**What You Should Look Out For When You Review Pull Request**



Photo by [Simon English](https://unsplash.com/@guernseyphotographer?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral" \t "_blank)

You just graduated with Computer Science degree and started your first job as junior developer. Thanks to your final year project, you equipped yourself with some coding experience which landed you in this job.

After a few days of honey moon period of onboarding, you are added to company enterprise Github. You know what Github is because it’s where pushed your final year project source code. You’re assigned to you first project with a few members working in the same code base. Coding starts.

A few hours later, you received email from GitHub. Someone in your team is requesting your review on his **pull request**. You don’t even know what pull request is or how to review it.



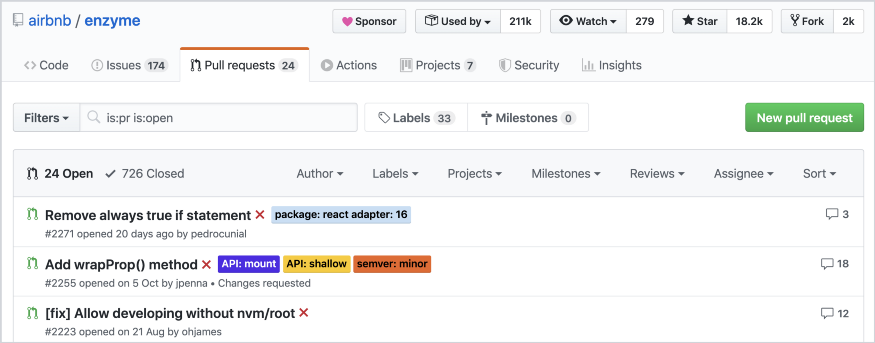
Photo by [Paul Hanaoka](https://unsplash.com/@paul_?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral" \t "_blank)

This story will briefly explain what pull request is and why it is important. If you’re already familiar with the definition, skip to the main point, which is the **9 questions** you should ask yourself when reviewing a pull request.

**What is Pull Request?**

*Pull requests let you tell others about changes you’ve pushed to a branch in a repository on GitHub. Once a pull request is opened, you can discuss and review the potential changes with collaborators and add follow-up commits before your changes are merged into the base branch.*[*About pull requests – GitHub Help*](https://help.github.com/en/github/collaborating-with-issues-and-pull-requests/about-pull-requests)

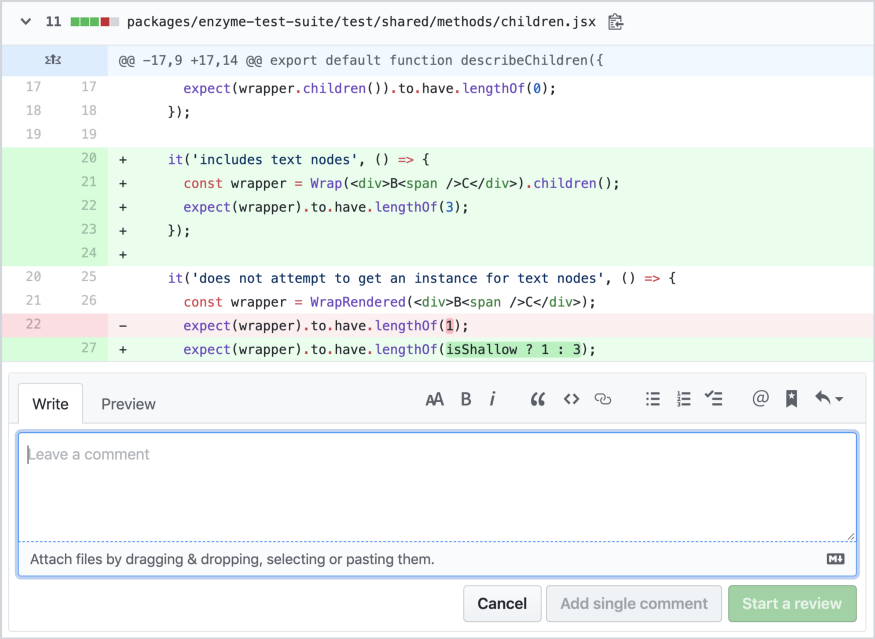
Pull request (hereinafter referred to as PR) is how you tell others about the changes you want to merge from your branch to *master* branch or *develop* branch.



airbnb / enzyme pull request page

The person who creates PR is called **author**. Author can request one or multiple people to review his PR. Those people are known as **reviewers**. Reviewer can be the engineer lead in the team, the project members, or anyone in the organization. Anyone can review a PR as long as enough context is given.

