Subject Name: **Source Code Management**

Subject Code: **CS181**

Cluster: **Zeta**

Department: **DCSE**

Submitted By:
Paras Chauhan
2110992032
G27

Submitted To:
Dr.Anuj Jain

List of Programs

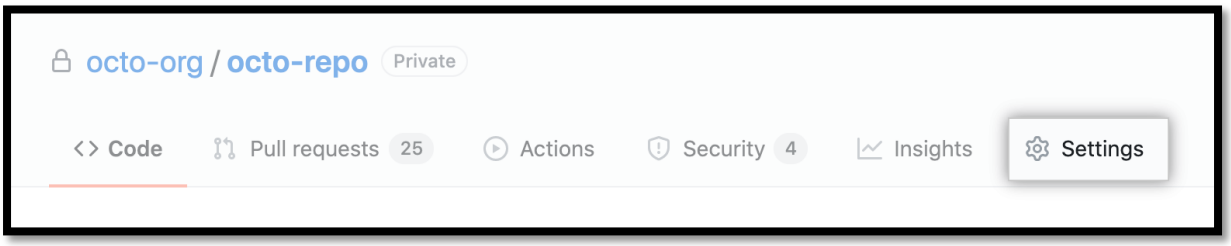
Experiment 1

Aim: Add collaborators on GitHub Repo.

What is repository GitHub?

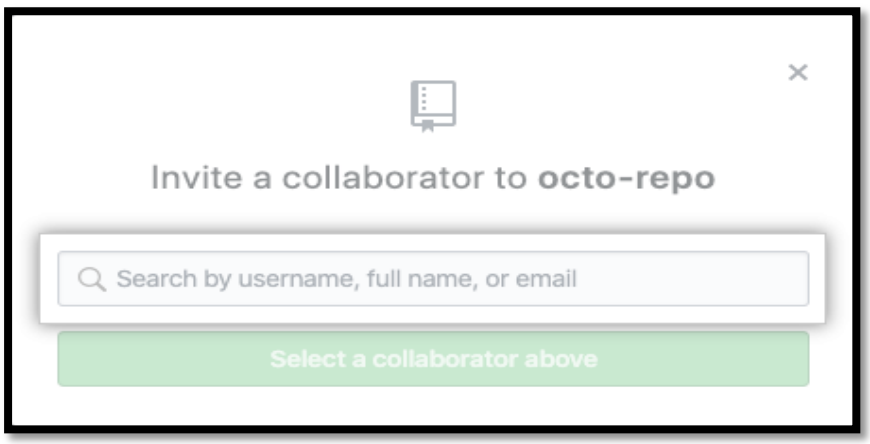
A repository **contains all of your project's files and each file's revision history**. You can discuss and manage your project's work within the repository.

1. Ask for the username of the person you're inviting as a collaborator. If they don't have a username yet, they can sign up for GitHub For more information, see "Signing up for a new GitHub account".
2. On GitHub.com, navigate to the main page of the repository.

