

COMS/SE 319: Software Construction and User Interface **Spring 2019**

Tutorial – GitLAB

1 Objectives

BEGINNER LEVEL

- Terms and Concepts (see <http://juristr.com/blog/2013/04/git-explained/>)
- Setup and Configure (2.1 and 2.2 of this document)
- Status of WD, Index, Repo
- Move to Stage and Then Repo
- File Level Undo actions
- Rename/Delete file
- Git GUI: Sourcetree (see <https://www.sourcetreeapp.com/>)

INTERMEDIATE LEVEL

- Create and Delete Branch
- Switch Branches
- Merge Branches

ADVANCED LEVEL

- Workflow

2 Setting up Git

2.1 Installing Git

Installing on Linux: Use package-management tool that comes with your distribution. If you're on Fedora for example, you can use yum:

\$ sudo yum install git

If you're on a Debian-based distribution like Ubuntu, try apt-get:

\$ sudo apt-get install git

Installing on Mac: An OSX Git installer is maintained and available for download at the Git website, at

<http://git-scm.com/download/mac>

Installing on Windows: A Windows binary installer for Git is available for download at the Git website at

<http://git-scm.com/download/win>

For detailed installation instructions refer

2.2 Configuring Git

Set the name and email for Git to use when you commit:

```
$ git config --global user.name <<"Your Name">>
```

```
$ git config --global user.email <<your-email@iastate.edu>>
```

You can execute the following command to verify these are set.

```
$ git config --list
```

The above configuration will be set globally for ALL Git projects you work with. If you want to use a different name/email address for a particular project, you can change it for just that project

- cd to the project directory
- Use the above commands, but leave out the --global

3 Warm Up: Try some basic commands

1. Create a directory/folder named **git-lab1** and navigate to this directory in your terminal (or git-bash terminal). Then execute the following command in your terminal to create a local git repository

```
git init
```

2. Create a file named **readme.txt** and type in some text in the text file.
3. To see the changes in your working directory you can execute the following

```
git status
```

4. Whenever we create a new file or modify existing files before we commit the file we must add it to staging area. We can add files to staging area by the following commands

```
git add readme.txt
```

Note: For staging all files from the current working directory you can use

```
git add .
```

5. You can list out the staged files using the following command

```
git ls-files --stage
```

6. Once we have added the desired files to the staging area we can make a commit using the following command

```
git commit -m "Initial Commit"
```

7. We can list all the commits in the local git repository using
git log --oneline
8. Edit the text in **readme.txt** and check ***git status*** (now git should show that readme.txt as modified).
9. We can list out the files that are modified by using the following command
git ls-files -m
10. Use Step 4 to stage the changes. To unstage a staged file use
git reset HEAD readme.txt
11. To un-modify a modified file use
git checkout readme.txt

Note: It's important to understand that git checkout [file] is a dangerous command. Any changes you made to that file will be gone once you use that command. You should use this command to discard your changes to the file.

12. Create a temp.txt and make a commit. To move/rename a file via git you can use
git mv <filename> <new_filename>
example: git mv temp.txt temp1.txt
13. To delete a file via git you can use
git rm <filename>
example: git rm temp1.txt

Note: When we move/delete using git, git will stage the moved/deleted files automatically.

4 Git Branches

1. The ***git branch*** command lets you create, list, rename, and delete branches. It doesn't let you switch between branches or put a forked history back together again. Create a branch named "firstBranch" using the below command
git branch <branch-name>
example: git branch firstBranch
2. To list the existing branches, you can use the following
git branch
3. Then execute the following command to delete the "firstBranch"
git branch -d <branch-name>
example: git branch -d firstBranch

Note: This is a "safe" operation in that Git prevents you from deleting the branch if it has unmerged changes.

4. Create a branch "secondBranch" using Step 1. To force delete the "secondBranch" branch, you can execute the following command

```
git branch -D <branch-name>  
git branch -D secondBranch
```

Note: The above commands force deletes the specified branch i.e. even if it has unmerged changes it will delete the branch.

