Common GIT scenarios

There are **three main sections** of a Git project: the Git directory, the working tree, and the staging area.

- **The Git directory** is where Git stores the metadata and object database for your project. it is what is copied when you *clone* a repository.
- **The working tree** is a single checkout of one version of the project. These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify. It is your scratch space, used to easily modify file content.
- **The staging area** is a file, generally contained in your Git directory, that stores information about what will go into your next commit. Its technical name in Git parlance is the "**index**", but the phrase "**staging area**" works just as well.

The Three States in Git: modified(deleted), staged and committed.

The HEAD in Git is the pointer to the current branch reference, which is in turn a pointer to the last commit you made or the last commit that was checked out into your working directory. That also means it will be the parent of the next commit you do. It's generally simplest to think of it as **HEAD is the snapshot of your last commit**.

The Index is your proposed next commit. Git populates it with a list of all the file contents that were last checked out into your working directory and what they looked like when they were originally checked out. It's not technically a tree structure, it's a flattened manifest, but for our purposes it's close enough. When you run `git commit`, that command only looks at your Index by default, not at anything in your working directory. So, it's simplest to think of it as **the Index is the snapshot of your next commit**.

Committing, Pushing and Pulling

Now that you've got a local copy and a copy on your GitHub account, there are four things that you'll need to know how to do in order to collaborate with SparkFun:

- **Commit** - committing is the process which records changes in the repository. Think of it as a snapshot of the current status of the project. Commits are done locally.
- **Push** - pushing sends the recent commit history from your local repository up to GitHub. If you're the only one working on a repository, pushing is fairly simple. If there are others accessing the repository, you may need to pull before you can push.
- **Pull** - a pull grabs any changes from the GitHub repository and merges them into your local repository.
- **Sync** - syncing is like pulling, but instead of connecting to your GitHub copy of the forked repo, it goes back to the original repository and brings in any changes. Once you've synced your repository, you need to push those changes back to your GitHub account.

git pull= git fetch + git merge

https://gist.github.com/grawity/4392747

Protocols to choose from when cloning:

plain Git, aka `git://github.com/`

- **Does not add security** beyond what Git itself provides. The server is not verified.

  If you clone a repository over git://, you should check if the latest commit's hash is correct.

- You **cannot push** over it. (But see "Mixing protocols" below.)

HTTPS, aka `https://github.com/`

- HTTPS **will always verify the server** automatically, using certificate authorities.
- (On the other hand, in the past years several certificate authorities have been broken into, and many people consider them not secure enough. Also, some important HTTPS security enhancements are only available in web browsers, but not in Git.)
- Uses **password** authentication for pushing, and still allows anonymous pull.
- Downside: You have to enter your GitHub password every time you push. Git can remember passwords for a few minutes, but you need to be careful when storing the password permanently – since it can be used to change anything in your GitHub account.
- If you have two-factor authentication enabled, you will have to use a personal access token instead of your regular password.
- HTTPS **works practically everywhere**, even in places which block SSH and plain-Git protocols. In some cases, it can even be **a little faster** than SSH, especially over high-latency connections.

SSH, aka `git@github.com:` or `ssh://git@github.com/`

- Uses **public-key** authentication. You have to generate a **keypair** (or "public key"), then add it to your GitHub account.

- Using keys is **more secure than passwords**, since you can add many to the same account (for example, a key for every computer you use GitHub from). The private keys on your computer can be protected with passphrases.
- On the other hand, since you do not use the password, GitHub does not require two-factor auth codes either – so whoever obtains your private key can push to your repositories without needing the code generator device.
- However, the keys only allow pushing/pulling, but *not* editing account details. If you lose the private key (or if it gets stolen), you can just remove it from your GitHub account.
- A minor downside is that authentication is needed for all connections, so you always **need a GitHub account** – even to pull or clone.
- You also need to **carefully verify the server's fingerprint** when connecting for the first time. Many people skip that and just type "yes", which is insecure.
- (Note: This description is about GitHub. On personal servers, SSH can use passwords, anonymous access, or various other mechanisms.)

git config

With Git, there are many configurations and settings possible. *git config* is how to assign these settings. Two important settings are user user.name and user.email. These values set what email address and name commits will be from on a local computer. With *git config*, a *--global* flag is used to write the settings to all repositories on a computer. Without a *--global* flag settings will only apply to the current repository that you are currently in.

There are many other variables available to edit in *git config*. From editing color outputs to changing the behavior of *git status*. Learn about *git config* settings in the official Git documentation.

