

Practical Series

UNDERSTANDING GIT AND GITHUB
AND USING BRACKETS TO MANAGE THEM

BY MICHAEL GLEDHILL



Revision:

R01—22 Sep 2017

Copyright 2017

Practical Series of Publications

All Rights Reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except under the terms of the Copyright Designs and Patents Act 1988 without the permission in writing of the publisher.

This publication was set using Equity, Concourse and Triplicate fonts.

Published By:



Practical Series of Publications

by Michael Gledhill

Published in the United Kingdom

Email: mg@practicalseries.com



First Release Edition

2017

Document No.: PS1002-5-2121-001

Based on Template: T0-1101-007 R01.01 Formal Publication (Equity).Dotm

Revision History

R01

22 Sep 2017

First formal release

A NOTE ON PRINTING & VIEWING THIS DOCUMENT

This document is designed to be printed two sided (it makes it look better—*I spent a lot of time on those margins*). If you are looking at the pdf version in Adobe Acrobat Reader and want it to look right, view it in *Two-Page View* ([VIEW → PAGE DISPLAY → TWO PAGE VIEW](#)).

To make it put odd pages on the right hand side (so it looks like the printed version), select [VIEW → PAGE DISPLAY → SHOW COVER IN TWO PAGE VIEW](#).

CONTENTS

A note by the author	11
1 Introduction	13
1.1 How to use this website	16
1.2 How to pay for this website	18
2 Git, the concept	19
2.1 What is version control?	24
2.1.1 Why use version control	25
Collaboration	25
Version storage	26
Tracking changes	26
Restoring changes and regression paths.....	27
Project backup	28
VCS the downside	28
2.2 Version control with Git.....	29
2.2.1 Working, staging and the local repository—how it works	31
2.2.2 Commit version numbers	40
2.2.3 How to view commits	41
2.2.4 Committing changes—best practice.....	42
2.3 Branching.....	43
2.3.1 What happens when you switch branches	46
2.3.2 Changing branches with uncommitted changes	47
2.4 More branches and merging	49
2.4.1 Conflict when merging.....	54
2.4.2 Branches and work flow—best practice.....	57
2.5 Recovering an older commit with a reset	58
2.5.1 Setting up a project to explain the reset command	58
2.5.2 A hard reset.....	62
2.5.3 Making changes after a reset.....	63
2.5.4 Using resets—best practice	66
2.5.5 A mixed reset	69
2.5.6 A soft reset	71
2.5.7 Reset, a summary.....	72

2.6	Tagging	73
2.6.1	Lightweight tags	73
2.6.2	Annotated tags	74
2.6.3	Using tags	74
2.6.4	Tag naming restrictions	75
2.7	Ignoring files with .gitignore	76
2.8	The remote repository.....	78
2.8.1	Local and remote repositories: the differences	78
2.8.2	Where does a remote repository live?	78
2.9	Moving data between local and remote repositories.....	79
2.9.1	Sending (<i>pushing</i>) data to a remote repository	80
2.9.2	Getting (<i>pulling</i>) data from a remote repository.....	81
2.9.3	What happens if there is a conflict with the remote.....	82
2.9.4	Creating a local repository from an existing remote	83
2.9.5	A note on remote connection names	84
2.9.6	Working with remotes—best practice	84
3	Installing Git	85
3.1	Downloading and installing Git	89
3.2	Downloading everything else you need	93
3.2.1	A default text editor for Git.....	93
3.2.2	A difference and merge tool.....	97
3.3	Changing the Git default directories	102
3.3.1	Changing the default directory for Git	103
	Changing where Git and Git Bash store the configuration files	103
	Setting the start directory	106
3.4	Changing the Git default configuration	109
3.4.1	Configuring a Username and email address.....	109
3.4.2	Configuring a default (core) text editor	111
	Adding Notepad++ to the PATH Environment Variable.....	111
	Setting the Git Bash core text editor.....	116
3.4.3	Configuring a default difference tool.....	118
3.4.4	Configuring a default merge tool.....	119
3.4.5	Defining an alias.....	120

