

Cloud Computing

Introduction to Git and Github

Suryo Adhi Wibowo, Ph.D.

Image Processing and Vision Laboratory
School of Electrical Engineering, Telkom University, Bandung, Indonesia
Email: suryoadhiwibowo@telkomuniversity.ac.id



Slide Sources: Google and Dicoding



IMAGE PROCESSING
AND VISION
LABORATORY



Before Version Control System

- Version Control Systems are crucial to maintaining a healthy codebase for all kinds of IT resources and to ensure a group of developers and IT specialist/system administrators can collaborate in the same coding projects together.
- The most initial form of version control is keeping historical copies.
- Before version control systems, track file changes using commands, such as **diff** and **patch**.

```
$ diff -u disk_usage_original.py disk_usage_fixed.py > disk_usage.diff
```

```
$ patch disk_usage.py < disk_usage.diff
```

Version Control System (VCS)

- A Version Control System keeps track changes applied into files, knows when the changes applies and who did them. It's able to revert a change if needed because of VCS stores the history of the changes.
- A **commit** is the edits of multiple files. It treats that collections of edits as a single change.
- A VCS can be invaluable not only to store codes, but also to manage system configurations, eg: Domain Name System (DNS) server records or DHCP configurations of your routers.

Git VCS

- Git is an Open Source VCS created in 2005 by Linus Torvalds, the developer who started the Linux kernel.
- Git can work as a standalone program as a server and as a client. Git clients can communicate with Git servers over network using HTTP, SSH, or Git special protocol.
- The official Git website use terms Source Control Management (SCM), another acronym similar to VCS.



* The official Git website <https://git-scm.com>

Git Configuration

- Start using git by setting some basic configurations.
- There are two ways to start working with a git repository:
 - Create one from scratch using the **git init** command.
 - Use the **git clone** command to make a copy of a repository that already exists. This will be explained later in remote repository topic.

Git Basic Configuration

```
$ git config --global user.email  
"your.email@gmail.com"
```

```
$ git config --global user.name "Your Name"
```

Initiate Git Project

```
$ mkdir project  
$ cd project  
$ git init  
Initialized empty Git repository in  
/home/bangkit/project/.git/
```

Git Directories

- Current working directory, also called the **working tree**, acting as **a sandbox**, that works with the current versions of the files.
- The **.git** directory acts as a database for all the changes tracked by Git. No direct interaction to these files, but through Git commands.

Working Tree and Git Directory

\$ ls -la

```
# stripped output (empty working tree)
drwxr-xr-x 7 user user 4096 Dec 1 10:11 .git
```

\$ ls -l .git/

```
# stripped output (initialized Git dir)
drwxr-xr-x 7 user user 4096 Dec 1 10:11 branches
drwxr-xr-x 7 user user 4096 Dec 1 10:11 config
drwxr-xr-x 7 user user 4096 Dec 1 10:11 description
drwxr-xr-x 7 user user 4096 Dec 1 10:11 HEAD
drwxr-xr-x 7 user user 4096 Dec 1 10:11 hooks
drwxr-xr-x 7 user user 4096 Dec 1 10:11 info
drwxr-xr-x 7 user user 4096 Dec 1 10:11 objects
drwxr-xr-x 7 user user 4096 Dec 1 10:11 refs
```

Using Git


- To start track files, add it to the project using the `git add` command. Use `git status` command to get some information about the current working tree and pending changes.
- Staging area, also known as the index is a file maintained by Git. Such staging area contains all information about what files and changes are heading to go into your next commit. Use `git commit` command to commit and put some messages.

First Steps with Git

```
$ cp ~/disk_usage.py .  
$ git add disk_usage.py  
$ git status
```

```
On branch master  
No commits yet  
Changes to be committed:  
  new file:   disk_usage.py
```

**means the changes in
the staging area**



```
$ git commit
```

```
# enter commit message on editor then save
```

```
$ git status
```

```
On branch master  
nothing to commit, working tree clean
```

Tracking Files Using Git

- Track changes applied into the files.
- With Git, files can be either tracked (part of snapshots) or untracked (not part of snapshots yet, Eg: new files).
- Each track file can be in one of the following three main states:
 - **Modified:** changed (added/modified/deleted contents of the file) but not committed yet
 - **Staged:** added to staging area, ready to be committed, will be the part of the next snapshot
 - **Committed:** the changes made to the files are safely stored in a snapshot in the Git directory

Git Commit Message

- Writing a clear informative commit message is important when using a VCS.
- A commit message is generally broken down into a few sections:
 - The first line is a short summary of the commit, usually kept to about 50 characters or less, followed by a blank line.
 - Then followed by a full description of the changes which details why they're necessary and anything that might uniquely be interesting about them or difficult to understand. Keep it under 72 characters per line in length. Lines started with pound symbol (#) are comments.
- Git command to display these commit messages is called **git log**.

