Table of contents

Git Operations

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

```
git init
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
```

- Content of this file is the checksum of last commit

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

- Verify the above checksum value with

```
git log
```

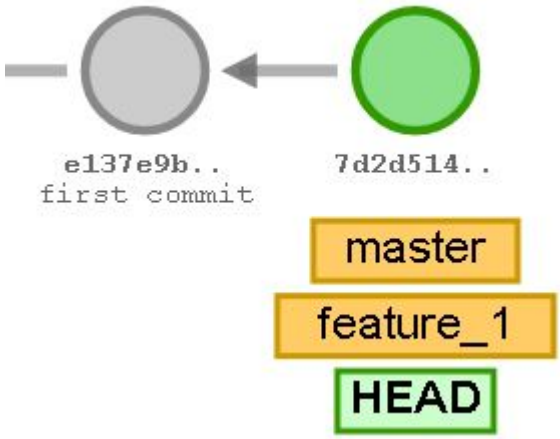
Git Branching

- `Branching` is the way to **work on different versions of a repository at one time.**
- By default your repository has one branch named `master` which is considered to be the definitive branch.
- We use branches to experiment and make edits before committing them to master.
- When you create a branch off the master branch, you're making a copy, or snapshot, of master 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.
- check your current branch

```
git branch
```

- Create a new branch using below command:

```
git branch feature_1
git checkout feature_1
OR
git checkout -b feature_1
```

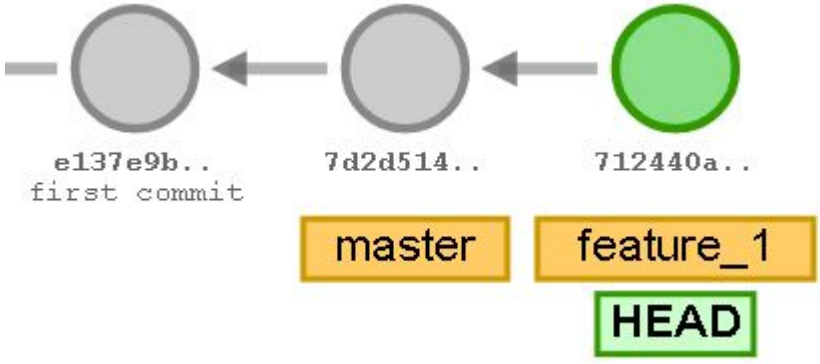


- check your current branch

```
git branch
```

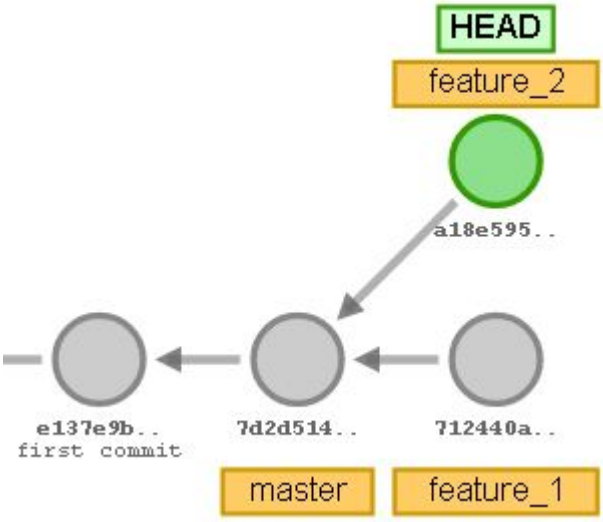
- Lets add some changes in this specific branch.

```
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"
```



- There can be multiple branches created from any specific branch/commit.

```
git checkout master
git branch feature_2
git checkout feature_2
echo "Adding this line only in feature_2 branch" >> file1.txt
git add file1.txt
git commit -m "added line in feature_2 branch file1.txt"
```



- Check all branches

```
git branch
```

```
ls -ltr .git/refs
ls -ltr .git/refs/heads
cat .git/refs/heads/feature_1
```

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

```
cat .git/HEAD
git checkout master
cat .git/HEAD
git branch
```