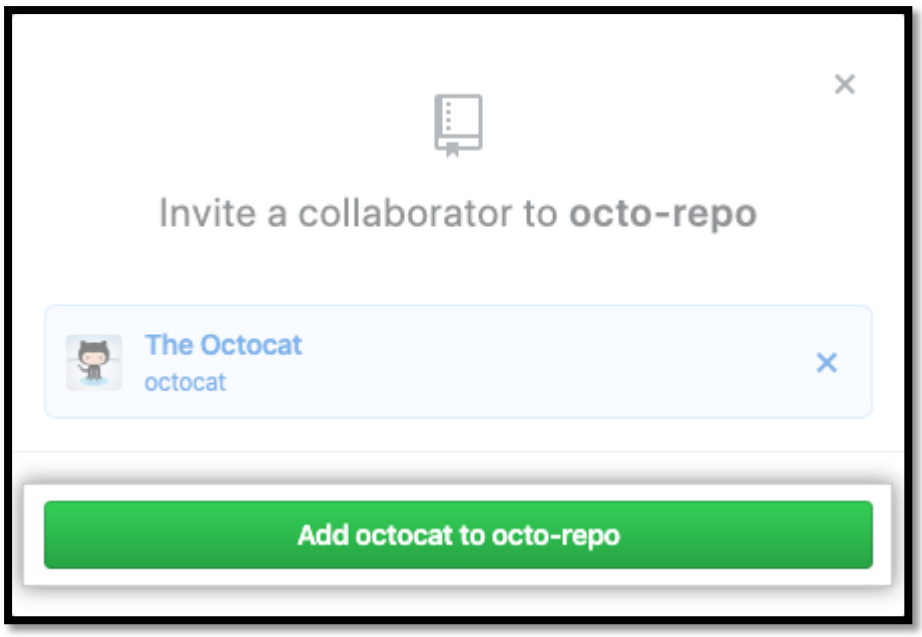


3. Under your repository name, click **Settings**.
4. In the "Access" section of the sidebar, click **Collaborators & teams**.
5. Click **Invite a collaborator**.

6. In the search field, start typing the name of person you want to invite, then click a name in the list of matches.



7. Click **Add NAME to REPOSITORY**.



8. The user will receive an email inviting them to the repository. Once they accept your invitation, they will have collaborator access to your repository.

Experiment 2

Aim: Fork and Commit.

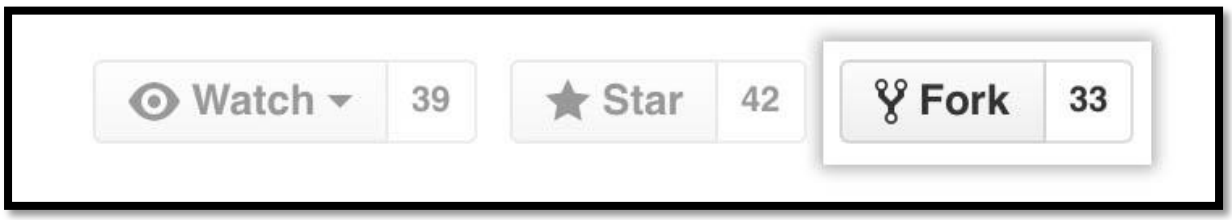
About forks

Most commonly, forks are used to either propose changes to someone else's project to which you don't have write access, 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."

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 repository you just forked.

1. On GitHub.com, navigate to the octocat/Spoon-Knife repository.
2. In the top-right corner of the page, click **Fork**.



Git Commit

`git commit` creates a commit, which 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.

How to Use Git Commit

Common usages and options for Git Commit

- `git commit`: This starts the commit process, but since it doesn't include a `-m` flag for the message, your default text editor will be opened for you to create the commit message. If you haven't configured anything, there's a good chance this will be VI or Vim. (To get out, press esc, then `:w`, and then Enter. :wink:)
- `git commit -m "descriptive commit message"`: This starts the commit process, and allows you to include the commit message at the same time.
- `git commit -am "descriptive commit message"`: In addition to including the commit message, this option allows you to skip the staging phase. The addition of `-a` will automatically stage any files that are already being tracked by Git (changes to files that you've committed before).
- `git commit --amend`: Replaces the most recent commit with a new commit. To see all of the possible options you have with `git commit`, check out Git's documentation.

```
ADMIN@DESKTOP-BOPQ8T2 MINGW64 ~
$ pwd
/c/Users/ADMIN

ADMIN@DESKTOP-BOPQ8T2 MINGW64 ~
$ mkdir git-project

ADMIN@DESKTOP-BOPQ8T2 MINGW64 ~
$ cd git-project

ADMIN@DESKTOP-BOPQ8T2 MINGW64 ~/git-project
$ git init
Initialized empty Git repository in C:/Users/ADMIN/git-project/.git/

ADMIN@DESKTOP-BOPQ8T2 MINGW64 ~/git-project (master)
$ touch file.txt

ADMIN@DESKTOP-BOPQ8T2 MINGW64 ~/git-project (master)
$ git add file.txt

ADMIN@DESKTOP-BOPQ8T2 MINGW64 ~/git-project (master)
$ git commit -m "master"
[master (root-commit) 305350c] master
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file.txt

ADMIN@DESKTOP-BOPQ8T2 MINGW64 ~/git-project (master)
$ git status
On branch master
nothing to commit, working tree clean

ADMIN@DESKTOP-BOPQ8T2 MINGW64 ~/git-project (master)
$
```

Experiment 3

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

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.

