A Brief Introduction to GitHub for Social Scientists using Stata and Dropbox[*]

Heitor S. Pellegrina

May 24, 2018

Introduction

GitHub is a widely used online platform that computer scientists use to keep track of their codes and to collaborate on their projects. The word GitHub is the combination of Git with Hub. Git is a version control system that stores all the history of a code. With Git, programmers can check what they changed in their codes and when they changed their codes. They can also restore previous versions of their projects if they do not like their new versions. GitHub is a web-based repository that keeps all the history of a code on a cloud using the Git system. Since everything is in a cloud, it allows many researchers to colaborate on a specific project.

Why should I use GitHub+Dropbox instead of just using Dropbox?

Dropbox is a great resource for teamwork. It allows many people to work on the same project and everything is automatically synchronized across computers. However, as projects expand in terms of size and collaborators, one may run into problems. First, if you have used Dropbox to work with many collaborators you probably noticed the generation of "conflict" copies. This happens when Dropbox is not sure which version it should use because there is more than one synchronization happening at the same time. As I explain below, GitHub tends to avoid this problem because files are not automatically synchronized. Second, Dropbox lacks a more sophisticated system to track what changes in each version of a code. If something changes, we do not know who changed a

[*] The purpose of this material is to serve just as an introduction to Github. I hope that this material can be useful for Social Scientists who are trying to integrate Github with Dropbox for the first time. This document should not be considered as representative of high standards of project management! For an advanced template of project management using GitHub, see the material here click here. I thank Yaw Nyarko, Andres Fajardo and Martin Smith for comments. Contact: heitor.pellegrina@nyu.edu

code and what exactly they changed in it. GitHub has an interface that makes it easy to identify these changes. Third, if you want to bring back previous versions of a project, GitHub can do it quickly, but with Dropbox, we can only bring back specific files one at a time. While some of these problems may be more likely to appear in large projects (imagine writing the codes for facebook for example...), as big empirical projects become more frequent in Economics, I believe that the use of Github will inevitably become more frequent as well. The main cost of using GitHub is that you have to be much more systematic about the way how you work since you have to manually synchronize your changes with the cloud.

What is the difference between Dropbox and GitHub?

For me, the main difference between Dropbox and GitHub is that now we have an intermediary step for the synchronization of files called *"commit"*. In Dropbox, we have

$$\text{Master Repo (on the web)} \iff \text{Local Repo (in your computer)},$$

where Master Repository is the cloud where the codes are and local repository is a copy of the Master Repository that you keep in your computer. If someone changes something in their Local Repository, things are automatically synchronized with the Master Repository and across all the Local Repositories of other researchers working on the project. Therefore, if someone makes a mistake, this mistake spreads across the project and there is no straighforward way of identifying this mistake and going back to previous versions of the project. Now, with Git, we have an intermediary step

$$\text{Master Repo (on the web)} \iff \text{Commit} \iff \text{Local Repo (in your computer)},$$

The way how synchronization works with Github is as follows. You first download all the repository that cointains the codes for the project to your computer. If it is the first time that you are downloading a project, you use a process called *"clone"*. If you are just updating the project to your local repo, you use a process called *"Pull"*. Both are essentially downloading the files from the cloud. After you make your changes to the files, you can then upload them to this intermediary step called commit using a process called *"add"*. Once you added everything that you want, you use a process called "commit", which basically packages all the "*adds*" that you made. You can upload this commit to the cloud using a process called "*Push*". GitHub keeps track of all the versions of the project. If someone makes a mistake, we can easily go back to the version of the project that existed before the mistake.

See the Big Picture in the Appendix A to get a sense of how this will work!

Can I use GitHub as a repository for my datasets?

GitHub is a tool for keeping track of codes, but it is not supposed to be a repository for datasets. This is where an integration of Dropbox with Github is useful. GitHub can keep track of the codes that are written for a project, and we can leave the datasets and output of the project (figures, tables and pdfs) untracked. For example, a researcher can keep track of do-files that clean the data, but only keep the final version of the cleaned dataset. The idea is that, by keeping track of the code, we can always go back to previous versions of the project and re-obtain previous versions of the cleaned dataset.

Prerequisites for this manual

Before you move forward, make sure that you do the following

- Go to GitHub and set up an account (click here).
- Follow their introduction guide.
- Go to Git website and install git (click here).