Usage:

```
# Running git config globally
$ git config --global user.email "my@emailaddress.com"
$ git config --global user.name "Brian Kerr"

# Running git config on the current repository settings
$ git config user.email "my@emailaddress.com"
$ git config user.name "Brian Kerr"
```

Use the “git config” tool to set the configuration variables

Set the user name
Pradip.Muhuri@HHSL4L772N2 MINGW64 ~
$ git config --global user.name "pkmedu"

Set the user email address
Pradip.Muhuri@HHSL4L772N2 MINGW64 ~
$ git config --global user.email "pradip.muhuri@ahrq.hhs.gov"

List the git configuration setting
Pradip.Muhuri@HHSL4L772N2 MINGW64 ~
$ git config --list
core.symlinks=false
core.autocrlf=true
core.fscache=true
color.diff=auto
color.status=auto
color.branch=auto
color.interactive=true
help.format=html
rebase.autosquash=true
http.sslcainfo=C:/Program Files/Git/mingw64/ssl/certs/ca-bundle.crt
http.sslbackend=openssl

```
diff.astextplain.textconv=astextplain
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
credential.helper=manager
user.name=pkmedu
user.email=pradip.muhuri@ahrq.hhs.gov
```

Check if Git has already been installed

Git is usually preinstalled on Mac and Linux.

Type the following command and then press enter:

```
git --version
```

You should receive a message that tells you which Git version you have on your computer. If you don't receive a "Git version" message, it means that you need to download Git.

If Git doesn't automatically download, there's an option on the website to download manually. Then follow the steps on the installation window.

After you are finished installing Git, open a new shell and type `git --version` again to verify that it was correctly installed.

Add your Git username and set your email

It is important to configure your Git username and email address, since every Git commit will use this information to identify you as the author.

On your shell, type the following command to add your username:

```
git config --global user.name "YOUR_USERNAME"
```

Then verify that you have the correct username:

```
git config --global user.name
```

To set your email address, type the following command:

```
git config --global user.email "your_email_address@example.com"
```

To verify that you entered your email correctly, type:

```
git config --global user.email
```

You'll need to do this only once, since you are using the `--global` option. It tells Git to always use this information for anything you do on that system. If you want to override this with a different username or email address for specific projects, you can run the command without the `--global`option when you're in that project.

Check your information

To view the information that you entered, along with other global options, type:

```
git config --global --list
```

Working with local repositories

git init

This command turns a directory into an empty Git repository. This is the first step in creating a repository. After running git init, adding and committing files/directories is possible.

```
# change directory to codebase
$ cd /Users/computer-name/Documents/website

# make directory a git repository
$ git init
Initialized empty Git repository in /Users/computer-name/Documents/website/.git/
```

git add

Adds files in the to the staging area for Git. Before a file is available to commit to a repository, the file needs to be added to the Git index (staging area). There are a few different ways to use git add, by adding entire directories, specific files, or all unstaged files.

```
# To add all files not staged:
$ git add .

# To stage a specific file:
$ git add index.html

# To stage an entire directory:
$ git add css
```

The ‘Git add command’ provides powerful ways to add modified files. You can use your codes natural directory hierarchy to control what gets added.

```
git add -u
```

updates all your changes

# add all modifications in the staging area
$ git add .

So, when you are using “git add .” command, it will add all the changes from that level. But when you use “git add -A” option it will look for modifications throughout the module and add them.

git commit

Record the changes made to the files to a local repository. For easy reference, each commit has a unique ID.

It’s best practice to include a message with each commit explaining the changes made in a commit. Adding a commit message helps to find a particular change or understanding the changes.

```
$ git commit -m "My first commit message"
[SecretTesting 0254c3d] My first commit message
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 homepage/index.html
```

Let us say that you have cloned the repository and the application is saved on your local computer in a particular folder. At this stage the site will be downloaded to your local computer and you can edit your files via your favorite editor. To commit the changes to your local GIT repository you can use the following command:

```
1 git commit -a -m "Commit comment."
```

The above command will commit the changes to your local repository and the comment will be added to the GIT logs. To push the changes to the server you have to use the following command:

```
1 git push
```

The system will connect to the server and upload the files that have been modified on your local computer.

