## 

## 

Subject Name: **Source Code Management**

Subject Code: **CS181**

Cluster: **Beta**

       Department: **DCSE**

## logo

**Submitted By: Submitted To:**

 Abhijeet Sambharwal Dr. Monit Kapoor

 2110990040

 G1

## Index

|  |  |  |  |
| --- | --- | --- | --- |
| S.No | Aim of Experiment | Date of submission | Remarks by teacher |
| 1 | Setting up of Git client |  |  |
| 2 | Setting up GitHub account |  |  |
| 3 | Program to generate log |  |  |
| 4 | Create and visualize branches |  |  |
| 5 | Git Lifecycle description |  |  |
| 6 | Add collaborators on Github repo |  |  |
| 7 | Fork and Commit |  |  |
| 8 | Merge and Resolve conflicts created due to own activity and collaborators activity |  |  |
| 9 | Reset and Revert |  |  |
| 10 | Project |  |  |



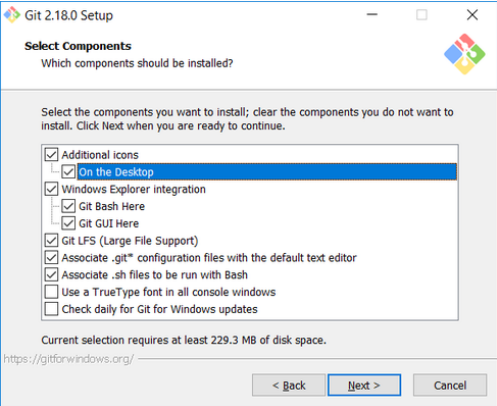
**Experiment No. 01**

**Aim:** Setting up of Git Client

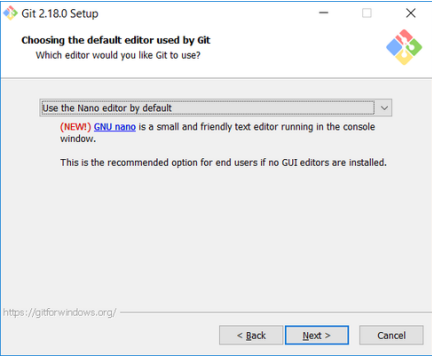
**Theory:**

* Need of GIT – As we made a code and we need to change and if we want to save it. We usually tend to copy the code and start the addition in new file. The issue in this that it takes a lot of time and uses very much space.
* What is GIT – Version Control System(VCS) 🡪 It is a software by using it we are able to track all the previous changes in the code.
* Advantages of GIT –

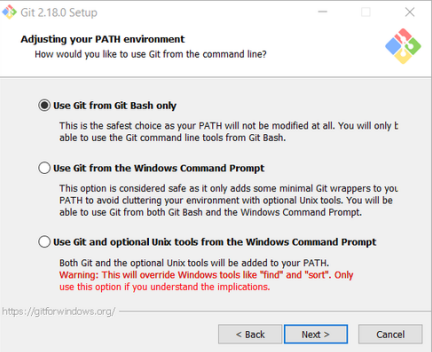
**Procedure:** 1. Git installation Download the Git installation program (Windows, Mac, or Linux) from http://git-scm.com/downloads. When running the installer, various screens appear (Windows screens shown). Generally, you can accept the default selections, except in the screens below where you do NOT want the default selections: In the Select Components screen, make sure Windows Explorer Integration is selected as shown: In the Choosing the default editor used by Git dialog, it is strongly recommended.



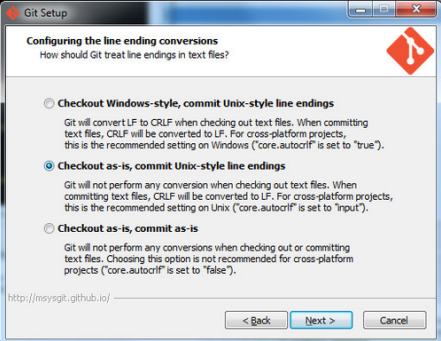
In the Choosing the default editor used by Git dialog, it is strongly recommended that you DO NOT select the default VIM editor - it is challenging to learn how to use it, and there are better modern editors available. Instead, choose Notepad++ or Nano - either of those is much easier to use. It is strongly recommended that you select Notepad++, BUT YOU MUST INSTALL NOTEPAD++ first! Find the installation with Google.



In the Adjusting your PATH screen, all three options are acceptable: Use Git from Git Bash only: no integration, and no extra commands in your command path Use Git from the Windows Command Prompt: adds flexibility - you can simply run git from a Windows command prompt, and is often the setting for people in industry - but this does add some extra commands. Use Git and optional Unix tools from the Windows Command Prompt: this is also a robust choice and useful if you like to use Unix commands like grep.

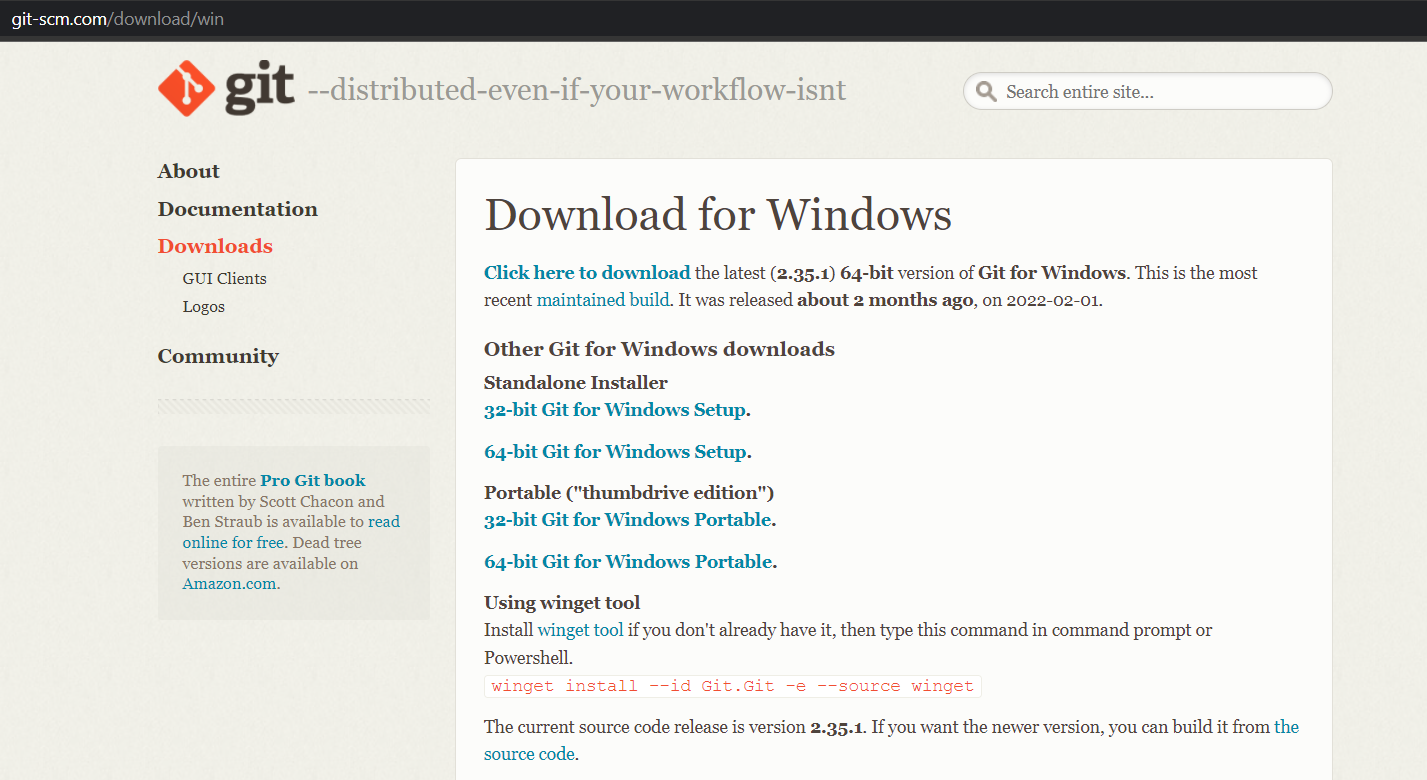


In the Configuring the line ending screen, select the middle option (Checkout as-is, commit Unix-style line endings) as shown. This helps migrate files towards the Unix-style (LF) terminators that most modern IDE's and editors support.The Windows convention (CR-LF line termination) is only important for Notepad (as opposed to Notepad++), but if you are using Notepad to edit your code you may need to ask your instructor for help.

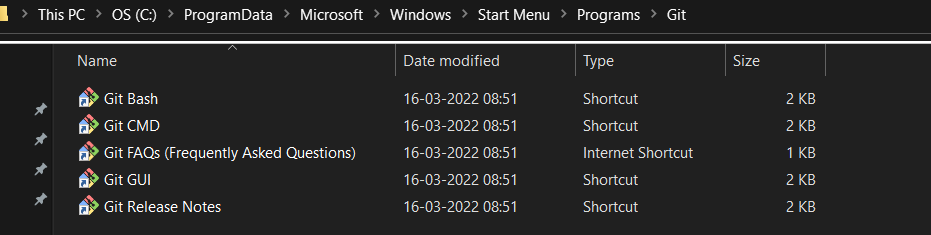


**Snapshots of download:**

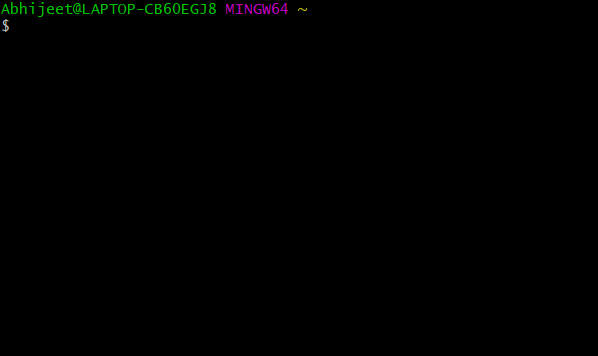
Snapshot of the site from where we have to download:

****

Location Of Git In my system:



Interface of GitBash:





**Experiment No. 02**

**Aim:** Setting up GitHub Account

**Theory:**

* **What is GitHub -** At a high level, GitHub is a website and cloud-based service that helps developers store and manage their code, as well as track and control changes to their code.
* **Advantages of GitHub -** GitHub’s interface is user-friendly enough so even novice coders can take advantage of Git. Without GitHub, using Git generally requires a bit more technical savvy and use of the command line. Additionally, anyone can sign up and host a public code repository for free, which makes GitHub especially popular with open-source projects.

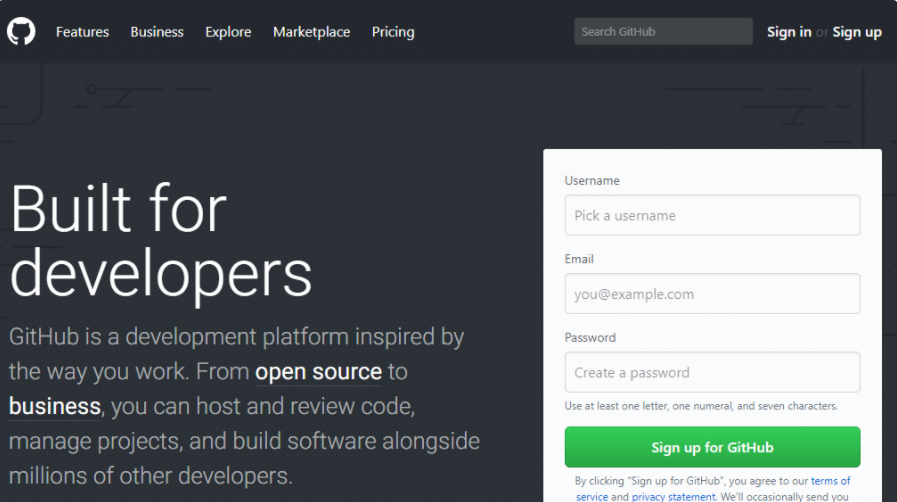
**Procedure:** The first steps in starting with GitHub are to create an account, choose a product that fits your needs best, verify your email, set up two-factor authentication, and view your profile.

