

SOURCE CONTROL

Created by:
Deepanshu Pasrija

Source control

Source control (or version control) is the practice of tracking and managing changes to code.

Source control management (SCM) systems provide a running history of code development and help to resolve conflicts when merging contributions from multiple sources.

Need

- Wikipedia defines Source control as "Revision control is the management of multiple revisions of the same unit of information."
- Source control allows distributed work in teams of any size, at different locations, while avoiding conflicts in source code changes.
- Collaboration
- Storing Versions (Properly)
- Restoring Previous Versions
- Understanding What Happened
- Backup

Basic linux commands

- Apt-get install
- Cd
- Ls
- Cd ..
- Mv
- Cp -pr
- Mkdir

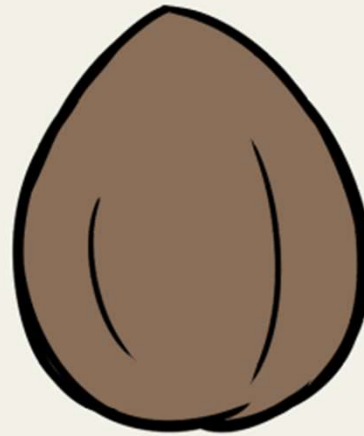
Git terminologies

- Repos, Staging, Commits
- Branch
- Feature Branch Workflow
- Gitflow Workflow
- HEAD
- Master
- Pull Request
- Repository
- Tag

- Getting Down to the “Source” of Source Control

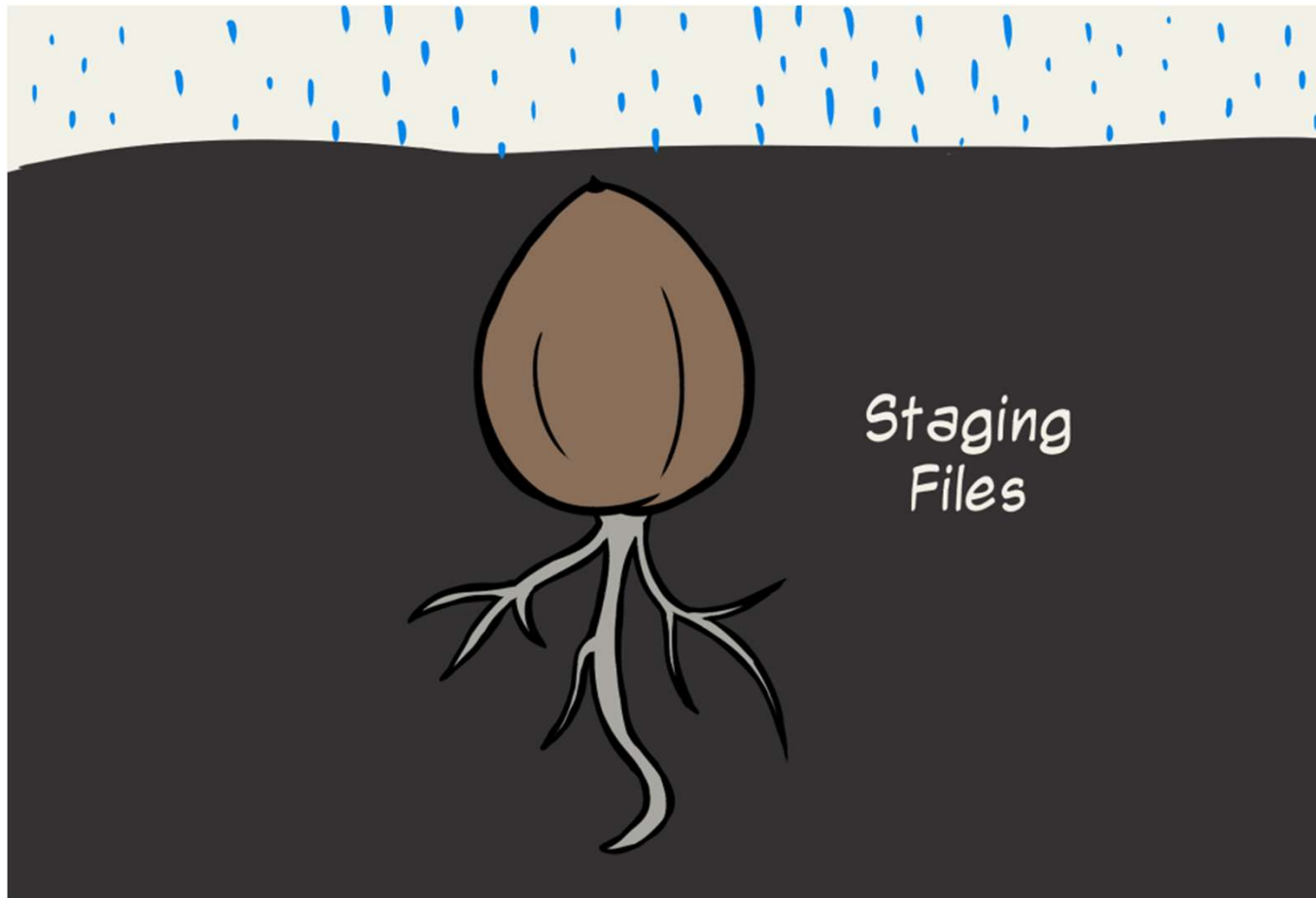
- Pinpointing the origin.
- Branches, commits, versions, let’s go through the steps of how we would implement a modern source control system into our workflow with an example. The explanations will be general, and we will be using a tree for an analogous comparison.

