Table of contents

- Table of contents
  - Git Operations
    - Git Branching
      - Checkout branch
    - Visualizing Git
    - Git Merging-Local
      - Fast Forward Merge
      - 3-Way Merge
      - Merge Conflicts
    - Git Merge Summary
    - Git Merging Remote-Github Pull Requests
      - Creating a Pull Request
      - Navigate information in PR
      - View PRs information.
      - Managing a branch protection rule
      - Adding Users to Github Repo
    - .gitignore
    - checkout using commit by creating branch
    - Git Tagging
      - Tagging Later
      - Sharing Tags
      - Checking out Tags
    - Understanding the .git directory
  - Related Terms

Git Operations

- initialize a new empty directory to understand branching and merging OR
- Clone a new Empty repository.

```
git clone <GITHUB_URL>
```

```
echo "Hello World-1a" >> file1.txt
echo "Hello World-1b" >> file1.txt
git add file1.txt
git commit -m "added hello world-1 to file1.txt"

echo "Hello World - 2nd commit" >> file1.txt
git add file1.txt

git commit -m "added 2nd hello world to file1.txt"
OR
```

```
#If you want to stage all modified files and commit in one line command
git commit -a -m "added 2nd hello world to file1.txt"
```

```
git log --all --decorate --graph
```

- if you dont want to type the above command everytime , create a alias for above command

```
alias mygraph="git log --all --decorate --graph"
```

- use this alias command everytime to check the HEAD pointer
- To see all the alias created in linux

```
alias
ls -ltr .git/refs
git branch
```

- Content of this file is the checksum of last commit

```
cat .git/refs/heads/master
cat .git/refs/heads/main
```

- Verify the above checksum value with

```
git log
```

Git Branching

- `Branching` is the way to **work on different versions of a repository at one time.**
- By default your repository has one branch named `main/master` which is considered to be the definitive branch.
- We use branches to experiment and make edits before committing them to `main/master`.
- When you create a branch off the `main` branch, you're making a copy, or snapshot, of `main` as it was at that point in time.
- Create new branch - usually feature branches, using below command all the commits currently present in the `master` branch, a copy of all the commits will be present in `feature_1` branch as well.

Checkout branch

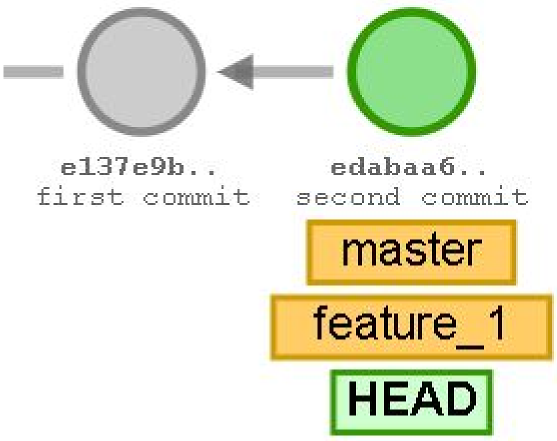
- To start working in a branch you have to checkout the branch. If you checkout a branch, the HEAD pointer moves to the last commit in this branch and the files in the working tree are set to the state of this commit.

- Checkout to your current branch

```
git checkout main
git branch
```

- Create a new branch using below command:

```
git branch
# Create a new branch
git branch feature_1
git checkout feature_1
OR
git checkout -b feature_1
```