4	GitHub and SSH Linking.....	123
4.1	Creating a GitHub account.....	126
4.2	Creating a first repository	132
4.3	Setting up an SSH key on the local machine	135
4.3.1	Creating an SSH key pair.....	135
4.3.2	Adding the public key to GitHub.....	142
4.3.3	Testing the SSH connection	145
5	Git inside Brackets	147
5.1	Installing Brackets-Git.....	150
5.2	Creating an empty GitHub repository.....	153
5.2.1	Create a new repository using GitHub	153
5.3	Cloning the GitHub repository with Brackets.....	156
5.4	Changes commits and push from Brackets-Git.....	159
5.4.1	Creating a new folder in the repository	159
5.4.2	Modifying a file.....	161
5.4.3	Committing the changes	163
5.4.4	Pushing the changes back to GitHub.....	164
6	Git & Brackets a worked example	167
6.1	Creating the local repository	170
6.2	Creating the folder structure and initial files.....	174
6.2.1	The <code>.gitignore</code> file	174
6.2.2	Creating a folder structure	176
6.2.3	Adding images to the project.....	178
6.2.4	The <code>README.md</code> file.....	179
6.2.5	The <code>index.html</code> file	184
6.2.6	The <code>style.css</code> file.....	185
6.3	The first commit	188
6.3.1	Staging the files	188
6.3.2	Making the first commit	190
6.3.3	A note on the first commit with Brackets	192
6.3.4	Viewing the commit history	194
6.3.5	Tagging the commit	197
6.3.5	The current workflow.....	198
6.4	Adding another commit	199
6.4.1	Viewing the file history	202
6.4.2	Viewing the contents of a file at a previous commit	205

6.5	Branches.....	206
6.5.1	Adding a branch in Brackets	206
6.5.2	Making a commit on a new branch	208
6.6	Switching branches	211
6.6.1	Switching branches with uncommitted changes	213
6.6.2	Discarding the latest changes	214
6.7	Merging, conflicts and deleting branches	215
6.7.1	Merging a branch with Brackets	220
6.7.2	Deleting a branch with Brackets	224
6.7.3	Merging a branch with a Conflict	225
7	Regression with Brackets.....	235
7.1	Brackets—enabling the reset and checkout functions.....	238
7.1.1	Date and time settings	239
7.2	Regressing to an earlier commit with Reset.....	241
7.2.1	Resetting to an earlier commit point	242
7.3	Regressing to an earlier commit with Checkout	252
8	Brackets and remote repositories	259
8.1	Extending the current project	262
8.2	Creating the remote repository	267
8.3	Pushing a local repository to a remote.....	270
8.3.1	Linking Brackets to the remote repository.....	270
8.3.2	Pushing the local repository to the remote repository	272
8.3.3	Pushing another branch to the remote.....	279
8.4	Pulling remote changes into the local repository.....	283
8.5	Conflicts between local and remote repositories.....	292
8.6	Deleting remote branches.....	304
8.6.1	Deleting the remote branch	308
8.6.2	Deleting the local branch.....	310

9	GitHub.....	313
9.1	GitHub—the basic pages	316
9.1.1	GitHub profile—newsfeed page.....	316
9.1.2	A repository home page	319
9.2	GitHub—finding and viewing files.....	321
9.2.1	File view.....	322
9.2.2	Viewing the contents of a file.....	323
	The <code>RAW</code> button	325
	The <code>HISTORY</code> button	326
	The <code>BLAME</code> button	327
	The <code>GITHUB DESKTOP</code> icon	328
	The <code>EDIT</code> and <code>DELETE</code> file icons	328
9.2.3	Viewing other types of files	329
9.3	GitHub—creating, modifying, deleting and committing files	331
9.3.1	Creating new folders and files.....	331
9.3.2	Committing files.....	333
9.3.3	Uploading a file	335
9.3.4	Editing a file	339
9.3.5	Renaming a file	342
9.3.6	Deleting a file	343
9.4	GitHub—navigating commits and regression.....	345
9.4.1	The commit history	347
	Making additional commit comments	349
9.4.2	Recovering data from an earlier commit.....	352
9.5	GitHub—branches. Merging and conflicts.....	354
9.5.1	Creating a new branch	354
9.5.2	Creating a file on the new branch	357
9.5.3	Merging branches with a <code>PULL REQUEST</code>	360
	Creating a pull request.....	361
	Reviewing the pull request	363
	Rejecting a pull request.....	364
	Starting a pull request conversation.....	365
	Accepting a pull request.....	365
9.5.4	Examining previous pull requests.....	367
9.5.5	Pull request to merge a branch with a conflict	369
9.5.6	Deleting a branch	376
9.6	GitHub—tags and releases.....	377
9.6.1	Tags and releases, what's the difference?.....	378
9.6.2	Viewing releases (and tags)	379
9.6.3	Creating a release	381
9.6.4	Pulling releases back into Brackets	384

10	Collaborative working.....	385
10.1	Forking.....	389
10.1.1	Forking (copying) a repository	391
10.1.2	Creating a pull request on a forked repository	395
10.1.3	Synchronising a forked repository (the limitations)	403
10.2	Issues and milestones.....	404
10.2.1	Creating issue labels	404
10.2.2	Creating milestones.....	405
10.2.3	Creating an issue	407
10.2.4	Closing an issue directly	409
10.2.5	Closing an issue from a commit.....	409
And finally.....		413
	Contact Michael Gledhill	415
	Acknowledgements.....	416
	Colophon	419
	Legal.....	420

