The art of using git

In this book I will describe how I use git. In doing so I hope to help others in learning git. Please take everything I write here with a grain of salt. Please don't blindly use this guide!

- Branches - Gitflow
- Commiting
- Merge - Rebase - Squash
- Typical workflow

Branches - Gitflow

Main and dev

For starters: Do as I say and not as I do.

I say this because I myself have forgot to use separate branches. It is really easy to forget and you will only notice it when it becomes a pain in the ass to fix.

Ok let's start this guide!

Separate branches are a must if you want to step up your git game. It will help you and others. Here are the rules for it:

Main should always be "bug free". Meaning, it should just work without any major problem.
Dev should be used as a testing ground for new features. Meaning, it can have problem.
Feature/<feature-name> should be used to develop feature in an isolated matter and later be merged into dev.

Depending on what workflow you want to use the names and rules can slightly vary.

The use of feature branches

Feature branches will help you develop faster and more time efficiently. The main reason for that is that you don't need to deal with the commits of other developer because your feature branch is isolated from dev.

Protected branches

You should always protect your main branch!

Protecting main as a number of advantages.

1. Less chances of breaking main

2. Requiring code reviews which can lead to finding bugs
3. etc

If you are working alone this will not really help you other than protect you from yourself.

If you are working in a team then this is a must.

The protection rule I use are as following:

1. Require a pull request before merging
2. Require approvals (Amount depends on team size) (Not really needed for dev)
3. Do not allow bypassing the above settings

Those rules can also be used for dev branches.

Other resources to look at

https://www.gitkraken.com/learn/git/git-flow (Last accessed 22.03.2023)

https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow (Last accessed 22.03.2023)

Commiting

How do I commit right?

What makes a good commit message?

1. Separate subject from body with a blank line
2. Limit the subject line to 50 characters
3. Capitalize the subject line
4. Do not end the subject line with a period
5. Use the imperative mood in the subject line
6. Wrap the body at 72 characters
7. Use the body to explain what and why vs. how

A good example would be:

```
Summarize changes in around 50 characters or less

More detailed explanatory text, if necessary. Wrap it to about 72
characters or so. In some contexts, the first line is treated as the
subject of the commit and the rest of the text as the body. The
blank line separating the summary from the body is critical (unless
you omit the body entirely); various tools like `log`, `shortlog`
and `rebase` can get confused if you run the two together.

Explain the problem that this commit is solving. Focus on why you
are making this change as opposed to how (the code explains that).
Are there side effects or other unintuitive consequences of this
change? Here's the place to explain them.

Further paragraphs come after blank lines.

 - Bullet points are okay, too

 - Typically a hyphen or asterisk is used for the bullet, preceded
```

```
   by a single space, with blank lines in between, but conventions
   vary here

If you use an issue tracker, put references to them at the bottom,
like this:

Resolves: #123
See also: #456, #789
```

Atomic commits

Each commit should express a single unit of work on a single feature. Don't bulk-commit all of the work you did today, or write your entire project in one go and then commit that all at once. Commit whole pieces of work, ideally leaving the application in a workable state.

When you need to think what happened as you are writing your commit message then you already have done to much for a single commit.

Tip: The same can also apply for pull request. Keep it short and don't try to merge 50 commits at ones, if you are working in a team.

