

6.2 GitHub - Contributing to a Project

Contributing to a Project

Now that our account is set up, let's walk through some details that could be useful in helping you contribute to an existing project.

Forking Projects

If you want to contribute to an existing project to which you don't have push access, you can "fork" the project. When you "fork" a project, GitHub will make a copy of the project that is entirely yours; it lives in your namespace, and you can push to it.

Note Historically, the term "fork" has been somewhat negative in context, meaning that someone took an open source project in a different direction, sometimes creating a competing project and splitting the contributors. In GitHub, a "fork" is simply the same project in your own namespace, allowing you to make changes to a project publicly as a way to contribute in a more open manner.

This way, projects don't have to worry about adding users as collaborators to give them push access. People can fork a project, push to it, and contribute their changes back to the original repository by creating what's called a Pull Request, which we'll cover next. This opens up a discussion thread with code review, and the owner and the contributor can then communicate about the change until the owner is happy with it, at which point the owner can merge it in.

To fork a project, visit the project page and click the "Fork" button at the top-right of the page.



Figure 88. The "Fork" button

After a few seconds, you'll be taken to your new project page, with your own writeable copy of the code.

The GitHub Flow

GitHub is designed around a particular collaboration workflow, centered on Pull Requests. This flow works whether you're collaborating with a tightly-knit team in a single shared repository, or a globally-distributed company or network of strangers contributing to a project through dozens of forks. It is centered on the [Topic Branches](#) workflow covered in [Git Branching](#).

Here's how it generally works:

1. Fork the project.
2. Create a topic branch from `master`.
3. Make some commits to improve the project.
4. Push this branch to your GitHub project.
5. Open a Pull Request on GitHub.
6. Discuss, and optionally continue committing.
7. The project owner merges or closes the Pull Request.
8. Sync the updated master back to your fork.

This is basically the Integration Manager workflow covered in [Integration-Manager Workflow](#), but instead of using email to communicate and review changes, teams use GitHub's web based tools.

Let's walk through an example of proposing a change to an open source project hosted on GitHub using this flow.

Tip You can use the official **GitHub CLI** tool instead of the GitHub web interface for most things. The tool can be used

on Windows, MacOS, and Linux systems. Go to the [GitHub CLI homepage](https://github.com/cli/cli) for installation instructions and the manual.

Creating a Pull Request

Tony is looking for code to run on his Arduino programmable microcontroller and has found a great program file on GitHub at <https://github.com/schacon/blink>.

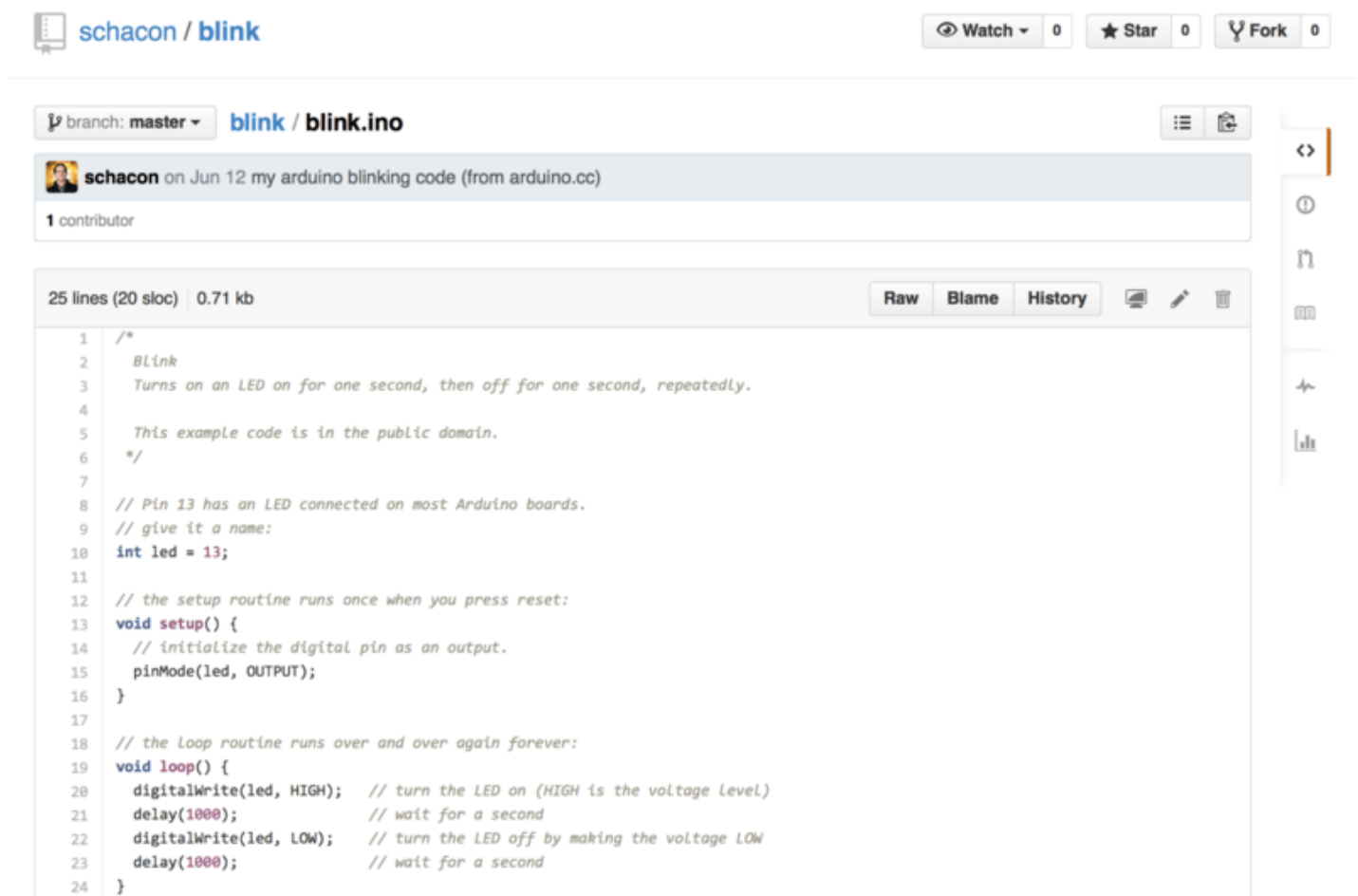


Figure 89. The project we want to contribute to

The only problem is that the blinking rate is too fast. We think it's much nicer to wait 3 seconds instead of 1 in between each state change. So let's improve the program and submit it back to the project as a proposed change.