Initialized Repository

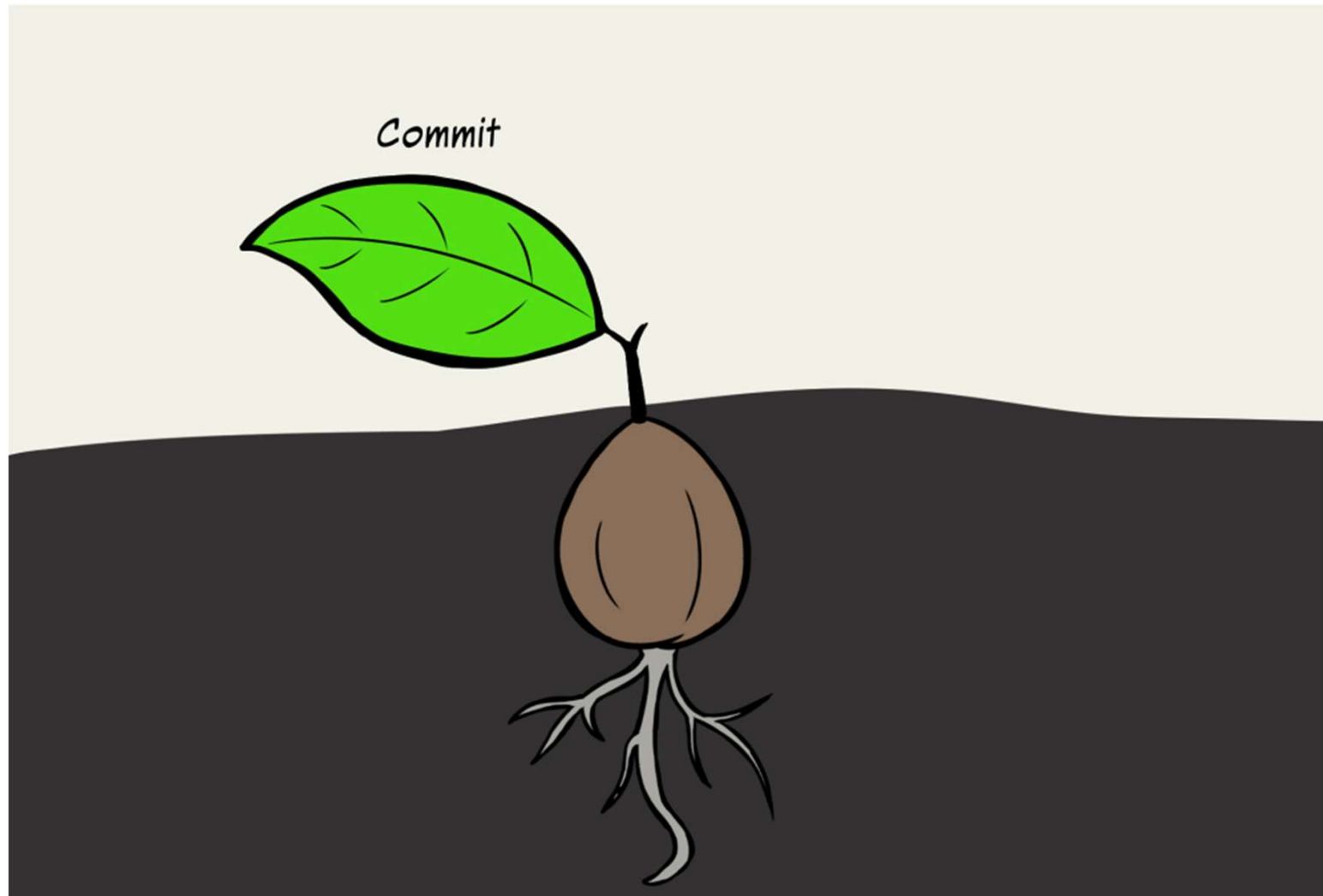


*Initialized
Repository*

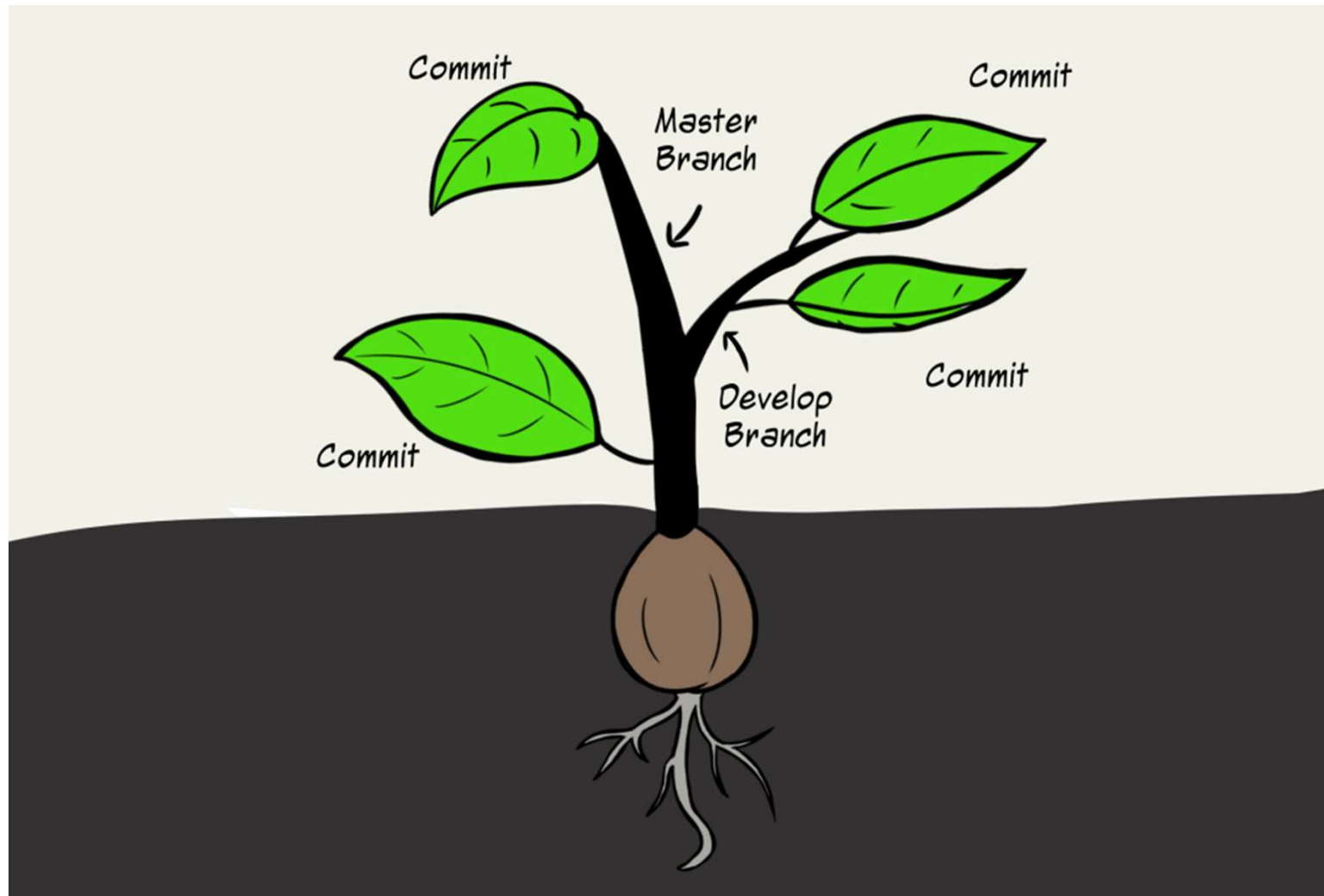
Staging Files



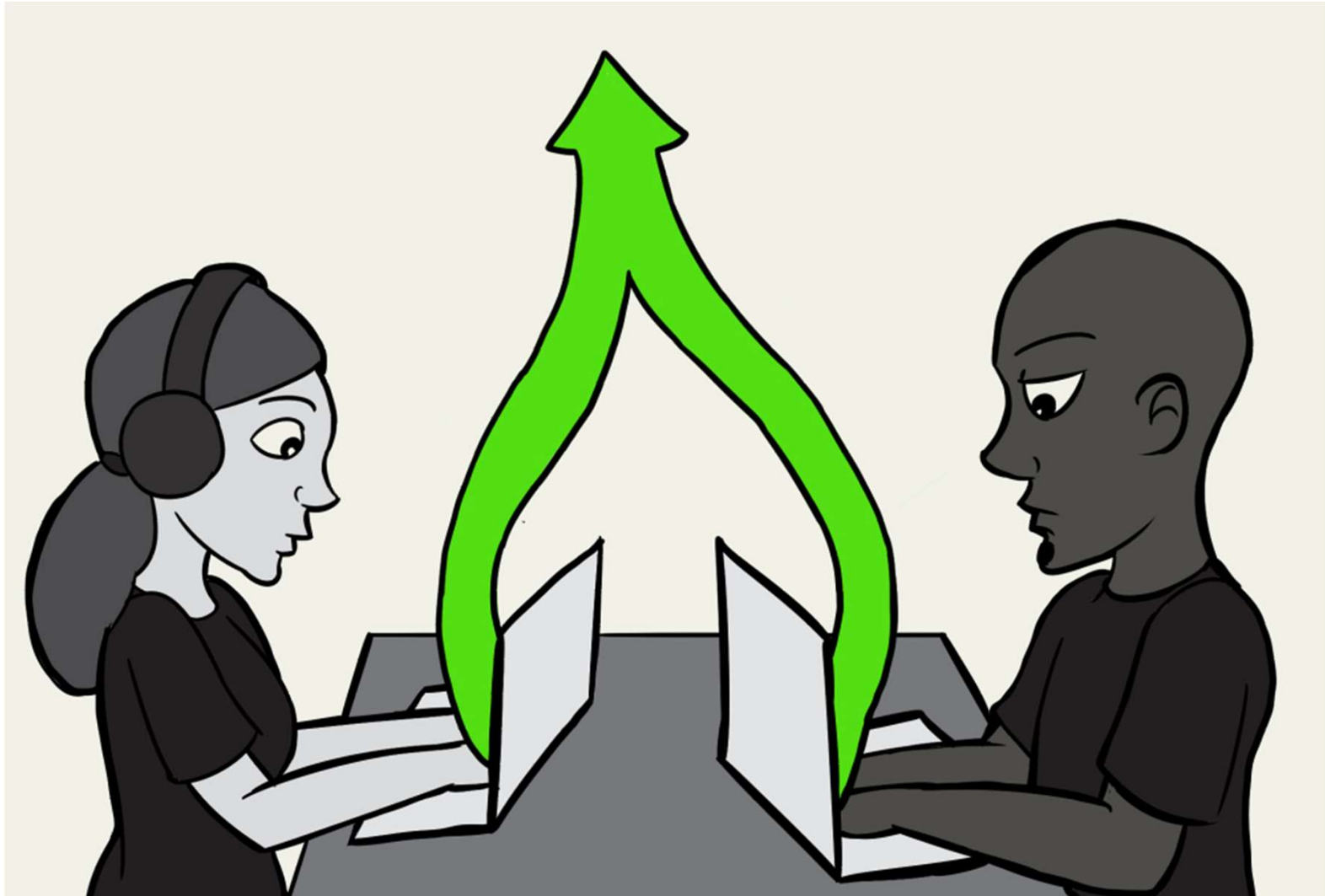
Commit



Develop Branch



Work in Sync



Git Installation

- apt-get update
- Install Git
- apt-get install git-core
- You may be asked to confirm the download and installation; simply enter y to confirm. It's that simple, git should be installed and ready to use!
- git --version
- git config --global user.name "testuser"
- git config --global user.email "testuser@example.com"
- cat ~/.gitconfig or git config --list

Centralized Workflow

- Initialize the central repository
- Hosted central repositories
- Clone the central repository
- Make changes and commit
- Push new commits to central repository
- Managing conflicts

EXAMPLE

- John works on his feature
- Mary works on her feature
- John publishes his feature - git push origin master
- Mary tries to publish her feature
- Mary rebases on top of John's commit(s) - git pull --rebase origin master
- Mary resolves a merge conflict
- Mary successfully publishes her feature

