Al-Azhar UNIVERSITY

Faculty of Engineering

Computers and Systems Engineering Department

EXPERIMENT 1 – Git

OBJECTIVES

- Understand the PC configuration required to use a git repository
- Understand how to create a local copy of a GitHub repository.
- Understand the conceptual areas: working tree, staging area
- Understand the basic update workflow: add to staging, commit with message, push to origin

MATERIALS/EQUIPMENT NEEDED

1. Git tool

INTRODUCTION

Git is a distributed revision control and source code management system with an emphasis on speed. Git was initially designed and developed by Linus Torvalds for Linux kernel development. Git is a free software distributed under the terms of the GNU General Public License version 2. This tutorial explains how to use Git for project version control in a distributed environment while working on web-based and non web-based applications development.

Version Control System (VCS) is a software that helps software developers to work together and maintain a complete history of their work.

Listed below are the functions of a VCS –

- Allows developers to work simultaneously.
- Does not allow overwriting each other's changes.
- Maintains a history of every version.

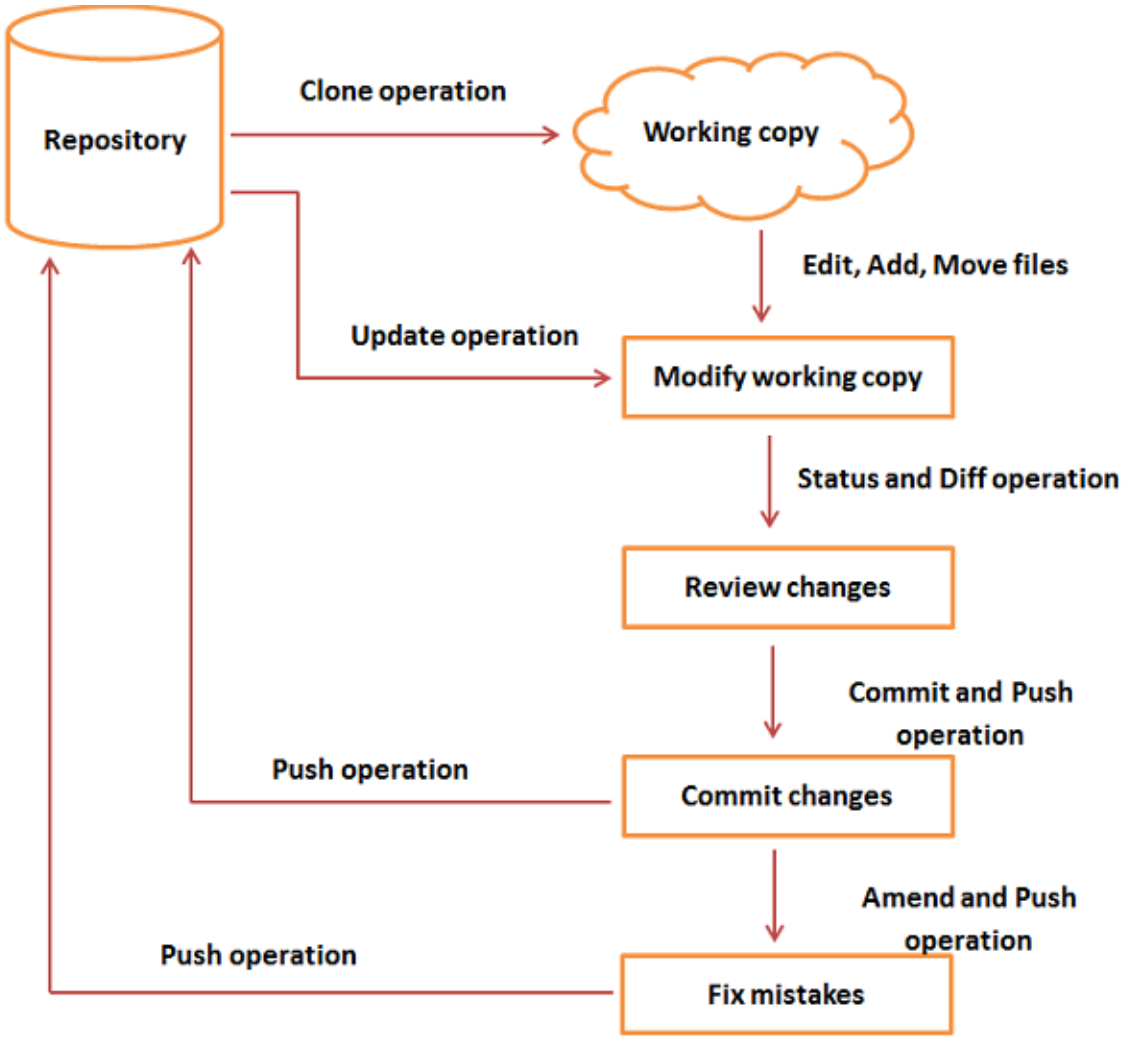
Git - Life Cycle

General workflow is as follows –

- You clone the Git repository as a working copy.
- You modify the working copy by adding/editing files.

- If necessary, you also update the working copy by taking other developer's changes.
- You review the changes before commit.
- You commit changes. If everything is fine, then you push the changes to the repository.
- After committing, if you realize something is wrong, then you correct the last commit and push the changes to the repository.

Shown below is the pictorial representation of the work-flow.



Git Operations

First, note that you can get documentation for a command such as

```
git help log
```

Create a Bare Repository in `myProject.git` folder

```
git init myProject.git --bare
```

Clone the project from Git server

```
git clone <path_to_your_server>
```

Making changes

Modify some files, then add their updated contents to the index or stage:

```
$ git add file1 file2 file3, or
$ git add . // add all files
```

You are now ready to commit. You can see what is about to be committed using **git diff** with the --cached option:

