

**Subject Name: Source Code Management**

**Subject code cse 181**

**Cluser beta**

**CHITKARA**  
UNIVERSITY



**Submitted By:**

DEVWANI

THAKUR

2110990430

G08

**Submitted To:**

Dr. Monit kapoor

## INDEX

<b>S. No</b>	<b>Program Title</b>	<b>Page No.</b>
1	Add Collaborators on GitHub Repository	2-11
2	Fork and Commit	12-15
3	Merge and Resolve conflicts created due to own activity and Collaborators activity	16-27
4	Reset and Revert	28-34

## AIM: Add Collaborators on GitHub Repository

In GitHub, We can invite other GitHub users to become collaborators to our private repositories(which expires after 7 days if not accepted, restoring any unclaimed licenses). Being a collaborator, of a personal repository you can pull (read) the contents of the repository and push (write) changes to the repository. You can add unlimited collaborators on public and private repositories(with some per day limit restrictions). But, in a private repository, the owner of the repo can only grant write-access to the collaborators, and they can't have the read-only access.

GitHub also restricts the number of collaborators we can invite within a period of 24 hours. If we exceed the limit, then either we have to wait for 24-hours or we can also create an organization to collaborate with more people.

### Actions that can be Performed by Collaborators

Collaborators can perform a number of actions into someone else's personal repositories, they have gained access to Some of them are,

- Create, merge, and close pull requests in the repository
- Publish, view, install the packages
- Fork the repositories
- Make the changes on the repositories as suggested by the Pull requests.
- Mark issues or pull requests as duplicate
- Create, edit, and delete any comments on commits, pull requests, and issues in the repository
- Removing themselves as collaborators on the repositories.
- Manage releases in the repositories.

Now, let's see how can we invite collaborators to our repositories.

### Inviting Collaborators to your personal repositories

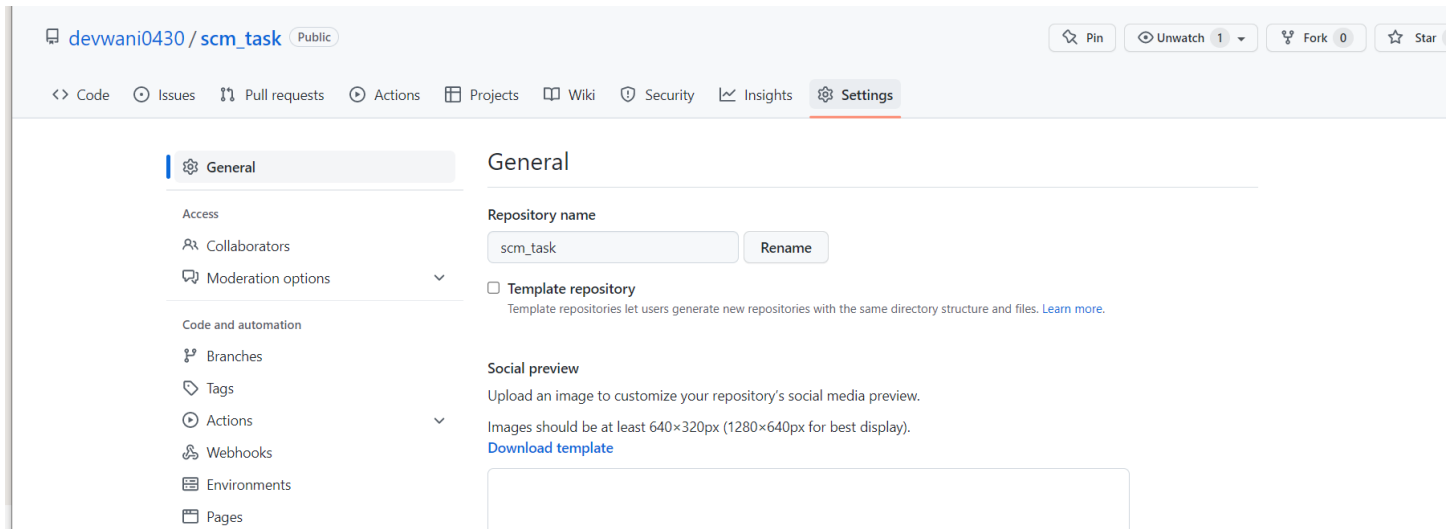
Follow the steps below to invite collaborators to your own repository (public or private).

**Step 1:** Get the usernames of the GitHub users you will be adding as collaborators. In case, they are not on GitHub, ask them to sign in to GitHub.