There are several types of accounts on GitHub. Every person who uses GitHub has their own user account, which can be part of multiple organizations and teams. Your user account is your identity on GitHub.com and represents you as an individual.

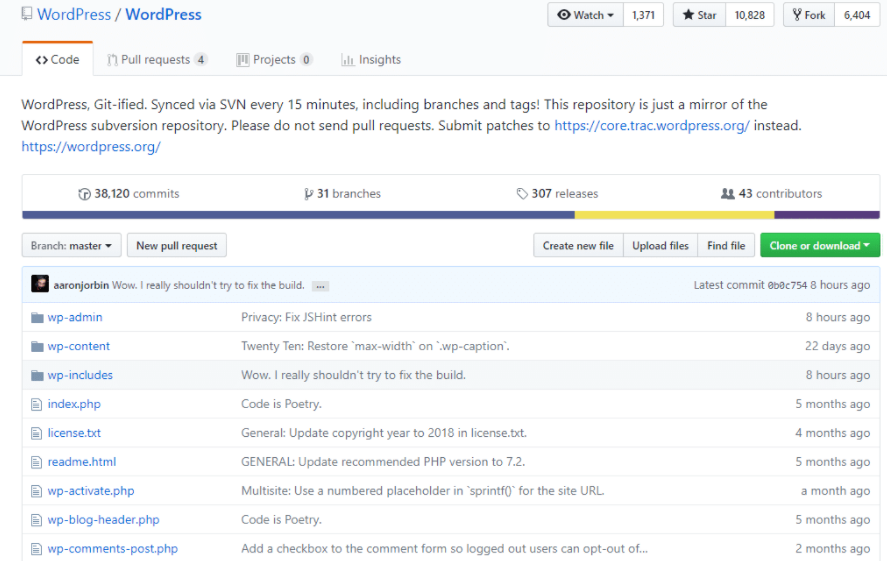
* **Creating an Account:** To sign up for an account on GitHub.com, navigate to <https://github.com/> and follow the prompts. To keep your GitHub account secure you should use a strong and unique password. For more information, see "[Creating a strong password](https://docs.github.com/en/github/authenticating-to-github/keeping-your-account-and-data-secure/creating-a-strong-password)."
* **Choosing your GitHub product:** You can choose GitHub Free or GitHub Pro to get access to different features for your personal account. You can upgrade at any time if you are unsure at first which product you want. For more information on all of GitHub's plans, see "[GitHub's products](https://docs.github.com/en/get-started/learning-about-github/githubs-products)."
* **Verifying your email address:** To ensure you can use all the features in your GitHub plan, verify your email address after signing up for a new account. For more information, see "[Verifying your email address](https://docs.github.com/en/github/getting-started-with-github/signing-up-for-github/verifying-your-email-address)."
* **Configuring two-factor authentication:** Two-factor authentication, or 2FA, is an extra layer of security used when logging into websites or apps. We strongly urge you to configure 2FA for the safety of your account. For more information, see "[About two-factor authentication](https://docs.github.com/en/github/authenticating-to-github/securing-your-account-with-two-factor-authentication-2fa/about-two-factor-authentication)."
* **Viewing your GitHub profile and contribution graph:** Your GitHub profile tells people the story of your work through the repositories and gists you've pinned, the organization memberships you've chosen to publicize, the contributions you've made, and the projects you've created. For more information, see "[About your profile](https://docs.github.com/en/github/setting-up-and-managing-your-github-profile/customizing-your-profile/about-your-profile)" and "[Viewing contributions on your profile](https://docs.github.com/en/github/setting-up-and-managing-your-github-profile/managing-contribution-graphs-on-your-profile/viewing-contributions-on-your-profile)."

**Snapshots** –

Sign-up Page of GitHub:



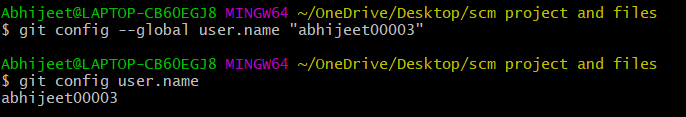
Interface of GitHub:



**To link GitHub account with Git bash –**

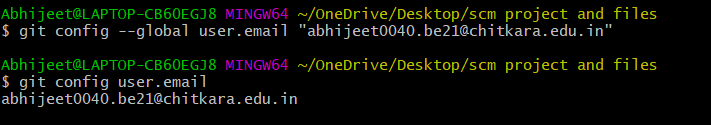
For username:

git config --global user.name “username”



For user email:

git config --global user.email “example@mail.com”



To verify:

git config user.name

git config user.email



**Experiment No. 03**

**Aim:** Program to Generate log

**Theory:-**

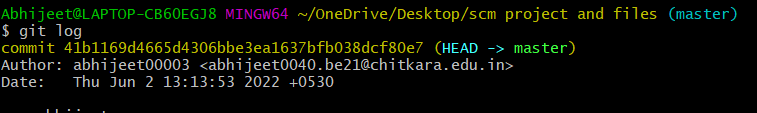
**Logs ->** Logs are nothing but the history which we can see in git by using the code git log.

It contains all the past commits, insertions and deletions in it which we can see any time.

**Why logs ->**  Logs helps to check that what were the changes in the code or any other file and by whom. It also contains the number of insertions and deletions including at which time it was changed.

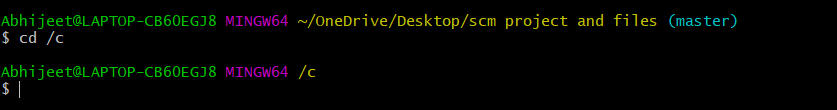
**Snapshots -**

git log:



pwd: It tells the present working directory that’s at which location it is present.

cd: It is sed to change the directory





**Experiment No. 04**

**Aim:** Create and visualize branches

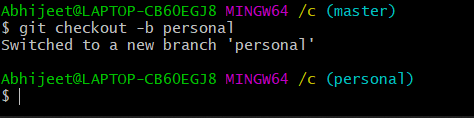
**Theory:** A branch represents an independent line of development. Branches serve as an abstraction for the edit/stage/commit process. You can think of them as a way to request a brand new working directory, staging area, and project history. New commits are recorded in the history for the current branch, which results in a fork in the history of the project.

**Procedure to make branches:**

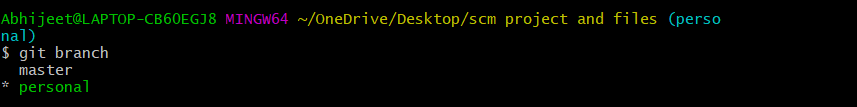
The main branch in git is called as master branch. But we can make branches out of this main master branch. All the files present in master can be shown in branch but the file which are created in branch are not shown in master branch. We can also merge both the parent (master) and child (other branches).

**Snapshots -**

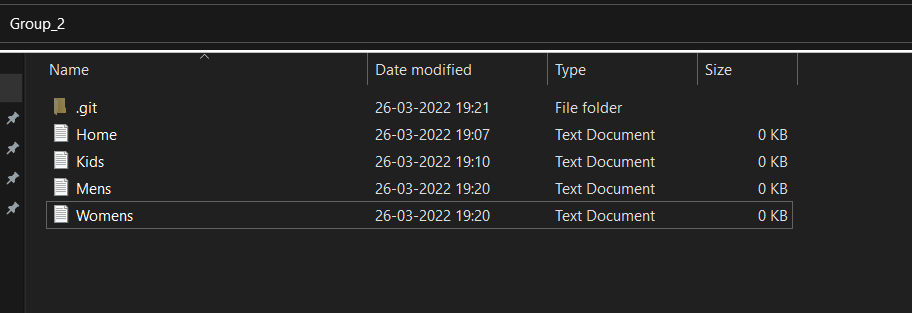
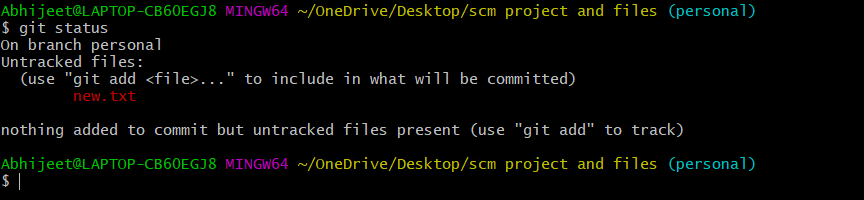
To change from one branch to another syntax:



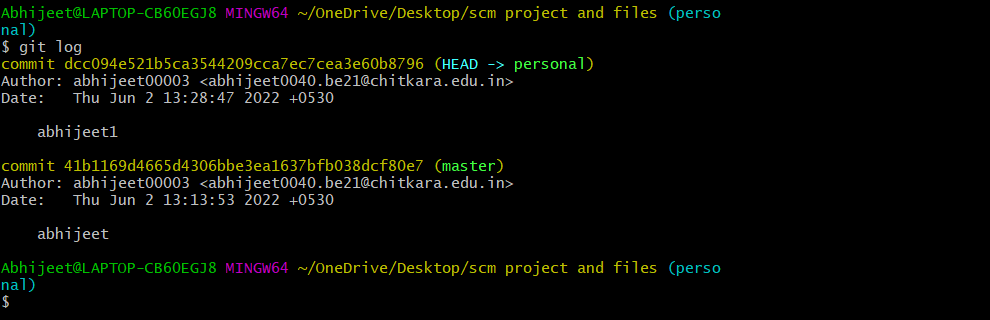
To see number of branches present syntax:



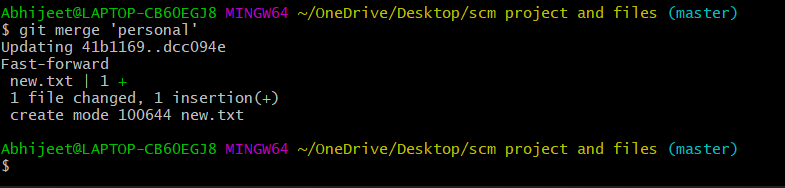
Added folders in personal1 branch:



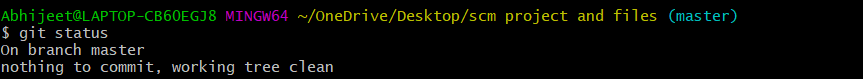
Git log:



Syntax to merge both branches:



git status for personal branch:

****

Syntax to link file from to GitHub to system:



**Experiment No. 05**

**Aim:** Git lifecycle description

**Theory:**

**Stages in GIT Life Cycle -** Files in a Git project have various stages like Creation, Modification, Refactoring, and Deletion and so on. Irrespective of whether this project is tracked by Git or not, these phases are still prevalent. However, when a project is under Git version control system, they are present in three major Git states in addition to these basic ones. Here are the three Git states:

* Working directory
* Staging area
* Git directory

### **Working Directory -**

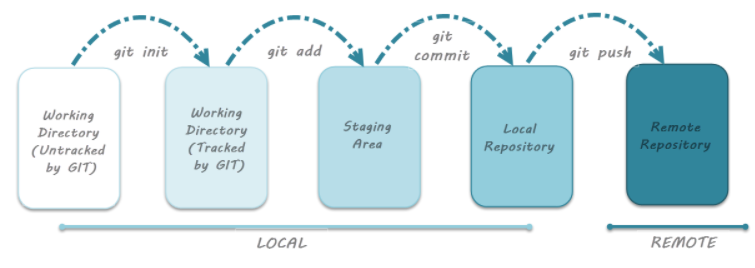
Consider a project residing in your local system. This project may or may not be tracked by Git. In either case, this project directory is called your Working directory.

