Subject Name: **Source Code Management**

Subject Code: **CS181**

Cluster: Zeta

Department: DCSE

Submitted By:

Nawed Alam
2110992080
G27

Submitted To:

Dr. Anuj Jain

CHITKARA
UNIVERSITY

List of Tasks

Task 1

Installing and Configuring the Git client

The following sections list the steps required to properly install and configure the Git clients - Git Bash and Git GUI - on a Windows 7 computer.

Git is also available for Linux and Mac. The remaining instructions here, however, are specific to the Windows installation.

Be sure to carefully follow all of the steps in the first five sections. The last section, 6, is optional.

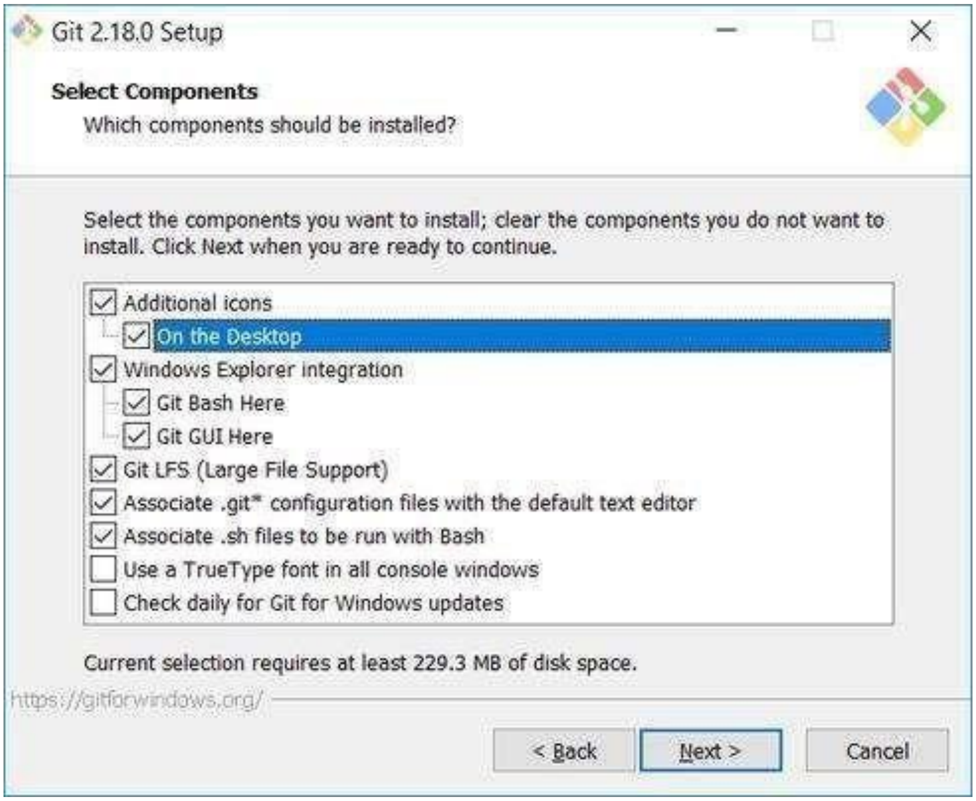
There is also a section on common problems and possible fixes at the bottom of the document.

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:



Git 2.18.0 Setup

Select Components

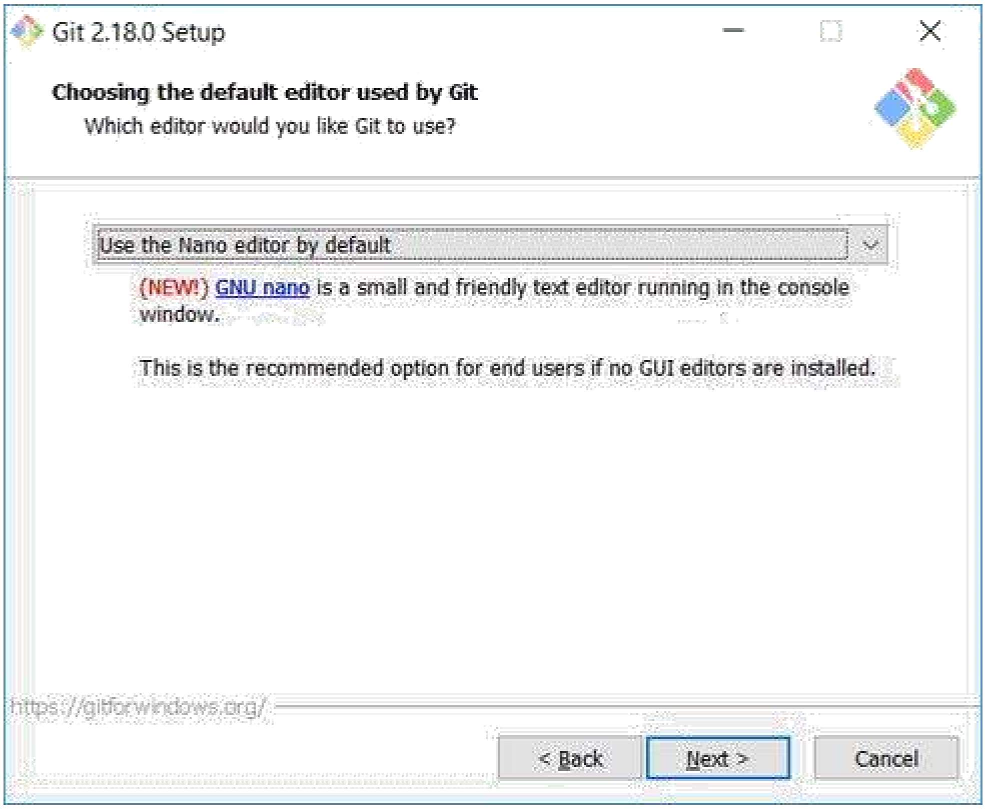
Which components should be installed?

Select the components you want to install; clear the components you do not want to install. Click Next when you are ready to continue.

- ☑ Additional icons
  - ☑ On the Desktop
- ☑ Windows Explorer integration
  - ☑ Git Bash Here
  - ☑ Git GUI Here
- ☑ Git LFS (Large File Support)
- ☑ Associate .git* configuration files with the default text editor
- ☑ Associate .sh files to be run with Bash
- □ Use a TrueType font in all console windows
- □ Check daily for Git for Windows updates

Current selection requires at least 229.3 MB of disk space.

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.



Git 2.18.0 Setup

Choosing the default editor used by Git

Which editor would you like Git to use?

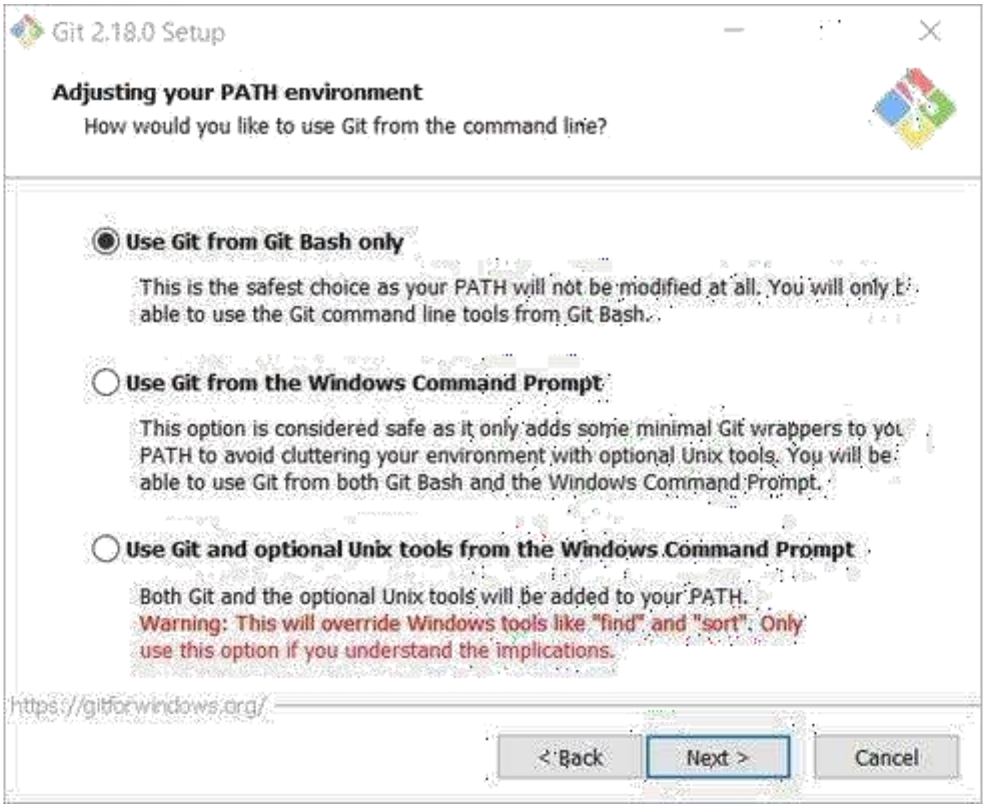
Use the Nano editor by default

(NEW!) GNU nano is a small and friendly text editor running in the console window.

This is the recommended option for end users if no GUI editors are installed.

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.



Adjusting your PATH environment

How would you like to use Git from the command line?

● **Use Git from Git Bash only**

This is the safest choice as your PATH will not be modified at all. You will only be able to use the Git command line tools from Git Bash.

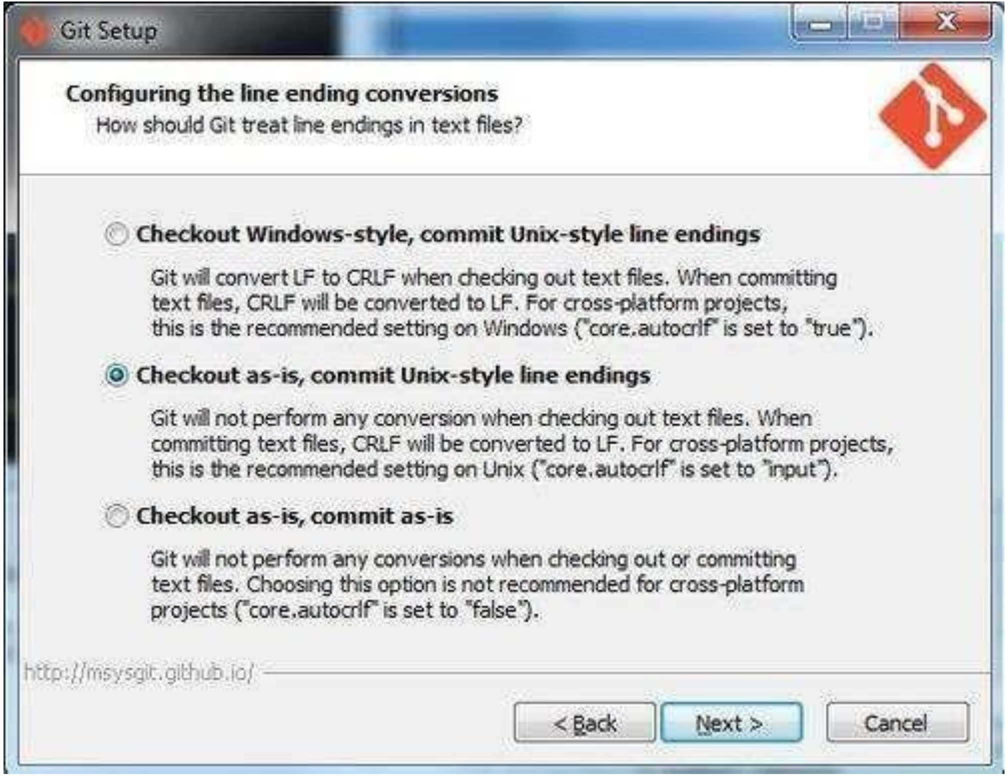
○ **Use Git from the Windows Command Prompt**

This option is considered safe as it only adds some minimal Git wrappers to your PATH to avoid cluttering your environment with optional Unix tools. You will be able to use Git from both Git Bash and the Windows Command Prompt.

○ **Use Git and optional Unix tools from the Windows Command Prompt**

Both Git and the optional Unix tools will be added to your PATH.
Warning: This will override Windows tools like "find" and "sort". Only use this option if you understand the implications.

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.



Configuring the line ending conversions

How should Git treat line endings in text files?

○ **Checkout Windows-style, commit Unix-style line endings**

Git will convert LF to CRLF when checking out text files. When committing text files, CRLF will be converted to LF. For cross-platform projects, this is the recommended setting on Windows ("core.autocrlf" is set to "true").

● **Checkout as-is, commit Unix-style line endings**

Git will not perform any conversion when checking out text files. When committing text files, CRLF will be converted to LF. For cross-platform projects, this is the recommended setting on Unix ("core.autocrlf" is set to "input").

○ **Checkout as-is, commit as-is**

Git will not perform any conversions when checking out or committing text files. Choosing this option is not recommended for cross-platform projects ("core.autocrlf" is set to "false").

2. Configuring Git to ignore certain files

This part is extra important and required so that your repository does not get cluttered with garbage files.

By default, Git tracks **all** files in a project. Typically, this is **NOT** what you want; rather, you want Git to ignore certain files such as **.bak** files created by an editor or **.class** files created by the Java compiler. To have Git automatically ignore particular files, create a file named **.gitignore** (note that the filename begins with a dot) in the **C:\users\name** folder (where name is your MSOE login name).

NOTE: The .gitignore file must NOT have any file extension (e.g. .txt). Windows normally tries to place a file extension (.txt) on a file you create from File Explorer - and then it (by default) HIDES the file extension. To avoid this, create the file from within a useful editor (e.g. Notepad++ or UltraEdit) and save the file without a file extension).

Edit this file and add the lines below (just copy/paste them from this screen); these are patterns for files to be ignored (taken from examples provided at https://github.com/github/gitignore.)

```
#Lines (like this one) that begin with # are comments; all other lines are rules

# common build products to be ignored at
MSOE *.o
*.obj
*.class
*.exe

# common IDE-generated files and folders to
ignore workspace.xml
bin/
out/
.classpath
# uncomment following for courses in which Eclipse .project files are not checked in
# .project

#ignore automatically generated files created by some common applications, operating
systems
*.bak
*.log
*.ldb
~*
.DS_Store*
._*
Thumbs.db

# Any files you do want not to ignore must be specified starting with !
# For example, if you didn't want to ignore .classpath, you'd uncomment the following rule:
# !.classpath
```

Note: You can always edit this file and add additional patterns for other types of files you might want to ignore. Note that you can also have a .gitignore file in any folder naming additional files to ignore. This is useful for project -specific build products.

3. Configuring Git default parameters

Once Git is installed, there is some remaining custom configuration you must do. Follow the steps below:

a. From within File Explorer, right-click on any folder. A context menu appears containing the commands "**Git Bash here**" and "**Git GUI here**". These commands permit you to launch either Git client. For now, select **Git Bash here**.

b. Enter the command (replacing name as appropriate) `git config --global core.excludesfile c:/users/name/. gitignore`

- This tells Git to use the **.gitignore** file you created in step 2
- NOTE: TO avoid typing errors, copy and paste the commands shown here into the Git Bash window, using the arrow keys to edit the red text to match your information.

c. Enter the command `git config --global user.email "name@msoe.edu"`

- This links your Git activity to your email address. Without this, your commits will often show up as "unknown login".
  Replace name with your own MSOE email name.

d. Enter the command `git config --global user.name "Your Name"`

- Git uses this to log your activity. Replace "Your Name" by your actual first and last name.

e. Enter the command `git config --global push.default simple`

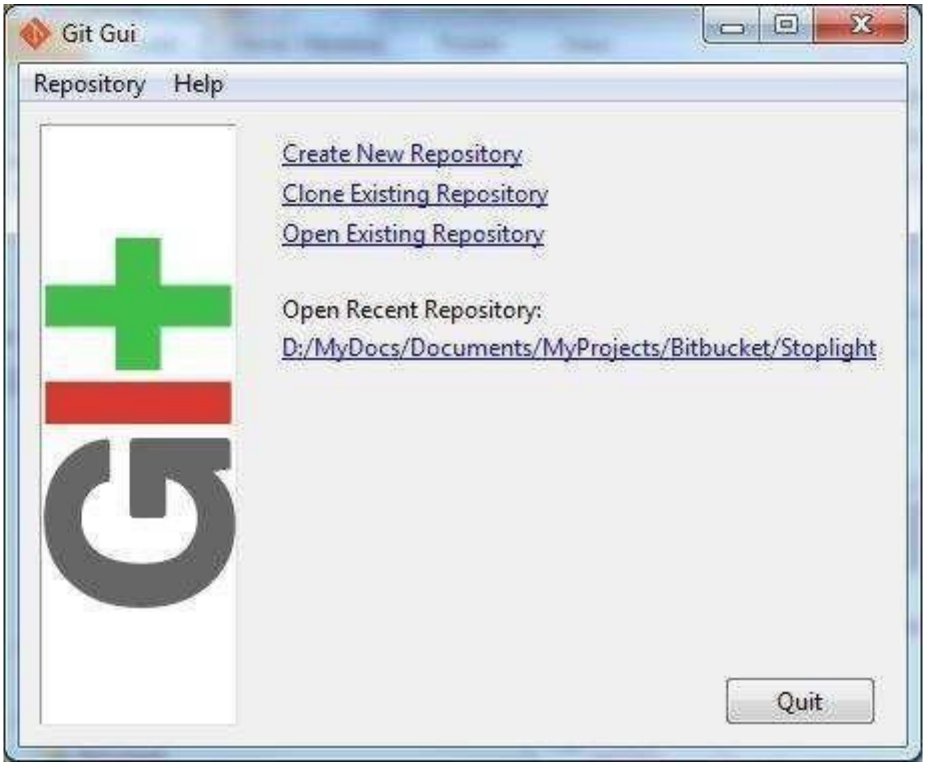
- This ensures that all pushes go back to the branch from which they were pulled. Otherwise pushes will go to the master branch, forcing a merge.