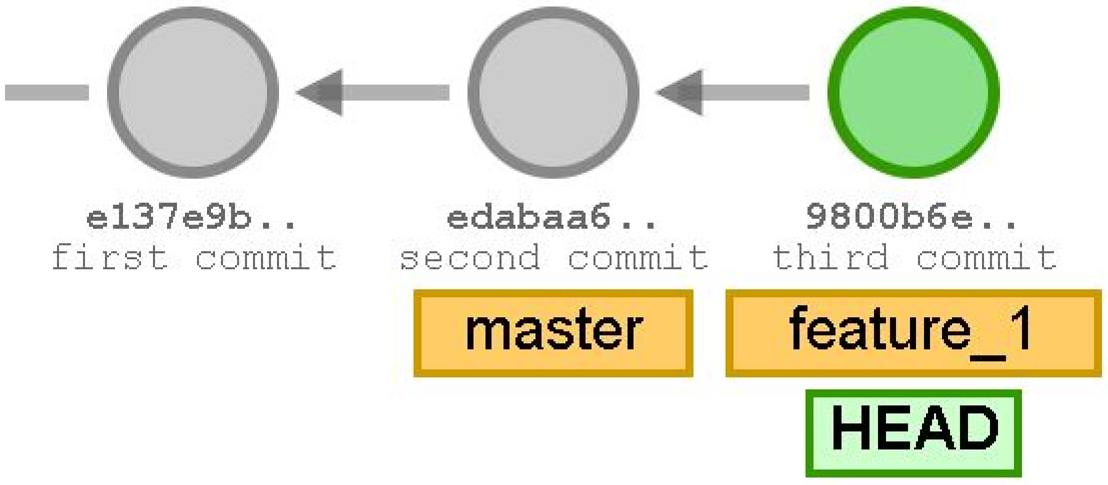


- checkout to your current branch

```
git log
git branch
```

- Lets add some changes in this specific branch.

```
echo "Adding this line only in feature_1 branch" >> file1.txt
git add file1.txt
git commit -m "added line in feature_1 branch file1.txt"
```



- As shown here, the commit id is only present in `feature_1` branch. Here , we can say that, `feature_1` branch is ahead of `main` branch.
- Use `git checkout <branch_name>` to switch different branches and `HEAD` as per its definition, **is a pointer to latest commit in the current checkout branch** .
- Thus `HEAD` pointer moves along with checkout branch.
- Check all branches locally.

```
git branch
```

- Since there are multiple branches now, i.e main and feature branches, git knows about the current branch using `HEAD`

```
git checkout main
cat .git/HEAD
git checkout feature_1
cat .git/HEAD
git branch
mygraph
```

- To View all changes using commit-id

```
git show <commit-id>
```

Visualizing Git

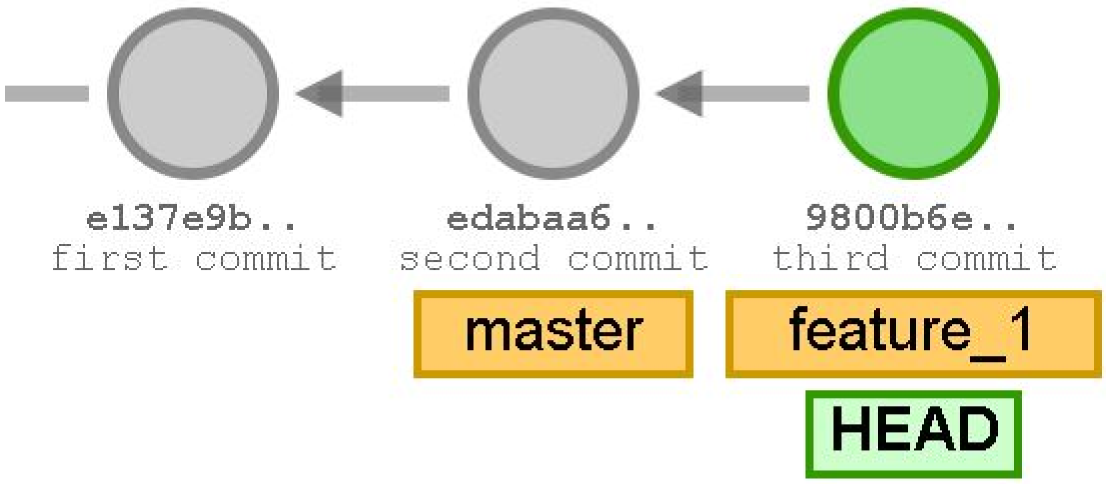
- Navigate to Visualizing Git

Git Merging-Local

- Git merge will combine multiple sequences of commits into one unified history.
- The `git merge` command lets you take the independent lines of development created by git branch and integrate them into a single branch.