First, we click the 'Fork' button as mentioned earlier to get our own copy of the project. Our user name here is "tonychacon" so our copy of this project is at <https://github.com/tonychacon/blink> and that's where we can edit it. We will clone it locally, create a topic branch, make the code change and finally push that change back up to GitHub.

```
$ git clone https://github.com/tonychacon/blink (1)
Cloning into 'blink'...
```

```
$ cd blink
$ git checkout -b slow-blink (2)
Switched to a new branch 'slow-blink'
```

```
$ sed -i 's/1000/3000/' blink.ino (macOS) (3)
# If you're on a Linux system, do this instead:
# $ sed -i 's/1000/3000/' blink.ino (3)
```

```
$ git diff --word-diff (4)
diff --git a/blink.ino b/blink.ino
index 15b9911..a6cc5a5 100644
--- a/blink.ino
+++ b/blink.ino
@@ -18,7 +18,7 @@ void setup() {
 // the loop routine runs over and over again forever:
void loop() {
  digitalWrite(led, HIGH);   // turn the LED on (HIGH is the voltage level)
```

```
[-delay(1000);-]{+delay(3000);+}          // wait for a second
digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
[-delay(1000);-]{+delay(3000);+}          // wait for a second
}
```

```
$ git commit -a -m 'Change delay to 3 seconds' (5)
[slow-blink 5ca509d] Change delay to 3 seconds
1 file changed, 2 insertions(+), 2 deletions(-)
```

```
$ git push origin slow-blink (6)
Username for 'https://github.com': tonychacon
Password for 'https://tonychacon@github.com':
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 340 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To https://github.com/tonychacon/blink
 * [new branch]      slow-blink -> slow-blink
```

1. Clone our fork of the project locally.
2. Create a descriptive topic branch.
3. Make our change to the code.
4. Check that the change is good.
5. Commit our change to the topic branch.
6. Push our new topic branch back up to our GitHub fork.

Now if we go back to our fork on GitHub, we can see that GitHub noticed that we pushed a new topic branch up and presents us with a big green button to check out our changes and open a Pull Request to the original project.

You can alternatively go to the “Branches” page at <https://github.com/<user>/<project>/branches> to locate your branch and open a new Pull Request from there.

The screenshot shows the GitHub interface for the repository 'tonychacon / blink'. At the top, there's a header with the repository name and a 'Compare & pull request' button. Below this, a section titled 'Your recently pushed branches:' lists the 'slow-blink' branch, which was pushed 'less than a minute ago'. A green bar highlights this branch, and a 'Compare & pull request' button is visible. Below the branches list, the README content is displayed, starting with 'Blink' and 'This repository has an example file to blink the LED on an Arduino board.' On the right sidebar, there are links for 'Code', 'Pull Requests', 'Wiki', 'Pulse', 'Graphs', and 'Settings'. At the bottom right, there are buttons for 'Clone in Desktop' and 'Download ZIP'.

Figure 90. Pull Request button

If we click that green button, we'll see a screen that asks us to give our Pull Request a title and description. It is almost always worthwhile to put some effort into this, since a good description helps the owner of the original project determine what you were trying to do, whether your proposed changes are correct, and whether accepting the changes would improve the original project.

We also see a list of the commits in our topic branch that are “ahead” of the master branch (in this case, just the one) and a unified diff of all the changes that will be made should this branch get merged by the project owner.

The screenshot shows the GitHub interface for creating a pull request. At the top, the repository is identified as 'tonychacon / blink', forked from 'schacon/blink'. It shows 1 Unwatch, 0 Stars, and 1 Fork. The pull request title is 'Three seconds is better'. The description area contains the text 'Studies have shown that 3 seconds is a far better LED delay than 1 second.' and a URL 'http://studies.example.com/optimal-led-delays.html'. A green button 'Create pull request' is visible. Below the description, it shows '1 commit', '1 file changed', '0 commit comments', and '1 contributor'. A diff for 'blink.ino' is displayed, showing changes to the delay values in the loop function.

```
@@ -18,7 +18,7 @@ void setup() {
18 18 // the loop routine runs over and over again forever:
19 19 void loop() {
20 20   digitalWrite(led, HIGH); // turn the LED on (HIGH is the voltage level)
21 21   - delay(1000);           // wait for a second
22 22   + delay(3000);           // wait for a second
23 23   digitalWrite(led, LOW);  // turn the LED off by making the voltage LOW
24 24   - delay(1000);           // wait for a second
25 25   + delay(3000);           // wait for a second
26 26 }
```

Figure 91. Pull Request creation page

When you hit the 'Create pull request' button on this screen, the owner of the project you forked will get a notification that someone is suggesting a change and will link to a page that has all of this information on it.

Note Though Pull Requests are used commonly for public projects like this when the contributor has a complete change ready to be made, it's also often used in internal projects *at the beginning* of the development cycle. Since you can keep pushing to the topic branch even **after** the Pull Request is opened, it's often opened early and used as a way to iterate on work as a team within a context, rather than opened at the very end of the process.

Iterating on a Pull Request

At this point, the project owner can look at the suggested change and merge it, reject it or comment on it. Let's say that he likes the idea, but would prefer a slightly longer time for the light to be off than on.

Where this conversation may take place over email in the workflows presented in [Distributed Git](#), on GitHub this happens online. The project owner can review the unified diff and leave a comment by clicking on any of the lines.