Appendices	421
A Installing Brackets	423
A.1 The Brackets text editor.....	425
A.1.1 Getting and installing Brackets	425
A.2 Adding extensions to Brackets	427
B A revision numbering mechanism.....	431
B.1 Workflow arrangements	434
B.2 Master branch revision states.....	435
B.3 Development branch revision states.....	436
B.4 Parallel development branches.....	439
B.5 Nested development branches.....	440
B.6 A note on numbering.....	441
B.7 A note on commit messages	442
C Markdown.....	443
C.1 General markdown syntax	445
C.1.1 Headings.....	445
C.1.2 Block quotes.....	445
C.1.3 Emphasis.....	446
C.1.4 Ordered lists	446
C.1.5 Unordered lists	447
C.1.6 Combined lists.....	447
C.1.7 Images &links	448
C.2 Git flavoured markdown syntax.....	449
C.2.1 Code fragment.....	449
C.2.2 Task lists	449
C.2.3 Tables	450
C.2.4 Special GitHub references (issues and users)	450
C.2.5 Escape characters.....	450

D	Git and Command Line.....	451
D.1	Git command line summary	453
D.2	Git Bash instruction summary	455
E	Git Advanced topics	457
E.1	Rebase	459
E.1.1	Rebase—the rules	460
E.1.2	Understanding rebase	461

A NOTE BY THE AUTHOR

This is my second Practical Series publication—this one happened by accident too. The first publication is all about building a website, you can see it [HERE](#). This publication came about because I wanted some sort of version control mechanism for the first publication.

There are lots of different *version control systems* (VCS) out there; some are free and some are commercial applications—just [GOOGLE](#) it. If you do, you will find that Git and GitHub show up again and again.

I also noticed something as I wrote and developed my first publication and used various third party open source software and applications:

- Normalise.css
- Lightbox
- MathJax
- Prettify

I noticed that all the people developing these applications did so by using GitHub—so, thought I, there must be something in it—*best have a look and see what it's all about*.

So I did, and while GitHub and its poor provincial, command line cousin, Git are powerful version control applications, it's fair to say that they're also very difficult to understand.

I had a couple of goes at it and gave up each time—Git and GitHub have a pretty steep learning curve, especially Git (the application that runs on a local machine rather than GitHub that is web based). Git is driven through a command line terminal emulator and it's a bugger to use.

So I did a bit more digging and I found that the Brackets text editor, the one I use to develop the Practical Series websites, has an extension that provides a Git and GitHub interface directly within the Brackets development environment.

This extension (it's called *Brackets-Git*) has the distinct advantage that it is easy to use and doesn't require command line inputs that are counter intuitive and hard to remember.

Git and GitHub, apart from having peculiar (and slightly rude sounding names), also have a wide range of in-house terminology and phrases, things like this:

- Gist
- Push
- Repo
- Fork
- Pull request
- Flirt
- Flagellate

Ok, I made the last two up, but you wouldn't necessarily know. It is all very confusing and it's not that well explained.

It takes a long time to get to grips with Git—to such an extent that I had to make lots of notes, do a lot of reading around the subject, try a lot of different things and experiment with it until I had an understanding of what it all meant and how it worked.

And since I had written it all down, I thought it might be useful to other people—so I decided to publish it, and here it is.

Michael Gledhill
April 2017

1

INTRODUCTION

The why, what and how of this website (book)
why I did it, what it's for and how to use it.

THIS PUBLICATION (WEBSITE) is, in effect an online book. The purpose of this book is to explain how the Git *version control system* (VCS) and its online partner in crime GitHub work; specifically how these two applications can be made to work from within the development environment of the Brackets text editor.

I explain everything from scratch:

- ① The concepts used by Git and GitHub for version control
- ② Installing Git
- ③ Using Git Bash (Git's command line terminal) to set up the default configuration for Git on your machine
- ④ Setting up a GitHub account
- ⑤ Installing Brackets and the tools needed to incorporate Git and GitHub
- ⑥ An example web project to demonstrate the proper way to use Brackets and the Git and GitHub version control system
- ⑦ Instructions for working with GitHub directly
- ⑧ Using GitHub in collaboration with others

I also provide additional material explaining the command line interface and some useful information for developing with GitHub.

1.1

How to use this website

THIS WEBSITE (and the document you are reading now) is an explanatory text on Git, GitHub and version control using the Brackets text editor.