Fast Forward Merge

- Lets say, we have the two branches and we want to merge `feature_1` into `main`



> Here main branch is pointing to a commit that is already in the history of the feature1 branch. This means that all of the commits on the main branch are already on the feature1 branch.

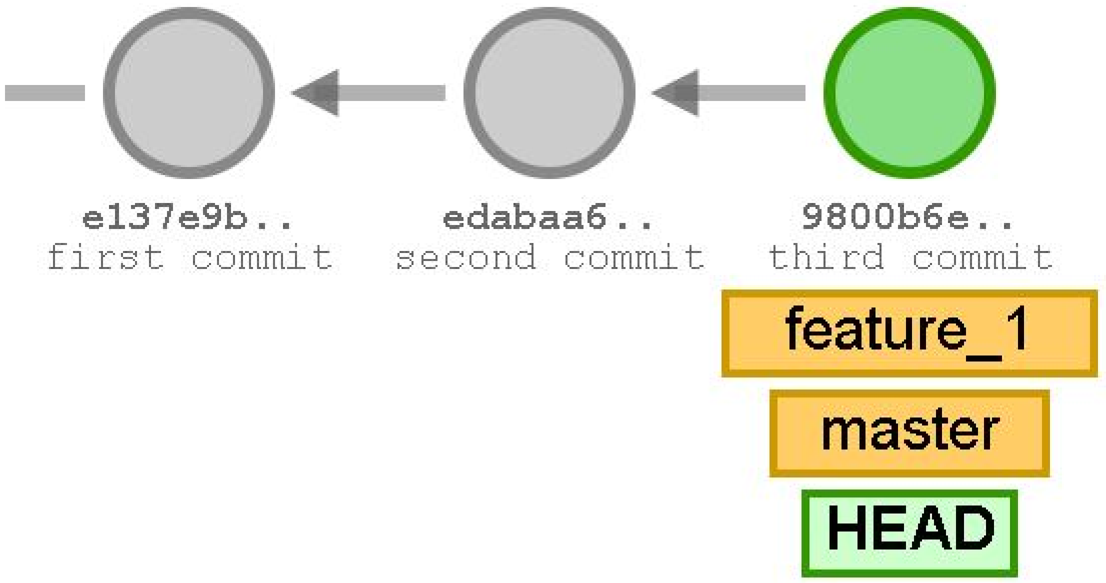
- This type of merge only be done when there is direct path available.
- In this scenario, when we use Git merge, it will do a `fast-forward` merge by default, meaning it will simply move (or fast-forward) the `main` and `HEAD` refs so that they point to the commit that `feature1` points to.
- Check diff between two branches, below command will shows what will change if we merge feature_1 into main

```
git diff main..feature_1
```

- Use below commands to merge `feature_1` branch into `main`.

```
git checkout main
cat file1.txt
git merge feature_1
mygraph
cat file1.txt
```

- Execute `git status` to ensure that `HEAD` is pointing to the correct `merge-receiving` branch.



- undo a recent merge

```
git reset --merge ORIG_HEAD
mygraph
cat file1.txt
git merge feature_1
mygraph
```

- The current branch i.e `main` will be updated to reflect the merge, but the target branch i.e `feature_1` will be completely unaffected.
- verify the branches that are merged

```
git branch --merged
```

- A best practice always followed is that if work is done on the `feature_1` branch and these changes are merged into base branch, we should be deleting the branch to avoid multiple unnecessary feature branches.

```
git branch -d feature_1
```

- Once changes from child branch are merged in main branch, these changes can be pushed to remote repo.

```
git push origin main
```

- When a branch is deleted, it provides with a SHA Id , to undo the deleted branch , simply create the branch with same name with the same commit id

