Terralogic
Expect Excellence

Git command

July 2016 - Thanh Nguyen

Embedded Systems
Mobile Strategy
Applications
Cloud Integration
Testing & Automation
Product Engineering

Outline Part 1

- ➢ Git introduction
- ➢ Setting up a repository
- ➢ Git workflow
- ➢ View file changes
- ➢ Stashing changes

Outline Part 2

- ➢ Saving changes
- ➢ Syncing
- ➢ View commit history
- ➢ Undo changes
- ➢ Rewriting history
- ➢ Git cherry-pick
- ➢ Branches

Terralogic Confidential

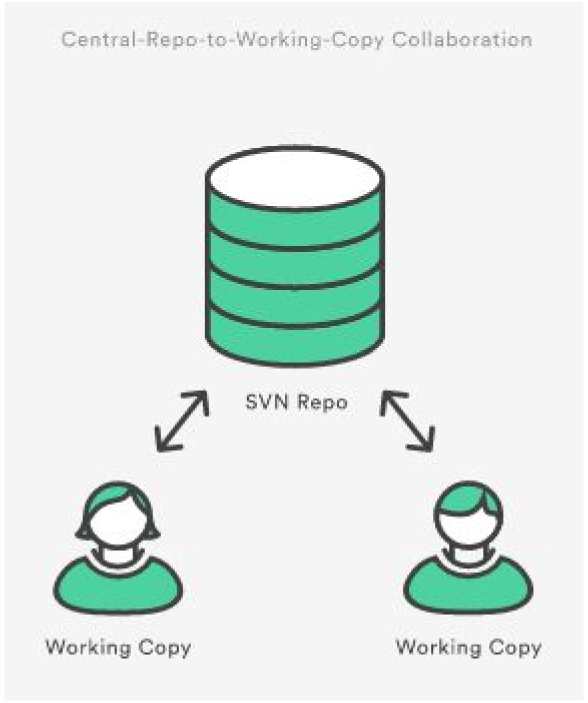
Git introduction

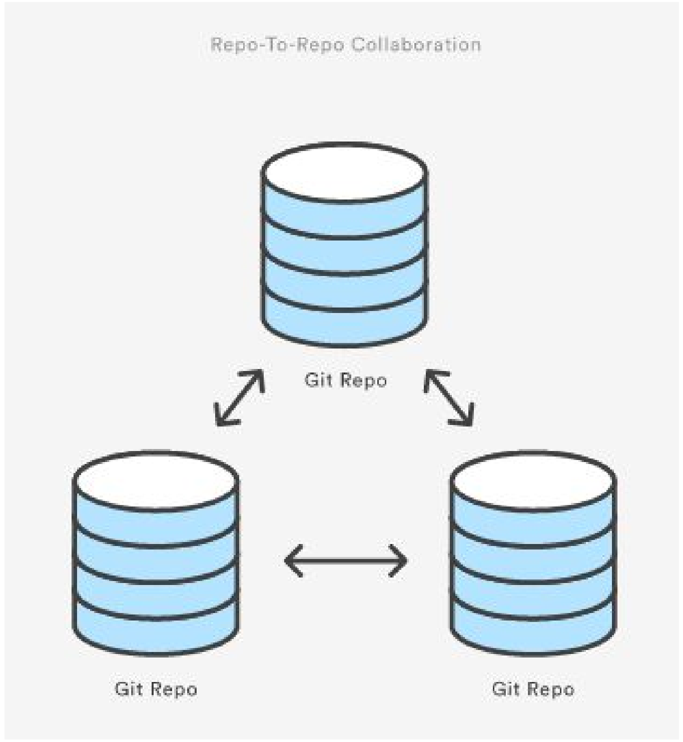
- What is git?
  - Git is a version control system that is used for software development.

- Why we use git?
  - Version control software allows you to have “versions” of a project, which show the changes that were made to the code over time, and allows you to backtrack if necessary and undo those changes. This ability alone – of being able to compare two versions or reverse changes, makes it fairly invaluable when working on larger projects.

- Compare with SVN?
  - Git: create new version-controlled projects easy with git init. SVN: need to do many steps: create a repository, import files, and check out a working copy.
  - Git: Interact with other repositories. SVN: just interact with central repo.