Installing Git on Windows

After downloading the installation file, execute the installer and leave the default options in every step **except the last one.** At this point, check the option "Enable symbolic links." Screen shots of this manual's recommended installing options are available below in Appendix A.

Synchronization using GitHub

There are two ways to synchronize your files with the master repository. First, you can use a software called GitHub Desktop. This is a software where you can "point-and-click" your way through synchronization. I found this software quite user friendly. Second, you can use the Terminal in Mac[1] (or the command line in windows). They may seem scary at first, but, in practice, you do not have to be a sophisticated programmer to use it. There are just a few commands that you have to use for your daily work. I see two main benefits in learning how to actually type the commands in Terminal. First, whenever you want to do something specific and you google about it, the answers that you get are mostly based on command lines. Therefore, to understand the answers that are given by the online community, you need to have a basic understanding of the language that

[1] If you are using a Mac computer and do not know what Terminal is, read this website here.

they use. Second, there are some functionalities that I was not able to obtain with the graphical interface.

Terminal use in Windows

After installing Git with the options mentioned above, you can use Git BASH to work with the same commands you would use if you were in a *NIX terminal. When this manual asks you to use 'the terminal', you should use Git BASH. There will be warnings when different commands are needed if working on a Windows machine.

Basic commands (MAC)

- *ls* (this command will **list** the files in the current directory)
- *cd Dropbox* (this command will change the **current directory** and enter the folder called "Dropbox")
- *cd ..* (this will go a folder back in the hierarchy of folders)
- *mkdir projects-git* (this command will **make a directory** called projects-git)
- *git clone https://github.com/hpellegrina/hello-world* (this command will make a clone of the online repository called hello-world to the current directory in your terminal)
- *git add table_1_summary_stat.do* (this command will add the do.file called *table_1_summary_stat.do* to the commit that you are preparing)
- *git remove table_1_summary_stat.do* (this command will remove *table_1_summary_stat.do* from the commit)
- *git add -A -m "Adding all files"* (this command **adds** to the commit all the new files that you have and creates a **message** associated with the commit saying "Adding all files")[2]
- *git commit -m "Adding table_1_summary_stat.do"* (this command will commit the changes that you made to the folder, the phrase inside the quotation marks is the comment that you are going to attach to your commit)
- *git push* (this command will **push** the commit that you prepared that is still in your computer to the git cloud)

[2] When you are looking throught the commits of a project, you can see the list of commits and the messages associated with each of them

- *git pull* (this command will **pull** the current version of the project to your desktop)
- *git status* (will show you the modifications/deletions that you made in your local repo)

How a typical day of work looks like?

- You start by going to your project's local folder in a terminal window and download (pull) any changes your collaborators may have uploaded to the online repository.

  ```
  $ cd local/path/to/the/project
  $ git pull
  ```

- You work on a task, i.e. you make changes to the project's files.
- You check the changes you have made and select (add) the changes you want to approve into a bundle of changed files.

  ```
  $ git status
  $ git add -A
  ```

- You approve (commit) the changes and add a comment so that your collaborators know what the changes consist in.

  ```
  $ git commit -m "Issue #1: added table_1_summary_stat.do"
  ```

- You work on other tasks and make further commits for each (steps 3 and 4 above).
- You upload all the commits you have made to the online repository.

  ```
  $ git push
  ```

Hello-World! I'm an Economist!

We will be following the tradition in computer science. Below you will have your first commits using GitHub, Stata and Dropbox. I taylored these commits to include some of the tricks to integrate Dropbox with GitHub.

Exercise #0 - Getting started.

In this exercise you will set things up so that you have access to the files needed for the subsequent exercises.

1. Before you start, make sure that you have a GitHub account and that you have installed Git into your machine. To double check that it is installed properly, go to terminal and type the following command. If it is already installed, it will show a list of commands that you can use.

   ```
   $ git
   ```

2. Link git to your GitHub account.

   ```
   $ git config --global user.email your@email.address
   ```

3. Make a copy of **hello-world-econ** into your computer. This is just to give you access to the files needed for this manual.

   ```
   $ cd /sample/path/for/files
   $ git clone https://github.com/hpellegrina/hello-world-econ
   ```

4. There is a folder called **my-hello-world-econ-shared** in **hello-world-econ.** Copy this folder into your Dropbox folder. This is the folder you will use to share data, documentation, etc., with your collaborators.

