Agenda for Day 2

1. Git in Production
2. Clean Code
3. Automated Testing

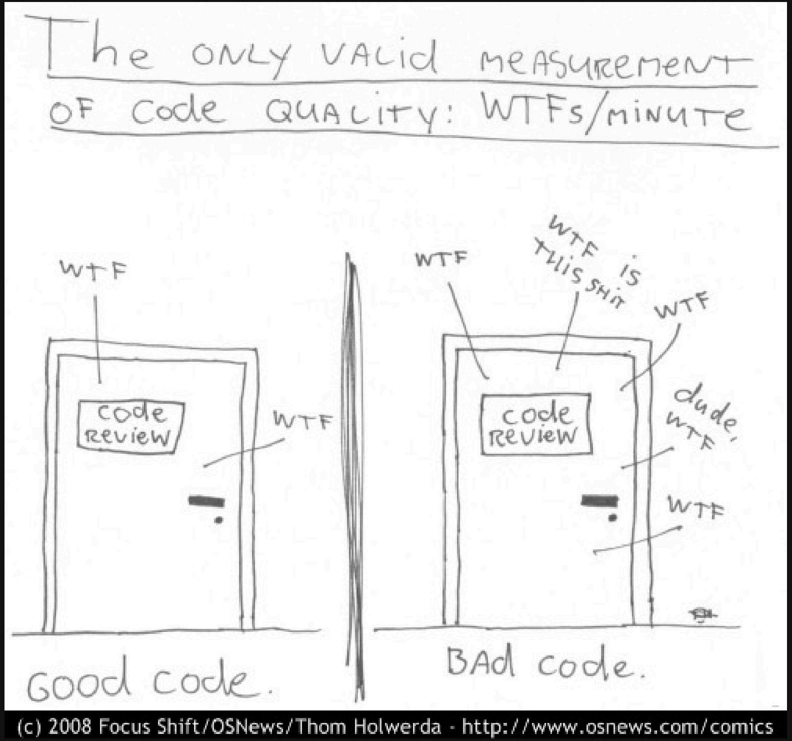
Disclaimer

1. View this workshop is a *starting point* for learning. I'm aiming to give a (somewhat) holistic overview of things to learn.

2. All topics of today are actively dicussed among the dev community. I will focus on things that are considered "best practice" among many but some people might still have different opinions and that's okay!

3. Don't be discouraged if you feel overwhelmed. I'm trying to have a high content density and move fast. Slides are published for later reference.

4. I'm also just a Coxi student with some work experience under my belt. If you find any mistakes throughout this workshop, let me know :)


The only valid measurement of code quality: WTFs/minute
WTF
WTF
code review
Good code.
WTF
WTF is this shit
WTF
code review
dude, WTF
WTF
BAd code.
(c) 2008 Focus Shift/OSNews/Thom Holwerda - http://www.osnews.com/comics

OSnews

Git in Production

Rough agenda:

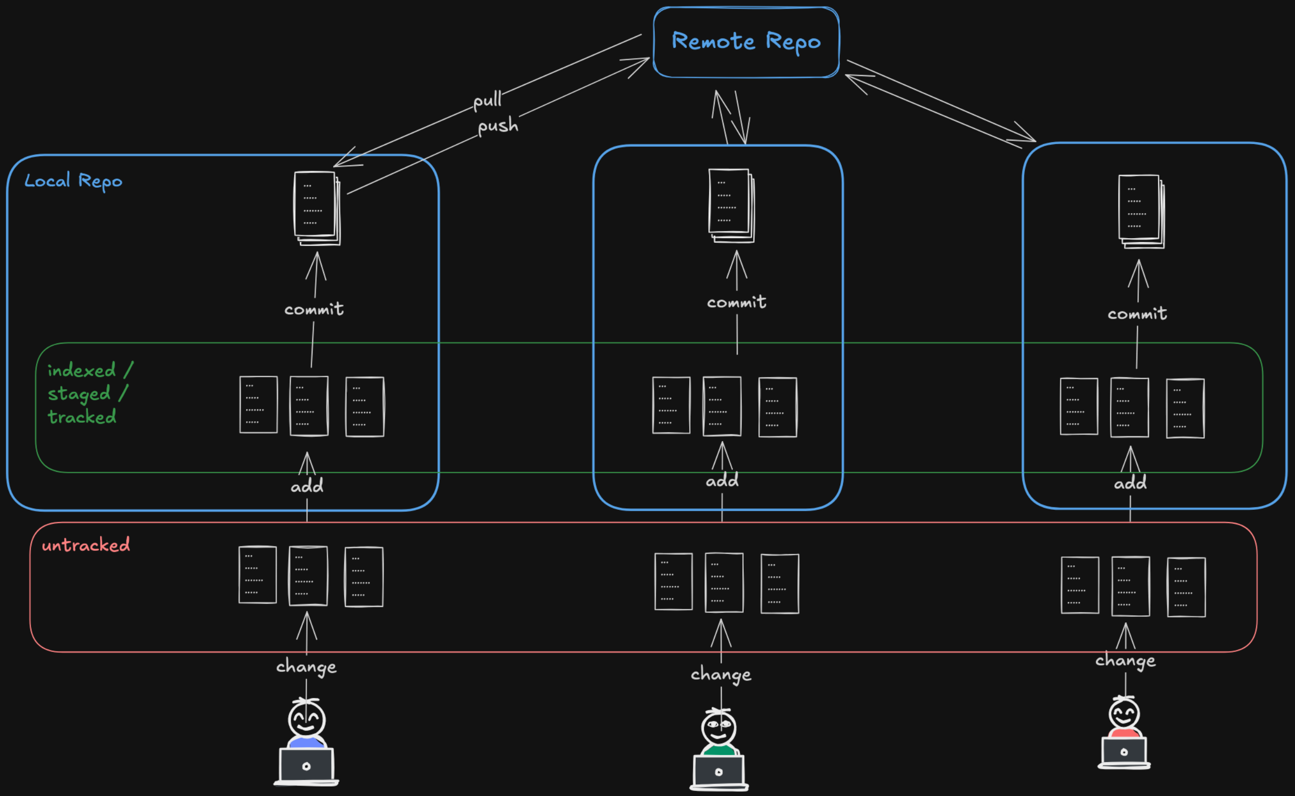
1. Requirements
2. Basics of git
3. Branches
4. Merging
5. Rewriting History
6. Remote(s)
7. Code review
8. Collaboration

Requirements

- Have git installed
- Have a github.com account connected to your machine (preferrably via SSH)
- Some basic knowledge of the CLI on your machine (how to move around, create/delete files, etc.)
- That’s it

What is git?

= free and open source distributed version control system (VCS)



Your first git repo

1. Create a local repo with

   ```
   git init
   ```

2. Configurate

   If the following cmd has no output:

   ```
   git config --get user.name
   ```

   You need to configure your username and email

   ```
   git config --add --local user.name "Louis Kapp"
   git config --add --local user.email "l.kapp@4eign.de"
   ```

   You can check your whole configuration using

   ```
   git config --list
   ```

The Basics

These simple commands suffice 80% of the time

- `git add <path-to-file | pattern>` will add zero or more files to the index (staging area)
- `git commit -m '<message>'` will commit what changes are present in the index.
- `git status` will describe the state of your git repo which will include tracked, staged, and untracked changes.