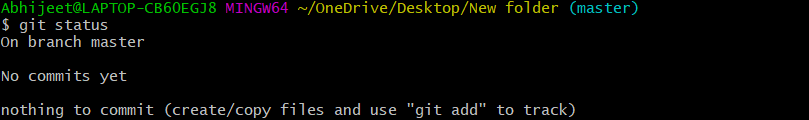
### **Staging Area -**

**Staging area is the playground where you group, add and organize the files to be committed to Git for tracking their versions.**

### **Git Directory -**

Now that the files to be committed are grouped and ready in the staging area, we can commit these files. So, we commit this group of files along with a commit message explaining what is the commit about. Apart from commit message, this step also records the author and time of the commit. Now, a snapshot of the files in the commit is recorded by Git. The information related to this commit is stored in the Git directory.

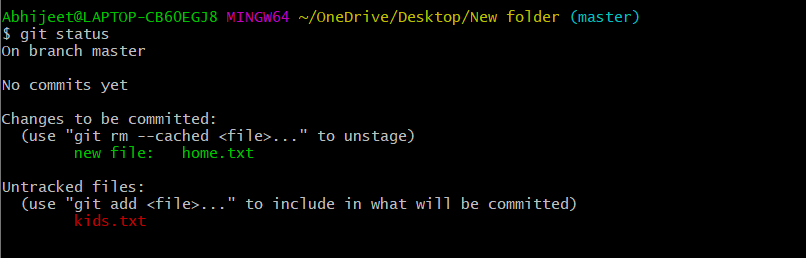
**Remote Repository means mirror or clone of the local Git repository in GitHub**. And **pushing means uploading the commits from local Git repository to remote repository hosted in GitHub.**

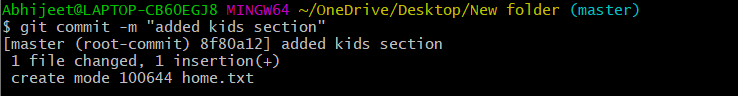
**Snapshots -**

Staging:

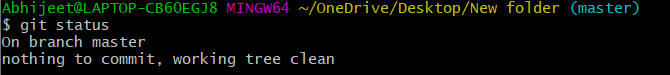
****

Committing:

****

****

git status: work tree clean means that commit we have done has been saved for lifetime.

****



**Experiment No. 06**

**Aim:** Add collaborators on Github Repo

To accept access to the Owner’s repo, the Collaborator needs to go to <https://github.com/notifications>or check for email notification. Once there she can accept access to the Owner’s repo.

Next, the Collaborator needs to download a copy of the Owner’s repository to her machine. This is called “cloning a repo”.

The Collaborator doesn’t want to overwrite her own version of planets.git, so needs to clone the Owner’s repository to a different location than her own repository with the same name.

To clone the Owner’s repo into her Desktop folder, the Collaborator enters:

$ git clone git@github.com:vlad/planets.git ~/Desktop/vlad-planets

Replace ‘vlad’ with the Owner’s username.

If you choose to clone without the clone path (~/Desktop/vlad-planets) specified at the end, you will clone inside your own planets folder! Make sure to navigate to the Desktop folder first.

The Collaborator can now make a change in her clone of the Owner’s repository, exactly the same way as we’ve been doing before:

$ cd ~/Desktop/vlad-planets

$ nano pluto.txt

$ cat pluto.txt

It is so a planet!

$ git add pluto.txt

$ git commit **-m** "Add notes about Pluto"

1 file changed, 1 insertion(+) create mode 100644 pluto.txt

Then push the change to the *Owner’s repository* on GitHub:

$ git push origin main

Enumerating objects: 4, done. Counting objects: 4, done.

Delta compression using up to 4 threads. Compressing objects: 100% (2/2), done.

Writing objects: 100% (3/3), 306 bytes, done. Total 3 (delta 0), reused 0 (delta 0)

To https://github.com/vlad/planets.git 9272da5..29aba7c main -> main

Note that we didn’t have to create a remote called origin: Git uses this name by default when we clone a repository. (This is why origin was a sensible choice earlier when we were setting up remotes by hand. Take a look at the Owner’s repository on GitHub again, and you should be able to see the new commit made by the Collaborator. You may need to refresh your browser to see the new commit.

# **Some more about remotes**

In this episode and the previous one, our local repository has had a single “remote”, called origin. A remote is a copy of the repository that is hosted somewhere else, that we can push to and pull from, and there’s no reason that you have to work with only one. For example, on some large projects you might have your own copy in your own GitHub account (you’d probably call this origin) and also the main “upstream” project repository (let’s call this upstream for the sake of examples). You would pull from upstream from time to time to get the latest updates that other people have committed. Remember that the name you give to a remote only exists locally. It’s an alias that you choose - whether origin, or upstream, or fred - and not something intrinstic to the remote repository.

The git remote family of commands is used to set up and alter the remotes associated with a repository. Here are some of the most useful ones:

* git remote -v lists all the remotes that are configured (we already used this in the last episode)
* git remote add [name] [url] is used to add a new remote
* git remote remove [name] removes a remote. Note that it doesn’t affect the remote repository at all - it just removes the link to it from the local repo.
* git remote set-url [name] [newurl] changes the URL that is associated with the remote. This is useful if it has moved, e.g. to a different GitHub account, or from GitHub to a different hosting service. Or, if we made a typo when adding it!
* git remote rename [oldname] [newname] changes the local alias by which a remote is known - its name. For example, one could use this to change upstream to fred.

To download the Collaborator’s changes from GitHub, the Owner now enters:

$ git pull origin main

remote: Enumerating objects: 4, done. remote: Counting objects: 100% (4/4), done.

remote: Compressing objects: 100% (2/2), done.

remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0 Unpacking objects: 100% (3/3), done.

From https://github.com/vlad/planets

\* branch main

9272da5..29aba7c main

Updating 9272da5..29aba7c Fast-forward

pluto.txt | 1 +

-> FETCH\_HEAD

-> origin/main

1 file changed, 1 insertion(+)

create mode 100644 pluto.txt

Now the three repositories (Owner’s local, Collaborator’s local, and Owner’s on GitHub) are back in sync.

# **A Basic Collaborative Workflow**

In practice, it is good to be sure that you have an updated version of the repository you are collaborating on, so you should git pull before making our changes. The basic collaborative workflow would be:

* update your local repo with git pull origin main,
* make your changes and stage them with git add,
* commit your changes with git commit -m, and
* upload the changes to GitHub with git push origin main

It is better to make many commits with smaller changes rather than of one commit with massive changes: small commits are easier to read and review.

# Switch Roles and Repeat-

Switch roles and repeat the whole process.

# Review Changes-

The Owner pushed commits to the repository without giving any information to the Collaborator. How can the Collaborator find out what has changed with command line? And on GitHub?

Comment Changes in GitHub-

The Collaborator has some questions about one line change made by the Owner and has some suggestions to propose. With GitHub, it is possible to comment the diff of a commit. Over the line of code to comment, a blue comment icon appears to open a comment window. The Collaborator posts its comments and suggestions using GitHub interface.

# Version History, Backup, and Version Control-

Some backup software can keep a history of the versions of your files. They also allows you to recover specific versions. How is this functionality different from version control? What are some of the benefits of using version control, Git and GitHub?



**Experiment No. 07**

**Aim:** Fork and Commit

# **About forks**

# Most commonly, forks are used to either propose changes to someone else's project or to use someone else's project as a starting point for your own idea. You can fork a repository to create a copy of the repository and make changes without affecting the upstream repository. For more information, see "[Working with forks](https://docs.github.com/en/github/collaborating-with-issues-and-pull-requests/working-with-forks)."

## Propose changes to someone else's project

## For example, you can use forks to propose changes related to fixing a bug. Rather than logging an issue for a bug you've found, you can:

* Fork the repository.
* Make the fix.
* Submit a pull request to the project owner.

**Use someone else's project as a starting point for your own idea**

Open source software is based on the idea that by sharing code, we can make better, more reliable software. For more information, see the "[About the Open Source Initiative](http://opensource.org/about)" on the Open Source Initiative.For more information about applying open source principles to your organization's development work on GitHub.com, see GitHub's white paper "[An introduction to](https://resources.github.com/whitepapers/introduction-to-innersource/) [innersource](https://resources.github.com/whitepapers/introduction-to-innersource/)."

When creating your public repository from a fork of someone's project, make sure to include a license file that determines how you want your project to be shared with others. For more information, see "[Choose an open source license](https://choosealicense.com/)" at choosealicense.com.

For more information on open source, specifically how to create and grow an open source project, we've created [Open Source Guides](https://opensource.guide/) that will help you foster a healthy open source community by recommending best practices for creating and maintaining repositories for your open source project. You can also take a free [GitHub Learning](https://lab.github.com/) [Lab](https://lab.github.com/) course on maintaining open source communities.

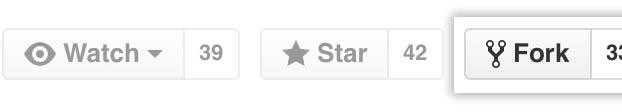
# **Prerequisites**

If you haven't yet, you should first [set up Git.](https://docs.github.com/en/articles/set-up-git) Don't forget to [set up authentication to](https://docs.github.com/en/articles/set-up-git#next-steps-authenticating-with-github-from-git) [GitHub.com from Git](https://docs.github.com/en/articles/set-up-git#next-steps-authenticating-with-github-from-git) as well.

# **Forking a repository**

# You might fork a project to propose changes to the upstream, or original, repository. In this case, it's good practice to regularly sync your fork with the upstream repository. To do this, you'll need to use Git on the command line. You can practice setting the upstream repository using the same [octocat/Spoon-Knife](https://github.com/octocat/Spoon-Knife) repository you just forked.

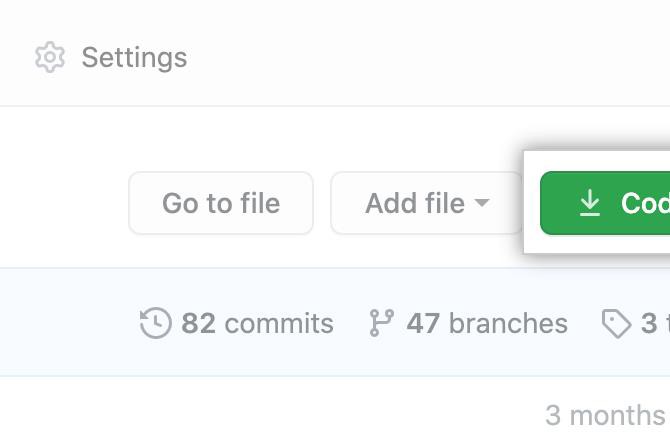
* On GitHub.com, navigate to the [octocat/Spoon-Knife](https://github.com/octocat/Spoon-Knife) repository.
* In the top-right corner of the page, click Fork.