PR shows changes in the files which have been modified by the author. Reviewers gives comments on certain line of code when they find any potential issue or they want to suggest better solution.



airbnb / enzyme

PR could be just a few lines of changes in one file, or it may involve changes across multiple file. The scope of changes is usually based on a feature, a bug fix, or a chore to deal with [technical debt](https://en.wikipedia.org/wiki/Technical_debt" \t "_blank).

**Why Need Pull Request?**

When a software engineering team work in the same code repository, it’s crucial that everyone is merging code with non-breaking logics, adherence to standards, and agreeable styling. You can use linters to enforce styling, but logic is something requiring additional pair of eyes to validate. In order to achieve that, PR is one of the solutions.

PR allows other developers to validate your logic in different perspective. If PR review is done carefully, the chance of bug in production is significantly low. PR is the last and strongest defensive line against bugs in production.



Photo by [Nathan Shively](https://unsplash.com/@shivelycreative?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral" \t "_blank)

Although you have a complete set of tests to fully cover your code, it doesn’t guarantee you have covered all the corner cases. QA testers usually can’t thoroughly spot all the minor errors which later snowball to larger defects. For example, using Math.round vs Math.floor.

**Does Experience Matter?**

Pull request is commonly used in open source projects to allow anyone to contribute with proper reviewing guidelines and process. In companies, it is also context sharing opportunity to make sure more than one developers are aware of the change. It can also be served as a learning opportunity for junior developers and senior developers to learn from each other.



Photo by [Vlad Sargu](https://unsplash.com/@vladsargu?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral" \t "_blank)

With that being said, junior developers should not feel discouraged or inferior when reviewing senior developers’ PR. Seniority and years of experiences doesn’t guarantee your code to be perfect. In fact, if a senior developer writes code which could not be understood by junior developer, it means the readability of the code needs improvement.

Now that you know how important PR is, let’s get to the questions you should ask yourself when reviewing PR.



Photo by [Emily Morter](https://unsplash.com/@emilymorter?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral" \t "_blank)

**1. What does this PR do?**

The first question is — **what?**and it’s followed by**why?**As a reviewer, you need to know what you’re reviewing. To get full context of what the changes are, PR must have a well-defined title and description.

Title must be concise, not too long and not too vague. Make sure the title follow a certain format standardized within your organization if any. For example:

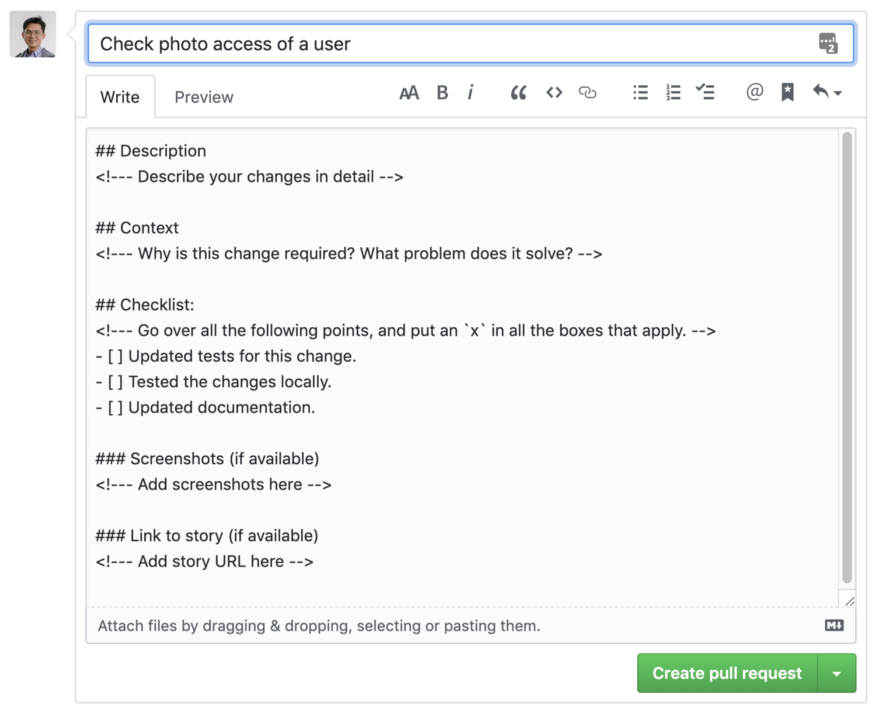
* Use a square bracket for ticket number
* Start the title with an action verb

