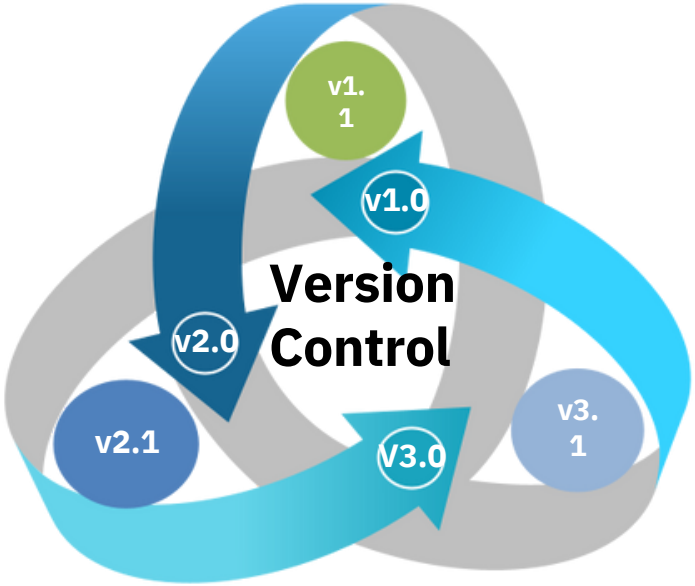
Git Version Control

www.nomadicmehul.com

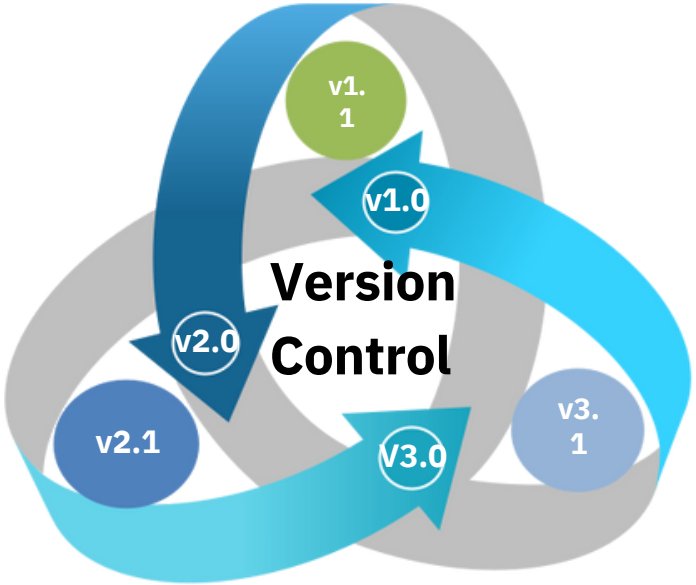
What is Version Control

Version Control is a system that documents changes made to a file or a set of files. It allows multiple users to manage multiple revisions of the same unit of information. It is a snapshot of your project over time.



What is Version Control

- Version Control is a system that documents changes made to a file or a set of files
- It allows multiple users to manage multiple revisions of the same unit of information
- It is a snapshot of your project over time

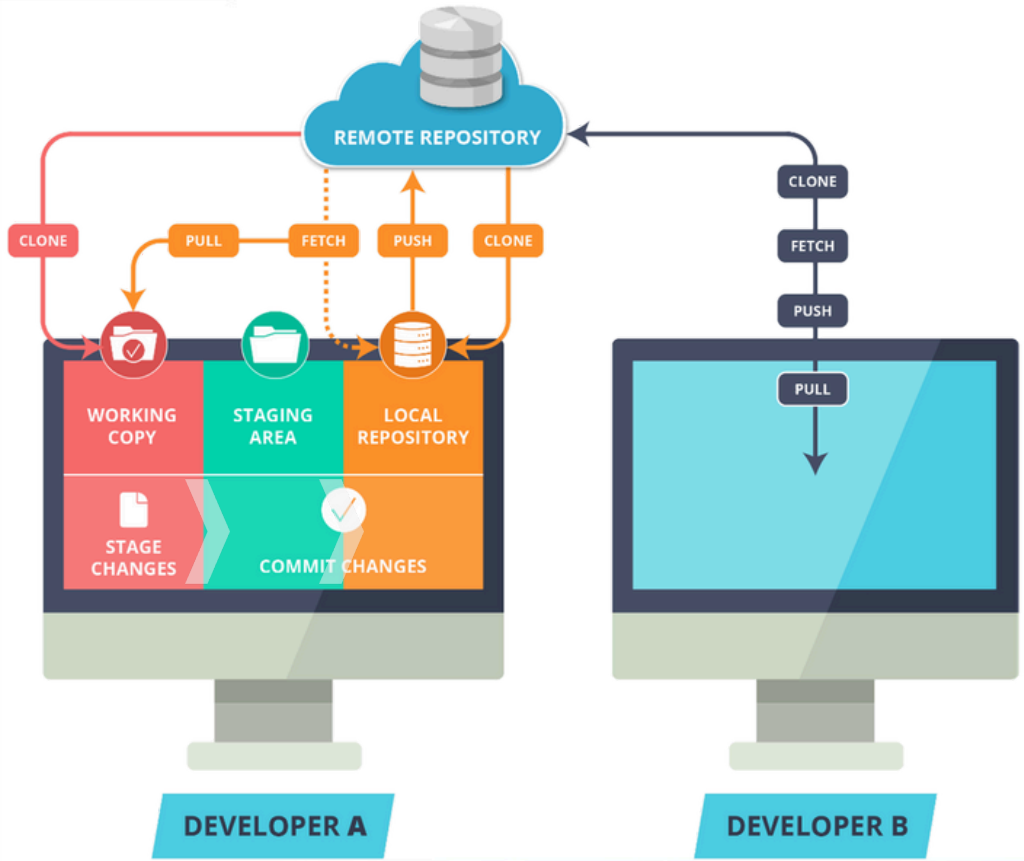


Git

Git is an open source Distributed Version Control System(DVCS) which records changes made to the files laying emphasis on **speed**, **data integrity** and **distributed**, **non-linear workflows**

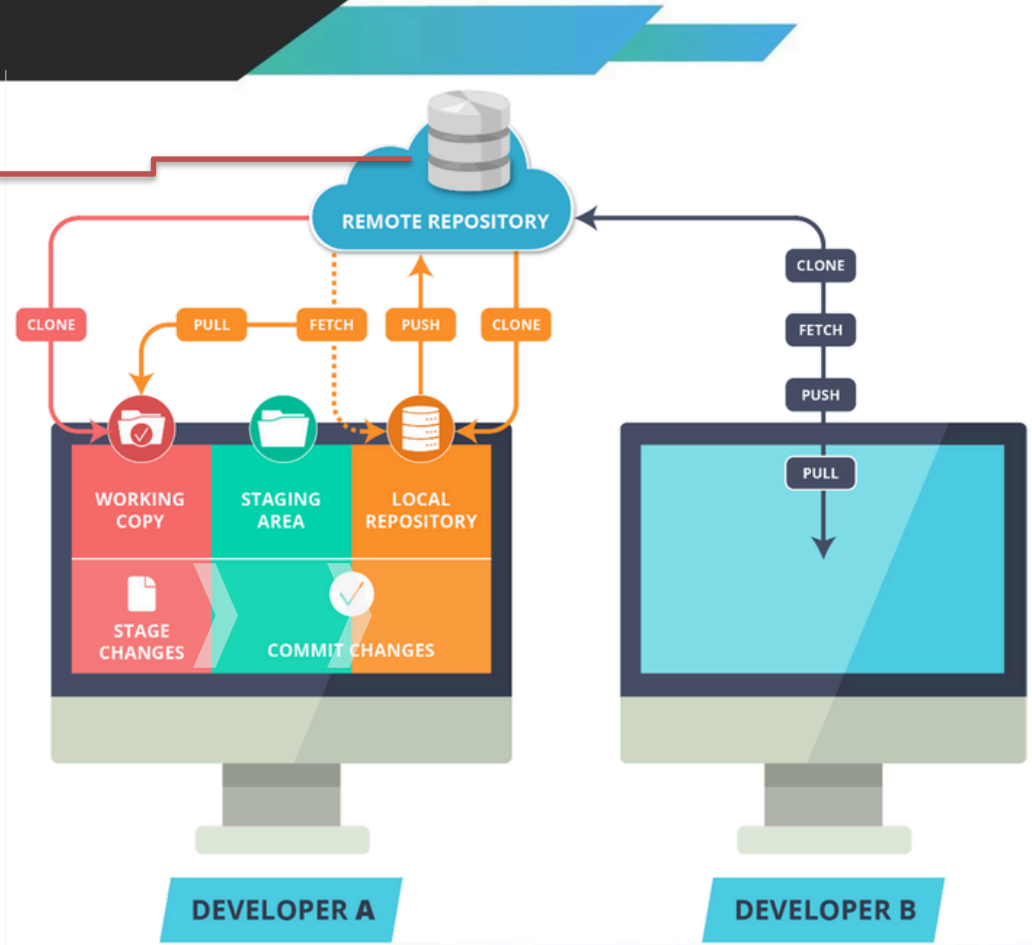
Git Workflow

- Use Git workflow to manage your project effectively
- Working with set of guidelines increases Git's consistency and productivity



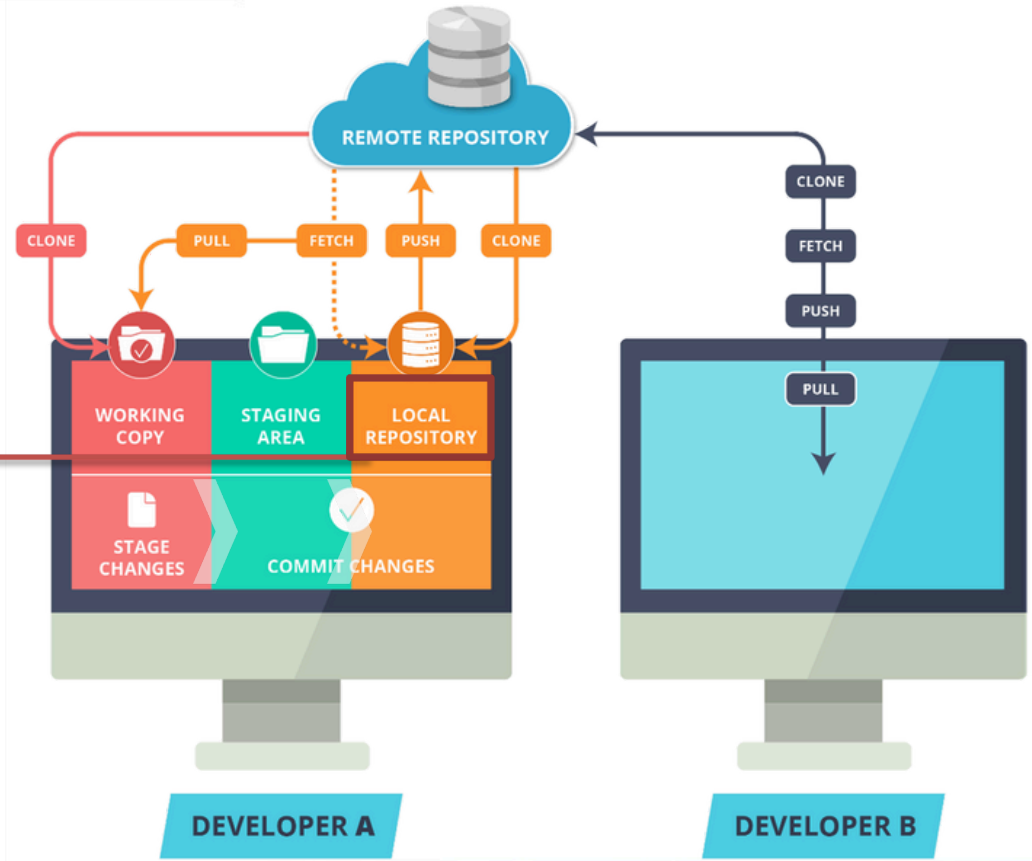
Git Workflow

- The Remote Repository is the server where all the collaborators upload changes made to the files



