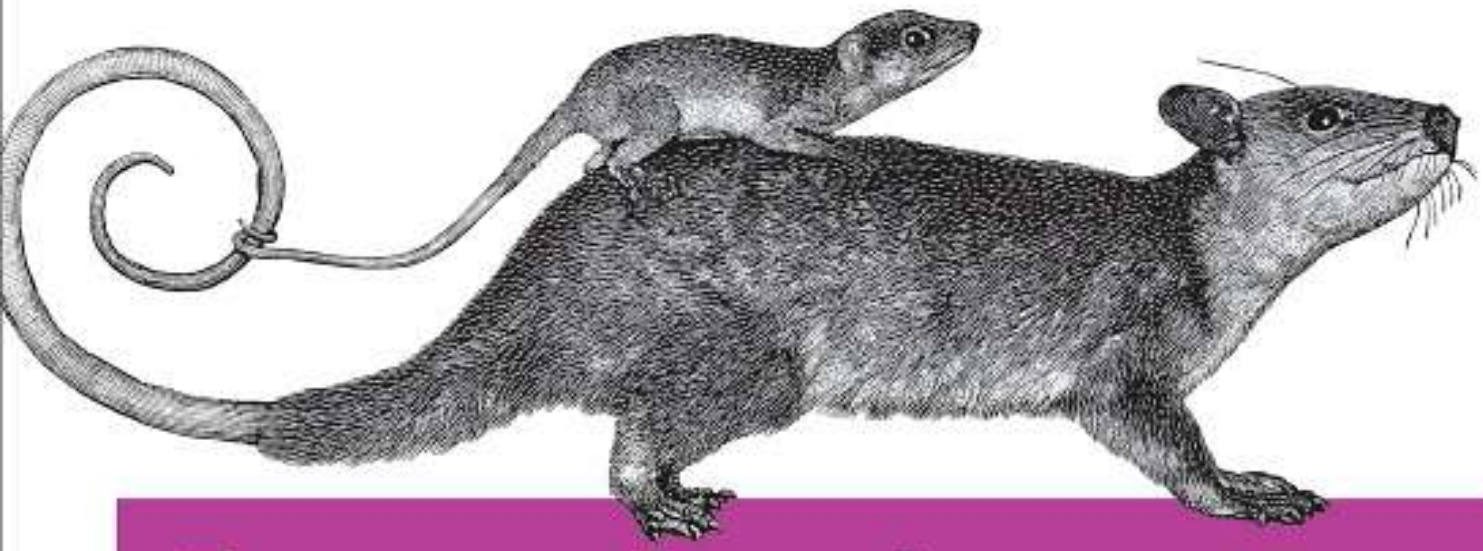
O'REILLY®

2nd Edition



Introducing GitHub

A NON-TECHNICAL GUIDE

Brent Beer

Introducing GitHub

Second Edition

A Non-Technical Guide

Brent Beer

Introducing GitHub

by Brent Beer

Printed in the United States of America.

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com/safari). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Virginia Wilson

Production Editor: Kristen Brown

Copyeditor: Rachel Head

Proofreader: Charles Roumeliotis

Indexer: Ellen Troutman-Zaig

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

November 2014: First Edition

December 2017: Second Edition

Revision History for the Second Edition

- 2017-11-28: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781491981818 for release details.

978-1-491-98181-8

[LSI]

Preface

GitHub is changing the way that software gets built. Conceived originally as a way to make it easier for developers to contribute to open source projects, GitHub is rapidly becoming the default platform for software development. More than just a tool for storing source code, GitHub provides a range of powerful tools for specifying, discussing, and reviewing software and other text-based documents.

Who This Book Is For

If you are working with developers on a software project, this book is for you, whether you are a:

- *Business stakeholder* who wants to have a sense of how your project is going
- *Product or project manager* who needs to ensure that software is delivered on time and within budget
- *Designer* who needs to deliver anything from mockups to HTML/CSS for a project
- *Copywriter* who's adding marketing copy or other content to a site or an app
- *Lawyer* who's reviewing the legal implications of a project or writing the terms and conditions or privacy policy
- *Team member* who needs to review, comment on, and/or contribute to the project
- *Technical writer* who's making sure a project's documentation is up to date for all collaborators to help them get their jobs done
- *Developer* who is new to using GitHub and wants to learn how to collaborate using GitHub in a team

If you need to view the progress of a piece of software while it's being developed, if you would like to be able to comment on the progress, and if you'd like to have the option of contributing changes to the project, this book will show you how to effectively collaborate with a software development team by using GitHub.

Beyond Software

While GitHub is still primarily used to collaborate on the development of software, it's also a great way for a team to collaborate on a wide range of projects. From the authoring of books (like this one) and the distribution of models for 3D printing to the crafting of legislation, whenever you have a

team of people collaborating on a collection of documents, you should consider using GitHub to manage the process. Our examples will assume that you’re working on software because that is currently the most common use case, but this book is the perfect guide to collaborating via GitHub on any kind of project.

Who This Book Is Not For

This book is designed to teach the core skills required to collaborate effectively using GitHub. If you are already familiar with forking, cloning, and using feature branches and pull requests for collaboration, you probably won't learn that much.

Equally, if you are looking for an in-depth introduction to the Git version control system, this is not the book that you are looking for. This book covers just enough Git to do the job of introducing GitHub, but it's not a comprehensive introduction to Git. For that you should read the excellent *Version Control with Git* by Jon Loeliger and Matthew McCullough (O'Reilly).

How to Use This Book

This book has deliberately been made as concise as possible. You should be able to read it pretty quickly. If you want to gain the confidence that comes from really understanding what GitHub is about and how to use it, try to read the book from start to finish.

However, I know that you're busy. If you're in a rush, start by skimming the first chapter. Chapter 1 gives you a brief introduction to Git, GitHub, and some key terms that you'll need to understand to make sense of the rest of the book. Then feel free to just jump into whatever chapters seem relevant. I've tried to write the book so that each chapter runs you through specific workflows, so you should be able to read just the chapter you need to complete a particular task.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

> Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`

Used for program listings, as well as within paragraphs to refer to program elements and the names of branches.

Tip

This element signifies a tip or suggestion.

Note

This element signifies a general note.

Warning

This element indicates a warning or caution.

O’Reilly Safari

Safari (formerly Safari Books Online) is a membership-based training and reference platform for enterprise, government, educators, and individuals.

Members have access to thousands of books, training videos, Learning Paths, interactive tutorials, and curated playlists from over 250 publishers, including O’Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Adobe, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, and Course Technology, among others.

For more information, please visit *http://oreilly.com/safari*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O’Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

There is a web page for this book, which lists errata, examples, and any additional information. You can access this page at *http://bit.ly/intro-github-2e*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

I’d first like to thank my wife Lindsay. Thank you for supporting my late nights while working on this book and encouraging me to do the best job that I could. You’re the best wife, partner, and support anyone could ask for.

I’d also like to thank my parents. My mom for her constant encouragement for reading, without which I may never have found a love for it. And my dad. Without him letting me watch him work on our computer, entertaining me with the Oscar the Grouch trash can utility on our Macintosh, and encouraging me to learn how to program, I would not be in the field I am today.

Lastly, I want to thank Peter Bell. Peter, without the hard work you put into the first edition and encouragement to do this second edition, this book wouldn’t exist. Thank you!

Chapter 1. Introduction

In this chapter I'll start by introducing Git and GitHub. What are they, what is the difference between them, and why would you want to use them? I'll then introduce some other common terms that you'll often hear mentioned when people are discussing GitHub. That way you'll be able to understand and participate in discussions about your projects more easily.

What Is Git?

Git is a version control system. A *version control system* is a piece of software designed to keep track of the changes made to files over time. More specifically, Git is a *distributed* version control system, which means that everyone working with a project in Git has a copy of the full history of the project, not just the current state of the files.

What Is GitHub?

GitHub is a platform where you can upload a copy of your Git repository (often shortened to *repo*), hosted either on GitHub.com, by your company on a cloud provider (like Azure, AWS, or IBM Bluemix), or on your company's own servers behind its firewall. But more than just uploading your Git repositories, it allows you to collaborate much more easily with other people on your projects. It does that by providing a centralized location to share the repository, a web-based interface to view it, and features like *forking*, *Pull Requests*, *Issues*, *Projects*, and *GitHub Wikis* that allow you to specify, discuss, and review changes with your team more effectively.

Why Use Git?

Even if you're working on your own, if you are editing text files, there are a number of benefits to using Git, including the following:

The ability to undo changes
: If you make a mistake, you can go back to a previous point in time to recover an earlier version of your work.

A complete history of all the changes
: If you ever want to see what your project looked like a day, week, month, or year ago, you can *check out* a previous version of the project

to see exactly what the state of the files was back then.

Documentation of why changes were made
: Often it's hard to remember *why* a change was made. With *commit messages* in Git, it's easy to document for future reference why you're making a change.

The confidence to change anything
: Because it's easy to recover a previous version of your project, you can have the confidence to make any changes you want. If they don't work out, you can always get back to an earlier version of your work.

Multiple streams of history
: You can create different *branches* of history to experiment with different changes to your content or to build out different features independently. You can then *merge* those back into the main project history (`master` branch) once they're done, or delete them if they end up not working out.

Working on a team, you get an even wider range of benefits when using Git to keep track of your changes. Some of the key benefits of Git when working with a team are:

The ability to resolve conflicts
: With Git, multiple people can work on the same file at the same time. Usually Git will be able to merge the changes automatically. If it can't, Git will show you what the conflicts are and you will hopefully be able to easily resolve them.

Independent streams of history
: Different people working on the project can work on different branches, allowing them to work on separate features independently and then merge the features when they're done.

Why Use GitHub?

GitHub is much more than just a place to store your Git repositories. It provides a number of additional benefits, including the ability to do the following:

Document requirements
: Using issues, you can either document bugs or specify new features that you'd like to have your team develop.

Collaborate on independent streams of history
: Using branches and pull requests, you can collaborate on different branches or features.

Review work in progress
: By looking at the list of pull requests, you can see all of the different features that are currently being worked on; by clicking any given pull request you can see the latest changes and all of the discussions about the changes, check the status of an integration like a Continuous Integration (CI) server, or even add your own review to approve changes before they are accepted.

See team progress
: Skimming the *pulse* or looking through the *commit history* allows you to see what the team has been working on.

Key Concepts

There are a number of key concepts that you'll need to understand to work effectively with Git and GitHub. Here is a list of some of the most common terms, with a short description of each and an example of how they might be used in conversation:

Commit
: Whenever you save your changes in one or more files, you can create a new commit in Git. A commit is like a snapshot of your entire repository at that point in time, not just of one or two files. So naturally, after you change those files, you will want to update the repository by taking a new snapshot. *Example usage: "Let's commit these changes and push them up to GitHub."*

Commit message
: Every time you make a commit, you need to supply a message that describes *why* the change was made. That commit message is invaluable

when trying to understand later why a certain change was implemented. *Example usage: “Make sure to include Susan’s comment about the new SEC guidelines in the commit message.”*

Branch
: A branch is an independent series of commits off to one side that you can use to try out an experiment or create a new feature. *Example usage: “Let’s create a branch to implement the new search functionality.”*

`master` branch
: Whenever you create a new Git project, there is a default branch created called `master`. This is the branch that your work should end up on eventually, once it’s ready to push to production. *Example usage: “Remember never to commit directly to `master`.”*

Feature (or topic) branch
: Whenever you’re building a new piece of functionality, you’ll create a branch to work on it. That is called a *feature branch*. *Example usage: “We’ve got way too many feature branches. Let’s focus on getting one or two of these finished and into production.”*

Release branch
: If you have a manual QA process or have to support old versions of your software for your customers, you might need a release branch as a place to make any necessary fixes or updates. There is no technical difference between a feature or release branch, but the distinction is useful when talking about a project with your team. *Example usage: “We’ve got to fix the security bug on all of our supported release branches.”*

Merge
: A merge is a way to take completed work from one branch and incorporate it into another branch. Most commonly you’ll merge a feature branch into the `master` branch. *Example usage: “Great job on the ‘my account’ feature. Could you merge it into `master` so we can push it to production?”*

Tag

A tag is a reference to a specific historic commit. Tags are most often used to document production releases so you know exactly which versions of the code went into production and when. *Example usage: "Let's tag this release and push it to production."*

Checkout

Checking out enables you to go to a different version of the project's history and see the files as of that point in time. Most commonly you'll check out a branch to see all of the work that has been done on it, but any commit can be something you check out. *Example usage: "Could you check out the last release tag? There's a bug in production that I need you to replicate and fix."*

Pull request

Originally, a pull request was used to request that someone else review the work you'd completed on a branch and then merge it into `master`. Now, pull requests are often used earlier in the process to start a discussion about a possible feature. *Example usage: "Go create a pull request for the new voting feature so we can see what the rest of the team thinks about it."*

Issue

GitHub has a feature called Issues that can be used to discuss features, track bugs, or both. *Example usage: "You're right, the login doesn't work on an iPhone. Could you create an issue on GitHub documenting the steps to replicate the bug?"*

Wiki

Originally developed by Ward Cunningham, wikis are a lightweight way of creating web pages with simple links between them. GitHub projects often use wikis for documentation. *Example usage: "Could you add a page to the wiki to explain how to configure the project to run on multiple servers?"*

Clone

Often you'll want to download a copy of a project from GitHub so you can work on it locally. The process of copying the repository to your computer is called *cloning*. *Example usage: "Could you clone the repo, fix the bug, and then push the fix back up to GitHub later tonight?"*

Fork

Sometimes you don't have the necessary permission to make changes directly to a project. Perhaps it's an open source project written by people you don't know, or a project written by another group at your company that you don't work with much. If you want to submit changes to such a project, first you need to make a copy of the project under *your* user account on GitHub. That process is called *forking* the repository. You can then clone it, make changes, and submit them back to the original project using a pull request. *Example usage: "I'd love to see how you'd rewrite the home page marketing copy. Fork the repo and submit a pull request with your proposed changes."*

Don't worry if all the terminology seems overwhelming at first. Once you start working with some real projects, it'll all make a lot more sense! In the next chapter we'll look at the various elements of a GitHub project and how you can use them to get a sense of progress on a project.

Chapter 2. Viewing

In this chapter we'll look at how you can view the state of a project to see what's going on, so you can find areas that could be improved or so you can be familiar with the interface when you're ready to contribute. We'll use the popular Bootstrap open source project as an example.

Introducing the Repository Page

Bootstrap is a repository that allows developers to quickly develop attractive web applications. Go view the repository on GitHub. There is a lot of information on the home page. Let's start by reviewing some of the most important elements on the page.

Take a look at Figure 2-1. One of the first things you see looking at the top left of the page is that the project name is "bootstrap" and that it's owned by a user (or in this case an organization) called "twbs." If you were to go to *https://github.com/twbs*, you'd see a list of all of the repositories hosted by that organization at GitHub. To the left of the organization name on the repository page you'll also see an icon that makes it clear that this is a public repository that anyone can see. A lot of the projects you work on will have a closed lock icon and the word "Private" at the end, signifying that they are private and can be viewed only by people who have been explicitly given access.

In the upper-right corner of the page in Figure 2-1, you can see that at the time the screenshot was taken, 7,065 people were *watching* the repository to get notified every time new changes were made to it; 115,203 people had *starred* it to mark it as one of their favorite repositories; and 53,899 people had *forked* the repository, making their own copy on GitHub where they could make changes to the code and share them with others if need be, or contribute back to this parent repository.

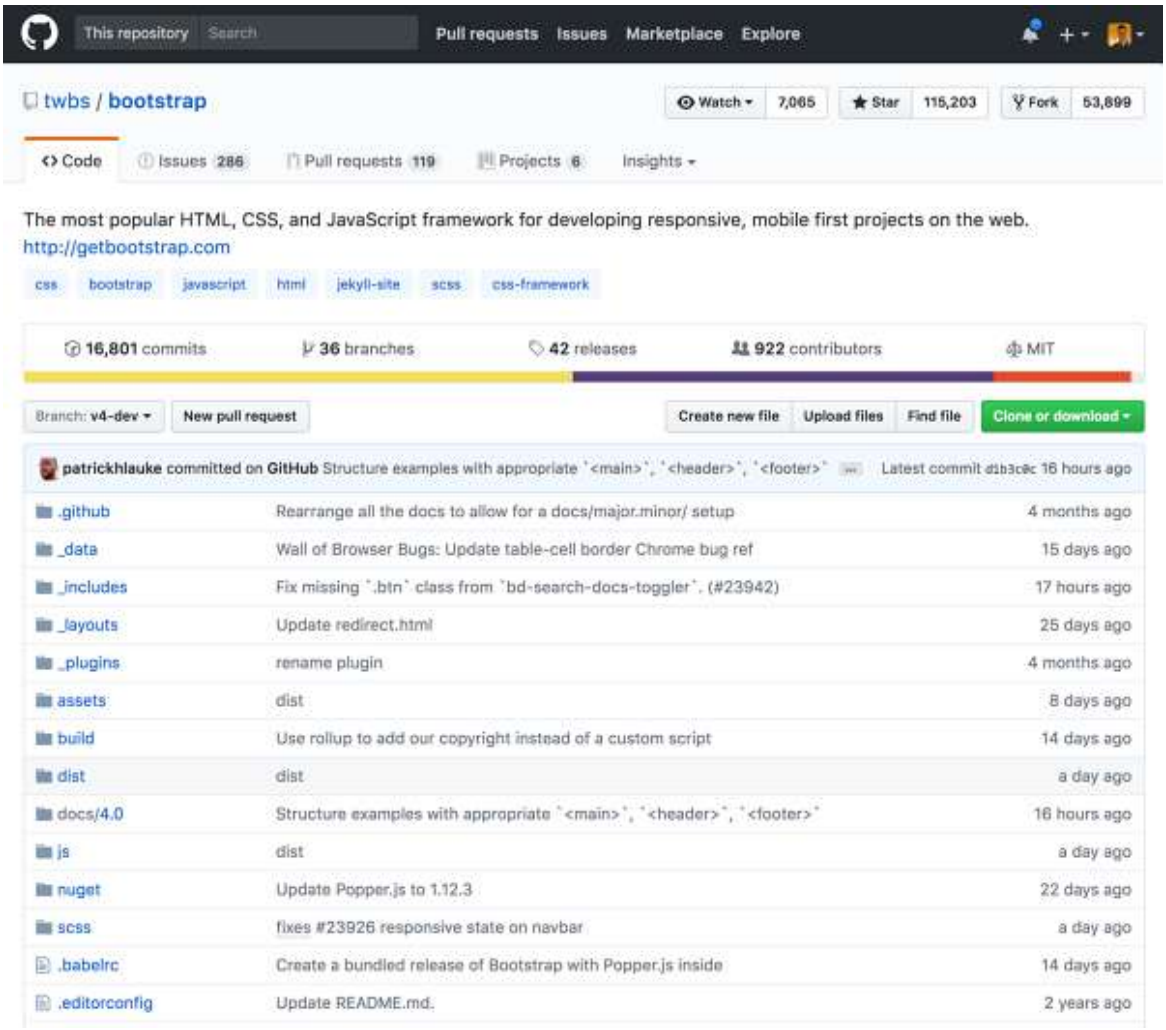


Figure 2-1. The Bootstrap home page on GitHub

Under the repository name and the tabs, you can see a short description of the repository and some topics that are related to it. Below that, you can see that there have been a total of 16,801 changes to the repository (commits), 36 different streams of history are currently being developed (branches), 42 versions of the software have been recommended over time for people to use (releases), and 922 people have written some part of the code (contributors).

You can also see that we're currently viewing the `v4-dev` branch (this repository's default branch), that we're in the root *bootstrap* folder, that the latest commit on `v4-dev` was "Structure examples with appropriate `<main>`, `<header>`, `<footer>`," and that the commit was made by

GitHub user “patrickhlauke”. As you look further down the figure, you can see the folders (sometimes called *directories*) and files that are in the root (top-level) folder in the project.

Viewing the README.md File

If there is a file in the root of a project named *README.md*, the contents of that file will be displayed just below the list of folders and files on the repository home page. This file provides an introduction to the project and additional information that may be useful to collaborators, such as how to install the software, how to run any automated tests, how to use the code, and how to make contributions to the project.

These days, *README* files will often also include badges—images used to show the current state of things like the automated test suite, to let you know the current state of the repository. In Figure 2-2, which shows the status section further down in the *README.md* file, you can see the versions of other projects that Bootstrap depends on. This section also shows details on the state of the dependencies (peerDependencies, devDependencies), package registries (meteor, packagist, nuget), chat service (Slack), and the versions of browsers and operating systems that Bootstrap should work for.

Figure 2-2. The status section of Bootstrap’s README.md file

Viewing the Commit History

Looking at the commit history is a great way to get a sense of the most recent small units of work that have been completed on any given branch.

Go to the Bootstrap page on GitHub and click the “16,801 commits” link (the number of commits will have changed by the time you do this). You’ll see a list of commits, with the most recent first (see Figure 2-3).

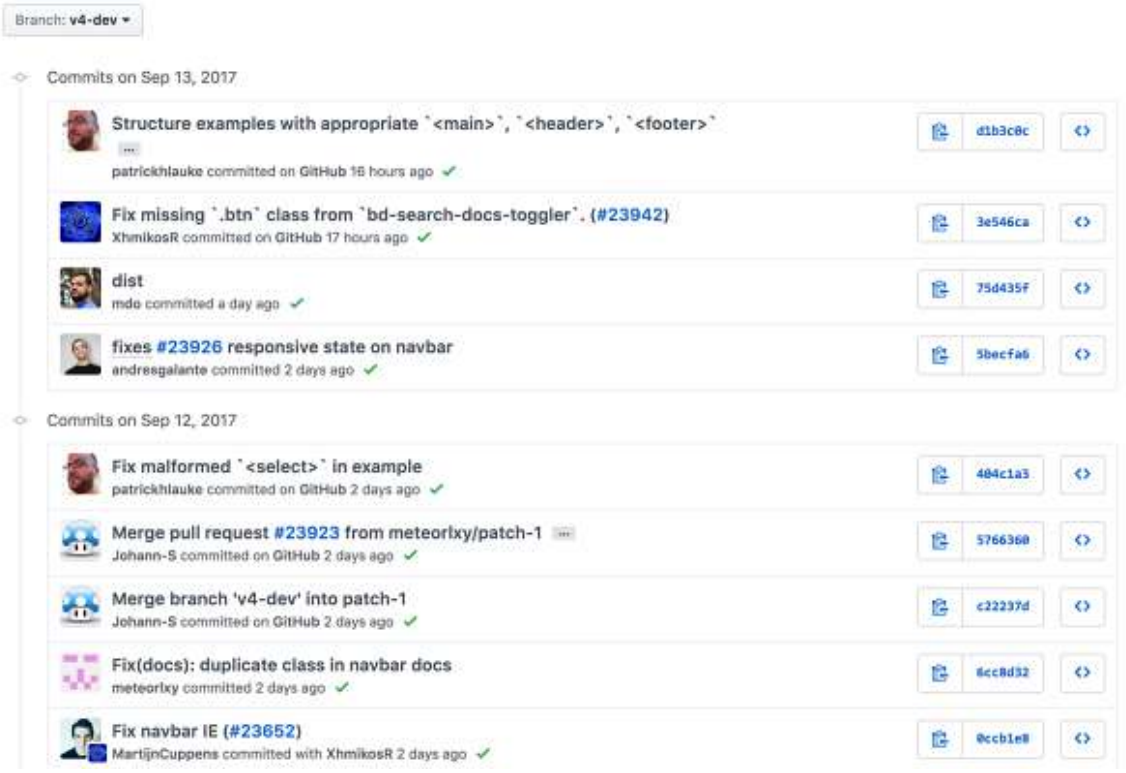


Figure 2-3. A list of the recent commits to the project

Clicking any of the commits will show you the commit message, which should explain *why* the change was made (see Figure 2-4). Below that you will see who made the change; which branch it is on now; the pull request it was brought in from, if any (#23899 in this case); and each file that was added, removed, or modified as part of the commit, with content that was removed displaying in red and content that was added displaying in green.

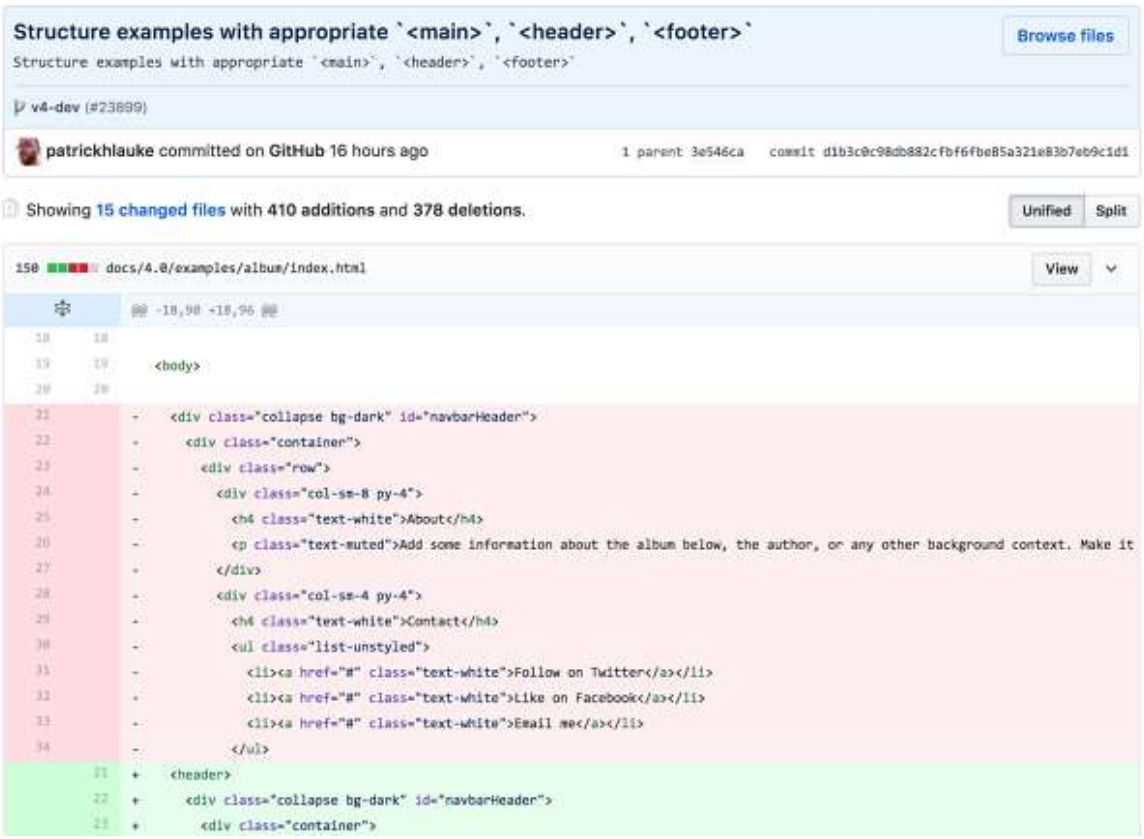


Figure 2-4. A recent commit in the repository

Viewing Pull Requests

Pull requests give you a sense of the current work in progress by showing you a comparison of work being done on one branch that is proposed to merge with another. Click the “Pull requests” tab along the top of the page and you’ll see a list of open pull requests. These are the outstanding features or fixes that people are currently working on (see Figure 2-5).

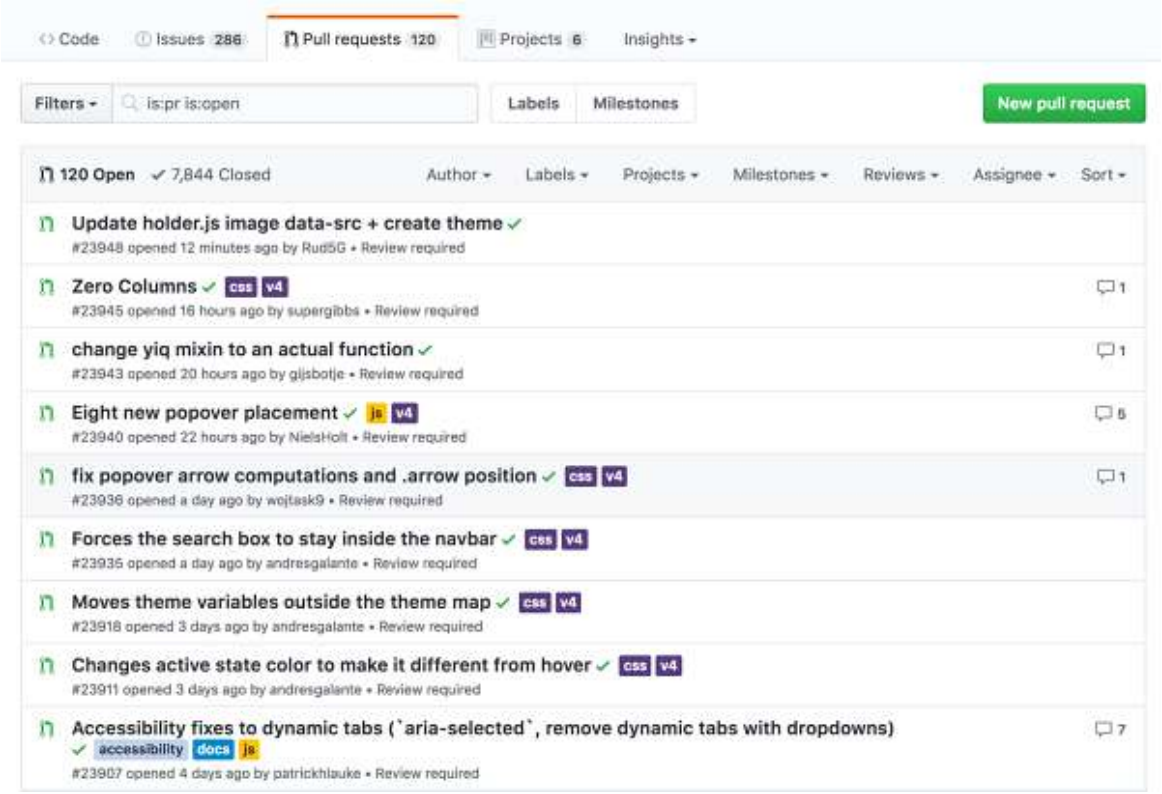


Figure 2-5. Open pull requests for Bootstrap

Click one of the pull requests, and you'll see a page displaying a short title describing the pull request and a longer description of it. There will be one or more commits with the proposed changes, and there may be a number of comments from people discussing the proposed changes (see Figure 2-6).

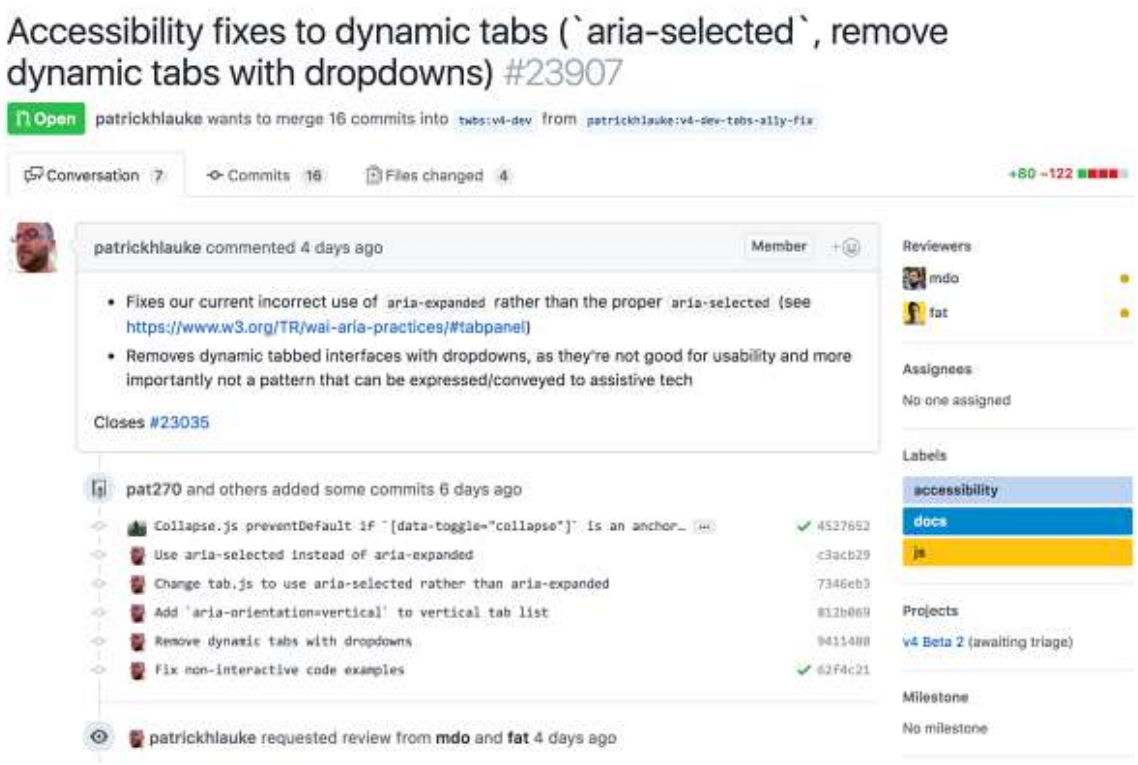


Figure 2-6. A recent pull request

Looking at the pull requests is a great way to get a sense of what people are working on now and the current state of those changes—whether bug fixes or proposed features.

Viewing Issues

While pull requests give you a sense of the current bug fixes and features being worked on, issues can give you a wider sense of the outstanding work that still needs to be done on a repository. Pull requests are often linked to an issue, but there will usually also be issues that nobody has started working on yet, so they don’t yet have associated pull requests.

If you click the Issues tab at the top of the page, by default you’ll see a list of all of the open issues (see Figure 2-7).

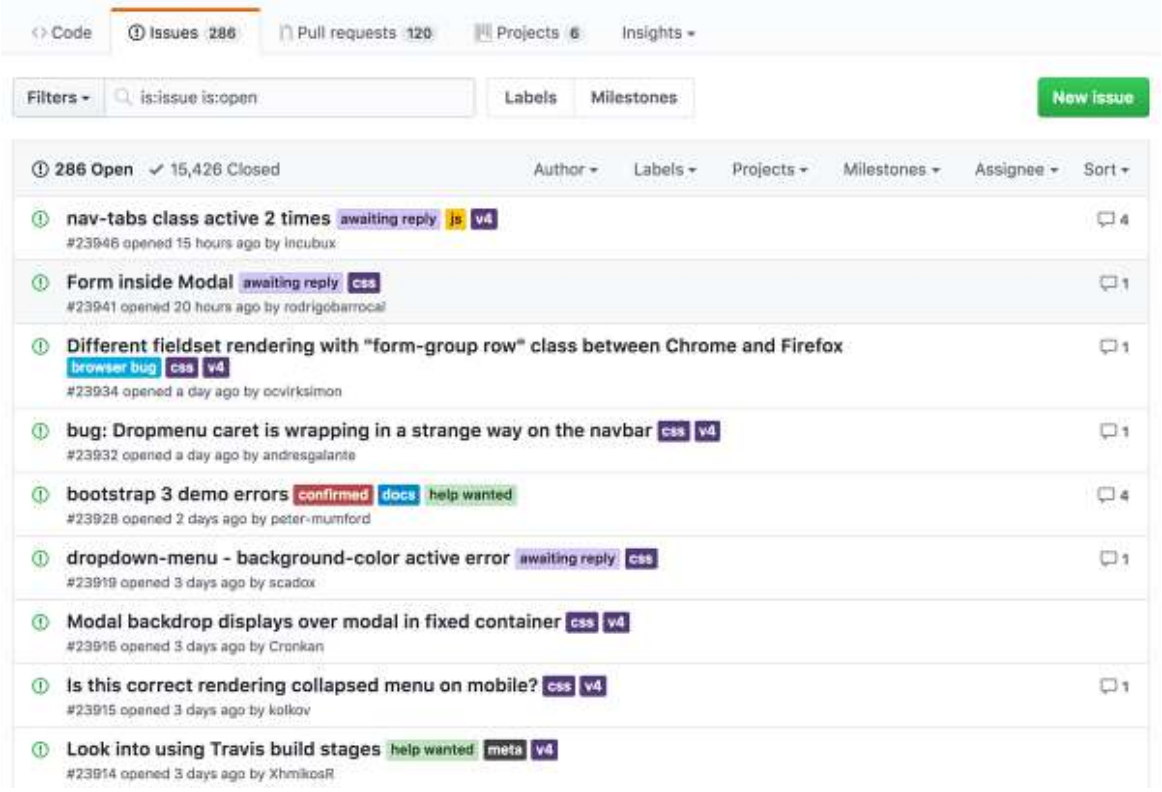


Figure 2-7. Open issues for the repository

Click an issue and, similar to a pull request, you'll see the title, description, and any comments related to the issue. If any work has been done and pushed to GitHub where the commit message references an issue, it'll show up on the Issues page so you can see what's being done. In Figure 2-8 someone appears to be having a problem with one of the Bootstrap features.

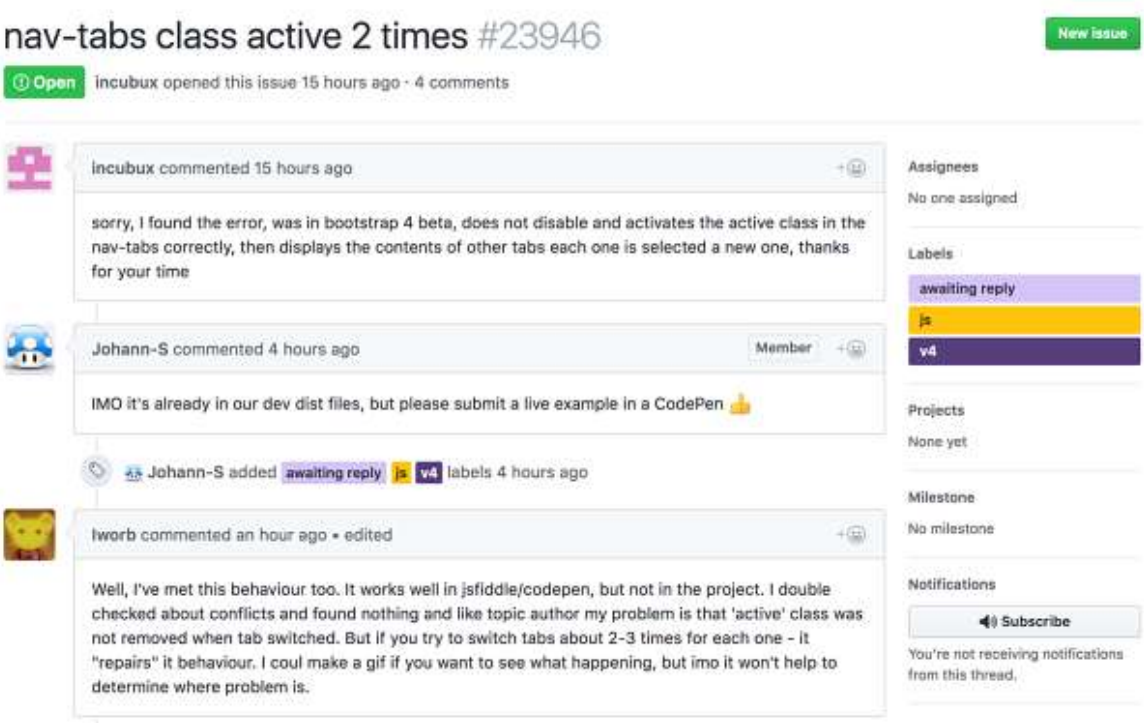


Figure 2-8. A recent issue

Viewing Projects

Issues and pull requests are great for individual work, but sometimes people want to do longer-form project work at a project management level. That's exactly what GitHub projects are for. Click on the Projects tab to get a look at any active projects being worked on inside of Bootstrap, and you'll be taken to a page like Figure 2-9.

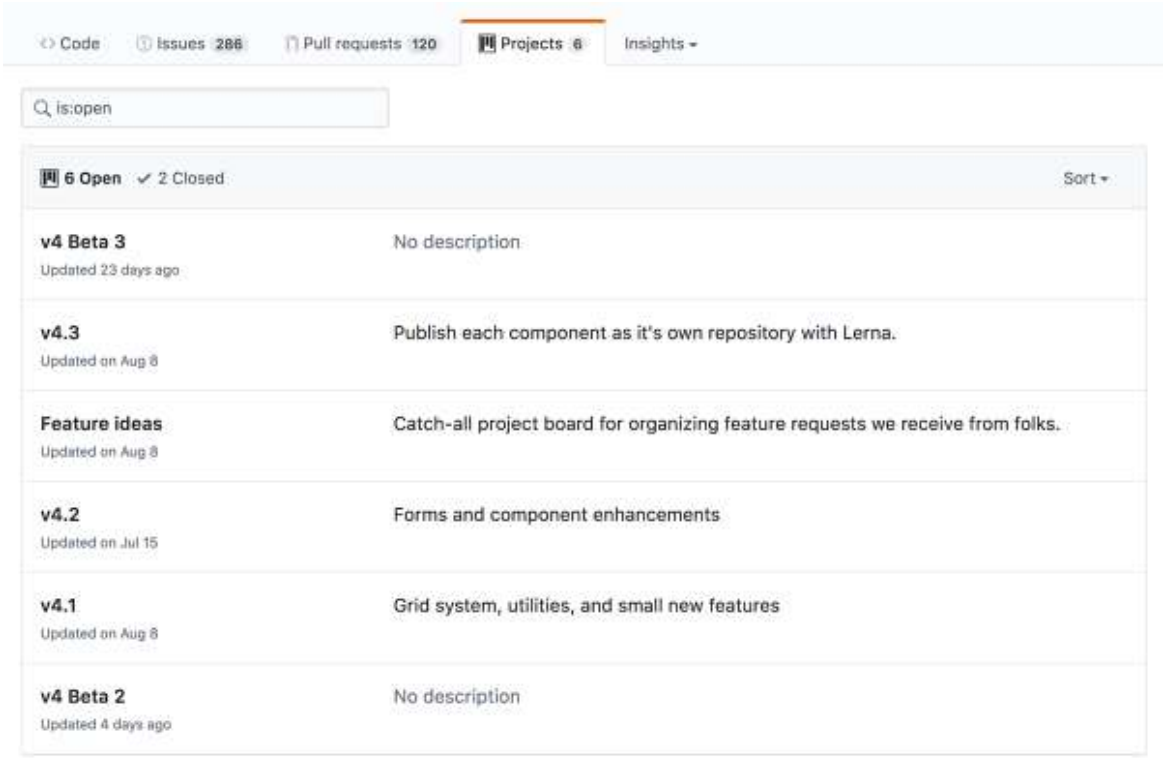


Figure 2-9. Bootstrap's projects

If you click an individual project, you'll be taken to its board, where you'll see the columns of work and the different cards (notes, issues, and pull requests) that are in those columns (see Figure 2-10).

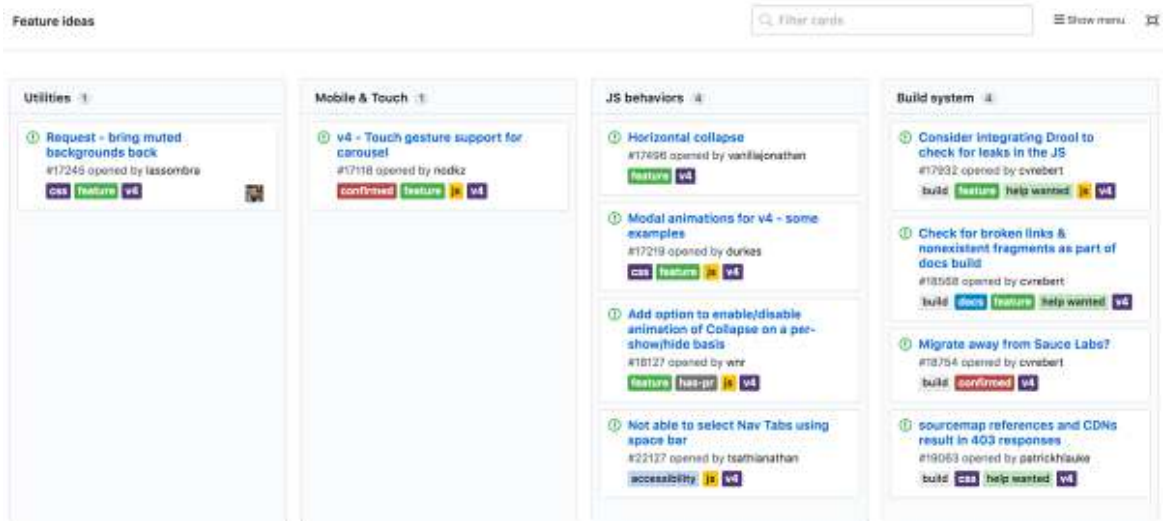


Figure 2-10. An active Bootstrap project board

Viewing Insights

GitHub’s Insights tab allows you to get a sense of the work that has been done on a repository over a longer period of time.

Viewing the Pulse

The *pulse* is a great way to get a sense of the recent activity on a project. This is the page displayed by default when you click the Insights tab. Notice in the top right of Figure 2-11 that you can customize the pulse to be for the last day, three days, week, or month.

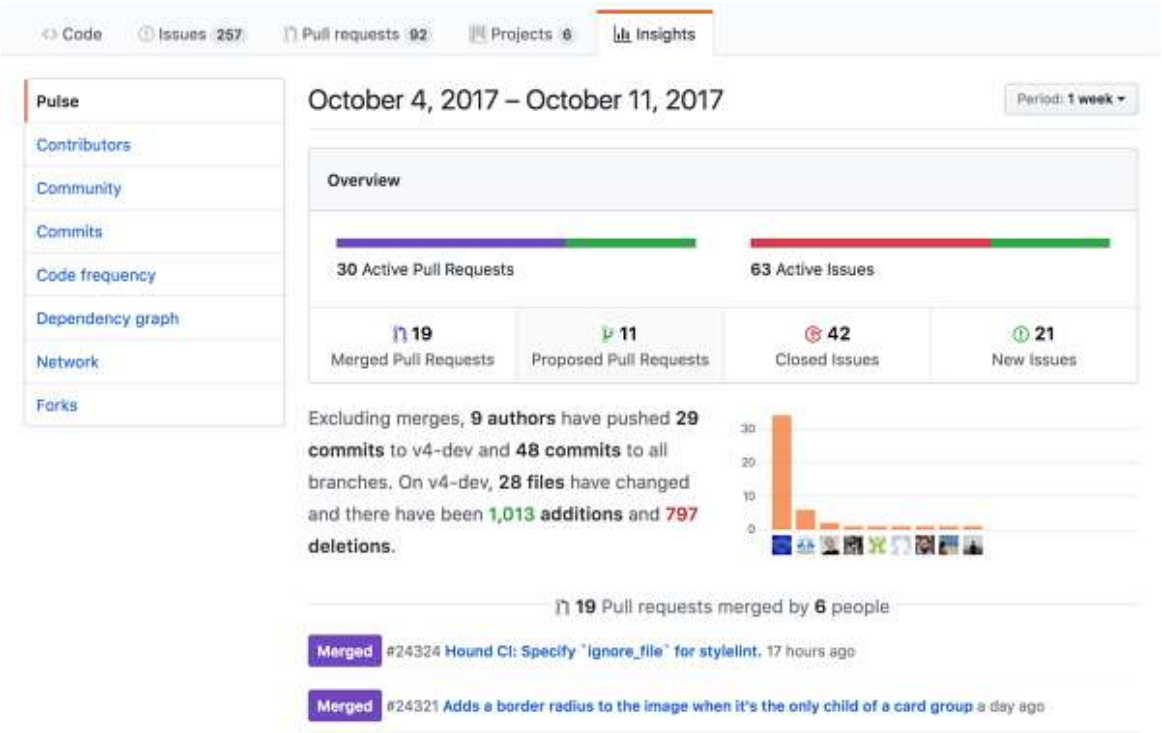


Figure 2-11. The pulse for Bootstrap

The pulse starts with an overview of the number of pull requests that have been merged (completed) and proposed (added) during the selected time period. It also shows how many issues were closed and opened. It’s important to understand that when the pulse refers to the number of active pull requests and issues, this is not the outstanding number of each but rather the number of requests and issues that have been started or finished in the time period you selected. For example, at the time of writing, Bootstrap had 19 merged and 11 proposed pull requests, for a total of 30 “active” pull requests in the last week, but it had a total of 92 open pull

requests (as indicated by the number on the “Pull requests” tab at the top of the page).

The next paragraph on the screen is a concise summary of recent changes, listing the number of authors, commits on `v4-dev`, total commits on all branches, and files modified on the `v4-dev` branch. It then gives you the number of lines of content that have been added or removed, although it’s important to realize that if a line of text in a file is modified, Git will treat it as if one line was removed and another, different line was added in its place.

To the right is a bar chart showing the contributors who have made the most commits during the specified period. Below is a list of the titles of the merged and proposed pull requests, followed by the closed and then opened issues. The pulse view ends with a list of “unresolved conversations,” which is a list of all of the issues and pull requests that have received additional comments but have not yet been closed.

The Contributors Graph

The contributors graph (Figure 2-12) shows you the number of contributions (commits, additions, or deletions) to the repository. It shows a graph for all of the contributions, followed by smaller graphs showing the contributions by individual developers—from the most to the least prolific.

Figure 2-12. The contributors graph for Bootstrap

The default commits graph shows the total number of commits that have been made to the `v4-dev` branch. It's important to realize that it shows only the commits that have been merged into the `v4-dev` branch. If you have someone on your team who has been working on a feature branch all week and whose work has not yet been merged in, none of those contributions will show up until they are ready for release and have been merged into the `v4-dev` branch.

By default, the time period for the graph is the entire lifetime of the repository. If you'd like to pick a shorter interval, just click the starting point you'd like on the main graph and then drag and release on the time you'd like the new graph to end. Figure 2-13 shows the results of doing this to focus on the commits over the last year or so. You can see that the main graph at the top of the page stays the same, but at the top left it shows the time period we're focused on (November 30, 2016, through October 12, 2017). The commit graphs of the individual contributors show the number of commits by each and how they were spread out over that time period.

Figure 2-13. Narrowing the interval of the contributors graph

There is no standard size for a commit. A good rule of thumb is that if developers are writing code, as opposed to researching a problem or testing something, they should probably be committing every 5 to 10 minutes. However, depending on the team you're working with, you might find that some developers create fewer commits than others, even if they're doing a similar amount of work. If that is the case, you might want to change the "contributions" type for your contributor graphs to additions or deletions. In that way, you'll get a sense of the number of lines of code that the developers have added to or removed from the repository. If they modify a line, it will show up as a deletion of the old line and an addition of the new one.

The Community Profile

Your repository's community profile is an insight into the way your repository presents itself to new contributors, as well as those looking to just use your repository. Take a look at Figure 2-14 to see how the Bootstrap repository is doing in this area.

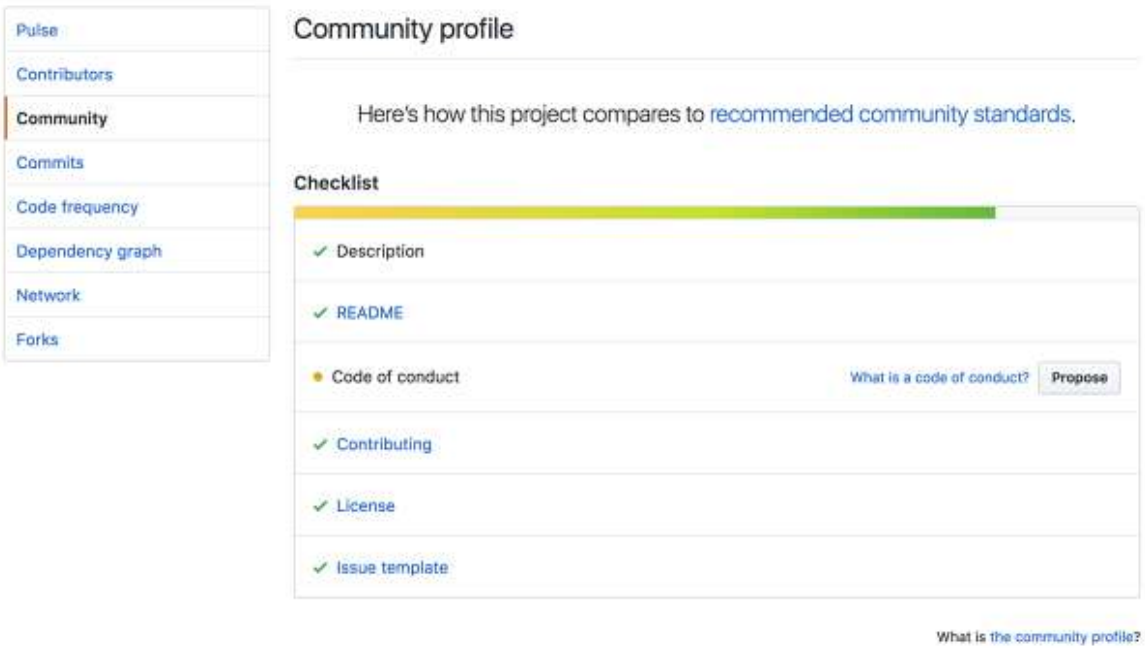


Figure 2-14. The community profile for Bootstrap

As of the writing of this book, community profiles are relatively new, so this means that not every repository is going to have a complete checklist. However, their importance is critical to fostering a safe and welcoming community in addition to having a solid open source initiative. Similar to documentation fixes in a repository, this is a great avenue to get started on making your first contribution once you feel comfortable doing so.

The Commits Graph

The commits graph (Figure 2-15) shows the number of commits per week over the past year, giving a very rough indication of activity and how it has varied over time.

The first reason to look at the commits graph is to get a sense of how many commits per week there have been over the past year. It starts with a bar graph showing one bar per week and is a great way to see cyclical or long-term trends. Is the number of commits in your repository slowly decreasing? If you have more developers, is the number of commits consistently increasing? Are most of your commits in the last week of every month, or are there seasonal trends? This graph can give you good insight

into how the number of commits—which is a very rough proxy for productivity—are varying over time.

Figure 2-15. The commits graph for Bootstrap

Below the bar graph is a line graph showing the average number of commits on each day of the week over the last year. This graph can be useful for getting a sense of the cadence over the course of an average week. Are people not committing on Mondays because of too many meetings? Are they making most of their commits on a Thursday ahead of your Friday "demo days," or are they working too much on the weekend, which isn't good for long-term sustainability?

The Code Frequency Graph

The code frequency graph (Figure 2-16) shows you the number of lines added to and removed from your repository over time and is particularly helpful for identifying large changes to your code base.

Often when developers are doing a big refactoring, they'll add and delete hundreds or even thousands of lines of code per commit, whereas in the usual course of business, a commit will probably contain only a few lines of added, modified, or deleted code. When such a refactoring is going on, the number of commits might not change much, but the number of lines added

and deleted will spike—so if you want to get a sense of when the biggest changes happened to your code base, you should start by having a look at the code frequency graph. For example, you can see in Figure 2-16 that a big refactoring was done in February/March of 2013.

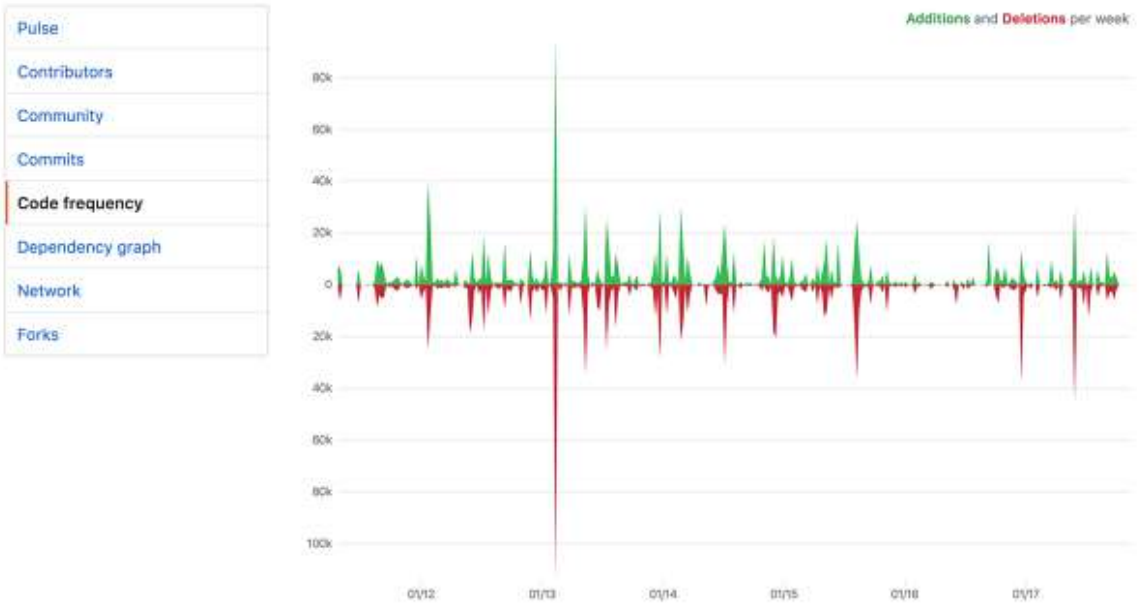


Figure 2-16. The code frequency graph for Bootstrap

The Dependency Graph

The Dependencies tab of the dependency graph page is a great place for you to get a well-formatted overview of what other components your repository depends on. This is important because you may want to contribute a patch to another repository your repository depends on to add some functionality. You can see a list of dependencies for GitHub's Linguist repository in Figure 2-17. This page is especially useful when you want to be informed about security vulnerabilities, which GitHub will display for you.

