Introduction to Git

Trivera Tech
TECHNOLOGY TRAINING

Table of Contents

Git Basics

Starting a Git Repository

In this lesson, you will learn about:

- Initializing a repository
- Cloning an existing repository
- Adding and committing files
- Pushing commits on remote repositories

Configuring Git

- Before you start working on Git, you have to configure your name and e-mail by using the following commands:

```
Erik@local:~$git config --global user.name  "Erik"
Erik@local:~$git config --global user.email erik@domain.com
```

Initializing a new repository

- If you want to create a repository in an existing project, just type the following command line:

```
Erik@local:~$ cd myProject
Erik@local:~/myProject$ git init .
```

- Otherwise, you have to create an empty directory and type git init inside it, as shown:

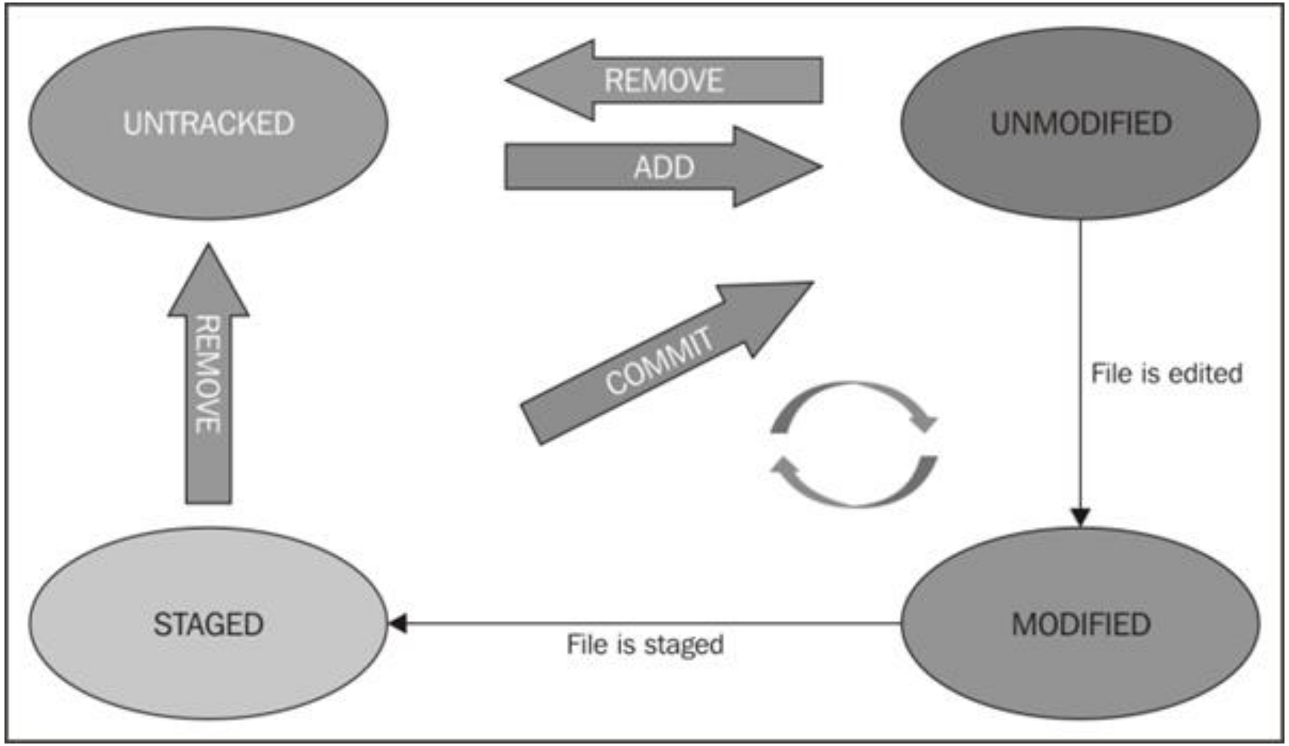
```
Erik@local:~$ mkdir myProject
Erik@local:~$ cd myProject
Erik@local:~/myProject$ git init
```

Cloning an existent repository

- For example, if you want to clone a Symfony2 repository, type this line to clone it using myProjectName as the folder name:

```
Erik@local:~/myProject$ git clone
https://github.com/symfony/symfony.gitmyProjectName
Initialized empty Git repository in /var/www/myProjectName/.git/
remote: Counting objects: 7820, done.
remote: Compressing objects: 100% (2490/2490), done.
remote: Total 7820 (delta 4610), reused 7711 (delta 4528)
Receiving objects: 100% (7820/7820), 1.40 MiB | 479 KiB/s, done.
Resolving deltas: 100% (4610/4610), done.
Checking out files: 100% (565/565), done.
```

Working with the repository



Adding a file

- When you start an empty repository and add a file, it will be in the untracked state, which means that it isn't in the Git repository.

- To track a file, you have to execute this command line:

```
Erik@local:~/myProject$ touch MyFileName.txt
Erik@local:~/myProject$ echo "test" > MyFileName.txt
Erik@local:~/myProject$ git add MyFileName.txt
```

Adding a file

- If you want to add all files because you already have something inside the directory while you create the repository, add a period (.) just after git add to specify to take all files inside the current directory:

```
Erik@local:~/myProject$ echo "hello" > MyFile2.txt
Erik@local:~/myProject$ echo "hello" > MyFile3.txt
Erik@local:~/myProject$ git add .
```

Committing a file

- The commit command is local to your own repository, nobody except you can see it.

- The commit command line offers various options. For example, you can commit a file, as shown in the following example:

```
Erik@local:~/myProject$ git commit -m 'This message explains the changes' MyFileName.txt
```

- To commit everything, use the following command:

```
Erik@local:~/myProject$ git commit -m 'My commit message'
```

Committing a file

- You will create a new commit object in the Git repository, This commit is referenced by an SHA-1 checksum and includes various data (content files, content directories, the commit history, the committer, and so on).
- You can show this information by executing the following command line:

```
Erik@local:~/myProject$ git log
```

- It will display something similar to the following:

```
Commit f658e5f22afe12ce75cde1h671b58d6703ab83f5
Author: Eric Pidoux <contact@eric-pidoux.com>
Date:  Mon Jun 2 22:54:04 2014 +0100
My commit message
```

TriveraTech
TECHNOLOGY TRAINING

Pushing a file

- Once committed, you can push the files in the remote repository.
- It can be on a bare repository, using init with the git init --bare command, so just type the following command:

```
Erik@local:~/myProject$ git push /home/erik/remote-repository.git
```

Pushing a file

- If you create a remote repository on another server, you have to configure your local Git repository.
- If you use Git 2.0 or later, the previous command will print out something like this on the screen:

```
Warning: push.default is unset; its implicit value is changing in
Git 2.0 from 'matching' to 'simple'. To squelch this message
and maintain the current behavior after the default changes, use:
gitconfig --global push.default matching
To squelch this message and adopt the new behavior now, use:
gitconfig --global push.default simple
```

Pushing a file

- Firstly, check if a remote repository is defined:

```
Erik@local:~/myProject$ git remote
```

- If it's not, define the remote repository named origin:

```
Erik@local:~/myProject$ git remote add origin
http://github.com/myRepoAddress.git
```

- Now, push the changes using the following command:

```
Erik@local:~/myProject$ git push -u origin master
```

Removing a file

- If you don't want a file anymore, there are two ways to remove it:

- Delete the file manually and commit the changes.
- This will delete the file locally and on the repository. Use the following command line

```
Erik@local:~/myProject$ git commit -m 'delete this file'
```

- Delete the file only through Git:

```
Erik@local:~/myProject$ git rm --cached MyFileName.txt
```

Checking the status

- There is a way to display the working tree status, that is, the files that have changed and those that need to be pushed, and of course, there is a way to display the conflicts:

```
Erik@local:~/myProject$ git status
```

- If everything is correct and up to date, you will get this result:

```
Erik@local:~/myProject$ git status
# On branch master
nothing to commit, working directory clean
```

Checking the status

- If you add a file, Git will warn you to track it by using the git add command:

```
Erik@local:~/myProject$ touch text5.txt
Erik@local:~/myProject$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   text5.txt
nothing added to commit but untracked files present (use "git add" to track)
```

TriveraTech TECHNOLOGY TRAINING

```
Erik@local:~/myProject$ echo "I am changing this file" > MyFile2.txt
Erik@local:~/myProject$ git status
# On branch master
# Changes to be committed:
#   (use "gitreset HEAD<file>..." to unstage)
#
#   new file: text5.txt
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
# modified: MyFile2.txt
#
```

Checking the status

Checking the status

- If you add text5.txt using the git add command, you will notice the following changes:

```
Erik@local:~/myProject$ git add MyFile2.txt
Erik@local:~/myProject$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   text5.txt
#   modified:   MyFile2.txt
#
```

Ignoring files

- Git can easily ignore some files or folders from your working tree.
- For example, consider a website on which you are working, and there is an upload folder that you might not push on the repository to avoid having test images in your repository.
- To do so, create a .gitignore file inside the root of your working tree:

```
Erik@local:~/myProject$ touch .gitignore
```

- Then, add this line in the file; it will untrack the upload folder and its contents:

```
upload
```

Ignoring files

- If the file is already pushed on the repository, the file is tracked by Git.

- To remove it, you will have to use the gitrmcommand line by typing this:

```
Erik@local:~/myProject$ git rm --cached MyFileName.txt
```

Summary

- In this lesson, we saw the basics of Git: how to create a Git repository, how to put content in it, and how to push data to a remote repository.

- In the next lesson, we will see how to use Git with a team and manage all interactions with a remote repository.

Git on the Server

Why Create Your Own Git Server?

Improved security

- GitHub, and other services, provide good protective parameters but accidents can happen, and have happened in the past.
- In general, having any sensitive data on third-party servers should be avoided where possible.