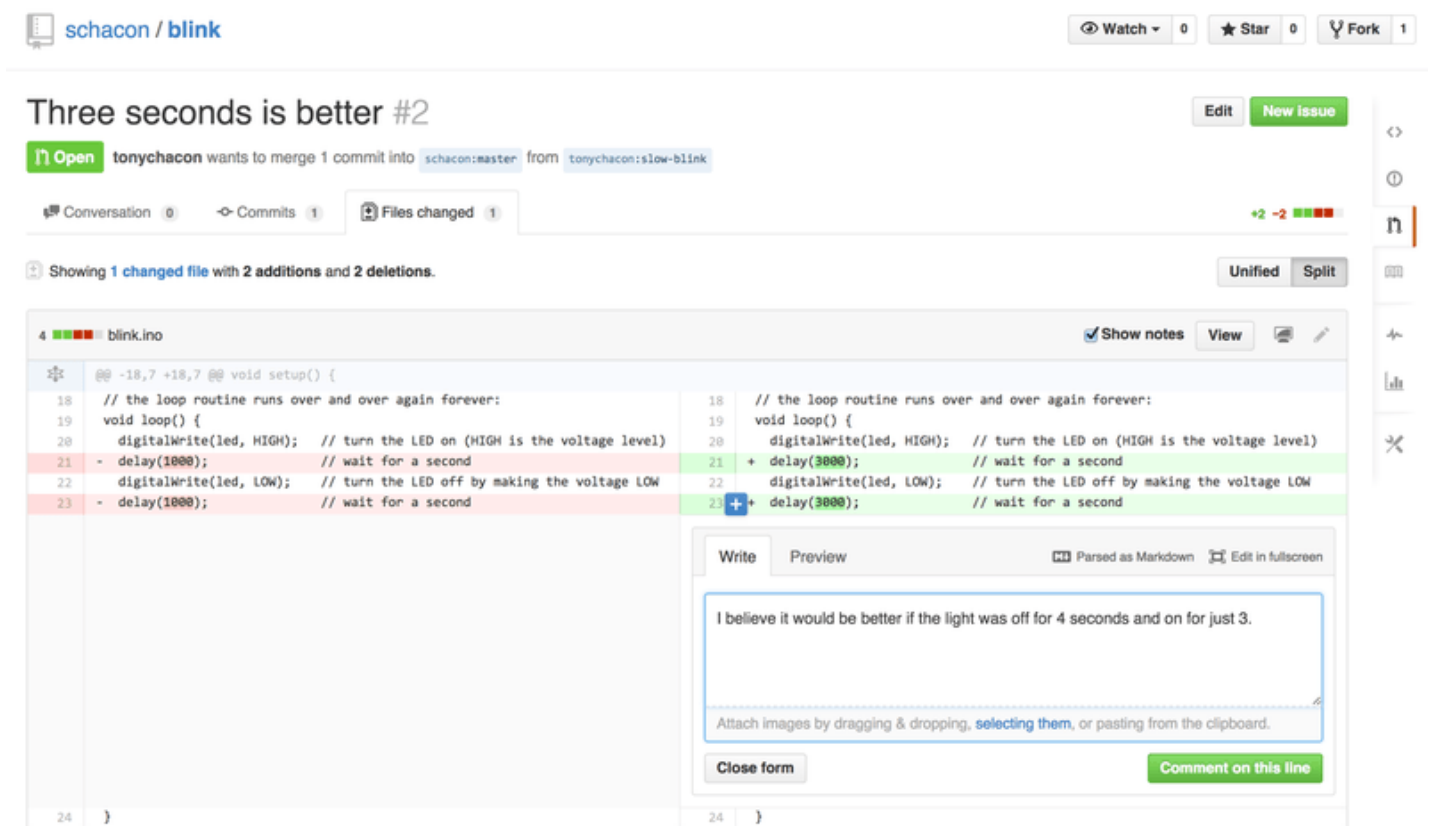


Figure 92. Comment on a specific line of code in a Pull Request

Once the maintainer makes this comment, the person who opened the Pull Request (and indeed, anyone else watching the repository) will get a notification. We'll go over customizing this later, but if he had email notifications turned on, Tony would get an email like this:



Figure 93. Comments sent as email notifications

Anyone can also leave general comments on the Pull Request. In [Pull Request discussion page](#) we can see an example of the project owner both commenting on a line of code and then leaving a general comment in the discussion section. You can see that the code comments are brought into the conversation as well.

Three seconds is better #2

Edit New Issue

Open tonychacon wants to merge 1 commit into schacon:master from tonychacon:slow-blink

Conversation 1 Commits 1 Files changed 1

+2 -2



tonychacon commented 6 minutes ago

Studies have shown that 3 seconds is a far better LED delay than 1 second.

<http://studies.example.com/optimal-led-delays.html>

three seconds is better

db44c53

schacon commented on the diff just now

blink.ino

[View full changes](#)

		((6 lines not shown))	
22	22	digitalWrite(led, LOW);	// turn the LED off by making the voltage LOW
23	-	delay(1000);	// wait for a second
	23	+ delay(3000);	// wait for a second

schacon added a note just now

Owner

I believe it would be better if the light was off for 4 seconds and on for just 3.

Add a line note



schacon commented just now

Owner

If you make that change, I'll be happy to merge this.

Labels

None yet

Milestone

No milestone

Assignee

No one—assign yourself

Notifications

Unsubscribe

You're receiving notifications because you commented.

2 participants




Lock pull request

Figure 94. Pull Request discussion page

Now the contributor can see what they need to do in order to get their change accepted. Luckily this is very straightforward. Where over email you may have to re-roll your series and resubmit it to the mailing list, with GitHub you simply commit to the topic branch again and push, which will automatically update the Pull Request. In [Pull Request final](#) you can also see that the old code comment has been collapsed in the updated Pull Request, since it was made on a line that has since been changed.

Adding commits to an existing Pull Request doesn't trigger a notification, so once Tony has pushed his corrections he decides to leave a comment to inform the project owner that he made the requested change.


Three seconds is better #2

 **tonychacon** wants to merge 3 commits into `schacon:master` from `tonychacon:slow-blink`

Conversation 3

Commits 3

Files changed 1




tonychacon commented 11 minutes ago


Studies have shown that 3 seconds is a far better LED delay than 1 second.
<http://studies.example.com/optimal-led-delays.html>

three seconds is better

db44c53

 **schacon** commented on an outdated diff 5 minutes ago

Show outdated diff




schacon commented 5 minutes ago

Owner

✎ ✕

If you make that change, I'll be happy to merge this.


 **tonychacon** added some commits 2 minutes ago

longer off time

0c1f66f


remove trailing whitespace

ef4725c



tonychacon commented 10 seconds ago

I changed it to 4 seconds and also removed some trailing whitespace that I found. Anything else you would like me to do?



This pull request can be automatically merged.

You can also merge branches on the [command line](#).


 **Merge pull request**

Figure 95. Pull Request final

An interesting thing to notice is that if you click on the “Files Changed” tab on this Pull Request, you’ll get the “unified” diff — that is, the total aggregate difference that would be introduced to your main branch if this topic branch was merged in. In `git diff` terms, it basically automatically shows you `git diff master...<branch>` for the branch this Pull Request is based on. See [Determining What Is Introduced](#) for more about this type of diff.

The other thing you’ll notice is that GitHub checks to see if the Pull Request merges cleanly and provides a button to do the merge for you on the server. This button only shows up if you have write access to the repository and a trivial merge is possible. If you click it GitHub will perform a “non-fast-forward” merge, meaning that even if the merge **could** be a fast-forward, it will still create a merge commit.