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 command line 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

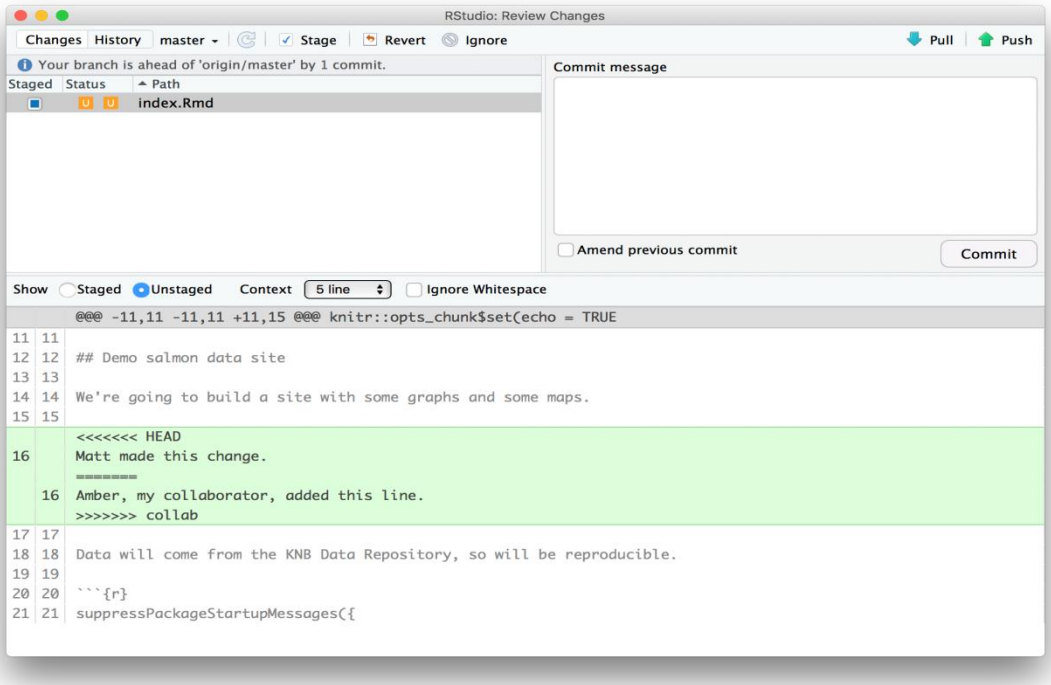
The simplest way to resolve a conflict, given that you know whose version of the file you want to keep, is to use the commandline `git` program to tell git to use either *your* changes (the person doing the merge), or *their* changes (the other collaborator).

- 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 R Studio, 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 `pull` the file with a

