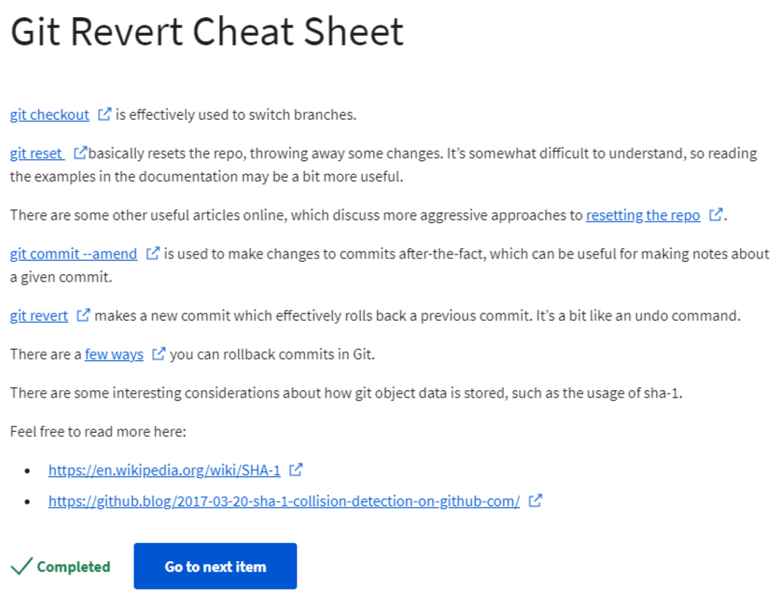
GitHub Cheat codes



Git Revert Cheat Sheet

git checkout is effectively used to switch branches.

git reset basically resets the repo, throwing away some changes. It's somewhat difficult to understand, so reading the examples in the documentation may be a bit more useful.

There are some other useful articles online, which discuss more aggressive approaches to resetting the repo.

git commit --amend is used to make changes to commits after-the-fact, which can be useful for making notes about a given commit.

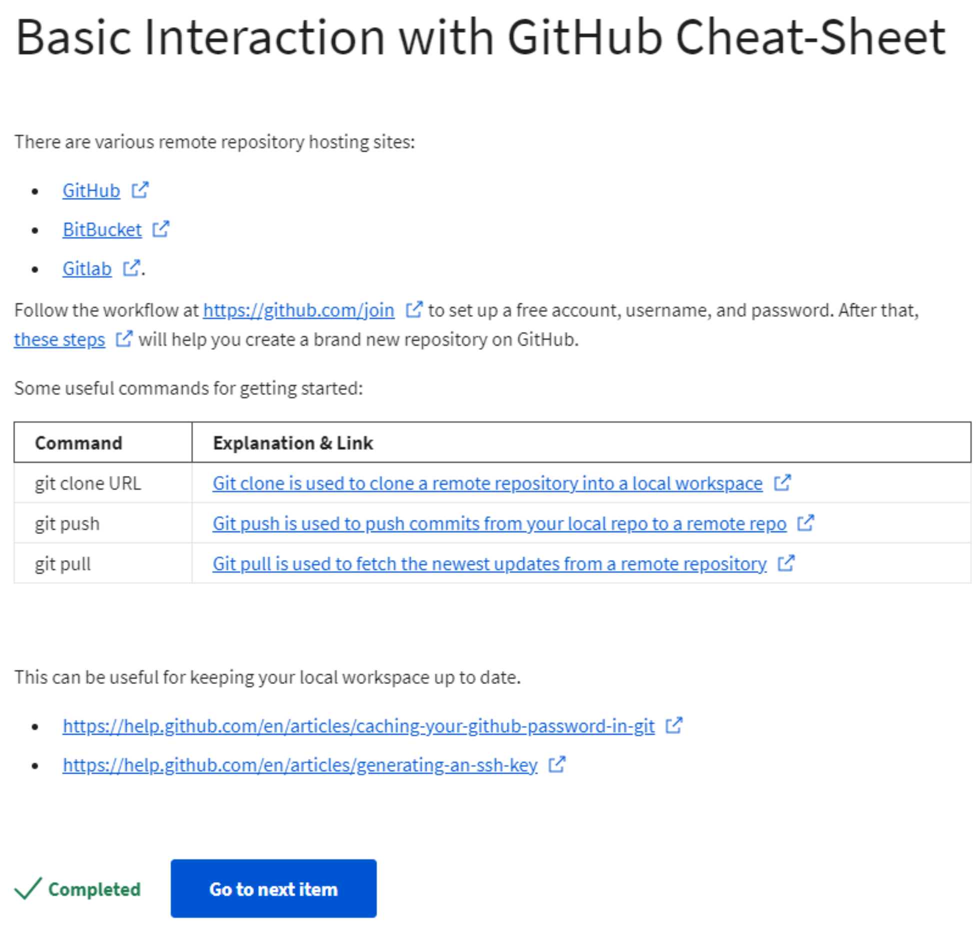
git revert makes a new commit which effectively rolls back a previous commit. It's a bit like an undo command.

There are a few ways you can rollback commits in Git.

There are some interesting considerations about how git object data is stored, such as the usage of sha-1.

Feel free to read more here:

- https://en.wikipedia.org/wiki/SHA-1
- https://github.blog/2017-03-20-sha-1-collision-detection-on-github-com/



Basic Interaction with GitHub Cheat-Sheet

There are various remote repository hosting sites:

- GitHub
- BitBucket
- Gitlab.