If you would prefer, you can simply pull the branch down and merge it locally. If you merge this branch into the masterbranch and push it to GitHub, the Pull Request will automatically be closed.

This is the basic workflow that most GitHub projects use. Topic branches are created, Pull Requests are opened on them, a discussion ensues, possibly more work is done on the branch and eventually the request is either closed or merged.

Note Not Only Forks

It's important to note that you can also open a Pull Request between two branches in the same repository. If you're working on a feature with someone and you both have write access to the project, you can push a topic branch to the repository and open a Pull Request on it to the master branch of that same project to initiate the code review and discussion process. No forking necessary.

Advanced Pull Requests

Now that we've covered the basics of contributing to a project on GitHub, let's cover a few interesting tips and tricks about Pull Requests so you can be more effective in using them.

Pull Requests as Patches

It's important to understand that many projects don't really think of Pull Requests as queues of perfect patches that should apply cleanly in order, as most mailing list-based projects think of patch series contributions. Most GitHub projects think about Pull Request branches as iterative conversations around a proposed change, culminating in a unified diff that is applied by merging.

This is an important distinction, because generally the change is suggested before the code is thought to be perfect, which is far more rare with mailing list based patch series contributions. This enables an earlier conversation with the maintainers so that arriving at the proper solution is more of a community effort. When code is proposed with a Pull Request and the maintainers or community suggest a change, the patch series is generally not re-rolled, but instead the difference is pushed as a new commit to the branch, moving the conversation forward with the context of the previous work intact.

For instance, if you go back and look again at [Pull Request final](#), you'll notice that the contributor did not rebase his commit and send another Pull Request. Instead they added new commits and pushed them to the existing branch. This way if you go back and look at this Pull Request in the future, you can easily find all of the context of why decisions were made. Pushing the "Merge" button on the site purposefully creates a merge commit that references the Pull Request so that it's easy to go back and research the original conversation if necessary.

Keeping up with Upstream

If your Pull Request becomes out of date or otherwise doesn't merge cleanly, you will want to fix it so the maintainer can easily merge it. GitHub will test this for you and let you know at the bottom of every Pull Request if the merge is trivial or not.



This pull request contains merge conflicts that must be resolved.
Only those with [write access](#) to this repository can merge pull requests.



Figure 96. Pull Request does not merge cleanly

If you see something like [Pull Request does not merge cleanly](#), you'll want to fix your branch so that it turns green and the maintainer doesn't have to do extra work.

You have two main options in order to do this. You can either rebase your branch on top of whatever the target branch is (normally the master branch of the repository you forked), or you can merge the target branch into your branch.

Most developers on GitHub will choose to do the latter, for the same reasons we just went over in the previous section. What matters is the history and the final merge, so rebasing isn't getting you much other than a slightly cleaner history and in return is **far** more difficult and error prone.

If you want to merge in the target branch to make your Pull Request mergeable, you would add the original repository as a new remote, fetch from it, merge the main branch of that repository into your topic branch, fix any issues and finally push it back up to the same branch you opened the Pull Request on.

For example, let's say that in the "tonychacon" example we were using before, the original author made a change that would create a conflict in the Pull Request. Let's go through those steps.

```
$ git remote add upstream https://github.com/schacon/blink (1)
```

```
$ git fetch upstream (2)
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
Unpacking objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
```



```

From https://github.com/schacon/blink
* [new branch]      master      -> upstream/master

$ git merge upstream/master (3)
Auto-merging blink.ino
CONFLICT (content): Merge conflict in blink.ino
Automatic merge failed; fix conflicts and then commit the result.

$ vim blink.ino (4)
$ git add blink.ino
$ git commit
[slow-blink 3c8d735] Merge remote-tracking branch 'upstream/master' \
    into slower-blink

$ git push origin slow-blink (5)
Counting objects: 6, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 682 bytes | 0 bytes/s, done.
Total 6 (delta 2), reused 0 (delta 0)
To https://github.com/tonychacon/blink
ef4725c..3c8d735  slower-blink -> slow-blink

```

1. Add the original repository as a remote named upstream.
2. Fetch the newest work from that remote.
3. Merge the main branch of that repository into your topic branch.
4. Fix the conflict that occurred.
5. Push back up to the same topic branch.

Once you do that, the Pull Request will be automatically updated and re-checked to see if it merges cleanly.

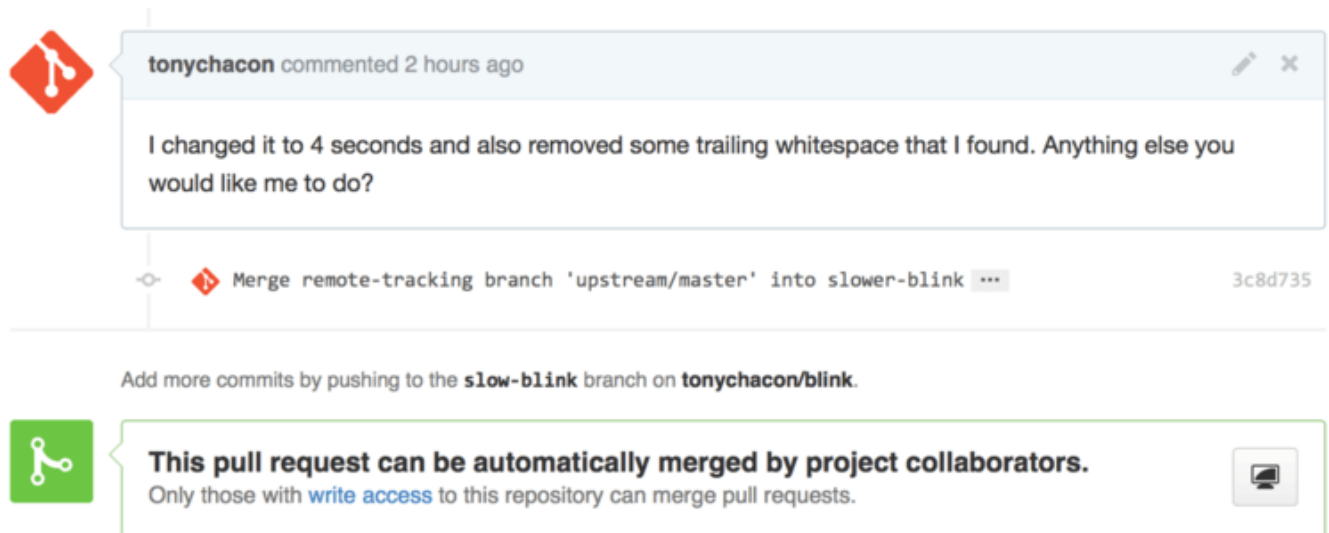


Figure 97. Pull Request now merges cleanly

One of the great things about Git is that you can do that continuously. If you have a very long-running project, you can easily merge from the target branch over and over again and only have to deal with conflicts that have arisen since the last time that you merged, making the process very manageable.