Error:

```
error: failed to push some refs to '/path/to/repo.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Merge the remote changes (e.g. 'git pull')
hint: before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Guidelines

- Short-lived branches - The longer a branch lives separate from the production branch, the higher the risk for merge conflicts and deployment challenges
- Minimize and simplify reverts-It's important to have a workflow that helps proactively prevent merges that will have to be reverted.
- Match a release schedule - A workflow should complement your business's software development release cycle.

How it works




- Start with the master branch
- `git checkout master`
- `git fetch origin`
- `git reset --hard origin/master`
- Create a new-branch
- `git checkout -b new-feature`
- Update, add, commit, and push changes
- `git status`
- `git add <some-file>`
- `git commit`
- Push feature branch to remote
- `git push -u origin new-feature`
- Resolve feedback
- Merge your pull request

Create Project

- Go into the directory containing the project.
- Type `git init`.
- Type `git add` to add all of the relevant files.
- You'll probably want to create a `.gitignore` file right away, to indicate all of the files you don't want to track. Use `git add .gitignore`, too.
- Type `git commit`.
- `git remote add origin`
`git@github.com:username/new_repo`
- `$ git push -u origin master`


- To create a new repository
- In the upper right corner, next to your avatar or identicon, click and then select New repository.
- Name your repository hello-world.


Owner **Repository name**

PUBLIC   **hubot** / 


Great repository names are short and memorable. Need inspiration? How about **petulant-shame**.

Description (optional)

☒  **Public**
Anyone can see this repository. You choose who can commit.

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

☒ **Initialize this repository with a README**
This will allow you to `git clone` the repository immediately. Skip this step if you have already run `git init` locally.

| 

Create repository

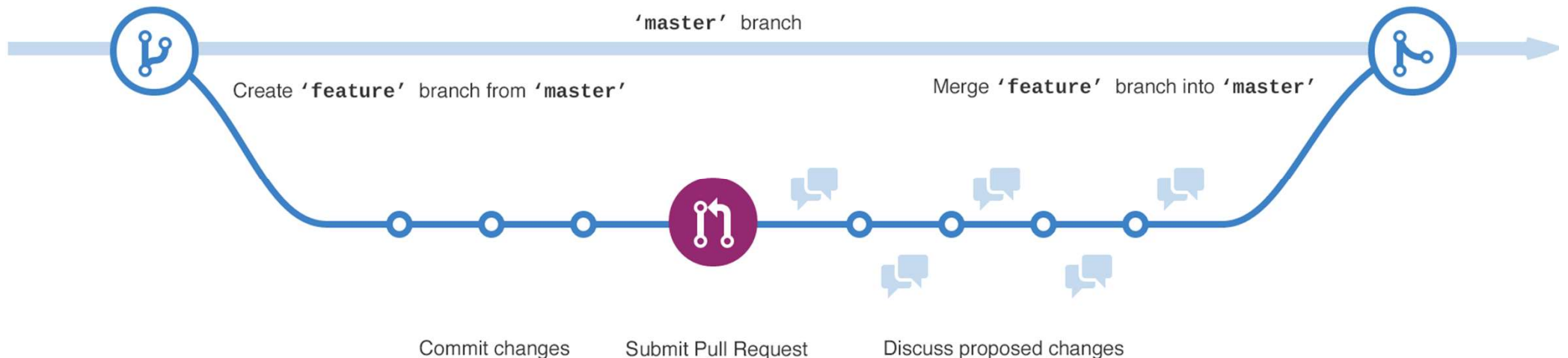
Step 2. Create a Branch

Go to your new repository hello-world.