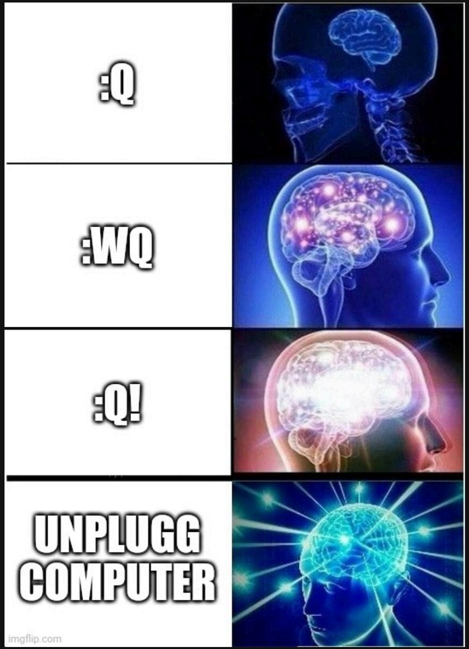
The vi experience

You’re cooked

You may wonder what happens if you just `git commit` without specifying a `--message` (`-m`):

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch main
#
# Initial commit
#
# Changes to be committed:
#       new file:   test.md
#
```

Quit vi using `:q!` or specify your favorite editor (e.g. VSCode) in your git config to never end up in this situation: `git config --global core.editor "code"`



r/ProgrammerHumor

Exercise

~ 3 minutes

1. Create and configure an empty repo
2. Create a file
3. Check the status of git to see that its untracked.
4. Add it to the index
5. Check the status of git to see that it is staged.
6. Commit it with a nice message
7. Check the status of git to see if there is anything left to do
8. View all your commits in the repo with a single command (bonus)

Git behind the curtain

wait, which curtain?

- Use `git log` to view all commits in your repo

```
commit ea7aad16a91e11d339df42c5c726b6761444c96a (HEAD -> main)
Author: Louis Kapp <l.kapp@4eign.de>
Date:   Wed Jan 22 17:07:48 2025 +0100

    test
```

- Each commit has unique ID in form of a SHA
- `git log --oneline` shows the shorthand version of the SHA which is suffient for unique identification

```
ea7aad1 (HEAD -> main) test
```

- Almost all git commands accept a wide range of options. You can explore them using `git your-command-name --help`. Some explanations are not the best though and you might be quicker by asking your LLM of choice.

Git behind the curtain

wait, which curtain?

- git has a command for *virtually everything*

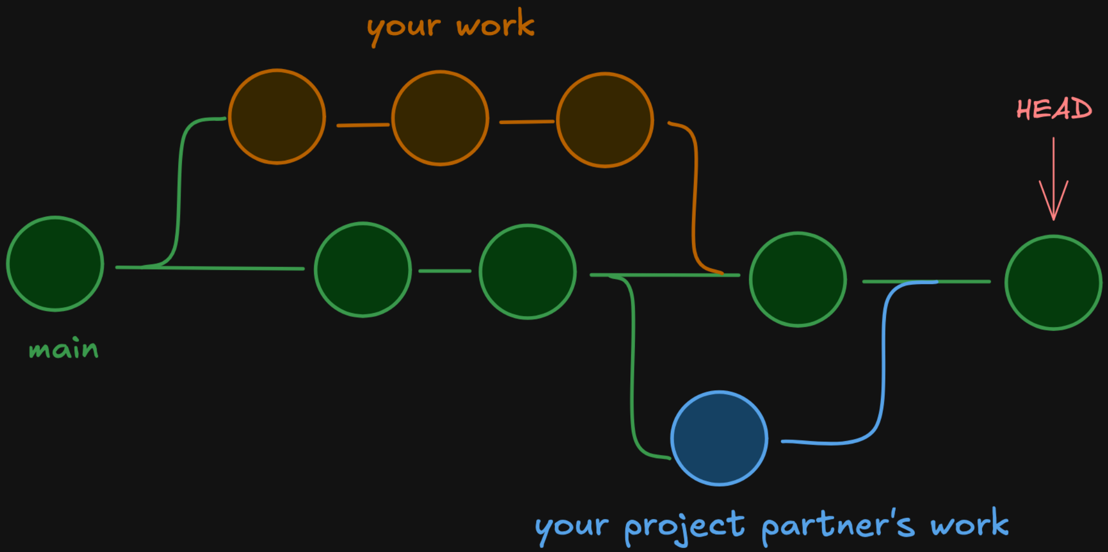
```
git
> zsh: do you wish to see all 142 possibilities (142 lines)?
```

- git has very few restrictions on what you can and can't do (which can sometimes be dangerous)
- you can explore all sorts of git's internal workings by taking a close look inside the hidden `.git` directory of your project and even view the full source code.

Branches

Why?

- Better code organization
- Minimizes friction during collaboration



Branches

How?

- Create a new branch

```
git branch foo
```

- You have 2 different options of changing your current branch to the new one

```
git checkout foo
git switch foo
> Switched to branch 'foo'
```

- View all your local branches with `git branch`

```
* foo
  main
```

The * marks your current branch

Branches

What?

Let’s create another file and commit it to our new branch

```
touch another.md && git add . && git commit -m "another commit
```

Now, what happened?

```
git log --oneline
> d4ee3ee (HEAD -> foo) 2nd commit
> ea7aad1 (main) test
```

If we switch back to *main*, we can’t see the new commit:

```
git checkout main
> Switched to branch 'main'
git log --oneline
> ea7aad1 (HEAD -> main) test
```

We can make changes to *main* branch as if our change on *foo* never existed!

Exercise

~ 5 minutes

1. Create a new branch and switch to it
2. Verify you are on the branch
3. Make two commits on the new branch
4. Verify that your new branch is two commits ahead of *main*

Something like this should be the result:

```
62ad496 (HEAD -> foo) 3rd commit
de59c28 2nd commit
2326e5e (main) test
```

We can see that *HEAD* points to our new branch *foo* while *main* is two commits behind.

Merge

Let’s say, in the meantime, our colleague worked on a different feature and commited the code to *main*.

```
git checkout main
touch new.md
git add .
git commit -m "4th commit"
git log --oneline
```

This is the *main* branch now:

```
fdadab0 (HEAD -> main) 4th commit
2326e5e test
```

How can we get our work from *foo* to *main*?

Merge

How can we get our work from *foo* to *main*?

By **merging** the **source** branch (*foo*) into the **target** (*main*) branch:

```
git merge foo
```

The branch you’re on is the *target* branch and the branch you’re specifying in `git merge <branchname>` is your *source* branch.

Upon merging, you should choose an informative commit message:

```
Feat: This is a great feature

# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
```

When working in larger codebases, there is usually some naming convention for merge commits (e.g. feat, fix, chore, …).

Merge

What happened? Let’s check with `git log --graph`

```
*   commit e15bddaf5515a5b89c04a1e9085718e8a81b0b4e (HEAD -> main)
|\  Merge: fdadab0 62ad496
| |
| |     Feat: This is a great feature
| |
| * commit 62ad496a99fd8c6be44cbe28e21747c85b8b54ae (foo)
| |
| |     3rd commit msg
| |
| * commit de59c2813ebe259e7d1fa91ccd83228fbd0e52c5
| |
| |     2nd commit msg
| |
* | commit fdadab02a0a1b9a282a8fd30f97221434a999c5d
|/
|       4th commit msg
|
* commit 2326e5eed074c32710ff3ff7d785b17eee2ee4f1

      1st commit msg
```

A merge is attempting to combine two histories together that have diverged at some point in the past (here: after the first commit).

git then merges the sets of commits onto the common ancestor (merge base) and creates a new commit (merge commit) at the tip of the branch that is being merged on with all the changes combined into one commit.

During this, a merge conflict can occur. More on that later.

Reflog

Lost your *HEAD* and trying to find it?

Using `git reflog` you can see where your `HEAD` has been previously. Each point in time (i.e. a checkout, commit, etc.) has it's own unique SHA.