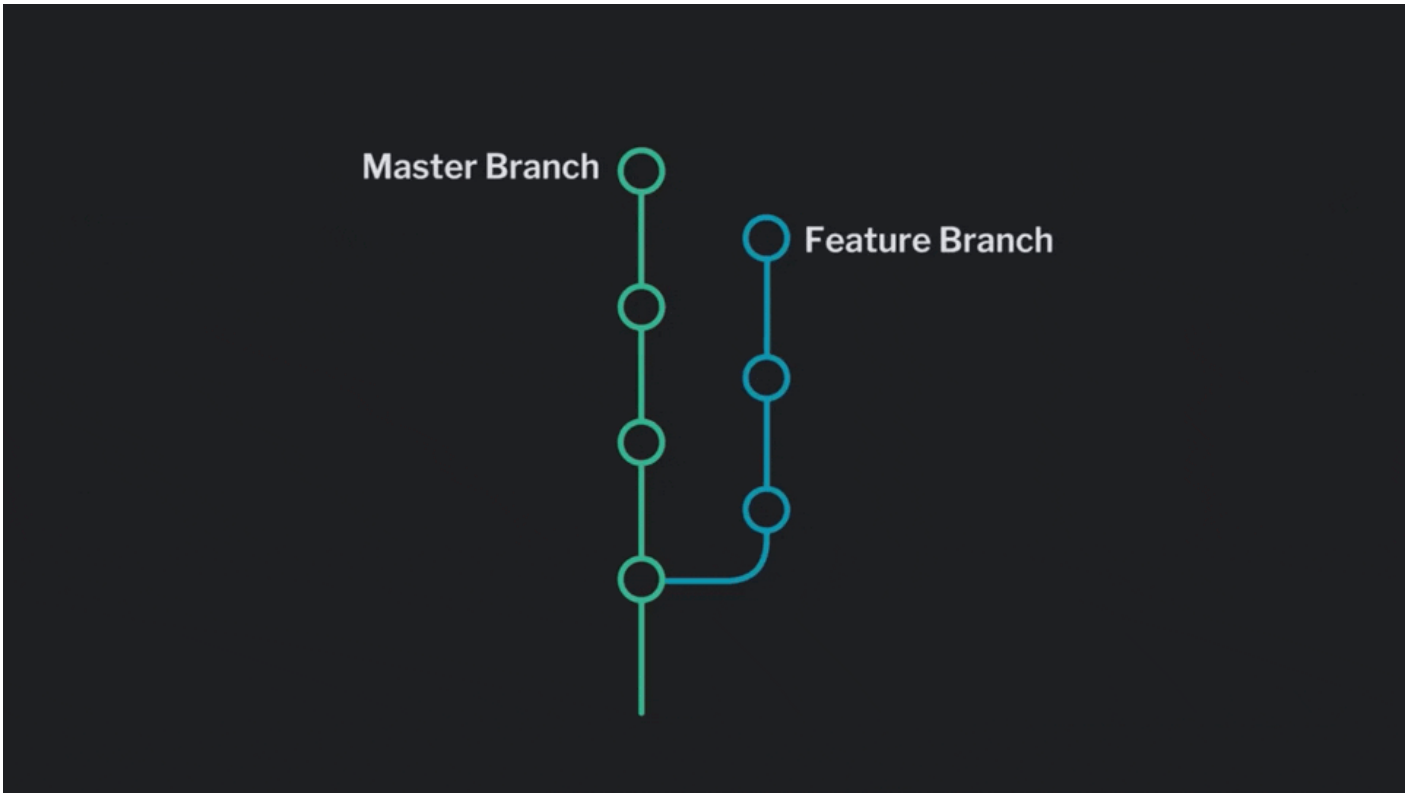
Sources

https://cbea.ms/git-commit/

Merge - Rebase - Squash

When should you use either of them?