**[XYZ-123] Add error message when request returns error**

Description should summarises what the changes are made and what they are for. Usually the changes are tied to one or more stories ([Pivotal Tracker](https://www.pivotaltracker.com/)) or tickets ([JIRA](https://www.atlassian.com/software/jira)). The description should include links to those stories or tickets.

If you don’t get enough context from title and description, ask the author to provide more details. If the changes are related to frontend, ask the author to provide a screenshot of before-change and after-change.

**Note**: Github allows you to [customize pull request template](https://help.github.com/en/github/building-a-strong-community/creating-a-pull-request-template-for-your-repository" \t "_blank) if you create **PULL\_REQUEST\_TEMPLATE.md** file inside **doc** or **.github** folder at your project root. Here is an example of [**PULL\_REQUEST\_TEMPLATE.md**](https://github.com/taingmeng/pull-request-review-medium/blob/master/.github/PULL_REQUEST_TEMPLATE.md):



**2. Does this PR do what it’s supposed to do?**

The title and description may **lie** to you. The author wrote the description in one way, but the actual changes may be in another way.



Photo by [Andrew Le](https://unsplash.com/@letanloc1941995?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral" \t "_blank)

Don’t get the author wrong. He did his job according the story requirement, but he may also have done something more. The author might have been tempted to fix indentation, spacing, or a typo. One line of actual change, 100 lines of indentation change. Can you spot which line is the actual change below?



One line of actual change, 100 lines of indentation change

It’s still acceptable if those changes are in the same line of what need to get done. Otherwise, you’ll have to spend more time and attention for those unrelated changes.

One PR should only have one purpose and do it precisely. As a reviewer, you need to do one quick round of bird-eye-view scanning through the changes. If you spot any unrelated changes, the first comment:

The change here doesn’t seem to relate to you PR. Can you create a separate PR for this?

**3. Does this PR do what it’s supposed to do correctly?**

Here is the most critical purpose of PR review. It’s also the most difficult item in the check list because you have to read the changes line by line carefully. This is question you should spend the most time on. The comment you give here is also the most valuable.

**Logic error could be lethal in production**. Such a mistake is hard to spot because the author may only handle simple cases.



Photo by [Daniel Tausis](https://unsplash.com/@greatmalinco?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral" \t "_blank)

If you feel any line of code might throw exception, clarify with the author whether there is any chance that exception could be thrown and how it is handled.

You have to fully understand code by just reading. A good code should not let the review feel the urge of checking out the branch and run it locally. However, if you do feel checking out the branch and running it locally help you understand the code faster, please do.

You should not be afraid to ask questions if you don’t understand what a function does. Write a comment to ask the author to explain.

Code should be readable, not just runnable.