4. Generating public/private key pairs for authentication

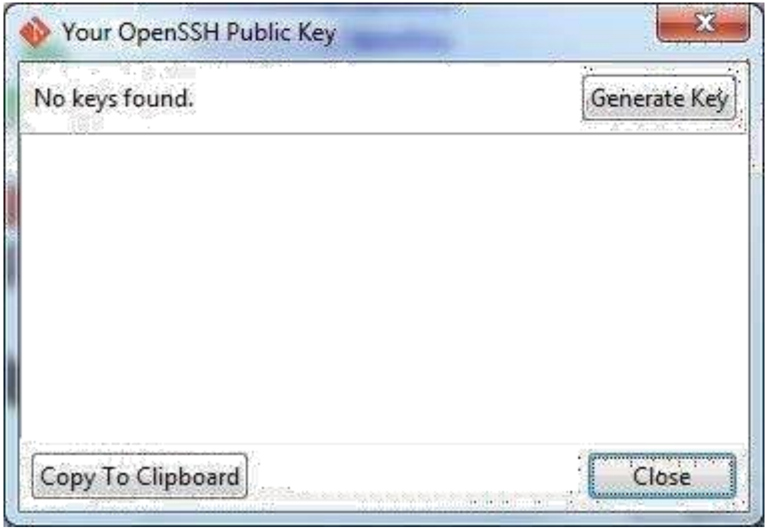
This part is critical and used to authenticate your access to the repository.

You will eventually be storing your project files on a remote Bitbucket or other server using a secure network connection. The remote server requires you to authenticate yourself whenever you communicate with it so that it can be sure it is you, and not someone else trying to steal or corrupt your files. Bitbucket and Git together user public key authentication; thus you have to generate a pair of keys: a public key that you (or your instructor) put on Bitbucket, and a private key you keep to yourself (and guard with your life).

Generating the key pair is easy: From within File Explorer, right-click on any folder. From the context menu, select **Git GUI Here** . The following appears:

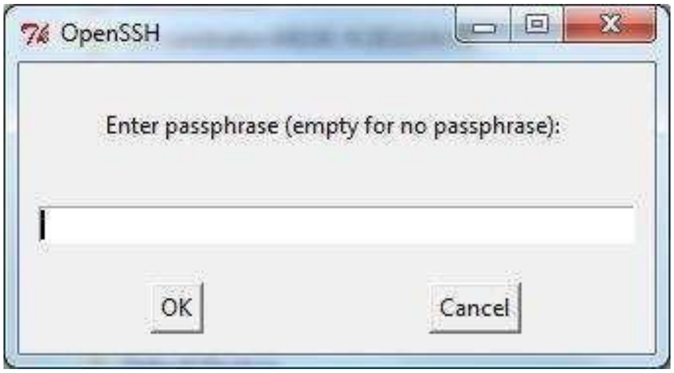


From the **Help** menu, select the **Show SSH Key** command. The following pup-up dialog appears:



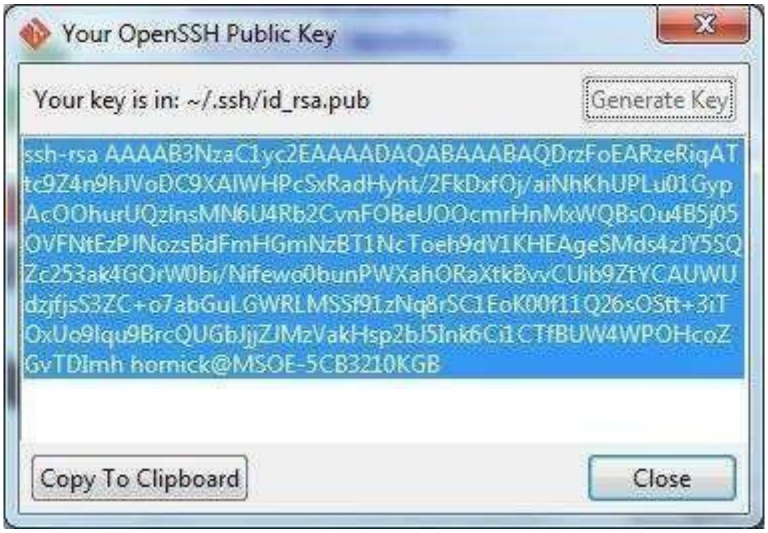
No keys found.

Initially, you have no public/private key pair; thus the message "**No keys found**" appears withing the dialog. Press the **Generate Key** button. The following dialog appears:



Enter passphrase (empty for no passphrase):

Do **NOT** enter a passphrase - just press **OK** twice. When you do, the dialog disappears and you should see something like the following - but your generated key will be different:



Your OpenSSH Public Key

Your key is in: ~/.ssh/id_rsa.pub

```
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDrzFoEARzeRiqAT
tc9Z4n9hJVoDC9XAIWHPcSxRadHyht/2FkDxfOj/aiNhKhUPLu01Gyp
AcOOhurUQzInsMN6U4Rb2CvnFOBeUOOcmrHnMxWQBsOu4B5j05
OVFNtEzPJNozsBdFmHGmNzBT1NcToeh9dV1KHEAgeSMds4zJY5SQ
Zc253ak4GOrW0bi/Nifewo0bunPWXahORaXtkBvvCUib9ZtYCAUWU
dzjfjsS3ZC+o7abGuLGWRLMSSf91zNq8rSC1EoK00f11Q26sOStt+3iT
OxUo9Iqu9BrcQUGbJjjZJMzVakHsp2bJ5Ink6Ci1CTfBUW4WPOHcoZ
GvTDImh hornick@MSOE-5CB3210KGB
```

The keys have been written into two files named **id_rsa** and **id_rsa.pub** in your **c:/Users/*username*/.ssh** folder (where ***username*** is your MSOE user name). Don't ever delete these files! To configure Bitbucket to use this key:

1. Click on the **Copy to Clipboard** button in the Git GUI Public Key dialog.
2. Log in to BItbucket
3. Click on your picture or the 👤 icon in the left pane and select **Settings**.
4. Select **SSH keys** under **Security**.
5. Click on the **Add key** button.
6. Enter a name for your key in the **Label** box in the Bitbucket window. If your key is ever compromised (such as someone gets a copy off of your laptop), having a clear name will help you know which key to delete. A good pattern to follow is to name the computer used to generate the key followed by the date you generated it; for instance: "MSOE laptop key 2012-02-28".
7. Paste the key from the Clipboard into the **Key** text box in the Bitbucket window, and add it.

You should now be able to access your repository from your laptop using the ssh protocol without having to enter a password. Protect the key files - other people can use them to access your repository as well! If you have another computer you use, you can copy the id_rsa.pub file to the .ssh folder on that computer or (better yet) you can generate another public/private key pair specific to that computer.

Configuring other repositories (such as GitLab) is very similar.

5. Authenticating with private keys

Linux, Mac users:

1. Open a terminal prompt.
2. Type the commands

```
eval `ssh    -agent`
ssh-add
```

Note that backticks (`) are used, not forward ticks ('). The second command assumes your key is in the default location, ~/.ssh/id_dsa. If it is somewhere else, type `ssh-add` *path-to-private-key-file.* Note that the directory containing the ssh key cannot be readable or writeable by other users; that is, it needs mode 700. You can add these commands to your .bashrc file so they are executed every time you log in.
Otherwise the keys only remain active until you close the shell you are using or log out.

Windows users:

1. Install Pageant if it is not installed. It is usually installed with PuTTY and PuTTYgen.
2. Start Pageant and select **Add Key**.
3. Browse to your .ppk file, open it, and enter the passphrase if prompted.

If git pull or get push cannot connect, you might need to add a system variable `GIT_SSH` set to the path to the `plink.exe` executable. Go to Windows Settings, enter "system environment" in the search box, open the "Edit the system environment variables" item, click on Environment Variables..., then New... in the System Variables section (the bottom half), enter `GIT_SSH` for the name, and browse to `plink.exe` for the value. Save the setting, then reboot your computer.

You will need to re-add the key to Pageant every time you log in to Windows (say, after a reboot). The command `start/b pageant c:\path\to\file.ppk` will open the file in pageant if you have pageant in your %PATH% variable.

6. Optional: Configure Git to use a custom application (WinMerge) for comparing file differences

It is recommended that you skip this step unless you really are attached to using WinMerge for file comparison tasks.

a. Enter the command `git config --global merge.tool winmerge`

This configures Git to use the application WinMerge to resolve merging conflicts. You must have WinMerge installed on your computer first. Get WinMerge at http://winmerge.org/downloads/.

B. Enter the following commands to complete the WinMerge configuration:

i. `git config --global mergetool.winmerge.name WinMerge`

ii. git config --global mergetool.winmerge.trustExitCode true

iii. If you install WinMerge to the default location (that is, C:\Program Files (x86)\WinMerge), enter git config --global mergetool.winmerge.cmd "\"C:\Program Files (x86)\WinMerge\WinMergeU.exe\" -u -e -dl \"Local\" -dr \"Remote\" \$LOCAL \$REMOTE \$MERGED"

iv. If you install WinMerge to an alternate location (for example, D:\WinMerge), enter git config --global mergetool.winmerge.cmd "/d/WinMerge/WinMergeU.exe -u -e -dl \"Local\" -dr \"Remote\" \$LOCAL \$REMOTE \$MERGED"

c. Enter the command `git config --global diff.tool winmerge`

This configures Git to use the application WinMerge to differences between versions of files.

d. Enter the commands to complete the WinMerge diff configuration:

i. `git config --global difftool.winmerge.name WinMerge`

ii. `git config --global difftool.winmerge.trustExitCode true`

iii. If you install WinMerge to the default location (that is, C:\Program Files (x86)\WinMerge), enter `git config --global difftool.winmerge.cmd "\"C:\Program Files (x86)\WinMerge\WinMergeU.exe\" -u -e \$LOCAL \$REMOTE"`

iv. If you install WinMerge to an alternate location (for example, D:\WinMerge), enter `git config --global difftool.winmerge.cmd "/d/WinMerge/WinMergeU.exe -u -e \$LOCAL \$REMOTE"`

Common problems and possible fixes

Problem	Fix
In Linux, pushes and pulls do not work with the error message "**permission denied".**	Check the ownership and permissions on the .git folder Use "`ls -ld .git`" to check this. If the owner is root, use the `chown` command to fix that. If the permissions are not rwx for the owner, use the `chmod` command to set the permissions to 770.
Even though your keys are set up and available through `ssh-agent` or `pageant`, git push and pull commands prompt you for your username and password.	Confirm the repository address by entering "`git remote -v show`". If it shows the HTTPS protocol (starts with HTTPS), then you need to clone the repository using the SSH protocol specifier.

Task 2

This guide will walk you through setting up your GitHub account and getting started with GitHub's features for collaboration and community.

Part 1: Configuring your GitHub account

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.

1. 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."

2. 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."

3. 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."

4. 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."

5. 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" and "Viewing contributions on your profile."

Part 2: Using GitHub's tools and processes

To best use GitHub, you'll need to set up Git. Git is responsible for everything GitHubrelated that happens locally on your computer. To effectively collaborate on GitHub, you'll write in issues and pull requests using GitHub Flavored Markdown.

1. Learning Git

GitHub's collaborative approach to development depends on publishing commits from your local repository to GitHub for other people to view, fetch, and update using Git. For more information about Git, see the "Git Handbook" guide. For more information about how Git is used on GitHub, see "GitHub flow."

2. Setting up Git

If you plan to use Git locally on your computer, whether through the command line, an IDE or text editor, you will need to install and set up Git. For more information, see "Set up Git."

If you prefer to use a visual interface, you can download and use GitHub Desktop. GitHub Desktop comes packaged with Git, so there is no need to install Git separately. For more information, see "Getting started with GitHub Desktop."

Once you install Git, you can connect to GitHub repositories from your local computer, whether your own repository or another user's fork. When you connect to a repository on GitHub.com from Git, you'll need to authenticate with GitHub using either HTTPS or SSH. For more information, see "About remote repositories."

3. Choosing how to interact with GitHub

Everyone has their own unique workflow for interacting with GitHub; the interfaces and methods you use depend on your preference and what works best for your needs.

For more information about how to authenticate to GitHub with each of these methods, see "About authentication to GitHub."

Method	Description	Use cases
Browse to GitHub.com	If you don't need to work with files locally, GitHub lets you complete most Git-related actions directly in the browser, from creating and forking repositories to editing and opening pull requests.	This method is useful if you want a visual interface and need to do quick, simple changes that don't require working files locally.

Method	Description	Use cases
GitHub Desktop	GitHub Desktop extends and simplifies your GitHub.com workflow, using a visual interface instead of text commands on the command line. For more information on getting started with GitHub Desktop, see "Getting started with GitHub Desktop."	This method is best if you need or want to work with files locally, but prefer using a visual interface to use Git and interact with GitHub.
IDE or text editor	You can set a default text editor, like Atom or Visual Studio Code to open and edit your files with Git, use extensions, and view the project structure. For more information, see "Associating text editors with Git."	This is convenient if you are working with more complex files and projects and want everything in one place, since text editors or IDEs often allow you to directly access the command line in the editor.
Command line, with or without GitHub CLI	For the most granular control and customization of how you use Git and interact with GitHub, you can use the command line. For more information on using Git commands, see "Git cheatsheet." GitHub CLI is a separate command-line tool you can install that brings pull requests, issues, GitHub Actions, and other GitHub features to your terminal, so you can do all your work in one place. For more information, see "GitHub CLI."	This is most convenient if you are already working from the command line, allowing you to avoid switching context, or if you are more comfortable using the command line.
GitHub API	GitHub has a REST API and GraphQL API that you can use to interact with GitHub. For more information, see "Getting started with the API."	The GitHub API would be most helpful if you wanted to automate common tasks, back up your data, or create integrations that extend GitHub.

4. Writing on GitHub

To make your communication clear and organized in issues and pull requests, you can use GitHub Flavored Markdown for formatting, which combines an easy-to-read, easyto-write syntax with some custom functionality. For more information, see "About writing and formatting on GitHub."

You can learn GitHub Flavored Markdown with the "Communicating using Markdown" course on GitHub Learning Lab.

5. Searching on GitHub

Our integrated search allows you to find what you are looking for among the many repositories, users and lines of code on GitHub. You can search globally across all of GitHub or limit your search to a particular repository or organization. For more information about the types of searches you can do on GitHub, see "About searching on GitHub."

Our search syntax allows you to construct queries using qualifiers to specify what you want to search for. For more information on the search syntax to use in search, see "Searching on GitHub."

6. Managing files on GitHub

With GitHub, you can create, edit, move and delete files in your repository or any repository you have write access to. You can also track the history of changes in a file line by line. For more information, see "Managing files on GitHub."

Part 3: Collaborating on GitHub

Any number of people can work together in repositories across GitHub. You can configure settings, create project boards, and manage your notifications to encourage effective collaboration.

1. Working with repositories

Creating a repository

A repository is like a folder for your project. You can have any number of public and private repositories in your user account. Repositories can contain folders and files, images, videos, spreadsheets, and data sets, as well as the revision history for all files in the repository. For more information, see "About repositories."

When you create a new repository, you should initialize the repository with a README file to let people know about your project. For more information, see "Creating a new repository."

Cloning a repository

You can clone an existing repository from GitHub to your local computer, making it easier to add or remove files, fix merge conflicts, or make complex commits. Cloning a repository pulls down a full copy of all the repository data that GitHub has at that point in time, including all versions of every file and folder for the project. For more information, see "Cloning a repository."

Forking a repository

A fork is a copy of a repository that you manage, where any changes you make will not affect the original repository unless you submit a pull request to the project owner. 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. For more information, see "Working with forks."

2. Importing your projects

If you have existing projects you'd like to move over to GitHub you can import projects using the GitHub Importer, the command line, or external migration tools. For more information, see "Importing source code to GitHub."

3. Managing collaborators and permissions

You can collaborate on your project with others using your repository's issues, pull requests, and project boards. You can invite other people to your repository as collaborators from the **Collaborators** tab in the repository settings. For more information, see "Inviting collaborators to a personal repository."

You are the owner of any repository you create in your user account and have full control of the repository. Collaborators have write access to your repository, limiting what they have permission to do. For more information, see "Permission levels for a user account repository."

4. Managing repository settings

As the owner of a repository you can configure several settings, including the repository's visibility, topics, and social media preview. For more information, see "Managing repository settings."

5. Setting up your project for healthy contributions

To encourage collaborators in your repository, you need a community that encourages people to use, contribute to, and evangelize your project. For more information, see "Building Welcoming Communities" in the Open Source Guides.

By adding files like contributing guidelines, a code of conduct, and a license to your repository you can create an environment where it's easier for collaborators to make meaningful, useful contributions. For more information, see "Setting up your project for healthy contributions."

6. Using GitHub Issues and project boards

You can use GitHub Issues to organize your work with issues and pull requests and manage your workflow with project boards. For more information, see "About issues" and "About project boards."

7. Managing notifications

Notifications provide updates about the activity on GitHub you've subscribed to or participated in. If you're no longer interested in a conversation, you can unsubscribe, unwatch, or customize the types of notifications you'll receive in the future. For more information, see "About notifications."

8. Working with GitHub Pages

You can use GitHub Pages to create and host a website directly from a repository on GitHub.com. For more information, see "About GitHub Pages."

9. Using GitHub Discussions

You can enable GitHub Discussions for your repository to help build a community around your project. Maintainers, contributors and visitors can use discussions to share announcements, ask and answer questions, and participate in conversations around goals. For more information, see "About discussions."

Part 4: Customizing and automating your work on GitHub

You can use tools from the GitHub Marketplace, the GitHub API, and existing GitHub features to customize and automate your work.

1. Using GitHub Marketplace

GitHub Marketplace contains integrations that add functionality and improve your workflow. You can discover, browse, and install free and paid tools, including GitHub Apps, OAuth Apps, and GitHub Actions, in GitHub Marketplace. For more information, see "About GitHub Marketplace."

2. Using the GitHub API

There are two versions of the GitHub API: the REST API and the GraphQL API. You can use the GitHub APIs to automate common tasks, back up your data, or create integrations that extend GitHub. For more information, see "About GitHub's APIs."

3. Building GitHub Actions

With GitHub Actions, you can automate and customize GitHub.com's development workflow on GitHub. You can create your own actions, and use and customize actions shared by the GitHub community. For more information, see "Learn GitHub Actions."

4. Publishing and managing GitHub Packages

GitHub Packages is a software package hosting service that allows you to host your software packages privately or publicly and use packages as dependencies in your projects. For more information, see "Introduction to GitHub Packages."

Part 5: Building securely on GitHub

GitHub has a variety of security features that help keep code and secrets secure in repositories. Some features are available for all repositories, while others are only available for public repositories and repositories with a GitHub Advanced Security license. For an overview of GitHub security features, see "GitHub security features."

1. Securing your repository

As a repository administrator, you can secure your repositories by configuring repository security settings. These include managing access to your repository, setting a security policy, and managing dependencies. For public repositories, and for private repositories owned by organizations where GitHub Advanced Security is enabled, you can also configure code and secret scanning to automatically identify vulnerabilities and ensure tokens and keys are not exposed.

For more information on steps you can take to secure your repositories, see "Securing your repository."