The site is divided into the following sections:

SECTION	NAME	PURPOSE
1	Introduction	This section, an introduction to the website and what it's for
2	Git, the concept	How the Git and GitHub version control system works and the best practice for using it
3	Installing Git and other packages	An explanation of how to get and install Git, the other packages you need and how to set it all up
4	GitHub and linking it to a local machine	Setting up a GitHub account and how to use the SSH protocol to link it to a local machine
5	Using Git and GitHub from within Brackets	An introduction to using Git and GitHub from within the Brackets text editor environment
6	Git and Brackets: A worked example	A fully worked example demonstrating all the aspects of using Git through Brackets with a local repository
7	Regression with Brackets	How to recover information from earlier stages of a project
8	Brackets and remote repositories	Using Brackets to control and work with both local and remote repositories
9	GitHub	Using GitHub, the online version of Git, to manage remote repositories
10	Collaborative working	Working in collaboration with others using GitHub
Appendices	Appendices	Defines common terms, references and an explanation of other aspects of Git and GitHub

Table 1.1 Document sections

The sections are designed to be read in order; however if you are familiar with Git and GitHub you can skip directly to section 5 to see how to implement a Git/GitHub interface directly from within brackets.

Those not familiar with Git and GitHub and how it works should start at section 2. I explain there how the version control system works and the terminology used in Git and GitHub.

Sections 3, 4 and 5 explain how to set up Git, GitHub and Brackets respectively. These sections contain very detailed step-by-step instructions with lots of pictures.

Sections 6, 7 and 8 contain fully worked examples for developing a small website under Git version control and deploying it on a remote repository with Brackets and GitHub.

Sections 9 and 10 give details of using GitHub (the online version of Git) by itself and how to work in collaboration with others.

The appendices contain other useful information:

- ① A best practice for version numbering
- ② A guide to markdown syntax
- ③ A guide to GitHub markdown syntax
- ④ Git Bash terminal and command line instructions summary
- ⑤ Understanding Git rebase

1.2

How to pay for this website

This website (book) is an experiment—I've written it, I've drawn the pictures, I've invested my time and effort in producing it, I've bought the fonts, I've paid for it to be hosted and I've enjoyed doing all of this (well the paying less so).

I wanted to produce something that had a certain quality to it, the sort of quality that would be present in a printed book. There are a great many websites that look good (graphically) and there are a great number that explain to the n^{th} degree how something works (and generally these don't look so good).

This, I hope, is a good compromise; I hope I've explained things well and that I have created a site that is pleasing to the eye, is easy to navigate, and, in short, I hope it is something you find agreeable and useful and worthy of your time.

But that doesn't mean it's free. If you do like the website and find it useful, all well and good—I want you to use it and learn from it; but I also want you to support it.

“*How do I do that?*” you ask—well that bit's easy, you can:

- ① **MAKE A DONATION** (I suggest £2-£15 though anything is welcome)—donate the amount that best reflects your appreciation of the site
- ② Tell other people (free)—I can't say this is as good as the other option but exposure is better than nothing

There is no advertising on this website and I don't want there to be. Please take the points above seriously; I've done my bit in producing the website, I like to think you will do your bit too by supporting it—enough said.

Michael Gledhill
April 2017

2

GIT, THE CONCEPT

How Git and GitHub work, the fundamentals of version control and the best practices for using Git and GitHub.

I AM BY PROFESSION an engineer, I build and programme control systems for a living (industrial machinery, pharmaceutical manufacturing, even tritium handling systems for nuclear reactors). The software that goes in these systems is (*usually*) rigorously tested and anything that goes into a regulated environment (pharmaceutical, nuclear, safety systems &c.) has quite stringent version control and traceability requirements.

Version control is important, particularly when the reactor melts down and you want to know who's to blame¹. It's also important when you've completely screwed things up and you need to back to an earlier, less screwed up version.

Now I write industrial software and these applications usually have their own development environments and version tracking mechanisms that are particular to the product being used (Siemens, ABB &c). Other things such as documents and drawings are controlled by the in-house version control system that we operate in the office.

When I started the Practical Series website, I wanted some form of version control, it was just me working on it, so I didn't need anything too ambitious—and in the early days I just kept increasing the revision number and backing up everything each time I did so.

And this was fine, up to a point. Websites are not particularly massive things, a few megabytes; and having multiple copies doesn't tax a modern hard drive particularly. *So what was the problem?*

Well the main problem was that I wasn't documenting the revisions properly; the website has a lot of files (way over a thousand) and while I had a copy of every file at every

¹

I worked for one company that shall be nameless (let's just call them Consolidated Gyroscope); they operated a blame-free culture intended to prevent accidents by encouraging people to report the smallest incident. The idea being if you stop the small things, you will prevent the larger things. This was fine for small things, nobody minds being told to use a coaster for their coffee cup. Bit different when the building burns down.

It did lead to a bit of a sub-culture—whilst they operated a blame-free policy, they did like to know whose fault it was.

revision I didn't necessarily know which files had changed from one revision to another.

