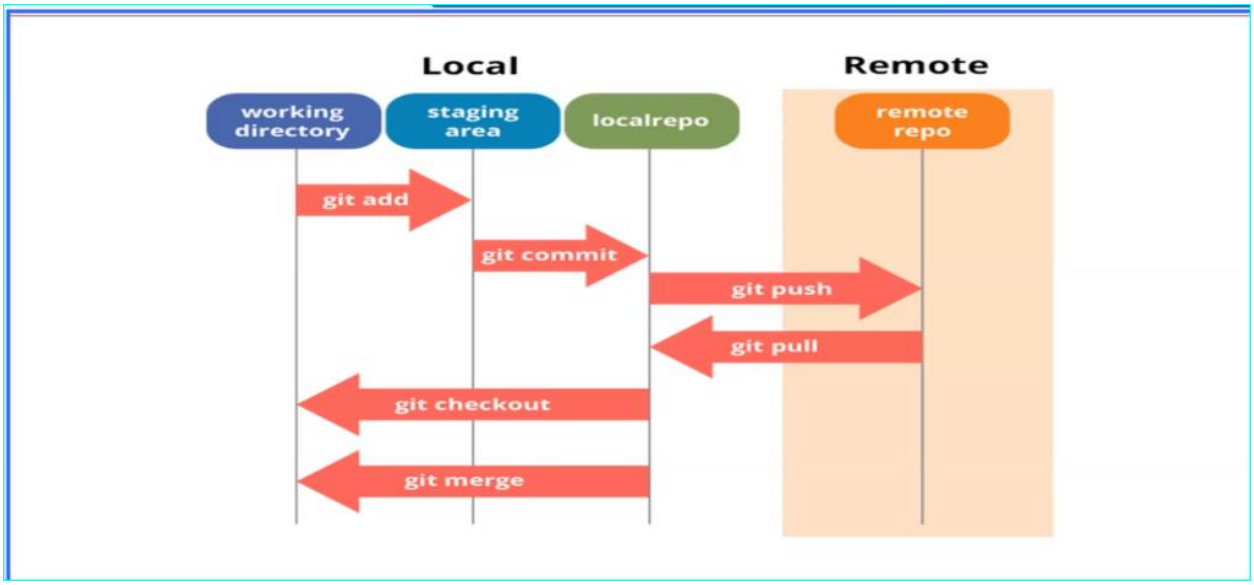


GIT WORKFLOW

The Three Trees

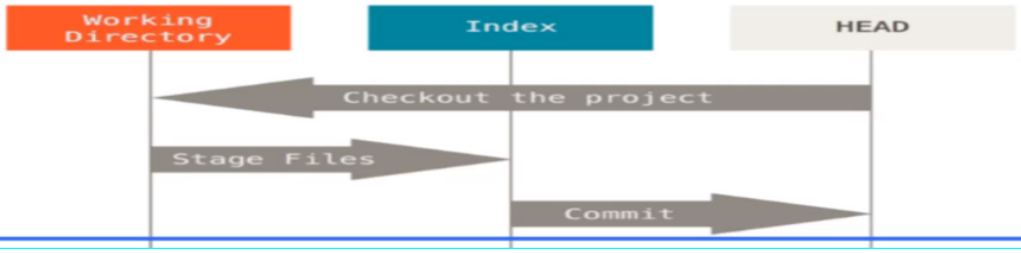
An easier way to think about `reset` and `checkout` is through the mental frame of Git being a content manager of three different trees. By "tree" here, we really mean "collection of files", not specifically the data structure. (There are a few cases where the index doesn't exactly act like a tree, but for our purposes it is easier to think about it this way for now.)

Git as a system manages and manipulates three trees in its normal operation:

Tree	Role
HEAD	Last commit snapshot, next parent
Index	Proposed next commit snapshot
Working Directory	Sandbox

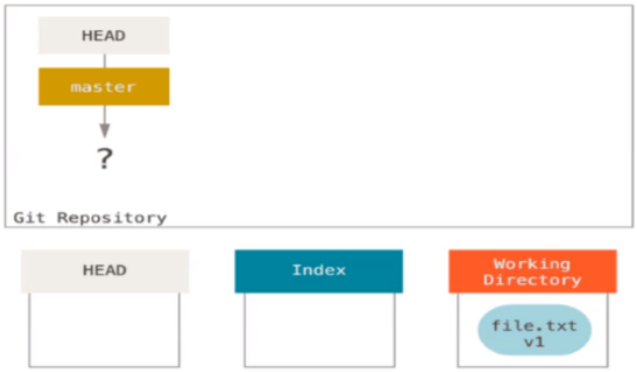
The Workflow

Git's main purpose is to record snapshots of your project in successively better states, by manipulating these three trees.



GIT WORKFLOW

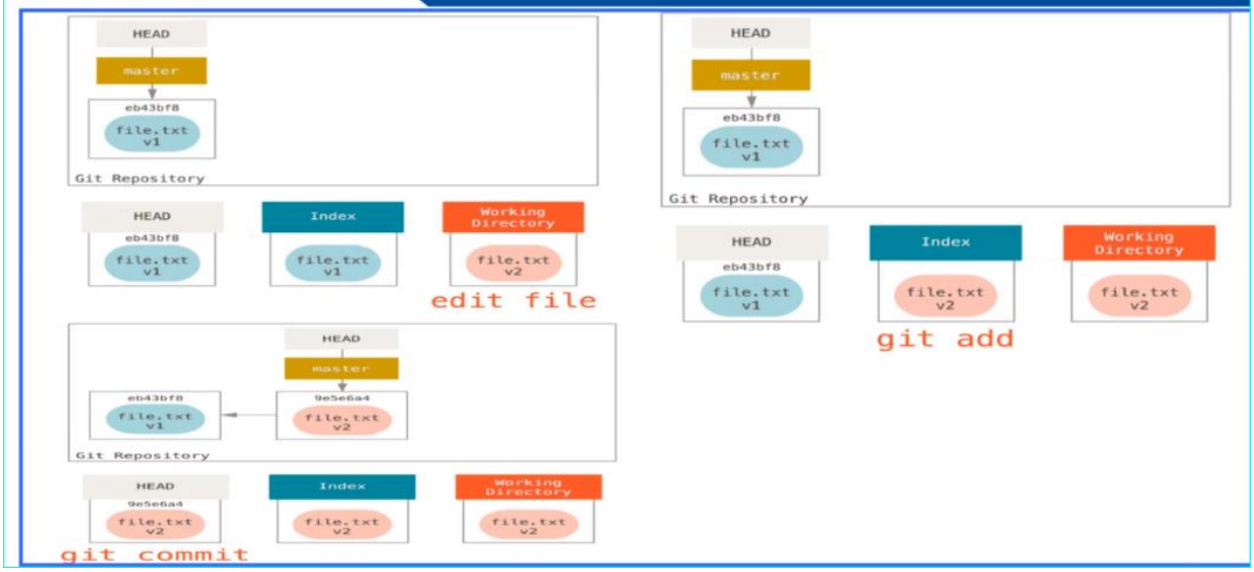
- Now we run git init, which will create a Git repository with a HEAD reference which points to an unborn branch (master doesn't exist yet).



GIT WORKFLOW

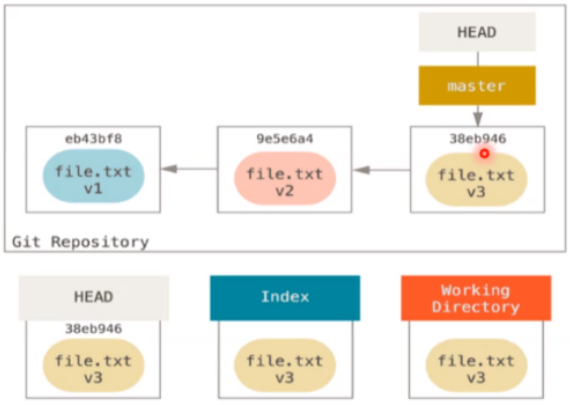
Then we run git commit, which takes the contents of the Index and saves it as a permanent snapshot, creates a commit object which points to that snapshot, and updates master to point to that commit.