5. Create a branch named "sampleBranch" using step 1. Now let's switch the branch to "sampleBranch" using the following

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

6. You can also create a branch and checkout the branch created using the following

```
git checkout -b "helloBranch"
```

7. Once you checkout a branch the workflow of modifying files, staging and making commits is same.

5 Git Merge

5.1 Scenario-1 (merging)

1. **Checkout "master" branch**
2. **Create a text file named "newfeature.txt" with some text and make a commit.**
3. **Checkout "helloBranch" and create a text file named "somefeature.txt" with some text and make a commit.**
4. **Checkout "master" branch. To merge "master" branch with "helloBranch", we will use the same command as above**
git merge helloBranch

Note: The "helloBranch" and "master" branch diverged in the above case i.e. there were changes in the "helloBranch" after branching and also there were changes in the "master" branch. In the above scenario we were changing different files in different branches so git was able to auto-merge as there were no conflicts.

5.2 Scenario-2 (handling conflict during merge)

Let's do the below scenario

1. **Checkout "master" branch.**
2. **Edit the "readme.txt" with some text and make a commit.**
3. **Checkout "helloBranch" branch.**
4. **Edit the "readme.txt" with some text and make a commit.**
5. **Checkout "master" branch. To merge "master" branch with**

"helloBranch", we will use the same command as before

6. `git merge helloBranch`

As we modified same file in both the branches Git won't be able to figure out which version to use. When such a situation occurs, it stops right before the merge commit so that you can resolve the conflicts manually. You would see something like the following

```
$ git merge helloBranch
Auto-merging readme.txt
CONFLICT (content): Merge conflict in readme.txt
Automatic merge failed; fix conflicts and then commit the result.
```

When you encounter a merge conflict, running the `git status` command shows you which files need to be resolved. For example, for the above scenario, you would see something like the following:

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:   readme.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

When you open an unmerged file, like for example "readme.txt" in the above scenario, you would see something like the following

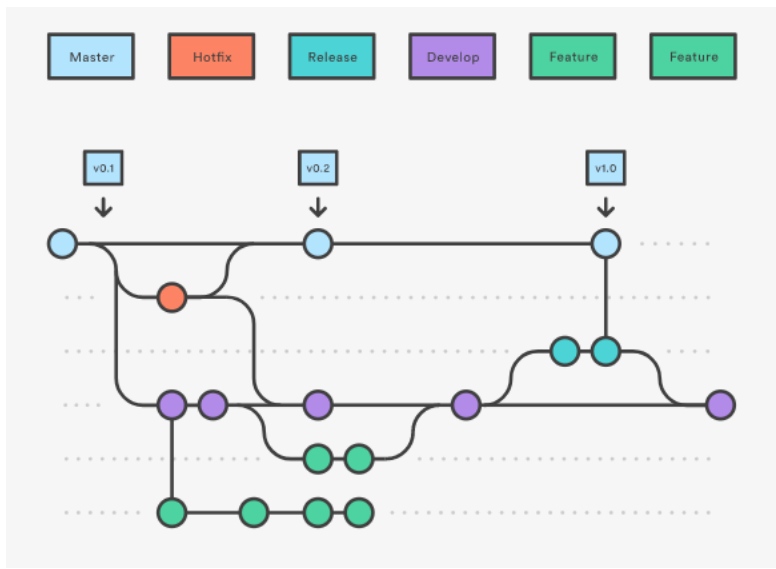
```
<<<<<< HEAD
hello from master branch
=====
hello from hellobranch
>>>>>> helloBranch
```

Then, you can fix the lines to your liking (for e.g. take changes from master and ignore the changes done in the "helloBranch" or whatever you need). When you're ready to finish the merge, all you have to do is run `git add` on the conflicted file(s) to tell Git they're resolved. Then, you run a normal `git commit` to generate the merge commit.

6 Workflow

A Git Workflow is a recipe or recommendation for how to use Git to accomplish work in a consistent and productive manner, especially for team programming.

Git Feature Branch Workflow (tutorial: <https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow>) and **Gitflow Workflow** (tutorial: <https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow>) are recommended to organize Git repo. These two workflows isolate new development from unfinished work, reduce the risk of merging codes.



Gitflow Workflow

7 Exercise


1. Create a folder with name of the format **yourLastName_yourFirstName_git_lab1** and create a git repository.
2. Create a file named **student.txt** with text as "Hello World" and make a commit with commit message "**Initial Commit**"
3. Modify the **student.txt** to have your name and commit with a commit message "**Modified student.txt**"
4. Create a file **temp.txt** with some text and modify **student.txt** to have your name in the format **lastName_FirstName**.
5. Make 2 separate commits with 1 commit having only **temp.txt** changes with a commit message "**Added Temp file**" and another commit which should have **student.txt** related changes with a commit message "**Modified student.txt 2**"
6. Rename **temp.txt** to **temp2.txt** and make a commit with a commit message "**Renamed temp.txt**".
7. Delete **temp2.txt** and make a commit with commit message "**Deleted temp2.txt**"
8. Create a branch name with format **yourLastName_experiments**. Create a file "exp.txt" with text as "**yourLastName experiments**". Make a commit and merge with "**master**" branch.
9. Simulate the merge conflict scenario (scenario 2) mentioned above with "**student.txt**" and resolve the conflict and merge with the branch you created above.