**4. Is the code readable?**

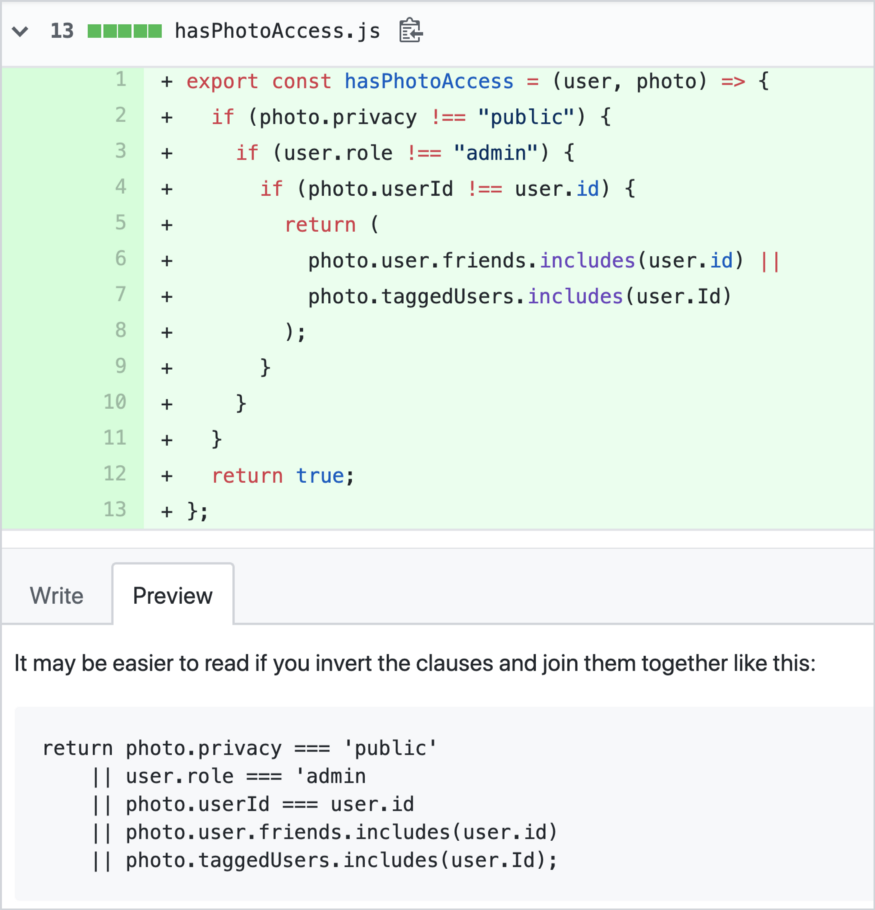
Assuming you know most of the syntax of a programming language, if you read through the code once and still don’t know what it does, it’s not readable. If unreadable code is merged, anyone who has to work with that code in the future will have a lot of WTF time. That anyone could be you.

Here are the examples of common unreadable code and possible suggestion:

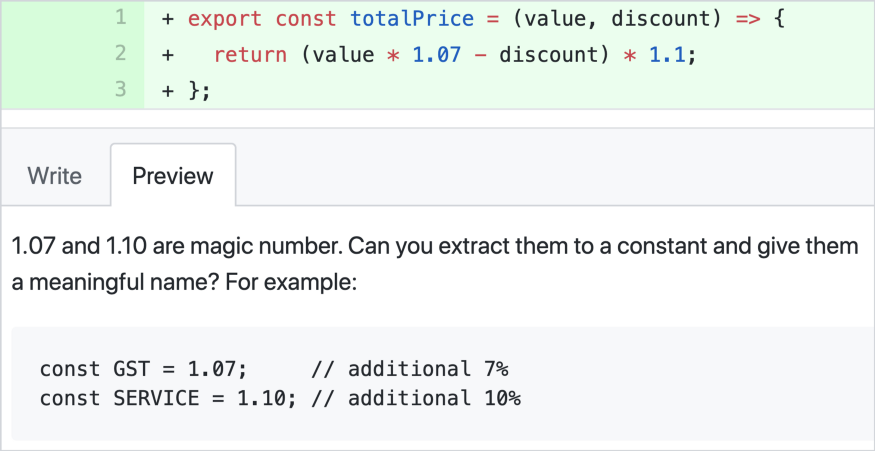
* **The function is too long.**Imagine a function has more than 100 lines of code. It may be doing too many things or even too many unrelated things. Ask the author to break the logic down into smaller functions. It could be better for both readability and testability.

We should extract these few lines to a separate function because they are handling different logic from your function.