**Step 2:** Go to your repository (intended to add collaborators)

**Step 3:** Click into the Settings.

**Step 4:** A settings page will appear. Here, into the left-sidebar click into the Collaborators.



**Step 5:** Then a confirm password page may appear, enter your password for the confirmation.

**Step 6:** Next, click into Add People.

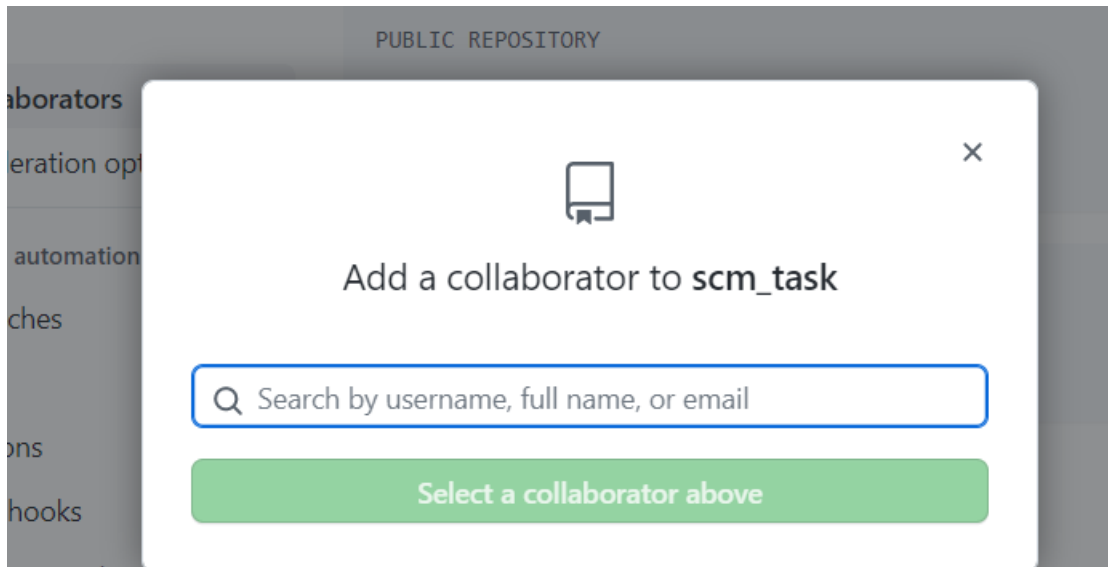
## Manage access

Add people

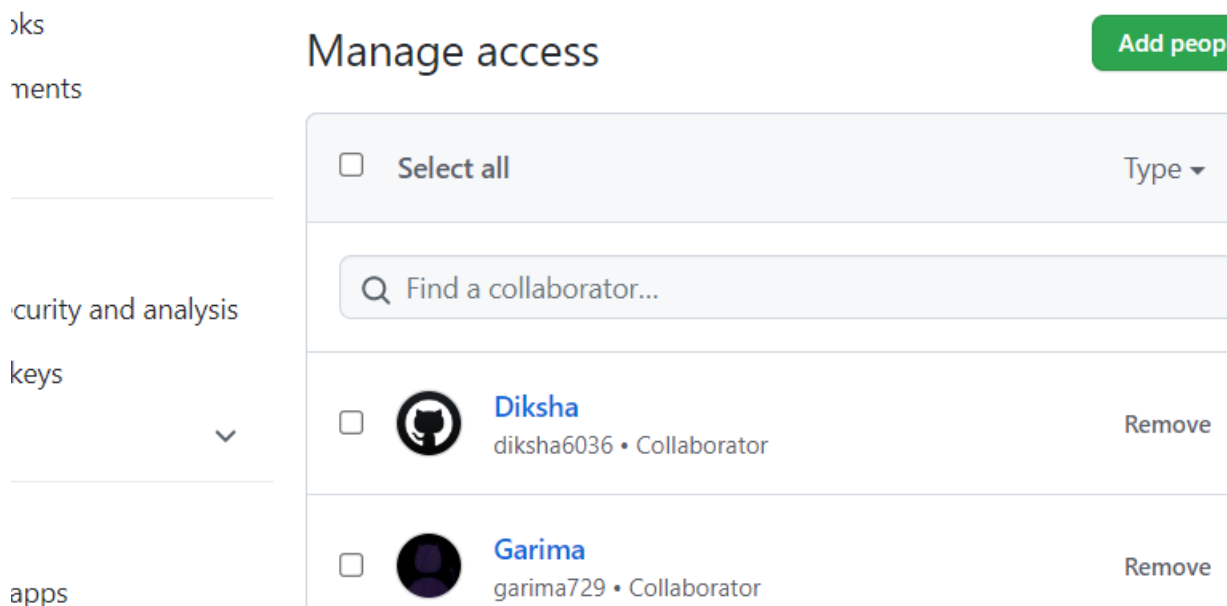
☐ Select all
 Type ▼

Find a collaborator...

**Step 7:** Then a search field will appear, where you can enter the username of the ones you want to add as collaborator.



**Step 8:** After selecting the people, add them as collaborator.



**Step 9:** After sending the request for collaboration to that person whom we wanted as our collaborator for our project will get an email from the Team leader (by GitHub). He/she has to open its Email to view the invitation.





@garima729 has invited you to join the  
@SCM-Team-7 organization



@garima729 has invited you to join the SCM-  
Team-7 organization

Hi devwani0430!

@garima729 has invited you to join the @SCM-Team-7 organization on  
GitHub. Head over to <https://github.com/SCM-Team-7> to check out  
@SCM-Team-7's profile.

This invitation will expire in 7 days.

[Join @SCM-Team-7](#)

**Note:** If you get a 404 page, make sure you're signed in as devwani0430.  
You can also accept the invitation by visiting the organization page directly.

**Step 10:** After Clicking the View invitation, He/she will be redirected to the GitHub Page for accepting the invitation sent by the team leader.

**Step 11:** We are done adding a single collaborator. Now, they will get a mail regarding invitation to your repository. Once they accept, they will have collaborator access to your repository. Till then it will be in pending invitation state. You can also add more collaborator and delete the existing one as depicted below.


## Manage access

Add people

☐ Select all


Type ▾

☐

**Chaarvi Jusroy**  
Cjusroy • Collaborator


Remove

☐


**Diksha**  
diksha6036 • Collaborator

Remove

☐

**Garima**  
garima729 • Collaborator

Remove



**Get team access controls and discussions for your contributors in an organization.**  
(NEW) Private repos and unlimited members are free.

