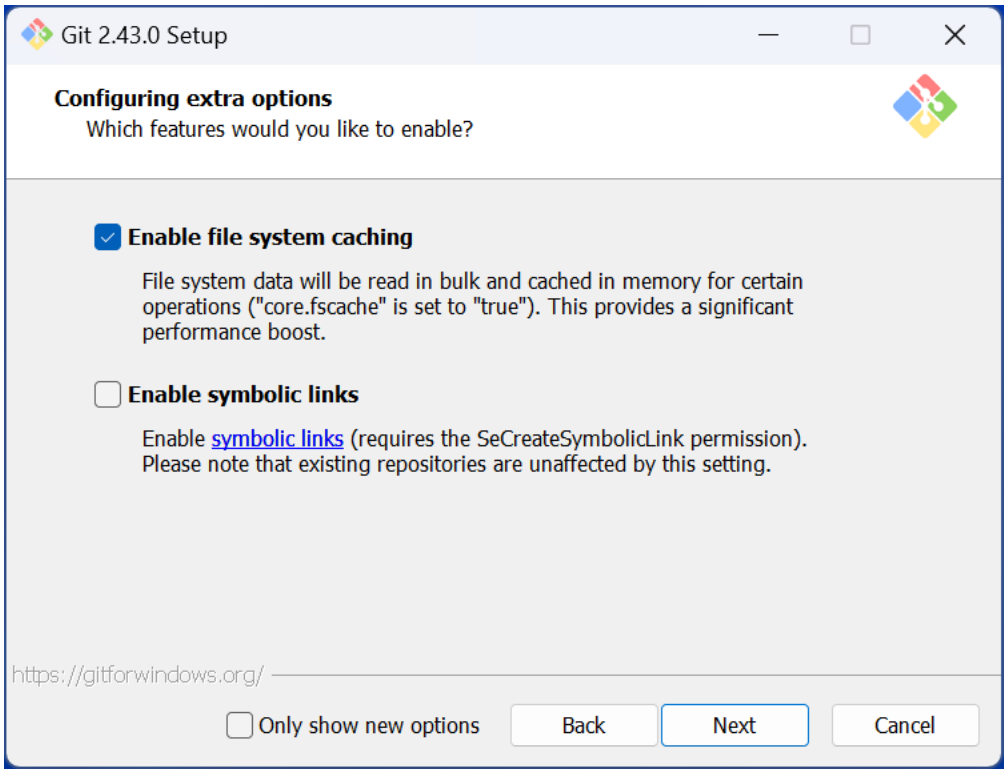
○ **None**
Do not use a credential helper.

https://gitforwindows.org/

☐ Only show new options | Back | Next | Cancel

This dialog allows choosing whether to use the Git Credential Manager or not. Especially if using Azure DevOps, the Git Credential Manager should be used. Select it.

Click Next.



Git 2.43.0 Setup

Configuring extra options
Which features would you like to enable?

[✓] **Enable file system caching**

File system data will be read in bulk and cached in memory for certain operations ("core.fscache" is set to "true"). This provides a significant performance boost.

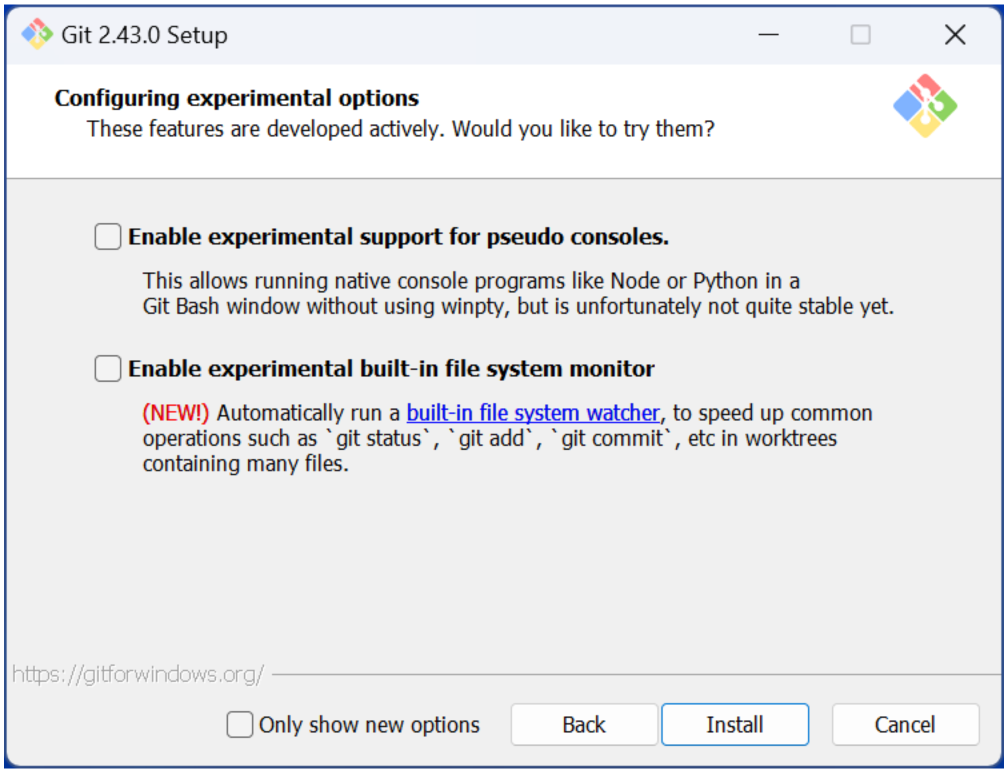
[ ] **Enable symbolic links**

Enable symbolic links (requires the SeCreateSymbolicLink permission). Please note that existing repositories are unaffected by this setting.

https://gitforwindows.org/

This dialog displays two options. The first is to enable file system caching which should be checked. The enable symbolic links option is not needed and can be left unchecked.

Click Next.



Git 2.43.0 Setup

Configuring experimental options

These features are developed actively. Would you like to try them?

[ ] **Enable experimental support for pseudo consoles.**

This allows running native console programs like Node or Python in a Git Bash window without using winpty, but is unfortunately not quite stable yet.

[ ] **Enable experimental built-in file system monitor**

(NEW!) Automatically run a built-in file system watcher, to speed up common operations such as `git status`, `git add`, `git commit`, etc in worktrees containing many files.

https://gitforwindows.org/

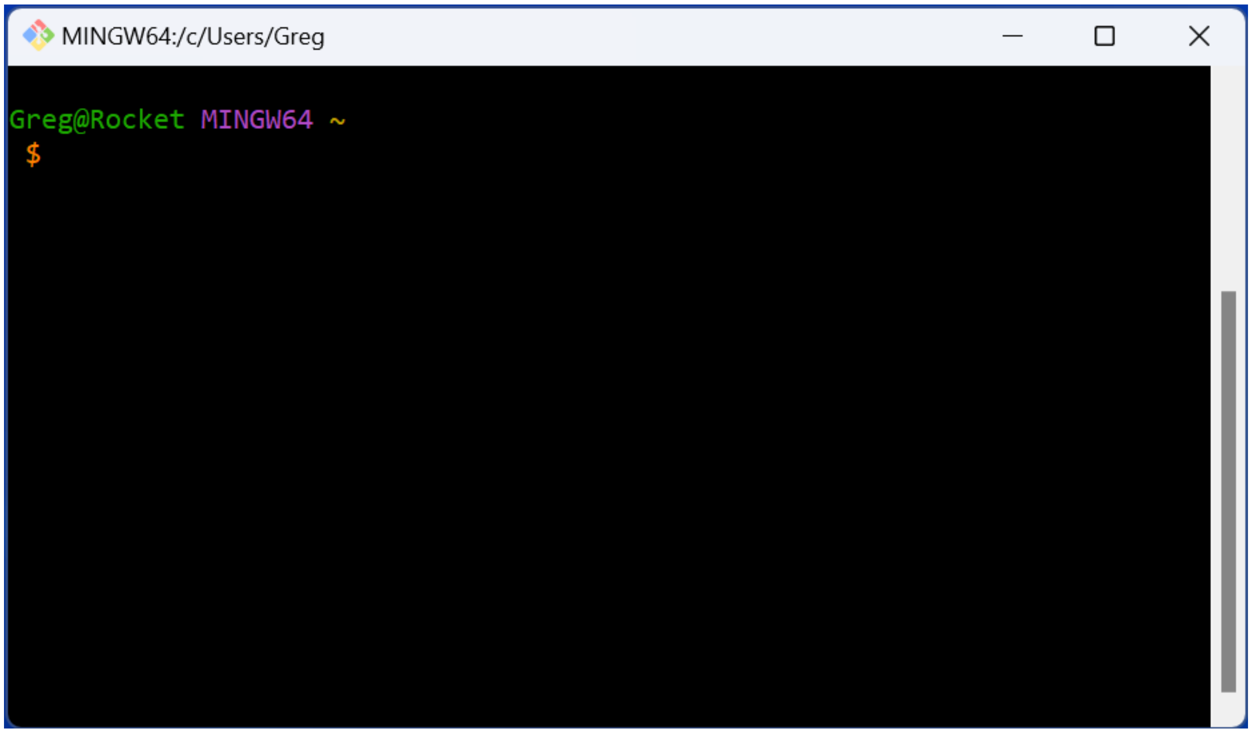
[ ] Only show new options

This dialog allows enabling experimental features. They should be left unchecked.

Click Install.

1.2. Verifying the Git install

When the installation is complete, open the program called “Git Bash”. You should see a text line with multiple colors and a prompt on the following line ending with a dollar sign ($).



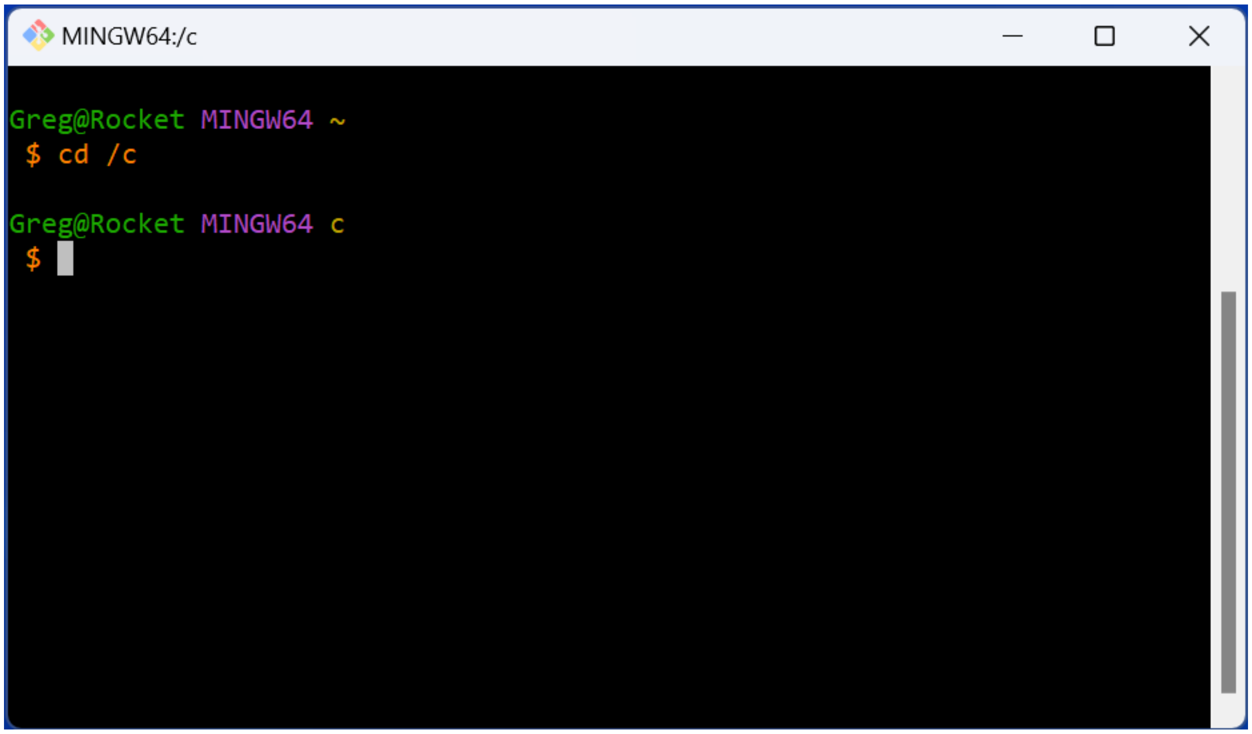
```
Greg@Rocket MINGW64 ~
 $
```

The green text is the user and name of the machine.

The purple text is the name of the bash shell program.

The gold text is the current directory. The tilde character (~) signifies the users home folder.

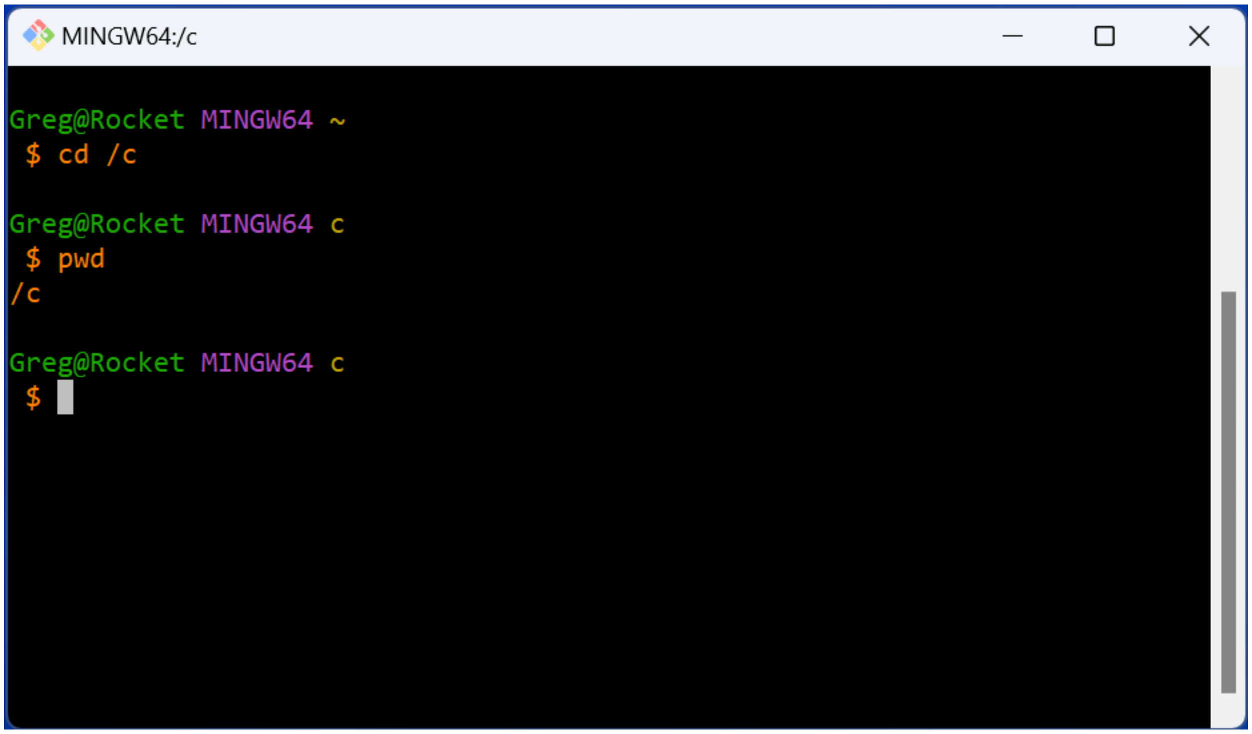
Type “cd /c” and enter.



```
Greg@Rocket MINGW64 ~
 $ cd /c

Greg@Rocket MINGW64 c
 $
```

The prompt should show "c" in gold

Type "pwd" and enter



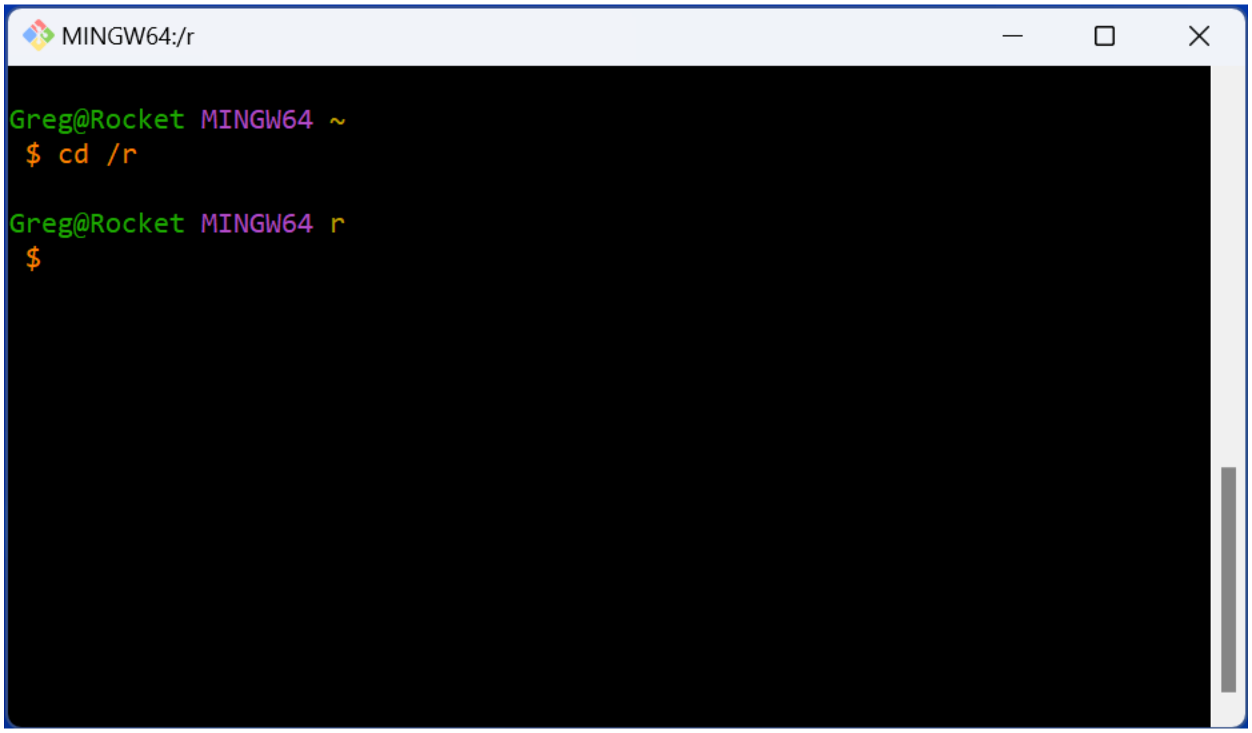
```
Greg@Rocket MINGW64 ~
 $ cd /c

Greg@Rocket MINGW64 c
 $ pwd
/c

Greg@Rocket MINGW64 c
 $
```

It should display "/c" on the next line.