If you use a private repository on GitHub.com, you have to opt in and choose to contribute your private information to GitHub and the community as a whole in order to enrich the ecosystem and allow others to see how useful their repositories may be.

The Dependents tab of the dependency graph page is useful to find out where your repository is being used and what packages may depend on it.

In Figure 2-18, which again shows GitHub's Linguist repository, you can see that a few applications and packages depend on this repository.

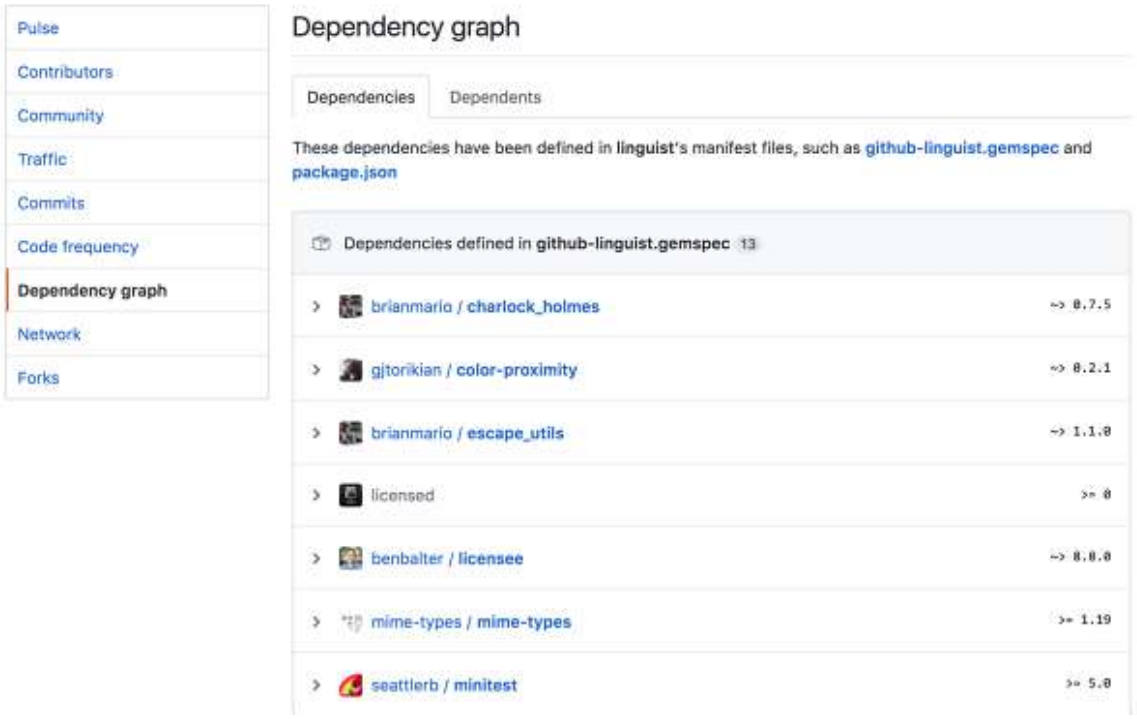


Figure 2-17. The dependencies for the github/linguist repository

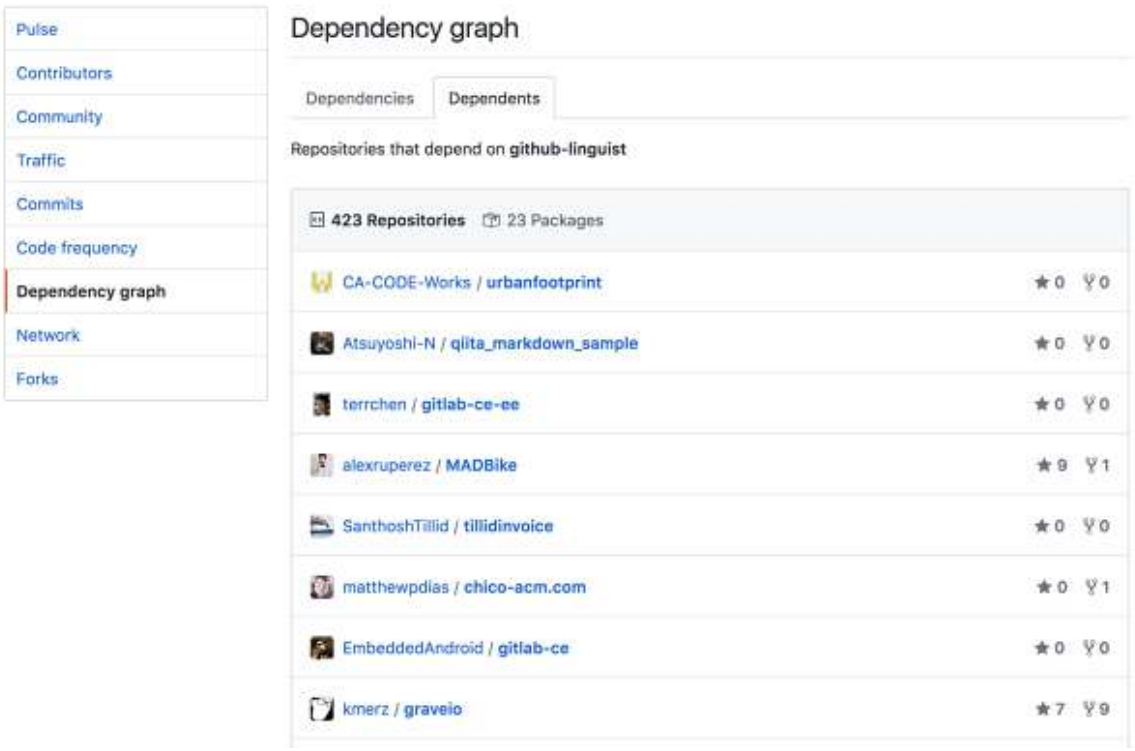


Figure 2-18. The dependents list for the github/linguist repository

Being aware of what repositories or packages depend on your code is important for beng a good citizen on GitHub, and also gives you a good idea of who else you may have an impact on when you make any updates to your code.

The Network Graph

The network graph (Figure 2-19) shows the number of branches and commits on those branches throughout a repository's history. It also shows any forks that contributors have created. Because Bootstrap is such a popular project, the network graph shown here is actually of the GitHub Desktop repository.

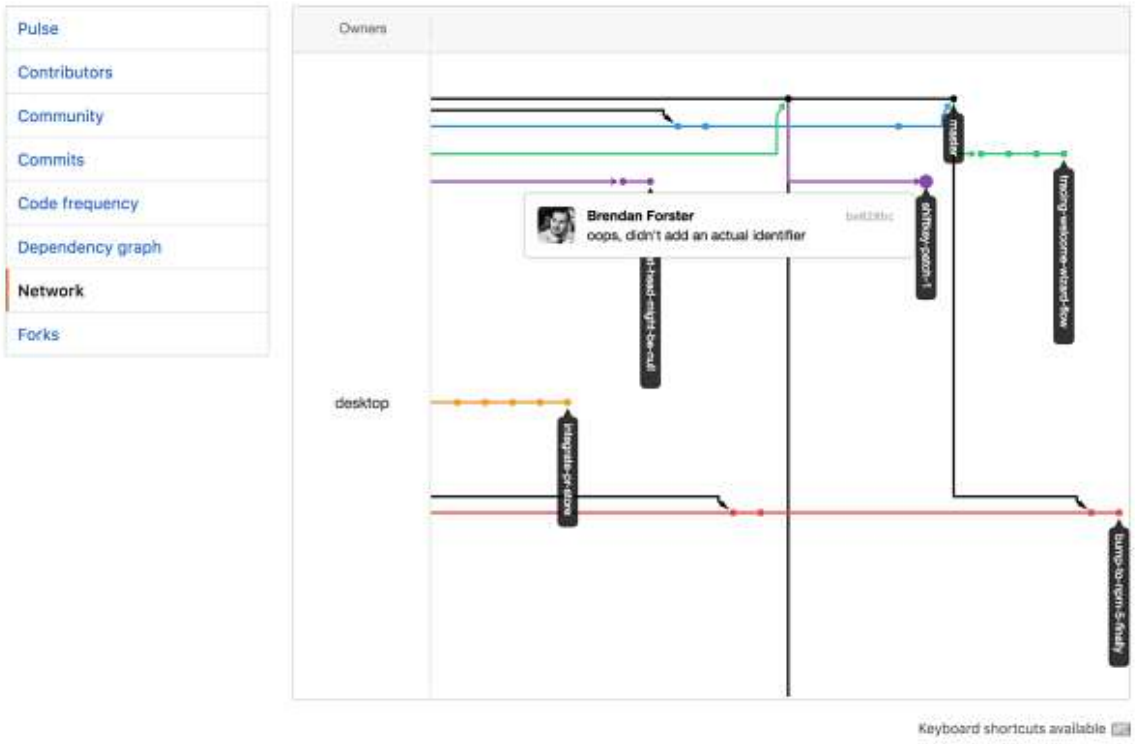


Figure 2-19. The network graph of the desktop/desktop repository

The network graph is useful for seeing how far ahead one branch may be, or what kind of work someone may be working on in their own fork. When these commits make their way back into the original repository's default branch, we'll see them come in with an arrow and a merge commit if they were done via a pull request. We can also mouse over these commits to see who wrote them and what the commit message was.

The Forks List

Though not a traditional graph of work, the forks list gives you an idea of which members have forked this repository. If there is an unusual number of forks, you'll see a message like the one in Figure 2-20, displaying only a partial list.

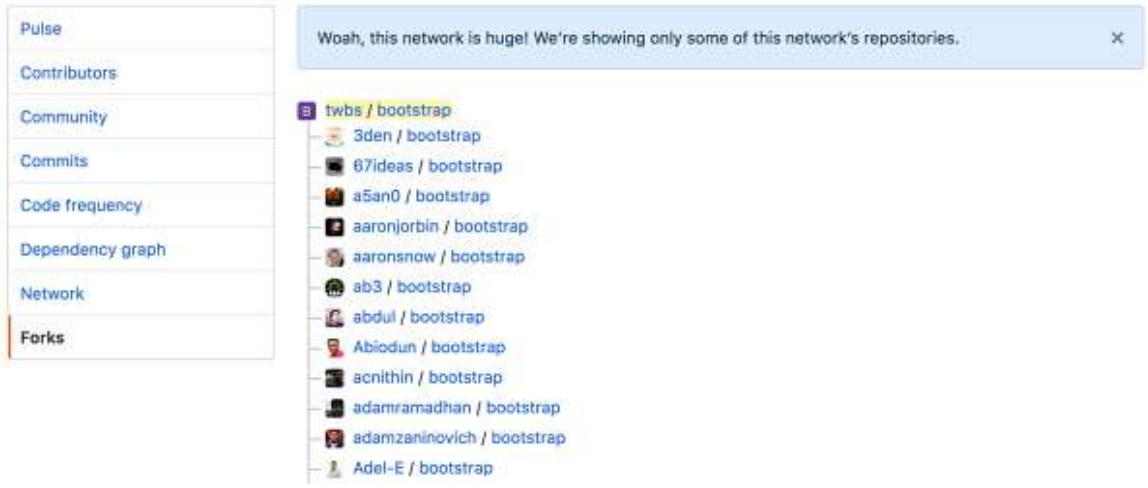


Figure 2-20. The list of Bootstrap forks

The forks list shows just the people who have forked the repository or forks of forks. These people aren't collaborators on the original parent repository and therefore needed their own copy of the repository in order to contribute to it through a pull request. To make your own copy, you would need to fork the repository. This action and workflow is covered in Chapter 4.

The Traffic Graph

One additional graph, which is available only to owners and collaborators on a project, is the traffic graph, shown in Figure 2-21.

The traffic graph shows you the number of Git clones, cloners, views, and unique visitors over time; it also lists the sites that people are linking from and highlights the most popular content on your GitHub project site. It can be a great way to get an idea of the popularity of open source projects.

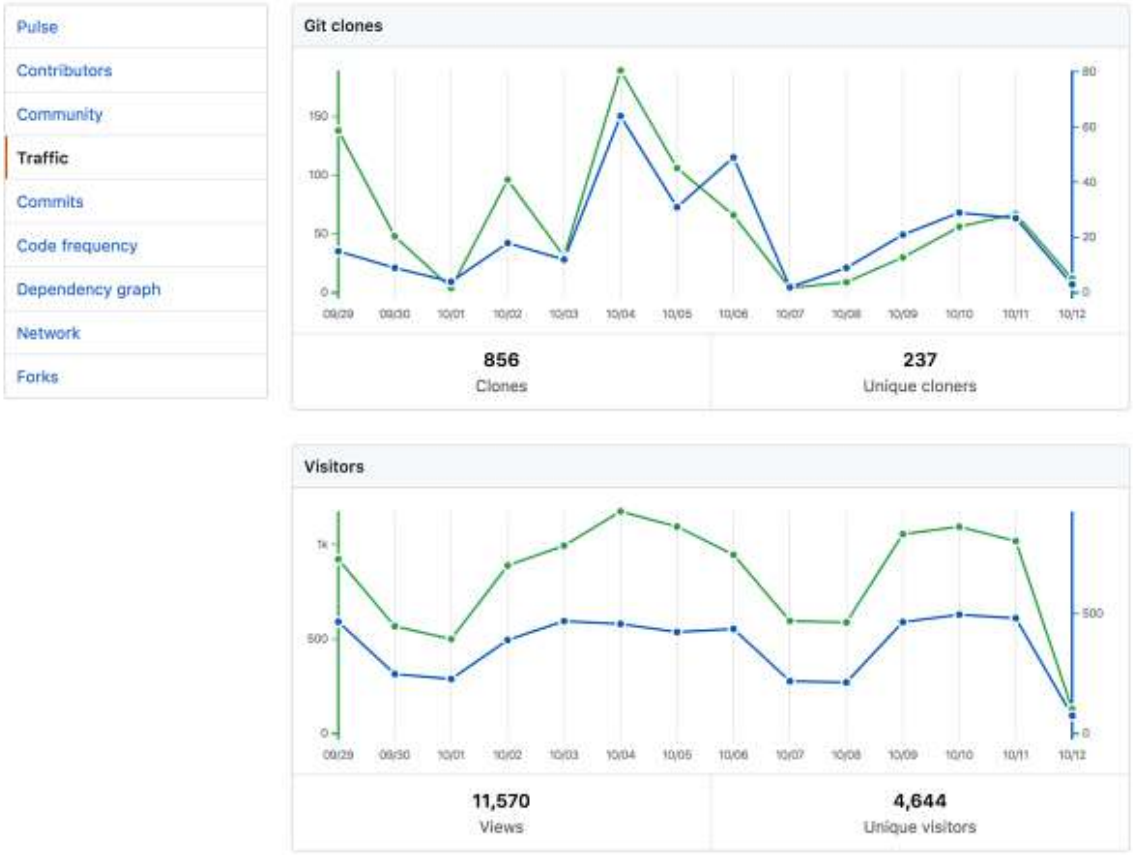


Figure 2-21. The traffic graph of the github/scientist repository

By now you should have a good sense of how to get up to speed with a repository by looking at the *README* file, commits, pull requests, issues, projects, the pulse, and the GitHub graphs. In the next chapter we'll look at how you can start to contribute to a repository.

Chapter 3. Creating and Editing

In this chapter we'll look at how you can create and work with your first repository. Creating this repo means that you'll always have access to the code and the additional information contained in pull requests, issues, projects, and wikis. We'll then look at how you can add, edit, rename, or delete a file directly on GitHub. We'll also look at how to work with directories on GitHub, and finally we'll discuss what to do when you want to make multiple changes as part of a single commit. From this point forward, if you want to follow along you'll need to create an account by going to *https://github.com/join*.

Creating a Repository

To create a new project on GitHub, click the + sign to the right of your username at the top right of the page. Then click the "New repository" option in the drop-down list. You'll see the new repository form, as shown in Figure 3-1.

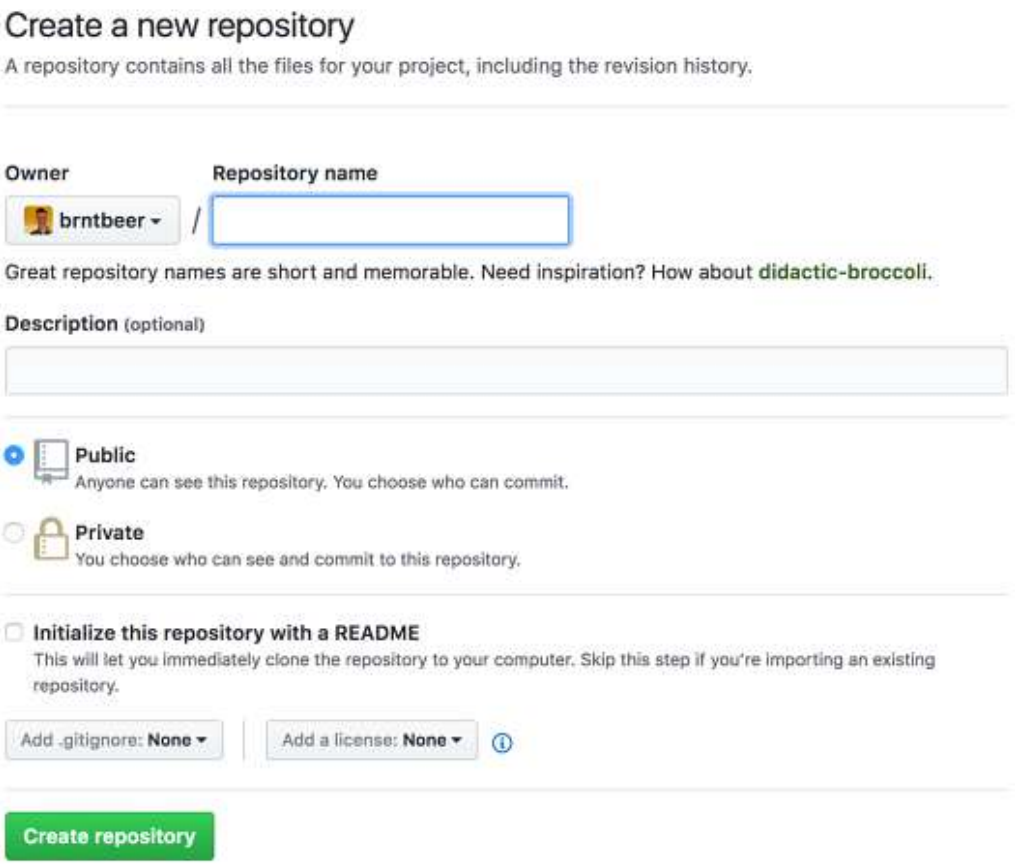


Figure 3-1. The new repository form

The first thing to do is decide whether to create the repository under your username or under an organization. You can see in Figure 3-2 a list of the possible organizations to which I could add a new repository. If you don't have access to any organizations, just leave this defaulted to your username. Remember, you'll always be able to transfer the project later if you want to.

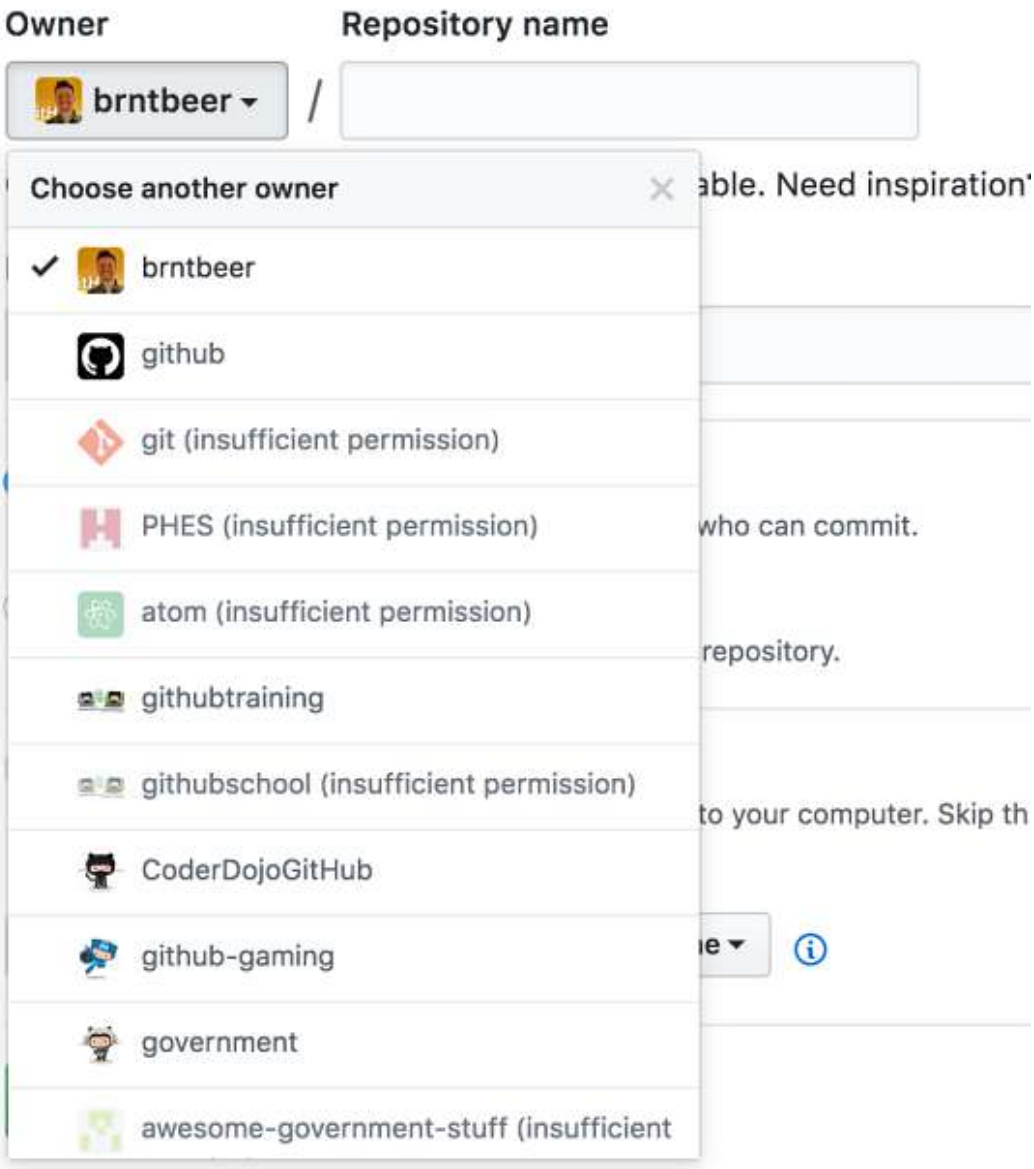


Figure 3-2. Selecting who should own the new repository

The next step is to give the repository a name and description. Names should be comprised of letters, numbers, hyphens, and/or underscores. Any other characters will be replaced with a hyphen. The description is a nice place to give people a short explanation of the purpose of the repository.

After entering the name and description, you need to decide whether to make the repository private or not. Public repositories can be viewed by anyone. Private repositories can be viewed only by people that you

specifically invite as collaborators. In either case, the project can be modified only by people you add as collaborators.

Generally, if your code is commercially sensitive, you'll pay the few dollars a month to be able to keep it private. If it isn't, you can just create a public repository, and it won't cost you a thing. If you don't see the option to make the repository private, you'll need to upgrade the user or organization you're creating the project under to allow it to host private repositories.

The final decision you'll need to make when creating a new repository is whether or not to initialize it with a *README* file by checking the checkbox, as shown in Figure 3-3.

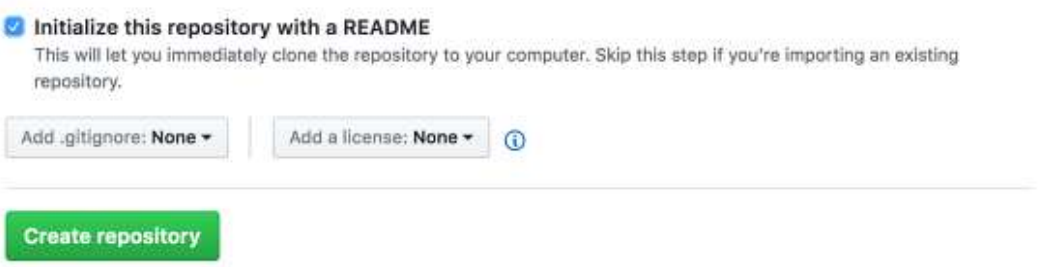


Figure 3-3. Initializing a repository with a README.md

Most developers will not check the box to initialize the repo. They'll just create a project locally, save it using Git, and then push their work up to GitHub. However, if you're not a developer, you'll probably want to initialize the project with a *README* as it allows you to create a project without having to create a local Git repository and upload it. Then your developers will be able to clone (download) the repo and add all of their code. Once you're ready, click the "Create repository" button, and the new repo will be created.

If you initialize the repo with a *README*, it will create a project and take you to a screen that looks something like Figure 3-4. That project is ready for your developers to clone and start committing to.

If you *don't* initialize your repo, you'll see a screen like Figure 3-5. Notice that you or someone on your team is going to have to upload an existing Git repository before anyone will be able to clone or work with this repository

or click on the links for *README*, *LICENSE*, or *.gitignore* near the top of the page, in the “Quick setup” section. Clicking one of those links will bring you to a page to add a new file. So, whether you already created the *README* or clicked one of those links, let’s look at adding a new file next.

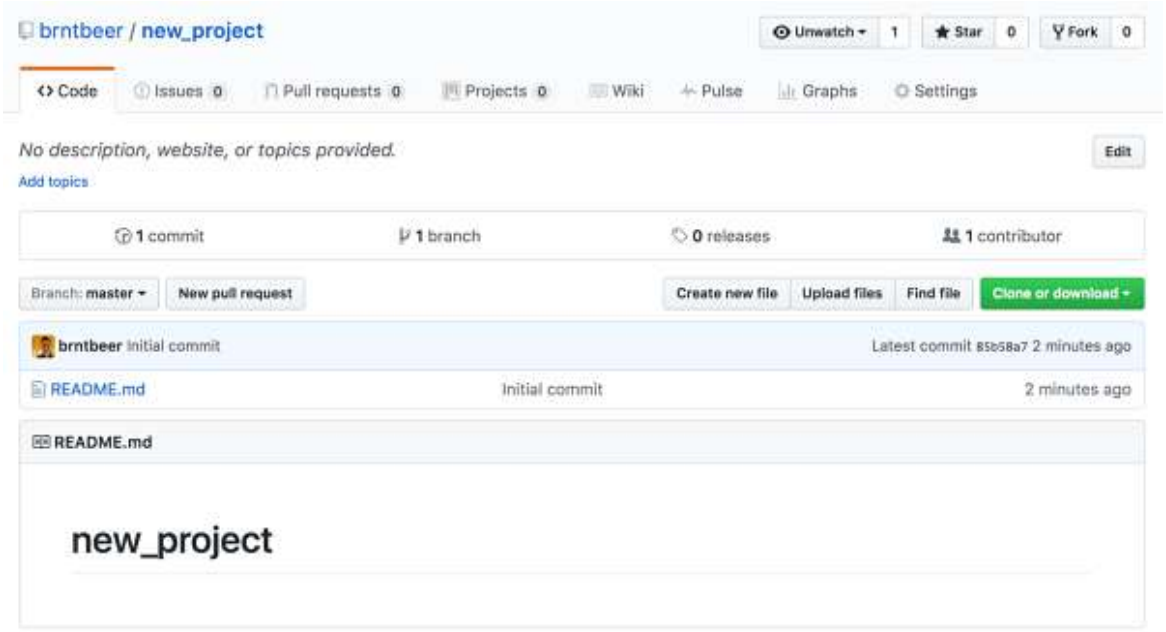


Figure 3-4. A new project initialized with a README.md

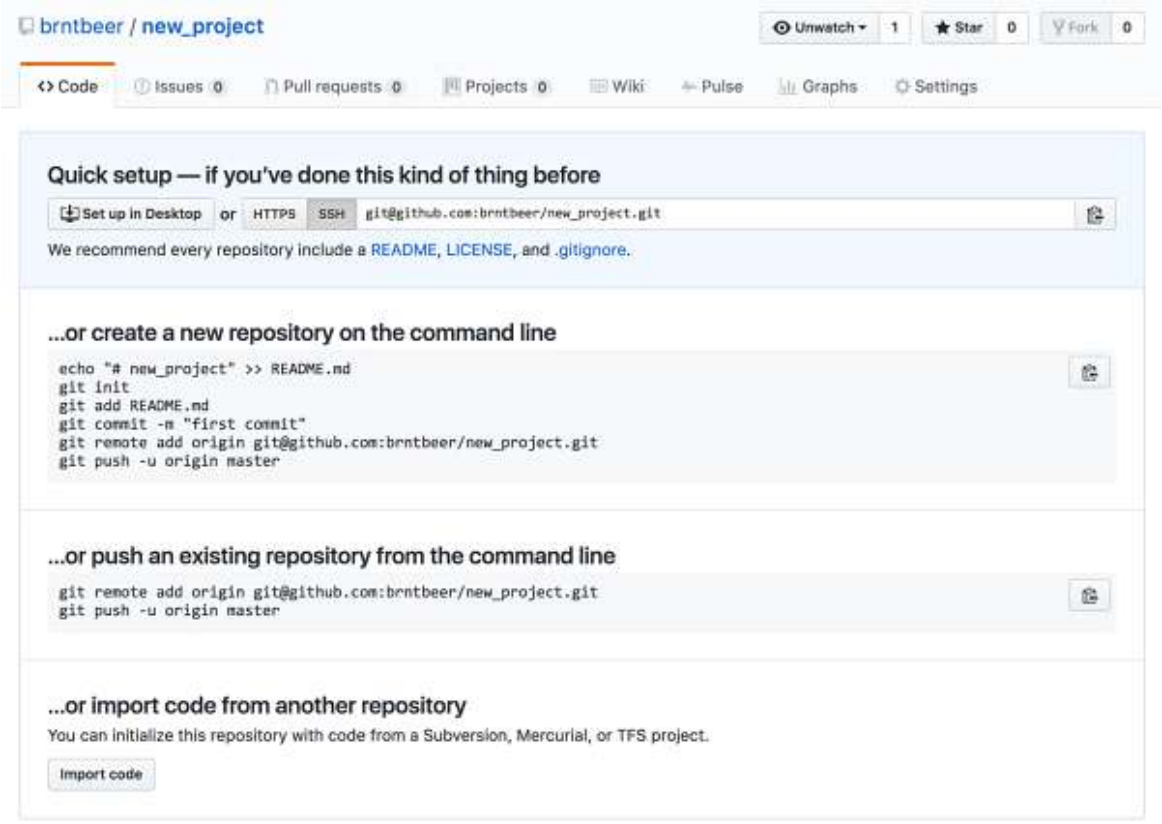


Figure 3-5. A new project that needs a repository to be uploaded

Adding a File

As you can see in Figure 3-6, there are a collection of buttons on the right side of the screen above the listing of files, and one of them says “Create new file.”

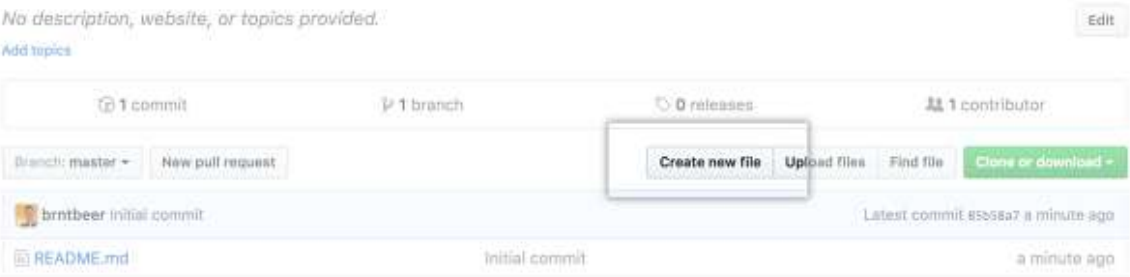


Figure 3-6. The “Create new file” button

Click this button and you’ll be taken to the screen in Figure 3-7.

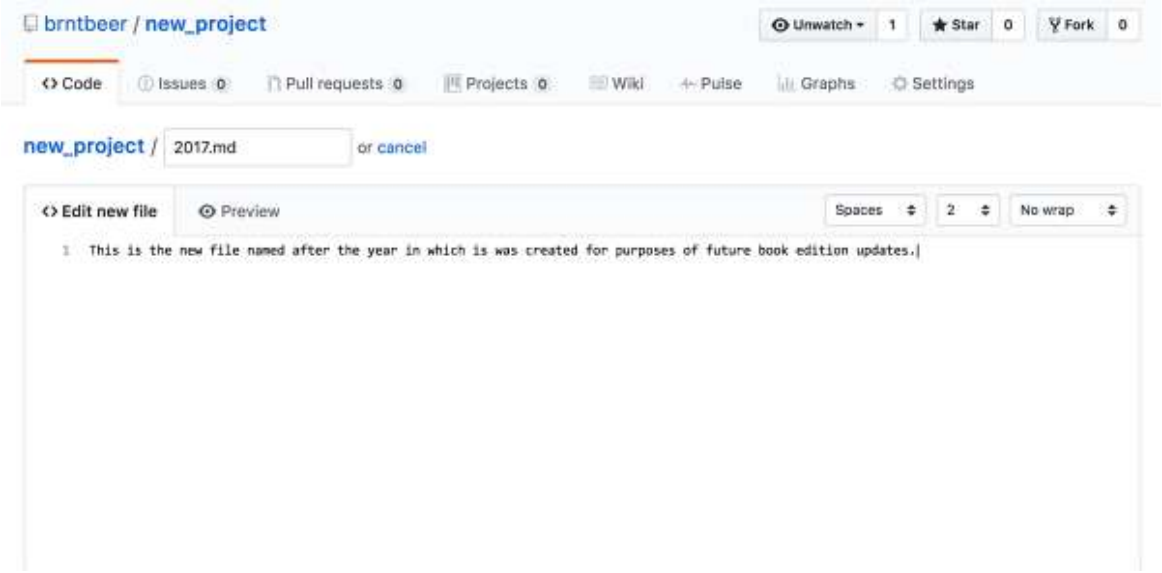


Figure 3-7. The "Create new file" screen

Toward the top of the page is a text box just to the right of the project name, where you can enter the name of the file you want to add to the project. Below that is a text area where you can enter the content you'd like to put in the file. Scroll down the page when you're done naming the file and entering the content and, as shown in Figure 3-8, you'll see a couple of text boxes where you can create a (required) short description and an optional extended description of the change that you're making.

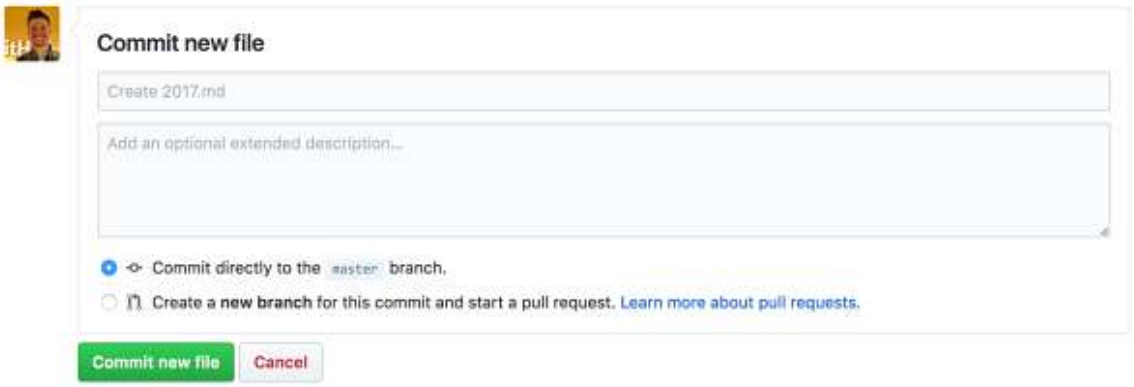


Figure 3-8. The bottom of the "Create new file" screen

These descriptions will be saved as the commit message for your commit. If you don't enter anything, the default commit message will be "Create (<*filename*>)." Generally, you'll want to enter a meaningful commit

message so other people viewing the project will understand what you did and why you did it. You have two options for where to commit this change: directly to the `master` branch or on a new branch to propose the change as a pull request. We'll see how to start a pull request another way in the next chapter, but this "quick pull request" flow exists for immediately starting a pull request from your edits. For now, click the green "Commit new file" button, and your new file will be added to the project and your commit will be added to the commit history. You can see in Figure 3-9 that *2017.md* has been added to the list of files.

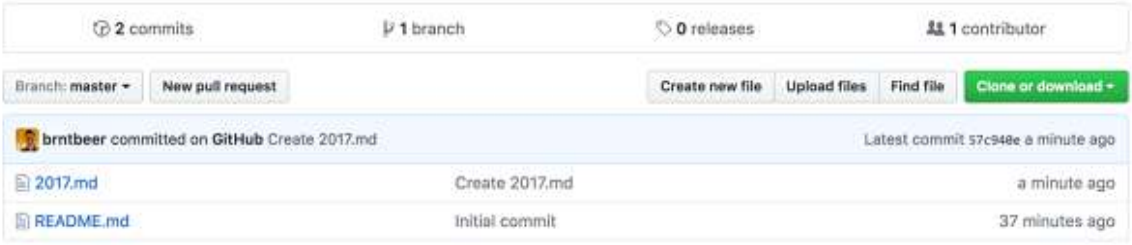


Figure 3-9. The project home page after adding the new file

Editing a File

Sometimes you might want to add a new file to a project because you're adding whole new functionality, but most of the time you'll find that you're going to change an existing file to expand some feature or fix some bug in existing code. Let's say you wanted to edit *README.md* to let people know how to contribute to the project. Starting on the home page of your project, if you click the *README.md* filename, it'll take you to a page like Figure 3-10.

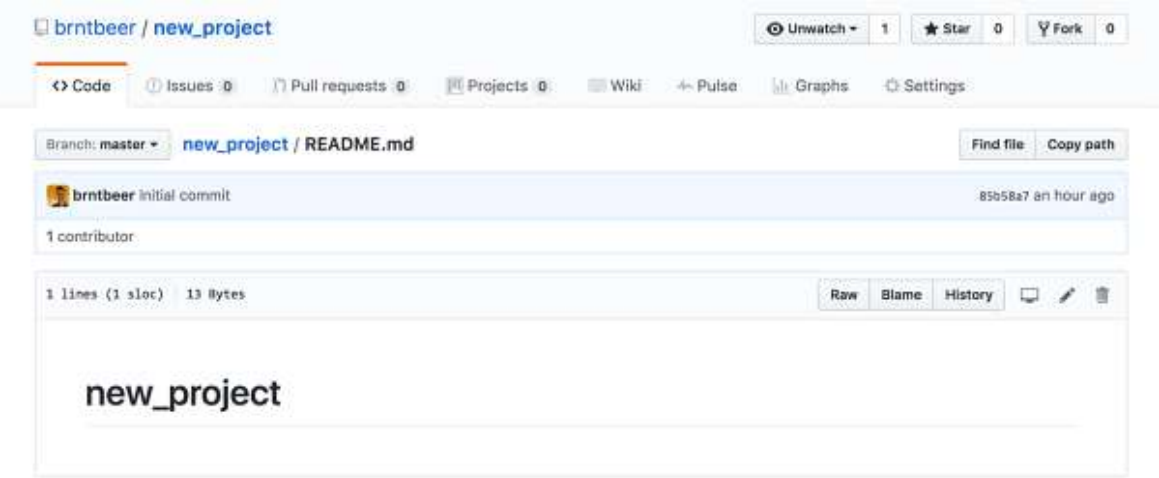


Figure 3-10. Viewing the README.md file

In Figure 3-10, you can see who last made a change to the file, how long ago the change was made, the first line of the commit message, and how many people have contributed content to the file. Above the display of the content are a number of buttons and icons. The option we're going to use right now is to edit by clicking the pencil icon, shown in Figure 3-11. Clicking that icon takes you to the screen shown in Figure 3-12, which will allow you to change the content of the file.

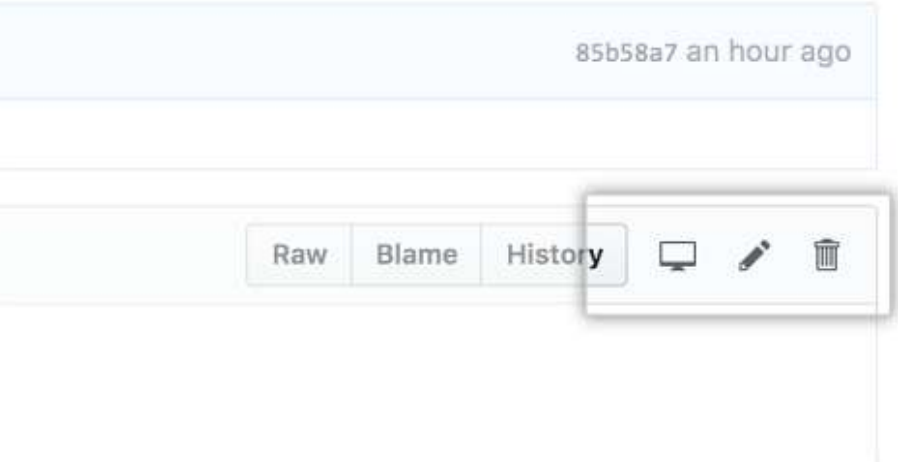


Figure 3-11. Pencil icon to edit the file

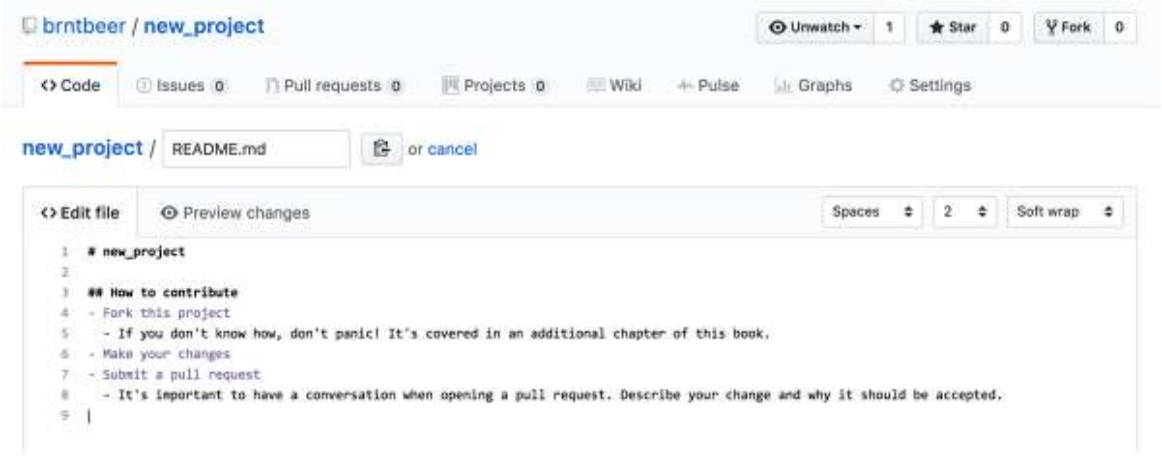


Figure 3-12. Editing the README.md file

As with the screen for adding a file, once you're done with your changes to explain how to contribute to your project, scroll down the page, enter a meaningful commit message, and click the "Commit changes" button. Once you've done that, you'll see the page displaying the *README.md* file and any additional content you added. In Figure 3-13 you can see the "How to contribute" information that was just added to the file.

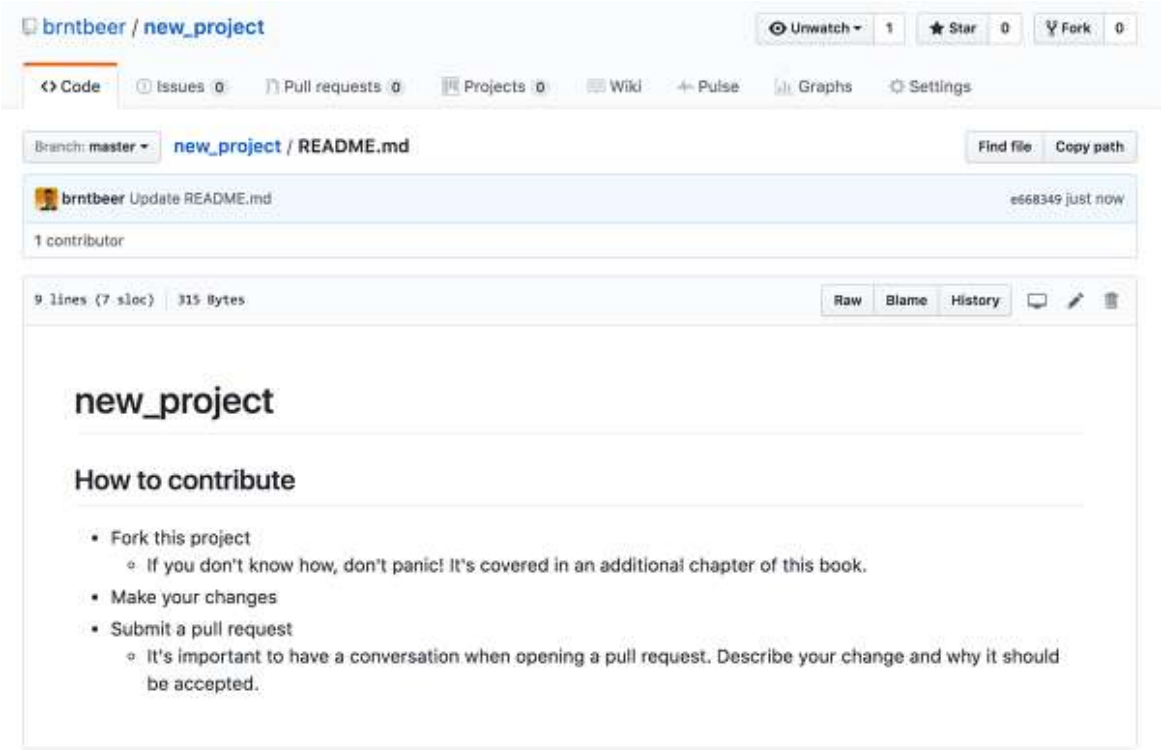


Figure 3-13. The updated contribution instructions in README.md

Renaming or Moving a File

Often you will want to rename a file, or move it from one folder to another. As far as Git and GitHub are concerned, both are the same process: you're changing the full name of the file, and optionally including the name of its folder. In this section, we're going to move the *2017.md* file we just created to a folder called *documentation* and then rename it *chapter_1.md*.

To start, go to the home page (the Code tab at the top of the page) of the repo, and then click the *2017.md* filename to go to the view page for *2017.md*. Then click the pencil icon to edit like you did in the previous section. Doing that gets you to a screen that looks like Figure 3-14.

Figure 3-14. The edit screen for 2017.md

This time, instead of editing the content of the file, you're going to go to the text box further up the page with the filename. If you just want to rename the file but keep it in the same folder, you can just change the name of the file. If you want to put this file in another folder (whether or not it exists already), all you have to do is include a forward slash (/) in the filename. So in this case, you just need to type *documentation/chapter_1.md* into the filename box. As you can see in Figure 3-15, as soon as you enter the forward slash, GitHub breaks that out as a new folder in the interface. If you later wanted to move the file up a folder, you could just start by typing ../ into the filename.

Figure 3-15. Editing the folder or filename for a file

If you misspell the folder name, just click the “cancel” button shown in Figure 3-15 to start over. If you’d like to edit the content of the file, you can also do that at this time. Once you’re done with the renaming or moving of a file, scroll down the page and commit the change. Figure 3-16 shows the renamed and edited file, now in the */documentation* folder.

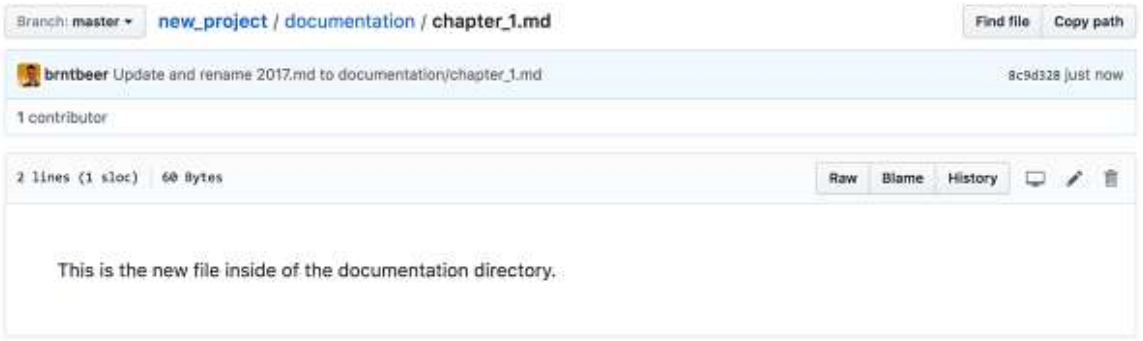


Figure 3-16. The renamed and edited file in the /documentation folder

Working with Folders

It is important to understand how Git thinks about folders—it doesn’t! Git is concerned only with files. As far as it is concerned, folders are simply a place to store those files. Because of that, there is no way to add a folder to a project unless it includes at least one file.

Sometimes this is a problem. For example, in many software projects there needs to be a */build* folder where automatically generated files will be saved when compiling the software. With some systems, if you don’t have such a folder, you’ll be unable to use the project.

Creating a Folder

A common pattern that has emerged is to create an empty file called *.gitkeep* in any folder that you need to create but that doesn’t really need to have any files in it. It seems a bit strange, but it works well and it is a well-understood convention—so if you ever need to create a folder, just create a *.gitkeep* file (see Figure 3-17).

Figure 3-17. Adding a .gitkeep file to create a /build folder

Renaming a Folder

You might have guessed that just as you can't create a folder directly, you can't rename it directly either. If you want to move a single file from one folder to another, you can do that by renaming it. For example, if I wanted to move *chapter_1.md* from */documentation* to */new_docs*, I could just go to the view page for the *chapter_1.md* file, click the pencil icon to edit the file, and at the start of the filename box type *../* to go up a folder, followed by *new_docs* to create or put the file into that folder instead. However, there is no way to just rename a folder on GitHub. You'd have to rename each of the files in the folder one at a time to move them to the new folder.

The Limits of Editing on GitHub

We have just run into one of the limitations of editing on GitHub. Originally GitHub was designed to allow developers to share their Git repositories with each other. Developers would make changes to their projects locally on their laptops, save those changes in Git, and then push the results to GitHub. Now that more and more nontechnical people are collaborating via GitHub, it's possible to do much of your editing right on the site, but there are a number of things that you can't do via the web-based interface.

Currently, GitHub doesn't allow you to rename folders or to make changes to more than one file in a single commit, nor does it give you the power of Git to rewrite history, or access to a handful of other workflows and commands more suitable for a terminal interface or desktop application. So, collaboration on projects often requires downloading (cloning) a copy of the repo, making some changes, and pushing them back up to GitHub.

If you want to learn the basics of working with Git locally, check out the instructions in Chapter 8 for getting started with GitHub on the desktop. For

now, though, we’re going to look at how to collaborate effectively with others or with your team using GitHub.

Chapter 4. Collaboration

In this chapter we'll start by looking at how to collaborate on a repository that you don't have permission to push to by creating a fork and a pull request. While forks are a good way to accept contributions from people you don't work with regularly, they are a bit too cumbersome for everyday use in a team that is working together closely. Because of this, later in the chapter we'll look at how to collaborate directly on a single repository without using forks. Lastly, we'll take some time to look more deeply into collaborating using pull requests and issues.

Contributing via a Fork

If you want to contribute directly to a repository, you either need to own it or have been added to it as a collaborator. If you want to contribute to a repository that you don't own and are not a collaborator on, you'll need to make a copy of it on GitHub under your user account. That process is called *forking*. Once you've forked a repository, you'll be able to make any changes you want to your fork (copy) and you'll be able to request that your changes get incorporated into the original repository by using a pull request. Let's go through that process now.

Go to *https://github.com/pragmaticlearning/github-example*. Click the Fork button in the top-right corner of the page, as shown in Figure 4-1.

Figure 4-1. The Fork button

When you click the Fork button, if you are a member of any organizations, you'll see a list of all of the organizations you're involved with as well as your username. You'll be asked where you want to fork the repository (as shown in Figure 4-2).

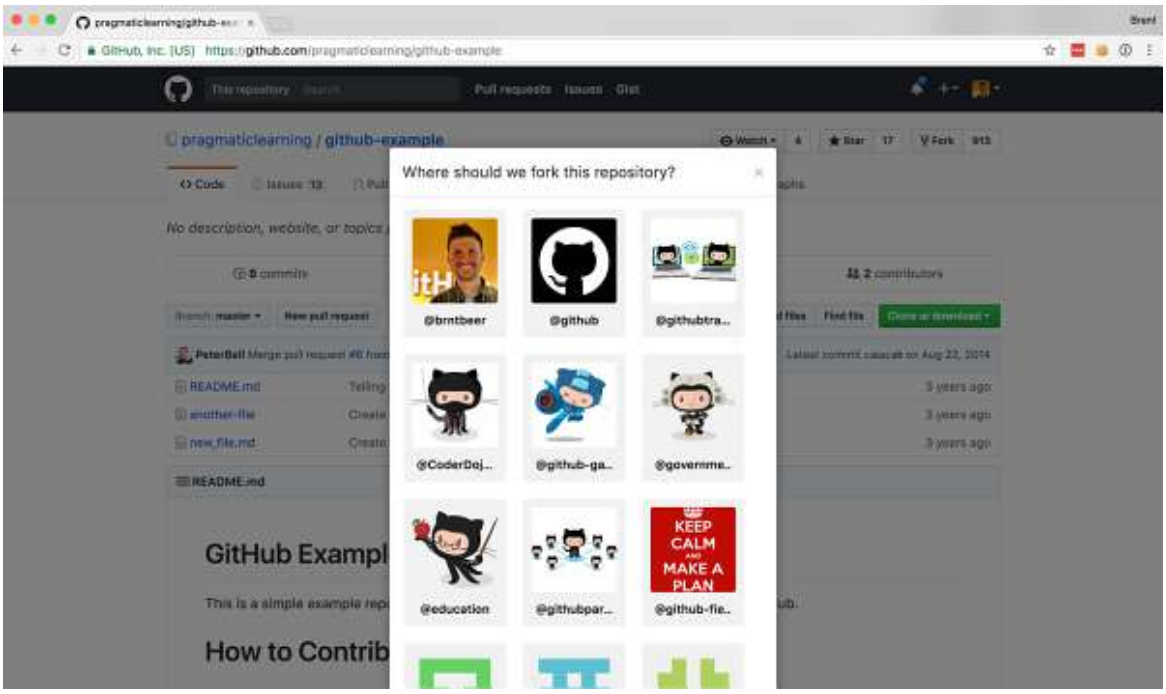


Figure 4-2. Selecting where to fork a repo

After you select where you want to fork the repository, or if you are not a member of any organizations, you'll be taken to your new repository page. Once you've forked the repo, you can make any changes you want to your fork (copy). In the next section we'll look at how you can add a new file, like we did in Chapter 3, and then how to create a pull request to try to get your change incorporated into the original repository.

Adding a File

In this section we'll look at how to add a new file to a repository. As a reminder, you can see in Figure 4-3 that there are a collection of buttons on the right side of the screen above the listing of files, one of which says "Create new file."

Figure 4-3. The "Create new file" button

Click this button and you'll be taken to the same new file screen as before, shown in Figure 4-4.

Figure 4-4. The "Create new file" screen

Toward the top of the page is a text box just to the right of the project name, where you can enter the name of the file you want to add to the project. Below that is a text area where you can enter the content you'd like to put in the file. Scroll down the page when you're done naming the file and entering the content; as shown in Figure 4-5, you'll see a couple of text boxes where you can create a (required) short description and an optional extended description of the change that you're making.

Figure 4-5. The bottom of the "Create new file" screen

These descriptions will be saved as the commit message for your commit. If you don't enter anything, the default commit message will be "Create

(<*filename*>).” As mentioned in Chapter 3, you’ll want to enter a meaningful commit message so other people viewing the project will understand what you did and why you did it. As shown in Figure 4-5, you have two options for where to commit this change: directly to the `master` branch or on a new branch to propose the change as a pull request. We’ll start a pull request another way in a moment, but this “quick pull request” flow exists for immediately starting a pull request from your edits. For now, click the green “Commit new file” button: your new file will be added to the project and your commit will be added to the commit history. You can see in Figure 4-6 that *first-file.md* has been added to the list of files.