Advanced **Git** Interaction

- On basic Git workflow the process is usually make changes, stage, and then commit.
- Another alternative is skipping the staging step and going directly to the commit. Use **git commit -a** command which will automatically stage every tracked-and-modified file. It is called "automatic" since it skips the git add command. Also note that tracked means new files are excluded.

Skipping the Staging Area

after made (small) changes/edit

```
$ git commit -a -m "minor revision"
```

```
[master fd0b38f] minor revision
```

```
1 file changed, 4 insertions(+), 1 deletion(-)
```

Advanced **Git** Interaction

- Git uses the **HEAD** alias to represent the currently checked-out snapshot of the project. Think about it as a bookmark.
 - In some cases, the current snapshot is the latest commit in the project.
 - In some other cases, (next on branches topic) head can be a commit in the different branch of the project.
 - In the use of git to go back in time, head represents old commit.

HEAD in git log

\$ git log

```
commit fd0b38f..f (HEAD -> master)
Author: Your Name <y..@gmail.com>
Date: Tue Dec 1 10:11:12 2020 +0800
```

```
    minor revision
```

```
commit ...
```

Advanced **Git** Interaction

- Use **git rm** command to remove files from repository, which will stop the file from being tracked and remove it from git directory. Then commit it with the same general workflow.
- To rename or move files use **git mv** command, continue with commit as general workflow.
- Create file **.gitignore** with the list of files or directory to be ignored from the git process, eg: output noise of git status. Also commit it as general workflow.

Undoing Things

- After editing files (before staging or commit), to change a file back to its earlier committed state by using **git checkout** command followed by the file name to revert.

Revert Files to Its Earlier Committed State

```
# after made changes/edit
```

```
$ git status
```

```
# stripped output
```

```
(use "git checkout ..." to discard changes)
```

```
modified: all_checks.py
```

```
$ git checkout all_checks.py
```

```
$ git status
```

```
On branch master
```

```
nothing to commit, working tree clean
```

Undoing Things

- If the changed files already added to the staging (by git add), unstage the changes by using the **git reset** command. Think reset as the counterpart to add.

Revert Files in Staging Area

```
$ git add *
```

```
# unintentionally add (too) many files
```

```
$ git status
```

```
# stripped output
```

```
(use "git reset HEAD ..." to unstage)
```

```
modified: all_checks.py
```

```
new file: debug.txt
```

```
$ git reset HEAD debug.txt
```

```
$ git status
```

```
# stripped output
```

```
Untracked files:
```

```
debug.txt
```

Undoing Things

- After committing files to git, you can revise it, (e.g.: to add files that belongs to the same change, or change the commit description) by using **git commit --amend**. With it, Git will take whatever is currently available in our staging area and run the git commit workflow to overwrite previous commit.
- Please avoid to use it on public commits because it can lead to confusing situations when working with other people.

Amending Commits

```
$ git add auto-update.py
$ git commit -m "add 2 new scripts"
[master 919d809] add 2 new scripts
1 file changed, ...
# there's only 1 file added to the commit

$ git add another-script.py
$ git commit --amend
# enter commit message on editor then save
[master f910c59] add two new scripts
2 file changed, ...
```

Undoing Things

- Sometime to swiftly fix a problem by rollback commits, you can use **git revert** command, among other ways.
- Git revert doesn't just mean undo. It creates a commit that contains the inverse of all the changes made in the previous commit to cancel them. It will create a new commit that is the opposite of everything in the given commit.

Rollbacks with git revert

```
$ git commit -a -m "new function call"
```

```
# add (untested) new code to the script  
[master 5aab0fd] new function call  
1 file changed, ...
```

```
$ git revert HEAD
```

```
# enter rollback message on editor then save  
[master 0832461] Revert "new code to the script"  
1 file changed, ...
```


Undoing Things

- To revert other commits farther back in time, target a specific commit by using its **commit ID**. It is a 40 chars long string, called a hash, calculated using SHA1 algorithm. Git will be smart enough to guess which commit ID if provided with around 4-6 characters, as long as it is matched.