It was also a very inefficient mechanism; there were probably some files in there (images for example) that were in at the first revision and were backed up without change in every subsequent revision up to the latest (there were over 150 when I stopped).

It also became complicated when I wanted to work on two different things at once; I might for example be correcting typos in one section that had just been proofread, while developing a new section from scratch. It was difficult to keep track of each.

So while what I had worked, *sort of*—I could always go back to an earlier version. It was laborious; I would have to guess which revision I wanted, unzip it and then look at the file I wanted to see if it was the right version, if it wasn't I had to guess another revision and do the same until I found the one I was looking for. This was ok in the early days; but at the point where I switched over to proper version control I had 78,000 files in 35,000 folders; and 150 zipped revisions. The whole thing was taking up 7 GB most of which was identical copies of files that hadn't changed between revisions.

It was at this point that I decided I needed some proper version control and while I could have used the office system, that didn't feel right; the website was nothing to do with my work (*more an expensive hobby really*) and I didn't want to take advantage. Neither did I want to buy the office version for home use, it was just two expensive and over the top for what I needed.

So I had a look around, there are lots of different version control systems out there; some are free, some are commercial applications. The question is: *which one to use?*

One thing that gave me a clue was the software I had used in developing the website; I had used various open source software (some CSS files, some Java Script code) that improved or added functionality to the website (things like lightbox images and formulae on web pages). I noticed as I researched these things, that virtually all of the people developing these applications used GitHub as their version control system.

So I decided to have a look at GitHub and I realised that this was the online version of Git, Git being a version control system that runs on a PC or Mac.

My conclusion is that both Git and GitHub are complicated bastards that are difficult to understand, especially Git. Git in its native form uses a command line interface (*just like MS-DOS and those text adventures from the eighties*). Both applications also use some fairly peculiar and non-intuitive terminology that gives the whole thing a pretty steep learning curve. It all takes a fair bit of hard work to understand properly.

There are three parts to it really:

- ① The first part is understanding the concepts of how Git manages version control (*the theory if you will*)
- ② The second part is installing it all and making it work (*and that's not as easy as it sounds*)
- ③ The third part is using something better than Git's command line to manage it all. I've chosen to use Brackets and some specialist extensions that make the whole thing much easier to use (*it moves away from the command line environment to something more modern*)

This chapter is concerned with the first of these three; I explain how Git works, how it operates as a version control system and what the hell all that peculiar terminology means.

I should say right from the start, that Git and GitHub are designed by geeks for geeks—and by God does it show. They belong to a bit of a club where those in the club don't really want to explain it to those outside; they say they do (because it's open source, left-wing and trendy) but they don't. There's lots of information but it's all designed to be a bit intimidating and to make you feel, well, stupid. It reminds me very much of the gramophone sketch by the **NOT THE NINE O'CLOCK NEWS** team. It's the same attitude.

So if you don't want a bag on your head, read on. I hope I've explained it better here.

2.1

What is version control?

Let's start with the basics—“*what is version control?*”

Version control is a mechanism for recording changes made to any files within a software project. It records all the changes, what files were affected by each change and a reason explaining why those changes were made. It also records who made the change and the time and date of the change.

It keeps a record of every change made within the project and allows any file that has been modified to be reverted back to a previous state. It means that if you change an image on a website, you can always go back and find the original version.

Version control systems do other things too, they can show the differences between two different versions of the software (even down to lines within a file), they allow multiple people to work on the project at the same time—even to work on the same file at the same time and they provide mechanisms for resolving conflicts (where two different people have modified the same section of a file for example).

Version control systems can be applied to any kind of project; it can be a website, a documentation project, a software application, engineering control system—anything at all, as long as it's a collection of files that can be stored on computer.

The version control system does not itself edit or modify any of the files within the project; it just records changes and, where it recognises a file type, is able to display the changes that have occurred to it.

The version control system does not care what software application is used to modify files within the project, it can be anything: text editor, word processor, file manager, graphics editor, specialist programming application &c. All it cares about is knowing a change has taken place and why.

Version control systems simply record any change made within a collection of files (the project), who made it, when it was made and the reason why. That's all.

2.1.1 Why use version control

Well, mainly for the reasons I illustrated at the start of this chapter, if you don't have a VCS, things get out of hand very quickly.

The proper reasons are:

- *Collaboration*, the VCS allows a team of people to work on the same project at the same time
- *Version storage*: the VCS manages all the versions of all the files, stores them, names them and can recover them
- *Tracking the changes*: the VCS records precisely what was changed and a reason must be given for the change
- *Restoring changes and regression paths*: the VCS allows individual files or groups of files to be restored to a previous version.

