02-Git | Git Commands

Gir Standard Practices, Git commands

Posted on February 22, 2021
Last updated on February 21, 2023

ALL Pdf Notes (https://github.com/amrit94/DataScience/tree/main/notes)

Git Standard Practices

1. Using .gitignore

The .gitignore file is a text file that tells Git which files or folders to ignore in a project. A `local .gitignore` file is usually placed in the root directory of a project. You can also create a `global .gitignore` file and any entries in that file will be ignored in all of your Git repositories.

To add or change your global .gitignore file, run the following command:

```
git config --global core.excludesfile ~/.gitignore_global
```

2. Good Commit Messages

Git Commit Messages (https://chris.beams.io/posts/git-commit/)

Git comment best practice

- `Feat/Feature` : The new feature you're adding to a particular application
- `Fix` : A bug fix
- `Style` : Feature and updates related to styling
- `Refactor` : Refactoring a specific section of the codebase
- `Test` : Everything related to testing
- `Docs` : Everything related to documentation
- `Chore` : Regular code maintenance.[You can also use emojis to represent commit types]
- `Revert` : Reverting things
- `perf` : A code change that improves performance

3. Making Effective Commits on Git

Small Commits

Make small commits. Let's say you got a task. `Instead of doing a 1 commit with 100 lines of code` , make sure you `break down your task` into micro-tasks and then commit for each microtask. This creates your work history and at the same time makes it easier for you to go back to a point where you wrote code that works.

Commit Often

`Keep committing after every micro-task` , don't commit at the end of the day. You will lose a lot of work history. If you use Pomodoro, then make a habit of committing after every Pomodoro.

Commit Working Code

Make sure you `always commit working code` . Never commit something that breaks the codebase. What I mean by this is for example you wrote a function/method for your task that is not being used elsewhere you can commit it. Because it won't break the working application. Alternatively, let's say you are using a method of a service file in controller, committed controller code, and didn't commit service file code. In this case when the controller action is used the application throws an error, which means code breaks.

Committing working code doesn't mean you have to commit a working feature. It just means whatever you are committing isn't breaking anything.

Diff your code

Before committing anything use your ide's diff tool or the integrated Git Version Control. Both VS Code and IntelliJ IDEA has their own diff tools. Make sure you `don't commit debug logs, comments` , or anything you wrote for your own convenience or testing changes.

Learn VS Code's Version Control (https://www.youtube.com/watch?v=AKNYgP0yEOY)

Learn IntelliJ's Version Control (https://www.youtube.com/watch?v=MaQnpCaiop0)

Diff in IntelliJ (https://www.jetbrains.com/help/idea/comparing-files-and-folders.html)

Always Test your code

You already know you need to commit working code. But how do you make sure about that? For this, once you make changes `test your changes thoroughly.` Apart from testing your own changes, make sure changes you made to code didn't break some other feature. The best way to do is to trace the usage of your changes in other places `using simple find feature` of your ide, and then test that feature changes as well.

Avoid Committing passwords & secrets

One common mistake beginner developers do is committing API Keys, secrets, and passwords in the repository. Never commit them, this will create a security breach and some hackers misuse them to exploit your server resources. `Never ever commit passwords, API keys, Secrets, or Bcrypt salts in the codebase`

4. Undoing Changes

What If You

- committed the wrong message?
- skipped file in the last commit?

- accidentally committed on master?
- accidentally committed to the wrong branch?
- want to undo the last 5 commits?
- want to undo changes to a file?

Answers to all these questions are well explained in this good resource (https://ohshitgit.com/). Please go through it and have it as a reference whenever you encounter such cases

General commands

```
git init
git add .
git commit -m 'comment'
git remote add origin get_url

# push
git push origin branch_name
git push origin master
```

Origin

```
# Shows detail for current REPO
vim .git/config

# Clone repo
git clone -b branch_name remote_repo_url

# To get Repo name
git remote show origin

# Change remote URL
git remote set-url origin new_git_url
```

Handle local changes

```
# Shows modified, staged, unstaged files
git status

# Shows all the changes in current Repo
git diff

# Shows comments
git log or git log --online


# # Remove/Disgard changes
# Remove all changes
git checkout -f

# Remove changes of cal.py
git checkout cal.py
```

Branch

```
# Switch to the branch last checked out
git checkout

# Switch to selected branch
git checkout branch_name
```

Create Branch

```
# Create new branch
git branch branch_name

# Create new branch from a commit_log
git checkout -b new_branch comment_id

# Create and switch to new branch
git checkout -b branch_name

# Clone a remote branch and switch to it
git checkout -b branch_name origin/branch_name
```

Delete a branch

```
# The -d option is an alias for --delete
git branch -d branch_name

# The -D option is an alias for --delete --force
git branch -D the_local_branch

# Delete a remote branch
git push remote_name --delete branch_name
git push origin --delete branchName
```

List and Rename branch

```
# List local branches
git branch

# List all branches (local and remote)
git branch -a

# Rename other branch:
git branch -m old new

# Rename current branch
git branch -m new
```

Pull branch

```
# Pull master branch changes to current brunch
git pull origin naster

# Pull remote branch to local
get fetch origin branch_name

# # Merge/Pull
# Merge a branch into the active branch
git merge branch_name
# Merge a branch into a target branch
git merge source_branch target_branch
# in branch to update branch
git merge master
```

Git Stash

This is used to `save changes` in current branch – without any commit. So, current branch becomes change free `and then able to switch` to other branch

```
# stash local changes
git stash

# show all stashes
git stash list

# unstash specific stash
# But will not remove the stash
git stash apply stash@{1}

# unstash the last stash
git stash pop

# Remove the last added stash
git stash drop stash@{1}

# Remove all the stash
git stash clear
```

Reset Commit

```
# SOFT - reset
git reset --soft HEAD~1
git reset --soft commit_id

# MIXED - Default reset
* git reset commit_id

# HARD
git reset --hard commit_id
```

Other

- git clean -df
    - Clean untracked dir/files

- git push origin master –force
- We want to reset the only commit on “master” branch
    - git update-ref -d HEAD
    - git rm –cached -r .

Git - How to fix a bad commit

- git commit –amend -m ‘push on same commit’
- git cherry-pick commit_id

Get some deleted logs/files

- git reflog
    - Shows deleted logs
    - May not remain forever(around 30 days)

Revert

- git revert commit_id
    - Create 1 more commit and revert previous changes