Subject Name: **Source Code Management**

Subject Code: **CS181**

Cluster: **Beta**

Department: **CSE**

****

|  |  |  |
| --- | --- | --- |
| **Submitted By:**  Anand Kumar  2110990172  G3 |  | **Submitted To:**  Dr. Deepak Thakur |

**INDEX**

|  |  |  |
| --- | --- | --- |
| **S. No** | **LIST OF PRACTICALS** | **Page No.** |
| 1 | How to install and configure the Git client | 3-16 |
| 2 | How to set up your GitHub account and getting started with GitHub's features for collaboration and community. | 17-27 |
| 3 | Package ‘logr’ | 28-43 |
| 4 | Learning about Git branches | 44-46 |
| 5 | Understanding Git-Life Cycle | 47-48 |

How to install and configure the Git client

**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.

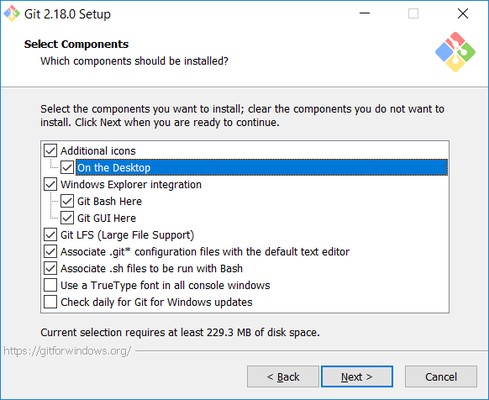
***Be sure to carefully follow all of the steps in the first five sections.*** The last section, 6, is optional. There is also a section on common problems and possible fixes at the bottom of the document.

# Git installation

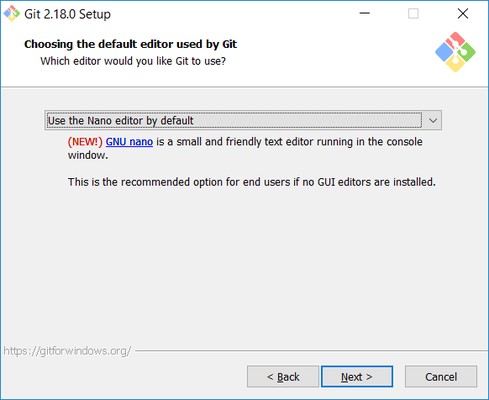
Download the Git installation program (Windows, Mac, or Linux) from <http://git-scm.com/downloads>.

When running the installer, various screens appear (Windows screens shown). Generally, you can accept the default selections, ***except in the screens below where you do NOT want the default selections:***

In the **Select Components** screen, make sure **Windows Explorer Integration** is selected as shown:



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

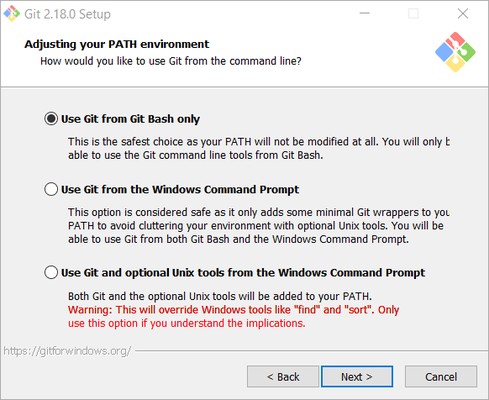


In the **Adjusting your PATH** screen, all three options are acceptable:

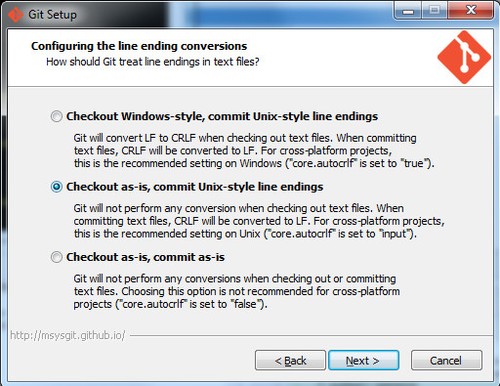
**Use Git from Git Bash only**: no integration, and no extra commands in your command path

**Use Git from the Windows Command Prompt**: adds flexibility - you can simply run git from a Windows command prompt, and is often the setting for people in industry - but this does add some extra commands.

**Use Git and optional Unix tools from the Windows Command Prompt**: this is also a robust choice and useful if you like to use Unix commands like grep.



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



# 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:

* 1. 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**.
  2. Enter the command (replacing name as appropriate)

gitignore

git config --global core.excludesfile c:/users/name/.