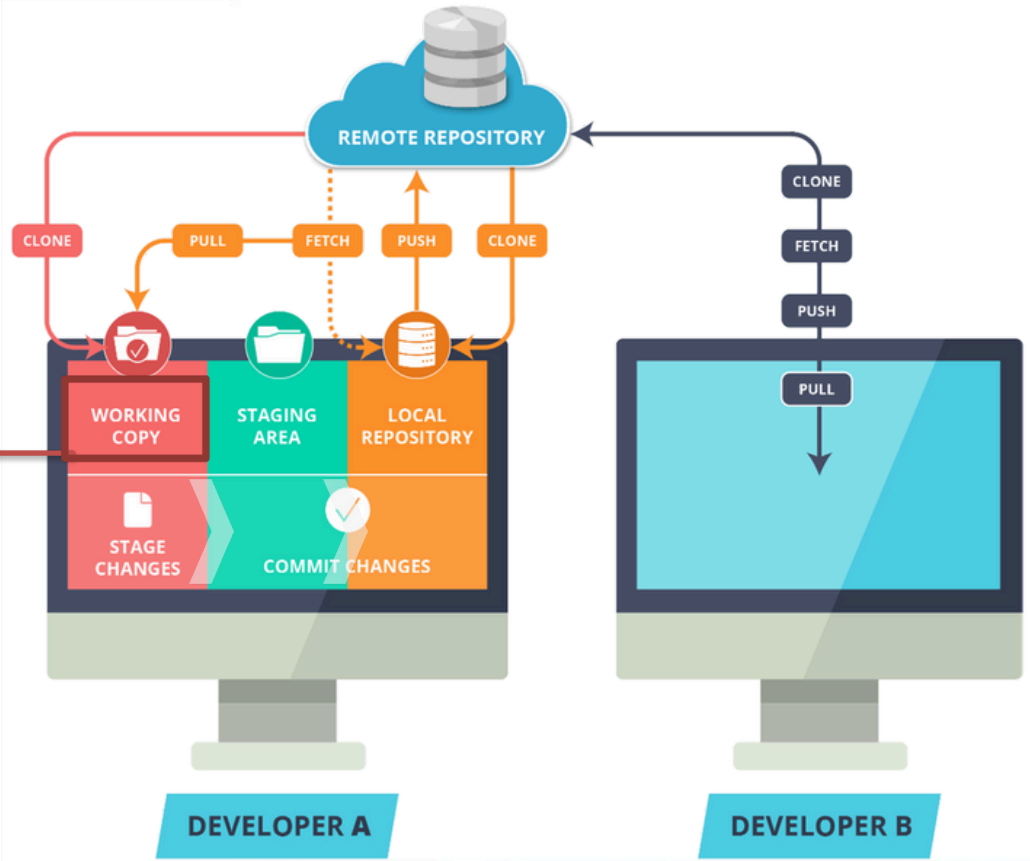
Git Workflow

- “**Local Repository**” is user’s copy of the Version Database
- The user accesses all the files through local repository and then push the change made to the “**Remote Repository**”



Git Workflow

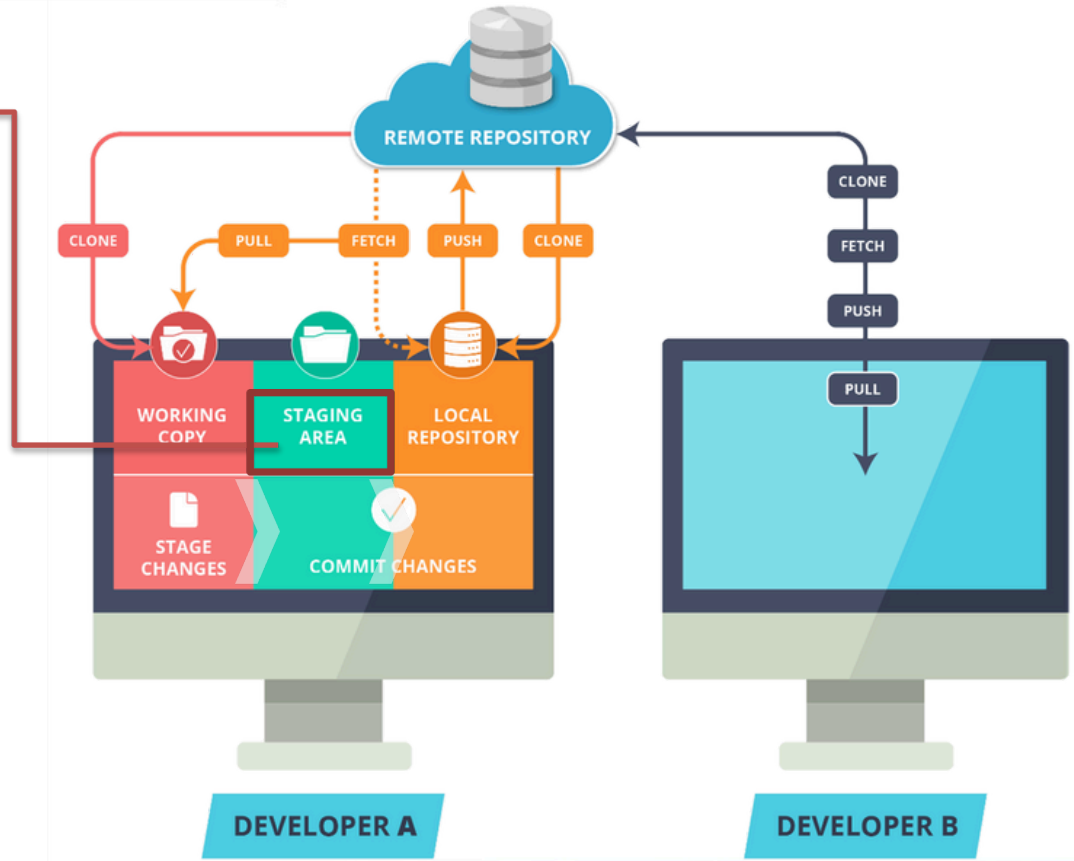
- **“Workspace”** is user’s active directory
- The user modifies existing files and creates new files in this space. Git tracks these changes compared to your Local Repository



Git Workflow

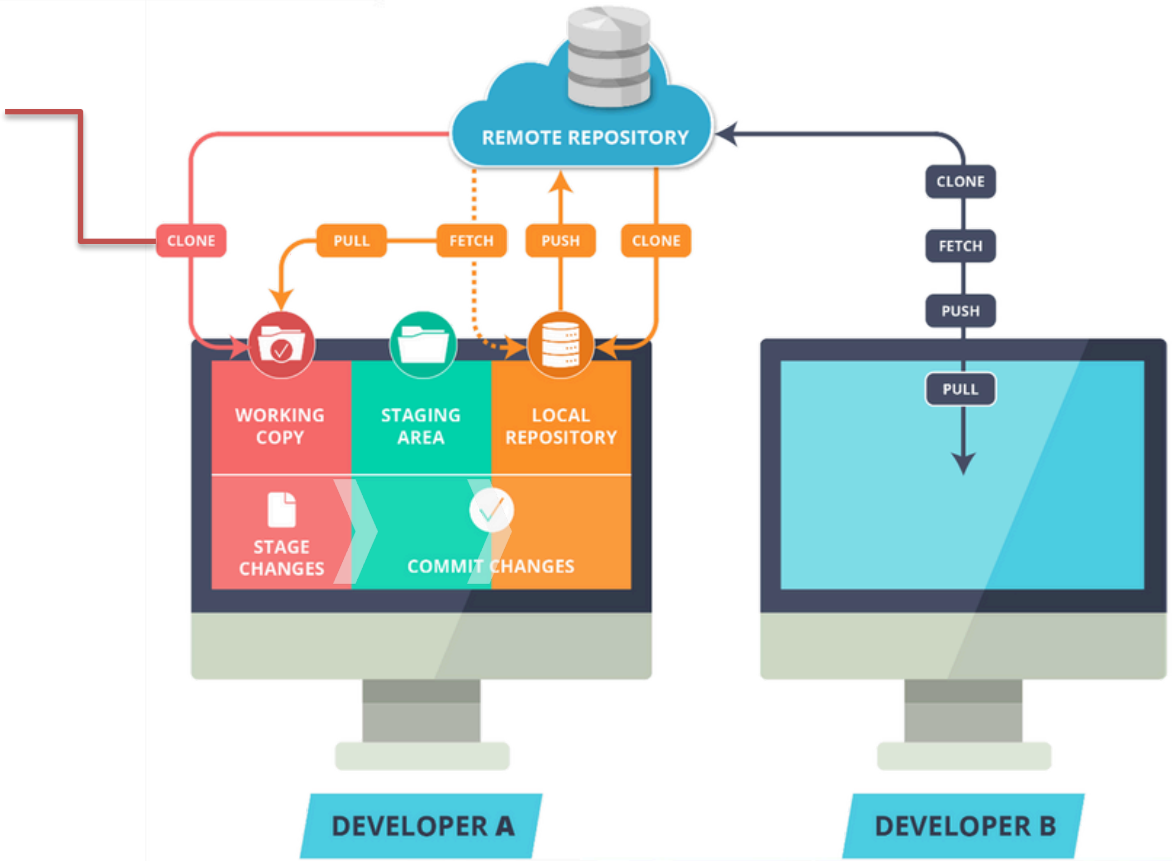
- Stage is a place where all the modified files marked to be

  committed are placed.