1.3. The "repos" folder

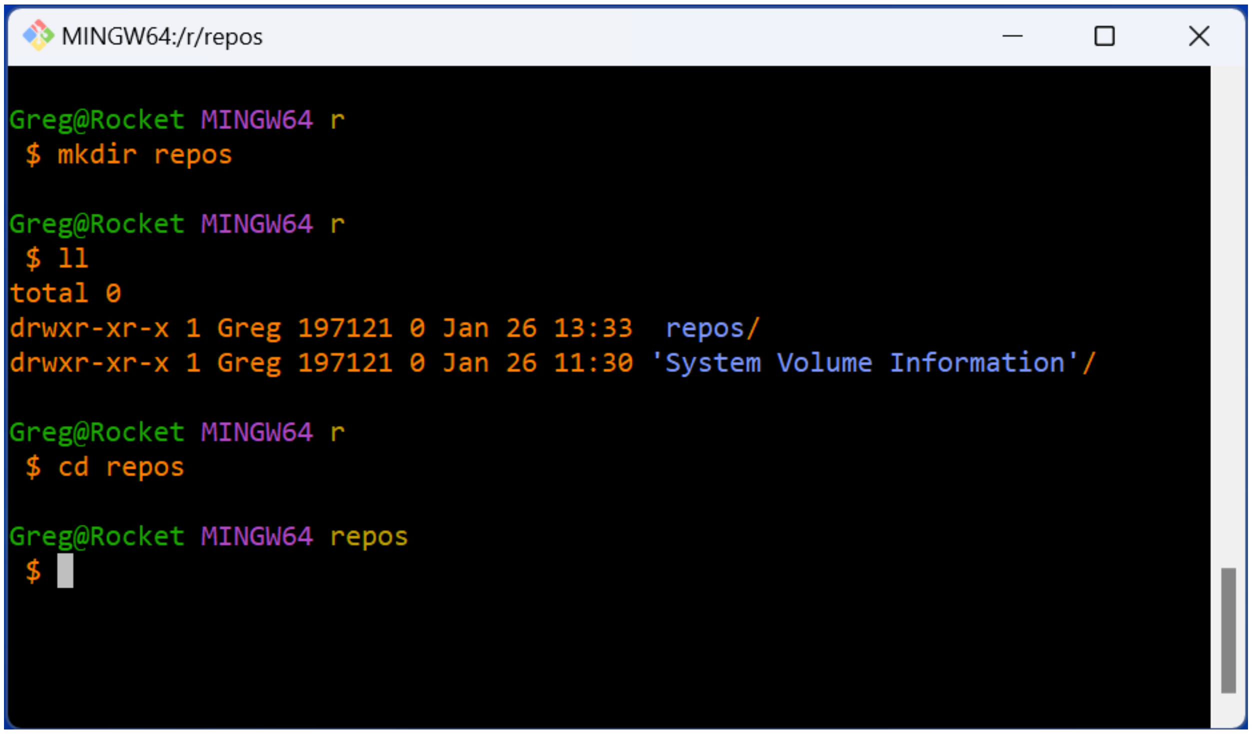
As more and more source code projects are added to the local disk, it will get harder and harder to keep track of where projects are on the disk. To help keep projects manageable, it is suggested that a folder named "repos" be created in the root of the C drive or any other drive and have all the projects created inside the repos folder or a subfolder of repos.

To create the repos folder, start by navigating to the root of the C (or any other drive) using the Git bash shell. Once there, enter the following command:


MINGW64:/r

```
Greg@Rocket MINGW64 ~
 $ cd /r

Greg@Rocket MINGW64 r
 $
```

Type: “mkdir repos” and press enter


MINGW64:/r/repos

```
Greg@Rocket MINGW64 r
 $ mkdir repos

Greg@Rocket MINGW64 r
 $ ll
total 0
drwxr-xr-x 1 Greg 197121 0 Jan 26 13:33  repos/
drwxr-xr-x 1 Greg 197121 0 Jan 26 11:30 'System Volume Information'/

Greg@Rocket MINGW64 r
 $ cd repos

Greg@Rocket MINGW64 repos
 $
```

There will be no messages from the “`mkdir repos`” command. With Git, most of the time, when there is no message it means everything worked fine.

Once that’s done, display the file and folder in the drive.

Type: “ll” (double lower-case L) and press enter.

You should see a line that ends with “repos/”. The trailing slash means it is a folder not a file.

Lastly, you can change the current folder and making the new repos folder current.

Type: “cd repos” and press enter.

Notice that the gold text has changed to “repos” which is the name of the current folder. That will always be displayed.

Overview of Source Control

This section will focus on source control describing what it is, how it works, and why it is so important to software developers.

2.1. What is source control?

Source control is a technology that provides code management and revision control of software programs. The text is stored in a special datastore the keeps all versions saved into the datastore and keeps tracks of each version and the change from the previous version.

Source control system usually supports multiple users editing the same file at the same time and merging all the changes into one version of the file. Many times this merging of multiple file changes is done automatically by the source control software.

All the changes made to each file are tracked by the system. When a new file is created and saved to the data store, it will be considered as version 1. When the file is changed and saved again, version 2 is created and marked as the most current version along with version 1 remaining in the data store. Every change to the file and saving it creates a newer version of the file. Older versions are never deleted by the system. The value here is that if the newest version of the file has significant problems, an older version can be retrieved and used while the newest version is fixed.

In addition to managing the versions of each file, source control systems support keeping multiple different copies of source files using the concept of “branches”. Branches allow a programmer to create a copy of some code to make changes while not touching any of the original code. The value is that anyone using the original code and has a problem, another programmer can still use that code to debug and fix the problem.

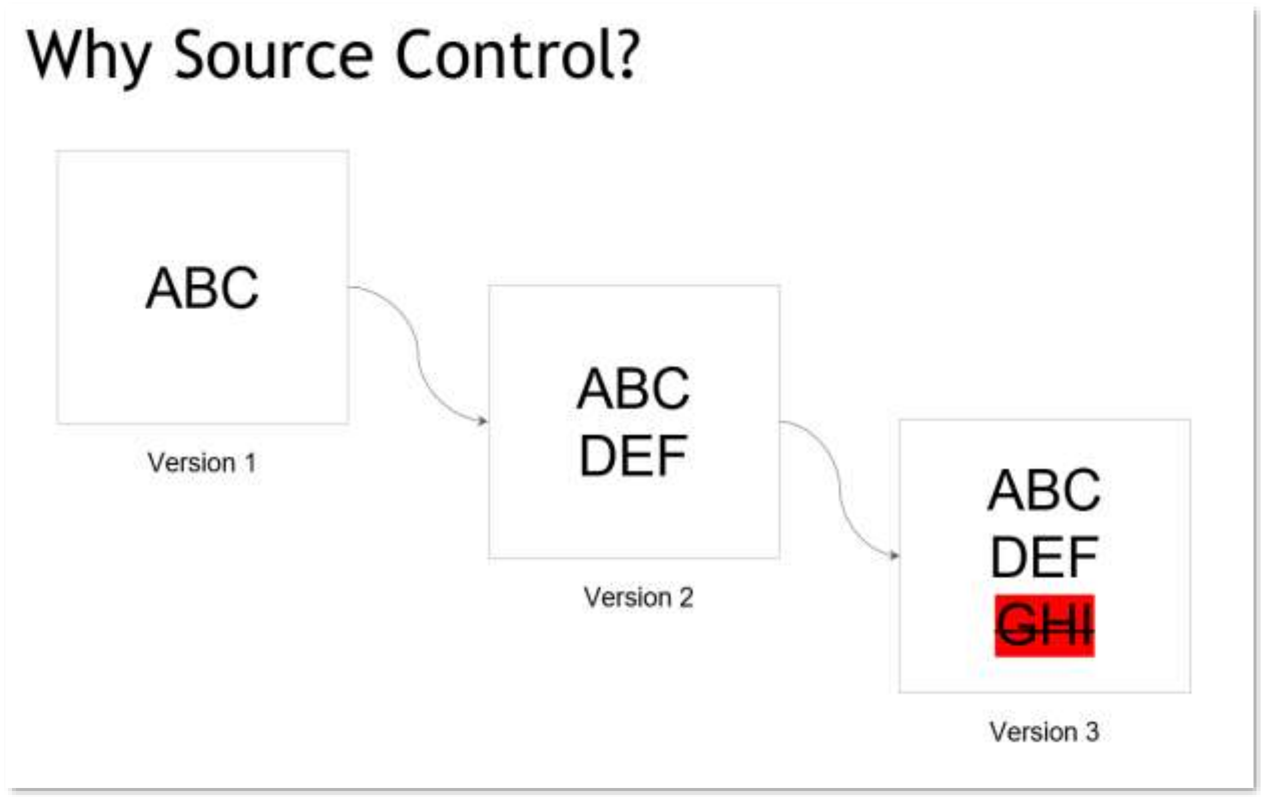


Figure 1

Figure 1 shows the process of a file being created and changed two times with each one being saved to source control. There are three versions of the file with version 3 being the most current.

But a problem is found in the version 3 code which is causing failures in the software that might take a long time to fix. In the meantime, version 2 code can be pulled and deployed since it doesn't contain the bad code. So, version 2 code runs fine but does not contains some new features that were in version 3. Programmers can continue working on version 3 to fix the issue then store the new version 4 code in the data store.

Git

Git was created by Linus Torvalds in 2005 as he and his team were building the Linux kernel. His team was not located in a particular location where they all could easily access a single source control server. So, Linus decided to create a new source control system. One where each programmer would have a data store on their own computer then occasionally copy the work over the Internet to a common server.

Git is a command line interface (CLI) where command are typed at a command prompt and results are display as text messages in the same terminal. Git is the most popular program for local source control.

Here is the format for Git commands:

Format: git [command] [options...]

[command] is lower-case with no spaces

[options] are additional information for some commands

3.1. Bash Shell

The bash shell is a program for Windows that was built to help those users of Linux and MAC adjust to Windows. It does this by allowing Linux & MAC disk commands to be entered and display properly on Windows. For example, using the Windows cmd shell, to display all the files and folders, the command is “dir”. In Linux & MAC, to do the same requires using the “ll” (double L). The bash shell allows entering the “ll” and getting the list of files and folders on Windows.

Here is a list of Linux/MAC commands.

ls :: Displays files and folders in a wide format.

ll :: Displays file and folders in detail format.

mkdir :: Creates a new folder in the current folder by default.

rmdir :: Deletes a folder if it is empty.

touch :: Creates an empty file.

cd :: Changed to another folder or to the home folder if no options.

rm :: Deletes one or more files.

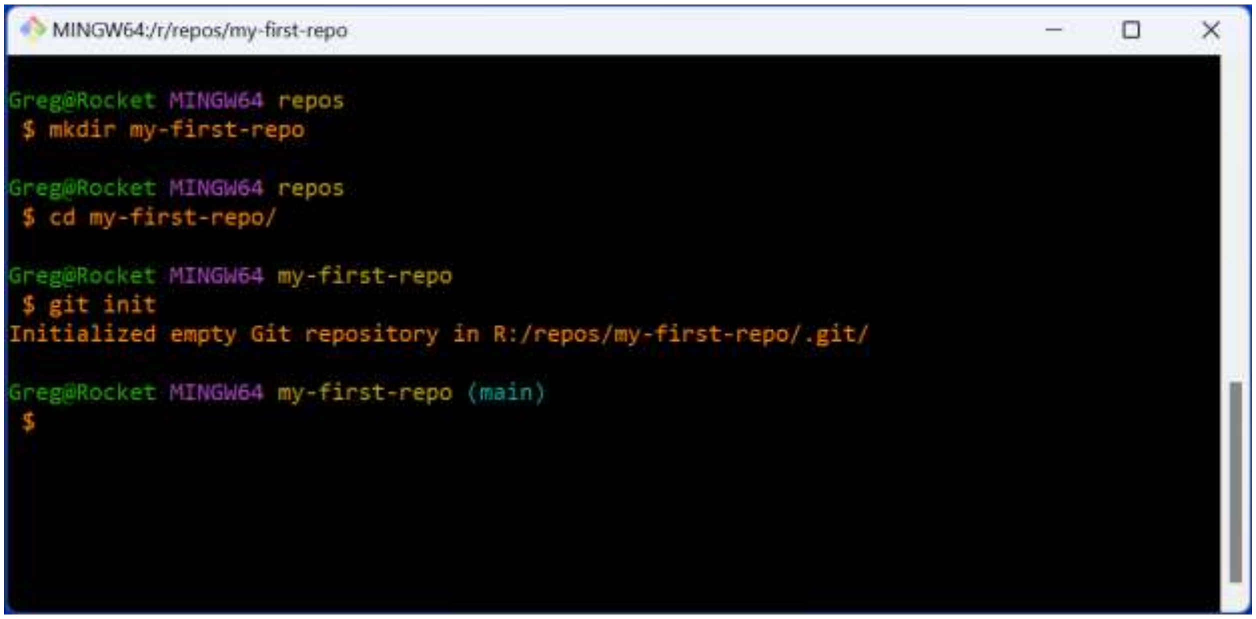
rm -rf .git :: Deletes the hidden Git folder (be careful!).

3.2. How does Git work?

The first step to use Git is to create a new folder on disk and make that folder a *repository*.

Repository (abbreviated as repo) is the name given to the data store used for Git and most source control systems.