Git introduction





Setting up a repository

- git init
- git clone
- git config

Setting up a repository git init

- The git init command creates a new git repository
- For most projects, git init only needs to be executed once to create a central repository, developers typically don‘t use git init to create their local repositories. Instead, they’ll usually use git clone to copy an existing repository onto their local machine

Usage:

```
$git init
```

Setting up a repository git clone

- The git clone command copies an existing Git repository.
- As a convenience, cloning automatically creates a remote connection called origin pointing back to the original repository. This makes it very easy to interact with a central repository.

Usage:

```
$git clone <repo>
```

Clone the repository located at <repo> onto the local machine.

```
$git clone <repo> <directory>
```

Clone the repository located at <repo> into the folder called <directory> on the local machine.

Setting up a repository git config

- The git config command lets you configure your Git installation from the command line.
- This command can define everything from user info to preferences to the behavior of a repository. Several common configuration options are listed below.

Usage

$git config --list Review the settings

$git config --global user.name <name> set name

$git config --global user.email <email> set email

$git config --global alias.<alias-name> <git-command> Create a shortcut for a Git command.

$git config --global credential.helper 'cache --timeout=360000’ configure the Credential helper and set appropriate timeout value

Setting up a repository git config (cont.)

Git stores configuration options in three separate files, which lets you scope options to individual repositories, users, or the entire system:

- <repo>/.git/config – Repository-specific settings
- ~/.gitconfig – User-specific settings. This is where options set with the --global flag are stored.
- $(prefix)/etc/gitconfig – System-wide settings.

When options in these files conflict, local settings override user settings, which override system-wide

Avi git workflow

1. Git fetch
2. Git checkout branch
3. Git pull --rebase
4. Edit file
5. Commit
6. Git push
7. Create and merge Pull request