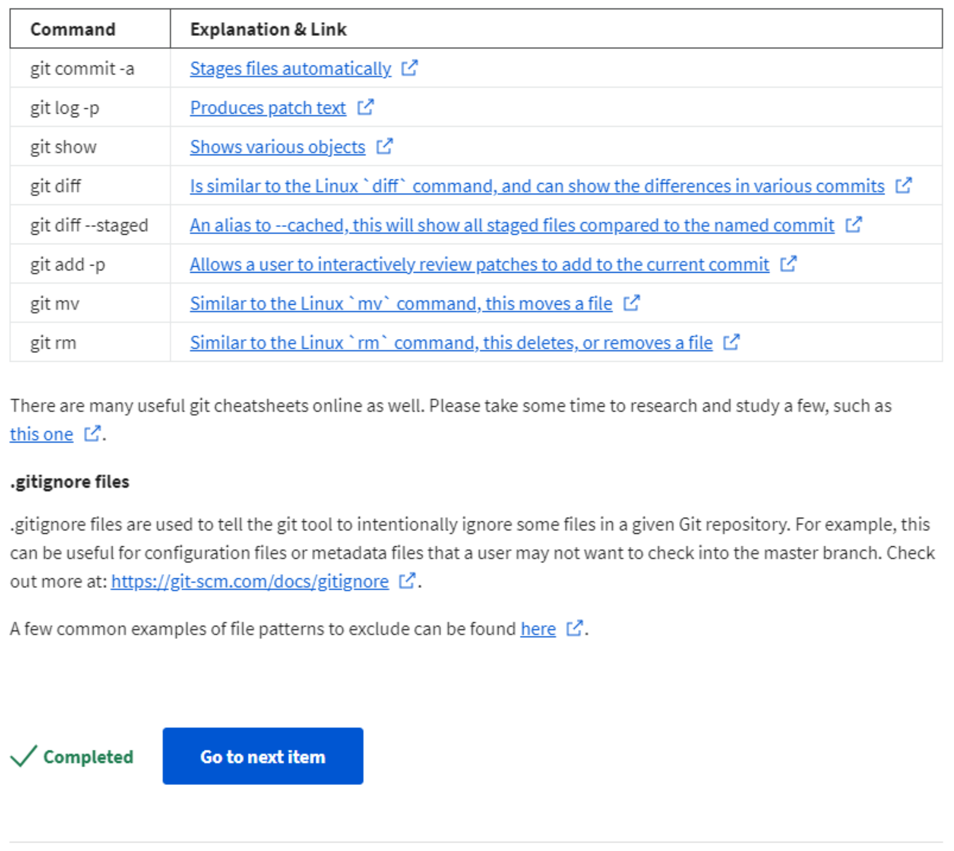
Follow the workflow at https://github.com/join to set up a free account, username, and password. After that, these steps will help you create a brand new repository on GitHub.

Some useful commands for getting started:

Command	Explanation & Link
git clone URL	Git clone is used to clone a remote repository into a local workspace
git push	Git push is used to push commits from your local repo to a remote repo
git pull	Git pull is used to fetch the newest updates from a remote repository

This can be useful for keeping your local workspace up to date.

- https://help.github.com/en/articles/caching-your-github-password-in-git
- https://help.github.com/en/articles/generating-an-ssh-key



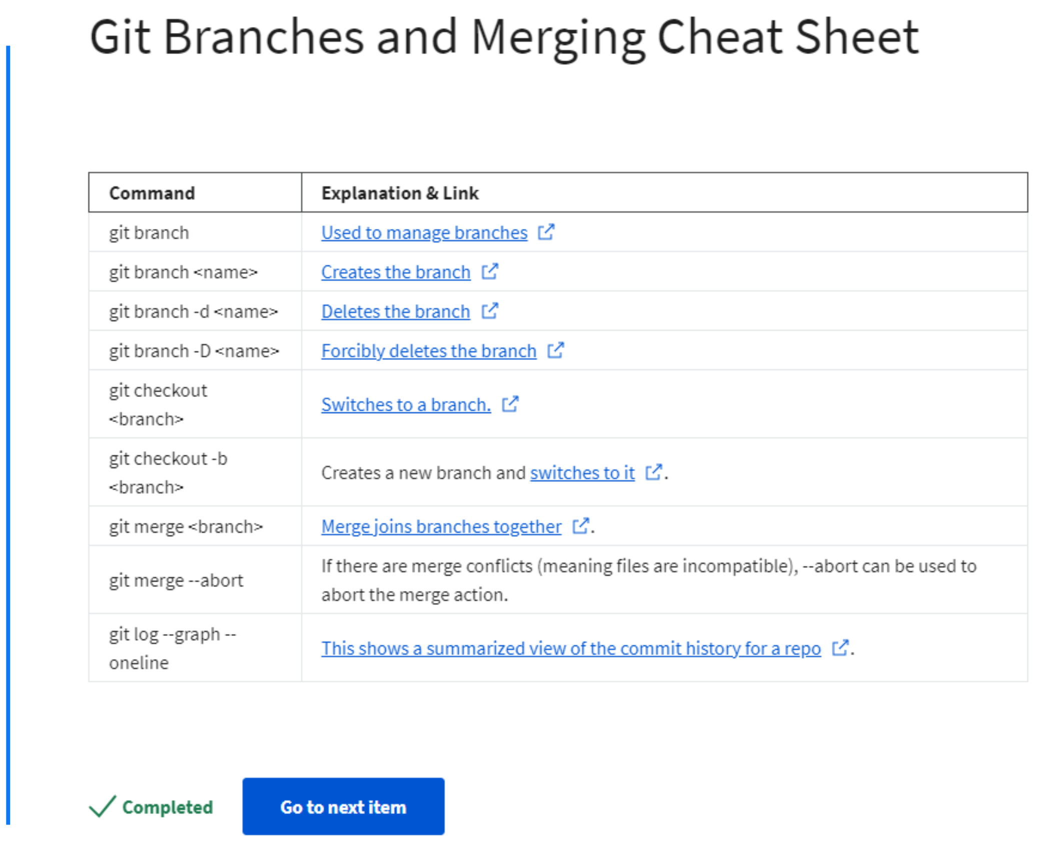
Command	Explanation & Link
git commit -a	Stages files automatically
git log -p	Produces patch text
git show	Shows various objects
git diff	Is similar to the Linux `diff` command, and can show the differences in various commits
git diff --staged	An alias to --cached, this will show all staged files compared to the named commit
git add -p	Allows a user to interactively review patches to add to the current commit
git mv	Similar to the Linux `mv` command, this moves a file
git rm	Similar to the Linux `rm` command, this deletes, or removes a file

There are many useful git cheatsheets online as well. Please take some time to research and study a few, such as this one.