Create an organization



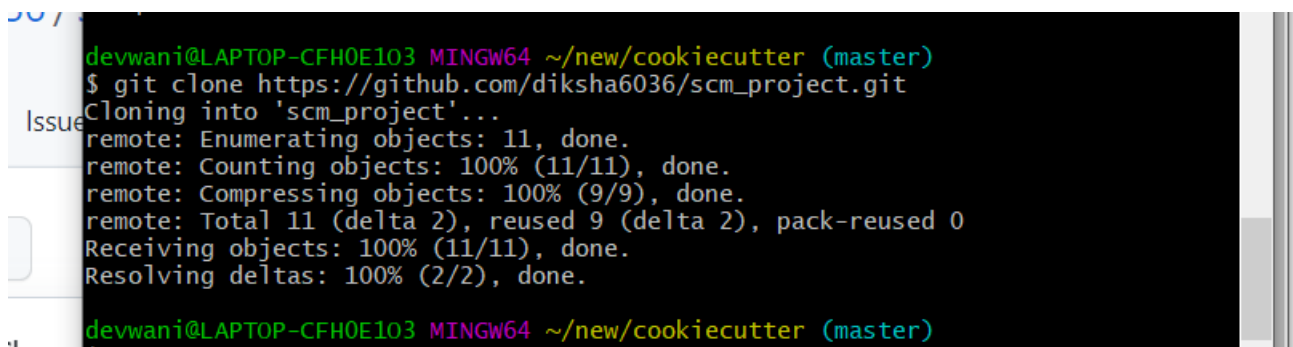
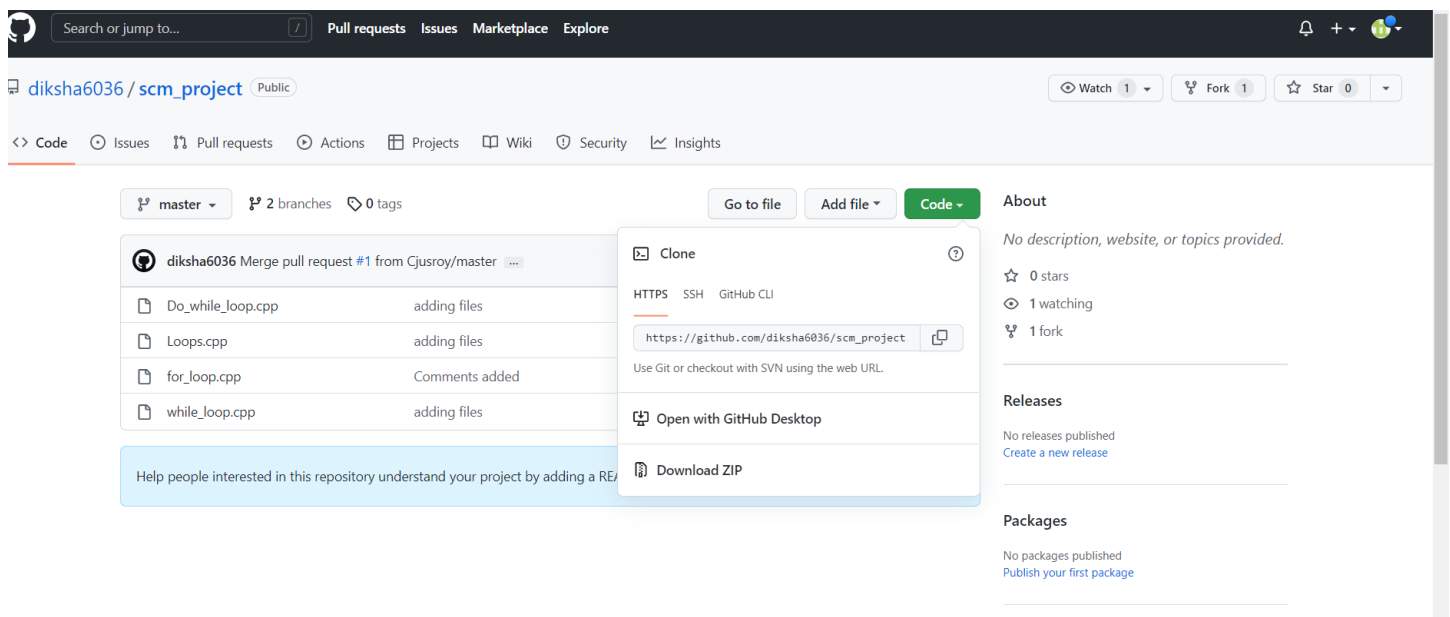
## Removing collaborator permissions from a person contributing to a repository

Similar to the above steps, go to Your **Repository** -> **Settings** -> **Manage Access** -> **Remove** (on the right side of collaborator username)

## Cloning a repository

Next, the Collaborator needs to download a copy of the Owner's repository to her machine. This is called "cloning a repository".

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



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. Then push the change to the *Owner's repository* on GitHub

git

## **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 intrinsic 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 remoterepository at all - it just removes the link to it from thelocal 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:

```
Everything up-to-date
devwani@LAPTOP-CFH0E103 MINGW64 /d/scm project/Team07 (main)
$ git pull origin master
From https://github.com/diksha6036/Team07
* branch          master      -> FETCH_HEAD
```

## 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 collaborativeworkflow 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?

## Solution

### Comment Changes in GitHub

The screenshot shows the GitHub Settings page for the user 'devwani0430'. The left sidebar contains navigation links: Public profile, Account, Appearance, Accessibility, Notifications, Access, Billing and plans, Emails, Password and authentication, SSH and GPG keys (highlighted), Organizations, and Moderation. Below these are links for Code, planning, and automation, Repositories, and Packages. The main content area has three sections: 'SSH keys' with a 'New SSH key' button and a list of keys (one key is shown for 'scm project' with a 'Delete' button), 'GPG keys' with a 'New GPG key' button and a message stating no GPG keys are associated, and 'Vigilant mode'. The browser address bar shows 'https://github.com/settings/gpg/new'.

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 allow 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?

### **Generate SSH Key**

```
devwani@LAPTOP-CFH0E103 MINGW64 ~ (master)
$ git remote -v
origin https://github.com/devwani0430/scm_task.git (fetch)
origin https://github.com/devwani0430/scm_task.git (push)

devwani@LAPTOP-CFH0E103 MINGW64 ~ (master)
$ ssh-keygen -t rsa -b 4096 -C "devwani0430.be21@chitkara.edu.in"
Generating public/private rsa key pair.
Enter file in which to save the key (/c/Users/thaku/.ssh/id_rsa):
Created directory '/c/Users/thaku/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /c/Users/thaku/.ssh/id_rsa
Your public key has been saved in /c/Users/thaku/.ssh/id_rsa.pub
The key's fingerprint is:
SHA256:uwbBRaXkMK1Hzyc6BE9i+peGRN5AdnEc33n3qsQcrWM devwani0430.be21@chitkara.edu.in
The key's randomart image is:
+---[RSA 4096]---+
| .o oo=+o.. |
| ."....o"=o . |
| = B o =++o . . |
| . o = + . o. o . |
| o o o oS . . . |
| o = . . o o . |
| o .. E . |
| .. o o |
| .. . |
+-----[SHA256]-----+

devwani@LAPTOP-CFH0E103 MINGW64 ~ (master)
$ eval "$(ssh-agent -s)"
Agent pid 732

devwani@LAPTOP-CFH0E103 MINGW64 ~ (master)
$
```

# AIM: FORK AND COMMIT

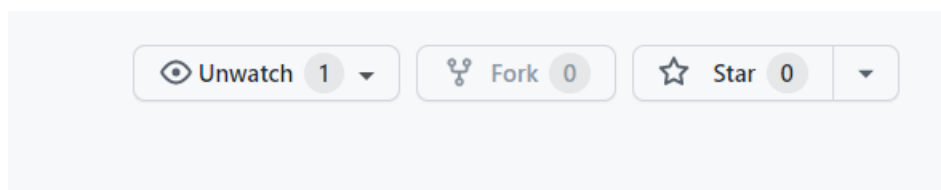
## What is Forking a Repository mean and why it is used?

**Forking a repository** means creating a copy of the repo. When you fork a repo, you create your own copy of the repository on your GitHub account. This is done for the following reasons:

1. You have your own copy of the project on which you may test your own changes without changing the original project.
2. This helps the maintainer of the project to better check the changes you made to the project and has the power to either accept, reject or suggest something.
3. When you clone an open source project, which isn't yours, you don't have the right to push code directly into the project.