Exercise #1 - Using GitHub to track codes.

Remember the big picture: you are interested in GitHub because you want to track the changes made to your project's codes and light files such as output tables and graphs. In this exercise, you will use GitHub to track the codes and graphs of a new project: **my-hello-world-econ**. Specifically, you will (1) create a remote repository of your project in GitHub, (2) configure git so that that it ignores *.dta Stata files, (3) set up a local repository to work on your project, and (3) learn how to upload local changes to the online repository.

1. Go to github.com and create a repository called **my-hello-world-econ.**

2. Click on 'Create new file' to create a file called **.gitignore** with the content below. Note that (1) this file should be in the main folder of your project, (2) the dot at the beginning of the name should be included, and (3) lines that start with '#' are comments. After creating the file, click on 'Commit changes to the master branch'.

   ```
   # Ignore all *.dta files in the project
   */.dta
   ```

   Important note: Changes made to **.gitignore** will only apply to untracked files. If there are any files within these folders that git is already tracking, you will need to untrack them first. This can be the case if, for example, you already added a file to a commit. Here is an example of how to untrack files.

3. Open a terminal window and use the following commands to create a local copy of your repository. This will create a folder called **my-hello-world-econ** in your computer.

   ```
   $ cd /sample/path/for/your/project
   $ git clone https://github.com/yourusername/my-hello-world-econ
   ```

4. There is a **my-hello-world-econ-contents.zip** file in **hello-world-econ** (the repository you copied in Exercise #0). Extract the contents of this folder into **my-hello-world-econ**. Your local project folder should now have two subfolders: **analysis** and **data**.

5. The files you just added to **my-hello-world-econ** are local changes you made to the project. You will now update your online repository so that it incorporates these changes. To do so, you need to select (stage) the changed files you wish to update into a bundle of changes that you will then approve (commit) and upload (push) to the online repository. Let's start by staging the files to the commit. With the second command below you can check if the changes were added to the commit.

   ```
   $ git add -A
   $ git status
   ```

6. Now, create a commit and add a comment that explains what changes are being made.

   ```
   $ git commit -m "Copied the contents of my-hello-world-econ-contents"
   ```

7. Push the commit to the online repository. You will be asked to input your GitHub credentials at this point.

   ```
   $ git push
   ```

8. Go to github.com and check that the changes are now displayed in your online repository.

Exercise # 2 - Using Dropbox to access data inputs.

Remember the big picture: you are interested in Dropbox because it allows you to share files with collaborators in one place. You're not interested in the history of these files because they are either static (e.g. raw data, documentation, related papers) or they can be replicated by running the code that produced them in the first place (e.g. output tables from a do-file or pdf files from latex code). In this exercise, you will work with your first data input: **macro_indicators.csv**. A collaborator uploaded this file to **my-hello-world-econ-shared**, the project's shared Dropbox folder you copied in Exercise #0.

1. As always, you must start by updating your local repository to download (pull) any changes (commits) made to the online repository. You will check that you're standing on the 'master' branch. If you're not, the third line of code below will take you to that branch.[3]

   ```
   $ cd your/path/to/my-hello-world-econ
   $ git pull
   ```

2. You will now 'connect' to the data input. Go to the folder **population** in your project's data folder and create a symbolic link called **input**. This link must point to the folder in **my-hello-world-econ-shared** that contains the file **macro_indicators.csv.** If you are using MacOS or *NIX, use the following commands in the terminal. See this link for a helpful review of symbolic links.

   ```
   $ cd your/path/to/my-hello-world-econ/data/macro
   $ ln -s path/to/Dropbox/my-hello-world-econ-shared/macro input
   ```

   If you're using Windows, open the Command Prompt as an administrator (not Git BASH) and use the following commands. Note that we are using backslashes now and not forward slashes.

   ```
   >> cd your\path\to\my-hello-world-econ\data\population
   >> mklink /D input \path\to\Dropbox\my-hello-world-econ-shared-macro
   ```

3. Open the file **import_population.do** and edit the path in the third line so that it points to its folder, that is, to **my-hello-world-econ/data/population/code**. Save and run the do-file.