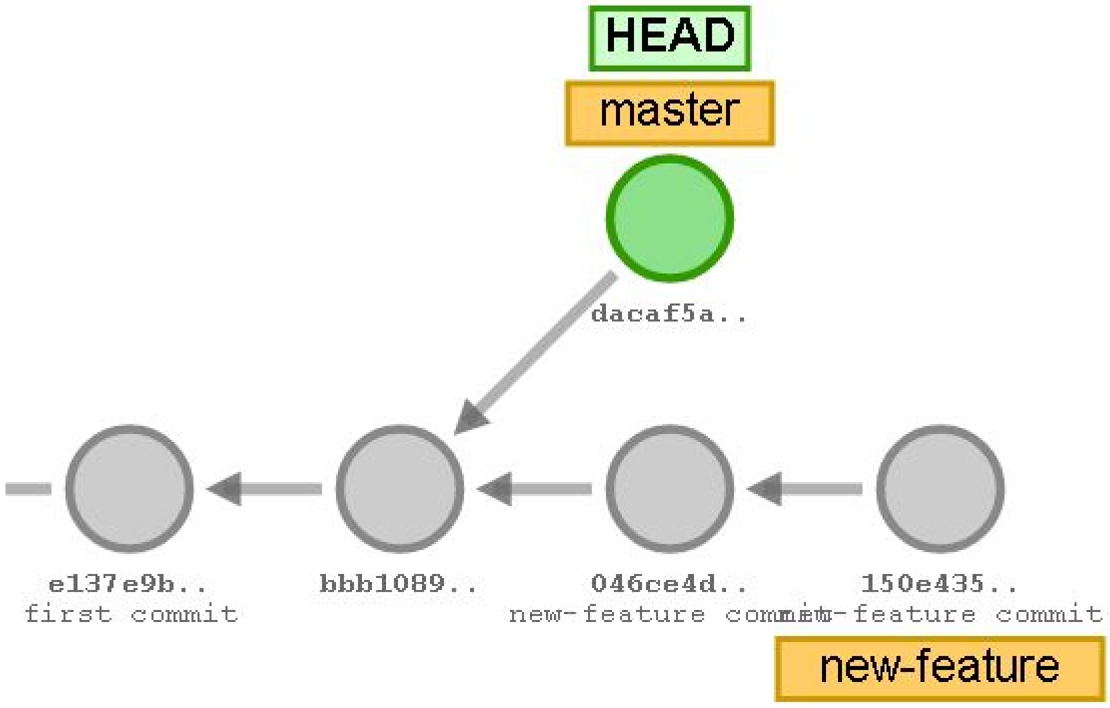
```
git branch feature_1 <COMMIT_FROM_ABOVE_DELETE_COMMAND>
```

3-Way Merge

- Git will perform a 3-Way merge incase the base branch moves ahead before the feature branch is already merged.
- 3-way merges use a dedicated commit to tie together the two histories.
- Here Git uses three commits to generate the merge commit: the two branch tips and their common ancestor.
- Check if we are in base branch.

```
git checkout main
git status
git checkout -b new-feature main

echo "Adding this line in a new feature branch" >> new_feature_file.txt
git add new_feature_file.txt
git commit -m "added line in new feature file"

echo "Adding another line in a new feature branch" >> new_feature_file.txt
git add new_feature_file.txt
git commit -m "added another line in new feature file"

git checkout main
echo "Adding this line in main_file.txt" >> main_file.txt
git add main_file.txt
git commit -m "added line in base branch file"

# At this point, latest commit changes in base branch are not known to `new-
feature` branch
```