Git Workflow



Git Workflow

- For the purposes of these examples, let's say that we've modified file.txt again and committed it a third time. So now our history looks like this:

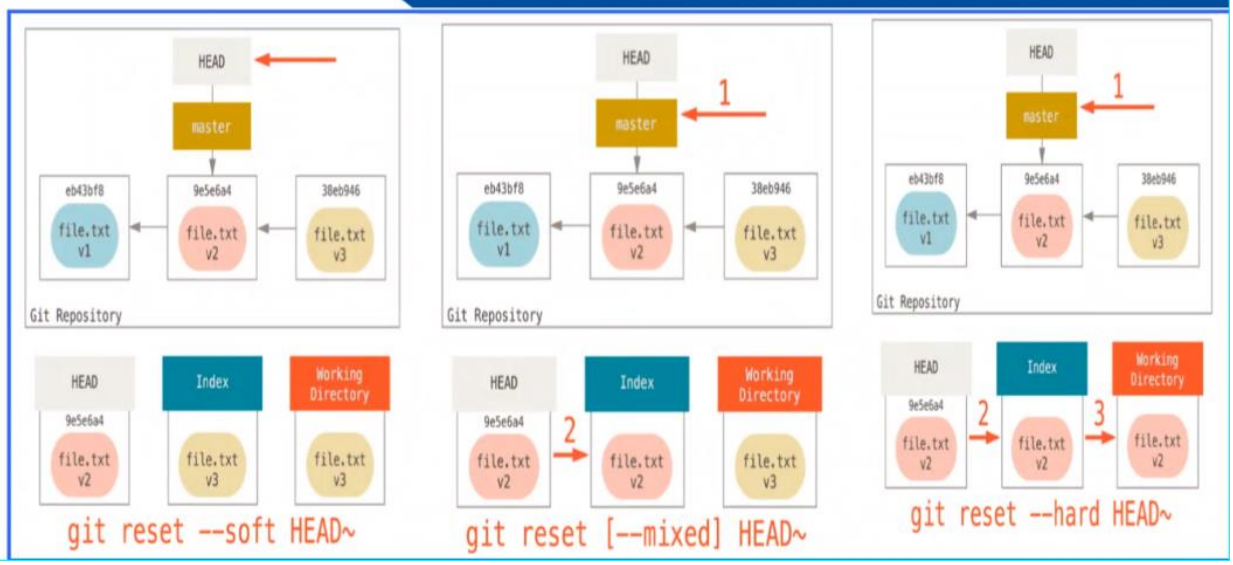


Git Branching

- A branch in Git is simply a lightweight movable pointer to one of the commits.
- The default branch name in Git is master.
- As you start making commits, you're given a master branch that points to the last commit you made.
- Every time you commit, the master branch pointer moves forward automatically.
- The "master" branch in Git is not a special branch. It is exactly like any other branch. The only reason nearly every repository has one is that the git init command creates it by default and most people don't bother to change it
- Because a branch in Git is actually a simple file that contains the 40 character SHA-1 checksum of the commit it points to, branches are cheap to create and destroy. Creating a new branch is as quick and simple as writing 41 bytes to a file (40 characters and a newline).

Figure 11. A branch and its commit history

GIT RESET



Git Branching and Merging

Let's go through a simple example of branching and merging with a workflow that you might use in the real world. You'll follow these steps:

1. Do some work on a website.
2. Create a branch for a new story you're working on.
3. Do some work in that branch.

At this stage, you'll receive a call that another issue is critical and you need a hotfix. You'll do the following:

1. Switch to your production branch.
2. Create a branch to add the hotfix.
3. After it's tested, merge the hotfix branch, and push to production.
4. Switch back to your original story and continue working.

First, let's say you're working on your project and have a couple of commits already on the master branch.

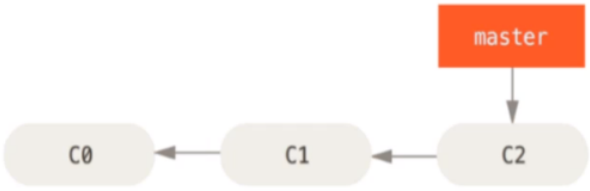


Figure 18. A simple commit history

Git Branching and Merging

You've decided that you're going to work on issue #53 in whatever issue-tracking system your company uses.

To create a new branch and switch to it at the same time, you can run the git checkout command with the -b switch:

$ git checkout -b iss53

Switched to a new branch "iss53"

This is shorthand for:

$ git branch iss53

$ git checkout iss53

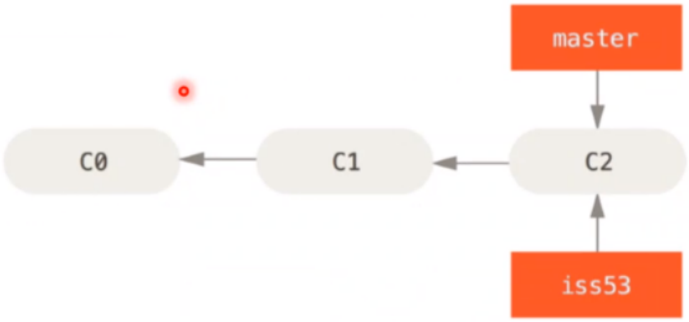


Figure 19. Creating a new branch pointer

Git Branching and Merging

You work on your website and do some commits. Doing so moves the iss53 branch forward, because you have it checked out (that is, your HEAD is pointing to it):

$ vim index.html

$ git commit -a -m 'added a new footer [issue 53]'

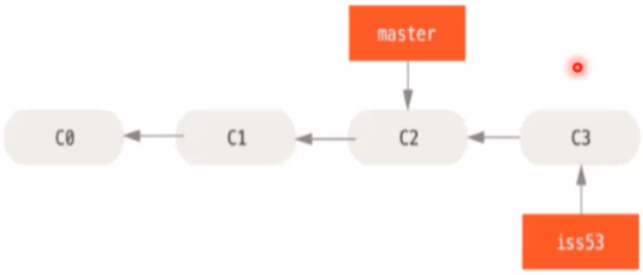


Figure 20. The iss53 branch has moved forward with your work

Git Branching and Merging

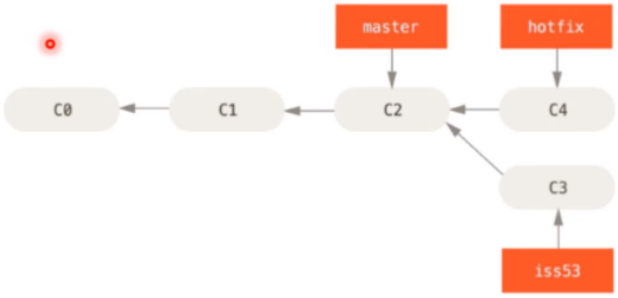
Now you get the call that there is an issue with the website, and you need to fix it immediately. With Git, you don't have to deploy your fix along with the iss53 changes you've made, and you don't have to put a lot of effort into reverting those changes before you can work on applying your fix to what is in production. All you have to do is switch back to your master branch.

However, before you do that, note that if your working directory or staging area has uncommitted changes that conflict with the branch you're checking out, Git won't let you switch branches. It's best to have a clean working state when you switch branches.

$ git checkout master

Switched to branch 'master'

Let's create a hotfix branch on which to work until it's completed:
$ git checkout -b hotfix
Switched to a new branch 'hotfix'
$ vim index.html
$ git commit -a -m 'fixed the broken email address'
[hotfix 1fb7853] fixed the broken email address
1 file changed, 2 insertions(+)



Git Branching and Merging

You can run your tests, make sure the hotfix is what you want, and finally merge the hotfix branch back into your master branch to deploy to production. You do this with the git merge command:

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
index.html | 2 ++
1 file changed, 2 insertions(+)
```

You'll notice the phrase "fast-forward" in that merge. Because the commit C4 pointed to by the branch hotfix you merged in was directly ahead of the commit C2 you're on, Git simply moves the pointer forward. To phrase that another way,
when you try to merge one commit with a commit that can be reached by following the first commit's history, Git simplifies things by moving the pointer forward because there is no divergent work to merge together — this is called a "fastforward."
Your change is now in the snapshot of the commit pointed to by the master branch, and you can deploy the fix.

Git Branching and Merging

After your super-important fix is deployed, you're ready to switch back to the work you were doing before you were interrupted. However, first you'll delete the hotfix branch, because you no longer need it — the master branch points at the same place. You can delete it with the -d option to git branch:

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

Now you can switch back to your work-in-progress branch on issue #53 and continue working on it.