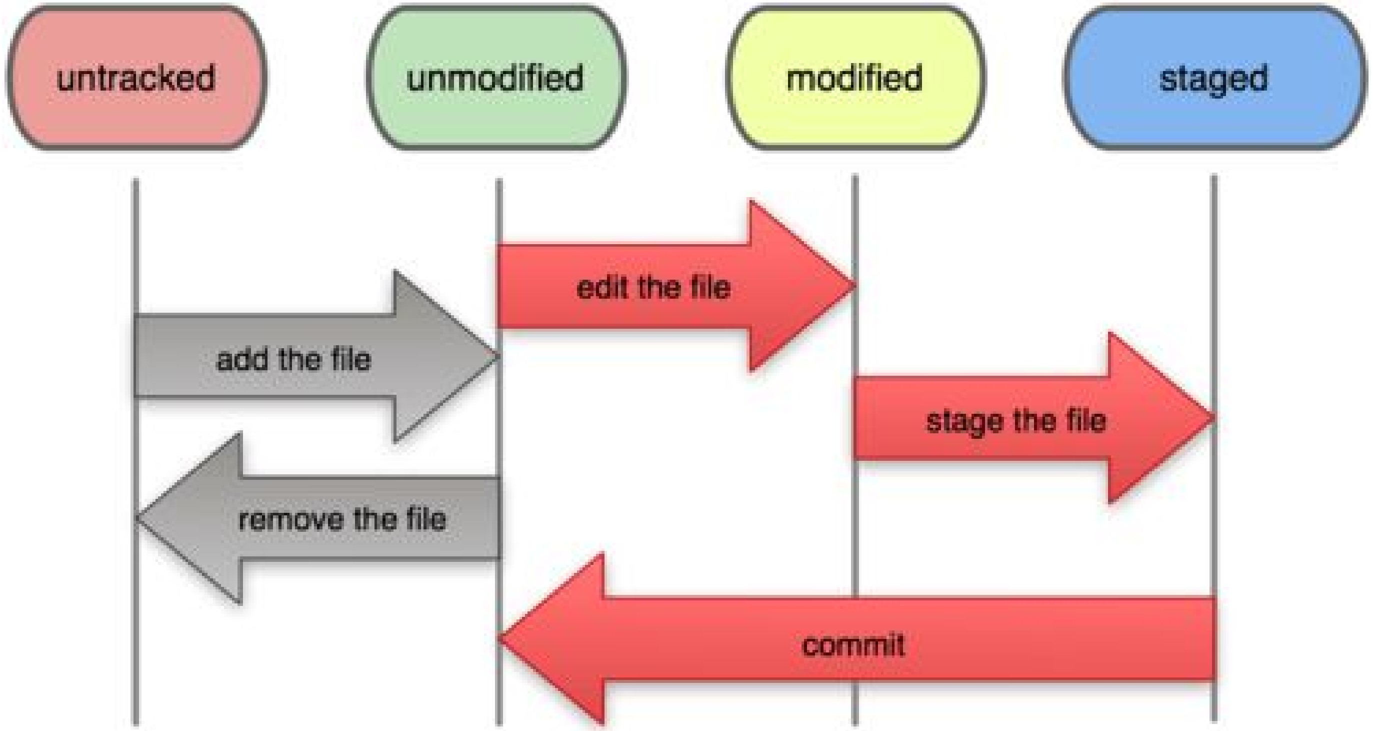
View changes

- Git File Status
- Ignoring files
- git status
- git diff

Viewing changes

Git File States

- **Untracked** vs **Tracked**
- **Unmodified** vs **Modified**
- **Staged** vs **Unstaged**



Viewing changes Ignoring files

Git lets you completely ignore files by placing paths in a special file called .gitignore. Any files that you'd like to ignore should be included on a separate line, and the * symbol can be used as a wildcard.

For example, adding the following to a .gitignore file in your project root will prevent compiled Python modules from appearing in git status: *.pyc

Usage

```
$vim /home/aviuser/avi-dev/.gitignore
```
View, edit gitignore file

Viewing changes git status

The git status command displays the state of the working directory and the staging area. It lets you see which changes have been staged, which haven’t, and which files aren’t being tracked by Git.

Usage

```
$git status
```

List which files are staged, unstaged, and untracked.
unmodified : not shown in this command

Viewing changes git diff

The git diff command shows the changes since last commit

Usage

```
$git diff
```

Show unstaged changes made since your last commit.

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

Show changes staged for commit

```
$git diff HEAD
```

Show changes (staged and unstaged) in working directory since last commit.

Stashing changes git stash

- Use “git stash” when you want to record the current state of the working directory and the index, but want to go back to a clean working directory.
- The command saves your local modifications away and reverts the working directory to match the HEAD commit.

Stashing changes git stash usage

$git stash

$git stash list - List all current stashes.

$git stash apply [id] - Restore the changes recorded in the stash on top of the current working tree state.

$git stash pop [id] - Remove a single stashed state from the stash list and apply on top of the current working tree state.

$git stash drop [id] - Remove a single stashed state from the stash list.

$git stash clear - Remove all the stashed states.

$git stash branch new-branch [id] - Creates and checks out a new branch named new-branch starting from the commit at which the stash was originally created, applies the changes recorded in stash to the new working tree and index, then drops the stash if that completes successfully.

Outline Part 2

- Saving changes
- Syncing
- View commit history
- Undo changes
- Rewriting history
- Branches
- Merging
- cherry-pick
- Others

Saving changes

- git add
- git commit

Saving changes git add

The git add command adds a change in the working directory to the staging area. It tells Git that you want to include updates to a particular file in the next commit. However, git add doesn't really affect the repository in any significant way—changes are not actually recorded until you run git commit.

Usage

```
$git add <file>
```

Stage all changes in <file> for the next commit.

```
$git add <directory>
```

Stage all changes in <directory> for the next commit.

```
$git add --all
```

Stage all changes in your working copy

Saving changes git commit

- The "git commit" command commits the staged snapshot to the project history.
- Along with git add, this is one of the most important Git commands.
- After a git commit, git status should show ‘nothing to commit’

Usage

```
$git commit file1 file2 | -a | -m "<message>"
```

- file1 file2: Commit the staged snapshot. This will launch a text editor prompting you for a commit message. After you’ve entered a message, save the file and close the editor to create the actual commit.
- -m "<message>" : Commit the staged snapshot, but instead of launching a text editor, use <message> as the commit message.
- -a :Commit a snapshot of all changes in the working directory. (exclude untracked files)

Syncing

- git remote
- git fetch
- git pull
- git push

Syncing git remote

The git remote command lets you create, view, and delete connections to other repositories. When you clone a repository with git clone, it automatically creates a remote connection called origin pointing back to the cloned repository.

Usage

- $git remote add <name> <url> - Create a new connection to a remote repository. After adding a remote, you'll be able to use <name> as a convenient shortcut for <url> in other Git commands.
- $git remote - List the remote connections you have to other repositories.
- $git remote -v - Same as the above command, but include the URL of each connection.
- $git remote rm <name> - Remove the connection to the remote repository called <name>.
- $git remote rename <old-name> <new-name> - Rename a remote connection from <old-name> to <new-name>.