- Check the HEAD pointer

```
mygraph
```

- To View all changes using commit-id

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

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

- This type of merge only be done when there is direct path available

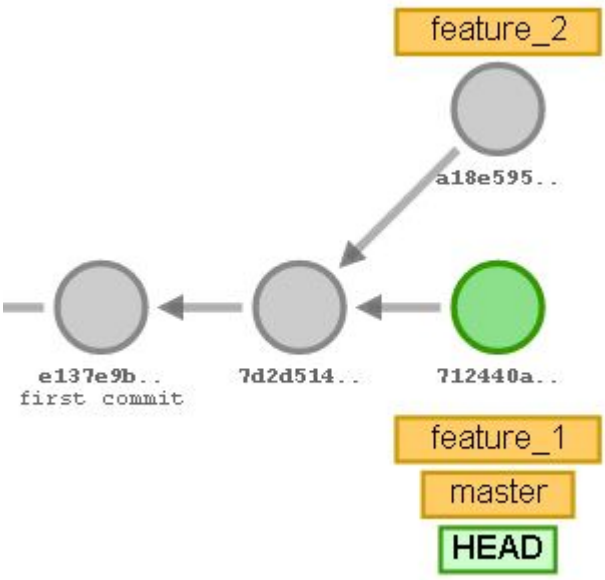
```
git checkout master
```

- Check diff between two branches, below command will shows what will change if we merge feature_1 into master

```
git diff master..feature_1
```

```
cat file1.txt
```

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



- undo a recent merge

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

```
git merge feature_1
mygraph
```

- The current branch i.e `master` 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 , we should be deleting the branch to avoid multiple unnecessary feature branches.

```
git branch -d feature_1
```

- if we try to delete the other feature branch i.e feature_2, git will display a "not fully merged" message

```
git branch -d feature_2
```

3-Way Merge

- Since master is now ahead of the path from where feature_2 was created, a direct `fast forward merge` is not possible
- if it is done, changes at the current master branch can be lost
- Check if we are in master branch

```
git checkout master
git status
```

- To merge feature_2 branch into master

```
git merge feature_2
```

- Check pointer with mygraph command

```
mygraph
```

- Confirm whether master and feature_2 are merged git branch --merged
- Now branch feature_2 can be deleted

```
git branch -d feature_2
```

- 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_2 <COMMIT_FROM_ABOVE_DELETE_COMMAND>
```

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

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

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

```
git checkout -b merge_branch
echo "replacing the content of this file to merge later" > merge.txt
git commit -am "modified entire content of merge.txt to create a merge conflict"
```

```
git checkout master
echo "appended some lines to merge.txt" >> merge.txt
git commit -am "appended some content to merge.txt"
```

```
git merge merge_branch

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 master 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"
```

- 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 master
```

- 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
```

- `.gitignore` file

```

```

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

- stage and commit the .gitignore files
- Push the changes to your Github/CodeCommit Repository

```
git push origin master
```

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.

```
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 v0.1 <COMMIT_ID>
git tag
```

Sharing Tags

- To push a specfic tag to remote repsitory

```
git push origin v1.0
```

- To push all the local tags to remote

```
git push origin --tags
```

Checking out Tags

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

```
git checkout v1.0
```

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

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

- This will create a branch and you have commits to this branch.

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
```

CodeCommit UI

> AWS CodeCommit is a version control service hosted by Amazon Web Services that you can use to privately store and manage assets

- Limit Pushes and Merges to Branches

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": [
                "codecommit:GitPush",
                "codecommit:DeleteBranch",
                "codecommit:PutFile",
                "codecommit:MergeBranchesByFastForward",
                "codecommit:MergeBranchesByThreeWay",
                "codecommit:MergePullRequestByFastForward",
            ],
            "Resource": "arn:aws:codecommit:us-east-2:<AWS_ACCOUNT_ID>:
<REOPO_NAME",
            "Condition": {
                "StringEqualsIfExists": {
                    "codecommit:References": [
                        "refs/heads/master"
                     ]
                },
                "Null": {
                    "codecommit:References": false
                }
            }
        }
    ]
}
```

- Apply the IAM Policy to an IAM Group or Role
- The policy has no effect until you apply it to an IAM user, group, or role.
- As a best practice, consider applying the policy when there is IAM User or IAM group
- Create a IAM group with name `developer` and add IAM users to the group, and assign this policy to the `developer` group.
- Attach policy to this IAM Group
- `Test the above scenario as below:`
- You should test the effects of the policy you've applied on the group to ensure that it acts as expected.

- Sign in to the CodeCommit console with an IAM user who is a member of an IAM group `developer` that has the policy applied. In the console, add a file on the branch where the restrictions apply. You should see an `error message` when you attempt to save or upload a file to that branch. Add a file to a different branch. The operation should succeed.
- From the terminal or command line or git bash, create a commit on the branch where the restrictions apply, and then push that commit to the CodeCommit repository. You should see an error message. Commits and pushes made from other branches should work as usual.