8 Git GUI

SourceTree (<https://www.sourcetreeapp.com/>) is a powerful Git desktop client for developers

using Mac or Windows.

You can also use Git GUI instead of command line to operate Git. Commit, push, pull and merge changes with the click of a button, organize your repos with the intuitive bookmarks window, and visualize how your work changes over time with SourceTree's log view.

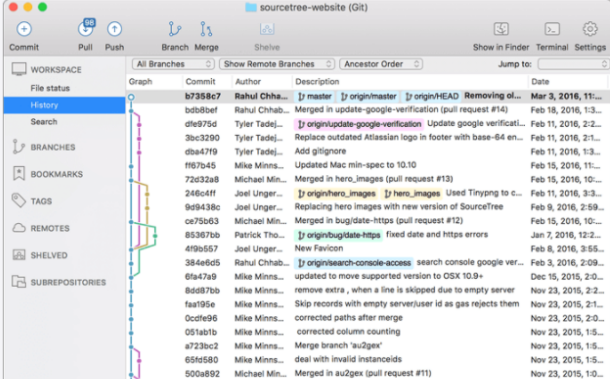
 **Sourcetree**

[Download free](#)

Simplicity and power in a beautiful Git GUI

[Download for Mac OS X](#)

Also available for Windows



The screenshot shows the Sourcetree application window for a repository named 'sourcetree-website (GIT)'. The interface includes a sidebar with navigation options: WORKSPACE, File status, History, Search, BRANCHES, BOOKMARKS, TAGS, REMOTES, SHELVED, and SUBREPOSITORIES. The main area displays a commit log with columns for Commit, Author, Description, and Date. The commit log shows a series of commits, with the most recent one being 'b7358c7' by 'Rahul Chhab...' on 'Mar 3, 2016, 11:00...'. The commit description for this commit is 'Merged in update-google-verification (pull request #14)'. The commit log also shows a branch graph on the left side of the main area.

A free Git client for Windows and Mac

Sourcetree simplifies how you interact with your Git repositories so you can focus on coding. Visualize and manage your repositories through Sourcetree's simple Git GUI.

1 Objectives

INTERMEDIATE LEVEL

- setting up remote repository
- pull
- push
- .gitignore

2 GitHub vs GitLab vs Git

GitHub: GitHub.com is a site for online storage of Git repositories. Many open source projects use it, such as the Linux kernel.

Question: Do I have to use github to use Git?

Answer: No!

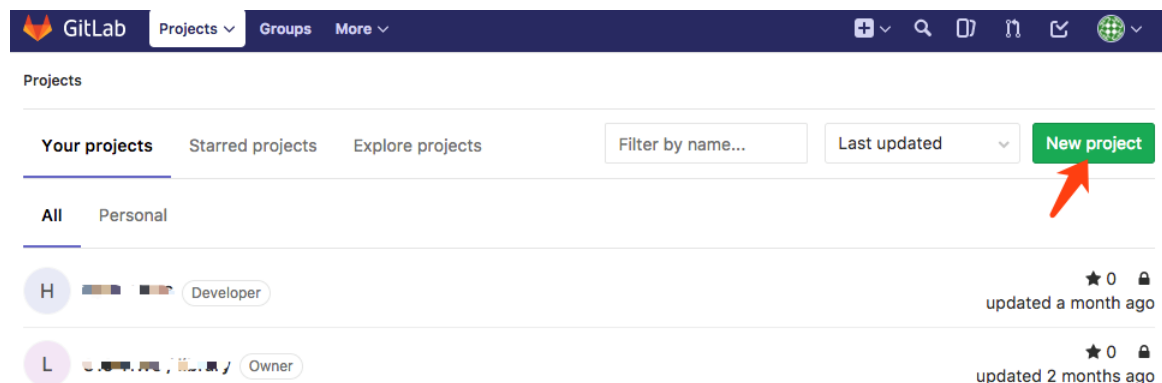
You can use Git completely locally for your own purposes, or you or someone else could set up a server to share files, or you could share a repo with users on the same file system, such as we did for lab 1.