.gitignore files

.gitignore files are used to tell the git tool to intentionally ignore some files in a given Git repository. For example, this can be useful for configuration files or metadata files that a user may not want to check into the master branch. Check out more at: https://git-scm.com/docs/gitignore.

A few common examples of file patterns to exclude can be found here.



Git Branches and Merging Cheat Sheet

Command	Explanation & Link
git branch	Used to manage branches
git branch <name>	Creates the branch
git branch -d <name>	Deletes the branch
git branch -D <name>	Forcibly deletes the branch
git checkout <branch>	Switches to a branch.
git checkout -b <branch>	Creates a new branch and switches to it.
git merge <branch>	Merge joins branches together.
git merge --abort	If there are merge conflicts (meaning files are incompatible), --abort can be used to abort the merge action.
git log --graph --oneline	This shows a summarized view of the commit history for a repo.

Command	Explanation & Links
git remote	Lists remote repos
git remote -v	List remote repos verbosely
git remote show <name>	Describes a single remote repo
git remote update	Fetches the most up-to-date objects
git fetch	Downloads specific objects
git branch -r	Lists remote branches; can be combined with other branch arguments to manage remote branches

You can also see more in the video Cryptography in Action from the course IT Security: Defense against the digital dark arts.

✓ Completed

Go to next item

Like Dislike Report an issue

About merge conflicts

Merge conflicts happen when you merge branches that have competing commits, and Git needs your help to decide which changes to incorporate in the final merge.

Git can often resolve differences between branches and merge them automatically. Usually, the changes are on different lines, or even in different files, which makes the merge simple for computers to understand. However, sometimes there are competing changes that Git can't resolve without your help. Often, merge conflicts happen when people make different changes to the same line of the same file, or when one person edits a file and another person deletes the same file.

You must resolve all merge conflicts before you can merge a pull request on GitHub. If you have a merge conflict between the compare branch and base branch in your pull request, you can view a list of the files with conflicting changes above the **Merge pull request** button. The **Merge pull request** button is deactivated until you've resolved all conflicts between the compare branch and base branch.

Resolving merge conflicts

To resolve a merge conflict, you must manually edit the conflicted file to select the changes that you want to keep in the final merge. There are a couple of different ways to resolve a merge conflict:

- If your merge conflict is caused by competing line changes, such as when people make different changes to the same line of the same file on different branches in your Git repository, you can resolve it on GitHub using the conflict editor. For more information, see "Resolving a merge conflict on GitHub."
- For all other types of merge conflicts, you must resolve the merge conflict in a local clone of the repository and push the change to your branch on GitHub. You can use the command line or a tool like GitHub Desktop to push the change. For more information, see "Resolving a merge conflict using the command line."

If you have a merge conflict on the command line, you cannot push your local changes to GitHub until you resolve the merge conflict locally on your computer. If you try merging branches on the command line that have a merge conflict, you'll get an error message. For more information, see "Resolving a merge conflict using the command line."

```
$ git merge BRANCH-NAME
> Auto-merging styleguide.md
> CONFLICT (content): Merge conflict in styleguide.md
> Automatic merge failed; fix conflicts and then commit the result
```

Further reading

- "About pull request merges"
- "About pull requests"
- "Resolving a merge conflict using the command line"
- "Resolving a merge conflict on GitHub"

Resolving a merge conflict using the command line

You can resolve merge conflicts using the command line and a text editor.

Mac **Windows** Linux

Merge conflicts occur when competing changes are made to the same line of a file, or when one person edits a file and another person deletes the same file. For more information, see "About merge conflicts."

Tip: You can use the conflict editor on GitHub to resolve competing line change merge conflicts between branches that are part of a pull request. For more information, see "Resolving a merge conflict on GitHub."

Competing line change merge conflicts

To resolve a merge conflict caused by competing line changes, you must choose which changes to incorporate from the different branches in a new commit.

For example, if you and another person both edited the file *styleguide.md* on the same lines in different branches of the same Git repository, you'll get a merge conflict error when you try to merge these branches. You must resolve this merge conflict with a new commit before you can merge these branches.