```
$ git checkout iss53
Switched to branch "iss53"
$ vim index.html
$ git commit -a -m 'finished the new footer [issue 53]'
[iss53 ad82d7a] finished the new footer [issue 53]
1 file changed, 1 insertion(+)
```

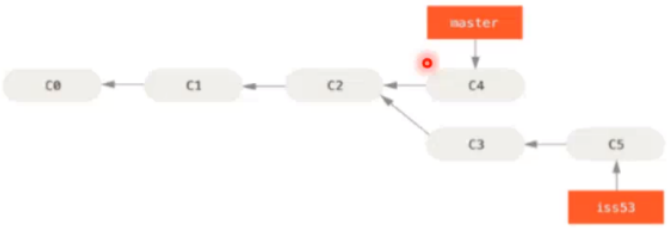


Figure 23. Work continues on iss53

Git Branching and Merging

Suppose you've decided that your issue #53 work is complete and ready to be merged into your master branch. In order to do that, you'll merge your iss53 branch into master, much like you merged your hotfix branch earlier. All you have to do is check out the branch you wish to merge into and then run the git merge command:

```
$ git checkout master
Switched to branch 'master'
$ git merge iss53
Merge made by the 'recursive' strategy.
index.html |    1 +
1 file changed, 1 insertion(+)
```

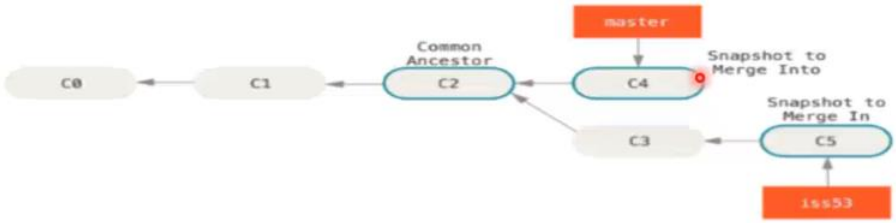


Figure 24. Three snapshots used in a typical merge

Git Branching and Merging

Instead of just moving the branch pointer forward, Git creates a new snapshot that results from this three-way merge and automatically creates a new commit that points to it. This is referred to as a merge commit, and is special in that it has more than one parent.

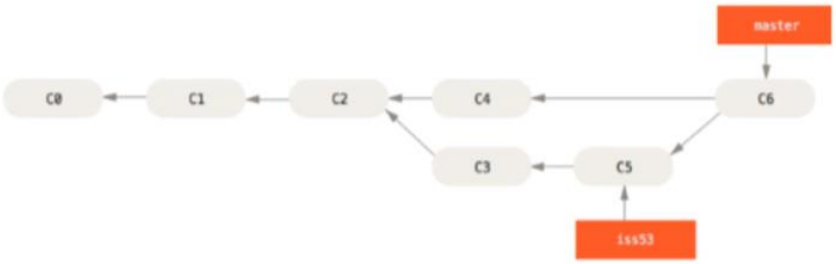


Figure 25. A merge commit

$ git branch -d iss53

Branch Management

The git branch command does more than just create and delete branches. If you run it with no arguments, you get a simple listing of your current branches:

$ git branch

```
iss53 *
master
Testing
```

To see the last commit on each branch, you can run git branch -v:

$ git branch -v