More control

- Managing your Git server means you can customize and optimize the hosting environment for your needs specifically.
- Many of the premium options you get on your own server for free will cost you extra on a hosted service.

Managing SSH Keys

- Using SSH keys for interacting with Git means you don’t have to manage passwords.
- If you have a team member who wants to contribute to the project, you can simply upload their public key to the Git server.
- When their work is done, or they are no longer contributing to the project, all you have to do is remove their key and they will no longer have access to the server.

How the Process Works

Here is the basic rundown of the Git server process:

- Create a “git” user and group
- Upload a public SSH key to the “git” user which will allow you (or others) to log in as the “git” user
- Create a bare repository on the server
- Add the remote repository (the one just created on the server) to a local Git project
- Push changes to the remote repository

The Theory Behind the Git User Account

- Why create a “git” Linux user? Can you just use “root”? The git user is required because we will be pushing content to our server using SSH.

- In order to log in with SSH, a user is required. You could use the root user, but this poses a security hazard, It’s considered a best practice to limit access to your root user account.

- By creating a specialized git user for this process, you could—if needed—open your project to more contributors without granting access to the root user account.

Creating the “Git” User

- First things first, it’s time to create a “git” user account with which to do all your git-related work.
- Remember, you can title this user account anything you want.
- Naming the account “git” is merely a simple way to indicate what this user account was created to do.
- Remember, as you are logging into your cloud VPS account, you are using the root user. If you are not using your root user, you will need to prepend these commands with “sudo.”
- Create the Git user with the adduser command:

```
adduser git
```

Creating the “Git” User

- Now that you have created the git user, you can assume the role by using the su command:

```
su git
```

- You will now need to create a .ssh directory in the “home” folder for the git user. In order to do that, you must make sure you are in the home directory, so run a cd command to make sure:

```
Cd
```

- Now, create the .ssh directory:

```
mkdir .ssh
```

Creating the “Git” User

- Next, make the directory only accessible by the git user:

```
chmod 700 .ssh
```

- Now you have an .ssh directory to hold your “authorized_keys” file, which is used to authenticate login requests. It’s time to create that file now:

```
touch .ssh/authorized_keys
```

- And, likewise, adjust the privileges to make the file accessible only to the git user:

```
chmod 600 .ssh/authorized_keys
```

Uploading Keys to the Git User

- When you hear about "uploading" or "managing" keys, you are basically appending these keys to the .ssh/authorized_keys file.
- You can put as many public keys as you need in this file, and there are a few different ways to get to them there.
- One way is to simply copy and paste the key from your local computer to the server.
- There is also an SSH command that will copy the key to the server for you:

```
ssh-copy-id -i ~/.ssh/key user@<hostname, IP address, or domain name>
```

Creating the "Bare" Git Repositories

- Once you've decided which directory will house your remote repositories, you can start creating repositories first as directories ending with a .git extension.
- The extension is important, as it distinguishes the directories containing Git repositories from regular directories.
- For example, to create a repository called "website," create a directory like so:

```
mkdir website.git
```

- Then, enter the directory with the cd command:

```
cd website.git
```

- Once inside the directory, create a "bare" Git repository:

```
git init --bare
```

Add Your Remote Repository

- Now, you just need to make your local Git installation aware of the new repository:

```
git remote add origin git@server:/path/to/website.git
```

- Let us break this command a little bit.
- The git remote add command initiates the addition of a repository. The “origin” part names the new repository.
- You can put any name you want, you don’t have to use “origin,” but it is merely a conventional naming scheme.

Add Your Remote Repository

- The rest of the command specifies the precise server file path to the bare Git repository you created.
- The final step is to push your project to the new repository.
- For example, if you are pushing the “master” branch, the command will look like this:

```
git push origin master
```

Git Branching

Working in a Team Using Git

This lesson introduces the aim of Git: team work.

In the examples of this lesson, we will use the following conditions:

- Three programmers working together on a simple website project
- They install Git, but nothing is created
- They own a dedicated server with Git, SSH, and GitLab installed on it

Creating a server repository

- Git can use four protocols to transport data:

- Local
- Secure Shell (SSH)
- Git
- HTTP

We will see how and when to use these protocols. We will also distinguish between the pros and cons of each protocol.

Creating a server repository

- For all protocols, we have to create the bare repository by executing these lines on the server's command lines.

```
Erik@server:~$ mkdir webproject
#Create the folder
Erik@server:~$ cd webproject
#go inside it
Erik@server:~/webproject$ git init --bare
Initialized empty Git repository in /home/erik/webproject
```

Local

- The local protocol is the basic protocol; the remote repository is a local directory.
- This protocol is used if all members have access to the remote repository (through NFS, for example).
- Now, every programmer has to clone it in local:

```
Erik@local:~$ git clone /opt/git/webproject.git
```

Local

- For example, we assume that Jim, one of the programmers, has already written some code lines. Jim has to initialize a local Git repository inside the directory and set a remote location for the bare repository:

```
Jim@local:~/webproject$ git init
Jim@local:~/webproject$ git remote add origin /opt/git/webproject.git
```

SSH

- Secure Shell (SSH) is the most used protocol, especially if, as in our example, the remote repository is on a distant server.

- Now, every programmer has to first clone it in local:

```
Erik@local:~$ git clone ssh://username@server/webproject.git
```

SSH

- Using the SSH protocol, programmers have to install their SSH keys on the remote repository in order to push to and pull from it.

- Otherwise, they have to specify the password on each remote command.

- For our Jim's case:

```
Jim@local:~/webproject$ git init
Jim@local:~/webproject$ git remote add origin
ssh://username@server/webproject.git
```

HTTPS

- The HTTPS protocol is the easiest to set up. Anyone who has access to the web server can clone it.

- The programmers start to clone it in local:

```
Erik@local:~$ git clone https://server/webproject.git
```

- And, of course, in our Jim's case:

```
Jim@local:~/webproject$ git init .
Jim@local:~/webproject$ git remote add origin
http://server/webproject.git
```

Pushing data on remote repositories – Jim's case

```
Jim@local:~/webproject$ git add .
Jim@local:~/webproject$ git commit -m 'add my code'
[master (commit racine) 83fcc8a] add my code
2 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 index.html
create mode 100644 readme.txt
Jim@local:~/webproject$ git push –u origin master
Counting objects: 3, done.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 225 bytes | 0 byte/s, done.
Total 3 (delta 0), reused 0 (delta 0)
```

Pulling data from the repository

- The following command is used to pull data:

```
Erik@local:~/webproject$ git pull origin master
```

- This command will check and compare your local commit hash to the remote hash.
- If the remote is the latest, it will try to merge data with the local master branch.
- This command is the equivalent of executing git fetch (get remote data) and git merge (merge to your branch).

Creating a patch

- Let's explain what a patch is with an example, An external programmer was called by Jim to make small fixes in a part of the project, but Jim didn't want to give him access to the repository, thus preventing him from pushing data.

- So, he decides to make a patch and sends it by e-mail to Jim:

```
External@local:~/webproject$ git format-patch origin patch-
webproject.patch
```

Creating a patch

- Jim can import the patch by executing this:

```
Jim@local:~/webproject$ git apply /tmp/patch-webproject.patch
```

- This command will apply the patch, but it doesn't create commits.

- So, to generate a series of commits, use the git am command:

```
Jim@local:~/webproject$ git am /tmp/patch-webproject.patch
```

Working with branches

- The first command to know is:

```
Jim@local:~/webproject$ git branch
```

- This command will display all available local branches for the repository. Inside the given list, the current working branch has the prefix *.
- If you want to see all branches, including the remote branches, you will have to execute the following command:

```
Jim@local:~/webproject$ git branch -a
```

Creating a branch

- You can create a branch with the git branch command, This command allows you to create a new branch using a commit.
- So, Git will create the branch and populate it with the files from the given commit.
- If you don't provide a commit, it will use the last (or HEAD) commit:

```
Jim@local:~/webproject$ git branch test
```

Checking out a branch

- To start using a branch, you have to check it out.
- If you do so, Git will ignore files from other branches and prepare to listen to changes on specific files only:

```
Jim@local:~/webproject$ git checkout test
#Do some changes inside the readme.txt file
Jim@local:~/webproject$ git commit -a -m 'edit readme'
Jim@local:~/webproject$ git checkout master
```

Playing with a branch

- There are several commands useful for some features of branches. Firstly, you can easily rename a branch executing this:

```
Jim@local:~/webproject$ git branch -m old_name new_name
```

- Then, you can delete a branch:

```
Jim@local:~/webproject$ git branch -d test
```

Playing with a branch

- You might get an error if there are uncommitted changes, You can force it:

```
Jim@local:~/webproject$ git branch -D test
```

- Finally, you can push the changes of a branch to a remote repository.