git remote

To connect a local repository with a remote repository. A remote repository can have a name set to avoid having to remember the URL of the repository.
Usage:

```
# Adding a remote repository with the name of beanstalk
$ git remote add origin
git@account_name.git.beanstalkapp.com:/acccount_name/repository_name.git

# List named remote repositories
$ git remote -v
origin git@account_name.git.beanstalkapp.com:/acccount_name/repository_name.git (fetch)
origin git@account_name.git.beanstalkapp.com:/acccount_name/repository_name.git (push)
```

Note: A remote repository can have any name. It's common practice to name the remote repository 'origin'.

git push

Sends local commits to the remote repository. *git push* requires two parameters: the remote repository and the branch that the push is for.

```
# Push a specific branch to a remote with named remote
$ git push origin staging
Counting objects: 5, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (5/5), 734 bytes | 0 bytes/s, done.
Total 5 (delta 2), reused 0 (delta 0)
To git@account_name.git.beanstalkapp.com:/acccount_name/repository_name.git
   ad189cb..0254c3d  SecretTesting -> SecretTesting

# Push all local branches to remote repository
$ git push --all
Counting objects: 4, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 373 bytes | 0 bytes/s, done.
Total 4 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To git@account_name.git.beanstalkapp.com:/acccount_name/repository_name.git
   0d56917..948ac97  master -> master
   ad189cb..0254c3d  SecretTesting -> SecretTesting
```

git status

This command returns the current state of the repository.

git status will return the current working branch. If a file is in the staging area, but not committed, it shows with *git status*. Or, if there are no changes it'll return *nothing to commit, working directory clean.*

```
# Message when files have not been staged (git add)
$ git status
On branch SecretTesting
Untracked files:
  (use "git add <file>..." to include in what will be committed)

            homepage/index.html

# Message when files have been not been committed (git commit)
$ git status
On branch SecretTesting
Your branch is up-to-date with 'origin/SecretTesting'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   homepage/index.html

# Message when all files have been staged and committed
$ git status
On branch SecretTesting
nothing to commit, working directory clean
```

git branch

To determine what branch the local repository is on, add a new branch, or delete a branch.

```
# Create a new branch
$ git branch new_feature

# List branches
$ git branch -a
* SecretTesting
  new_feature
  remotes/origin/stable
  remotes/origin/staging
  remotes/origin/master -> origin/SecretTesting

# Delete a branch
$ git branch -d new_feature
Deleted branch new_feature (was 0254c3d).
```

```
# Switching to branch 'new_feature'
$ git checkout new_feature
Switched to branch 'new_feature'

# Creating and switching to branch 'staging'
$ git checkout -b staging
Switched to a new branch 'staging'
```

git merge

Integrate branches together. *git merge* combines the changes from one branch to another branch. For example, merge the changes made in a staging branch into the stable branch.

Usage:

```
# Merge changes into current branch
$ git merge new_feature
Updating 0254c3d..4c0f37c
Fast-forward
 homepage/index.html | 297 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 1 file changed, 297 insertions(+)
 create mode10/16/2018 100644October 16, 2018 homepage/index.html
```

Working with remote repositories

git clone

To create a local working copy of an existing remote repository, use *git clone* to copy and download the repository to a computer. Cloning is the equivalent of *git init* when working with a remote repository. Git will create a directory locally with all files and repository history.

Usage:

```
$ git clone git@account_name.git.beanstalkapp.com:/acccount_name/repository_name.git
Cloning into 'repository_name'...
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 5 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (5/5), 3.08 KiB | 0 bytes/s, done.
Checking connectivity... done.
```

git pull

To get the latest version of a repository run *git pull*. This pulls the changes from the remote repository to the local computer.

Usage:

```
$ git pull <branch_name> <remote_URL/remote_name>
```

In Practice:

```
# Pull from named remote
$ git pull origin staging
From account_name.git.beanstalkapp.com:/account_name/repository_name
 * branch            staging    -> FETCH_HEAD
 * [new branch]      staging    -> origin/staging
Already up-to-date.

# Pull from URL (not frequently used)
$ git pull git@account_name.git.beanstalkapp.com:/acccount_name/repository_name.git staging
From account_name.git.beanstalkapp.com:/account_name/repository_name
 * branch            staging    -> FETCH_HEAD
 * [new branch]      staging    -> origin/staging
Already up-to-date.
```

