Subject Name: **Source Code Management**

Subject Code: **CS181**

Cluster: **Beta**

Department: **CSE**

****

|  |  |  |
| --- | --- | --- |
| **Submitted By:**  Harsh Raj  2110990564  G3 |  | **Submitted To:**  Dr. Deepak Thakur |

**INDEX**

|  |  |  |
| --- | --- | --- |
| **S. No** | **LIST OF PRACTICALS** | **Page No.** |
| 1 | How to install and configure the Git client | 3-11 |
| 2 | How to set up your GitHub account and getting started with GitHub's features for collaboration and community. | 12-22 |
| 3 | Package ‘logr’ | 23-34 |
| 4 | Learning about Git branches | 35-40 |
| 5 | Understanding Git-Life Cycle | 41-42 |

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.

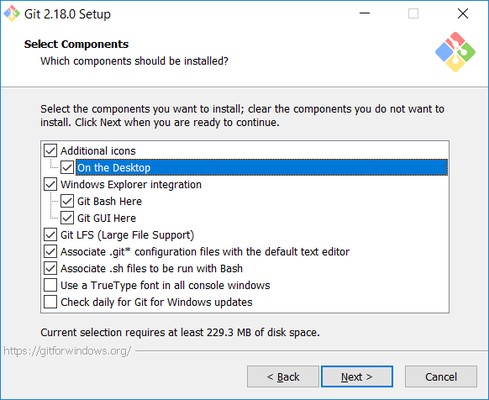
***Be sure to carefully follow all of the steps in the first five sections.*** The last section, 6, is optional. There is also a section on common problems and possible fixes at the bottom of the document.

# Git installation

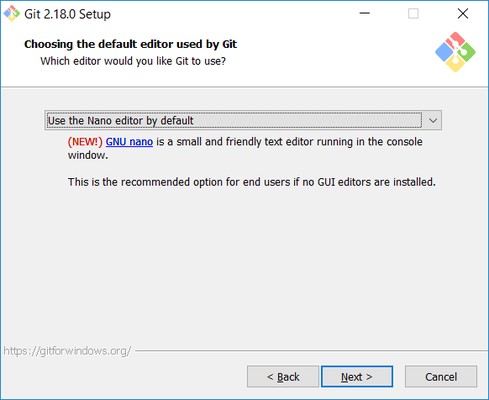
Download the Git installation program (Windows, Mac, or Linux) from <http://git-scm.com/downloads>.

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.

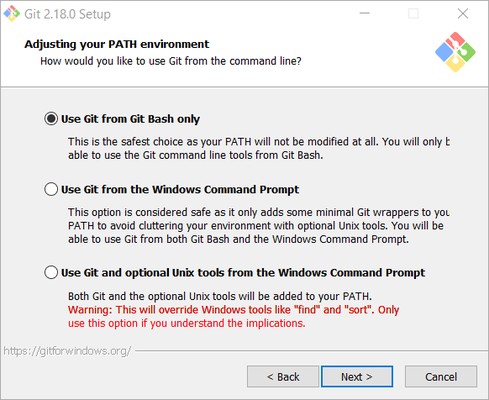


In the **Adjusting your PATH** screen, all three options are acceptable:

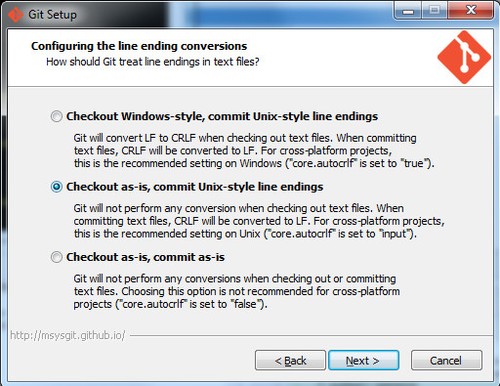
**Use Git from Git Bash only**: no integration, and no extra commands in your command path

**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.



**How to set 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)."

### Learning about Git branches

### 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.

# 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

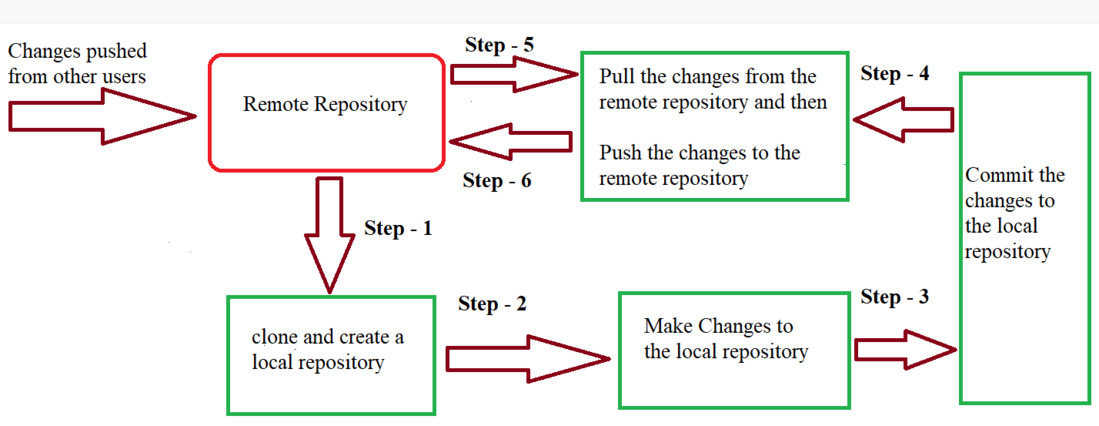
Or

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.

**Understanding 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 –



* ***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*

**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 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>*