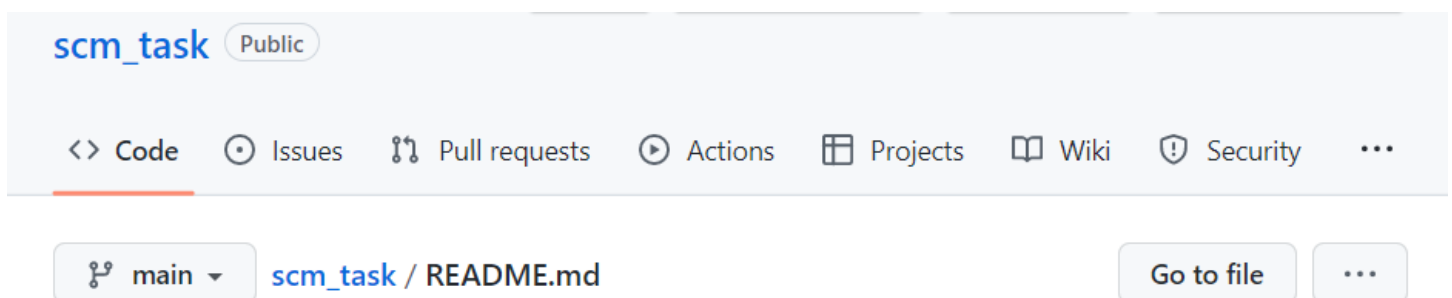
For these reasons, you are always suggested to FORK. Let's have a screenshot walkthrough of the whole process. When getting started with a contribution to Open source Project, you have been advised to first FORK the repository(repo). But what is a fork?

You must have seen this icon on every repository in the top right corner. Now, this button is used to Fork the repo. But again, what is a fork or forking a repository in GitHub as shown in the below media as follows:



## Procedure:

**Step 1:** Go to task1.2 official repository.

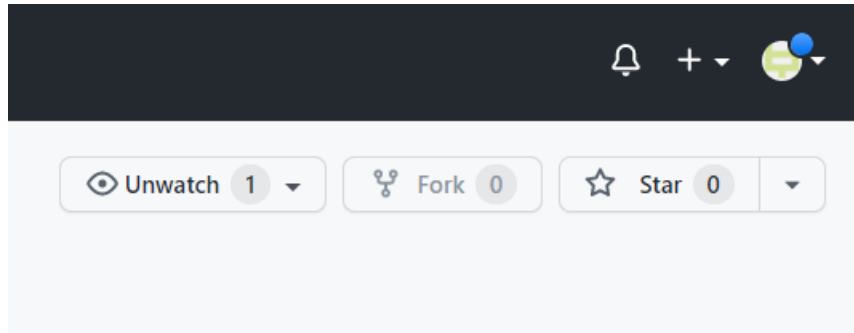


You can see **devwaqni0430?scm\_task** This means **devwanithakur** is the maintainer and

**Scm\_task** is the project's name.



**Step 2:** Find the Fork button on the top right corner.



**Step 3:** Click on **Fork**.

**Step 4:** Now you have your own copy of the repository.

## 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.

1. On GitHub.com, navigate to **your fork** of the Spoon-Knife repository.
2. Above the list of files, click **Code**.
3. To clone the repository using HTTPS, click under "Clone with HTTPS". To clone the repository using an SSH key, including a certificate issued by your organization's SSH certificate authority, click **Use SSH**. To clone a repository using GitHub CLI, click **Use GitHub CLI**.
4. Open Git Bash.
5. Change the current working directory to the location where you want the cloned directory.
6. 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
```

7. Press **Enter**. Your local clone will be created.

```
devwani@LAPTOP-CFH0E103 MINGW64 ~ (master)
$ git clone https://github.com/diksha6036/Team07.git
Cloning into 'Team07'...
remote: Enumerating objects: 37, done.
remote: Counting objects: 100% (37/37), done.
remote: Compressing objects: 100% (33/33), done.
remote: Total 37 (delta 7), reused 8 (delta 0), pack-reused 0
Receiving objects: 100% (37/37), 9.18 KiB | 9.18 MiB/s, done.
Resolving deltas: 100% (7/7), done.
```

```
devwani@LAPTOP-CFH0E103 MINGW64 ~ (master)
$
```

```
devwani@LAPTOP-CFH0E103 MINGW64 ~ (master)
$ git remote -v
origin https://github.com/devwani0430/scm_task.git (fetch)
origin https://github.com/devwani0430/scm_task.git (push)
devwani@LAPTOP-CFH0E103 MINGW64 ~ (master)
```

## What is COMMIT in GitHub?

Commit is like a snapshot of your repository. These commits are snapshots of your entire repository at specific times. You should make new commits often, based around logical units of change.

Over time, commits should tell a story of the history of your repository and how it came to be the way that it currently is.

Commits include lots of metadata in addition to the contents and message, like the author, timestamp and more.

It is similar to saving a file that's been edited, a commit records changes to one or more files in your branch. Git assigns each commit a unique ID, called a SHA or hash, that identifies:

- The Specific Changes
- When the Changes were made
- Who created the changes.

When you make a commit, you must include a commit message that briefly describes the changes, you can also add a co-author on any commits you collaborate on.

```
devwani@LAPTOP-CFH0E103 MINGW64 ~ (master)
$ touch file.txt

devwani@LAPTOP-CFH0E103 MINGW64 ~ (master)
$ git add file.txt

devwani@LAPTOP-CFH0E103 MINGW64 ~ (master)
$ git commit -m "added commit"
[master 2974631] added commit
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 file.txt

devwani@LAPTOP-CFH0E103 MINGW64 ~ (master)
$ git push origin master
Enumerating objects: 11, done.
Counting objects: 100% (11/11), done.
Delta compression using up to 8 threads
Compressing objects: 100% (7/7), done.
Writing objects: 100% (11/11), 932 bytes | 932.00 KiB/s, done.
Total 11 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), done.
remote:
remote: Create a pull request for 'master' on GitHub by visiting:
remote:   https://github.com/devwani0430/scm_task/pull/new/master
remote:
To https://github.com/devwani0430/scm_task.git
 * [new branch]      master -> master

devwani@LAPTOP-CFH0E103 MINGW64 ~ (master)
$ git pull origin master
From https://github.com/devwani0430/scm_task
 * branch            master       -> FETCH_HEAD
Already up to date.