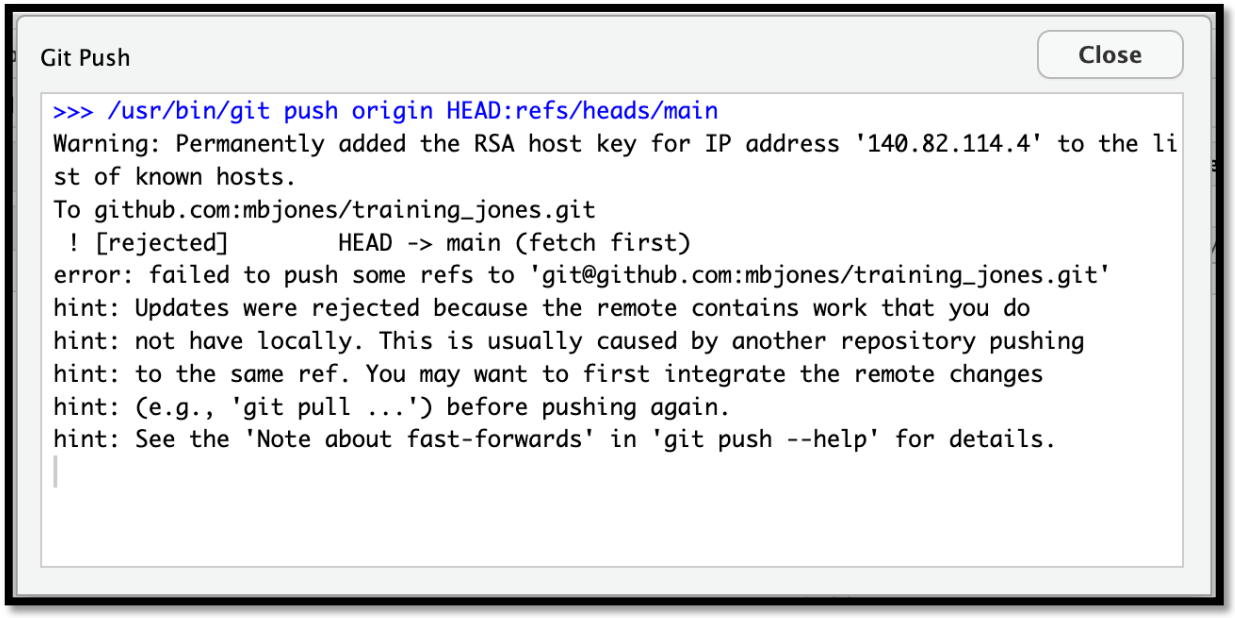


conflict, git notices that there is a conflict and modifies the file to show both your own changes and your collaborator's changes in the file. It also shows the file in the Git tab with an orange `U` icon, which indicates that the file is `Unmerged`, and therefore awaiting you help to resolve the conflict. 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.

1. Producing and resolving merge conflicts

- Owner and collaborator ensure all changes are updated
- Owner makes a change and commits
- Collaborator makes a change and commits **on the same line**
- Collaborator pushes the file to GitHub
- Owner pushes their changes and gets an error



- Owner pulls from GitHub to get Collaborator changes
- Owner edits the file to resolve the conflict
- Owner commits the resolved changes
- Owner pushes the resolved changes to GitHub
- Collaborator pulls the resolved changes from GitHub
- Both can view commit history

Experiment 4

Aim: Reset and Revert

Reset and Revert

The purpose of the git revert command is to remove all the changes a single commit made to your source code repository. For example, if a past commit added a file named index.html to the repo, a git revert on that commit will remove the index.html file from the repo. If a past commit added a new line of code to a Java file, a git revert on that commit will remove the added line.

A git revert commit example

To really understand how to undo Git commits, look at this git revert example. We will start with a git init command to create a completely clean repository:

```
git@commit /c/revert example/

$ git init

Initialized empty Git repo in C:/git revert example
```

With the repository initialized, we'll add five files to the repo. Each time a new file is created, we add it to the Git index and create a new commit with a meaningful message.

```
git@commit /c/revert example/

$ touch alpha.html

$ git add . && git commit -m "1st git commit: 1 file"


$ touch beta.html

$ git add . && git commit -m "2nd git commit: 2 files"


$ touch charlie.html

$ git add . && git commit -m "3rd git commit: 3 files"


$ touch delta.html

$ git add . && git commit -m "4th git commit: 4 files"
```

```
$ touch edison.html

$ git add . && git commit -m "5th git commit: 5 files"
```

A quick directory listing following the initial command batch shows five files in the current folder:

```
git@commit /c/revert example/

$ ls

alpha.html  beta.html  charlie.html delta.html  edison.html
```

A call to the git reflog command will show us our current commit history:

```
git@commit /c/revert example/

$ git reflog

(HEAD -> master)

d846aa8 HEAD@{0}: commit: 5th git commit: 5 files

0c59891 HEAD@{1}: commit: 4th git commit: 4 files

4945db2 HEAD@{2}: commit: 3rd git commit: 3 files

defc4eb HEAD@{3}: commit: 2nd git commit: 2 files

2938ee3 HEAD@{4}: commit: 1st git commit: 1 file
```

How to revert a Git commit

What do you think would happen if we did a git revert on the third commit with ID 4945db2? This was the git commit that added the charlie.html file.

```
git@commit /c/revert example/

$ git revert 4945db2
```

Will the git revert of commit 4945db2
remove charlie.html, delta.html and edison.html from the local workspace?

Will the git revert of commit 4945db2 remove delta.html and edison.html from the local workspace, but leave the other files alone?

Or will this git revert example leave four files in the local workspace and remove only the charlie.html file?

If you chose the last outcome, you'd be correct. Here's why.

`The git revert command will undo only the changes associated with a specific commit. In this git revert example, the third commit added the charlie.html file. When we revert said Git commit, the only file removed from our repository is charlie.html.

```
git@commit /c/revert example/
$ ls
alpha.html  beta.html  delta.html  edison.html
```

Developers also need to know that when they git revert a commit, the reverted commit is deleted from their local workspace, but not deleted from the local repository. The code associated with the reverted Git commit remains stored in the repository's history of changes, which means reverted code is still referenceable if it ever needs to be accessed or reviewed in the future.