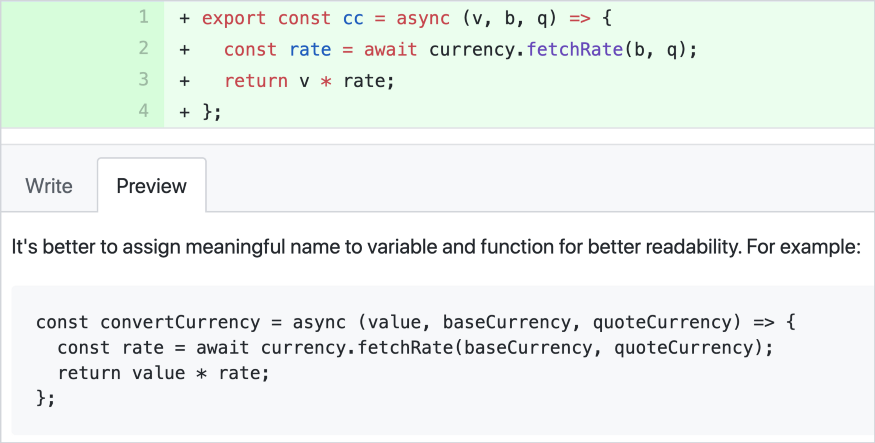
* **Too many nested conditions**. When you read too many nested conditions, it could literally damage your brain cells, and a lot of them.



* **Magic number.** Any number which comes out of nowhere should be extracted as a constant and given a name. The constant could be local or exported depending on whether it’s used somewhere else.



* **Ambiguous naming.** Variable and function naming should be concise and meaningful. One letter acronym is usually unacceptable because other developers may not what it means.

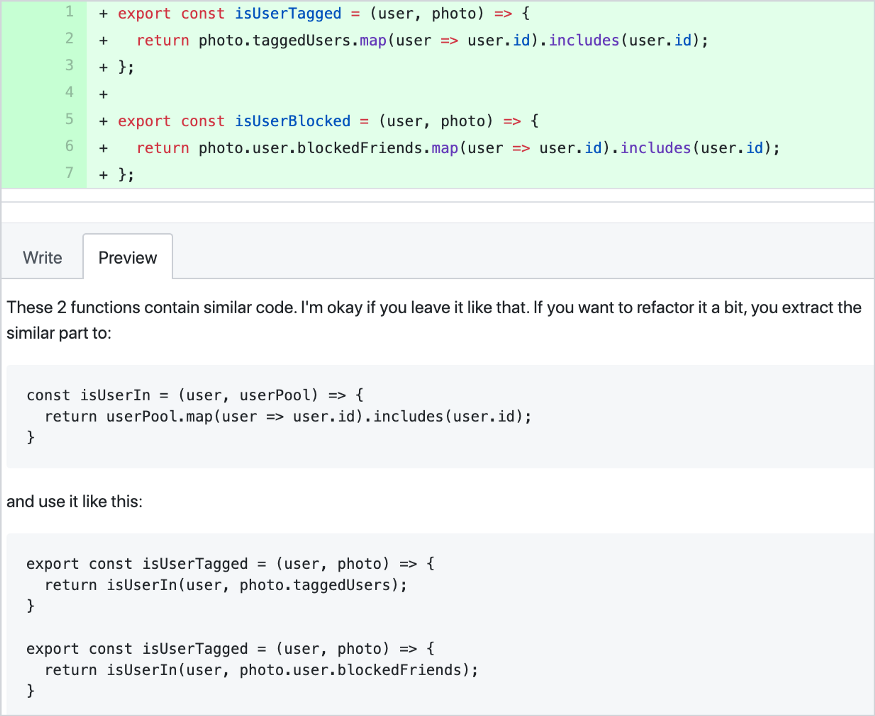


**5. Can the code be shorter?**

This is where you can suggest refactoring if necessary. Remember the word **necessary**is subjective because people define necessity differently. There are a million ways to do one thing. People usually like the original idea they come up by themselves. You should make sure your comment here sounds like a suggestion more than enforcement.

There are a few principles to shorten the code.