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.

* 1. Enter the command git config --global user.email "[name@msoe.edu"](mailto: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.

* 1. 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.

* 1. Enter the command git config --global push.default simple

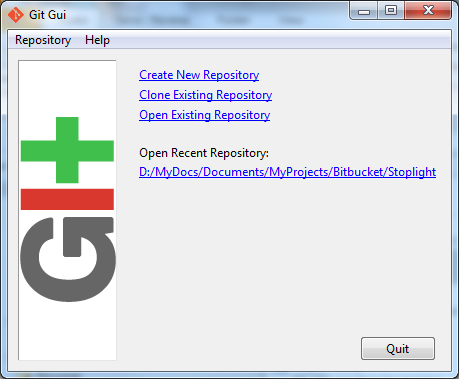
This ensures that all pushes go back to the branch from which they were pulled. Otherwise pushes will go to the master branch, forcing a merge.

# Generating public/private key pairs for authentication

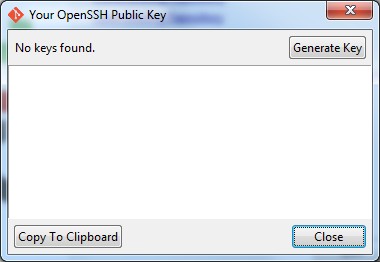
### This part is critical and used to authenticate your access to the repository.

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).

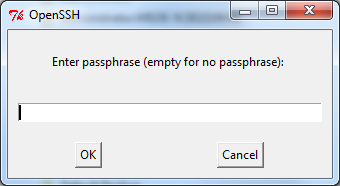
Generating the key pair is easy: From within File Explorer, right-click on any folder. From the context menu, select **Git GUI Here**. The following appears:



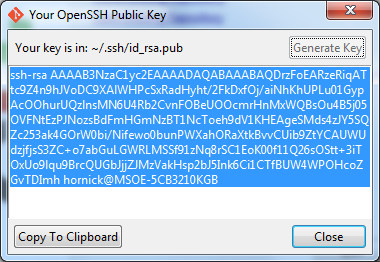
From the **Help** menu, select the **Show SSH Key** command. The following pup-up dialog appears:



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:



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:



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.

# 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.

# 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.

1. Enter the command git config --global merge[.](http://user.name/)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](http://winmerge.org/downloads/) at <http://winmerge.org/downloads/>.

1. Enter the following commands to complete the WinMerge configuration:
   1. git config --global [mergetool.winmerge.name](http://mergetool.winmerge.name/) WinMerge
   2. git config --global mergetool.winmerge.trustExitCode true
   3. 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"

* 1. 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"

1. Enter the command git config --global diff[.](http://user.name/)tool winmerge

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

1. Enter the commands to complete the WinMerge diff configuration:
   1. git config --global [difftool.winmerge.name](http://difftool.winmerge.name/) WinMerge
   2. git config --global difftool.winmerge.trustExitCode true
   3. 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"
   4. 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. |

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](https://docs.github.com/en/github/authenticating-to-github/keeping-your-account-and-data-secure/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](https://docs.github.com/en/get-started/learning-about-github/githubs-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](https://docs.github.com/en/github/getting-started-with-github/signing-up-for-github/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](https://docs.github.com/en/github/authenticating-to-github/securing-your-account-with-two-factor-authentication-2fa/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](https://docs.github.com/en/github/setting-up-and-managing-your-github-profile/customizing-your-profile/about-your-profile)" and "[Viewing contributions on your profile](https://docs.github.com/en/github/setting-up-and-managing-your-github-profile/managing-contribution-graphs-on-your-profile/viewing-contributions-on-your-profile)."

**Part 2: Using GitHub's tools and processes**

To best use GitHub, you'll need to set up Git. Git is responsible for everything GitHub-related 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](https://guides.github.com/introduction/git-handbook/)" guide. For more information about how Git is used on GitHub, see "[GitHub flow](https://docs.github.com/en/get-started/quickstart/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](https://docs.github.com/en/get-started/quickstart/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](https://docs.github.com/en/desktop/installing-and-configuring-github-desktop/overview/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](https://docs.github.com/en/get-started/getting-started-with-git/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](https://docs.github.com/en/github/authenticating-to-github/keeping-your-account-and-data-secure/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 files 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 locally. |
| 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](https://docs.github.com/en/desktop/installing-and-configuring-github-desktop/overview/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](https://atom.io/) or [Visual Studio Code](https://code.visualstudio.com/) to open and edit your files with Git, use extensions, and view the project structure. For more information, see "[Associating text editors with Git](https://docs.github.com/en/github/using-git/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](https://docs.github.com/en/github/getting-started-with-github/quickstart/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](https://docs.github.com/en/github/getting-started-with-github/using-github/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](https://docs.github.com/en/github/extending-github/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, easy-to-write syntax with some custom functionality. For more information, see "[About writing and formatting on GitHub](https://docs.github.com/en/github/writing-on-github/about-writing-and-formatting-on-github)."