- While executing the push command, you can specify the remote branch to use:

```
Jim@local:~/webproject$ gitpush origin test
```

The difference between branches

- To see the difference between two branches, you can execute this:

```
Jim@local:~/webproject$ git diff master test
```

TriveraTech TECHNOLOGY TRAINING

Tracking branches

To set up a tracking branch, execute this:

```
Jim@local:~/webproject$ git checkout -b new_branch
origin/branch_to_track
#Or you can use this
Jim@local:~/webproject$ git branch new_branch origin/master
Jim@local:~/webproject$ git branch --track new_branch origin/master
```

TriveraTech TECHNOLOGY TRAINING

Tracking branches

- Similarly, you can specify to not track a remote branch:

```
Jim@local:~/webproject$ git branch --no-track new_branch
origin/master
#You can later update this branch and track origin/master
Jim@local:~/webproject$ git branch -u origin/master new_branch
```

Deleting a branch from the remote

- Use this command if you want to delete a branch in the remote repository:

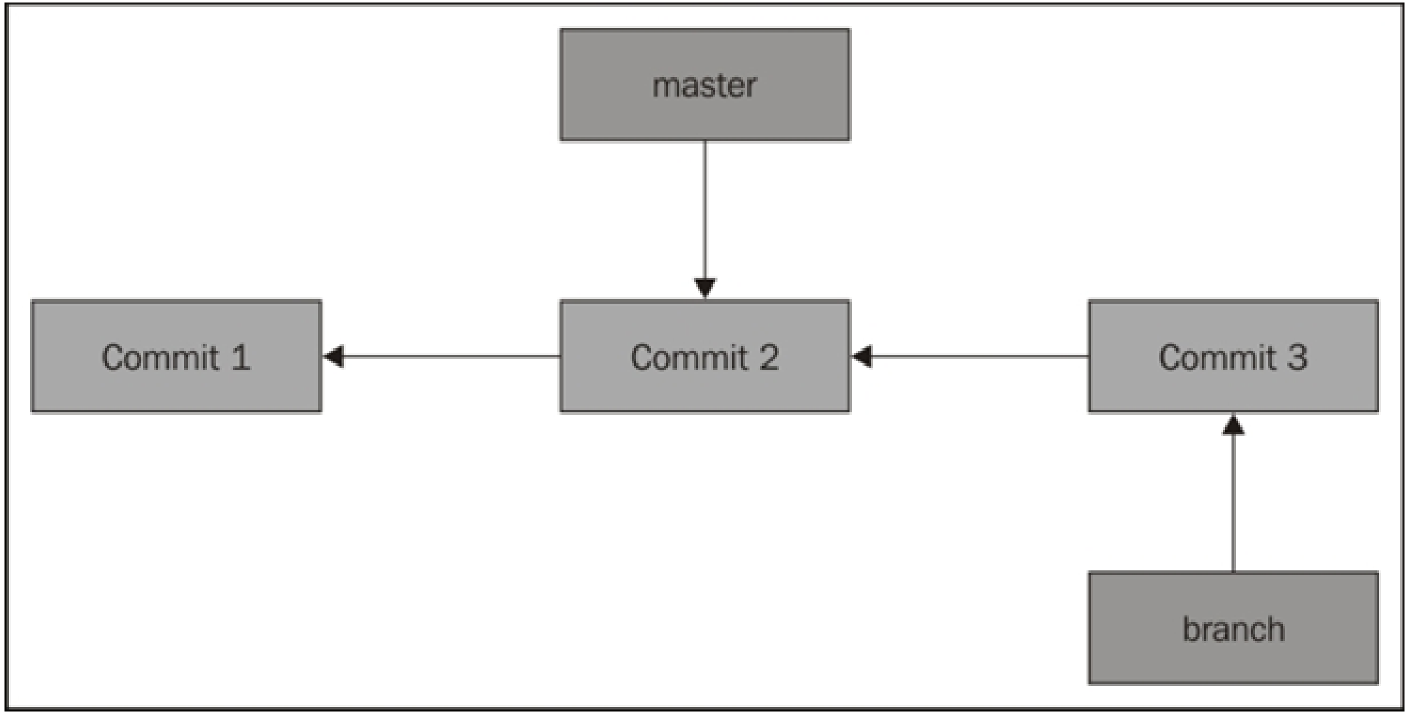
```
Erik@local:~/webproject$ git branch -d origin/test
```

Merging

- A very nice process in Git allows you to combine the changes of two branches, This is called merging.
- The git merge command performs a merge.
- You can merge changes from one branch to the current branch via the following command.
- Your local master branch is created as a tracking branch for the master branch of the origin repository:

```
Erik@local:~/webproject$ git checkout master
Erik@local:~/webproject$ git merge test
```

Fast forward merge



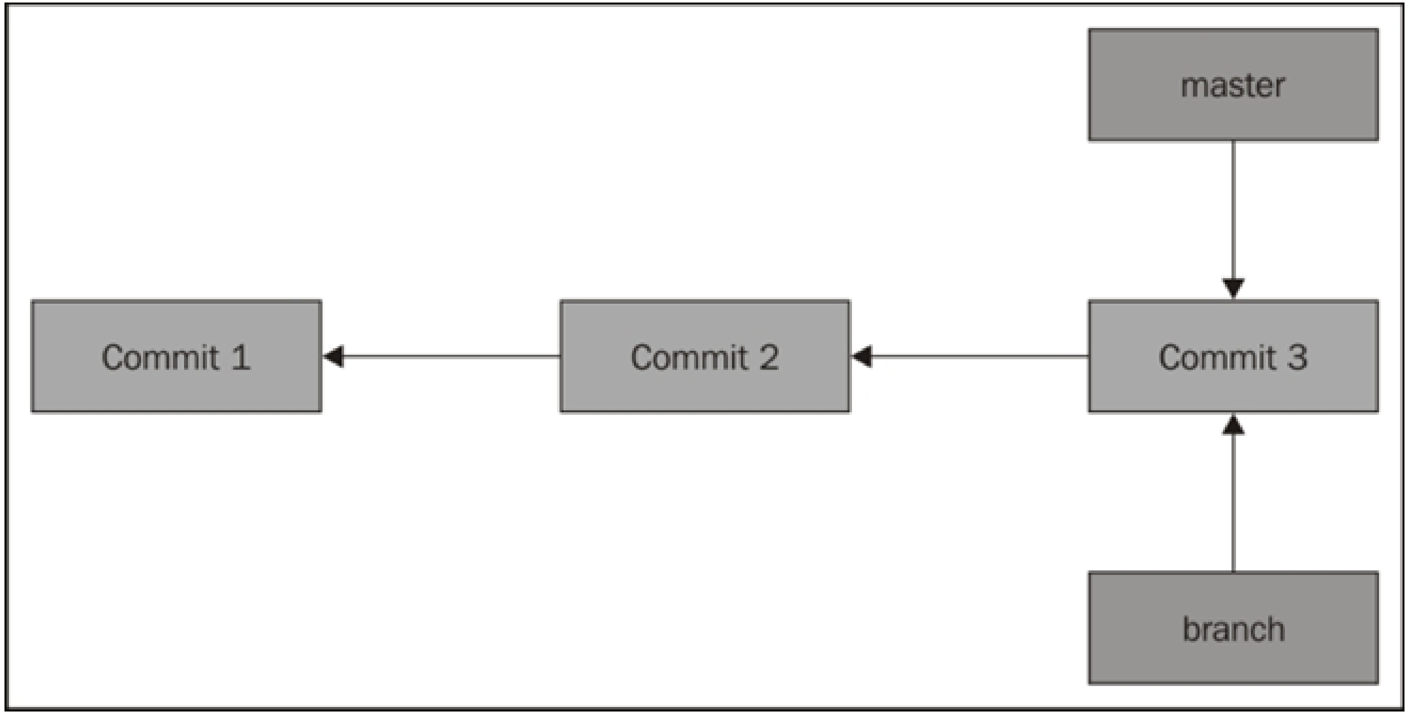
Fast forward merge

- The diagram shows the current state of the repository.

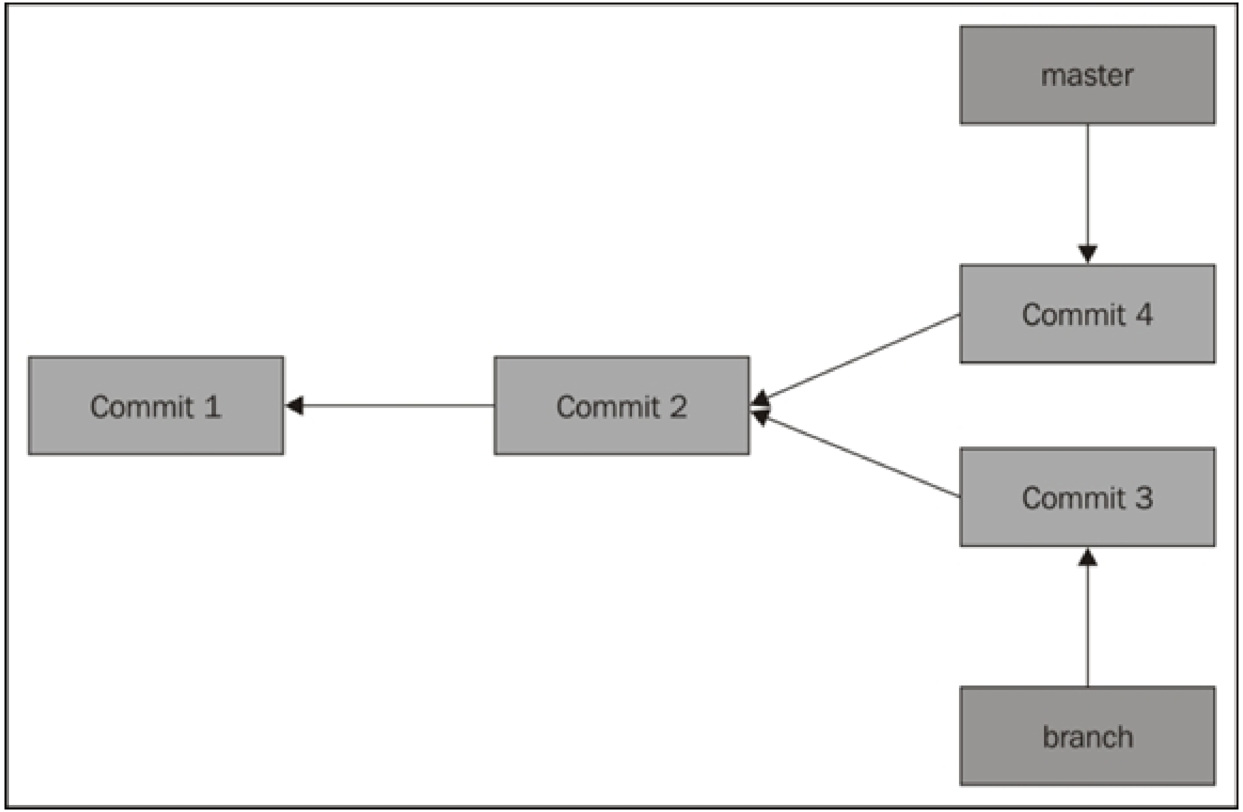
- Now, we have to checkout to the master branch and merge the branch on it.

```
Erik@local:~/webproject$ git checkout master
Erik@local:~/webproject$ git merge branch
```