   ```
   1. clear all
   2. * Write down the directory below
   3. cd "your/path/to/my-hello-world-econ/data/population/code"
   ```

4. Repeat the last step for the files **import_agricultural_output.do**, **make_data_for_analysis.do** and **fig_va_urbanization.do** (in that order). You can find the last one in the **analysis** folder.

5. The codes you edited and ran produced new files. You will now upload (push) the changes you have made to your project to the online repository. Remember that we configured git to ignore *.dta files, so the files **population.dta**, **agricultural_output.dta** and **data_for_analysis.dta** should not be uploaded to the online repository.

   ```
   $ git add -A
   $ git status
   $ git commit -m "Changed do-files"
   $ git push
   ```

6. Go to github.com and check that the changes in the online repository do not include the creation of *.dta files.

[3] Later on we will talk about branches and how sometimes it ! See this for a discussion on pull vs fetch and rebase.

You have made your first contribution to my-hello-world-econ! Make sure you understand what each of the do-files are doing and how their relative paths work. The number of folders may seem excessive at first, but as a project grows you will find it helpful.

Additional Features for Teamwork

Adding collaborators

To add collaborators into your GitHub repository, you can just use the github.com website.

- Click on the repository of the project.
- Click on the settings of the repository (one of the tabs on the top).
- Click on collaborators.
- Type the email of you collaborator and add him/her there.

Issues

A useful tool for the organization of tasks in GitHub is the use of "Issues". Any collaborator in a project can create a new issue on the website associated with a repository and link their commits with these issues. For example, one can go to a repository and create an issue such as "Add clustered standard errors in table 1". When the issue is created it comes with a number. Let's say this number is 1. When someone in the project commits something related to the issue, this person can directly link his commit with the issue by writing in the beginning of the message #1. When you do so, the website will display a small notification under the issue with the user who updated the issue and also what was added there. You can also assign issues to specific people in the project.

Branches

Another useful tool for teamwork is the creation of branches. Let's say that you want to test the results from your paper using a restricted sample, but you do not want to affect your current results because it is possible that the restricted sample contains problems that you may find while working on it. In this case, you can create a "*branch*". This is a parallel repo on the cloud that your co-authors can use. In that case, we have

$$\begin{array}{rcccl}
\text{Master Repo (on the web)} & \Longleftrightarrow & \text{Commit} & \Longleftrightarrow & \text{Local Repo (in your computer)} \\
\uparrow\downarrow & & & & \\
\text{Branch Repo (on the web)} & \Longleftrightarrow & \text{Commit} & \Longleftrightarrow & \text{Local Repo (in your computer)}
\end{array}$$

If the new sample works well, you may merge the branch with the master repo. If not, you can just delete the branch and the master repo will still be intact.