```
$ git diff --cached
```

(Without --cached, **git diff** will show you any changes that you've made but not yet added to the index.) You can also get a brief summary of the situation with **git status**:

```
$ git status
```

If you need to make any further adjustments, do so now, and then add any newly modified content to the index. Finally, commit your changes with:

```
$ git commit ,or
$ git commit -m 'message'
```

This will again prompt you for a message describing the change, and then record a new version of the project.

Alternatively, instead of running **git add** beforehand, you can use

```
$ git commit -a
```

which will automatically notice any modified (but not new) files, add them to the index, and commit, all in one step.

Git tracks content not files

Many revision control systems provide an `add` command that tells the system to start tracking changes to a new file. Git's `add` command does something simpler and more powerful: **git add** is used both for new and newly modified files, and in both cases it takes a snapshot of the given files and stages that content in the index, ready for inclusion in the next commit.

Viewing project history

At any point you can view the history of your changes using

```
$ git log
```

If you also want to see complete diffs at each step, use

```
$ git log -p
```

Often the overview of the change is useful to get a feel of each step

```
$ git log --stat --summary
```

Push your changes into master repository

```
$ git push origin master
```

Managing branches

A single Git repository can maintain multiple branches of development. To create a new branch named "experimental", use

```
$ git branch experimental
```

If you now run

```
$ git branch
```

you'll get a list of all existing branches:

```
  experimental
* master
```

The "experimental" branch is the one you just created, and the "master" branch is a default branch that was created for you automatically. The asterisk marks the branch you are currently on; type

```
$ git checkout experimental ,or git switch experimental
```

to switch to the experimental branch. Now edit a file, commit the change, and switch back to the master branch:

```
(edit file)

$ git commit -a
$ git switch master
```

Check that the change you made is no longer visible, since it was made on the experimental branch and you're back on the master branch.