```
iss53   93b412c fix javascript issue
master  7a98805 Merge branch 'iss53'
testing 782fd34 add scott to the author list in the readmes
```

$ git branch --merged

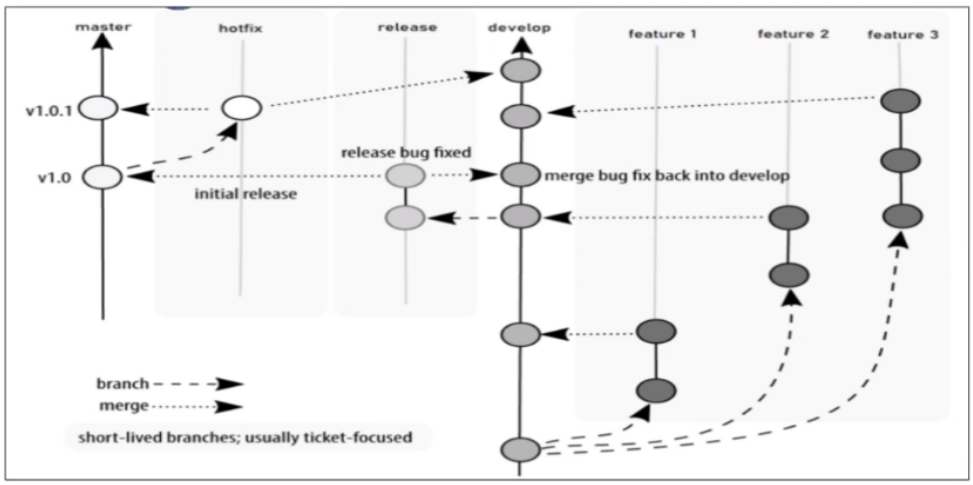
```
iss53
* master
```

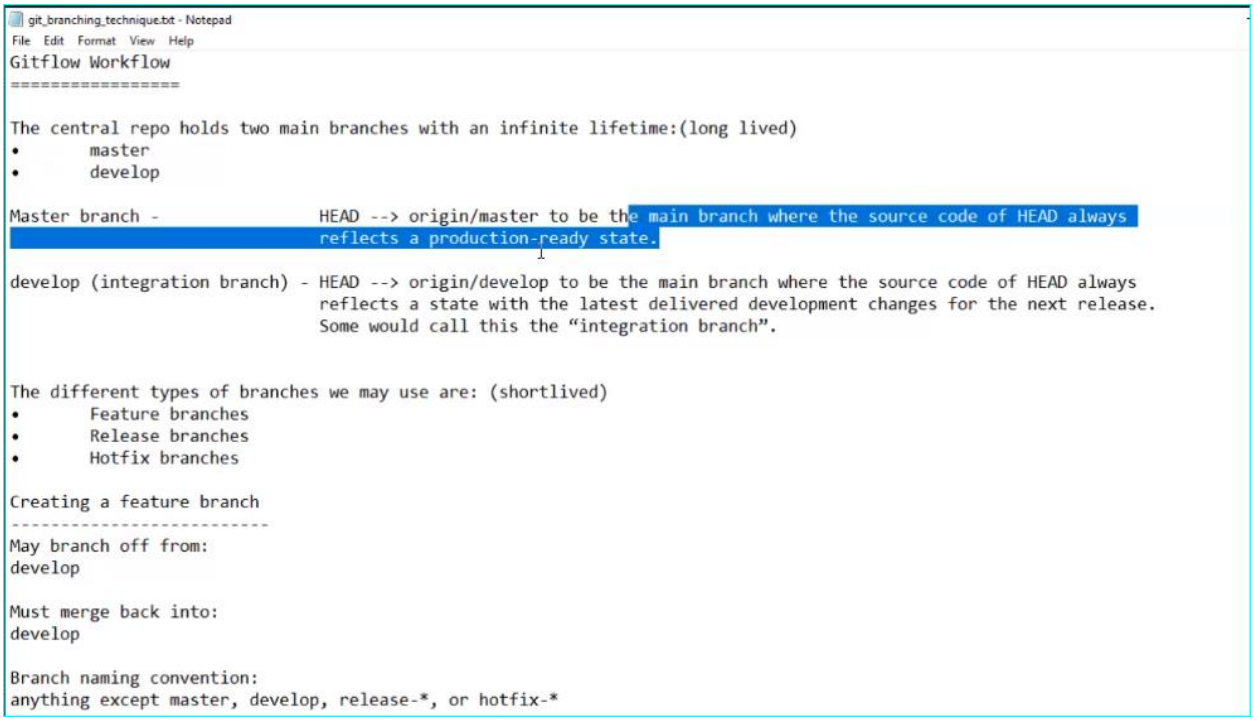
$ git branch --no-merged

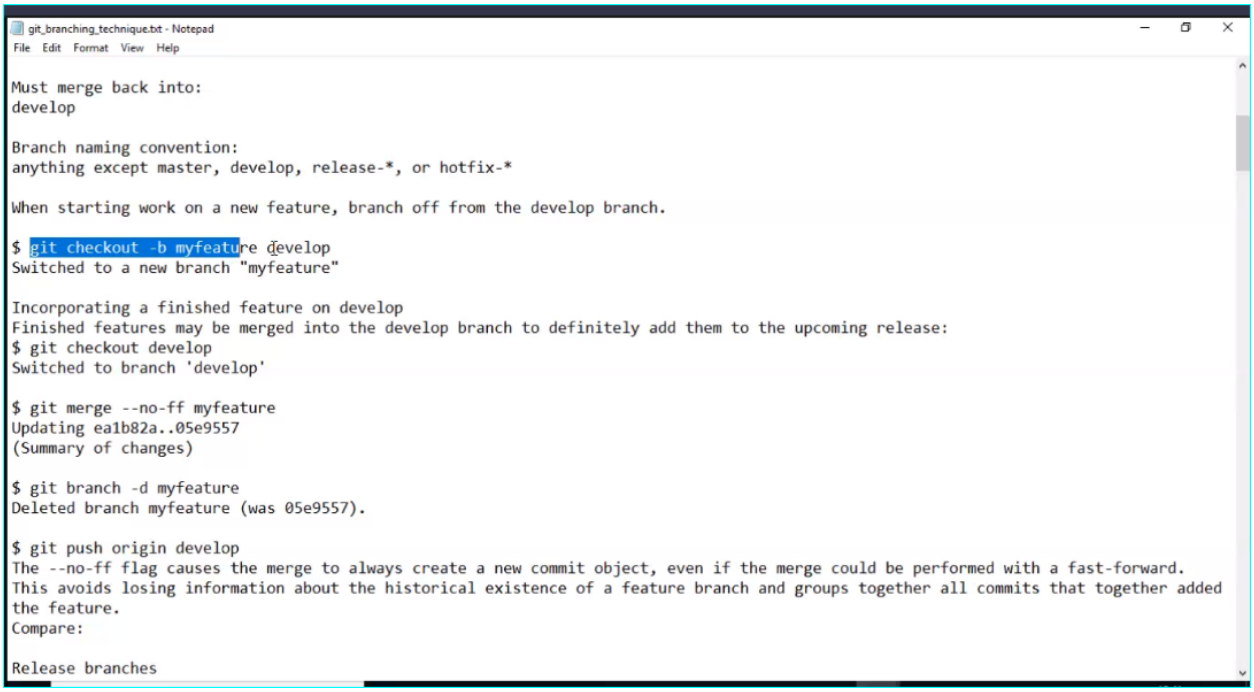
```
testing
```

$ git branch -d testing

GitFlow Workflow



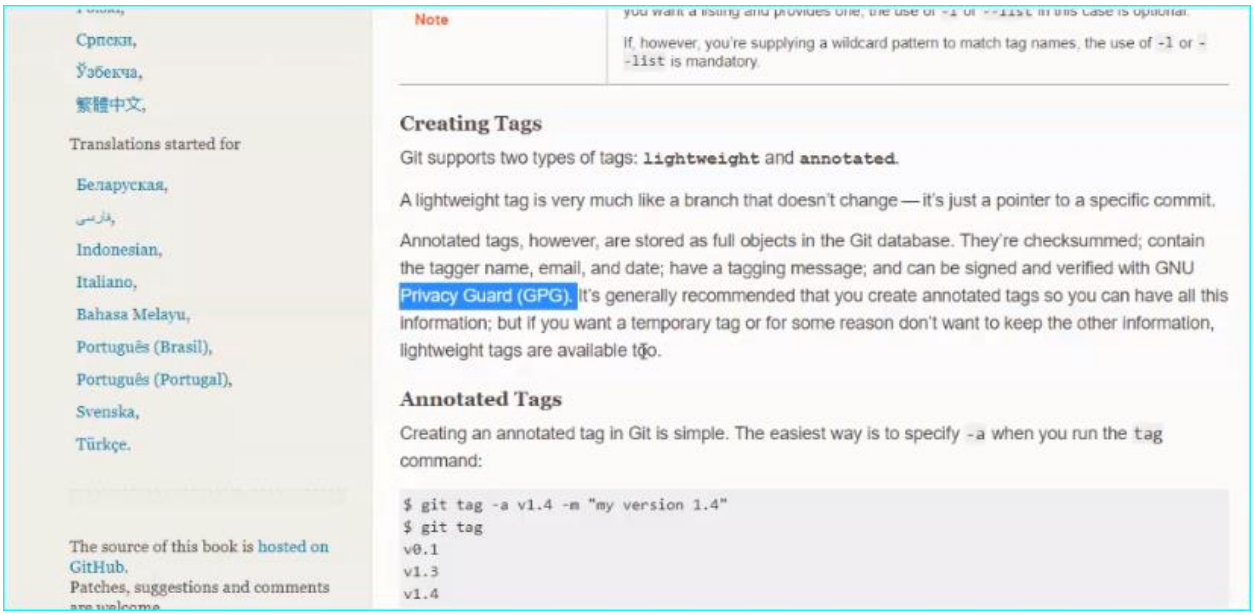

git_branching_technique.txt - Notepad
File Edit Format View Help
Gitflow Workflow
==========
The central repo holds two main branches with an infinite lifetime:(long lived)
• master
• develop
Master branch - HEAD --> origin/master to be the main branch where the source code of HEAD always reflects a production-ready state.
develop (integration branch) - HEAD --> origin/develop to be the main branch where the source code of HEAD always reflects a state with the latest delivered development changes for the next release. Some would call this the “integration branch”.
The different types of branches we may use are: (shortlived)
• Feature branches
• Release branches
• Hotfix branches
Creating a feature branch
Must branch off from:
develop
Must merge back into:
develop
Branch naming convention:
anything except master, develop, release-*, or hotfix-*


git_branching_technique.txt - Notepad
File Edit Format View Help
Must merge back into:
develop
Branch naming convention:
anything except master, develop, release-*, or hotfix-*
When starting work on a new feature, branch off from the develop branch.
$ git checkout -b myfeature develop
Switched to a new branch "myfeature"
Incorporating a finished feature on develop
Finished features may be merged into the develop branch to definitely add them to the upcoming release:
$ git checkout develop
Switched to branch 'develop'
$ git merge --no-ff myfeature
Updating ea1b82a..05e9557
(Summary of changes)
$ git branch -d myfeature
Deleted branch myfeature (was 05e9557).
$ git push origin develop
The --no-ff flag causes the merge to always create a new commit object, even if the merge could be performed with a fast-forward.
This avoids losing information about the historical existence of a feature branch and groups together all commits that together added the feature.
Compare:
Release branches

```
$ git checkout develop
Switched to branch 'develop'

$ git merge --no-ff release-1.2
Merge made by recursive.
(Summary of changes)

$ git branch -d release-1.2
Deleted branch release-1.2 (was ff452fe).


Hotfix branch (Hotfix branches are very much like release branches in that they are also meant to prepare for a new production
release,
albeit unplanned)
--------------
May branch off from:
master

Must merge back into:
develop and master

Branch naming convention:
hotfix-*

Hotfix branches are created from the master branch. For example, say version 1.2 is the current production release running
live and causing troubles due to a severe bug. But changes on develop are yet unstable. We may then branch off a hotfix branch and
start fixing the problem:
$ git checkout -b hotfix-1.2.1 master
Switched to a new branch "hotfix-1.2.1"
```

```
Release branches
-------------------
May branch off from:

develop

Must merge back into:
develop and master

Branch naming convention:
release-*

Release branches are created from the develop branch. For example, say version 1.1.5 is the current production release and we have a
big release coming up. The state of develop is ready for the “next release” and we have decided that this will become version 1.2
(rather than 1.1.6 or 2.0). So we branch off and give the release branch a name reflecting the new version number:

$ git checkout -b release-1.2 develop
Switched to a new branch "release-1.2"

$ ./bump-version.sh 1.2
Files modified successfully, version bumped to 1.2.

$ git commit -a -m "Bumped version number to 1.2"
[release-1.2 74d9424] Bumped version number to 1.2
1 files changed, 1 insertions(+), 1 deletions(-)

Finishing a release branch
--------------------------
$ git checkout master
Switched to branch 'master'
```

```
$ git checkout -b release-1.2 develop
Switched to a new branch "release-1.2"

$ ./bump-version.sh 1.2
Files modified successfully, version bumped to 1.2.