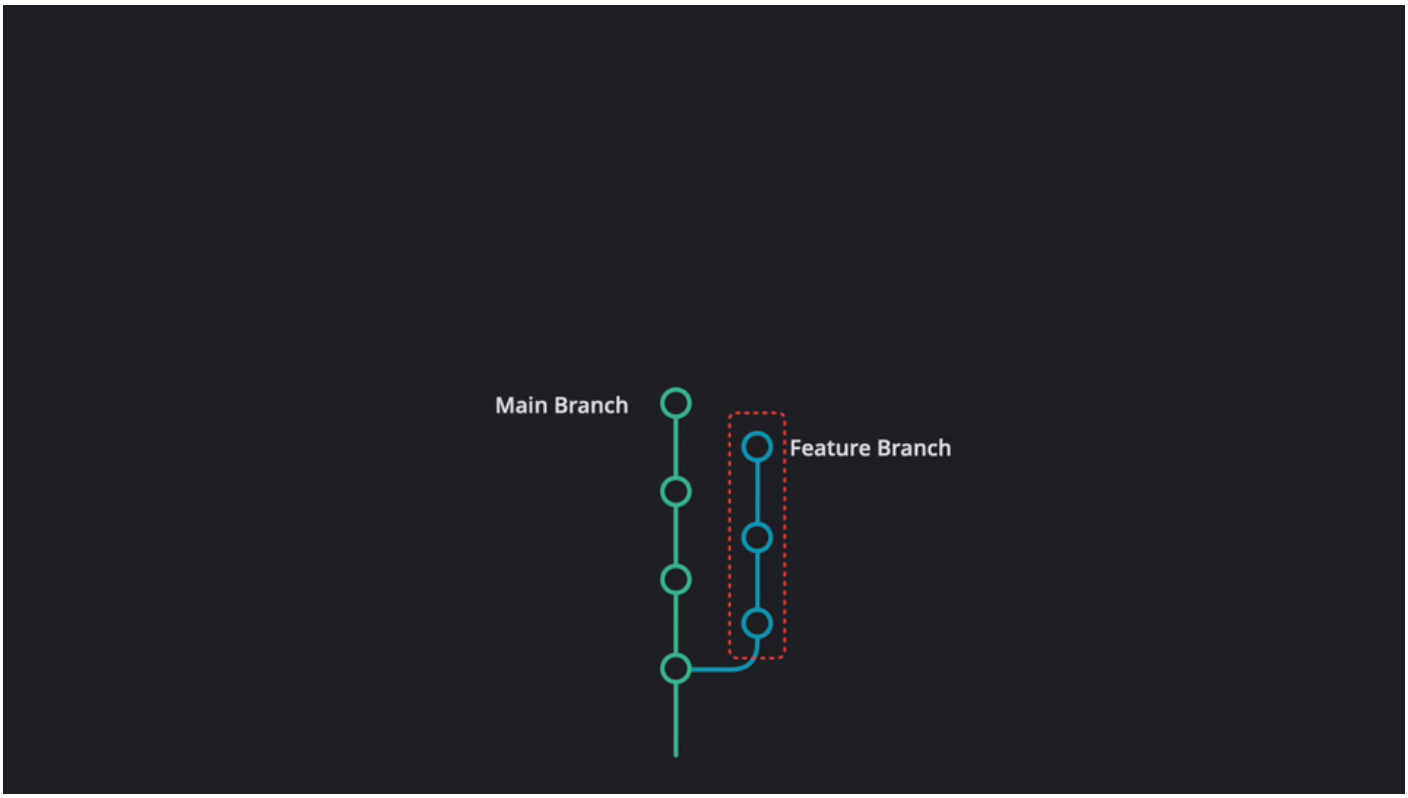
Merge



Source: https://www.gitkraken.com/learn/git/git-merge (Last accessed 22.03.2023)

Merging is used to merge two branches and preserve their history. It also makes it easier to undo mistakes.