Advanced Git Commands

git stash

To save changes made when they're not in a state to commit them to a repository. This will store the work and give a clean working directory. For instance, when working on a new

```
# Store current work
$ git stash -u
Saved working directory and index state WIP on SecretTesting: 4c0f37c Adding new file to branch
HEAD is now at 4c0f37c Adding new file to branch

# Bring stashed work back to the working directory
$ git stash pop
On branch SecretTesting
Your branch and 'origin/SecretTesting' have diverged,
and have 1 and 1 different commit each, respectively.
  (use "git pull" to merge the remote branch into yours)
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   index.html

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (3561897724c1f448ae001edf3ef57415778755ec)
```

git log

To show the chronological commit history for a repository. This helps give context and history for a repository. *git log* is available immediately on a recently cloned repository to see history.

```
# Show entire git log
$ git log
commit 4c0f37c711623d20fc60b9cbcf393d515945952f
Author: Brian Kerr <my@emailaddress.com>
Date:   Tue Oct 25 17:46:11 2016 -0500

    Updating the wording of the homepage footer

commit 0254c3da3add4ebe9d7e1f2e76f015a209e1ef67
Author: Ashley Harpp <my@emailaddress.com>
Date:   Wed Oct 19 16:27:27 2016 -0500

    My first commit message

# Show git log with date pameters
$ git log --before="Oct 20"
commit 0254c3da3add4ebe9d7e1f2e76f015a209e1ef67
Author: Ashley Harpp <my@emailaddress.com>
Date:   Wed Oct 19 16:27:27 2016 -0500

    My first commit message
```

```
# Show git log based on commit author
$ git log --author="Brian Kerr"
commit 4c0f37c711623d20fc60b9cbcf393d515945952f
Author: Brian Kerr <my@emailaddress.com>
Date:   Tue Oct 25 17:46:11 2016 -0500

    Updating the wording of the homepage footer
```

git rm

Remove files or directories from the working index (staging area). With *git rm*, there are two options to keep in mind: force and cached. Running the command with force deletes the file. The cached command removes the file from the working index. When removing an entire directory, a recursive command is necessary.

Usage:

```
# To remove a file from the working index:
$ git rm --cached css/style.css
rm 'css/style.css'

# To delete a file (force):
$ git rm -f css/style.css
rm 'css/style.css'

# To remove an entire directory from the working index (cached):
$ git rm -r --cached css/
rm 'css/style.css'
rm 'css/style.min.css'

# To delete an entire directory (force):
$ git rm -r -f css/
rm 'css/style.css'
rm 'css/style.min.css'
```

Basic Git commands

Go to the master branch to pull the latest changes from there

```
git checkout master
```

Download the latest changes in the project

This is for you to work on an up-to-date copy (it is important to do this every time you start working on a project), while you set up tracking branches. You pull from remote repositories to get all the changes made by users since the last time you cloned or pulled the project. Later, you can push your local commits to the remote repositories.

```
git pull REMOTE NAME-OF-BRANCH
```

When you first clone a repository, REMOTE is typically "origin". This is where the repository came from, and it indicates the SSH or HTTPS URL of the repository on the remote server. NAME-OF-BRANCH is usually "master", but it may be any existing branch.

View your remote repositories

To view your remote repositories, type:

```
git remote -v
```

Create a branch

To create a branch, type the following (spaces won't be recognized in the branch name, so you will need to use a hyphen or underscore):

```
git checkout -b NAME-OF-BRANCH
```

Work on an existing branch

To switch to an existing branch, so you can work on it:

```
git checkout NAME-OF-BRANCH
```

View the changes you've made

It's important to be aware of what's happening and the status of your changes. When you add, change, or delete files/folders, Git knows about it. To check the status of your changes:

```
git status
```

View differences

To view the differences between your local, unstaged changes and the repository versions that you cloned or pulled, type:

```
git diff
```

Add and commit local changes

You'll see your local changes in red when you type `git status`. These changes may be new, modified, or deleted files/folders. Use `git add` to stage a local file/folder for committing. Then use `git commit` to commit the staged files:

```
git add FILE OR FOLDER

git commit -m "COMMENT TO DESCRIBE THE INTENTION OF THE COMMIT"
```

Add all changes to commit

To add and commit all local changes in one command:

```
git add .

git commit -m "COMMENT TO DESCRIBE THE INTENTION OF THE COMMIT"
```

Note: The `.` character typically means *all* in Git.