$ git commit -a -m "Bumped version number to 1.2"
[release-1.2 74d9424] Bumped version number to 1.2
1 files changed, 1 insertions(+), 1 deletions(-)

Finishing a release branch
--------------------------
$ git checkout master
Switched to branch 'master'

$ git merge --no-ff release-1.2
Merge made by recursive.
(Summary of changes)

$ git tag -a 1.2  -- annotation  -- labelling a release
The release is now done, and tagged for future reference.

we need to merge those back into develop, though. In Git:

$ git checkout develop
Switched to branch 'develop'

$ git merge --no-ff release-1.2
Merge made by recursive.
(Summary of changes)
```



Српски,
Ўзбекча,
繁體中文,

Translations started for

Беларуская,
فارسی,
Indonesian,
Italiano,
Bahasa Melayu,
Português (Brasil),
Português (Portugal),
Svenska,
Türkçe.

The source of this book is hosted on GitHub.
Patches, suggestions and comments are welcome.

Note

you want a listing and provides one, the use of -l or --list in this case is optional.

If, however, you're supplying a wildcard pattern to match tag names, the use of -l or --list is mandatory.

Creating Tags

Git supports two types of tags: **lightweight** and **annotated**.

A lightweight tag is very much like a branch that doesn't change — it's just a pointer to a specific commit.

Annotated tags, however, are stored as full objects in the Git database. They're checksummed; contain the tagger name, email, and date; have a tagging message; and can be signed and verified with GNU Privacy Guard (GPG). It's generally recommended that you create annotated tags so you can have all this information; but if you want a temporary tag or for some reason don't want to keep the other information, lightweight tags are available too.

Annotated Tags

Creating an annotated tag in Git is simple. The easiest way is to specify `-a` when you run the `tag` command:

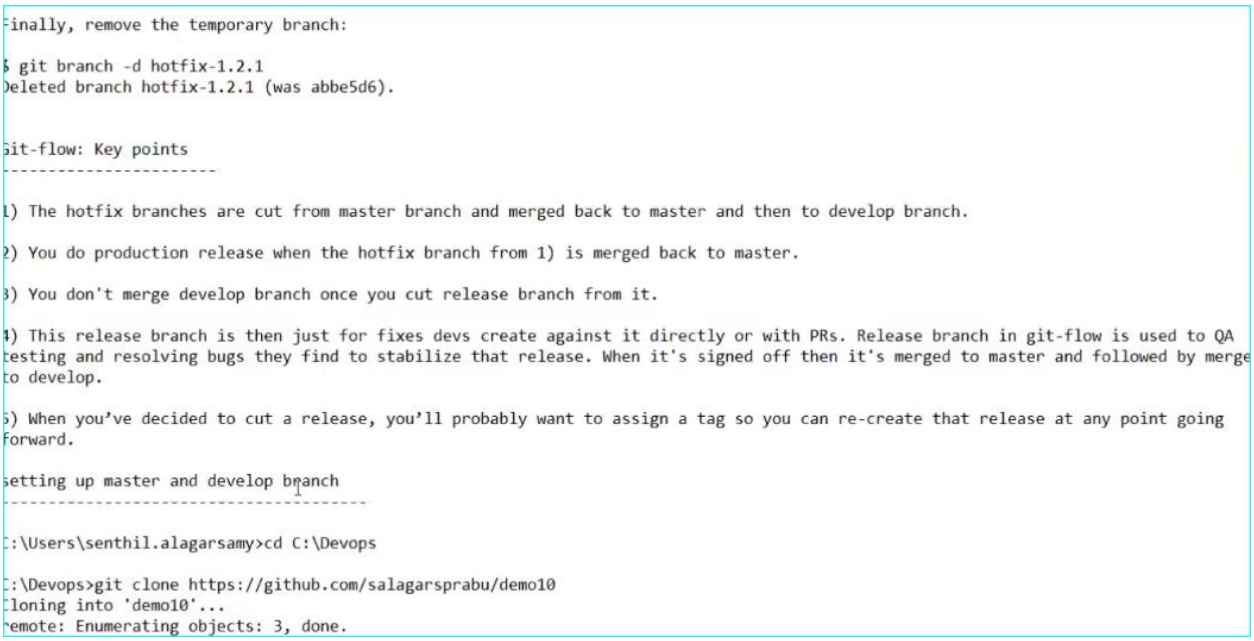
```
$ git tag -a v1.4 -m "my version 1.4"
$ git tag
v0.1
v1.3
v1.4
```

Your microphone is muted.

```
Gitflow Workflow
================

The central repo holds two main branches with an infinite lifetime:(long lived)
•       master
•       develop

Master branch -                 HEAD --> origin/master to be the main branch where the source code of HEAD always
                                reflects a production-ready state.

develop (integration branch) - HEAD --> origin/develop to be the main branch where the source code of HEAD always
                                reflects a state with the latest delivered development changes for the next release.
                                Some would call this the “integration branch”.


The different types of branches we may use are: (shortlived)
•       Feature branches
•       Release branches
•       Hotfix branches

Creating a feature branch
--------------------------
May branch off from:
develop

Must merge back into:
develop

Branch naming convention:
anything except master, develop, release-*, or hotfix-*
```

```
develop

Must merge back into:
develop and master

Branch naming convention:
release-*

Release branches are created from the develop branch. For example, say version 1.1.5 is the current production release and we have a
big release coming up. The state of develop is ready for the “next release” and we have decided that this will become version 1.2
(rather than 1.1.6 or 2.0). So we branch off and give the release branch a name reflecting the new version number:

$ git checkout -b release-1.2 develop
Switched to a new branch "release-1.2"

$ ./bump-version.sh 1.2
Files modified successfully, version bumped to 1.2.

$ git commit -a -m "Bumped version number to 1.2"
[release-1.2 74d9424] Bumped version number to 1.2
1 files changed, 1 insertions(+), 1 deletions(-)

Finishing a release branch
--------------------------
$ git checkout master
Switched to branch 'master'

$ git merge --no-ff release-1.2
Merge made by recursive.
(Summary of changes)
```

```
Finally, remove the temporary branch:

$ git branch -d hotfix-1.2.1
Deleted branch hotfix-1.2.1 (was abbe5d6).


Git-flow: Key points
-----------------------

1) The hotfix branches are cut from master branch and merged back to master and then to develop branch.

2) You do production release when the hotfix branch from 1) is merged back to master.

3) You don't merge develop branch once you cut release branch from it.

4) This release branch is then just for fixes devs create against it directly or with PRs. Release branch in git-flow is used to QA
testing and resolving bugs they find to stabilize that release. When it's signed off then it's merged to master and followed by merge
to develop.

5) When you've decided to cut a release, you'll probably want to assign a tag so you can re-create that release at any point going
forward.