Git Workflow

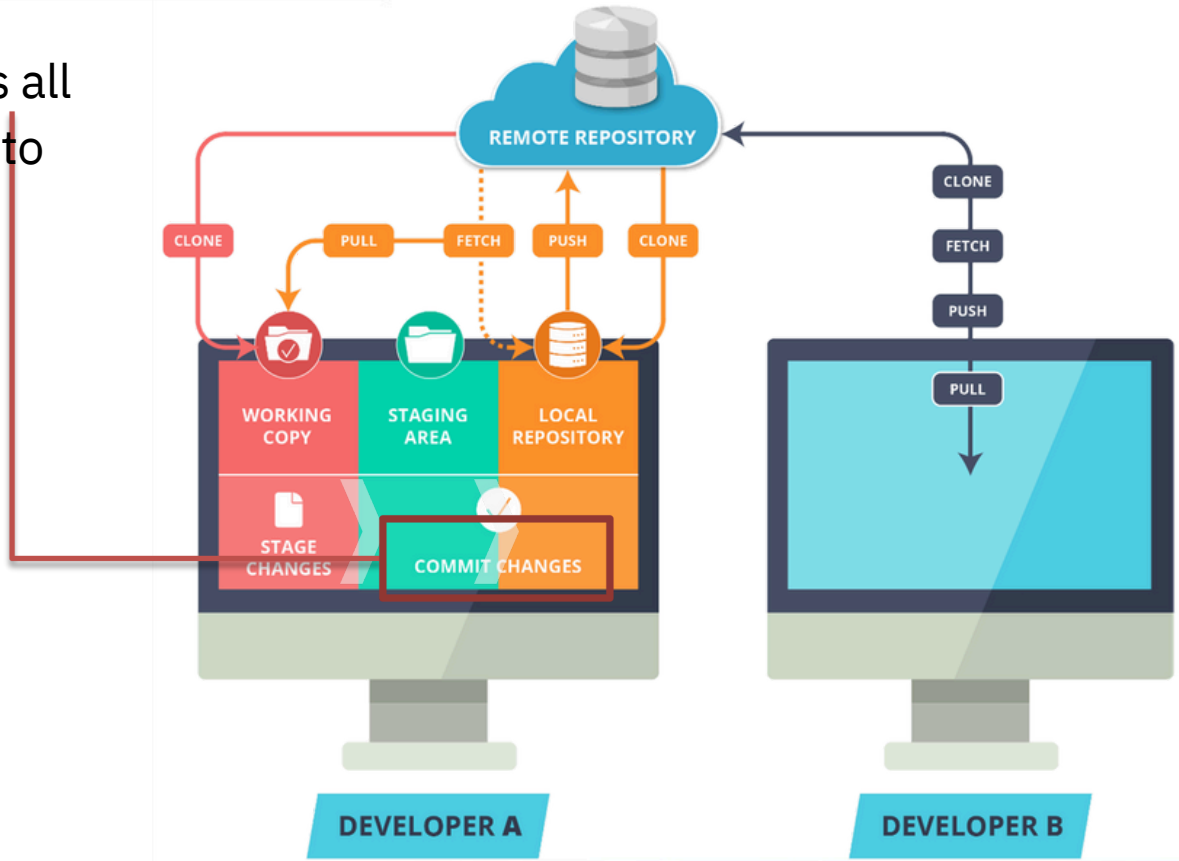
- Clone command creates a copy of an existing Remote Repository inside the Local Repository.



Git Workflow

- Commit command commits all the files in the staging area to

  the local repository.

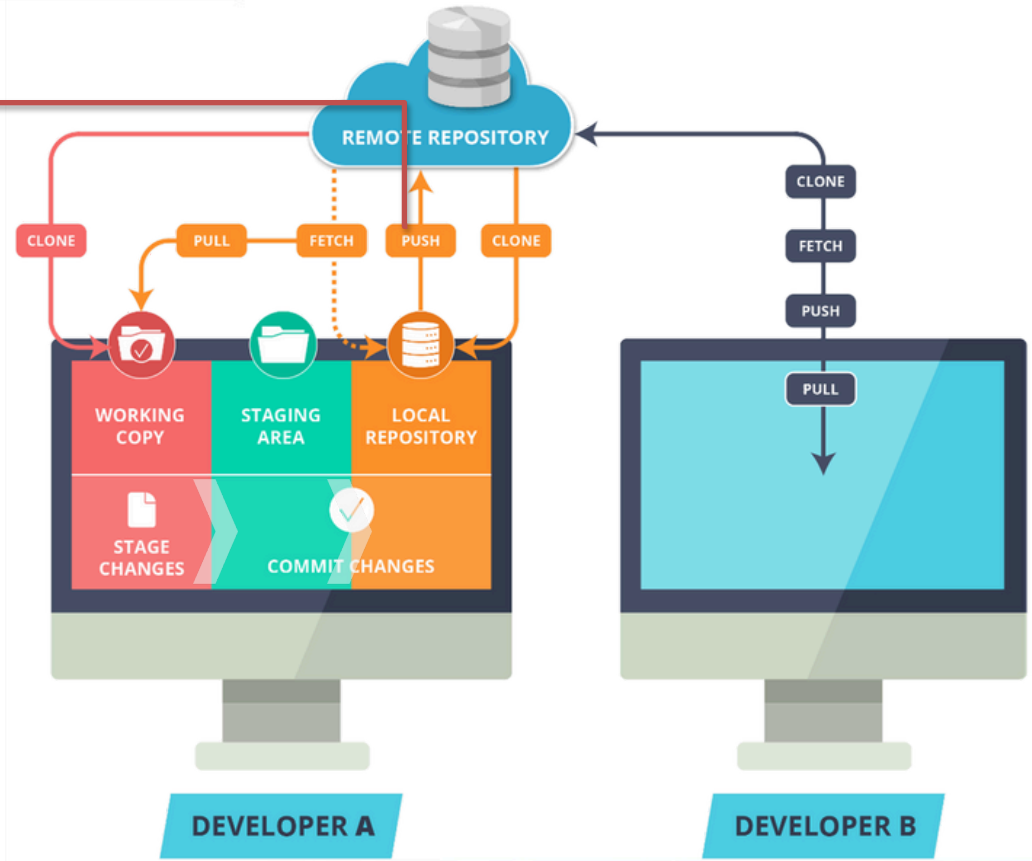


Git Workflow

- Push command pushes all the changes made in the Local

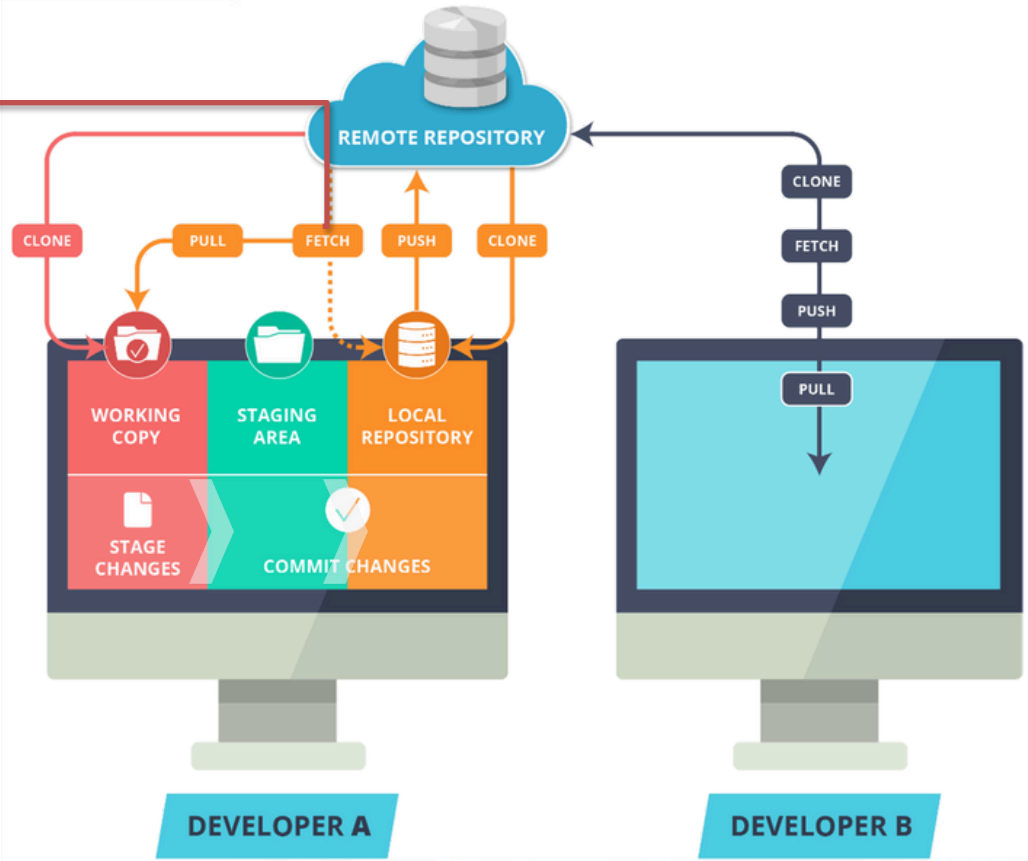
  Repository to the Remote

  Repository



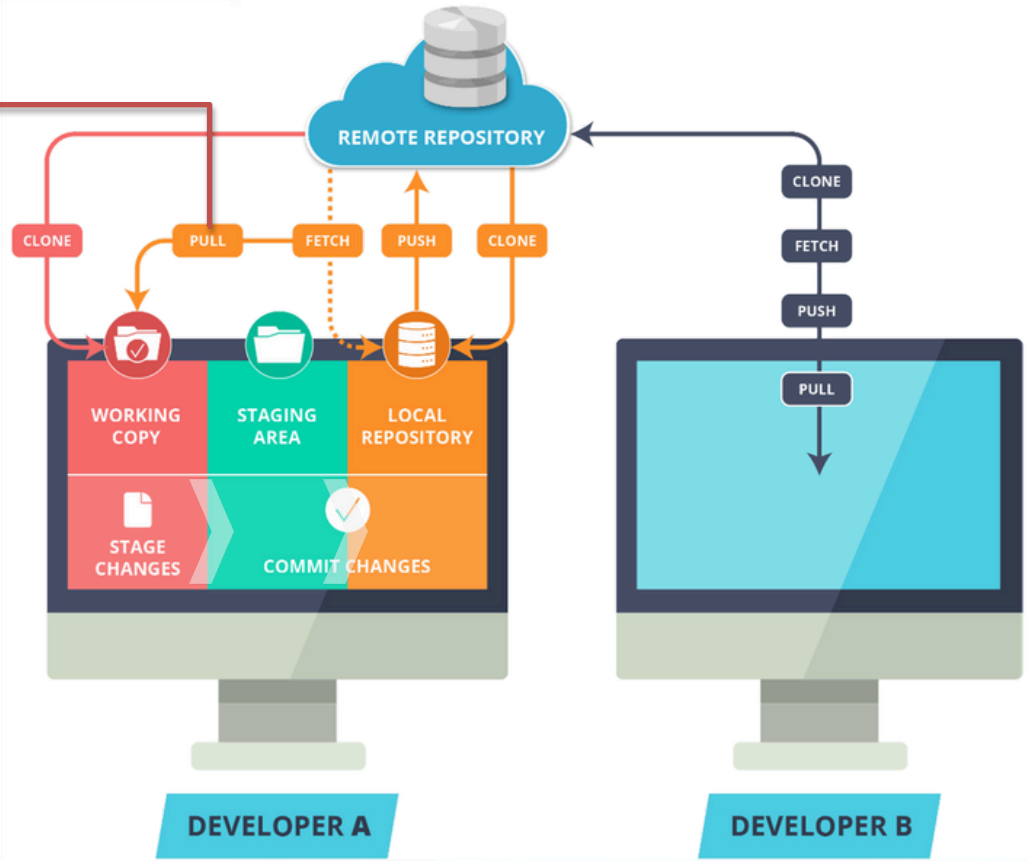
Git Workflow

- Fetch command collects the changes made in the Remote repository and copies them to the Local Repository. This command doesn't affect our Workspace.



