GIT BEST PRACTICES

git

1- COMMIT EARLY, COMMIT OFTEN

It would be best if you commit your changes early and often. Early commit helps to cut the risk of conflicts between two concurrent changes.

Additionally, having periodic checkpoints means that you can understand how you broke something.

If you do small commits, it allows you to use useful tools like git-bisect.

Your message will be clear and focused.

And easy for other developers to understand the code history.

2- DO NOT COMMIT GENERATED FILES

You should commit your code to your repository. If you're checking in generated files, you're doing something wrong. It's more hassle than it's worth to save it in source control.

Your code repository stays clean and organized.

The size of the storage remains under control.

swipe

3- DO NOT COMMIT LOCAL CONFIGURATION FILES

You should not commit your local config files into the source control for many good reasons.

Config files might hold secrets or personal preferences. And configuration files might change from environment to environment.

Different users have different local settings.

Config files might contain secrets like passwords, API keys. So if you will not check them inside your Git repo, protecting them.

4- DO NOT COMMIT BROKEN CODE

You should not commit broken code in the repository before leaving the office at the end of the day.
You can consider Git's "Stash" feature instead of Git commit & push.

Your code repository will stay stable.

At any given time, there is no risk of halting the team's productivity because of the broken code.

swipe

5- TEST YOUR CHANGES BEFORE COMMITTING

It's always good to test your changes before you commit them to your local repository.

Testing is essentially an extra layer of security before you commit. It allows you to identify mistakes and bugs quicker before they go to your production server.

Dev testing helps to decrease software faults.

Save the expense of the project in the long term.

6- WRITE USEFUL COMMIT MESSAGES

Creating useful commit messages is one of the best things you can do every day.

The right commit message helps other developers in many ways. They can understand the code better even without looking at it.

Good commit messages act as documentation for other developers.

Useful commit Messages help maintain the project in the long run.

swipe

7- REVIEW YOUR COMMIT (CODE REVIEW)

In general, you as a developer have no permission to commit or push your changes directly in the main branch. Dev team checks the sets of modifications before merging in the main branch.

Code review helps keep the company's coding style intact.

Finding bugs early, when it is cheap to patch them.

8- USE REFERENCE NUMBER WITH YOUR COMMIT

For writing commit messages, all teams have different expectations. So it isn't easy to achieve one size which fits all the commit messages.

Using a reference number with your commit messages is a good idea.

The reference may be an external bug tracker reference number or your VSTS task number, or any other.

It adds clarity to why you made some changes.

It helps others to review your code.

swipe

9- USE PULL REQUEST

The use of pull requests based development is good to have.

Also, pull requests help others to review your changes. It improves the quality and helps to make sure only stable code is going into the master branch.

Enough testing and it improves the stability of your codebase.

Also, pull request help in reducing conflicts.

10- DO NOT DELAY IN PULL REQUEST

Pull requests should not be unattended for a long time. Also, this delay will increase the chances of more conflicts.

So It is important to review the pull request soon for any merge or to deploy. It will fast and smooth the process.

swipe

11- COMPLETE YOUR WORK BEFORE A PULL REQUEST

The use of pull requests based development is good to have.

Also, pull requests help others to review your changes. It improves the quality and helps to make sure only stable code is going into the master branch.

Enough testing and it improves the stability of your codebase.

Also, pull request help in reducing conflicts.

12- DO NOT DELAY IN PULL REQUEST

Pull requests must be atomic in nature. First stuff first. It's the job of the author to make the code review quick.

You need to present the code changes in a pleasant way. So do not create a Pull request for your broken code.

And in case of a large pull request, split by features. Tidy up your work and all commits before a pull request.

swipe

13- DEFINE CODE OWNERS

Useful code review is crucial for any successful project. But sometimes it's not clear who should review it.

And this practice depends from team to team. You can assign owners for different codebases.

An extra layer of security for your code

Review Process become more effective

14- USE GITIGNORE FILE

Gitignore is a text file that helps to ignore predefined files and directories. It may be config files, user settings, and other unwanted files.

You can choose a relevant template from Gitignore.io to get started quickly.

This file helps to keep your repository clean.

Also it ensures files that are not monitored by git remain untracked.

swipe

15- USE REBASE

As you continue to grow your feature branch, rebase it frequently against the main.

This means routinely performing the following steps:

```
$ git checkout main
$ git pull
$ git checkout your-feature-branch
$ git rebase main
```

A cleaner project history, and it's easier to review.

Commit history remains stacked up as an exact sequence.

16- USE GIT STASH WITH A MESSAGE

You can stash your uncommitted modifications (both staged and unstaged) for later usage. You can use them back from your working copy.

Git stash is excellent for storing changes temporarily.

You should use Git stash with a meaningful message.

```
$ git stash save “Your meaningful stash message”.
```

It helps you to keep track of what’s what easily!

swipe

17- USE BRANCHES

The most important feature of git is its branching model. These branches are more like a new copy of your code's current state.

It would help if you used branches.

The use of branches lets you manage the workflow more quickly and easily.

18- DELETE STALE BRANCHES

Without the possibility of losing any code, you can securely remove a git branch.

And You should remove old git branches that are no longer in use.

If you do not delete old branches, you may end up with many stale-branches, and it's become hard to manage.

Help in keeping your repository clean.

Avoid confusion with having few branches in use.

swipe

19- PROTECT YOUR MAIN BRANCH

By protecting the main branch, nobody should make a direct check in to the main branch. Or can rewrite the history of the main branch.

Also, only merge requests should be allowed in master after a code review process. So no git push straight to the main branch.

It's your production code, ready for the world to roll out.

The main branch always stays in stable mode.

20- TEST BEFORE YOU PUSH

Test your code changes before you push is a good practice. In fact, it's part of my favorites - AFTER technique. (which I will share in the Bonus section)

Instead of depending on QA to find any bug or issues, a successful developer checks their code.

If you did not test your code and property, it has a knockout effect. You might block the entire development team. (and it might happen)

So it's always worth spending a few minutes to test your code before you push.

swipe

21- DO NOT USE ONLY NUMBERS

As part of the branch naming convention, do not use only numbers.

It's hard to know from these numbers what this branch is about. If you have to associate an external bug tracker reference, then use more details.
Here are few examples

cr-12312
bug-0912

22- USE SEPARATORS

You can use a separator in your branch name.

Many developers use slash, and others use hyphens or underscores. Which one to use depends on the needs of both you and your team.

It makes the branch name easier to read.

It helps in avoiding confusion.