Click the drop down at the top of the file list that says branch: master.

Type a branch name, readme-edits, into the new branch text box.

Select the blue Create branch box or hit “Enter” on your keyboard.



Make and commit changes

- Click the README.md file.
- Click the pencil icon in the upper right corner of the file view to edit.
- In the editor, write a bit about yourself.
- Write a commit message that describes your changes.
- Click Commit changes button.

Open a Pull Request

- Click the Pull Request tab, then from the Pull Request page, click the green New pull request button.
- In the Example Comparisons box, select the branch you made, readme-edits, to compare with master (the original).
- Look over your changes in the diffs on the Compare page, make sure they're what you want to submit.
- When you're satisfied that these are the changes you want to submit, click the big green Create Pull Request button.
- Give your pull request a title and write a brief description of your changes. pr-form
- When you're done with your message, click Create pull request!

Step 5. Merge your Pull Request

- In this final step, it's time to bring your changes together – merging your readme-edits branch into the master branch.
- Click the green Merge pull request button to merge the changes into master.
- Click Confirm merge.
- Go ahead and delete the branch, since its changes have been incorporated, with the Delete branch button in the purple box.
-

•Here's what you accomplished

- Create and use a repository
- Started and managed a new branch
- Changed a file and committed those changes to GitHub as commits
- Opened and merged a Pull Request

HISTORY

- `Git log --full-diff file.txt`
- `git commit --amend --no-edit=` to changelast commit message
- `git rebase -i HEAD~3`
- Stopped at f7f3f6d... changed my name a bit
- You can amend the commit now, with
 - `git commit --amend`
- Once you're satisfied with your changes, run
 - `git rebase --continue`

Branching

- `git checkout -b iss53`
- Switched to a new branch "iss53"
- or
- `git branch iss53`
- `$ git checkout iss53`
- `Vi file.txt`
- `git commit -a -m 'added a new footer [issue 53]'`

DELETE A BRANCH

- `git branch -d hotfix`
- Deleted branch hotfix (3a0874c).