```
e15bdda (HEAD -> main) HEAD@{0}: merge foo: Merge made by the 'ort' strategy.
fdadab0 HEAD@{1}: merge foo: updating HEAD
fdadab0 HEAD@{2}: commit: 4th commit
2326e5e HEAD@{3}: checkout: moving from foo to main
62ad496 (foo) HEAD@{4}: commit: 3rd commit
...
```

Let's create a new branch, commit something to it and delete the branch:

```
git checkout -b new-to-be-deleted
touch new-to-be-deleted.md && git add . && git commit -m "lost commit"
git checkout main
git branch -D new-to-be-deleted
```

Can we bring back our lost commit?

Cherry pick

Bringing back our lost commit

Let's check where our *HEAD* has been with `git reflog`

```
git reflog
> e15bdda (HEAD -> main) HEAD@{0}: checkout: moving from new-to-be-deleted to main
> 5f5bcec HEAD@{1}: commit: lost commit
> e15bdda (HEAD -> main) HEAD@{2}: checkout: moving from main to new-to-be-deleted
```

Our lost commit, along with it's SHA is still available! Using `git cherry-pick <SHA>` we can bring it back to the branch that we are currently on (if history didn't diverge too much):

```
git cherry-pick 5f5bcec
git log --oneline
> d589cc2 (HEAD -> main) lost commit
> e15bdda Feat: This is a great feature
> fdadab0 4th commit
> 62ad496 (foo) 3rd commit
```

Great! Our *HEAD* now points to the commit that we've thought to be lost.

Remember: You can learn more about `cherry-pick` just like with any other command using `git cherry-pick --help`

Reset

Time travel has never been so easy

Using `git reset <SHA>` you can reset your *HEAD* to any previous point SHA from `log` or `reflog`.

Say, you decide that you don’t want to keep the commit that we just recovered. You can just move back to the commit before it!

```
git log --oneline
> d589cc2 (HEAD → main) lost commit
> e15bdda Feat: This is a great feature
 ...
```

```
git reset e15bdda
> HEAD is now at e15bdda Feat: This is a great feature
```

But wait… Why are there uncommited files now?

```
git status
> On branch main
> new file:   new-to-be-deleted.md
```

Reset

Why are there uncommited files now?

```
git status
> On branch main
> new file:   new-to-be-deleted.md
```

This is the exact file which we added via the commit that we just got rid of!

The file is still there, because `git reset` does a *soft* reset per default (= `git reset --soft`). This keeps the files from the removed commits in our staging area. It's a safety feature that prevents us from losing our work unintentionally.

However, in this case, we want to remove the commit **along** with all it's changes. We can achieve this with the `--hard` flag. Use with caution!

```
git reset --hard e15bdda
git status
> On branch main
> nothing to commit, working tree clean
```

Reset

a word of caution

This commands does something that is commonly called rewriting history.

Only rewrite history on **your own branches**! These include branches that are one of the following:

- Not published yet (= you did not use `git push` yet)
- Others will definitely not base their work on top of your branch (e.g. a feature-branch that **only you** are working on)

Exercise

~ 10 minutes

1. Merge *foo* into *main*
2. Explore the git log and highlight the different branches
3. Make some changes to main and commit them
4. Delete the commit you've made
5. Verify that *HEAD* now points to your merge commit again
6. "Undo" the deletion and bring back the commit
7. Verify that *HEAD* now points to the brought-back commit
8. Bonus: Explore different options for the new commands you just learned

Remote

Let's start collaborating!

Remote just means that there is a copy of your local git repo accessible *somewhere* else! Typically, remotes are hosted on GitHub, GitLab, Bitbucket and so on. However, there are plenty of other options and you certainly don't need any of these big players. You can even host your own git server if you like.

For the sake of simplicity, we will use GitHub for this course. If you've followed the requirements, you should already have set it up.

1. Create an **empty** repo on GitHub
2. Add the remote to your repo as a "source of truth". A common naming convention for such remote repos is *origin*:

   ```
   git remote add origin git@github.com:<YOUR-USER-NAME>/<YOUR-REPO-NAME>.git
   ```

3. Push your local changes (for once) on main upstream:

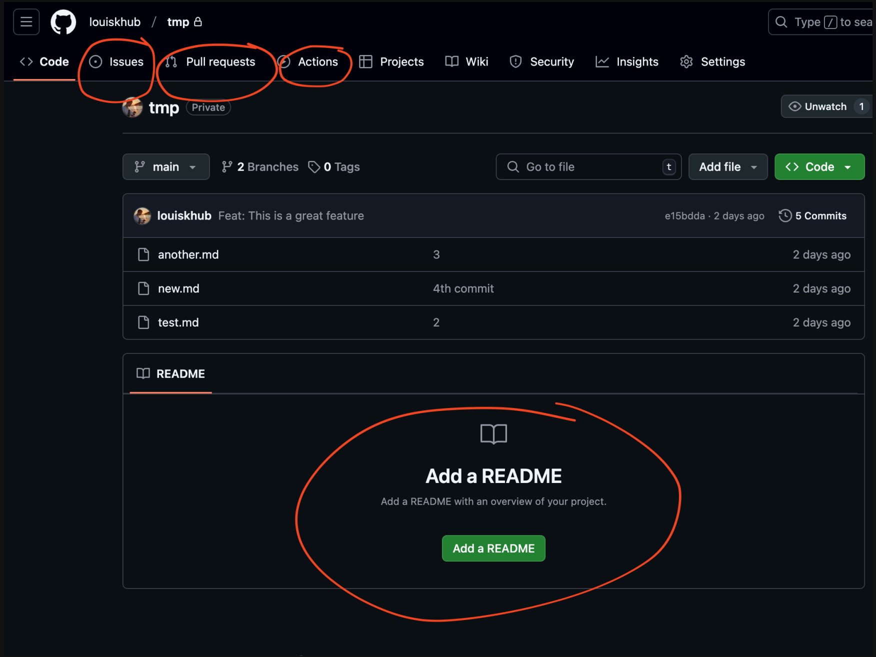
   ```
   git push -u origin main
   ```

4. Now you should be able to see your files and commits in the browser!

GitHub

has some neat features

The cool thing about using GitHub (or any other established hosting service) is that it comes with a lot of features in addition to version control.



louiskhub Feat: This is a great feature e15bdda · 2 days ago 5 Commits

another.md	3	2 days ago
new.md	4th commit	2 days ago
test.md	2	2 days ago

Add a README

Add a README with an overview of your project.

Add a README

- We can write issues to track ideas, feedback, tasks, or bugs
- We can create pull requests for our contributions
- We can automate tasks with actions
- Small hints, etc. (e.g. we are immediately reminded to upload a README)

Exercise

~ 5 min

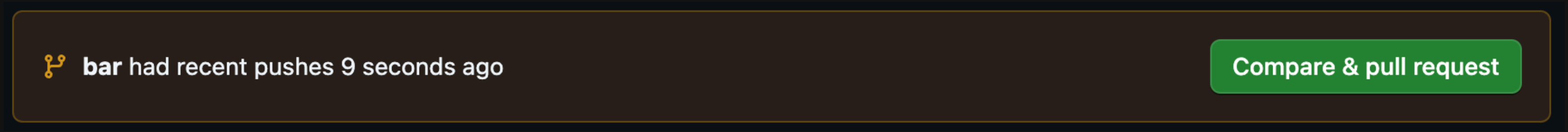
Create a new branch along with 3-4 commits.

1. In the first commit you add some text to your existing files
2. In the second commit, you delete some of the text you just added.
3. In the third commit, you add a README.md file with some content (can be any markdown text) In the first commit. In the other commits, you can add/delete more text and files.

After this, push the branch upstream. Hint: you first have to link an upstream branch to your local branch in order to do this:

```
git push --set-upstream origin "<YOUR-BRANCH-NAME>"
```

Now, you should see a notification on GitHub which tells you that there has been a push to your new branch.



bar had recent pushes 9 seconds ago

Rewriting history on a public branch

and how git tries to protect you

Say, after we already pushed our changes upstream (i.e. made our branch public), we decide to remove a commit. As mentioned previously, this can be done via `git reset <SHA>` but **it rewrites history**.

Git implements some guardrails that prevent you from accidentally pushing a diverged git history and potentially overwriting someones work:

```
git push -u origin bar
> ! [rejected]        bar -> bar (non-fast-forward)
> error: failed to push some refs to 'github.com:louiskhub/tmp.git'
> hint: Updates were rejected because the tip of your current branch is behind
> hint: its remote counterpart. Integrate the remote changes (e.g.
> hint: 'git pull ...') before pushing again.
```

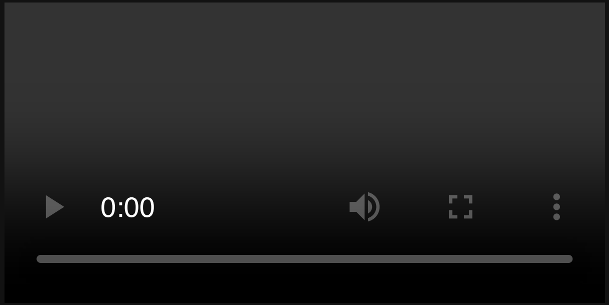
If you're absolutely sure no one will use your public branch until it's merged into *main*, you can force git to push your changes with the `--force-with-lease` flag:

```
git push -u origin bar --force-with-lease
```

Exercise: Issue & Pull Request

~ 4 min

1. Create a new issue where we specify some arbitrary requirements for a feature
2. Create a pull request (PR) from your new branch to *main*
3. Link the issue in the PR description



Code review

Why it is important (with descending priority)

1. **Distributing code ownership**

   The reviewer has to answer the question: "If the person who wrote this code leaves the team, would you be okay with maintaining the code?".

2. **Ensure proper documentation and minimal complexity**

   If the reviewer has a really hard time understanding the code, it's probably too complex and/or required (more) documentation.

3. **Test if everything works as expected**

   Many larger companies have dedicated QA teams and testing is not the reviewer's responsibility. Also, if there is good automated test coverage, manual testing is quick and straightforward. However, both is usually not the case in startups!

4. **Find potential vulnerabilities and bugs**

   It's really hard to catch subtle bugs while trying to keep review-time managable. Hence, this is not the main benefit of review.

Code review

How to do it properly

1. Get the author to self-review the code before you even look at the PR
2. Keep **the reasons** for code review in mind
3. Read the PR description
4. Read corresponding issue/ticket and check if basic requirements are met by the PR
5. Manually test the feature (if there is no dedicated QA) for bugs
6. Step through the commit history to understand the changes on a **high level**
7. Check out the code locally and use your editor to deep dive into sections that deserve more attention
8. Ask questions and write comments about **stuff that matters** (no nits)

The lie of small PRs

and what you can do to cope

Small PRs are much easier to review. Large PRs introduce a kind of *code review fatigue*.

Try to keep PRs small and compact.

Often, this is impossible! In those cases, make sure the PRs changes are within a well-defined **context**!

Additionally, having an atomic commit history helps a lot with review.

*Code reviews at a startup
Senior dev be like
+27 −4
*Efficiency improvements tips
*pls refactor
*rename the function to something more appropriate
+2735 −424

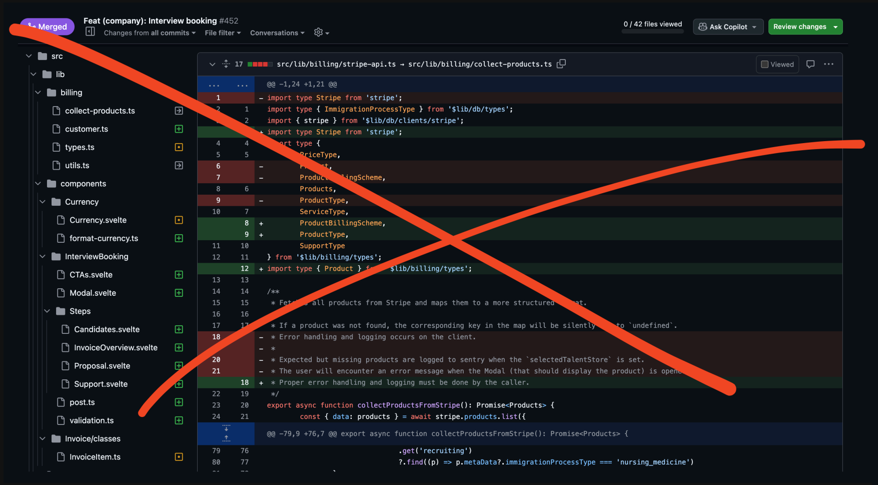
r/ProgrammerHumor

Atomic commit history

makes a code review so much easier

> = series of commits where each commit represents a single, complete, and independent change that maintains a working codebase.

This makes it much easier for the reviewer to step through small changes once at a time rather than everything at once!

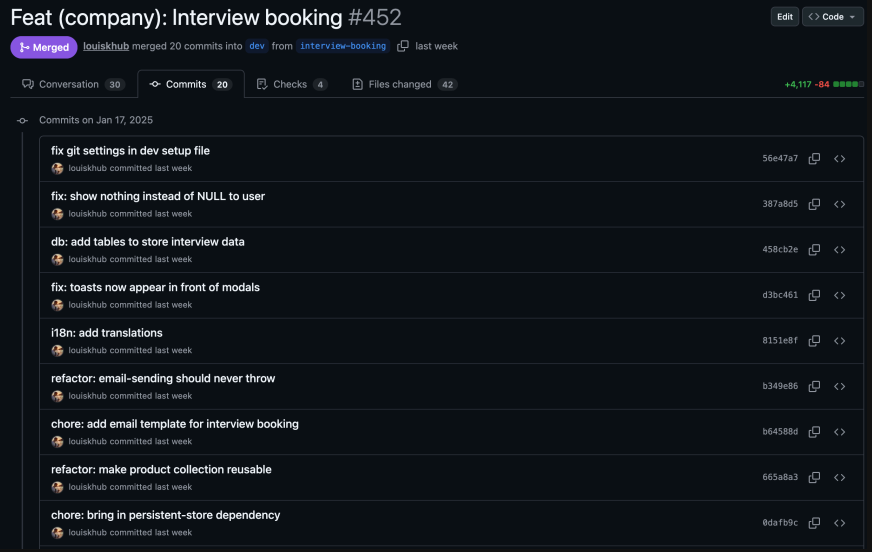


17 src/lib/billing/stripe-api.ts → src/lib/billing/collect-products.ts

```
@@ -1,24 +1,21 @@
- import type Stripe from 'stripe';
  import type { ImmigrationProcessType } from '$lib/db/types';
  import { stripe } from '$lib/db/clients/stripe';
+ import type Stripe from 'stripe';
  import type {
          PriceType,
-         Product,
-         ProductBillingScheme,
          Products,
-         ProductType,
          ServiceType,
+         ProductBillingScheme,
+         ProductType,
          SupportType
  } from '$lib/billing/types';
+ import type { Product } from '$lib/billing/types';

  /**
   * Fetches all products from Stripe and maps them to a more structured format.