To make a folder a repository required initializing it using the git init command:



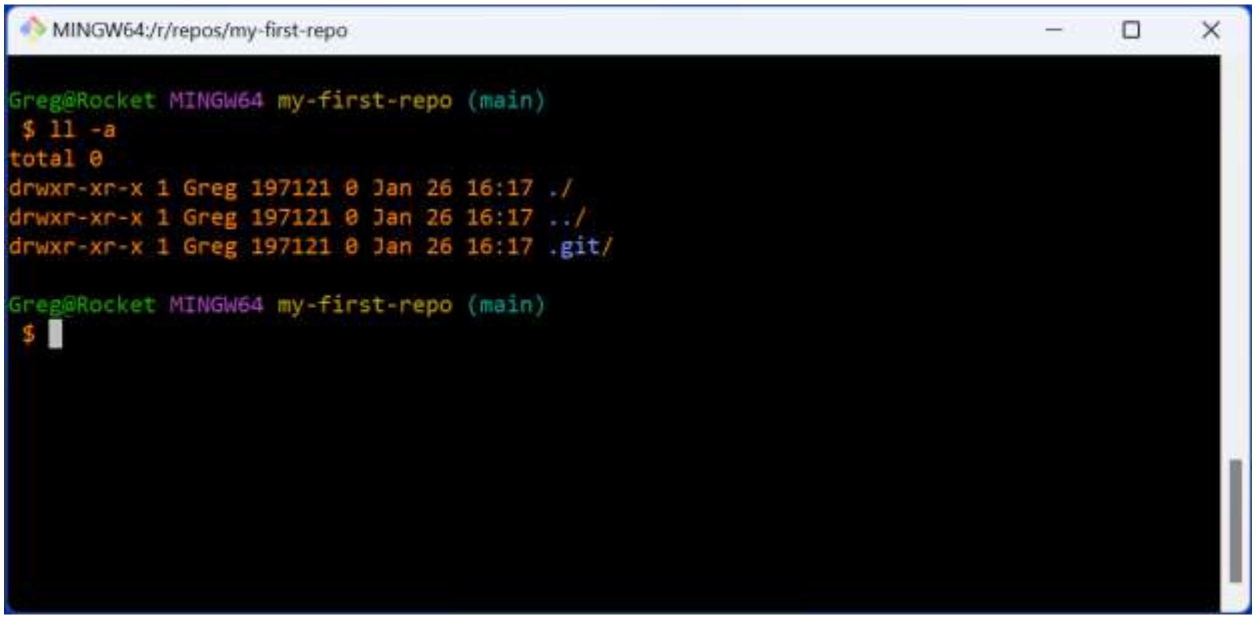
```
Greg@Rocket MINGW64 repos
 $ mkdir my-first-repo

Greg@Rocket MINGW64 repos
 $ cd my-first-repo/

Greg@Rocket MINGW64 my-first-repo
 $ git init
Initialized empty Git repository in R:/repos/my-first-repo/.git/

Greg@Rocket MINGW64 my-first-repo (main)
 $
```

Starting from the repos folder, the first line creates a new folder in repos called 'my-first-repo'. Then we change the current folder from repos to my-first-repo. Then we run the 'git init' which transforms that folder into a git repository. Notice that there is a new piece of blue text in the message which is the default branch that is automatically created.

It must be noted that folders and files created should use lower case letters, numbers, dashes, and underscores. Names should not have upper case letters or blanks in the name.

If we display the contents of our new repository, we'll see that it contains no files or folders. What makes it a repository if there is nothing in the folder? Actually, the folder does contain one item. It is a directory that is hidden from view by default. The folder is named ".git". The period before the g is part of the name. We can display this by executing the command:

```
ll -a
```



MINGW64:/r/repos/my-first-repo

```
Greg@Rocket MINGW64 my-first-repo (main)
 $ ll -a
total 0
drwxr-xr-x 1 Greg 197121 0 Jan 26 16:17 ./
drwxr-xr-x 1 Greg 197121 0 Jan 26 16:17 ../
drwxr-xr-x 1 Greg 197121 0 Jan 26 16:17 .git/

Greg@Rocket MINGW64 my-first-repo (main)
 $
```

On the third line is that hidden folder that makes this folder a Git repo. You can browse through that folder and there are many subfolders but there is not much to see. One thing for sure, you should NOT make any changes anywhere inside the ".git" folder.

Since the ".git" folder makes the folder a repo, removing it reverts the folder back to a plain folder. It could be initialized again, but all that was lost cannot be recovered.

3.3. Workspaces & Branches

When working with Git, there are some terms that should be understood. Workspaces and branches are two of them. If you read the Git documentation, you'll see these terms used and you should know what they mean.

3.3.1. Workspaces

Workspaces are simply the disk on the local computer. For many computers, this is the C-drive and some folder on Windows. The workspace is the repository created when it is initialized by Git. Developers can use many operating system tools to view and manage the files in the workspace.

3.3.2. Branches

Branches are something that exists in every source control system. It supports making a copy of a folder and all the files within allowing the developer to make changes to the files in the new folder without touching the original files in the copied folder.

In Git, you can think of branches are virtual folders within the repository. Git will keep track of everything contained in each branch. The developer can have as many branches as needed.

The folders and files within a branch are available in the workspace. Changing folders or files on the workspace does NOT automatically change them in the repository.

When a copy of the folders and files is created, the developer must give it a name. For example, assume a repository is initialized and the default branch name is *main*. A developer that is charged with creating an enhancement to the code would not want to start making changes to the code in the *main* branch because the *main* branch could not be used to fix bugs that might have been discovered after the enhancements were in process. Instead, the developer would make a new branch called something like *enhancement* from the existing *main* branch then make changes to the files in the *enhancement* branch and not touching any of the files in the *main* branch.

3.3.3. Merging

Normally, when a developer has to make changes to some existing code, the developer will do so in a new branch which starts as a copy of the original branch. The developer may add new code, change existing code, and/or delete code in the new branch. The new branch code will be tested and, when it is deemed ready, needs to be merged back into the original branch. This process is called a *merge*. The merge will copy any new or changed files from the new branch back into the original branch. Any files that have been deleted in the new branch will be deleted in the original branch. Any files not changed in the new branch are ignored.

When more than one developer is working on files from the original branch, each developer will be created their own branch and may make changes to the same file. When that occurs, the possibility of a conflict when multiple developers attempt to merge their new branches back into the same original branch. When this problem occurs, it is called a *merge conflict*. Here is an example of a merge conflict.

The main branch contents:

```
// main branch
// index.html
<h1>Production: v1.0</h1>
```

The main branch has only a single file named *index.html* and it contains a single H1 tag showing that it is the production version 1.0.

Assume dev1 is asked to modify the code to version 2.0. Dev1 will create a new branch called *enh* which will contain a copy of the main branch.

```
// Dev1: enh branch
// index.html
<h1>Production: v1.0</h1>
```

Dev1 makes the required changes to the index.html file.

```
// Dev1: enh branch
// index.html
<h1>Production: v1.0 > Enh: v2.0</h1>
```

At the same time, Dev2 has been asked to make a bug fix to the code in the main branch. So Dev2 creates a branch called *bugfix* from the main branch.

```
// Dev2: bugfix branch
// index.html
<h1>Production: v1.0</h1>
```

Dev2 makes the required changes to fix the bug and tests in the *bugfix* branch.

```
// Dev2: bugfix branch
// index.html
<h1>Production: v1.0 > Bugfix: v1.1</h1>
```

Because the *bugfix* changes are completed more quickly, Dev2 decides to merge his branch back into the main branch. This merge completes successfully and the code in the main branch looks like this:

```
// main branch
// index.html
<h1>Production: v1.0 > Bugfix: v1.1</h1>
```

After this merge, Dev1 completes the enhancements and test. Dev1 then decides to merge the changes back into the main branch by doing a merge. The result is a merge conflict because the index.html file has been changed since Dev1 copied it from the main branch. If the merge of Dev1's code were applied to index.html, the code from Dev2 would be overwritten and lost. Git knows this because it keeps track of the date and time on the origin file. Recognizing that the file has been changed since Dev1 copied it, Git does not do the merge but show Dev1 there is a conflict. The index.html file is modified by Git and looks like this:

```
// main branch
// index.html

<<<<< Current
<h1>Production: v1.0 > Bugfix: v1.1</h1>
=====
<h1>Production: v1.0 > Enh: v2.0</h1>
>>>>> Local
```

Since Git cannot merge automatically, it displays the conflict of what the main branch file looks like and what the merge is trying to change it to. Both are displayed and it is up to Dev1 to manually modify the code to include both changes. Dev1 decides to keep the previous changes of Dev2 and add Dev1 changes, the code is modified like this:

```
// main branch
// index.html

<h1>Production: v1.0 > Bugfix: v1.1 > Enh: v2.0</h1>
```

Dev1 manually changes the code so that it contains both Dev2 and Dev1's code. Then Dev1 would stage the file and commit is to the main branch to complete the process.

3.3. Git Commands

Git is a command-line interface (CLI) program. Meaning you execute Git command by typing them into a shell program like "cmd" (Windows) or "terminal" (Linux/MAC) and press "enter" or "return". The command *may* display a response message showing the results of the command or an error if it occurred. Some Git commands do not show a message if the command is successful.

Here is the syntax of Git commands:

```
Git format:

     git [command] [options]

     - [command] is the function to execute
     - [options] varies with the command.

     ex: git commit -m 'this is a message'
```

Once Git is installed, the name of the program to execute is "git" which is in all lowercase characters followed by the command and, optionally, the options for the command. At least one space is required between "git", the command, and each of the options.

Git is a case-sensitive program meaning the program name, the commands, and the options must be entered in the correct case. Trying to execute "git Status" rather then "git status" will not work.

Some options require string data to be entered. It is best to surround the string with double quotes, but single quotes can be used. Everything within the quotes is part of the string including any spaces.

It is important that when entering a string, the trailing quote much match the starting quote. If the starting quote is a double but the trailing quote is a single, the system will assume the trailing quote is part of the string and will continue looking for the trailing double quote.

Typical Workflow

```
// initialize a repository (just once)
git init

// link to a remote repository
git remote add origin https://github.com/developer/my-repo.git

// stage and commit files multiple times in small batches
git add [filename(s)]
git commit -m [message]

// push commits to remote repository (occasionally)
git push -u origin main
```

3.3.1. *config*

The *config* command contains values that set up Git for operations by setting default values for a single repository or for all repositories. For example, it is used to define the default text editor to use. One setup that must be done before commits can be donein a repository is to set up the default user's name and user's email. These values are used to identify who created the commit. This is especially important when a repository is being shared by multiple developers.

If the configuration is for all repositories, the commands are:

```
git config --global user.name "Jane Doe"
git config --global user.email "janedoe@gmail.com"
```

If the configuration is for a single repository, the commands are:

```
git config user.name "Jane Doe"
git config user.email "janedoe@gmail.com"
```

Again, these parameters must be set before the first commit.

You can view all the configuration data by issuing this command:

```
git config { --global }} --list
```

3.3.2. *Init*

The *init* command will initialize a folder as a Git repository by creating the *.git* hidden folder within the current folder. Init must be done before any other Git commands can be executed. The format of the command it:

```
git init
```

If a new subfolder is created but no files are added to the subfolder, Git will not add the empty folder to the repository until at least one file is added.

When a folder is initialized as a repository, every file and folder is automatically part of the repository.

Subfolders within a repository should NEVER be initialized as a new repository.

If a subfolder was accidently initialized as a repository, the correction is to delete the *.git* hidden folder from the subfolder.

3.3.3. *status*

The *status* command displays information for the files in the workspace. A file is always in one of three states: 'untracked', 'staged', or 'committed'.

When a new file is created in a repository and *status* is executed, the new file will be displayed as an untracked file. This means that Git knows the file exists in the repository but will ignore the file until the file is staged. For how to stage a file, see the *add* command.

After the new file is staged, when *status* is run, the file will show as staged. A staged file is on that will be saved to the repository when the next *commit* is done.

Once the file is staged and after a commit is done, the file is stored in the repository and the file is *committed*. This file does not show in the *status* display because the file in the workspace is identical to that file in the repository.

```
git status
git status -s // short display
git status -u // displays untracked files
```

3.3.4. add

The add command stages one or more files. This staging takes a snapshot of the file contents when the *add* command is executed. The file can be changed after the *add* is executed but, unless another *add* command is executed, the snapshot of the file when the last *add* command was executed will be what gets put into the repository by the next commit.

There are multiple ways to select the files to be staged. One or more individual file names can be listed. Also, file specs can be use. File specs have partial file names along with an asterisk for the remaining file name. For example, to stage all the files that end in ".js", the file spec would be "*.js". Multiple file specs can also be included in a single *add* command.

The most used method of staging all the changed files is to use the period. This command stages all files in all folders easily.

```
// stages only the one file
git add [file]

// stages two files in folders
git add src/[file] src/comp/[file]

// stages all files ending in .js or .css
git add *.js *.css"

// stages all untracked or modified files in all folders

git add .
```

3.3.5. commit

The *commit* command is how *all* staged files are added or updated in the repository. The only way to control which staged files get updated in the repository is to limit the files that are staged before the *commit* is executed.

Each commit includes a number of information items:

- A unique 41-character identifier.
- The branch that stored the file.
- The author's name and email.

- The current date and time the commit was done.
- The message attached to the commit.

A message is required when executing the *commit*. This message is to detail what changes in which files have occurred with this commit. The first line of the message should be short and no more than 50 characters. Consider it a summary of the commit. To add additional details such as info on individual files, follow the summary with a blank line followed by the more detailed message lines. Here is an example:

```
Fixed issue #3838297: No output from program


* index.html changed to check for invalid data
* main.js added additional error checking
* output.js checked for error before outputting data
```

The first line is a short summary while the last three lines are more detailed about the three files that were updated in this commit.

Here is the typical commit command:

```
git commit -m "initial commit"
```

If the "-m [message]" is omitted, Git will use the default text editor to display text that can be modified by the developer and, when the file is saved, it will be used by Git as the message for the commit.

If a commit is executed and the developer would like to add more files to the commit instead of creating another commit, the previous commit can be *amended* with additional files. After the additional files are staged, the *commit* is executed again but with the "-a" switch:

```
git commit -a -m "initial commit (amended)"
```

This will merge all the files into a single commit.

Note: Amending a commit is NOT recommended if another developer has used the commit before it has been amended. The amended files would not be included in the other developer's code and could very well cause problems for that developer.

3.3.6. remote

The *remote* command creates a link between the local repository and some other repository which is usually in the cloud. Making this link allows copying commits from either repository to the other (see the *push / pull* command). Any number of links can be created as long as the short names are unique for each URL. The short names may contain dashes but may not contain blanks.

To make the link requires the URL of the repository to be connected to. For example, if the new repository is in GitHub, the URL will end with ".git".

When creating a link to another repository requires creating a short name to represent the longer URL. Usually, when the first remote is created, it is common to make the short name "origin" but any short name is allowed.

Here is the syntax for the command:

```
// syntax of the remote command
git remote add [short name] [URL]

// adds a URL to the local repository
git remote add origin https://github.com/programmer/my-repo.git

// removes a URL from the repository
git remote remove old-repo

// displays all the linked URLs
git remote -v
```

3.3.7. push / pull

The *push* and *pull* commands will copy commits from one repository to the other.

The *push* command will copy the commits that exist only in the local repository to the remote repository. The commits in the local repository that are already in the remote repository will not be copied.

The *pull* command will copy the commits that exist only in the remote repository to the local repository. The commits in the remote repository that are already in the local repository will not be copied.

As a way to remember which way the commits are copied, think of the local repository as the main repository (actually both repositories are the same priority). From the local repository, commits are *pushed* to the remote repository and *pulled* from the remote repository to the local.

When *pushing* or *pulling*, it is important that the branch of the local repository is selected. If the commits are pushed, they will be from the current branch. If the commits are pulled, they will be placed in the current branch.

The branch used in the command designates the branch in the remote repository. If the command is a *push*, the branch is the name of the branch in the remote repository. If that branch name does not exist in the remote repository, it will be created. If the command is a *pull*, the commits will be copied from that branch in the remote repository.

Here is the syntax:

```
// the general syntax
git { push | pull } [short name] [branch]

// copies commits from the local repository to the remote
// repository and stores them in the main branch
git push origin main
```

```
// copies commits from the remote repository to the local
// repository and stores them in the main branch
git pull origin main
```

3.3.8. branch

Creating a branch

When a new branch is made, it makes an identical copy of an existing branch. The new branch is stored in the hidden folder (.git) in the Git repository with the name chosen. In the example above, creating the *enhancement* branch from the *main* branch creates a copy of everything in the *main* branch and stored the copy in the new *enhancement* branch. Only one branch can be selected at a time so with the creation of the *enhancement* branch, it becomes the *current* branch.

To create a new branch, the branch to be copied must be the current branch. Then the command to create the branch is done this way:

```
git branch enhancement
```

This will copy all the folders and files from the current branch into the new *enhancement* branch. With this command, the current branch will remain current.

Switching branches

Switching branches is easy to do. To make the *enhancement* branch current, execute this command:

```
git switch enhancement
```

While this is easy, it is important that you know what is happening "under the covers" of Git.

The first thing that happens when switching from the *main* branch to the *enhancement* branch is that the folders and files in the workspace from the main branch are copied to the *main* branch within the Git repository then are deleted from the workspace. Then Git copies all the folders and files from *enhancement* branch in the Git repository to the workspace.

When this switch, the developer looking at the folders and files in the workspace may see some files disappear, some new files appear, and some existing files change content.

It is important to remember that if files are changed in the workspace but not committed to the repository, Git will not allow the switch to another branch. That's because if the switch was made, any changes in the workspace would be lost because Git will delete all the workspace files before restoring the files in the new branch.

Deleting branches

Branches that are no longer needed should be deleted. The command is:

```
git branch {-d |-D} [branch]
```

Of course, the branch to be deleted cannot be the current branch.

