DST-Labs:
Collaboration and Version Control using Git on an Offline Server

Anders Humlum* Bjørn Bjørnsson Meyer† Jonathan Leisner‡

December 3, 2021

Abstract

This is a manual for project collaboration and version control using Git on servers without internet access like those supplied by Statistics Denmark.

Contents

*University of Chicago, humlum@uchicago.edu.
†University of Copenhagen, bjorn.bjornsson.meyer@econ.ku.dk.
‡University of Copenhagen, jl@econ.ku.dk.

1 Introduction

As economic researchers, we increasingly find ourselves collaborating on computer code and data files. Every step of our work, from data cleaning, statistical analysis, model simulation, or the formatting of outputs involves writing, debugging, and sharing of code. Yet, few of us have ever received any formal training in computer science. Most of what we do is self-taught or passed from researcher to researcher. In Code and Data for the Social Sciences: A Practitioner's Guide, Gentzkow and Shapiro take invaluable steps toward breaking bad coding habits. We greatly recommend their guidelines, although it is difficult not to take their points personally.

The purpose of this manual is to facilitate collaboration and version control with the Git software on the research servers at Statistics Denmark (DST). Most empirical economists interested in Denmark will at some point use the vast opportunities in the register data at DST.

On a typical DST server, collaborators can access the same folders, but many things are made difficult by the lack of opportunities to install software or access the internet. For example, the lack of internet access prevents the use of online hosting services for remote repositories like GitHub.

For many researchers, including ourselves, the current situation is that several people work on the same folders and files without any system of version control. Researchers instead resort to copying the code files to a separate folder at important milestones. Git offers many advantages both in terms of version control and collaboration between researchers.

Version control with Git works with a repository where all the codes are stored. A user checks out the codes from the repository, works on them, and at some point checks them back in. This creates the waypoints of the version control. The defining feature of Git is that the version control is "distributed", such that each user has their own local version of the code base. This delivers a host of advantages for us. Most of us have probably at some point worked as a RA and received the following instructions from a senior: "you can find it in my folder - please don't change or run any codes!". Needless to say, with such a system for file sharing, small human mistakes can jeopardize a whole research project.

Git solves these problems as new users will get their own distributed version of the code base to use, and potentially build onto, before adding it back to the main branch. Furthermore, Git allows different users to run and alter the same codes simultaneously, solving the usual coordination issues with conflicting copies when researchers collaborate around the same codes.

As part of this manual, we supply templates of our folder structure and other useful hacks in the spirit of the much more advanced GSLab Manual.[1]

2 Command-Line Shell

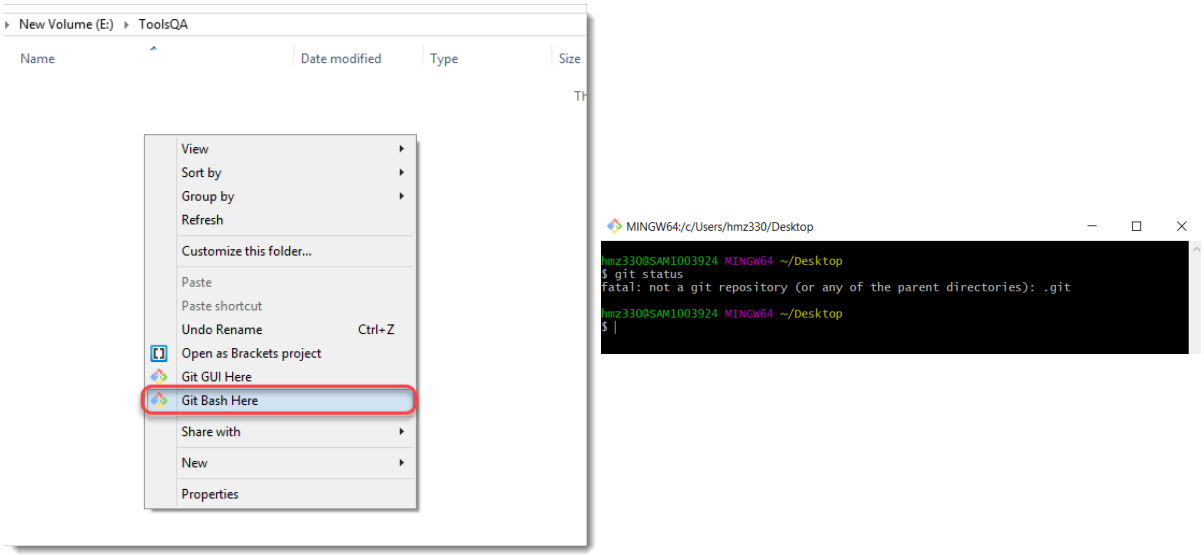
Throughout this manual, the user executes some basic shell commands. We recommend and supply the syntax for the Git Bash shell. Git Bash is a Unix-based terminal for Windows that usually comes with the Git software installation. It is also possible to use Command Prompt, Powershell, or a range of other shells on the servers of DST.

