Introduction to Version Control with GIT

Anja Gerbes

Goethe University in Frankfurt/Main
Center for Scientific Computing

March 8, 2019
Overview

What is version control?

Key Concepts

First Steps in GIT

GIT Workflow

Branching

Extras
What is version control?

- A system that keeps records of your changes
- Allows for collaborative development
- Able to see **who** made changes and **when**
- Can revert any changes back to a previous state
Snapshots

- The way git keeps track of your code history
- Essentially records what all your files look like at a given point in time
- You decide **when** to take a snapshot and of **what** files
- Have the ability to go back to visit any snapshot
Commits

- The act of creating a snapshot
- Essentially, a project is made up of a bunch of commits
- Commits contain three pieces of information:
  1. Information about how the files changed from previously
  2. A reference to the commit that came before it (called the parent commit)
  3. A hash code name (Will look like: edfec504eb864dc557f3f5b9d3d301617036d15f3a)

Commits as small as possible or as big as necessary
Repositories

- A collection of all the files and the history of those files
  - consists of all your commits
  - place where all your hard work is stored
install git

Setup

Linux (Ubuntu)  sudo apt-get install git
Linux (Fedora)  sudo yum install git

Mac  https://git-scm.com/download/mac

Windows  https://gitforwindows.org/

Get Help

git --help
man git
$ git config --global user.name "Anja Gerbes"
$ git config --global user.email "gerbes@csc.uni-frankfurt.de"

# ~/.gitconfig
[user]
    name = Anja Gerbes
    email = gerbes@csc.uni-frankfurt.de

$ git config color.ui true
$ git config format.prettify oneline
create repositories

$ mkdir myrepo
$ cd myrepo

directory will be become the working tree for the repository

$ git init

Initialized empty Git repository in ../myrepo/.git

repository is created without a working tree and it is used as a remote repository that is sharing a repository among teammates

$ git init --bare

permanently fixing permissions on a shared git repository

$ git init --bare --shared=group

For shared repositories pay attention to the file permissions. It is recommended to prohibit changing the history.
create a file

$ echo "Hello World" > doc.md
$ git status

# On branch master
#
# Initial commit
#
# Changes to be committed:
# (use "git rm --cached <file>..." to unstage)
#
# new file: doc.md
#
# 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: doc.md
`git add`

add files to the staging area

```bash
$ git add doc.md
$ git add *.md
```

create as small as possible, logically separated commits

```bash
$ git add --patch or git add -p
$ git add --interactive or git add -i
```

stage all changes (incl. deleted files)

```bash
$ git add --all
```
git commit
put files from staging area into repository and make a snapshot

a commit should contain a single, self contained idea

$ git commit -m "My first commit"

[master 8345967] changed
1 files changed, 1 insertions(+), 1 deletions (-)

automatically stage files that have been modified

$ git commit -a -m "My first commit"

[master 8345967] changed
1 files changed, 1 insertions(+), 1 deletions (-)

edit last commit-message

$ git commit --amend
$ git clone git@host:/path/to/repository/testing.git
mydir

Cloning into 'mydir'
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (3/3), done.
$ git push origin master

Counting objects: 5, done.
Writing objects: 100% (3/3), 272 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@host:/path/to/repository/testing.git
    edfec50..2fc284e  master --> master

$ git push [remote-name] [remote-branch-name]
$ git pull

remote: Counting objects: 7, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 2), reused 0 (delta 0)
Updating 361303d..f2cd831
Fast forward
doc.md | 1 +
1 files changed, 1 insertions (+), 0 deletions (-)
git remote
manage set of tracked repositories

$ git remote add origin <server>
$ git remote add origin git@host:/path/to/repository/testing.git
$ git push origin master

Counting objects: 3, done.
Writing objects: 100% (3/3), 231 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@host:/path/to/repository/testing.git
  [new branch]  master  -->  master
Branching

create branch my-feature

$ git branch my-feature

rename branch my-oldfeature to my-newfeature

$ git branch -m my-oldfeature my-newfeature

delete branch my-feature

$ git branch -d my-feature

create & switch to new branch

$ git checkout -b my-feature

$ git checkout master

Switched to branch 'my-feature'

$ echo "My Branch is different" > doc.md
$ git commit -a -m "changed content to my branch"
List Branches

list all local branches

$ git branch

master
* my-feature

list all branches (local + remote)

$ git branch -a

master
* my-feature
remotes/origin/HEAD -> origin/master
remotes/origin/master
Developer creates an empty develop branch locally & pushes in to server:

```
git branch develop
git push -u origin develop
```

Other developers should now clone central repository & create a tracking branch for develop branch:

```
git clone ssh://user@host/path/to/repo.git
git checkout -b develop origin/develop
```

Now everyone has a local copy of historical branches set up.
Branch Workflow

eg. 2 developers: Both require separate branches that are based on develop branch instead of master branch

    git checkout -b some-feature develop

Both of them add commits to the feature branches according to the usual procedure: Edit, stage, commit.

    git status
    git add
    git commit
Branch Workflow

- after several commits, a developer feels that feature is ready
- he can merge it into his local develop branch & push it into central repository as follows:

  ```
git pull origin develop
git checkout develop
git merge some-feature
git push
git branch -d some-feature
  ```