Big Picture

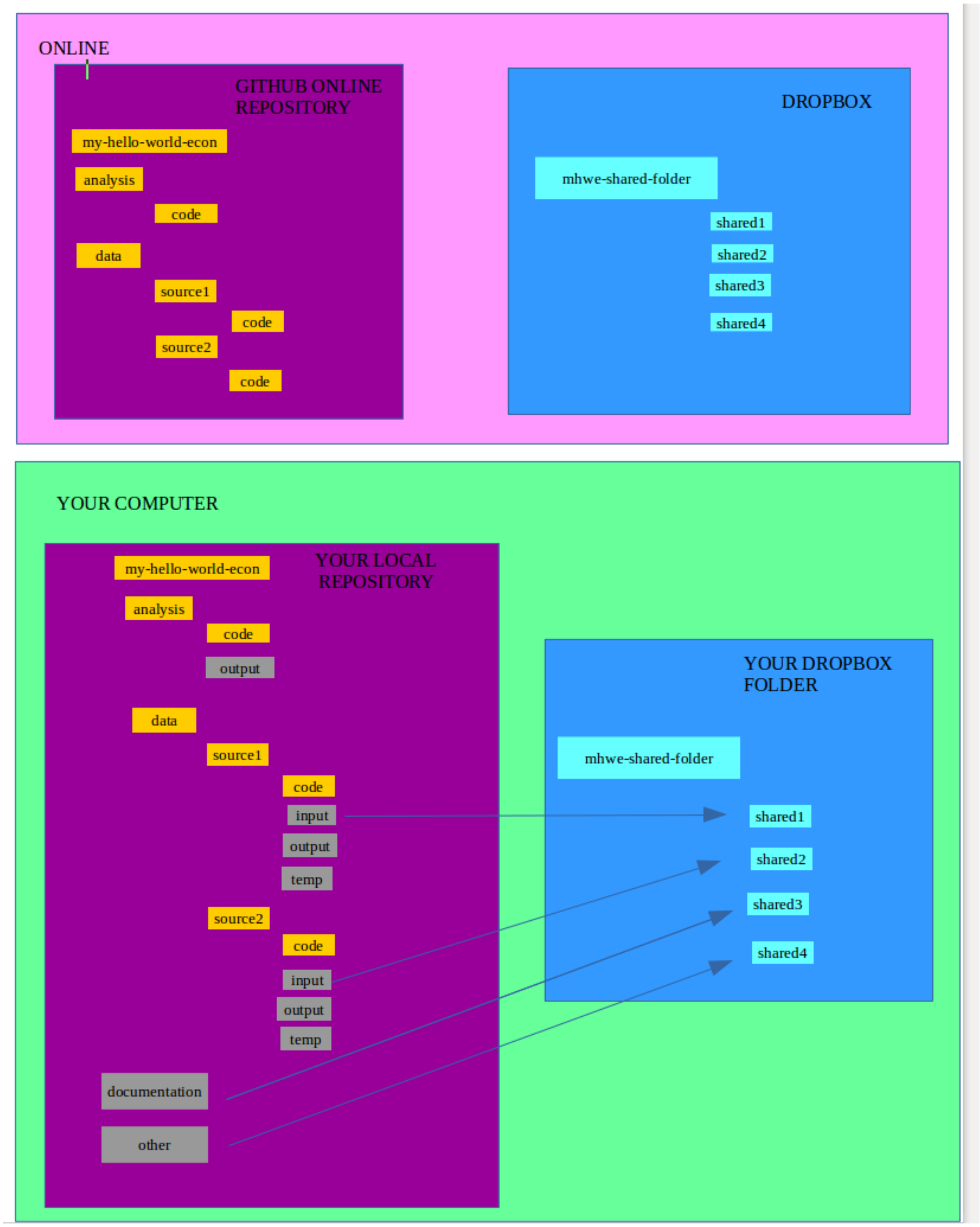
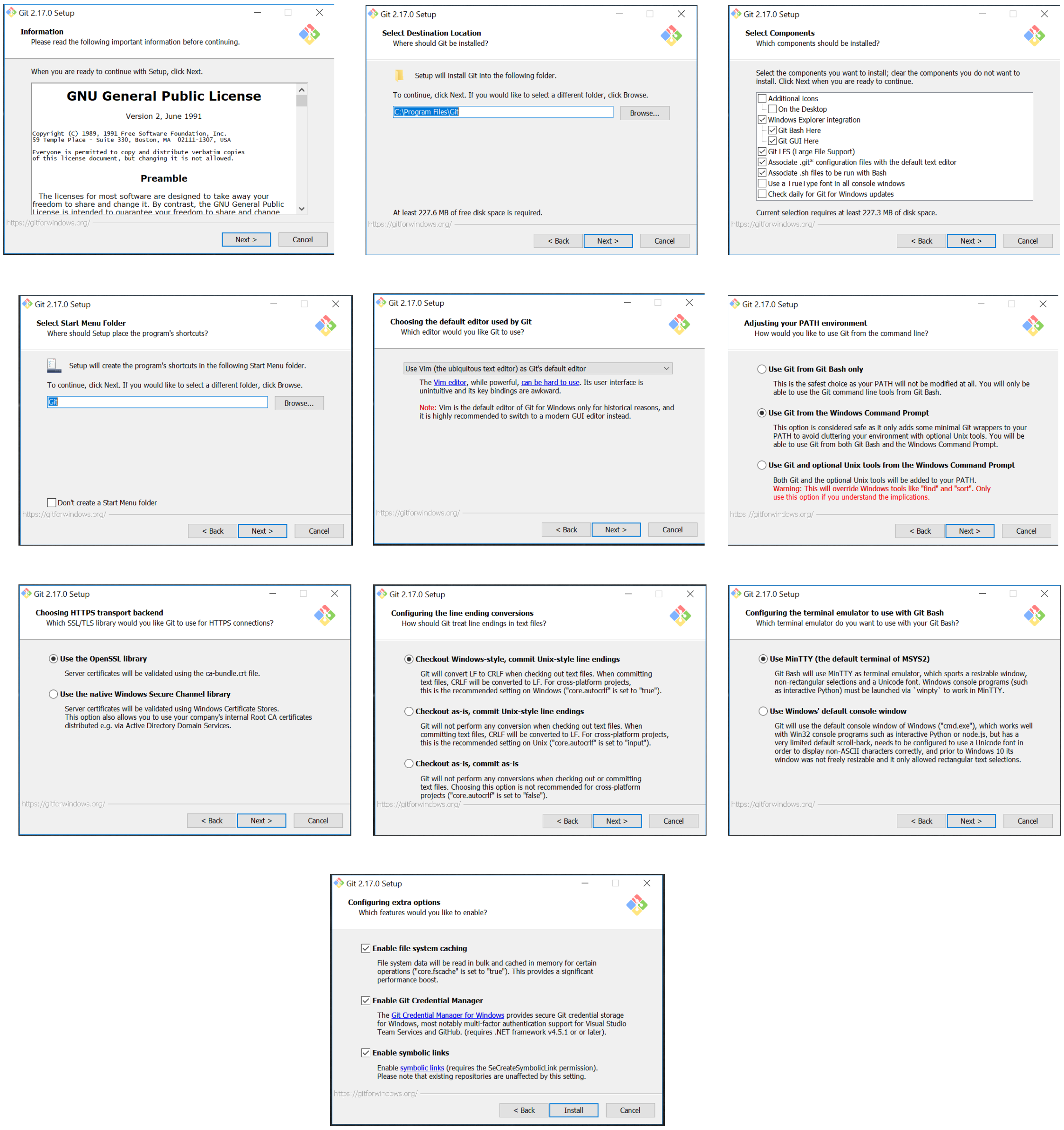