Identifying a Commit

```
$ git log -2
```

```
# show last 2 commits
```

```
$ git show 30e70
```

```
# show specific commit with id match 30e70
```

```
$ git revert 30e70
```

```
# revert back to commit 30e70
```

```
# enter rollback message on editor then save
```

```
[master 7d1de19] Revert "New name for d...py"
```

```
1 file changed, ...
```

Branching and Merging

- In Git, a **branch** at the most basic level is just a pointer to a particular commit. It represents an independent line of development in a project.
- The default branch that Git creates for you when a new repository is initialized is called master.
- The master branch is commonly used to represent the known good state of a project. When developing a feature or trying something new in a project, create a **separate branch** to do the work without worrying about messing up the current working state. Then lastly, merge back into the master branch, if needed.

Branching and Merging

- Running command **git branch** will show a list of all the branches in your repository. The git branch command also can create, delete, and manipulate branches.

asterisk indicates the
current branch



Branching

```
$ git branch
* master

$ git branch new-feature
$ git branch
* master
  new-feature

$ git checkout new-feature
$ git branch
  master
* new-feature
```

Branching and Merging

- Add new file and commit to new branch.
- By switching to master branch, the result is that git changes where head is pointing.
- The commit from other branches doesn't show up at all. Git also change files in the working directory or working tree to whatever snapshot head is currently pointing at.

Working with Branches

```
$ cp ~/free_memory.py .  
# copy script from home directory  
$ git add free_memory.py  
$ git commit -m "add free_memory.py"  
[new-feature 4361880] add free_memory.py
```

```
...  
$ ls  
disk_usage.py  free_memory.py
```

```
$ git checkout master  
Switched to branch "master"
```

```
$ ls  
disk_usage.py
```



**free_memory.py not found
in branch master**

Branching and Merging

- Merging is the term employed by Git to combine branched data and history together. Use **git merge** command. It takes the independent snapshots and history of one Git branch and tangle them into another.
- Two different algorithms to perform a merge: **fast-forward** and **three-way** merge.
- Delete branch after merge with **git branch -d** command.

Merging Branches

```
$ git merge new-feature
```

```
Updating 7d1de19..3261880
```

```
Fast-forward
```

```
free_memory.py | ...
```

```
1 file changed, ...
```

```
$ git branch -d new-feature
```

```
Deleted branch new-feature ...
```

Remote Repository with GitHub

Introduction to GitHub

- Git is a distributed version control system, which means that each developer has a copy of the whole repository on their local machine.
- GitHub (<https://github.com>) is a web-based Git repository hosting service, similar like BitBucket and GitLab.
- The course using GitHub for the examples. GitHub provides free access to a Git server for public and private repositories.
- A word of caution on how to manage these repos. For real configuration and development work, use a secure and private Git server and limit the people authorized to work on it.

Introduction to GitHub

- After creating an account and signing-in at GitHub, please create new repository with GitHub web interface. Guided repository name on the course is **health-checks**. Complete it with README.md.
- Use **git clone** command to create a local copy of the repo.
- Send the changes to remote repository by using **git push** command.

Basic Interaction with GitHub

```
$ git clone https://github.com/myaccount/health-checks.git
```

```
Cloning into 'health-checks'
```

```
Username for 'https://github.com': myaccount
```

```
Password for 'https://myaccount@github.com':
```

```
remote: ...
```

```
Unpacking objects: 100%, done.
```

```
$ cd health-checks/
```

```
# edit README with markdown format
```

```
$ git commit -a -m "update README"
```

```
[master 807cb50] update README
```

```
1 file changed, ...
```

```
$ git push
```

```
Username for 'https://github.com': myaccount
```

```
Password for 'https://myaccount@github.com':
```