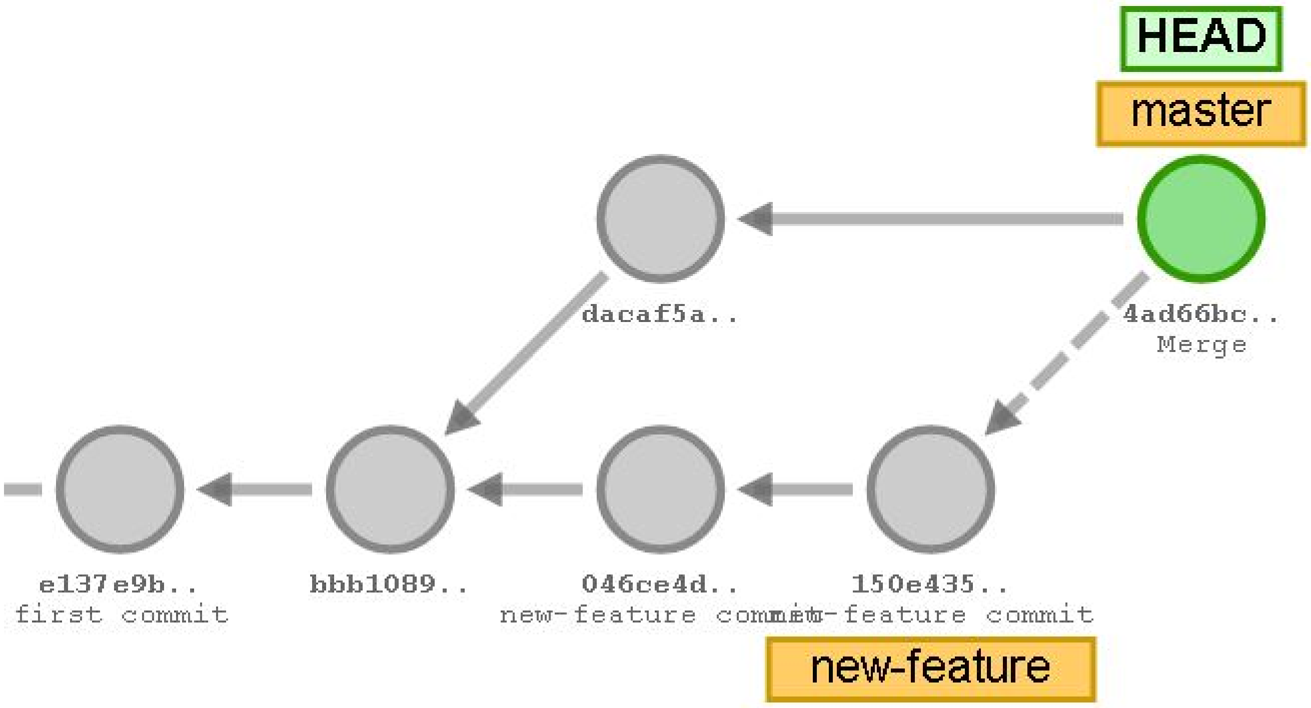
Since main is now ahead of the path from where `new-feature` was created, a direct `fast forward merge` is



not possible

- Lets try to merge `new-feature` into `main`

```
git merge new-feature
```



> Make sure no one else has to work on that feature branch.

- if we try to delete a feature branch that is not yet merged, git will display a "`not fully merged`" message

---

Merge Conflicts

> Merging two branches that have changes in the same lines in same files can create merge conflicts

- lets create a new file for merge conflict scenario

```
git checkout main
echo "1st line content in merge.txt" > merge.txt
git commit -am "added a new file with merge.txt with 1st line content"
```

- Lets checkout to a new branch and modify similar content to same file

```
git checkout -b merge_branch
echo "replacing the content of this file to merge later" > merge.txt
git add merge.txt
git commit -m "modified entire content of merge.txt to create a merge conflict"
git checkout main
echo "appended some lines to merge.txt" >> merge.txt
git commit -am "appended some content to merge.txt"
git merge merge_branch
# Auto-merging merge.txt
# CONFLICT (content): Merge conflict in merge.txt
# Automatic merge failed; fix conflicts and then commit the result.

git status
cat merge.txt
```

- Below lines indicate

```
   <<<<<<< HEAD
    =======
    >>>>>>> merge_branch
```

> The ======= line is the "center" of the conflict. All the content between the center and the <<<<<<< HEAD line is content that exists in the current branch `main` which the HEAD ref is pointing to.

> Alternatively all content between the center and >>>>>>> new_branch_to_merge_later is content that is present in our merging branch.

- To resolve the merge conflict, edit the file, keep the line that is required from specific branch OR remove the dividers.

```
git add merge.txt
git commit -m "conflicts in merge.txt are merged and resolved"
```

Git Merge Summary

- Git merging combines sequences of commits into one unified history of commits.
- There are two main ways Git will merge: Fast Forward and Three way
- Git can automatically merge commits unless there are changes that conflict in both commit sequences.