Normally, the "-d" switch is used when all the changes to files in the branch have been committed. If any file changes have not be committed, Git will refuse to delete the branch because the changes would be lost. If the developer is certain that losing the uncommitted changes is ok, the "-D" switch will force the delete.

3.3.9. merge

The merge command copies new or changed files from one local branch to the current local branch. If files have been deleted in a branch, they are deleted from the current branch. When the *merge* is successful, a new commit will be added to the current branch.

Once one branch is merged into another branch, the source branch is no longer needed and may be deleted.

If one or more files in the current branch have been changed since the new branch was created, attempting to merge will cause a *merge conflict* (see *Merging branches*).

Here is the syntax:

```
git merge [branch]
```

It is important that the branch to merge into is the current branch.

3.3.10. reset

The *reset* command is used to modify the working tree (the collection of commits) and changes the most current commit to a previous commit. It is used when one or more of the existing commits is a problem. *reset* will remove one or more of the most current commits from the working tree. Here is a scenario which could be resolved by doing a *reset*.

Suppose a developer creates an enhancement to the code in a repository. The changed code is tested and merged back into the original code base which causes a commit to be added to the original branch working tree of commits. But when the updated branch is deployed to production, a significant bug is discovered which causes the code to fail. After a quick look at the new code, the bug is discovered but fixing the bug will be a significant effort that will take some time. A temporary solution is to revert the code back before the new code was added. This is done by doing a *reset*.

Assuming the commit with the bug is in commit with an id of 1234 and the previous commit is id 1233. The reset command to execute would look like this:

```
git reset --mixed 1233
```

By executing this *reset*, commit 1234 is removed from the working tree and commit 1233 becomes the most current commit. The code is copied from commit 1233 and redeployed to users can continue working with the code but without the enhancements attempted in commit 1234. The files that were changed in commit 1234 still exist on the developers machine and can be used to fix the bug.

Here is the syntax:

```
git reset  [mode] [commit id]
```

The [commit id] is always the entire 41 characters that make up the commit id. To display the commit id, execute the *log* command.

--mixed

This is the default. When --mixed is used, the files from the commit(s) that are removed remain in the workspace (on disk) but are changed to untracked.

--soft

When --soft is used, the files from the commit(s) that are removed remain in the workspace and remain staged.

--hard

When --hard is used, the files from the commit(s) that are removed are also removed from the workspace. Files in the workspace that are not in the removed commits are untouched.

3.3.11. log

The log command displays commits starting with most recent and displaying the older commits in descending order. A typical commit display looks like this:

```
commit 2d6ff62cee840906f337595eee5535d051ec866e (HEAD -> main)
Author: developer <developer@gmail.com>
Date:   Sun Feb 4 11:53:50 2024 -0500


    added file.txt
```

This displays the entire commit id and the branch (main) that contains the most current commit. The author's name and email are displayed along with the data and time of the commit. Finally, the message of the commit is displayed.

The syntax is:

```
// display all commits
git log

// displays only the most current three commits
git log -3

// displays only most current three commits
// showing only the last 7 characters of the commit id
git log -3 --oneline
```

3.3.12. diff

The *diff* command displays what is different between two or more objects in the git workspace (what is current on disk) and/or commits.

One thing to know about the Git display is that it will only shows adds (green) and deletes (red). If a statement has been updated, it will display as a delete followed by an add.

Here is an example of the output from the command *git diff* after a file has been updated in the workspace:

```
$ git diff
diff --git a/index.css b/index.css
index e69de29..fa3b867 100644
--- a/index.css
+++ b/index.css
@@ -0,0 +1,4 @@
+body {
+    color: red;
+    font-family: Arial, Helvetica, sans-serif;
+}
```

The displayed text shows the changes that will be added to the repository if the file is staged and committed.

3.3.12.1. Differences between a commit and unstaged files

```
git diff {commit}
```

One of the most common uses for the *diff* command is to display the difference between a commit and the unstaged files in the workspace. If not commit id is included, HEAD is used automatically. By executing *git diff*, Git will display the changes to the file contents with green code being added and red code being deleted.

3.3.12.2. Differences between the last commit and staged files

```
git diff --cached
```

This *diff* command is similar to the previous command, but it displays the difference between the last commit and staged files.

3.3.13. checkout & switch

The *checkout* command has two purposes: to switch from one branch to another and to create a new branch and immediately switch to the new branch.

The *switch* command will simply switch branches.

3.3.13.1. Switching to a branch

Using *checkout* or *switch* for switching from one branch to another is the most common use. The command is:

```
git checkout existing-branch

git switch existing-branch
```

This changes the current branch to the *existing-branch*. This can only be done if all files are committed or have been stashed.

3.3.13.2. Creating and switching to a new branch

Another use for *checkout* is to create a new branch and switch to it.

```
git checkout -b new-branch-name
```

This version of the command will create the new branch and switch to that branch.

3.3.14. clone

The *clone* command will copy a repository making a new repository on the local disk. The command is:

```
git clone https://github.com/developer/my-repo.git
```

This will copy the main or master branch and create a new repository that includes the branch that was copied. If the remote repository has other branches, they can be copied down by executing the command:

```
git checkout [remotebranchname]
```

3.3.15. stash

The *stash* command copies all the changed but unstaged files in the current branch and removes the changes from those files resetting them to the state before the changes were made. The changes are saved together in a temporary list inside Git.

```
git stash
```

The *stash pop* command will take the stashed changes and reapply them to the files in the current branch.

```
git stash pop
```

One use of this command is when file changes are made in the wrong branch.

Assume a repository has a *main* branch and a *dev* branch and a developer intended to make changes in the *dev* branch but mistakenly made multiple changes in the *main* branch. To move the changes to the *dev* branch, the developer could execute:

```
git stash
```

in the *main* branch. This would store the changes and reset the files in that branch. Then the developer could switch to the *dev* branch and execute:

```
git stash pop
```

and all the changes would be applied to the files in the correct branch.

3.3.15. gitignore

The. gitignore is not a Git command. It is an optional, hidden text file that directs Git to ignore one or more files or folders from being added to the repository. For example, the developer may choose to not store a file containing a database connection string that includes a password to the database. By adding the file name to the *.gitignore* file, Git will never store that file in the repository. Assume the connection string is stored is a file called *conn-str.db*. In the .gitignore file, a new line would be added like this:

```
// .gitignore file

conn-str.db
```

Lines in the *.gitignore* can contain individual file names, file specs (i.e. *.exe), or folders (i.e. /bin) Next is an example of telling Git that any file that has an extension of *.token* should be ignored.

```
// .gitignore file
*.token
```

GitHub

GitHub is a repository in the cloud. Each repository has many more features including:

- Issues that can be used to log enhancements or bugs related to the code.
- Pull Requests that is used by developers that want code changed they made to be merged into other branches.

GitHub is free to use, but there are some features that can only be used with a paid account.

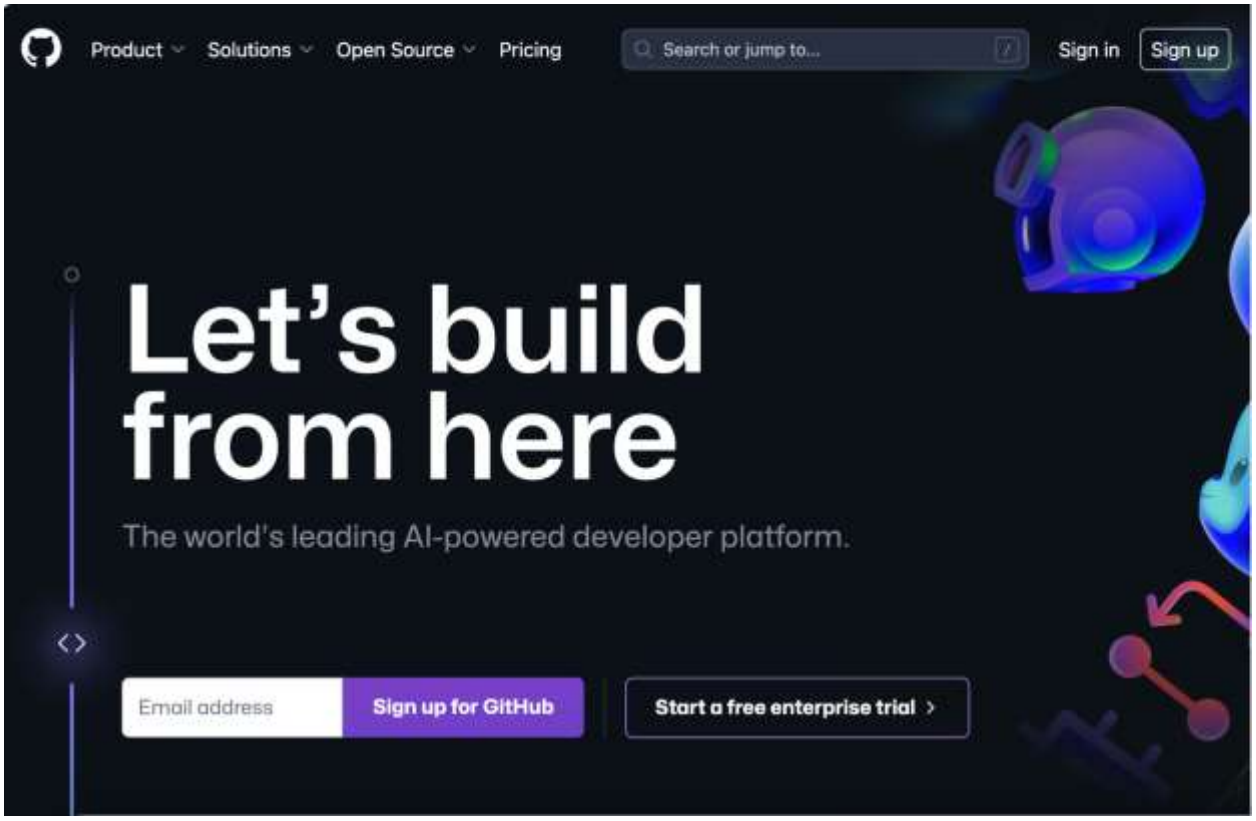
This course will show using GitHub as a cloud repository that can stored a copy of all the commits from the developer's local Git repository. This is a protection from losing the source code if the local workstation gets destroyed. If that happens and the GitHub repository contains a copy of the local repository, the GitHub repository can copy all the commits to a new workstation so that no source code will be lost.

GitHub is very compatible with Git. The biggest difference is that Git is a single user repository while GitHub can be used as a single user repository, but it can also support multiple developers using the same repository.

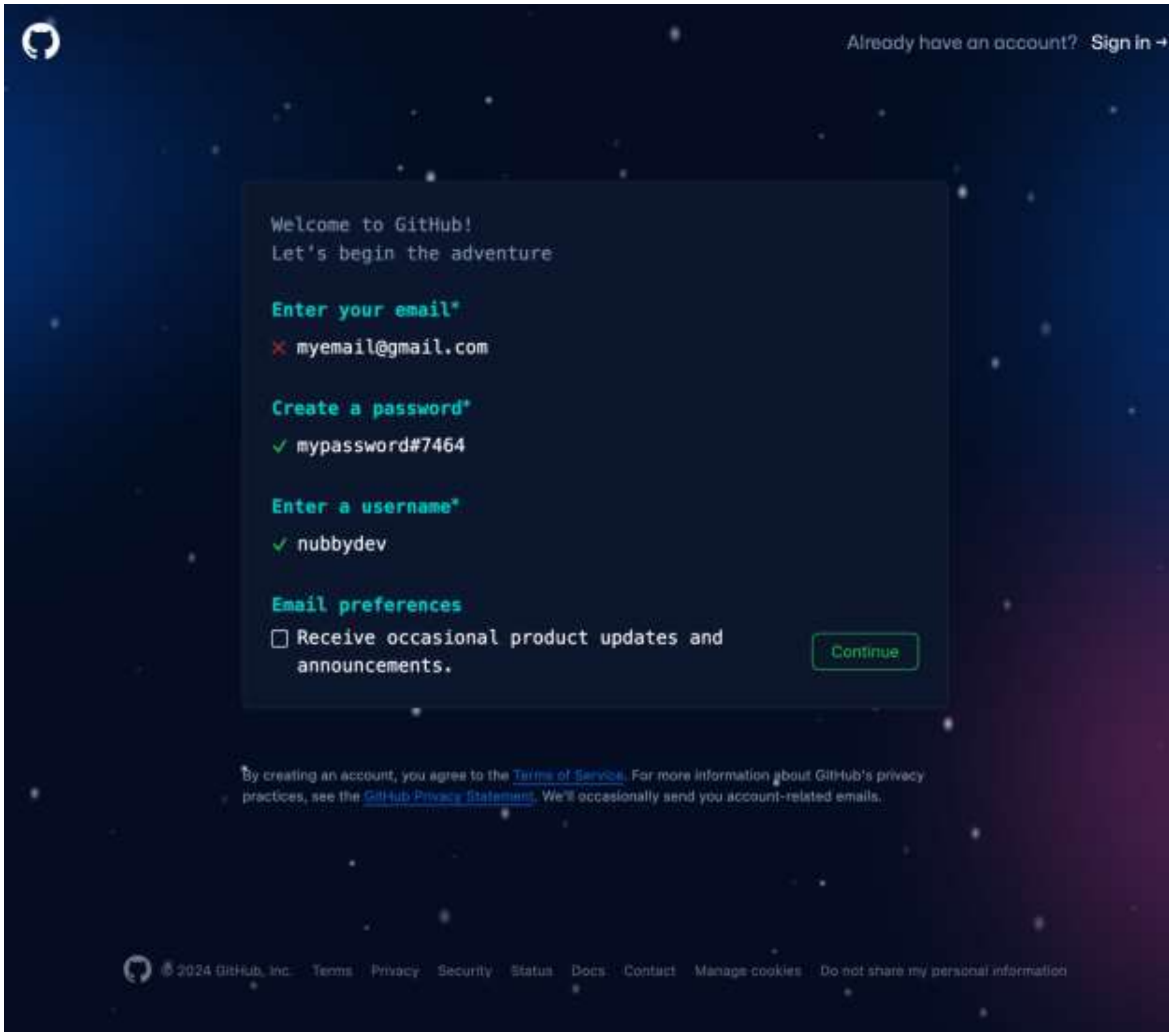
4.1. Creating a GitHub account

Creating an account in GitHub is easy and the basic plan is free. You can upgrade to a paid plan that provides additional features, but the basic plan is a great starting point.

To create a new GitHub Account, start by navigating to the GitHub.com home page.



In the upper right corner, click on *Sign up*.



Already have an account? Sign in →

Welcome to GitHub!
Let's begin the adventure

Enter your email*
✗ myemail@gmail.com

Create a password*
✓ mypassword#7464

Enter a username*
✓ nubbydev

Email preferences
☐ Receive occasional product updates and announcements.

By creating an account, you agree to the Terms of Service. For more information about GitHub's privacy practices, see the GitHub Privacy Statement. We'll occasionally send you account-related emails.

© 2024 GitHub, Inc. Terms Privacy Security Status Docs Contact Manage cookies Do not share my personal information

Right under *Enter your email*, you should enter an email address that you have access to because GitHub will be sending an email that you'll need to respond to. Then click *Continue*.

After your email, you need to create a password for accessing GitHub. It needs to be at least 8 characters that include both a lower case letter and a number or you can create a 15 character password with any characters you want. Then click *Continue*.

Next, you'll need to create a username. You'll login to GitHub with the username and password. Make sure the username you create is available. Then click *Continue*.

Then you can choose to receive emails from GitHub if you wish by checking the box. Then click *Continue*.