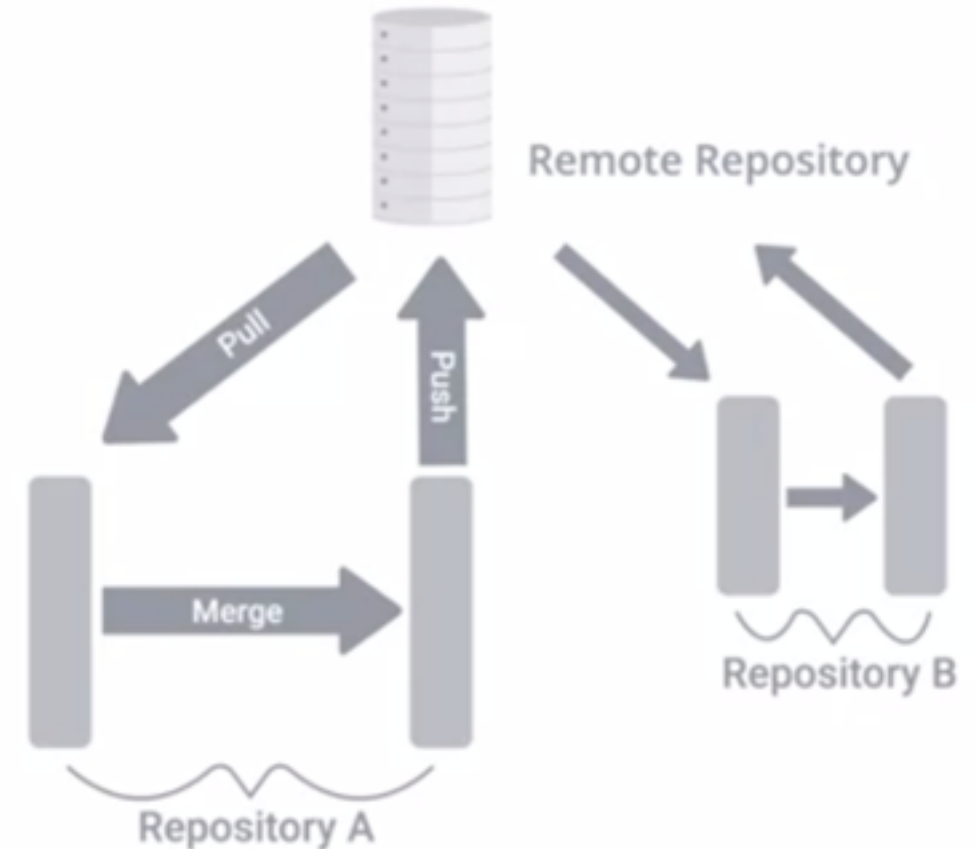
```
...
```

```
Writing Objects: 100%, done....
```

```
3d9f86c..807cb50 master -> master
```


Using a Remote Repository

- When working with remotes, the workflow for making changes has some **extra steps**. The workflow still includes modifying stage and committing of local changes. The additional step happened after commit, consists of **fetch** any new changes from the remote repo, manually **merge** if necessary, and only then you may push changes to the remote repo.



Using a Remote Repository

- With `git clone`, Git sets up remote repository with default **origin** name. Check the configuration by running **git remote -v** in the repo directory. Two remote URLs consist of one to fetch data & the other one is to push data to remote.
- `git status` up to date inform that **local master branch** has the same commits with the **master branch in remote** called origin.

Working with Remotes

```
$ cd health-checks/
```

```
$ git remote -v
```

```
origin https://github.com/myaccount/health-checks.git (fetch)  
origin https://github.com/myaccount/health-check.git (push)
```

```
$ git status
```

```
On branch master  
Your branch is up to date with 'origin/master'.  
nothing to commit, working tree clean
```

Using a Remote Repository

- Git doesn't keep remote and local branches in sync automatically. Instead, it waits until we execute **git fetch** command.
- To integrate the branches into the master branch, perform **git merge**, which merge "origin/master" branch into local master branch.

Fetching New Changes

\$ git fetch

```
remote: Enumerating objects: ...  
from https://github.com/myaccount/health-checks  
807cb50..b62dc2e master -> origin/master
```

\$ git status

```
On branch master  
Your branch is behind 'origin/master' by 1 commit, and can  
be fast-forwarded.  
(use "git pull" to update your local branch)  
...
```

\$ git merge origin/master

```
Updating 807cb50..b62dc2e  
Fast-forward  
...
```

Using a Remote Repository

- Other than the basic workflow with remotes: **fetch** changes manually, **merge** if necessary, and only then you may push any changes. Git has the **git pull** command to fetching and merging at once.
- Running **git pull** will fetch the remote copy of the current branch and automatically try to merge it into the current local branch.