You can learn GitHub Flavored Markdown with the "[Communicating using Markdown](https://lab.github.com/githubtraining/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](https://docs.github.com/en/github/searching-for-information-on-github/getting-started-with-searching-on-github/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](https://docs.github.com/en/github/searching-for-information-on-github/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](https://docs.github.com/en/github/managing-files-in-a-repository/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](https://docs.github.com/en/github/creating-cloning-and-archiving-repositories/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](https://docs.github.com/en/github/creating-cloning-and-archiving-repositories/creating-a-repository-on-github/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](https://docs.github.com/en/github/creating-cloning-and-archiving-repositories/cloning-a-repository-from-github/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](https://docs.github.com/en/github/collaborating-with-pull-requests/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](https://docs.github.com/en/github/importing-your-projects-to-github/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](https://docs.github.com/en/github/setting-up-and-managing-your-github-user-account/managing-access-to-your-personal-repositories/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](https://docs.github.com/en/github/setting-up-and-managing-your-github-user-account/managing-user-account-settings/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](https://docs.github.com/en/github/administering-a-repository/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](https://opensource.guide/building-community/)" 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](https://docs.github.com/en/communities/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](https://docs.github.com/en/issues/tracking-your-work-with-issues/about-issues)" and "[About project boards](https://docs.github.com/en/issues/organizing-your-work-with-project-boards/managing-project-boards/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](https://docs.github.com/en/github/managing-subscriptions-and-notifications-on-github/setting-up-notifications/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](https://docs.github.com/en/pages/getting-started-with-github-pages/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](https://docs.github.com/en/discussions/collaborating-with-your-community-using-discussions/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](https://github.com/marketplace). For more information, see "[About GitHub Marketplace](https://docs.github.com/en/github/customizing-your-github-workflow/exploring-integrations/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](https://docs.github.com/en/github/creating-cloning-and-archiving-repositories/archiving-a-github-repository/backing-up-a-repository), or [create integrations](https://docs.github.com/en/github/customizing-your-github-workflow/exploring-integrations/about-integrations) that extend GitHub. For more information, see "[About GitHub's APIs](https://docs.github.com/en/developers/overview/about-githubs-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](https://docs.github.com/en/actions/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](https://docs.github.com/en/packages/learn-github-packages/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](https://docs.github.com/en/code-security/getting-started/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](https://docs.github.com/en/code-security/getting-started/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](https://docs.github.com/en/code-security/supply-chain-security)."

**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](https://opensource.guide/how-to-contribute/)" 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](https://github.com/explore). For more information, see "[Finding ways to contribute to open source on GitHub](https://docs.github.com/en/github/getting-started-with-github/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](https://github.community/) 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](https://docs.github.com/en/get-started/learning-about-github/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](https://lab.github.com/). 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](https://docs.github.com/en/github/getting-started-with-github/quickstart/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](https://docs.github.com/en/sponsors/getting-started-with-github-sponsors/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](https://docs.github.com/en/github/working-with-github-support/about-github-support)."

**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](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>](mailto:dbosak01@gmail.com)

**Repository** CRAN

**Date/Publication** 2022-03-08 16:10:02 UTC

R **topics documented:**

logr [2](#_bookmark0)

log\_close [2](#_bookmark0)

log\_code [3](#_bookmark2)

log\_open [4](#_bookmark3)

log\_path [6](#_bookmark5)

log\_print [7](#_bookmark6)

log\_status [9](#_bookmark9)

Index [11](#_bookmark10)

2 *log\_close*

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:

* [log\_open](#_bookmark4)
* [log\_print](#_bookmark7)
* [log\_close](#_bookmark1)

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()

*log\_code* 3

## 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](#_bookmark4) to open the log, and [log\_print](#_bookmark7) 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)

# Print data to log log\_print(hmc)

# Close log log\_close()

# 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](#_bookmark4). 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 written.

4 *log\_open*

## Usage

log\_code()

## Value

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

## See Also

[log\_open](#_bookmark4) to open the log, and [log\_close](#_bookmark1) to close the log.

## Examples

# Create temp file location

tmp <- file.path(tempdir(), "test.log")

# Open log

lf <- log\_open(tmp)

# Write code to the log log\_code()

# 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\_open *Open a log*

## Description

A function to initialize the log file.