Git Workflow

- Pull like Fetch, gets all the changes from the remote repository and copies them to the Local Repository
- Pull merges those changes to the current working directory



Installing Git

- To install Git on your Linux Machine you can type in the following command in Terminal:

```
Syntax: sudo apt-get install git
```

- For Windows download the installer from http://git-scm.com

Setting up Git User

- Set up username for your repository

```
Syntax: git config --global user.name “username”
```

- Set up user-email for your repository

```
Syntax: git config --global user.email “useremail@example.com”
```

Initialize a Git Repository

- Select or create the Directory where you want to initialize Git
- **Initialize Git** in the Directory

```
Syntax: git init
```

Adding Files & Checking Status

- To **add** a file to the **staging area**

```
Syntax: git add <filename>
```

- To check the working tree **status**

```
Syntax: git status
```

Committing Changes

- To **commit** the **staged** files to you **local repository:**

```
Syntax: git commit
```

- The git **diff** command displays all the changes made to the **tracked** files

```
Syntax: git diff
```

Staging & Committing Multiple Files

- To **stage and commit** multiple files at once we use -a flag with the commit command
- Commit with -a flag automatically stages all the modified files and commits changes to the local repository

```
Syntax: git commit -a -m ‘message’
```

Staging & Committing Multiple Files

- The **git rm** command deletes the file from git repository as well as users system

```
Syntax: git rm <filename>
```

- To remove the file from git repository but not from the system **--cached** option

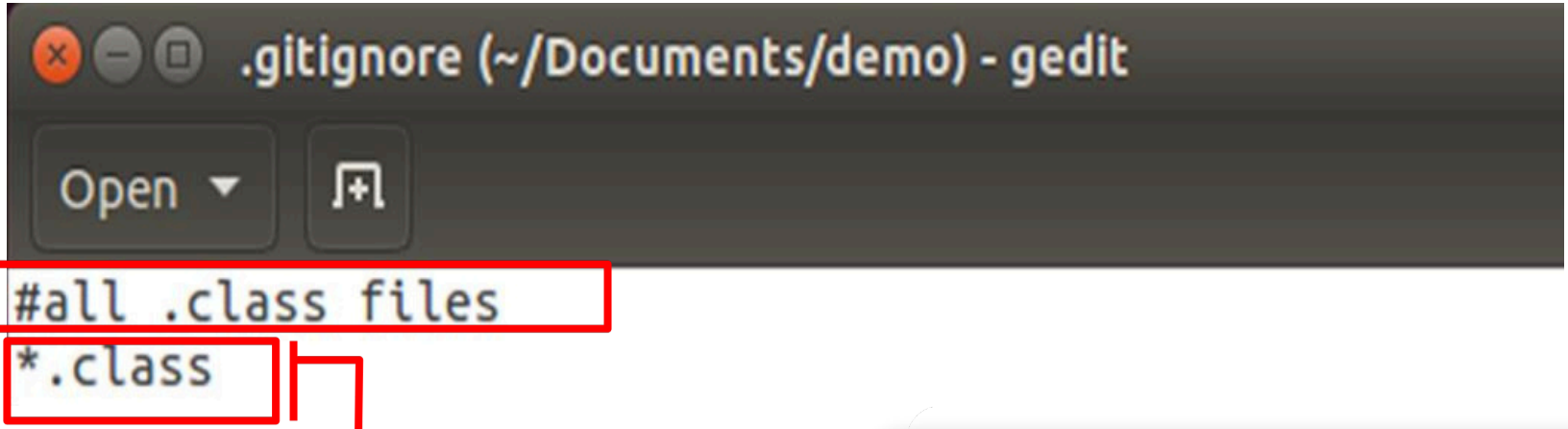
```
Syntax: git rm --cached <filename>
```

- An error shows up if you try to delete a staged file
- You can force remove a staged file by using **–f** flag

```
Syntax: git rm -f <filename>
```

Creating a Gitignore File

- You can create a **.gitignore** file and add all the untracked files you want Git to ignore



Comments can be made using #

You can add file names or ignore a specific type of file as shown in the example

Git Log

- The **git log** command shows all the commits so far on the current branch

```
Syntax: git log
```

- The **git log --oneline** command shows only one line for all the commits so far on the current branch

```
Syntax: git log --oneline
```

```
Syntax: git log –-oneline --decorate --graph
```

Git Tag

- Commit tags provide an alias for commitID

```
Syntax: git tag --a <annotation> --m <message>
```

- You can also view all the tags you have created

```
Syntax: git tag
```

Git Tag

- To give a commit in history, a tag

```
Syntax: git tag --a <annotation> <commit id> --m <message>
```

- Commit id can be obtained from git logs
- All these tags can be viewed in git

```
Syntax: git show <tag-name>
```

Commit Tag

- To give a commit in history, a tag

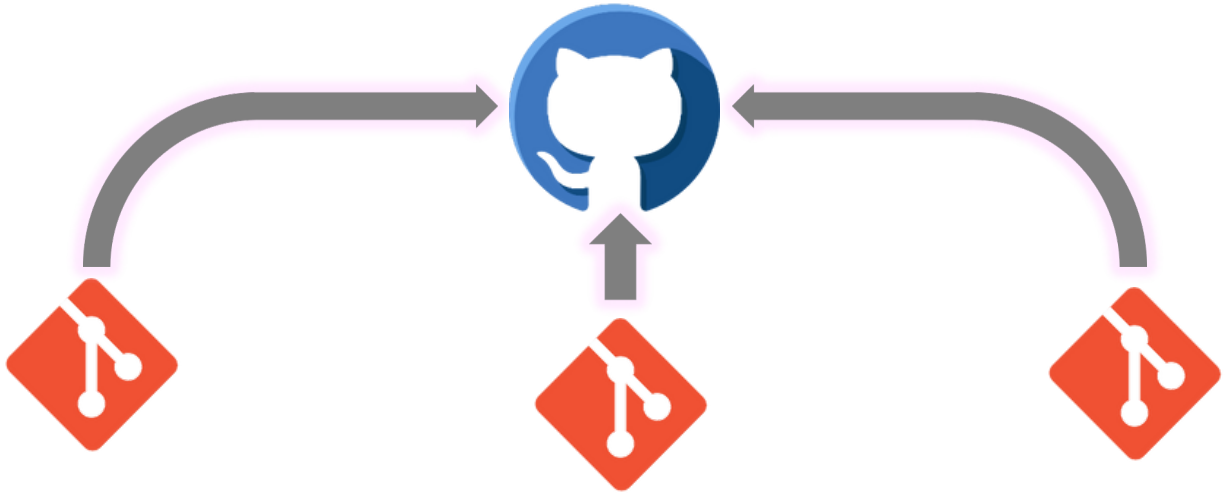
```
Syntax: git tag --a <annotation> <commit id> --m <message>
```

- Commit id can be obtained from git logs
- All these tags can be viewed in git

```
Syntax: git show <tag-name>
```

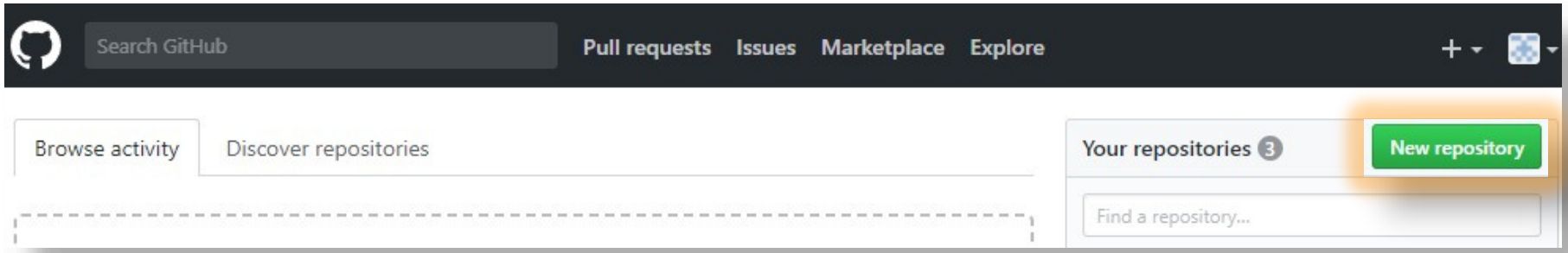
Remote Repository

Mostly the users work on a local repository. But in order to collaborate with other people, we use a remote repository. A remote repository is place where the users upload and share their commits with other collaborators.