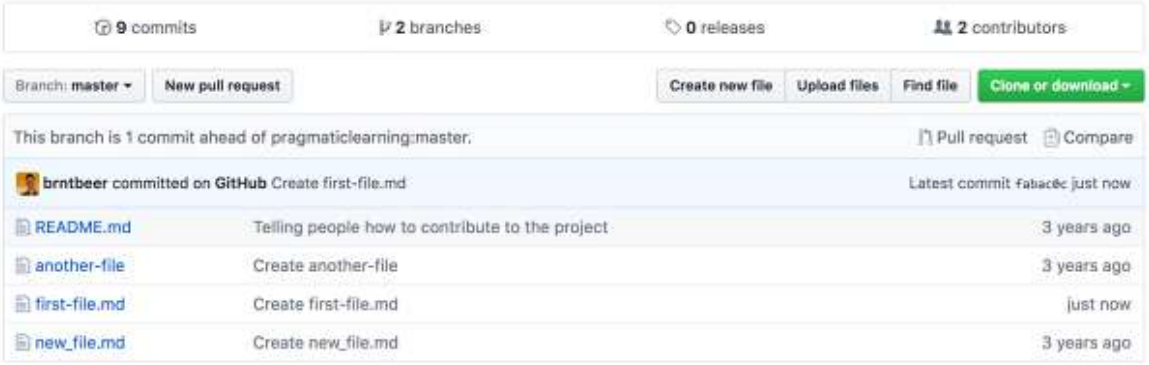


Figure 4-6. The project home page after adding the new file

Creating a Pull Request

You’ve made a change to your fork of the project, but the change hasn’t propagated back to the original project yet. That makes sense. Anyone can fork any public project, and the project owner wouldn’t want just anyone editing all of their files. However, sometimes it’s great to allow other people to *propose* changes to a project. This allows a large number of people to easily contribute to an open source project or a smaller team to work together on an internal project. That is what pull requests are for. Some maintainers send pull requests to their own repositories for the sake of documenting how they’ve developed their projects, even if they’re not waiting for anyone else’s approval! Many more approaches to maintaining, contributing, and working with open source software in general can be found at *https://opensource.guide*.

With a pull request, you can request that changes you've made on a fork be incorporated into the original project. Let's go through the process now. As you can see in Figure 4-7, at the top of the page there's a "Pull requests" tab.

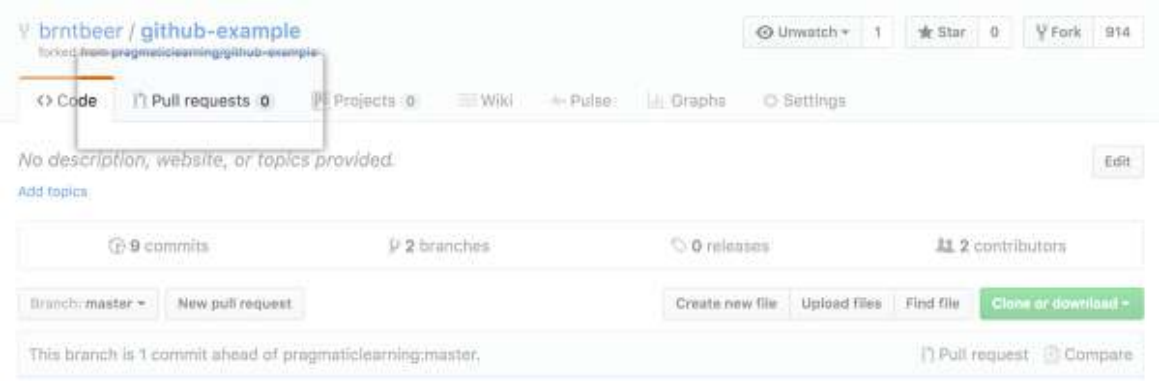


Figure 4-7. The "Pull requests" tab on the project home page

Click the "Pull requests" tab, and you'll see a screen similar to Figure 4-8 showing that currently you don't have any outstanding pull requests. Click the green "New pull request" button at the top right of the screen.

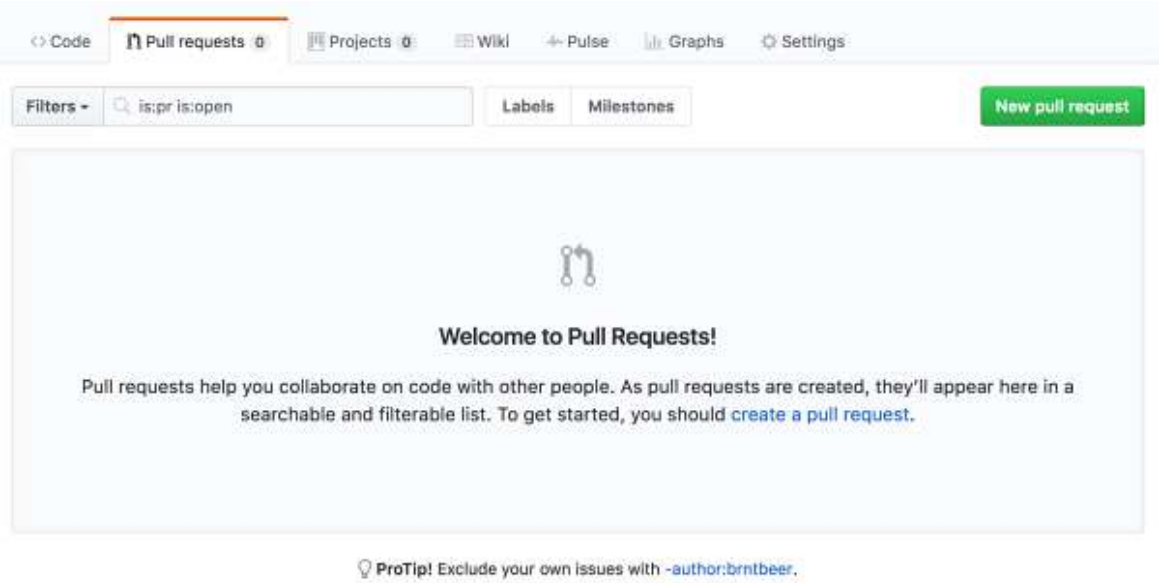


Figure 4-8. The pull requests screen

When you click the button, you'll see a screen similar to Figure 4-9.

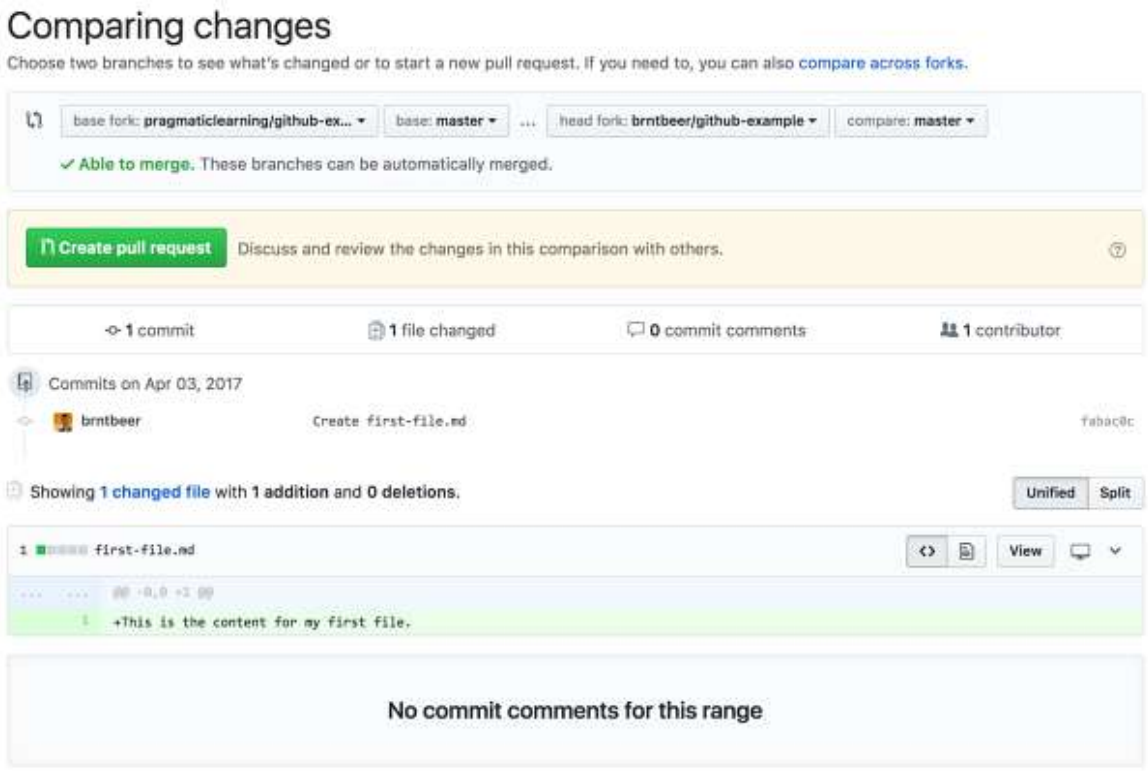


Figure 4-9. The "New pull request" screen

One of the first things you'll see in Figure 4-9 is that it is proposing a pull request between `pragmaticlearning:master` and `brntbeer:master`. Pull requests are requests to incorporate the changes from one branch (stream of history) into another. In this case, GitHub has correctly guessed that I want to take the change that I made on the `master` branch on my fork (the new file I added) and have that merged back into the `master` branch on the original project that I forked from. Note that the branch with the changes that you want merged in is on the right, and the target branch you'd like it to be merged into is on the left.

As you look lower down in Figure 4-9, you'll also see that it provides a summary of the changes that would occur if that pull request was merged—I did indeed make one commit that changed a single file. It even shows in green the new content that would be added to *first-file.md*—this is often called the *diff* or *difference*. If I click the Split or Unified button, it will change the way in which the difference that is being proposed is rendered.

This setting is “sticky,” meaning that GitHub will remember it for future diff renderings.

Once you’ve confirmed that the proposed pull request is the one you want to create, the next step is to click the large green “Create pull request” button. Doing so will take you to a page similar to Figure 4-10.

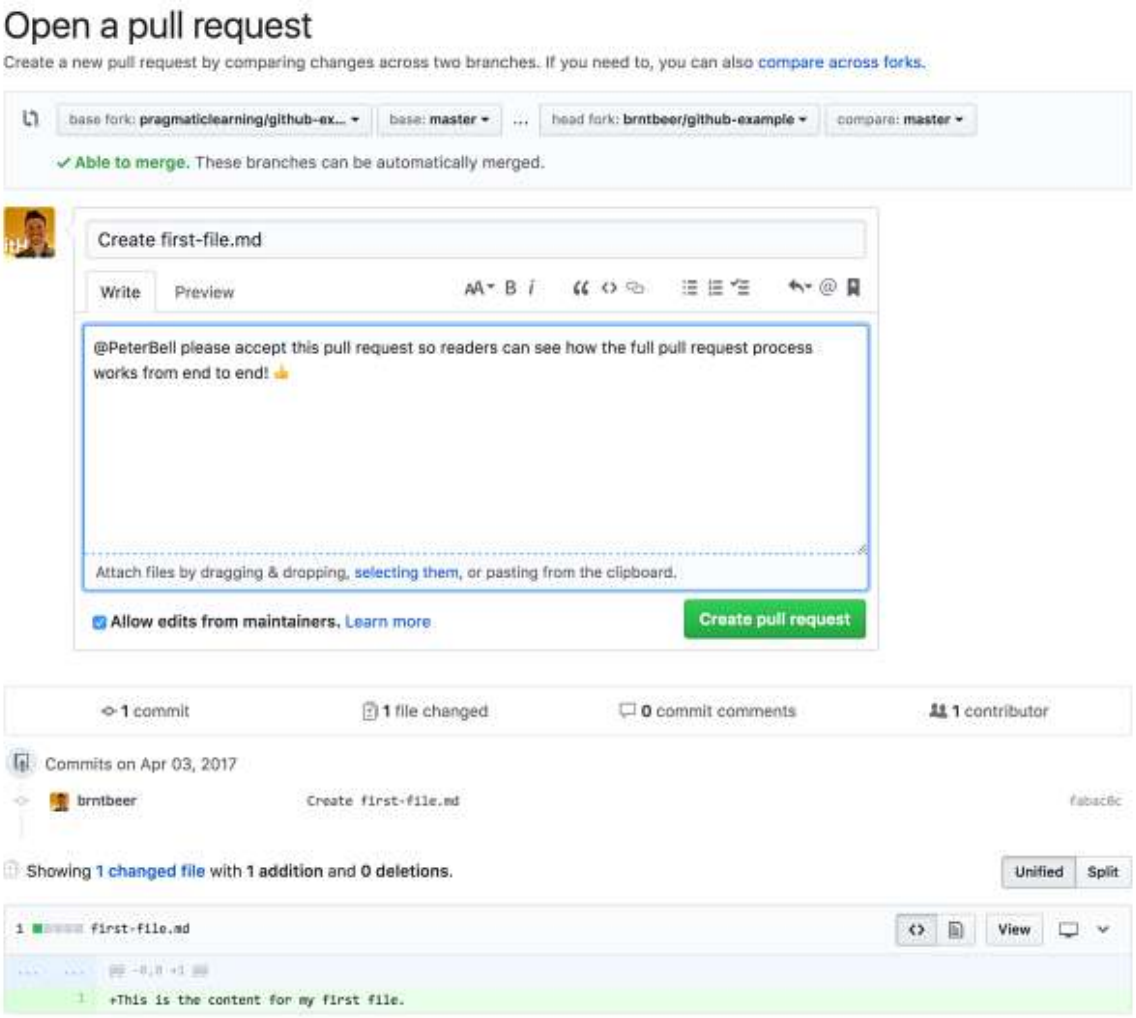


Figure 4-10. The “Create pull request” screen

This screen is your chance to tell the story and start a conversation about why your changes should be incorporated into the other project, so take the time to create a meaningful title and description of the changes you’ve made. By default the title will be the first line of your commit message for your most recent commit, and if you’ve made more than one commit on the branch you’re trying to have merged, the description will contain a bulleted

list of the first lines of all of the commit messages that are part of the pull request. That's a fine starting point, but you're going to want to take a little bit of time to describe not only what changes you've made, but why you made them and why they'd be a good addition to the project. Since this is the start of a conversation, it's often best to @mention (pronounced "at-mention") the maintainer to ensure they see your request, and to let them know if you still have some changes you'd like to make or not. If you're lucky, the repository you're contributing to will have made use of an issue or pull request template to fill in some information or give you some instructions on contributing. For more information about these templates, please see the GitHub documentation.

You may also notice the "Allow edits from maintainers" checkbox below the description section, to the left of the "Create pull request" button. This allows the maintainers to make changes directly on your branch. This may sound scary at first, but it really helps the maintainers, and you, if there are small changes to be made before accepting your changes. Sometimes there may be simple stylistic changes or complicated changes that are easier for them to do to finish the pull request. If you don't select this option here, it also can be turned on or off from the pull request screen after the pull request is created.

Once you've finished describing your pull request, click the "Create pull request" button and you'll see a page that looks like Figure 4-11.

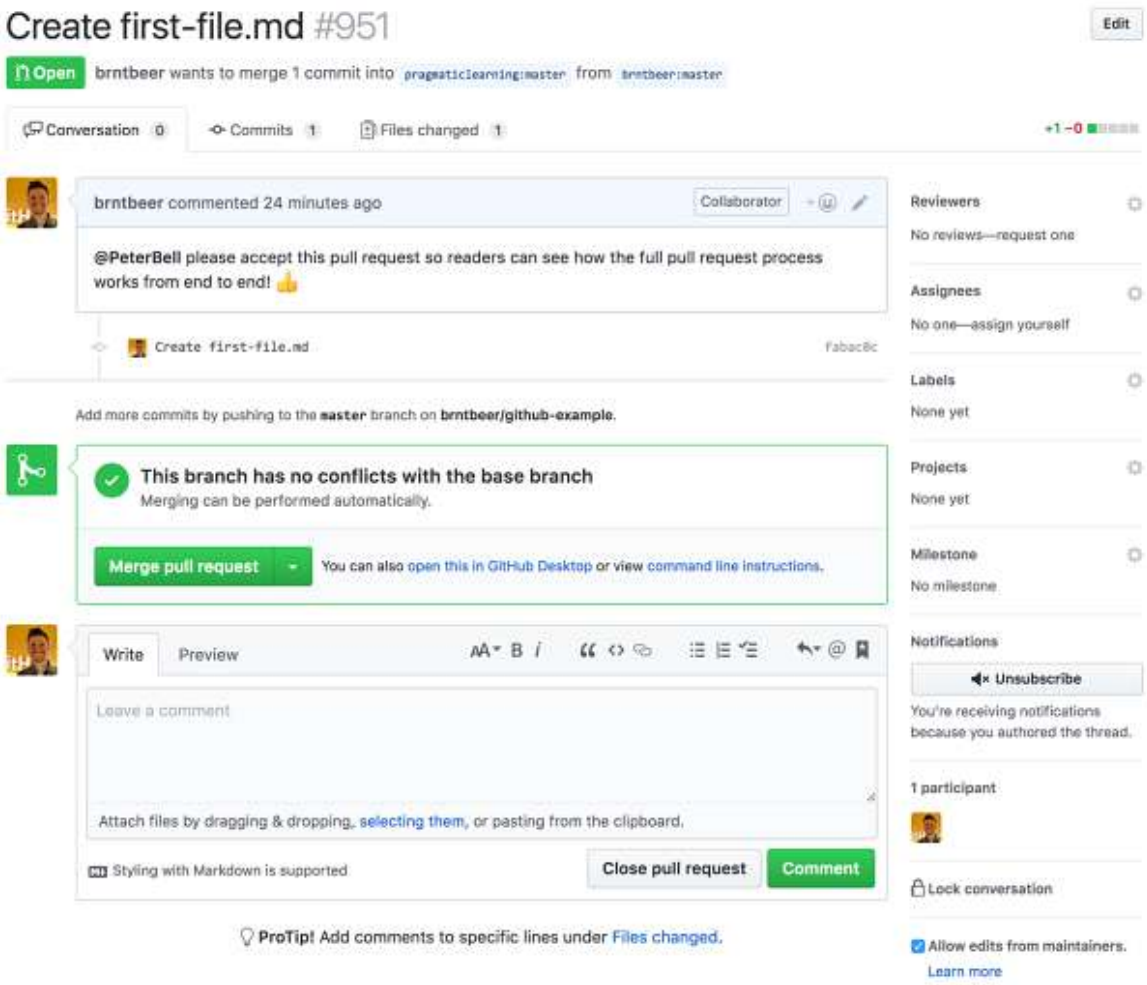


Figure 4-11. A created pull request

There are a couple of things that you should notice in Figure 4-11. First, notice that you're now in the original project, under `pragmaticlearning`. This makes sense. You wanted to create a request to pull your work into that project, so the pull request is part of that project—not your fork. You can see that "brntbeer wants to merge 1 commit into `pragmaticlearning:master` from `brntbeer:master`," and it shows you the pull request (title and description) followed by the commit that was made. Clicking that commit displays the details of the commit in a review workflow, as you can see in Figure 4-12.

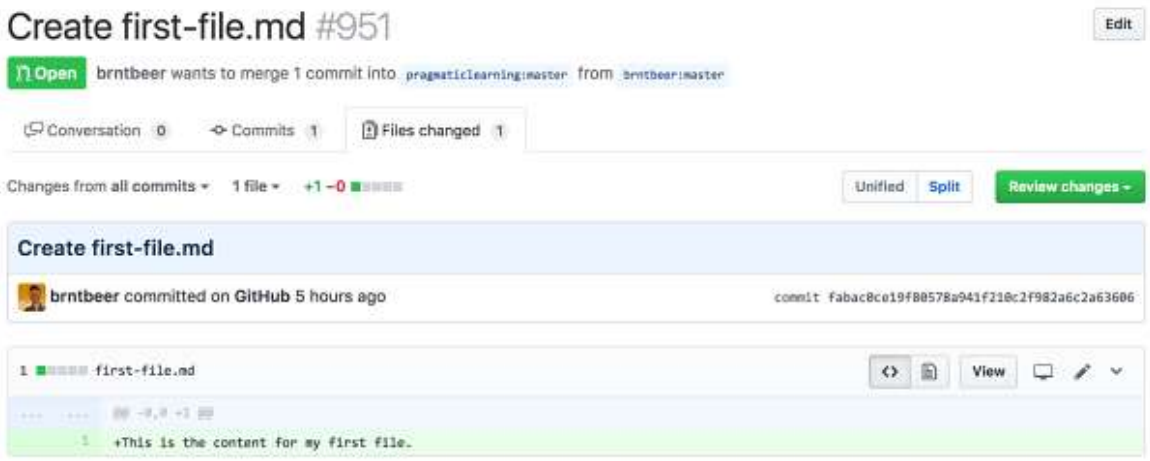


Figure 4-12. Viewing the commit from the pull request

Notice that the commit link has taken you to the “Files changed” tab of the pull request. This tab is where you can see all of your proposed edits for this pull request. However, in this situation you see just that one commit because that’s what you clicked on. This is a useful review workflow when you want to jump through some work one commit at a time. We’ll cover code reviews for pull requests in more detail later in this chapter.

Going back to the pull request in Figure 4-11, you’ll see that there is an option to merge the pull request. That option is visible only to the owner of the project or to anyone the owner has added as a collaborator with “write” or “admin” permission. If someone without those permissions—for example, yourself—was looking at the page, he would not be able to merge the pull request. To illustrate, in Figure 4-13 I’ve logged in as another user. When I view the same page, I don’t get the option to merge in the pull request, although I can still comment on it if I want.

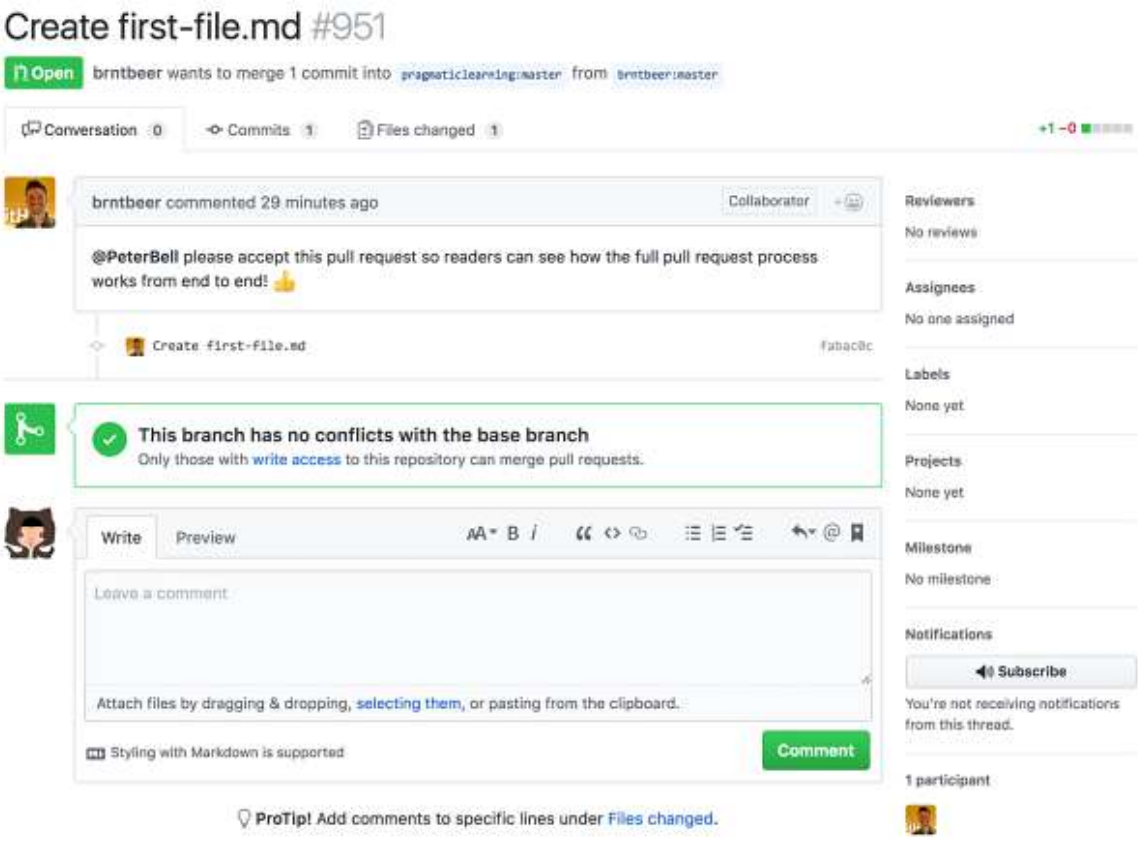


Figure 4-13. Viewing a pull request without being able to merge it

Often there will be a discussion before a pull request is merged, but we'll look at that more later in this chapter. For now I'm just going to accept the pull request and merge it in. Clicking the "Merge pull request" button adds a text box where I get the option to customize the commit message for merging the pull request, as shown in Figure 4-14.

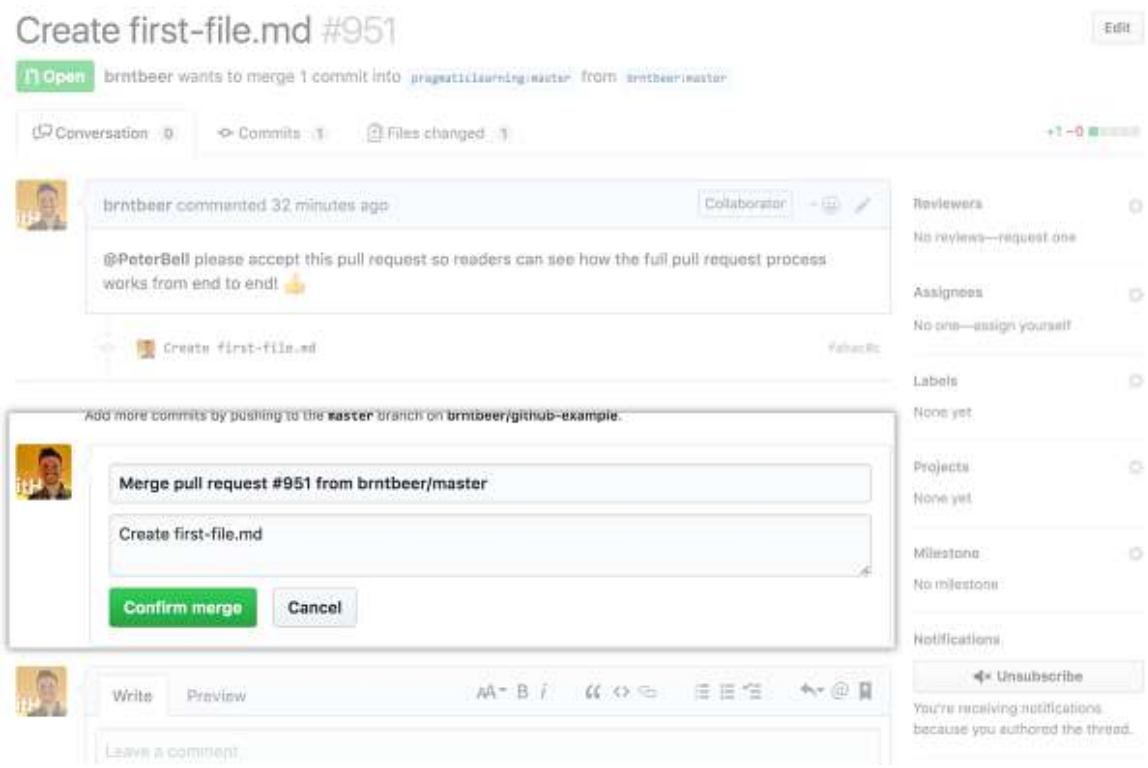


Figure 4-14. Getting ready to merge a pull request

Once I've made any changes I want to the merge commit's message, I can just click the "Confirm merge" button below and to the left. The pull request is then merged, and the output is displayed, as in Figure 4-15.

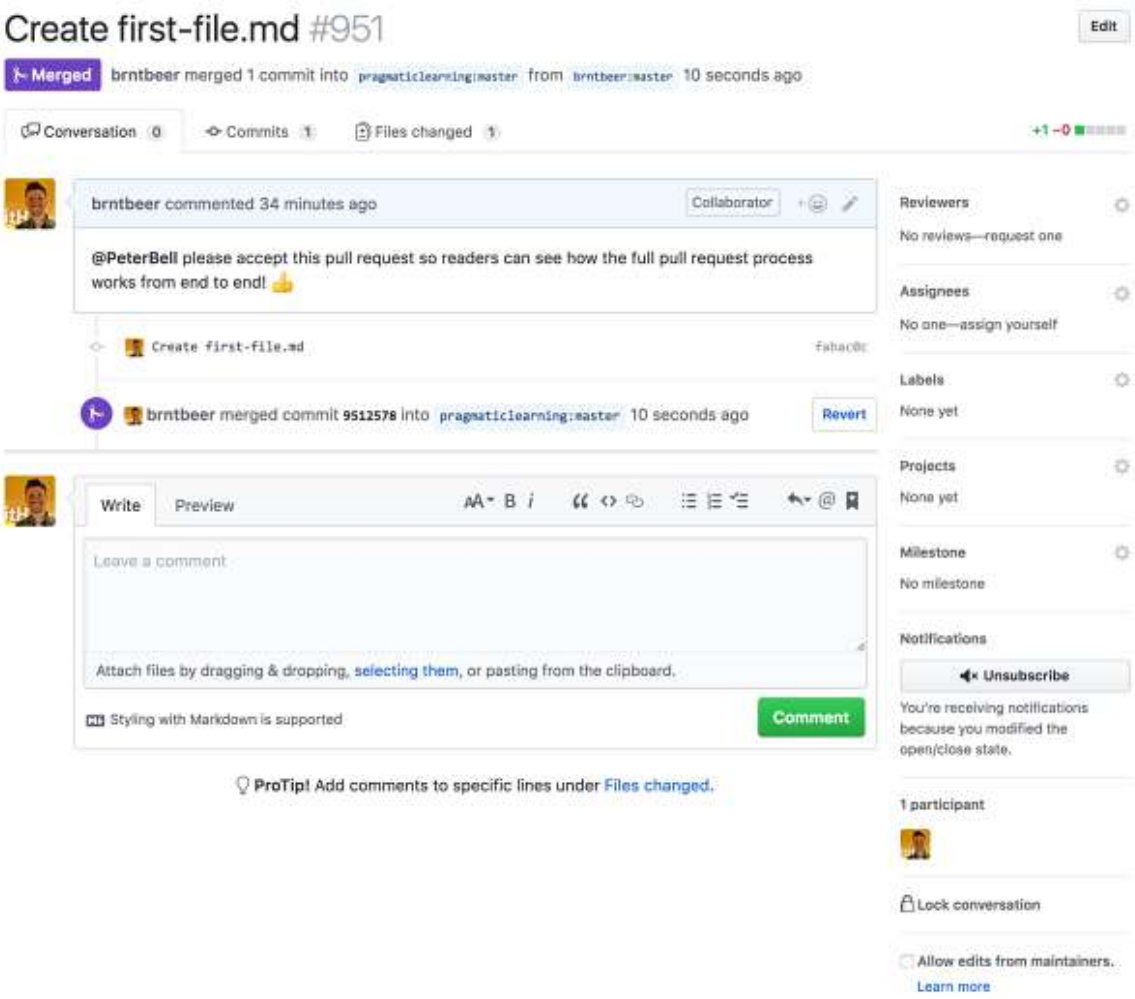


Figure 4-15. Viewing a closed (merged) pull request

Notice that you can still see the pull request message and the commit, but now you can also see who merged in the pull request and approximately when they did so. I also have the ability to revert this pull request if the merge was done in error. The Revert button here will allow me to open a new pull request that does the inverse of the work I just did. Finally, if you look at the project page in Figure 4-16, you'll notice a couple of things.

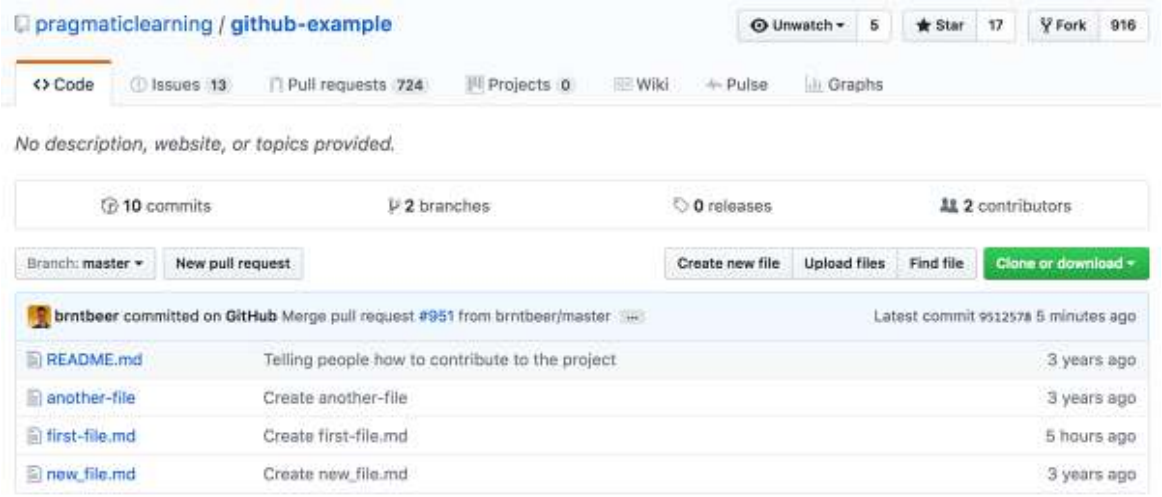


Figure 4-16. The original (pragmaticlearning) project after merging the pull request

First, *first-file.md* has been added to the project. Second, there are *10* commits now in the original project. Clicking the “10 commits” link shows why (see Figure 4-17).

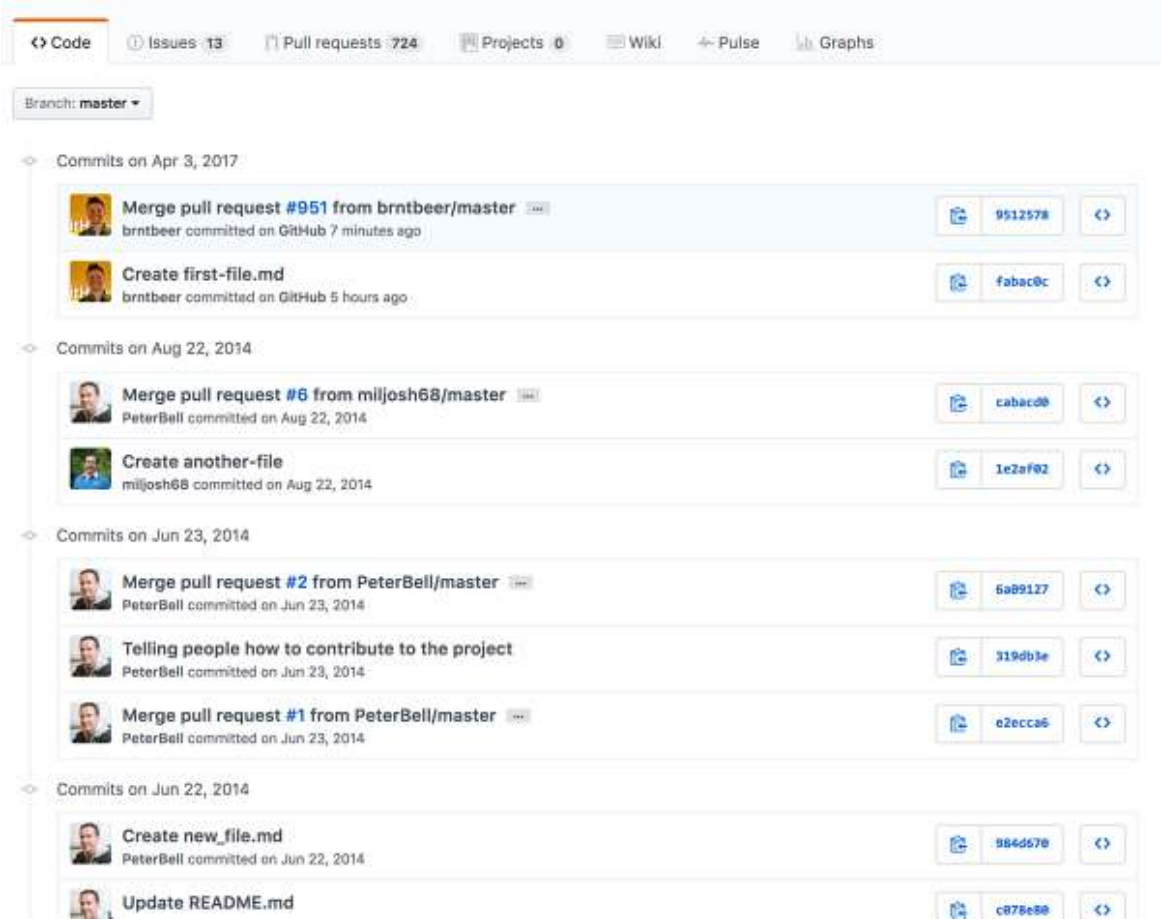


Figure 4-17. The project's commit history

In addition to the eight original commits in the project, there is the "Create first-file.md" commit that was made on my fork and a new merge commit that brought the work into the original project when I merged the pull request. By default, whenever you merge a pull request, it will create one of these merge commits (it is possible to configure the repository to have different functionality, but we'll leave that as an advanced topic for you to learn about later). They are really useful because the commit message, which you can edit when you merge a pull request, allows you to document *why* you decided to include the work.

Note

If you ever wanted to get rid of all of the work you merged in from a pull request, you could ask one of your developers to "revert the merge commit for that pull request" and she'd be able to easily remove all of the changes that got merged in. If you have permission to merge the pull request, you

will also have the ability to bring in the changes by squashing them all together before merging, or *rebasing* before merging. These are more advanced workflows that we won't be covering in this book. If you are interested in learning more, you should check out the GitHub learning resources.

Committing to a Branch

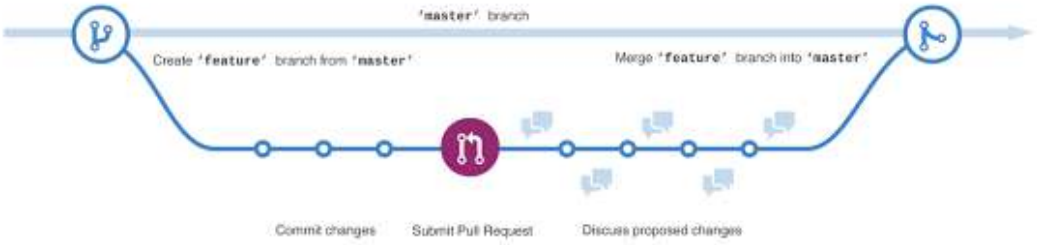


Figure 4-18. GitHub Flow, a basic branching collaboration structure

Now that we've seen how to contribute via a fork, we'll look at a more common team-based workflow: committing directly to a repository that you have access to push code directly to. To some people, this workflow may seem like a combination of working on a fork as well as working on your own repository, with one exception: feature branching. In Git, everything is committed onto a branch, and the branch we've done all of our commits on so far has been `master`. An alternative to committing to `master` is to create a branch that is often named after what you're working on. Sometimes this is something simple, like `update-readme-with-contact-info`, and other times it's named directly after a work item that's been assigned to you by a project management team, like `15363-change-login-flow`.

Besides allowing you to work safely and experiment on changes without affecting the `master` branch (which often signifies the safe, nonbuggy, stable code base), feature branching allows you to start a pull request in the same repository you're in. This is the part that is similar to the forking workflow. An example of the type of workflow you will be using is shown in Figure 4-18; it's called the GitHub Flow. Keep this image in mind as you dive in and experience it for yourself.

I've created a simple single-repo-example repository under the pragmaticlearning organization, as you can see in Figure 4-19.

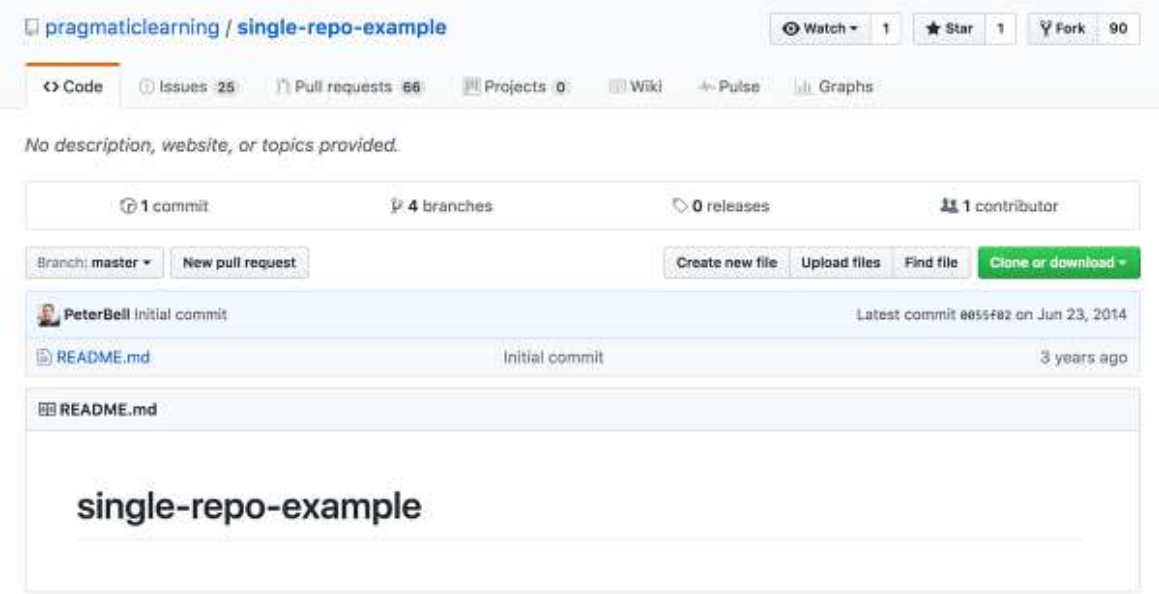


Figure 4-19. The single-repo-example repository

If I want to augment the *README.md* file, the first thing I should to do is create a branch. That way I'll be able to keep my changes separate while I'm working on them and can open my pull request later. To do that, I can just click the "Branch: master" drop-down, which lists the current branches in the project and provides a text box for entering the name of an existing branch or the new branch that I want to create (see Figure 4-20).

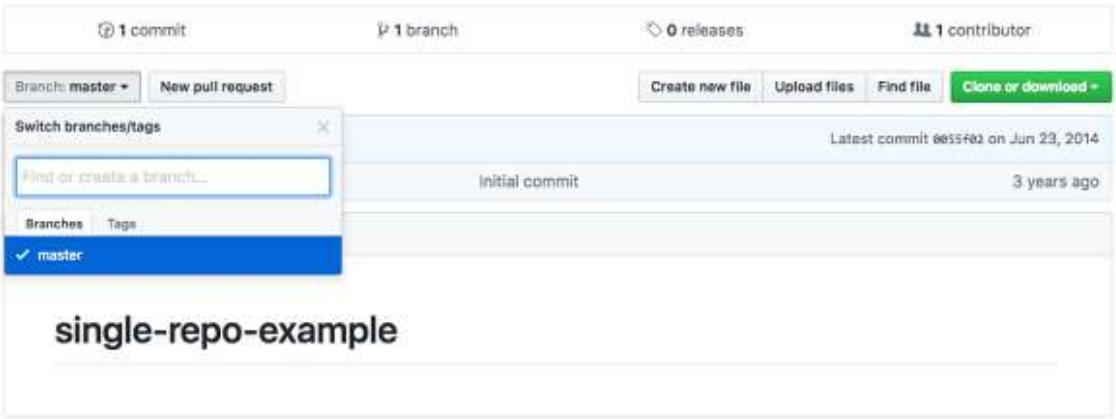


Figure 4-20. The Branch drop-down list

If I create an `update_readme` branch, as you can see in Figure 4-21, GitHub automatically checks out that new branch. You can see this both on the Branch pull-down where the current branch is displayed, and in the browser URL bar: the address ends with *tree/update_readme*, signifying that we're on the `update_readme` branch.

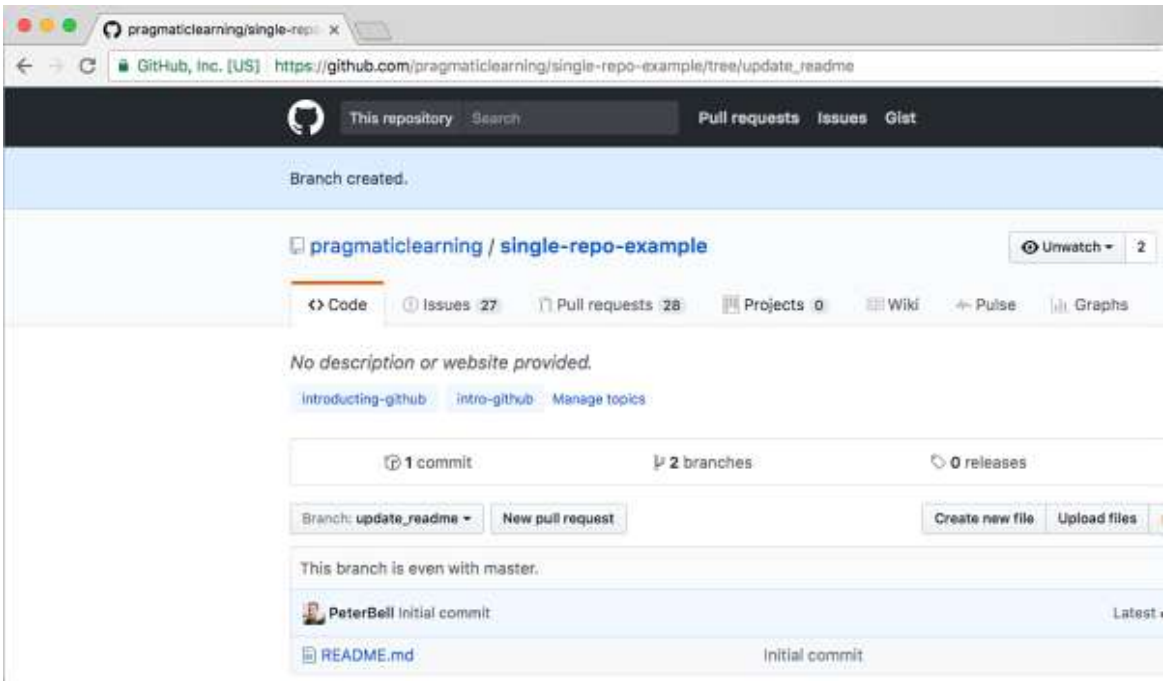


Figure 4-21. On the update_readme branch

The next step is to start to make changes on the `update_readme` branch. I'll edit the *README.md* file and commit the changes like we've done a few times up to this point. As you can see in Figure 4-22, I have only one commit on the `master` branch, but if you look at Figure 4-23, where I've changed the branch to `update_readme`, in addition to the initial commit you can also see the new commit that I made on the `update_readme` branch.

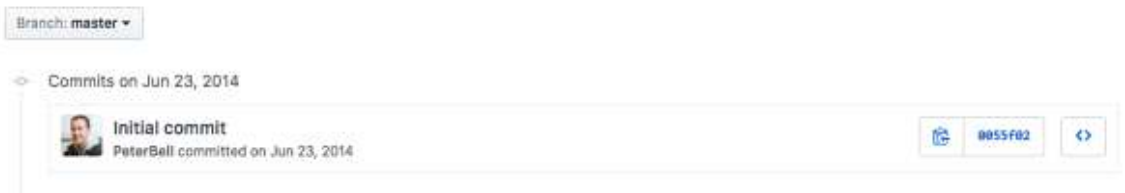


Figure 4-22. There's still only one commit on the master branch

Figure 4-23. But there are two commits on the update_readme branch

I might continue to work on the branch for a while, getting my changes just right. Once I'm ready to get some input, I'll want to create a pull request to start a conversation about my proposed changes.

Creating a Pull Request from a Branch

There are many ways to create a pull request, but as in the previous chapter, I'll click the Pull Request tab on the top part of the page and then click the green "New pull request" button. When I do this, as you can see in Figure 4-24, the experience is slightly different. Now GitHub isn't sure what branches I want to create a pull request between, so I have to tell it.

On the left you can see "base: master." That is perfect as it means that if I create a pull request, once it is accepted, it will get merged into the `master` branch, which is exactly what I want. However, I do need to click the "compare: master" drop-down to tell GitHub what branch I want to create a pull request for, as you can see in Figure 4-25. The "compare:" branch is the one that I'd like people to consider merging into `master`.

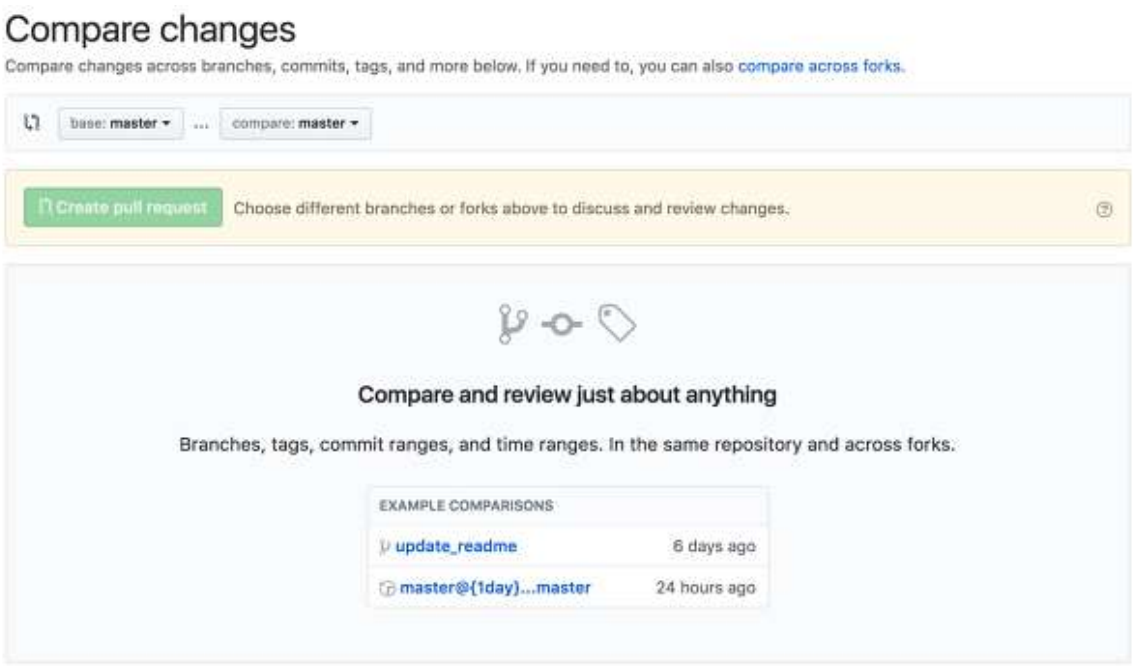


Figure 4-24. Starting to create a pull request from a branch

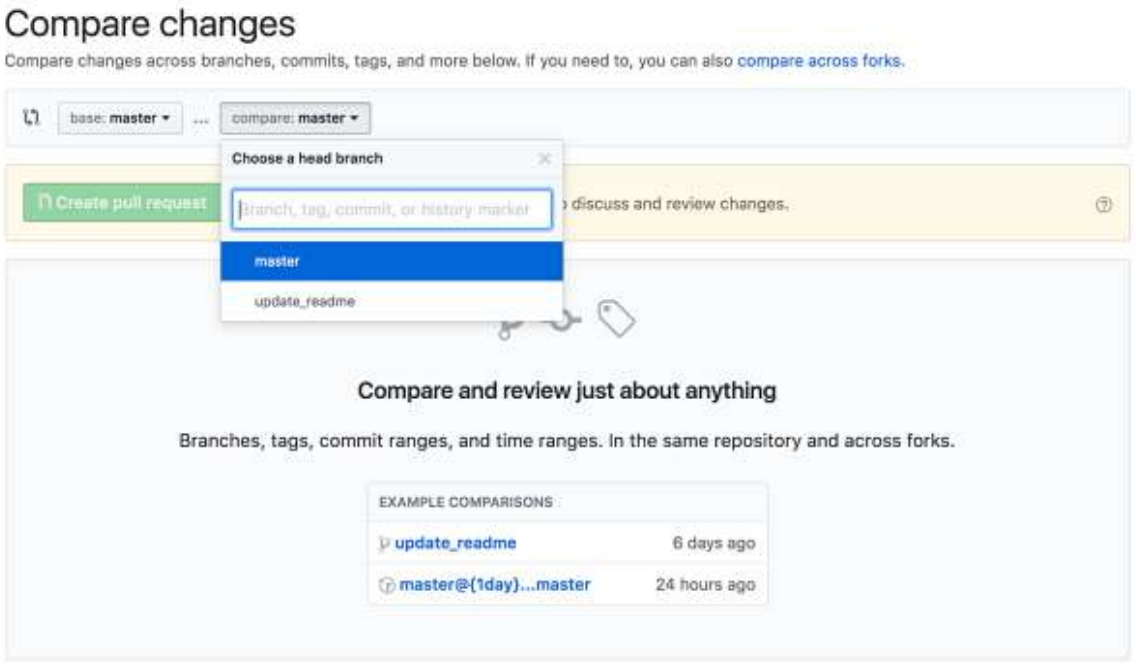


Figure 4-25. Selecting the branch for the pull request

Once I've selected a branch, the process is just the same as it was earlier in this chapter when creating a pull request from a fork. I click the green "Create pull request" button, enter a title and description to explain the

reason for the pull request, and then click the “Create pull request” button. This creates the pull request shown in Figure 4-26.

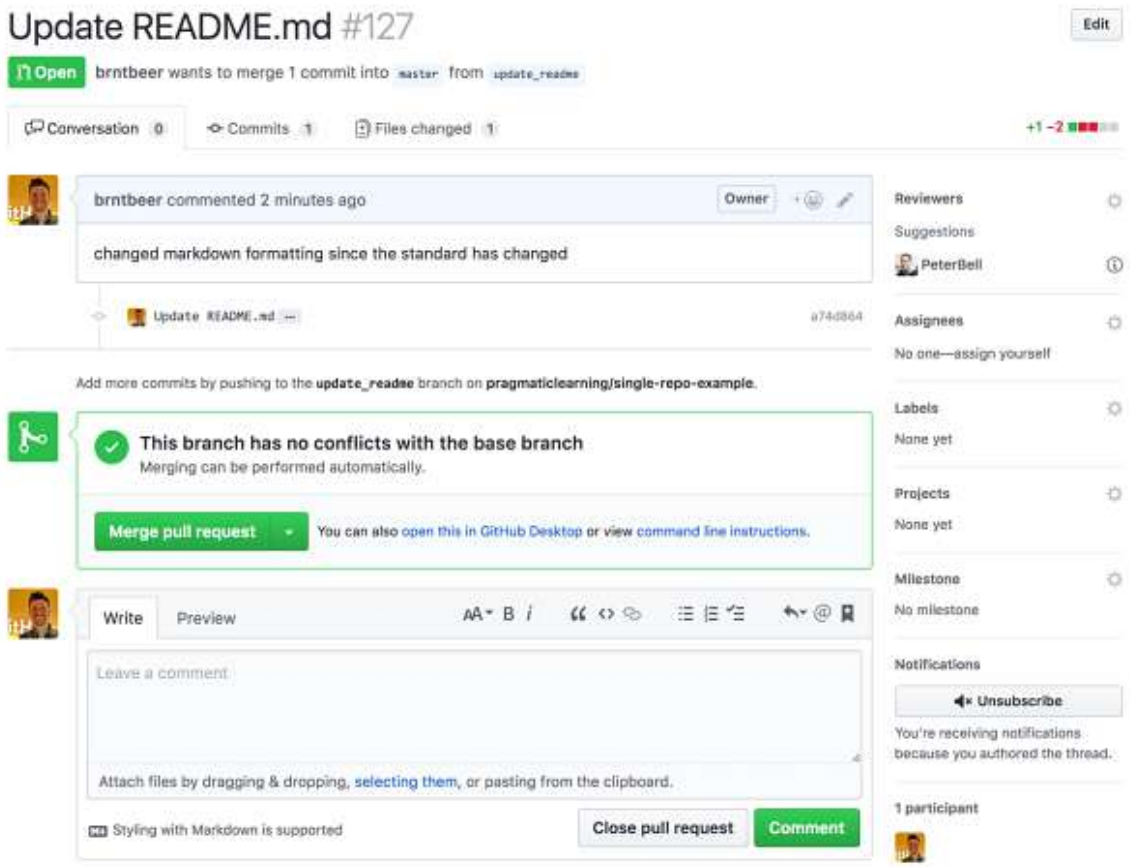


Figure 4-26. The new pull request

Collaborating on Pull Requests

Pull requests are designed to start a conversation about a proposed change—usually either a new feature or a bug fix. Originally, pull requests were created only when coding was completed to ask someone to incorporate a completed set of changes, but these days pull requests are used in a couple of different ways.