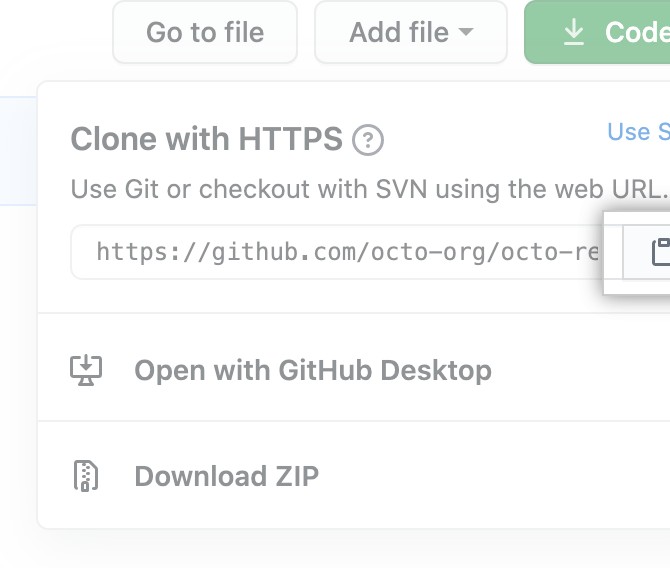
# **Cloning your forked repository**

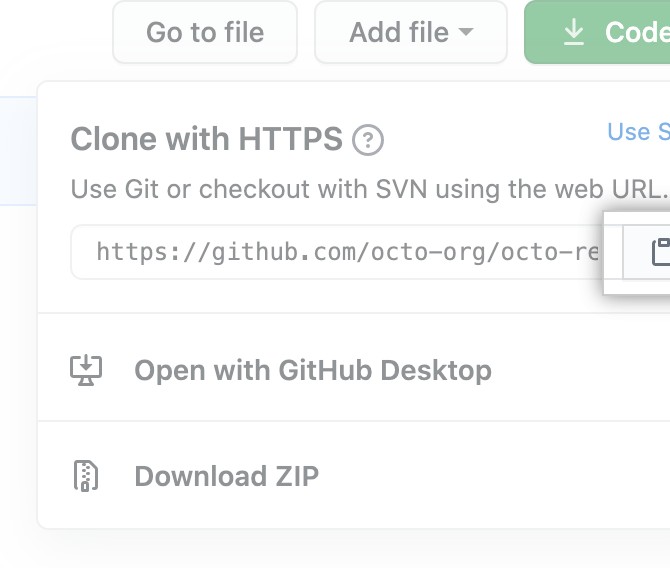
Right now, you have a fork of the Spoon-Knife repository, but you don't have the files in that repository locally on your computer.

* On GitHub.com, navigate to your fork of the Spoon-Knife repository.
* Above the list of files, click Code



* To clone the repository using HTTPS, under "Clone with HTTPS", click . To clone the repository using an SSH key, including a certificate issued by your organization's SSH certificate authority, click **Use SSH**, then click . To clone a repository using GitHub CLI, click **Use GitHub CLI**, then click .



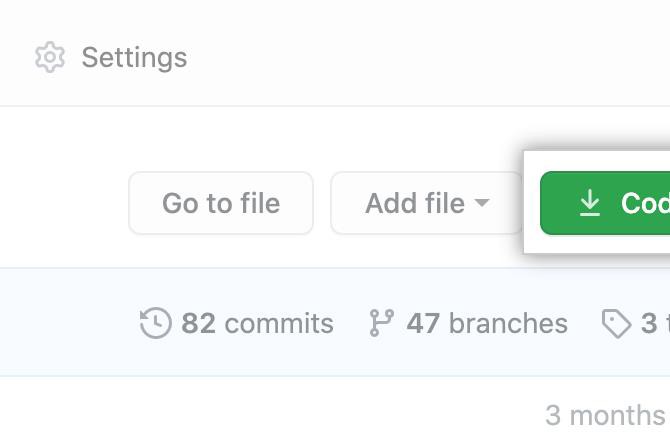


* Open Git Bash.
* Change the current working directory to the location where you want the cloned directory.
* Type git clone, and then paste the URL you copied earlier. It will look like this, with your GitHub username instead of YOUR-USERNAME:
* $ git clone https://github.com/*YOUR-USERNAME*/Spoon-Knife
* Press Enter. Your local clone will be created.
* $ git clone https://github.com/*YOUR-USERNAME*/Spoon-Knife
* > Cloning into `Spoon-Knife`...
* > remote: Counting objects: 10, done.
* > remote: Compressing objects: 100% (8/8), done.
* > remove: Total 10 (delta 1), reused 10 (delta 1)
* > Unpacking objects: 100% (10/10), done.

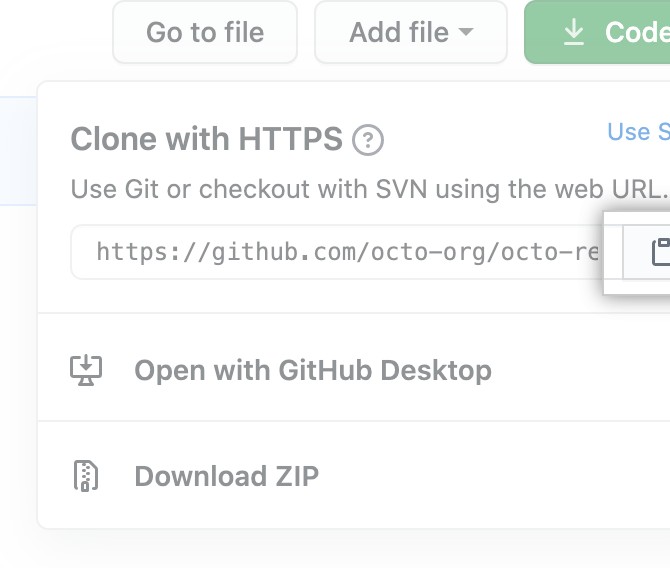
# Configuring Git to sync your fork with the original repository

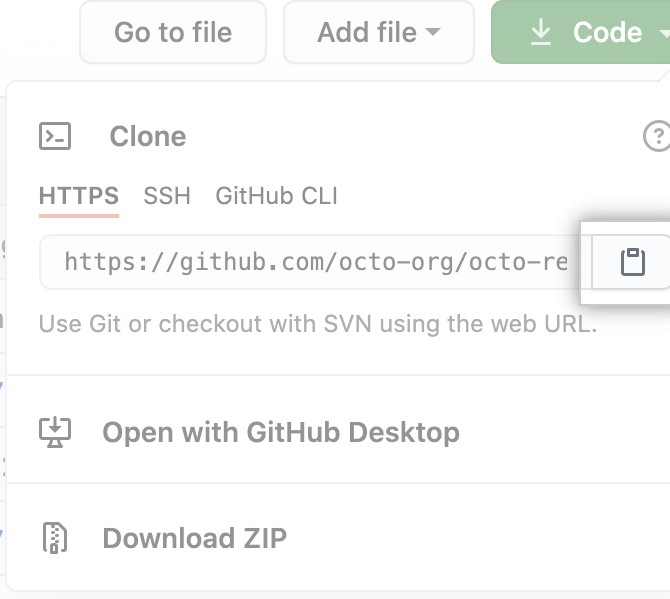
# When you fork a project in order to propose changes to the original repository, you can configure Git to pull changes from the original, or upstream, repository into the local clone of your fork.

1. On GitHub.com, navigate to the [octocat/Spoon-Knife](https://github.com/octocat/Spoon-Knife) repository.
2. Above the list of files, click Code



1. To clone the repository using HTTPS, under "Clone with HTTPS", click . To clone the repository using an SSH key, including a certificate issued by your organization's SSH certificate authority, click **Use SSH**, then click . To clone a repository using GitHub CLI, click **Use GitHub CLI**, then click.





1. Open Git Bash.
2. Change directories to the location of the fork you cloned.
   * To go to your home directory, type just cd with no other text.
   * To list the files and folders in your current directory, type ls.
   * To go into one of your listed directories, type cd your\_listed\_directory.
   * To go up one directory, type cd ...
3. Type git remote -v and press Enter. You'll see the current configured remote repository for your fork.
4. $ git remote -v
5. > origin https://github.com/YOUR\_USERNAME/YOUR\_FORK.git (fetch)

> origin https://github.com/YOUR\_USERNAME/YOUR\_FORK.git (push)

9. Type git remote add upstream, and then paste the URL you copied in Step 2 and press Enter. It will look like this:

$ git remote add upstream https://github.com/octocat/Spoon-Knife.git

1. To verify the new upstream repository you've specified for your fork, type git remote -v again. You should see the URL for your fork as origin, and the URL for the original repository as upstream.
2. $ git remote -v
3. > origin https://github.com/YOUR\_USERNAME/YOUR\_FORK.git (fetch)
4. > origin https://github.com/YOUR\_USERNAME/YOUR\_FORK.git (push)
5. > upstream https://github.com/ORIGINAL\_OWNER/ORIGINAL\_REPOSITORY.git (fetch)

> upstream https://github.com/ORIGINAL\_OWNER/ORIGINAL\_REPOSITORY.git (push)

Now, you can keep your fork synced with the upstream repository with a few Git commands. For more information, see "[Syncing a fork](https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/working-with-forks/syncing-a-fork)."

## Next steps

## You can make any changes to a fork, including:

* Creating branches: [*Branches*](https://docs.github.com/en/articles/creating-and-deleting-branches-within-your-repository) allow you to build new features or test out ideas without putting your main project at risk.
* Opening pull requests: If you are hoping to contribute back to the original repository, you can send a request to the original author to pull your fork into their repository by submitting a [pull request.](https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests)

# **Find another repository to fork**

# Fork a repository to start contributing to a project. You can fork a repository to your user account or any organization where you have repository creation permissions. For more information, see "[Roles in an organization](https://docs.github.com/en/organizations/managing-peoples-access-to-your-organization-with-roles/roles-in-an-organization)." If you have access to a private repository and the owner permits forking, you can fork the repository to your user account or any organization on GitHub Team where you have repository creation permissions. You cannot fork a private repository to an organization using GitHub Free. For more information, see "[GitHub's products](https://docs.github.com/en/articles/githubs-products)."

You can browse [Explore](https://github.com/explore) to find projects and start contributing to open source repositories. For more information, see "[Finding ways to contribute to open source on](https://docs.github.com/en/github/getting-started-with-github/finding-ways-to-contribute-to-open-source-on-github) [GitHub](https://docs.github.com/en/github/getting-started-with-github/finding-ways-to-contribute-to-open-source-on-github)."

# **Celebrate**

You have now forked a repository, practiced cloning your fork, and configured an upstream repository. For more information about cloning the fork and syncing the changes in a forked repository from your computer see "[Set up Git](https://docs.github.com/en/articles/set-up-git)." You can also create a new repository where you can put all your projects and share the code on GitHub. For more information see, "[Create a repository](https://docs.github.com/en/articles/create-a-repo)."