- `git checkout iss53`

BRANCHING AND MERGING

- `git checkout -b hotfix`
- `vi index.html`
- `$ git commit -a -m 'fixed the broken email address'`
- `git checkout master`
- `$ git merge hotfix`

Working Dir, Staged Area

- Working Directory → files in your working directory.
- Staging Area (aka cache, index) → a temp area that git add is placed into.
- HEAD → A reference to a specific. Normally, it points to the last commit in local repository. (that is, after you did git commit).

LOCAL AND REMOTE

- A local branch is a branch that only you (the local user) can see. It exists only on your local machine.
- A remote tracking branch is a local copy of a remote branch. When myNewBranch is pushed to origin using the command above, a remote tracking branch named origin/myNewBranch is created on your machine.
- `git push -u origin myNewBranch`

•Branching

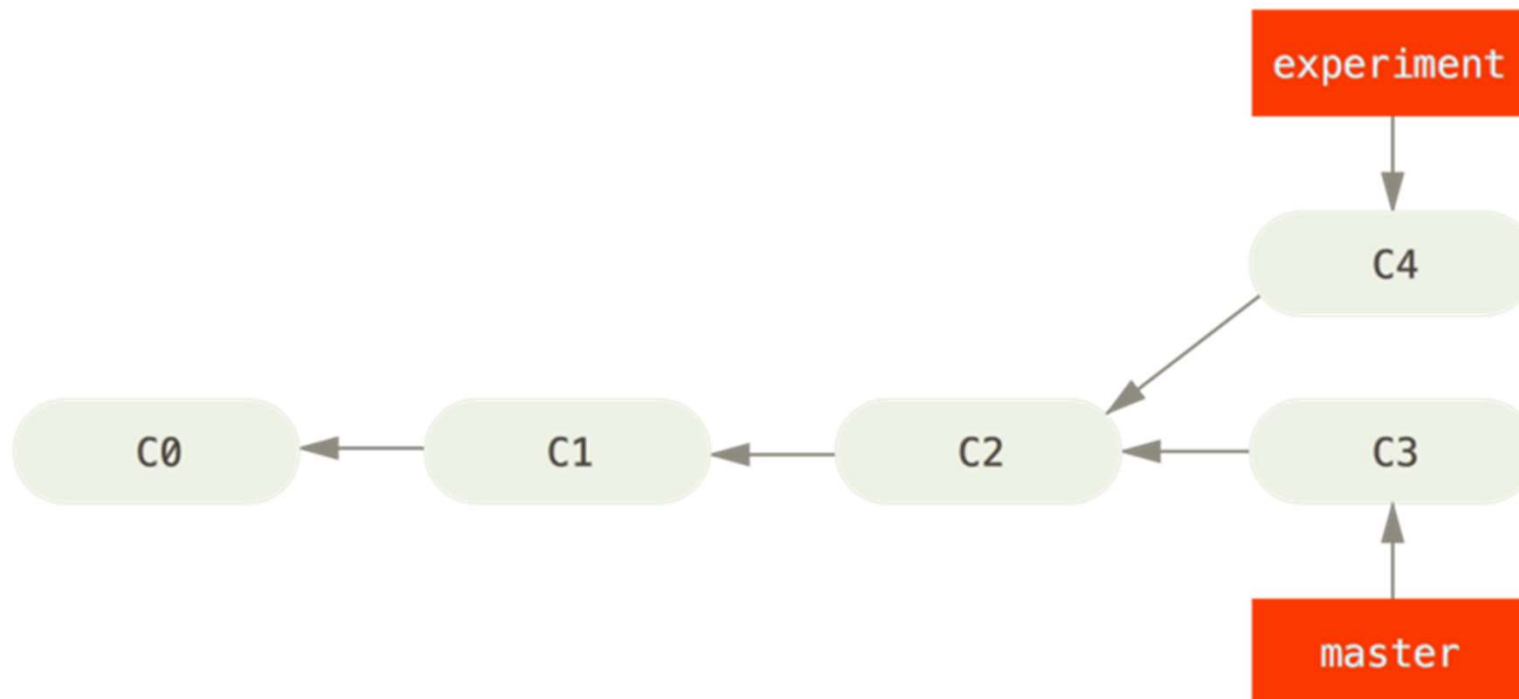
- Branching, in revision control and software configuration management, is the duplication of an object under revision control (such as a source code file or a directory tree) so that modifications can happen in parallel along both branches. Branches are also known as trees, streams or codelines

Merging

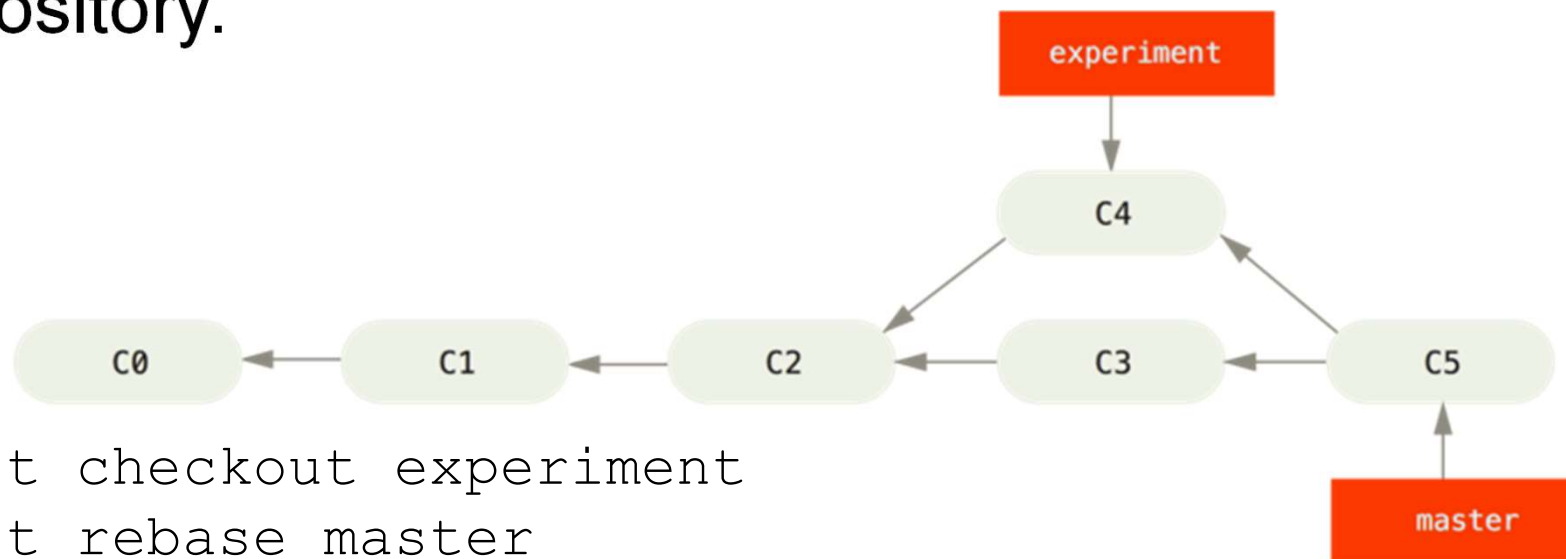
- In the most frequent use cases, git merge is used to combine two branches. The following examples in this document will focus on this branch merging pattern. In these scenarios, git merge takes two commit pointers, usually the branch tips, and will find a common base commit between them
- git checkout master
- Switched to branch 'master'
- \$ git merge iss53

Rebasing

- The Basic Rebase-The easiest way to integrate the branches, as we've already covered, is the merge command. It performs a three-way merge between the two latest branch snapshots (C3 and C4) and the most recent common ancestor of the two (C2), creating a new snapshot (and commit).



- another way: you can take the patch of the change that was introduced in C4 and reapply it on top of C3. In Git, this is called rebasing. With the rebase command, you can take all the changes that were committed on one branch and replay them on another one.
- Do not rebase commits that exist outside your repository.