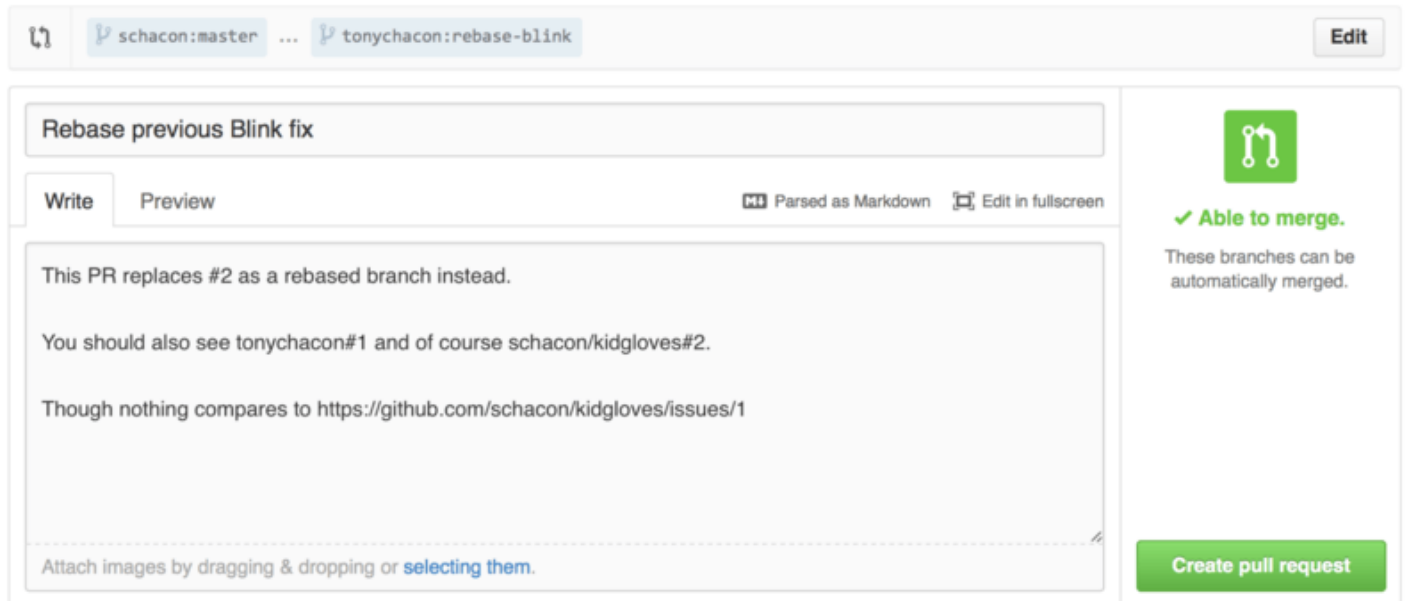
If you absolutely wish to rebase the branch to clean it up, you can certainly do so, but it is highly encouraged to not force push over the branch that the Pull Request is already opened on. If other people have pulled it down and done more work on it, you run into all of the issues outlined in [The Perils of Rebasing](#). Instead, push the rebased branch to a new branch on GitHub and open a brand new Pull Request referencing the old one, then close the original.

References

Your next question may be “How do I reference the old Pull Request?”. It turns out there are many, many ways to reference other things almost anywhere you can write in GitHub.

Let's start with how to cross-reference another Pull Request or an Issue. All Pull Requests and Issues are assigned numbers and they are unique within the project. For example, you can't have Pull Request #3 *and* Issue #3. If you want to reference any Pull Request or Issue from any other one, you can simply put #<num> in any comment or description. You can also be more specific if the Issue or Pull request lives somewhere else; write username#<num> if you're referring to an Issue or Pull Request in a fork of the repository you're in, or username/repo#<num> to reference something in another repository.

Let's look at an example. Say we rebased the branch in the previous example, created a new pull request for it, and now we want to reference the old pull request from the new one. We also want to reference an issue in the fork of the repository and an issue in a completely different project. We can fill out the description just like [Cross references in a Pull Request](#).



Rebase previous Blink fix

Write Preview

Parsed as Markdown Edit in fullscreen

This PR replaces #2 as a rebased branch instead.

You should also see tonychacon#1 and of course schacon/kidgloves#2.

Though nothing compares to https://github.com/schacon/kidgloves/issues/1

Attach images by dragging & dropping or selecting them.

✓ Able to merge.
These branches can be automatically merged.

Create pull request

Figure 98. Cross references in a Pull Request

When we submit this pull request, we'll see all of that rendered like [Cross references rendered in a Pull Request](#).

Rebase previous Blink fix #4



tonychacon wants to merge 2 commits into schacon:master from tonychacon:rebase-blink



Conversation 0



Commits 2



Files changed 1



tonychacon commented just now

This PR replaces #2 as a rebased branch instead.

You should also see tonychacon#1 and of course schacon/kidgloves#2.

Though nothing compares to schacon/kidgloves#1



tonychacon added some commits 4 hours ago



three seconds is better

afe904a



remove trailing whitespace

a5a7751

Figure 99. Cross references rendered in a Pull Request

Notice that the full GitHub URL we put in there was shortened to just the information needed.

Now if Tony goes back and closes out the original Pull Request, we can see that by mentioning it in the new one, GitHub has automatically created a traceback event in the Pull Request timeline. This means that anyone who visits this Pull

Request and sees that it is closed can easily link back to the one that superseded it. The link will look something like [Link back to the new Pull Request in the closed Pull Request timeline](#).