Each repository in GitHub is owned by a person or an organization. You can interact with the people, repositories, and organizations by connecting and following them on GitHub. For more information see "[Be social](https://docs.github.com/en/articles/be-social)." GitHub has a great support community where you can ask for help and talk to people from around the world. Join the conversation on [Github Support Community.](https://github.community/)



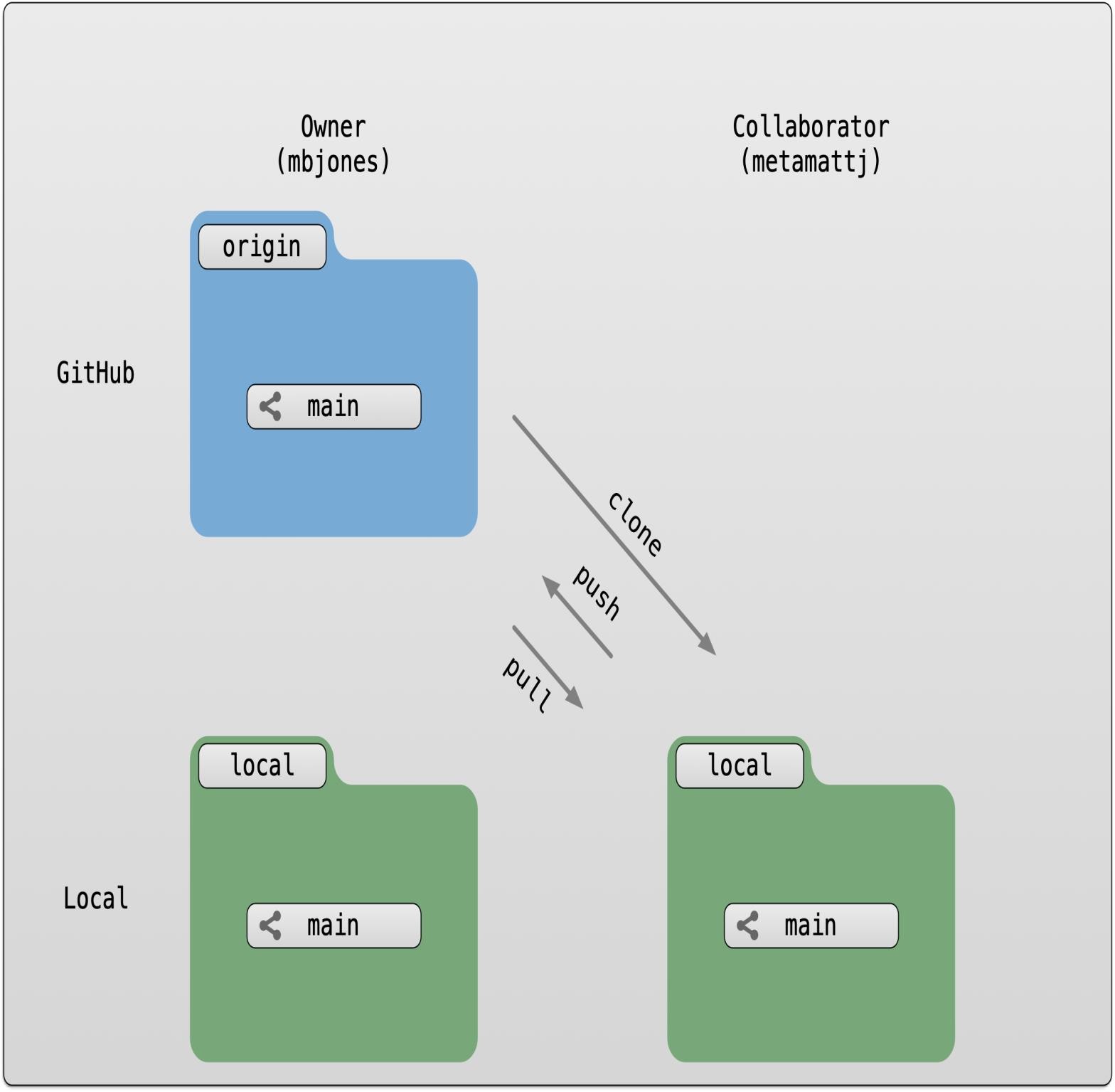
**Experiment No. 08**

**Aim:** Merge and Resolve conflicts created due to own activity and collaborators activity

Git is a great tool for working on your own, but even better for working with friends and colleagues. Git allows you to work with confidence on your own local copy of files with the confidence that you will be able to successfully synchronize your changes with the changes made by others.

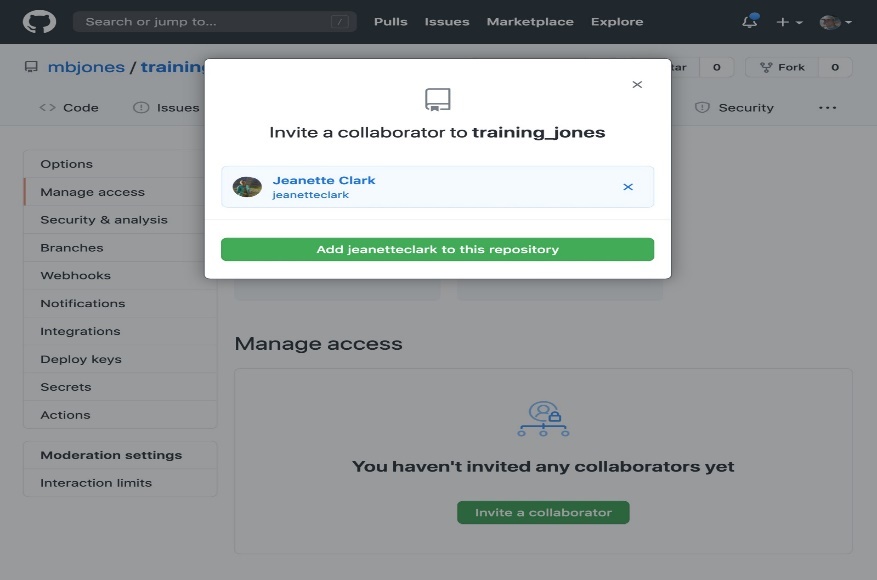
The simplest way to collaborate with Git is to use a shared repository on a hosting service such as [GitHub,](https://github.com/) and use this shared repository as the mechanism to move changes from one collaborator to another. While there are other more advanced ways to sync git repositories, this “hub and spoke” model works really well due to its simplicity.

In this model, the collaborator will clone a copy of the owner’s repository from GitHub, and the owner will grant them collaborator status, enabling the collaborator to directly pull and push from the owner’s GitHub repository.



#### **Collaborating with a trusted colleague without conflicts**

We start by enabling collaboration with a trusted colleague. We will designate the Owner as the person who owns the shared repository, and the Collaborator as the person that they wish to grant the ability to make changes to their reposity. We start by giving that person access to our GitHub repository.

****

We will start by having the collaborator make some changes and share those with the Owner without generating any conflicts, In an ideal world, this would be the normal workflow. Here are the typical steps.

##### **Step 1: Collaborator clone**

##### To be able to contribute to a repository, the collaborator must clone the repository from the Owner’s github account.

##### To do this, the Collaborator should visit the github page for the Owner’s repository, and then copy the clone URL.

##### In R Studio, the Collaborator will create a new project from version control by pasting this clone URL into the

##### appropriate dialog (see the earlier chapter introducing GitHub).

##### **Step 2: Collaborator Edits**

With a clone copied locally, the Collaborator can now make changes to the index.Rmd file in the repository, adding a line or statment somewhere noticeable near the top. Save your changes.

##### **Step 3: Collaborator commit and push**

To sync changes, the collaborator will need to add, commit, and push their changes to the Owner’s repository. But before doing so, its good practice to pull immediately before committing to ensure you have the most recent changes from the owner. So, in R Studio’s Git tab, first click the “Diff” button to open the git window, and then press the green “Pull” down arrow button. This will fetch any recent changes from the origin repository and merge them. Next, add the changed index.Rmd file to be committed by cicking the checkbox next to it, type in a commit message, and click ‘Commit.’ Once that finishes, then the collaborator can immediately click ‘Push’ to send the commits to the Owner’s GitHub repository.

##### **Step 4: Owner pull**

Now, the owner can open their local working copy of the code in RStudio, and pull those changes down to their local copy. Congrats, the owner now has your changes!

##### **Step 5: Owner edits, commit, and push**

Next, the owner should do the same. Make changes to a file in the repository, save it, pull to make sure no new changes have been made while editing, and then add, commit, and push the Owner changes to GitHub.

##### **Step 6: Collaborator pull**

The collaborator can now pull down those owner changes, and all copies are once again fully synced. And you’re off to collaborating.

**Challenge**

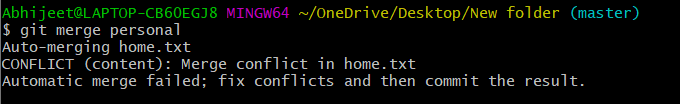
Now that the instructors have demonstrated this conflict-free process, break into pairs and try the same with your partner. Start by designating one person as the Owner and one as the Collaborator, and then repeat the steps described above:

* Step 0: Setup permissions for your collaborator
* Step 1: Collaborator clones the Owner repository
* Step 2: Collaborator Edits the README file
* Step 3: Collaborator commits and pushes the file to GitHub
* Step 4: Owner pulls the changes that the Collaborator made
* Step 5: Owner edits, commits, and pushes some new changes
* Step 6: Collaborator pulls the owners changes from GitHub

#### **Merge conflicts**

#### So things can go wrong, which usually starts with a merge conflict, due to both collaborators making incompatible changes to a file. While the error messages from merge conflicts can be daunting, getting things back to a normal state can be straightforward once you’ve got an idea where the problem lies.

A merge conflict occurs when both the owner and collaborator change the same lines in the same file without first pulling the changes that the other has made. This is most easily avoided by good communication about who is working on various sections of each file, and trying to avoid overlaps. But sometimes it happens, and git is there to warn you about potential problems. And git will not allow you to overwrite one person’s changes to a file with another’s changes to the same file if they were based on the same version.



The main problem with merge conflicts is that, when the Owner and Collaborator both make changes to the same line of a file, git doesn’t know whose changes take precedence. You have to tell git whose changes to use for that line.

#### **How to resolve a conflict**

##### Abort, abort, abort…

##### Sometimes you just made a mistake. When you get a merge conflict, the repository is placed in a ‘Merging’ state until you resolve it. There’s a commandline command to abort doing the merge altogether:

git merge --abort

Of course, after doing that you stull haven’t synced with your collaborator’s changes, so things are still unresolved. But at least your repository is now usable on your local machine.