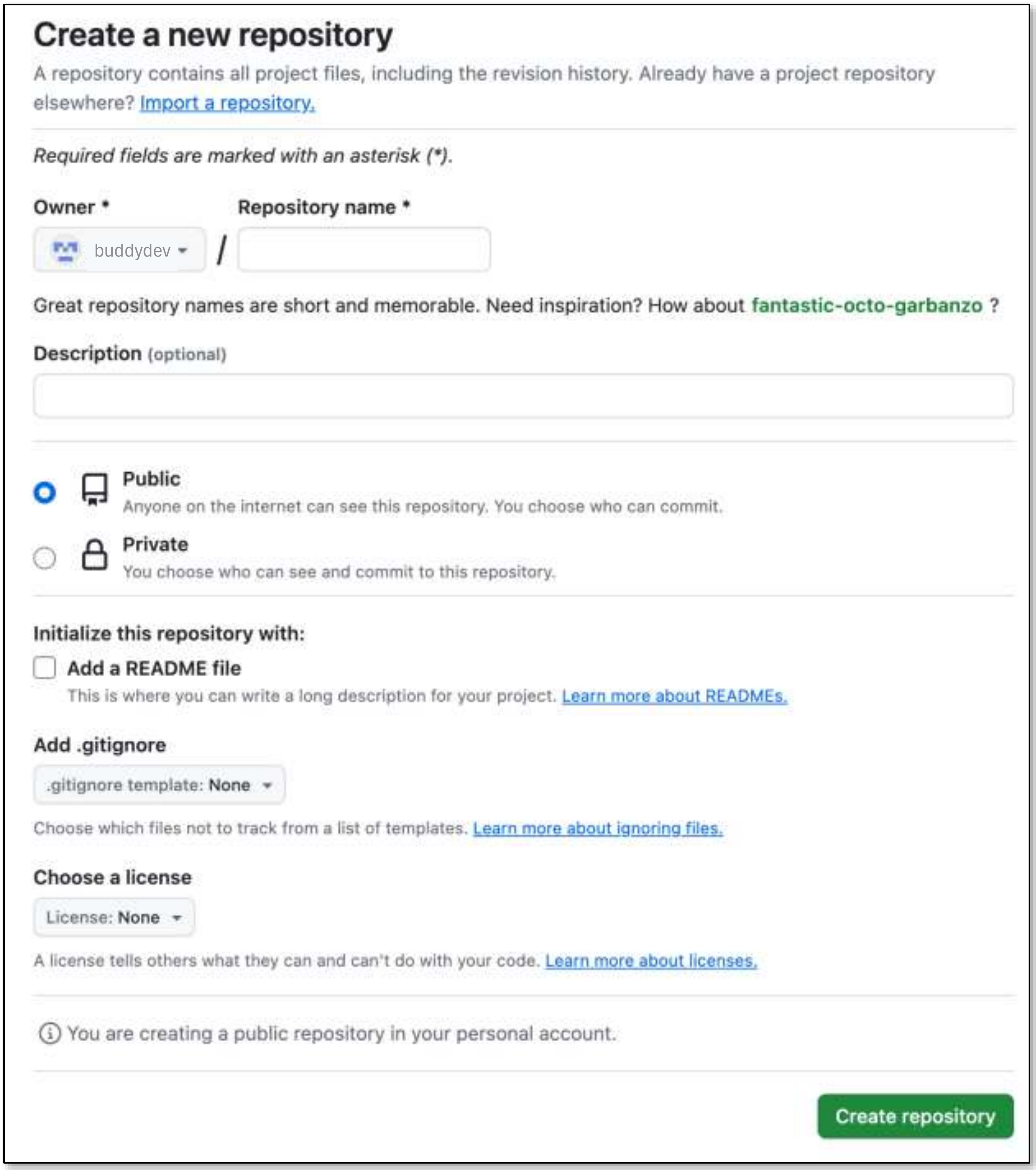
To verify that you are not a robot, you'll have to complete a task that involves orienting one image the same as another image.

The final step is GitHub will send an email to your email address with a number that must be entered into GitHub. Once that is completed successfully, your account will be created.

4.2. Creating a repository

Creating a repository is easy.

Start by clicking the plus sign button to the right of the search box and select *New Repository*.



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 *: buddydev / Repository name *:

Great repository names are short and memorable. Need inspiration? How about fantastic-octo-garbanzo ?

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.

The *Create New Repository* page displays.

In the *Repository Name* textbox, you must give the repository a name and it must be a unique name in your GitHub account.

The *Description* lets you provide some text that describes what the repository is about. It can be changed at any time.

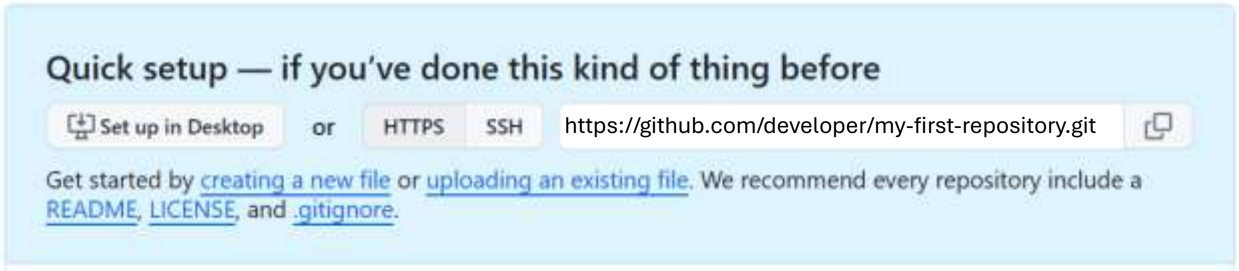
New repositories access defaults to *public*. That means that anyone can view and get a copy of the repository, but they cannot change the contents in any way. Setting the access to *private* makes the repository invisible to all others. If you are working on a project for another person or organization, you should make the access *private* because the code is likely the property of the person or organization.

The checkbox next to *Add a README file* should be left unchecked. A file with the name README.md is special within GitHub. Its contents of that file will be displayed when someone looks at the list of files within the repository. If you check this, a README.md file will be created but you will be required to do a *clone* or *pull* before you can *push* any commits to the repository. You can always create the README.md file on your local machine then push it to the repository.

The *Add .gitignore* drop-down will display a variety of code types. The *.gitignore* file will define files that will not be added to the repository. The file is a simple text file where each line is a single file name, a file spec (i.e. *.exe), or a folder (i.e. /bin). If the drop-down contains the code type you're using, select it. If not, you can create the *.gitignore* yourself in the project root.

The *Choose a license* drop-down lets you select the licensing you want for your code. If you want to control your code, you should select one of the license types or you can just leave it as *None*.

Click on the green button


Quick setup — if you've done this kind of thing before

https://github.com/developer/my-first-repository.git

Get started by creating a new file or uploading an existing file. We recommend every repository include a README, LICENSE, and .gitignore.

Next you'll see a page with this block. The https://github.com/developer/my-first-repository.git URL (yours will be different) points to your new repository in GitHub.

Click the double squares to the right of the URL. This will copy the URL to the clipboard.

Return to the Git Bash shell and the local repository to connect it with the new GitHub repository.

Execute this to make the connection:

```
git remote add origin https://github.com/developer/my-first-
repository.git
```

There is no message when it worked. To verify, execute this:

```
git remote -v

origin  https://github.com/developer/my-first-repository.git (fetch)
origin  https://github.com/developer/my-first-repository.git (push)
```

It displays the URL twice with the only difference being the text in the parentheses at the end. *Fetch* is for reading the repository and *push* is for updating the repository. By default, they are the same URL. But GitHub does allow them to be different.

4.3. Creating a Pull Request

While a GitHub repository can be used by a single developer, a valuable use it for multiple developers to share a single GitHub repository. Having multiple developers updating code requires a different code management process than when only a single developer is using it.

When a team is sharing the repository, there is typically a limited number of admin developers that can update the main branch. So, when any developer makes changes to the code, they will create a new branch from the main branch and make changes in the new branch. They'll test their code and when they feel like it is ready to merge into the main code branch, instead of doing that merge directly, they will push their branch into the GitHub repository then create a pull request.

A *Pull Request* is a request created by the developer to have their branch merged into the main branch. Most developers do not have the authority to update the main branch. Only those few admin developers who are responsible for the main branch have the authority to merge code into the main branch. Creating a pull request alerts the admin developers that a developer wants their branch to be merged into main. The admin developers can review the developers code and choose to merge it or reject is with comments about why they won't merge the current code. If the pull request is merged, all developers need to pull the updated main branch to their local workstations. If the pull request is rejected, the developer can update their code and push the changed code up to the same branch and ask the admin developers to reconsider merging.

Merge Conflict

When a developer makes changes to existing code, it is normal that the developer will create a new branch then make changes in that branch. When the code changes are complete and tested, the developer will want to merge the changed back into the original branch. When the repository is used by a single developer, there is little chance that any problems will occur. But when multiple developers are changing code, the possibility of a merge conflict is much more likely.

When a new branch is created, there is a timestamp of the last time the file was updated. When the attempt to merge the changed code back into the original branch, the timestamp on the changed files is compared to the timestamp on the files in the origin branch. If the timestamps are the same, the merge is done.

```
File name        Orig Timestamp   Dev Timestamp        Diff

============     ==============   =============        =====
index.html       202403091625     202403011430         No Problem
```

If the timestamps are different, it means that the files in the origin branch have been changed by another developer since the new branch was created.

```
File name        Orig Timestamp   Dev Timestamp        Diff

============     ==============   =============        =====
index.html       202403091625     202403011430         Conflict
```

If the original files have been updated by another branch, merging the current branch will overwrite the previous merge.

To address this situation, the code is annotated with changes from the original branch and the current branch.

```
// index.html with merge conflict
<html>
     <body>
          <<<<<<< HEAD
          <div></div>
          =======
          <span></span>
          >>>>>>> DEV
     </body>
</html>
```

In this simple example, the index.html file has had a single line changed by an update after the Dev branch was created and the one line was changed to a `<div></div>`. The developer changed that same line to a `<span></span>`. The lines a with the less than characters, greater than characters and the equal signs show the differences between the current branch and the new branch. The developer must change the code manually, making whatever changes are needed. Once those changes are complete, the developer must stage and commit the changes just like any normal change.

```
// index.html with merge conflict resolved
<html>
     <body>
          <div></div>
          <span></span>
     </body>
</html>
```

In this example, the developer decided to include both the origin code plus the new code.

5.1. Create and resolve a merge conflict

```
// *** create a new Git repos on the workstation ***

// create a new folder
$ mkdir /repos/merge-conflict-example

// change into the new folder
$ cd /repos/merge-conflict-example
```

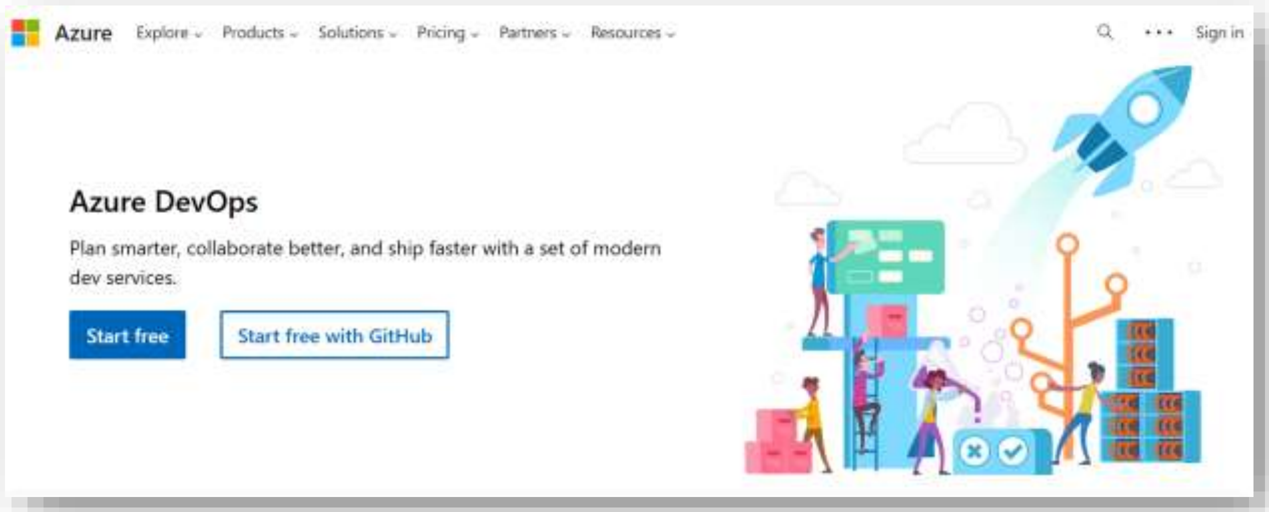
```
// make the new folder a Git repository
// main is current branch
$ git init

// create a single text file
$ echo “Prod v1.0” > source-file.code

// stage and commit the changes
$ git add .
$ git commit -m “initial commit”

// create a bugfix branch
$ git checkout -b bugfix

// bugfix branch is current
// change file to v1.1
$ echo “ | Bugfix v1.1” >> source-file.code

// stage and commit the changes
$ git add .
$ git commit -m “fixed bug to v1.1”

// switch back to main branch
$ git switch main

// main branch is current
// create enhancement branch
$ git checkout -b enhancement

// enhancement branch is current
// change file to v2.0
$ echo “ | Enhancement v2.0” >> source-file.code

// stage and commit the changes
$ git add .
$ git commit -m “enhancement to v2.0”

// switch back to main branch
$ git switch main

// main branch is current
// Merge bugfix into main
$ git merge bugfix

// merge is successful!
// Merge enhancement into main
$ git merge enhancement

// MERGE CONFLICT!!!
// fix the merge conflict so it
// includes both the bugfix and enhancement
$ echo “Prod 1.0 | Bugfix v1.1 | Enhancement v2.0” > source-file.code
```

```
// stage and commit the changes
$ git add .
$ git commit -m "merge conflict to v2.0"

// merge conflict resolved!
```

Azure DevOps

Azure DevOps provides a collection of services for the software development workflows. The focus of this section is on Azure Repos. This service is equivalent to GitHub.

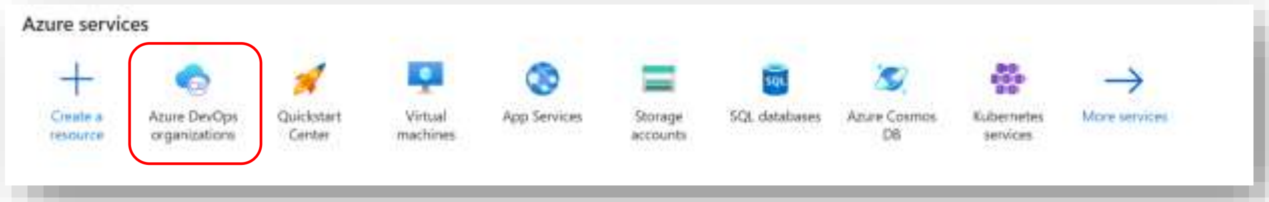
Start by navigating to http://dev.azure.com. This will take you to the main page for Azure DevOps.



Azure DevOps

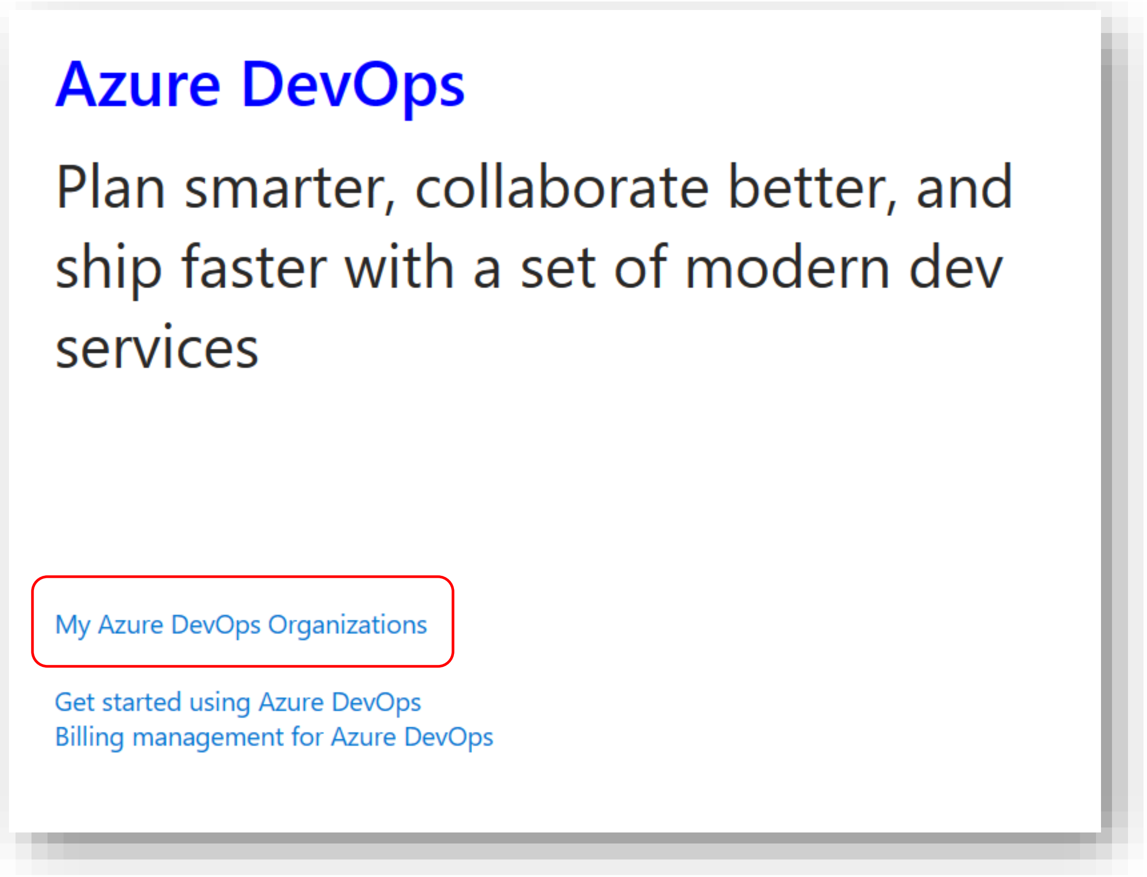
Plan smarter, collaborate better, and ship faster with a set of modern dev services.