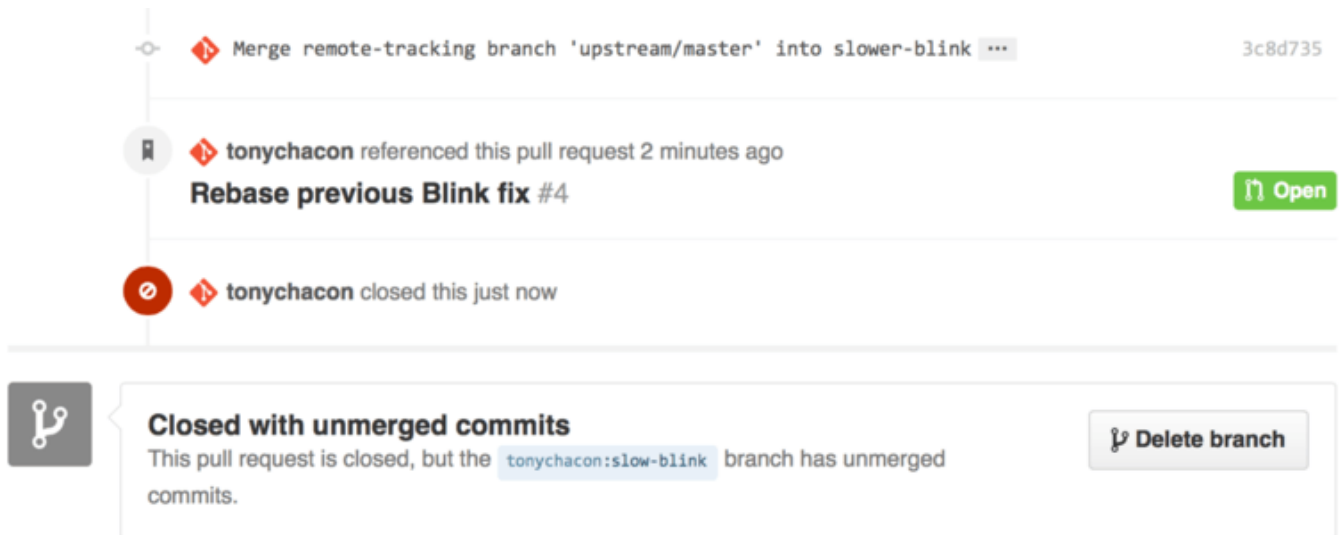


Figure 100. Link back to the new Pull Request in the closed Pull Request timeline

In addition to issue numbers, you can also reference a specific commit by SHA-1. You have to specify a full 40 character SHA-1, but if GitHub sees that in a comment, it will link directly to the commit. Again, you can reference commits in forks or other repositories in the same way you did with issues.

GitHub Flavored Markdown

Linking to other Issues is just the beginning of interesting things you can do with almost any text box on GitHub. In Issue and Pull Request descriptions, comments, code comments and more, you can use what is called “GitHub Flavored Markdown”. Markdown is like writing in plain text but which is rendered richly.

See [An example of GitHub Flavored Markdown as written and as rendered](#) for an example of how comments or text can be written and then rendered using Markdown.

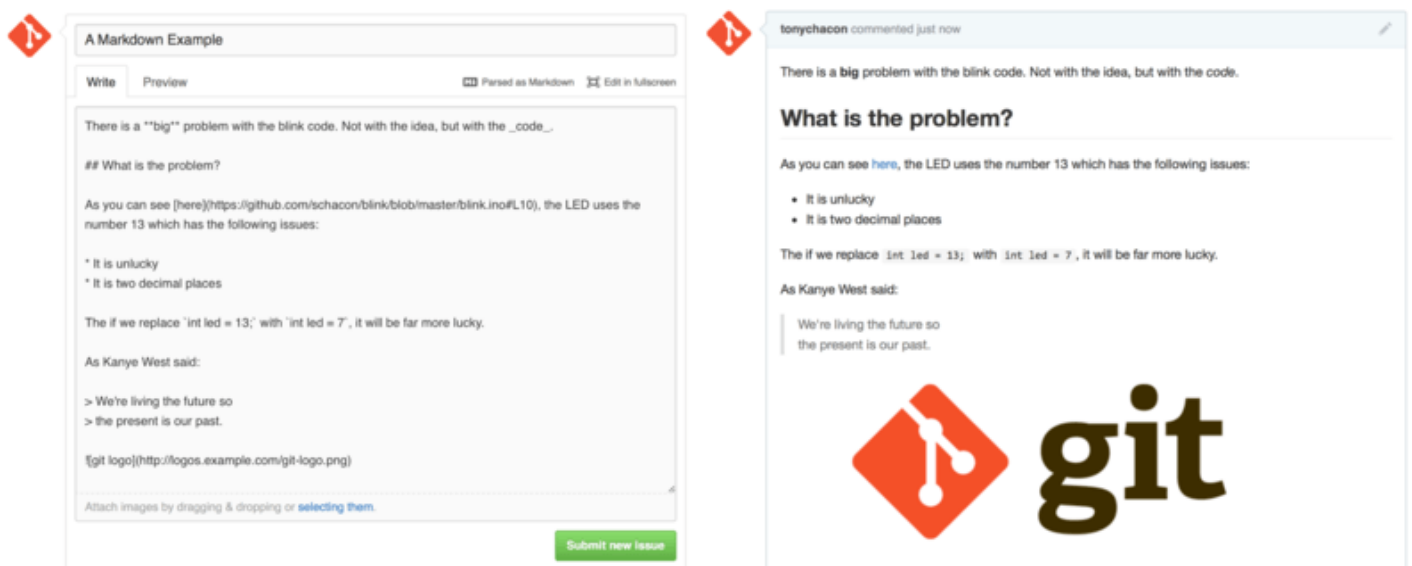


Figure 101. An example of GitHub Flavored Markdown as written and as rendered

The GitHub flavor of Markdown adds more things you can do beyond the basic Markdown syntax. These can all be really useful when creating useful Pull Request or Issue comments or descriptions.

Task Lists

The first really useful GitHub specific Markdown feature, especially for use in Pull Requests, is the Task List. A task list is a list of checkboxes of things you want to get done. Putting them into an Issue or Pull Request normally indicates things that you want to get done before you consider the item complete.

You can create a task list like this:

- [X] Write the code
- [] Write all the tests
- [] Document the code

If we include this in the description of our Pull Request or Issue, we'll see it rendered like [Task lists rendered in a Markdown comment](#).



Figure 102. Task lists rendered in a Markdown comment

This is often used in Pull Requests to indicate what all you would like to get done on the branch before the Pull Request will be ready to merge. The really cool part is that you can simply click the checkboxes to update the comment — you don't have to edit the Markdown directly to check tasks off.

What's more, GitHub will look for task lists in your Issues and Pull Requests and show them as metadata on the pages that list them out. For example, if you have a Pull Request with tasks and you look at the overview page of all Pull Requests, you can see how far done it is. This helps people break down Pull Requests into subtasks and helps other people track the progress of the branch. You can see an example of this in [Task list summary in the Pull Request list](#).