[1] Our folder template can be downloaded here.

A convenient way to launch Git Bash is to right-click in any folder folder and select "Git Bash Here", see Figure 1. This way, Git Bash will launch with the folder as the Working Directory, sidestepping the following shell navigation commands:

```
# Print the current working directory
pwd
#List content of the working directory
ls
# Change the working directory
cd <path_name>
# Change the working directory to a subfolder
cd ./<subfolder_name>
# Move one level up in folder structure
cd ../
```

Figure 1: Git Bash



Note: Launching Git Bash and executing a Git Status command in the current working directory (desktop).

3 Git Setup

The Internet contains an ocean of resources on Git and version control. Start with this introductory video. Useful beginner tutorials can be found here and here. Chapters 1 to 3 of the Pro Git Book. Hands-on tutorial here.

Many of the online guides are provided by cloud-based hosting services for remote repositories, such as GitHub or Bitbucket. We will, of course, not use these hosting services as we work on a server without internet access. Instead, we will set up our own "remote" ("bare") repository on the server, located in a centralized project directory, that each of us will clone to our user work directories.

Before a new user starts to use Git, she must type the following in Git Bash:

```
# Configure Git user
git config --global user.name <your_username>
git config --global user.email <your_email_address@example.com>
```

4 Creating A New Project Repository

In this section, we describe how to create a new project folder from our template, initialize a local/"bare" Git repository, and set up the first user folder.[2] The following series of Bash commands will copy folder structures and settings from our template, and initiate the new Git repository. We describe the folder structure below.

1. Create the project folder (`project_name`), and copy the "shared" template folder into it[3]

   ```
   cp -r <path>/Projects/template/shared <path>/Projects/project_name
   ```

2. Initialize a bare repository[4] in the new project folder (write out full paths, no relative paths with "../" here)

   ```
   git init --bare <path>/Projects/project_name/repo.git
   ```

3. Create the project folder in the user's folder and clone the repo

   ```
   git clone <path>/Projects/project_name/repo.git <path2>/user_name/project_name
   ```

4. Copy the content of the user template folder into the user's project folder

   ```
   cp -r <path>/Projects/template/user/. <path2>/user_name/project_name
   ```

5. In this final step, we commit the changes in the user folder, and push them to repository (to include settings for additional users)

   ```
   cd <path2>/user_name/project_name #Navigates to the user folder
   git add . #Stages all files for Git
   git commit -m "Initialized repository with the DST-Labs template"
   git push origin master
   ```

4.1 Folder Structures

Your user folder should now look like depicted in Figure 2. We adhere to the folder structure in Gentzkow and Shapiro (2014), where subproject folders are organized into build and analysis directories, which in turn are organized into code, temp, and output folders. The `.git` is where the Git software works, and should not be touch by the user. The `.gitignore` and `.gitattributes` files are for the user to supply settings for Git. We provide these in the templates. In particular, we suggest making Git ignore the tracking of all data and output