2. Managing your dependencies

A large part of building securely is maintaining your project's dependencies to ensure that all packages and applications you depend on are updated and secure. You can manage your repository's dependencies on GitHub by exploring the dependency graph for your repository, using Dependabot to automatically raise pull requests to keep your dependencies up-to-date, and receiving Dependabot alerts and security updates for vulnerable dependencies.

For more information, see "Securing your software supply chain."

Part 6: Participating in GitHub's community

There are many ways to participate in the GitHub community. You can contribute to open source projects, interact with people in the GitHub Community Support, or learn with GitHub Learning Lab.

1. Contributing to open source projects

Contributing to open source projects on GitHub can be a rewarding way to learn, teach, and build experience in just about any skill you can imagine. For more information, see "How to Contribute to Open Source" in the Open Source Guides.

You can find personalized recommendations for projects and good first issues based on your past contributions, stars, and other activities in Explore. For more information, see "Finding ways to contribute to open source on GitHub."

2. Interacting with GitHub Community Support

You can connect with developers around the world in GitHub Community Support to ask and answer questions, learn, and interact directly with GitHub staff.

3. Reading about GitHub on GitHub Docs

You can read documentation that reflects the features available to you on GitHub. For more information, see "About versions of GitHub Docs."

4. Learning with GitHub Learning Lab

You can learn new skills by completing fun, realistic projects in your very own GitHub repository with GitHub Learning Lab. Each course is a hands-on lesson created by the GitHub community and taught by the friendly Learning Lab bot.

For more information, see "Git and GitHub learning resources."

5. Supporting the open source community

GitHub Sponsors allows you to make a monthly recurring payment to a developer or organization who designs, creates, or maintains open source projects you depend on. For more information, see "About GitHub Sponsors."

6. Contacting GitHub Support

GitHub Support can help you troubleshoot issues you run into while using GitHub. For more information, see "About GitHub Support."

Task 3

Package ‘logr’

Title Creates Log Files
Version 1.2.9
Description Contains functions to help create log files. The
package aims to overcome the difficulty of the base R sink() command. The
log_print() function will print to both the console and the file log, without
interfering in other write operations.
License CC0
Encoding UTF-8

URL https://logr.r-sassy.org

BugReports https://github.com/dbosak01/logr/issues
Depends R (>= 3.4.0)
Suggests knitr, rmarkdown, testthat, tidylog, dplyr, covr
Imports withr, utils, this.path
VignetteBuilder knitr
RoxygenNote 7.1.2
NeedsCompilation no
Author David Bosak [aut, cre]
Maintainer David Bosak <dbosak01@gmail.com>
Repository CRAN
Date/Publication 2022-03-08 16:10:02 UTC

R topics documented:

logr Creates log files

Description

The logr package contains functions to easily create log files.

Details

The logr package helps create log files for R scripts. The package provides easy logging, without the complexity of other logging systems. It is designed for analysts who simply want a written log of the their program execution. The package is designed as a wrapper to the base R sink() function.

How to use

There are only three logr functions:

8. log_open

9. log_print

10. log_close

The log_open() function initiates the log. The log_print() function prints an object to the log. The log_close() function closes the log. In normal situations, a user would place the call to log_open at the top of the program, call log_print() as needed in the program body, and call log_close() once at the end of the program.

Logging may be controlled globally using the options "logr.on" and "logr.notes". Both options accept TRUE or FALSE values, and control log printing or log notes, respectively.

See function documentation for additional details.

log_close Close the log

Description

The log_close function closes the log file.

Usage

```
log_close()
```

Details

The log_close function terminates logging. As part of the termination process, the function prints any outstanding warnings to the log. Errors are printed at the point at which they occur. But warnings can be captured only at the end of the logging session. Therefore, any warning messages will only be printed at the bottom of the log.

The function also prints the log footer. The log footer contains a date-time stamp of when the log was closed.

Value

None

See Also

log_open to open the log, and log_print for printing to the log.

Examples

```
# Create temp file location
tmp <- file.path(tempdir(), "test.log")

# Open log
lf <- log_open(tmp)

# Send message to log
log_print("High Mileage Cars Subset")

# Perform operations
hmc <- subset(mtcars, mtcars$mpg > 20)
```

3. Print data to log log_print(hmc)

4. Close log

```
log_close()
```

4. View results

```
writeLines(readLines(lf))
```

log_code Log the current program code

Description

A function to send the program/script code to the currently opened log. The log must be opened first with log_open. Code will be prefixed with a right arrow (">") to differentiate it from standard logging lines. The log_code function may be called from anywhere within the program. Code will be inserted into the log at the point where it is called. The log_code function will log the code as it is saved on disk. It will not capture any unsaved changes in the editor. If the current program file cannot be found, the function will return FALSE and no code will be writtten

Usage

```
log_code()
```

Value

A TRUE or FALSE value to indicate success or failure of the function.

See Also

log_open to open the log, and log_close to close the log.

Examples

```
# Create temp file location
tmp <- file.path(tempdir(), "test.log")

# Open log
lf <- log_open(tmp)
```

7. Write code to the log log_code()

8. Send message to log

```
log_print("High Mileage Cars Subset")

# Perform operations
hmc <- subset(mtcars, mtcars$mpg > 20)
```

c. Print data to log

```
log_print(hmc)
```

d. Close log

```
log_close()
```

v. View results

```
writeLines(readLines(lf))
```

log_open Open a log

Description

A function to initialize the log file.

Usage

```
log_open(file_name = "", logdir = TRUE, show_notes = TRUE, autolog = NULL)
```

Arguments

file_name
: The name of the log file. If no path is specified, the working directory will be used. As of v1.2.7, the name and path of the program or script will be used as a default if the file_name parameter is not supplied.

logdir
: Send the log to a log directory named "log". If the log directory does not exist, the function will create it. Valid values are TRUE and FALSE. The default is TRUE.

show_notes
: If true, will write notes to the log. Valid values are TRUE and FALSE. Default is TRUE.

autolog
: Whether to turn on autolog functionality. Autolog automatically logs functions from the dplyr, tidyr, and sassy family of packages. To enable autolog, either set this parameter to TRUE or set the "logr.autolog" option to TRUE. A FALSE value on this parameter will override the global option. The global option will

Details

The log_open function initializes and opens the log file. This function must be called first, before any logging can occur. The function determines the log path, attaches event handlers, clears existing log files, and initiates a new log.

The file_name parameter may be a full path, a relative path, or a file name. An relative path or file name will be assumed to be relative to the current working directory. If the file_name does not have a ’.log’ extension, the log_open function will add it.

As of v1.2.7, if the file_name parameter is not supplied, the function will use the program/script name as the default log file name, and the program/script path as the default path.

If requested in the logdir parameter, the log_open function will write to a ’log’ subdirectory of the path specified in the file_name. If the ’log’ subdirectory does not exist, the function will create it.

The log file will be initialized with a header that shows the log file name, the current working directory, the current user, and a timestamp of when the log_open function was called.

All errors, the last warning, and any log_print output will be written to the log. The log file will exist in the location specified in the file_name parameter, and will normally have a '.log' extension.

If errors or warnings are generated, a second file will be written that contains only error and warning messages. This second file will have a '.msg' extension and will exist in the specified log directory. If the log is clean, the msg file will not be created. The purpose of the msg file is to give the user a visual indicator from the file system that an error or warning occurred. This indicator msg file is useful when running programs in batch.

To use logr, call log_open, and then make calls to log_print as needed to print variables or data frames to the log. The log_print function can be used in place of a standard print function. Anything printed with log_print will be printed to the log, and to the console if working interactively.

This package provides the functionality of sink, but in much more user-friendly way. Recom-mended usage is to call log_open at the top of the script, call log_print as needed to log interim state, and call log_close at the bottom of the script.
Logging may be controlled globally using the "logr.on" option. This option accepts a TRUE or FALSE value. If the option is set to FALSE, logr will print to the console, but not to the log. Example: options("logr.on" = TRUE)

Notes may be controlled globally using the "logr.notes" option. This option also accepts a TRUE or FALSE value, and determines whether or not to print notes in the log. The global option will override the show_notes parameter on the log_open function. Example: options("logr.notes" = FALSE)

Version v1.2.0 of the logr package introduced autolog. The autolog feature provides automatic log-ging for dplyr, tidyr, and the sassy family of packages. To use autolog, set the autolog parameter to TRUE, or set the global option logr.autolog to TRUE. To maintain backward compatibility with prior versions, autolog is disabled by default.

Value

The path of the log.

See Also

log_print for printing to the log (and console), and log_close to close the log.

Examples

```
# Create temp file location
tmp <- file.path(tempdir(), "test.log")

# Open log
lf <- log_open(tmp)

# Send message to log
log_print("High Mileage Cars Subset")

# Perform operations
hmc <- subset(mtcars, mtcars$mpg > 20)

# Print data to log log_print(hmc)

# Close log
log_close()
```

```
# View results
writeLines(readLines(lf))
```

log_path Get the path of the current log

Description

The log_path function gets the path to the currently opened log. This function may be useful when you want to manipulate the log in some way, and need the path. The function takes no parameters.

Usage

```
log_path()
```

Value

The full path to the currently opened log, or NULL if no log is open.

Examples

```
# Create temp file location
tmp <- file.path(tempdir(), "test.log")

# Open log log_open(tmp)

 # Get path
lf <- log_path()

# Close log log_close()

lf
```

log_print Print an object to the log

Description

The log_print function prints an object to the currently opened log.

Usage

```
log_print(x, ..., console = TRUE, blank_after = TRUE, msg = FALSE, hide_notes = FALSE)
put(x, ..., console = TRUE, blank_after = TRUE, msg = FALSE, hide_notes = FALSE)
sep(x, console = TRUE)
```

```
log_hook(x)
```

Arguments

x The object to print.

... Any parameters to pass to the print function.

console Whether or not to print to the console. Valid values are TRUE and FALSE. Default is TRUE.

blank_after Whether or not to print a blank line following the printed object. The blank line helps readability of the log. Valid values are TRUE and FALSE. Default is TRUE.

msg Whether to print the object to the msg log. This parameter is intended to be used internally. Value values are TRUE and FALSE. The default value is FALSE.

hide_notes If notes are on, this parameter gives you the option of not printing notes for a particular log entry. Default is FALSE, meaning notes will be displayed. Used internally.

Details

The log is initialized with log_open. Once the log is open, objects like variables and data frames can be printed to the log to monitor execution of your script. If working interactively, the function will print both to the log and to the console. The log_print function is useful when writing and debugging batch scripts, and in situations where some record of a scripts' execution is required.

If requested in the log_open function, log_print will print a note after each call. The note will contain a date-time stamp and elapsed time since the last call to log_print. When printing a data frame, the log_print function will also print the number and rows and column in the data frame. These counts may also be useful in debugging.

Notes may be turned off either by setting the show_notes parameter on log_open to FALSE, or by setting the global option "logr.notes" to FALSE.

The put function is a shorthand alias for log_print. You can use put anywhere you would use log_print. The functionality is identical.

The sep function is also a shorthand alias for log_print, except it will print a separator before and after the printed text. This function is intended for documentation purposes, and you can use it to help organize your log into sections.

The log_hook function is for other packages that wish to integrate with logr. The function prints to the log only if autolog is enabled. It will not print to the console.

Value

The object, invisibly

See Also

log_open to open the log, and log_close to close the log.

Examples

```
# Create temp file location
tmp <- file.path(tempdir(), "test.log")

# Open log
lf <- log_open(tmp)

# Send message to log
log_print("High Mileage Cars Subset")

# Perform operations
hmc <- subset(mtcars, mtcars$mpg > 20)

# Print data to log log_print(hmc)

# Close log
log_close()

# View results
writeLines(readLines(lf))
```

log_status Get the status of the log

Description

The log_status function gets the status of the log. Possible status values are ’on’, ’off’, ’open’, or ’closed’. The function takes no parameters.

Usage

```
log_status()
```

Value

The status of the log as a character string.

Examples

```
# Check status before the log is opened
log_status()
# [1] "closed"

# Create temp file location
tmp <- file.path(tempdir(), "test.log")

# Open log
```

```
lf <- log_open(t

# Check status after log is opened
log_status()
# [1] "open"

# Close log log_close()
```

Task 4

This document is an in-depth review of the `git branch` command and a discussion of the overall Git branching model. Branching is a feature available in most modern version control systems. Branching in other VCS's can be an expensive operation in both time and disk space. In Git, branches are a part of your everyday development process. Git branches are effectively a pointer to a snapshot of your changes. When you want to add a new feature or fix a bug—no matter how big or how small—you spawn a new branch to encapsulate your changes. This makes it harder for unstable code to get merged into the main code base, and it gives you the chance to clean up your future's history before merging it into the main branch.

The diagram above visualizes a repository with two isolated lines of development, one for a little feature, and one for a longer-running feature. By developing them in branches, it’s not only possible to work on both of them in parallel, but it also keeps the `main` branch free from questionable code.

The implementation behind Git branches is much more lightweight than other version control system models. Instead of copying files from directory to directory, Git stores a branch as a reference to a commit. In this sense, a branch represents the tip of a series of commits—it's not a container for commits. The history for a branch is extrapolated through the commit relationships.

As you read, remember that Git branches aren't like SVN branches. Whereas SVN branches are only used to capture the occasional largescale development effort, Git branches are an integral part of your everyday workflow. The following content will expand on the internal Git branching architecture.

How it works

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.

The `git branch` command lets you create, list, rename, and delete branches. It doesn't let you switch between branches or put a forked history back together again. For this reason, `git branch` is tightly integrated with the git checkout and git merge commands.

Common Options

```
git branch
```

List all of the branches in your repository. This is synonymous with `git branch --list`.

```
git branch <branch>
```

Create a new branch called <`branch`>. This does *not* check out the new branch.

```
git branch -d <branch>
```

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

```
git branch -D <branch>
```

Force delete the specified branch, even if it has unmerged changes. This is the command to use if you want to permanently throw away all of the commits associated with a particular line of development.

```
git branch -m <branch>
```

Rename the current branch to <`branch`>.

```
git branch -a
```

List all remote branches.

Creating Branches

It's important to understand that branches are just pointers to commits. When you create a branch, all Git needs to do is create a new pointer, it doesn’t change the repository in any other way. If you start with a repository that looks like this:

Then, you create a branch using the following command:

```
git branch crazy-experiment
```

The repository history remains unchanged. All you get is a new pointer to the current commit:

Note that this only *creates* the new branch. To start adding commits to it, you need to select it with `git checkout`, and then use the standard `git add` and `git commit` commands.

Creating remote branches

So far these examples have all demonstrated local branch operations. The `git branch` command also works on remote branches. In order to operate on remote branches, a remote repo must first be configured and added to the local repo config.

```
$ git remote add new-remote-
repo https://bitbucket.com/user/repo.git
# Add remote repo to local repo config
$ git push <new-remote-repo> crazy-experiment~
# pushes the crazy-experiment branch to new-remote-repo
```

This command will push a copy of the local branch `crazyexperiment` to the remote repo `<remote>`.

Deleting Branches

Once you've finished working on a branch and have merged it into the main code base, you're free to delete the branch without losing any history:

```
git branch -d crazy-experiment
```

However, if the branch hasn't been merged, the above command will output an error message:

```
error: The branch 'crazy-experiment' is not fully merged. If
you are sure you want to delete it, run 'git branch -D
crazy-experiment'.
```

This protects you from losing access to that entire line of development. If you really want to delete the branch (e.g., it's a failed experiment), you can use the capital `-D` flag:

```
git branch -D crazy-experiment
```

This deletes the branch regardless of its status and without warnings, so use it judiciously.

The previous commands will delete a local copy of a branch. The branch may still exist in remote repos. To delete a remote branch execute the following.

```
git push origin --delete crazy-experiment
```

Or

```
git push origin :crazy-experiment
```

This will push a delete signal to the remote origin repository that triggers a delete of the remote `crazy-experiment` branch.

Summary

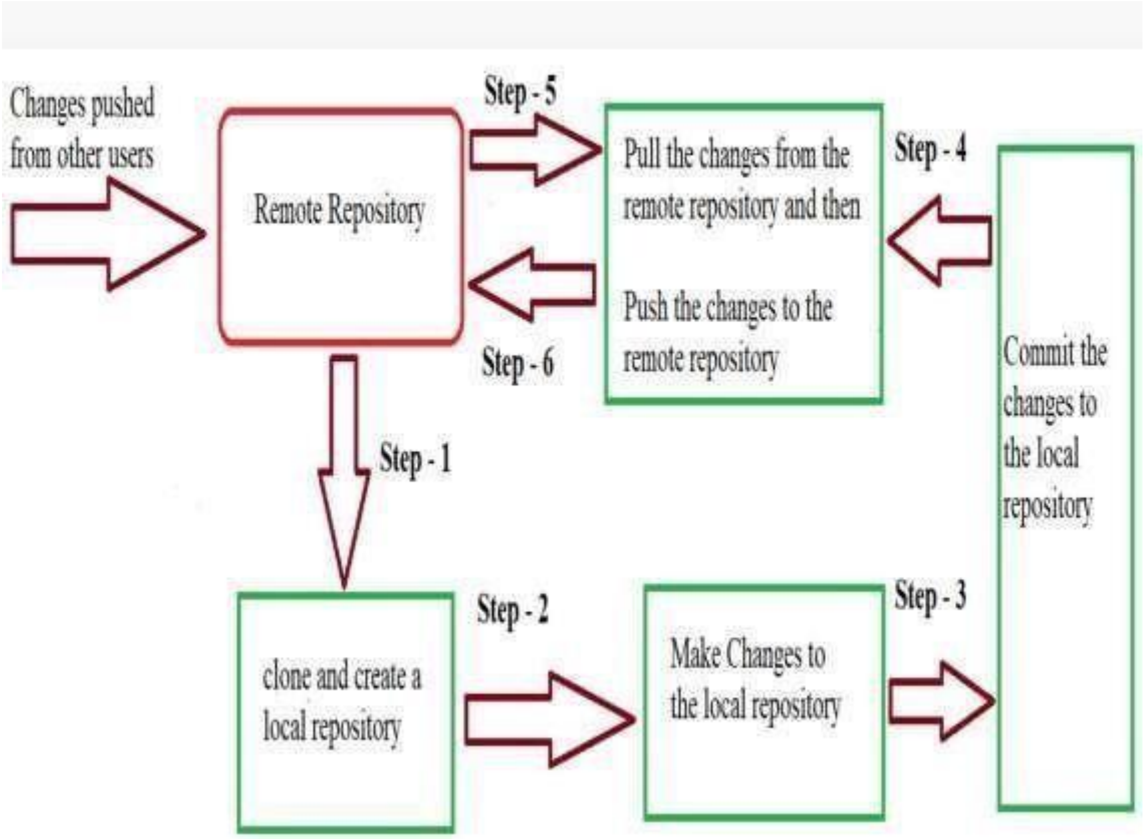
In this document we discussed Git's branching behavior and the `git branch` command. The `git branch` commands primary functions are to create, list, rename and delete branches. To operate further on the resulting branches the command is commonly used with other commands like `git checkout`. Learn more about `git checkout` branch operations; such as switching branches and merging branches, on the git checkout page.