Git Merging Remote-Github Pull Requests

> It is possible to merge one branch changes to another in Remote SCM

- Create a new branch and push it to remote.
  - `git push origin branch_name`

Creating a Pull Request

- Follow below steps to create PR
  - Login to Github Portal UI
  - Navigate to specific Repository
  - Click on **Pull request**.
  - Click on New Pull request
  - Put PR pull from branch: `feature_branch`
  - Put PR merge into branch: `main`
  - Click on New Pull Request
  - Add PR title as `Merge from feature into main`
  - Click on **Create Pull request**

Navigate information in PR

- Navigate to Repository to view existing Pull Requests
  - Select the PR(in opened/closed state)
  - This allows to add/review comments and view Commits and Files changed in the feature branch.
  - Click on the green button Merge Pull Request and then confirm again by clicking on the green button
  - Merge Pull Request to merge PR
  - Here changes from feature branch gets merged into base branch in Remote Repository.

View PRs information.

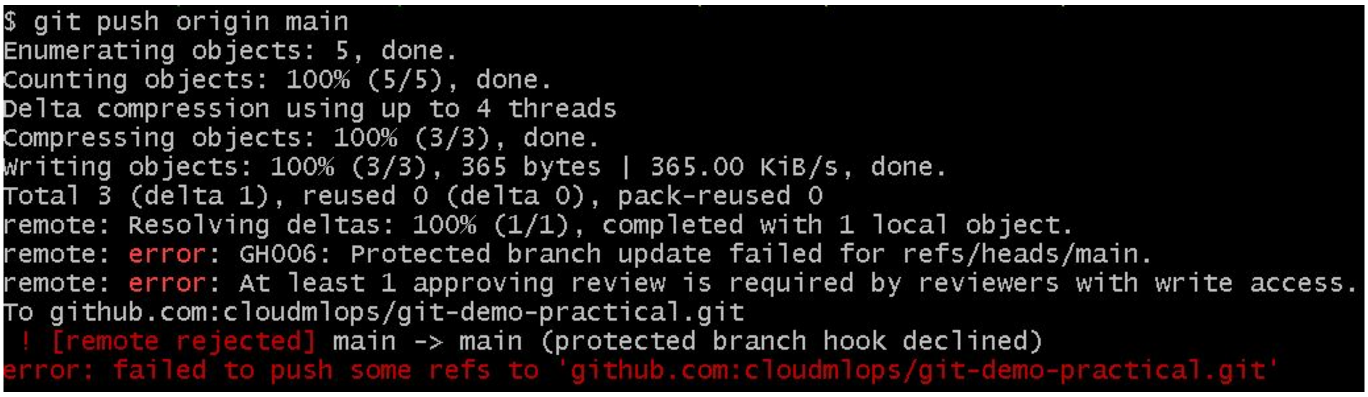
- View PRs information for one of the Open Source Github Repository i.e boto3

Managing a branch protection rule

- On GitHub, navigate to the main page of the repository.
- Under your repository name, click `Settings`.
- In the left menu, click Branches.
- Next to `Branch protection rules`, click Add rule.
- Under `Branch name pattern`, type the branch name or pattern you want to protect.
- Enable required pull request reviews.

- Under `Protect matching branches`, select `Require pull request reviews before merging`
    - select the `number of approving reviews`
    - select `Include administrators`.

> Once above settings are enabled for `main` branch, and these changes are saved. If you try to push commits directly to main branch, you will get below error.



```
$ git push origin main
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 365 bytes | 365.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote: error: GH006: Protected branch update failed for refs/heads/main.
remote: error: At least 1 approving review is required by reviewers with write access.
To github.com:cloudmlops/git-demo-practical.git
 ! [remote rejected] main -> main (protected branch hook declined)
error: failed to push some refs to 'github.com:cloudmlops/git-demo-practical.git'
```

Adding Users to Github Repo

- Navigate to Github Repository `Settings` > `Manage Access` > `Add People` > `Search for Collaborator using Github Username` > `Add user to this repository`

- The collaborator will get a email invitation to accept/reject.