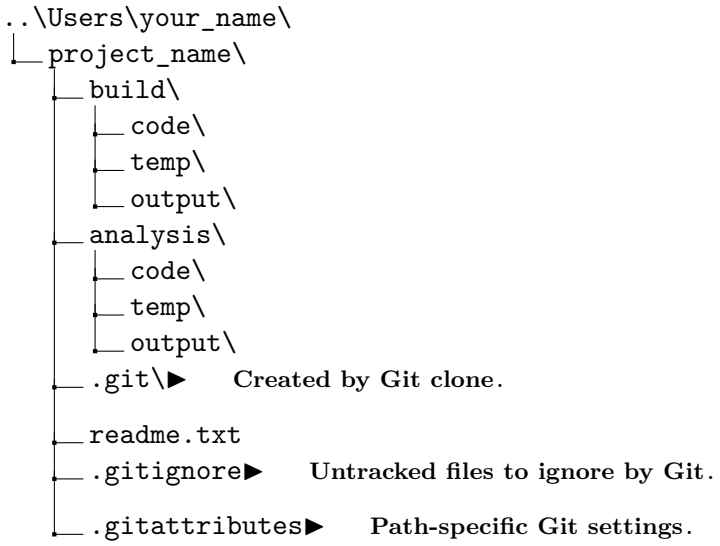
[2] This section is based on Stack Overflow (scroll down to the answer by 'adelphus').

[3] -r means recursive and includes subfolders. If the bash directory is <path>, it can be substituted with the relative path "./"

[4] It is conventional to give bare repositories the extension `.git`.

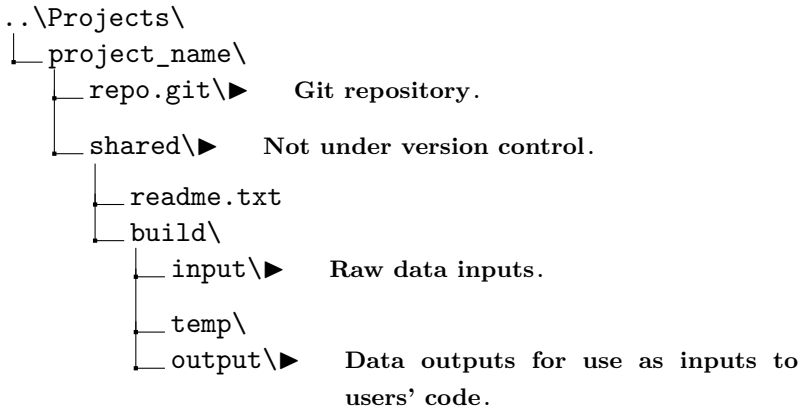
formats, like `.csv` or `.pdf`. Omitted from Figure 2, we also have a `readme.txt` file in all the empty folders in the tree. It is good practice to document throughout the folder structure, and it secures that the otherwise empty folders will be tracked in Git.

Figure 2: User Folder



We now turn to the project folder, depicted in Figure 3. The `repo.git` folder is the central Git repository, and should never be touched by users. The `shared` folder should hardly be touched either, or there should at least not be run codes in it. We will describe the ideas and use of the `shared` folder in section 6 on workflows.

Figure 3: Project Folder



5 Adding a User to an Existing Project

To initiate a new user, we simply clone the Git repository in the project folder. This is the same procedure as cloning a repository from GitHub, the only difference being that we supply the path to the project repository instead of an URL to GitHub. The user creates a folder with the projects name, and runs Step 3 of Section 4 with this command:

```
git clone ../Projects/project_name/repo.git ../Users/your_name/project_name
```

6 Workflows

6.1 The `shared` Folder

As the name suggests, the `shared` folder contains files that are shared between users and thus not distributed to each user's personal folder. It is placed on the server in a folder accessible to all users. The folder is mainly intended for large data files that we, out of storage concerns, do not want to maintain distributed copies of. It is also the most natural place to keep all raw data inputs to the project. Furthermore, we also keep transformations of raw data that are too time-consuming to execute in `shared`. In our case, pulling data from registers and transforming it can easily take more than a day. The `shared` folder is confined to building data sets, not conducting any analysis on these. For this reason, the template for `shared` does not contain an analysis subfolder. Furthermore, we do not run codes located in the shared folder; these are all distributed to the users' `build/code` folders. A good reason for not keeping code in `shared` is that the folder is not under version control through Git. That said, it might be worthwhile to expand version control to the data files in `shared`. We suggest this as an add-on in Section 7.

6.2 Example of Workflow with Git

Section 3 lists a handful of online tutorials for Git. Here is an example of a common workflow with Git where a subproject is finalized in a new branch of the code development.