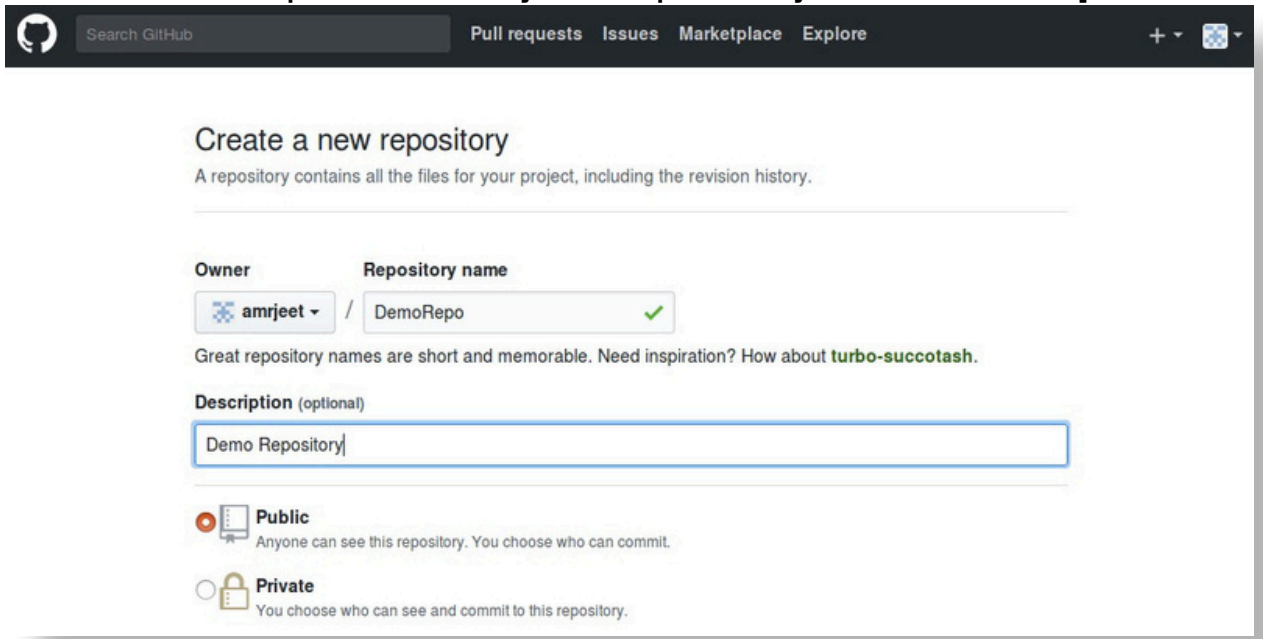
Creating a Remote Repository

- Sign-up at **github.com**
- Click on New repository to create a new repository



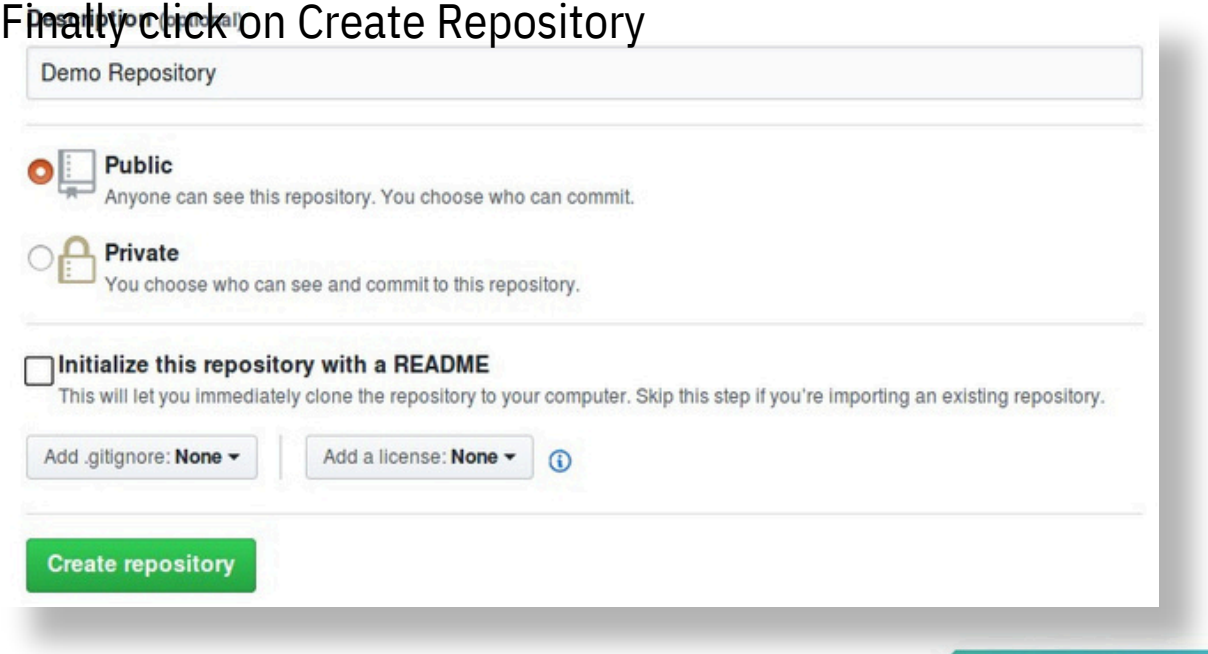
Creating a Remote Repository

- Under **Repository name**, give a name to your repository
- Give some Description about your repository under **Description** section.



Creating a Remote Repository

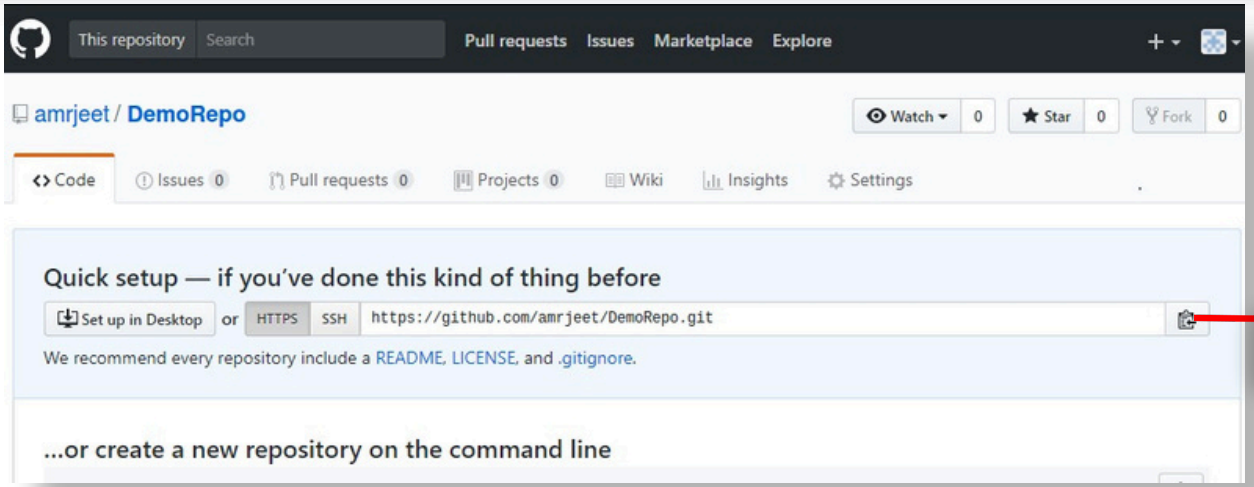
- For a free repository choose public
- For a private repository a monthly premium needs to be paid
- Finally click on Create Repository



Adding Remote Repository To Local Repository

- To add Remote repository to local use **git add remote** followed by remote link

```
Syntax: git add remote origin <remote link>
```



Adding Remote Repository To Local Repository

- To push **Local repository** to remote use **push** command

```
Syntax: git push origin master
```

•Origin is an alias for your remote

•Master is the name of the branch you are pushing from local to remote

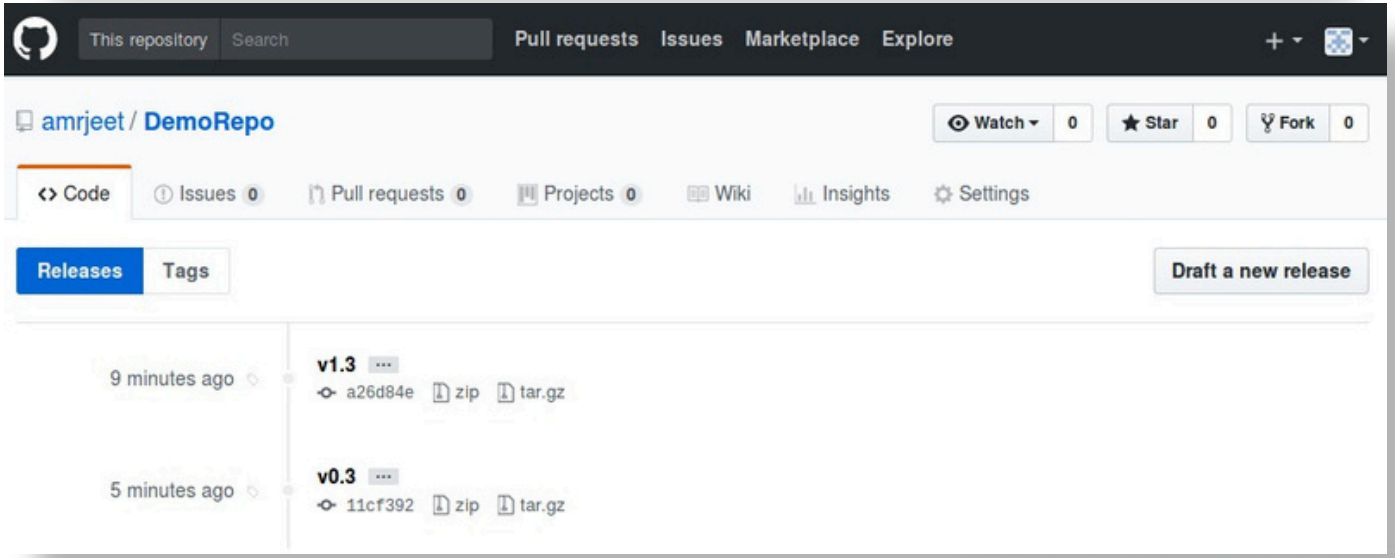
- To push other branches to remote use the following command

```
Syntax: git push –u origin <branch-name>
```

Pushing Tags to Remote Repository

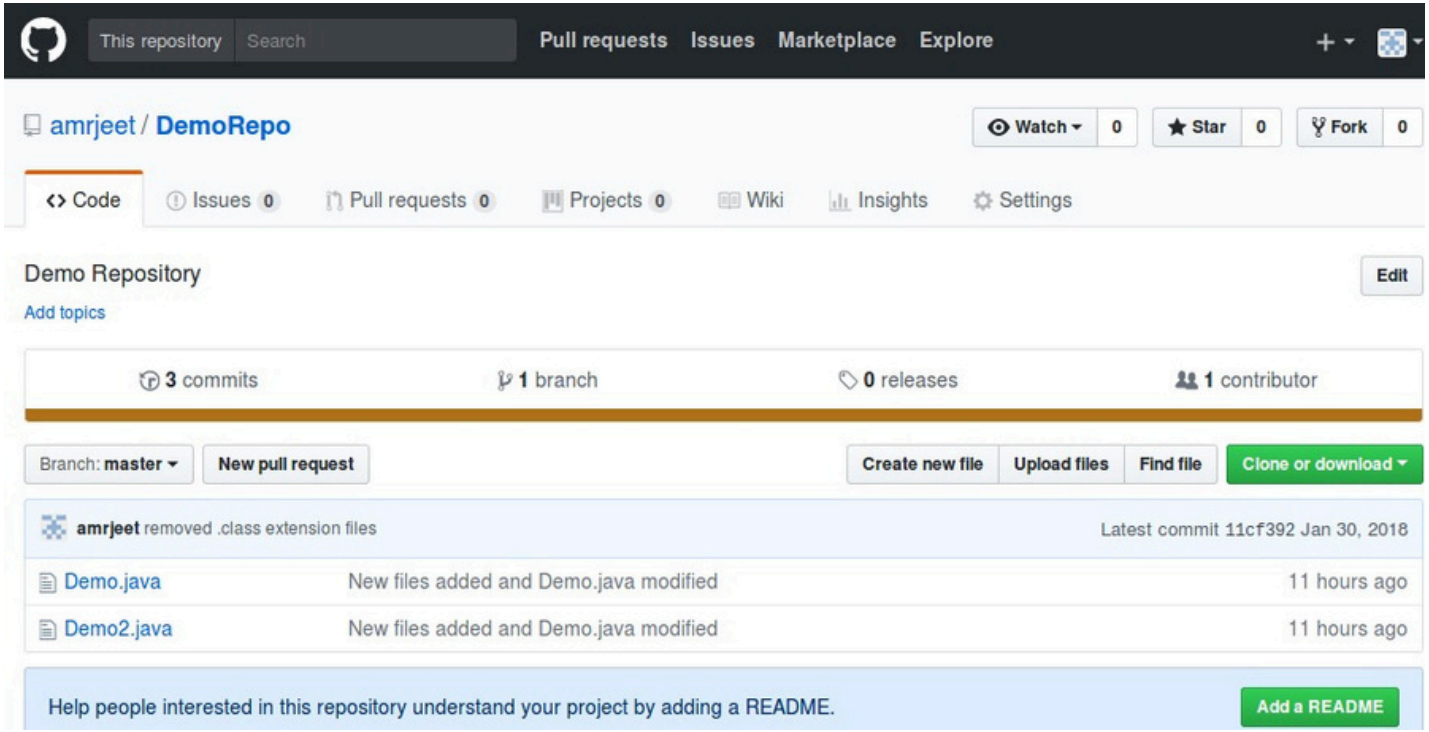
- Tags can be pushed, viewed and shared on Remote

```
Syntax: git push origin --tags
```