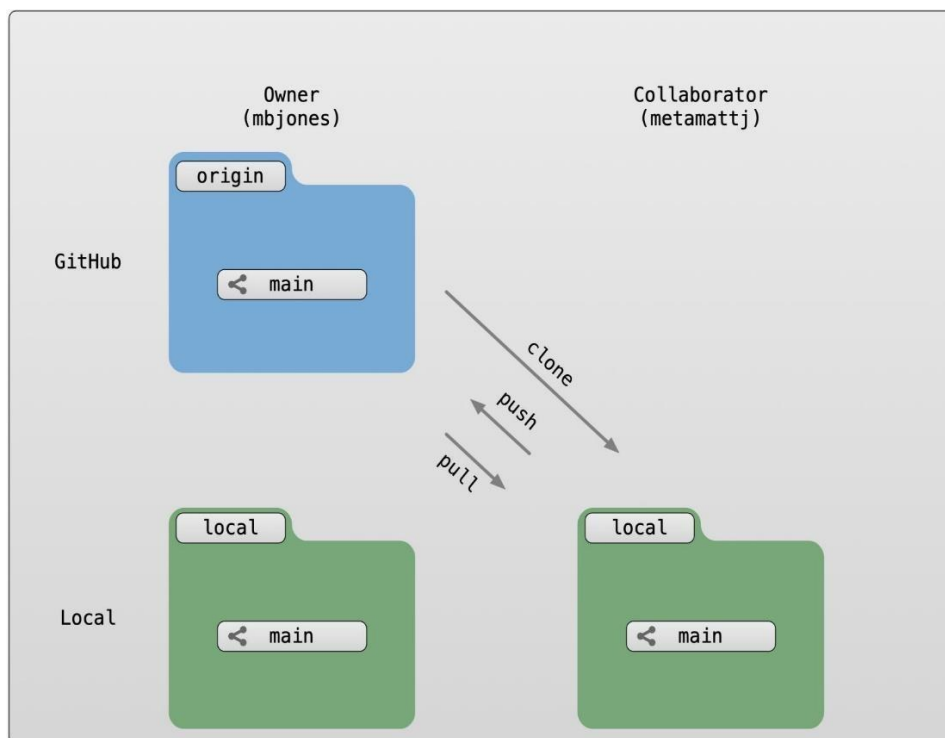
devwani@LAPTOP-CFH0E103 MINGW64 ~ (master)
```

## 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, 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.



## 5.3 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 repository. We start by giving that person access to our GitHub repository.

### Setup

We will break you into pairs, so choose one person as the owner and one as the Collaborator

Log into GitHub as the owner

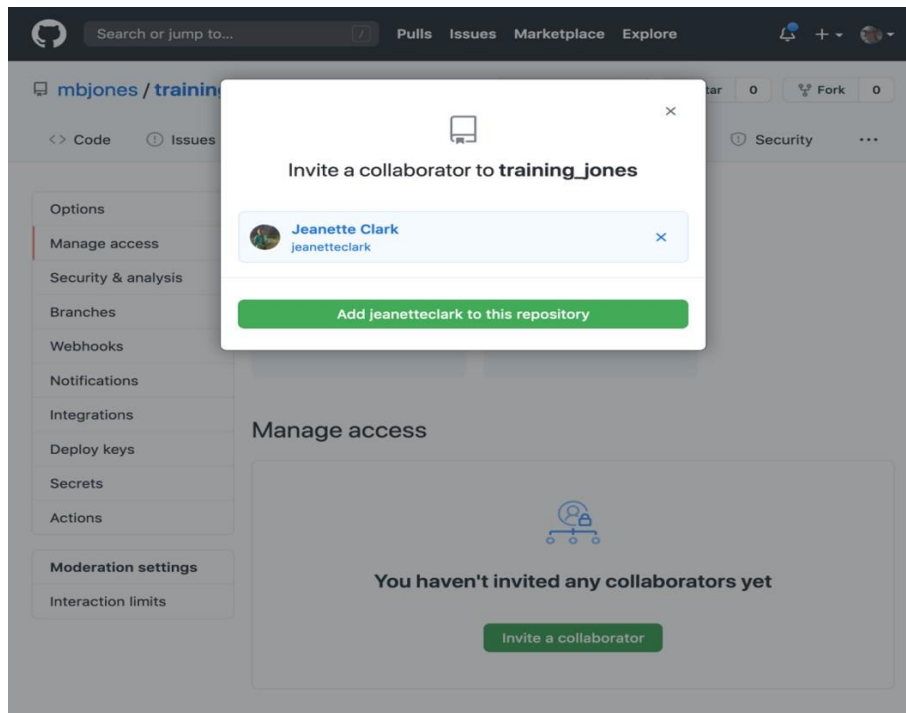
Navigate to the owner's training repository (e.g., training Jones)

Then, have the Owner visit their training repository created earlier, and visit the *Settings* page, and select the *Manage access* screen, and add the username of your Collaborator in the box.

Once the collaborator has been added, they should check their email for an invitation from GitHub, and click on the acceptance link, which will enable them to collaborate on the 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.

### 5.3.1 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).

### 5.3.2 Step 2: Collaborator Edits

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

### 5.3.3 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, it's 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 clicking 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.

### 5.3.4 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!**

### 5.3.5 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.

### 5.3.6 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

## 5.4 Merge conflicts

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.

## 5.5 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 still 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



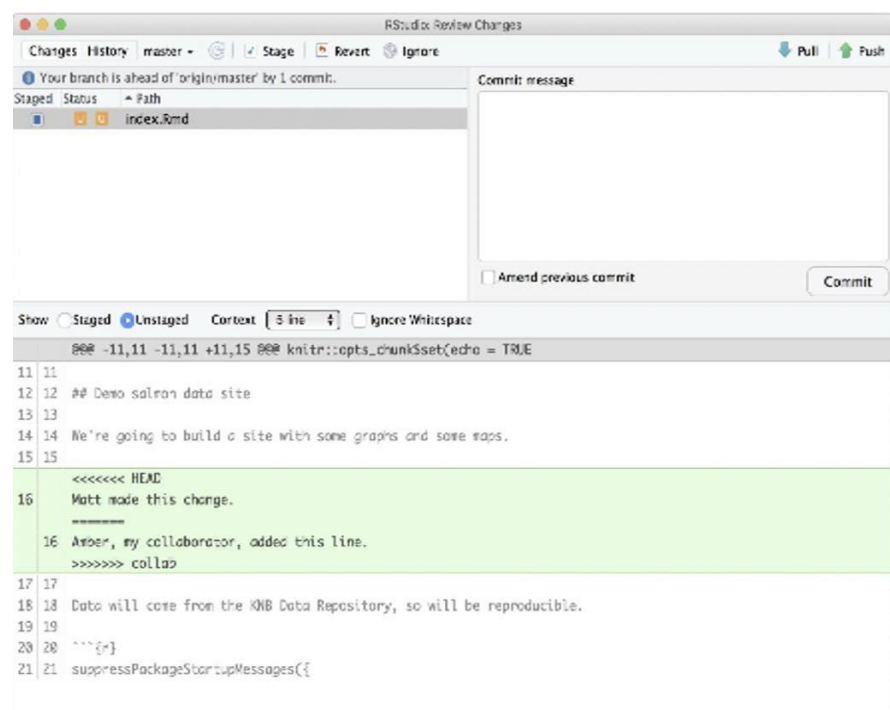
The simplest way to resolve a conflict, given that you know whose version of the file you want to keep, is to use the commandline git program to tell git to use either *your* changes (the person doing the merge), or *their* changes (the other collaborator).

- 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 command line. 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 conflicts, 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.

## 5.5.1 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.

### 5.5.1.1 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 repository 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.

### 5.5.1.2 Owner makes a change and commits

From that clean slate, the Owner first modifies and commits a small change including 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 cannot yet see.

### 5.5.1.3 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.

### 5.5.1.4 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.

### 5.5.1.5 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).

