Lab 1 – Configuring Git & Cmder

Contents

[1 Introduction & Overview of Labs 1](#_Toc103077707)

[2 Introductory Videos 3](#_Toc103077708)

[3 Download & Install Git and Cmder 3](#_Toc103077709)

To make this and subsequent Labs easier to read, it is recommended that you turn off spell checking and grammar checking in Word:

1. Choose: File, Option, Proofing
2. At the very bottom, check: “Hide spelling errors…” and “Hide grammar errors…”

# Introduction & Overview of Labs

A *Version Control System* (VCS) is a system that keeps track of changes to code (or any digital artifact) and is a fundamental component of [*software configuration management*](https://en.wikipedia.org/wiki/Software_configuration_management)*.* Benefits of a VCS:

1. When problems occur, you can revert back to a previous (working) version.
2. Over time, you can see what changes were made, by whom, when, and why the change was made.
3. It is the most efficient way for a group of individuals to collaboratively develop a software system. All professional entities (company, government agency, *etc.*) that develop software use a VCS.

In some sense, a VCS is a database of the changes made to code over time, where each change records who made the change, when, and why. When a change is made to code, a VCS does not typically store a new version of the entire file, it only stores the changes. However, a VCS can provide the most current version of a file, or it can revert back to any version of a file after any change that has been made. Popular VCS’s: [Git](https://en.wikipedia.org/wiki/Git), [CVS](https://en.wikipedia.org/wiki/Concurrent_Versions_System), [SVN](https://en.wikipedia.org/wiki/Apache_Subversion), and [Mercurial](https://en.wikipedia.org/wiki/Mercurial) have features that enhance the management of changes to code. We will use Git in this class. Some terminology

|  |  |
| --- | --- |
| Term | Description |
| Repository | A database of the changes to a code base. When you use Git to manage a project, it uses a repository (repo). Each project has its own repository. A repository can be stored on a local computer and/or on the internet. |
| Commit | A change to a code base that records the change, who made it, when, and why. Commits are recommended to be small. Sometimes that might be a method, and sometimes that might be the beginning version of a class. A commit would generally not be an entire, complex class. |
| GitHub | A website where repositories are stored and managed. Other common hosts are BitBucket and SourceForge. |

This is a high-level view of the way Git and GitHub are used in practice

* A GitHub repo stores the code base for a project.
* A developer *pulls* code from GitHub to their local machine. Typically, you pull the entire code base. In some sense, they have checked out the code. However, this doesn’t mean someone else can’t pull the same code and be working on it at the same time.
* The developer works locally, committing often to a *local repo.*
* Meanwhile, another developer pulls code to their local machine and begins working and committing locally.
* At some point, the developers *push* their commits to GitHub

Another useful feature that both Git and GitHub support is *branching.* A repo has a *master* branch which is generally a fully working and tested code base. Then, a developer creates a *branch*, which is a copy of the master branch. Each developer has their own branch and this is where they work. In other words, they *pull* code from their branch and *push* *commits* to their branch. Eventually, a developer’s branch is *merged* with the master branch. The merging process preserves all the commits so that there is a complete history of the code. This process continues. One complication that arises is when attempting to merge a branch where there is conflicting code between the branch and the master branch. For example, if the master branch has method that accepts one parameter and the branch has the same method, but an additional parameter, then this is a conflict. This can be a major source of frustration; however, if you follow these labs carefully, you’ll know how to handle them when they come up.

Finally, *GitHub* supports *pull-requests*. A pull-request results when you attempt to merge one branch into another. It is a way for other team members to review the changes you have made, provide feedback, and ultimately approve or deny the merge.

This is a summary of the Labs:

|  |  |  |
| --- | --- | --- |
| Labs | Title | Description |
| 1 | Configuring Git and Cmder | Provides an introduction to VCS (above) and then shows how to install Git locally, and how to install Cmder to use for shell commands |
| 2 | Basic Git Commands | Learn how to create a local repo, put a code base under local source control. How to add and remove files, and commit these and other changes. |
| 3 | Rolling Back Commits | Learn how to inspect commits and revert back to a previous version, locally. In other words, how to undo any number of sequential commits. You’ll also use *gitk,* a GUI based program that allows you to view information about commits. |
| 4 | Branching & Merging | Learn how to create and use branches, and how to merge them back into master, or another branch. |
| 5 | Resolving Merge Conflicts | Learn how to deal with a merge where there are conflicting pieces of code in each branch. |
| 6 | Rebasing | This is another form of merging two branches and is an important part of most workflows. |
| 7 | Pushing to GitHub | Learn how to push code from a local repository to a GitHub branch. |
| 8 | Pulling Changes from GitHub | Learn how to pull changes from a branch on GitHub to a local repo. If there is not a local repo, you will first *clone,* which is the next lab. *Pulling* assumes you have a local repo and can be initiated if there have been changes on the GitHub branch since the last time you pulled code. In other words, if two people are working on the same branch and one person pushes to that branch on GitHub, the other person can pull those changes so that they have the most recent version. |
| 9 | Cloning a Repository | This is done once, to create an initial local repo of a branch on GitHub. |
| 10 | Pull Requests | Learn how to do code reviews will attempting to merge two branches |
| 11 | Eclipse GitHub Workflow 1 | Learn how to use Eclipse to manage a local repo and push and pull to/from GitHub. This is presented in the form of a workflow your team can use to get started working on your project. |
| 12 | Eclipse GitHub Workflow 2 | This is similar to Lab 11, with a few small differences. |
| 13 | Project Management | Learn how to create and use a project board to manage the work for a project. |
| 14 | GitHub Desktop | Similar to Labs 11 & 12, except that instead of using Eclipse, it uses a stand-alone software, GitHub Desktop to manage pushing and pulling. |