Send changes to gitlab.com

To push all local commits to the remote repository:

```
git push REMOTE NAME-OF-BRANCH
```

For example, to push your local commits to the *master* branch of the *origin* remote:

```
git push origin master
```

Delete all changes in the Git repository

To delete all local changes in the repository that have not been added to the staging area, and leave unstaged files/folders, type:

```
git checkout .
```

Delete all untracked changes in the Git repository

```
git clean -f
```

Unstage all changes that have been added to the staging area

To undo the most recent add, but not committed, files/folders:

```
git reset .
```

Undo most recent commit

To undo the most recent commit, type:

```
git reset HEAD~1
```

This leaves the files and folders unstaged in your local repository.

Warning: A Git commit is mostly irreversible, particularly if you already pushed it to the remote repository. Although you can undo a commit, the best option is to avoid the situation altogether.

Merge created branch with master branch

You need to be in the created branch.

```
git checkout NAME-OF-BRANCH

git merge master
```

Merge master branch with created branch

You need to be in the master branch.

```
git checkout master

git merge NAME-OF-BRANCH
```

Differences between user and organization accounts

Your user account is your identity on GitHub. Your user account can be a member of any number of organizations, regardless of whether the account is on a free or paid plan.

Personal user accounts

Every person who uses GitHub has their own user account. These accounts include:

- Unlimited public repositories and collaborators for free
- Paid plan with unlimited private repositories
- Ability to invite unlimited repository collaborators

Tips:

- It may be tempting to have more than one user account, such as for personal use and business use, but you only need one account. For more information, see "Merging multiple user accounts."
- User accounts are intended for humans, but you can give one to a robot, such as a continuous integration bot, if necessary.

Organizations

Organizations are shared accounts where groups of people can collaborate across many projects at once. Owners and administrators can manage member access to the organization's data and projects with sophisticated security and administrative features.

Organizations include:

- A free plan with unlimited collaborators on unlimited public repositories
- The option to upgrade to paid plans with unlimited private repositories, sophisticated user authentication and management, 24/5 support, and a service level agreement for uptime availability

- Unlimited membership with a variety of roles that grant different levels of access to the organization and its data
- The ability to give members a range of access permissions to your organization's repositories
- Nested teams that reflect your company or group's structure with cascading access permissions and mentions
- The ability for organization owners to view members' two-factor authentication (2FA) status
- The option to require all organization members to use two-factor authentication

Can I give read only (pull only) access to specific user(s) in private repository?

Want to evaluate Docker Hub for our project, but we definitely need this feature.

Yes,

1. Go To Teams
2. Create a Team
3. Add the user to that team.
4. Go to your repositories
5. Click on repository
6. Click on Collaborators
7. Choose the team created from step 3 and Access rights.

1. What is GIT version control ? (GIT Interview Questions)
Answer: With the help of GIT version control, you can track the history of a collection of files and includes the functionality to revert the collection of files to another version. Each version captures a snapshot of the file system at a certain point of time. A collection of files and their complete history are stored in a repository. (GIT Interview Questions) (*GIT Interview Questions*)

2. What is the difference between 'git remote' and 'git clone'?
Answer: 'git remote add' just creates an entry in your git config that specifies a name for a particular URL. While, 'git clone' creates a new git repository by copying and existing one located at the URI.

3. How can you bring a new feature in the main branch ?
Answer: To bring a new feature in the main branch, you can use a command "git merge" or "git pull command".

4. Mention some of the best graphical GIT client for LINUX ?
Answer: Every node (or Puppet Agent) has got its configuration details in Puppet Master, written in the native Puppet language. These details are written in the language which Puppet can understand and are termed as Manifests. Manifests are composed of Puppet code and their filenames use the .pp extension.

Now give an example, you can write a manifest in Puppet Master that creates a file and installs apache on all Puppet Agents (Slaves) connected to the Puppet Master.

5. What is the function of 'git diff ' in git ?
Answer: 'git diff ' shows the changes between commits, commit and working tree etc.

6. Skipped.

7. What is 'git status' is used for ?
Answer: As 'Git Status' shows you the difference between the working directory and the index, it is helpful in understanding a git more comprehensively.

8. What is the difference between the 'git diff 'and 'git status' ?
Answer: 'git diff' is similar to 'git status', but it shows the differences between various commits and also between the working directory and index. (GIT Interview Questions)
9. What is the function of 'git checkout' in git ?
Answer: A 'git checkout' command is used to update directories or specific files in your working tree with those from another branch without merging it in the whole branch.