Compared to other VCSs, Git's branch operations are inexpensive and frequently used. This flexibility enables powerful Git workflow customization. For more info on Git workflows visit our extended workflow discussion pages: The Feature Branch Workflow, GitFlow Workflow, and Forking Workflow.

Task 5

Git – Life Cycle

Git is used in our day-to-day work, we use git for keeping a track of our files, working in a collaboration with our team, to go back to our previous code versions if we face some error. Git helps us in many ways. Let us look at the Life Cycle that git has and understand more about its life cycle. Let us see some of the basic steps that we follow while working with Git –


Changes pushed from other users
Remote Repository
Step - 5
Pull the changes from the remote repository and then
Push the changes to the remote repository
Step - 6
Step - 4
Commit the changes to the local repository
Step - 1
Step - 2
clone and create a local repository
Make Changes to the local repository
Step - 3

- ***In Step – 1***, *We first clone any of the code residing in the remote reposit ory to make our won local repository.*
- ***In Step-2*** *we edit the files that we have cloned in our local repository an d make the necessary changes in it.*
- ***In Step-3*** *we commit our changes by first adding them to our staging ar ea and committing them with a commit message.*
- ***In Step – 4 and Step-5*** *we first check whether there are any of the chang es done in the remote repository by some other users and we first pull that c hanges.*
- *If there are no changes we directly proceed with* ***Step – 6*** *in which we p ush our changes to the remote repository and we are done with our work.*

When a directory is made a git repository, there are mainly 3 states which make the essence of Git Version Control System. The three states are –

- Working Directory
- Staging Area
- Git Directory

Let us understand in detail about each state.

1. Working Directory

Whenever we want to initialize our local project directory to make it a git repository, we use the ***git init*** command. After this command, git becomes aware of the files in the project although it doesn’t track the files yet. The files are further tracked in the staging area.

git init

2. Staging Area

Now, to track the different versions of our files we use the command ***git add***. We can term a staging area as a place where different versions of our files arestored. ***git add*** command copies the version of your file from your working directory to the staging area. We can, however, choose which files we need toadd to the staging area because in our working directory there are some files
that we don’t want to get tracked, examples include node modules, env files, temporary files, etc. Indexing in Git is the one that helps Git in understandingwhich files need to be added or sent. You can find your staging area in the
.git folder inside the ***index*** file.
// to specify which file to add to the staging area gitadd <filename>

// to add all files of the working directory to the staging area gitadd .

3. Git Directory

Now since we have all the files that are to be tracked and are ready in the staging area, we are ready to commit our files using the ***git commit*** command. Commit helps us in keeping the track of the metadata of the files in our staging area. We specify every commit with a message which tells what the commit is about. Git preserves the information or the metadata of the files thatwere committed in a Git Directory which helps Git in tracking files and basically it preserves the photocopy of the committed files. Commit also storesthe name of the author who did the commit, files that are committed, and the date at which they are committed along with the commit message. *git commit -m <Commit Message>*

Subject Name: **Source Code Management**
Subject Code: **CS181**
Cluster: **Zeta**
Department: **DCSE**

Submitted By: Nawed Alam
211099208
0G27

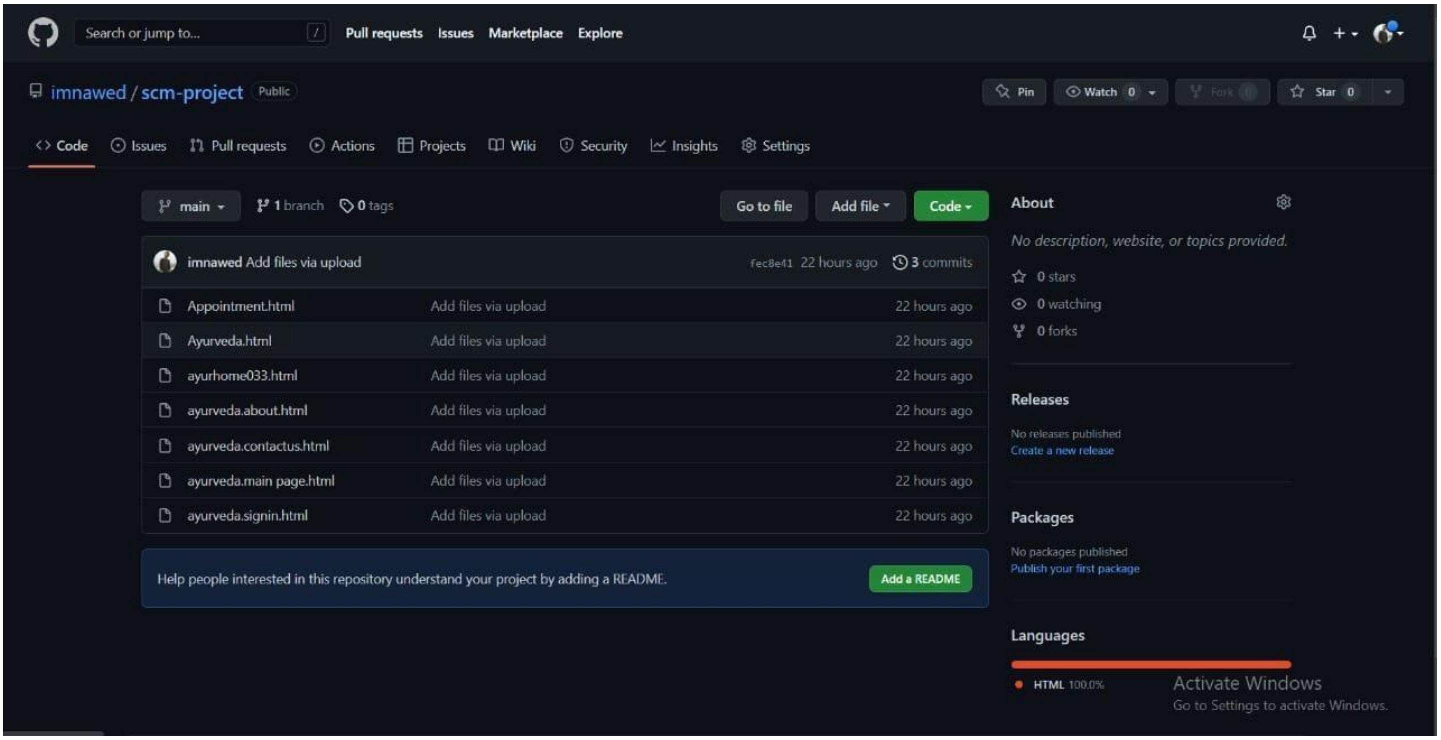
Submitted To:
Dr. Anuj Jain

Experiment 1: Add collaborators on GitHub

Ask for the username of the person you're inviting as a collaborator. If they don't have ausername yet, they can sign up for GitHub For more information, see "[Signing up for a new GitHub account](#)".

- On GitHub.com, navigate to the main page of the repository.

- **\Under your repository name, click Settings.**



imnawed / scm-project Public

imnawed Add files via upload fec8e41 22 hours ago 3 commits

Appointment.html	Add files via upload	22 hours ago
Ayurveda.html	Add files via upload	22 hours ago
ayurhome033.html	Add files via upload	22 hours ago
ayurveda.about.html	Add files via upload	22 hours ago
ayurveda.contactus.html	Add files via upload	22 hours ago
ayurveda.main page.html	Add files via upload	22 hours ago
ayurveda.signin.html	Add files via upload	22 hours ago

Help people interested in this repository understand your project by adding a README.

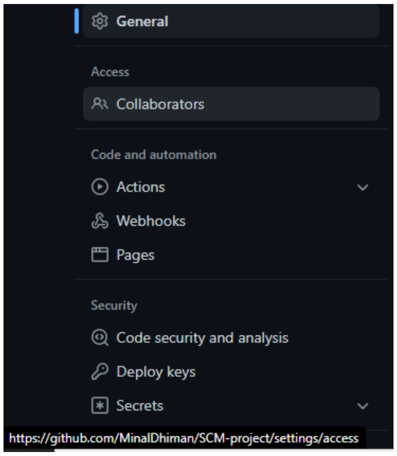
No description, website, or topics provided.

No releases published
Create a new release

No packages published
Publish your first package

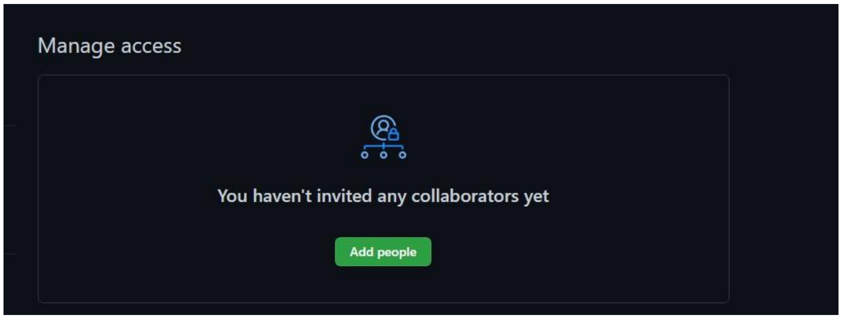
Activate Windows
Go to Settings to activate Windows.

- In the "Access" section of the sidebar, click **Collaborators**.



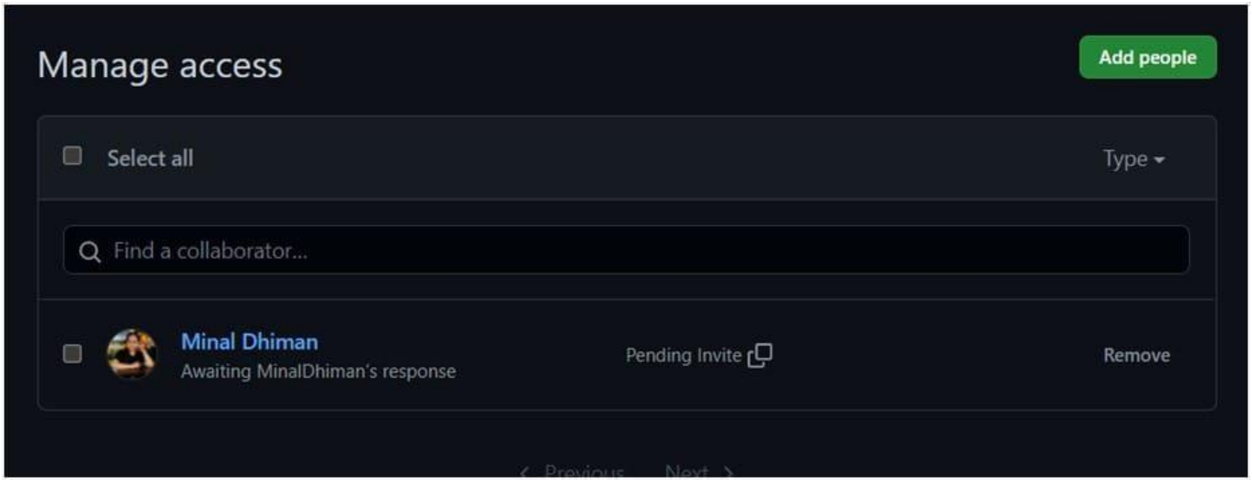
https://github.com/MinalDhiman/SCM-project/settings/access

- Click **Add people**.


Manage access

You haven't invited any collaborators yet

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



- Click **Add** .

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

Experiment 2: Fork and Commit

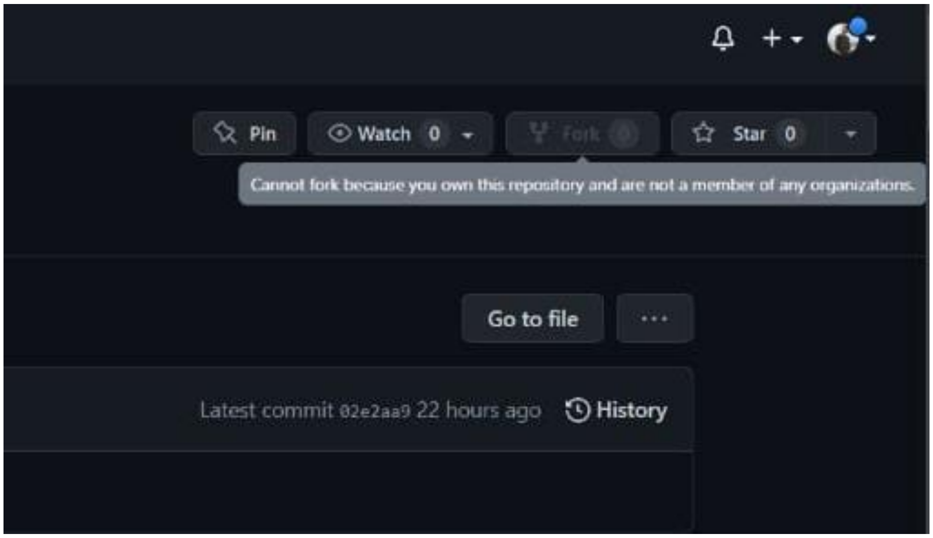
Forking a GitHub Repository

The first step is to fork the GitHub repository with which you'd like to work. For example, if you were interested in helping contribute content to the Open vSwitch web site, which is itselfhosted as a GitHub repository, you would first fork it. Forking it is basically making a copy ofthe repository, but with a link back to the original.

Forking a repository is *really* straightforward:

1. Make sure you're logged into GitHub with your account.
2. Find the GitHub repository with which you'd like to work.
3. Click the Fork button on the upper right-hand side of the repository's page.

That's it—you now have a copy of the original repository in your GitHub account.



Cannot fork because you own this repository and are not a member of any organizations.

Latest commit 02e2aa9 22 hours ago

Commit

`git commit` creates a commit, which is like a snapshot of your repository. These commitsare 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.

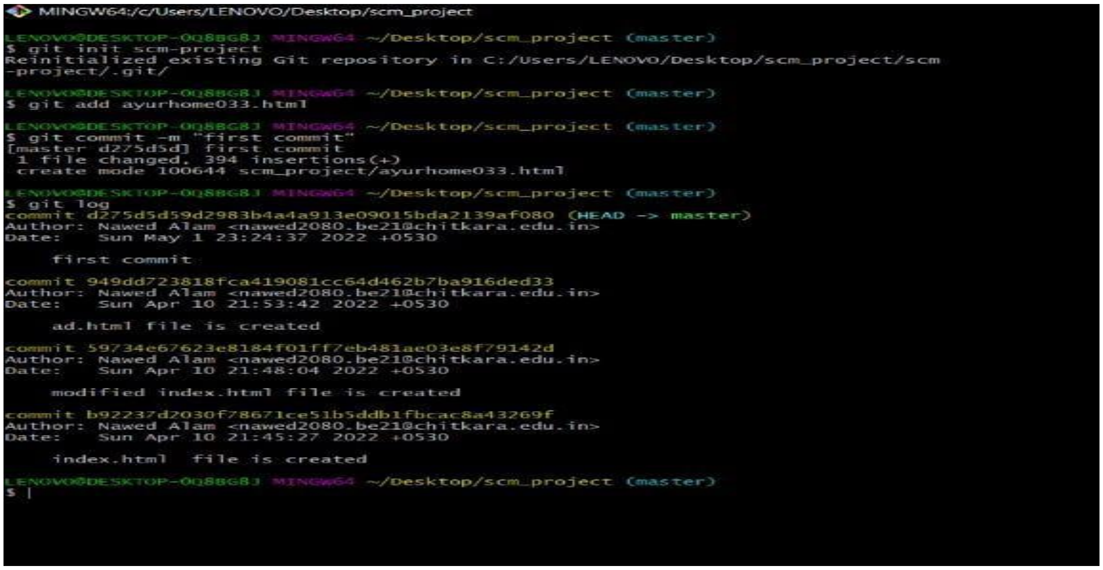
How does it works.......

Commits are the building blocks of "save points" within Git's version control.

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 getout, press esc, then `:w`, and then Enter. :wink:)
- `git commit -m "descriptive commit message"`: This starts the commit process, andallows 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 thatyou've committed before).
- `git commit --amend`: Replaces the most recent commit with a new commit.

Examples of Git Commit



```
LENOVO@DESKTOP-0Q8BG8J MINGW64 ~/Desktop/scm_project (master)
$ git init scm-project
Reinitialized existing Git repository in C:/Users/LENOVO/Desktop/scm_project/scm
-project/.git/

LENOVO@DESKTOP-0Q8BG8J MINGW64 ~/Desktop/scm_project (master)
$ git add ayurhome033.html

LENOVO@DESKTOP-0Q8BG8J MINGW64 ~/Desktop/scm_project (master)
$ git commit -m "first commit"
[master d275d5d] first commit
 1 file changed, 394 insertions(+)
 create mode 100644 scm_project/ayurhome033.html

LENOVO@DESKTOP-0Q8BG8J MINGW64 ~/Desktop/scm_project (master)
$ git log
commit d275d5d59d2983b4a4a913e09015bda2139af080 (HEAD -> master)
Author: Nawed Alam <nawed2080.be21@chitkara.edu.in>
Date:   Sun May 1 23:24:37 2022 +0530

    first commit

commit 949dd723818fca419081cc64d462b7ba916ded33
Author: Nawed Alam <nawed2080.be21@chitkara.edu.in>
Date:   Sun Apr 10 21:53:42 2022 +0530

    ad.html file is created

commit 59734e67623e8184f01ff7eb481ae03e8f79142d
Author: Nawed Alam <nawed2080.be21@chitkara.edu.in>
Date:   Sun Apr 10 21:48:04 2022 +0530

    modified index.html file is created

commit b92237d2030f78671ce51b5ddb1fbcac8a43269f
Author: Nawed Alam <nawed2080.be21@chitkara.edu.in>
Date:   Sun Apr 10 21:45:27 2022 +0530

    index.html  file is created

LENOVO@DESKTOP-0Q8BG8J MINGW64 ~/Desktop/scm_project (master)
$ |
```

Experiment 3: Merge and Resolve conflicts created due to own activity and collaborators

.

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.

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

Adding Collaborator as we have done earlier.