You can make a different change on the master branch:

```
(edit file)

$ git commit -a
```

at this point the two branches have diverged, with different changes made in each. To merge the changes made in experimental into master, run

```
$ git merge experimental
```

If the changes don't conflict, you're done. If there are conflicts, markers will be left in the problematic files showing the conflict.

```
$ git diff
```

will show this. Once you've edited the files to resolve the conflicts,

```
$ git commit -a
```

will commit the result of the merge. Finally,

```
$ gitk
```

will show a nice graphical representation of the resulting history.

At this point you could delete the experimental branch with

```
$ git branch -d experimental
```

This command ensures that the changes in the experimental branch are already in the current branch.

If you develop on a branch crazy-idea, then regret it, you can always delete the branch with

```
$ git branch -D crazy-idea
```

Branches are cheap and easy, so this is a good way to try something out.

Stash Operation

Suppose you are implementing a new feature for your product. Your code is in progress and suddenly a customer escalation comes. Because of this, you have to keep aside your new feature work for a few hours. You cannot commit your partial code and also cannot throw away your changes. So you need some temporary space, where you can store your partial changes and later on commit it.

In Git, the stash operation takes your modified tracked files, stages changes, and saves them on a stack of unfinished changes that you can reapply at any time.

```
git stash
```

Now, your working directory is clean and all the changes are saved on a stack.

Now you can safely switch the branch and work elsewhere. We can view a list of stashed changes by using the **git stash list** command.

```
git stash list
```

Suppose you have resolved the customer escalation and you are back on your new feature looking for your half-done code, just execute the **git stash pop** command, to remove the changes from the stack and place them in the current working directory.

```
git stash pop
```

Exploring history

Git history is represented as a series of interrelated commits. We have already seen that the **git log** command can list those commits. Note that first line of each git log entry also gives a name for the commit:

```
$ git log

commit c82a22c39cbc32576f64f5c6b3f24b99ea8149c7
Author: Junio C Hamano <junkio@cox.net>
Date:   Tue May 16 17:18:22 2006 -0700
    merge-base: Clarify the comments on post processing.
```

We can give this name to **git show** to see the details about this commit.

```
$ git show c82a22c39cbc32576f64f5c6b3f24b99ea8149c7 # You can use
any initial part of the name that is long enough to uniquely
identify the commit
```

Every commit usually has one "parent" commit which points to the previous state of the project:

```
$ git show HEAD^  # to see the parent of HEAD
$ git show HEAD^^ # to see the grandparent of HEAD
$ git show HEAD~4 # to see the great-great grandparent of HEAD
```

Note that merge commits may have more than one parent:

```
$ git show HEAD^1 # show the first parent of HEAD (same as HEAD^)
$ git show HEAD^2 # show the second parent of HEAD
```

You can also give commits names of your own; after running

```
$ git tag v2.5 1b2e1d63ff # git tag -a 'Release_1_0' -m 'Tagged
basic string operation code' HEAD
```

you can refer to 1b2e1d63ff by the name "v2.5". If you intend to share this name with other people (for example, to identify a release version), you should create a "tag" object, and perhaps sign it.

Any Git command that needs to know a commit can take any of these names. For example:

```
$ git diff v2.5 HEAD   # compare the current HEAD to v2.5
$ git branch stable v2.5 # start a new branch named "stable" based
                         # at v2.5

$ git reset --hard HEAD^ # reset your current branch and working
                         # directory to its state at HEAD^
```

Be careful with that last command: in addition to losing any changes in the working directory, it will also remove all later commits from this branch. If this branch is the only branch containing those commits, they will be lost. Also, don't use **git reset** on a publicly-visible branch that other developers pull from, as it will force needless merges on other developers to clean up the history. If you need to undo changes that you have pushed, use **git revert** instead.

The **git grep** command can search for strings in any version of your project, so

```
$ git grep "hello" v2.5
```

searches for all occurrences of "hello" in v2.5.

If you leave out the commit name, **git grep** will search any of the files it manages in your current directory. So

```
$ git grep "hello"
```

is a quick way to search just the files that are tracked by Git.