10. What is the function of 'git rm' ?

Answer: To remove the file from the staging area and also off your disk 'git rm' is used.(*GIT Interview Questions*)

11. What is the function of 'git stash apply' ?
Answer: When you want to continue working where you have left your work, 'git stash apply' command is used to bring back the saved changes onto the working directory.(*GIT Interview Questions*)

12. What is the use of 'git log' ?
Answer: To find specific commits in your project history- by author, date, content or history 'git log' is used.

13. What is 'git add' is used for ?
Answer: 'git add' adds file changes in your existing directory to your index.

14. What is the function of 'git reset' ?
Answer: The function of 'Git Reset' is to reset your index as well as the working directory to the state of your last commit.

15. What is git Is-tree ?
Answer: 'git Is-tree' represents a tree object including the mode and the name of each item and the SHA-1 value of the blob or the tree.

(*GIT Interview Questions*)
16. How git instaweb is used ?
Answer: 'Git Instaweb' automatically directs a web browser and runs webserver with an interface into your local repository. (GIT Interview Questions)

17. What does 'hooks' consist of in git ?
Answer: This directory consists of Shell scripts which are activated after running the corresponding Git commands. For example, git will try to execute the post-commit script after you run a commit.

18. Explain what is commit message ?
Answer: Commit message is a feature of git which appears when you commit a change. Git provides you a text editor where you can enter the modifications made in commits.(*GIT Interview Questions*)
19. How can you fix a broken commit ?
Answer: To fix any broken commit, you will use the command "git commit—amend". By running this command, you can fix the broken commit message in the editor.

20. Why is it advisable to create an additional commit rather than amending an existing commit ?
Answer: There are a couple of reason

a.The amend operation will damage the case that was earlier accumulated in a commit. If it's simply the commit information being converted then that's not a problem. However, if the contents are being amended then possibilities of reducing something significant prevails more.
b. Exploiting "git commit- amend" can generate a small commit to develop and obtain irrelevant variations.(*GIT Interview Questions*)

21. What is 'bare repository' in GIT ?
Answer: To co-ordinate with the distributed development and developers team, especially when you are working on a project from multiple computers 'Bare Repository' is used. A bare repository comprises of a version history of your code.(*GIT Interview Questions*)

22. Name a few Git repository hosting services?
Answer:

- Pikacode
- Visual Studio Online
- GitHub
- GitEnterprise
- SourceForge.net

23. What is git rerere?
Answer: In GIT, rerere is a hidden feature. The full form of rerere is "reuse recorded resolution".
By using rerere, GIT remembers how we've resolved a hunk conflict. The next time GIT sees the same conflict, it can automatically resolve it for us.
(GIT Interview Questions)

24. What does a commit object contain?
Answer: Whenever we do a commit in GIT by using git commit command, GIT creates a new commit object. This commit objects is saved to GIT repository.
The commit object contains following information:
HASH: The SHA1 hash of the Git tree that refers to the state of index at commit time.
Commit Author: The name of person/process doing the commit and date/time.
Comment: Some text messages that contains the reason for the commit.

25. What is cherry-pick in GIT?
Answer: A git cherry-pick is a very useful feature in GIT. By using this command we can selectively apply the changes done by existing commits.
In case we want to selectively release a feature, we can remove the unwanted files and apply only selected commits. (*GIT Interview Questions*)

26. What is shortlog in GIT?

Answer: A shortlog in GIT is a command that summarizes the git log output. The output of git shortlog is in a format suitable for release announcements.

27. What is git grep ?
Answer: GIT is shipped along with a grep command that allows us to search for a string or regular expression in any committed tree or the working directory. By default, it works on the files in your current working directory.

28. How can your reorder commits in GIT?
Answer: We can use git rebase command to reorder commits in GIT. It can work interactively and you can also select the ordering of commits.

29. What is filter-branch in GIT?
Answer: In GIT, filter-branch is another option to rewrite history. It can scrub the entire history. When we have large number of commits, we can use this tool.
It gives many options like removing the commit related changes to a specific file from history.
You can even set you name and email in the commit history by using filter-branch.

30. What is a submodule in GIT?
Answer: In GIT, we can create sub modules inside a repository by using git submodule command.
By using submodule command, we can keep a Git repository as a subdirectory of another Git repository.
It allows us to keep our commits to submodule separate from the commits to main Git repository.

GitHub Glossary

Below are a list of some Git and GitHub specific terms we use across our sites and documentation.

Blame

The "blame" feature in Git describes the last modification to each line of a file, which generally displays the revision, author and time. This is helpful, for example, in tracking down when a feature was added, or which commit led to a particular bug.