Syncing git fetch

The git fetch command imports commits from a remote repository into your local repo

Usage

```
$git fetch <remote>
```

Fetch all of the branches from the repository. This also downloads all of the required commits and files from the other repository.

```
$git fetch <remote> <branch>
```

Same as the above command, but only fetch the specified branch.

Syncing git pull

- Merge upstream changes into your local repository

Usage

```
$git pull <remote>
```

Fetch the specified remote's copy of the current branch and immediately merge it into the local copy.

Syncing git push

- Pushing is how you transfer commits from your local repository to a remote repo.

Usage

```
$git push <remote> <branch>
```

Push the specified branch to <remote>

```
$git push <remote> --all
```

Push all of your local branches to the specified remote.

View history

- git log
- git blame

Terralogic Confidential

Viewing history git log

- The git log command displays committed snapshots. It lets you list the project history, filter it, and search for specific changes.

Usage:

```
$git log - Display the entire commit history using the default formatting.
$git log -n <limit> - Limit the number of commits by <limit>. For example, git log -n 3 will display only 3 commits.
$git log --oneline - prints each commit on a single line, which is useful if you're looking at a lot of commits
$git log -p - Shows diff introduced in each commit
$git log --author="<pattern>" - Search for commits by a particular author.
$git log --grep="<pattern>" - Search for commits with a commit message that matches <pattern>
$git log <file> - Only display commits that include the specified file.
$git log --after="MMM DD YYYY" Show commits that occur after a certain date, e.g. "Jun 20 2008".
$git log --before="MMM DD YYYY" Show commits that occur before a certain date.
```

Viewing history git blame

Git blame command shows who authored each line in file.

Usage:

```
$git blame <file>
```

Undo changes

- git checkout
- git revert
- git reset
- git clean

Undo changes git checkout

The git checkout command serves three distinct functions: checking out files, checking out commits, and checking out branches.
Usage:

```
$git checkout <file>
```

Drop all changes on <file>

```
$git checkout <branch>
```

Return to the selected branch.

```
$git checkout <commit> <file>
```

Check out a previous version of a file. This turns the <file> that resides in the working directory into an exact copy of the one from <commit> and adds it to the staging area.

Undo changes git revert

The git revert command undoes a committed snapshot. But, instead of removing the commit from the project history, it will generate a new commit that undoes all of the changes.

Usage:

```
$git revert <commit>
```

Generate a new commit that undoes all of the changes introduced in <commit>, then apply it to the current branch.

Undo changes git reset

When you undo with git reset, there is no way to retrieve the original copy—it is a permanent undo.

Usage

$git reset <file> - Remove the specified file from the staging area, but leave the working directory unchanged.

$git reset - Reset the staging area to match the most recent commit, but leave the working directory unchanged.

$git reset --hard - Reset the staging area and the working directory to match the most recent commit. The --hard flag tells Git to overwrite all changes in the working directory. This obliterates all uncommitted changes

$git reset <commit> Move the current branch backward to <commit>, reset the staging area to match, but leave the working directory alone.