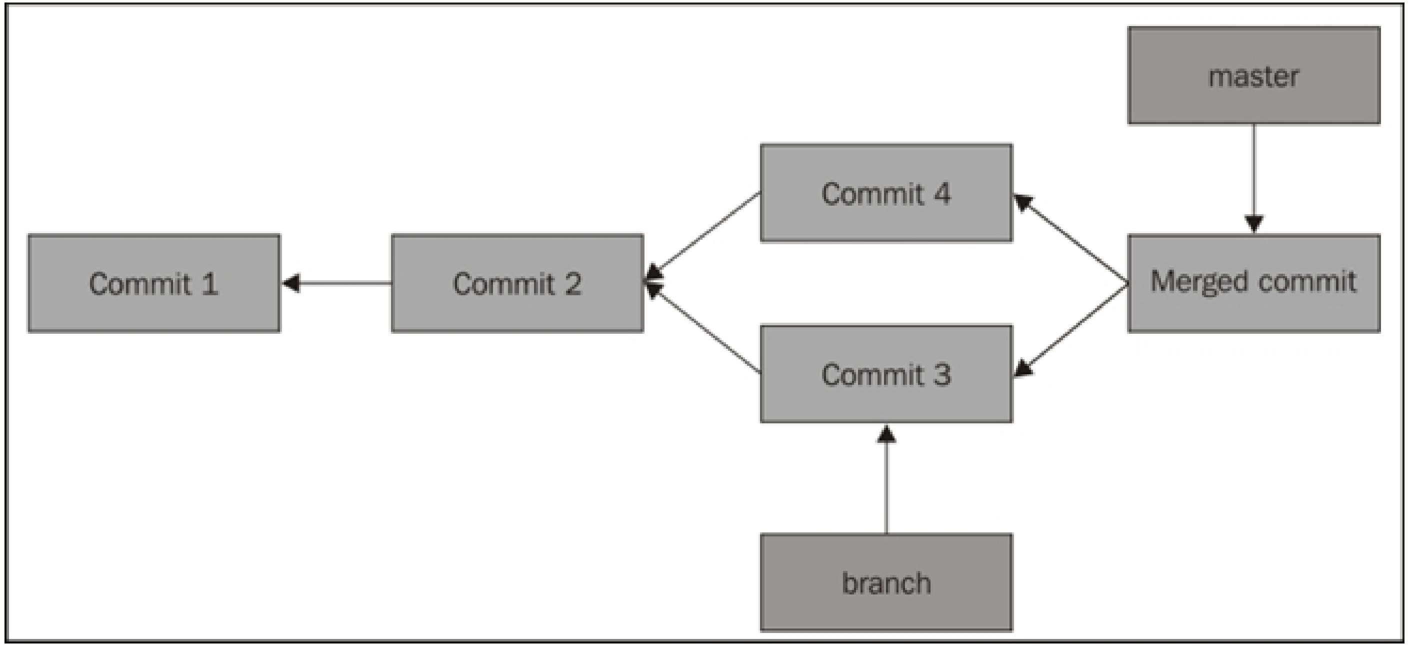
Fast forward merge



Merge commit



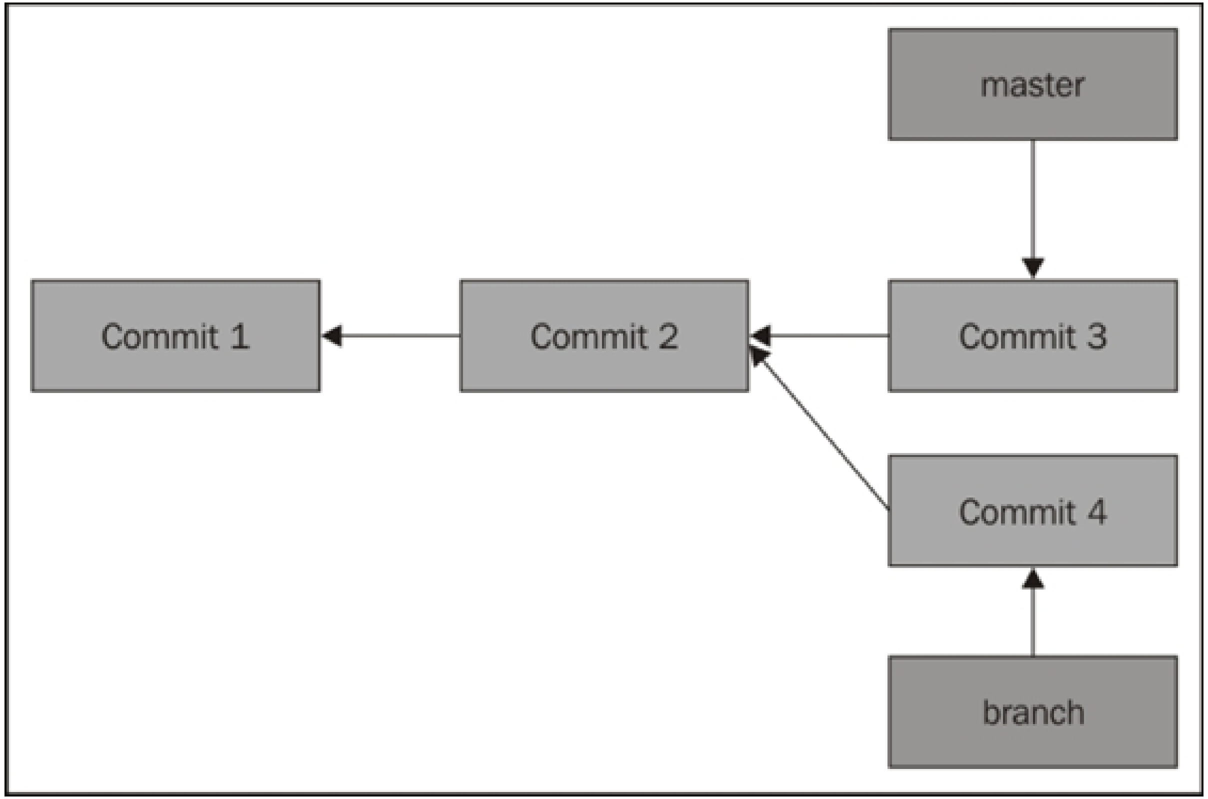
Merge commit



- To use this strategy (without fast forward), you can use this command:

```
Erik@local:~/webproject$ git merge --no-ff
```

Rebase

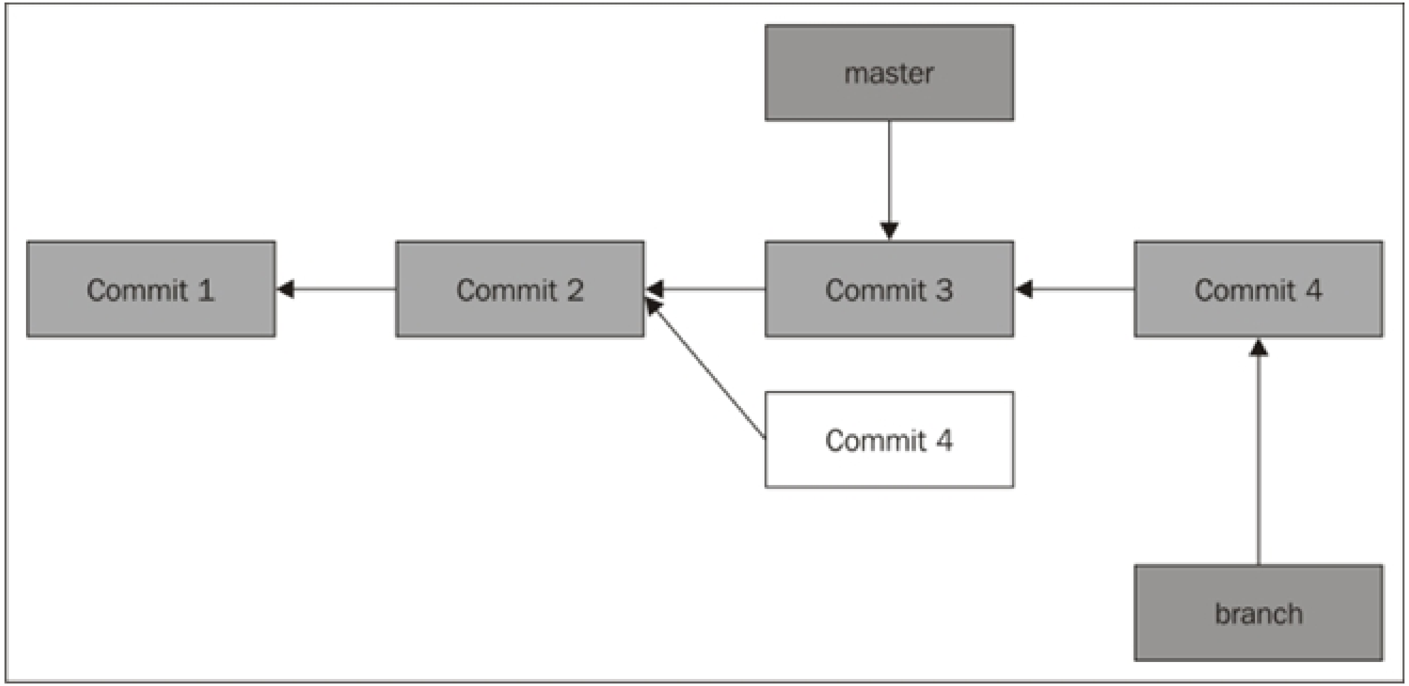


Rebase

- Our goal is to rebase branch from master, so we need to checkout on master and rebase it:

```
Erik@local:~/webproject$ git checkout branch
Erik@local:~/webproject$ git rebase master
```

Rebase



Cherry-pick

- Let's try to understand this by exploring the following example: Jim creates the jim branch from master and adds a new file in it:

```
Jim@local:~/webproject$ git checkout -b jim
Jim@local:~/webproject$ touch home.html
Jim@local:~/webproject$ git add home.html
Jim@local:~/webproject$ git commit -m 'add homepage'
Jim@local:~/webproject$ echo "<html>...</html>" > home.html
Jim@local:~/webproject$ git commit -a -m 'add content inside home'
```

Cherry-pick

- Jim creates the home.html file, adds it into Git, and commits it.