   * If a product was not found, the corresponding key in the map will be silently [illegible] to `undefined`.
-  * Error handling and logging occurs on the client.
-  *
-  * Expected but missing products are logged to sentry when the `selectedTalentStore` is set.
-  * The user will encounter an error message when the Modal (that should display the product) is opened.
+  * Proper error handling and logging must be done by the caller.
   */
  export async function collectProductsFromStripe(): Promise<Products> {
          const { data: products } = await stripe.products.list({
@@ -79,9 +76,7 @@ export async function collectProductsFromStripe(): Promise<Products> {
                                  .get('recruiting')
                                  ?.find((p) => p.metaData?.immigrationProcessType === 'nursing_medicine')
```



Feat (company): Interview booking #452

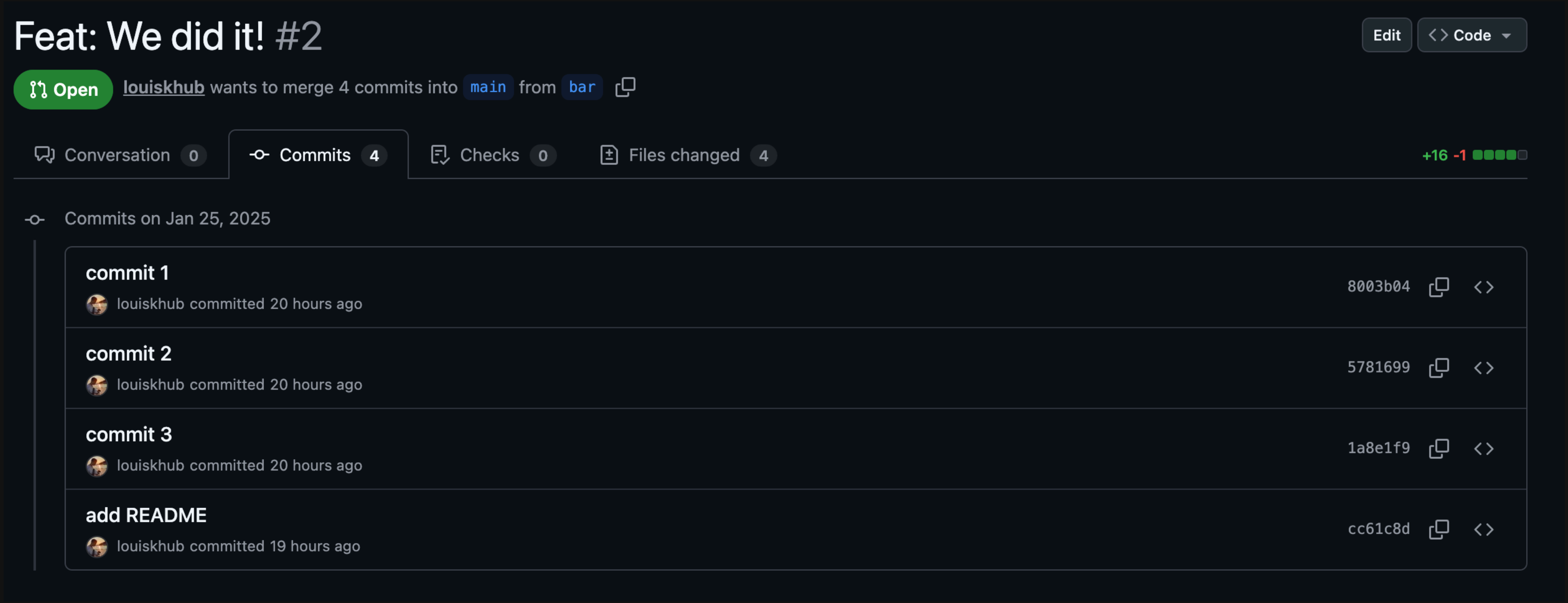
louiskhub merged 20 commits into dev from interview-booking last week

Commits on Jan 17, 2025

- fix git settings in dev setup file — louiskhub committed last week — 56e47a7
- fix: show nothing instead of NULL to user — louiskhub committed last week — 387a8d5
- db: add tables to store interview data — louiskhub committed last week — 458cb2e
- fix: toasts now appear in front of modals — louiskhub committed last week — d3bc461
- i18n: add translations — louiskhub committed last week — 8151e8f
- refactor: email-sending should never throw — louiskhub committed last week — b349e86
- chore: add email template for interview booking — louiskhub committed last week — b64588d
- refactor: make product collection reusable — louiskhub committed last week — 665a8a3
- chore: bring in persistent-store dependency — louiskhub committed last week — 0dafb9c

Question

What could we improve about our PRs commits to make them easier to review?



Feat: We did it! #2

louiskhub wants to merge 4 commits into main from bar

Commits on Jan 25, 2025

commit 1
louiskhub committed 20 hours ago
8003b04

commit 2
louiskhub committed 20 hours ago
5781699

commit 3
louiskhub committed 20 hours ago
1a8e1f9

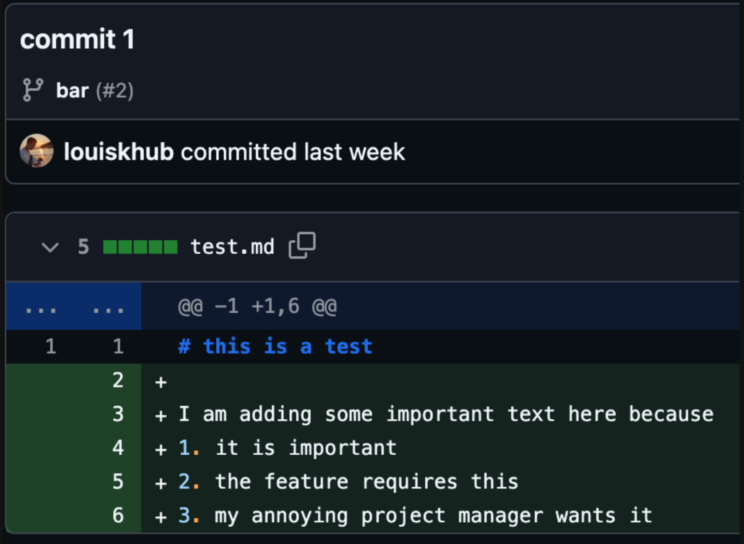
add README
louiskhub committed 19 hours ago
cc61c8d

Answer

What could we improve about our PRs commits to make them easier to review?

1. Use descriptive commit messages (maybe even with a standardized naming convention)
2. Don't make changes in some commits that you undo in a later commit!

E.g. in *commit 1* we add a line in the `test.md` file that we remove in *commit 2*:



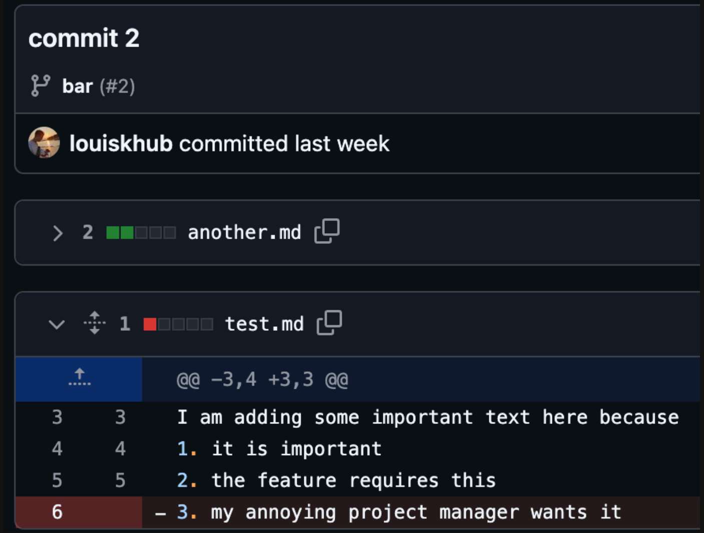
commit 1

bar (#2)

louiskhub committed last week

5 🟩🟩🟩🟩🟩 test.md