$git reset --hard <commit>

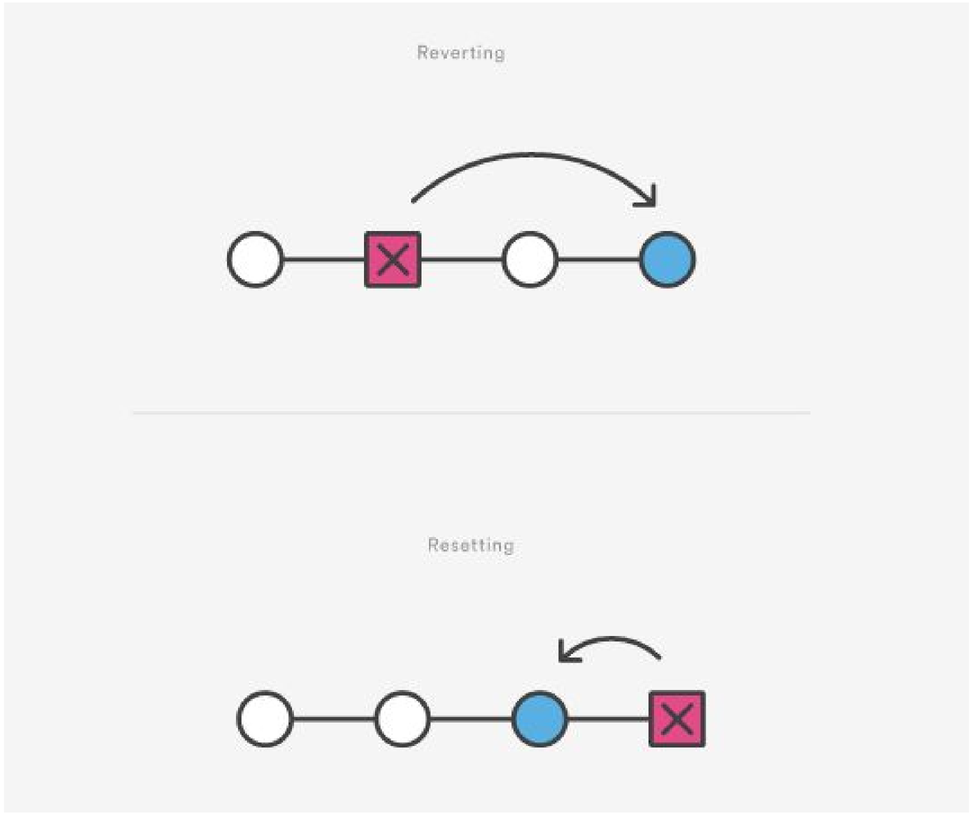
Move the current branch backward to <commit> and reset both the staging area and the working directory to match. This obliterates not only the uncommitted changes, but all commits after <commit>, as well.

$git reset HEAD^ Undo the Previous Commit

Undo changes

git revert vs git reset

- git revert is designed to safely undo a public commit
- git reset is designed to undo local changes.



Undo changes git clean

The git clean command removes untracked files from your working directory.

Usage:

```
$git clean -h
-n, --dry-run   Dry run - Show which files are going to be removed without actually doing it.
-f, --force      force - Remove all untracked files
-d               Remove whole directories
-x               Removed ignored files, too
-X               Remove only ignored files
-q, --quiet      Do not print names of files removed
```

Rewriting history

- git commit --amend
- git rebase -i

Rewriting history git commit --amend

This command will let you combine the staged changes with the previous commit or let you edit the previous commit's message.

Usage:

```
git commit --amend
```

Rewriting history git rebase -i

`git rebase -i` will let user edit the list of commits to rebase

Usage:

```
git rebase -i <commid_id>
```

Git cherry-pick

As always please cherry pick commits from eng to the branch.

For example: To cherry-pick to 15.1-beta-1

1) While on Eng branch, push the commit as usual to eng

```
git push origin eng
```

2) Get the SHA-ID of your commit-id
3) git fetch (This fetches all branches from origin)
4) git checkout 15.1-beta-1 (Run "git status" and verify you are on 15.1-beta-1)
5) git pull --rebase (Gets the latest 15.1-beta-1 branch)
5) git cherry-pick <SHA-ID-OF-ENG-COMMIT>
6) git log (Verify your commit is on top. Note, the SHA-ID of your commit on 15.1-beta-1 will be different)
7) git push origin 15.1-beta-1

Branches

- List the branches
- Create new branches
- Delete branches

Branches List the branches

git branch
shows all local branches and the current branch has * before its name

git branch -r
shows remote branches

git branch -a
shows all remote and local branches

Branches

Creating branches

```
$git branch new-branch
```

Create a new branch named new-branch, based on current branch.

```
$git branch --track new-branch remote/remote-branch
```

Create a new tracking branch named new-branch, referencing, and pushing/pulling from, the branch named remote-branch on remote repository named remote.

```
$git checkout -b <new-branch>
```

Create and switch <new-branch>.

The -b option tells Git to run git branch <new-branch> before running git checkout <new-branch>

Branches Deleting branches

```
git branch -d branch
```
Delete the local branch named branch (fails if branch is not reachable from the current branch).

```
git branch -D branch
```
Force delete of the local branch named branch (works even if branch is not reachable from the current branch).

```
git branch -d -r remote/branch
```
Delete a “local remote” branch, i.e. a local tracking branch.

```
git push remote :heads/branch
```
Delete a branch named branch from a remote repository.