If you have a change that you’re confident about, you can still create a new branch, make all your changes, and wait to create a pull request until you’re done with the work. In such a case, the purpose of the pull request is just as a double-check to make sure that the rest of your team agrees with the

changes you made before the changes get merged into `master` and pushed to production.

However, there is another way to use pull requests. In many companies, employees will often create pull requests for features that they'd like to discuss. So, if you have an idea for a change but aren't sure whether it's a good idea, consider creating a branch and making the simplest possible start on the work—maybe just a small text file describing it. Once you have a commit on the branch, you can then create a pull request to kick off a discussion about the idea.

Involving People with Pull Requests

If you've created a pull request and would like feedback from specific people on a team, @mention them. To do this, within the pull request itself or in a comment on the pull request, type @ and then type in the GitHub username. If the person is the owner or a collaborator on the repository, the username will autocomplete as you start to type. You can also begin typing the user's displayed name (which can be set in your public profile).

If you wanted to get feedback from me on some work you'd been doing, you might create a comment like "hey @brntbeer, mind looking at this PR and letting me know what you think?" The formality of the language will depend on the people you're working with, but pull request comments are often written in a fairly informal style.

Reviewing Pull Requests

If you want to see what people are working on within a repository, go to the home page and click the "Pull requests" tab at the top, and you'll see a list of all of the currently open pull requests.

On most projects there should be only a few pull requests open at any one time. A good rule of thumb for a private repository is that you shouldn't have more than a few open pull requests per developer. Generally, the fewer pull requests you have open, the better, as it is more valuable to keep the team focused on finishing up existing features than on starting new ones. Also, pull requests should be for small, iterative changes, to make them easier to review. The more changes that go into a branch, the longer it will

live and the more difficult the changes will be to review properly. These “long-lived” branches aren’t always avoidable, but you should be on the lookout for them.

Note

The number of open pull requests on open source projects will typically be much larger, as anyone can create a pull request, and sometimes it takes a while for the core project team to review, accept, and/or close them.

When you find a pull request that you want to review, click it to view the pull request detail page.

Commenting on Pull Requests

A really important part of working with a development team is to take the time to review all of the pull requests that you might care about. Nothing is more disheartening than to work on a feature for a couple of days, create a pull request, and then get no feedback at all. Also remember that by default anyone can merge their own pull request into `master` so long as they have write permission, so make sure to take the time to review people’s work so they aren’t tempted to merge it in without at least one or two people having a look at it. Or, if you’d like this not to be the case, you can use protected branches to require certain approval workflows. We’ll cover some of these options in Chapter 7.

Whenever you get an email or a web notification that you’ve been @mentioned in a pull request, make sure to take the time to check it out as soon as you can and provide some useful feedback. Sometimes useful feedback may even be to let the person know you’ll give it a good review soon. Even if you’re not named personally, taking a little bit of time out of your day to make sure that you review any outstanding pull requests and provide your thoughts to ensure everyone is on the same page with where the project is going is a good idea.

Commenting on pull requests is pretty simple. Skim down the pull request page, go to the comment box, type in your feedback, and click the Comment button.

Adding Color to Comments

Especially for a team that doesn't work in the same office all of the time, commenting on pull requests is often one of the more frequent ways that team gets to interact. Because of that, it's often a good idea to add a little bit of fun to the interactions.

GitHub has built-in support for emoji. Emoji are small images that are often used for displaying a mood or emotion graphically. If you look at Figure 4-27, you'll see that this comment has the :+1: (I'm in support of this feature) emoji and the :ship: (let's merge this in and "ship" it) emoji.

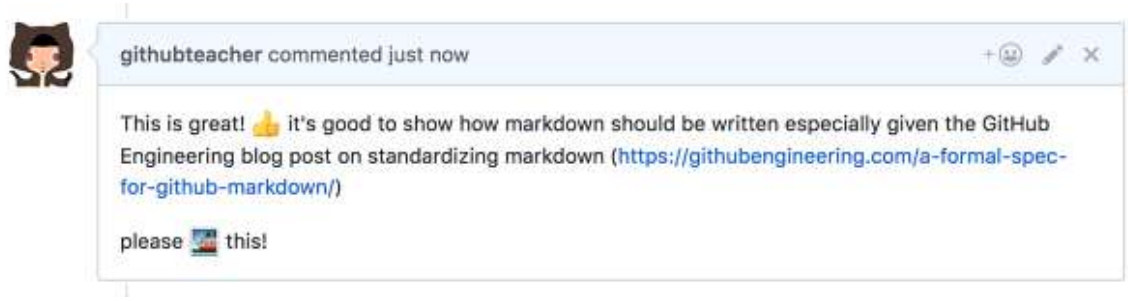


Figure 4-27. A comment with some emoji

Another way to add some more color to your comments on GitHub is by using animated GIFs. While emoji are subtle, most animated GIFs are much larger and more striking—they're often a great way to really lighten the mood or show strong support (or disapproval) for a change or a comment. To add an animated GIF (or any other image) to a pull request, just drag and drop it into the comment box and it'll get uploaded automatically.

Contributing to Pull Requests

Sometimes you'll want to make a change directly to a pull request. Perhaps someone has added a new page and you'd like to fix up the marketing copy, the legal disclaimer, or even the CSS to make it display better in your favorite browser. It's easy to make a change to someone else's pull request.

The process is the same as for editing a file, which we covered in the previous chapter. The only difference is that you must be on the correct branch. In this case I'm looking at the `update_readme` pull request for adding some content to the *README.md* file, as you can see in Figure 4-28.

If I decided that it would be great if the file contained a brief description, rather than just commenting that the *README* was missing a contributors guide, I could add one.

To make the change, all I need to do is go to the repository home page and select the `update_readme` branch from the drop-down list of branches. I can then click the file and click the edit icon, and I'll get the edit screen, as you can see in Figure 4-29.

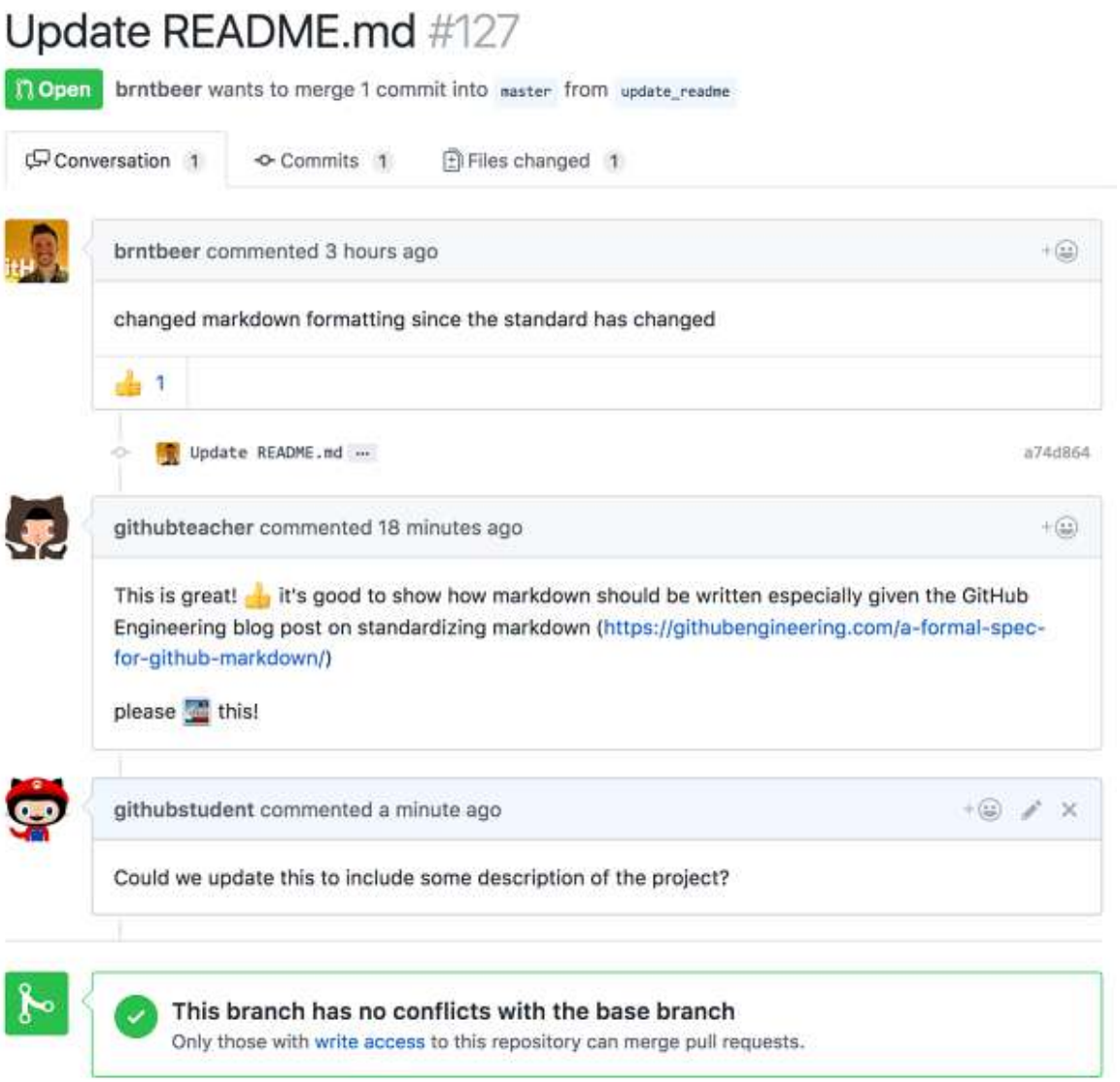


Figure 4-28. The update_readme pull request

Figure 4-29. Editing README.md on the update_readme branch

I can then make my changes, scroll down the page, and enter some kind of commit message, as shown in Figure 4-30.

Figure 4-30. Adding a commit message

Now if I go back to the pull request page, you can see in Figure 4-31 that my commit has been added to the pull request. Anyone who is watching the pull request will get a notification that it has been updated so they can review my change and provide their feedback.

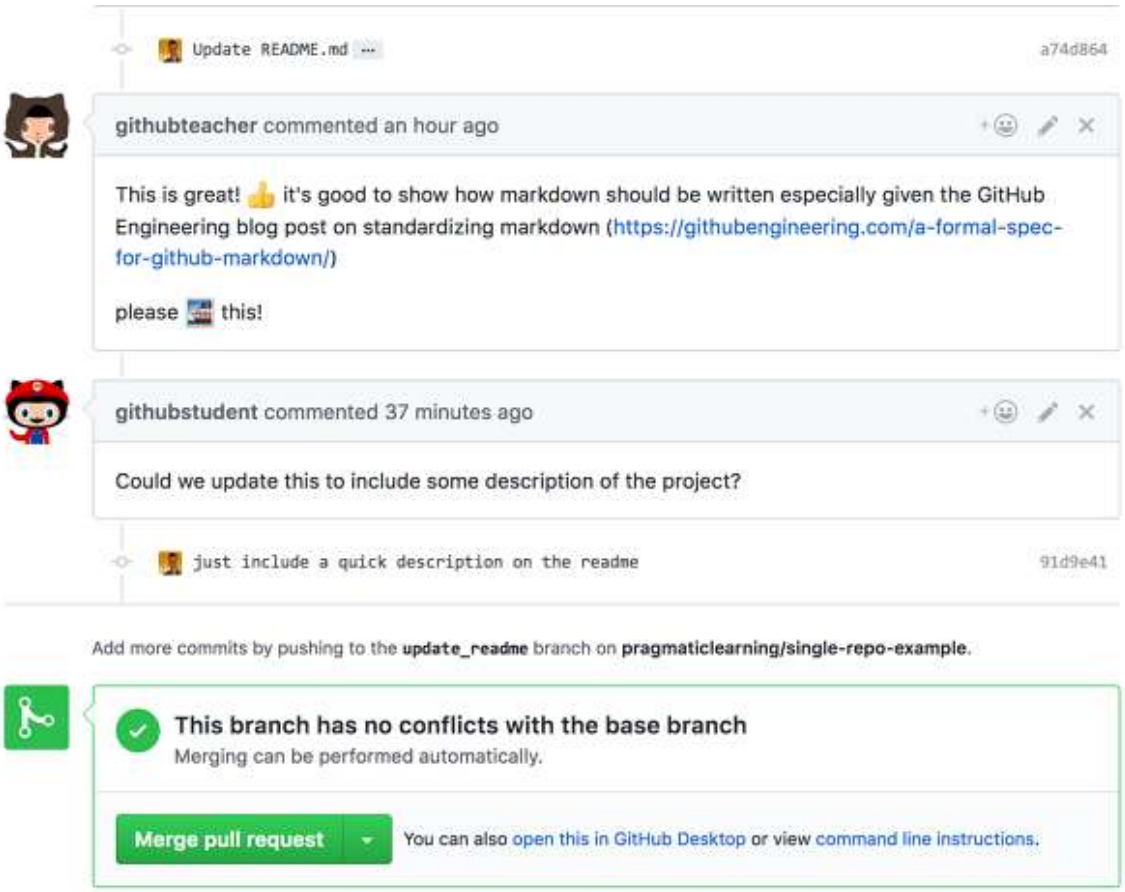


Figure 4-31. The new commit in the pull request

Testing a Pull Request

If you have the appropriate permissions, before you approve a pull request that includes substantive code changes that you can't just review visually, you're going to want to download a copy of the repository (clone the repo). Then check out the branch that the pull request relates to, run the automated tests to make sure they're all passing, and then run the code and maybe do a little bit of manual testing just to make sure it seems solid. We'll cover cloning repositories in Chapter 8.

If you're not a developer, you could leave this to your development team, but you do want to make sure that at least one or two people are downloading the code, running the test suite, and maybe doing a little manual testing before approving a pull request. Alternatively, an easier and best practice option is to set up automated testing that will run for you and report its status back to the pull request. This, as well as required reviews,

can be configured inside of the protected branches settings of a repository, which will be talked about in Chapter 7.

Merging a Pull Request

When you're ready to merge a pull request, just click the large green "Merge pull request" button, as shown in Figure 4-32.

Figure 4-32. The "Merge pull request" button

When you do so, GitHub will ask for a commit message (the default will be the title of the pull request and an indication that this commit came in from a pull request merge), as shown in Figure 4-33. Once you've entered that, just click the "Confirm merge" button and the pull request will get merged and closed, as described earlier in this chapter.

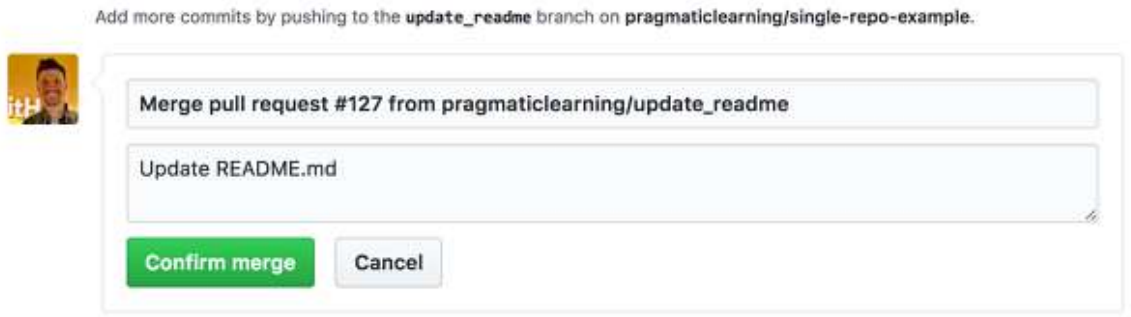


Figure 4-33. Closing and merging a pull request

You should have some kind of policy for closing pull requests. Many teams will require one or two people other than the primary author of the pull request to provide a :+1: before a pull request is merged. Have some kind of process, but keep it as lightweight as needed. Remember, you can always revert a merge, so if the code you're working on does not have the possibility of endangering anyone's life, it's generally better to "move fast

and (occasionally) break things" than have a list of 27 people who need to approve every single pull request before it can be merged. However, as mentioned a few times already, if there are strict requirements they can be added in the protected branches and required statuses.

Who Should Merge a Pull Request?

One question that often comes up is whether a pull request should be merged by the person who created the pull request or by someone else. I generally recommend that pull requests be merged by the person who created them. Here's why.

Many companies have the rule that "the person who created a pull request can't merge it." The reason for this is to make sure that someone doesn't just create a pull request and merge it in without getting any feedback. The idea is good, but I don't think the recommendation is ideal.

Most of the time, the person who created the pull request is the person who knows the most about it. As such, I always want to have that person available when her work is merged in just in case it breaks something unexpected. One of the easiest ways of making *sure* that she's around is to ask her to do the merge. So I'd recommend asking people to merge in their own pull requests, but making it clear that they shouldn't do so until they've got at least a couple of :+1:s from the rest of the team, or any other required workflows and statuses have happened.

Pull Request Notifications

If you create a pull request, comment on one, commit to one, or are @mentioned in one, by default you'll be subscribed to the pull request. This means that whenever anyone comments on, commits to, merges, or closes the pull request, you'll be sent a notification. You can see on the right side of Figure 4-34 that I am currently subscribed to this pull request.

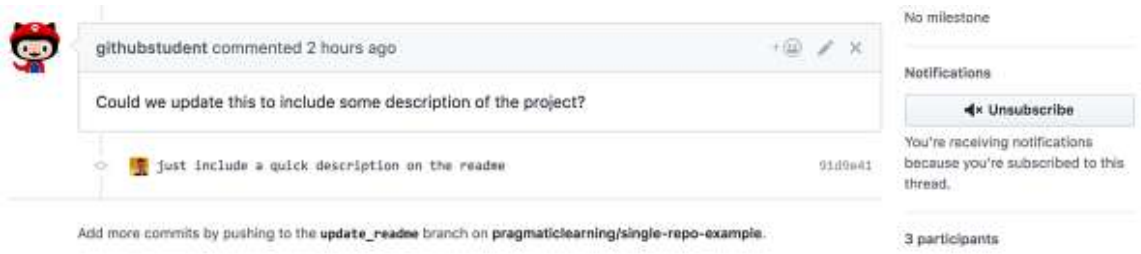


Figure 4-34. I'm subscribed to this pull request

If you're no longer interested in a pull request that you've been subscribed to, just click the Unsubscribe button and you'll stop receiving notifications. You will get re-subscribed automatically if anyone @mentions you again in the comments. If you're *not* subscribed to a pull request that you'd like to keep an eye on, just click the Subscribe button, as shown on the right in Figure 4-35, and you will start getting notifications of any activity on that pull request.

Figure 4-35. The Subscribe button on a pull request

Best Practices for Pull Requests

There are a few best practices that are worth bearing in mind when working with pull requests:

Create pull requests for everything
: Anytime you want to fix a bug or add a new feature, make sure to do it on a branch and then create a pull request to get input before merging your work into `master`.

Make the titles descriptive
: Other team members will be looking at the pull requests to get a sense of what's going on. The title should give them a good idea of what you're working on.

Take the time to comment

: Do this even if you're not @mentioned. It'll give you a good sense of what's going on with the project and will improve the overall quality of the work.

@mention key people

: If you want feedback from marketing, legal, and the operations team, @mention the necessary users to ensure they see the pull request and make it more likely you get feedback.

Run the tests

: Make sure that at least one developer downloads the latest changes from a pull request, checks out the appropriate branch, and runs your automated tests. It isn't enough just to look at the code visually for nontrivial changes.

Have a clear policy for approving pull requests

: Most companies require that one or two people other than the primary author of the pull request review and provide a :+1: before the pull request is merged in.

Up to this point, you've had an overview of the repository, working by yourself, and working with others. Most of the actual work on GitHub is conducted around pull requests, but what about when you just want to discuss an idea before it becomes work, or if you notice a bug in someone's software and you want to talk about it? That's where GitHub issues come in. If pull requests are where you discuss and collaborate around code, GitHub issues are for discussing ideas and planning before a pull request is created. Continue on to the next chapter to find out more!

Chapter 5. Project Management

In this chapter, we'll take a look at an overview of project management with GitHub so you can better stay involved with software development even if you're not writing code. Project management on GitHub typically starts in the form of simple task items that need to be worked on with GitHub Issues, organizing GitHub issues by applying labels to them and giving them deadlines with milestones. Finally, if your work involves project managers or more teams, you may decide to collect your issues and pull requests onto boards with GitHub Projects.

GitHub Issues

GitHub Issues provides a lightweight, easy-to-use tool for managing outstanding work—whether it's bugs that need to be fixed or new features that need to be built. Generally, when starting a new project, someone may begin by managing both bugs and features using GitHub Issues. Later they may move to another tool like ZenHub, Waffle.io, or JIRA if they need features that GitHub Issues does not provide.

Creating a New Issue

To create a new issue, you can start by clicking the Issues tab from any repository to visit the issues page, though it may be best to test this out on your own repository first. Once there, click the "New issue" button, shown in Figure 5-1.

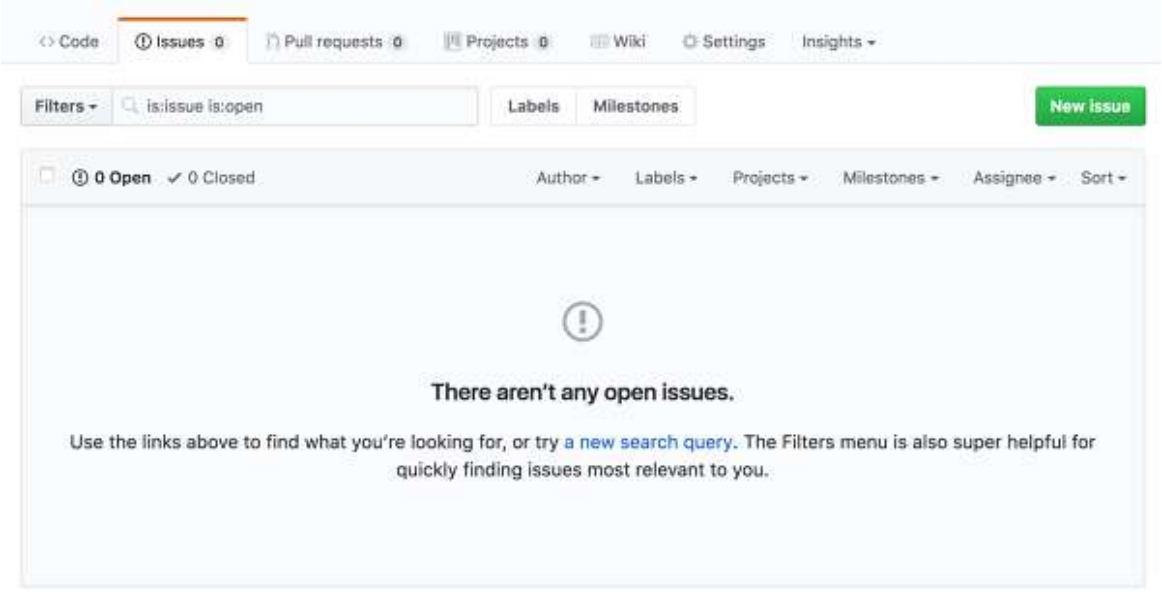


Figure 5-1. The Issues tab

When you click the green “New issue” button on the right side of the screen, you’ll see a form similar to Figure 5-2 for entering the details of the issue you want to document.

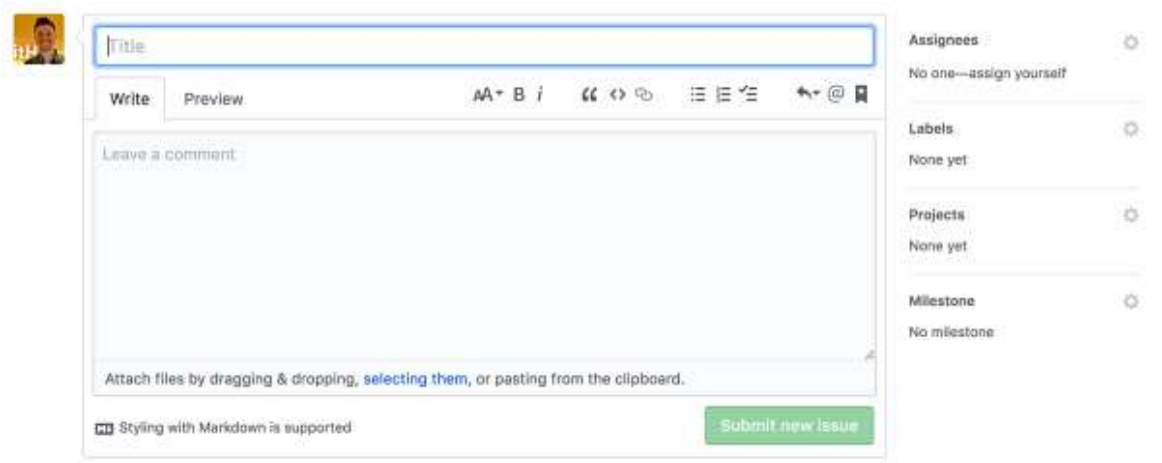


Figure 5-2. The “New issue” form

Enter a descriptive title that will quickly give people a sense of the bug or feature you want to describe, and then enter a more comprehensive description below in the comment field, @mentioning any teams or individuals that it would be appropriate to start a conversation with. If you have access and know who should be working on the issue, you can select

that person from a drop-down list of collaborators by clicking “Assignees,” in the top-right corner of the screen. Additionally, you can select a milestone if you’re assigning issues to sprints or other deadlines, select all of the labels that apply, and even assign the issue to a project if you know of one. When you’re finished, click the green “Submit new issue” button at the bottom of the page to create the issue.

Managing Milestones for Issues

The milestones feature of Issues is often used to assign issues to a particular sprint or an external deadline, like “Sprint week 34.” To add a new milestone, revisit the issues page of your repository by clicking on the Issues tab at the top of the page. Then click the Milestones button in the middle of the screen, to the right of the issues search box and the Labels button. On the righthand side of the milestones page you’ll see a button to create a new milestone, as you can see in Figure 5-3.

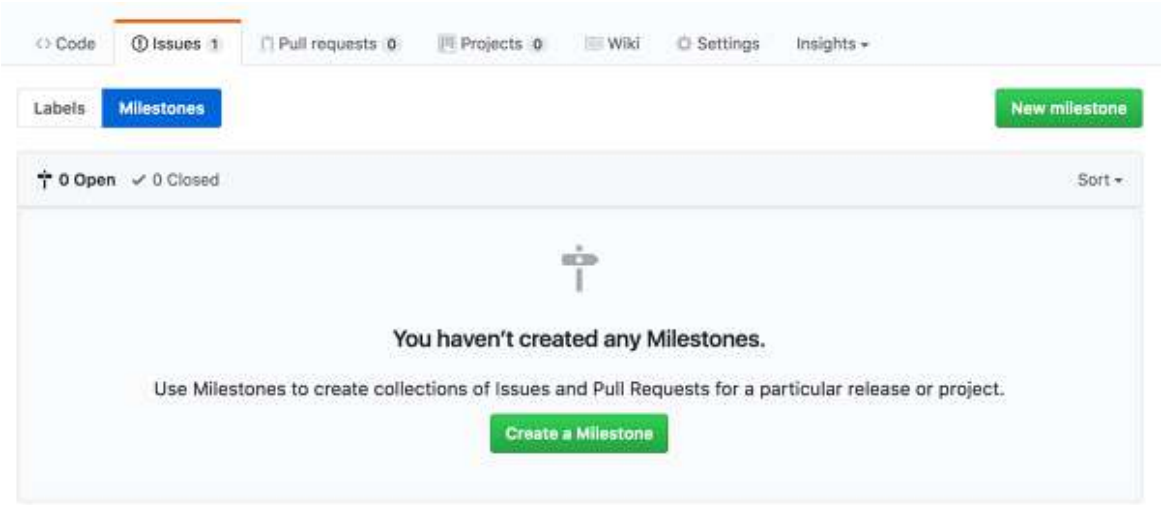


Figure 5-3. The milestones page

Click the “New milestone” button and you’ll see a form similar to Figure 5-4 asking you for a title, an optional description, and an optional due date.

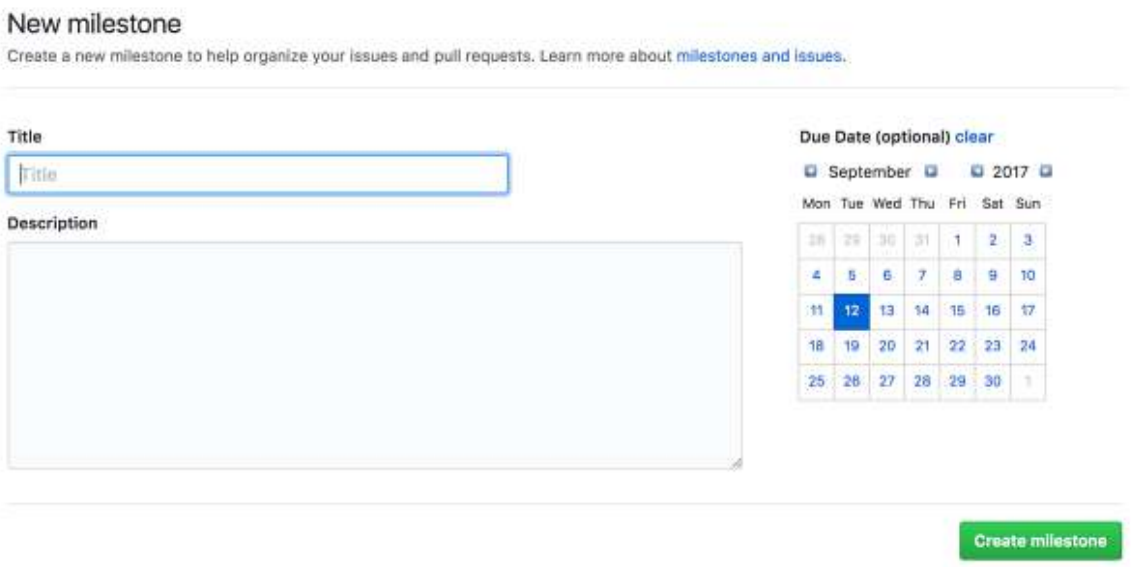


Figure 5-4. Adding a new milestone

Enter at least a title and click the “Create milestone” button at the bottom right of the page; you’ll see the new milestone added to your list of milestones, as shown in Figure 5-5. You can now edit the milestone, close it, delete it, or browse a list of the issues associated with the milestone.

Figure 5-5. The new milestone created

Managing Labels for Issues

You’ll probably also want to create some custom labels for your project. Click the Labels button in the upper-left portion of the screen next to Milestones. From this page, shown in Figure 5-6, you’ll be able to edit titles and colors, as well as delete and create new labels.

To delete a label, click Delete on the right side of that label’s row. To edit a label, click Edit; the view will change to allow you to edit both the text and the color for the label, as shown in Figure 5-7.

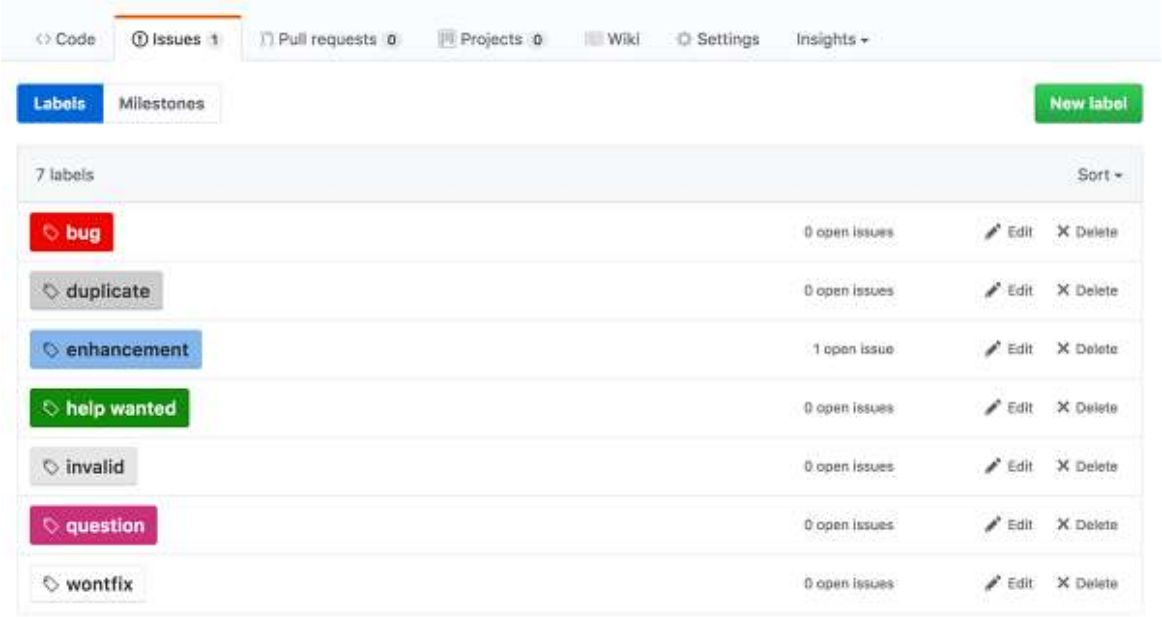


Figure 5-6. The labels page

Figure 5-7. Editing a label

If you want to add a new label, click the “New label” button and you’ll see a text box, a set of colors to choose from, and a “Create label” button, as shown in Figure 5-8.

Figure 5-8. Creating a new label

Commenting on Issues

As with pull requests:

- To comment on an issue, just visit that particular issue, scroll down to the comment box, enter your comment, and click the Comment button.
- Make sure to take a little time every day to see if there are any new issues in your repository, and respond to any @mentions as soon as you can. You will probably receive an email notification or web notification about these comments, which will make checking issues easier!
- Feel free to use emoji and animated GIFs to add a little fun to the process of collaboration.

Referencing Issues in a Commit

If you make a commit that either relates to or fixes an issue, just include a hash symbol (#) followed by the number of the issue in the commit message, and the commit will show up in the issue's history. Prefix the issue number with a word like "closes," "fixes," or "resolves" if the commit solves the issue, and when that commit is merged into your default branch (usually `master`), the issue will be closed automatically!

Best Practices for Issues

Here are some best practices to consider when thinking about how best to use GitHub Issues:

Create descriptive labels like "feature," "blocker," or "docs"
: This will make it easier to understand what the issue is about.

Use milestones if they fit your workflow
: If you have either external deadlines or an internal cadence based around something like sprints, feel free to use milestones to assign issues to delivery dates. If you don't use date-based deliveries, consider using milestones (without dates) to group like pieces of work. For example, you could have a milestone for "Complete site redesign" and another one for "Launch ecommerce features."

Don’t be afraid to reassign issues or add more assignees

: If someone can’t complete the task anymore, someone else needs to complete it, or maybe it requires more than one person to complete, reassign it and have a conversation about that change in workflow.

Make extensive use of labels

: In addition to high-level labels to distinguish “bugs,” “features,” and other work, you can use labels for many other purposes. Consider adding labels to track the status of work, to assign the work to different groups (“iOS,” “server side,” “frontend,” etc.), and even for tracking other interesting information like the severity of a bug or the business objective that the new feature is designed to support.

GitHub Projects

Using GitHub Projects can be a great way to organize your work into clear buckets to define states. You can create project boards for specific feature work, comprehensive roadmaps, or even release checklists. With project boards, you have the flexibility to create customized workflows that suit your needs. Project boards are made up of issues, pull requests, and notes that are categorized as cards in columns of your choosing. Cards can be moved from column to column and reordered according to your needs.

Projects can be created for just a single repository or, as is mentioned in Chapter 7, across multiple repositories within a single organization to track really large projects. We’re going to focus on creating just a single project first to get the hang of things.

Creating a Project Board

Anyone with read access to a repository can view the repository’s project boards. To create a project board, you must have write access to the repository. So, when it’s your own repository you’re working with, this of course means you can create a project and add cards, issues, or pull requests to it. If you click on the Projects tab at the top of the page, you should see a helpful screen like in Figure 5-9 to get you started. Go ahead and click the “Create a project” button.

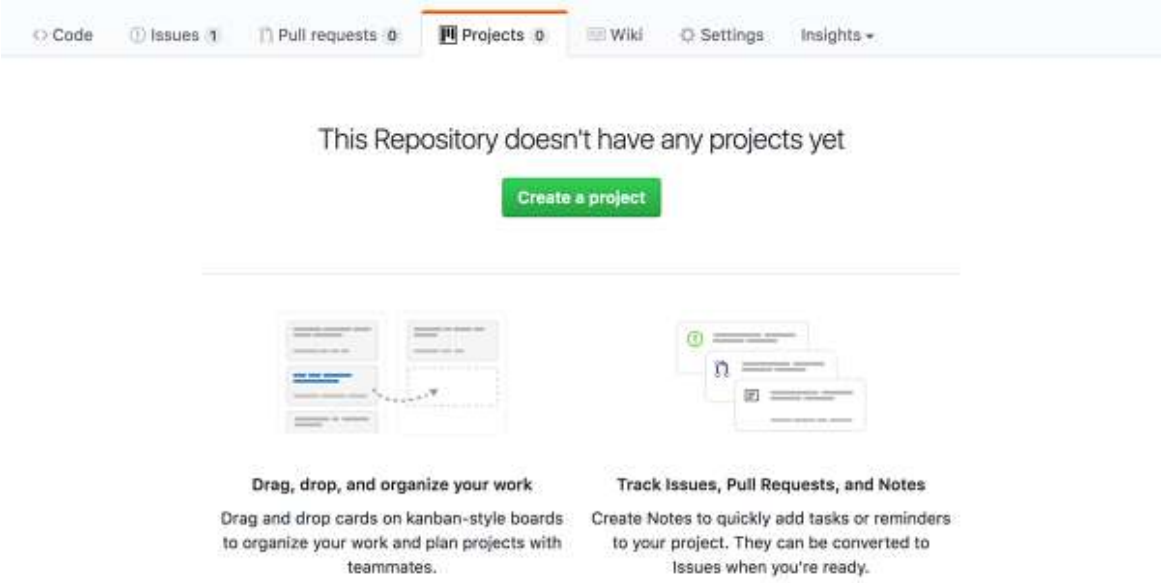


Figure 5-9. Empty project page

The page you get taken to, which should look like Figure 5-10, is similar to the "New milestone" page, and it's just as easy to create one. Give your project a name and a brief description to get started.

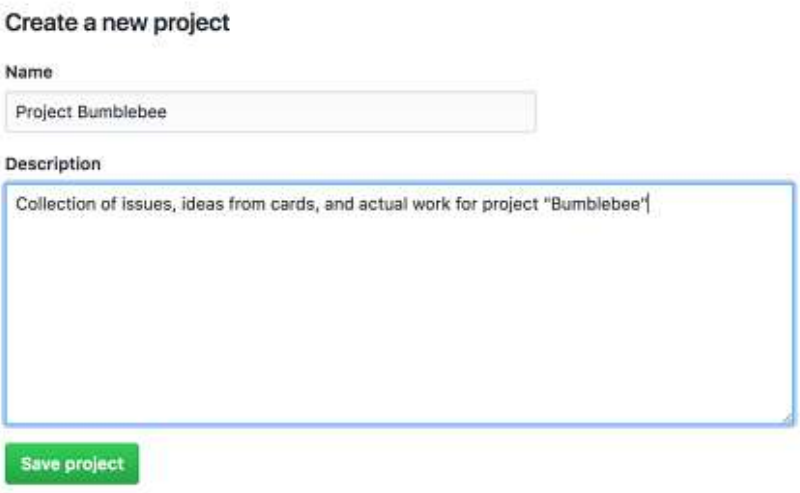


Figure 5-10. The "Create a new project" page

Creating Columns and Adding Cards

Now that there's a project board, you can make some traditional columns to start putting some work into. To begin, click the "+ Add column" link. You

will see a page similar to Figure 5-11 asking you to add a title. I'm going to do this a few times to create columns named "To Do," "In Progress," and "Done." You can see what the finished product looks like in Figure 5-12.

Figure 5-11. Create a column to start adding cards

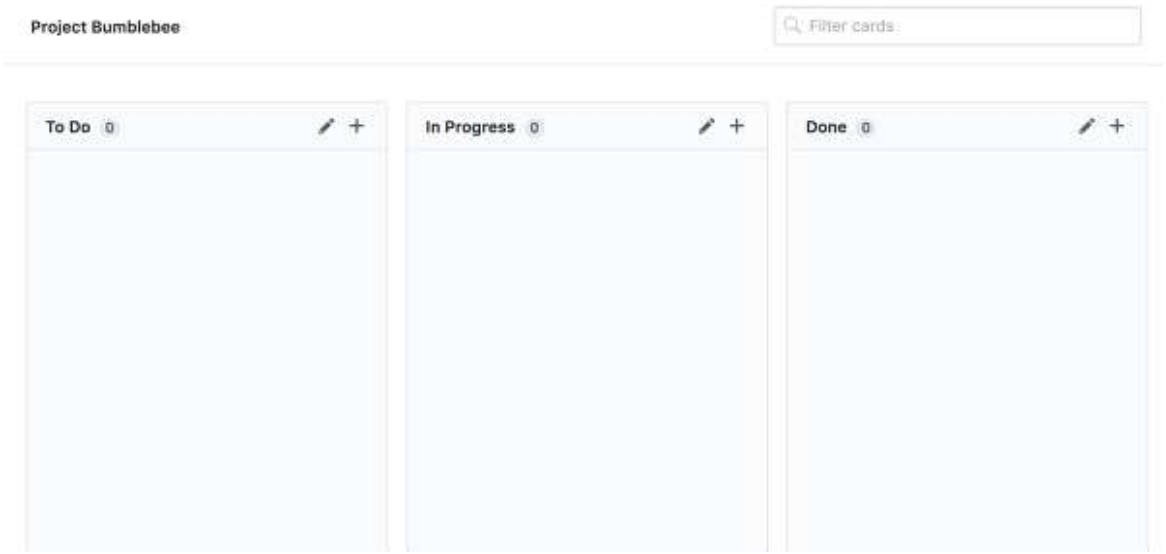


Figure 5-12. Traditional project board columns

Now that there are columns in your project board, you have two ways to add items: you can add notes from the top of a given column by clicking the "+", or add issues and pull requests by clicking "+ Add cards" on the righthand side of the page. You should explore this a bit to get familiar with both options. If you don't have any issues or pull requests, there won't be any to add from that view. Once you've added some cards, you can move them from one column to another by clicking and dragging them around. A more complete board with some cards added is shown in Figure 5-13.

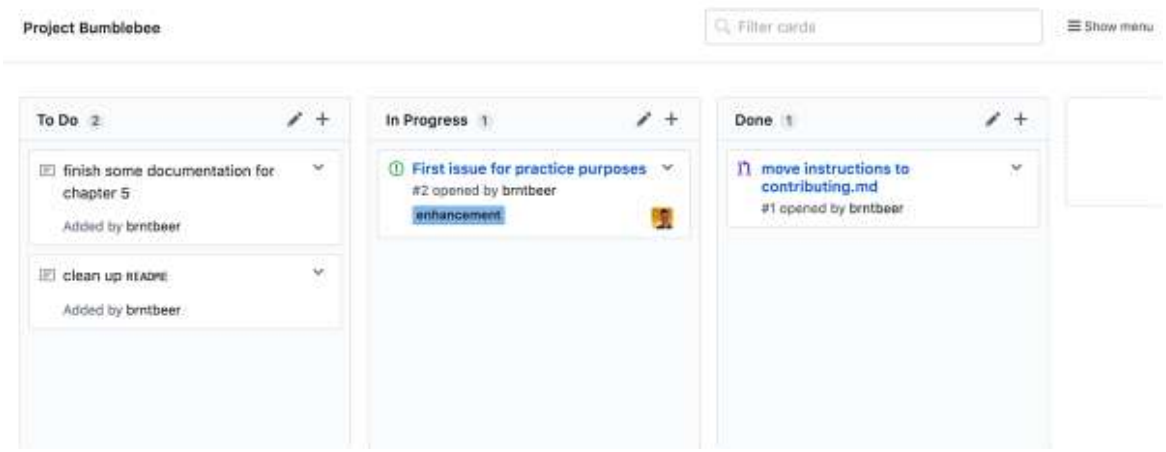


Figure 5-13. A project board with some cards in it

Closing, Editing, or Deleting Project Boards

There are a few more administrative tasks that can be done for the project board as a whole. You may want to close your project when all the work is completed, edit the description or title if it changes, or even delete it if you made a mistake in creating it in the first place.

Closing a project can be done from the “Show menu” button above the columns on the right side of the screen. Clicking it will show you a screen like Figure 5-14, and from there you can select “Close project” to show your project is completed, or reopen it if it needs to be revisited.

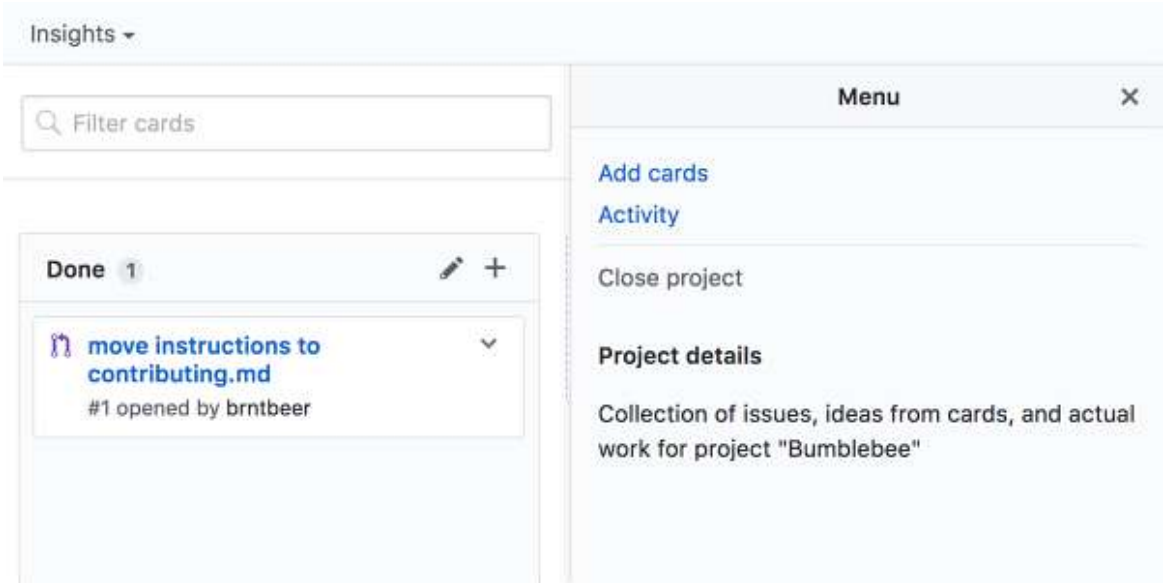


Figure 5-14. Close a project to show that it’s done

Lastly, editing and deleting a project can be done from the settings icon that looks like a gear on the far right of the screen, to the right of the Fullscreen button. Clicking it will bring you to a screen like the one shown in Figure 5-15, where you can change the title or description, or click the “Delete project” button to delete the project. Don’t worry, you’ll be asked to confirm your choice if you click this button!

As you become more and more comfortable with GitHub Projects, you may want to create more boards, edit the column titles, and convert notes to issues by clicking the drop-down arrow on them. Every action that you take on a project board can be tracked, as seen in Figure 5-16, by going to “Show menu” and then clicking on the Activity link.

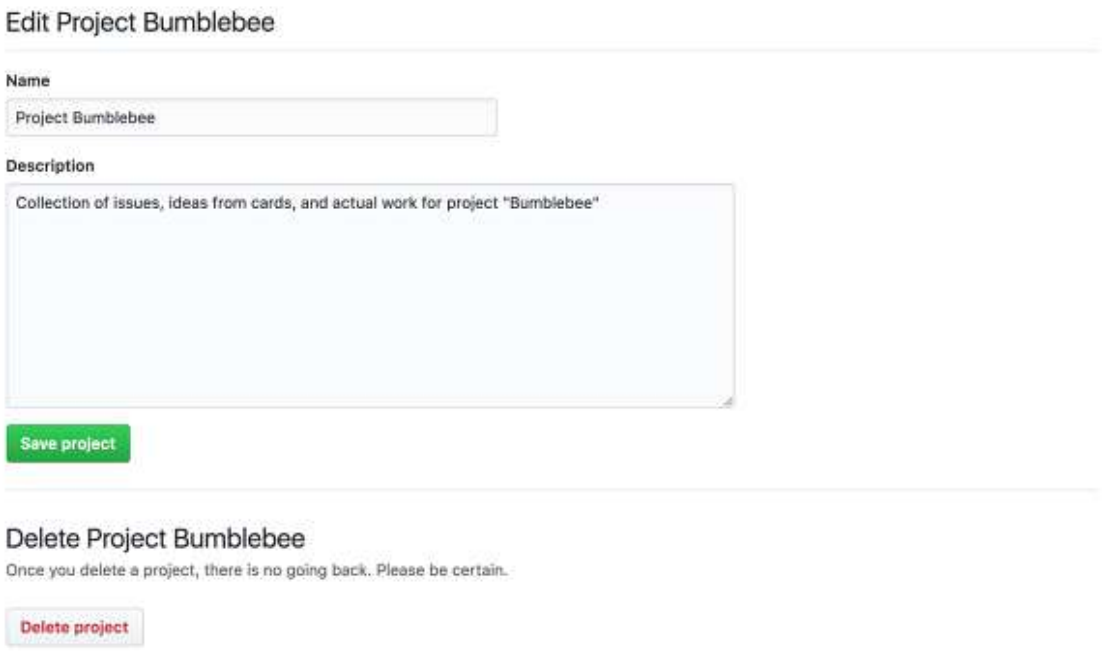


Figure 5-15. Editing or deleting a project

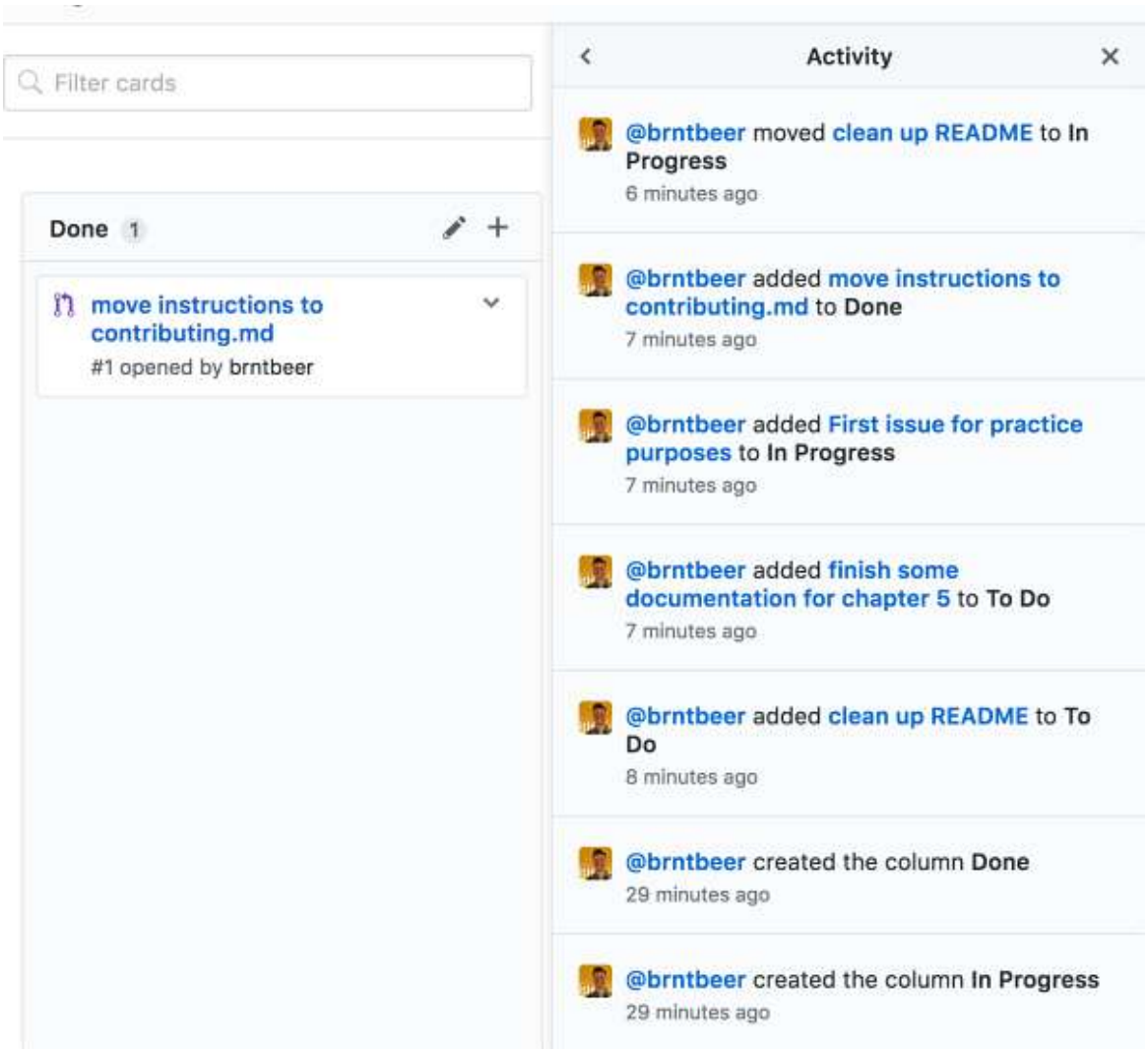


Figure 5-16. Viewing the activity feed of a project board

As of the writing of this book, GitHub projects are still pretty new and changing as more people adopt them into their workflow. So don't forget to visit your project boards regularly to see if anything has changed!

Chapter 6. Publishing Content

Providing good documentation for how to get started on a project, troubleshoot certain problems, and share ideas, or just providing a better landing page for people to find out about your project, can be just as important as the code itself. It can help new developers or users learn how to fix common problems and how to contribute back to build a stronger ecosystem, or allow you to find new contributors for your project. In this chapter we'll be taking a look at writing and maintaining higher-level documentation on GitHub so that you can encourage more people to contribute to your repository or more effectively use your code and applications.

Wikis

If you're familiar with wikis, continuing to use them is a great way to have some additional long-form documentation. Though the workflow for using them and accepting changes to them is different from any other contribution workflow on GitHub (you can't use pull requests), it's a good way for your users to find out more about your project if you have not yet created a GitHub Pages site.

A wiki is a very simple content management system that makes it easy for a group of collaborators to build a set of interlinked pages. Typically, GitHub's wikis are used for capturing end user documentation, developer documentation, or both so that all of the information relating to a project is accessible through the project's GitHub page.

Getting Started with a Wiki

If your project doesn't yet have a wiki, start by going to Settings and scrolling down to the Features area, as shown in Figure 6-1. Make sure that the Wikis checkbox is selected. This is also a chance to check the next box if you're going to be creating a public project and want to limit it so that only collaborators on the project are able to update the documentation on the wiki.

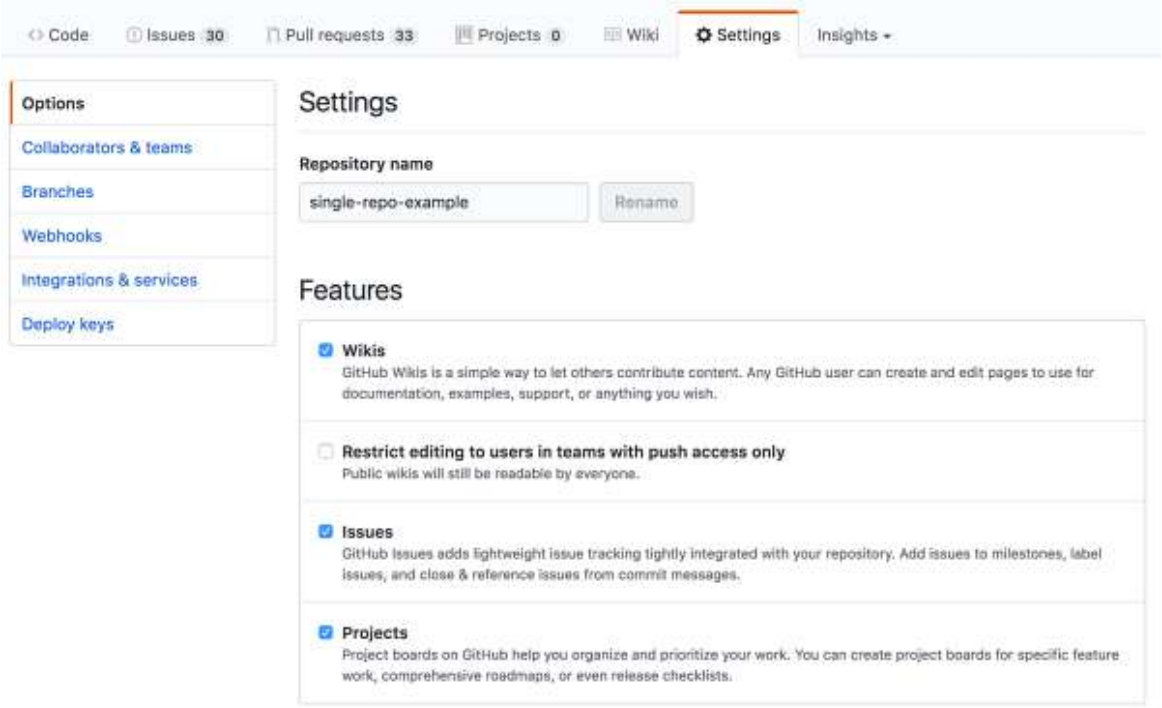


Figure 6-1. Ensuring that wikis are enabled

Once you've ensured that you have wikis enabled, click the Wiki tab at the top of the page. If you haven't yet added any content, you'll see a page like Figure 6-2.