1. Set user work folder for project as current working directory

```
right-click and launch Git Bash in ../Users/<your-name>/<project-name>
```

2. Update list of remote branches

```
git remote update
```

3. Create branch for subproject

```
git branch <branch-name>
```

4. Switch to branch for subproject

```
git checkout <branch-name>
```

5. Whenever a subtask is completed, make a commit[5]

```
# Stage files
git add .
# Commit changes
git commit -m "<Content of completed subtask for commit>"
```

6. Whenever a task of a subproject is completed, push to branch

```
git pull origin <branch-name> # Not relevant if first push to branch
git push -u origin <branch-name> # Push to subproject branch
```

7. When subproject is finalized, merge branch onto `master`

[5]Here, we stage the entire work folder. Other times, you will only want to commit specific files that relate to the completed subtask.

```
# Merge finalized <branch-name> onto master branch
git checkout master
git pull origin master
git merge <branch-name>
git push origin master
```

8. Delete subproject branch

```
# Delete finalized <branch-name> locally
git branch -d <branch-name>
# Delete <branch-name> remotely
git push -d origin <branch-name>
```

7 Add-Ons

7.1 Run Codes with a Script

It is useful to be able to run a whole build process or analysis from the beginning to the end at once. Often it will cover many different files and without a cookbook it can be difficult for a new collaborator to replicate. To complicate things, the steps often uses different scripts, potentially in different folders, like SAS scripts for retrieving data, python scripts for transforming data, etc.

We provide the template "run.sh" for a shell script in our `Add-ons` folder that executes a series of scripts after double-clicking. It contains the following syntax:[6]

```
#Invoke bash from OS (# needed):
#!/bin/bash
#Open the SAS software and execute (in background) the SAS script in subfolder sas:
"C:\Program Files\SASHome\SASFoundation\9.4\sas.exe" ".\sas\sas_script.sas"
#Run a python script from the python subfolder:
python ./python/python_script.py
```

7.2 Jupyter Notebooks and Python

Jupyter Notebooks is a great way to work with data interactively, and document the processes for collaborators. Jupyter integrates many languages, but here we focus on Python. The notebooks can be used as an intermediate step before producing Python code to run, or they can serve as end-products in the form of nice-looking report.

Jupyter is usually invoked by entering "jupyter notebook" in some shell window. On the DST server, it is recommended to use the corresponding Anaconda Prompt as it makes sure Jupyter opens in the user's chosen root folder. To make Jupyter Notebook open in your current folder, you can place the windows command prompt file "open_jupyter_here.cmd" in your folder and double-click. It contains:

```
jupyter notebook --notebook-dir=%cd%
```

The problem with Python notebooks (.Ipynb) and Git is that there is a ton of meta-data in the notebook, making it difficult to compare changes in Git. If you do not need the features of a notebook, we recommend a Python script (.py). If not, we suggest the following workaround:

[6] We write the full path since SAS is not added to the Windows environment on the server. Python is and can launch with its name.

1. End all notebooks with a line that converts the current notebook to a Python script with the same name.

```
!Jypyter nbconvert --to script notebook_name.ipynb
```

2. Stage and commit both the .ipynb and .py files.
3. You can then use the notebooks as usual, and use the .py files for shell runs and Git code comparison.

7.3 Git in the `shared` Folder

In some situations, it can be useful to have version control of the data files in the `shared`. These are not covered above, as they are not distributed and tracked with Git. This is probably unnecessary in most cases outside DST, where you have full control with the raw input data files. A problem that researchers face at DST is that the registers we pull raw data from are sometimes are revised. Often you want to pull new data to get the latest available year of the registers. Here, it would be nice to be able to check whether something else unexpectedly has changed since the previous version.

An ad-hoc solution is to run Git separately and undistributed in the `shared` folder:

```
#launch Git Bash:
right-click in the shared folder and open Git Bash.
#Inititialize a regular Git repository:
git init
```

In this setup, it is important that users commit to this Git system whenever they update files in the `shared` folder, for example pulling updated raw data from the registers. This can be done from the user folder by double-clicking on the shell script shared_commit.sh containing:

```
cd "<project folder>\shared"
git add .
git commit -m "Stage and commit from user dir"
```

7.4 Text Editors with Git Integration

A modern text editor for code writing has many advantages. One is that the Git system can be integrated and enable you to stage and commit without opening Git Bash. Popular choices are Visual Studio Code, Atom, and Sublime Text. The full functionalities of these editors are somewhat limited at DST's remote server. We use Visual Studio Code, and the Git integration works well.