Looking at these in turn:

Collaboration

If your idea of working in collaboration is shouting across the office that you're working on Function Block 12 and no one else should touch it until you're finished; then you are probably doing it wrong.

With a version control system anybody within the project team should be able to work on any file at any time; even the same file¹.

The VCS should be clever enough to work out what has changed and allow all these changes to be merged back into a common version.

¹

I'm not sure I recommend this though—sometimes it's inevitable, but try to avoid it as much as possible.

The VCS should allow any member of the team to take a copy of the project, work on it locally on their machine and then recombine all the changes back into the main project.

Version storage

Storing a version of software isn't as easy as you think, look at my example at the start of this section.

When you store a version, it begs the question what do you actually store; do you store the whole project and every file within it (even though most of these files won't have changed) or do you just store the files you've modified (hoping that you've remembered to include all the files that you've changed).

If it's the first case, you will have as I did lots of unnecessary information and files stored on your system.

In the second case, it is very difficult to have a complete picture of the whole project at any given point.

This is what the VCS does for a living; it keeps track of the changes and is able to reconstruct the whole project at any given revision. It does it for you and provides the complete set of project files (or indeed just individual files) from any particular revision.

Tracking changes

This is important; it comes down to how revisions are numbered, and what is in each revision.

You would think numbering would be straight forward start at V01.00 and work up. Unfortunately, this generally only works when you have one person working on a project in a linear fashion.

What version does a file get if two people work on it at the same time? Perhaps files should be revised with the date and time? Again this falls down if you need two different versions of the same file (perhaps one for internal use and one for customers &c.).

It's fairly certain that unless it is a single person project, things will get out of hand quickly.

The next thing is knowing what changes are contained within each revision; in my example at the start, I recorded the changes made to each file in a header at the start of each file. And that is ok when there are only a small number of files, but by the end I had thousands of files and it meant I had to open each file to see when it had changed and what those changes were.

Even trying to keep a separate log (in say, a spreadsheet) doesn't work, you eventually forget to record something.

A VCS keeps a log of all the changes and ensures that no change is made without a corresponding record being made when the change is stored (it won't accept the change without a statement from the person making the change, *that doesn't stop the person entering complete bollocks, but it's a start*).

This log is visible and easy to navigate, it shows the changes made, when they were made, what was changed, who made the change and most importantly a statement explaining why the changes were made.

Restoring changes and regression paths

Being able to restore older versions of files or even the whole project makes it difficult to screw things up. It's always possible to go back to an earlier version and start again.

A regression path shows the changes that have been made and the order in which they were made, it allows a user to track the changes backwards and if a bug were introduced regress to an earlier version.

There is one final point:

Project backup

A VCS doesn't necessarily provide backup facilities, if it exists on a single machine and the VCS repository is deleted, then there will be no backup (unless the operator made one). However, if the VCS is distributed (like Git and GitHub) or indeed if more than one person is working on the project, then each team member will have a local copy of the project that can be restored. With Git and GitHub it is possible to keep an online copy of the project that is always up to date and can be copied to another PC.

Well, two final points:

VCS the downside

Yes, there is a downside—it all takes time—you have to think when to add a new version. I also spend some time composing the messages that get stored with the changes; *I like them to be right.*

I find that managing the requirements of the VCS spoils the flow of the work—interrupts the creative juices as it were.

In short, managing the VCS is time consuming and disruptive.

2.2

Version control with Git

Git is a version control system. It tracks changes in all the files contained within a project. The Git terminology for a project is a *repository*, but this terminology itself is a bit confusing, it looks like this (Figure 2.1).



Figure 2.1 Git repository structure

The *working copy*, *staging area* and the *local repository* are generally collectively referred to as a *repository* (sometimes just *repo*) or sometimes a *project* (*I tend to use project in this document*). All three of these are stored locally on the machine you are using.

The remote repository is a copy of the local repository that is stored elsewhere either on a server or, in the case of this publication, on the GitHub website.

A NOTE ON TERMINOLOGY

The terminology is a bit confusing with Git, there is a *repository* and this generally refers to the whole structure (the working copy, the staging area and the local repository). Using repository in this sense is referring to the whole *project* (a project is always held in a folder on your machine and it contains everything, all the files and folders that make up the project and the local repository and all the other bits that go with it).

Then there is the *local repository*, this is a database (*sort of*) of all the versions of every tracked file, all the metadata (such as change logs, change statements &c.) and everything else the VCS needs. The local repository lives in its own folder within the main project folder (with Git it lives in a `.git` folder, note the leading full stop).

Then there is the *remote repository*, this is a copy (*sort of*) of the local repository on some remote server, in our case it is on the GitHub website, but it could be a server in your office. I say *sort of* when calling it a copy, this is because it might hold more things than just what is within *your* local repository if other people are working on the same project—the remote repository is generally considered to be the master repository.