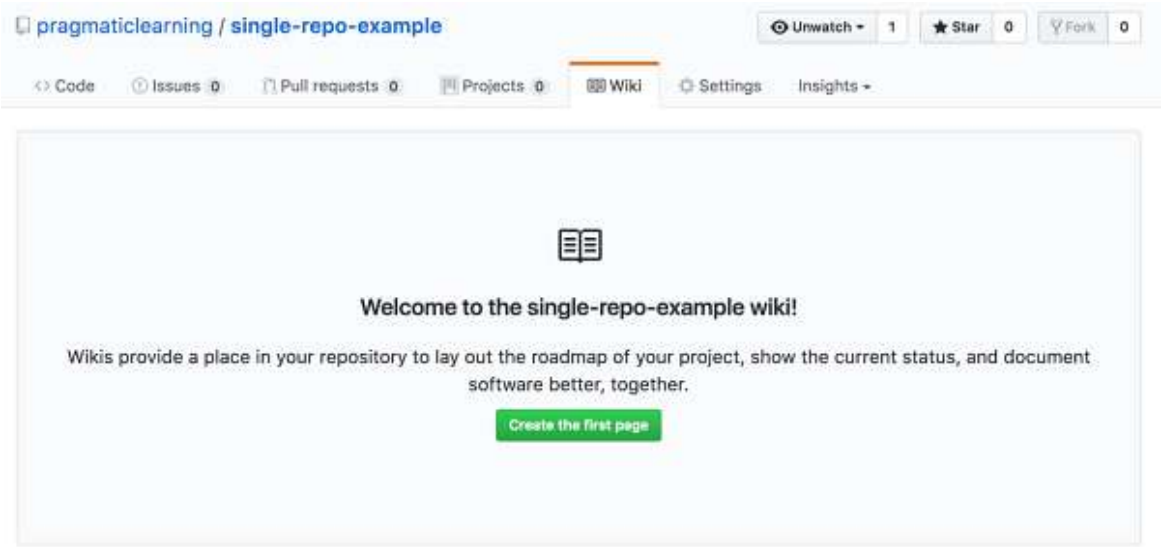


Figure 6-2. The default wiki page

Click the green “Create the first page” button, and you’ll see a page similar to Figure 6-3.

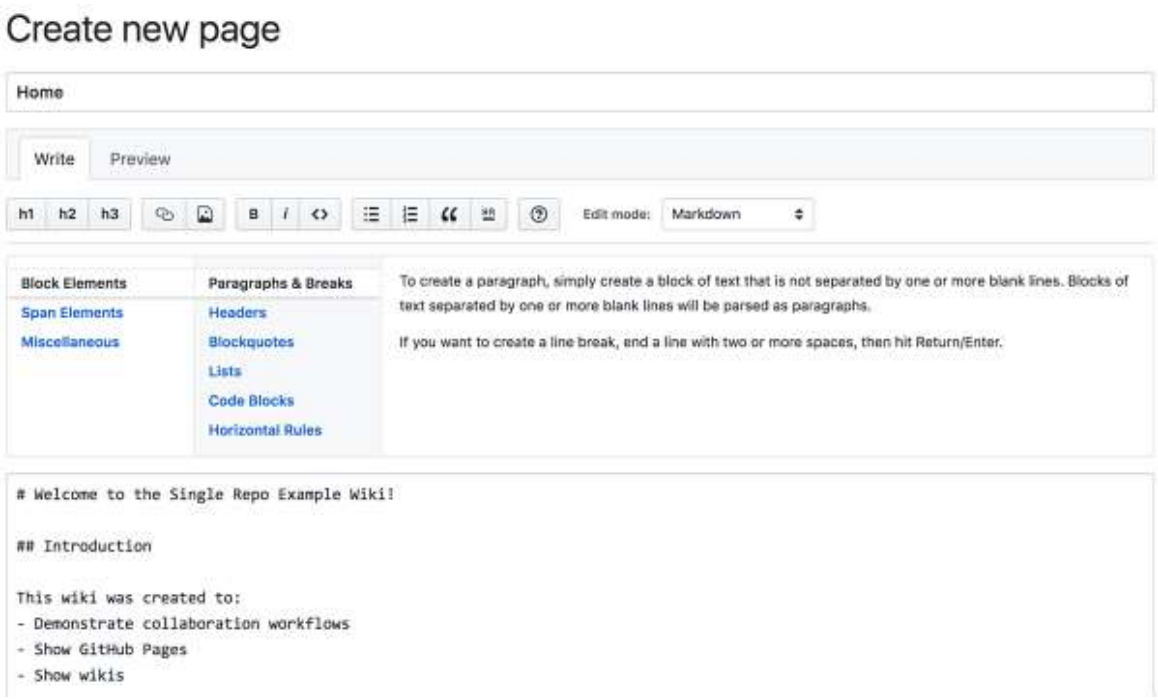


Figure 6-3. Creating your first wiki page

By default the first page is called “Home,” although you can change this by editing the title. Then you can enter your content in the text area. You’ll notice that there are a number of buttons above the text area for styling, but this is deliberately not a full, in-place WYSIWYG (what-you-see-is-what-you-get) editor. Instead, the buttons will just insert the appropriate Markdown into the text area. If you want to see what it’ll look like, click the Preview tab above the formatting buttons and you’ll see the Markdown rendered, as shown in Figure 6-4.

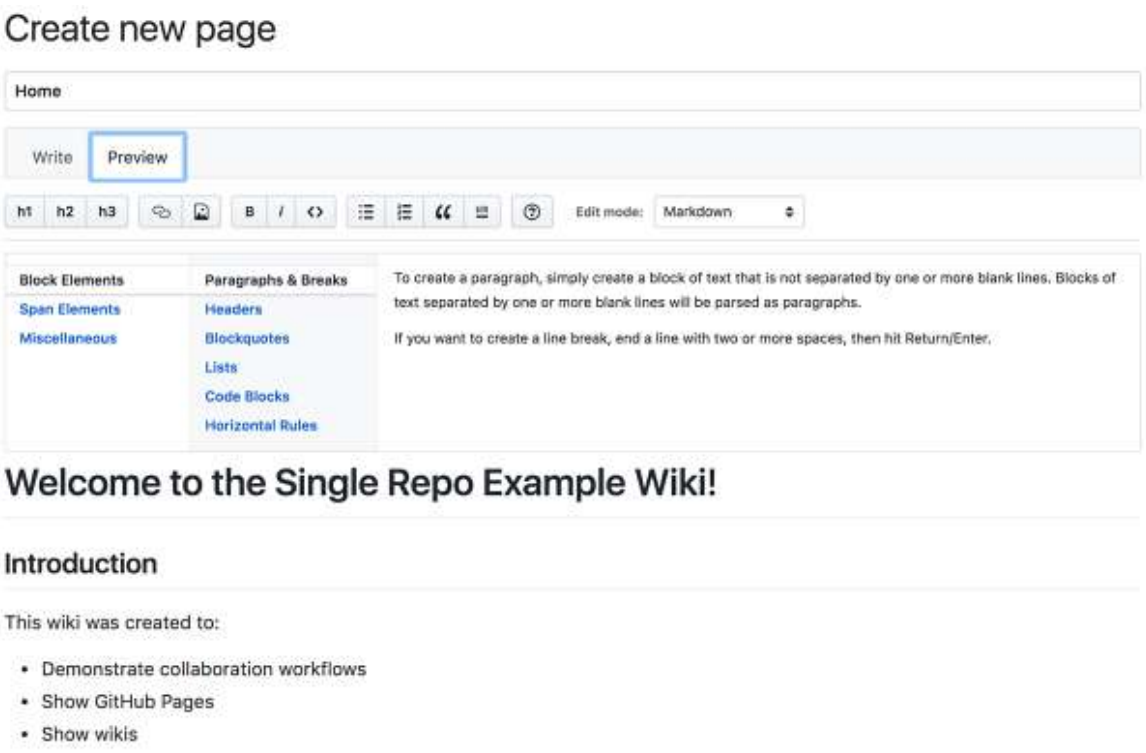


Figure 6-4. Previewing your new wiki page

If you click the “Edit mode” drop-down list, you get the option of changing to a range of different selected formatting syntaxes, as you can see in Figure 6-5. However, I’d recommend using Markdown as it’s the same format used by the GitHub team and is used in other areas within GitHub, such as issues and pull request comments.

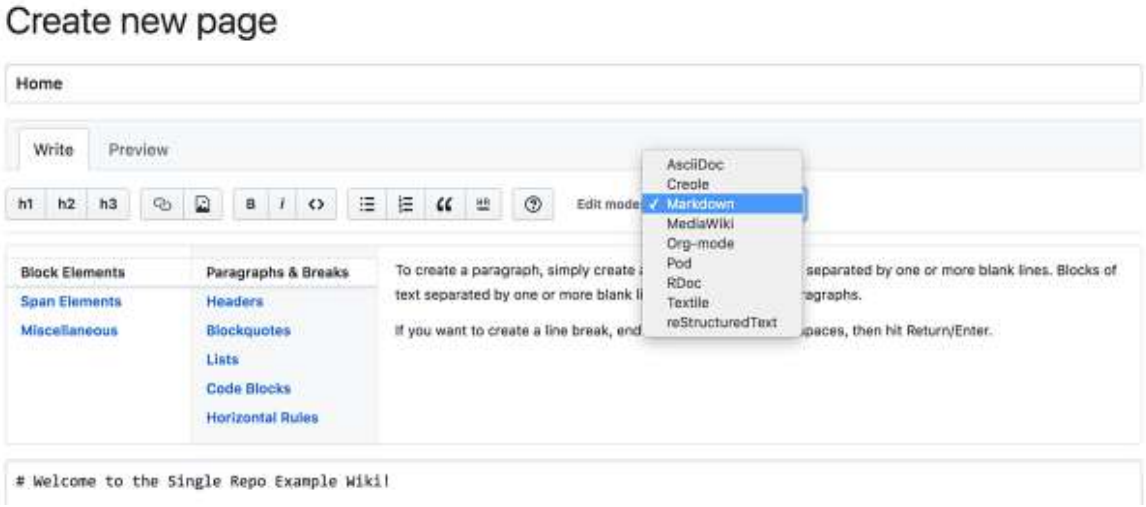


Figure 6-5. Alternative editing formats

When you're done with the content, enter a short (optional) description in the "Edit message" text box to describe why you made the change, and click the "Save page" button.

Adding and Linking to a Page on Your Wiki

Anytime you want to add a new page to your wiki, just click the New Page button at the top right of any wiki page and it'll allow you to add a page to the site. Once you've added the page, it will appear in the Pages section to the right of the screen, as you can see in Figure 6-6.

Figure 6-6. The Pages list in a wiki

To add a link to a new page from an existing page, start by using the Pages list to navigate to the page you want to add a link *to*. Then copy the URL for that page to the clipboard, from your browser—you'll need that in a moment. Next, use the Pages list to navigate to the page you want to add the link *on*. Click the Edit button at the top of that page, to the right. Go to the place in the content area where you want to add the link and click the link button in the top bar (it looks like two circles linked together). Clicking it pops up a dialog box, as shown in Figure 6-7.

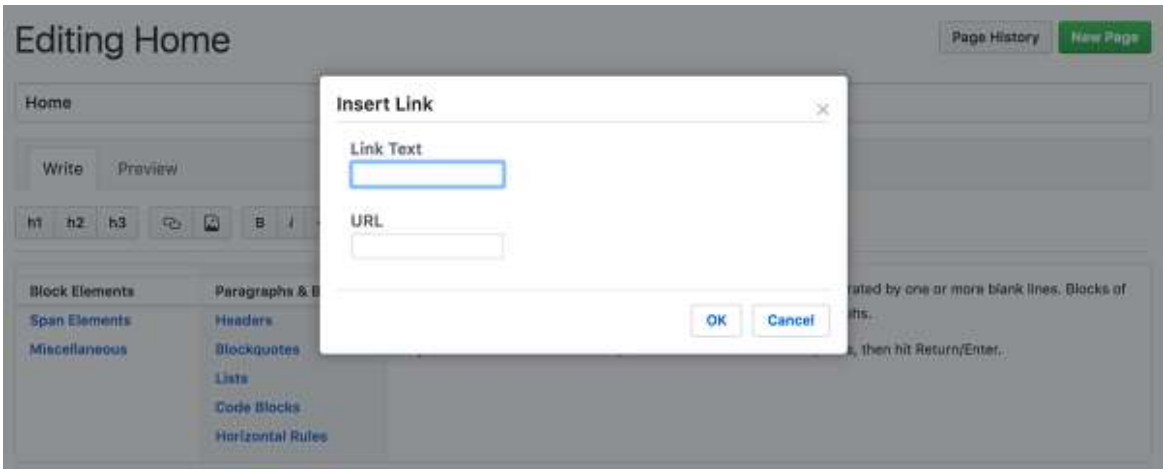


Figure 6-7. The Insert Link dialog box

In the first box, type whatever you want the link text to be—ideally something that describes the page it's linking to. Then, in the URL text box, paste the URL of the wiki page you want to link to from your clipboard.

If you'd really like to make the most of your wiki (or issues or pull requests), make sure to check out the Mastering Markdown guide, which provides a really good introduction to GitHub-flavored Markdown.

GitHub Pages

If you want to create a more customized website to share information about yourself, your organization, or your project, that's where GitHub Pages comes in. GitHub Pages is a feature that allows you to create and host web pages right on GitHub by either turning that rich Markdown documentation into a beautifully rendered web page or writing HTML and CSS yourself.

Creating a Website for Your Project

Whenever you create a repository on GitHub, you have the option of using GitHub Pages to add web pages for promoting or describing the project. There are typically four types of GitHub pages for a project: you can create a marketing site to describe the project in better detail, end user documentation on how to use the project, developer documentation that describes how the project works in deep detail, and community documentation for how to contribute and get support within that project's community. To get started with GitHub Pages to just build better documentation for your project, click Settings, scroll down to the GitHub

Pages area, and click the “Choose a theme” button. You’ll see a screen similar to Figure 6-8.

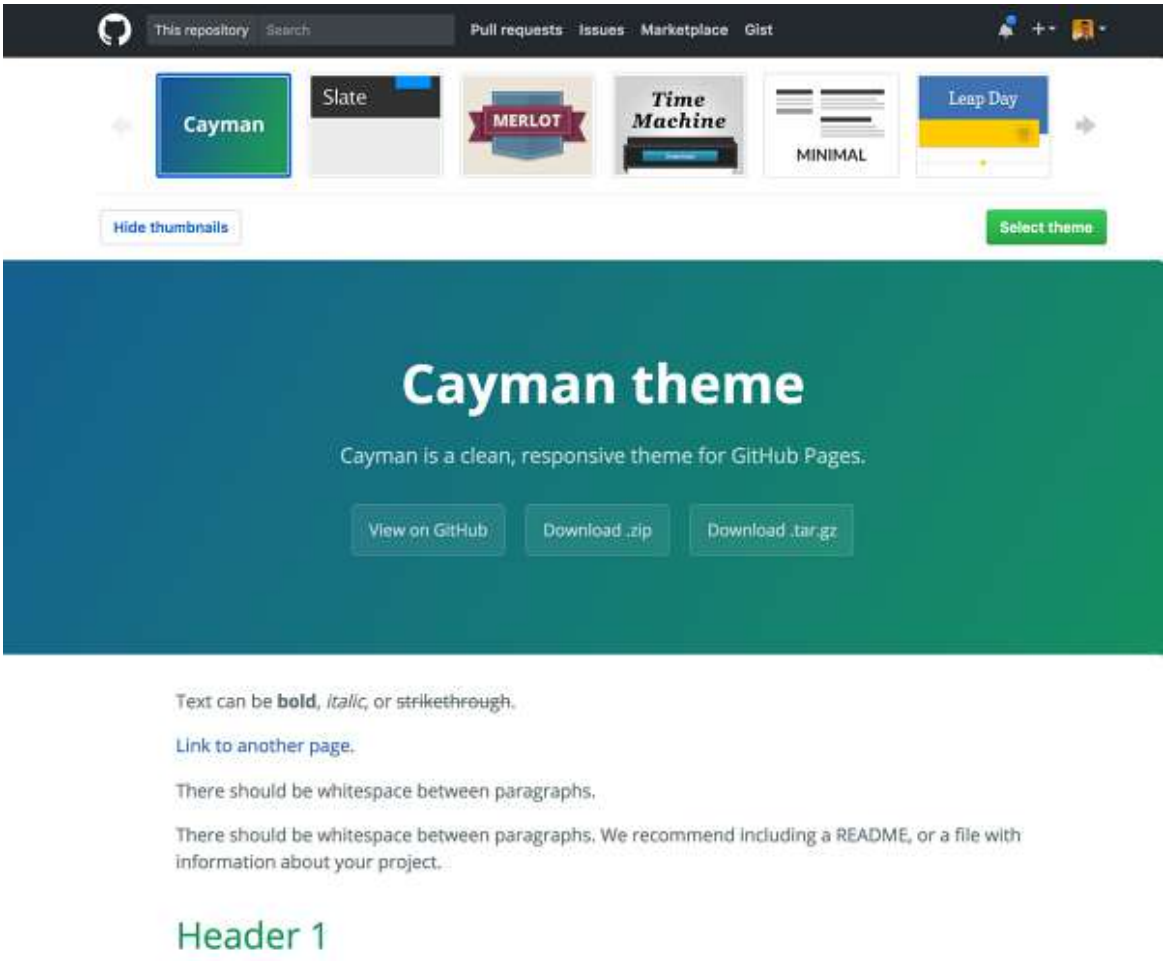


Figure 6-8. The GitHub Pages themes

When you’re happy with the look and feel, click the “Select theme” button toward the top right of the page, and your website will be created. That’s all there is to it! You can view the website at *http://<organization_name>.github.io/<projectname>*. For example, I just created a web page for the simple-repo-example project under the pragmaticlearning organization, available at *http://pragmaticlearning.github.io/single-repo-example/*.

By default, when you create a GitHub page for your repository, it looks at the files and folder structure based on the `master` branch. Some people choose to render their Pages site from their *docs/* folder to segregate their

long-form documentation from the code that drives the project. At this point, you just have *README.md* and no *docs/* folder, so that will be what ends up getting rendered in your example website. If you want to get more complex, you could start adding files like *index.html* and some CSS files for styling your Pages site. I'll save that as an exercise for the reader, but an example can be found in the facebook/react-native repository.

Creating a Website for Yourself or Your Organization

If you want to create a website for yourself or your organization using GitHub Pages, you need to create a project named "<*username*>.github.io" or "<*organization_name*>.github.io." For example, IBM has created a portal for its website, *https://ibm.github.io*, at *https://github.com/IBM/ibm.github.io* by following this exact method. Just like with a repository GitHub Pages site, you can choose to have this site built from the contents in the `master` branch, the *docs/* folder within the `master` branch, or the `gh-pages` branch.

If you want to create a website for your organization, go to the organization home page and click the New button on the righthand side of the screen, above the Top Languages and People section. Make sure to make the repository name "<*organization_name*>.github.io," and then check the "Initialize this repository with a README" checkbox, as shown in Figure 6-9.

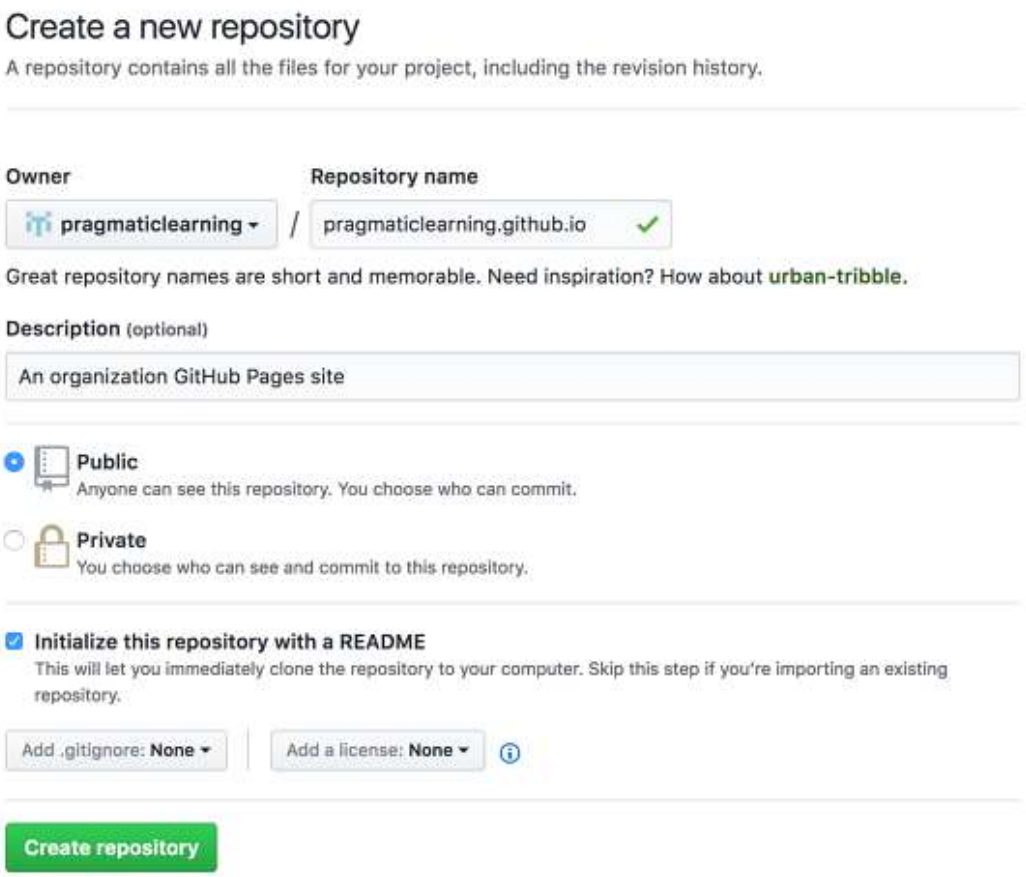


Figure 6-9. Creating a GitHub Pages repo for an organization

If you create such a project, click the Settings tab, and scroll down to the GitHub Pages section, you'll see that it shows that the site has already been published as a GitHub Pages website (see Figure 6-10).

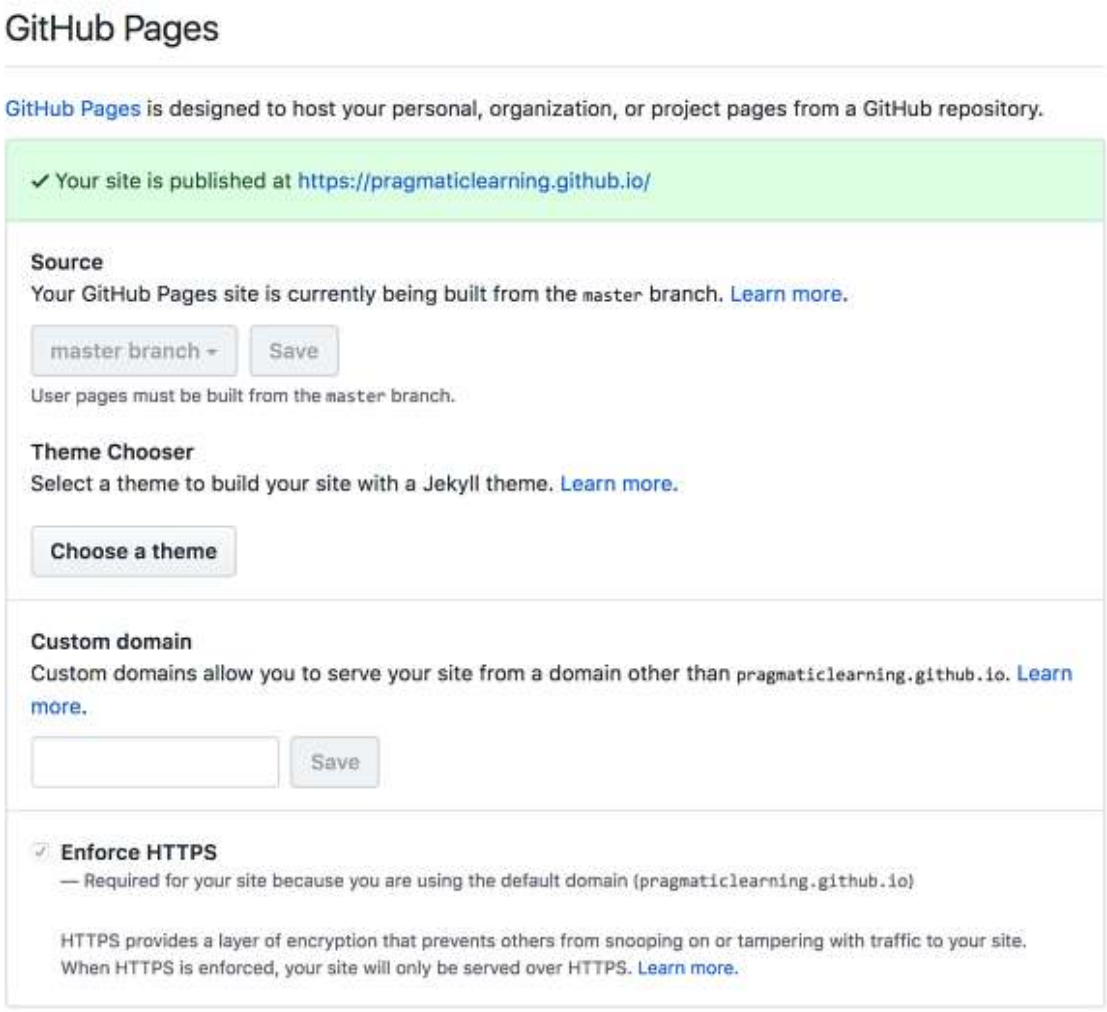


Figure 6-10. The Settings tab for a GitHub Pages organization site

If you know HTML and CSS, you can continue to build your website here. However, if you'd like to make it a bit easier to build a website without just HTML and CSS, like the GitHub Training Team has done, you may also want to check out the Jekyll project. Jekyll allows you to easily structure your website and keep some of those pages in Markdown so it remains easier to update them and make changes. Switching to Jekyll or having a more robust strategy for the documentation of your project will not only provide a place for people to find out about your project, but will also ensure your project is contributing to a healthy ecosystem on GitHub.

Chapter 7. Configuring Repositories and Organizations

So far we've looked at how to view, edit, and collaborate on repositories. In this chapter we're going to take a step back and go through the process of configuring a GitHub repository for a new project.

Warning

If you're working with developers on a contract basis, you'll want to create the repository they use to work on. Creating the repository means that you'll always have access to the code and the additional information contained in pull requests, issues, projects, and wikis. Once you've created it, you can then add the developers as collaborators so they'll have access to the repository—until you decide to revoke it. You do *not* want contract developers to create the repository for you. If they do, they'll be able to remove *you* from the repository at any time.

Configuring a Repository

To configure a repository, start by clicking the Settings tab at the top of the page. By default you'll go to the Options menu within Settings, as shown in Figure 7-1, which allows you to configure some high-level settings.

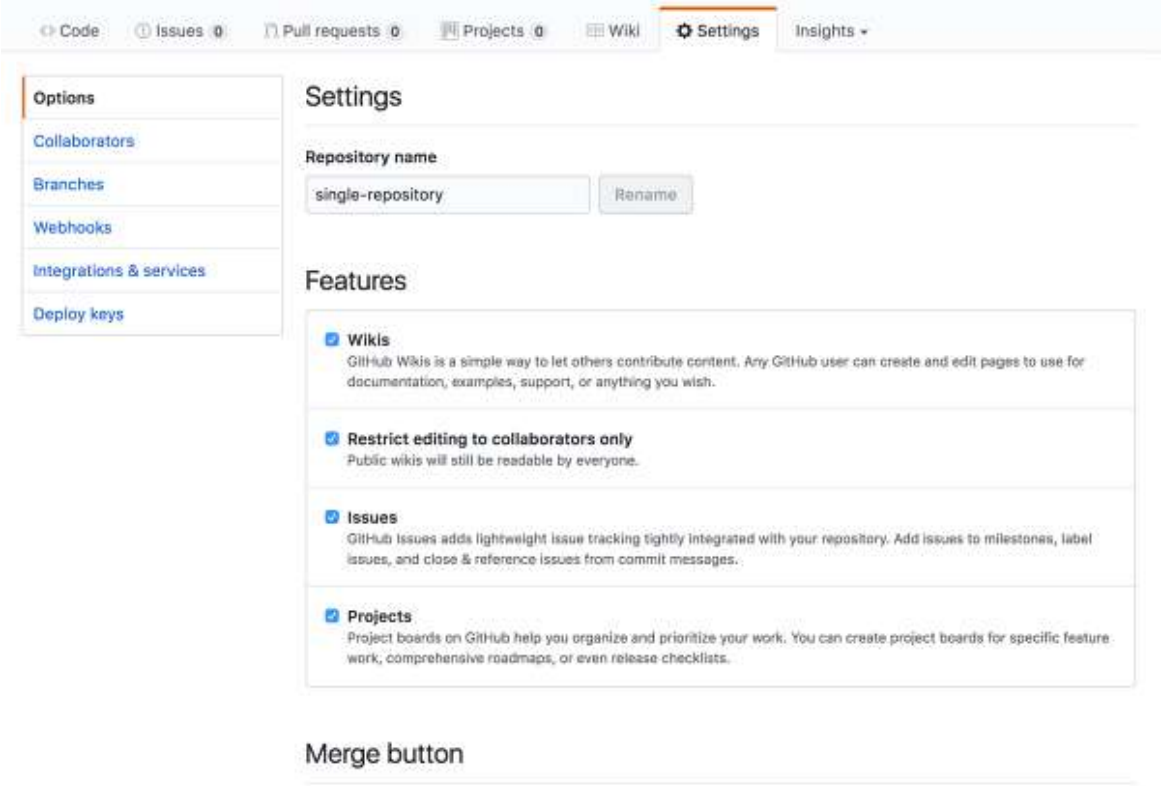


Figure 7-1. The Settings→Options screen

The first available setting to change is the name of the repository. If you change the repo name in the text box, the Rename button will become active, allowing you to change the name of the project. Don’t worry if your developers are already connected to the project. They won’t have to change anything to visit the project online—anybody using the old name or URL to access the project will be redirected automatically. However, anyone who is accessing the repository from the command line or from a graphical user interface like GitHub Desktop will need to update some settings to point their local copy to the correct URL.

On the Settings→Options screen, you also get the chance to configure wikis and issues. By default, new repositories have wikis, projects, and issues enabled. Just uncheck the boxes to disable them if, for instance, you would rather use GitHub Pages for documentation and aren’t looking for other users to open issues. If you want to limit the wiki on a public project so that only collaborators can edit the content, check the necessary box.

As you go further down the Settings→Options screen, you'll see some additional configuration settings: "Merge button," "Temporary interaction limits," GitHub Pages, and the "Danger zone." We discussed using GitHub Pages in Chapter 6, so let's take a look at the other sections.

As shown in Figure 7-2, the "Merge button" section has additional options for how you want work to be brought in from pull requests. As you and any other developers working on your project are likely to have your own preferences in terms of workflows, it may be important to be able to merge this work in different ways. Though not every permutation is possible in the web interface, you can allow for a better experience for common workflows. If none of these options is preferred by you or your team, you may have to set up additional automation or perform the actions yourself on the command line to finish a merge.

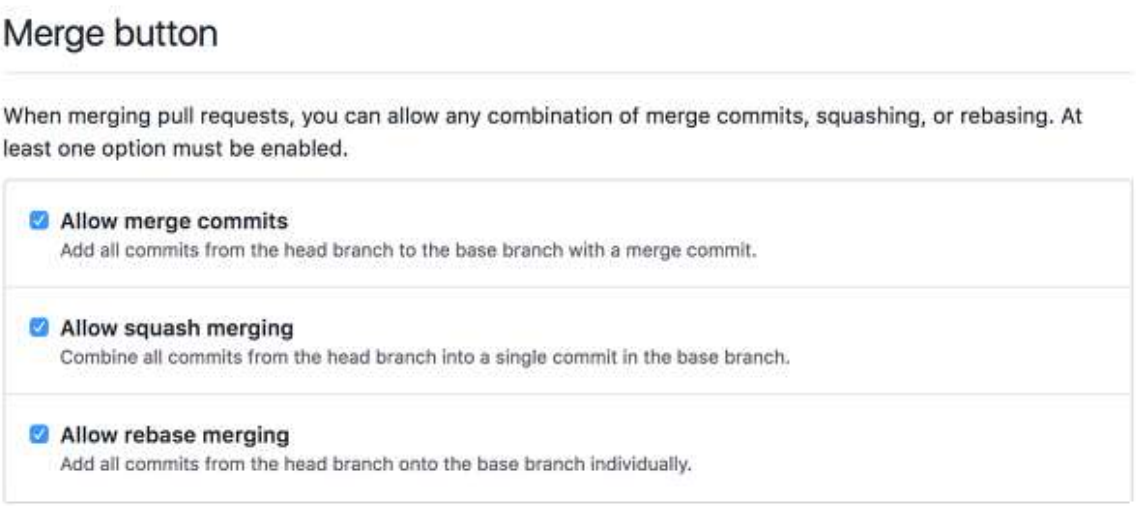


Figure 7-2. Merge button options

Sometimes when a conversation in an issue or pull request starts to get quite heated, a maintainer may want to let people cool down for a while until a good resolution can be thought up. That's exactly why the "Temporary interaction limits" section exists. It limits who can comment, create pull requests, or create issues for a 24-hour window and will automatically release the lock once that window is over. This is similar to locking an issue or pull request for comments, but applies to all conversations and even the creation of new ones until the time limit is up. This is another way that

maintainers can help manage good software development practices for everyone involved in a project, regardless of whether it's open source or private. Different ways to configure these limits can be seen in Figure 7-3.

Finally, we come to the "Danger zone." This section allows you to change the accessibility of a project between private and public. It also gives you the option to transfer the ownership of the project to another user or organization and, if you really want, to delete or archive the repository. Don't worry about hitting the "Delete this repository" or "Archive this repository" button accidentally. If you click either of them, you'll be asked to confirm that you really want to do that, as shown in Figure 7-4.

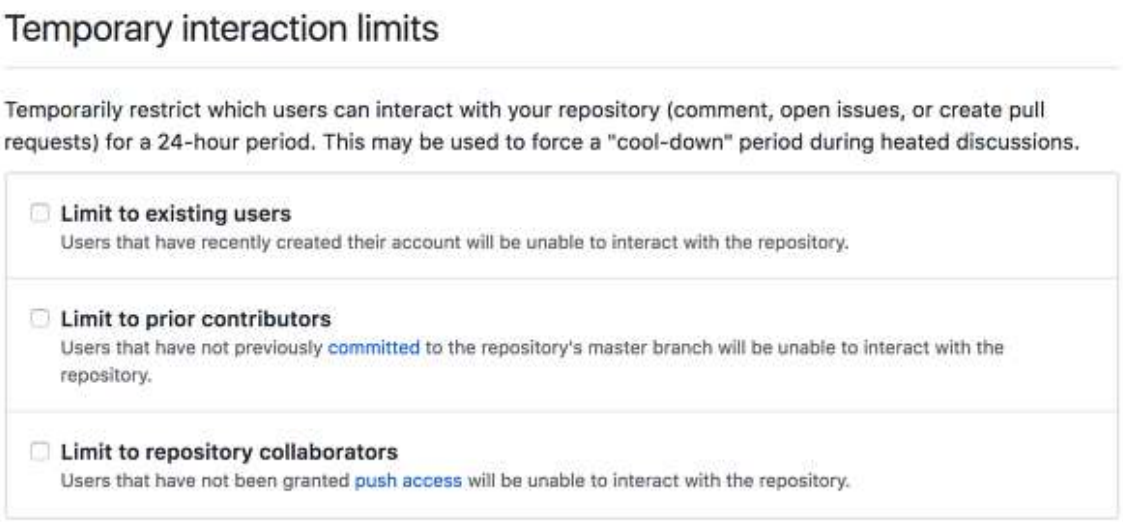


Figure 7-3. Temporary interaction limits

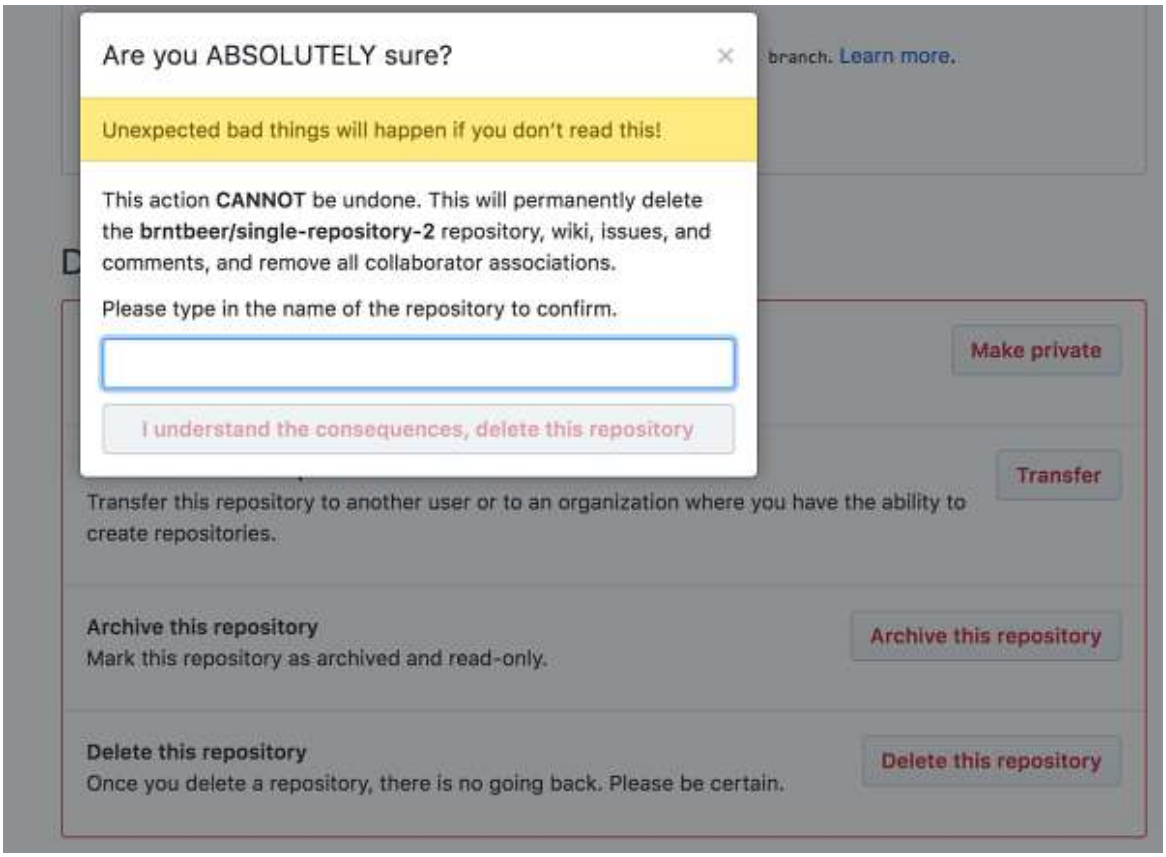


Figure 7-4. The "Delete this repository" confirmation pop-up

Adding Collaborators

Once you've created and initialized your repository, typically the next step is to add any collaborators. If you've created a public repository, you may not need to add collaborators, especially if you're just working with people occasionally. Ask them to fork your repository and send you a pull request any time they have a contribution to make. However, if you created a private repository or you have people who will be working on your code regularly, you should add them as collaborators.

If you've added the repository to an organization, you can manage access using teams, which we'll look at later in this chapter. However, if you just added the repository to your personal account, you'll have to add collaborators individually.

To add collaborators, click the Settings tab of the screen and then click the Collaborators link on the left, as shown in Figure 7-5. You may be asked for

your password just to confirm that it's you making the changes.

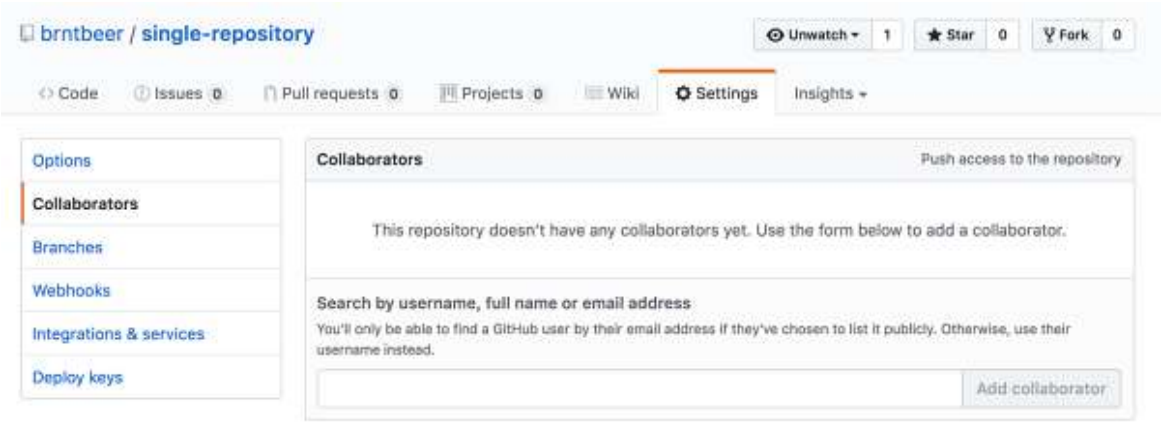


Figure 7-5. The Settings→Collaborators screen

To add collaborators, you'll need to know the GitHub usernames of the people you want to work with. Start typing a username, and the name will autocomplete, as shown in Figure 7-6. Select the autocompleted name and then click the "Add collaborator" button.

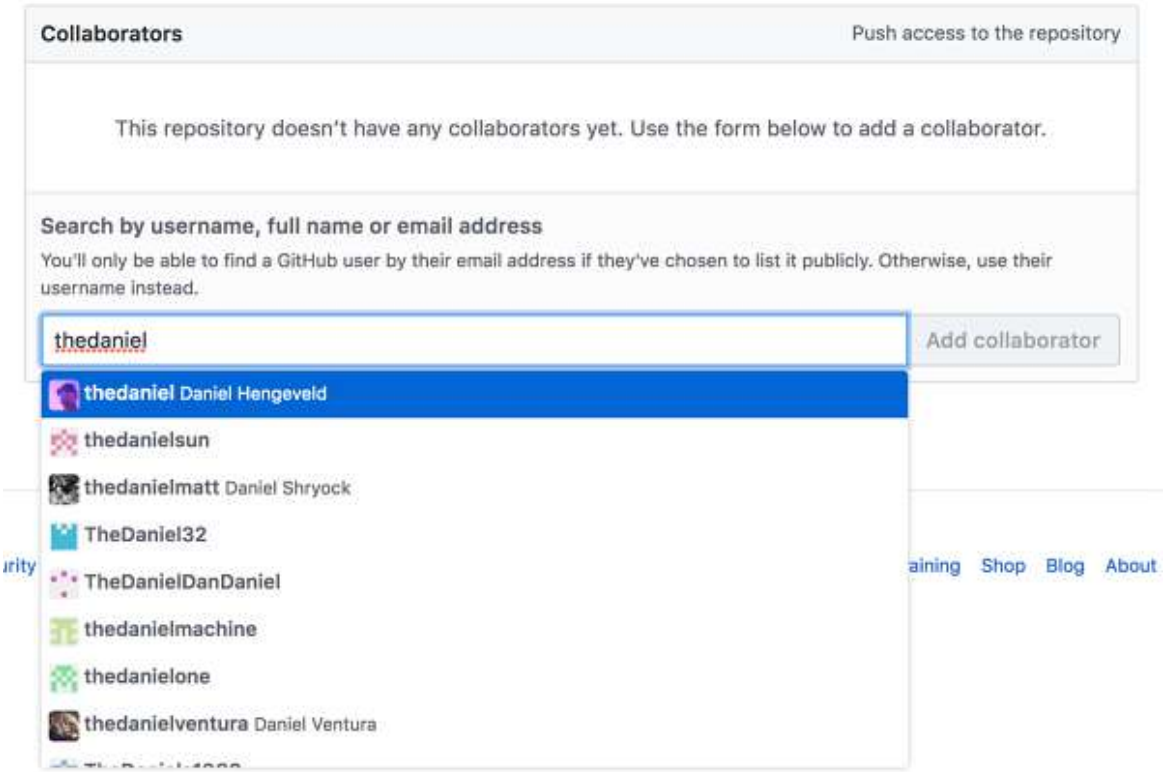


Figure 7-6. Autocompletion of a collaborator

Once you've invited collaborators and they've joined, it's worth taking a little bit of time to go through the other configuration options to see if there's anything else you want to set up.

Configuring Branches

The Branches section of the Settings tab is where you can configure your default branch, or enable some branches to be protected. For most repositories, `master` is the default branch. As discussed in other chapters and shown in Figure 7-7, the default branch setting controls which branch a pull request will be sent to by default. Generally it's best to leave this option alone, but if your development team really wanted to create a new default branch (for example, if you have a workflow that describes pull requests needing to go into an intermediate branch before going into `master` for some deployment or testing purposes), they could do so and you could make it the default branch here.

The default branch is used for features like auto-closing of issues and some workflows around Continuous Integration (CI) and Continuous Delivery or Deployment (CD). Usually, when you have a commit message that says something like "closes #10" or "fixed #10," when that commit is merged into the `master` branch, it will automatically close issue #10. However, the trigger is really when the commit gets merged into the default branch, so if you wanted to have a default branch named `trunk` or something else, you could do that if you really wanted to.

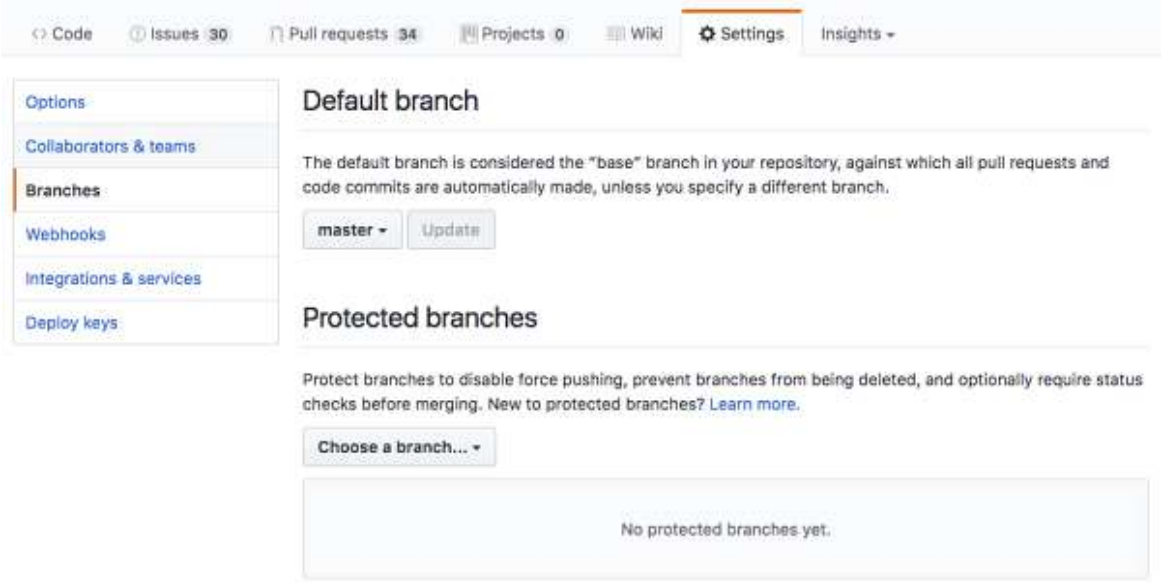


Figure 7-7. The Settings→Branches screen

Besides editing the default branch, on this screen you can also set up additional branches to be protected so that they cannot be deleted, or require other actions to happen before work ends up in those branches. Let’s take a look at protected branches in more detail.

Protected Branches

Protected branches are a way to enforce and ensure that certain workflows happen within your repository and that irrevocable changes are not made to the designated branches. Common reasons for protecting a branch are to require peer review, ensure that code quality is high or all your tests are passing by integrating with third-party services, ensure that the branch cannot be deleted or overwritten, and more. A few of these settings can be seen in Figure 7-8.

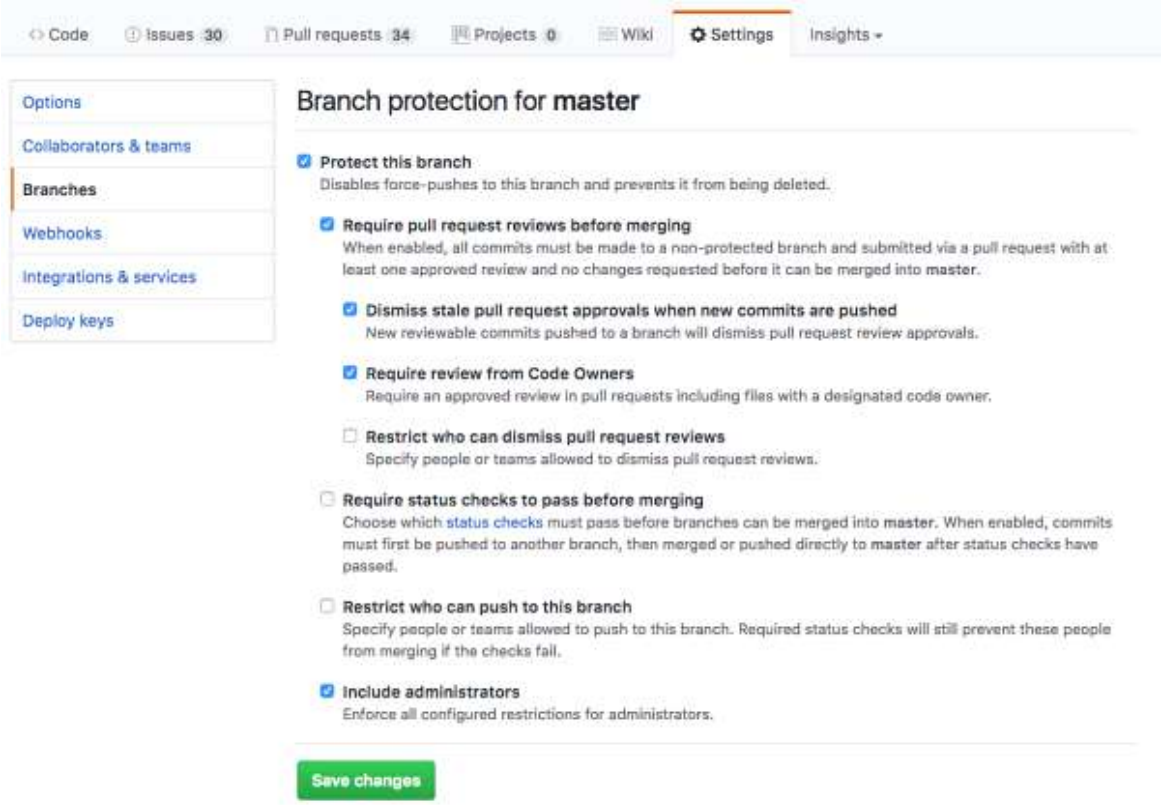


Figure 7-8. Branch protection options

Integrating with Other Systems

Sometimes you'll want to connect GitHub to other tools that fit alongside and enhance the software development process. These tools range from CI servers that regularly run automated tests to project management or bug tracking software. Let's look at some ways you can enhance your software development practices by expanding beyond GitHub.

One option is the GitHub API. Go to *https://developer.github.com*, as shown in Figure 7-9. From here, there are actually two interfaces to the GitHub API: GraphQL and REST. They both allow a developer to access data, but the way in which they do it is different. If you click through to either of these pages from the top of the page, you'll find guides to help you learn how to use the GitHub API to extend GitHub to do pretty much any custom workflow your team may have.

Figure 7-9. The GitHub Developer page

Though it's a pretty advanced topic, the API allows your developers to query and change almost anything they want in a repository or an organization—but sometimes they'll just want to be notified when a specific action occurs. For example, they might want an internal tool or some personal server to get notified every time someone adds a new issue or pushes work up to GitHub. If they want to have notifications automatically sent, they should be using the webhooks option that can be configured by going to Settings→Webhooks, as shown in Figure 7-10.

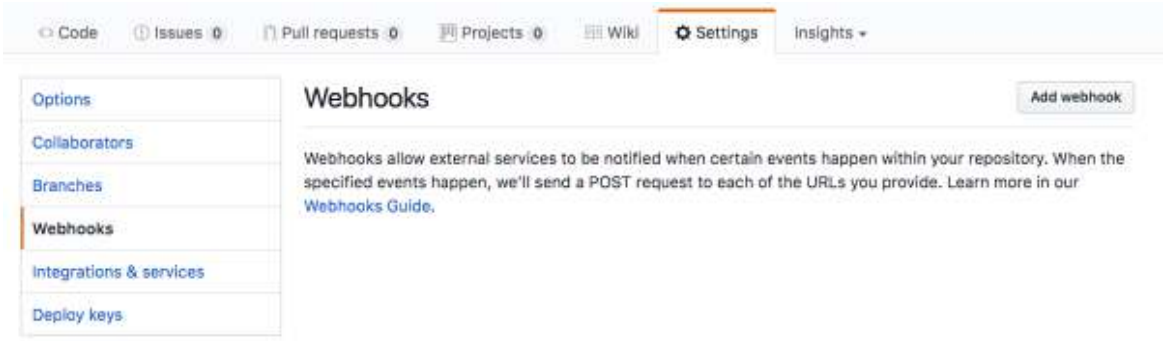


Figure 7-10. The Settings→Webhooks screen

Clicking the “Add webhook” button toward the top-right corner of the screen takes you to the “Add webhook” screen, as shown in Figure 7-11.

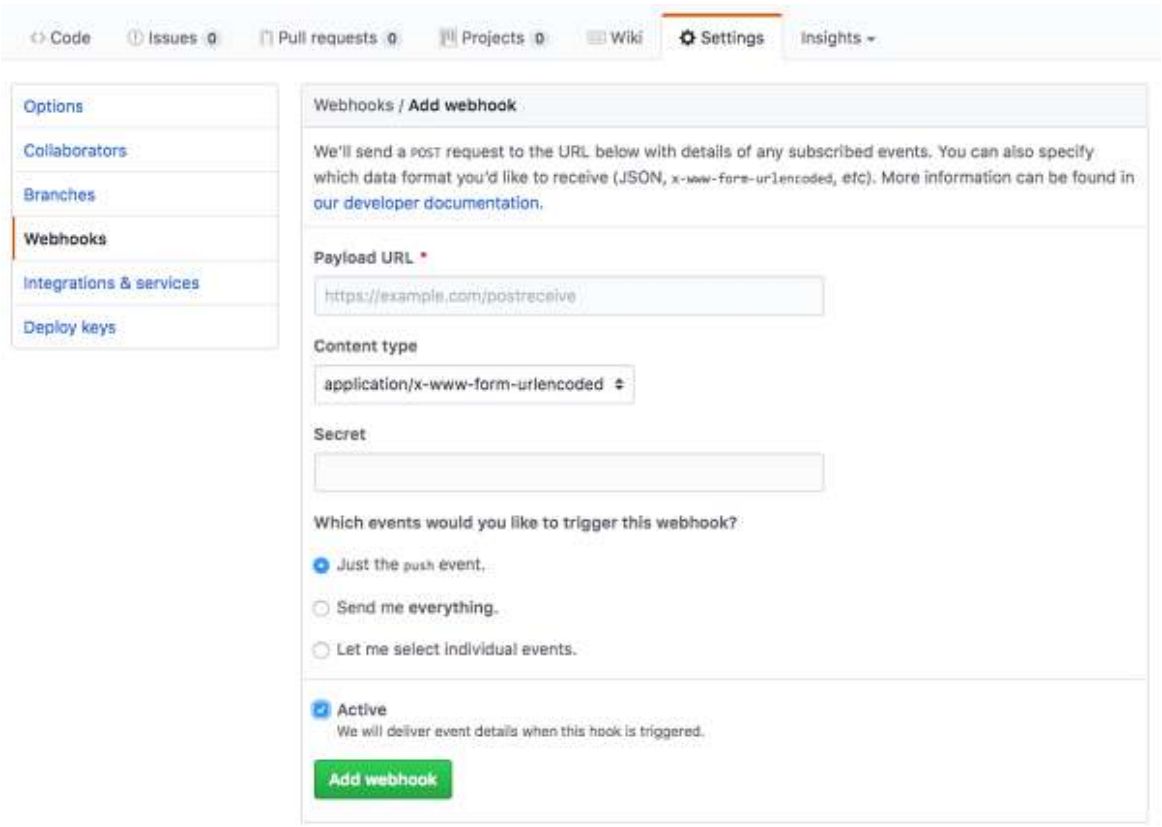


Figure 7-11. The “Add webhook” screen

This screen allows you to tell GitHub to send a notification to your custom server or application every time a particular type of event occurs. You need to provide the URL that your software will be listening on, the kind of content you want delivered, an optional secret (so that not just anyone can send fake information to that URL), and what kinds of events you’d like to have the software be notified about. If you or your developers are implementing a custom integration, you’ll probably know how the webhook(s) should be configured and what events to select from the “Let me select individual events” option. Even if you don’t have any webhooks or aren’t going to be building them yourself, knowing about these options is great so you can customize your workflows as you grow with GitHub.