```

Git Push
Close

>>> /usr/bin/git push origin HEAD:refs/heads/main
Warning: Permanently added the RSA host key for IP address '140.82.114.4' to the list of known hosts.
To github.com:mbjones/training_jones.git
 ! [rejected]          HEAD -> main (fetch first)
error: failed to push some refs to 'git@github.com:mbjones/training_jones.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.

```

### 5.5.1.6 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.

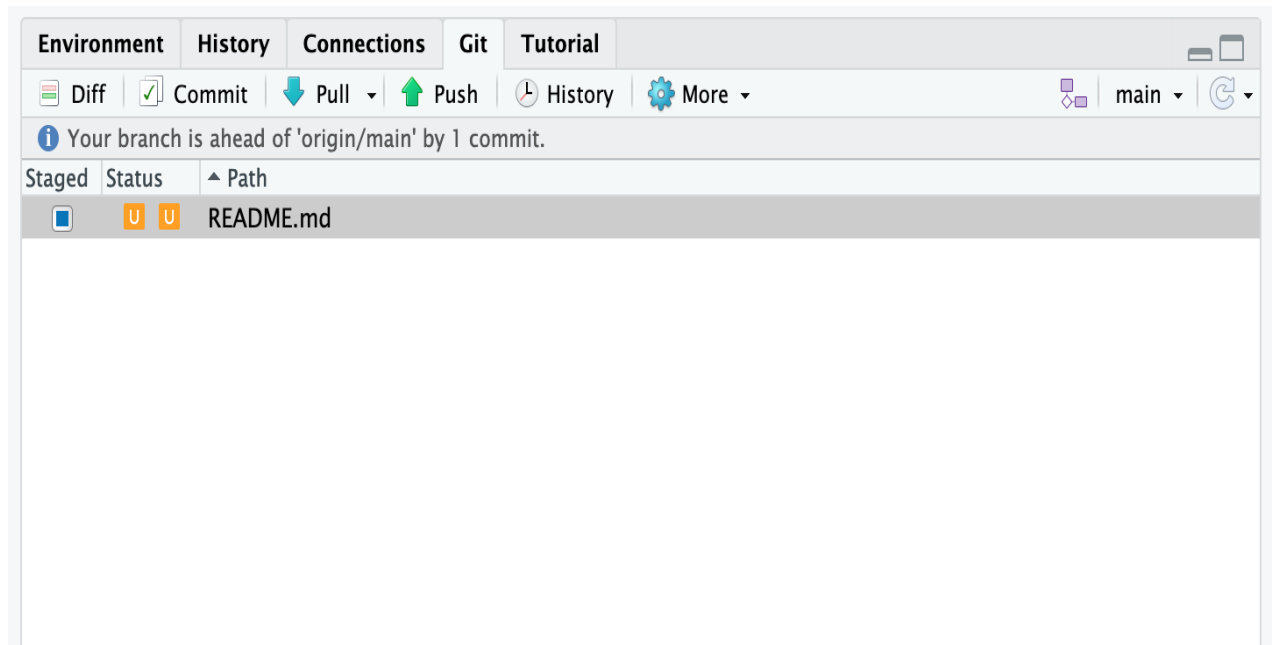
```

Git Pull
Close

>>> /usr/bin/git pull
From github.com:mbjones/training_jones
 0c471c8..659d6da  main      -> origin/main
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.

```

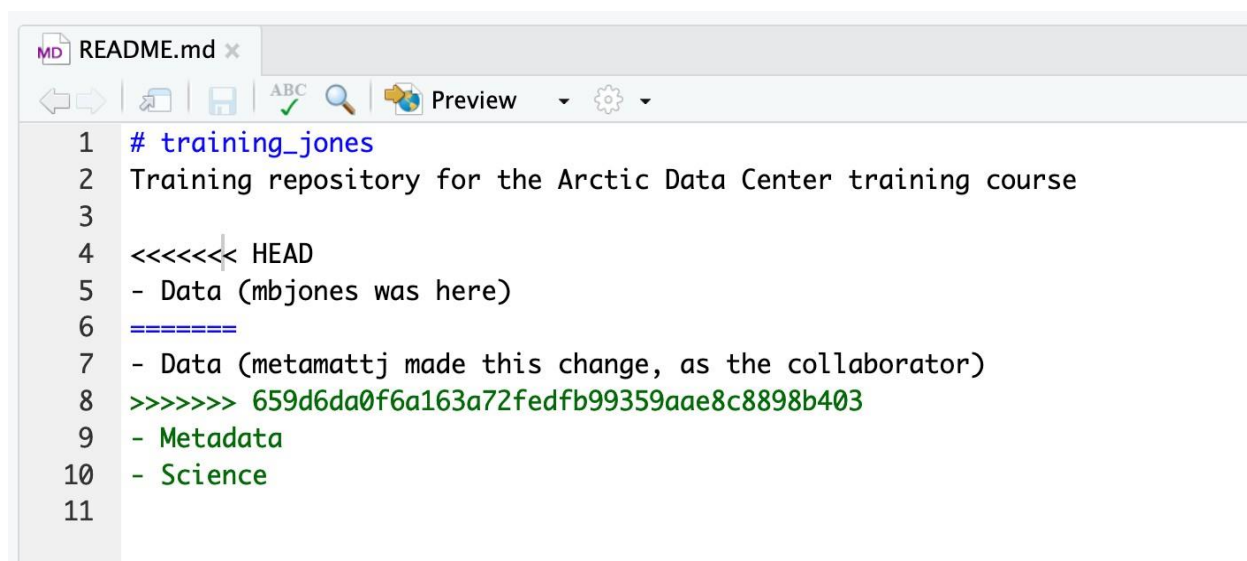
In the Git pane of RStudio, the file is also flagged with an orange 'U', which stands for an unresolved merge conflict.



### 5.5.1.7 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 occurred 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.