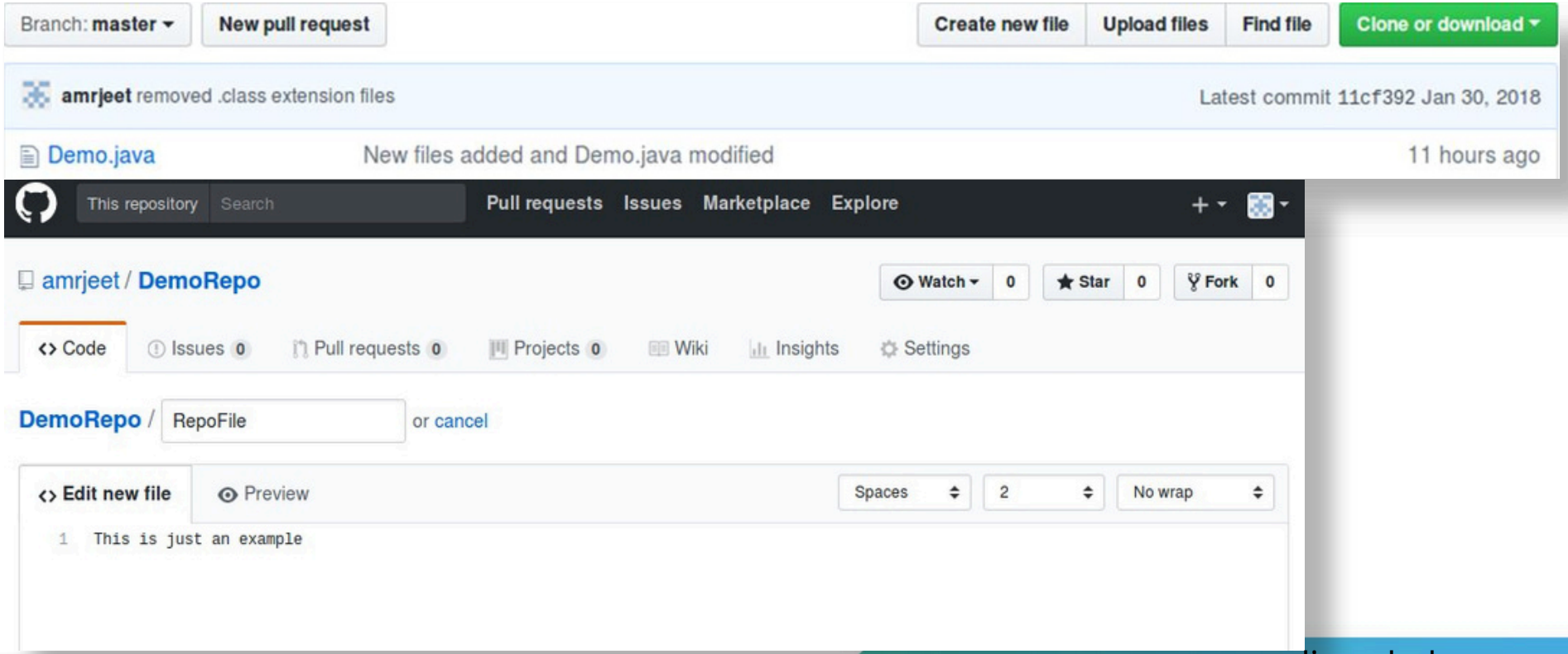
Push Local Repository To Remote

- The changes can be seen in Remote repository



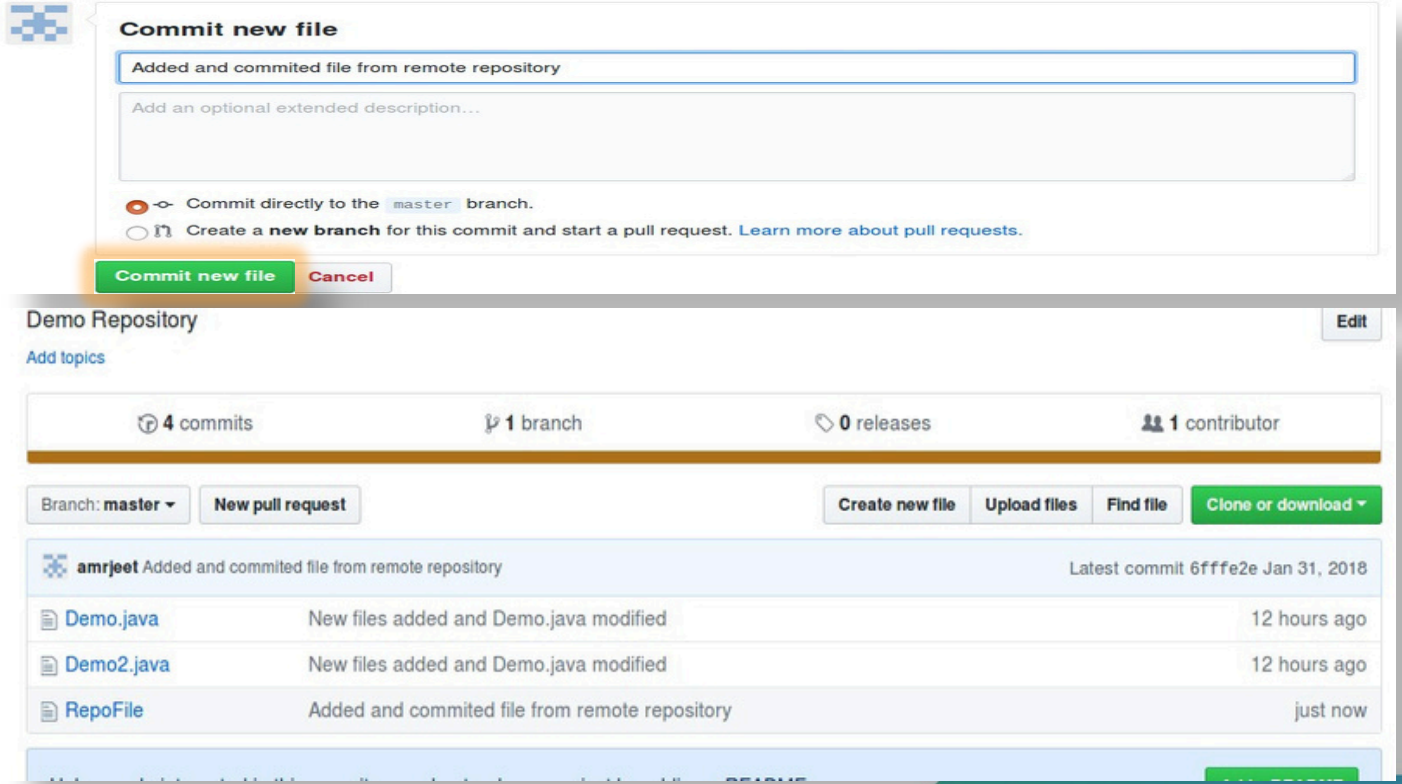
Working on Remote

- Files can be created and edited on remote



Working on Remote

- These files can then be committed on the remote



Remote List

- To list all the remotes attached to your Local repository

```
Syntax: git remote -v
```

Git Fetch

- **Fetch** command copies the changes from remote to local repository

```
Syntax: git fetch origin
```

- Fetch **does not** affect the present working directory

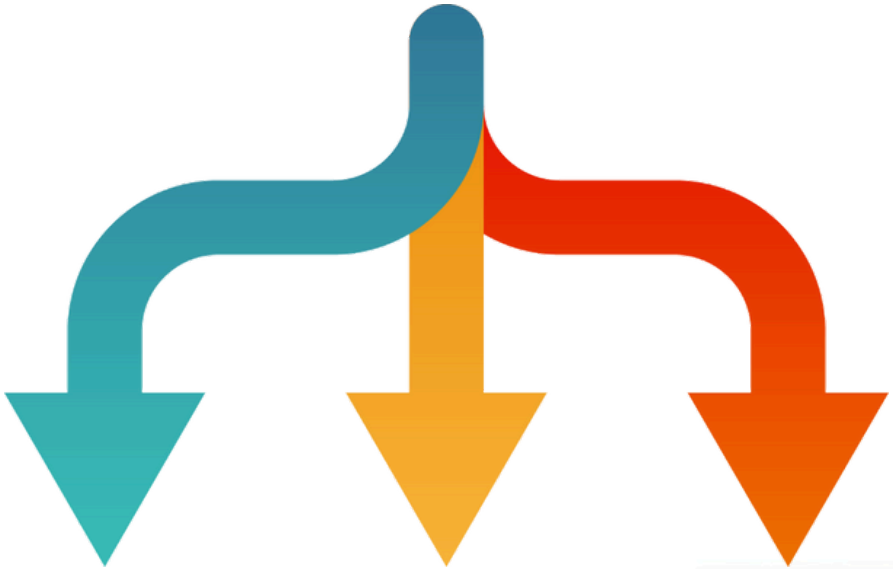
Git Pull

- Pull copies all the changes from remote to local repository
- It then merges the changes with the present working directory

```
Syntax: git pull origin
```

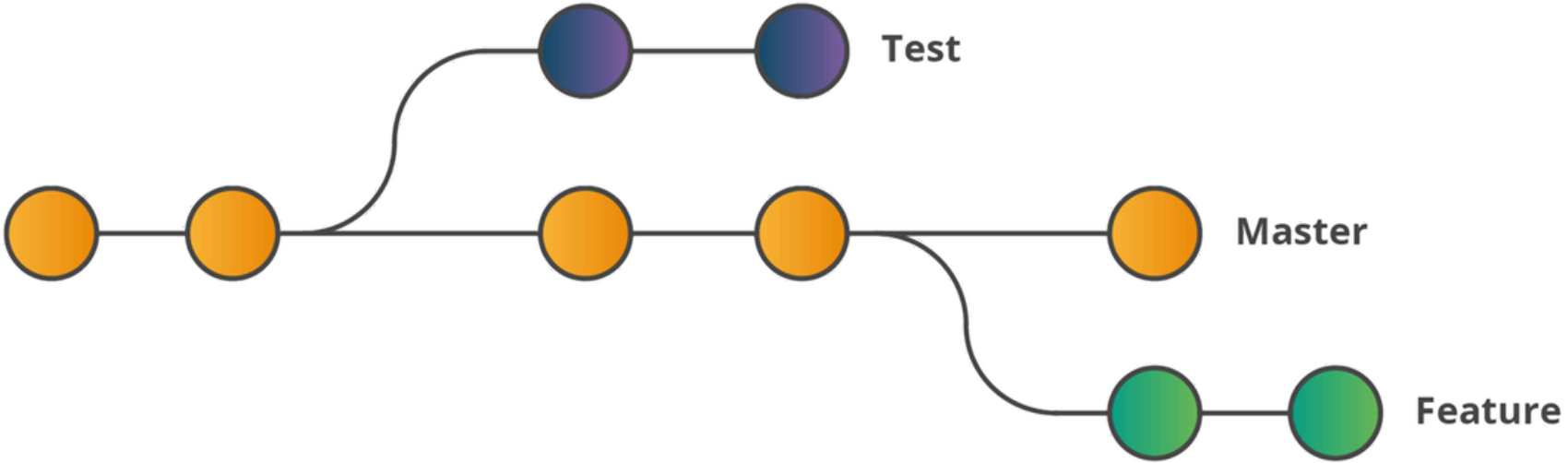
Git Branches

A project in its development could take multiple different paths to achieve its goal. Branching helps us take these different directions, test them out and in the end achieve the required goal.