You will need to sign in with a Microsoft account that provides privileges to Azure DevOps. The *Sign in* link is at the top right of the page.



6.1 Create an organization.

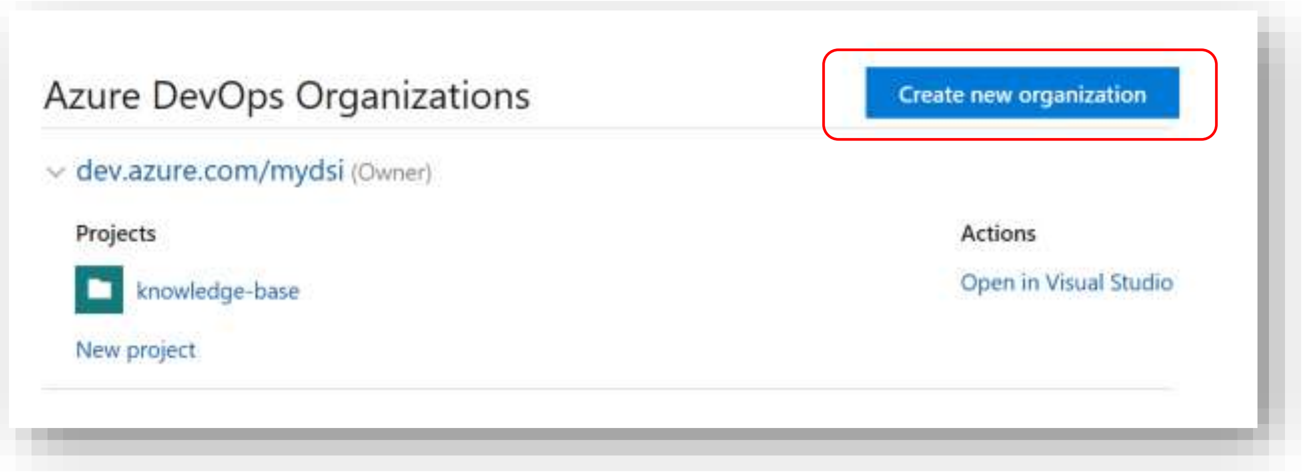
Once in the Azure portal, click "Azure DevOps organizations" icon.


Azure DevOps
Plan smarter, collaborate better, and ship faster with a set of modern dev services
My Azure DevOps Organizations
Get started using Azure DevOps
Billing management for Azure DevOps

To get started with Azure Repos, you need an organization.

Click “My Azure DevOps Organizations”

If you don’t already have an organization, you must create one. Click the “Create a new organization.”



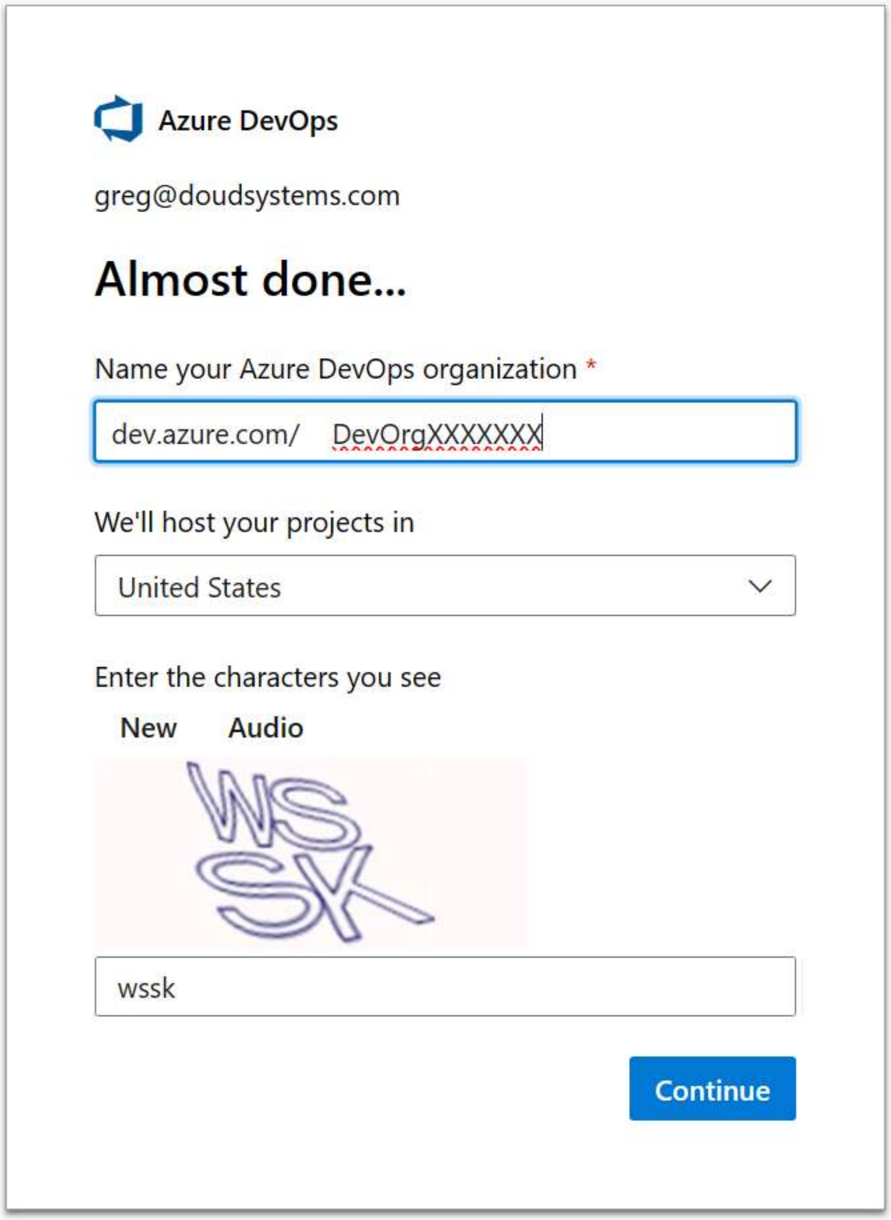
A new page will open to create an organization.

Start by creating a new organization name after the “dev.azure.com/”. Organization names must start with a letter or number, followed by letters, numbers, or hyphens, and must end with a letter or number.

Next, drop down the list of countries to host your projects. In this country, select “United States.”

Finally, you need to type the characters in the text box.

Click the “Continue” button.



greg@doudsystems.com

Almost done...

Name your Azure DevOps organization *

dev.azure.com/ DevOrgXXXXXXX

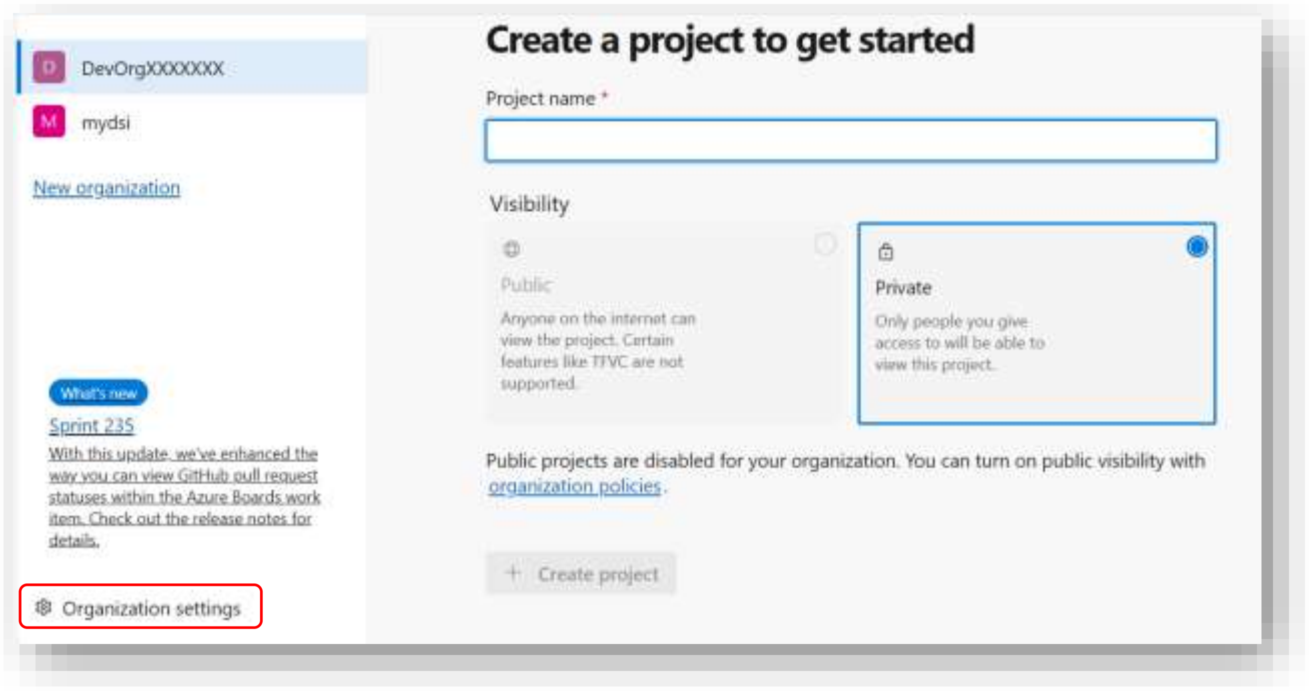
We'll host your projects in

United States

Enter the characters you see

wssk

When the creation of the organization is complete, you'll be taken to Azure DevOps and asked to create your first project. But first, we want to change a setting in your new organization that will allow you to create public repositories.



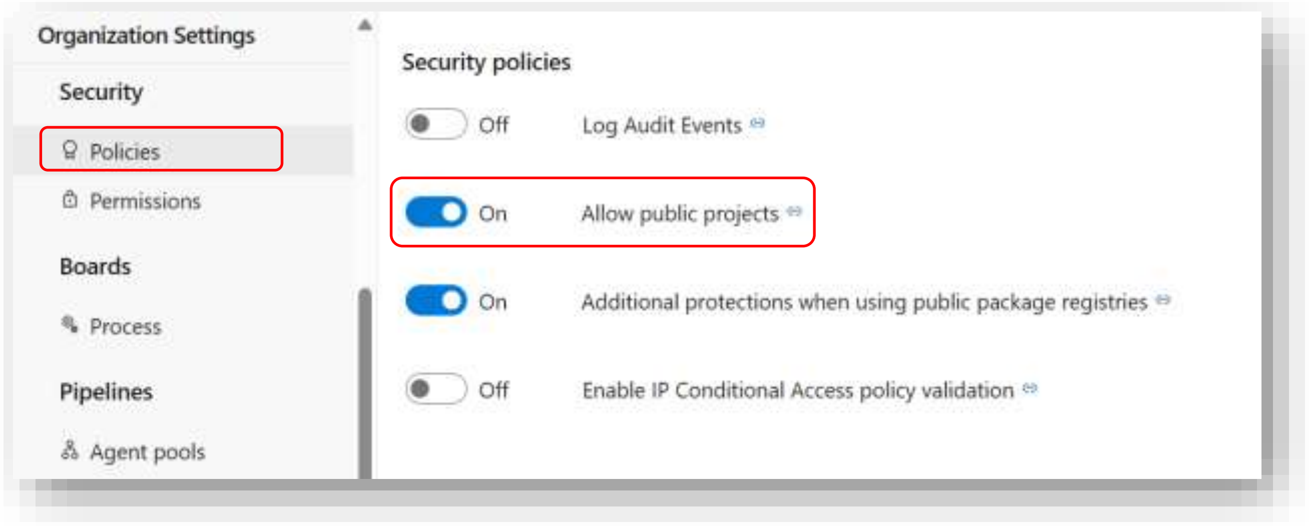
With this update, we've enhanced the way you can view GitHub pull request statuses within the Azure Boards work item. Check out the release notes for details.

Anyone on the internet can view the project. Certain features like TFVC are not supported.

Only people you give access to will be able to view this project.

Public projects are disabled for your organization. You can turn on public visibility with organization policies.

Click the "Organization settings" link at the bottom left.



Off Log Audit Events

On Allow public projects

On Additional protections when using public package registries

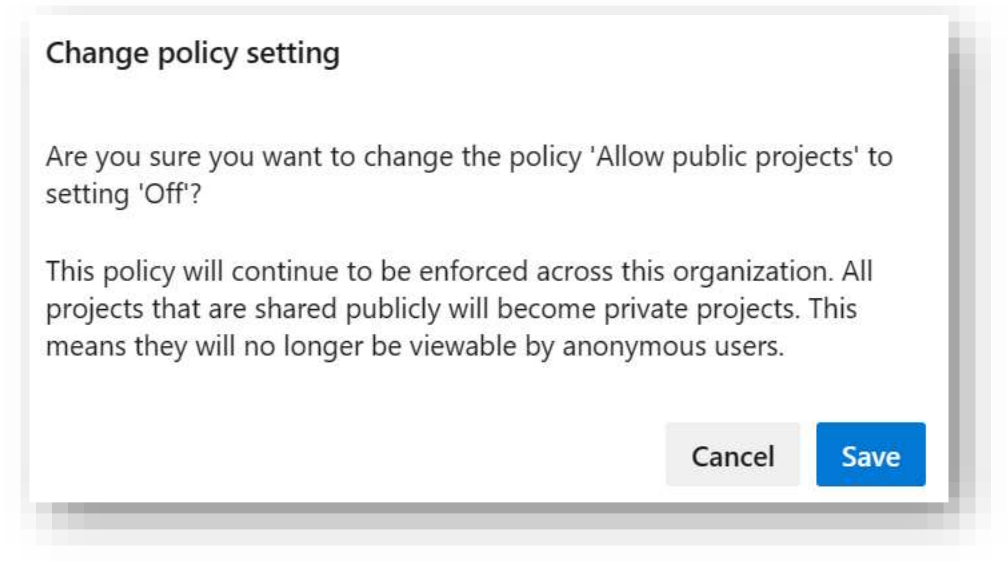
Off Enable IP Conditional Access policy validation

Scroll down the left sidebar and click on "Policies".

Then scroll down to the "Security Policies" and click the toggle button to "On" next to "Allow public projects". The "Change policy settings" dialog will display.

Then scroll down the left sidebar and click Repositories in the "Repos".

In the “All Repositories Settings” block, go to the “Default branch name for new repositories” and toggle the switch “On” and enter “main” in the texbox. This will name the default branch “main”.



Change policy setting

Are you sure you want to change the policy 'Allow public projects' to setting 'Off'?

This policy will continue to be enforced across this organization. All projects that are shared publicly will become private projects. This means they will no longer be viewable by anonymous users.

To affect the policy change, click the “Save” button.

To navigate back to the projects, click on the organization name next to “Azure DevOps”.

6.2. Create a project

It is simple to create a project in Azure DevOps.

Start by creating a name for the project in the “Project name” text box. The name should be unique within your repository.

Next, you can enter an optional description of the project or enter it later.

