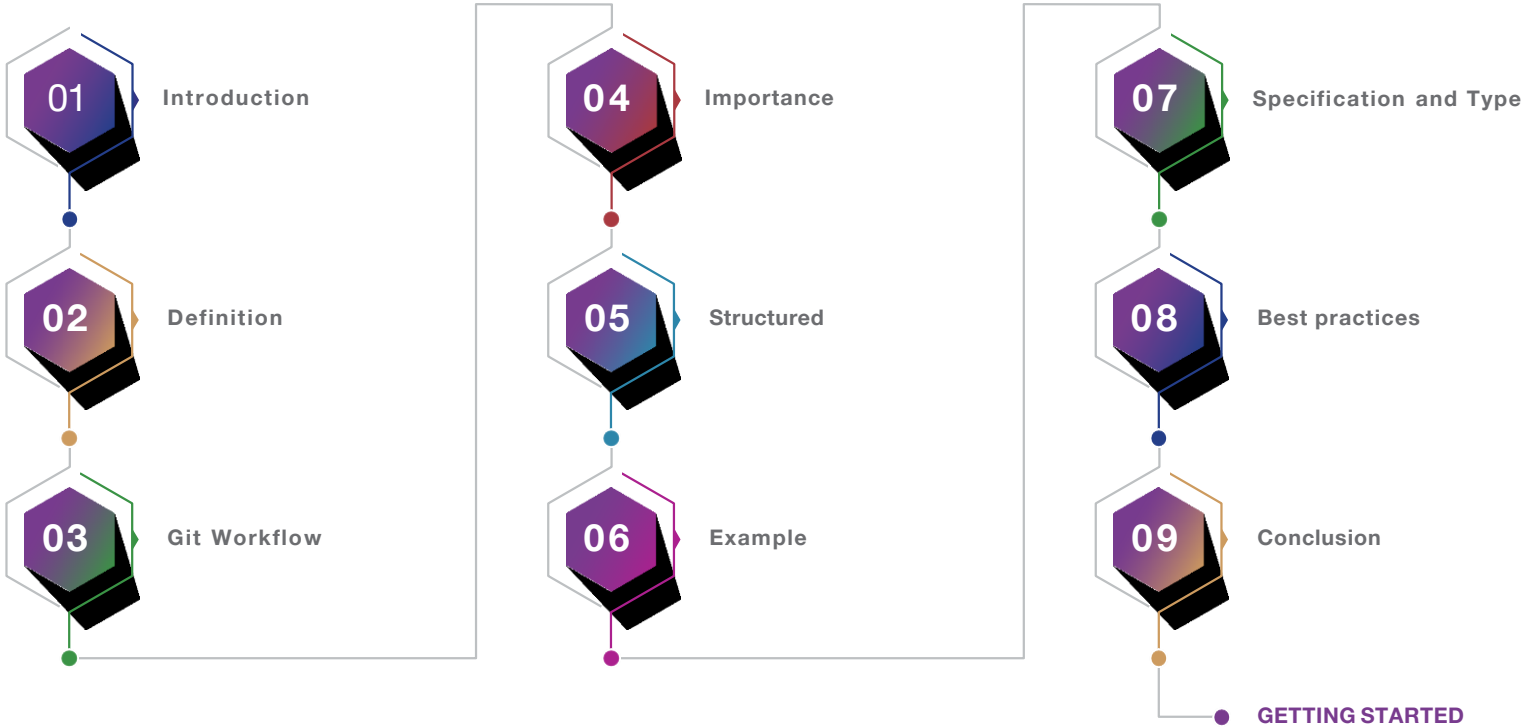
AppSec Training – Conventional Commits

a ultimate developers guide.

Developed By : Prasad Keluskar

Agenda



INTRODUCTION

WHAT IS COMMIT?