- Then he edits it and commits again. Now, let's see the commit history for this branch:

```
Jim@local:~/webproject$ git log --oneline
4f6ec45 add content inside home
22c45b7 add homepage
```

- Now, Jim will apply the first commit to the master branch:

```
Jim@local:~/webproject$ git checkout master
Jim@local:~/webproject$ git cherry-pick 22c45b7
```

TriveraTech TECHNOLOGY TRAINING

Creating and deleting tags

- The command to create a tag is very easy:

```
Jim@local:~/webproject$ git tag 1.0.0
#Annotated  tag contains a small description
Jim@local:~/webproject$ git tag 1.0.0 -m 'Release 1.0.0'
#Use a commit
Jim@local:~/webproject$ git tag 1.0.0 -m 'Release 1.0.0' commit_hash
```

Creating and deleting tags

```
Jim@local:~/webproject$ git tag
0.1.0
0.1.5
0.2.0
0.9.0
1.0.0
Jim@local:~/webproject$ git tag -l 0.1.*
0.1.0
0.1.5
Jim@local:~/webproject$ git tag -d 1.0.0
#Push it remotely
Jim@local:~/webproject$ git push origin tag 1.0.0
```

Summary

- In this lesson, we saw how to work within a team using Git, which is very common for most developers.
- Now, you understand what a branch is, how we can merge them, and how to rebase one branch on another.
- We also saw how to tag a commit. Now you are ready to prepare and work on your Git repository, but there is something that has been left behind: what should you do if there are conflicts?

Distributed Git

TriveraTech
TECHNOLOGY TRAINING

Distributed Git

- Git is a fully distributed revision control system (RCS), like Mecurial, Bazaar, and Darcs.

- Git was created by Linus Torvalds, the creator of the Linux operating system to help support the development of the Linux kernel.

- The purpose of all revision control systems is to manage sequences of changes over time to a collection of text documents in an orderly and meaningful way, granting the ability examine the documents at any point in time, moving forward and backward through file histories.

THE GIT OBJECT MODEL

OBJECT STORE

- At the heart of a git repository is the object store, It contains files, log messages, information about commiters, timestamps, file permissions, and other information necessary to rebuild any version or branch of the document collection.

- The git object store is organized as a content-addressable storage system, Each of the objects in the store is identified with a unique identifier created by hashing the object's content using the cryptographically Secure Hash Function (SHA1).

GIT OBJECT TYPES

- There are only four data types in the object store: blobs, trees, commits, and tags.
- Despite git work flows heavily revolving around branching and merging, there is no branch object in the object store, branches are tag objects held in a special reference directory.
- Each of these four atomic objects form the foundation for higher level data structures in git.

THE STAGING AREA

- In git parlance, the working directory is the directory holding your document collection managed by git.
- The staging area, or index as it is sometimes called, is the buffer between your working directory and the git object store.
- The staging area holds a set of prospective modifications to the object store, File additions, file removals, file renames, file permission changes, and file edits are held in the staging area until a commit is made.

MODIFYING THE STAGING AREA AND THE DATA STORE

- To view the contents of your staging area and the status of your working directory, use the status command.

```
$ git status
```

- The status command displays the contents of the staging area in green at the top of the output.
- Modifications to your working directory not in the staging area (and thus not in the next commit) are displayed in red at the bottom of the output, If you don’t have a colorized console, don’t worry, the staging area contents and the working directory status are also labeled.
- After a file has been edited, we add it to the staging area using the add command.

```
$ git add modified_file.c
```

MODIFYING THE STAGING AREA AND THE DATA STORE

- You need to provide a commit message to create a commit in the data store.
- By default git uses vim for commit messages, you can configure it to use your favorite editor with the environment variable GIT_EDITOR, You can also supply the commit message using the -m option.

```
$ git commit -m "Adding filename.c to the repository."
```

- Git does not manage your working directory using file directories like hierarchical revision control systems do. Instead, git creates tree objects in the data store as needed when a file path indicates a new sub-directory, The tree objects are created in the data store behind the scenes.
- For example, adding a new file under a new sub-directory like so:

```
$ git add subdir/path.c
```

MODIFYING THE STAGING AREA AND THE DATA STORE

- To remove a file from the working directory and the repository, we use the rm command.

```
$ git rm filename.c
```

- This adds an entry to the staging index indicating filename.c is going to be removed.
- Once the staging area is committed, git will delete filename.c from the repository by removing its SHA1 object identifier from the appropriate tree object in the data store, it will then physically delete the file from disk on your behalf.
- Renaming or moving a file is done with the mv command.

```
$ git mv filename.c newname.c
```

MODIFYING THE STAGING AREA AND THE DATA STORE

- To create a tag in the data store, we use the tag command, supplying a human readable label and possibly a message.

```
$ git tag v1.0 -m "Release version 1.0“
```

- By default, the tag command creates the tag object referencing the last commit in the current working branch (referred to as HEAD), however, a specific commit object can also be supplied.

```
$ git tag v1.0 deadbeef -m "Release version 1.0.“
```

- The tag command is also used to print a list of tags in the repository, This is the behavior when no arguments are supplied.

```
$ git tag
```

BRANCHES AND MERGING

- In git, branches are merely tag objects stored in a special reference directory.
- We can create or view branches using the branch command, To list all branches in our copy of the repository (called the local repository), we run the branch command with no arguments.

```
$ git branch
```

- All branches that exist in our local repository will be listed. By default, the only branch in a git repository is themaster branch.
- Git is distributed, If a git repository knows about a remote copy, it can also display those branches, To display, all known branches we run the branch command with the -a option.

```
$ git branch -a
```

TriveraTech TECHNOLOGY TRAINING

BRANCH CREATION

- To create a branch from the current branch, we give the branch command a parameter - the name of the new branch.

```
$ git branch new_branch
```

- Now listing the branches will show our new branch.

- We can checkout our new branch and start working on our changes by using the checkout command.

```
$ git checkout new_branch
```

TriveraTech TECHNOLOGY TRAINING

BRANCH CREATION

- This checks out our new branch, Git protects you from losing changes. It will not let you switch branches if you have changes you'd lose by doing so.
- Make sure you've committed all changes on your current branch before switching to another branch to do work.
- To create a branch and check it out in one step, perform the following:

```
$ git checkout -b new_branch branch_from
```

BRANCH INTEGRATION

- After you've implemented your changes and commits, integrate your feature branch using the mergecommand.

- Before merging, you checkout the branch you wish to integrate into.

```
$ git checkout develop
$ git merge --no-ff new_branch
```

- If there are no conflicts, you will be prompted for a commit message for the merge commit.

BRANCH INTEGRATION

- If there are conflicts and you don’t wish to keep the merge, you can back out the changes by using the –abort option while in merge resolution mode.

```
$ git merge --abort
```

- Git will cleanup you working directory and put you in a known good state - as if the merge never occurred.

BRANCHING MODEL

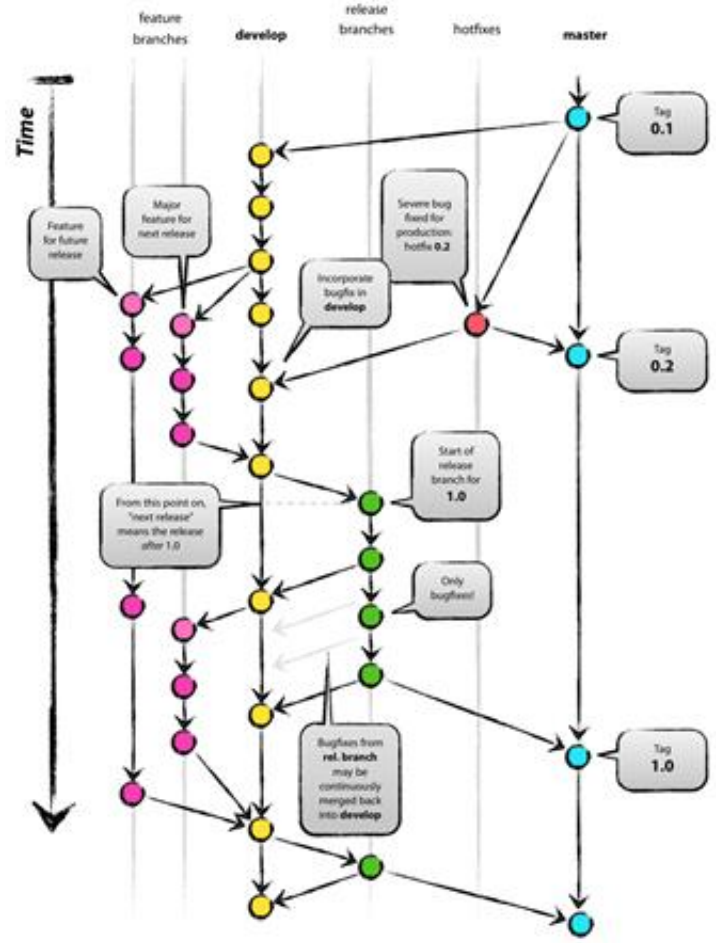
- Branches that have been integrated into develop can be deleted using the -d option.

```
$ branch -d branch_name
```