```
...  ...   @@ -1 +1,6 @@
 1    1    # this is a test
      2  +
      3  + I am adding some important text here because
      4  + 1. it is important
      5  + 2. the feature requires this
      6  + 3. my annoying project manager wants it
```



commit 2

bar (#2)

louiskhub committed last week

2 🟩🟩□□□ another.md

1 🟥□□□□ test.md

```
           @@ -3,4 +3,3 @@
 3    3    I am adding some important text here because
 4    4    1. it is important
 5    5    2. the feature requires this
 6       - 3. my annoying project manager wants it
```

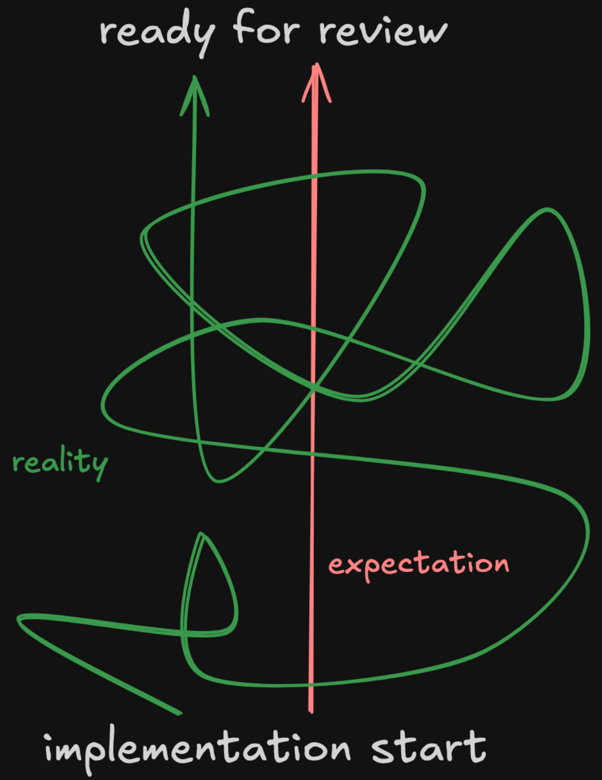
A workflow for getting a nice commit history

The reviewer expectation differs from reality during implementation

Problem: During implementation, it's hard to maintain a nice commit history (especially if there's a lot of *trial and error*)

Solution: Random commits just to backup your work. No one, except you, needs to understand them. You can also push these to an unused branch of your *remote*.

Implementation finished?
→ Use `git reset --soft` with the `SHA` of the last decent commit that you want the reviewer to see (usually the last commit of the branch where you've based your new branch on).



A workflow for getting a nice commit history

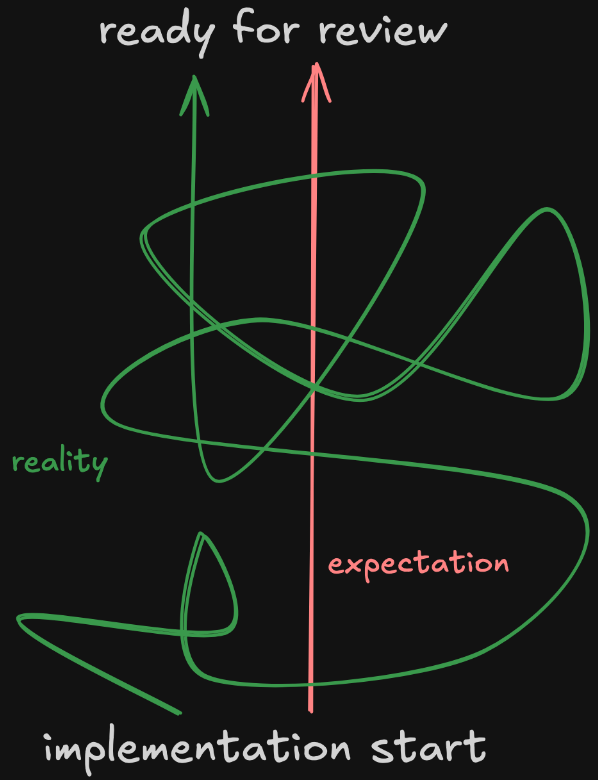
Implementation is finished, now what?

1. Use `git reset --soft` with the `SHA` of the last decent commit that you want the reviewer to see (usually the last commit of the branch where you've based your new branch on).

2. Group your changes into atomic commits.

   - Think about the changes you've made and how to group them into different contexts
   - Combine this with self-review of your code and check for unnecessary complexity

3. Use `git push --force-with-lease` to rewrite the history on **your** remote branch



A nice tool for (self)-review

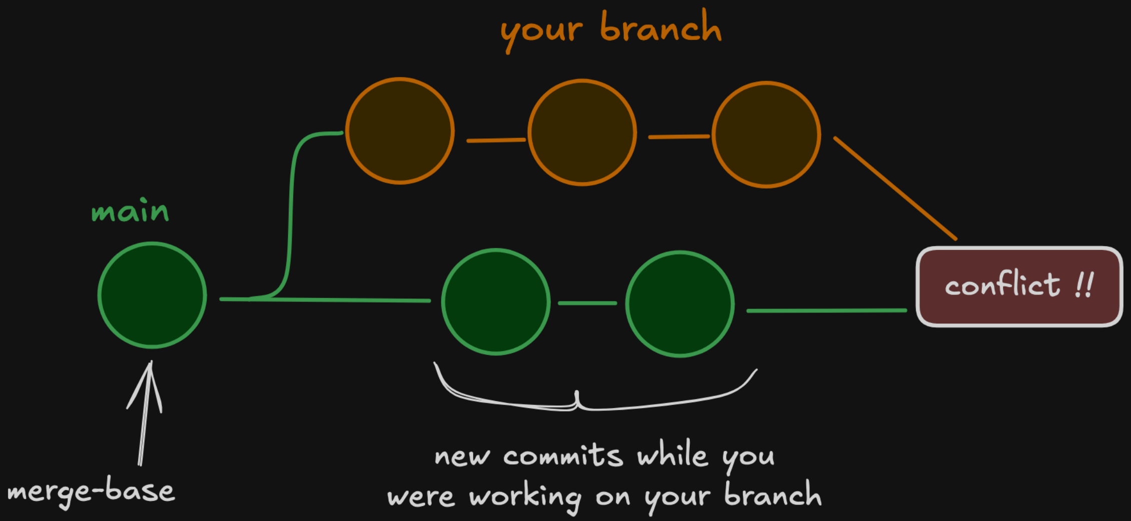
Or how to get a nice diff

Viewing and commit all your changes via the CLI (`git diff`) can be hard. Your IDE likely has a diff view that makes it easier to compare large amounts of changes. Below is an example from VSCode.

Resolving conflicts

`main` changed, now what?

Previously, merge conflicts were briefly mentioned. This happens a lot if you work with multiple devs on one project and it's *essential* to know how to resolve them.

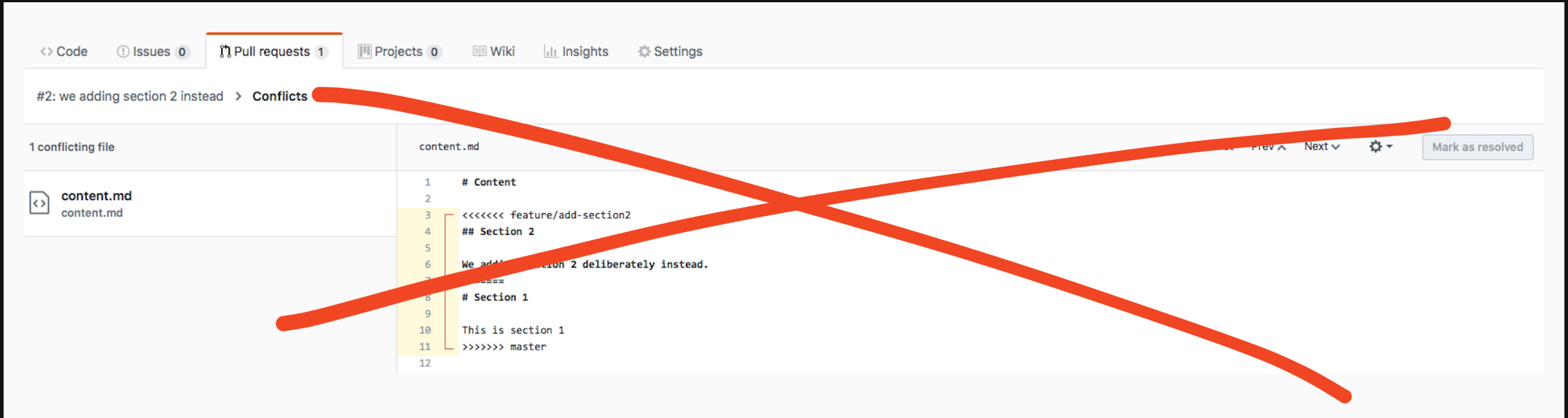


The new commits on *main* included some changes that conflicted with the work you did on your branch.

Resolving conflicts

Tips and tricks

Some remotes like GitHub enable you to resolve conflicts directly in the browser.