To avoid confusion I always refer to:

- | | |
|---|--|
| the <i>repository</i> or <i>project</i> | When referring to the whole project |
| the <i>local repository</i> | When referring to just the tracked files that have been committed to the local repository itself (not working copies or staged area files) |
| the <i>remote repository</i> | When referring to just the remote repository stored on GitHub (or at least a repository not on the local machine) |

Let's forget the remote repository for the time being and concentrate on just the local machine; we're left with Figure 2.2:

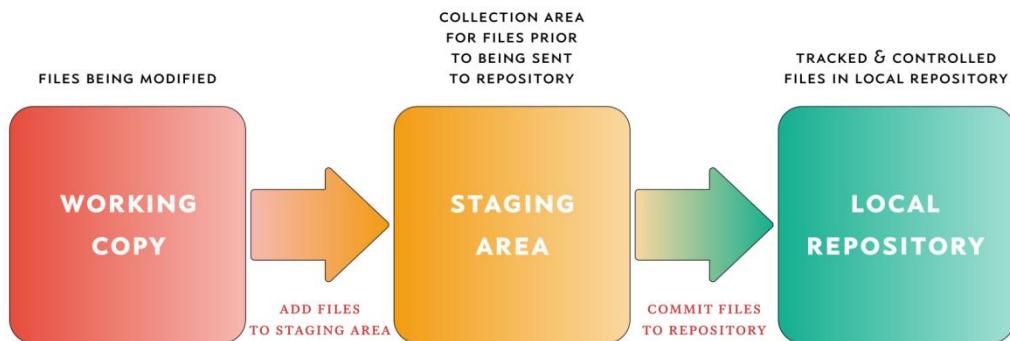


Figure 2.2 Git repository structure on a local machine

So how does this all work, well that's next.

2.2.1 Working, staging and the local repository—how it works

Let's start with a very simple example, a single page website with a picture; I'm going to call it `lab-01-website`.

On my machine I keep all my Git repositories under a single directory, that directory is on my D: drive and is called `2500 Git Projects`. Like this:

D:\2500 Git Projects

Yes I number my directories, yes it's embarrassing, but I am an engineer—we like to number things—I discuss this peccadillo further [HERE](#).

Next we need a directory to keep the repository we're creating in; this will be called `lab-01-website` and will live under the `2500 Git Projects` directory, thus:

D:\2500 Git Projects\lab-01-website

So far so good, we've created a new directory and it's completely empty.

Now we have to tell Git that it is a new repository.

Note: I'm not going to explain exactly how to do this yet, this is a high level discussion of how Git works and I want to explain it from the point of view of doing it through Brackets and we haven't covered this yet. The terms I use, like initialise, are valid though and you will see them in Brackets when we come to look at it there. I show some of the more common Git commands in the sidebar.

We do this by **initialising** the repository. Once initialised, the repository will contain a new hidden folder called `.git`. It is this folder (created by Git itself) that holds the local repository. On my system it looks like this, Figure 2.3:

Git command init

```
$ cd <path-to-folder>
$ git init
```

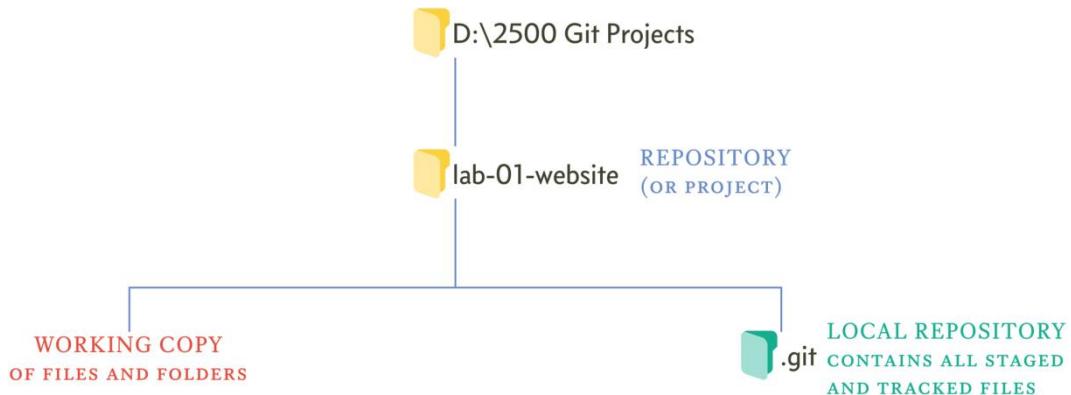


Figure 2.3 Git repository directory structure

This is now a Git repository, it hasn't got much in it, but it is ready to go.