Branch

A branch is a parallel version of a repository. It is contained within the repository, but does not affect the primary or `master` branch allowing you to work freely without disrupting the "live" version. When you've made the changes you want to make, you can merge your branch back into the `master` branch to publish your changes. For more information, see "About branches."

Check

A check is a type of status check on GitHub. See "Status checks."

Clone

A clone is a copy of a repository that lives on your computer instead of on a website's server somewhere, or the act of making that copy. With your clone you can edit the files in your preferred editor and use Git to keep track of your changes without having to be online. It is, however, connected to the remote version so that changes can be synced between the two. You can push your local changes to the remote to keep them synced when you're online.

Collaborator

A collaborator is a person with read and write access to a repository who has been invited to contribute by the repository owner.

Commit

A commit, or "revision", is an individual change to a file (or set of files). It's like when you *save* a file, except with Git, every time you save it creates a unique ID (a.k.a. the "SHA" or "hash") that allows you to keep record of what changes were made when and by who. Commits usually contain a commit message which is a brief description of what changes were made.

Contributor

A contributor is someone who has contributed to a project by having a pull request merged but does not have collaborator access.

Dashboard

Your personal dashboard is the main hub of your activity on GitHub. From your personal dashboard, you can keep track of issues and pull requests you're following or working on, navigate to your top repositories and team pages, and learn about recent activity in

repositories you're watching or participating in. You can also discover new repositories, which are recommended based on users you're following and repositories you have starred. To only view activity for a specific organization, visit your organization's dashboard. For more information, see "About your personal dashboard" or "About your organization dashboard."

Diff

A diff is the *difference* in changes between two commits, or saved changes. The diff will visually describe what was added or removed from a file since its last commit.

Fetch

Fetching refers to getting the latest changes from an online repository without merging them in. Once these changes are fetched you can compare them to your local branches (the code residing on your local machine).

Fork

A fork is a personal copy of another user's repository that lives on your account. Forks allow you to freely make changes to a project without affecting the original. Forks remain attached to the original, allowing you to submit a pull request to the original's author to update with your changes. You can also keep your fork up to date by pulling in updates from the original.

Git

Git is an open source program for tracking changes in text files, and is the core technology that GitHub, the social and user interface, is built on top of.

Issue

Issues are suggested improvements, tasks or questions related to the repository. Issues can be created by anyone (for public repositories), and are moderated by repository collaborators. Each issue contains its own discussion forum, can be labeled and assigned to a user.

Markdown

Markdown is a simple semantic file format, not too dissimilar from .doc, .rtf and .txt. Markdown makes it easy for even those without a web-publishing background to write prose (including with links, lists, bullets, etc.) and have it displayed like a website. GitHub supports Markdown, and you can learn about the semantics.

Merge

Merging takes the changes from one branch (in the same repository or from a fork), and applies them into another. This often happens as a pull request (which can be thought of as a request to merge), or via the command line. A merge can be done automatically via a pull request via the GitHub web interface if there are no conflicting changes, or can always be done via the command line. For more information, see "Merging a pull request."

Open source

Open source software is software that can be freely used, modified, and shared (in both modified and unmodified form) by anyone. Today the concept of "open source" is often

extended beyond software, to represent a philosophy of collaboration in which working materials are made available online for anyone to fork, modify, discuss, and contribute to. For more information on open source, specifically how to create and grow an open source project, we've created Open Source Guides that will help you foster a healthy open source community. You can also take a free GitHub Learning Lab course on maintaining open source communities.

Organizations
Organizations are shared accounts where businesses and open-source projects can collaborate across many projects at once. Owners and administrators can manage member access to the organization's data and projects with sophisticated security and administrative features.

Private repository
Private repositories are repositories that can only be viewed or contributed to by their creator and collaborators the creator specified.

Pull
Pull refers to when you are fetching *in* changes *and* merging them. For instance, if someone has edited the remote file you're both working on, you'll want to *pull* in those changes to your local copy so that it's up to date.

Pull request
Pull requests are proposed changes to a repository submitted by a user and accepted or rejected by a repository's collaborators. Like issues, pull requests each have their own discussion forum. For more information, see "About pull requests."

Push
Pushing refers to sending your committed changes to a remote repository, such as a repository hosted on GitHub. For instance, if you change something locally, you'd want to then *push* those changes so that others may access them.

Remote
This is the version of something that is hosted on a server, most likely GitHub. It can be connected to local clones so that changes can be synced.