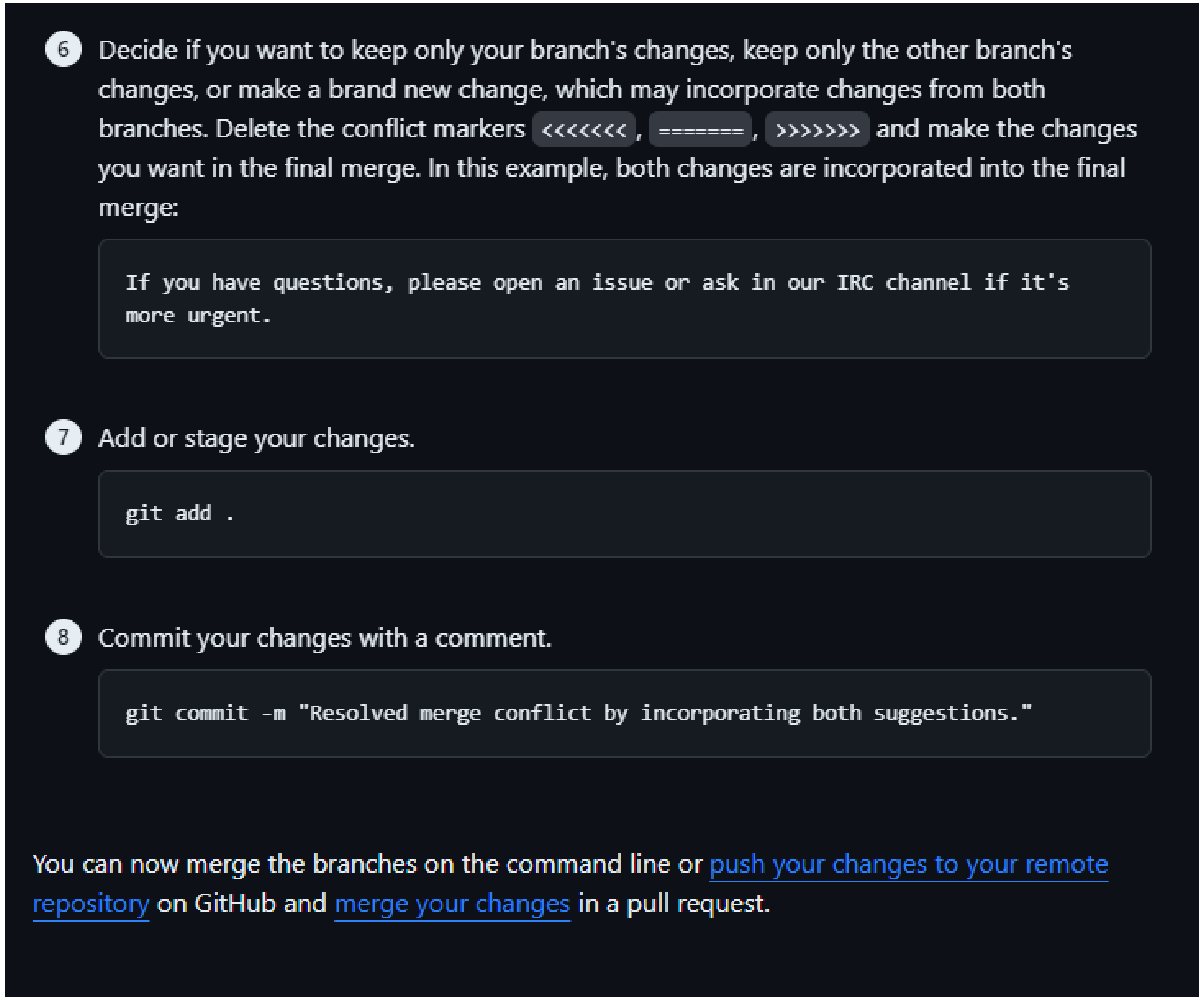
1. Open Git Bash.
2. Navigate into the local Git repository that has the merge conflict.

```
cd REPOSITORY-NAME
```

3. Generate a list of the files affected by the merge conflict. In this example, the file *styleguide.md* has a merge conflict.

```
$ git status
> # On branch branch-b
> # You have unmerged paths.
> #   (fix conflicts and run "git commit")
> #
> # Unmerged paths:
> #   (use "git add <file>..." to mark resolution)
> #
> # both modified:      styleguide.md
> #
> no changes added to commit (use "git add" and/or "git commit -a")
```

4. Open your favorite text editor, such as Visual Studio Code, and navigate to the file that has merge conflicts.
5. To see the beginning of the merge conflict in your file, search the file for the conflict marker `<<<<<<<`. When you open the file in your text editor, you'll see the changes from the HEAD or base branch after the line `<<<<<<< HEAD`. Next, you'll see `=======`, which divides your changes from the changes in the other branch, followed by `>>>>>>> BRANCH-NAME`. In this example, one person wrote "open an issue" in the base or HEAD branch and another person wrote "ask your question in IRC" in the compare branch or `branch-a`.



6. Decide if you want to keep only your branch's changes, keep only the other branch's changes, or make a brand new change, which may incorporate changes from both branches. Delete the conflict markers `<<<<<<<`, `=======`, `>>>>>>>` and make the changes you want in the final merge. In this example, both changes are incorporated into the final merge:

```
If you have questions, please open an issue or ask in our IRC channel if it's
more urgent.
```

7. Add or stage your changes.

```
git add .
```

8. Commit your changes with a comment.

```
git commit -m "Resolved merge conflict by incorporating both suggestions."
```

You can now merge the branches on the command line or push your changes to your remote repository on GitHub and merge your changes in a pull request.

Removed file merge conflicts

To resolve a merge conflict caused by competing changes to a file, where a person deletes a file in one branch and another person edits the same file, you must choose whether to delete or keep the removed file in a new commit.

For example, if you edited a file, such as *README.md*, and another person removed the same file in another branch in the same Git repository, you'll get a merge conflict error when you try to merge these branches. You must resolve this merge conflict with a new commit before you can merge these branches.

1. Open Git Bash.
2. Navigate into the local Git repository that has the merge conflict.

```
cd REPOSITORY-NAME
```

3. Generate a list of the files affected by the merge conflict. In this example, the file *README.md* has a merge conflict.

```
$ git status
> # On branch main
> # Your branch and 'origin/main' have diverged,
> # and have 1 and 2 different commits each, respectively.
> #  (use "git pull" to merge the remote branch into yours)
> # You have unmerged paths.
> #  (fix conflicts and run "git commit")
> #
> # Unmerged paths:
> #  (use "git add/rm <file>..." as appropriate to mark resolution)
> #
> #     deleted by us:   README.md
> #
> # no changes added to commit (use "git add" and/or "git commit -a")
```

4. Open your favorite text editor, such as Visual Studio Code, and navigate to the file that has merge conflicts.
5. Decide if you want to keep the removed file. You may want to view the latest changes made to the removed file in your text editor.

   To add the removed file back to your repository:

```
git add README.md
```

   To remove this file from your repository:

```
$ git rm README.md
> README.md: needs merge
> rm 'README.md'
```

6. Commit your changes with a comment.

```
$ git commit -m "Resolved merge conflict by keeping README.md file."
> [branch-d 6f89e49] Merge branch 'branch-c' into branch-d
```