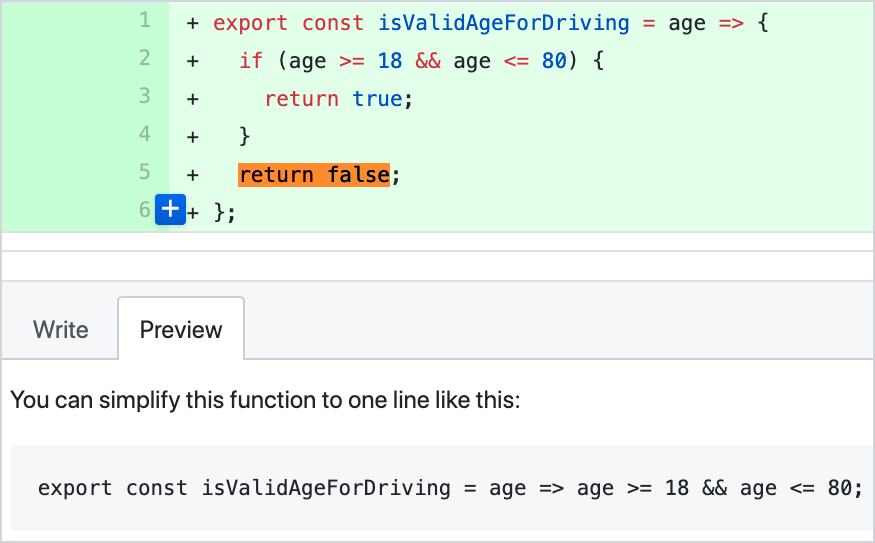
* **Do not repeat yourself** (**DRY**). Look for duplicated code. If two functions are doing similar task, there is a chance that some code can be reused.



* **Shorthand syntax.**Every language has its own sugar syntax to reduce verbosity. Some developers simply apply the knowledge from one language they are familiar with to a new language without knowing there is better syntax.



* **Return condition instead of boolean.** If you find a function which returns true or false based on a certain condition, you can suggest compressing the code by returning that condition instead.



**6. Can the code be smarter?**

This is where you could suggest optimisation or prevent deterioration. If new changes make the existing code execute slower even for a few hundreds of milliseconds, you should prevent that by giving suggestion or asking the author to consider different approach.

Different languages and frameworks have different ways of improving run time. This is where experience may matter between junior and senior level.



Photo by [Austin Distel](https://unsplash.com/@austindistel?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral" \t "_blank)

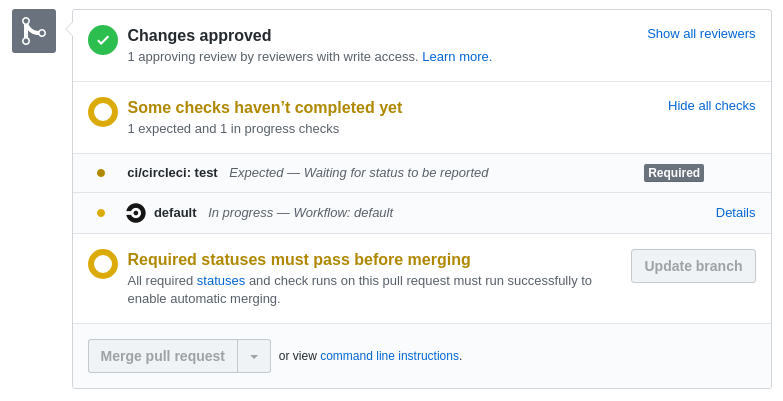
Similar to how you tackle Google whiteboard interview, the author may use a brute force approach, so you may suggest better solution by trading off between run time and memory. For example, use HashMap vs List.

In Javascript, if you find the author wait for multiplePromise one after another, you can suggest him use Promise.all.

For frontend development, if the author is writing a long running task which blocks UI update, the better approach you could suggest to the author is making the code run asynchronously or in different thread.

**7. Are changes covered by tests?**

If the project you’re working on has continuous integration workflow configured, it should run all the test and show a green tick on when all tests passed. You don’t have to run tests locally.



If the project requires high test coverage, you need to check whether the new changes are covered by existing tests or new tests.

In most programming languages, unit test file has the same name as its actual file with the word *test* or *spec*appended somewhere in between*.*For example:

* *database\_service.py → test\_database\_service.py*
* *date\_formatter.rb → date\_formatter\_test.rb*
* *StringUtil.js → StringUtil.test.js*
* *Person.java → TestPerson.java*

There should be 1-to-1 mapping between actual file and test file. The change in one file should have its corresponding change in its test file change, and vice versa. If you don’t see the corresponding, there are 3 possible scenarios:

* **The new change is not testable or should not be tested** (environment config change, documentation change, etc). If you agree that no test should be written for this change, move on.
* **The new change doesn’t break existing tests**. You need to check further if existing tests already cover the new change. If not, ask the author to add more tests.
* **No existing tests**. Untested code was merged in because someone didn’t do a full check in one of the PRs. Ask author to create the test file and add tests for his change. He should not add tests for the part which is out of the scope of this PR. Those missing tests should be handled in another PR.