Figure 103. Task list summary in the Pull Request list

These are incredibly useful when you open a Pull Request early and use it to track your progress through the implementation of the feature.

Code Snippets

You can also add code snippets to comments. This is especially useful if you want to present something that you *could* try to do before actually implementing it as a commit on your branch. This is also often used to add example code of what is not working or what this Pull Request could implement.

To add a snippet of code you have to “fence” it in backticks.

```
```java
for(int i=0 ; i < 5 ; i++)
{
 System.out.println("i is : " + i);
}
```
```

If you add a language name like we did there with 'java', GitHub will also try to syntax highlight the snippet. In the case of the above example, it would end up rendering like [Rendered fenced code example](#).



tonychacon commented just now



Perhaps we should try something like:

```
for(int i=0 ; i < 5 ; i++)  
{  
    System.out.println("i is : " + i);  
}
```

Figure 104. Rendered fenced code example

Quoting

If you're responding to a small part of a long comment, you can selectively quote out of the other comment by preceding the lines with the > character. In fact, this is so common and so useful that there is a keyboard shortcut for it. If you highlight text in a comment that you want to directly reply to and hit the `r` key, it will quote that text in the comment box for you.

The quotes look something like this:

```
> Whether 'tis Nobler in the mind to suffer  
> The Slings and Arrows of outrageous Fortune,
```

How big are these slings and in particular, these arrows?

Once rendered, the comment will look like [Rendered quoting example](#).



schacon commented 2 minutes ago

Owner

That is the question—
Whether 'tis Nobler in the mind to suffer
The Slings and Arrows of outrageous Fortune,
Or to take Arms against a Sea of troubles,
And by opposing, end them? To die, to sleep—
No more; and by a sleep, to say we end
The Heart-ache, and the thousand Natural shocks
That Flesh is heir to?



tonychacon commented 10 seconds ago



```
> Whether 'tis Nobler in the mind to suffer  
> The Slings and Arrows of outrageous Fortune,  
  
How big are these slings and in particular, these arrows?
```

Figure 105. Rendered quoting example

Emoji

Finally, you can also use emoji in your comments. This is actually used quite extensively in comments you see on many GitHub Issues and Pull Requests. There is even an emoji helper in GitHub. If you are typing a comment and you start with a `:` character, an autocompleter will help you find what you're looking for.

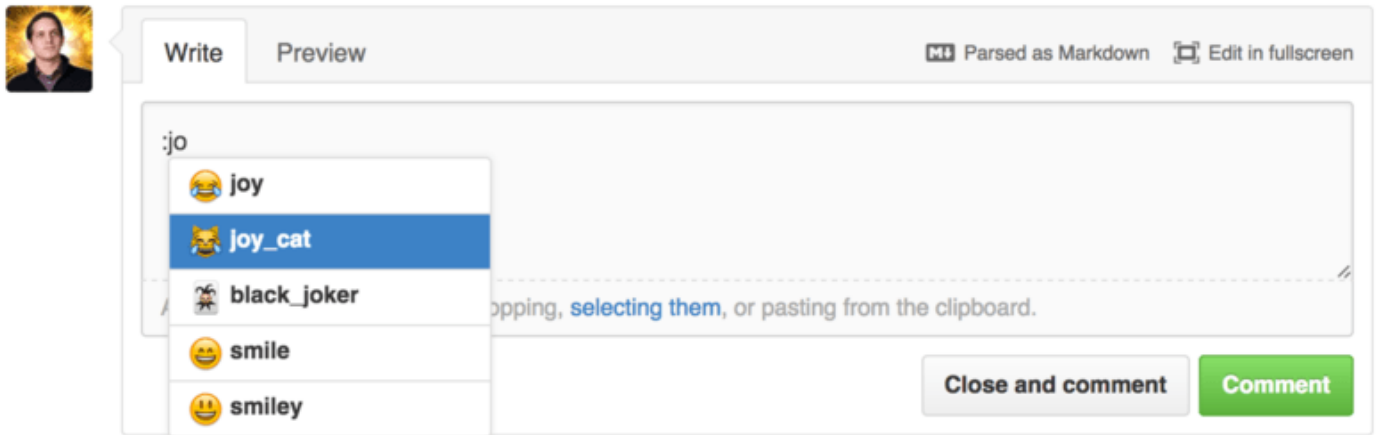


Figure 106. Emoji autoCompleter in action

Emojis take the form of `:<name>:` anywhere in the comment. For instance, you could write something like this:

```
I :eyes: that :bug: and I :cold_sweat:.
```

```
:trophy: for :microscope: it.
```

```
:+1: and :sparkles: on this :ship:, it's :fire::poop:!
```

```
:clap::tada::panda_face:
```

When rendered, it would look something like [Heavy emoji commenting](#).

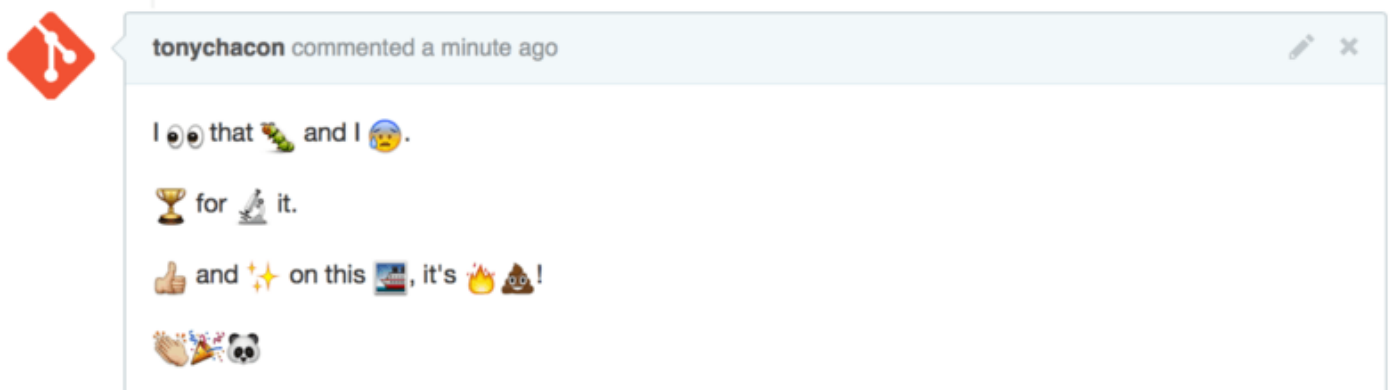


Figure 107. Heavy emoji commenting

Not that this is incredibly useful, but it does add an element of fun and emotion to a medium that is otherwise hard to convey emotion in.