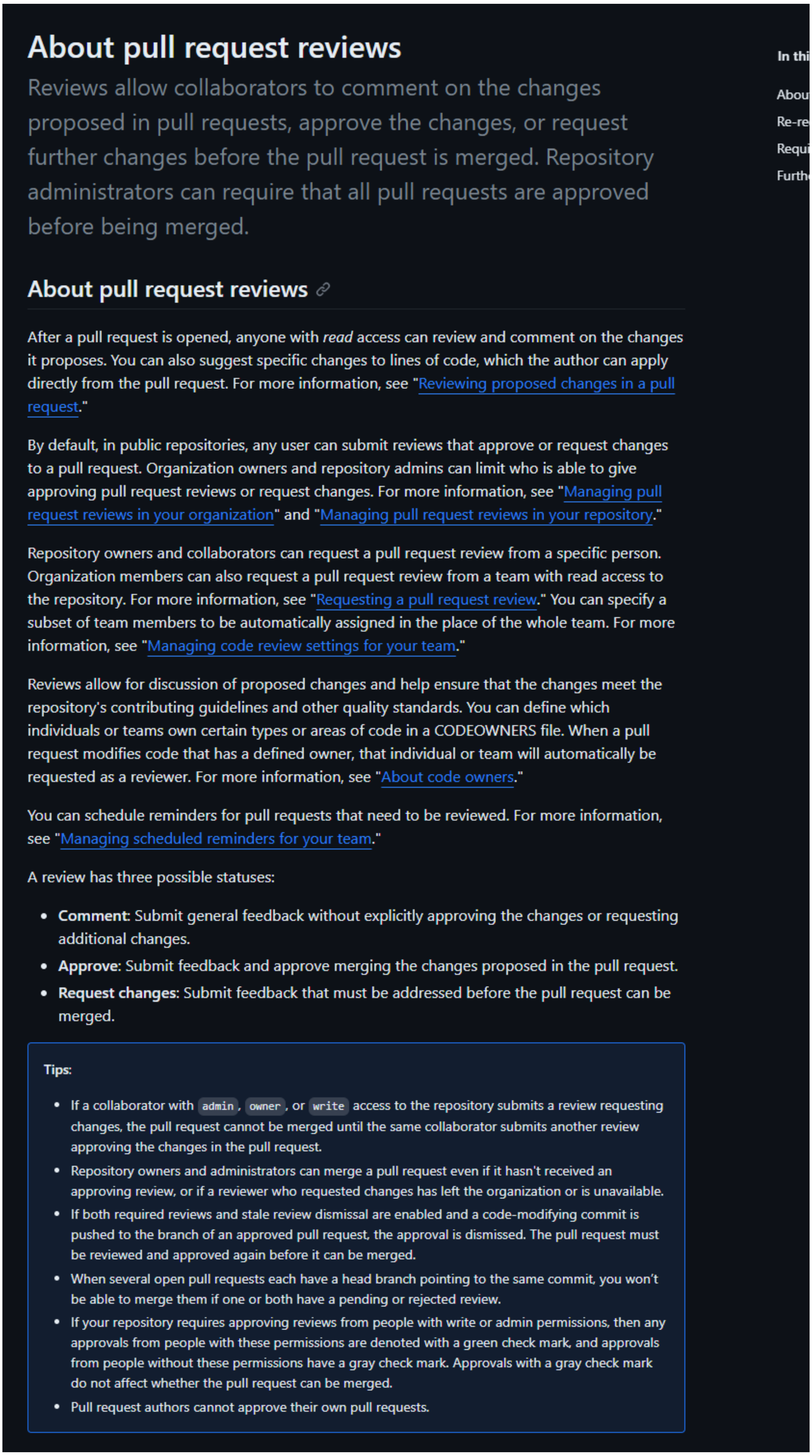
You can now merge the branches on the command line or push your changes to your remote repository on GitHub and merge your changes in a pull request.

Further reading

Code Review

https://google.github.io/styleguide/pyguide.html

https://google.github.io/styleguide/Rguide.html



About pull request reviews

Reviews allow collaborators to comment on the changes proposed in pull requests, approve the changes, or request further changes before the pull request is merged. Repository administrators can require that all pull requests are approved before being merged.

About pull request reviews

After a pull request is opened, anyone with *read* access can review and comment on the changes it proposes. You can also suggest specific changes to lines of code, which the author can apply directly from the pull request. For more information, see "Reviewing proposed changes in a pull request."

By default, in public repositories, any user can submit reviews that approve or request changes to a pull request. Organization owners and repository admins can limit who is able to give approving pull request reviews or request changes. For more information, see "Managing pull request reviews in your organization" and "Managing pull request reviews in your repository."

Repository owners and collaborators can request a pull request review from a specific person. Organization members can also request a pull request review from a team with read access to the repository. For more information, see "Requesting a pull request review." You can specify a subset of team members to be automatically assigned in the place of the whole team. For more information, see "Managing code review settings for your team."

Reviews allow for discussion of proposed changes and help ensure that the changes meet the repository's contributing guidelines and other quality standards. You can define which individuals or teams own certain types or areas of code in a CODEOWNERS file. When a pull request modifies code that has a defined owner, that individual or team will automatically be requested as a reviewer. For more information, see "About code owners."

You can schedule reminders for pull requests that need to be reviewed. For more information, see "Managing scheduled reminders for your team."

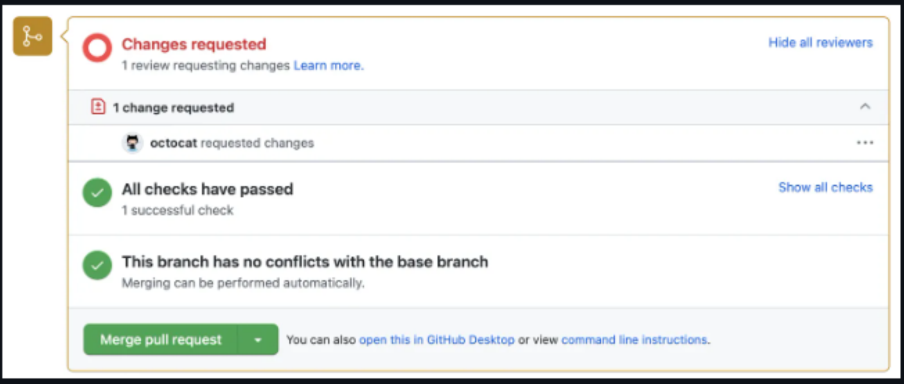
A review has three possible statuses:

- **Comment**: Submit general feedback without explicitly approving the changes or requesting additional changes.
- **Approve**: Submit feedback and approve merging the changes proposed in the pull request.
- **Request changes**: Submit feedback that must be addressed before the pull request can be merged.