**8. Are changes covered by enough tests?**

100% test coverage doesn’t mean the code is fully tested. It simply means every line has been run through at least once during the test execution.

In the example above, the test fully covers the code. Is there anything wrong? Your developer instinct should post this question in your mind:

Could the array be null, empty or contain only 1 item?

Although you see one tested is added for the function, you should always ask yourself if one test is enough.

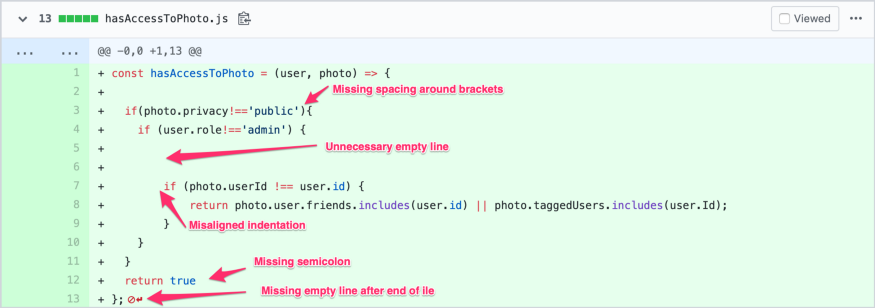


Photo by [Paweł Czerwiński](https://unsplash.com/@pawel_czerwinski?utm_source=medium&utm_medium=referral" \t "_blank) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral" \t "_blank)

**9. Is everything clean?**

Last but not least, check the follow styling inconsistency:

* Misaligned indentation
* Extra spacing or no spacing
* Extra line break
* Missing semicolon
* Missing empty line at end of file



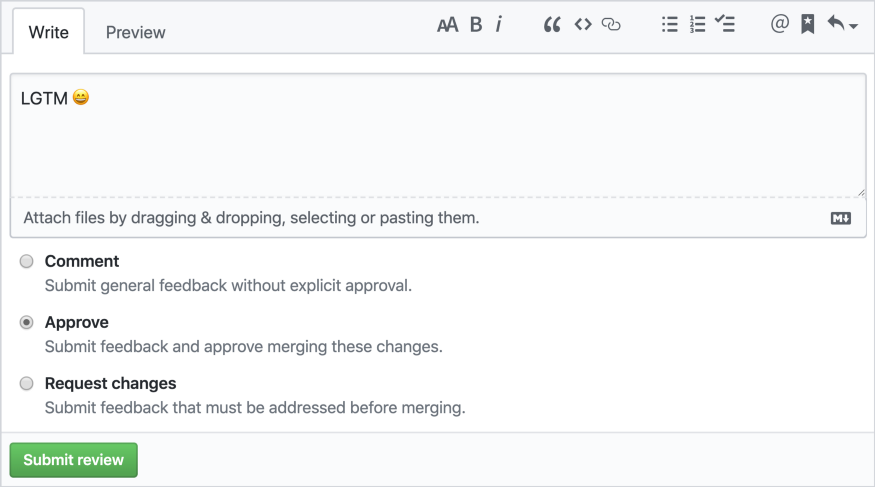
You should have formatter and linter setup at project level and editor level to handle this. Everyone in the project should using the same configuration. Otherwise, there will be no ending in commenting on this on every PR.

Unfortunately, there is no guarantee that linter and formatter works all the time. Developers might accidentally disable them locally. There could be other plugin overriding them.

Remind the author to check whether linter or formatter in their editor still works as expected.

**What Have You Learned?**

At the end of the day, if you still don’t know how to review, there is one ultimate solution. Just approve with **LGTM** (look good to me). Just kidding 😜.



The most straightforward way to review pul request

Let’s recap the questionnaires:

* What does this PR do?
* Do this PR do what it’s supposed?
* Do this PR do what it’s supposed to do correctly?
* Is the code readable?
* Can the code be shorter?
* Can the code be smarter?
* Are changes covered by tests?
* Are changes covered by enough tests?
* Is everything clean?