Figure 1: Boxes represent folders and their contents. Yellow folders are synced with GitHub unless they are grey, which means their contents are ignored by git. Light blue folders are stored in Dropbox. Arrows represent symbolic links.

Windows installation screenshots

Figure 2: Windows installation options



Git 2.17.0 Setup

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

Git 2.17.0 Setup

Select Destination Location

Where should Git be installed?

Setup will install Git into the following folder.

To continue, click Next. If you would like to select a different folder, click Browse.

C:\Program Files\Git

At least 227.6 MB of free disk space is required.

Git 2.17.0 Setup

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
  - ☑ Git Bash Here
  - ☑ 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
- □ Use a TrueType font in all console windows
- □ Check daily for Git for Windows updates

Current selection requires at least 227.3 MB of disk space.

Git 2.17.0 Setup

Select Start Menu Folder

Where should Setup place the program's shortcuts?

Setup will create the program's shortcuts in the following Start Menu folder.

To continue, click Next. If you would like to select a different folder, click Browse.

Git

□ Don't create a Start Menu folder

Git 2.17.0 Setup

Choosing the default editor used by Git

Which editor would you like Git to use?

Use Vim (the ubiquitous text editor) as Git's default editor

The Vim editor, while powerful, can be hard to use. Its user interface is unintuitive and its key bindings are awkward.

Note: Vim is the default editor of Git for Windows only for historical reasons, and it is highly recommended to switch to a modern GUI editor instead.

Git 2.17.0 Setup

Adjusting your PATH environment

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

○ **Use Git from Git Bash only**

This is the safest 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.

● **Use Git from the Windows Command Prompt**

This option is considered safe as it only adds some minimal Git wrappers to your PATH to avoid cluttering your environment with optional Unix tools. You will be able to use Git from both Git Bash and the Windows Command Prompt.

○ **Use Git and optional Unix tools from the Windows 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.

Git 2.17.0 Setup

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.

Git 2.17.0 Setup

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").

Git 2.17.0 Setup

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.

Git 2.17.0 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 Git Credential Manager**

The Git Credential Manager for Windows provides secure Git credential storage for Windows, most notably multi-factor authentication support for Visual Studio Team Services and GitHub. (requires .NET framework v4.5.1 or or later).

☑ **Enable symbolic links**

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

Useful links

Here is a list of helpful online tutorials:

- Git & GitHub Crash Course For Beginners
- Git Basics: Merge and Rebase
- Git Handout - Houtan Bastani