GIT CheatSheet

Clone

Clones remote repository to current directory

```
git clone <repository>
```

```
git clone https://github.com/marcoom/license-plate-detector.git
```

New Branch

After cloning, you can create a new branch based on master / main and move to that branch to work there:

```
git checkout -b <branch_name> master
```

```
git checkout -b feature/issue-123-google-oauth master
```

In this new branch, you can perform your changes to the repository

Branch naming convention (see annex for details):

```
<type>/<optional-issue-id>-<description>
```

type: main/master/develop, feature, bugfix, hotfix, release, chore/

Pull changes

To fetch changes from remote to the local repo and merge with current branch

```
git pull origin master
```

Add and restore files (staging)

To add files to staging:

```
git add "<file_name.ext>" # adds a single file
```

```
git add . # adds all files
```

To remove files from staging, simply replace “add” with “restore”

Commit changes

After adding files to staging, commit them using:

```
git commit -m "<Your comment>"
```

Comments convention (see annex for details):

```
git commit -m "type(opt scope): description" -m "opt body" -m "opt footer"
```

type: fix, feat, build, chore, ci, docs, style, refactor, perf, test

Push to remote

To push the committed changes to the remote repository:

```
git push origin <branch_name>
```

```
git push origin feature/issue-123-google-oauth master
```

Other useful commands

See all available branches (the current one is marked with *), use this command:

```
git branch
```

See the current status of your work:

```
git status
```

Initialize a folder as a git repository

```
git init
```

See differences between two commits or branches

```
git diff <commit1/branch1> <commit2/branch2>
```

Stash the changes

```
git stash
```

Show last n commits logs

```
git log -<n> --oneline
```

Show remote repositories whose branches you track

```
git remote -v
```

Pull/Merge Request (PR/MR)

After pushing to the remote, the new branch should be in the remote repository, accessible through your provider's web interface (GitHub, GitLab or BitBucket for example). To merge the changes to master you must create a Pull Request (GitHub) or Merge Request (GitLab) and then merge your changes (follow your team's standards for merging, for example merge only after a Peer Review). After merging your changes to master, it is good practice to delete your remote branch.

Merge Conflicts

A Merge Conflict occurs when Git can't auto-merge because both sides changed the same part of a file (or one deleted/renamed what the other modified), so it needs your criteria to decide which changes to keep. It can happen for example when the master branch in the remote is updated, and you have also modified your local branch, in this scenario, a merge conflict can happen when you are pulling changes from remote to your local branch, or when you are trying to merge your branch with master in the remote repository.
To resolve a conflict, you must modify the conflicting files and try to merge again. You can follow this guide from your local branch:

```
git status  # see conflicted files
# open each conflicted file and fix it
git add . # stage resolved files
git commit -m "Resolve merge conflicts"
git push origin my_branch
```

Conflicted files look like this (GIT adds markers to guide you):

```
def greet(name):
<<<<<<< HEAD
    return f"Hell0 {name}!"
=======
    return f"Hello, {name}."
>>>>>>> origin/master
```

Fixing conflicts means removing <<<<<<<, =======, >>>>>>> markers, and keep the right parts. Your resolved file should then look like this:

```
def greet(name):
    return f"Hello, {name}!"
```

Versioning

```
MAJOR.MINOR.PATCH
```

If your next release contains commits with:

- Breaking Changes incremented the MAJOR version (regardless of type)
- API relevant changes (feat type) increment the MINOR version (backward-compatible additions)
- Else increment the PATCH version (backward-compatible bug fixes)

Use leading "v" for tagging in GIT (e.g. Tag: "v1.2.3"). Do not use it for storing code artifacts (e.g. my-lib-1.2.3.tar.gz or __version__ = "1.2.3")

Source: https://semver.org/

Branch naming convention

For following good practices on naming conventions, your comment should be structured as follows (note that there are two blank lines):

```
<type>/<optional-issue-id>-<description>
```

Type (mandatory):

- main: The main development branch (e.g., main, master, or develop)
- feature/: For new features (e.g., feature/add-login-page)
- bugfix/: For bug fixes (e.g., bugfix/fix-header-bug)
- hotfix/: For urgent fixes (e.g., hotfix/security-patch)
- release/: For branches preparing a release (e.g., release/v1.2.0)
- chore/: For non-code tasks like dependency, docs updates (e.g., chore/update-dependencies)

Issue ID (optional): If applicable, include the ticket number from your project management tool to make tracking easier. For example, for a ticket issue-123, the branch name could be feature/issue-123-new-login.

Description (mandatory): should be descriptive yet concise, clearly indicating the purpose of the work.

PERMITTED CHARACTERS: lowercase letters (a-z), numbers (0-9), and hyphens(-). Dots only when versioning (release type). AVOID underscores (_)

Examples

```
feature/issue-123-google-oauth
bugfix/fix-null-pointer-on-startup
hotfix/production-crash-on-payment
chore/update-dependencies-2025-07
release/1.4.0
```

Source: https://conventional-branch.github.io/

Comment convention

For following good practices on comments conventions, your comment should be structured as follows (note that there are two blank lines):

```
<type>(optional scope)[!]: <description>

[optional body]

[optional footer(s)]
```

This can be achieved with the following command:

```
git commit -m "type(opt scope): description" -m "opt body" -m "opt footer"
```

Type (mandatory):

- fix: patch a bug
- feat: add a new feature
- build: changes to build system or external dependencies (e.g., npm, pip, Docker)
- chore: routine tasks not affecting src/tests (e.g., bumping versions, renaming files)
- ci: changes to CI configuration or scripts (GitHub Actions, GitLab CI, etc.)
- docs: documentation-only changes
- style: code style/formatting (no logic change)
- refactor: code change that neither fixes a bug nor adds a feature
- perf: performance improvements
- test: add or modify tests (no production code change)

Scope (optional):
Short, lowercase noun that pinpoints the area affected (e.g. api, release, changelog). Can be a module, package, layer, or feature. DO NOT USE issue identifiers as scope.

BREAKING CHANGE: when the commit introduces a modification that breaks compatibility with existing code/users (e.g. modifying an API contract, removing an endpoint), you must add an exclamation mark (!) before the (:) and an (optional) footer that begins with “BREAKING CHANGE: “ and then explains the changes and how to migrate.

Description (mandatory): contains a concise description of the change.

- Use the imperative, present tense: "change" not "changed" nor "changes".
- DO NOT capitalize first letter
- DO NOT end with a dot (.)

Body (optional): Explains the motivation for the change (the “why”) and the contrast with previous behaviour. Is also written in imperative present tense.

Footer (optional): Holds structured metadata/trailers. Can be: BREAKING CHANGE: notes, issue references (Closes #123), co-authors, or sign-offs—one item per line (multiple footers are permitted).

Example 1

```
docs: correct spelling of CHANGELOG
```

Example 2

```
feat(auth)!: add Google login
Replace legacy session-based login with OAuth2 to enable SSO and reduce
credential handling
BREAKING CHANGE: legacy /login endpoint removed
Closes #42
```

Source: https://www.conventionalcommits.org/en/v1.0.0/