**Advanced GIT**

**Git log**

|  |  |
| --- | --- |
| **Command** | **Description** |
| Git log | Shows a log of commits |
| Git log -n 2 | Returns the most recent # commits |
| Git log --since=YYYY-MM-DD | Search commits since date specified |
| Git log --author=”Ali” | Search commits by author |
| Git log --grep”regexp” | Global Regular ExPression search |
| Git log --oneline | Displays each commit on one line |

Each commit has an ID, which is a SHA1 hash key, an author, date and a commit message.

**The three-tree architecture**

Repository

Commit

Checkout

Staging

Add

Working

The repository stores all the versions of files. The working directory only shows the current local/working version.

**Working with the HEAD pointer**

HEAD always points to the tip off the current branch in repository.

Example:

|  |  |  |
| --- | --- | --- |
| **Master branch** | | |
| Commit #1 | Commit #2 (HEAD) |  |
| **Sub branch** | | |
| Commit #1 | Commit #2 | Commit #3 (HEAD) |

**Adding files**

Git status shows the difference between the 3 trees. If it is ‘clean’, this means our working tree matches exactly the repository that the HEAD is pointing to.

Git add <file> adds a file to the staging area.

To view changes

|  |  |
| --- | --- |
| **Command** | **Description** |
| Git diff | Shows changes between the repository and the working trees.   * Repo   + Working/new version |
| Git diff --staged | Compares the repo and staging trees |

Renaming

Git mv initialfile.txt newname.txt

Git will automatically send this to staging area and will know it is a rename.

*- OR -*

Git rm initialfile.txt

Git add newname.txt

Undo changes

If changes are made in the working directory but you have decided you no longer want them, you can undo the changes and revert back to the repository version via:

Git checkout --filename.txt

Unstage staged files

Git reset HEAD filename.txt

**Commits**

Amending commits

You can *only* amend the last commit message:

Git commit --amend -m”message”

Retrieving old versions

Git checkout SHA1() --filename.txt

*Where SHA1() is the commit id, e.g. 290ab7d*

This will put the file in to the staged area where you can commit it back in to the repository, and this will now be at the HEAD, i.e. the new updated file in the repo.

Reverting a commit

Git revert SHA1()

To examine a commit

Git show SHA1()

Using reset to undo commits

\*\*USE WITH CAUTION\*\*

By default, git positions the HEAD automatically. Git reset puts you in control!

--soft Does not change the staging index or working directory

--mixed Default. Changes the staging index to match the repo. Not the working dir

--hard Changes the staging and working directories to match the repo

e.g. git reset --soft SHA1()

Removing untracked files

If you create files which you no longer want, you can remove them prior to adding and committing them with:

Git clean -n //tells you what will be removed

Git clean -f //removes the files

Referencing commits

Git ls-tree HEAD

//instead of HEAD you can use other keywords such as commit id, branch name, paths…

This will display a list of files in the HEAD (or other keyword).

*Blob* A file

*Tree* Directories

Getting more from commit log

|  |  |
| --- | --- |
| **Command** | **Description** |
| Git log --oneline | Displays commits on single lines |
| Git log --online -1 | Displays the specified number of commits |
| Git log --author=”Ali”  Git log --since=”YYYY-MM-DD”  Git log --since=”2 days ago” |  |
| Git log --grep=”regexp” -i | -i = case insensitive |
| Git log --graph | Shows a graph of branches |

*These commands can be binded together*

**.gitignore**

You can use regular expressions: \* ? [aeiou] [0-9]

You can negate expressions with: !

E.g.

#ignore php files

\*.php

#do not ignore index.php

!index.php

**Branch**

When you are not trying to checkout a branch, it is good practice to put -- before the filename. This tells git to stay on the current branch. This is because you may have a branch with the same name as a file.

**Merging**

Go to the branch which you would like to merge the other branch with, for example; if you would like to merge a branch in with the master branch:

Git checkout master

Git merge <branchtomerge>

Use: *git branch --merged* to see what branches are merged with the current branch. You should only merge with clean trees. Once merged you can now consider deleting the branch.

Conflicts

When trying to merge with conflicts, git will inform you of this. *Git status* will show the conflict status.

Open the conflicted file in an IDE and git will show you where the conflicts are.