There are actually quite a number of web services that make use of emoji characters these days. A great cheat sheet to reference to find emoji that expresses what you want to say can be found at:

<https://www.webfx.com/tools/emoji-cheat-sheet/>

Images

This isn't technically GitHub Flavored Markdown, but it is incredibly useful. In addition to adding Markdown image links to comments, which can be difficult to find and embed URLs for, GitHub allows you to drag and drop images into text areas to embed them.

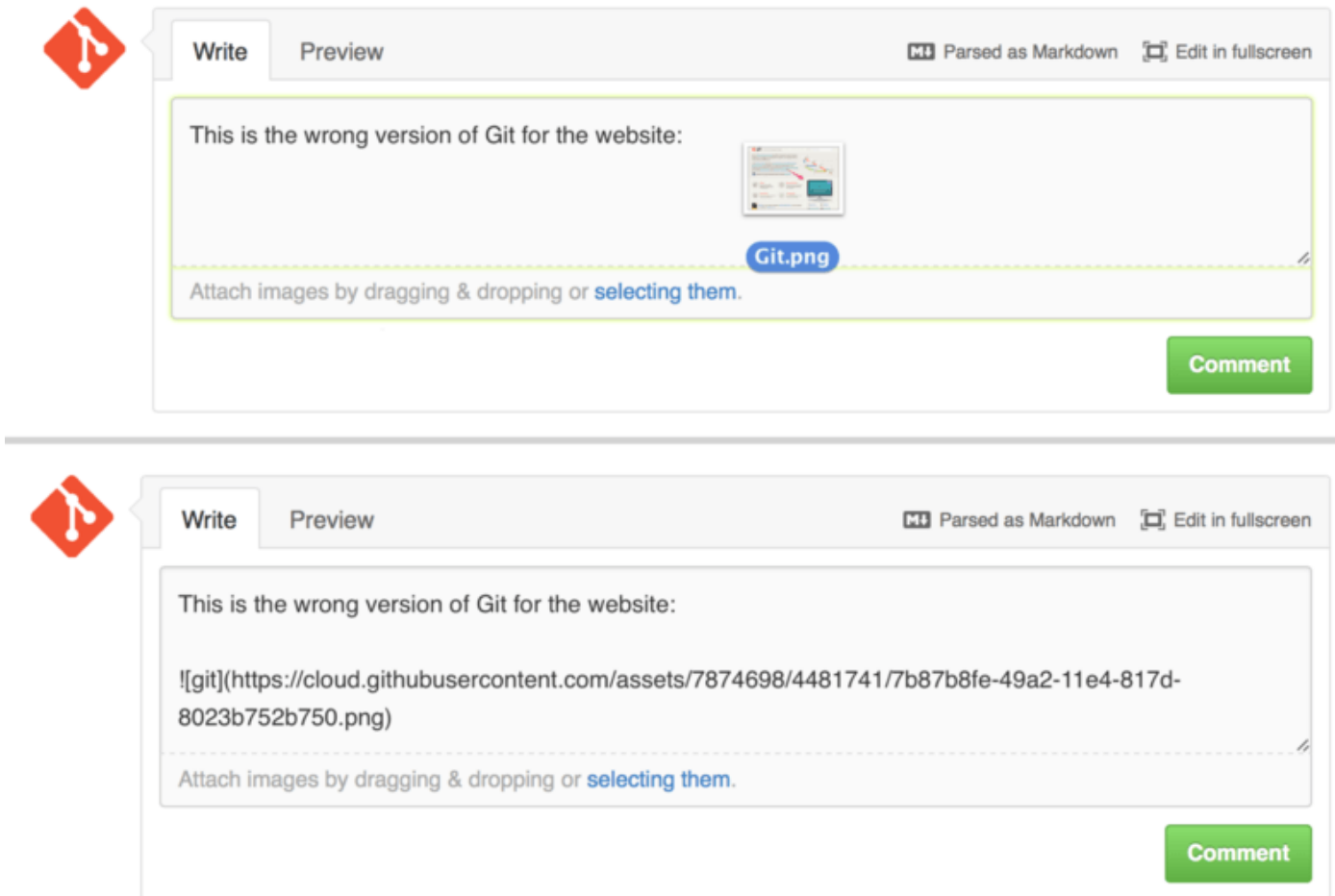


Figure 108. Drag and drop images to upload them and auto-embed them

If you look at [Drag and drop images to upload them and auto-embed them](#), you can see a small “Parsed as Markdown” hint above the text area. Clicking on that will give you a full cheat sheet of everything you can do with Markdown on GitHub.

Keep your GitHub public repository up-to-date

Once you’ve forked a GitHub repository, your repository (your “fork”) exists independently from the original. In particular, when the original repository has new commits, GitHub informs you by a message like:

This branch is 5 commits behind progit:master.

But your GitHub repository will never be automatically updated by GitHub; this is something that you must do yourself. Fortunately, this is very easy to do.

One possibility to do this requires no configuration. For example, if you forked from <https://github.com/progit/progit2.git>, you can keep your master branch up-to-date like this:

```
$ git checkout master (1)
$ git pull https://github.com/progit/progit2.git (2)
$ git push origin master (3)
```

1. If you were on another branch, return to master.
2. Fetch changes from <https://github.com/progit/progit2.git> and merge them into master.
3. Push your master branch to origin.

This works, but it is a little tedious having to spell out the fetch URL every time. You can automate this work with a bit of configuration:

```
$ git remote add progit https://github.com/progit/progit2.git (1)
$ git branch --set-upstream-to=progit/master master (2)
$ git config --local remote.pushDefault origin (3)
```

1. Add the source repository and give it a name. Here, I have chosen to call it progit.

2. Set your master branch to fetch from the `progit` remote.
3. Define the default push repository to `origin`.

Once this is done, the workflow becomes much simpler:

```
$ git checkout master (1)
$ git pull (2)
$ git push (3)
```

1. If you were on another branch, return to master.
2. Fetch changes from `progit` and merge changes into master.
3. Push your master branch to `origin`.

This approach can be useful, but it's not without downsides. Git will happily do this work for you silently, but it won't warn you if you make a commit to `master`, pull from `progit`, then push to `origin` — all of those operations are valid with this setup. So you'll have to take care never to commit directly to `master`, since that branch effectively belongs to the upstream repository.

[prev](#) | [next](#)