setting up master and develop branch
---------------------------------------

C:\Users\senthil.alagarsamy>cd C:\Devops

C:\Devops>git clone https://github.com/salagarsprabu/demo10
Cloning into 'demo10'...
remote: Enumerating objects: 3, done.
```


Finally, remove the temporary branch:

$ git branch -d hotfix-1.2.1
Deleted branch hotfix-1.2.1 (was abbe5d6).

Git-flow: Key points
----

1) The hotfix branches are cut from master branch and merged back to master and then to develop branch.

2) You do production release when the hotfix branch from 1) is merged back to master.

3) You don't merge develop branch once you cut release branch from it.

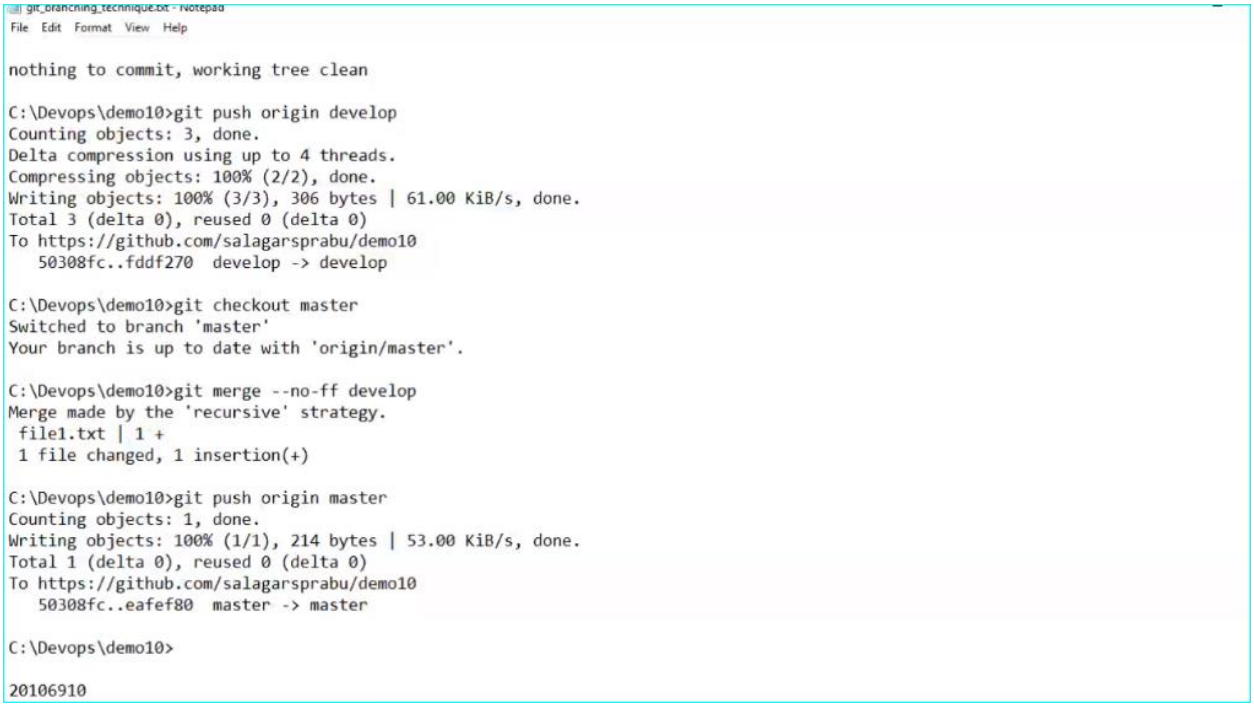
4) This release branch is then just for fixes devs create against it directly or with PRs. Release branch in git-flow is used to QA
testing and resolving bugs they find to stabilize that release. When it's signed off then it's merged to master and followed by merge
to develop.

5) When you've decided to cut a release, you'll probably want to assign a tag so you can re-create that release at any point going
forward.

setting up master and develop branch

C:\Users\senthil.alagarsamy>cd C:\Devops

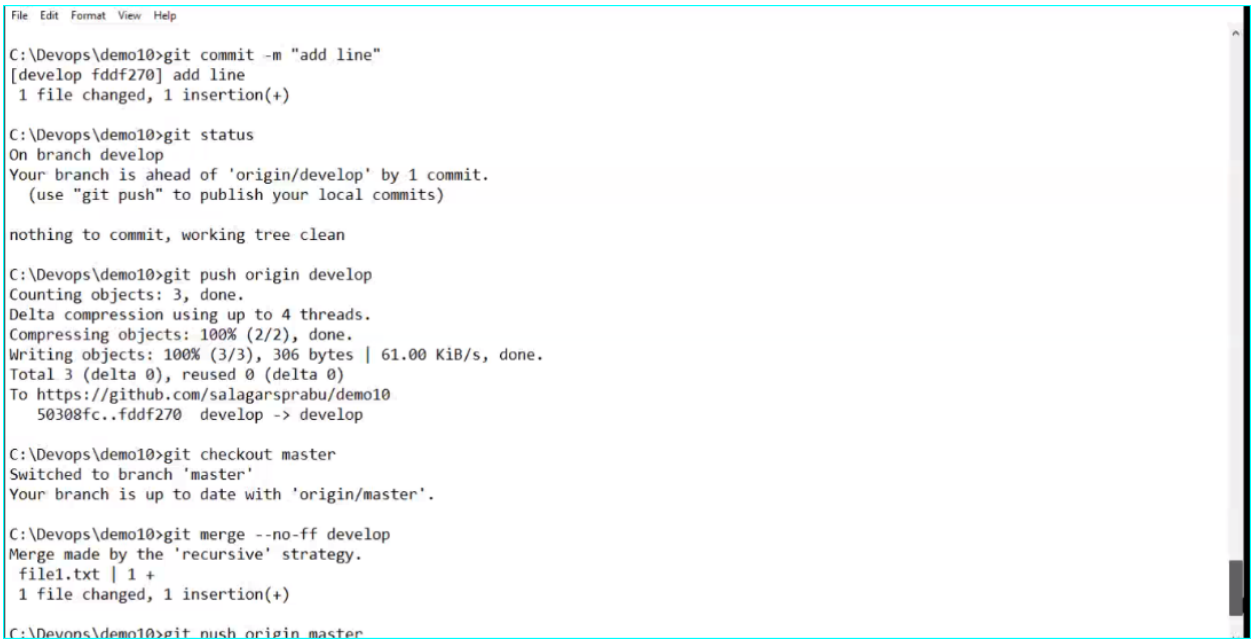
C:\Devops>git clone https://github.com/salagarsprabu/demo10
Cloning into 'demo10'...
remote: Enumerating objects: 3, done.



git_branching_technique.txt - Notepad

File Edit Format View Help

```
nothing to commit, working tree clean

C:\Devops\demo10>git push origin develop
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 306 bytes | 61.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/salagarsprabu/demo10
   50308fc..fddf270  develop -> develop

C:\Devops\demo10>git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.

C:\Devops\demo10>git merge --no-ff develop
Merge made by the 'recursive' strategy.
 file1.txt | 1 +
 1 file changed, 1 insertion(+)

C:\Devops\demo10>git push origin master
Counting objects: 1, done.
Writing objects: 100% (1/1), 214 bytes | 53.00 KiB/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To https://github.com/salagarsprabu/demo10
   50308fc..eafef80  master -> master

C:\Devops\demo10>

20106910
```



File Edit Format View Help

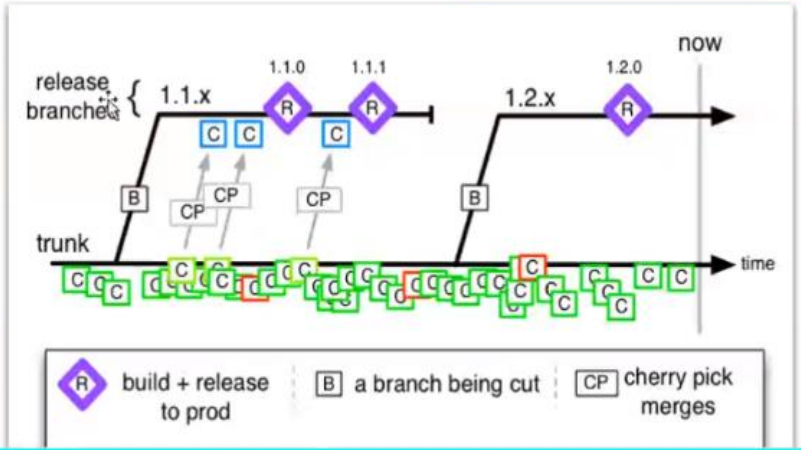
```
C:\Devops\demo10>git commit -m "add line"
[develop fddf270] add line
 1 file changed, 1 insertion(+)

C:\Devops\demo10>git status
On branch develop
Your branch is ahead of 'origin/develop' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean

C:\Devops\demo10>git push origin develop
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 306 bytes | 61.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/salagarsprabu/demo10
   50308fc..fddf270  develop -> develop

C:\Devops\demo10>git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.

C:\Devops\demo10>git merge --no-ff develop
Merge made by the 'recursive' strategy.
 file1.txt | 1 +
 1 file changed, 1 insertion(+)

C:\Devops\demo10>git push origin master
```

git_branching_technique.txt - Notepad

File Edit Format View Help

```
$ git branch -d hotfix-1.2.1
Deleted branch hotfix-1.2.1 (was abbe5d6).


Git-flow: Key points
------------------------

1) The hotfix branches are cut from master branch and merged back to master and then to develop branch.

2) You do production release when the hotfix branch from 1) is merged back to master.

3) You don't merge develop branch once you cut release branch from it.

4) This release branch is then just for fixes devs create against it directly or with PRs. Release branch in git-flow is used to QA
testing and resolving bugs they find to stabilize that release. When it's signed off then it's merged to master and followed by merge
to develop.

5) When you've decided to cut a release, you'll probably want to assign a tag so you can re-create that release at any point going
forward.

setting up master and develop branch
---------------------------------------

C:\Users\senthil.alagarsamy>cd C:\Devops

C:\Devops>git clone https://github.com/salagarsprabu/demo10
Cloning into 'demo10'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
```

TRUNK-BASED DEVELOPMENT



All project teams develop on a s
Branches happen for releases c
a tag was not enough), with che
TO the release branch for defec

Teams probably use **Feature To**
and **Branch by Abstraction** too
fit that concurrent development
consecutive releases goal.