```

1 # training_jones
2 Training repository for the Arctic Data Center training course
3
4 - Data (mbjones and metamattj reconciled)
5 - Metadata
6 - Science
7

```

### 5.5.1.8 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.

Changes History main Stage Revert Ignore Pull Push

Your branch is ahead of 'origin/main' by 1 commit.

Staged	Status	Path
<input checked="" type="checkbox"/>	M	README.md

Commit message 43 characters

Merged changes from Owner and Collaborator.

☐ Amend previous commit Commit

Show Staged Unstaged Context 5 line Ignore Whitespace Unstage All

```

@@ -1,6 +1,6 @@
1 # training_jones
2 Training repository for the Arctic Data Center training course
3
4 - Data (mbjones was here)
4 + Data (mbjones and metamattj reconciled)
5 - Metadata
6 - Science

```

### 5.5.1.9 Owner pushes the resolved changes to GitHub

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

Git Push

Close

```

>>> /usr/bin/git push origin HEAD:refs/heads/main
To github.com:mbjones/training_jones.git
659d6da..a164bbf HEAD -> main

```

### 5.5.1.10 Collaborator pulls the resolved changes from GitHub

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

### 5.5.1.11 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.

Changes History main (all commits)

Search Pull

Subject	Author	Date	SHA
origin/main Merged changes from Owner and Collaborator.	Matt Jones <gitcode@magisa.org>	2020-10-20	a164bbf4
Change made by collaborator.	metamattj <matt@magisa.org>	2020-10-20	659d6da0
Changed made by owner.	Matt Jones <gitcode@magisa.org>	2020-10-20	19a85340
Fix spelling	Matt Jones <gitcode@magisa.org>	2020-10-20	0c471c8b
Initial readme	metamattj <matt@magisa.org>	2020-10-20	979ba526
Initial RMarkdown lesson.	Matt Jones <gitcode@magisa.org>	2020-10-20	d6434629

Commits 1-6 of 6

SHA a164bbf4f18fb3a1982d9ba7f23f5d763dba94a

Author Matt Jones <gitcode@magisa.org>

Date 2020-10-20 01:43

Subject Merged changes from Owner and Collaborator.

Parent 19a853406b46eced71f88fc71d677c9d94108a73 659d6da0f6a163a72fedfb99359aae8c8898b403

README.md

View file @ a164bbf4

```

@@@ -1,6 +1,6 @@@
1 1 # training_jones
2 2 Training repository for the Arctic Data Center training course
3 3
4 - Data (mbjones was here)
4 - Data (metamattj made this change, as the collaborator)
4 - Data (mbjones and metamattj reconciled)
5 5 - Metadata
6 6 - Science

```

## 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

## 5.6 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 the 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.

## AIM: RESET AND REVERT

While Working with Git in certain situations we want to undo changes in the working area or index area, sometimes remove commits locally or remotely and we need to reverse those changes. There are 3 different ways in which we can undo the changes in our repository, these are ***git reset***, ***git checkout***, and ***git revert***. *git checkout* and *git reset* in fact can be used to manipulate commits or individual files. These commands can be confusing so it's important to find out the difference between them and to know which command should be used at a particular point of time.

```
devwani@LAPTOP-CPH0E103 MINGW64 ~ (master)
$ git log -p -1
commit 2974631310657b9a42eff5fdedf7ca51a17d2aea (HEAD -> master, origin/master,
scm)
Author: devwani0430 <devwani0430.be21@chitkara.edu.in>
Date:   Sun Jun 5 14:24:07 2022 +0530

    added commit

diff --git a/file.txt b/file.txt
new file mode 100644
index 0000000..e69de29
```



```
$ git status exp.txt
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

We can see that we have a single commit is done and the text document that has been committed with Hello Geeks in it. Now let's add some more text to our text document. Let's add a line inside the exp.txt. Doing this change, our file now needs to be added to the staging area for getting the commit done. These updates are currently in the working area and to see them we will see those using ***git status***.

We want to unstage a file and the command that we would be using to unstage our file is –

## 1. git reset

git reset is used when we want to unstage a file and bring our changes back to the working directory. git reset can also be used to remove commits from the local repository.

```
git reset HEAD <filename>
```

Whenever we unstage a file, all the changes are kept in the working area.

We are back to the working directory, where our changes are present but the file is now unstaged. Now there are also some commits that we don't want to get committed and we want to remove them from our local repository. To see how to remove the commit from our local repository let's stage and commit the changes that we just did and then remove that commit.

We have 2 commits now, with the latest being the Added Hello World commit which we are going to remove. The command that we would be using now is -

```
git reset HEAD~1
```

Points to be noted -

- HEAD~1 here means that we are going to remove the topmost commit or the latest commit that we have done.
- We cannot remove a specific commit with the help of git reset , for ex : we cannot say that we want to remove the second commit or the third commit , we can only remove latest commit or latest 2 commits ... latest N commits.(HEAD~n) [n here means n recent commits that needs to be deleted].

After using the above command, we can see that our commit is being deleted and also our file is again unstaged and is back to the working directory. There are different ways in which git reset can actually keep your changes.

- **git reset --soft HEAD~1** - This command will remove the commit but would not unstage a file. Our changes still would be in the staging area.
- **git reset --mixed HEAD~1** or **git reset HEAD~1** - This is the default command that we have used in the above example which removes the commit as well as unstages the file and our changes are stored in the working directory.
- **git reset --hard HEAD~1** - This command removes the commit as well as the changes from your working directory. This command can also be called destructive command as we would not be able to get back the changes so be careful while using this command.

Points to keep in mind while using git reset command -

- If our commits are not published to remote repository , then we can use git reset.
- Use git reset only for removing commits that are present in our local directory and not in remote directory.
- We cannot remove a specific commit with the help of git reset , for ex : we cannot say that we want to remove the second commit or the third commit , we can only remove latest commit or latest 2 commits ... latest N commits.(HEAD~n) [n here means n recent commits that needs to be deleted].