You can also use **git show** to see any such file:

```
$ git show v2.5:Makefile
```

.getignore

Git sees every file in your working copy as one of three things:

- tracked - a file which has been previously staged or committed;
- untracked - a file which has not been staged or committed; or
- ignored - a file which Git has been explicitly told to ignore.

Ignored files are usually build artifacts and machine generated files that can be derived from your repository source or should otherwise not be committed. Some common examples are:

- dependency caches, such as the contents of `/node_modules` or `/packages`
- compiled code, such as `.o`, `.pyc`, and `.class` files
- build output directories, such as `/bin`, `/out`, or `/target`
- files generated at runtime, such as `.log`, `.lock`, or `.tmp`
- hidden system files, such as `.DS_Store` or `Thumbs.db`
- personal IDE config files, such as `.idea/workspace.xml`

Ignored files are tracked in a special file named `.gitignore` that is checked in at the root of your repository.

Git ignore patterns

Pattern	Example matches	Explanation*

`**/logs`	`logs/debug.log`
	`logs/monday/foo.bar`
	`build/logs/debug.log`
		You can prepend a pattern with a double asterisk to match directories anywhere in the repository.

`**/logs/debug.log`	`logs/debug.log`
	`build/logs/debug.log`
	but not
	`logs/build/debug.log`
		You can also use a double asterisk to match files based on their name and the name of their parent directory.

Pattern	Example matches	Explanation*

`*.log`	`debug.log`
	`foo.log`
	`.log`
	`logs/debug.log`
		An asterisk is a wildcard that matches zero or more characters.

`*.log`
`!important.log`
	`debug.log`
	`trace.log`
	but not
	`important.log`
	`logs/important.log`
		Prepending an exclamation mark to a pattern negates it. If a file matches a pattern, but *also* matches a negating pattern defined later in the file, it will not be ignored.

`*.log`
`!important/*.log`
`trace.*`
	`debug.log`
	`important/trace.log`
	but not
	`important/debug.log`
		Patterns defined after a negating pattern will re-ignore any previously negated files.

`/debug.log`	`debug.log`
	but not
	`logs/debug.log`
		Prepending a slash matches files only in the repository root.

`debug.log`	`debug.log` `logs/debug.log`	By default, patterns match files in any directory

`debug?.log`	`debug0.log`
	`debugg.log`
	but not
	`debug10.log`
		A question mark matches exactly one character.

`debug[0-9].log`	`debug0.log`
	`debug1.log`
	but not
	`debug10.log`
		Square brackets can also be used to match a single character from a specified range.

`debug[01].log`	`debug0.log`
	`debug1.log`
	but not
	`debug2.log`
		Square brackets match a single character form the specified set.

Pattern	Example matches	Explanation*
	`debug01.log`	

`logs`	`logs`
	`logs/debug.log`
	`logs/latest/foo.bar`
	`build/logs`
	`build/logs/debug.log`
		If you don't append a slash, the pattern will match both files and the contents of directories with that name. In the example matches on the left, both directories and files named *logs* are ignored

logs/	`logs/debug.log`
	`logs/latest/foo.bar`
	`build/logs/foo.bar`
	`build/logs/latest/debug.log`
		Appending a slash indicates the pattern is a directory. The entire contents of any directory in the repository matching that name – including all of its files and subdirectories – will be ignored

`logs/**/debug.log`	`logs/debug.log`
	`logs/monday/debug.log`
	`logs/monday/pm/debug.log`
		A double asterisk matches zero or more directories.

`logs/*day/debug.log`	`logs/monday/debug.log`
	`logs/tuesday/debug.log`
	but not
	`logs/latest/debug.log`
		Wildcards can be used in directory names as well.

POST-LAB

- How do I get a local copy of my repository?
- How do I record changes to my files using git?
- How do I view the status of the repository?
- How do I synchronize the history with the remote repo?