# Introductory Videos

Watch the videos in *a* or *b* below (or watch both):

1. Watch the four videos (about 24 minutes total) here: <https://git-scm.com/videos>, which provide a high-level overview of version control, Git, and GitHub. In subsequent labs we will explore these things in more detail.
2. Watch the P1L4-Version Control video on Udacity. You can just watch clips 1-8, & 11 (total of 20-30 min). The remainder of the clips are covered in more detail in subsequent labs. We will be watching a bunch of videos on Udacity. So, do this now or later:
3. Visit: <https://www.udacity.com/course/software-development-process--ud805>
4. Choose: Start Free Course
5. Fill out the data to signup
6. You may have to prowl around to find this video: P1L4-Version Control (if you get lost, the course is: Software Development Process)

Several optional references that might be useful:

1. The tutorial below is similar to what my tutorials will cover except that mine will go into a bit more detail.

<http://product.hubspot.com/blog/git-and-github-tutorial-for-beginners>

1. The “Pro Git” book (Chacon and Straub) is available at the link below. I have read through the first 5 chapters and I highly recommend it. <https://git-scm.com/book/en/v2>
2. GitHub.com Documentation/Guides. <https://docs.github.com/en/github>
3. <https://stackoverflow.com/questions/315911/git-for-beginners-the-definitive-practical-guide>
4. <https://stackoverflow.com/questions/tagged/git>

# Download & Install Git and Cmder

1. (Read, no action required) As stated earlier, Git is used to manage a repository on a local machine. Git consists of a set of commands which we will learn in the next few tutorials. Git can be used in three ways on a Windows machine: from the command line (or bash shell), a Gui interface, through an IDE that is Git enabled such as Eclipse and Visual Studio. In this and subsequent tutorials you will use Git only from the command line as this will give you a stronger understanding of Git and the fact that it gives you more control.
2. [Download Git](https://git-scm.com/downloads) and run the installation. I opted in for “Windows Explorer Integration,” though I don’t think we will use this. I accepted all other defaults.
3. (Read, no action required) You will use the console emulator, *Cmder*. This is very similar to the DOS command prompt; however, it has some advantages for the types of things we are going to do. As well as Git, we will need a few basic DOS commands: dir, mkdir, cd. If you are not familiar with DOS, you will easily see how these commands work in the tutorials.
4. Download and install Cmder.

**Note: we will use Cmder to do issue command line git commands. There are a number of alternatives to Cmder which you are welcome to use if you know what you are doing.**

1. [Download Cmder](https://cmder.app/) (I choose the “Full” version, though I don’t think we will use Git for Windows):
2. Unzip into a folder. (I chose: C:\cmder\_mini)
3. Navigate into that folder and right-click *Cmder.exe* and choose: Send to/Desktop (create shortcut).
4. Use the shortcut on your desktop to launch Cmder. The result will be similar to the figure shown on the right. Notes about Cmder:
5. The command prompt is: **λ**
6. Select text in the usual way by dragging the mouse and then *Ctrl+c* and *Ctrl+v* to copy and paste.
7. Use arrow keys to cycle through previous commands.
8. You can create new console tabs if needed.
9. You can change the font size (and other things) using the System Menu and then Settings.
10. To test that *git* is accessible type “git” at the prompt.

**λ**  git

As shown on the right, you should see text showing the different parameters that can be used with the *git* command.

1. To see what version of *git* you have installed:

**λ**  git --version

|  |
| --- |
| Note: You can copy/paste these commands into cmder. However, sometimes, the “-“ is copied as long dash which will not be recognized when the command tries to execute. If so, click your mouse at the end of the command, and then use the arrow keys to move back to the long dash and replace. |

1. Specify your identity in *git*:
2. Issue the command to specify your name and **gmail** address (substituting your name and email address):

**λ**  git config --global user.name "Dave Gibson"

**λ**  git config --global user.email davegibson2@gmail.com

1. ****Verify that your identity is stored properly by issuing this command:

**λ**  git config -l

1. (Reference, Optional)

Git comes with a tool called git config that lets you get and set configuration variables that control all aspects of how Git looks and operates. These variables can be stored in three different places. See: <https://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup>

1. **Do the following:**
2. Make a screen shot similar to the one in Step 7b above. Your name and email should be displayed.
3. Place the image in the *HW VCS* document in the appropriate place.
4. The image should easily readable without zooming in or out.