We just discussed above that the git reset command cannot be used to delete commits from the remote repository, then how do we remove the unwanted commits from the remote repository T

```
$ git status exp.txt
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   exp.txt
```

## 2. git revert

git revert is used to remove the commits from the remote repository. Since now our changes are in the working directory, let's add those changes to the staging area and commit them.

Now we want to delete the commit that we just added to the remote repository. We could have used the git reset command but that would have deleted the commit just from the local repository and not the remote repository. If we do this then we would get conflict that the remote commit is not present locally. So, we do not use git reset here. The best we can use here is git revert.

git revert <commit id of the commit that needs to be removed>

```
commit 2974631310657b9a42eff5fdedf7ca51a17d2aea (HEAD -> master, origin/master,
SCM)
Author: devwani0430 <devwani0430.be21@chitkara.edu.in>
Date: Sun Jun 5 14:24:07 2022 +0530

    added commit

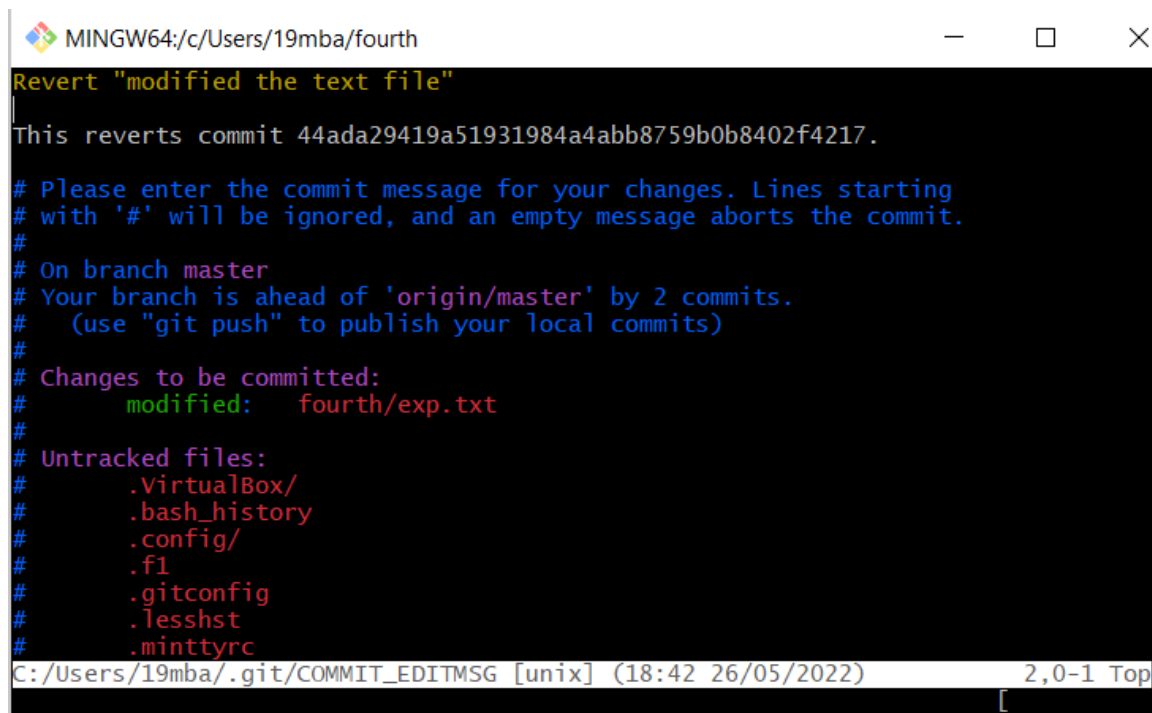
commit defd70f4c3d8c9936e7eb11280e923168d91cc9e (main)
Author: devwani0430 <devwani0430.be21@chitkara.edu.in>
Date: Sun Apr 10 23:06:33 2022 +0530

    hi

commit 3aa889944b8d7666938d0ac29c2750c2416931a6 (checkout)
Author: devwani0430 India <devwani0430.be21@chitkara.edu.in>
Date: Sun Apr 10 19:28:26 2022 +0530

    Hello

commit a8b1451892558ad62ed19a9d75360af733581f27
Author: Devwani <devwani0430.be21@chitkara.edu.in>
Date: Sun Apr 10 16:53:53 2022 +0530
```



```

MINGW64:/c/Users/19mba/fourth
Revert "modified the text file"

This reverts commit 44ada29419a51931984a4abb8759b0b8402f4217.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch master
# Your branch is ahead of 'origin/master' by 2 commits.
#   (use "git push" to publish your local commits)
#
# Changes to be committed:
#   modified:   fourth/exp.txt
#
# Untracked files:
#   .VirtualBox/
#   .bash_history
#   .config/
#   .fl
#   .gitconfig
#   .lessht
#   .minttyrc
C:/Users/19mba/.git/COMMIT_EDITMSG [unix] (18:42 26/05/2022) 2,0-1 Top

```

Figure 1 Git log befo

Figure 2 Git log after revert

Points to keep in mind -

- Using git revert we can undo any commit , not like git reset where we could just remove "n" recent commits.

Now let's first understand what git revert does, git revert removes the commit that we have done but adds one more commit which tells us that the revert has been done. Let's look at the example –

**Note:** If you see the lines as we got after the git revert command, just visit your default text editor for git and commit the message from there, or it could directly take to you your default editor. We want a message here because when using git revert, it does not delete the commit instead makes a new commit that contains the removed changes from the commit.

We can see that the new commit is being added. However since this commit is in local repository so we need to do **git push** so that our remote repository also notices that the change has been done.

And as we can see we have a new commit in our remote repository and **Hello World** which we added in our 2nd commit is being removed from the local as well as the remote repository. Let's summarize the points that we saw above -

## Difference Table

<b>git checkout</b>	<b>git reset</b>	<b>git revert</b>
Discards the changes in the working repository.	Unstages a file and bring our changes back to the working directory	Removes the commits from the remote repository.
Used in the local repository.	Used in local repository	Used in the remote repository
Does not make any changes to the commit history.	Alters the existing commit history	Adds a new commit to the existing commit history.
Moves HEAD pointer to a specific commit.	Discards the uncommitted changes.	Rollbacks the changes which we have committed.
Can be used to manipulate commits or files.	Can be used to manipulate commits or files.	Does not manipulate your commits or files.