```
# Content

<<<<<<< feature/add-section2
## Section 2

We added section 2 deliberately instead.
=======
# Section 1

This is section 1
>>>>>>> master

```

Avoid this! You generally have much less control over what you are doing and it’s easier to make mistakes.

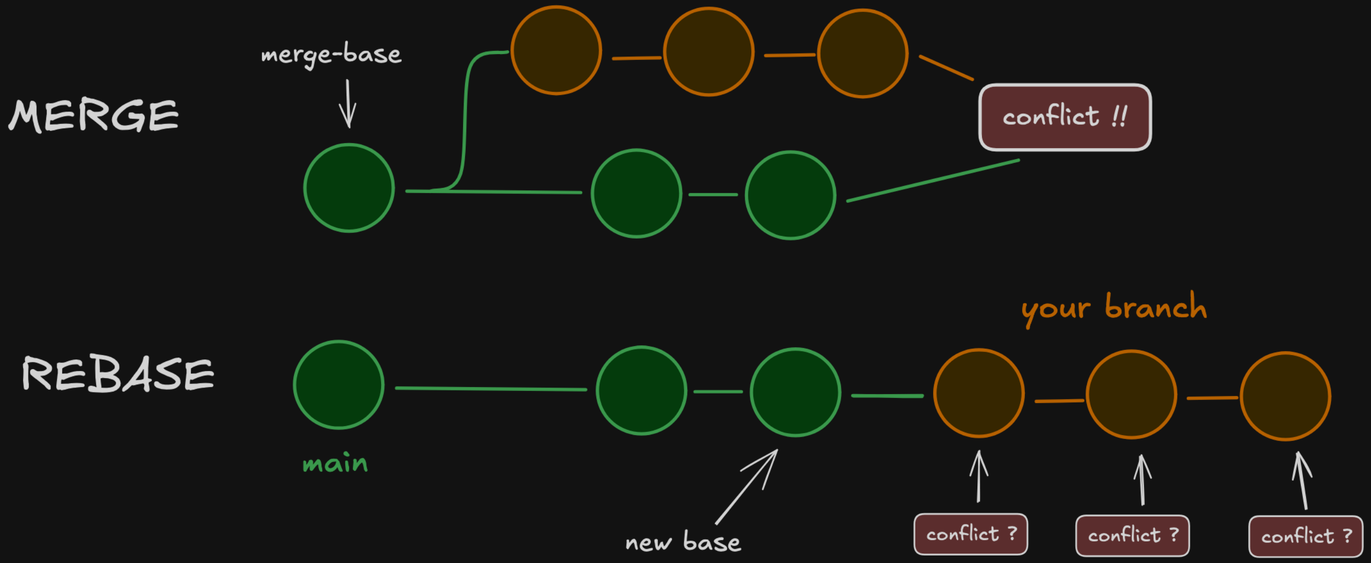
Instead, do it locally. First, we have to fetch all new changes from the remote with `git fetch origin`.

Resolving conflicts

Tips and tricks

After we fetched all changes, we can either use `git merge <branch>` or `git rebase <branch>`. Here, `<branch>` is the name of the branch you want to integrate the changes from.

What is the difference between merging and rebasing?



Rebasing can resolve your PR’s conflicts

This creates a *linear* history and may force you to rewrite some commits

When should you prefer `rebase` over `merge`?

Rebasing *rewrites history* (if you have a merge conflict that needs to be resolved). As mentioned previously, this should never be done on branches that are used by other devs!
This is also the reason why rebasing is considered "dangerous" and avoided by some devs.

However, on **your own** feature branch, that is about to be merged as a PR, rebasing can be a really nice option because there won’t be a merge commit and our nice atomic commit history is preserved!

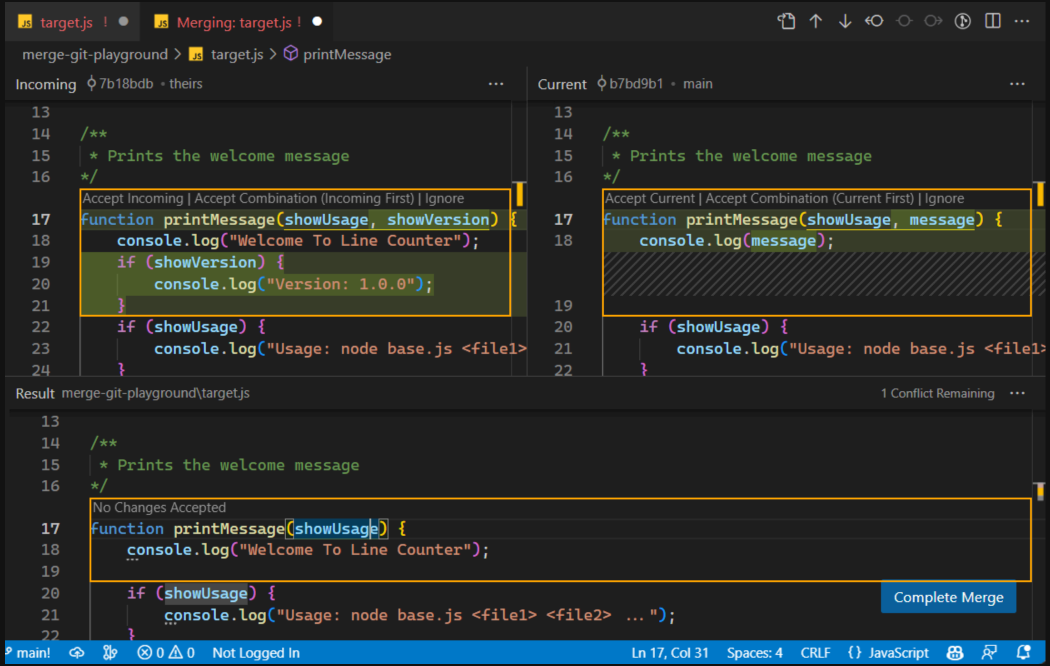
For large, complex PR’s with multiple conflicts, a merge commit can get messy and will be hard to understand in the future. Rebasing forces you to adapt the individual commits that are affected by the conflicts.

This might mean more work for you but could save your mates and future-self a lot of time!

A nice tool for resolving conflicts

is again (drum roll): Your IDE!

Your IDE likely has some built in tooling to visualize and resolve conflicts much more easily than in the CLI. Below is an example from VSCode.


target.js !
Merging: target.js !
merge-git-playground > target.js > printMessage
Incoming 7b18bdb • theirs
Current b7bd9b1 • main
Accept Incoming | Accept Combination (Incoming First) | Ignore
Accept Current | Accept Combination (Current First) | Ignore
Result merge-git-playground\target.js
1 Conflict Remaining
No Changes Accepted
Complete Merge
main!
⊗ 0 ⚠ 0
Not Logged In
Ln 17, Col 31
Spaces: 4
CRLF
{} JavaScript