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

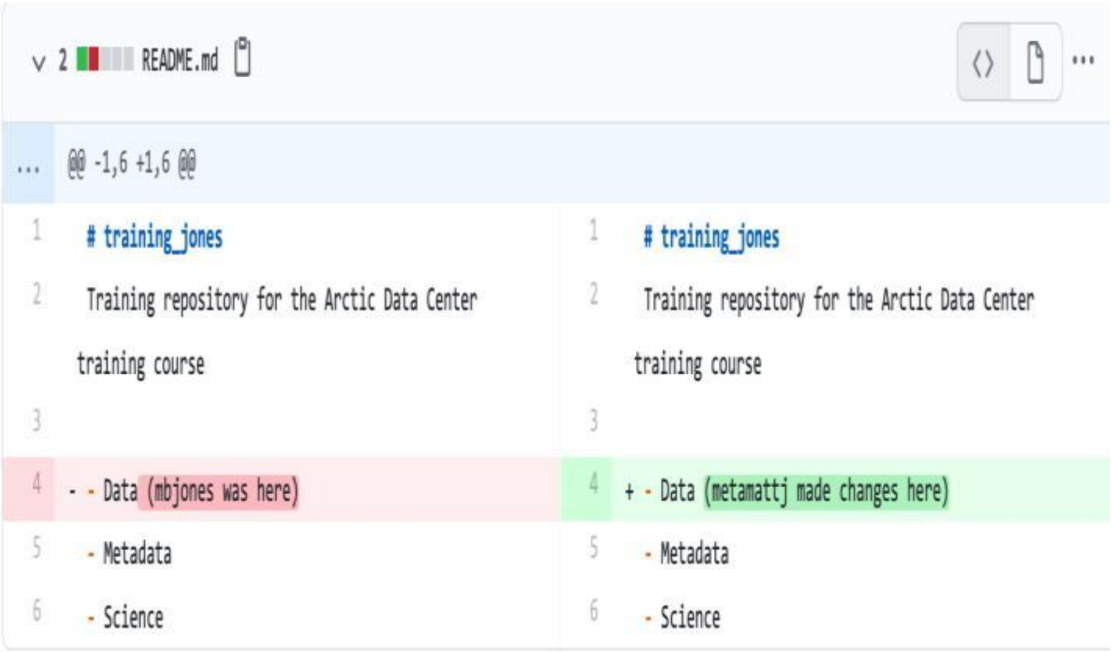
Now that the instructors have demonstrated this conflict-free process, break into pairs and try the same with your partner. Start by designating one person as the Owner and one as the Collborator , and then repeat the steps described above:

- 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 youto overwrite one person's changes to a file with another's changes to the same file if they were based on the same version.



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

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

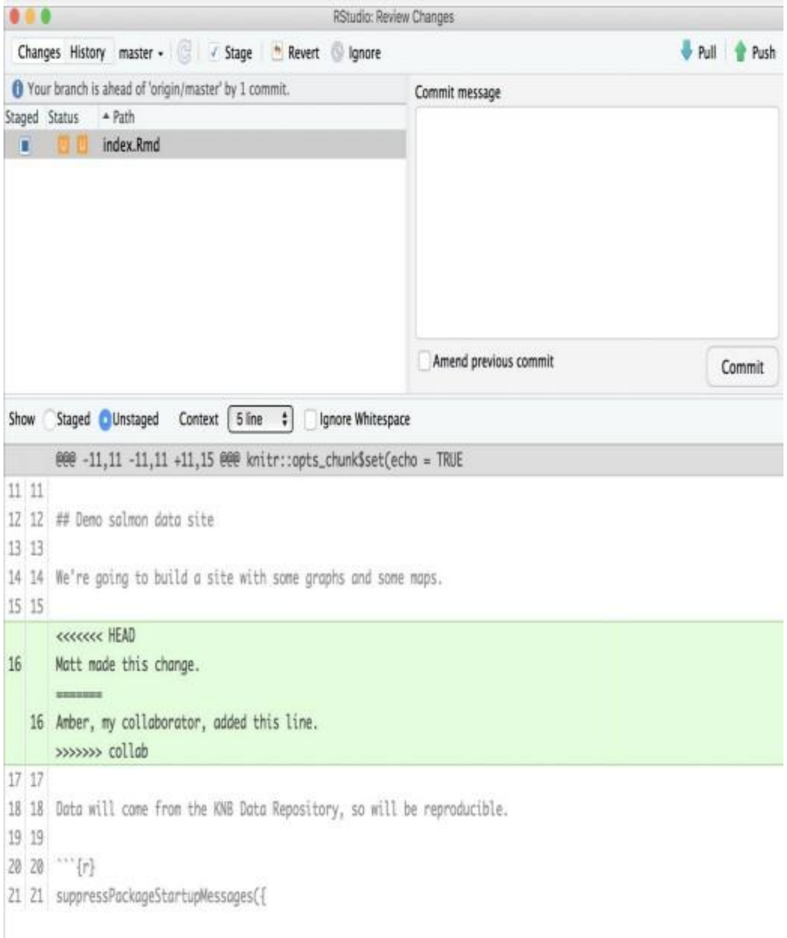
The simplest way to resolve a conflict, given that you know whose version of the file you want to keep, is to use the commandline git program to tell git to use either your changes (the person doing the merge), or their changes (the other collaborator).

- 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 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 youhelp to resolve the conflict. It delimits these blocks with a series of less than and greater than signs, so they are easy to find:



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

~~~
@@@ -11,11 -11,11 +11,15 @@@ knitr::opts_chunk$set(echo = TRUE

## Demo salmon data site

We're going to build a site with some graphs and some maps.

<<<<<<< HEAD
Matt made this change.
=======
Amber, my collaborator, added this line.
>>>>>>> collab

Data will come from the KNB Data Repository, so will be reproducible.

```{r}
suppressPackageStartupMessages({
~~~

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 <<<<<<>>>>>>. 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 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.

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

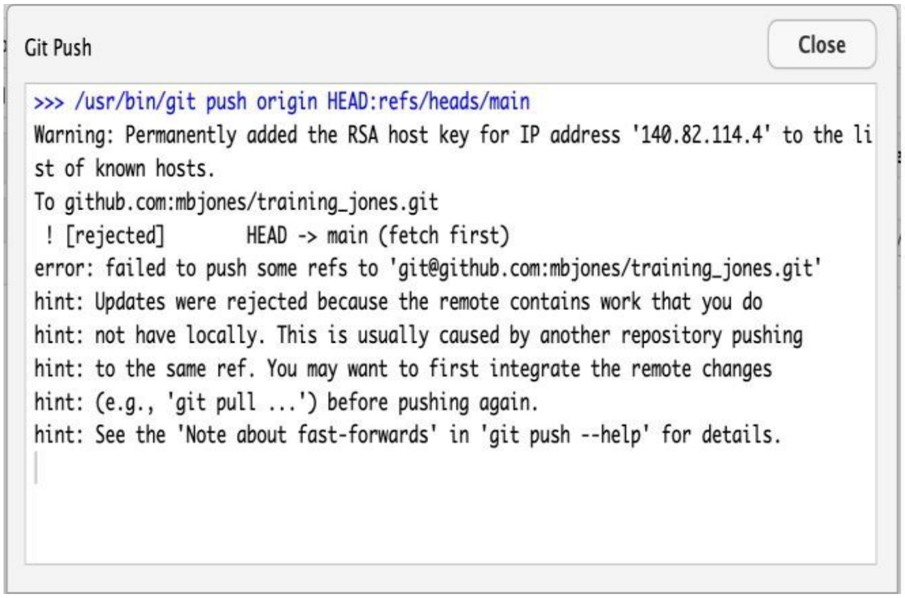
Now the collaborator also makes changes to the same (line 4) of the README.md file in their RStudio copy of the project, adding their name to the line. They then commit. At this point, both the owner and collaborator have committed changes based on their shared version of the README.md file, but neither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

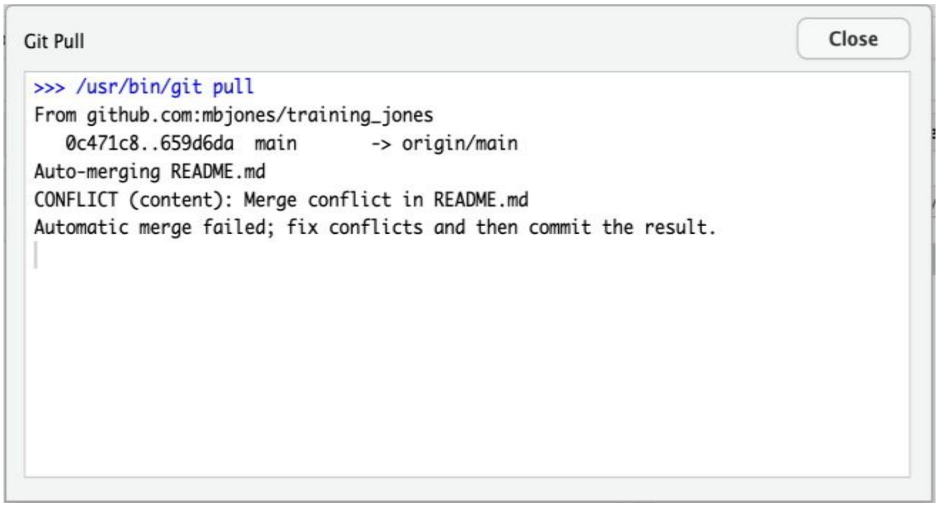
At this point; the owner tries to push their change to the repository, which triggers an error from GitHub. While the error message is long, it basically tells you everything needed (that the owner's repository doesn't reflect the changes on GitHub, and that they need to pull before they can push).



```
>>> /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 li
st 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.
```

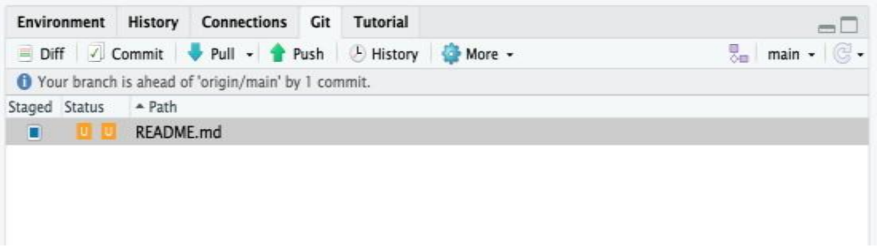
Owner pulls from GitHub to get Collaborator changes

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.



```
>>> /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.

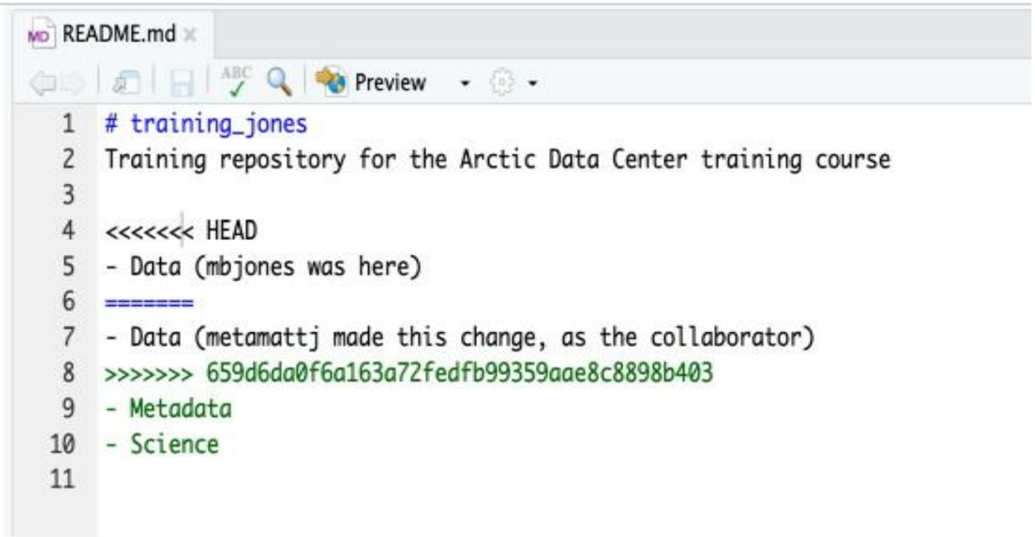


Your branch is ahead of 'origin/main' by 1 commit.

Staged	Status	Path
■	U U	README.md

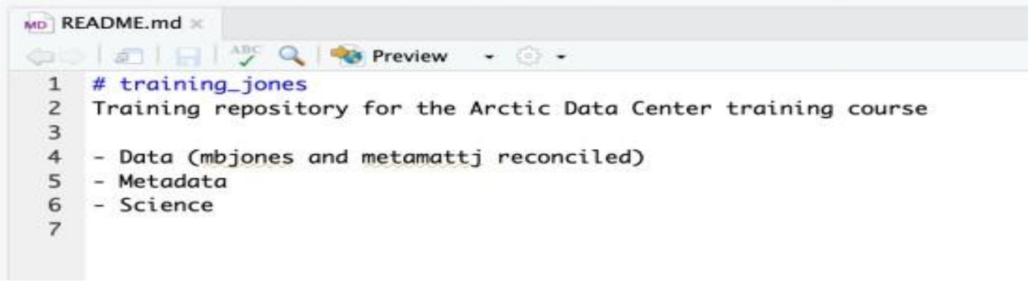
Owner edits the file to resolve the conflict

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 <<<<<<>>>>>>. 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 <<<<<<>>>>>>.



```
# training_jones
Training repository for the Arctic Data Center training course

<<<<<<< HEAD
- Data (mbjones was here)
=======
- Data (metamattj made this change, as the collaborator)
>>>>>>> 659d6da0f6a163a72fedfb99359aae8c8898b403
- Metadata
- Science

```

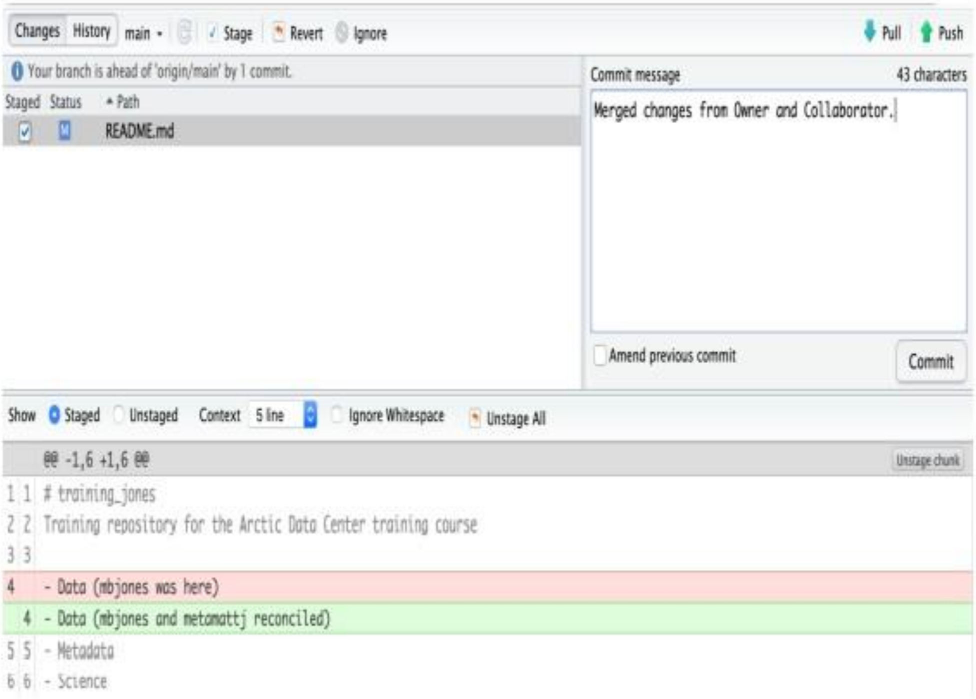
Of course, for scripts and programs, resolving the changes means more than just merging the text – whoever is doing the merging should make sure that the code runs properly and none of the logic of the program has been broken.



```
# training_jones
Training repository for the Arctic Data Center training course

- Data (mbjones and metamattj reconciled)
- Metadata
- Science

```

3.3.1.8 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.



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

Staged	Status	Path
☑	M	README.md

Commit message: 43 characters

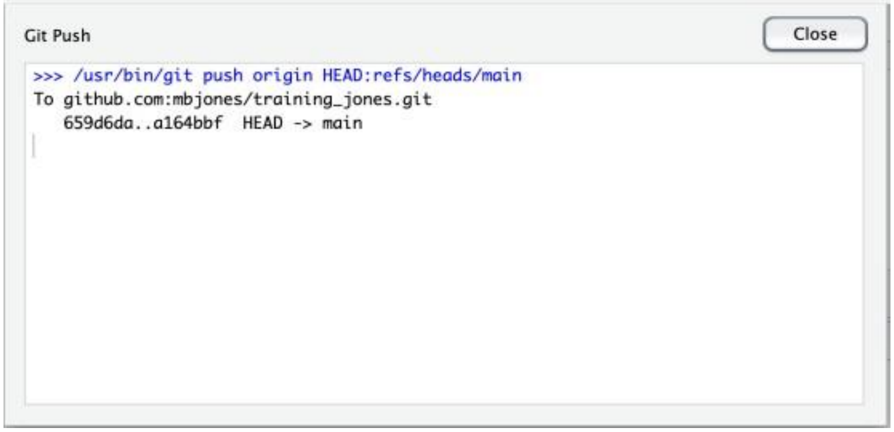
Merged changes from Owner and Collaborator.