The final integration option is one that developers don’t have to configure: GitHub Apps. GitHub Apps are applications that other third-party

developers have already created and that exist within the GitHub Marketplace, which can be seen in Figure 7-12.

Figure 7-12. GitHub Marketplace

When you visit *https://github.com/marketplace*, there are many categories of GitHub Apps you or your developers can choose from. If you don't know which category to start with, my suggestion is to look at Continuous Integration. CI apps handle automatically running the tests that your team would otherwise have to run manually, *and* they will report the status back to the pull request you are working on to make your code more robust and code review easier. Setting these up typically requires some additional configuration in the code of the repository itself, but your team will thank you for it. You can see some examples from this category in Figure 7-13.

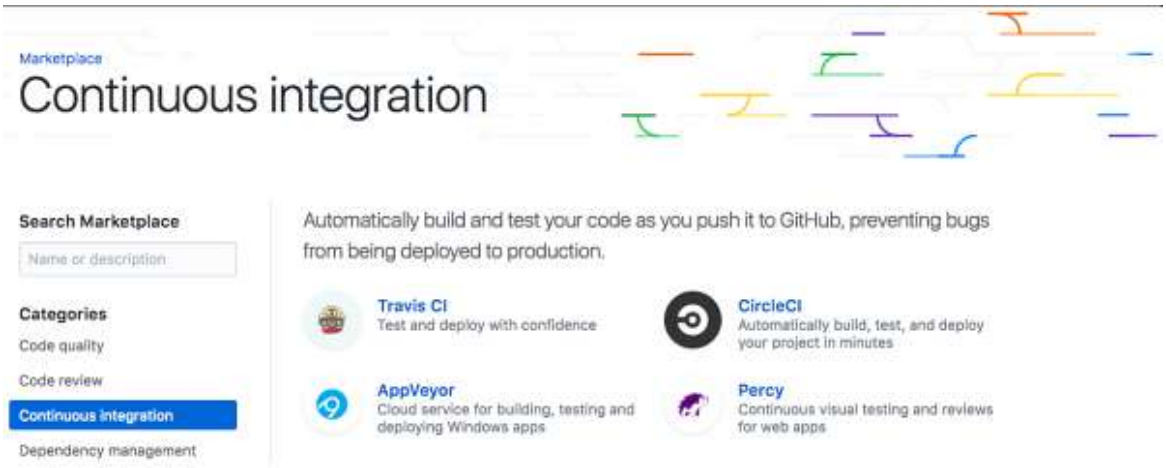


Figure 7-13. Some GitHub Apps within the Continuous Integration category

Once you set up and connect one of these GitHub Apps to any repository you have administrative access to, it should be listed on the repository's Settings→"Integrations & services" screen. For example, if you were to add the Codecov app, you would see something similar to Figure 7-14.

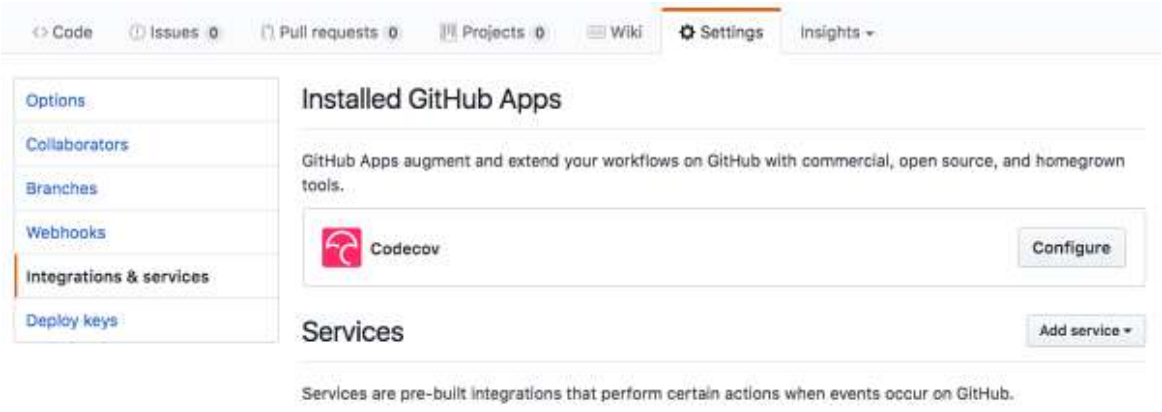


Figure 7-14. A third-party integration installed and configured

"Deploy keys" is the last menu option on the Settings tab for a GitHub repository. Clicking the link displays a page similar to Figure 7-15.

Figure 7-15. Configuring deploy keys

In addition to other people needing access to your repository, sometimes you'll want to provide the ability for other software to connect to it. For example, your development team will probably create an automated build system that will allow them to just click a button to deploy the latest changes from GitHub to your production server.

If they do that, the build system will need the ability to access the repository. There are a number of ways of providing that access. One option

is to create a *machine account*. This is where you create a new GitHub user just for your build machine and add that user as a collaborator. That's a particularly good approach if your build system needs access to a number of different repositories.

Another option is just to create a *deploy key*. A deploy key is a Secure Shell (SSH) key that is created to allow a particular piece of software to access a single repository on GitHub. Don't worry about this too much, but if your development team asks you to set up a deploy key, just ask them to email you the public SSH key and to give you a name for the key (e.g., "build server"); you can then use that information to fill out the "Add deploy key" screen, as shown in Figure 7-16. In a few situations (like merging branches), this system may need to push a change back up to the repository. If that's the case, be sure to give this deploy key write access so that it can send these changes back up.

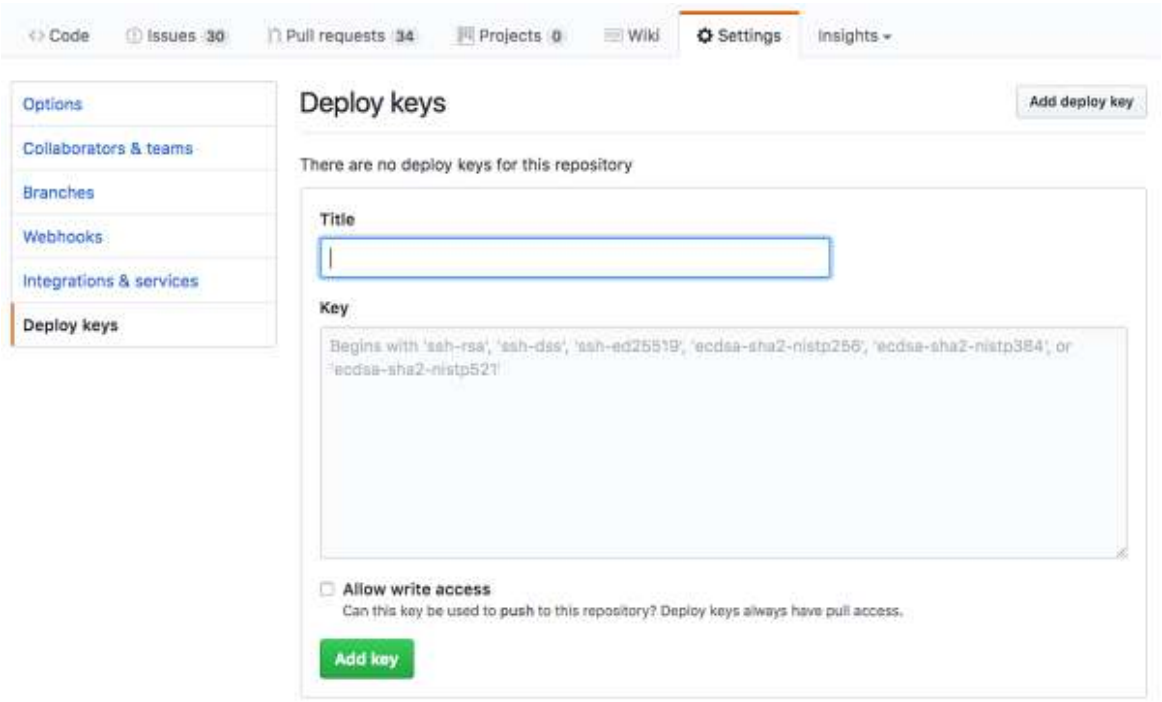


Figure 7-16. Adding a deploy key to a GitHub repo

Personal Versus Organizational

When you create a repository, the first question you need to answer is whether you should add the repository to your personal user account or

whether you should add it to an organization instead.

If you are creating a personal project, you probably want to just create it under your personal GitHub account. However, if you're creating a project that you know you will want to be owned and/or managed by an entity other than yourself—whether a whole team or a company—you should probably create an organization first and then create the project under the organization so you can easily transfer ownership of the project over time.

This isn't the most important decision. You can always transfer the ownership of a repository, so if in doubt, feel free to just create the repo under your user account. However, if you *know* that you're going to be building a project for an organization, you might want to create the organization first.

Creating an Organization

You can create an organization no matter where you are on the site, so long as you are logged in. To do so, click the + sign to the right of your username at the top right of the page, and from the drop-down list shown in Figure 7-17, click the "New organization" option. Though the options you see may be slightly different, you can always create an organization or a new repository.

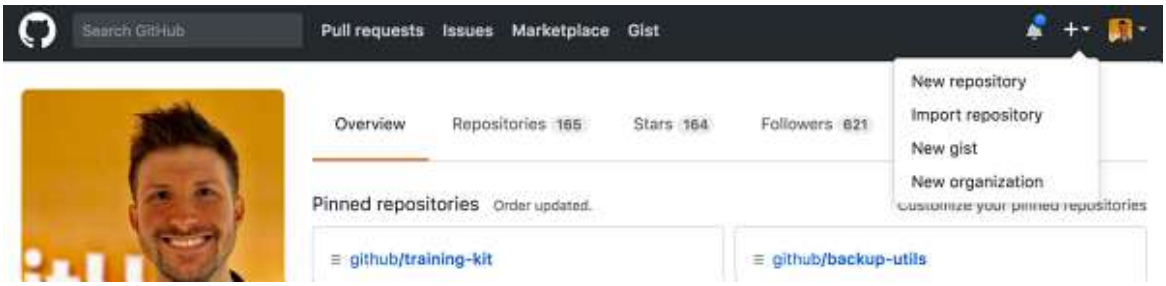


Figure 7-17. The first step in adding a new organization

Clicking this option will take you to a page similar to Figure 7-18 that will allow you to create a new organization.

Start by giving the organization a name and entering the email address for the billing contact. You'll then want to select a plan. If all of your projects are openly accessible, you can create an open source organization for free. If you want to work on private repositories with a team of people, you'll

want to select the Team plan to allow them to better coordinate around those repositories. Lastly, if you want more business features, such as SAML for single sign-on, an uptime service-level agreement, and guaranteed support response time, then the Business plan may be the one for you. If you're interested in the Business plan features but need to host the code on your own servers behind your firewall, following the link at the bottom to contact GitHub about the GitHub Enterprise plan would be best.

If you choose to create an organization for anything other than public code (fully open source), you'll be asked for either credit card or PayPal information to make the monthly payments.

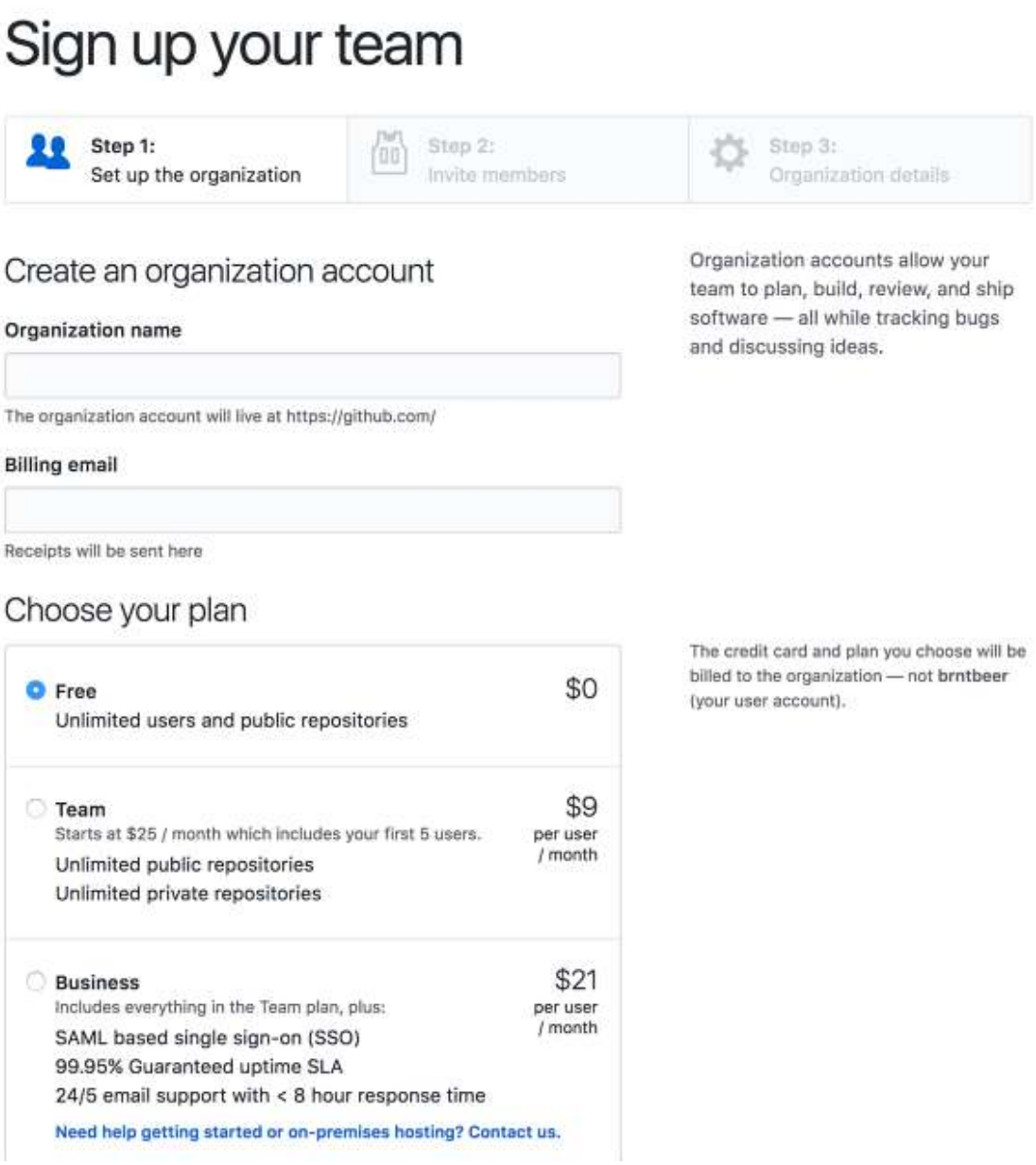


Figure 7-18. Creating a new organization by selecting a name, billing information, and plan type

Once you've filled in the fields and selected a plan, the next thing you'll want to do is invite the first members, who will have the role of *owners* for the organization. As described on the right in Figure 7-19, these members will have administrative privileges over all of the organization's repositories and control over other members' actions inside the organization once they accept the invite.

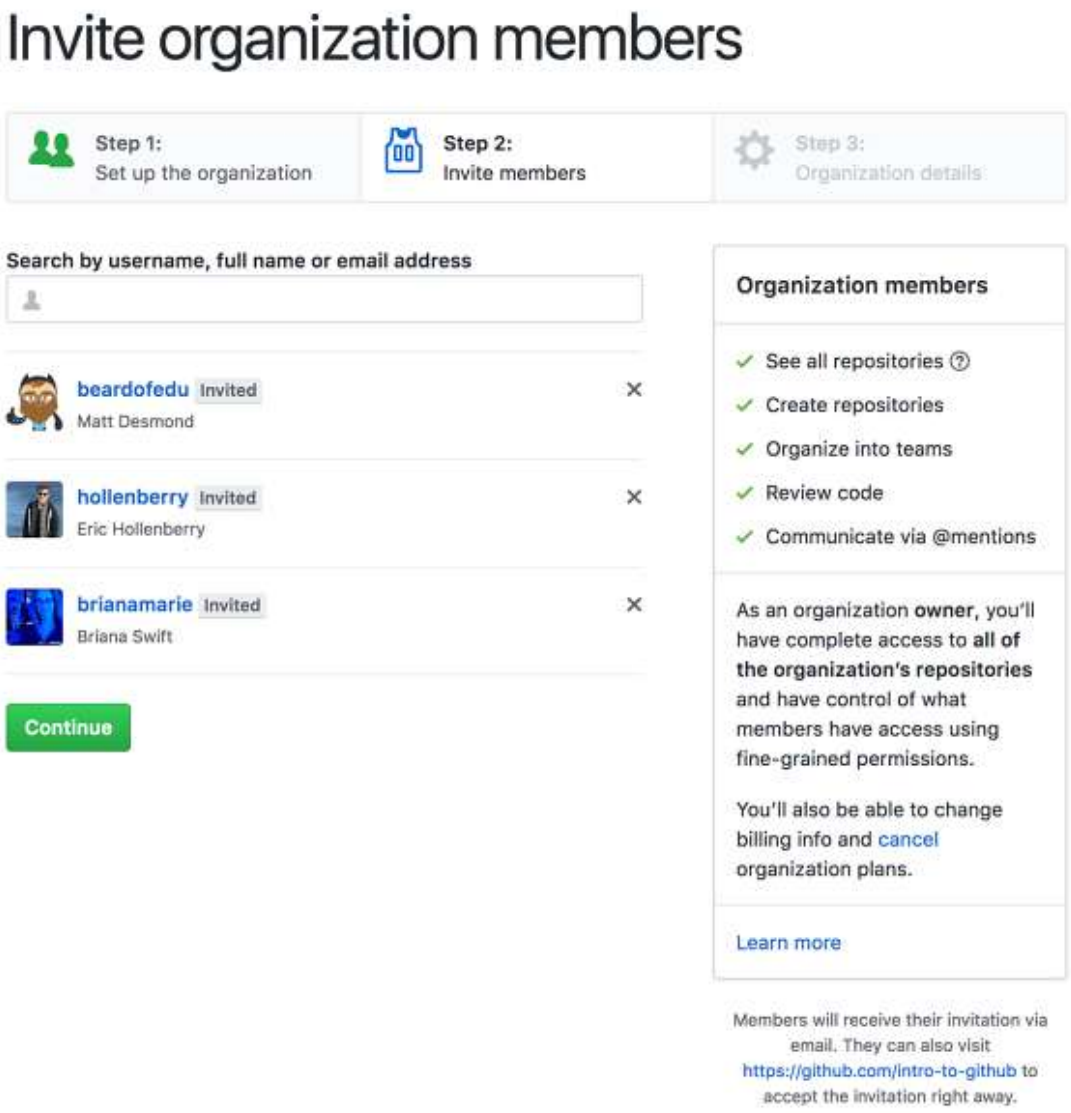


Figure 7-19. Inviting owners for the organization

Lastly, you'll be asked a few questions about the purpose of the organization, how long you expect it to be used for, and how many people may end up inside of it. These are just for GitHub to have a better idea of who the users may be and how to support you. You can skip this step if you'd like!

Once you've created an organization, the next thing you'll want to do is double-check some organizational settings and create some teams.

Configuring Your Organization

As an organization owner, it’s important to know about the possible organization configurations. If you’re going to be managing many teams of developers or if you’re responsible for how your company’s GitHub presence is growing, these settings can make your growth safer, as well as easier, for everyone involved.

Organization Profile

Configuring an organization starts by clicking the Settings tab at the top of the page. By default you’ll go to the Profile page within Settings, as shown in Figure 7-20, which allows you to configure some high-level settings.

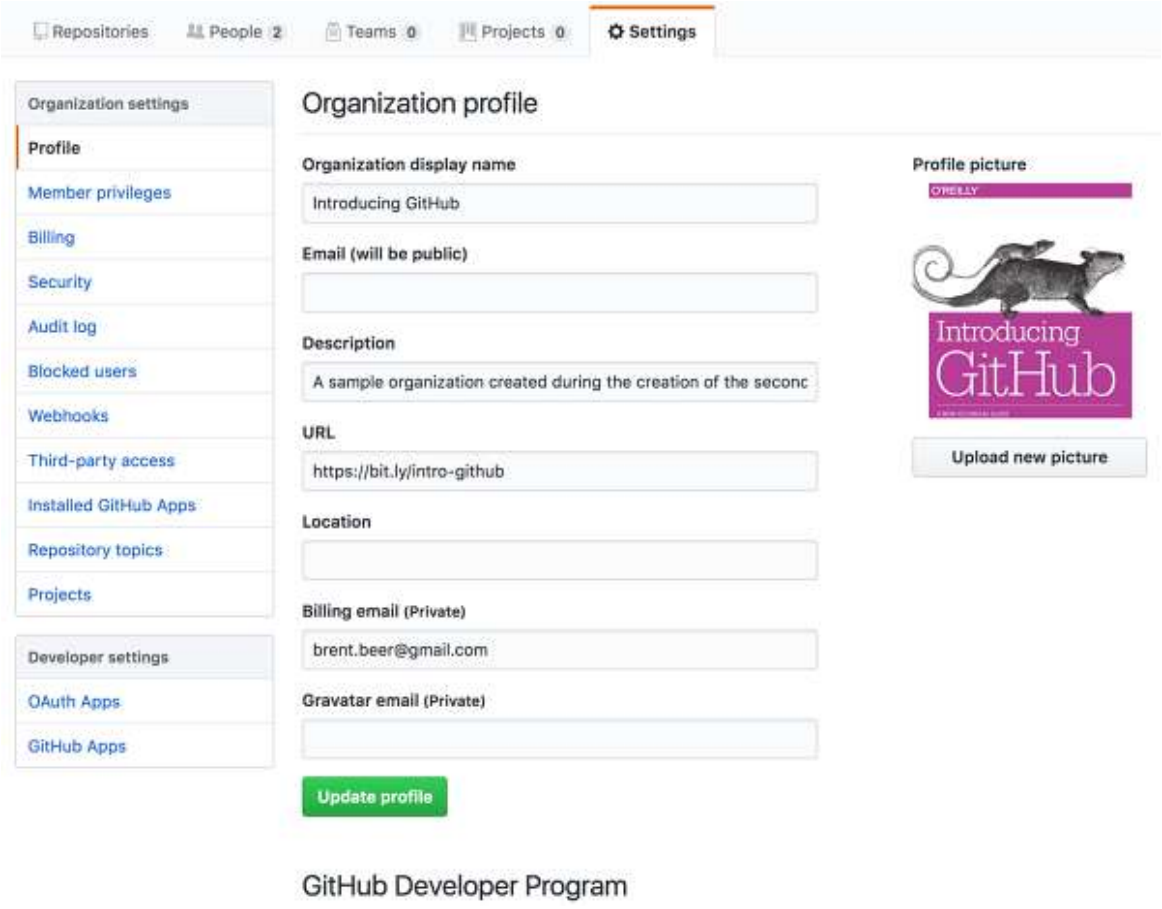


Figure 7-20. Give others an idea of what your organization is about by filling out the profile

The first couple of input boxes are what allow others to know who you are, where they can get more information, and maybe where your company is located. If you created a GitHub Pages site in Chapter 6, the URL box could be a good place to add that URL.

Further down on the Profile page, you'll see information about the GitHub Developer Program. If you have an integration with GitHub that your company built, you'll definitely want to sign up here. Below that is a familiar section from the repository settings earlier in this chapter, the "Danger zone." If you want to delete your organization, or change the name due to an error or company name change, this is where to do it. As the name suggests, proceed with caution when taking these actions. These two sections are presented in Figure 7-21.

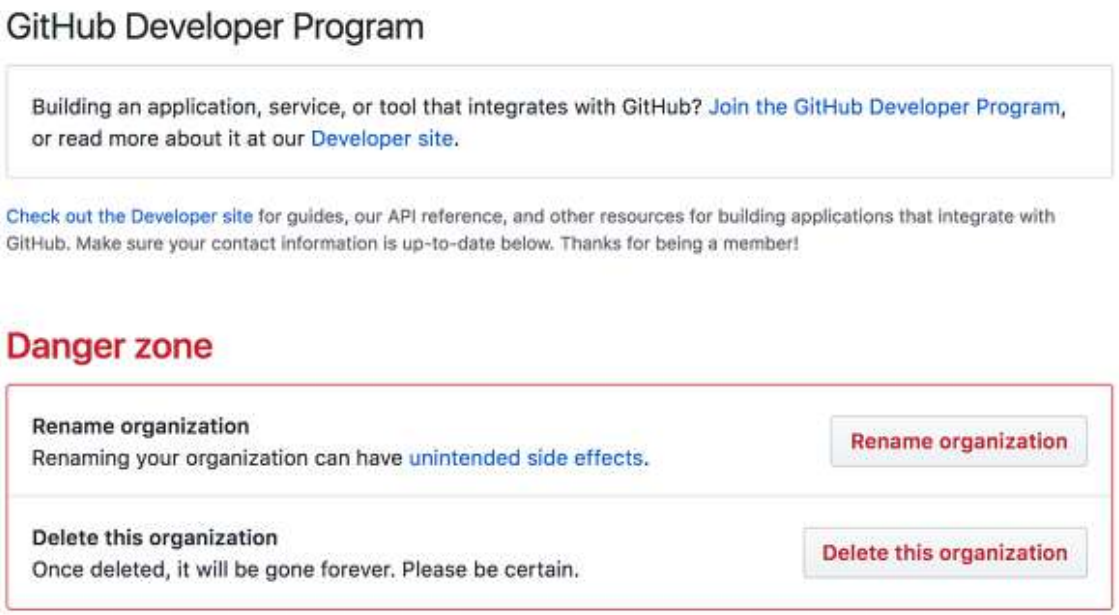


Figure 7-21. GitHub Developer Program and "Danger zone" sections

Member Privileges

The next menu option within Settings is for configuring the default behaviors of members within your organization. As shown in Figure 7-22, repository creation, deletion, and visibility control are important features for distributing responsibility within your organization. Allowing any member to create a repository belonging to the organization (and also delete one, for example if it was created in error) is a great way to encourage new ideas. You may also want the developers that work on a repository to be able to decide for themselves when it's ready to be shared with the world, and when to change the visibility from private to public. Alternatively, these

requests would have to go to an organization owner, which may be more secure but can take longer and seem like a barrier.

At the bottom of the “Member privileges” screen are the default permissions for each individual repository. These options, shown in Figure 7-23, allow you to control how each new member of your organization will interact with the repositories it owns. Because these are the defaults, they can be overwritten by a team’s settings to be more or less restrictive depending on that team and repository’s needs.

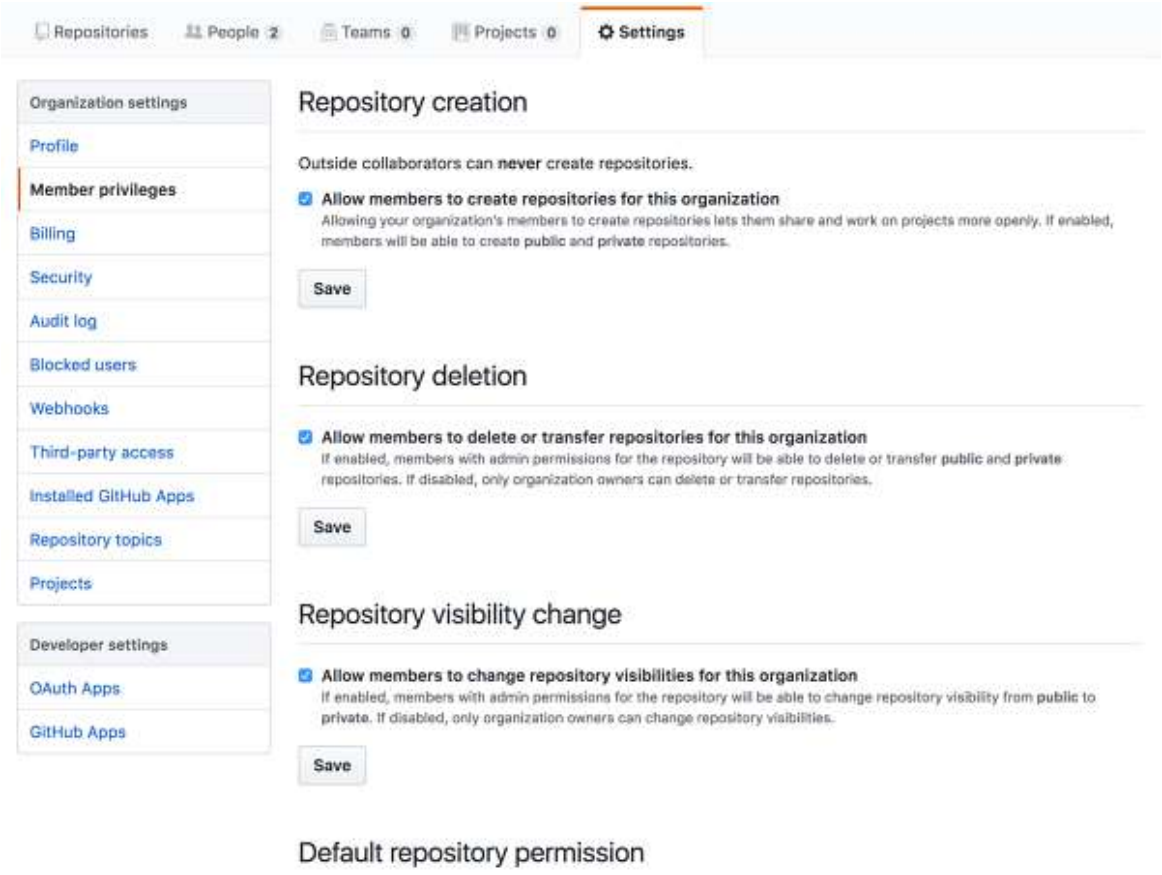


Figure 7-22. Member privileges for repository creation, deletion, and privacy changes

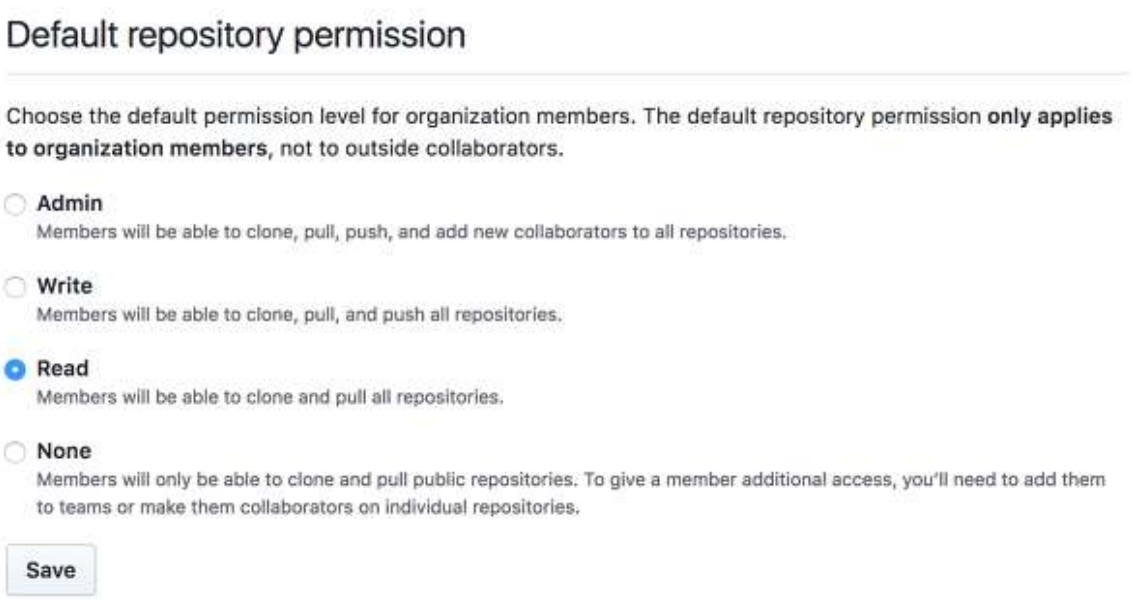


Figure 7-23. Default repository permissions

Billing

Changing or reviewing your billing is done from the Billing screen, the overview section of which you can see in Figure 7-24. As your organization grows, you may need to change your plan from Free to Team or Business, or you may need additional purchasing add-ons like Git LFS. All of these options can be changed and edited within the “Billing overview” section.

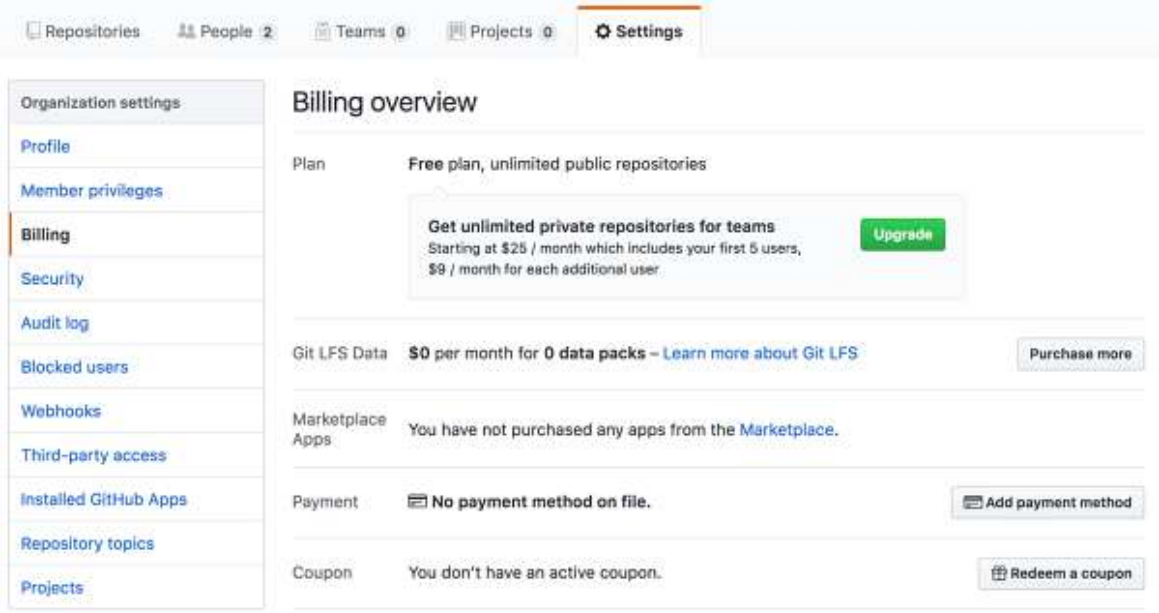


Figure 7-24. Billing overview

As you may have to audit your billing, this work could be assigned to one or more users within the organization as billing managers. This specific role is only allowed to change any of the billing details, as well as to view past transactions in the "Payment history" section. These two sections can be seen in Figure 7-25.

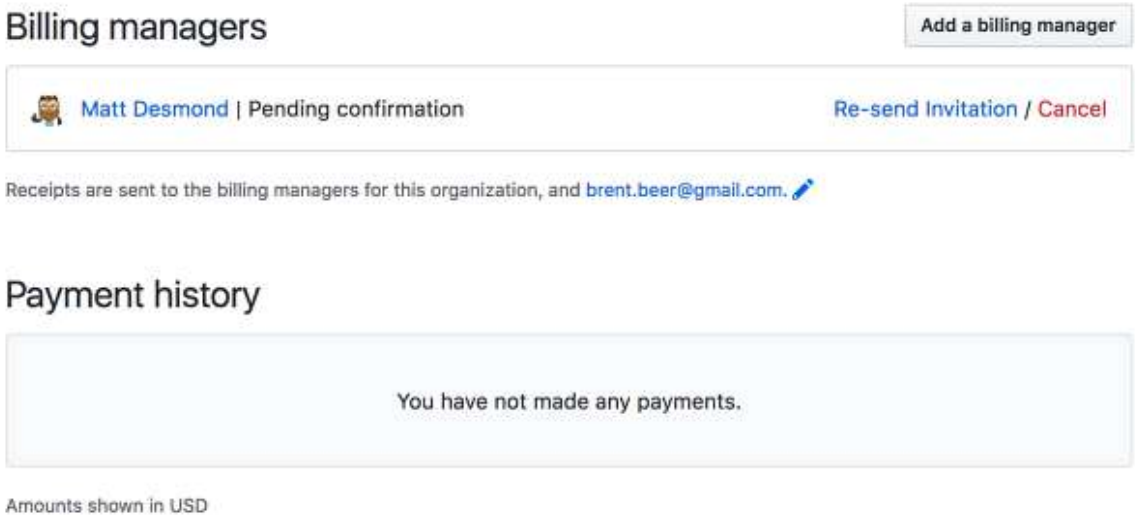


Figure 7-25. Billing managers and payment history

Security

Setting different security requirements, like two-factor authentication and, if you are on the Business plan on GitHub.com, enforcing SAML for members, is important for a healthy organization. These settings can be configured on the Settings → Security screen, shown in Figure 7-26.

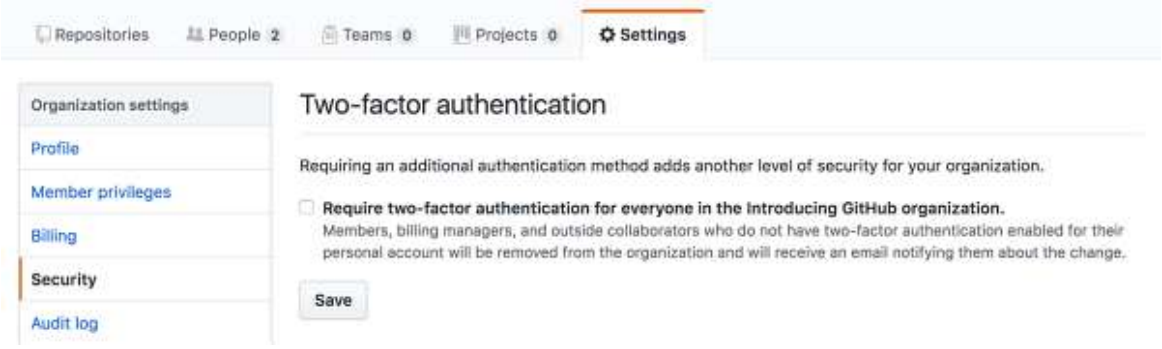


Figure 7-26. Security requirements

Audit Log

It can be important at times to track down when certain actions took place and who took them. Imagine a user reports his laptop as stolen while he's away on a trip, and you don't know if you need to temporarily remove this user from your organization until he's reset his passwords. If you looked into the audit logs and saw access to the account coming from an area of the world that it shouldn't, that would be a clear sign to remove that member from your organization and double-check what actions had been taken so you could do some damage control. As shown in Figure 7-27, the "Audit log" section of the Settings tab allows you to not only filter by certain common categories, but also any search criteria that may be helpful to look for, like a username or repository name.

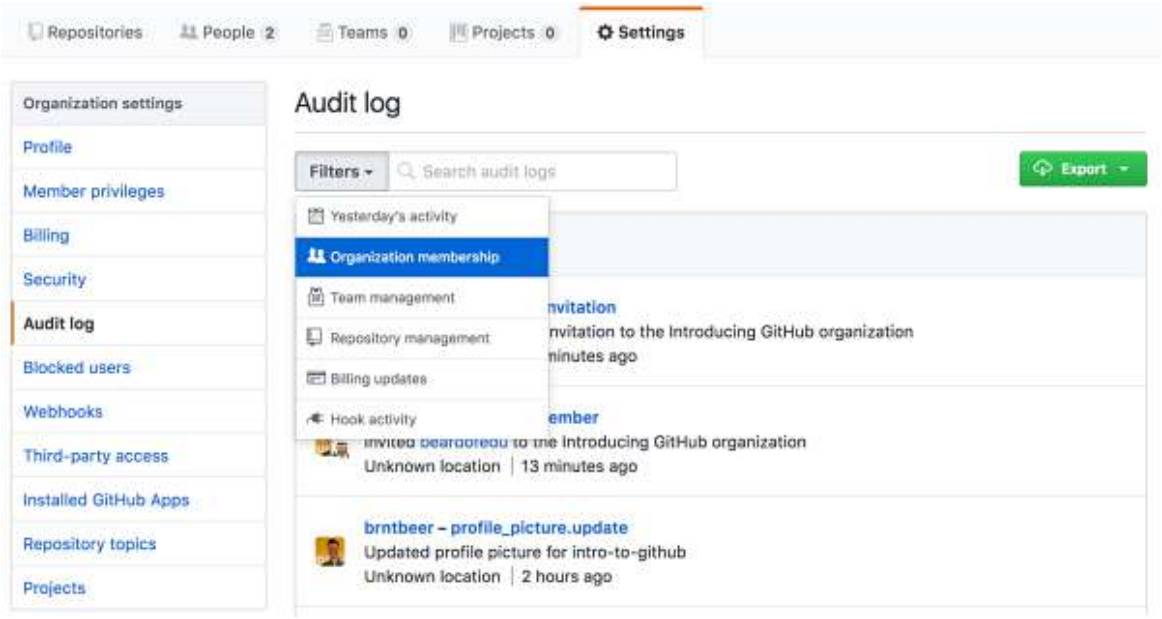


Figure 7-27. The Settings → "Audit log" screen

Blocked Users

Blocking malicious users at the organization level is a good way to ensure those users will no longer be able to interact with your organization's repositories, or even watch and star them. This should not necessarily be a feature used to counteract a heated discussion, as it is a severe punishment. You should instead use this option if you wish to block someone from interacting with you and your organization completely, for your own or your members' safety. On the "Blocked users" screen, you can review who is already blocked and an overview of the actions they are restricted from (see Figure 7-28).

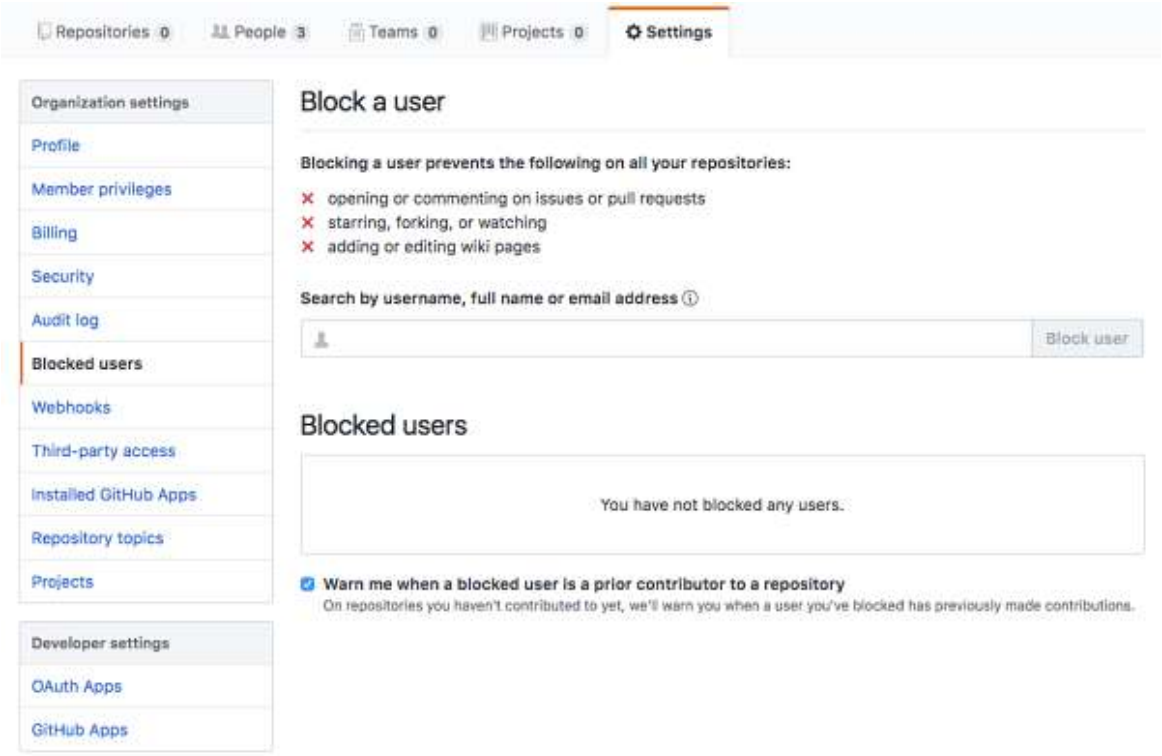


Figure 7-28. The Settings→"Blocked users" screen

Webhooks

Organization webhooks are similar to repository webhooks in that they allow certain actions to be taken given some event, but at an organizational level. Using the previous section as an example, you may want to notify an external server every time a user has been blocked by the organization in order to take additional action. You might also want notifications to be sent when a repository changes from public to private, or when membership within the organization has changed. There are a lot of possibilities to expand your team management with webhooks at the organizational level; to explore these, select the "Let me select individual events" option as shown in Figure 7-29.

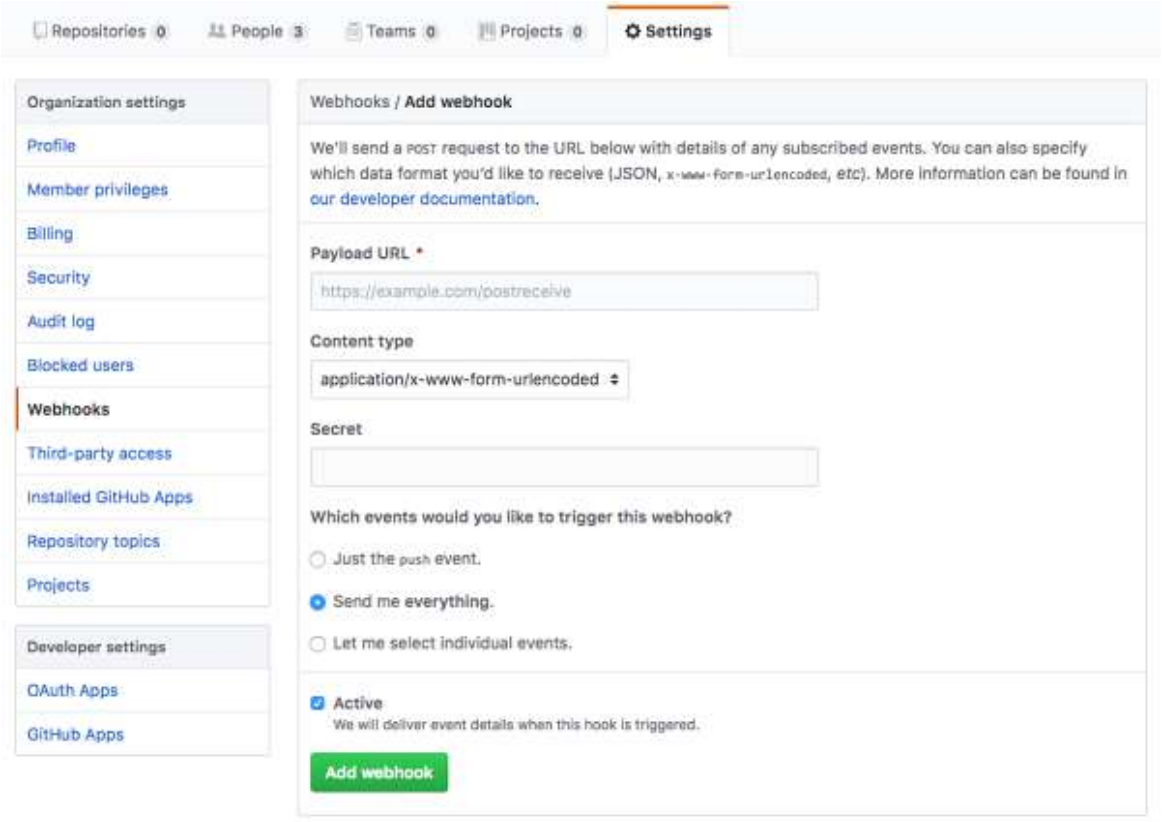


Figure 7-29. Adding organization webhooks

Third-Party Access and Installed GitHub Apps

When members of your organization integrate and interact with third-party software on GitHub, they may have to sign in to those applications and authorize them to access certain data from GitHub. Sometimes, these applications need organization-level information that, as an owner of your organization, you want to have control over. Controlling these requests and viewing any pending ones is done via the Settings→"Third-party access" screen, seen in Figure 7-30.

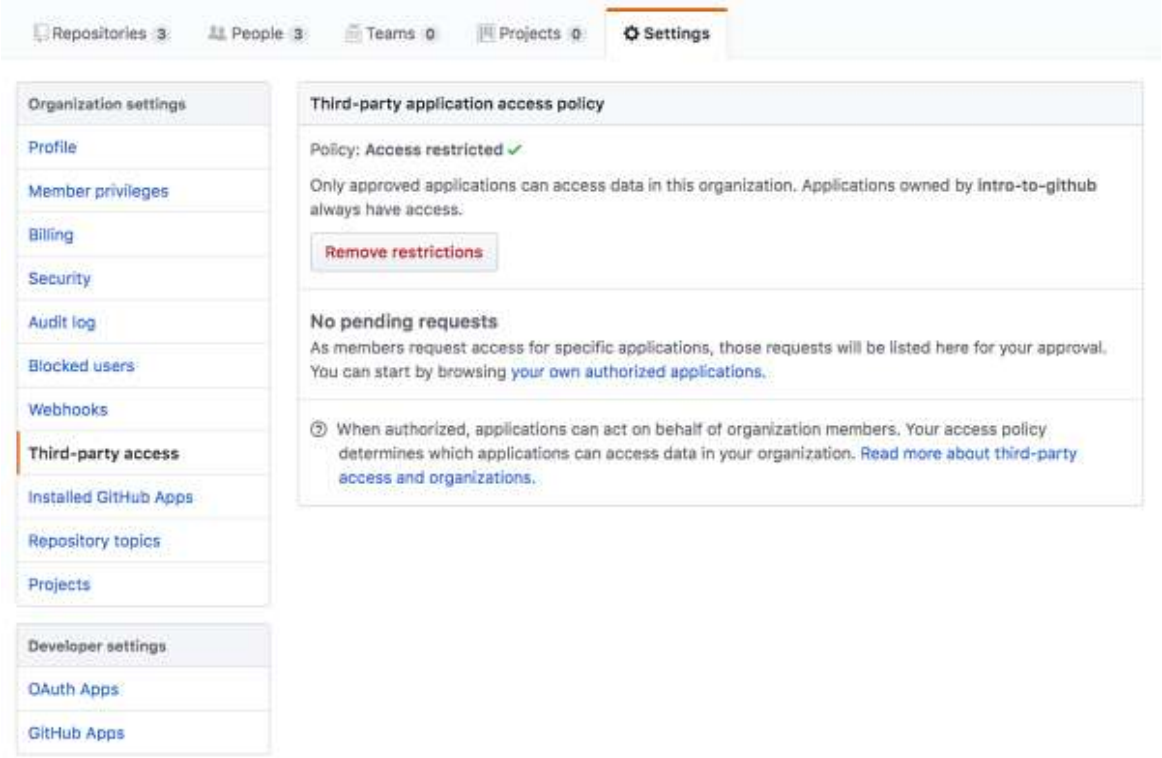


Figure 7-30. Viewing third-party access restrictions and pending requests

As an organization owner, if you've purchased any applications from the Marketplace (as discussed in "Integrating with Other Systems"), those items will appear on the Installed GitHub Apps screen.

Repository Topics

Repository topics, which help people discover your repositories and know what software topics a repository may cover, can be managed all at once from the Settings→"Repository topics" screen, shown in Figure 7-31. Adding topics to your repositories can be important for attracting new developers and making it easier for your current developers to find repositories within your organization.

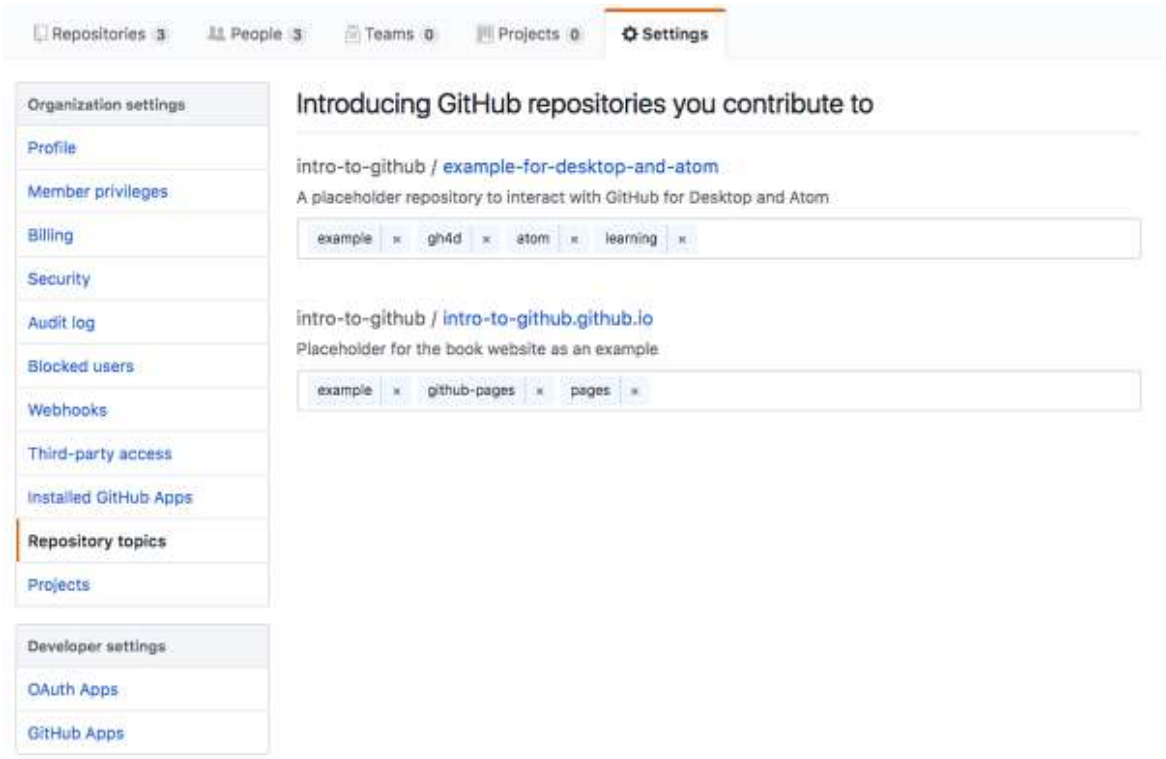


Figure 7-31. Organization-owned repositories and their topics

Projects

We talked about GitHub Projects at the repository level back in Chapter 5, but sometimes the projects you're working on span across multiple repositories. Tracking the work across those repositories can be made more efficient by having an organization-wide project and collecting issues and pull requests from multiple locations. To allow this, you need to enable the settings on the Projects screen, shown in Figure 7-32. If you want to disable GitHub Projects for all of your repositories, you can also do so from this screen.

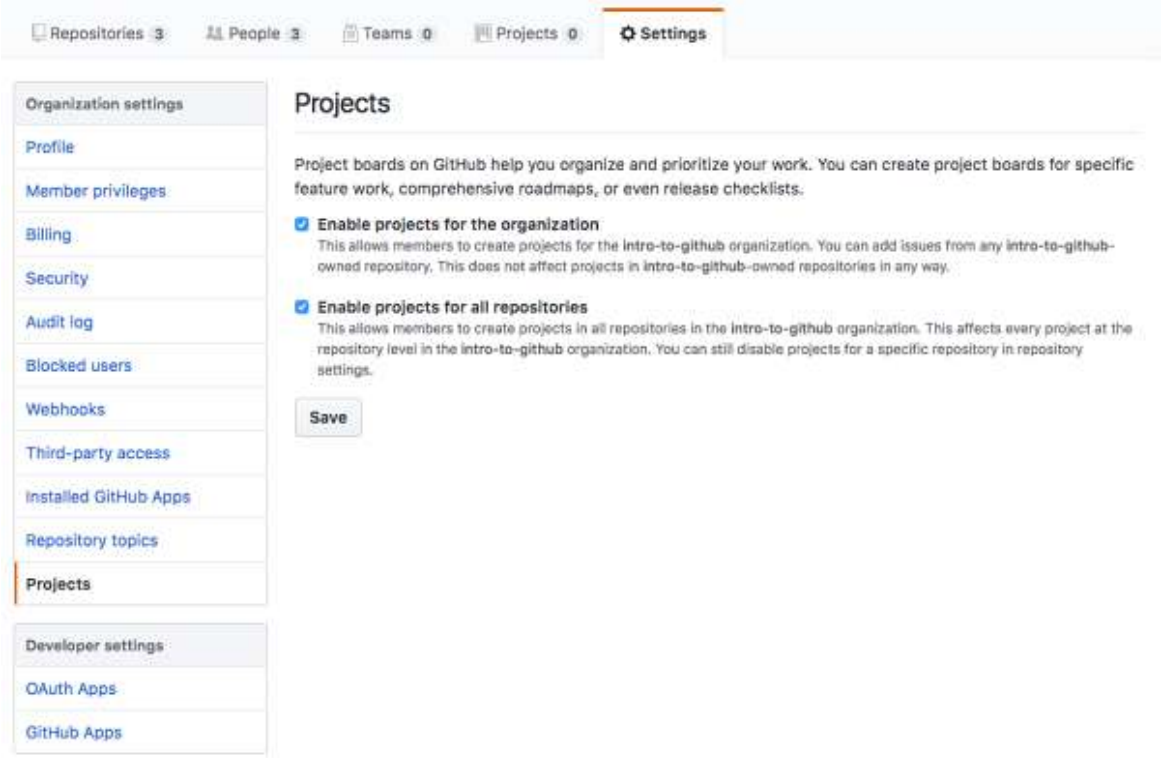


Figure 7-32. Enabling organization-wide projects

Managing Members and Teams

If you create a repository under your user account, you can just add collaborators directly to a project. However, if you create a repository under an organization and want to allow other people to access it, either they'll have to be members of your organization (see "Member Privileges" for details on setting permissions) or you'll have to create teams.