Branching in Git

- Branching is an integral part of any Version Control(VC) System
- Unlike other VC's Git **does not** create a copy of existing files for new branch
- It **points** to snapshot of the changes you have made in the system



Creating a Branch

- To create a new branch from your current branch

```
Syntax: git branch <branchname>
```

- You can then switch to this newly created branch

```
Syntax: git checkout <branchname>
```

Creating a Branch

- Creating and switching to a new branch can be done with using **-b flag**

```
Syntax: git checkout -b <branchname>
```

- **Branch** command lists all the branches and also points to the current working branch

```
Syntax: git branch
```

Merging in Git

- Merging integrates the changes made in different branches into one single branch

Merging in Git

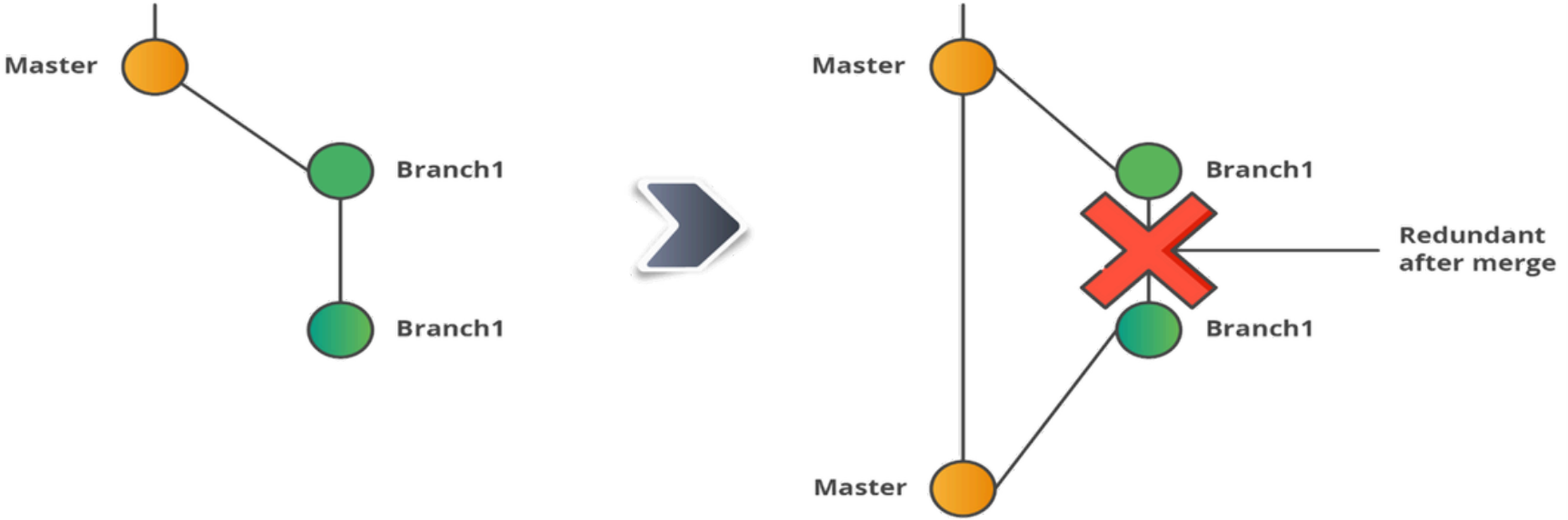
- Different modified branches can be **merged** together using merge

```
Syntax: git merge <branchname>
```

- The branch mentioned is merged into the current branch

Merging in Git

- All the changes made in **Branch1** after merging are available in the Merged branch(Master)
- Branch1 becomes redundant after merging, hence it can be deleted



Deleting a Branch

- **Merged** branches can be deleted using -**d** flag

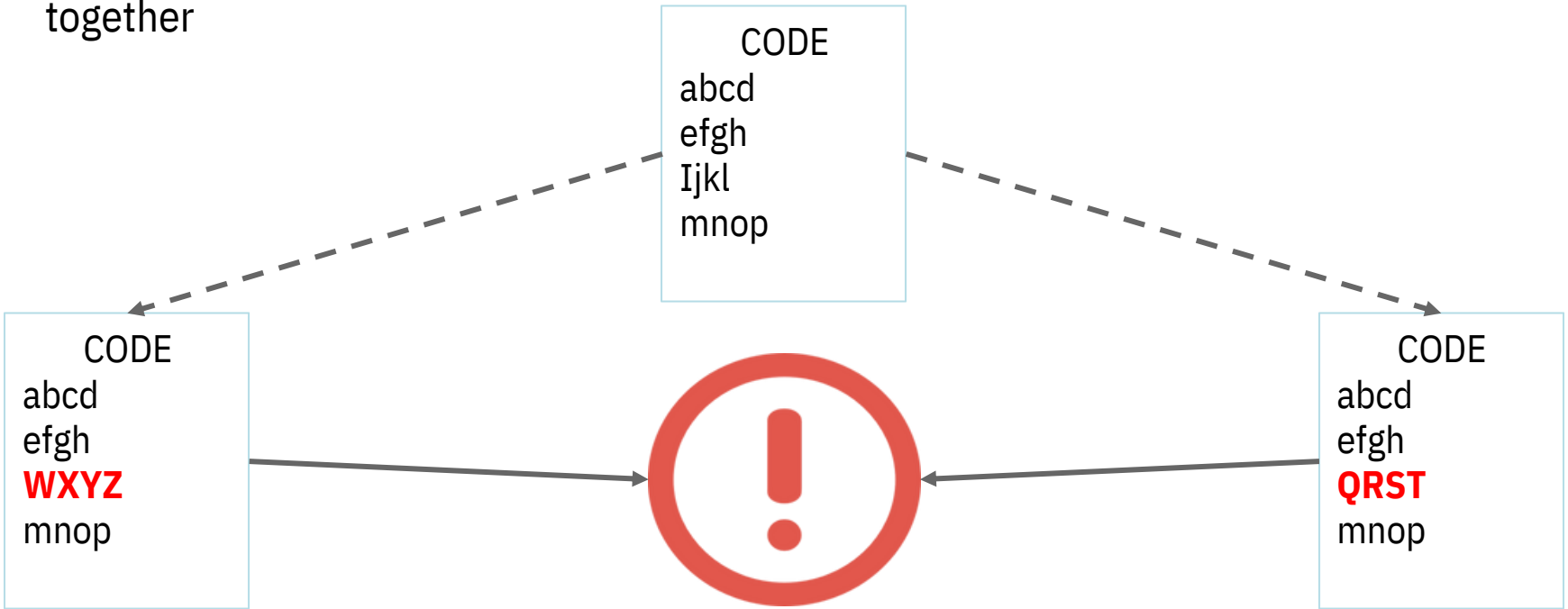
```
Syntax: git branch -d <branchname>
```

- **Unmerged** branches can be deleted using **-D** flag

```
Syntax: git branch -D <branchname>
```

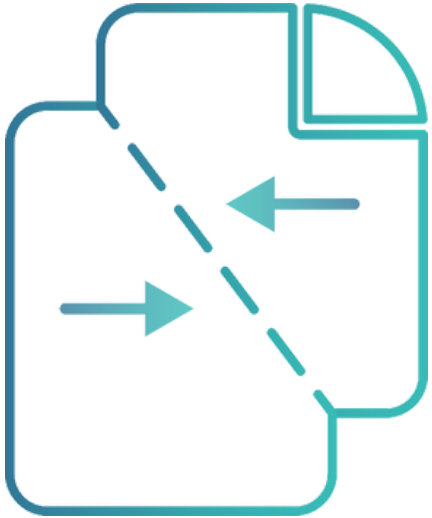
Merge Conflicts

- Merge conflicts arise when two files having same content modified are merged
- Merge conflicts can occur on merging branches or when merging forked history together

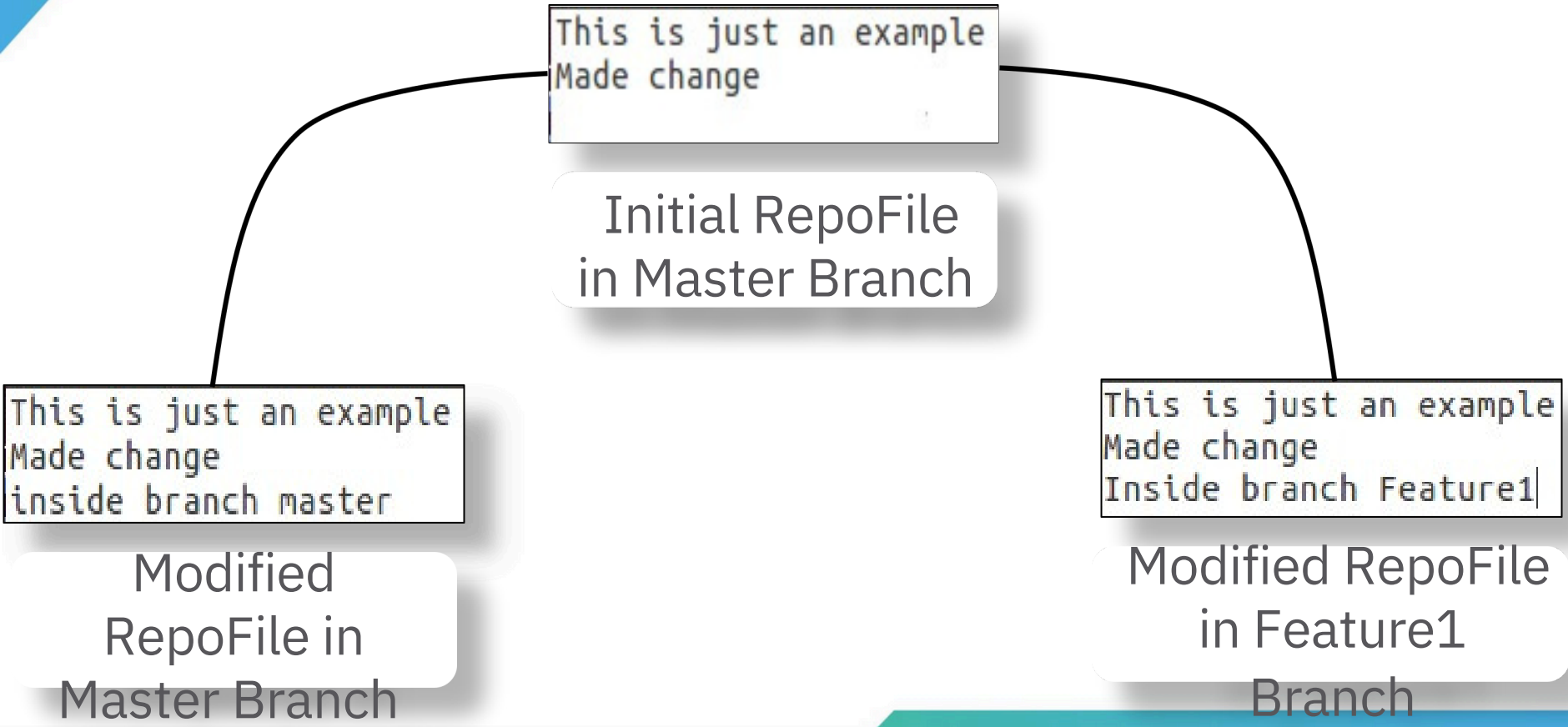


Resolving Merge Conflicts

- Merge Conflicts are resolved manually by users
- Git provides different Merge-Tools to compare and choose the required changes
- User can also use third party Merge-Tools with Git



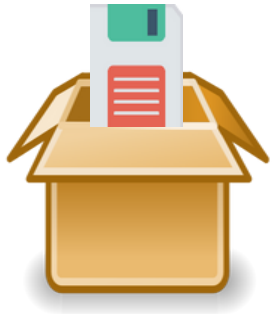
Merge Conflicts



Resolving Merge Conflicts

- Merge Conflict arises on merging branches
- Set a default merge tool before resolving conflict
- Commit the resolved changes
- Merge the branch again

Git Stashing is a way of creating a checkpoint for non-committed changes. It saves all the changes to a temporary location so the user can perform other tasks such as switching branches, reverting etc. These changes can then be reapplied anywhere.