Show ● Staged ○ Unstaged

```
@@ -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 (mbjones and metamattj reconciled)
5 5 - Metadata
6 6 - Science
```

3.3.1.9 Owner pushes the resolved changes to GitHub

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



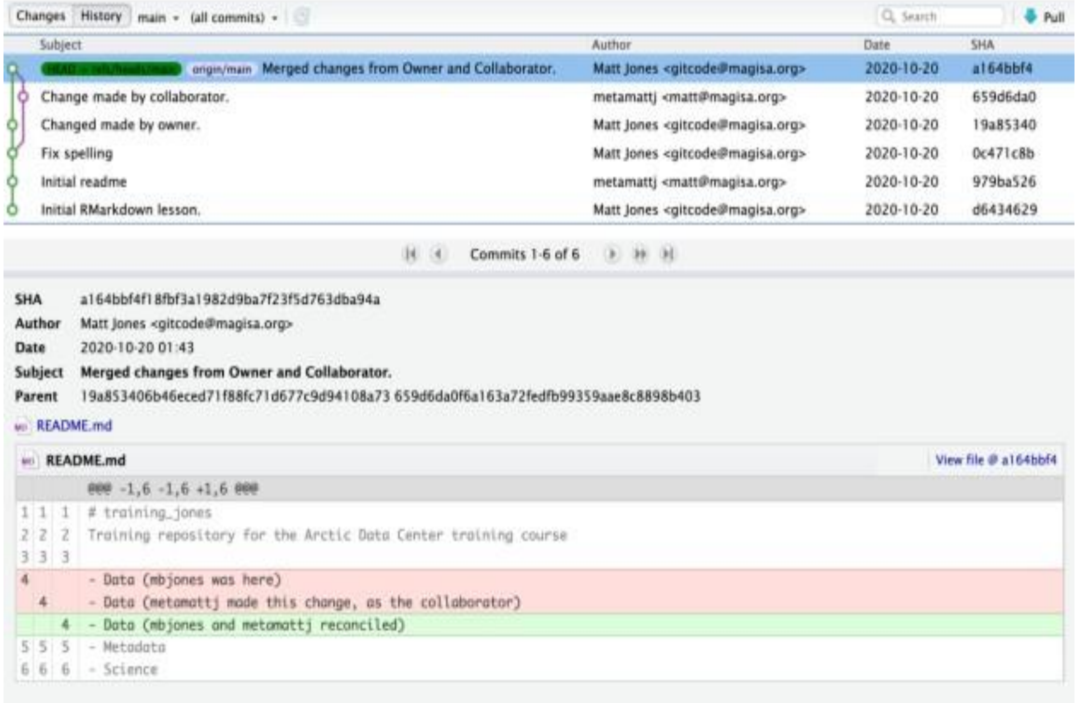
```
>>> /usr/bin/git push origin HEAD:refs/heads/main
To github.com:mbjones/training_jones.git
   659d6da..a164bbf  HEAD -> main
```

3.3.1.1 Collaborator pulls the resolved changes from GitHub

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

3.3.1.1 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.



Subject	Author	Date	SHA
HEAD -> refs/heads/main 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 a164bbf4f18fbf3a1982d9ba7f23f5d763dba94a
Author Matt Jones <gitcode@magisa.org>
Date 2020-10-20 01:43
Subject **Merged changes from Owner and Collaborator.**
Parent 19a853406b46eced71f88fc71d677c9d94108a73 659d6da0f6a163a72fedfb99359aae8c8898b403

README.md

README.md View file @ a164bbf4

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

3. 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
```

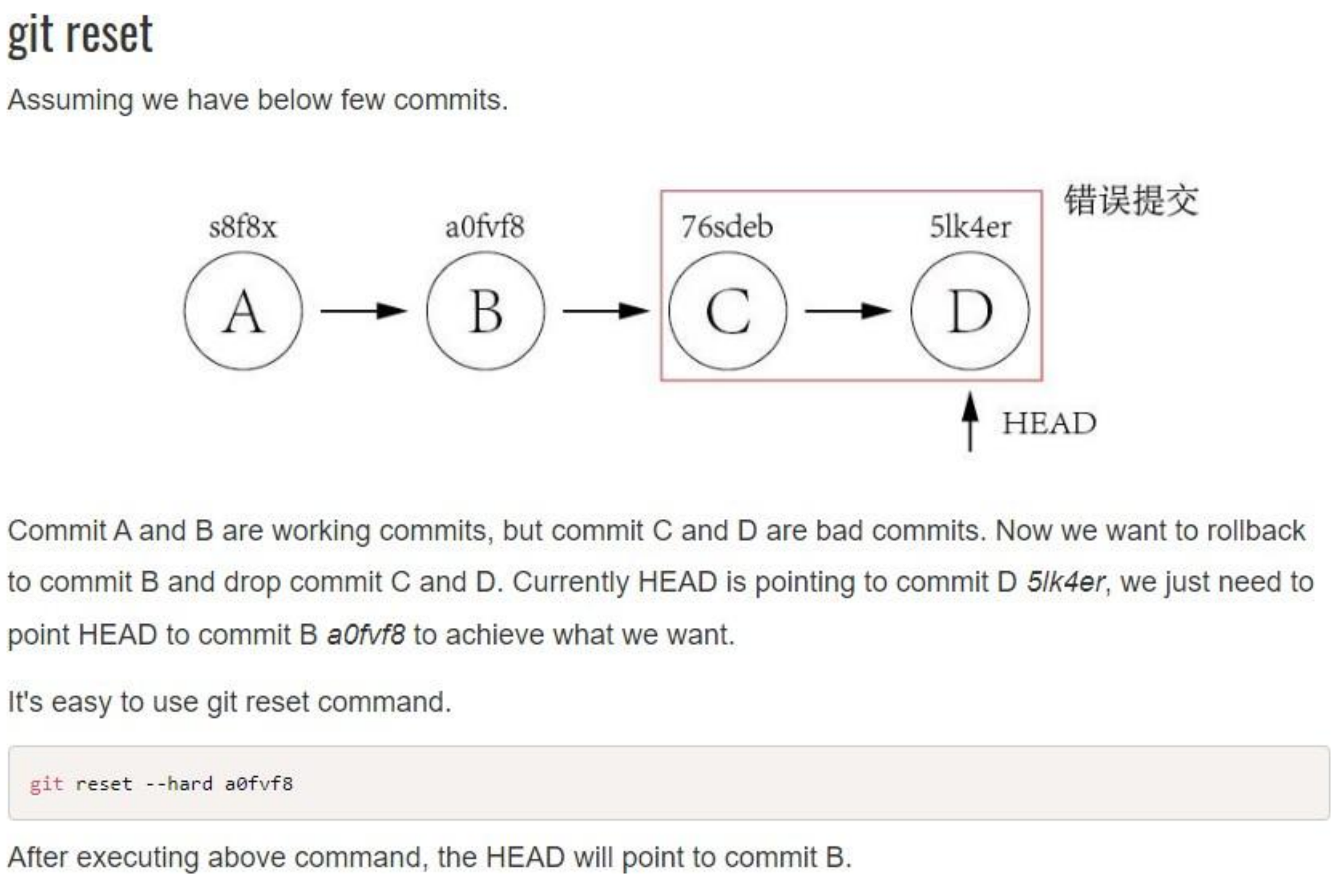
Always start your working sessions with a pull to get any outstanding changes, then start doing your editing and work. Stage your changes, but before you commit, Pull again to see if any new changes have arrived. If so, they should merge in easily if you are working in different parts of the program. 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 4. Reset and Revert

When maintaining code using version control systems such as **git**, it is unavoidable that we need to rollback some wrong commits either due to bugs or temp code revert. In this case, rookie developers would be very nervous because they may get lost on what they should do to rollback their changes without affecting others, but to veteran developers, this is their routine work and they can show you different ways of doing that.

In this post, we will introduce two major ones used frequently by developers.

- git reset
- git revert



git reset

Assuming we have below few commits.

Commit A and B are working commits, but commit C and D are bad commits. Now we want to rollback to commit B and drop commit C and D. Currently HEAD is pointing to commit D *5lk4er*, we just need to point HEAD to commit B *a0fvf8* to achieve what we want.

It's easy to use git reset command.

```
git reset --hard a0fvf8
```

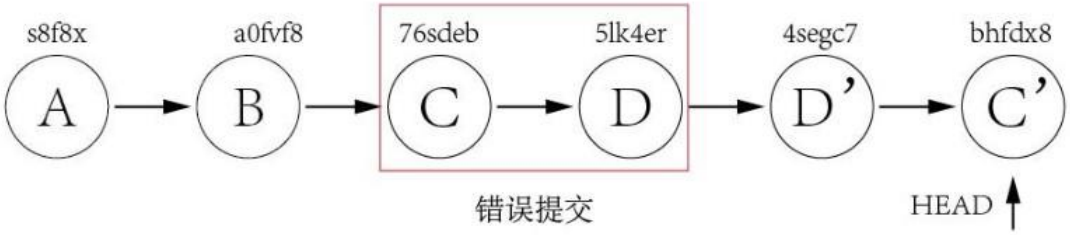
After executing above command, the HEAD will point to commit B.

git revert

For the example of git reset above, what we need to do is just reverting commit D and then reverting commit C.

```
git revert 5lk4er
git revert 76sdeb
```

Now it creates two new commit D' and C',

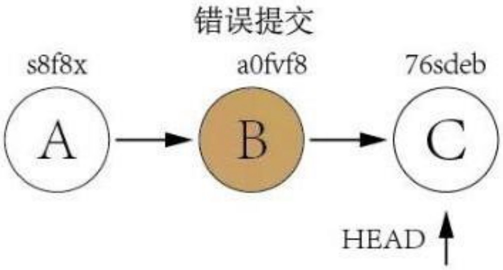


In above example, we have only two commits to revert, so we can revert one by one. But what if there are lots of commits to revert? We can revert a range indeed.