Tips:

- If a collaborator with `admin`, `owner`, or `write` access to the repository submits a review requesting changes, the pull request cannot be merged until the same collaborator submits another review approving the changes in the pull request.
- Repository owners and administrators can merge a pull request even if it hasn't received an approving review, or if a reviewer who requested changes has left the organization or is unavailable.
- If both required reviews and stale review dismissal are enabled and a code-modifying commit is pushed to the branch of an approved pull request, the approval is dismissed. The pull request must be reviewed and approved again before it can be merged.
- When several open pull requests each have a head branch pointing to the same commit, you won't be able to merge them if one or both have a pending or rejected review.
- If your repository requires approving reviews from people with write or admin permissions, then any approvals from people with these permissions are denoted with a green check mark, and approvals from people without these permissions have a gray check mark. Approvals with a gray check mark do not affect whether the pull request can be merged.
- Pull request authors cannot approve their own pull requests.

You can view all of the reviews a pull request has received in the Conversation timeline, and you can see reviews by repository owners and collaborators in the pull request's merge box.


Changes requested
1 review requesting changes Learn more.
Hide all reviewers
1 change requested
octocat requested changes
All checks have passed
1 successful check
Show all checks
This branch has no conflicts with the base branch
Merging can be performed automatically.
Merge pull request
You can also open this in GitHub Desktop or view command line instructions.

Tip: You can find a pull request where you or a team you're a member of is requested for review with the search qualifier `review-requested:[USERNAME]` or `team-review-requested:[TEAMNAME]`. For more information, see "Searching issues and pull requests."

Resolving conversations

You can resolve a conversation in a pull request if you opened the pull request or if you have write access to the repository where the pull request was opened.

To indicate that a conversation on the **Files changed** tab is complete, click **Resolve conversation**.

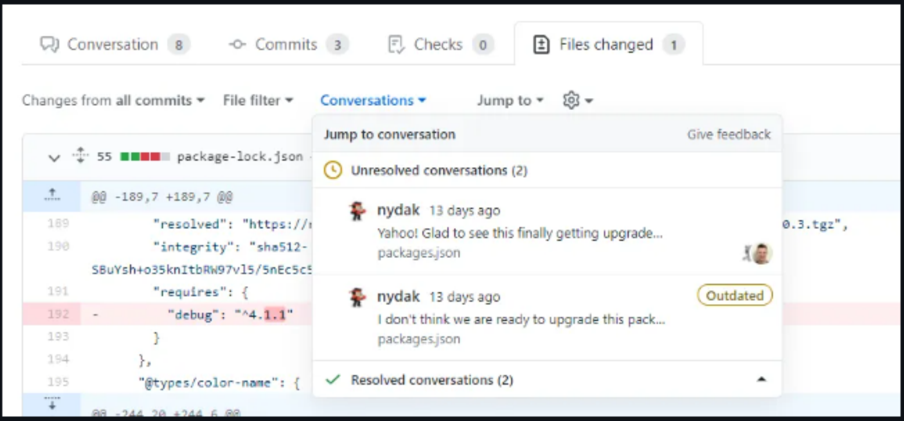
The entire conversation will be collapsed and marked as resolved, making it easier to find conversations that still need to be addressed.

If the suggestion in a comment is out of your pull request's scope, you can open a new issue that tracks the feedback and links back to the original comment. For more information, see "Creating an issue."

Discovering and navigating conversations

You can discover and navigate to all the conversations in your pull request using the **Conversations** menu that's shown at the top of the **Files Changed** tab.

From this view, you can see which conversations are unresolved, resolved, and outdated. This makes it easy to discover and resolve conversations.



nydak 13 days ago
Yahoo! Glad to see this finally getting upgrade...
packages.json

nydak 13 days ago
I don't think we are ready to upgrade this pack...
packages.json

```
55 package-lock.json
@@ -189,7 +189,7 @@
      "resolved": "https://r                                    0.3.tgz",
      "integrity": "sha512-
SBuYsh+o35knItbRW97vl5/5nEc5c
      "requires": {
-       "debug": "^4.1.1"
      }
    },
    "@types/color-name": {
```

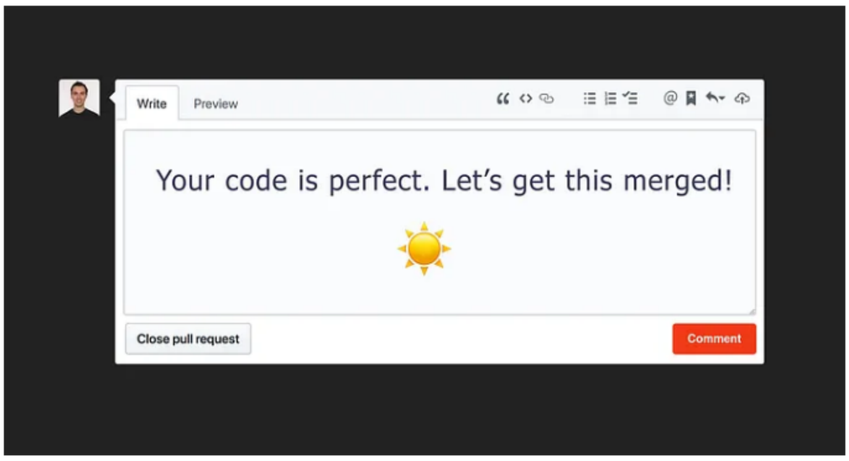
Re-requesting a review

You can re-request a review, for example, after you've made substantial changes to your pull request. To request a fresh review from a reviewer, in the sidebar of the **Conversation** tab, click the icon.

Required reviews

Repository administrators or custom roles with the "edit repository rules" permission can require that all pull requests receive a specific number of approving reviews before someone merges the pull request into a protected branch. You can require approving reviews from people with write permissions in the repository or from a designated code owner. For more information, see "About protected branches."

Tip: If necessary, people with *admin* or *write* access to a repository can dismiss a pull request review. For more information, see "Dismissing a pull request review."