- Once Accepted, the collaborator user will be able to access the repository present in another account.

- Here, the collaborator can create multiple feature branches and raise a PR to merge into main.

- Reviwers can be added for code review and changes can be merged once PR is approved.

- Create a new child branch, make some commits in local child branch, push the child branch, raise PR and add collaborator as Reviewer.

- Once PR is approved, then only Merge Option is enabled.

.gitignore

- You will never have to stage or commit files that match regular expression mentioned in `.gitignore`
- `.gitignore` file

```
echo "test" >> ignore.txt
vi .gitignore
- Enter `*.png , *.jpg, *.pdf`
- enter the name of the files that git should not consider for staging,commit etc
git status
```

- stage and commit the .gitignore file

- Push the changes to your Github/CodeCommit Repository

```
git push origin main
```

checkout using commit by creating branch

- Create a new branch by using commit id
  - `git branch older_branch_ref COMMIT_ID`
- To look at all the files at a particular commit id:

```
git checkout commit-id
```

- Here, `HEAD` pointer is pointing to commit directly instead of a branch
- To point `HEAD` again to branch , use below

```
git checkout main
```

- If you want to create a new branch from any other previous commit-id

```
git checkout commit-id
```

- Create a new branch from where the `HEAD` is pointing to a commit id

```
git branch commit-branch
```

- Checkout the new branch for `HEAD` to point to it

```
git checkout commit-branch
```

Git Tagging

Git has the ability to tag specific points in a repository's history as being important. Typically, people use this functionality to mark release points (v1.0, v2.0 and so on). In this section, you'll learn how to list existing tags, how to create and delete tags, and what the different types of tags are.

Listing Your Tags

```
git tag
```

- Creating Tags
    - Annotated Tags Annotated tags, however, are stored as full objects in the Git database. They contain the tagger name, email, and date; have a tagging message. It’s generally recommended that you create annotated tags so you can have all this information.
- Whenever you create a tag, local main and remote main is in sync

```
git tag -a v1.0 -m "app version 1.0"
```

- The -m specifies a tagging message, which is stored with the tag.
- To view the tag data

```
git show v1.0
```