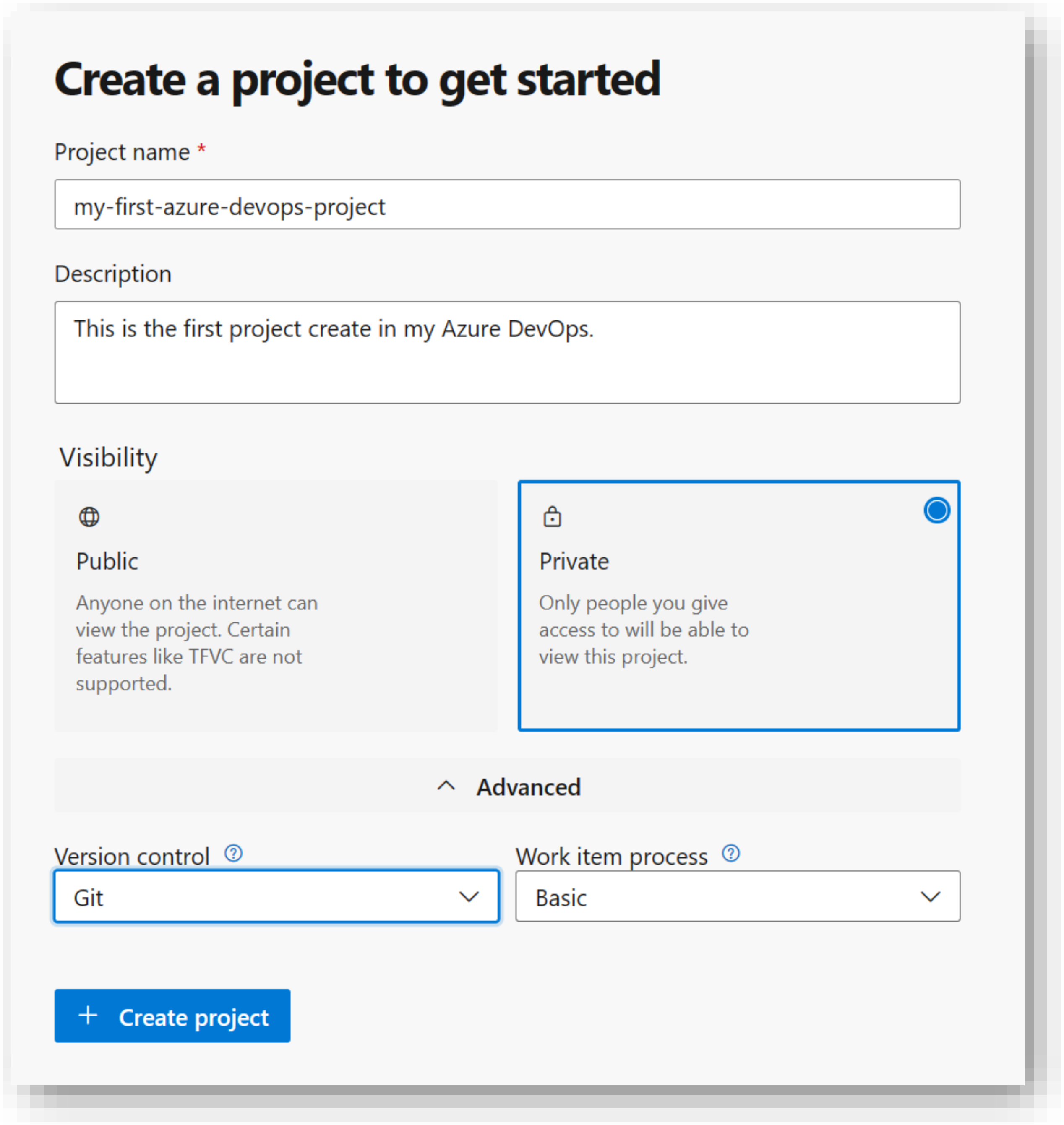
A project repository defaults to *private* meaning it is only available to the owner. If the policy change is made in the organization, the project can be *public* making it accessible to others.

In the “Advanced” drop-down you’ll find a drop-down for version control and work item process.

The “Version control” drop-down can select *Git* or *Team Foundation Version Control*. For this instruction, select *Git*.

In the “Work item process” drop-down, the options are: *Agile, Basic, CMMI, and Scrum*. For this instruction, select *Basic*.

Click the “Create project” button


Create a project to get started

Project name *

my-first-azure-devops-project

Description

This is the first project create in my Azure DevOps.

Visibility

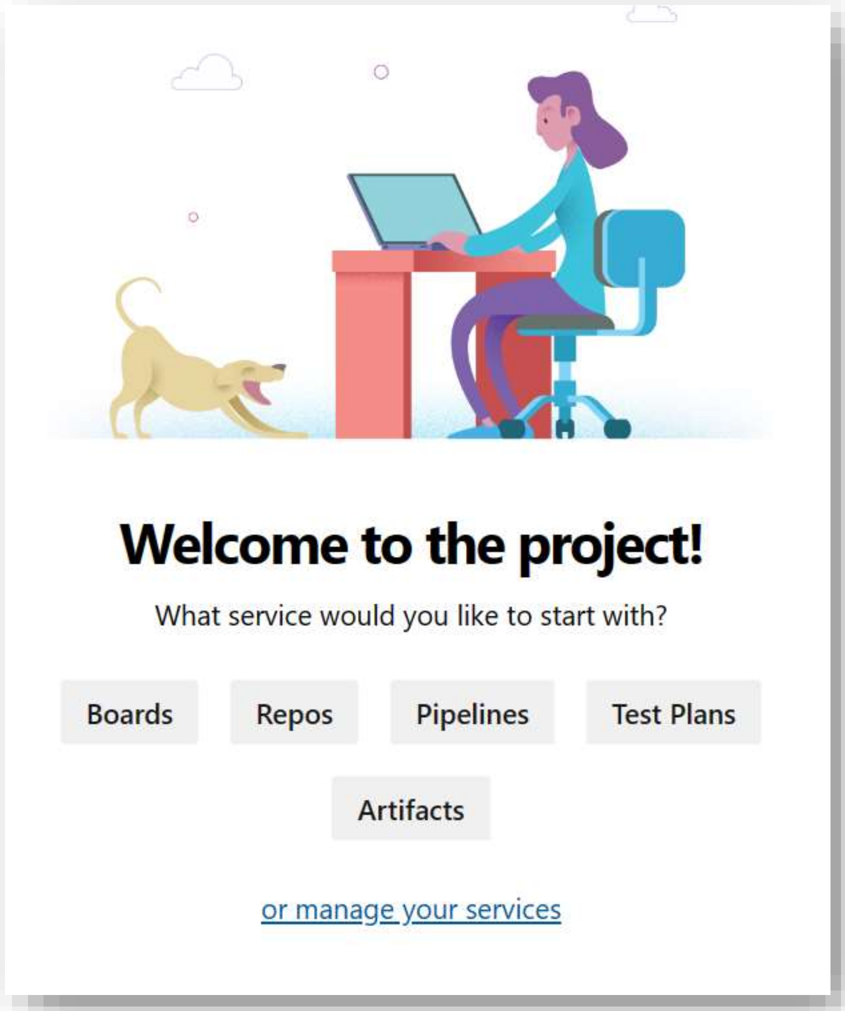
Public
Anyone on the internet can view the project. Certain features like TFVC are not supported.

[✓ Private]
Only people you give access to will be able to view this project.

Version control: Git

Work item process: Basic

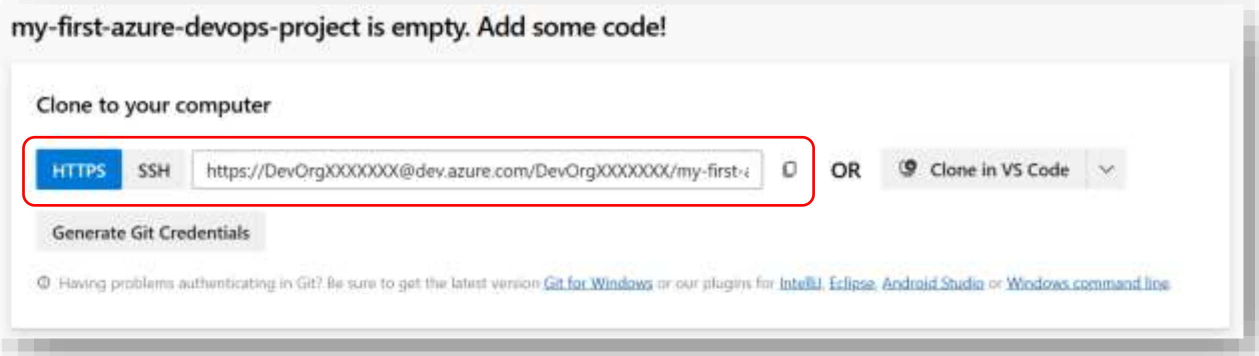
When your project is created, it will navigate to your project and you'll see a message "Welcome to the project!"


Welcome to the project!

What service would you like to start with?

In the left sidebar, click on “Repos” to begin managing your repository.

Inside the Repos category, clicking Files (if it is not already selected) will display the URL for your repository.


my-first-azure-devops-project is empty. Add some code!

Clone to your computer

https://DevOrgXXXXXXX@dev.azure.com/DevOrgXXXXXXX/my-first-

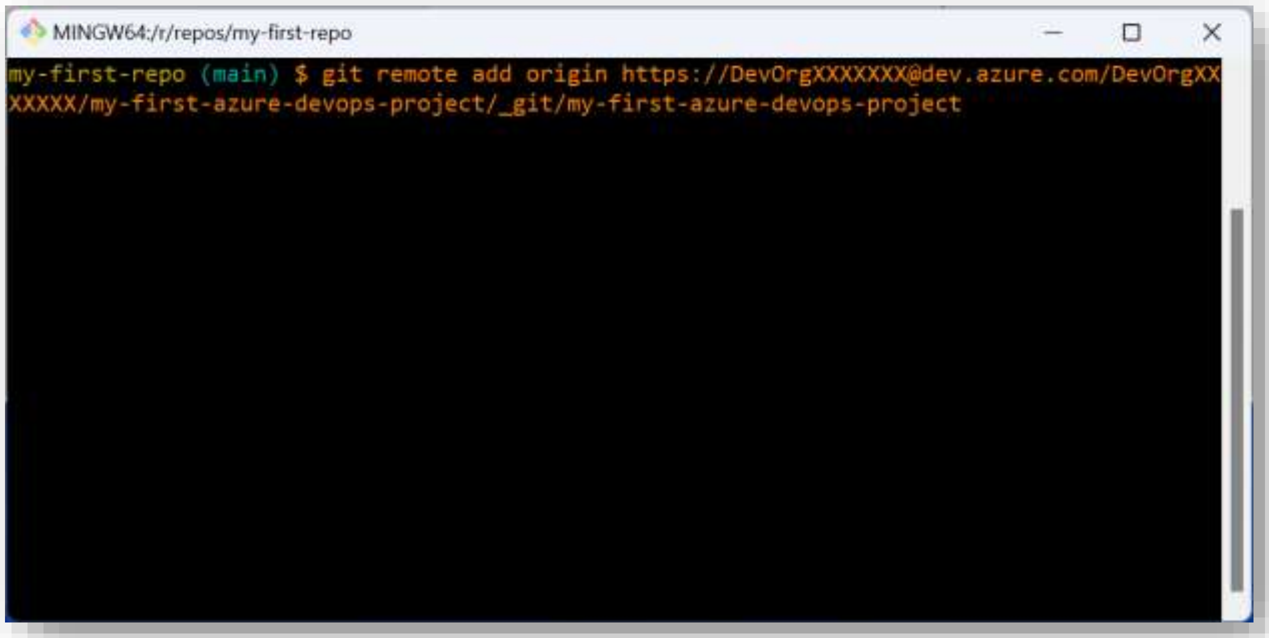
Having problems authenticating in Git? Be sure to get the latest version Git for Windows or our plugins for IntelliJ, Eclipse, Android Studio or Windows command line.

In the read-only textbox, there is the URL for the new repository. You should make sure that the HTTPS button is highlighted. If not, click it. Then click on the small document button just left of the word "OR". Clicking this button will copy the URL to the clipboard then it can be used to clone the repository or connect to an existing local workstation repository using the *remote* Git command

6.3. Attach to local repository

After copying the URL, go to the local Git repository and link the Azure DevOps repository to the local Git repository using the *remote* command. The command is:

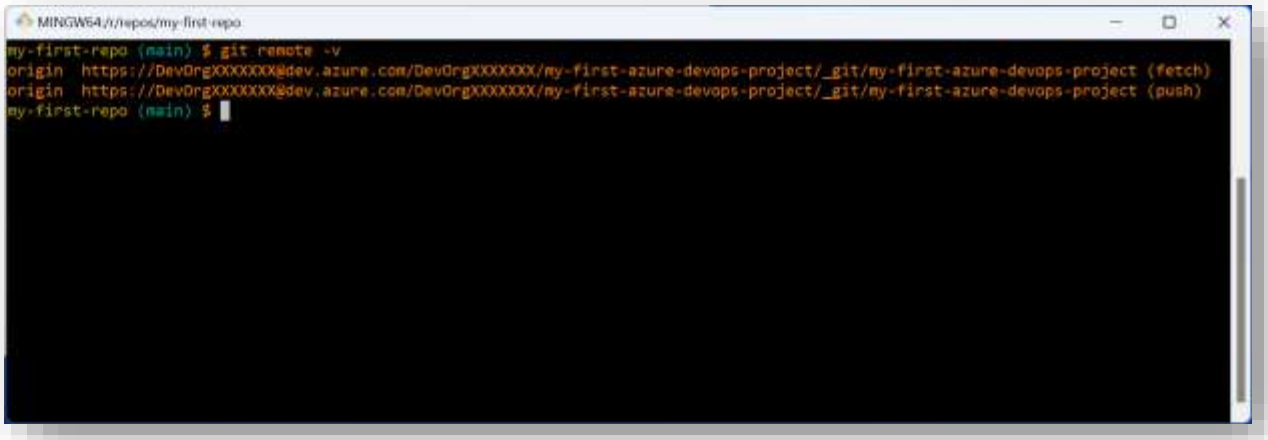
```
$ git remote add origin [URL]
```



```
my-first-repo (main) $ git remote add origin https://DevOrgXXXXXXX@dev.azure.com/DevOrgXX
XXXXX/my-first-azure-devops-project/_git/my-first-azure-devops-project
```

To verify that the remote repository was linked property, execute this command:

```
$ git remote -v
```

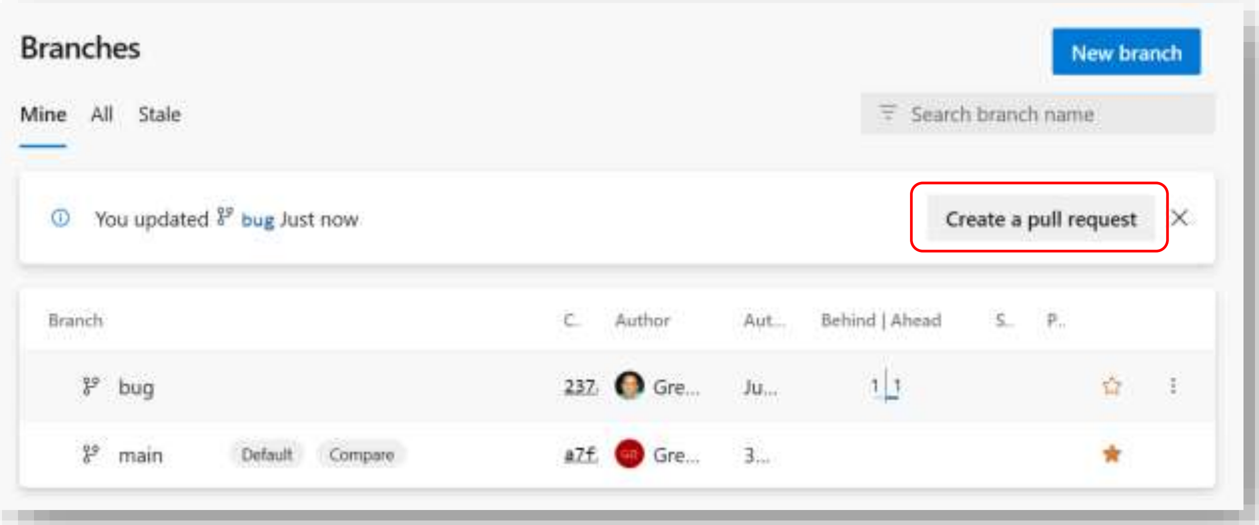


```
my-first-repo (main) $ git remote -v
origin  https://DevOrgXXXXXXX@dev.azure.com/DevOrgXXXXXXX/my-first-azure-devops-project/_git/my-first-azure-devops-project (fetch)
origin  https://DevOrgXXXXXXX@dev.azure.com/DevOrgXXXXXXX/my-first-azure-devops-project/_git/my-first-azure-devops-project (push)
my-first-repo (main) $
```

You should see two lines both with the same text except for the *fetch* or *push* in parentheses at the end.