- branch_name is the name of the feature branch we wish to delete. When using the -d option, git will prevent you from deleting branches which haven't been merged back into their parent branches.
- This prevents you from accidentally deleting work, If we decide a branch will never be integrated and we still wish to delete it, we can force deletion using the -D option.

```
$ branch -D branch_name
```

BRANCHING MODEL


feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0

REMOTE REPOSITORIES

- Git is fully distributed. Everyone gets an entire copy of the repository history when they clone a repository.
- Any changes users make to a repository are local to their repository until they are explicitly shared.
- Technically, git has no centralized authority for determining the official version of the repository, The distinction is made only by convention, Typically, the maintainer of the software has the official copy of the repository.
- Typically, users begin working with a git repository by invoking the clone command.

```
$ git clone url
```

REMOTE REPOSITORIES

- To view remote repositories use the remote command.

```
$ git remote
```

- If you wish to see the URLs of the remotes in addition to the labels, use the -v option.

```
$ git remote -v
```

REMOTE REPOSITORIES

- The following is example output from running git branch -a:

```
  master
* develop
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
  remotes/origin/develop
```

- To add a new remote, we again use the remote command.
- This time supply a name for the remote and a URL where the repository is available. For example:

```
$ git remote bobs_repo https://github.com/bob/repo.git
```

GETTING AND VIEWING REMOTE CHANGES

- To download the latest changes from a remote branch, use the fetch command.

```
$ git fetch origin/develop
```

- This command would grab the latest commits to the develop branch from the repository origin.
- To view the changes, use the log command with the branch name.

```
$ git log origin/develop
```

GETTING AND VIEWING REMOTE CHANGES

- Viewing specific changes can be accomplished with the show command, the diff command, or the difftool command.

- The show and diff commands display a commit's diff using git's internal diff engine, the difftool command uses your configured diff program to display the changes.

- When viewing changes on remote branches, it is easiest to provide the hash values referenced in the log message to view changes.

```
$ git difftool deadbeef~ deadbeef
```

TriveraTech TECHNOLOGY TRAINING

GETTING AND VIEWING REMOTE CHANGES

- To download and integrate changes from a remote branch, use the pull command.

- This command would be used to keep your integration and master branches in sync with origin. For example:

```
$ git checkout master
$ git pull origin
```

SHARING LOCAL CHANGES WITH A REMOTE

- After you’ve completed a feature and integrated it into your develop branch, you’d do a git pull origin to get any new changes from the remote repository, and then you’d want to share your changes.

- To share a copy of your changes with a remote repository use the push command.

- The long-form syntax for push is as follows:

```
$ git push origin develop:develop
```

SHARING LOCAL CHANGES WITH A REMOTE

- If the remote branch does not exist, git will create the branch in the remote repository.
- Configuring the “push.default” setting to “simple” will allow us to use the shorthand version for a push.

```
$ git push
```

COLLABORATING WITH GIT

- Being fully distributed has many attractive benefits, but having a central copy of the repository can aid in collaboration.

- In a corporate environment, where there are a handful of maintainers working on a project (for some definition of handful) and everyone needs write access to the central repository, the best approach is to host the central copy of the repository on a sever to which everyone has Secure Shell (SSH) access.

- Then manage write access to the repository using gitolite.

SUMMARY

- We’ve reviewed the git distributed revision control system.
- We’ve described its object model and data store, as well as the staging area, branching and merging functionality, and the intended branching model.
- We’ve also discussed distributed collaboration using git and how to manage remote repositories.

Git Tools

Cleaning your mistakes

- Let's start this section with how to remove untracked files:

```
Erik@server:~$ git clean -n
```

- The –n option will make a dry-run (it's always important to see what will happen before you regret it).
- If you want to also remove directories and hidden files, use this one:

```
Erik@server:~$ git clean -fdx
```

- With these options, you will delete new directories (-d) and hidden files (-x) and be able to force them (-f).

Reverting uncommitted changes

- Let's suppose you edited a file on the production working directory, but didn't commit it, On your last push, you edited it, and the changes in production aren't needed anymore.
- So, your goal is to erase changes on this file and reset the file to the last committed version:

```
Erik@server:~$ git checkout the_filename
```

- This command is really nice if you want to restore a deleted file. You can also specify a commit pointer to use (useful if you stash your changes):

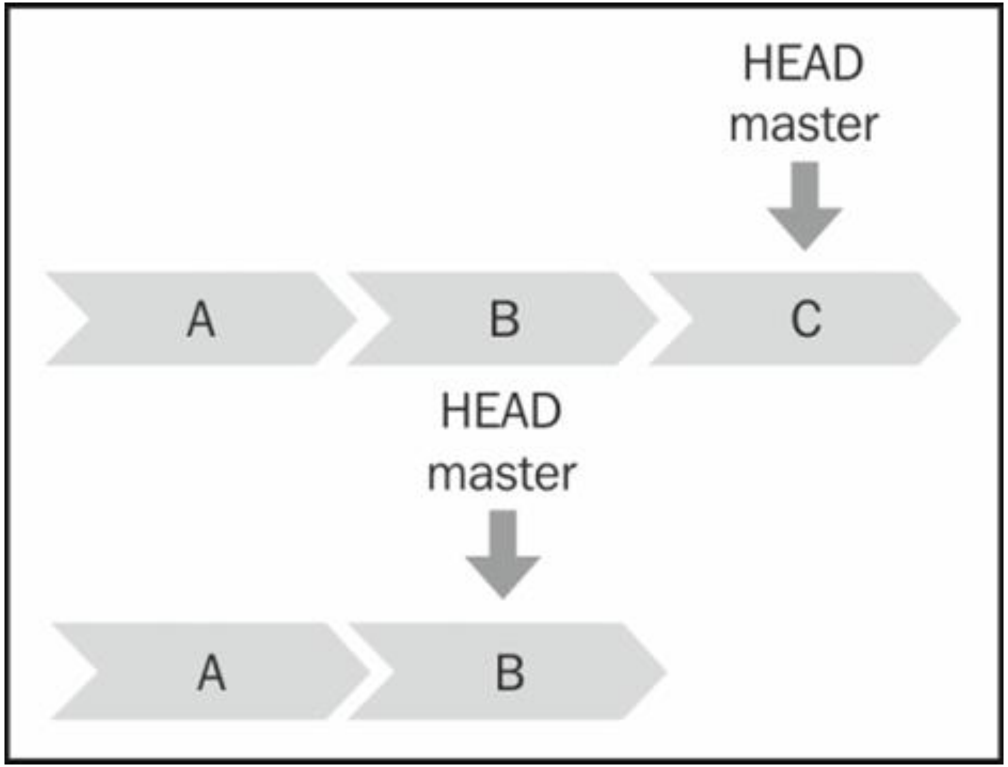
```
Erik@server:~$ rm myfile.txt
Erik@server:~$ git checkout HEAD myfile.txt
```

The git reset command

- The git reset command will allow you to go back to a previous state (for example, commit), The git reset command has three options (soft, hard, or mixed, by default).
- In general, the git reset command's aim is to take the current branch, reset it to point somewhere else, and possibly bring the index and work tree along.
- More concretely, if the master branch (currently checked out) looks like the first row (in the following figure) and you want it to point to B and not C, you will use this command:

```
Erik@server:~$ git reset B
```

The git reset command



The git reset command

- The following table explains what the options really move:

Option	Head pointer	Working tree	Staging area
Soft	Yes	No	No
Mixed	Yes	No	Yes
Hard	Yes	Yes	Yes

Editing a commit

- There are several tricks to edit your last commit.
- If you want to edit the description of your last commit, use the following line:

```
Erik@server:~$ git commit --amend
```

- Let's suppose that your last commit contains buggy code; you can specify that your changes on the file are part of the last commit:

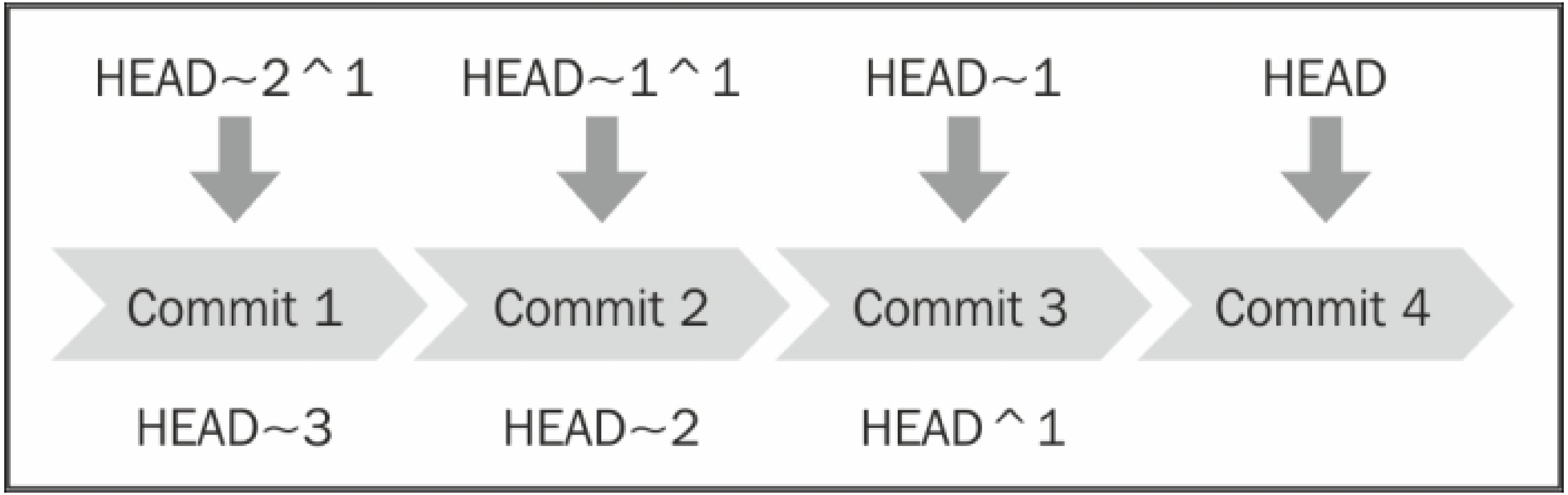
```
Erik@server:~$ git add filename.txt
Erik@server:~$ git commit -v -amend
```