##### Checkout

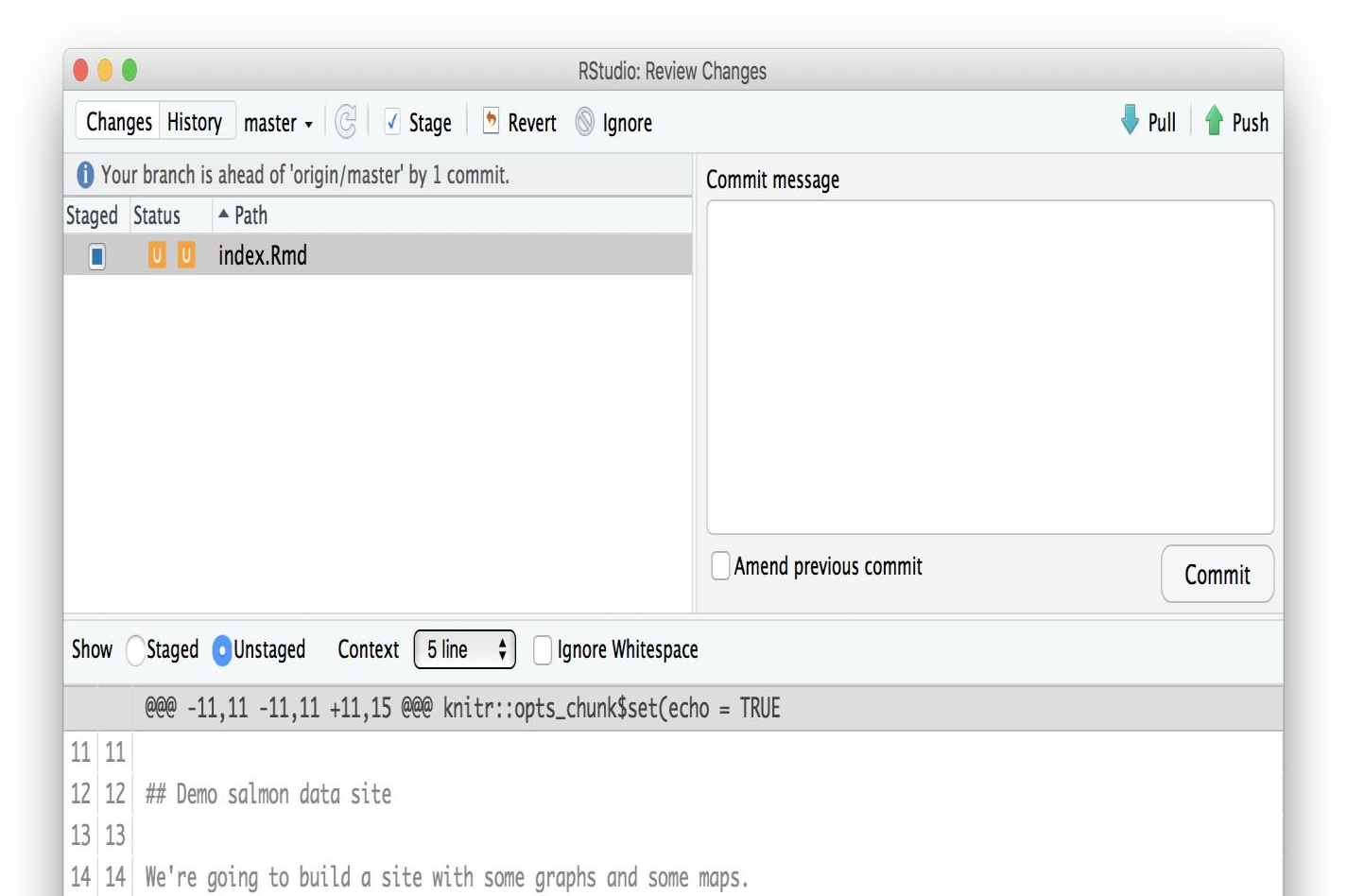
The simplest way to resolve a conflict, given that you know whose version of the file you want to keep, is to use the commandline git program to tell git to use either your changes (the person doing the merge), or their changes (the other collaborator).

* keep your collaborators file: git checkout --theirs conflicted\_file.Rmd
* keep your own file: git checkout --ours conflicted\_file.Rmd

Once you have run that command, then run add, commit, and push the changes as normal.

##### **Pull and edit the file**

But that requires the commandline. If you want to resolve from RStudio, or if you want to pick and choose some of your changes and some of your collaborator’s, then instead you can manually edit and fix the file. When you pulled the file with a conflict, git notices that there is a conflict and modifies the file to show both your own changes and your collaborator’s changes in the file. It also shows the file in the Git tab with an orange U icon, which indicates that the file is Unmerged, and therefore awaiting you help to resolve the conflict. It delimits these blocks with a series of less than and greater than signs, so they are easy to find:



To resolve the conficts, simply find all of these blocks, and edit them so that the file looks how you want (either pick your lines, your collaborators lines, some combination, or something altogether new), and save. Be sure you removed the delimiter lines that started with <<<<<<<, =======, and >>>>>>>.

Once you have made those changes, you simply add, commit, and push the files to resolve the conflict.

##### **Producing and resolving merge conflicts**

To illustrate this process, we’re going to carefully create a merge conflict step by step, show how to resolve it, and show how to see the results of the successful merge after it is complete. First, we will walk through the exercise to demonstrate the issues.

###### **Owner and collaborator ensure all changes are updated**

First, start the exercise by ensuring that both the Owner and Collaborator have all of the changes synced to their local copies of the Owner’s repositoriy in RStudio. This includes doing a git pull to ensure that you have all changes local, and make sure that the Git tab in RStudio doesn’t show any changes needing to be committed.

###### **Owner makes a change and commits**

###### From that clean slate, the Owner first modifies and commits a small change inlcuding their name on a specific line of the README.md file (we will change line 4). Work to only change that one line, and add your username to the line in some form and commit the changes (but DO NOT push). We are now in the situation where the owner has unpushed changes that the collaborator can not yet see.

###### **Collaborator makes a change and commits on the same line**

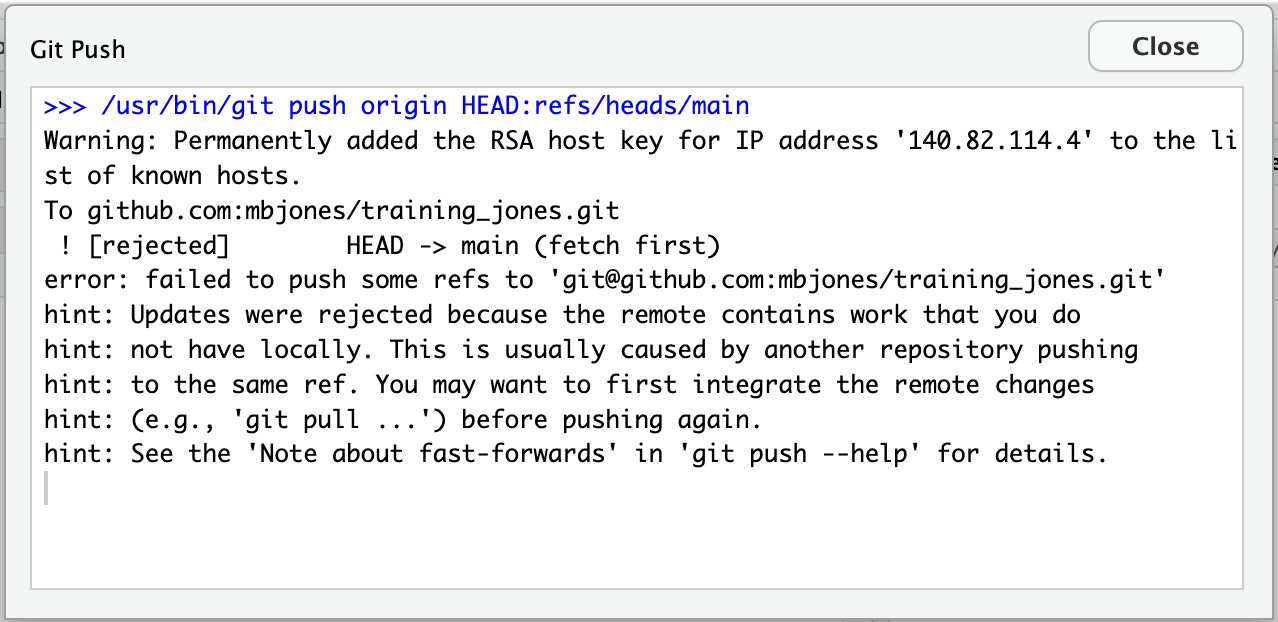
###### Now the collaborator also makes changes to the same (line 4) of the README.md file in their RStudio copy of the project, adding their name to the line. They then commit. At this point, both the owner and collaborator have committed changes based on their shared version of the README.md file, but neither has tried to share their changes via GitHub.

###### **Collaborator pushes the file to GitHub**

###### Sharing starts when the Collaborator pushes their changes to the GitHub repo, which updates GitHub to their version of the file. The owner is now one revision behind, but doesn’t yet know it.

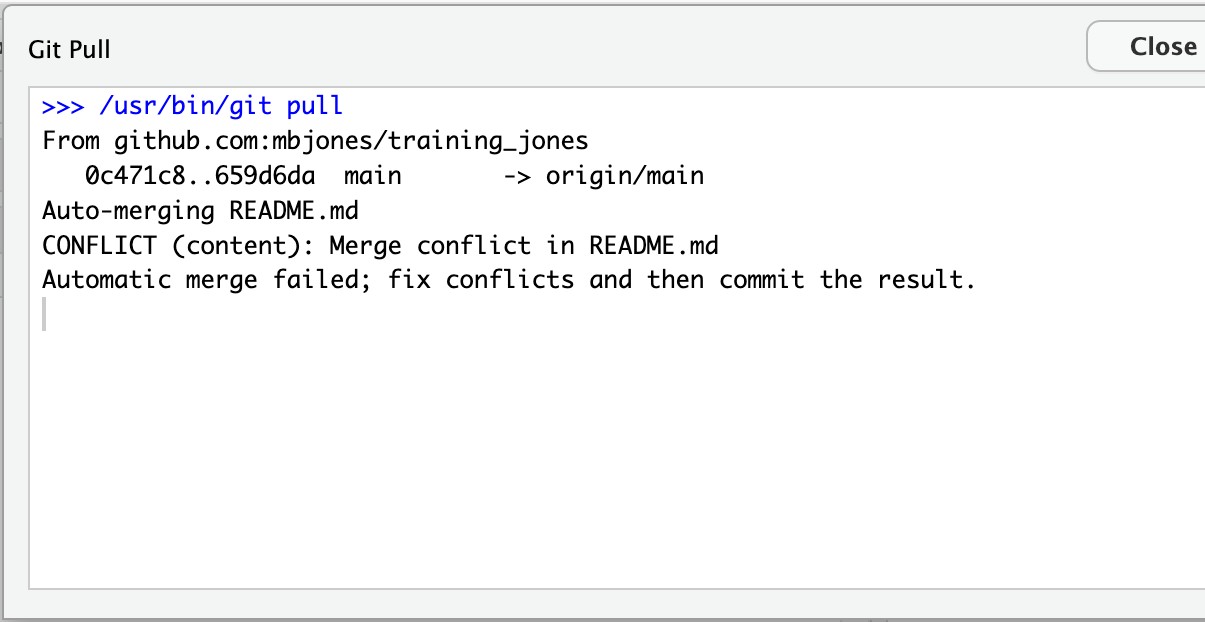
###### **Owner pushes their changes and gets an error**

###### At this point, the owner tries to push their change to the repository, which triggers an error from GitHub. While the error message is long, it basically tells you everything needed ( that the owner’s repository doesn’t reflect the changes on GitHub, and that they need to pull before they can push).