The first command ensures that the develop branch is up to date before attempting to merge the feature into it.
Feature Branch Workflow

- each new feature should be developed in its own branch
- branch can be pushed into central repository for backup and collaboration purposes
- develop branch is used as a source and branches are created here not on the master branch
- once new features are completed they are merged back into the develop branch
- new features never interact directly with master branch
If develop branch contains enough features for a release, develop branch suspends a release branch

this starts next release cycle; new features should not be added, only bugfixes & similar release-oriented changes

release is ready for delivery, it is merged into master branch & tagged with a version number
using a dedicated release preparation branch allows one team to fine-tune current release while other team continues to work on features for next release

certain stages of development can be defined very well; e.g. “Prepare this week like version 4.0” & see it in the structure of the repository
Branch → Remote

store branch in remote

$ git push origin <branch>
$ git push origin my-feature

Counting objects: 6, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (6/6), 482 bytes, done.
Total 6 (delta 0), reused 0 (delta 0)
To git@host:/path/to/repository/testing.git
  * [new branch]   my-feature -> my-feature
Update & Merge a Branch

$ git pull
$ git merge <branch>
$ git add <dateiname>
$ git diff <branch> <branch>
Merging Branches

```bash
# the working tree is in the same state as git HEAD
$ git checkout master

Switched to branch 'master'

$ git merge my-feature

Updating edfec50..2bc1785
Fast-forward
doc.md | 2+-
1 files changed, 1 insertions (+), 0 deletions (-)

changes can be enacted at the remote server by typing

git push origin master
```
Dealing with Merge Conflicts

- Handling a git pull request with merge conflict
- When working with git, the relatively complex tasks are issuing a pull request & then merging with conflicts

Step 1 Verify your local repository
- `git checkout pinc`
- `git pull origin pinc`
Ensure that the files on local repository are in-sync with your remote git repository

Step 2 Switch to branch
- `git checkout feature-1`
- `git pull origin feature-1`
Switch to the branch that you want to merge
Ensure that you pull the latest files from your remote server

Step 3 Try to merge
- `git merge pinc`

Step 4 Resolve the merge conflict
If you get the message, that there is a merge conflict & it cannot automatically merge the change, you can resolve the conflict manually. Open the file & you'll need to fix this.

Step 5 Check in changes
Commit the fixes to the branch
- `git add file.py`
- `git commit -m "some comment"`
- `git push origin feature-1`

Step 6 Merge the branch
Restore

restore deleted files

```
$ git checkout -- <filename>
$ git checkout -- doc.md

$ git fetch origin
$ git reset --hard origin/master
```
git stash

$ git stash <command>

**Syntax:**

- `save "Your message"` stashes with a message
- `save -u | save --include-untracked` stashes untracked files
- `list` view list of stashes you made at any time
- `apply [stash@{stash\_id}]` apply specific stash
- `pop [stash@{stash\_id}]` deletes stash from stack after it is applied
- `show` shows summary of stash diffs
- `show -p` shows full diffs
- `show stash@{stash\_id}` use stash_id to get diff summary
- `branch <name>` creates new branch
- `branch <name> stash@{stash\_id}` then deletes latest stash
- useful, conflicts after you've applied stash to latest version of your branch
- `clear` deletes all stashes made in repository impossible to revert
- `drop stash@{stash\_id}` deletes latest stashes from stack
- use it with caution, difficult to revert

**Example commands:**

- `git stash save "Your message"`
- `git stash list`
- `git stash apply stash@{stash\_id}`
- `git stash pop stash@{stash\_id}`
- `git stash show`
- `git stash show -p`
- `git stash branch <name>`
$ git log

commit edfec504eb864dc557f3f5b9d3d301617036d15f3a
Author: Anja Gerbes <gerbes@csc.uni-frankfurt.de>
Date: Thu Oct 18 14:00:20 2018 +0200

My First Commit

search in history

$ git log --pretty=short --since=2weeks
$ git log --pretty=short --author="Anja Gerbes"
   --grep="comment"
Regular expression for git repository

```plaintext
git@github.com:someone/someproject.git

[user]@[server]:[project].git
```

Git accepts a large range of repository URL expressions:

```plaintext
ssh://user@host.xz:port/path/to/repo.git/
file:///path/to/repo.git/
```

1. `'(\w+://)(.+@)([\w\d.]+)(:[\d]+){0,1}/*(.*)'`
2. `'file://(.*)'`
3. `'(.+@)([\w\d.]+):(.*).'`
4. `'((git|ssh|http(s)?)|(git@[\w\d.]+))(://)?([\w\@:\/-~]+)(\..git)/(//)?'`

look at https://www.debuggex.com
$ cat .gitignore

e.g. \LaTeX\-generated files:

```
*.aux
*.log
*.nav
*.out
*.pdf
*.snm
*.toc
*.vrb
```

global .gitignore

$ git config --global core.excludesfile ~/.gitignore
A worktree gives you an extra working copy of your repository for parallel development

```
git worktree add ../new-worktree-dir some-existing-branch
```

- `../new-worktree-dir` is a clone of your repository
- it should be somewhere outside of your main repository!
- You can then proceed to use the worktree directory as usual, checking out branches, pushing upstream, etc.
- You are finished with a worktree, just delete its directory then run `git worktree prune` from main repository directory