Editing a commit

- Now I want to remove a file I included accidentally in the last commit (because this file deserves a new commit):

```
Erik@server:~$ git reset HEAD^1 filename.txt
Erik@server:~$ git commit --amend -v
Erik@server:~$ git commit -v filename.txt
```

Editing a commit



Canceling a commit

- Firstly, let's have a look at the last commits:

```
Erik@server:~$ git log
commite4bac680c5818c70ced1205cfc46545d48ae687e
Author: Eric Pidoux
Date:   Sun Jul 20 19:00:47 2014 +0200
replace all
commit0335a5f13b937e8367eff35d78c259cf2c4d10f7
Author: Eric Pidoux
Date:   Sun Jul 20 18:23:06 2014 +0200
commitindex.php
```

- We want to cancel the 0335… commit:

```
Erik@server:~$ git revert 0335a5f13
```

Rewriting commit history

- Sometimes a situation will occur where you want to remove a file from all commits because it contains confidential information.

- You can do it by using git filter-branch:

```
Erik@server:~$ git filter-branch --index-filter 'git rm --cached --ignore-
unmatch myconfidentialfilename.txt' HEAD
```

Solving merge conflicts

- If it occurs, Git will mark the conflict and you have to resolve it.
- For example, Jim modified the index.html file on a feature branch and Erik has to edit it on another branch, When Erik merges the two branches, the conflict occurs.
- Git will tell you to edit the file to resolve the conflict. In this file, you will find the following:

```
<<<<<<< HEAD
Changes from Erik
=======
Changes from Jim
>>>>>>> b2919weg63bfd125627gre1911c8b08127c85f8
```

Solving merge conflicts

- Git's diff helps you to find differences:

```
Diff --git erik/mergetestjim/mergetest
Index.html 88h3d45..92f62w 130634
--- erik/mergetest
+++ jim/mergetest
@@ -1,3 +1,4 @@
<body>
+I added this code between
This is the file content
-I added a third line of code
+And this is the last one
```

Searching errors with git bisect

- For example, you pulled the last commits and the website isn't working anymore.
- You know that before the last pull everything was okay! So you have to find the commit ID before it crashes and the last ID after the pull:

```
Erik@server:~$ git bisect start
Erik@server:~$ git bisect bad commitIDAfterThePull
Erik@server:~$ git bisect good commitIDBefore
```

- Now, the bisecting loop begins and Git will check for an alternative commit. Reset the given commit and tell Git whether the website is working:

```
Erik@server:~$git bisect badcommitID
```

TriveraTech TECHNOLOGY TRAINING

Searching errors with git bisect

- Git will search again and again to find which commit crashed the website:

```
Erik@server:~$ git show theCommitID
```

Searching errors with git bisect

- For example, we want to check the existence of a file:

```
#!/bin/bash
FILE=$1
If [ -f $FILE];
Then
  Exit 0;
Else
  Exit 1;
fi
```

Searching errors with git bisect

- Now it's time to run git bisect to specify the last 10 commits:

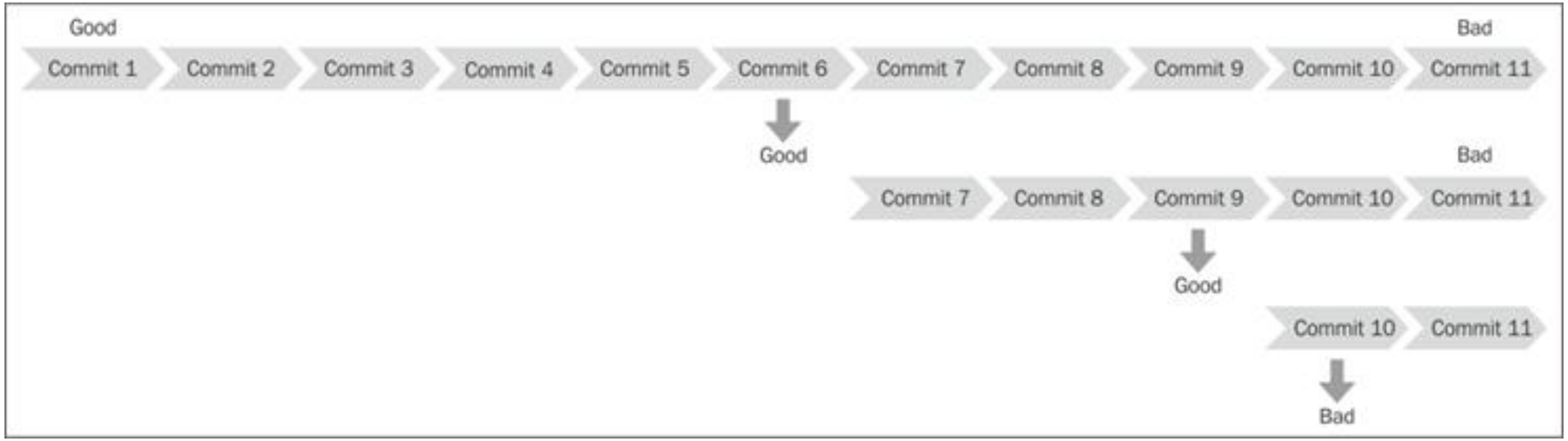
```
Erik@server:~$ git bisect start HEAD HEAD~10
Erik@server:~$ git bisect run ./check_file.sh index.html
```

- The algorithm used by git bisect always returns the commit that is at the middle position of the array.

TriveraTech TECHNOLOGY TRAINING

Searching errors with git bisect

- In the following diagram, you will see how the algorithm found the good commit in three steps:



Adding a submodule

- Let's imagine you are working on a website and you want to add the fpdf library that helps you create a PDF file in PHP.
- The first thing to do is to clone the library's Git repository inside your subfolder:

```
Erik@server:~/mySite/$ git submodule add
https://github.com/lsolesen/fpdf.git fpdf
Cloning in 'fpdf'
remote: Counting objects: 966, done.
remote: Total 966 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (966/966), 5.96 MiB | 1.13 MiB/s, done.
Resolving deltas: 100% (292/292), done.
```

Adding a submodule

- While you add the Git submodule, Git adds two files, Fpdf and .gitmodules. Let's see this with the git status command:

```
Erik@server:~/mySite/$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#     new file: .gitmodules
#    new file: Fpdf
```

Cloning a project with submodules

```
•   Erik@server:~$ git clone git://theurl.com/mySite.git
•   Initialized empty Git repository in /var/www/mySite/.git/
•   remote: Counting objects: 6, done.
•   remote: Compressing objects: 100% (4/4), done.
•   remote: Total 6 (delta 0), reused 0 (delta 0)
•   Receiving objects: 100% (6/6), done.
•   $ cd mySite
•   $ ls -l
•   total 8
•   -rw-r--r--  1 epidoux  admin   3 Aug  19 10:24 index.html
•   drwxr-xr-x  2 epidoux  admin  68 Aug  19 10:24 fpdf
```

Cloning a project with submodules

- The fpdf folder is created, but it is empty. You will have to execute these two commands to initialize the import submodule:

```
Erik@server:~$ git submodule init
Submodule 'fpdf' (git://github.com/lsolesen/fpdf.git) registered for path
'fpdf'
Erik@server:~$ git submodule update
Initialized empty Git repository in /var/www/mySite/fpdf/.git/
…
```

Removing a submodule

To remove a submodule from your project, you have to execute these steps:

- Delete the lines of the submodule from the .gitmodules file.
- Delete the submodule part from .git/config.
- Delete the submodule from Git by executing this command:

```
Erik@server:~$ git rm -cached submodule_path
```

Adding a subproject with a subtree

- Firstly, we need to tell Git that we want to include a project as a subtree.
- We use the git remote command to specify where the remote repository of this subtree is:

```
Erik@server:~$ git remote add –f fpdf_remote
git://github.com/lsolesen/fpdf.git
```

- Now, you can add the subtree inside your project using the remote repository:

```
Erik@server:~$ git subtree add --prefix fpdf fpdf_remote master --squash
```

Adding a subproject with a subtree

- Will create the subproject.
- If you want to update it later, you will have to use the fetch and subtree pull commands:

```
Erik@server:~$ git fetch fpdf_remote master
Erik@server:~$ git subtree pull fpdf fpdf_remote master --squash
```

Contributing on a subtree

- Obviously, you can commit your fixes to the subproject in the local directory, but when you push back upstream, you will have to use another remote repository:

```
Erik@server:~$ git remote add epidoux-fpdf
ssh://git@github.com/epidoux/fpdf.git
Erik@server:~$ git subtree push --prefix=fpdf epidoux-fpdf master
Git push using: epidoux-fpdf master
Counting objects: 1, done.
Delta compression using up to 1 thread.
Compressing objects: 100% (1/1), done.
Writing objects: 100% (1/1), 150 bytes, done.
Total 1 (delta 1), reused 0 (delta 0)
To ssh://git@github.com/epidoux/fpdf.git
  243ab46..dca35db dbca35db 21fe51c9b5824370b3b224c440b3470cb -} master
```

Creating and applying patches

- A patch is a piece of code that can apply a set of changes (that is, a commit) to any branch and any order of a project.
- The main goal of this feature is to share the changes with other developers, and it gives control to the project maintainer (whether they choose to incorporate the contribution or not).
- Creating a patch in Git is very easy, as we will see right now.