Rebase

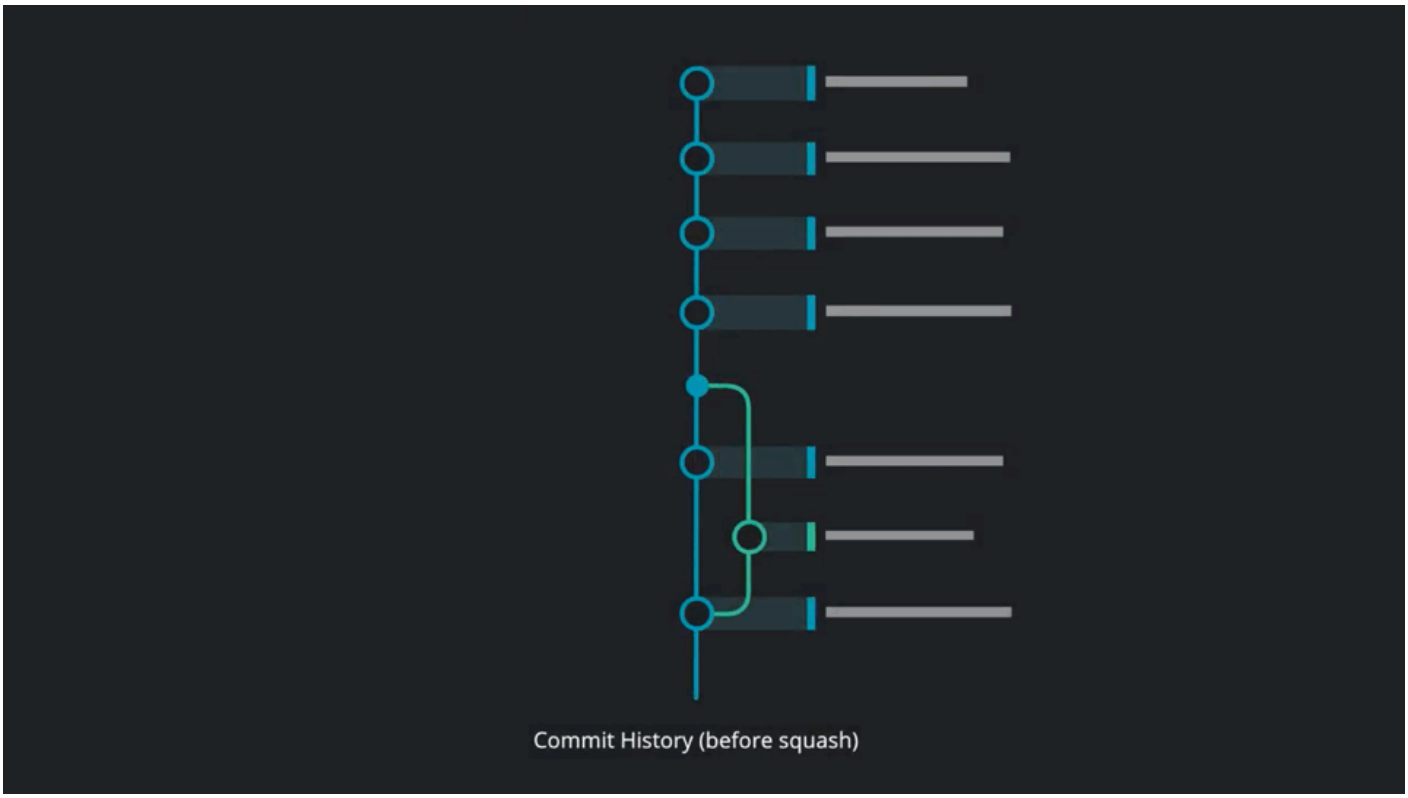


Source: https://www.gitkraken.com/learn/git/git-rebase (Last accessed 22.03.2023)

Rebasing can be beneficial in feature branches. Dev --> feature/<feature-name>

Rabase will rewrite commit history so please don't use it on shared branches like dev and main.

Squash



Source: https://www.gitkraken.com/learn/git/git-squash (Last accessed 22.03.2023)

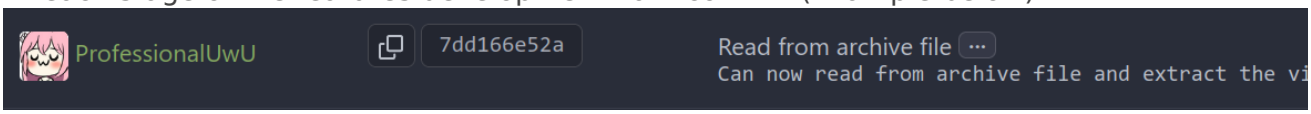
Squash can be used to condense a lot of commits into one. This can be useful if want to merge dev into main and you have 50 commits.

Typical workflow

How does a typical workflow look like?

I am going to take one of my repo's as an example on how one would go about doing all the things I described in this book.

- I look at the feature I want to implement
- In my code I begin writing what I want to do in plain English
- I begin with the easiest aspect of it
  - At each stage of the features development I am commit (Example below)



- When I am finished with my feature I rebase my with dev
  - I do this to fix potential issues (merge conflict)
- If everything works as expected I will start a pull request
  - I personally do a merge commit so I can keep the commit history
- When all features are tested together and everything works one can begin a pull request into main
  - For this I would recommend squash
- Then I would recommend to make a tag for the current state of development
  - As well as a release if wanted