GitLab: GitLab is a great way to manage git repositories on a centralized server. It something similar to GitHub, but each organization can host the gitlab on their own infrastructure. GitLab gives you complete control over your repositories and allows you to decide whether they are public or private for free.

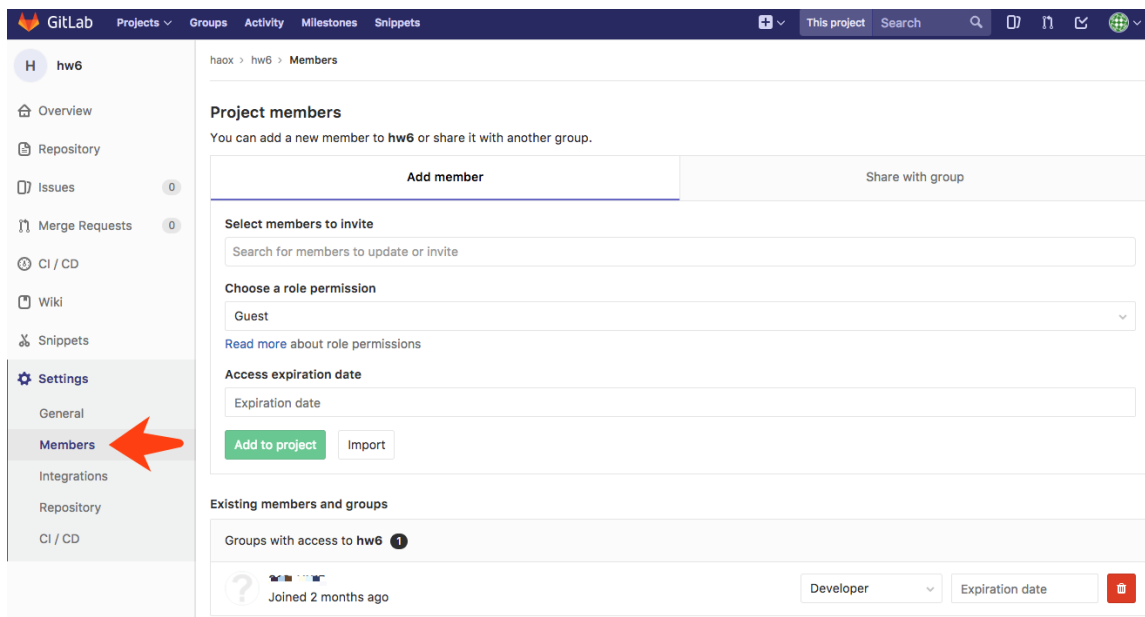
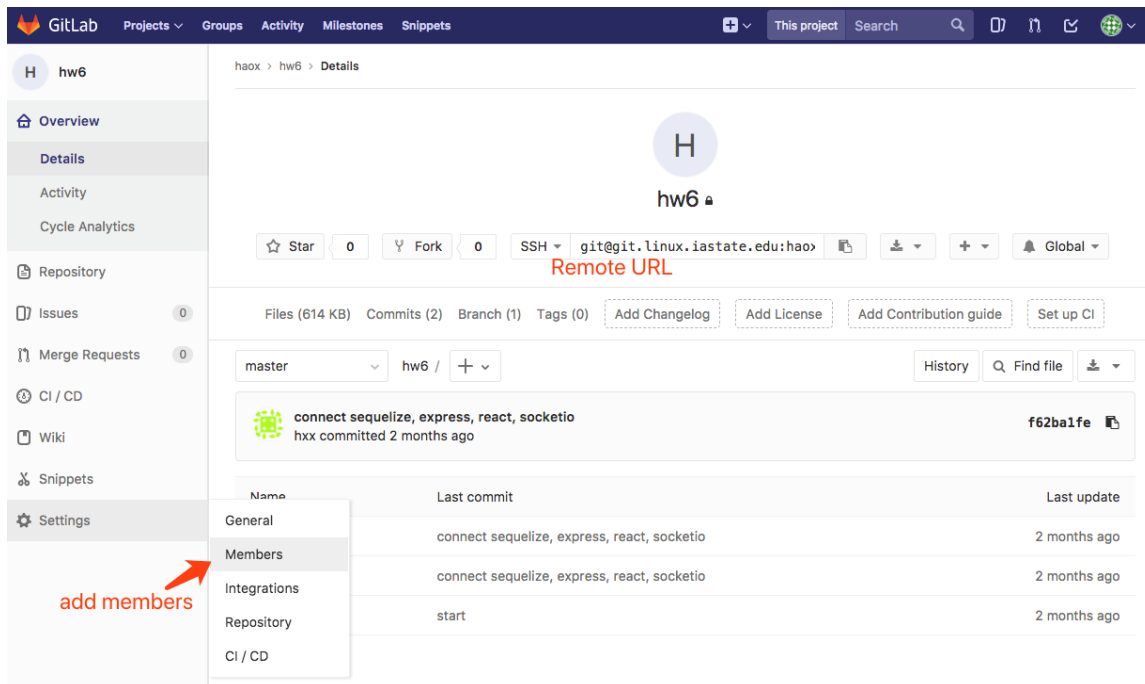
2.1 Created Remote Repository on GitLab