```
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: added staged command
```

This operation works by going to the common ancestor of the two branches (the one you're on and the one you're rebasing onto), getting the diff introduced by each commit of the branch you're on, saving those diffs to temporary files, resetting the current branch to the same commit as the branch you are rebasing onto, and finally applying each change in turn.

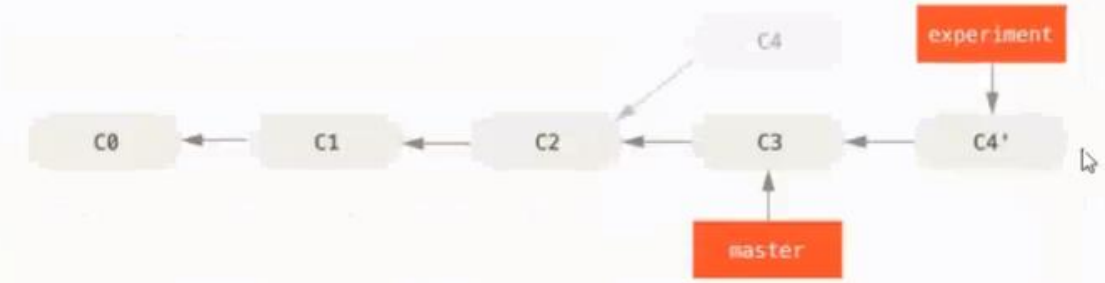


Figure 37. Rebasing the change introduced in `C4` onto `C3`

At this point, you can go back to the `master` branch and do a fast-forward merge.

```
$ git checkout master
$ git merge experiment
```

experiment

https://help.github.com/en/articles/resolving-a-merge-conflict-using-the-command-line#competing-line-change-merge-conflicts
Ctrl+Click to follow link

Resolving a merge conflict using the command line

a) Competing line change merge conflicts

For example, if you and another person both edited the file *merge.txt* on the same lines in different branches of the same Git repository, you'll get a merge conflict error when you try to merge these branches. You must resolve this merge conflict with a new commit before you can merge these branches.

$ mkdir git-merge-test2

$ cd git-merge-test2

$ git init .

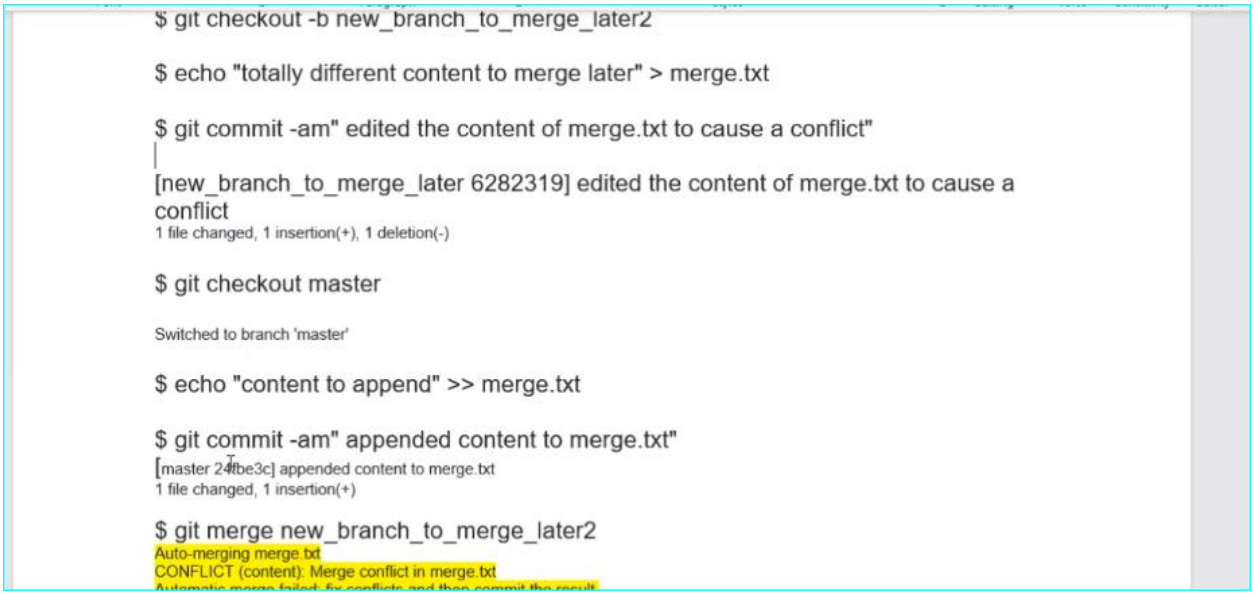
$ echo "this is some content to mess with" > merge.txt

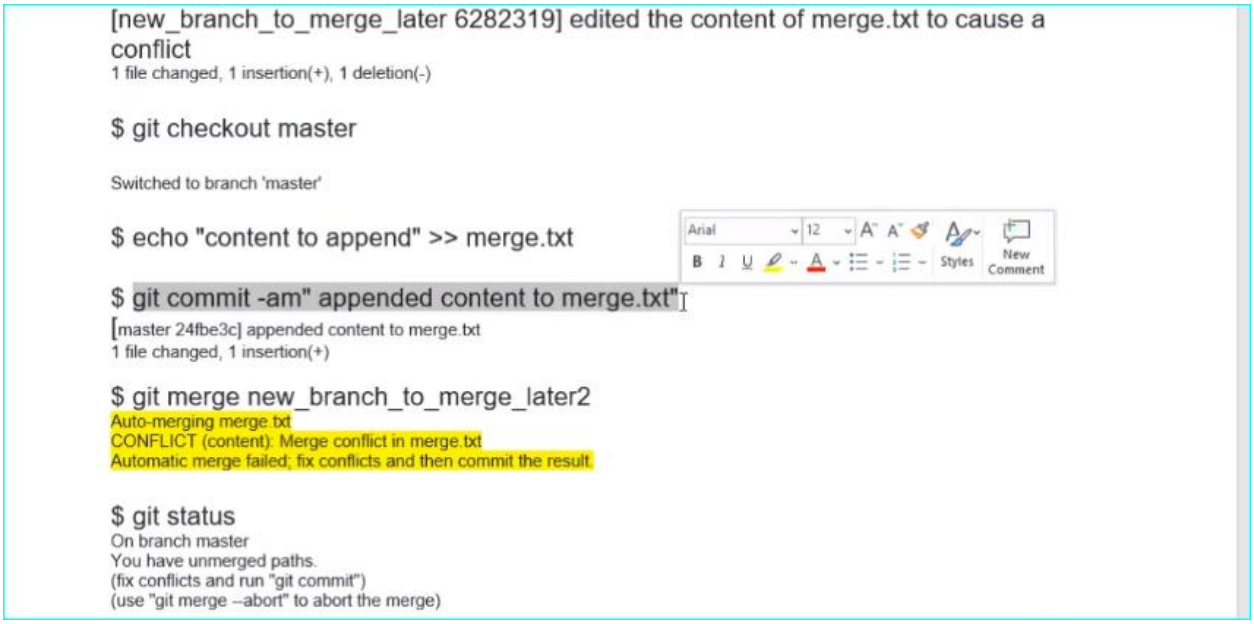
```
$ git add merge.txt

$ git commit -am" we are commiting the inital content"
[master (root-commit) d48e74c] we are commiting the inital content
1 file changed, 1 insertion(+)
create mode 100644 merge.txt

$ git checkout -b new_branch_to_merge_later2

$ echo "totally different content to merge later" > merge.txt

$ git commit -am" edited the content of merge.txt to cause a conflict"

[new_branch_to_merge_later 6282319] edited the content of merge.txt to cause a
conflict
1 file changed, 1 insertion(+), 1 deletion(-)

$ git checkout master

Switched to branch 'master'
```

```
$ mkdir git-merge-test2

$ cd git-merge-test2

$ git init .

$ echo "this is some content to mess with" > merge.txt

$ git add merge.txt

$ git commit -am" we are commiting the inital content"
[master (root-commit) d48e74c] we are commiting the inital content
1 file changed, 1 insertion(+)
create mode 100644 merge.txt

$ git checkout -b new_branch_to_merge_later2

$ echo "totally different content to merge later" > merge.txt

$ git commit -am" edited the content of merge.txt to cause a conflict"
```

```
$ git init .