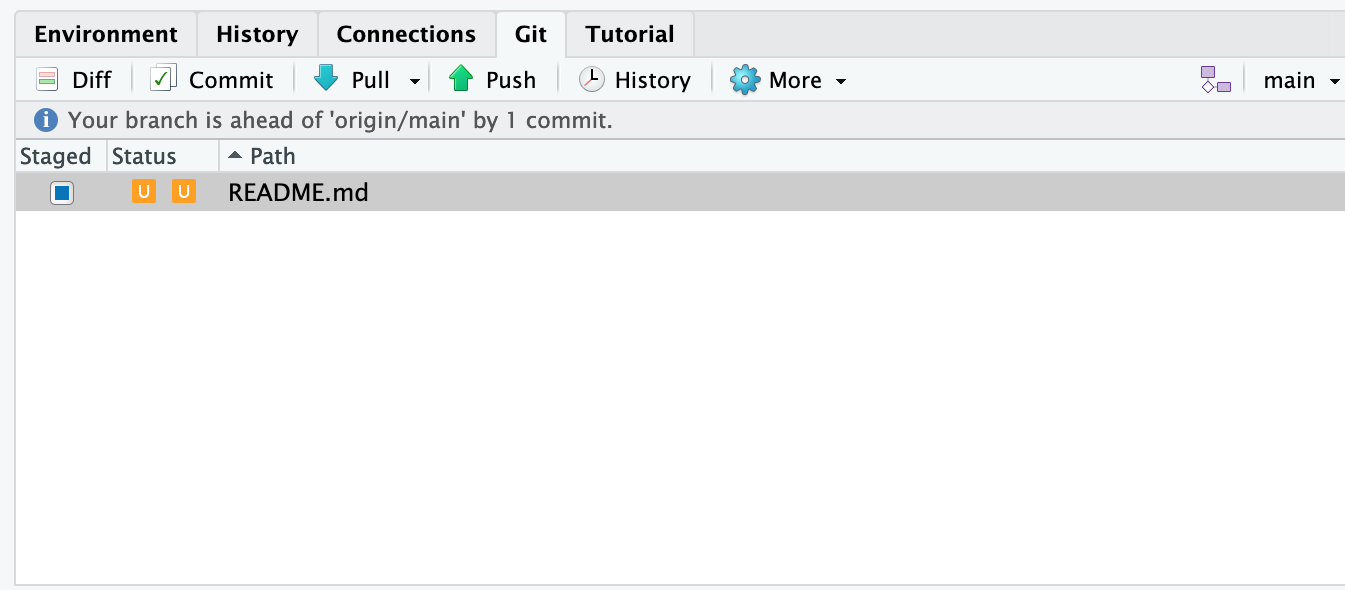


###### Owner pulls from GitHub to get Collaborator changes

Doing what the message says, the Owner pulls the changes from GitHub, and gets another, different error message. In this case, it indicates that there is a merge conflict because of the conflicting lines.

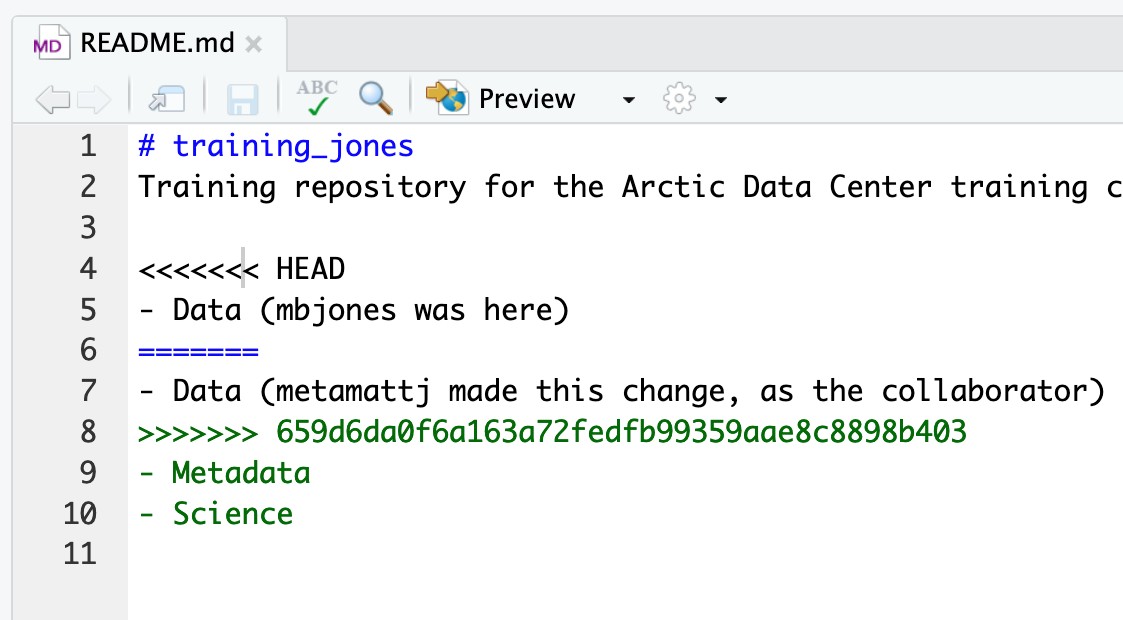


In the Git pane of RStudio, the file is also flagged with an orange ‘U’, which stands for an unresolved merge conflict.

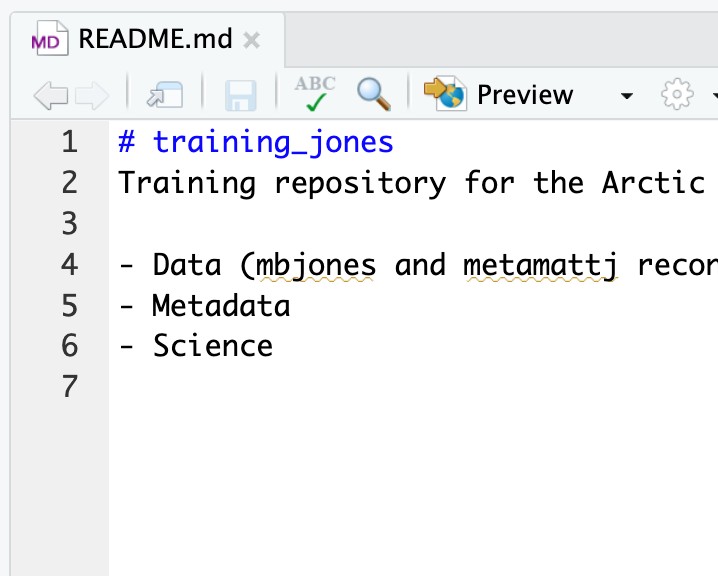


###### **Owner edits the file to resolve the conflict**

To resolve the conflict, the Owner now needs to edit the file. Again, as indicated above, git has flagged the locations in the file where a conflict occcurred with <<<<<<<, =======, and >>>>>>>. The Owner should edit the file, merging whatever changes are appropriate until the conflicting lines read how they should, and eliminate all of the marker lines with with <<<<<<<, =======, and >>>>>>>.



Of course, for scripts and programs, resolving the changes means more than just merging the text – whoever is doing the merging should make sure that the code runs properly and none of the logic of the program has been broken.

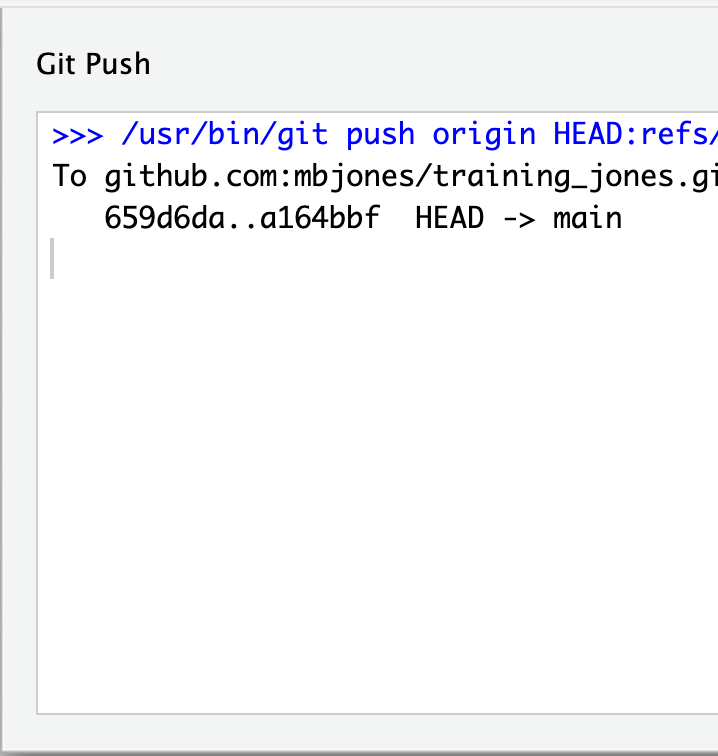


###### **Owner commits the resolved changes**

From this point forward, things proceed as normal. The owner first ‘Adds’ the file changes to be made, which changes the orange U to a blue M for modified, and then commits the changes locally. The owner now has a resolved version of the file on their system.

###### Owner pushes the resolved changes to GitHub

Have the Owner push the changes, and it should replicate the changes to GitHub without error.



###### **Collaborator pulls the resolved changes from GitHub**

Finally, the Collaborator can pull from GitHub to get the changes the owner made.

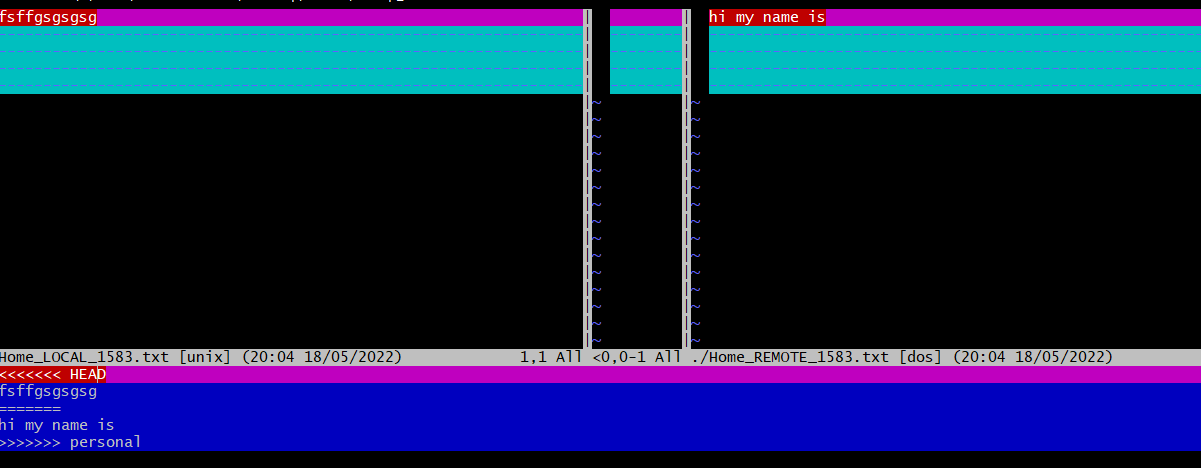
###### Both can view commit history

When either the Collaborator or the Owner view the history, the conflict, associated branch, and the merged changes are clearly visible in the history.

Merge Conflict Challenge

Now it’s your turn. In pairs, intentionally create a merge conflict, and then go through the steps needed to resolve the issues and continue developing with the merged files. See the sections above for help with each of these steps:

* Step 0: Owner and collaborator ensure all changes are updated
* Step 1: Owner makes a change and commits
* Step 2: Collaborator makes a change and commits on the same line
* Step 3: Collaborator pushes the file to GitHub
* Step 4: Owner pushes their changes and gets an error
* Step 5: Owner pulls from GitHub to get Collaborator changes
* Step 6: Owner edits the file to resolve the conflict
* Step 7: Owner commits the resolved changes
* Step 8: Owner pushes the resolved changes to GitHub
* Step 9: Collaborator pulls the resolved changes from GitHub
* Step 10: Both can view commit history

****

#### **Workflows to avoid merge conflicts**

Some basic rules of thumb can avoid the vast majority of merge conflicts, saving a lot of time and frustration. These are words our teams live by:

* Communicate often
* Tell each other what you are working on
* Pull immediately before you commit or push
* Commit often in small chunks.

**A good workflow is encapsulated as follows**:

Pull -> Edit -> Add -> Pull -> Commit -> Push

Always start your working sessions with a pull to get any outstanding changes, then start doing your editing and work. Stage your changes, but before you commit, Pull again to see if any new changes have arrived. If so, they should merge in easily if you are working in different parts of the program. You can then Commit and immediately Push your changes safely. Good luck, and try to not get frustrated. Once you figure out how to handle merge conflicts, they can be avoided or dispatched when they occur, but it does take a bit of practice.