## Usage

log\_open(file\_name = "", logdir = TRUE, show\_notes = TRUE, autolog = NULL)

*log\_open* 5

## 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 override a NULL on this parameter. Default is that autolog is disabled.

## 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. Any- thing 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.

6 *log\_path*

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](#_bookmark7) for printing to the log (and console), and [log\_close](#_bookmark1) 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*

*log\_print* 7

## 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)

8 *log\_print*

## 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](#_bookmark4) to open the log, and [log\_close](#_bookmark1) to close the log.

*log\_status* 9

## 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(tmp)

10 *log\_status*

# Check status after log is opened log\_status()

# [1] "open"

# Close log log\_close(

# Index

log\_close, [*2*](#_bookmark0), [2](#_bookmark0), [*4*](#_bookmark3), [*6*](#_bookmark5), [*8*](#_bookmark8)

log\_code, [3](#_bookmark2)

log\_hook *(*log\_print*)*, [7](#_bookmark6)

log\_open, [*2*](#_bookmark0)*–*[*4*](#_bookmark3), [4](#_bookmark3), [*8*](#_bookmark8)

log\_path, [6](#_bookmark5)

log\_print, [*2*](#_bookmark0)*,* [*3*](#_bookmark2), [*6*](#_bookmark5), [7](#_bookmark6)

log\_status, [9](#_bookmark9)

logr, [2](#_bookmark0)

put *(*log\_print*)*, [7](#_bookmark6)

sep *(*log\_print*)*, [7](#_bookmark6)

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 large-scale 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](https://www.atlassian.com/git/tutorials/using-branches/git-checkout) and [git merge](https://www.atlassian.com/git/tutorials/using-branches/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 crazy-experiment 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

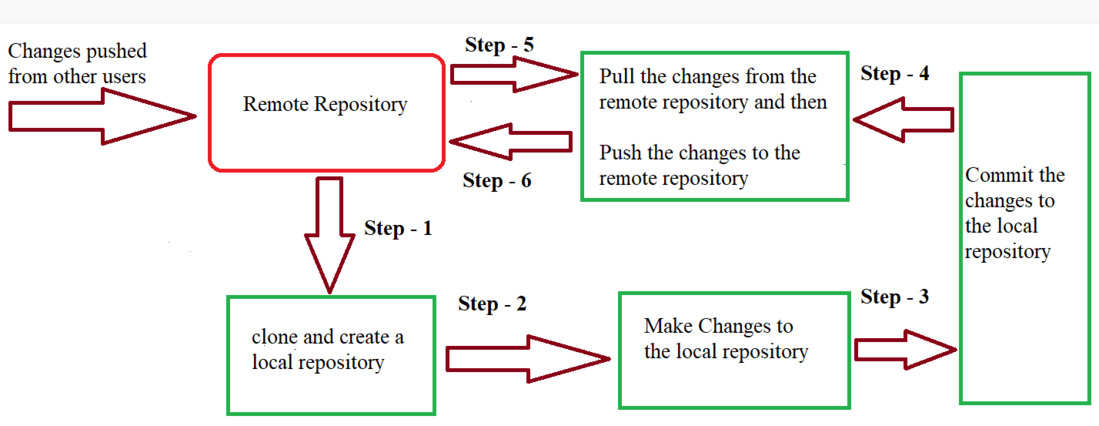
In this document we discussed Git's branching behavior and the git branch command. The git branch commands primary functions are to create, list, rename and delete branches. To operate further on the resulting branches the command is commonly used with other commands like git checkout. Learn more about git checkout branch operations; such as switching branches and merging branches, on the [git checkout](https://www.atlassian.com/git/tutorials/using-branches/git-checkout) page.

Compared to other VCSs, Git's branch operations are inexpensive and frequently used. This flexibility enables powerful [Git workflow](https://www.atlassian.com/git/tutorials/comparing-workflows) customization. For more info on Git workflows visit our extended workflow discussion pages: [The Feature Branch Workflow](https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow), [GitFlow Workflow](https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow), and [Forking Workflow](https://www.atlassian.com/git/tutorials/comparing-workflows/forking-workflow).

**Git – Life Cycle**

* Last Updated : 30 Jan, 2022

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 –



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

* 1. **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 are stored. ***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 to add 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 understanding which 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*

*git add <filename>*

*// to add all files of the working directory to the staging area*

*git add .*

**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 that were committed in a Git Directory which helps Git in tracking files and basically it preserves the photocopy of the committed files. Commit also stores the 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>*