Creating and applying patches

- The git log command, as we saw in lesson 2, Working in a Team Using Git, shows the commits:

```
Erik@server:~$ git log --pretty=oneline -5
```

- Now, we have the commit key to create the patch:

```
Erik@server:~$ git format-patch -1 theSha1 --stdout > myPatch.patch
```

Creating and applying patches

- For example, let's say Erik wants to change the title of the index.html file, He has no write access to the remote repository.
- He edits the file and commits his change. So, his file looks like this:

```
<html>
  <head>
    <title>Erik new title</title>
  </head>
  <body>

  </body>
</html>
```

Creating and applying patches

- Now that the file is edited, we can commit the change and generate a patch:

```
Erik@server:~$ git commit –a –m "Change the title"
Erik@server:~$ git format-patch master
```

Creating and applying patches

- The last command will create a file called 0001-Change-the-title.
- If you open the generated patch file, you will find something like this:

```
Index 98e10a1..854cc34 100643
--- a/index.html
+++ b/index.html
@@ -3,4 +3,3 @@
  <head>
-  <title> The old common title </title>
+  <title>Erik new title</title>
  </head>
```

Applying the patch

- Jim is the maintainer of this project and decided to apply the patch made by Erik.
- You have to move to the right repository, and after you check the patch that is applicable, you just have to use the git am command with the signoff option.
- This option will use the identity metadata from the patch, and not your metadata. You can use the k option too, which keeps the flags:

```
Jim@local:~$ git checkout –b patch-branch
Jim@local:~$ git am --signoff -k < 001-Change-the-title.patch
```

Customizing Git

Git hooks

There are two types of hooks, depending on the commands:

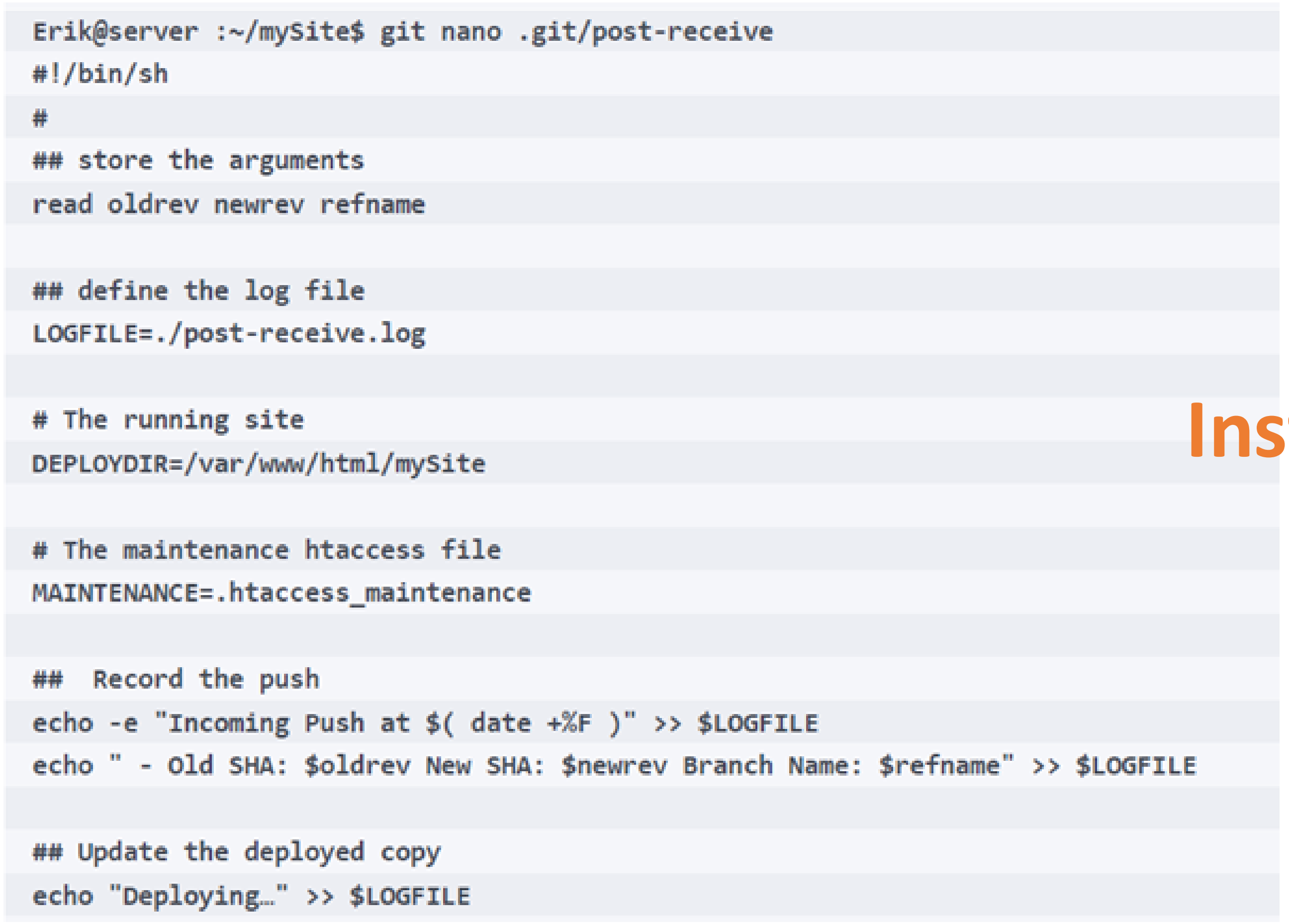
- Client hooks: These kinds of hooks are for client operations, such as the commit or merge command

- Server hooks: These hooks are for Git server-side operations, such as the push command

More about hooks

Hook	Type	Trigger	Can reject?
pre-commit	Client	Before the commit	Yes, the commit
post-commit	Client	After the commit	No
commit-msg	Client	During the commit	No
update	Server	While receiving the push	Yes, the push
pre-receive	Server	Before a receive pack	Yes
post-receive	Server	After a receive pack	No
post-update	Server	During a push from the client side	No
post-checkout	Client	After a checkout	No
post-merge	Client	After a merge	No

Installing a hook



```
Erik@server :~/mySite$ git nano .git/post-receive
#!/bin/sh
#
## store the arguments
read oldrev newrev refname

## define the log file
LOGFILE=./post-receive.log

# The running site
DEPLOYDIR=/var/www/html/mySite

# The maintenance htaccess file
MAINTENANCE=.htaccess_maintenance

##  Record the push
echo -e "Incoming Push at $( date +%F )" >> $LOGFILE
echo " - Old SHA: $oldrev New SHA: $newrev Branch Name: $refname" >> $LOGFILE

## Update the deployed copy
echo "Deploying…" >> $LOGFILE
```

Installing a hook

```
echo " - Entering the maintenance mode "
cd /var/www/html/mySite
mv .htaccess .htaccess_prod
mv $MAINTENANCE .htaccess
echo " - Updating code"
GIT_WORK_TREE="$DEPLOYDIR" git checkout -f

echo " – Exiting maintenance mode"
mv .htaccess $MAINTENANCE
mv .htaccess_prod .htaccess

echo "Finished Deploying" >> $LOGFILE
```

Installing a hook

- Previous script, written in bash, will be fired after the push is received by the remote repository.
- It manages the entire deployment process for us by enabling and disabling the maintenance mode and updating the code.
- Let's look at this more closely:

```
read oldrev newrev refname
```

Installing a hook

- Helps us to track what's being deployed, and roll back, if necessary.
- However, this script doesn't handle a rollback:

```
LOGFILE=./post-receive.log
DEPLOYDIR=/var/www/html/mySite
MAINTENANCE=.htaccess_maintenance
```

- Next, we set up a variable to log the output of the deployment and deployment directory:

```
echo -e "Incoming Push at $( date +%F )" >> $LOGFILE
echo " - Old SHA: $oldrev New SHA: $newrev Branch Name: $refname"
>> $LOGFILE
```

Installing a hook

- Here, we log when a push request and its details are received:

```
GIT_WORK_TREE="$DEPLOYDIR" git checkout -f
```

- Here, we tell Git where the working tree is, and instruct Git to check out the latest copy of the code in the directory, removing any changes that might have been made manually:

```
echo "Finished Deploying" >> $LOGFILE
```

Installing a hook

- You can change this hook to suit your needs, and don't forget to make the file executable, otherwise it won't run.

- If you're not familiar with this process, this is how it's done:

```
Erik@server~:$ chmod +x post-receive
```

Customizing Git

- As you already know, when you install Git, you have to configure your username and e-mail, but this isn't the only one configuration you can do:

```
Erik@server~:$ git config --global user.name 'Erik'
Erik@server~:$ git config --global user.email 'erik@mymail.com'
```

Installing a hook

- There are a lot of options available, but we will cover only the commonly used. You can see a list of all options executing this:

```
Erik@server~:$ git config -help
```

Here are the most important ones:

- Editor: You can use editors such as Emacs, Vi, nano, and so on. The following example shows how to use your favorite editor:

```
Erik@server~:$ git config --global core.editor nano
```

- Commit template: You can specify a message when you commit something:

```
Erik@server~:$ git config --global commit.template ~/gitmsg.txt
```

Here are the most important ones:

- Autocorrect: When you type a wrong command, Git will try to understand which command you actually meant to type:

```
Erik@server~:$ git pul
Git : 'pul' is not a git-command. See 'git --help'.
Did you m
ean this ?
   Pull
```

If you set help.autorrect to 1, Git will run the match command.

- Add colors: You are now forced to use a white font inside your terminal. To add colors, enable it with this command:

```
Erik@server~:$ git config --global color.ui true
```