```
git revert OLDER_COMMIT^..NEWER_COMMIT
```

This method would not have the disadvantage of **git reset**, it would point HEAD to newly created reverting commit and it is ok to directly push the changes to remote without using the **-f** option.

Now let's take a look at a more difficult example. Assuming we have three commits but the bad commit is the second commit.



It's not a good idea to use **git reset** to rollback the commit B since we need to keep commit C as it is a good commit. Now we can revert commit C and B and then use **cherry-pick** to commit C again.

A Project

report on

“AYURVEDA-The Pride of India”

with

Source Code Management

(CS181)

Submitted
by

Team Member 1	**Minal Dhiman**	**2110992101**
Team Member 2	**Nawed Alam**	**2110992080**
Team Member 3	**Chirag**	**2110992082**
Team Member 4	**Aryan Shukla**	**2110992063**

Department of Computer Science & Engineering

Chitkara University Institute of Engineering and Technology, Punjab

Jan- June
(2021-22)

CHITKARA
UNIVERSITY

Institute/School Name	**Chitkara University Institute of Engineering andTechnology**		
Department Name	**Department of Computer Science & Engineering**		
Programme Name	**Bachelor of Engineering (B.E.), ComputerScience & Engineering**		
Course Name	**Source CodeManagement**	Session	**2021-22**
Course Code	**CS181**	Semester/Batch	**2nd/2021**
Vertical Name	**Zeta**	Group No	**G27**
Course Coordinator	**Dr. Neeraj Singla**		
Faculty Name	**Dr. SachendraChouhan**		

Submission

Name:
Signature:

Date:

Table of Content

1. Version control with Git

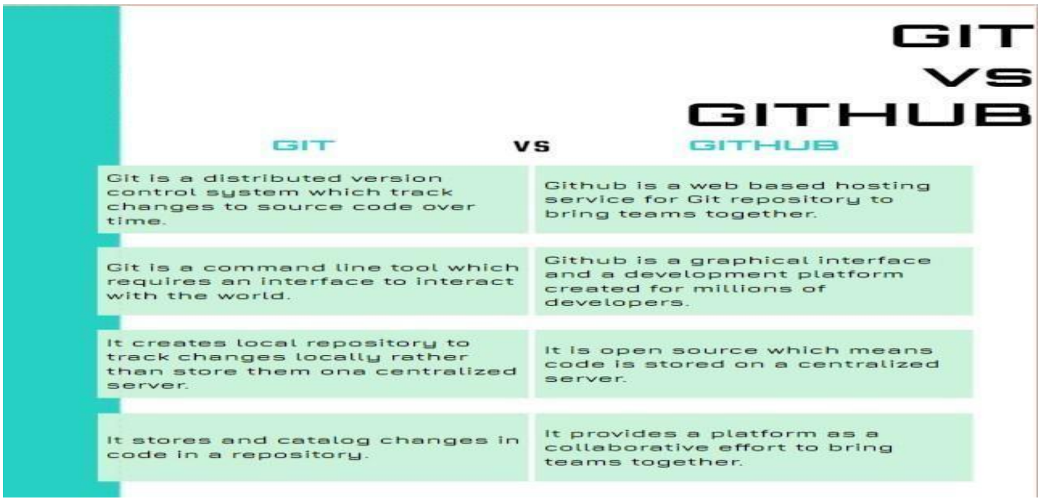
What is GIT and why is it used?

Git is a version control system that is widely used in the programming world. It is used for tracking changes in the source code during software development. It wasdeveloped in 2005 by Linus Torvalds, the creator of the Linux operating system kernel.
Git is a speedy and efficient distributed VCS tool that can handle projects of anysize, from small to very large ones. Git provides cheap local branching, convenient staging areas, and multiple workflows. It is free, open-source software that lowers the cost because developers can use Git without paying money. It provides support for non-linear development. Git enables multiple developers or teams to work separately without having an impact on the work ofothers.
Git is an example of a distributed version control system (DVCS) (henceDistributed Version Control System).

What is GITHUB?

It is the world's largest open-source software developer community platformwhere the users upload their projects using the software Git.

What is the difference between GIT and GITHUB?



GIT VS GITHUB

GIT	GITHUB
Git is a distributed version control system which track changes to source code over time.	Github is a web based hosting service for Git repository to bring teams together.
Git is a command line tool which requires an interface to interact with the world.	Github is a graphical interface and a development platform created for millions of developers.
It creates local repository to track changes locally rather than store them ona centralized server.	It is open source which means code is stored on a centralized server.
It stores and catalog changes in code in a repository.	It provides a platform as a collaborative effort to bring teams together.

What is Repository?

A repository is a directory or storage space where your projects can live. Sometimes GitHub users shorten this to "repo." It can be local to a folder onyour computer, or it can be a storage space on GitHub or another online host.
You can keep code files, text files, image files, you name it, inside a repository.

What is Version Control System (VCS)?

A version control system is a tool that helps you manage "versions" of your code or changes to your code while working with a team over remote distances.Version control keeps track of every modification in a special kind of database that is accessible to the version control software. Version control software (VCS) helps you revert back to an older version just in case a bug or issue is introduced to the system or fixing a mistake without disrupting the work of other team members.

Types of VCS

1. Local Version Control System
2. Centralized Version Control System
3. Distributed Version Control System

I. **Local Version Control System:** Local Version Control System is located inyour local machine. If the local machine crashes, it would not be possible to retrieve the files, and all the information will be lost. If anything happens to a single version, all the versions made after that will be lost.

II. **Centralized Version Control System:** In the Centralized Version Control Systems, there will be a single central server that contains all the files related to the project, and many collaborators checkout files from this single server(you will only have a working copy). The problem with the Centralized Version Control Systems is if the central server crashes, almost everything related to the project will be lost.

III. **Distributed Version Control System:** In a distributed version control system, there will be one or more servers and many collaborators similar to the centralized system. But the difference is, not only do they check out the latest version, but each collaborator will have an exact copy of the main repository on their local machines. Each user has their own repository and aworking copy. This is very useful because even if the server crashes we would not lose everything as several copies are residing in several other computers.

2. Problem Statement

Be it a common disease or severe diseases,there is some risk regarding medicinesand their side-effects. People who want to adopt Ayurveda,have to go with drugsbecause of less knowledge and experience.Here this website solves this issue easily,giving better knowledge (from experienced gurus)
.
Apartfrom this you'll not need to open and register on multiple websites to accessall information.

3. Objective

PROJECT NAME: AYURVEDA-The Pride of India

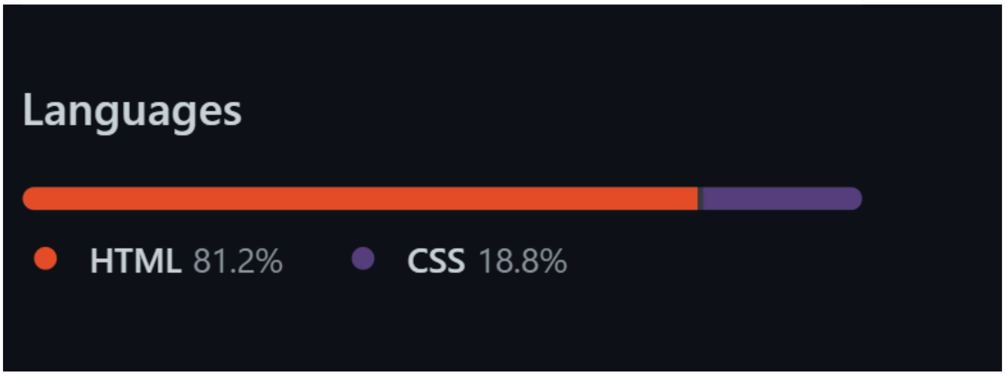
Our goal is to create a web app for our users that will provide them complete information related to Ayurveda, its origin, diseases and their remedies. Our prime objective is to create a lightweight online app with a great experience of getting whole knowledge of ayurvedic centres, institutes alog with appointment features onthe same platform without paying a large fee or being bombarded with annoying adverts.

4. Resources Required.

Frontend – HTML5,

CSS3Backend –

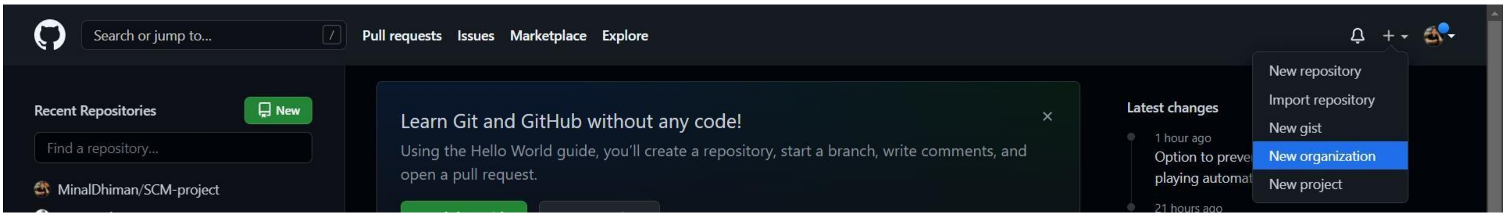
NodeJS



5. Concepts , Commands, Workflow and Discussions.

Aim: Create a distributed Repository and add members in projectteam

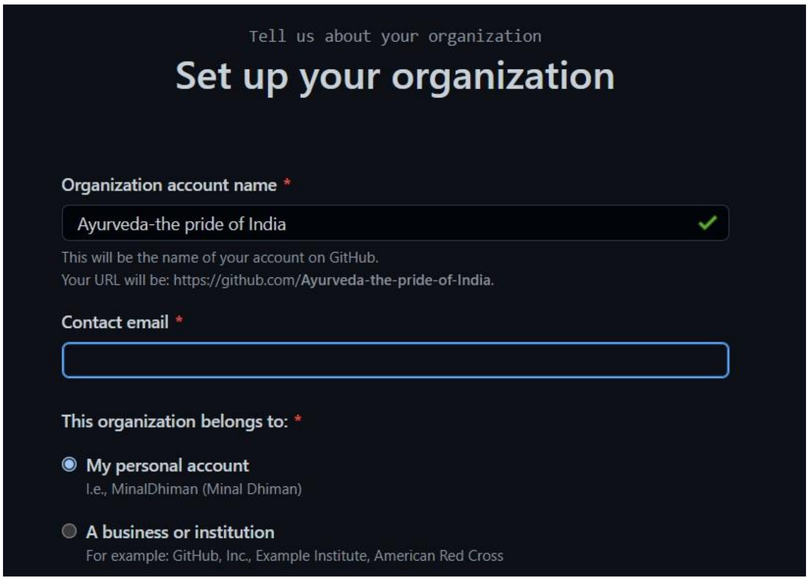
- Login to your GitHub account and you will land on the homepage as shown below. Click on the button shown in the menu bar and then click on New Organization.



Learn Git and GitHub without any code!

Using the Hello World guide, you'll create a repository, start a branch, write comments, and open a pull request.

- Set Up Your Organization. Fill Your Organization's Name and Other Details. .



Tell us about your organization

Set up your organization

Organization account name *

Ayurveda-the pride of India ✓

This will be the name of your account on GitHub.
Your URL will be: https://github.com/Ayurveda-the-pride-of-India.

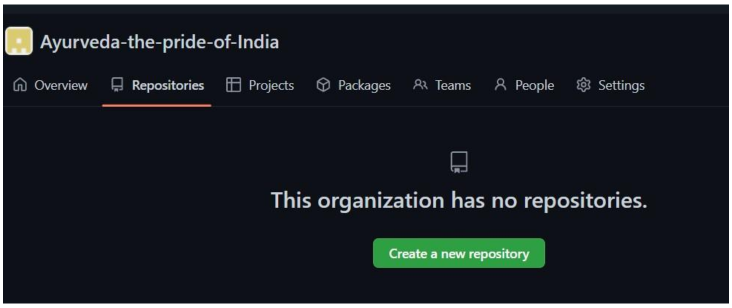
Contact email *

This organization belongs to: *

● My personal account
I.e., MinalDhiman (Minal Dhiman)

○ A business or institution
For example: GitHub, Inc., Example Institute, American Red Cross

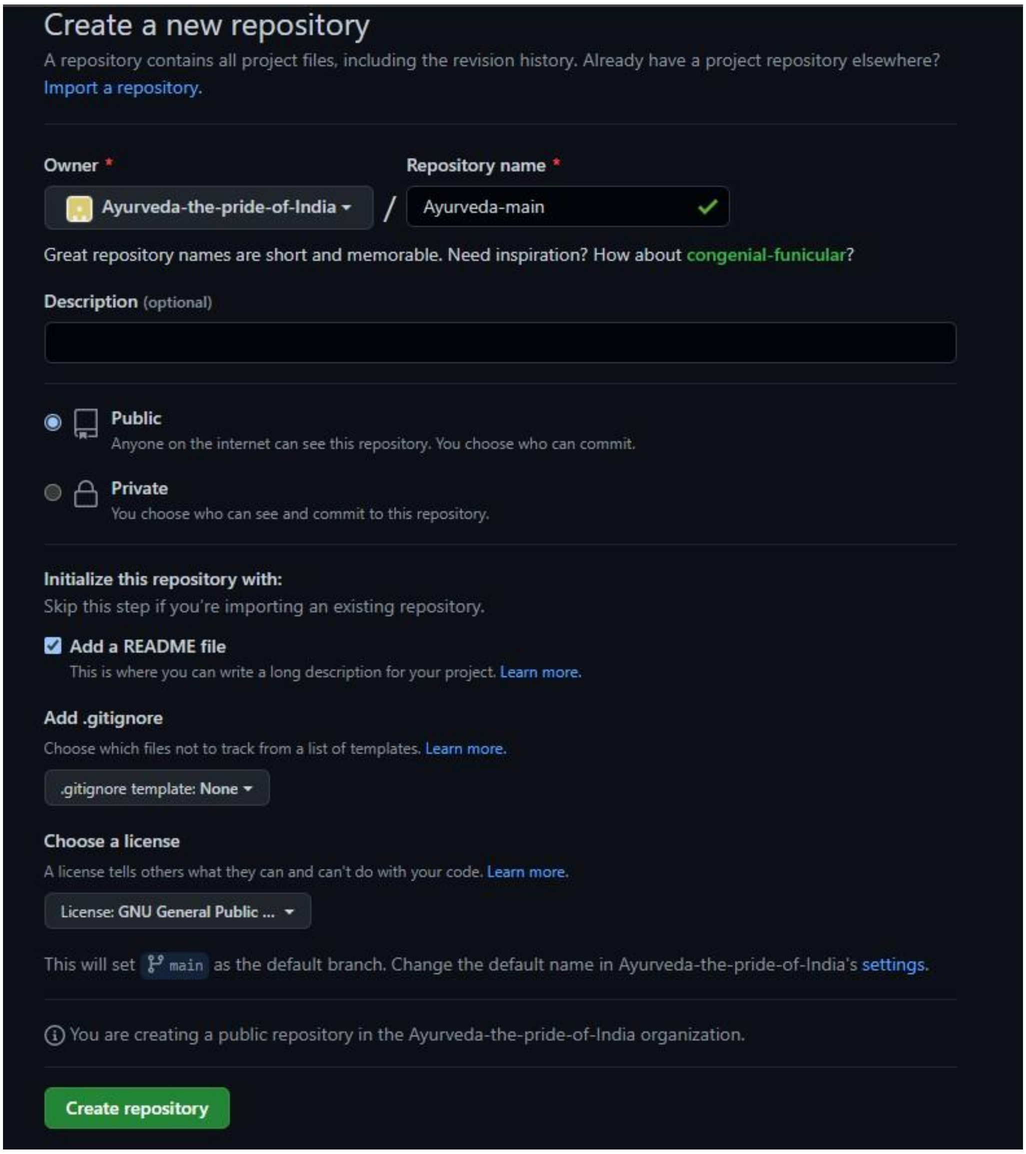
- After creating the repository, we have to create a Repository.



Ayurveda-the-pride-of-India

This organization has no repositories.

- Create a New Repo by Pressing the New repository button. Fillinthe required details.



Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.

Owner *: Ayurveda-the-pride-of-India / Repository name *: Ayurveda-main ✓

Great repository names are short and memorable. Need inspiration? How about congenial-funicular?

Description (optional)

● Public
Anyone on the internet can see this repository. You choose who can commit.

○ Private
You choose who can see and commit to this repository.

Initialize this repository with:
Skip this step if you're importing an existing repository.

☑ Add a README file
This is where you can write a long description for your project. Learn more.

Add .gitignore
Choose which files not to track from a list of templates. Learn more.

.gitignore template: None

Choose a license
A license tells others what they can and can't do with your code. Learn more.

License: GNU General Public ...

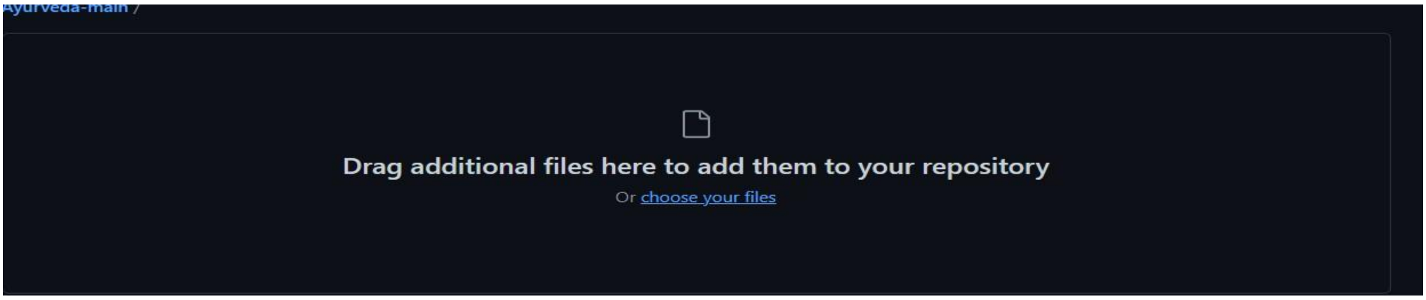
This will set `main` as the default branch. Change the default name in Ayurveda-the-pride-of-India's settings.

You are creating a public repository in the Ayurveda-the-pride-of-India organization.

-

- If you want to import code from an existing repository select theimport code option or you can also use drag and drop option.

- To create a new file or upload an existing file into yourrepository select the option in the following box.



Ayurveda-main /

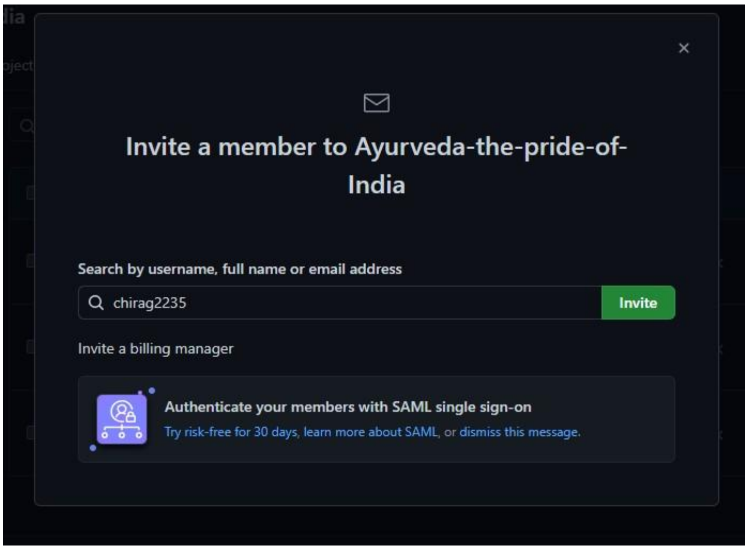
Drag additional files here to add them to your repository

Or choose your files

- Now, you have created your repository successfully.
- To add members to your repository, open your Organizationand select People option in the navigation bar.

- Click on Collaborators option under the access tab.

- To add members click on the add people option and searchthe id of your respective team member.



Invite a member to Ayurveda-the-pride-of-India

Search by username, full name or email address

chirag2235

Invite a billing manager

Authenticate your members with SAML single sign-on

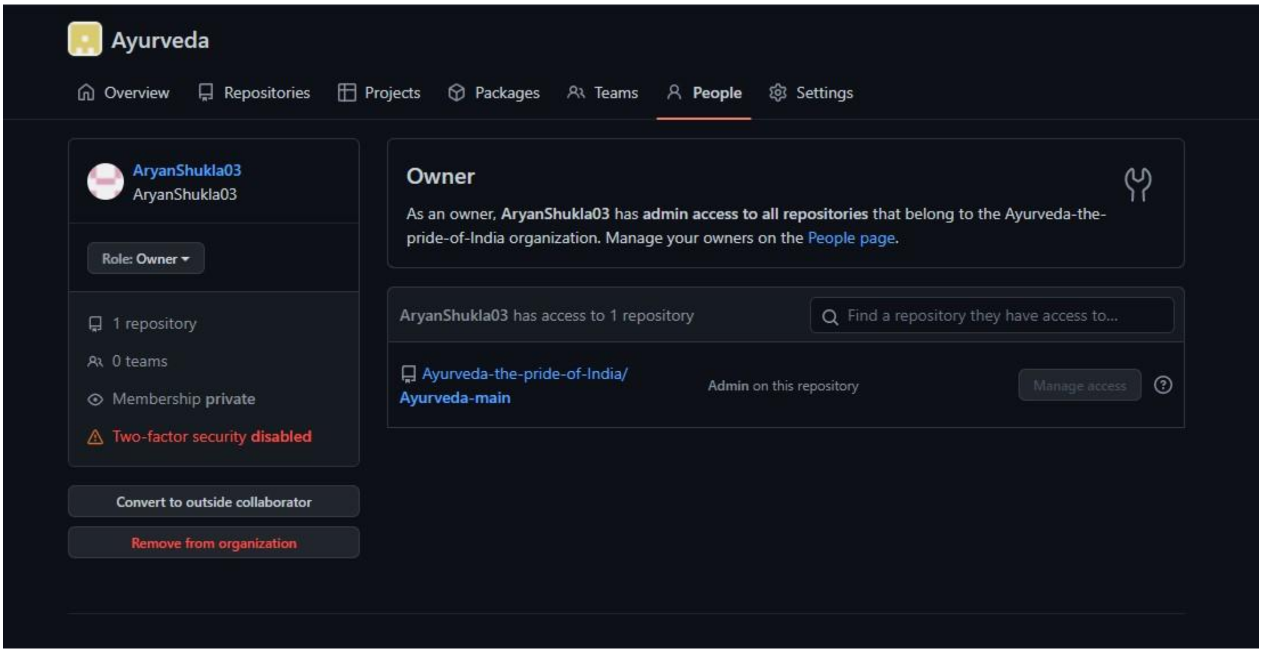
Try risk-free for 30 days, learn more about SAML, or dismiss this message.

- To accept the invitation from your team member, open youremail registered with GitHub.

- You will receive an invitation mail from the repositoryowner. Open the email and click on accept invitation.

- You will be redirected to GitHub where you can either selectto accept or decline the invitation.

- Next, Open the desired Repository in the Organisation. Look and click on Settings -> Collaborators and Teams. Here you can Manage the role of each collaborator.



Owner

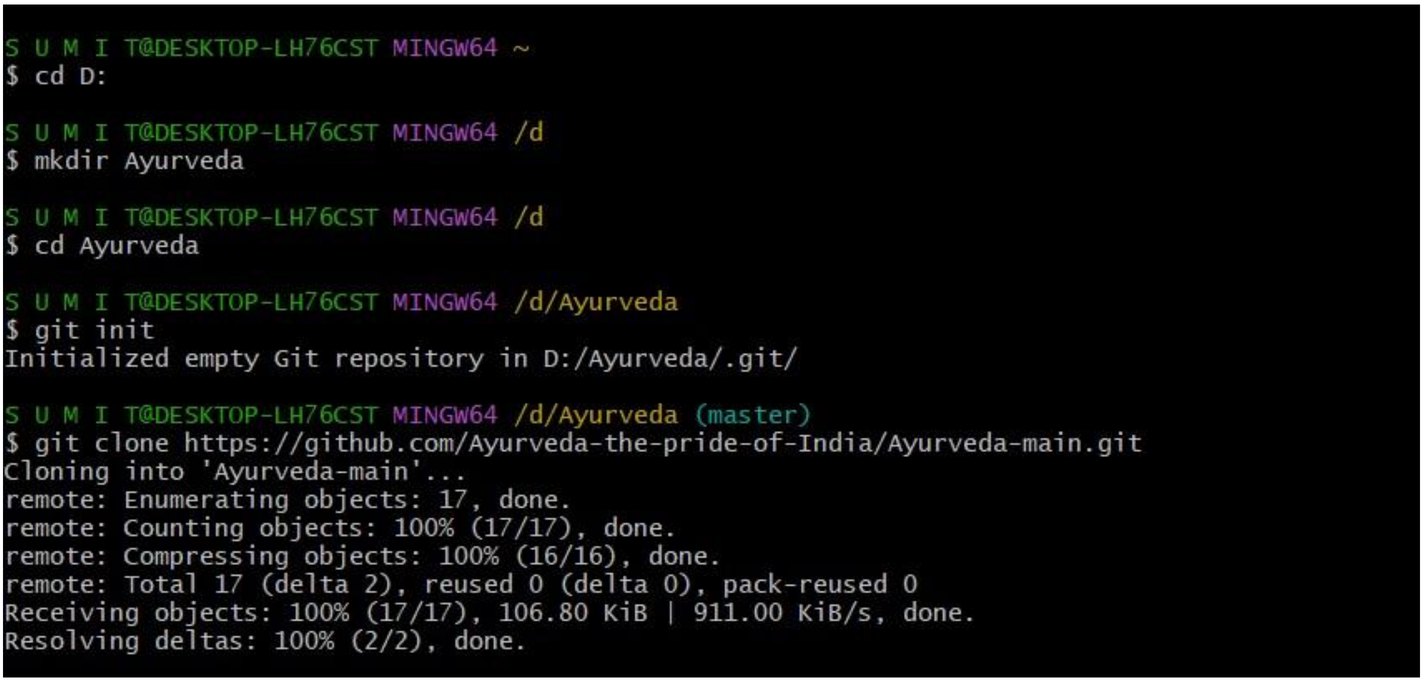
As an owner, **AryanShukla03** has **admin access to all repositories** that belong to the Ayurveda-the-pride-of-India organization. Manage your owners on the People page.

AryanShukla03 has access to 1 repository s

Experiment No. 02

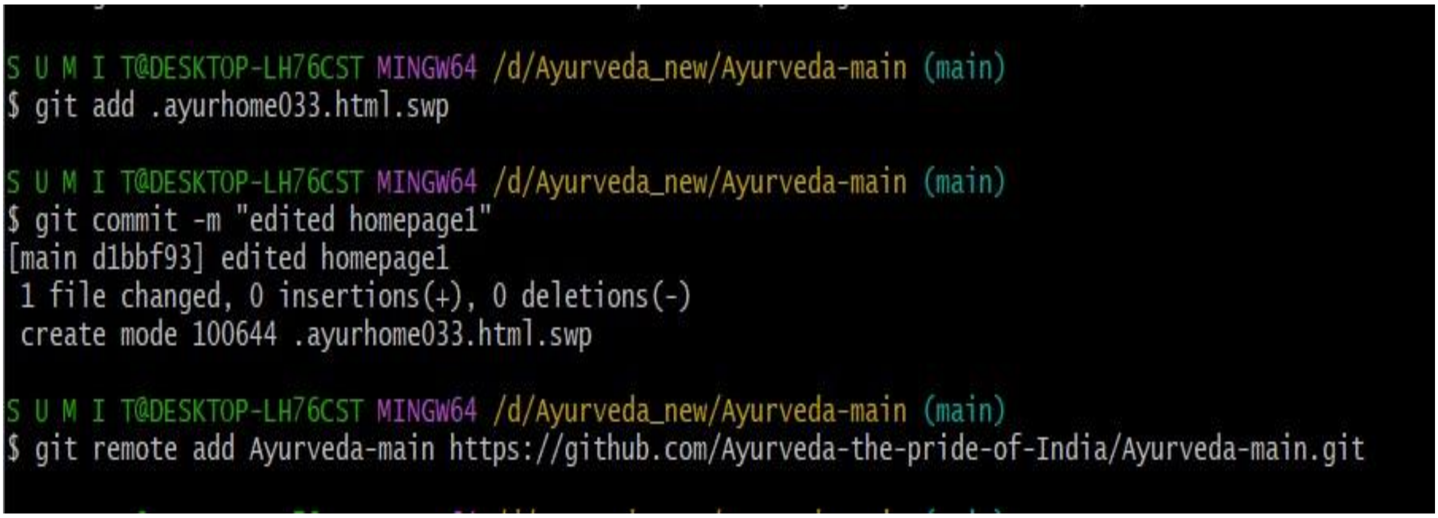
Aim: Open And Close a Pull Request

- To Open a Pull Request, First of All, it will be required to fork the repositoryand commit changes into your own.



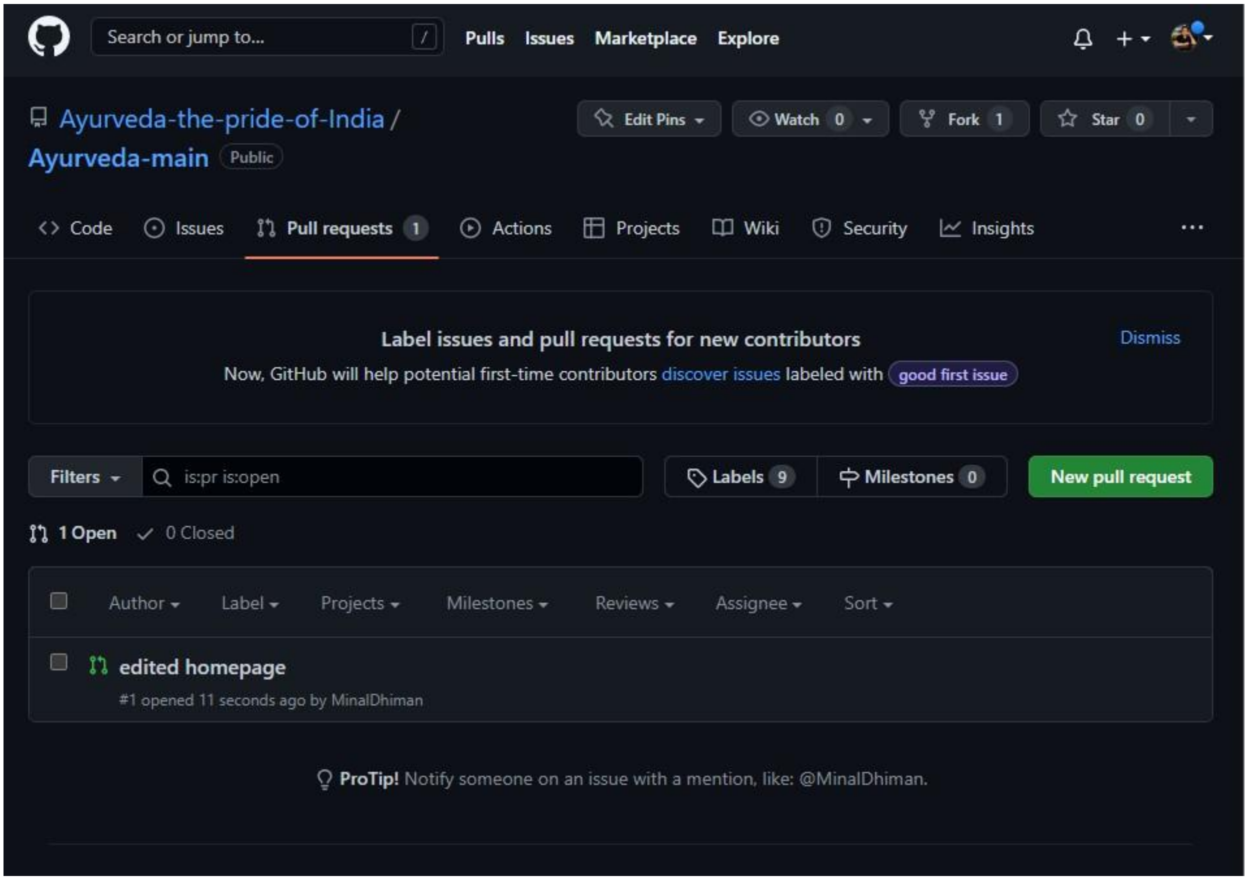
```
S U M I T@DESKTOP-LH76CST MINGW64 ~
$ cd D:

S U M I T@DESKTOP-LH76CST MINGW64 /d
$ mkdir Ayurveda

S U M I T@DESKTOP-LH76CST MINGW64 /d
$ cd Ayurveda

S U M I T@DESKTOP-LH76CST MINGW64 /d/Ayurveda
$ git init
Initialized empty Git repository in D:/Ayurveda/.git/

S U M I T@DESKTOP-LH76CST MINGW64 /d/Ayurveda (master)
$ git clone https://github.com/Ayurveda-the-pride-of-India/Ayurveda-main.git
Cloning into 'Ayurveda-main'...
remote: Enumerating objects: 17, done.
remote: Counting objects: 100% (17/17), done.
remote: Compressing objects: 100% (16/16), done.
remote: Total 17 (delta 2), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (17/17), 106.80 KiB | 911.00 KiB/s, done.
Resolving deltas: 100% (2/2), done.
```

- Add and commit the changes to the local repository
- Use git push origin branch name option to push the new branch to the mainrepository.
- After pushing new branch GitHub will either automatically ask you tocreate apull request or you can create your own pull request.



```
S U M I T@DESKTOP-LH76CST MINGW64 /d/Ayurveda_new/Ayurveda-main (main)
$ git add .ayurhome033.html.swp

S U M I T@DESKTOP-LH76CST MINGW64 /d/Ayurveda_new/Ayurveda-main (main)
$ git commit -m "edited homepage1"
[main d1bbf93] edited homepage1
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 .ayurhome033.html.swp

S U M I T@DESKTOP-LH76CST MINGW64 /d/Ayurveda_new/Ayurveda-main (main)
$ git remote add Ayurveda-main https://github.com/Ayurveda-the-pride-of-India/Ayurveda-main.git
```

- To create your own pull request click on pull request option.



Label issues and pull requests for new contributors

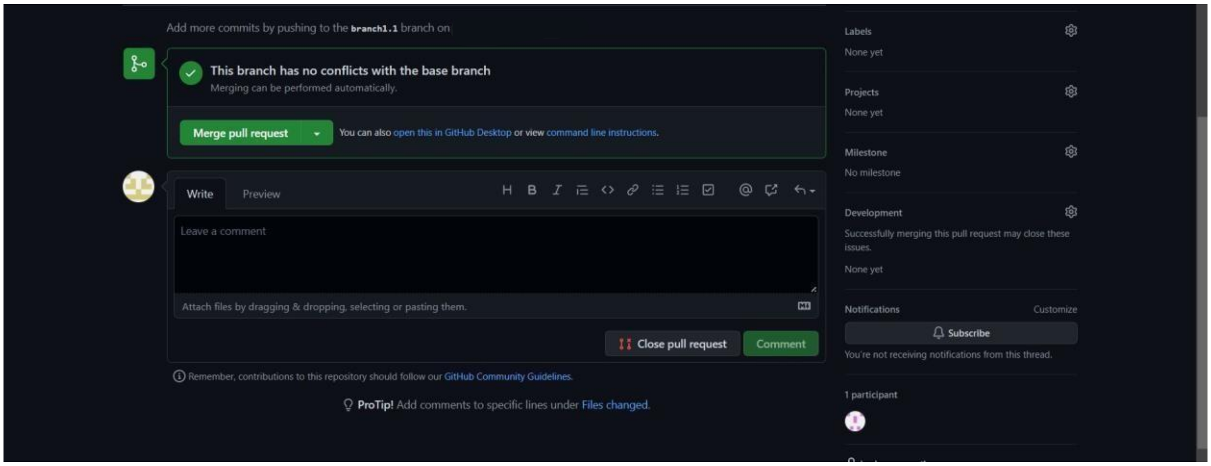
Now, GitHub will help potential first-time contributors discover issues labeled with good first issue

edited homepage

#1 opened 11 seconds ago by MinalDhiman

ProTip! Notify someone on an issue with a mention, like: @MinalDhiman.

- GitHub will detect any conflicts and ask you to enter a description of your pullrequest.
- After opening a pull request all the team members will be sentthe request if theywant to merge or close the request.



Add more commits by pushing to the branch1.1 branch on

This branch has no conflicts with the base branch

Merging can be performed automatically.

You can also open this in GitHub Desktop or view command line instructions.

Leave a comment

Attach files by dragging & dropping, selecting or pasting them.

Remember, contributions to this repository should follow our GitHub Community Guidelines.

ProTip! Add comments to specific lines under Files changed.

Labels: None yet

Projects: None yet

Milestone: No milestone

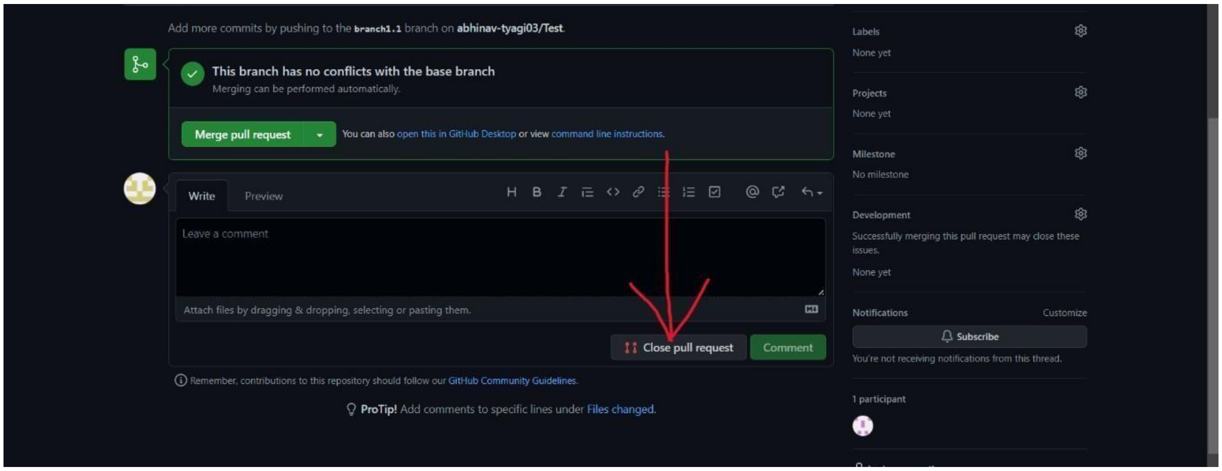
Development: Successfully merging this pull request may close these issues. None yet

You're not receiving notifications from this thread.

1 participant

- If the team member chooses not to merge your pull requestthey will close you’re the pull request.

- To close the pull request simply click on close pull request andadd comment/ reason why you closed the pull request.



Add more commits by pushing to the branch1.1 branch on abhinav-tyagi03/Test.

This branch has no conflicts with the base branch

Merging can be performed automatically.

You can also open this in GitHub Desktop or view command line instructions.

Leave a comment

Attach files by dragging & dropping, selecting or pasting them.

Remember, contributions to this repository should follow our GitHub Community Guidelines.

ProTip! Add comments to specific lines under Files changed.

Labels: None yet

Projects: None yet

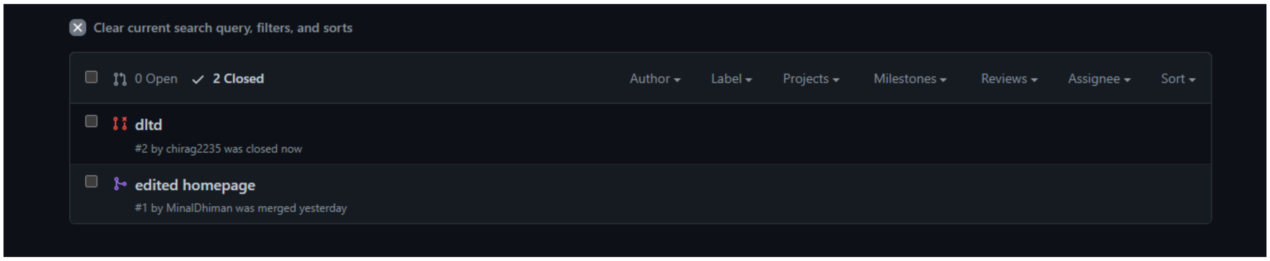
Milestone: No milestone

Development: Successfully merging this pull request may close these issues. None yet

You're not receiving notifications from this thread.

1 participant

- You can see all the pull request generated and how they weredealt with by clicking on pull request option.



□ 0 Open ✓ 2 Closed

□ dltd
#2 by chirag2235 was closed now

□ edited homepage
#1 by MinalDhiman was merged yesterday

Experiment No. 03

Aim: Publish and print network graphs

The network graph is one of the useful features for developers on GitHub. It is used to display the branch history of the entire repository network, including branches of the root repository and branches of forks that contain commits unique to the network.

A repository's graphs give you information on traffic, projects thatdepend on the repository, contributors and commits to the repository, and a repository's forks and network. If you maintain arepository, you can use this data to get a better understanding of who's using your repository and why they're using it.

Some repository graphs are available only in public repositorieswith GitHub Free:

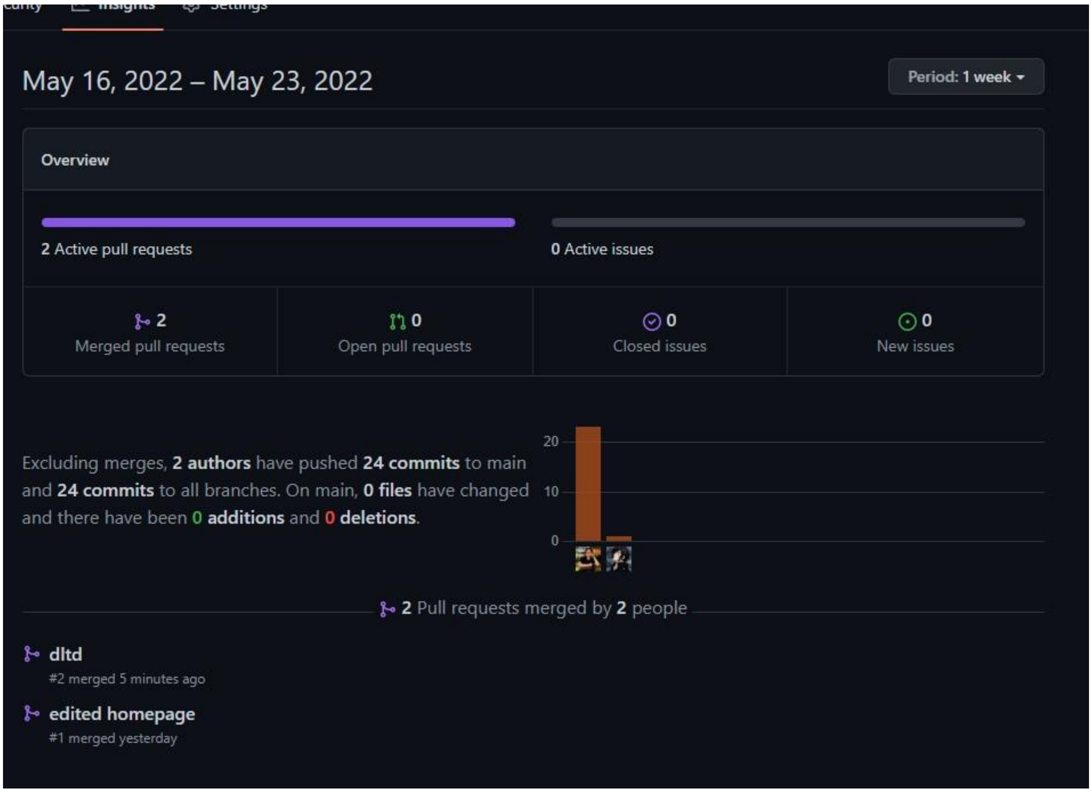
- Pulse
- Contributors
- Traffic
- Commits
- Code frequency
- Network

Steps to access network graphs of respective repository

1. On GitHub.com, navigate to the main page of the repository.

2. Under your repository name, click Insights.

3. At the left sidebar, click on Network.



May 16, 2022 – May 23, 2022

Overview

2 Active pull requests

0 Active issues

2 Merged pull requests

0 Open pull requests

0 Closed issues

0 New issues

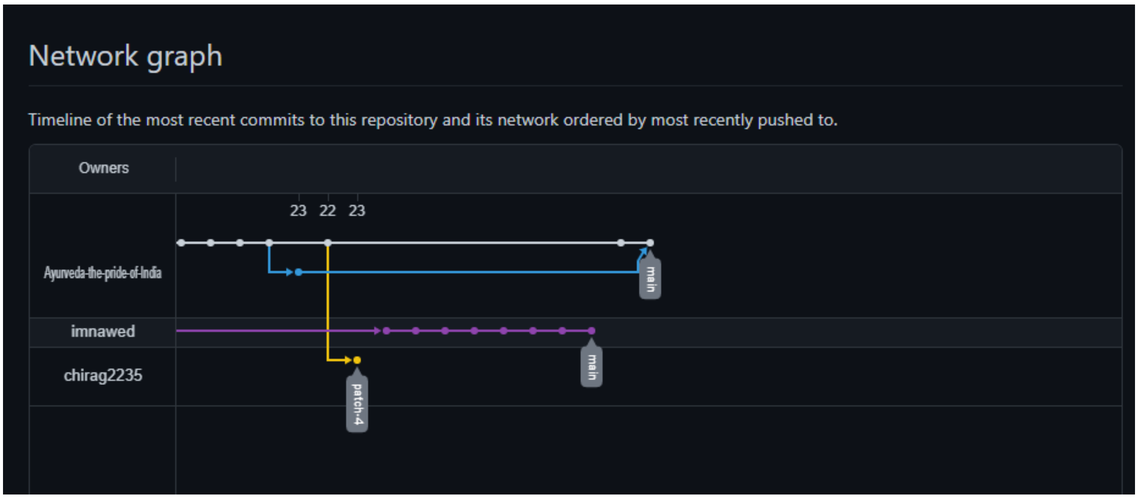
Excluding merges, **2 authors** have pushed **24 commits** to main and **24 commits** to all branches. On main, **0 files** have changed and there have been **0 additions** and **0 deletions**.

2 Pull requests merged by **2** people

dltd
#2 merged 5 minutes ago

edited homepage
#1 merged yesterday

You will get the network graph of your repository which displays the branchhistory of the entire repository network, including branches of the root repository and branches of forks that contain commits unique to the network.



Network graph

Timeline of the most recent commits to this repository and its network ordered by most recently pushed to.