By default, when you create an organization, GitHub will assign you to a role called "owners." Owners have complete administrative access to the organization. These people will help with managing the organization's settings, and by default will have administrative access to all of the repositories owned by the organization without needing to be given further permissions within a team or by any other means. "Members," on the other hand, is a permission title for everyone else who belongs to the organization. This shouldn't be confused with the role of an outside collaborator, who is someone who was added to a single repository owned by the organization. While members typically have default permission to

read and view all repositories owned by the organization (public and private), it is usually a best practice to create a new team to handle organizational structure and control over who can write to the organization's repositories. If you'd like to only invite members, you can do so by clicking the "Invite member" button within the People tab on the organization page, as shown in Figure 7-33.

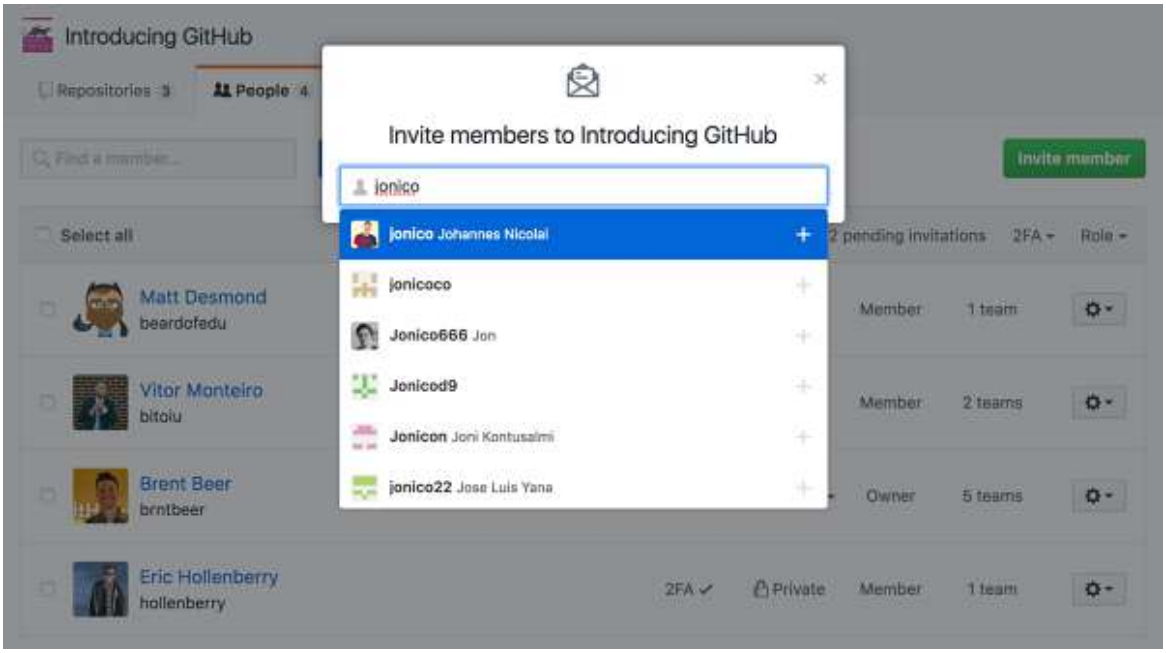


Figure 7-33. Inviting members to your organization

After you've invited the new member, you'll see a screen similar to Figure 7-34 where you can then decide if this new member should be an owner or add them to some teams if you have teams already. This makes it quick and easy to grow your organization once you have teams set up.

If you don't have teams in your organization yet, you'll want to create some to more easily handle permissions for individual repositories when adding members. When creating the team names, you may want to start with traditional teams like "engineering," "design," "product-design," "frontend-engineers," or "platform-engineering." Once you have these teams set up, you may want to create additional ad hoc teams like "senior-developers" for the more senior developers, "infrastructure" for the people knowledgeable about your operations or infrastructure, "github-experts" for people who

can assist with Git and GitHub questions, or even “code-review” to make it easy for that team to review code in the repository. If I’m working within a larger company, I might also create teams for business units or functions like marketing and legal. Besides handling permissions, these teams make it easy to bring relevant people into appropriate discussions by @mentioning the team. The @mention format is slightly different for teams than it is for an individual: the format it follows is @*<organization>*/*<team_name>*.

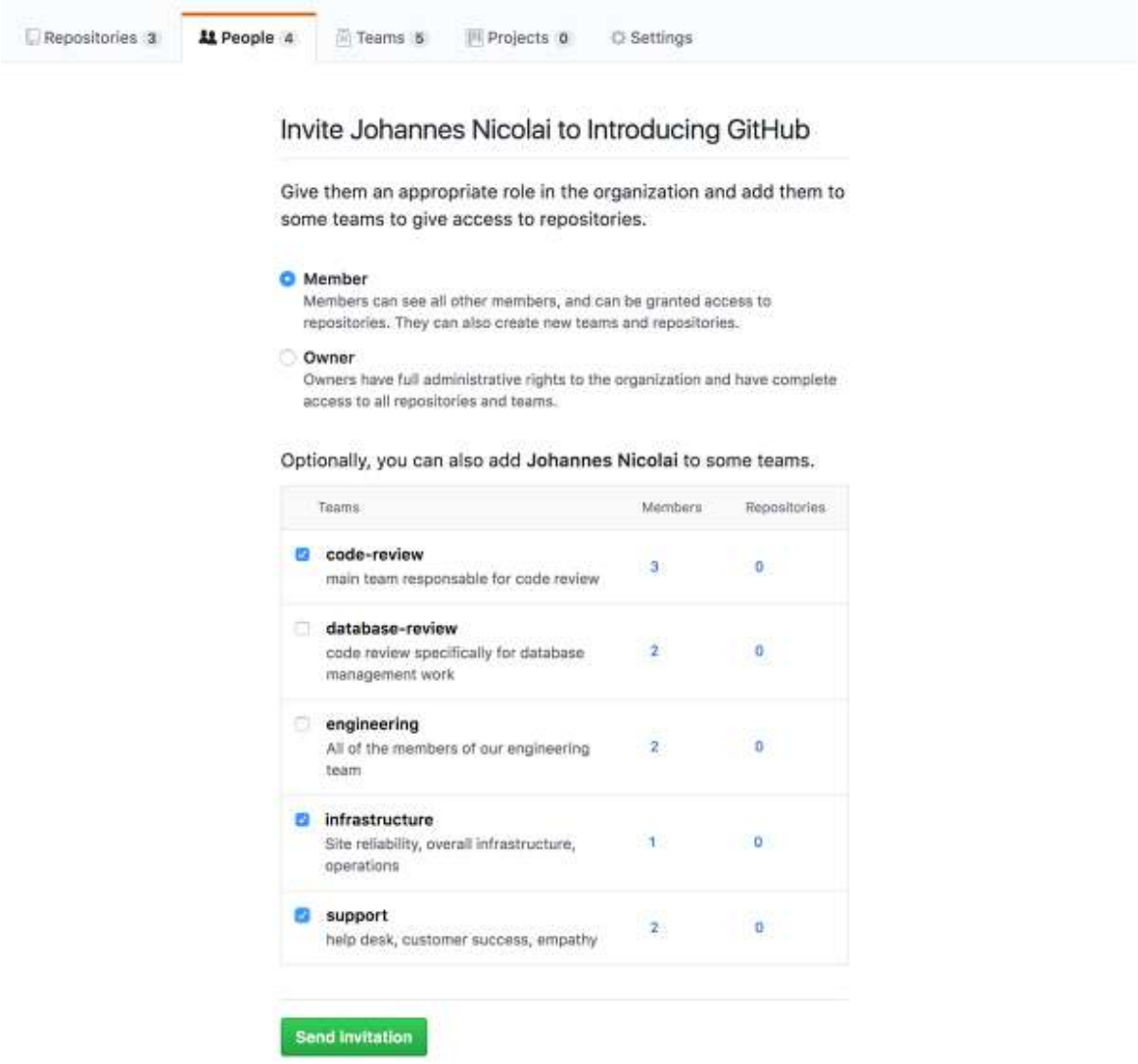


Figure 7-34. Adding a new member to some existing teams

To create a team, go to the organization’s home page and click the Teams tab at the top of the page, highlighted in Figure 7-35.

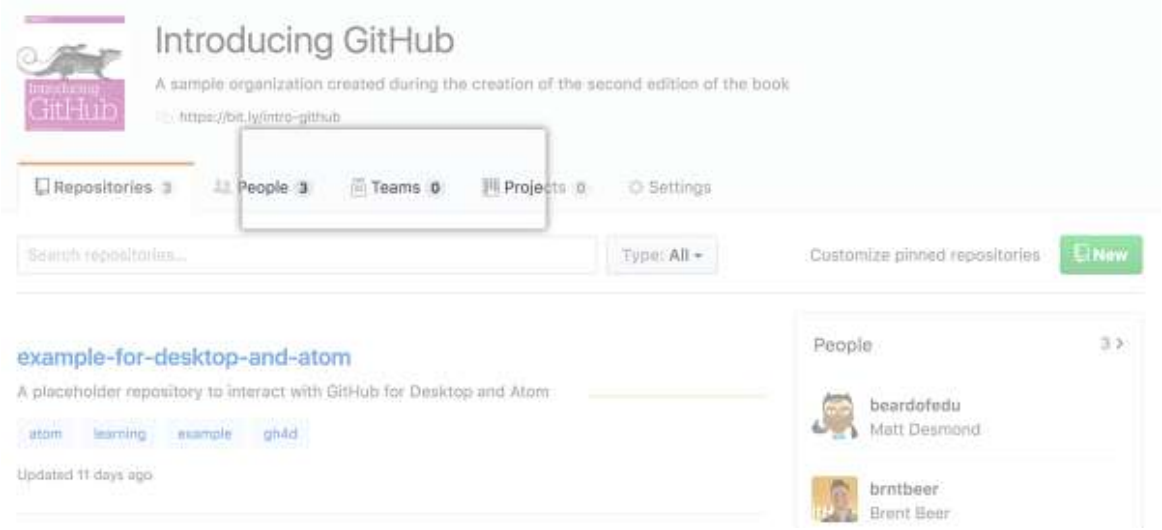


Figure 7-35. Locating the Teams tab on the organization's home page

When you visit the Teams page for the first time, you should see a screen similar to Figure 7-36 to help you get started on creating teams, because you haven't created one yet. Once you have created some teams it will show a list of all of the teams within your organization, each with a row of small profile pictures showing some or all members of the team, depending on its size, and any nested child teams. An example of a Teams page with teams is shown in Figure 7-37.

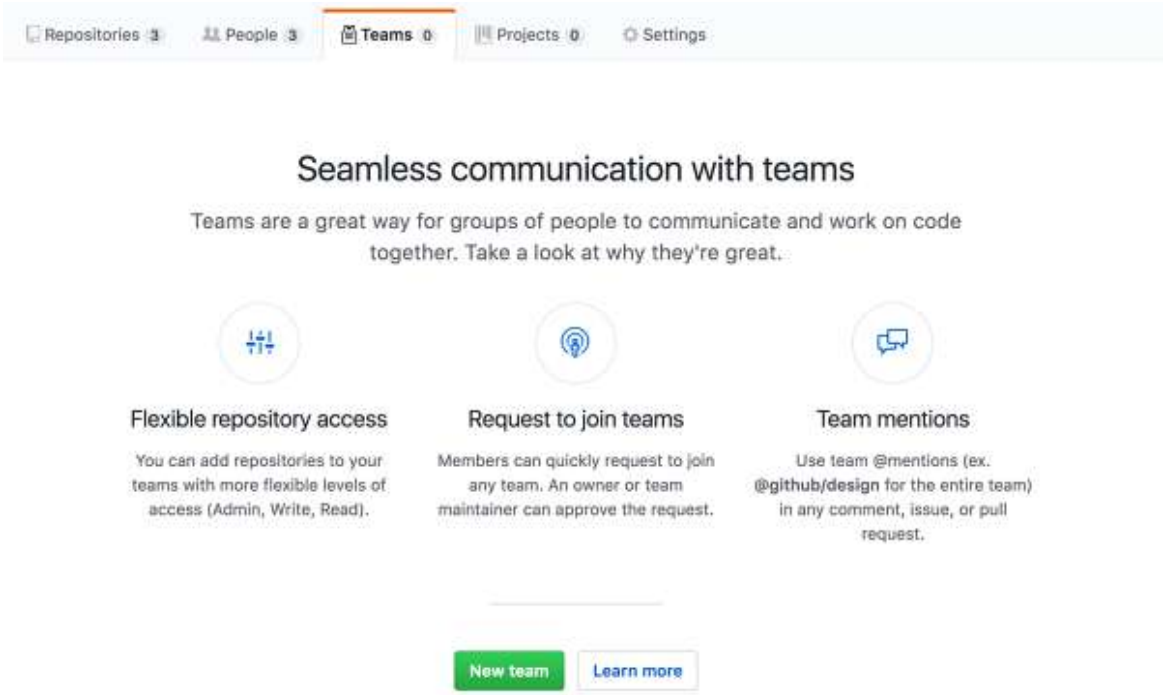


Figure 7-36. Bare Teams page

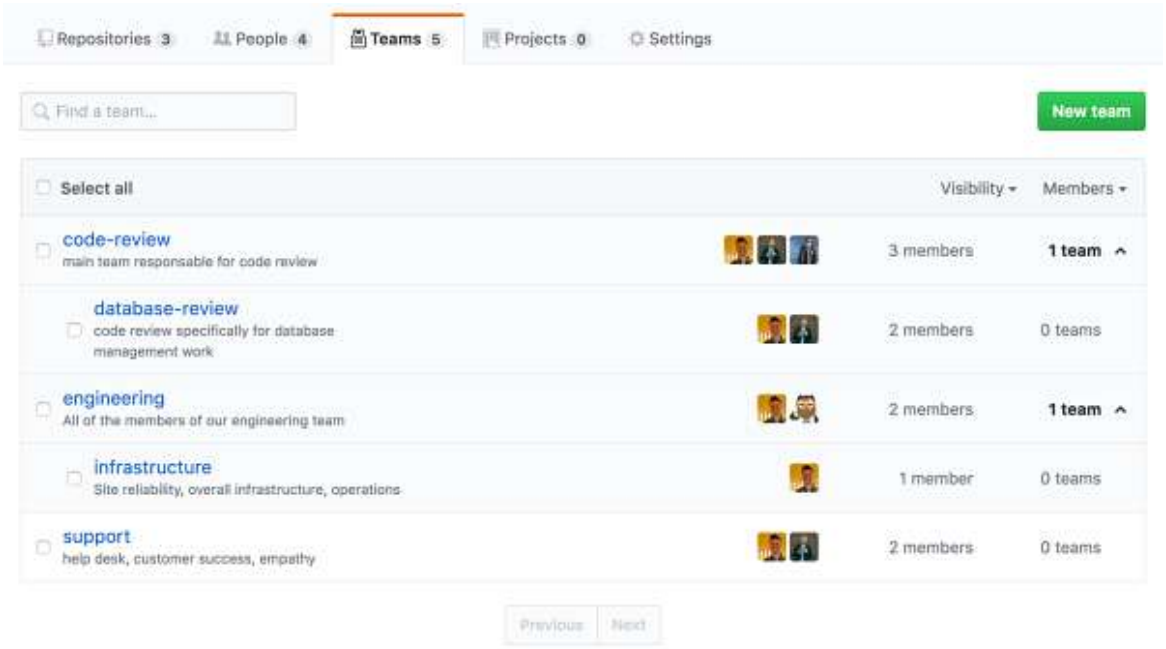


Figure 7-37. Viewing the existing teams within the organization

To create a new team, click the green “New team” button at the bottom of the content area if this is your first team, or on the right side of the page

above the team listing if this is for an additional team. Afterwards, you should see a screen similar to Figure 7-38.

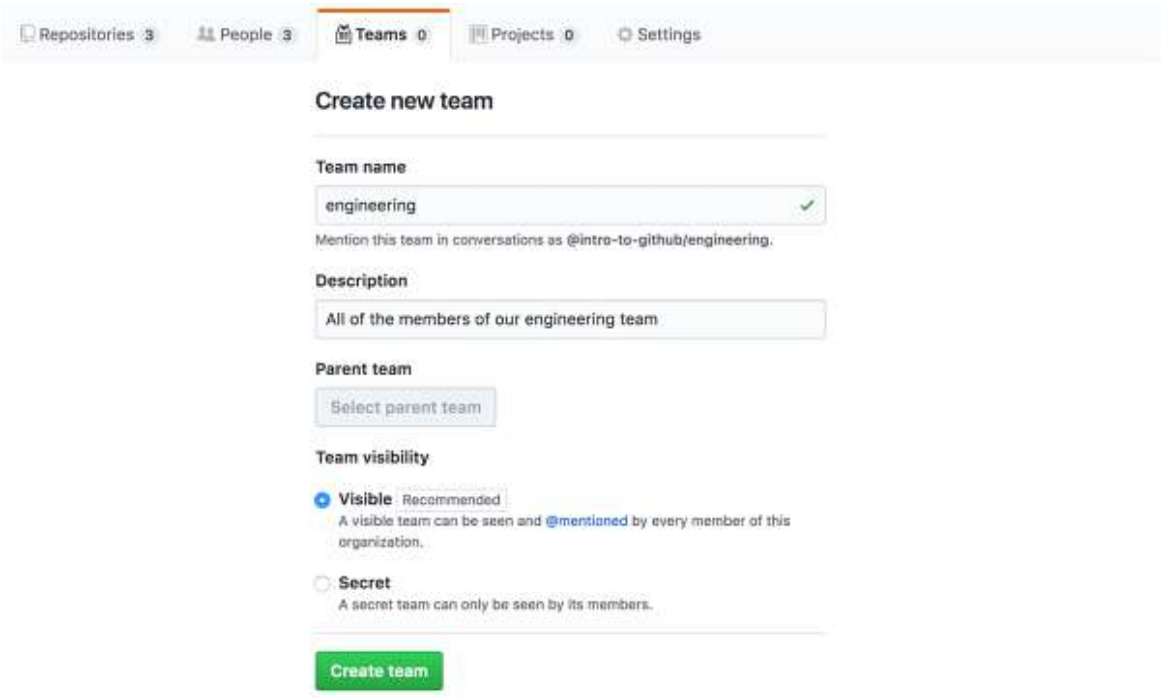


Figure 7-38. Adding a new team to an organization

Give your team a name. If you're just working with a couple of developers on a single project, it might be something as simple as "developers." If you are part of a larger organization, the name could be "engineering" for your entire engineering department, or it could be the name of the project team: "mobile-devs," "api," "designers," etc.

You can add an optional description if the intent of the team might not be obvious to members of your organization. Additionally, this description can make it easier to @mention the team if someone starts typing some words from the description instead of the name itself. For a smaller, focused team belonging to a larger hierarchical component of your company, like an "api" team that's part of the larger "engineering" team, you may want to select a parent team to nest this child team under. Child teams inherit the parent's access permissions, simplifying permissions management for large groups: you can be very broad at the high level and give more granular access to more restrictive repositories to lower child teams. Members of child teams

also receive notifications when the parent team is @mentioned, simplifying communication with multiple groups of people. When you've finished, click the "Create team" button.

Once you've created a team, the next step is to add members to the team. To do this, click the "Add a member" button on the team page. As shown in Figure 7-39, just start to enter the GitHub username or the name the user has displayed on their profile for each person you want to add to the team, and the name should autocomplete as you're typing. Usernames will be easier for GitHub to autocomplete, as they are fully unique. If the people you want to add to this team are not members of the organization yet, this will also invite them to the organization. Once they accept the invite, they will be members of the organization as well as being on the team.

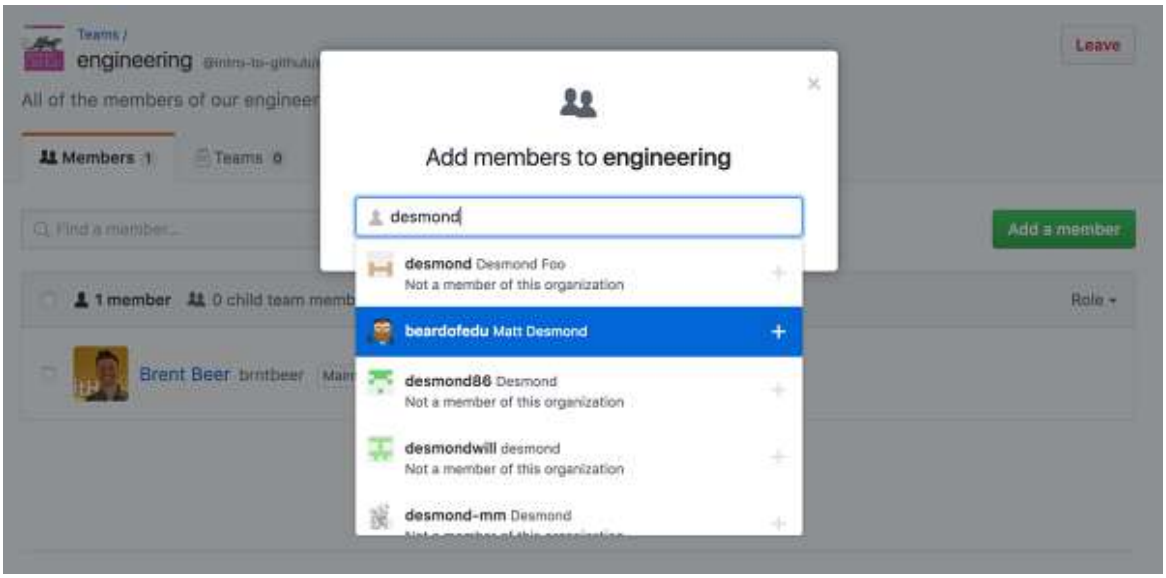


Figure 7-39. Adding a user to a team

Next, you will want to give the team some repositories to work with so you can have more granular access for your members. To add repositories to the team, click the Repositories tab on the team page. Once there, press the "Add repository" button to see a screen similar to Figure 7-40. Again, you can start typing the name of the repository within the organization to add to the team and it will autocomplete.

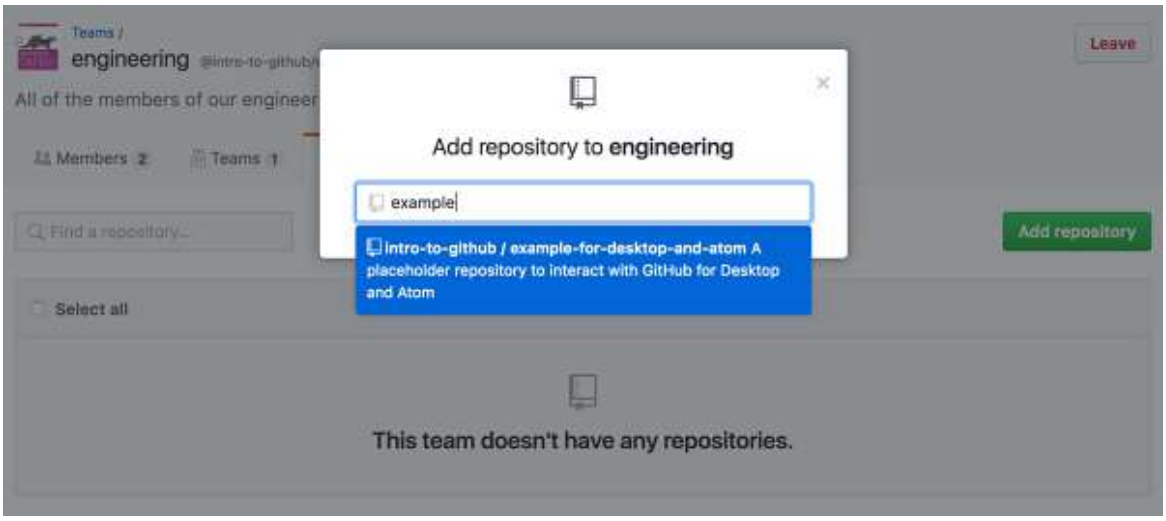


Figure 7-40. Adding repositories to a team

Once a team has a repository, you may want to change the permissions for how that team interacts with it. You can see some of the different permission options in Figure 7-41.

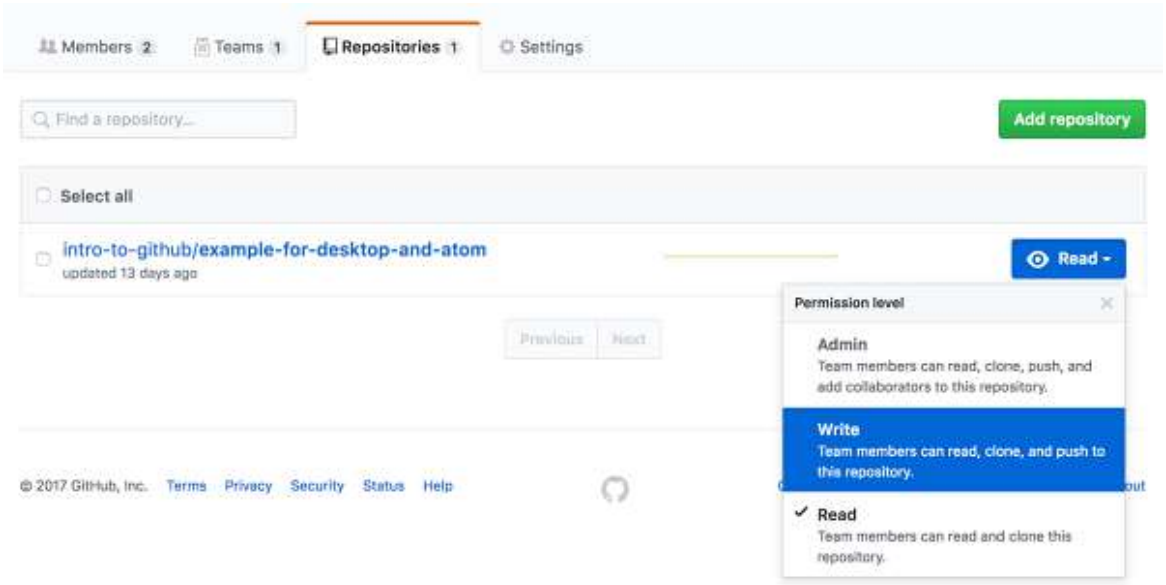


Figure 7-41. Editing team permissions for a repository

Changing these permissions is what will enable you to have more fine-grained access for the members within your organization. If this team just needs to consume and read code, review code in a pull request, or leave comments and feedback, perhaps “read” permission is sufficient. If they are contributing to the repository by writing and pushing code in addition to

reading from it, they'll need "write" access. Finally, if you want them to not only be able to read and write the code but also configure the integrations, invite outside collaborators, or control some of the code review for the repository, perhaps "admin" is the correct permission level.

If you ever need to edit a team's name, description, visibility, or parent team, or delete it altogether, that's done from the Settings tab. Click the Settings tab at the top of the team page and you'll see a screen similar to Figure 7-42.

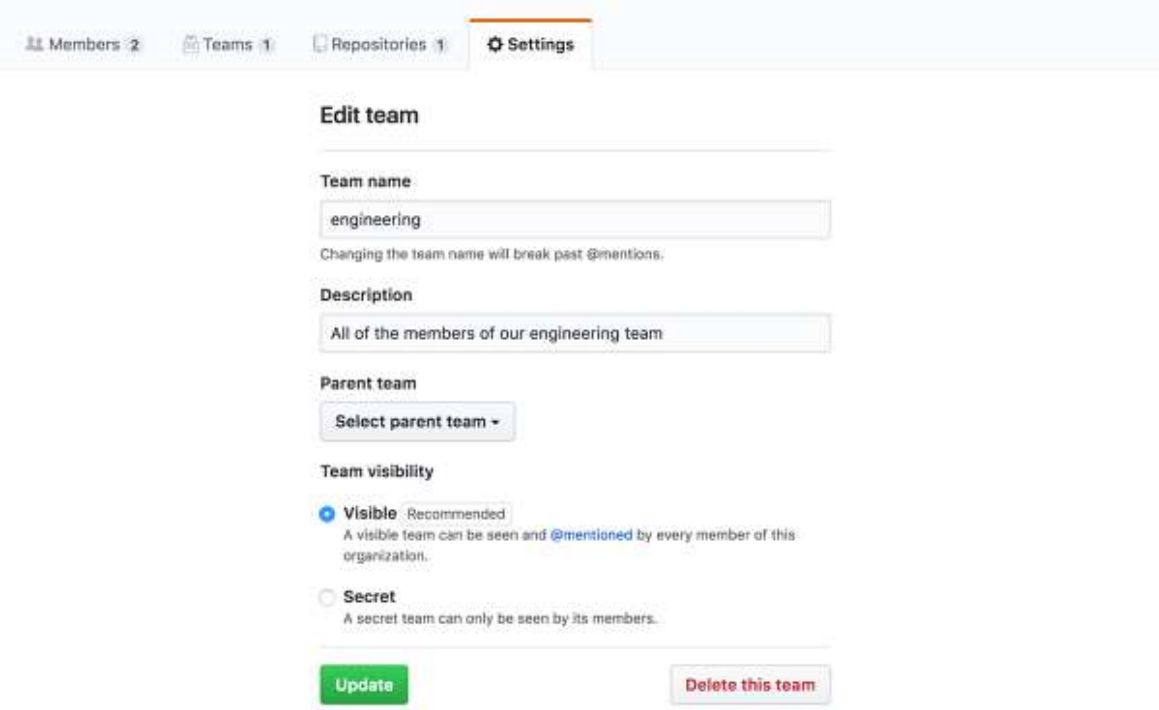


Figure 7-42. The team Settings page

If you want to delete the team, click the "Delete this team" button at the bottom of the page. You'll be asked whether you're sure. Just click OK, and the team will be deleted.

As well as deleting teams, you may want to remove a member from a team, or from the organization. Removing someone from a single team is best done from the team page itself. Visit the individual team page by clicking the team name in the organization's list of teams. On the Members tab of the team page, select the checkboxes next to the members you wish to

remove and click the drop-down at the top of the member list, as shown in Figure 7-43. This batch user management option will allow you to change the selected members' role to team maintainers or remove them from the team completely. Once you select "Remove from team," you'll be given a warning about those members losing their forks, as well as a summary of all the users you're removing. Press "Remove members" to confirm.

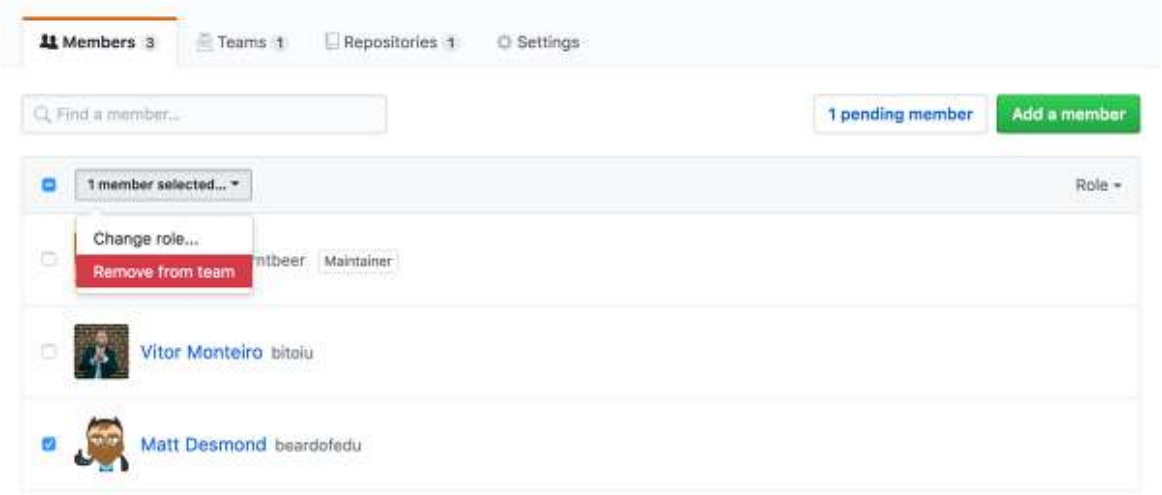


Figure 7-43. Selecting a member to remove from the team

Removing a member from the organization completely can be done by selecting the individual user on the People tab, or in a similar bulk-edit fashion like when removing members from a team. You can see an example of this in Figure 7-44. Additionally, if you want to keep the member as an outside collaborator who can work on only the repositories she has explicit permissions to access, you can do that here as well.

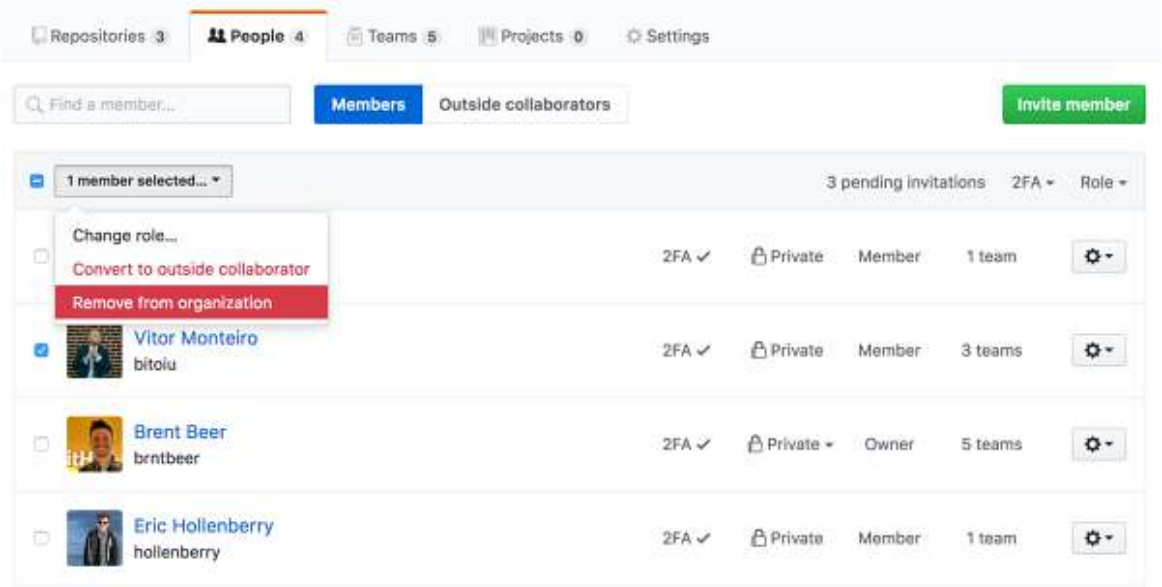


Figure 7-44. Selecting a member to remove from the organization

Congratulations! If you've gotten this far in the book, you should be ready to do almost anything with a GitHub repo. You should be able to view the state of a project, edit the files in a project, collaborate with your team, and create and configure a new repository. In the next chapter, we're going to look at how you can use GitHub Desktop and the Atom text editor to download a copy of a GitHub repository and make some simple changes to it on your laptop, to take your software development experience a step further.

Chapter 8. Downloading and Working Offline

You may never need to clone (download) a copy of a repository at all. As we've seen in this book, you can use the GitHub web interface to view the state of a project, edit content, collaborate with your team, and set up and configure a repository. However, sometimes it's necessary to clone a repository. In this chapter we'll look at why you might want to clone a repo and how you would do so using GitHub Desktop. If you're running Linux, though there are web interfaces out there, you'll probably be better off just installing Git directly and learning the command-line interface for working with Git repositories—but that's outside of the scope of this book.

Why Clone a Repository?

There are a number of reasons why you might decide to clone a repository. Some of the most common ones include the following:

Creating a backup
: When you clone a repository, it creates a full copy of the project—including all branches, tags, and history—on your computer. Sometimes it's worth cloning a repository and pulling the changes down regularly just to know that you have a full copy of the project safely on your machine. This does not grab any GitHub-specific items like issues or pull requests, however.

Editing in an IDE or text editor
: The web-based interface isn't as powerful as editing in an IDE (integrated development environment) or your favorite text editor, so if you're editing content all day, you're going to want to do that locally on your machine.

Editing offline
: You can't edit directly on GitHub unless you have an Internet connection, so if you want to be able to keep working on your project whether or not you're connected, you're going to want to clone your repo and work on it locally.

Editing multiple files

One of the current key limitations when editing on GitHub directly is that there is no way to group a set of related changes and make them as a single commit.

Running the code
: Sometimes you'll want to be able to run the code locally to test exactly how it works.

Running the tests
: If you have automated tests for a project, you may want to be able to run those tests locally to confirm that recent changes haven't broken the software.

If you need to do any of the preceding things, you'll need to either install the Git version control system directly onto your computer or install a GUI (graphical user interface) that makes it easier for you to use Git to perform common operations.

A number of different applications are available that provide GUIs for working with your Git repositories. In this chapter, we're going to cover the GUI provided by GitHub: GitHub Desktop.

GitHub Desktop

GitHub's native client is completely open source—if you or your developers ever want to be involved with the community, you should check out the repository for this application. On the repository page, you'll find additional install instructions as well as ways to contribute and report bugs. However, the traditional way to get a copy of GitHub Desktop is to go to *https://desktop.github.com*. You should see a screen similar to Figure 8-1.

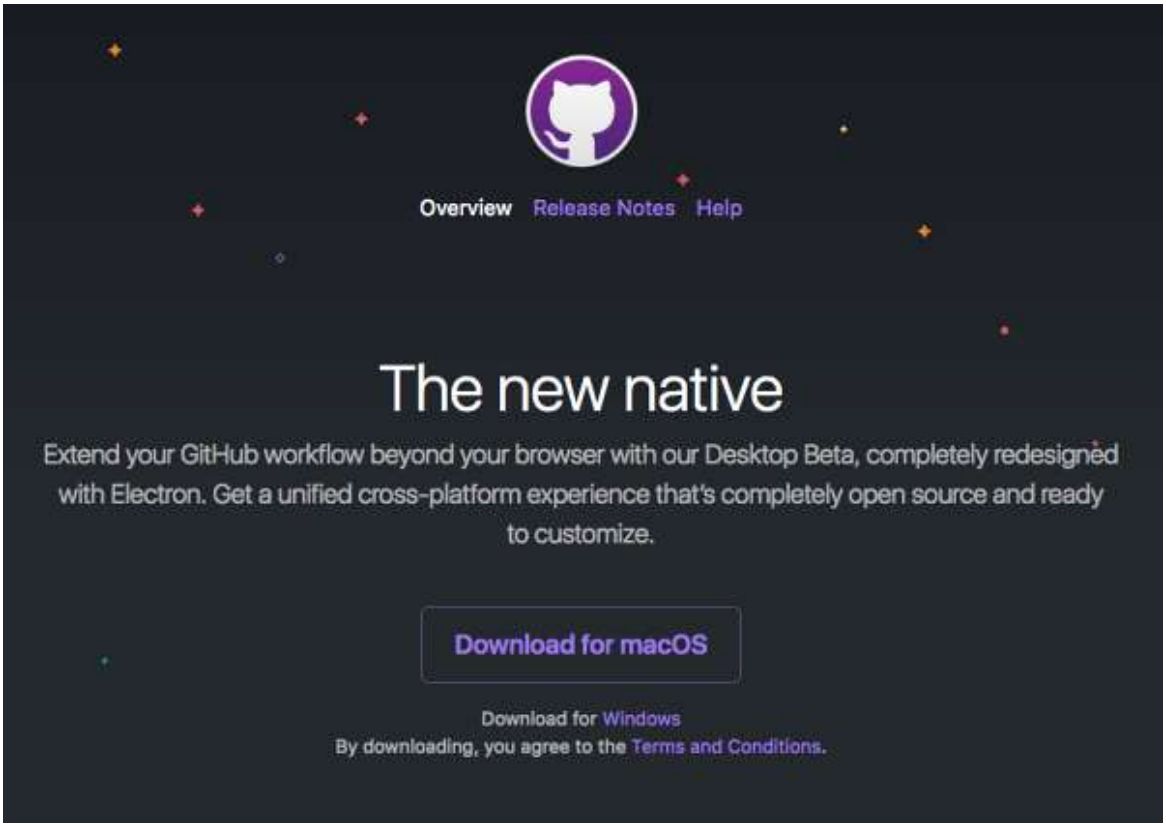


Figure 8-1. The GitHub Desktop web page

Click the “Download for macOS” or “Download for Windows” link to get started. This will download either a ZIP file (macOS version) or a setup installer (Windows) to the location on your computer that your browser typically downloads files to—usually your *Downloads* folder. If you have the ZIP file, just double-click it and it should expand to a file called *GitHub Desktop* for you to work with. On Windows you’ll just have a file called *GitHubDesktopSetup* that will take care of everything for you by bringing you through an install wizard. An example of what these files should look like in either operating system is shown in Figure 8-2.

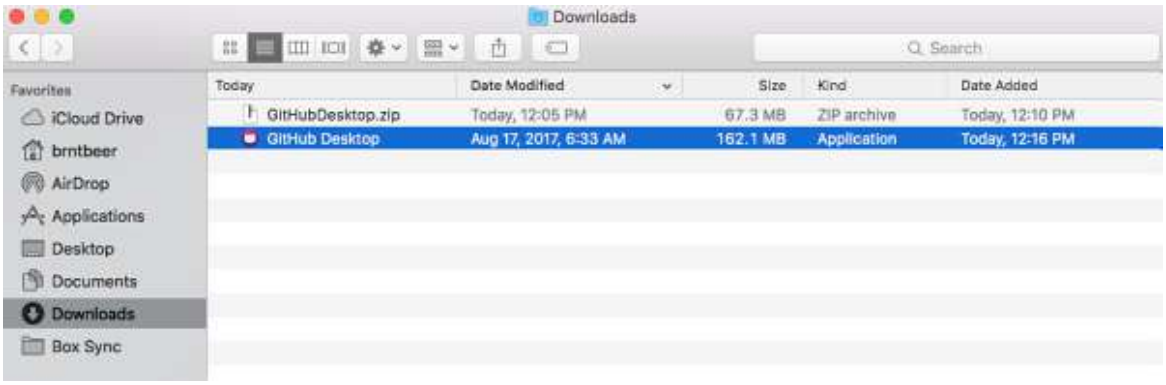


Figure 8-2. The GitHub Desktop ZIP file and application file

Drag the *GitHub Desktop* file into your *Applications* folder for long-term keeping if you're installing on macOS. Regardless of your operating system, double-clicking either the *GitHub Desktop* file in your *Applications* folder or the *GitHubDesktopSetup* in your *Downloads* folder will start the GitHub Desktop installation and setup process. You should see a screen similar to Figure 8-3 welcoming you to GitHub Desktop.

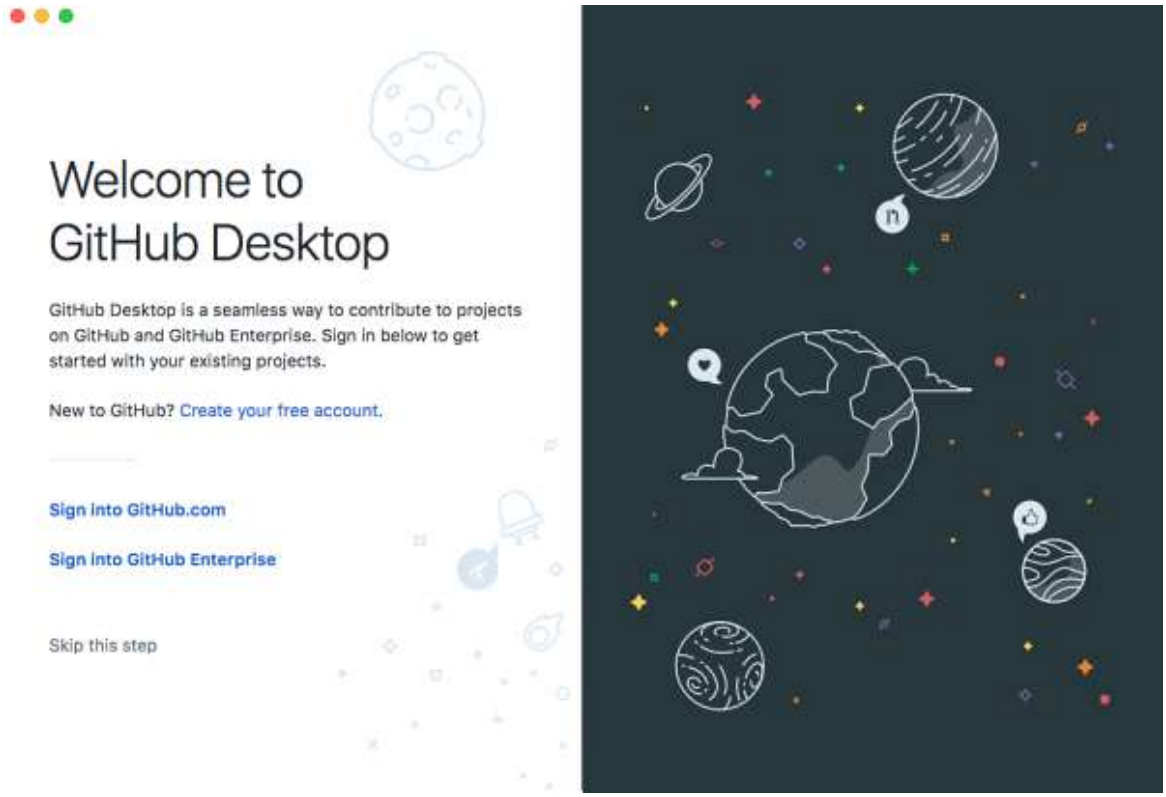


Figure 8-3. Welcome to GitHub Desktop

Unless your company has a GitHub Enterprise setup and you're going through this setup specifically for that, it would be best to first sign into GitHub.com by clicking the "Sign into GitHub.com" link. If you later need to sign into GitHub Enterprise, you'll be able to do that as well. If you have enabled two-factor authentication to make your account more secure, you'll be asked to enter the code that either was texted to your mobile phone or is available in the two-factor authentication app you set up.

Once you've done this, you should see a screen similar to Figure 8-4.

Click the "Sign in" button, and you'll be prompted for some information to configure Git. In the first text box, enter the name you want to be known by, and in the second, enter the email address you'd like your commits to be associated with. Usually you'll enter your full name into the first text box and the same email address you use for your GitHub account in the second one, as I've done in Figure 8-5.

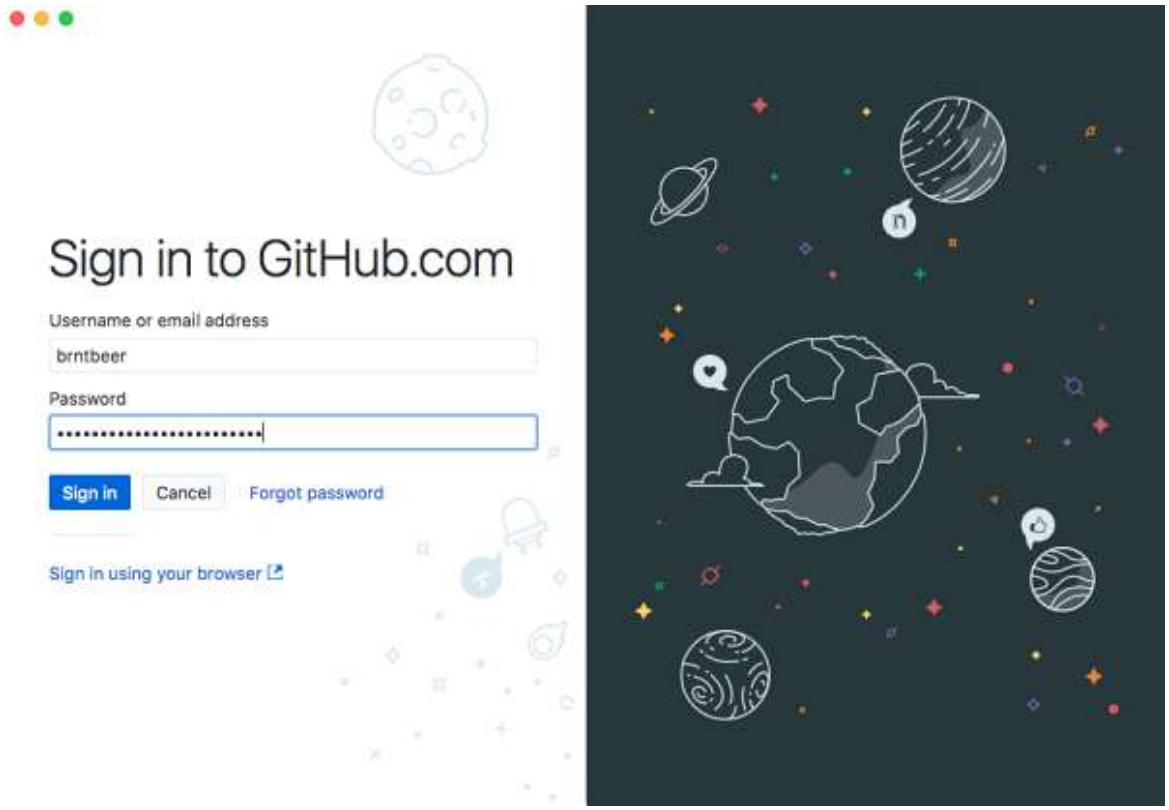


Figure 8-4. Sign into your GitHub account

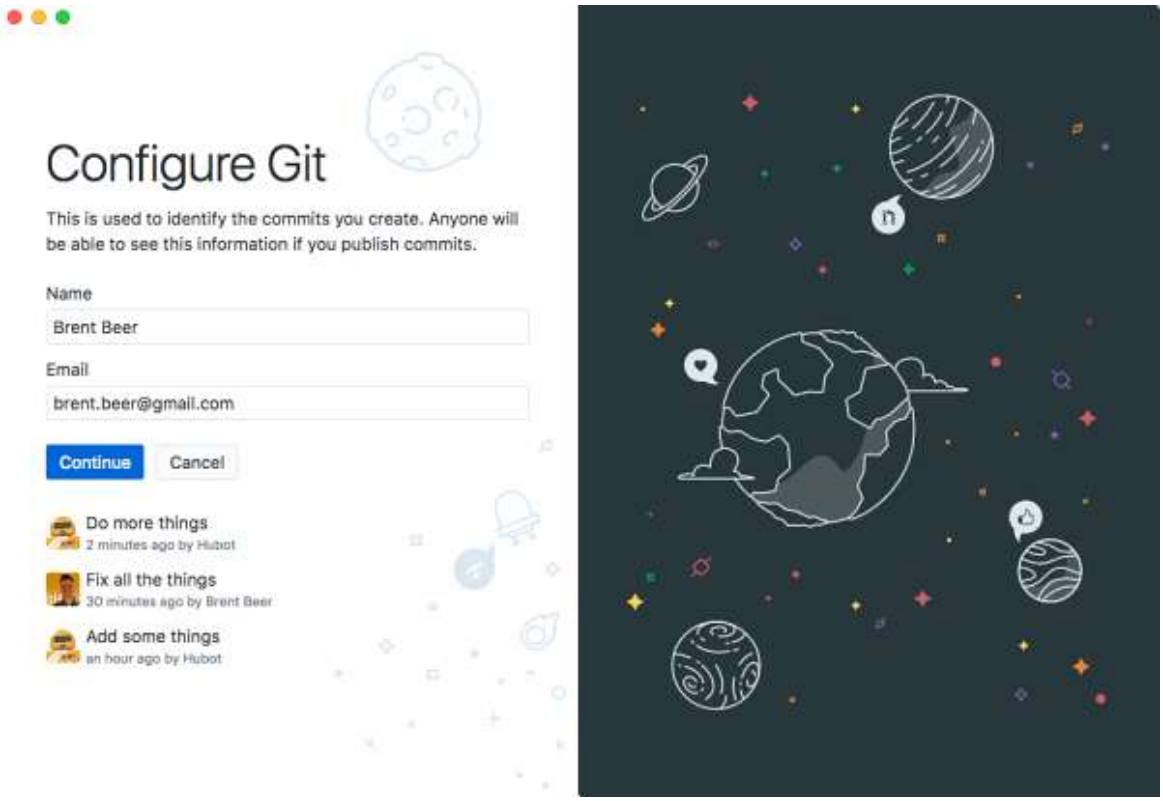


Figure 8-5. Configuring your Git settings

Click the Continue button and you'll be taken to a screen that allows you to create a new repository, add an existing repository you have on your computer, or clone a repository from GitHub. As you have more repositories that you work with locally, you'll see different repositories here; for now, you should see a blank screen that looks like Figure 8-6.

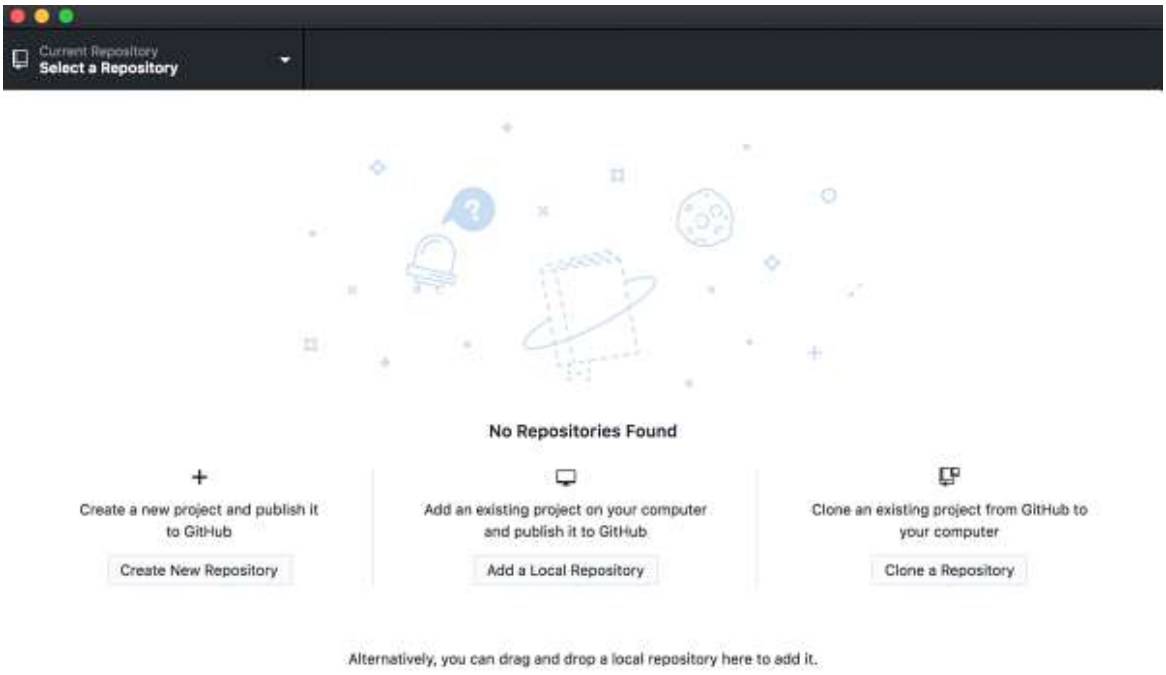


Figure 8-6. The home screen for GitHub Desktop

Now that you've installed GitHub Desktop, go to a repository that you'd like to clone (download) and that you own or are a collaborator on. You can clone any public repo, but you won't be able to push your changes back up to GitHub unless you're either an owner, a collaborator, or have sufficient write permissions within the organization if it belongs to one. I'm going to reuse the repository I created back in Chapter 3 to continue where we left off.

If you look at the right side of the page above the file list, you should see the "Clone or download" button that, when pressed, will give you two options: "Open in Desktop" or "Download ZIP" as shown in Figure 8-7. Go ahead and press "Open in Desktop" to get this repository onto your computer.

When you click "Open in Desktop," GitHub Desktop will open up and present you with some options for where to save this repository and what the URL of the repository is on GitHub. This information will be prefilled for you and should look similar to Figure 8-8. It may be best to keep those

defaults for now; if you later decide to move this folder to a different location, you will just have to let GitHub Desktop know where you've moved it to.