Your code is perfect. Let's get this merged! ☀️

Chances are usually pretty low that you see this image on a pull request

The Perfect Code Review Process

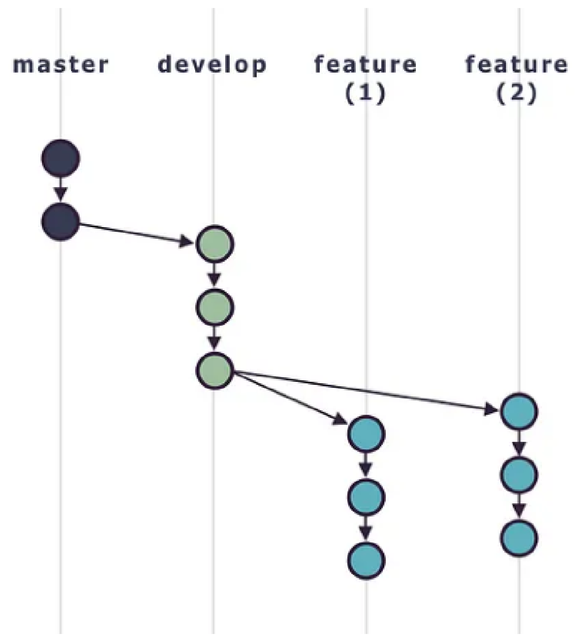
Robert Cooper · Follow
Published in Osedea · 6 min read · Jul 10, 2018

847 8

This article talks about a fictional scenario that tries to represent a near-ideal code review process. The story revolves around app development for a company and not an open source project. Also, it should be noted that this is **an opinionated article** of what makes a good code review. There are many different approaches that can make up an excellent code review process and the optimal process will vary depending on a variety of factors.

That being said, enjoy the story 📖

Meet Jimmy. Jimmy is working away on a new feature for his project. Jimmy and his team are using Git as their version control system and he is making his commits on a feature branch that will eventually get merged into the development (*dev*) branch (the team follows a GitFlow branching model). Before Jimmy can get his feature merged into the dev branch, his code needs to go through a **code review**.



Simplified representation of the branches in Jimmy's project. Jimmy is currently working on a feature branch and will soon be merging his feature into dev.

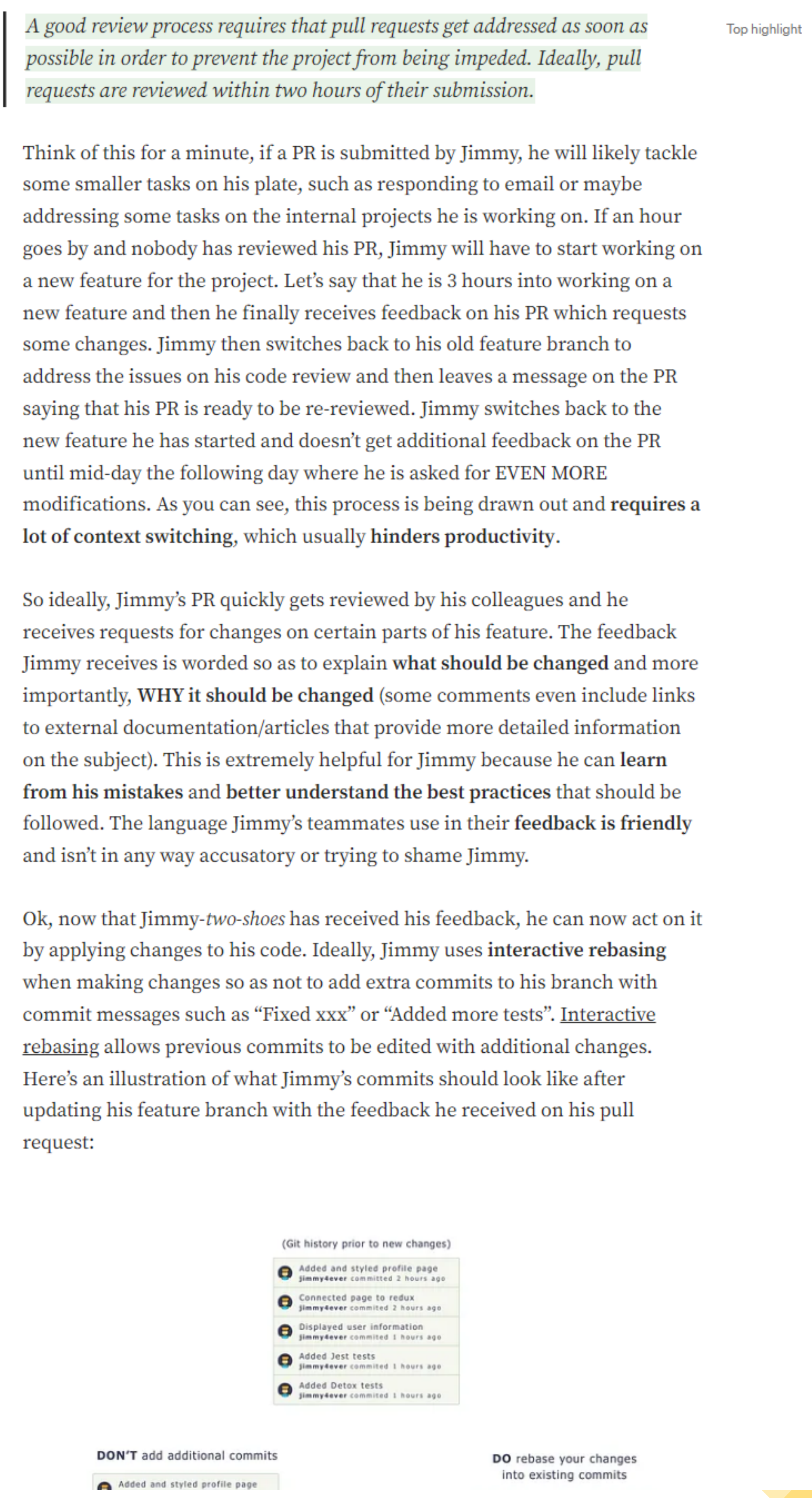
Why does Jimmy's code need to be reviewed before merging? Well, all the code in the dev branch should leave the project in a functional state and follow certain project level standards. Some examples of project level standards include **adding the appropriate tests**, correctly **defining variable types**, and having the code **logic in the appropriate files/functions.** Since Jimmy isn't a perfect programmer (especially after a late night of binge watching the latest season of Black Mirror), his code needs to be checked by his team members.