$ echo "this is some content to mess with" > merge.txt

$ git add merge.txt

$ git commit -am" we are commiting the inital content"
[master (root-commit) d48e74c] we are commiting the inital content
1 file changed, 1 insertion(+)
create mode 100644 merge.txt

$ git checkout -b new_branch_to_merge_later2

$ echo "totally different content to merge later" > merge.txt

$ git commit -am" edited the content of merge.txt to cause a conflict"

[new_branch_to_merge_later 6282319] edited the content of merge.txt to cause a
conflict
1 file changed, 1 insertion(+), 1 deletion(-)

$ git checkout master
```

```
$ git commit -am" we are commiting the inital content"
[master (root-commit) d48e74c] we are commiting the inital content
1 file changed, 1 insertion(+)
create mode 100644 merge.txt

$ git checkout -b new_branch_to_merge_later2

$ echo "totally different content to merge later" > merge.txt

$ git commit -am" edited the content of merge.txt to cause a conflict"

[new_branch_to_merge_later 6282319] edited the content of merge.txt to cause a
conflict
1 file changed, 1 insertion(+), 1 deletion(-)

$ git checkout master

Switched to branch 'master'

$ echo "content to append" >> merge.txt

$ git commit -am" appended content to merge.txt"
```


$ git checkout -b new_branch_to_merge_later2
$ echo "totally different content to merge later" > merge.txt
$ git commit -am" edited the content of merge.txt to cause a conflict"
[new_branch_to_merge_later 6282319] edited the content of merge.txt to cause a conflict
1 file changed, 1 insertion(+), 1 deletion(-)
$ git checkout master
Switched to branch 'master'
$ echo "content to append" >> merge.txt
$ git commit -am" appended content to merge.txt"
[master 24fbe3c] appended content to merge.txt
1 file changed, 1 insertion(+)
$ git merge new_branch_to_merge_later2
Auto-merging merge.txt
CONFLICT (content): Merge conflict in merge.txt


[new_branch_to_merge_later 6282319] edited the content of merge.txt to cause a conflict
1 file changed, 1 insertion(+), 1 deletion(-)
$ git checkout master
Switched to branch 'master'
$ echo "content to append" >> merge.txt
$ git commit -am" appended content to merge.txt"
[master 24fbe3c] appended content to merge.txt
1 file changed, 1 insertion(+)
$ git merge new_branch_to_merge_later2
Auto-merging merge.txt
CONFLICT (content): Merge conflict in merge.txt
Automatic merge failed; fix conflicts and then commit the result.
$ git status
On branch master
You have unmerged paths.
(fix conflicts and run "git commit")
(use "git merge --abort" to abort the merge)
Arial
12
Styles
New Comment

```
[new_branch_to_merge_later 6282319] edited the content of merge.txt to cause a
conflict
1 file changed, 1 insertion(+), 1 deletion(-)

$ git checkout master

Switched to branch 'master'

$ echo "content to append" >> merge.txt

$ git commit -am" appended content to merge.txt"
[master 24fbe3c] appended content to merge.txt
1 file changed, 1 insertion(+)

$ git merge new_branch_to_merge_later2
Auto-merging merge.txt
CONFLICT (content): Merge conflict in merge.txt
Automatic merge failed; fix conflicts and then commit the result.

$ git status
On branch master
You have unmerged paths.
(fix conflicts and run "git commit")
(use "git merge --abort" to abort the merge)
```

```
$ echo "content to append" >> merge.txt

$ git commit -am" appended content to merge.txt"
[master 24fbe3c] appended content to merge.txt
1 file changed, 1 insertion(+)

$ git merge new_branch_to_merge_later2
Auto-merging merge.txt
CONFLICT (content): Merge conflict in merge.txt
Automatic merge failed; fix conflicts and then commit the result.

$ git status
On branch master
You have unmerged paths.
(fix conflicts and run "git commit")
(use "git merge --abort" to abort the merge)

Unmerged paths:
(use "git add <file>..." to mark resolution)

both modified:   merge.txt

$ cat merge.txt
<<<<<<< HEAD
this is some content to mess with
content to append
```

Automatic merge failed; fix conflicts and then commit the result.

```
$ git status
On branch master
You have unmerged paths.
(fix conflicts and run "git commit")
(use "git merge --abort" to abort the merge)

Unmerged paths:
(use "git add <file>..." to mark resolution)

both modified:   merge.txt

$ cat merge.txt
<<<<<<< HEAD
this is some content to mess with
content to append
=======
totally different content to merge later
>>>>>>> new_branch_to_merge_later
```

Resolve merge conflicts using the command line

this is some content to mess with

both modified: merge.txt

$ cat merge.txt
<<<<<<< HEAD
this is some content to mess with
content to append
=======
totally different content to merge later
>>>>>>> new_branch_to_merge_later

Resolve merge conflicts using the command line

this is some content to mess with
content to append
totally different content to merge later

```
$ git commit -m "merged and resolved the conflict in merge.txt"
```

Removed file merge conflicts

Senthil Alagarsamy@LP-5CD730817M MINGW64 /c/Devops/git-merge-test (master)

```
both modified:   merge.txt

$ cat merge.txt
<<<<<<< HEAD
this is some content to mess with
content to append
=======
totally different content to merge later
>>>>>>> new_branch_to_merge_later
```

Resolve merge conflicts using the command line

```
this is some content to mess with
content to append
totally different content to merge later

$ git commit -m "merged and resolved the conflict in merge.txt"
```

Removed file merge conflicts

```
create mode 100644 merge.txt

$ git checkout -b new_branch_to_merge_later2

$ echo "totally different content to merge later" > merge.txt

$ git commit -am" edited the content of merge.txt to cause a conflict"

[new_branch_to_merge_later 6282319] edited the content of merge.txt to cause a
conflict
1 file changed, 1 insertion(+), 1 deletion(-)

$ git checkout master

Switched to branch 'master'

$ echo "content to append" >> merge.txt

$ git commit -am" appended content to merge.txt"
[master 24fbe3c] appended content to merge.txt
1 file changed, 1 insertion(+)
```

totally different content to merge later

>>>>>>> new_branch_to_merge_later

Resolve merge conflicts using the command line

this is some content to mess with
content to append
totally different content to merge later

$ git commit -m "merged and resolved the conflict in merge.txt"

Removed file merge conflicts

Senthil.Alagarsamy@LP-5CD73081ZM MINGW64 /c/Devops/git-merge-test (master)
$ mkdir git-test4

Senthil.Alagarsamy@LP-5CD73081ZM MINGW64 /c/Devops/git-merge-test (master)
$ cd git-test4

Senthil.Alagarsamy@LP-5CD73081ZM MINGW64 /c/Devops/git-merge-test/git-test4 (master)
$ git init .
Initialized empty Git repository in C:/Devops/git-merge-test/git-test4/.git/

Resolving a merge conflict using the command line

a) **Competing line change merge conflicts**

For example, if you and another person both edited the file *merge.txt* on the same lines in different branches of the same Git repository, you'll get a merge conflict error when you try to merge these branches. You must resolve this merge conflict with a new commit before you can merge these branches.

$ mkdir git-merge-test2

$ cd git-merge-test2

$ git init .

$ echo "this is some content to mess with" > merge.txt

$ git add merge.txt

```
$ git init .

$ echo "this is some content to mess with" > merge.txt

$ git add merge.txt

$ git commit -am" we are commiting the inital content"
[master (root-commit) d48e74c] we are commiting the inital content
1 file changed, 1 insertion(+)
create mode 100644 merge.txt

$ git checkout -b nẽw_branch_to_merge_later2

$ echo "totally different content to merge later" > merge.txt

$ git commit -am" edited the content of merge.txt to cause a conflict"

[new_branch_to_merge_later 6282319] edited the content of merge.txt to cause a
conflict
1 file changed, 1 insertion(+), 1 deletion(-)

$ git checkout master
```

```
$ echo "content to append" >> merge.txt

$ git commit -am" appended content to merge.txt"
[master 24fbe3c] appended content to merge.txt
1 file changed, 1 insertion(+)

$ git merge new_branch_to_merge_later2
Auto-merging merge.txt
CONFLICT (content): Merge conflict in merge.txt
Automatic merge failed; fix conflicts and then commit the result.

$ git status
On branch master
You have unmerged paths.
(fix conflicts and run "git commit")
(use "git merge --abort" to abort the merge)

Unmerged paths:
(use "git add <file>..." to mark resolution)

both modified:   merge.txt

$ cat merge.txt
<<<<<<< HEAD
this is some content to mess with
```