Repository
A repository is the most basic element of GitHub. They're easiest to imagine as a project's folder. A repository contains all of the project files (including documentation), and stores each file's revision history. Repositories can have multiple collaborators and can be either public or private.

SSH key
SSH keys are a way to identify yourself to an online server, using an encrypted message. It's as if your computer has its own unique password to another service. GitHub uses SSH keys to securely transfer information to your computer.

Status
A status is a type of status check on GitHub. See "Status checks."

Status checks
Status checks are external processes, such as continuous integration builds, which run for each commit you make in a repository. For more information, see "About status checks."

Team
Teams are groups of organization members that reflect your company or group's structure with cascading access permissions and mentions.

Upstream
When talking about a branch or a fork, the primary branch on the original repository is often referred to as the "upstream", since that is the main place that other changes will come in from. The branch/fork you are working on is then called the "downstream".

User
Users are personal GitHub accounts. Each user has a personal profile, and can own multiple repositories, public or private. They can create or be invited to join organizations or collaborate on another user's repository.

https://git-scm.com/book/en/v2/Git-Internals-Environment-Variables

Git Internals - Transfer Protocols

Transfer Protocols
Git can transfer data between two repositories in two major ways: the “dumb” protocol and the “smart” protocol. This section will quickly cover how these two main protocols operate.

https://gist.github.com/P7h/d5631d640ab91ed4a8e2e4732ff691d9
https://www.skorks.com/2009/09/bash-shortcuts-for-maximum-productivity/

Bash command line Shortcuts

Picked these from here

Command Editing Shortcuts

Command	Note
`Ctrl` + `a`	go to the start of the command line
`Ctrl` + `e`	go to the end of the command line
`Ctrl` + `k`	delete from cursor to the end of the command line
`Ctrl` + `u`	delete from cursor to the start of the command line
`Ctrl` + `w`	delete from cursor to start of word (i.e. delete backwards one word)
`Ctrl` + `y`	paste word or text that was cut using one of the deletion shortcuts (such as the one above) after
`Ctrl` + `xx`	move between start of command line and current cursor position (and back again)
`Alt` + `b`	move backward one word (or go to start of word the cursor is currently on)
`Alt` + `f`	move forward one word (or go to end of word the cursor is currently on)
`Alt` + `d`	delete to end of word starting at cursor (whole word if cursor is at the beginning of word)
`Alt` + `c`	capitalize to end of word starting at cursor (whole word if cursor is at the beginning of word)
`Alt` + `u`	make uppercase from cursor to end of word
`Alt` + `l`	make lowercase from cursor to end of word
`Alt` + `t`	swap current word with previous
`Ctrl` + `f`	move forward one character
`Ctrl` + `b`	move backward one character
`Ctrl` + `d`	delete character under the cursor
`Ctrl` + `h`	delete character before the cursor
`Ctrl` + `t`	swap character under cursor with the previous one

Command Recall Shortcuts

CommandNote

`Ctrl` + `r` search the history backwards
`Ctrl` + `g` escape from history searching mode
`Ctrl` + `p` previous command in history (i.e. walk back through the command history)
`Ctrl` + `n` next command in history (i.e. walk forward through the command history)
`Alt` + `.` use the last word of the previous command

Command Control Shortcuts

CommandNote

`Ctrl` + `l` clear the screen
`Ctrl` + `s` stops the output to the screen (for long running verbose command)
`Ctrl` + `q` allow output to the screen (if previously stopped using command above)
`Ctrl` + `c` terminate the command
`Ctrl` + `z` suspend/stop the command

Bash Bang (!) Commands

Bash also has some handy features that use the `!` (bang) to allow you to do some funky stuff with bash commands.

CommandNote

`!!` run last command
`!blah` run the most recent command that starts with 'blah' (e.g. `!ls`)
`!blah:p` print out the command that !blah would run (also adds it as the latest command in the command history)
`!$` the last word of the previous command (same as `Alt` + `.`)
`!$:p` print out the word that `!$` would substitute
`!*` the previous command except for the last word (e.g. if you type 'find some_file.txt /', then `!|` would give you 'find some_file.txt')
`!*:p` print out what `!|` would substitute

References

1. **Gunther, Tobias. Learn Version Control with Git. https://www.git-tower.com/learn/assets/files/ebook-learn_version_control_with_git-SAMPLE.pdf.**

2. **Atten, Jon. Basic Git Command Line Reference for Windows Users.**