Readable commit history

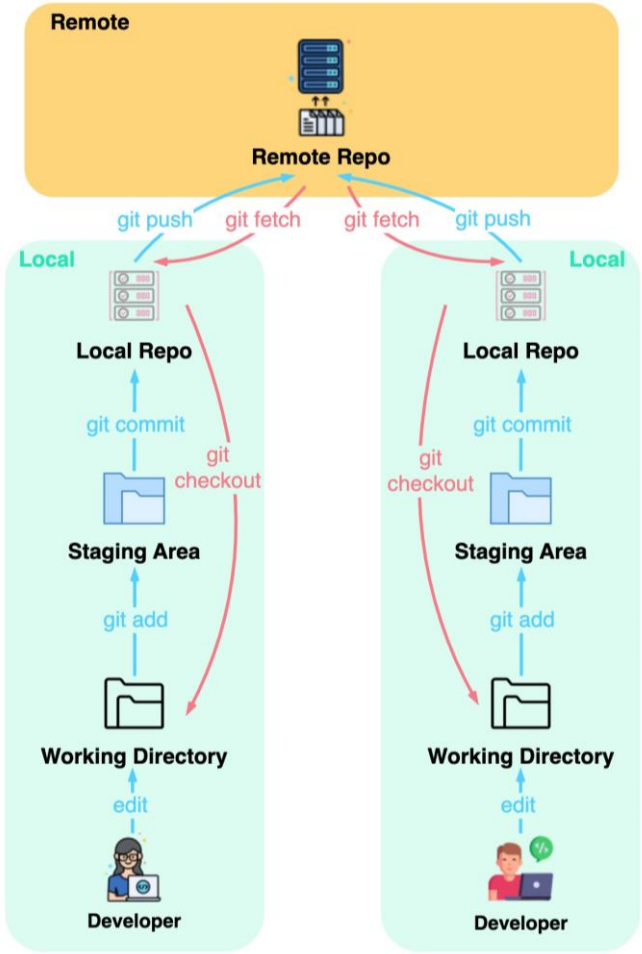
Conventional Commits

“A commit refers to the action of saving changes made to the code in a version control system. This is often done when a feature or bug fix is complete, or when a certain milestone has been reached.

The changes are saved with a commit message, which is a brief description of the changes made. This allows developers to keep track of what changes were made and why. In version control systems like Git, each commit is given a unique identifier, which allows you to reference that specific set of changes in the future”.

How does Git Work?

Git is a distributed version control system. Every developer maintains a local copy of the main repository and edits and commits to the local copy. The commit is very fast because the operation doesn't interact with the remote repository. If the remote repository crashes, the files can be recovered from the local repositories.



Git Flow?

Here are the key branches in Standard Git Flow:

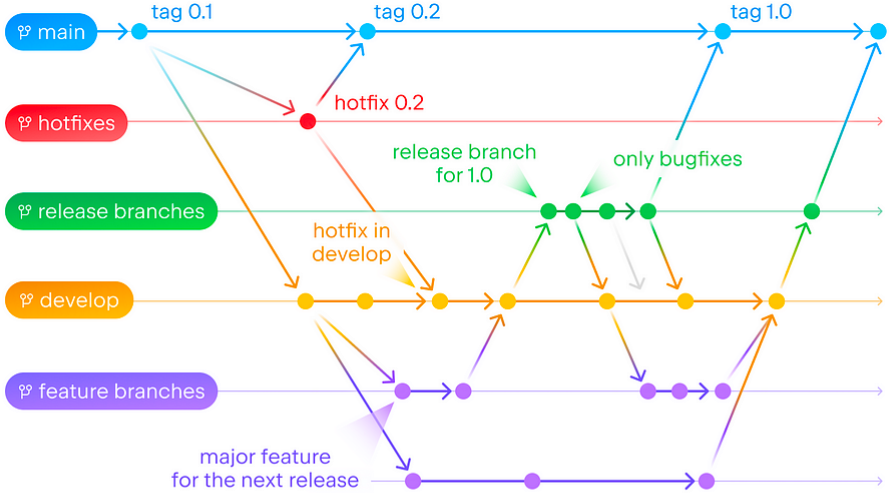
1.Master: The master branch represents the production-ready code and should always contain the latest stable release.

2.Develop: The develop branch is used for ongoing development. Feature branches are created from and merged back into develop.

3.Feature Branches: Feature branches are created for developing new features or enhancements. They branch off from develop and are merged back into develop when the feature is complete.

4.Release Branches: Release branches are used to prepare for a new release. They branch off from develop and are used for bug fixes, testing, and documentation updates. Once a release is ready, the release branch is merged into both master and develop.