Jim-*bo* works with 3 other team members. When Jimmy has completed his feature, he **pushes his feature branch to Github and makes a Pull Request** (PR) to initiate a code review. Jimmy then adds one or two of his teammates as reviewers to his pull request.

Why doesn't Jimmy add all of his team members as reviewers? Well the **more people that are asked to review** the code, the **more hours/resources you are dedicating** to reviewing code and those hours can probably best be used on other things than having 3 people review the same code.

> *If the purpose for adding all team members to a PR is to receive the quickest response, this usually doesn't work out as expected. There is usually one person that is the first to always review a PR and after this happens for many consecutive PRs, everyone expects the same person to review the pull request and then the requests for review end up getting ignored by most. This isn't so bad for a team of 4, but the problem gets amplifies with larger teams.*

If one of Jimmy's teammates is a "senior" level developer and Jimmy is a



A good review process requires that pull requests get addressed as soon as possible in order to prevent the project from being impeded. Ideally, pull requests are reviewed within two hours of their submission.

Top highlight

Think of this for a minute, if a PR is submitted by Jimmy, he will likely tackle some smaller tasks on his plate, such as responding to email or maybe addressing some tasks on the internal projects he is working on. If an hour goes by and nobody has reviewed his PR, Jimmy will have to start working on a new feature for the project. Let's say that he is 3 hours into working on a new feature and then he finally receives feedback on his PR which requests some changes. Jimmy then switches back to his old feature branch to address the issues on his code review and then leaves a message on the PR saying that his PR is ready to be re-reviewed. Jimmy switches back to the new feature he has started and doesn't get additional feedback on the PR until mid-day the following day where he is asked for EVEN MORE modifications. As you can see, this process is being drawn out and **requires a lot of context switching**, which usually **hinders productivity**.

So ideally, Jimmy's PR quickly gets reviewed by his colleagues and he receives requests for changes on certain parts of his feature. The feedback Jimmy receives is worded so as to explain **what should be changed** and more importantly, **WHY it should be changed** (some comments even include links to external documentation/articles that provide more detailed information on the subject). This is extremely helpful for Jimmy because he can **learn from his mistakes** and **better understand the best practices** that should be followed. The language Jimmy's teammates use in their **feedback is friendly** and isn't in any way accusatory or trying to shame Jimmy.

Ok, now that Jimmy-*two-shoes* has received his feedback, he can now act on it by applying changes to his code. Ideally, Jimmy uses **interactive rebasing** when making changes so as not to add extra commits to his branch with commit messages such as "Fixed xxx" or "Added more tests". Interactive rebasing allows previous commits to be edited with additional changes. Here's an illustration of what Jimmy's commits should look like after updating his feature branch with the feedback he received on his pull request:

(Git history prior to new changes)

Added and styled profile page
jimmy4ever committed 2 hours ago

Connected page to redux
jimmy4ever commited 2 hours ago

Displayed user information
jimmy4ever commited 1 hours ago

Added Jest tests
jimmy4ever commited 1 hours ago

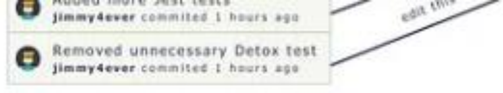
Added Detox tests
jimmy4ever commited 1 hours ago

DON'T add additional commits

Added and styled profile page

DO rebase your changes into existing commits

https://smartbear.com/learn/code-review/what-is-code-review/


Jimmy4ever commited 1 hours ago

Removed unnecessary Detox test
Jimmy4ever commited 1 hours ago

Notice how, for the recommended way of doing things, the commits are the same before and after the changes have been made to the feature branch? This is because an **interactive rebase was used to edit individual commits**. So instead of making a commit such as “Added more Jest tests”, the commit called “Added Jest tests” would be modified to include the added tests. This way, **all the commit messages are still relevant** and then there **are not a bunch of extra commits** spread about the feature branch.

Ok, so Jimmy made his changes, he pushed his changes to Github and has notified the other reviewers of his changes. If there are additional changes requested by Jimmy’s teammates, he will address those, otherwise Jimmy can merge his changes into the development branch. Jimmy proceeds to high-five all his teammates, bust out some office break-dance moves, and can continue living a full life.

Jimmy on a beach walking into the sunset and living a full life after having his PR merged into dev — Photo by Dekeister Leopold on Unsplash

That’s what I would describe as an ideal code review process. Does this always end up being followed? Unfortunately, no. Sometimes reviewers are too busy and don’t make the time to promptly review PRs. Other times the details of a PR are not clearly outlined in the Pull Request message and this makes it more difficult for the reviewers to provide a beneficial review. That being said, it is something to strive for and will make the development process more enjoyable.

Be like Jimmy and his perfect teammates and make the development world a better place for everyone 👫👬👭.

. . .

1. https://docs.github.com/en/issues/tracking-your-work-with-issues/linking-a-pull-request-to-an-issue
2. https://docs.github.com/en/issues/tracking-your-work-with-issues/linking-a-pull-request-to-an-issue
3. https://docs.github.com/en/communities/setting-up-your-project-for-healthy-contributions/setting-guidelines-for-repository-contributors

4. https://www.infoworld.com/article/3271126/what-is-cicd-continuous-integration-and-continuous-delivery-explained.html
5. https://stackify.com/what-is-cicd-whats-important-and-how-to-get-it-right/
6. https://docs.travis-ci.com/user/tutorial/
7. https://docs.travis-ci.com/user/build-stages/
8. https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/incorporating-changes-from-a-pull-request/about-pull-request-merges