- To Create a stash of your current working directory

```
Syntax: git stash save ‘message’
```

- To list all the saved stashes

```
Syntax: git stash list
```

- Stash list uses a stack structure to save the list

Applying a Stash

- Saved stashes can be applied at anytime on any branch

```
Syntax: git stash apply <stash id>
```

- After applying a stash it can still be accessed elsewhere because it remains in the stash

- Pop command can be used to apply the most recent stash and removing it from the stash stack

```
Syntax: git stash pop
```

Deleting Stashes

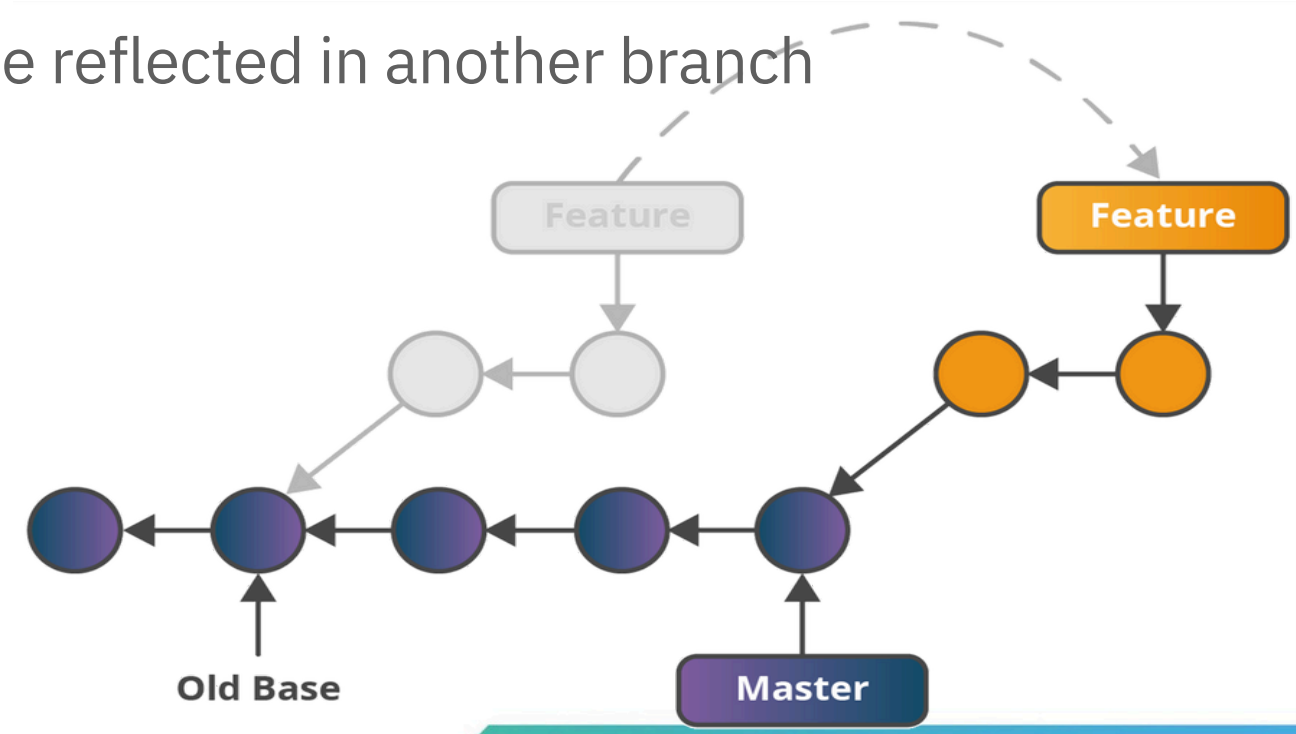
- Stashes can be deleted from the stash list

```
Syntax: git stash drop <stack id>
```

- The entire stack can also be deleted using one command

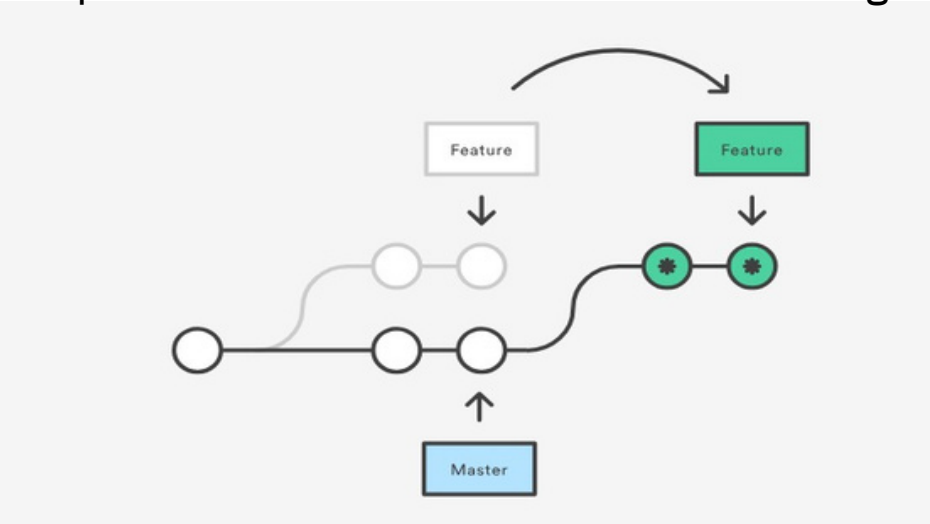
```
Syntax: git stash clear
```

Git rebasing is used, when changes made in one branch needs to be reflected in another branch



Rebase

- Rebasing is the process of moving or combining a sequence of commits to a new base commit
- Rebasing is most useful and easily visualized in the context of a feature branching workflow
- The general process can be visualized as the following:



Rebase

- From a content perspective, rebasing is changing the base of your branch from one commit to another making it appear as if you'd created your branch from a different commit
- Internally, Git accomplishes this by creating new commits and applying them to the
- specified base
  The primary reason for rebasing is to maintain a linear project history
- For example, consider a situation where the master branch has progressed since you started working on a feature branch
  You want to get the latest updates to the master branch in your feature branch, but you want to keep your branch's history clean so it appears as if you've been working off the latest master branch
- This gives the later benefit of a clean merge of your feature branch back into the master branch

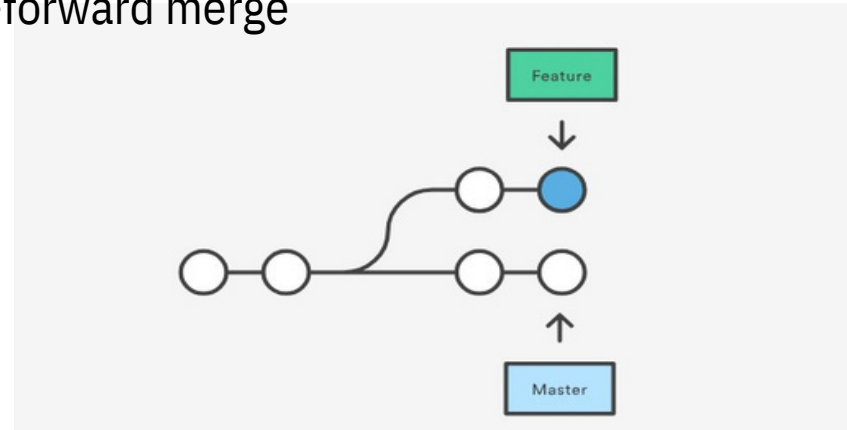
Maintain Clean History

- The benefits of having a clean history become tangible when performing Git operations to investigate the introduction of a regression. A more real-world scenario would be:
  - A bug is identified in the master branch. A feature that was working successfully is now broken.
  - A developer examines the history of the master branch using git log because of the "clean history" the developer is quickly able to reason about the history of the project.
  - The developer can not identify when the bug was introduced using git log so the developer executes a git bisect.
  - Because the git history is clean, git bisect has a refined set of commits to compare when looking for the regression. The developer quickly finds the commit that introduced the bug and is able to act accordingly.

Two Options for Integrating Your Code

- You have two options for integrating your feature into the master branch:
  - merging directly or
  - rebasing and then merging
- The former option results in a 3-way merge and a merge commit, while the latter results in a fast-forward merge and a perfectly linear history

The following diagram demonstrates how rebasing onto the master branch facilitates a fast-forward merge



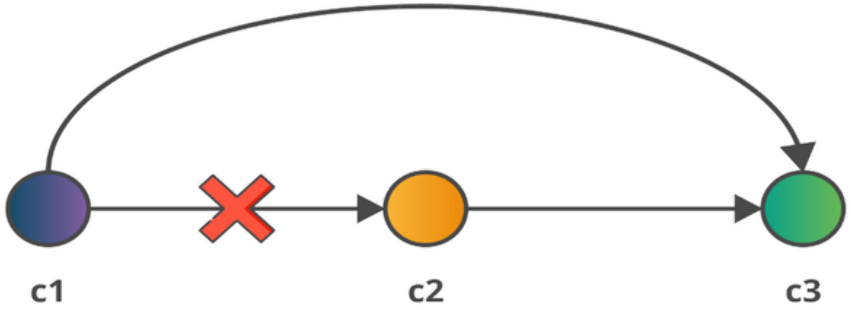
- To Rebase a branch

```
Syntax: git rebase <rebase branch>
```

- Revert or **undo** the changes made in the previous commit
- New commit is created without the changes made in the other commit

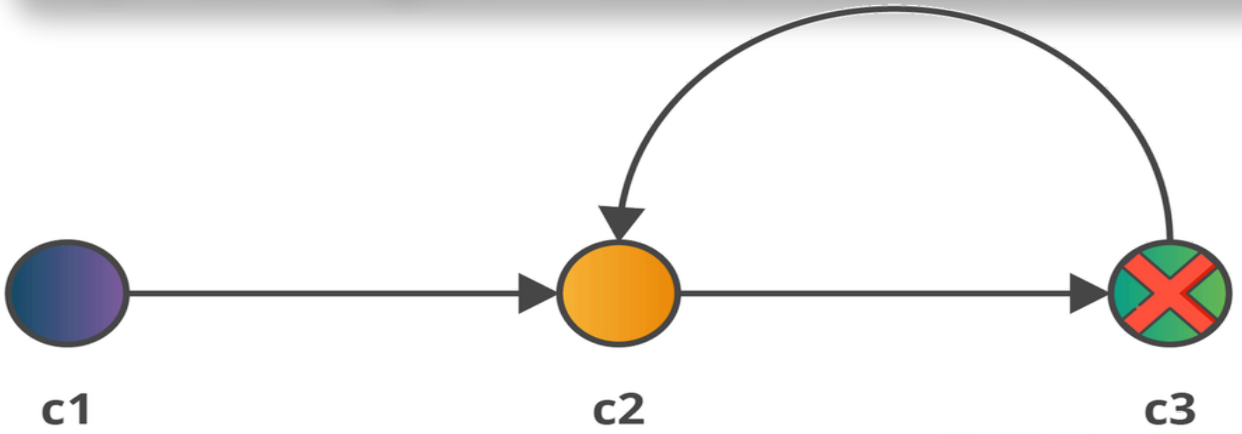
```
Syntax: git revert <commit id>
```

- Old commit still resides in the history



- Reset command can be used to undo changes at different levels
- Modifiers like --hard, --soft and --mixed can be used to decide the degree to which to reset

```
Syntax: git reset <modifier> <commit id>
```




THANK YOU
45%