5.Hotfix Branches: Hotfix branches are created to address critical issues in the master branch (i.e., production). They branch off from master, and once the fix is complete, they are merged back into both master and develop.

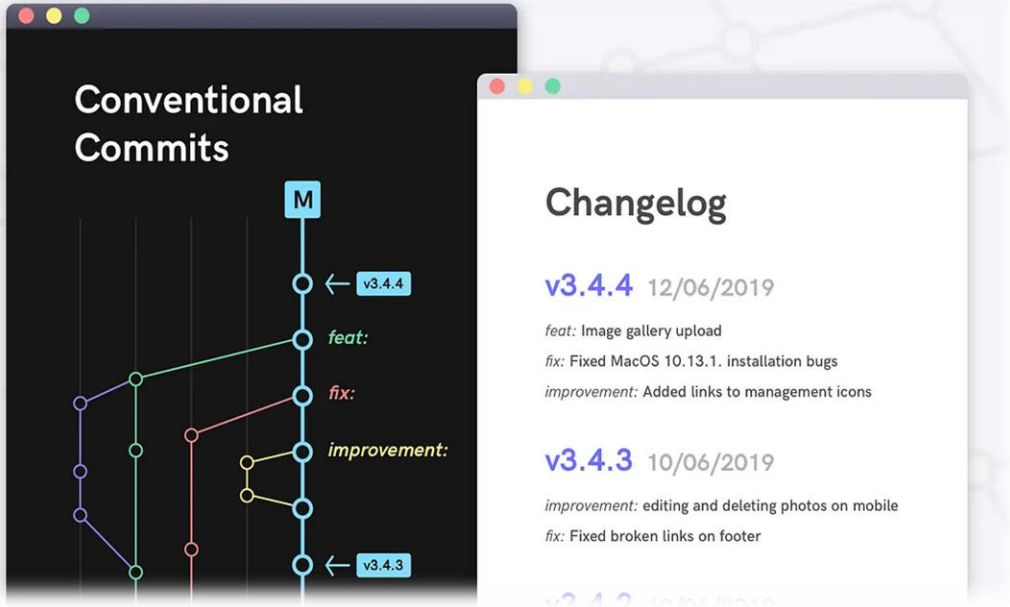


- The `main` branch is for production code only.
- The `develop` branch is for development code.
- `feature` branches are created from the `develop` branch.
- `hotfix` branches are created from the `main` branch.
- `release` branches are created from the `develop` branch.

Why Should You Write Better Commit Messages?

When you're working on a project or any repository and execute **git log** to see a list of old commit messages, the majority of us who have gone through tutorials or made quick fixes will confess, "Yep… I have no clue what I meant by 'Fix login' a month ago."

You might have come across code in a professional setting where its purpose or function was a mystery. You've been left in the dark without any code comments or a traceable history, even speculating, "What are the chances this will cause everything to fail if I delete this line?" Moreover, if you're working on code fixed by a previous developer, it can be cumbersome to understand what changes were made, when and where they were made, and whether those changes are functioning correctly.



Example

Commit Structured

commit message should be structured as follows:

<type>[optional scope]: <description> [optional body] [optional footer]

The commit contains the following structural elements, to communicate intent to the consumers of your library:

1.fix: a commit of the type fix patches a bug in your codebase (this correlates with **PATCH** in semantic versioning).

2.feat: a commit of the *type* feat introduces a new feature to the codebase (this correlates with **MINOR** in semantic versioning).

3.BREAKING CHANGE: a commit that has the text BREAKING CHANGE: at the beginning of its optional body or footer section introduces a breaking API change (correlating with **MAJOR** in semantic versioning). A breaking change can be part of commits of any *type*. e.g., a fix:, feat: & chore: types would all be valid, in addition to any other *type*.

4.Others: commit types other than fix: and feat: are allowed, for example @commitlint/config-conventional (based on the Angular convention) recommends chore:, docs:, style:, refactor:, perf:, test:, and others. We also recommend improvement for commits that improve a current implementation without adding a new feature or fixing a bug. Notice these types are not mandated by the conventional commits specification, and have no implicit effect in semantic versioning (unless they include a **BREAKING CHANGE**, which is **NOT** recommended). A scope may be provided to a commit's type, to provide additional contextual information and is contained within parenthesis, e.g., feat(parser): add ability to parse arrays.