```
$ git checkout experiment
```

```
$ git rebase master
```

First, rewinding head to replay your work on top of it...

Applying: added staged command

```
git checkout master
```

```
$ git merge experiment
```

REBASE ABORT

- You can run `git rebase --abort` to completely undo the rebase. Git will return you to your branch's state as it was before `git rebase` was called.
- You can run `git rebase --skip` to completely skip the commit. That means that none of the changes introduced by the problematic commit will be included. It is very rare that you would choose this option.
- You can fix the conflict.

REBASE ABORT

- You can run `git rebase --abort` to completely undo the rebase. Git will return you to your branch's state as it was before `git rebase` was called.
- You can run `git rebase --skip` to completely skip the commit. That means that none of the changes introduced by the problematic commit will be included. It is very rare that you would choose this option.
- You can fix the conflict.

REBASE CONFLICT

- Open Terminal.
- Navigate into the local Git repository that has the merge conflict.
- Generate a list of the files affected by the merge conflict.
- Open your text editor and navigate to the file that has merge conflicts.
- To see the beginning of the merge conflict in your file, search the file for the conflict marker <<<<<<. When you open the file in your text editor, you'll see the changes from the HEAD or base branch after the line <<<<<< HEAD. Next, you'll see =====, which divides your changes from the changes in the other branch, followed by >>>>>> BRANCH-NAME
- Decide if you want to keep only your branch's changes,
- `git add .`
- `git commit -m ""`

Steps to add a new file

- Add a file and go to directory
- `git status`
- `git add .`
- `git commit -m "Latest Commit"`
- `git pull`
- `git push`

Steps to add a New File in New Repo

- Add yourself in the collaborators of the new repo from settings -> Collaborators
- Take a clone of the repo using the URL in clone or download option of new Repo
- Add or change a file.
- Check git status to check your change.
- Type `git add .` Filename and then commit ur file using `git commit -m "Message"`
- Type `git pull`- You will get a conflict if the file is same and changes have been made else you will get up-to-date.
- Make the changes if required again and push the file by
- `git push`
- Type username and password. File will be pushed to the New Repository

STASHING

-
- Often, when you've been working on part of your project, things are in a messy state and you want to switch branches for a bit to work on something else. The problem is, you don't want to do a commit of half-done work just so you can get back to this point later. The answer to this issue is the git stash command.
-
- How to get commit ID?
- `git log -3`
-

Stashing Your Work

-
- `git status`
- `$ git stash`
- `$ git status`
- `git stash list`
- `git stash apply`
- `git stash drop stash@{0}`

- Git stash save
- This command is like Git stash. But this command comes with various options.
- Git stash with message
- `git stash save "Your stash message"`.
- The above command stashes with a message.
- Stashing untracked files
- You can also stash untracked files.
- `git stash save -u` or `git stash save --include-untracked`
- Git stash list
- When you `Git stash` or `Git stash save`, Git will actually create a Git commit object with some name and then save it in your repo.
- So it means that you can view the list of stashes you made at any time.
- `git stash list`

- Git stash apply
- This command takes the top most stash in the stack and applies it to the repo. In our case it is `stash@{0}`
- Git stash pop
- This command is very similar to `stash apply` but it deletes the stash from the stack after it is applied.
- Git stash show
- This command shows the summary of the stash diffs. The above command considers only the latest stash.
- Git stash branch <name>
- This command creates a new branch with the latest stash, and then deletes the latest stash (like `stash pop`).
- Git stash clear
- This command deletes all the stashes made in the repo. It maybe impossible to revert.
- Git stash drop
- This command deletes the latest stash from the stack. But use it with caution, it maybe be difficult to revert.
- `git stash drop stash@{1}`

•Creating a Branch from a Stash

- git stash branch testchanges
- Switched to a new branch 'testchanges'
-
- Cleaning your Working Directory
- git clean -d -n

Un-applying a Stash

- `git stash show -p stash@{0} | git apply -R`
- or `git stash show -p | git apply -R`
- `git config --global alias.stash-unapply '!git stash show -p | git apply -R'`
- `$ git stash apply` – To apply
- `$ git stash-unapply`
- Creating a Branch from a Stash = `git stash branch testchanges`

Tags

- Tags are ref's that point to specific points in Git history. Tagging is generally used to capture a point in history that is used for a marked version release (i.e. v1.0.1). A tag is like a branch that doesn't change. Unlike branches, tags, after being created, have no further history of commits.
-

- Creating a tag
- `git tag <tagname>`
- `git tag -a v1.4`

- git merge iss53
- Auto-merging index.html
- CONFLICT (content): Merge conflict in index.html
- Automatic merge failed; fix conflicts and then commit the result.
- git status
- REMOVE <<<<<< HEAD AND =====
AND >>>>.
- COMMIT AGAIN

Coding Standards and Best Practices

- Write comments and documentation
- Write readable yet efficient code
- Use helper methods
- If avoidable, do NOT hard-code!
- Write test cases. Don't forget the edge cases: 0s, empty strings/lists, nulls, etc.
- Write readable yet efficient code. Conform to the coding standards of your current project

THANK YOU