**Experiment No. 09**

**Aim:** Reset and Revert

One of the lesser understood (and appreciated) aspects of working with Git is how easy it is to get back to where you were before—that is, how easy it is to undo even major changes in a repository. In this article, we'll take a quick look at how to reset, revert, and completely return to previous states, all with the simplicity and elegance of individual Git commands.

# How to reset a Git commit

Let's start with the Git command reset. Practically, you can think of it as a "rollback"—it points your local environment back to a previous commit. By "local environment," we mean your local repository, staging area, and working directory.

Take a look at Figure 1. Here we have a representation of a series of commits in Git. A branch in Git is simply a named, movable pointer to a specific commit. In this case, our branch master is a pointer to the latest commit in the chain.

If we look at what's in our *master* branch now, we can see the chain of commits made so far.

$ git log --oneline

b764644 File with three lines 7c709f0 File with two lines 9ef9173 File with one line

## Programming and development

* Red
* [Programming cheat sheets](https://opensource.com/downloads/cheat-sheets?intcmp=7016000000127cYAAQ)
* [Try for free: Red Hat Learning Subscription](https://www.redhat.com/en/services/training/learning-subscription?intcmp=7016000000127cYAAQ)
* [eBook: An introduction to programming with Bash](https://opensource.com/downloads/bash-programming-guide?intcmp=7016000000127cYAAQ)
* [Bash Shell Scripting Cheat Sheet](https://developers.redhat.com/cheat-sheets/bash-shell-cheat-sheet?intcmp=7016000000127cYAAQ)
* [eBook: Modernizing Enterprise Java](https://developers.redhat.com/e-books/modernizing-enterprise-java?intcmp=7016000000127cYAAQ)

What happens if we want to roll back to a previous commit. Simple—we can just move the branch pointer. Git supplies the reset command to do this for us. For example, if we want to reset *master* to point to the commit two back from the current commit, we could use either of the following methods:

$ git reset 9ef9173 (using an absolute commit SHA1 value 9ef9173) or

$ git reset current~2 (using a relative value -2 before the "current" tag)

Figure 2 shows the results of this operation. After this, if we execute a git log command on the current branch (*master*), we'll see just the one commit

$ git log --oneline 9ef9173 File with one line

The git reset command also includes options to update the other parts of your local environment with the contents of the commit where you end up. These options include: hard to reset the commit being pointed to in the repository, populate the working directory with the contents of the commit, and reset the staging area; soft to only reset the pointer in the repository; and mixed (the default) to reset the pointer and the staging area.

Using these options can be useful in targeted circumstances such as git reset --hard <commit sha1 | reference>. This overwrites any local changes you haven't committed. In effect, it resets (clears out) the staging area and overwrites content in the working directory with the content from the commit you reset to. Before you use the hard option, be sure that's what you really want to do, since the command overwrites any uncommitted changes.

# **How to revert a Git commit**

The net effect of the git revert command is similar to reset, but its approach is different. Where the reset command moves the branch pointer back in the chain (typically) to "undo" changes, the revert command adds a new commit at the end of the chain to "cancel" changes. The effect is most easily seen by looking at Figure 1 again. If we add a line to a file in each commit in the chain, one way to get back to the version with only two lines is to reset to that commit, i.e., git reset HEAD~1.

Another way to end up with the two-line version is to add a new commit that has the third line removed—effectively canceling out that change. This can be done with a git revert command, such as:

$ git revert HEAD

Because this adds a new commit, Git will prompt for the commit message:

Revert "File with three lines"

This reverts commit b764644bad524b804577684bf74e7bca3117f554.

# Please enter the commit message for your changes. Lines starting

# with '#' will be ignored, and an empty message aborts the commit.

# On branch master

# Changes to be committed:

# modified: file1.txt #

Figure 3 (below) shows the result after the revert operation is completed.

If we do a git log now, we'll see a new commit that reflects the contents before the previous commit.

$ git log --oneline

11b7712 Revert "File with three lines" b764644 File with three lines

7c709f0 File with two lines 9ef9173 File with one line

Here are the current contents of the file in the working directory:

$ cat <filename> Line 1

Line 2

## Revert or reset?

Why would you choose to do a revert over a reset operation? If you have already pushed your chain of commits to the remote repository (where others may have pulled your code and started working with it), a revert is a nicer way to cancel out changes for them. This is because the Git workflow works well for picking up additional commits at the end of a branch, but it can be challenging if a set of commits is no longer seen in the chain when someone resets the branch pointer back.

This brings us to one of the fundamental rules when working with Git in this manner: Making these kinds of changes in your *local repository* to code you haven't pushed yet is fine. But avoid making changes that rewrite history if the commits have already been pushed to the remote repository and others may be working with them.

In short, if you rollback, undo, or rewrite the history of a commit chain that others are working with, your colleagues may have a lot more work when they try to merge in changes based on the original chain they pulled. If you must make changes against code that has already been pushed and is being used by others, consider communicating before you make the changes and give people the chance to merge their changes first. Then they can pull a fresh copy after the infringing operation without needing to merge.

You may have noticed that the original chain of commits was still there after we did the reset. We moved the pointer and reset the code back to a previous commit, but it did not delete any commits. This means that, as long as we know the original commit we were pointing to, we can "restore" back to the previous point by simply resetting back to the original head of the branch:

git reset <sha1 of commit>

A similar thing happens in most other operations we do in Git when commits are replaced. New commits are created, and the appropriate pointer is moved to the new chain. But the old chain of commits still exists.

# **Rebase**

Now let's look at a branch rebase. Consider that we have two branches—

*master* and *feature*—with the chain of commits shown in Figure 4

below. *Master* has the chain C4->C2->C1->C0 and *feature* has the chain C5->C3-

>C2->C1->C0.

If we look at the log of commits in the branches, they might look like the following. (The C designators for the commit messages are used to make this easier to understand.)

$ git log --oneline master 6a92e7a C4

259bf36 C2

f33ae68 C1 5043e79 C0

$ git log --oneline feature 79768b8 C5

000f9ae C3 259bf36 C2

f33ae68 C1 5043e79 C0

I tell people to think of a rebase as a "merge with history" in Git. Essentially what Git does is take each different commit in one branch and attempt to "replay" the differences onto the other branch.

So, we can rebase a feature onto master to pick up C4 (e.g., insert it into feature's chain). Using the basic Git commands, it might look like this:

$ git checkout feature

$ git rebase master

First, rewinding head to replay your work on top of it... Applying: C3

Applying: C5

Afterward, our chain of commits would look like Figure 5.

Again, looking at the log of commits, we can see the changes.

$ git log --oneline master 6a92e7a C4

259bf36 C2

f33ae68 C1 5043e79 C0

$ git log --oneline feature c4533a5 C5

64f2047 C3

6a92e7a C4 259bf36 C2

f33ae68 C1 5043e79 C0

Notice that we have C3' and C5'—new commits created as a result of making the changes from the originals "on top of" the existing chain in master. But also notice that the "original" C3 and C5 are still there—they just don't have a branch pointing to them anymore.

If we did this rebase, then decided we didn't like the results and wanted to undo it, it would be as simple as:

$ git reset 79768b8

With this simple change, our branch would now point back to the same set of commits as before the rebase operation—effectively undoing it (Figure 6).

What happens if you can't recall what commit a branch pointed to before an operation? Fortunately, Git again helps us out. For most operations that modify pointers in this way, Git remembers the original commit for you. In fact, it stores it in a special reference named ORIG\_HEAD within the .git repository directory. That path is a file containing the most recent reference before it was modified. If we cat the file, we can see its contents.

$ cat .git/ORIG\_HEAD 79768b891f47ce06f13456a7e222536ee47ad2fe

We could use the reset command, as before, to point back to the original chain. Then the log would show this:

$ git log --oneline feature 79768b8 C5

000f9ae C3 259bf36 C2

f33ae68 C1 5043e79 C0

Another place to get this information is in the reflog. The reflog is a play-by- play listing of switches or changes to references in your local repository. To see it, you can use the git reflog command:

|  |  |  |  |
| --- | --- | --- | --- |
| $ git reflog  79768b8 HEAD@{0}: | | reset: moving to | 79768b |
| c4533a5 HEAD@{1}: rebase finished: refs/heads/feature | | | returning to |
| c4533a5 64f2047  6a92e7a | HEAD@{2}:  HEAD@{3}:  HEAD@{4}: | rebase: C5 rebase: C3  rebase: checkout | master |
| 79768b8  feature  79768b8 | HEAD@{5}:  HEAD@{6}: | checkout: moving  commit: C5 | from feature to |
| 000f9ae 6a92e7a  259bf36 | HEAD@{7}:  HEAD@{8}:  HEAD@{9}: | checkout: moving commit: C4  checkout: moving | from master to feature  from feature to master |
| 000f9ae HEAD@{10}: commit: C3  259bf36 HEAD@{11}: checkout: moving from master to feature  259bf36 HEAD@{12}: commit: C2 f33ae68 HEAD@{13}: commit: C1  5043e79 HEAD@{14}: commit (initial): C0 | | | |

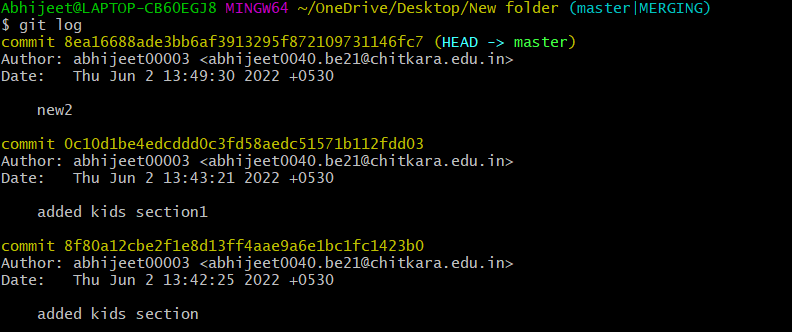
You can then reset to any of the items in that list using the special relative naming format you see in the log:

$ git reset HEAD@{1}

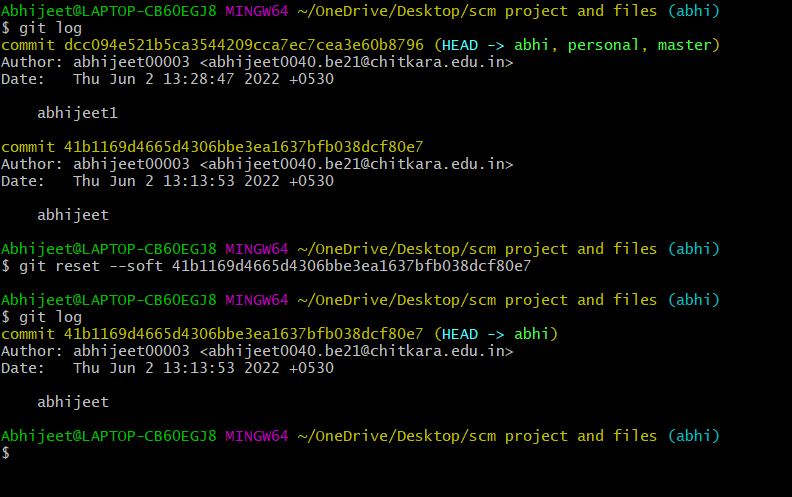
Once you understand that Git keeps the original chain of commits around when operations "modify" the chain, making changes in Git becomes much less scary. This is one of Git's core strengths: being able to quickly and easily try things out and undo them if they don't work.

**Snapshots**

**Git log for file before reset**

****

git reset --soft

****

git reset mixed(default)

## 

git reset --hard