```
/**
 * Prints the welcome message
*/
function printMessage(showUsage, showVersion) {
    console.log("Welcome To Line Counter");
    if (showVersion) {
        console.log("Version: 1.0.0");
    }
    if (showUsage) {
        console.log("Usage: node base.js <file1>
    }
```

```
/**
 * Prints the welcome message
*/
function printMessage(showUsage, message) {
    console.log(message);

    if (showUsage) {
        console.log("Usage: node base.js <file1>
    }
```

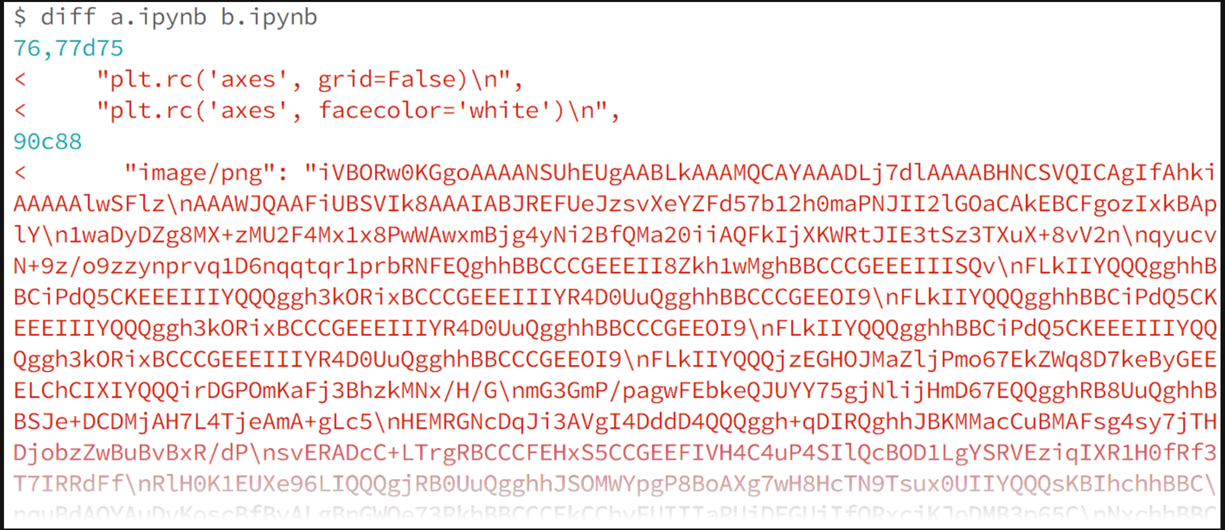
```
/**
 * Prints the welcome message
*/
function printMessage(showUsage) {
    console.log("Welcome To Line Counter");

    if (showUsage) {
        console.log("Usage: node base.js <file1> <file2> ...");
    }
```

Going full circle

Why Jupyter Notebooks suck for production

They yield gruesome `diff`'s. Good luck trying to resolve conflicts!



```
$ diff a.ipynb b.ipynb
76,77d75
<     "plt.rc('axes', grid=False)\n",
<     "plt.rc('axes', facecolor='white')\n",
90c88
<       "image/png": "iVBORw0KGgoAAAANSUhEUgAABLkAAAMQCAYAAADLj7dlAAAABHNCSVQICAgIfAhki
AAAAAlwSFlz\nAAAWJQAAFiUBSVIk8AAAIABJREFUeJzsvXeYZFd57b12h0maPNJII2lGOaCAkEBCFgozIxkBAp
lY\n1waDyDZg8MX+zMU2F4Mx1x8PwWAwxmBjg4yNi2BfQMa20iiAQFkIjXKWRtJIE3tSz3TXuX+8vV2n\nqyucv
N+9z/o9zzynprvq1D6nqqtqr1prbRNFEQghhBBCCCGEEEII8Zkh1wMghBBCCCGEEEIIISQv\nFLkIIYQQQgghhB
BCiPdQ5CKEEEIIIYQQQggh3kORixBCCCGEEEIIIYR4D0UuQgghBBCCCGEEOI9\nFLkIIYQQQgghhBBCiPdQ5CK
EEEIIIYQQQggh3kORixBCCCGEEEIIIYR4D0UuQgghhBBCCCGEEOI9\nFLkIIYQQQgghhBBCiPdQ5CKEEEIIIYQQ
Qggh3kORixBCCCGEEEIIIYR4D0UuQgghhBBCCCGEEOI9\nFLkIIYQQQjzEGHOJMaZljPmo67EkZWq8D7keBGyGEE
ELChCIXIYQQQirDGPOmKaFj3BhzkMNx/H/G\nmG3GmP/pagwFEbkeQJUYY75gjNlHmD67EQQgghRB8UuQghhB
BSJe+DCDMjAH7L4TjeAmA+gLc5\nHEMRGNcDqJi3AVgI4DddD4QQQgh+qDIRQghhJBKMMacCuBMAFsg4sy7jTH
DjobzZwBuBvBxR/dP\nsvERADcC+LTrgRBCCCFEHxS5CCGEEFIVH4C4uP4SIlQcBOD1LgYSRVEziqIXR1H0fRf3
T7IRRdFf\nRlH0K1EUXe96LIQQQgjRB0UuQgghJSOMWYpgP8BoAXg7wH8Hc
nguBdAOYAuDyKoscBfByAl gBnGWOe73RkhBBCCCFkCChyFUTTTaRUiDFGUiTfORxciKJoDMB3p65C\nNxchhBBC
```

To be fair: There are tools (e.g. nbdime) that make *diffing* easier for Jupyter Notebooks. However, these tools also come with their own caveats.

There is a looooot more to learn about git

But that's enough for today

As with most larger tools, one will hardly ever understand every little detail of git.

That's no problem at all.

Just never stop learning and exploring! This is what makes the best developers.

wikipedia