Updating the Local Repository

```
$ git pull
```

```
remote: Enumerating objects: ...
```


```
From https://github.com/myaccount/health-checks
```

```
b62dc2e..922d659 master -> origin/master
```

```
Updating b62dc2e..922d659
```

```
Fast-forward
```

```
...
```

 **fetch** **merge**

Solving Conflicts

- When running **git push** command Git rejects the change. Why? The remote repo contains changes that not available in local branch so Git can't fast forward. Run **git pull**.
- Git will try to exert all possible automatic merges and only leave manual conflicts for us to resolve when the automatic merge fails.

Pull-Merge-Push Workflow

```
<<<<<< HEAD
    if percent_free < min_percent or gigabytes_free <
min_gb:
=====
    if gigabytes_free < min_absolute or percent_free <
min_percent:
>>>>>> a2dc11
```

Solving Conflicts

- It's a good idea to always synchronize the branches before starting any work. It minimizes the chances of conflict or the need for rebasing.
- Another common practice is trying to avoid having very large changes that can modify a lot of different things. Try to make changes as small as possible as long as they're self-contained.
- When working on a big changes, it makes sense to have a separate feature branch. Regularly merge changes made on the master branch back onto the feature branch.
- To maintain more than one version of a project, it's a common practice to have the latest version of the project in the master branch and a stable version of the project on a separate branch.

Pull Requests

- When collaborating on projects that hosted on GitHub, the typical workflows are: create a fork of the repo and then work on that local fork. To merge the changes back into the main repo, create a **pull request**.
- Forking is a way to create a copy of the given repository so that it belongs to our user.
- **Pull Request** is a commit or series of commits being sent to the owner of the repo so they can incorporate it into their tree. The owner of the repo will review the changes. If accepted then they will merge them.

Pull Requests

- Create simple Pull Request directly through the GitHub interface:
 - Create fork of the repo and then work on that local fork. Fix the codes or add new features.
 - Make a change proposal. Scroll down and fill the description of the change. Clicking on Proposed file change button will create a commit in our (forked) repo.
 - After creating a commit, there will be some information about the change, like repositories and branches involved in the pull request creation. Github automatically creates a **branch** called **patch-1**.
 - Create pull request by pressing the button, which opens a text box to enter comments about the change, also allow edit by maintainer.

Pull Requests

- In other cases, to work with the forked repo on local computer, process with **git clone** command with the URL of the forked repo.
- Start working with local copy of the project by creating a new branch (Eg: add-readme) using command **git checkout -b add-readme**
- Save the working file(s) and add to staging. Eg: **git add README.md**
- Commit the changes with command: **git commit -m 'add README file'**
- Push the branch to remote forked repo: **git push -u origin add-readme**
- Check the new branch on GitHub forked repo and continue the pull request process by clicking pull request link from the GitHub interface like the previous steps (on previous slide).

Code Reviews

- Doing a code review means going through someone else's code, documentation or configuration and checking that it all makes sense and follows the expected patterns. The goal is to improve the project by making sure that changes are of high quality.
- On platforms like GitHub, it's common for projects to only require reviews for people that don't have commit access, while the project maintainers can commit directly.
- With reviewing tools, reviewer add comments to the changes, explaining what needs to be fixed and how. After addressing a comment, reply with another comment to ask another reviews or mark it as resolved. Once all the comments have been resolved and the reviewer is satisfied, the change is approved and incorporated.

Managing Projects

- Work/project documentation is super important. Documenting **what you do** and **why you do** is essential otherwise you'll spend much time answering everybody else's questions.
- As project maintainer, let others know how to interact with the project by creating readme.md. It is also important to reply pull requests. Especially for open source project, make sure to understand the changes before accepting it and be careful with which patches to accept or reject.
- For many years, most projects used mailing list and IRC channels for communication. Recently, new forms of communicating have gained popularity like Slack channels or Telegram groups.

Managing Projects

- Tool like issue tracker or bug tracker can help to better coordinate the work. An issue tracker tells the tasks that need to be done, the state of the issue and who's working on the issue. It includes comments mechanism related to the issue.
- Issue trackers aren't just useful for people actively working on projects, but also for users to report bugs, even if they don't know how to solve the problem.
- GitHub platform has a built in issue tracker. Each issue or pull request has a unique number associated with it. It will be referenced automatically if mentioned on comments by using the hashtag number.

***THANK
YOU***