Resolving merge conflicts

3 options:

1. Abort the merge with *git merge --abort*
2. Resolve the conflicts manually (most likely)
3. Use a merge tool

When you have conflicts, they will show in your files like the following:

<<<<<<<< HEAD

Conflicting code here

=================

You need to make edits and remove what is not needed from the conflicts, including the git inputted markers. From here you can add to the staging area and commit as usual. You can now check it has merged using *git branch --merged*

To avoid merges, make sure to *git pull* often.

Stashing changes

The stash is a place we can store changes temporarily without having to commit them to the repository. It is separate from the 3 trees. However, they are similar to commits, but they do not have a SHA1 associated to them.

Git stash save “message”

The message doesn’t get stored when committed, this is just for your benefit so the description doesn’t matter all that much.

To view stash: *git stash list*

\*\*You can be on any branch to pull items from the stash\*\*

To show changes: *git stash stashId //*Stash id e.g.: stash@{1}

Retrieving a stash

Note that there may be conflicts when pulling in a stash, these work the same as other conflicts and will have to be resolved accordingly.

1. Git stash pop stash@{0}
2. Git stash apply stash@{0}

*Apply* keeps the file in the stash, whereas *pop* removes the file from the stash after retrieving it. If you do not specify the stash item, git will get the first one.

Deleting stashed items

1. Git stash drop stash@{0} //delete individual
2. Get stash clear //delete all items in the stash

**Remotes**

|  |  |  |
| --- | --- | --- |
| **Remote branch** | | |
|  |  |  |
| **Local branch** | | |
| Commit #1 | Commit #2 | Commit #3 |

The remote branch will not receive the commits on the local branch until you *push* the commits up to it via *git push.*

When collaborators make local changes on their machine and push them to the remote server, we need to *fetch* them with: *git fetch*.

When you create a remote server and connect to it, your local repository creates another branch called *origin* (by default). When you *fetch* data from a repository, that data is stored in the origin branch but not yet in the working directory/master branch. At this point, we need to do a *merge* to merge the origin branch with the master branch.

*Git pull* basically does a *git fetch & git merge origin/master*

Adding a remote

*Git remote add <alias> <url>* //alias = branch name (default = origin)

//url = e.g. github repository

You can have more than one remote branch.

To see all remotes: *git remote*

To push your local changes to the remote branch: *git push -u origin master*

Where *origin* is the name of the remote branch, and *master* is the branch to push up to the remote branch. Note that you can push other branches up to the remotes, not just the master branch.

Cloning

Cloning is when there is already a remote branch and you want to clone/copy it to your local machine in order to work on it locally.

For example; find a project you wish to work on, on GitHub:

*Git clone <url> <local folder name>*

This creates a local working directory, it will bring down the master branch.

Collaborating

Find an open source project you want to work on and fork it. Forking a project puts it on your local GitHub/other repository, from here you can clone it and work on it locally.

By default, you will only have read access on other users’ repositories (not write access). Therefore, you need to fork it, clone it and then make changes. In order to propose you changes to the open source/original authors repository, you need to submit a pull request with a message of your changes. The author can then reply, accept/merge or decline your changes.

To get a remote branch from a repository that another user created:

*Git checkout -b <local branch name> <remote branch name>*

*Git checkout -b feedback\_form origin/feedback\_form*

**TEST**

**Part 1**

1. Git log:
   1. Display on a single line
   2. Display the last 3 commits
   3. Grep
   4. Display as graph
2. Explain the 3-tree architecture
3. Describe the HEAD pointer
4. How do you see the changes between the repository and the working tree?
5. How do you see the changes between the repository and the staging area?
6. Rename a file using git
7. How can you undo changes made in the working directory?
8. How do you un-stage a staged file?

**Part 2**

1. Amend the last commit message
2. Checkout an old commit, edit it and commit it back in
3. How do you examine a commit further?
4. Explain *git reset*
5. How can you delete untracked files using git
6. How can you see all the files changed in a:
   1. Commit
   2. Branch
   3. HEAD
7. In .gitignore, ignore all the PHP files except for index.php
8. How should you checkout a file (as opposed to a branch)

**Part 3**

1. Create a sub-branch of master, edit and commit it, then merge it with the master branch
2. Create a conflict, resolve it and then commit it
3. Explain what a stash is and create one
4. List all your stashes
5. Retrieve a stash (2 ways)
6. Remove a stash:
   1. Individually
   2. All
7. What does *git fetch* do?
8. What 2 (exact) commands make up a *git pull*
9. How do you add a remote branch?
10. Push a sub-branch to a remote branch
11. What is the difference between *forking* and *cloning*?
12. How do you get a remote sub-branch to your local/working directory?