Log in <https://git.linux.iastate.edu/> -[GitLab Community Edition for ISU] by your ISU NetID.

NOTE: we need to Install the AnyConnect VPN Client (<https://www.it.iastate.edu/howtos/vpn>) to get access ISU's GIT server outside campus network.



After creating the project, you can add some members to your private projects.



3 Using Remote Repository

3.1 Setting Up Remote Repository

There are two ways to configure the remote repository to your local repository

1. Adding remote to exiting local repository i.e. if you have already have a git local repository you can just add the remote repository using the following command

```
git remote add <name> <remote_url>
```

Note: Many of the Git servers support two protocols (git protocol & https protocol). If the remote url starts with something like “git://some_url”, it means you are using git protocol else if it starts with “https://some_url”, means you are using https protocol. https use username/password to authenticate, whereas git protocol uses ssh keys to authenticate. To use git protocol you need to generate ssh keys and configure your ssh key in gitlab or whatever server you use. The instructions to generate ssh keys are at <http://doc.gitlab.com/ce/ssh/README.html>. For using https protocol, you can just use your net-id/password to authenticate.

Git clone: If you don’t have a local copy of the repository you can clone the repository from the remote. Doing this pulls the full repository locally, and sets up the remote connection information. You can clone the remote repository using the following command

git clone <remote_url>

Note: These make an exact copy of the repository at the given URL in the specified path.

git clone <https://git.linux.iastate.edu/309Spring2016/ProjectName.git>

3.2 Pull

Merging changes in remote repository into your local repository is a common task in Git-based collaboration workflows. You can use ***git pull*** command to do that

git pull <remote>

Fetch the specified remote’s **copy of the current branch** and immediately merge it into the local copy.

To achieve the same, we can alternatively use ***git fetch <remote>*** followed by ***git merge origin/<current-branch>***.

3.3 Push

Pushing is how you transfer commits from your local repository to a remote repository. This is the counterpart of git pull (specifically git fetch). Git push exports commits to remote branches (remote repositories).

git push <remote> <branch-name>

git push origin master

Push the specified branch to **<remote>**, along with all of the necessary commits and internal objects. This creates a local branch in the destination repository. To prevent you from overwriting commits, Git won’t let you push if remote repository has changes which your local repository doesn’t have, then git push will be rejected. Note: It’s a good practice to do a git pull first before you push.

If you need to push a local git branch to remote, you can use the following

git push -u origin feature_branch_name

To push all of your local branches to the specified remote.

git push <remote> --all

3.4 .gitignore

It's important to tell Git what files you do not want to track. Temp files, executable files, Generated files, project settings etc. do not need version control (and can cause major issues when merging!) Some examples to ignore are *.class, .project, .classpath, bin/ etc.

You can find detailed instructions to create .gitignore at <https://help.github.com/articles/ignoring-files/>

Also for various kinds of projects and platforms, languages git has some predefined. gitignore files which you can find at <https://github.com/github/gitignore>

Note: It's very important to have .gitignore to avoid merge conflicts of un-necessary files like project settings, generated code etc.

4 Exercise

1. Create a new project using your NetID on ISU GitLab.
2. Clone your remote repository to your local.
3. Create a local branch with name of the format "**yourname_experiments**".
4. Checkout the branch "**yourname_experiments**" and create a text file (**your_name.txt**) with your name in it and make a commit and push the branch to remote repository.
5. Checkout master and create a file (**README.txt**). If file is already present, simply add your name to the file and make a commit and push to "**master**".
6. Depending on your project platform/language/ide create a .gitignore file accordingly and push the .gitignore file.