Examples

Commit message with description and breaking change in body

feat: allow provided config object to extend other configs
BREAKING CHANGE: `extends` key in config file is now used for extending other config files

Commit message with no body

docs: correct spelling of CHANGELOG

Commit message for a fix using an (optional) issue number.

fix: minor typos in code see the issue for details on the typos fixed fixes issue #12

Commit message with scope

feat(lang): added polish language

Full Conventional Commit Example

fix: fix foo to enable bar This fixes the broken behavior of the component by doing xyz. BREAKING CHANGE Before this fix foo wasn't enabled at all, behavior changes from <old> to <new> Closes JiraTicket ID

Commit Message Comparisons

Review the following messages and see how many of the suggested guidelines they check off in each category

- fixed bug on landing page
- Changed style
- oops
- I think I fixed it this time?
- empty commit messages

- feat: improve performance with lazy load implementation for images
- chore: update npm dependency to latest version
- Fix bug preventing users from submitting the subscribe form
- Update incorrect client phone number within footer body per client request

Benefits and Best Practices

Steps to Write Better Commit Messages

Capitalization and Punctuation	Mood	Type of Commit	Length	Content
Capitalize the first word and do not end in punctuation. If using Conventional Commits, remember to use all lowercase	Use imperative mood in the subject line. Example – Add fix for dark mode toggle state. Imperative mood gives the tone you are giving an order or request.	Specify the type of commit. It is recommended and can be even more beneficial to have a consistent set of words to describe your changes. Example: Bugfix, Update, Refactor, Bump, and so on	The first line should ideally be no longer than 50 characters, and the body should be restricted to 72 characters.	Consider the following: • Why have I made these changes? • What effect have my changes made? • Why was the change needed? • What are the changes in reference to?

Remember, the goal of a commit message is to explain what you did and why you did it. A good commit message can make it easier for you and others to understand the history of your code.

Commit Type and Specification

Type	Meaning	Description
feat	Features	A new feature
fix	Bug Fixes	A bug fix
docs	Documentation	Documentation only changes
style	Styles	Changes that do not affect the meaning of the code (white-space, formatting, missing semi-colons, etc)
refactor	Code Refactoring	A code change that neither fixes a bug nor adds a feature
perf	Performance Improvements	A code change that improves performance
test	Tests	Adding missing tests or correcting existing tests
build	Builds	Changes that affect the build system or external dependencies (example scopes: gulp, npm)
ci	Continuous Integrations	Changes to our CI configuration files and scripts (example scopes: Travis, Circle, BrowserStack)
chore	Chores	Other changes that don’t modify src or test files
revert	Reverts	Reverts a previous commit

Benefits and Best Practices

Readability and Understanding

commits can improve the readability and comprehension of your commit history.

Code Review and Collaboration

commits can simplify the code review process. It presents reviewers with a higher-level view of the changes, reducing the cognitive load of reviewing numerous smaller commits.

Granularity and Debugging

commits sacrifices granularity, as individual changes are combined into a single commit. This can make it more challenging to pinpoint specific changes or debug issues that might arise

Branch Management and Merging

commits helps maintain a clean and concise history, particularly when merging branches. It reduces the clutter of multiple small commits, minimizes conflicts, and enhances the clarity of the branch's purpose..

Extension & Tools

Top Git Extensions and Tools

By utilizing the following tools and plugins, you can compose and scrutinize your project commits, thereby indirectly boosting your productivity.

1. Conventional Commits
2. Git History
3. Git Lens
4. Git Blame
5. gitlint (Python)

1. Commitlint (Node application)
2. git-commit-plugin
3. Git Graph
4. Gitlens Code Lens
5. Better Comments

Conclusion

Conclusion

“Cultivating the ability to craft effective commit messages is a valuable skill that enhances your communication and collaboration with your team. Commit messages act as a record of modifications, serving as a historical document that aids us in understanding past actions and making informed decisions for the future.

While there are established standards that we can adhere to, the key is to settle on a convention that your team finds descriptive and considers future readers. This approach will undoubtedly yield benefits in the long run”.

THANKS