Tagging Later

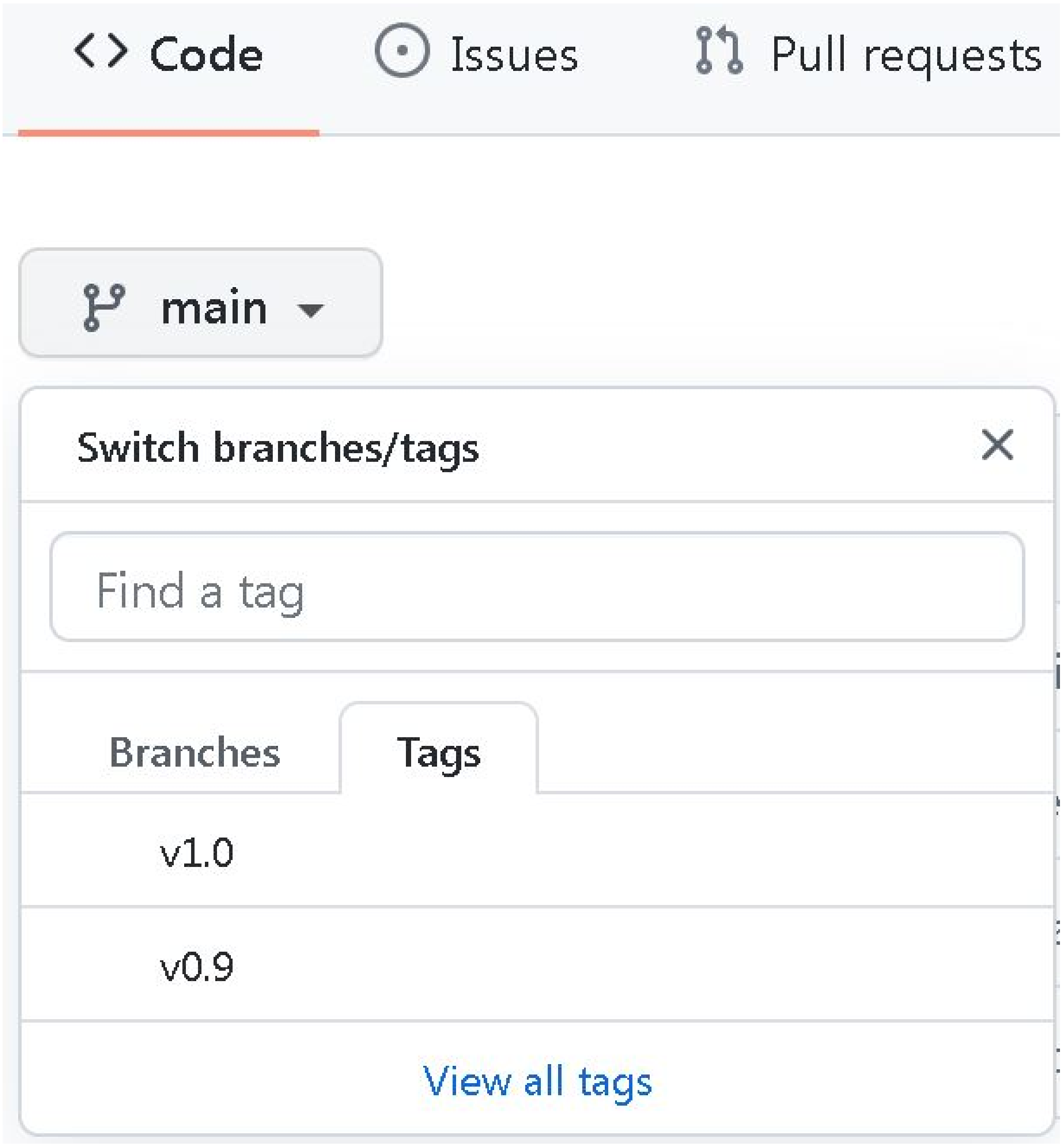
- In case you have many commits already done, and you forgot to tag one of the commit id

```
git tag -a v0.9 <COMMIT_ID> -m "app version 0.9"
git tag
```

Sharing Tags

```
# push a specfic tag to remote repsitory
git push origin v1.0

# push all the local tags to remote
git push origin --tags
```

- Above commands will push the local tags into remote repo.



Checking out Tags

To view all the versions of the files pointing to a tag

```
git checkout v1.0
```

- If you want to make some changes after this tag release

```
git checkout -b branch1.0 v1.0
```

- This will create a branch and you can commits to this branch.
- This new tag branch can be pushed to remote, and same lifecycle of Remote Merge can be followed

Understanding the .git directory

- When we create a git repo, using git init, git creates this directory: the `.git`.
- Below is the directory structure for .git folder before we make any first commit:

```
├── HEAD
├── branches
├── config
├── description
├── hooks
│ ├── pre-commit.sample
│ ├── pre-push.sample
│ └── ...
├── info
│ └── exclude
├── objects
│ ├── info
│ └── pack
└── refs
    ├── heads
    └── tags
```

Related Terms

- `git clone [url]`: Clone (download) a repository that already exists on GitHub, including all of the files, branches, and commits.
- `git status`: Always a good idea, this command shows you what branch you're on, what files are in the working or staging directory, and any other important information.
- `git branch`: This shows the existing branches in your local repository. You can also use `git branch [banch-name]` to create a branch from your current location, or `git branch --all` to see all branches, both the local ones on your machine, and the remote tracking branches stored from the last `git pull` or `git fetch` from the remote.
- `git push`: Uploads all local branch commits to the remote.
- `git log`: Browse and inspect the evolution of project files.
- `git remote -v`: Show the associated remote repositories and their stored name, like `origin`.
- `git checkout [branch-name]`: Switches to the specified branch and updates the working directory.