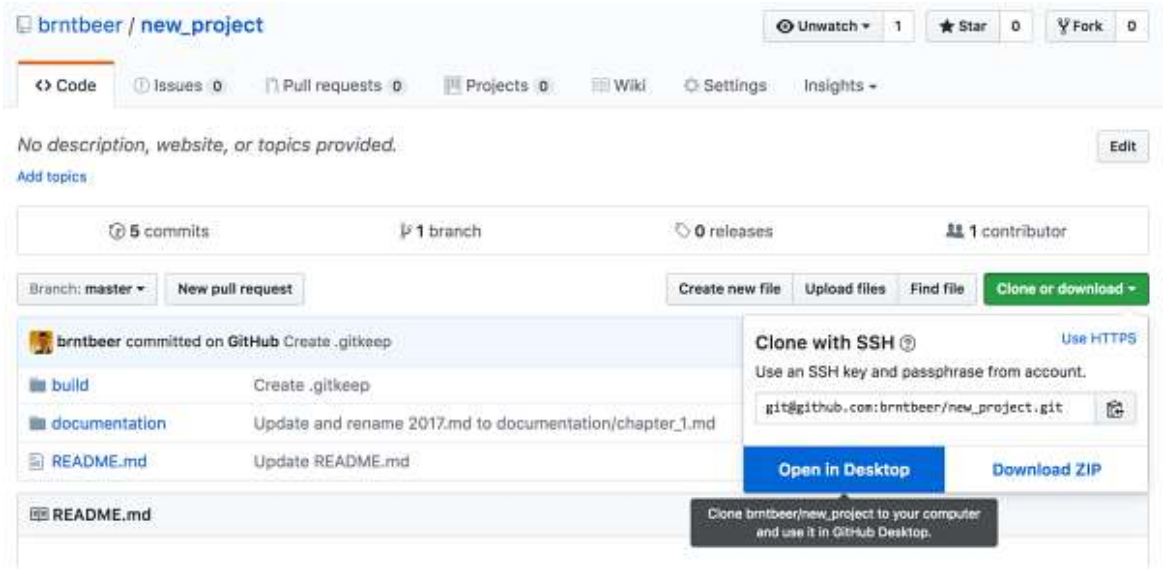


Figure 8-7. Clone or download options in a repository

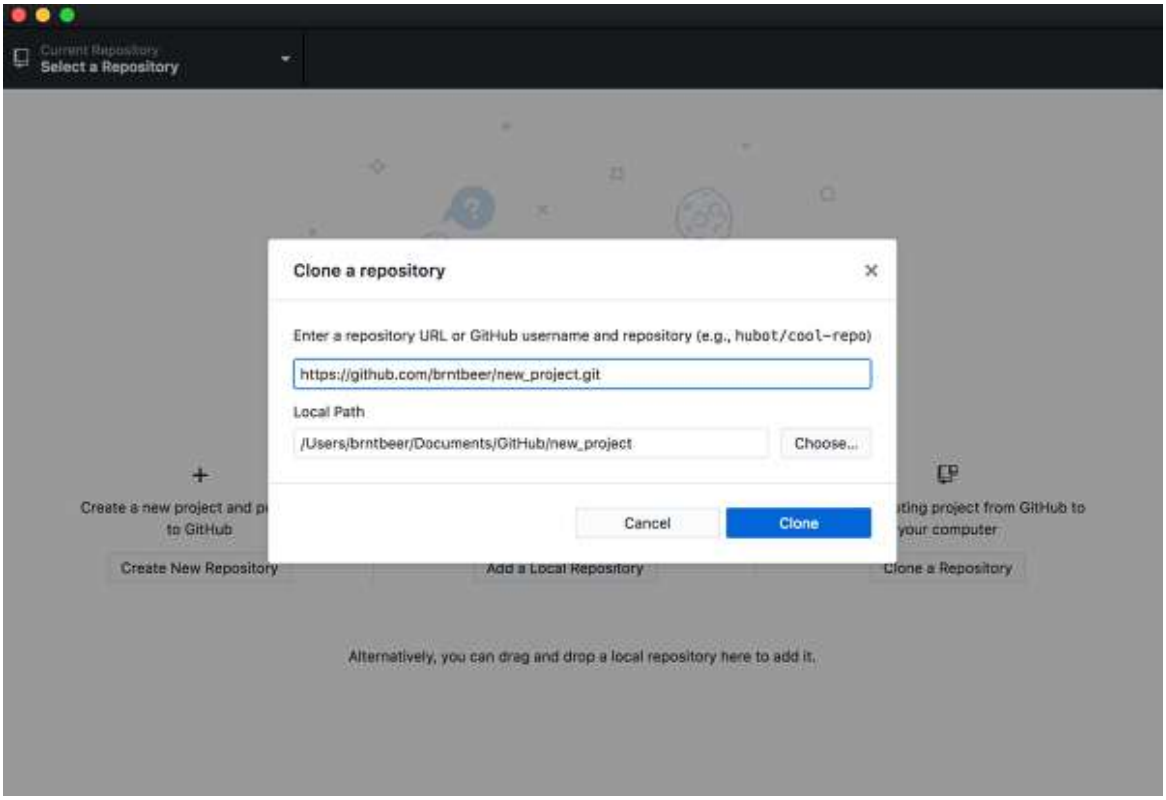


Figure 8-8. Confirming where to clone the repository

Once the repo has been successfully cloned, you should see a screen similar to Figure 8-9.

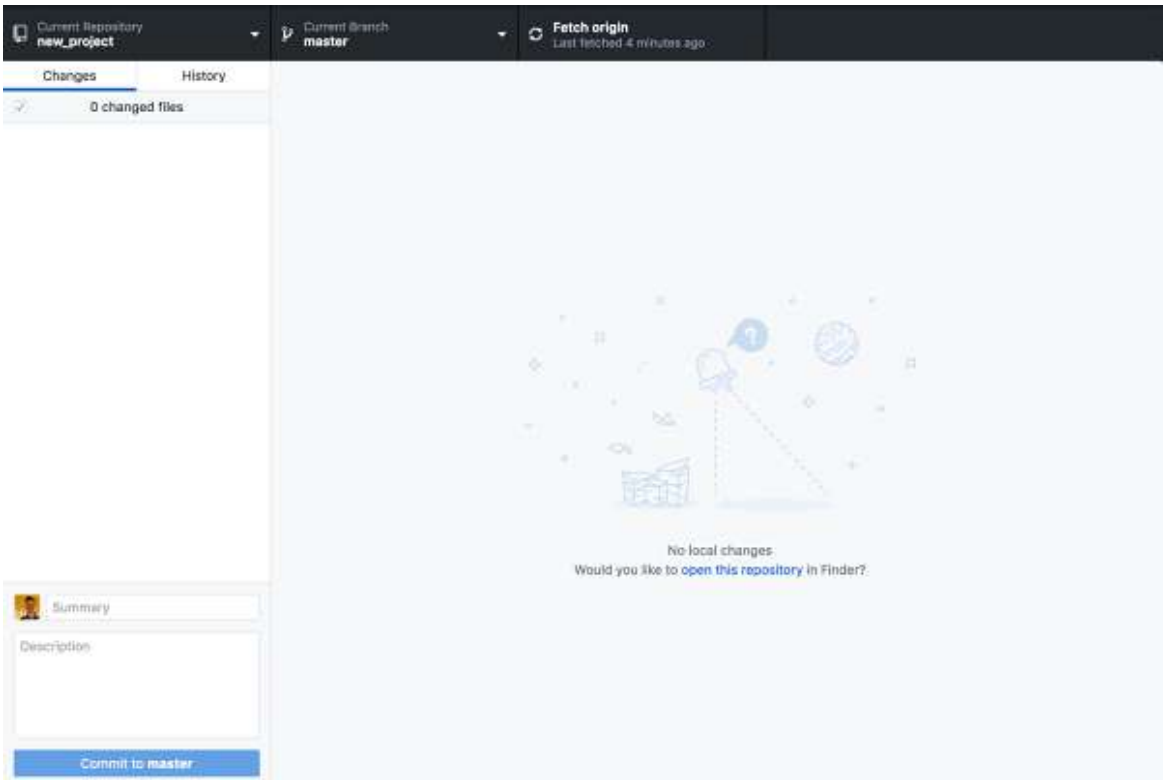


Figure 8-9. Viewing a repo in GitHub Desktop

If at any time you want to switch to a different repository you have a copy of locally, you can do so from the drop-down in the upper-left corner. For now, let's take a look at some of the other areas of GitHub Desktop before we start making changes to push back to GitHub.

Viewing Changes

There are two tabs on the home screen of a repository in GitHub Desktop: Changes and History. You'll notice that initially there's not much to see, but that's because you haven't actually changed any files yet. This will change in just a bit as you start editing files. If you click History, it will show you a list of commits on your current branch, as shown in Figure 8-10.

Figure 8-10. The History view

If you click the middle drop-down menu that mentions the current branch you're on, you'll see a screen similar to Figure 8-11.

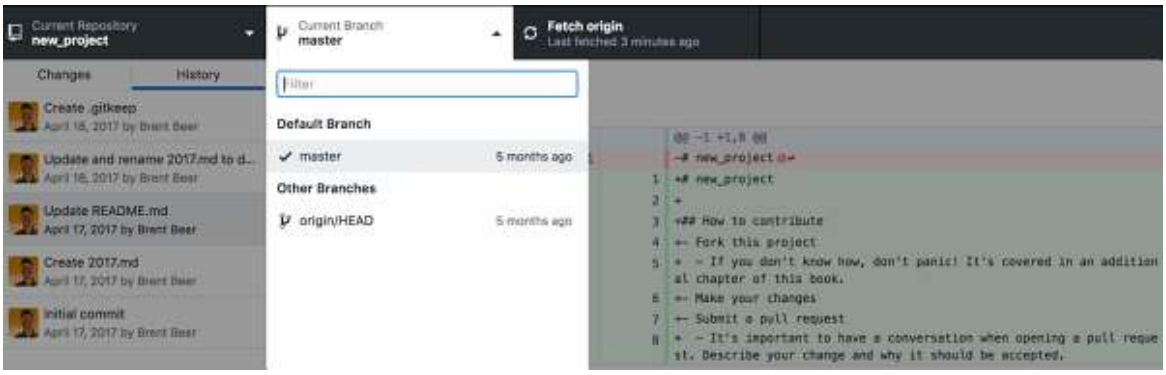


Figure 8-11. The branch list

If you clicked History earlier and didn't see some changes you expected, you may have needed to switch to a new branch—maybe that work was only on a different branch and hadn't yet been merged into where you were looking. This branch view shows a list of all of the branches that you've created locally and all of the other branches that are on GitHub, and clicking on one will switch you to it. Finally, to view the repository settings, you need to click on the Repository drop-down in the menu bar and select Repository Settings, as shown in Figure 8-12.

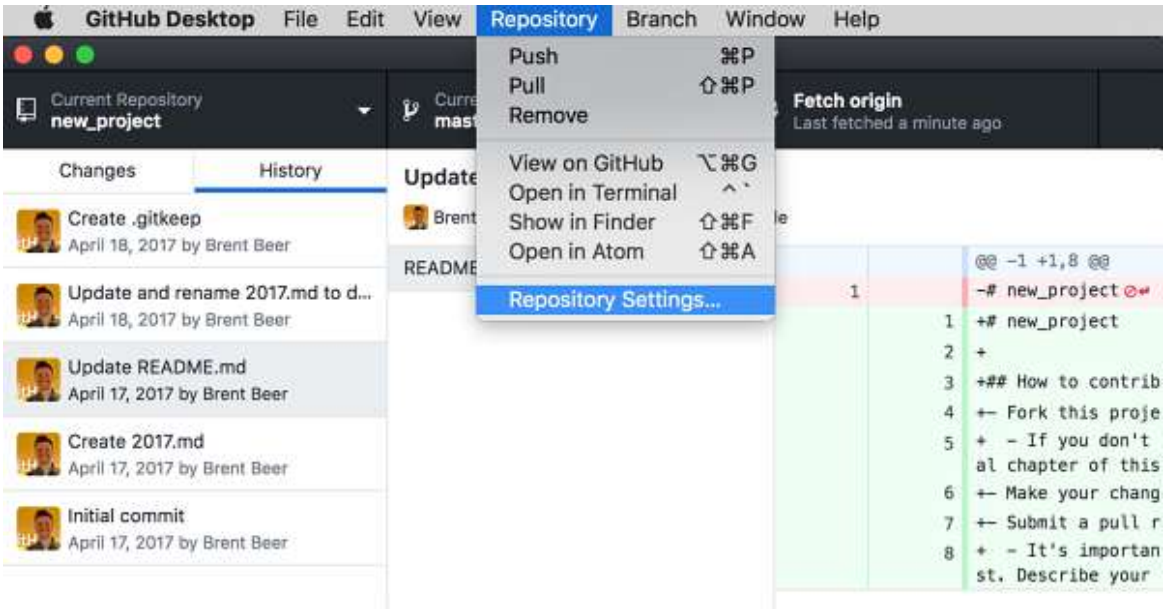


Figure 8-12. Finding the repository settings

Once you click on Repository Settings a small box will appear with two tabs, Remote and Ignored Files, as shown in Figure 8-13.

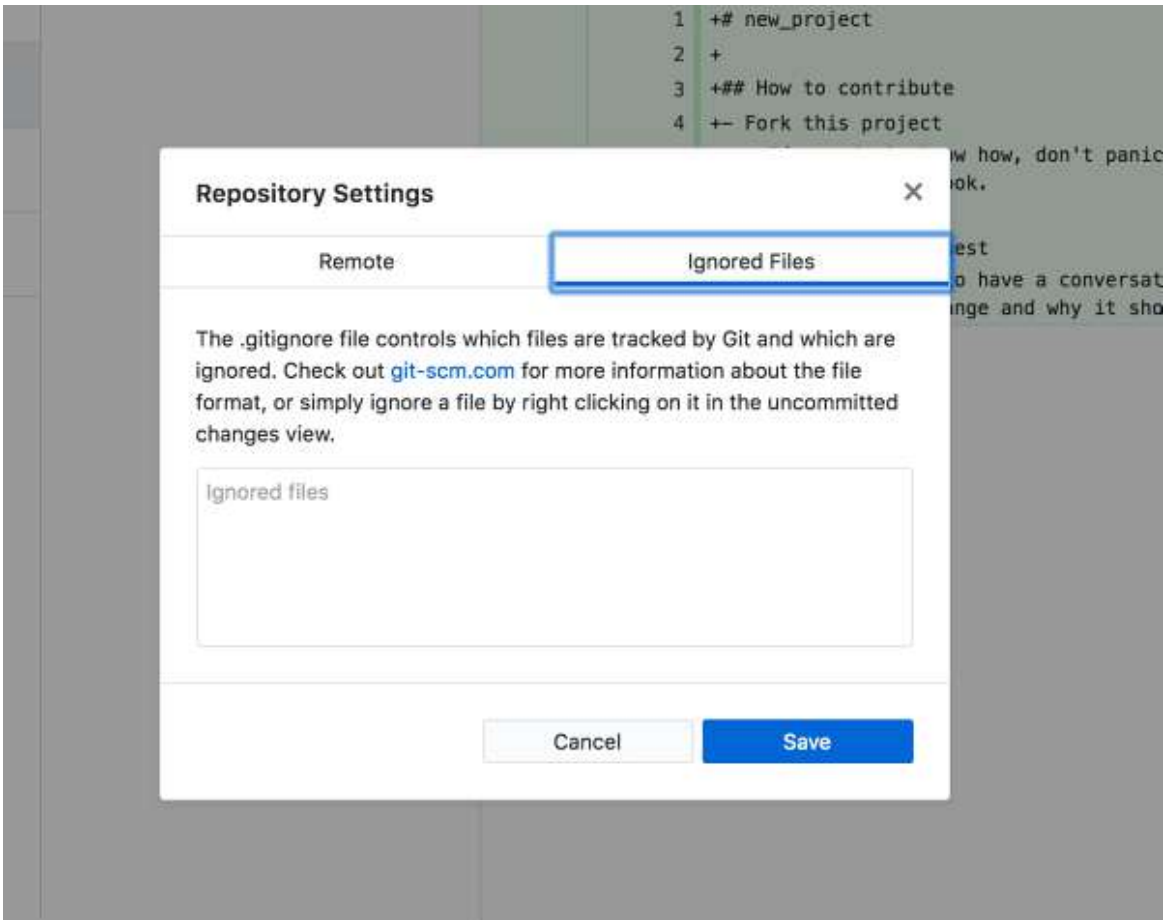


Figure 8-13. Files ignored by this repository

The Remote here is just in case you need to change where your repository is on GitHub (because you renamed it or transferred it). The Ignored Files tab is for files you don't want Git to pay attention to and track changes of. These could be personal note files, IDE configurations, executable files, log files, really anything! Lots of different coding languages and IDEs often have standard files they wish to ignore; you can see examples of these at *https://github.com/github/gitignore*. You may also see this populated already if the repository you cloned was not brand new.

Hopefully, now you have enough familiarity with the interface to know how to clone a repository and where certain functionality is should you need to do so using the GitHub Desktop application. The next section will take you a step further in actually editing some additional content with a more realistic workflow.

GitHub Desktop and Atom

To really use GitHub Desktop to its full potential, you need to start changing files with a text editor. You may have your preference of text editor, but to continue we're going to be using Atom, GitHub's open source text editor. To get started with Atom, download it from the Atom website, which should look similar to Figure 8-14.

Figure 8-14. The Atom text editor's home page

We're just going to use it to familiarize you with workflows in GitHub Desktop for now, but Atom itself has Git and GitHub functionality built into it. Additionally, this functionality and its appearance is similar to that of GitHub Desktop because it's built with the same underlying technology, which is called Electron. If you ever have a website that you need to have a desktop version of, you may want to check out *https://electron.atom.io*.

Getting started with Atom is similar to GitHub Desktop, though you don't have to go through a sign-in and setup process. Start by clicking the "Download For Mac" or "Download Windows 64-bit Installer" link on the home page. Just like with GitHub Desktop, this will download a ZIP file for Mac or a setup file for Windows. In Windows, once you click this setup file everything is taken care of. On a Mac, you will need to double-click the ZIP

file, and then you should move the application that's contained within into your *Applications* folder and double-click that file to open and install Atom. Once this finishes on either operating system, you should see a welcome screen with some information just like Figure 8-15.

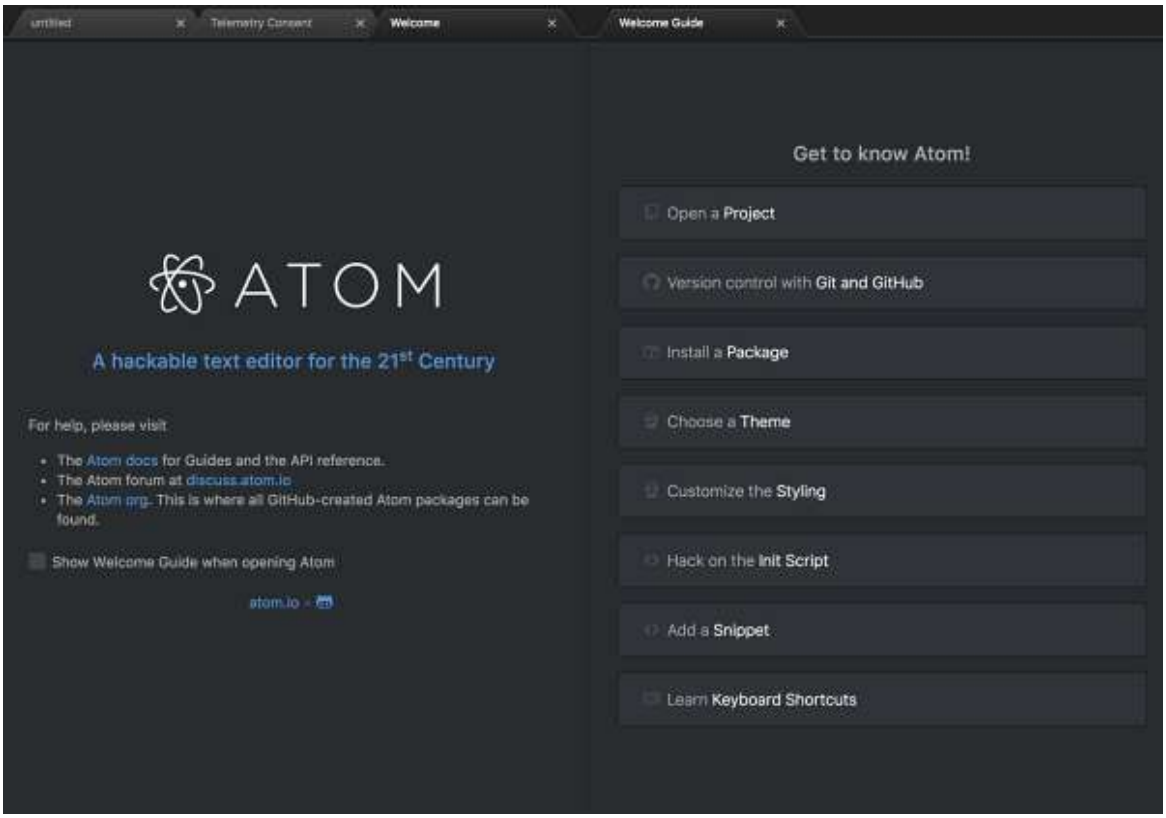


Figure 8-15. Welcome to Atom

This book won't be going through and explaining how to use Atom, with its seemingly limitless extensibility. However, a lot can be learned by going through the Welcome Guide you see in Figure 8-15. If you ever need to get back to it, you can always find it in the Help menu.

Creating a Branch and Editing Files

To make a new commit using GitHub Desktop, you will want to start by creating a new branch. Creating a branch before you edit any files and make any commits is not only required for making a pull request, it's also a best practice. In Figure 8-16 you can see that to do so I need to select New Branch from the Branch menu, and then in Figure 8-17 I'm creating the branch and naming it `updating-documentation`.

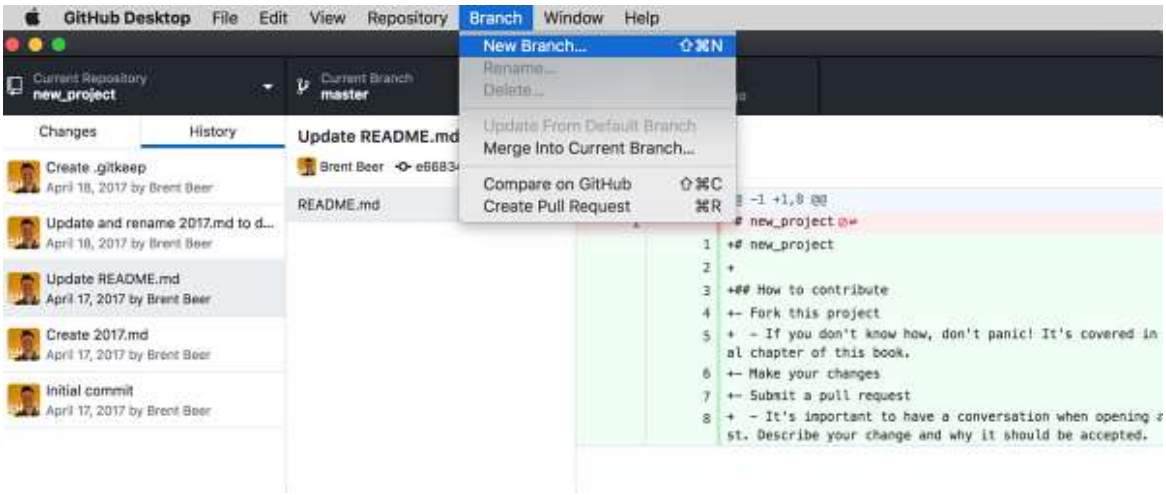


Figure 8-16. The menu option to create a new branch

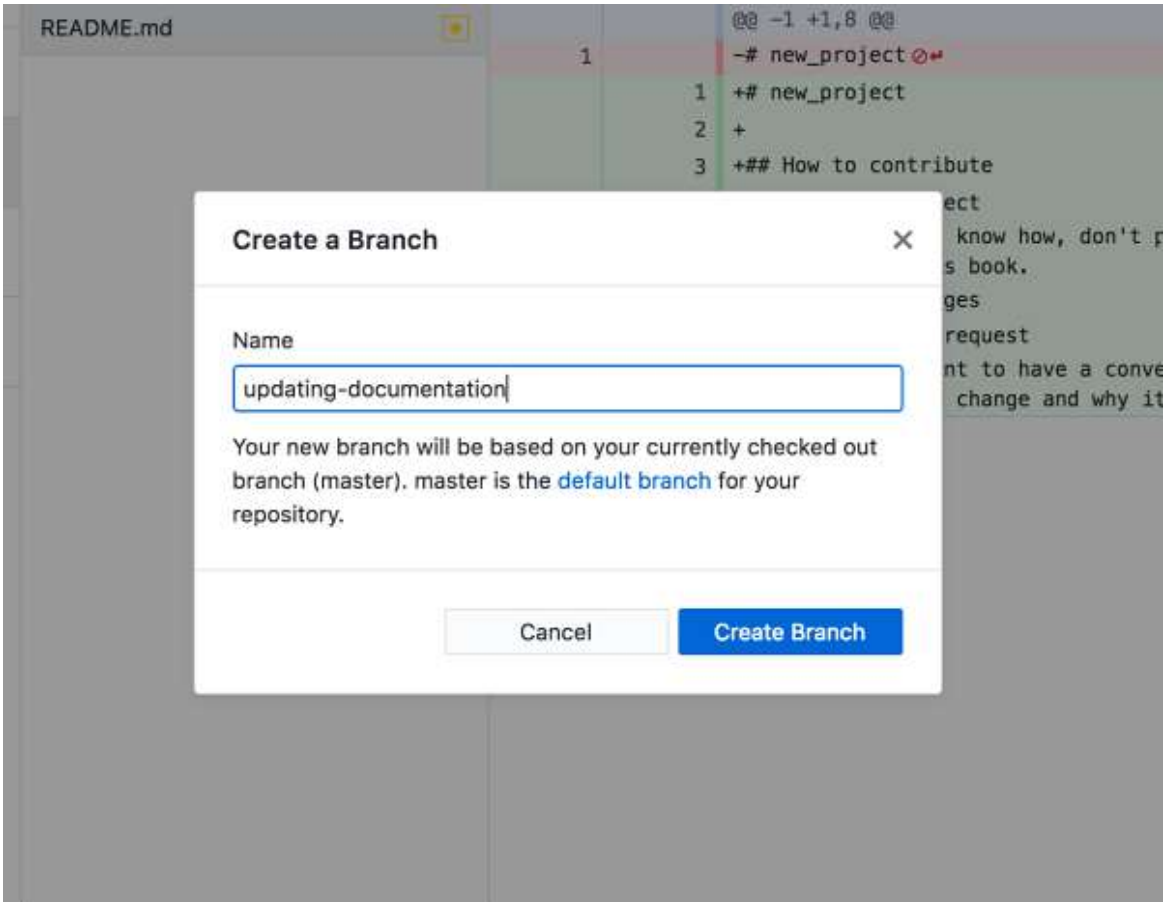


Figure 8-17. Creating a new branch

If you look at the middle drop-down menu on your screen, you should see that you're on the branch you just created. Now you need to add or edit

some content. To do that, you need to open Atom. GitHub Desktop makes that easy to do right from the Repository menu, as seen in Figure 8-18.

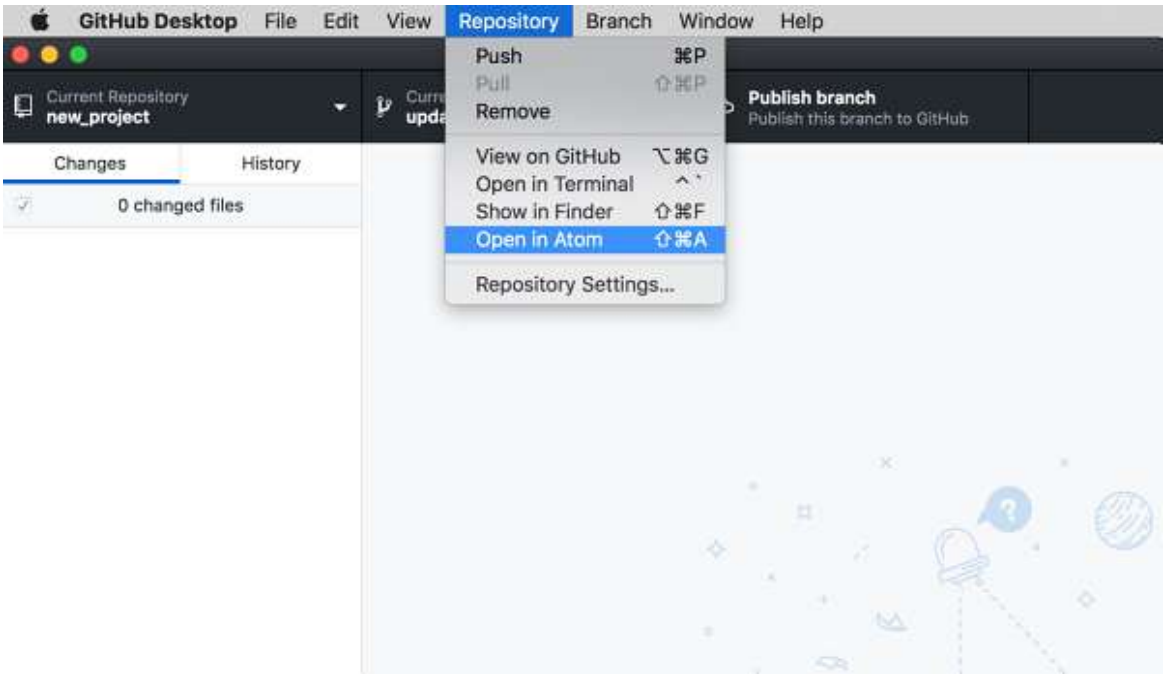


Figure 8-18. Open your current repository directly in Atom

Let's walk through an example together. Once Atom opens, we're going to update the repository's documentation a little bit, just like was hinted at in the branch name. Though we could make changes to the repository in the web editor, the changes we'll make are not possible to do in a single step. We want to create a new file and edit *README.md* to point to that new file. I'd have to do this in at least two steps in the web editor, and I might not even have a connection to the internet when I'm doing changes like this.

The first thing we should do is to create a new file for our contributing guidelines. Though these are in *README.md*, GitHub will link to the contributing guidelines in every pull request sent to this repository if we add these guidelines to a file called *CONTRIBUTING.md*. As you can see in Figure 8-19, the first step is to create the new file by right-clicking the project name in the left panel (the tree view). This creates the new file at the top level of the repository and not in any folder. When you select New File, you should see a dialog box like the one in Figure 8-20 asking you to name the file. Name it *CONTRIBUTING.md*—be careful not to forget the file

extension if you want your content to highlight and render appropriately! If you do forget the file extension, you can always right-click the file and select “Rename” to rename it.

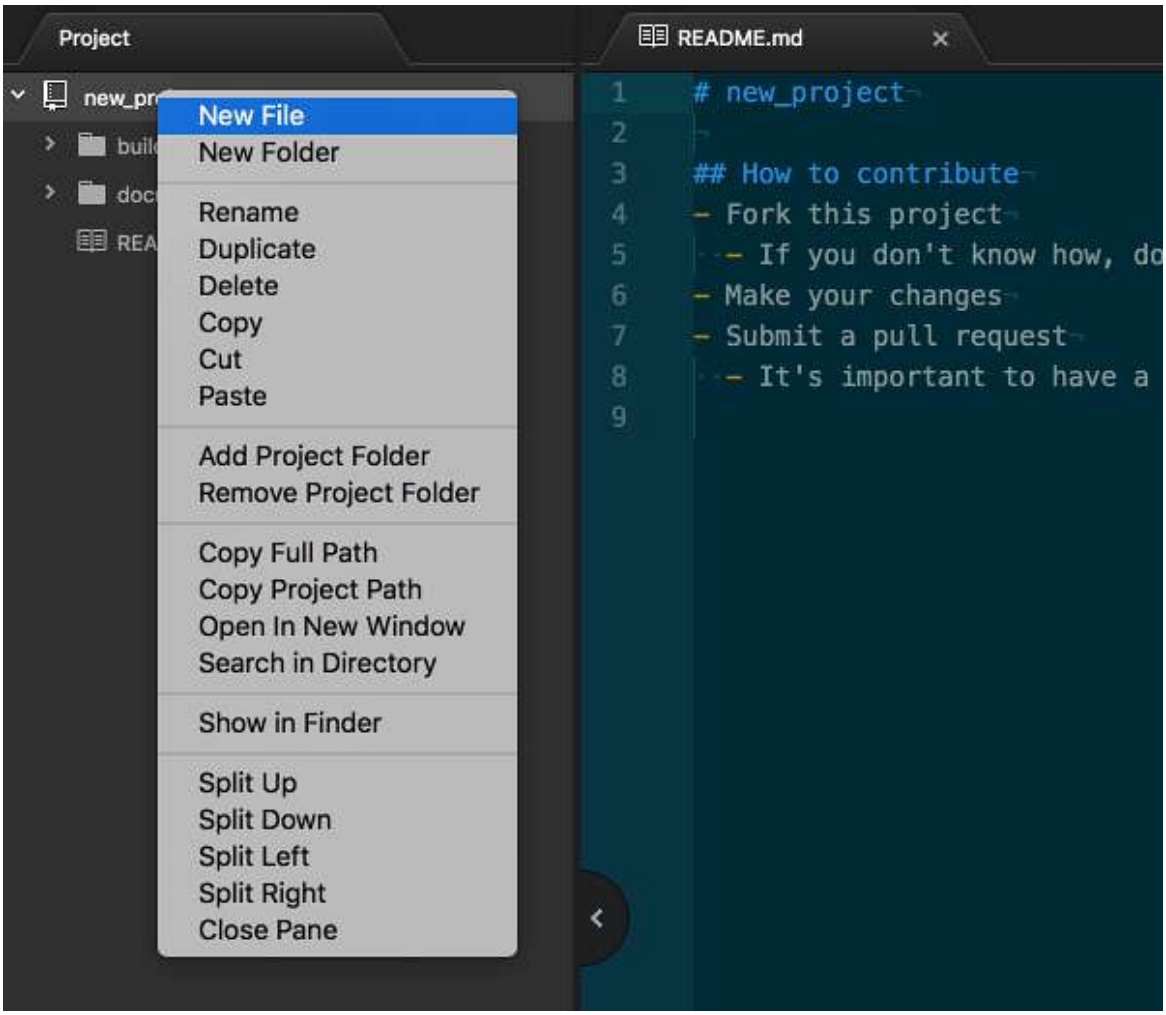


Figure 8-19. Right-click in the tree view to create a new file

Figure 8-20. Name the file CONTRIBUTING.md

Once the new file is created, we need to change the section “How to contribute” in *README.md* to point to this new file using basic Markdown formatting, which you can see in Figure 8-21.

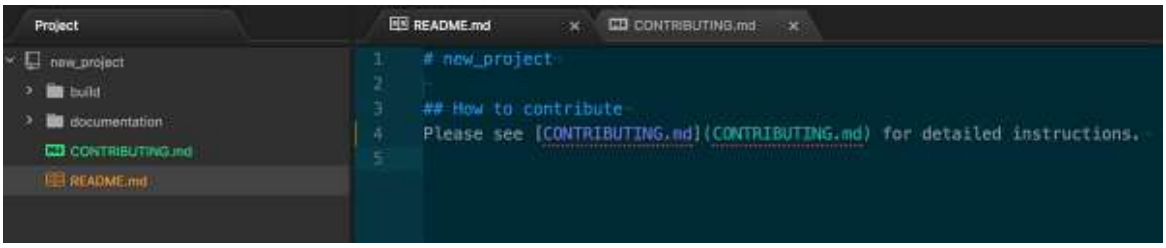


Figure 8-21. Markdown link to CONTRIBUTING.md

To finish out the feature—and this is the part that can’t be done in the web browser as easily without already having a commit—copy and paste the original contributing steps into *CONTRIBUTING.md*, as seen in Figure 8-22. Once this is there, make sure both files are saved before switching back to looking at GitHub Desktop to make the commit.

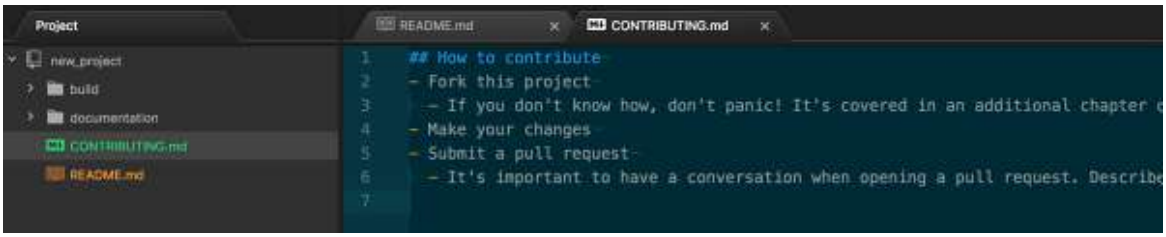


Figure 8-22. Content copied into CONTRIBUTING.md

Creating a Commit

Now that the files have been edited, let’s create a commit to record in the history that these changes have happened. The reason we want these changes to be recorded together is because they are the same logical unit of work. There are many preferences when making commits in the software development community, and this is only a rule of thumb, but it can make it easier for others to follow our work. If we switch back to the GitHub Desktop application we should see that it has already detected our changes, like in Figure 8-23.

One important thing to notice in Figure 8-23 is that both files that have been edited are checked in the Changes tab. If one of them were to be unselected,

then that file would not be committed. This can be useful when you are editing multiple files at once but are not yet ready to make a commit on one of them. Similarly, when you click on either file to view the changes in that file, you can see the diff of the changes, just like when you view a pull request.

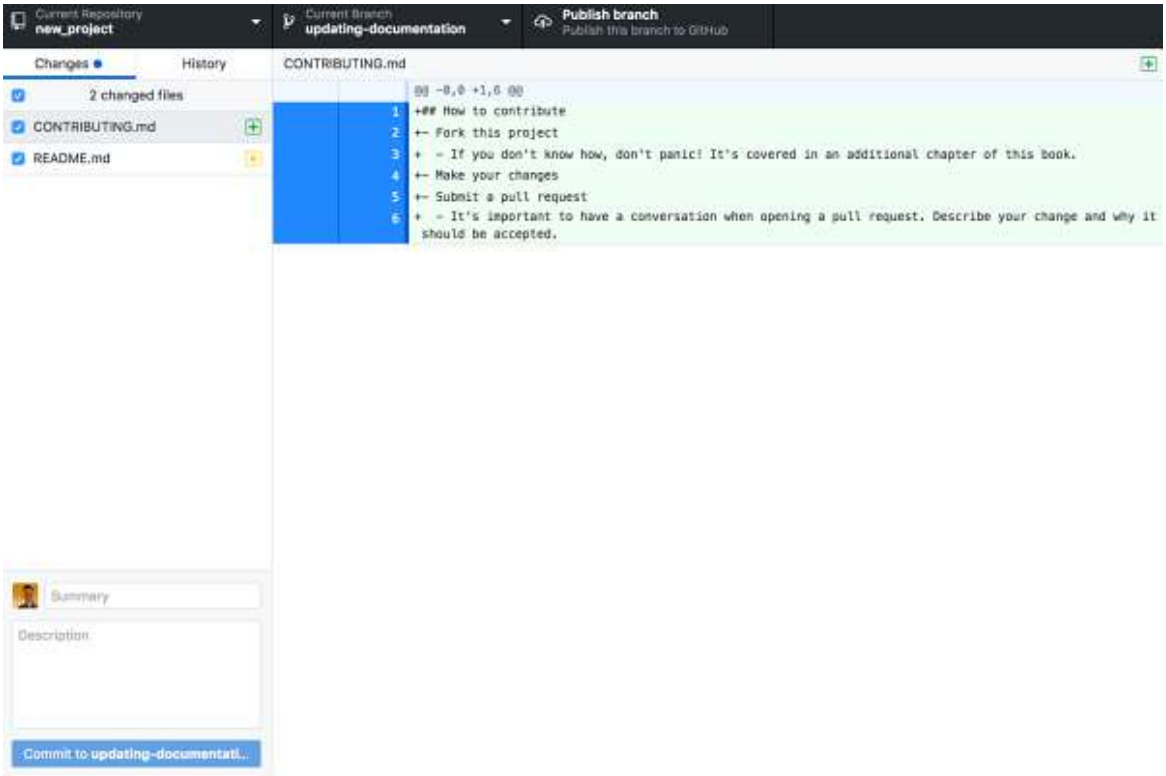


Figure 8-23. GitHub Desktop has detected the changes

Just like in the workflow outlined at the beginning of Chapter 3, we need to write a commit message to record this change. An example commit message can be seen in Figure 8-24. Notice that just like before, you do not need to include anything in the “description” part of the commit message, though it may be useful for complicated changes.

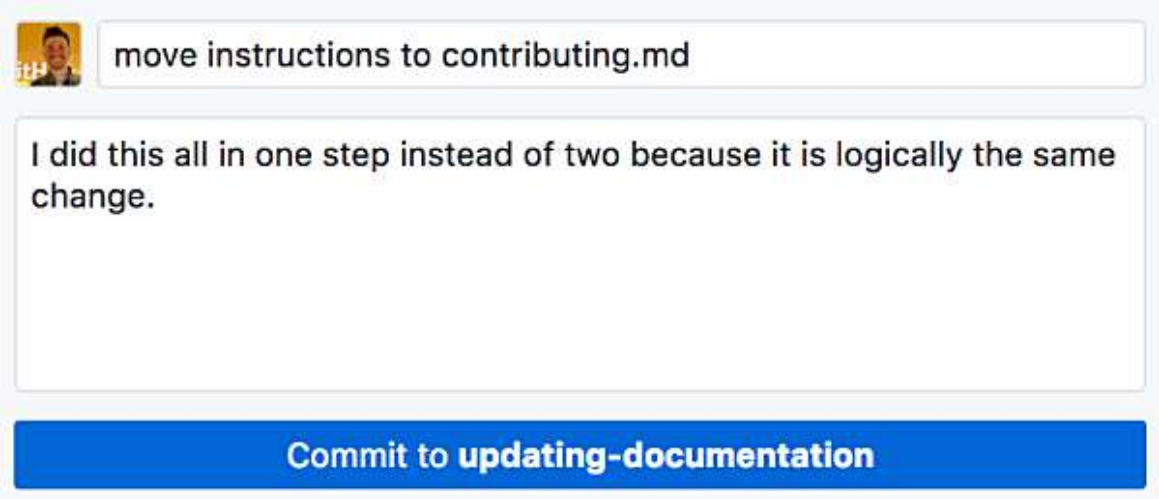


Figure 8-24. Sample commit message

Now that we have some changes to discuss, we need to push our changes up to GitHub to open a pull request.

Creating a Pull Request from GitHub Desktop

There are a few ways to create a pull request, but one step is required before we can begin. So far the work we've created, the files we've edited, and the changes we've committed only exist locally on our computers. We need to send them up to GitHub so others can work on them, but also so we can create the pull request. Previously, we were already working in the web browser, so there was no need to push these changes up to GitHub.

Whenever you need to send work to GitHub, or grab some work a collaborator has sent to the repository, you can do this from the rightmost menu, which should look similar to Figure 8-25. When you work on a branch that starts out just on your computer first, like we've been doing here, that menu will appear as it does in Figure 8-25. Afterwards, you'll see a syncing option that will send any changes you've made on your computer or receive any changes that have been pushed up to GitHub onto that branch.

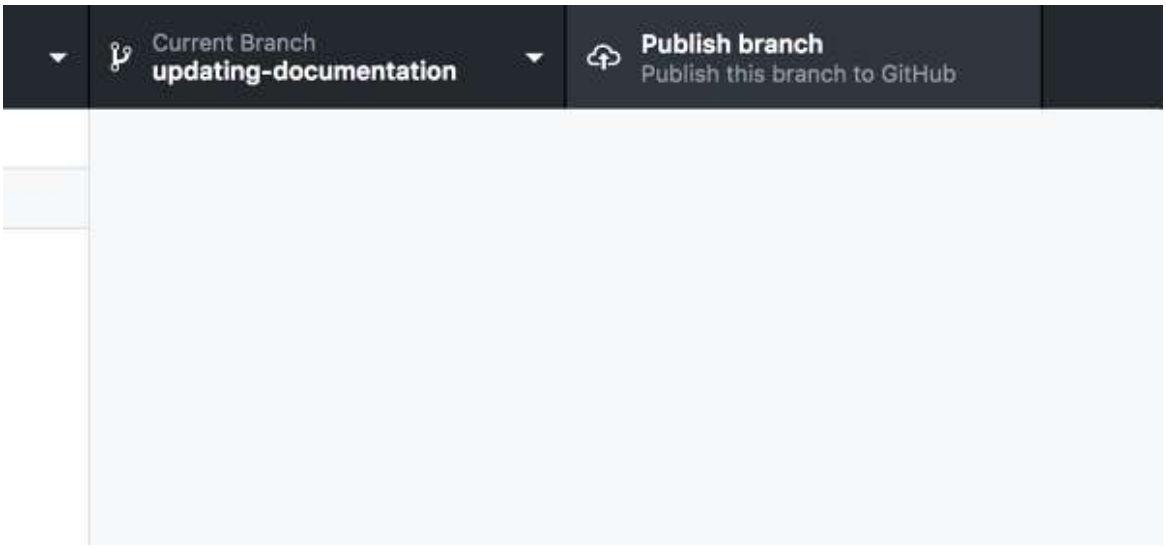


Figure 8-25. The menu option for publishing your changes

Though pushing our changes is important, we have yet to start our conversation about these changes in a pull request. This can be done from the GitHub Desktop Branch menu, as seen in Figure 8-26. Choosing the Create Pull Request option will open a browser window to the repository with a new pull request started. If you wanted to, you could also have done this manually from the repository home page in a browser window.

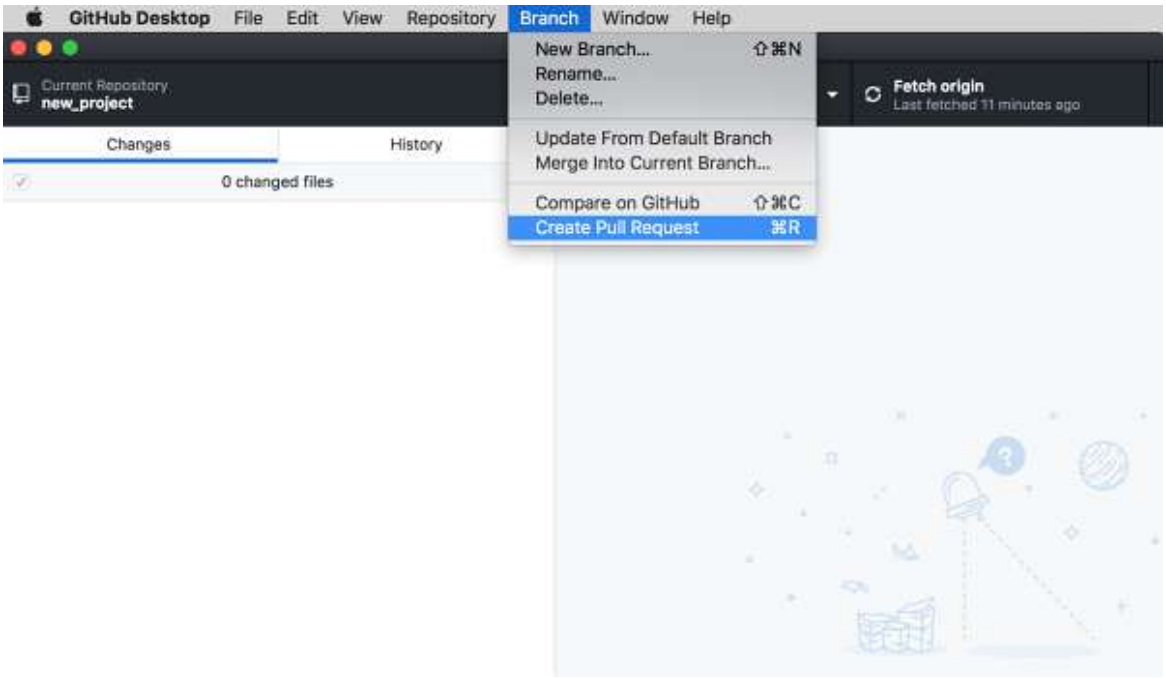


Figure 8-26. The menu option to start a pull request

Just like we did at the beginning of Chapter 4, we need to fill in some information to start our pull request. I've filled in some sample information in Figure 8-27.

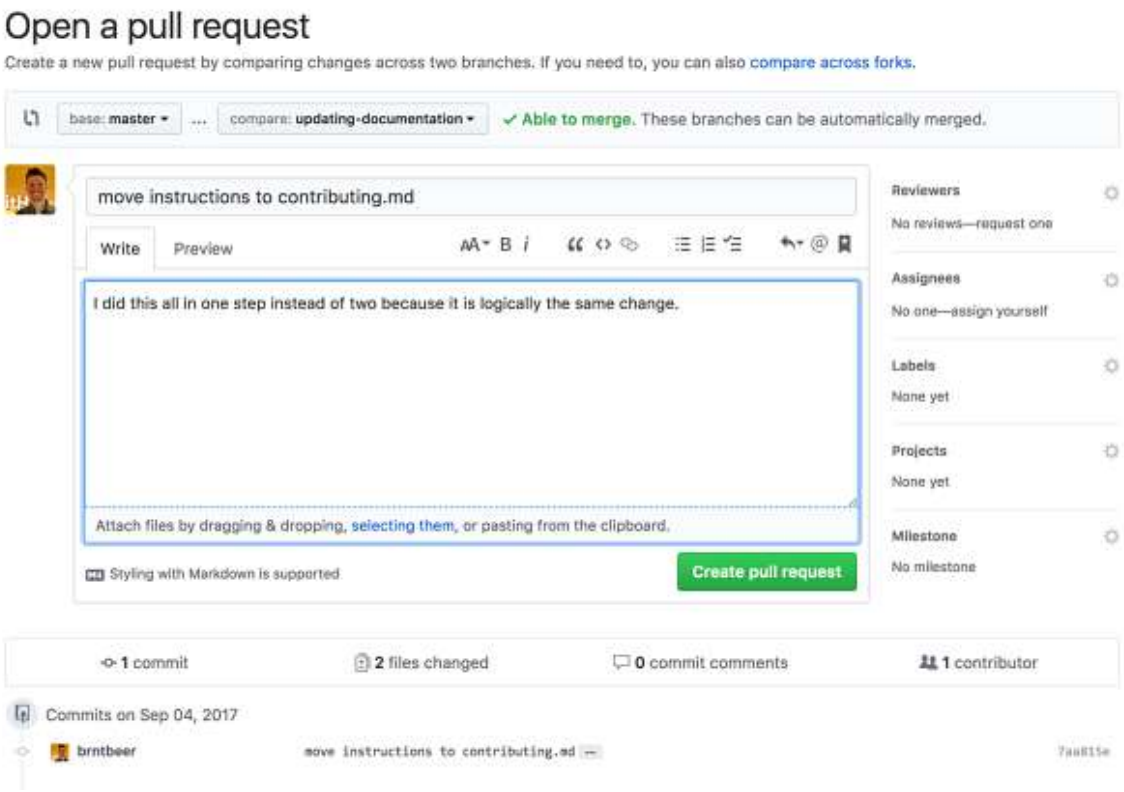


Figure 8-27. Start the pull request by starting the conversation

If this was a more serious repository with collaborators, I might go through some proper code review and wait for some integration tests to run. However, this is just my own sample repository, so I'm going to merge the changes as soon as the pull request is created.

The last part of this workflow is to ensure all of the work that has been done on GitHub is brought down to your computer. The easiest way to do this is to switch back to the `master` branch from the middle menu, which displays the current branch, and make sure you fetch your changes. Once you've done this, you'll see a notification of how many changes there are on the server that you need to pull in; as you can see in Figure 8-28, there are two changes. After these changes are pulled in, the History tab should show those changes in its log.

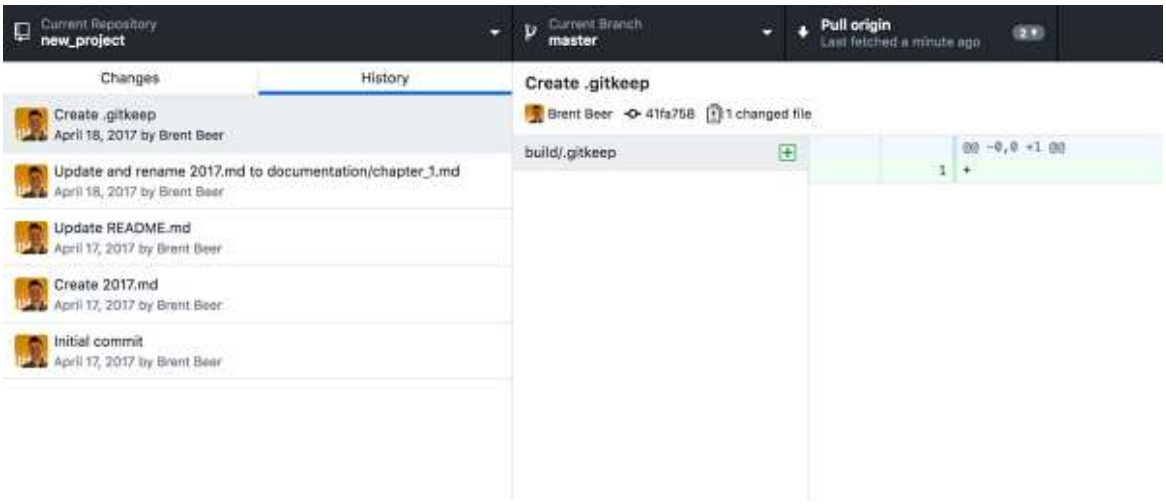


Figure 8-28. Pull changes from GitHub to ensure your master branch is up to date

At this point, you have all the tools you need available to you for continued software development and expanding your workflows and collaboration even further!

Chapter 9. Next Steps

We've covered a lot of ground in this book. We started by looking at how to view a project and then moved through the process of forking a project, making edits and offering them back, and then collaborating on a single repository, all with pull requests. We looked at how to create and configure a new repository, as well as an organization, and how to use the GitHub Desktop client to download and work on repositories locally.

For many people, this is all you'll need to know. The important next step is to practice until the skills become second nature and collaborating via GitHub becomes a natural way for you to work with teams of people on text-based files—whether source code or other projects.

There are some things that can only or best be done on the command line. You may never need to make the jump to the command line, but if you're working on projects in Git all day, every day, it makes sense to learn how to use Git from the command line. Jon Loeliger and Matthew McCullough have created a great book called *Version Control with Git* (O'Reilly), which would be an excellent next step if you wanted to learn more about using Git from the command line.

GitHub also provides a number of resources for learning more about both Git and GitHub. For more information, go to *https://services.github.com*.

GitHub is going to become an increasingly important part of the workflow of many companies. This is a great time to get familiar with it. Best of luck with the journey!

Brent Beer
Amsterdam
September 2017

Index

Symbols

A

B

D

E

F

G

N

W

About the Author

Brent Beer has used Git and GitHub for over five years through university classes, contributions to open source projects, and professionally as a web developer. He previously worked on the GitHub Training Team, where he taught the world to use Git and GitHub to their full potential, and now works as a solutions engineer for GitHub to help bring Git and GitHub to developers inside their companies across the world.

Colophon

The animal on the cover of *Introducing GitHub* is a bare-tailed woolly opossum (*Caluromys philander*), an arboreal species of marsupial also known as the white-eared opossum. This species is restricted to moist forests, and can be found in Brazil, Bolivia, French Guiana, Guyana, Suriname, Trinidad and Tobago, and Venezuela. With its prehensile tail—which allows it to climb, balance, and grasp objects—the white-eared opossum is rarely, if ever, found on the ground and seldom found in the understory.

Ranging in weight from 140 to 390 grams, the female bare-tailed woolly opossum is typically smaller than males. It generally has soft and thick fur, which differs depending on the animal's habitat and location. It has a reddish-brown back with gray gradations along its flanks and a yellow-orange belly. It has a gray head with distinct dark brown stripes that run down the bridge of its muzzle and out from the dark brown eye rings to the nose. About a quarter of its tail has fur; the rest is furless and cream to dark gray or brown in color with brown or white spots.

The mating ritual of the bare-tailed woolly opossum is a bit of a mystery. Generally, individuals are solitary except when males court females. White-eared opossums have up to three litters per year, depending on resource availability. Females can have up to seven young at one time, averaging around four young per litter in the wild; this, too, depends on resource availability. Bare-tailed woolly opossums have short gestation periods (24 days) and extended periods of parental care (up to 120 days of pouch time and 30–45 days in the mother's nest). Leaving the mother's nest is an important behavior, as demonstrated in captivity when young who have not been removed cannibalize their mother.

The bare-tailed woolly opossum is not listed as a species of concern, which is credited to its small size and adaptability to various types of neotropical forest. This could change as deforestation of neotropical regions continues.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to

animals.oreilly.com.

The cover image is from *Meyers Kleines Lexicon*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.