6.4. Pull Request

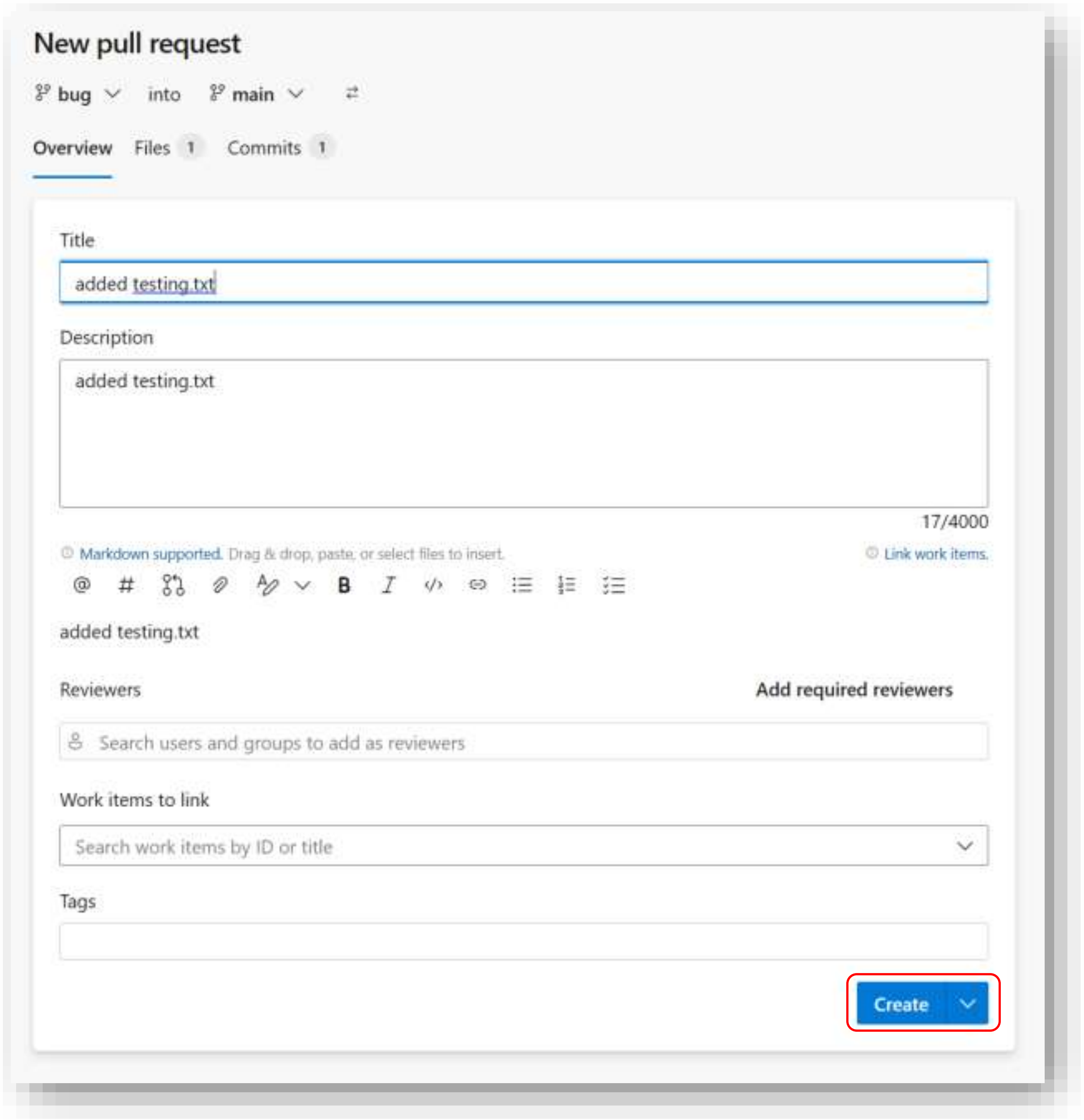
To create a *Pull Request* in ADO, click on "Branches" in the left sidebar and you'll see a page that looks like this



You updated bug Just now

Branch	C..	Author	Aut...	Behind | Ahead	S..	P..
bug	237.	Gre...	Ju...	1 | 1		
main Default Compare	a7f.	Gre...	3...			

Click on the "Create a pull request" button to display the *New pull request* page.


New pull request

bug into main

Overview Files 1 Commits 1

Title

added testing.txt

Description

added testing.txt

17/4000

Markdown supported. Drag & drop, paste, or select files to insert. Link work items.

added testing.txt

Reviewers Add required reviewers

Search users and groups to add as reviewers

Work items to link

Search work items by ID or title

Tags

Create

The ADO pull request page is a bit different from the GitHub pull request page.

The first detail to be verified is which branch is being merged and which branch is being merged into. At the top left, you see in the example that the “bug” branch is being merged into the “main” branch. The order of these branches is just the opposite of the order in GitHub. If the two branches are in the wrong order, click the double arrows button on the right.

The Overview tab is displayed above.

The *Title* is for providing a short describing of the pull request.

The *Description* is for a more detailed description which could include all the changes to all the files being merged.

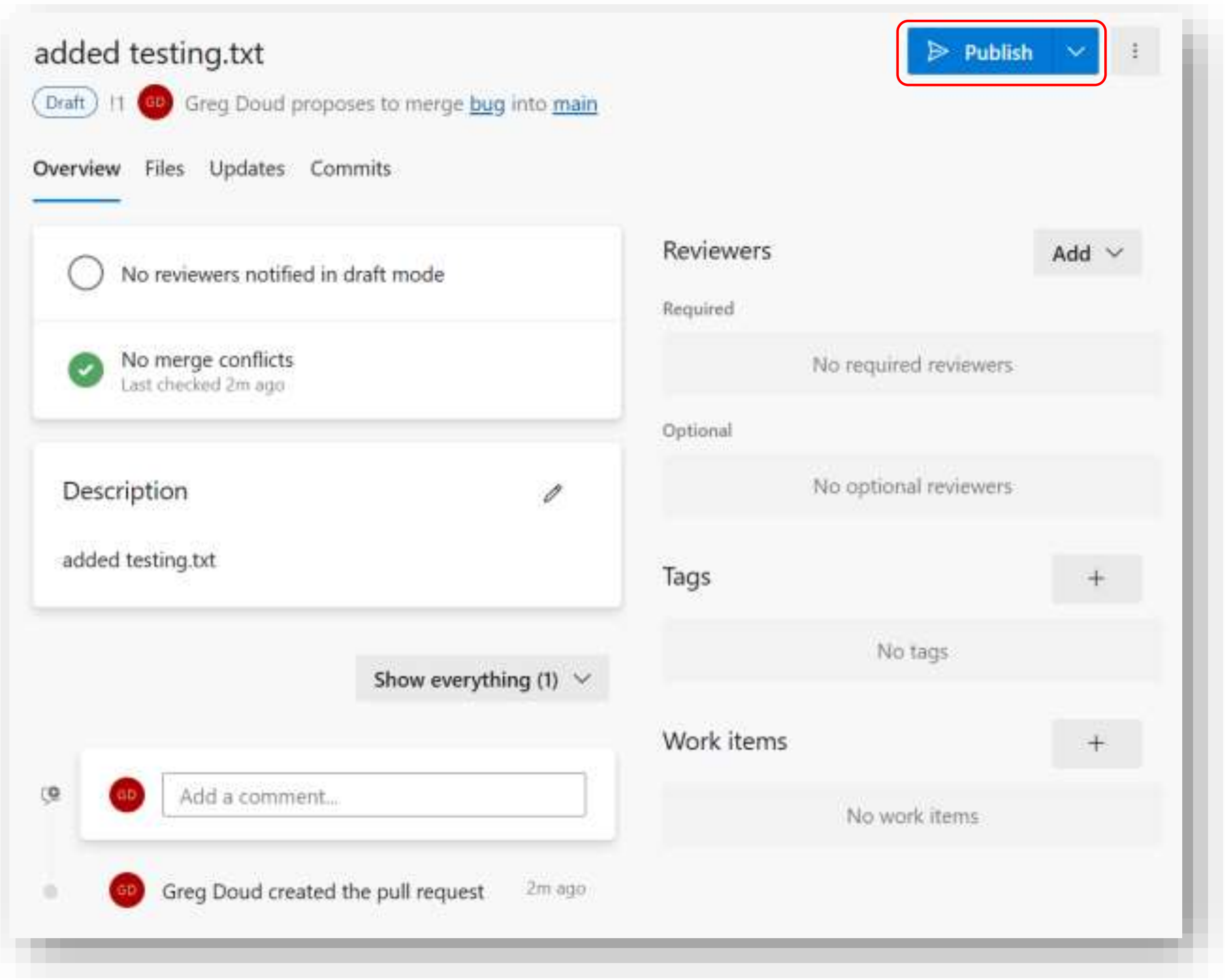
The optional *Reviewers* are those who must verify and approve of the pull request before it can be merged. They are not required.

The optional *Work items to link* can link this pull request to an existing work items board.

The optional *Tags* TBD.

When all data has been entered, click the “Create” button. This will create the pull request.

If the pull request data is not quite complete, click the down arrow and select “Create as draft”. This will save the data for the pull request as a draft but will not publish it. To publish is, select the “Pull requests” and the specific pull request. Make any changes needed and click the “Publish” button.


added testing.txt

Draft !1 GD Greg Doud proposes to merge bug into main

No reviewers notified in draft mode

No merge conflicts
Last checked 2m ago

Description

added testing.txt

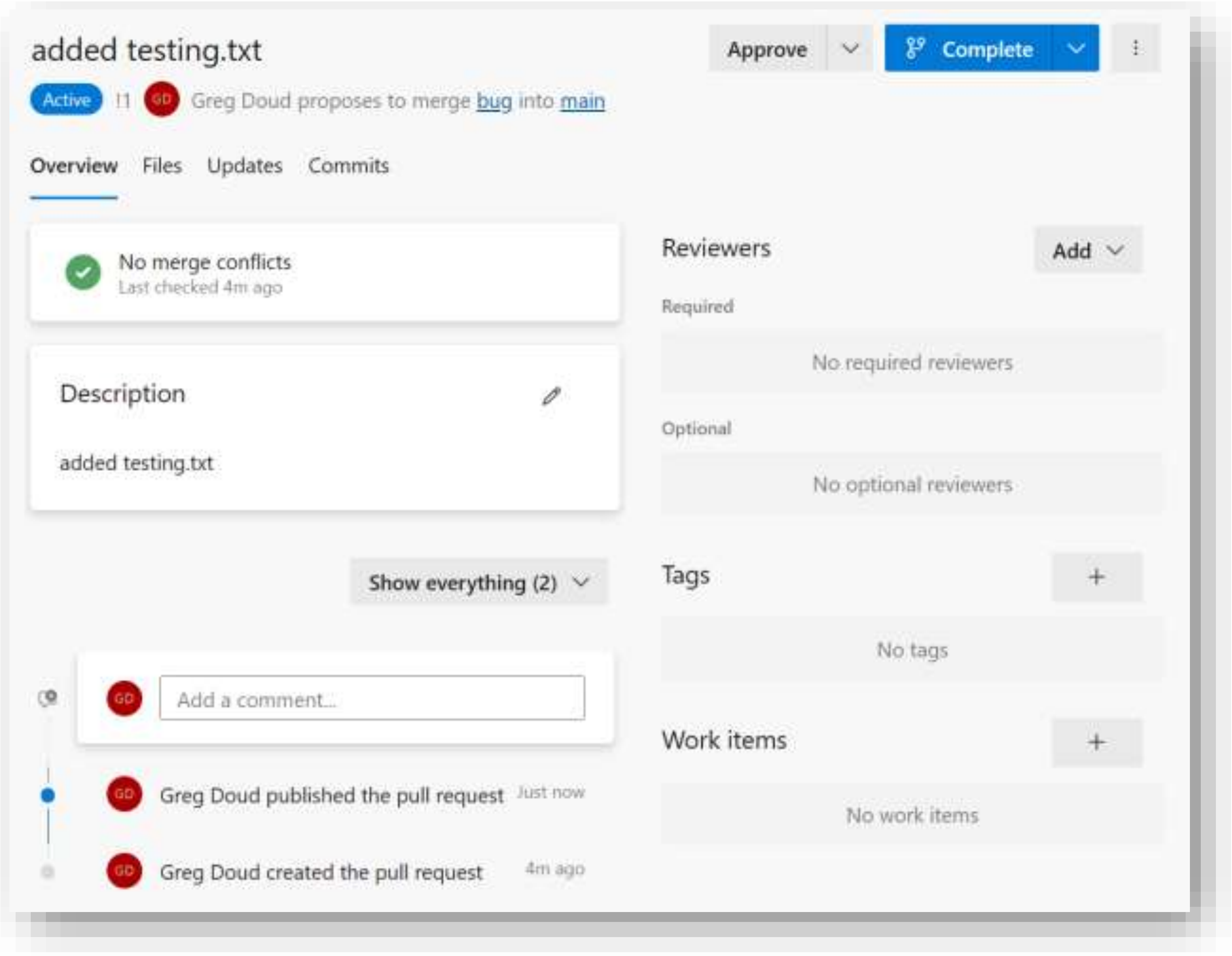
Reviewers
Required
No required reviewers
Optional
No optional reviewers

Tags
No tags

Work items
No work items

Greg Doud created the pull request 2m ago

Once the pull request is ready, an authorized user can process the pull request.


added testing.txt

Greg Doud proposes to merge bug into main

No merge conflicts
Last checked 4m ago

Description

added testing.txt

Greg Doud published the pull request Just now

Greg Doud created the pull request 4m ago

At the top right are the "Approve" and "Complete" buttons. Both allow dropping-down and selecting different items.

The values for the "Approve" button are:

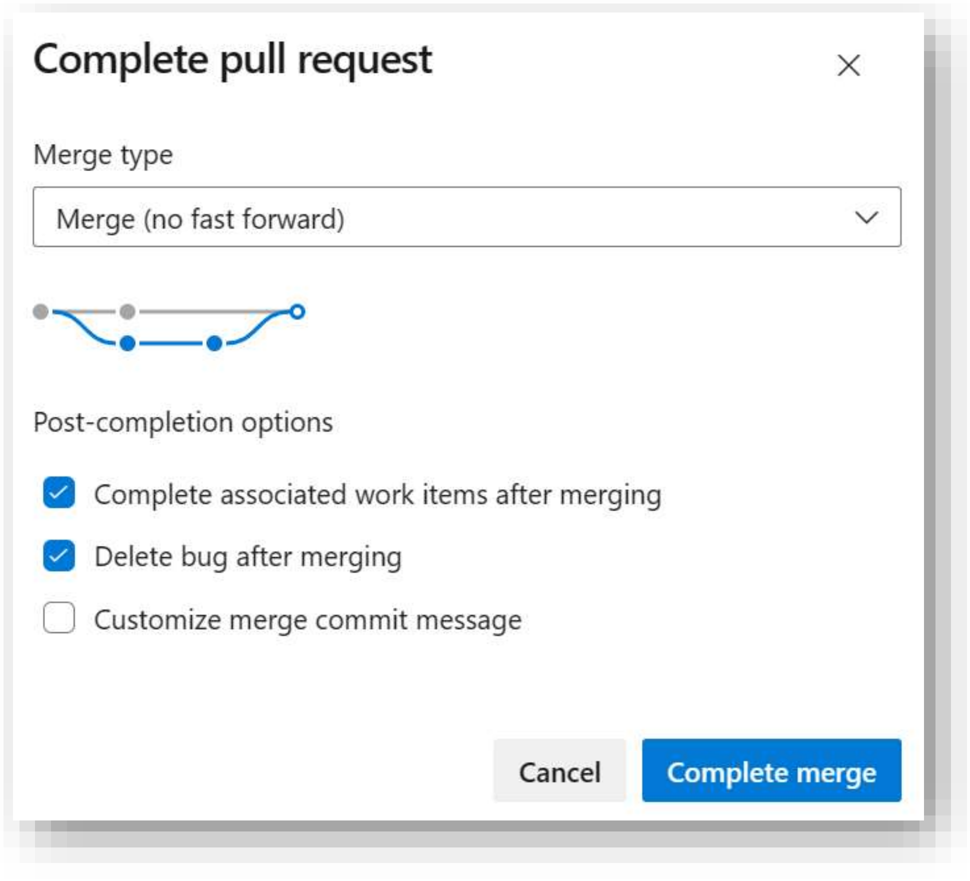
- **Approve** – A reviewer approves the pull request without any comments.
- **Approve with suggestions** – A reviewer approves the pull request with comments.
- **Wait for the author** – Waiting for the author of the pull request to make some changes.
- **Reject** – The pull request is rejected and must be changed or abandoned.
- **Reset feedback** – Reset the pull request to its original state.

The values for the "Complete" button are:

- **Complete** – Merge the pull request.
- **Mark as draft** – Return the pull request status to draft.
- **Abandon** – The pull request will not be merged and can be deleted.

6.5. Completing the pull request

When all is well with the pull request, the “Complete” button is clicked. The “Complete pull request” dialog displays.



Complete pull request

Merge type

Merge (no fast forward)

Post-completion options

- [x] Complete associated work items after merging
- [x] Delete bug after merging
- [ ] Customize merge commit message

The *Merge type* has four options:

- Merge (no fast forward) – The changes from the merging branch are replaced in the branch being merged into.
- Squash commit – Multiple commits in the merging branch are grouped into a single commit
- Rebase and fast-forward – The commits in the merging branch are just copied to the HEAD of the branch being merged into.
- Semi-linear merge