THE .GIT FOLDER—A GOLDEN RULE

The `.git` folder is a hidden folder in the root directory of the repository. It contains all that is important: the local repository, any staged files, all the metadata associated with the repository (change records, logs &c.) and all the important bits for a tracked project.

**There is one golden rule concerning the `.git` folder:
DON'T FUCK WITH IT.**

Best thing is, don't even look inside it. If you delete it, you delete everything, if you change it, everything gets screwed up.

If we ignore the `.git` folder (*and this is always the best thing to do, see above*), everything else in the `lab-01-website` folder is our working copy, we can do what we like here: create files and directories, delete them modify them, rename and move them. Anything we like and at the minute Git will ignore everything we do.

So let's start, let's create a folder structure for our project and add some files to it. I want it to look like this:

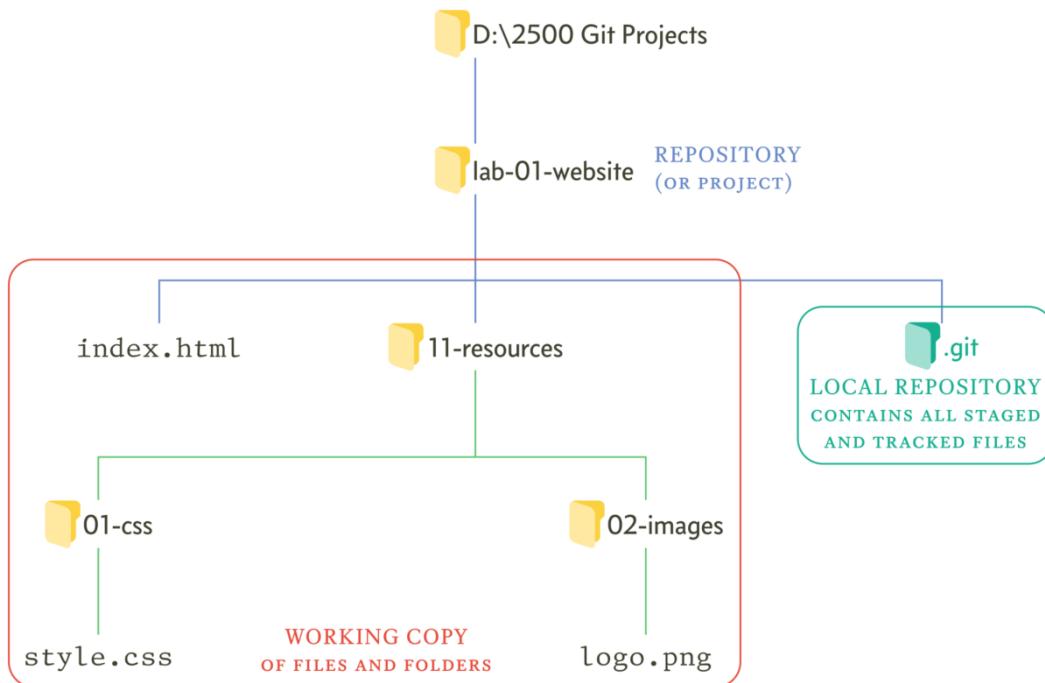


Figure 2.4 lab-01-website repository structure

Create the files and folders in whatever manner you like (Notepad and Windows Explorer will do), or download the finished article here [↓](#).

Having done all this, we are free to modify these files as much as we like. Git is ignoring the lot of them. It knows they're there, but it isn't tracking them in any way. In Git parlance, these are **untracked** files; it also knows that there are three of them:

- `index.html`
- `11-resources\01-css\style.css`
- `11-resources\02-image\logo.png`

Let's modify one of these files, `index.html`. I've added some very basic code (Code 2.1).

```

16<html lang="en">
165    <head>
166        <meta charset="utf-8">
167
168        <link rel="stylesheet" type="text/css" href="11-resources/01-css/style.css">
169
170        <title>PracticalSeries: Git Lab</title>
171    </head>
172
173    <body>
174        <h1>A Practical Series Website</h1>
175
176        <figure class="cover-fig">
177            
178        </figure>
179
180        <h3>A note by the author</h3>
181
182        <p>This is my second Practical Series publication—this one happened by accident
183        too. The first publication is all about building a website, you can see it here.
184        This publication came about because I wanted some sort of version control
185        mechanism for the first publication.</p>
186
187        </body>
188    </html>

```

Code 2.1 index.html

Git command add

```
$ git add index.html
```

Now let's say that we've finished `index.html` and we want to start tracking it, well the first thing to do is move it to the staging area. We do this with the Git command `add`.

Nothing has actually happened to the file, it's still there in the working area, however, a copy of the file exactly as it was at the time of the `add` has been placed in the staging area (we can't see this, it's inside the `.git` folder and we don't go there).

Once we add a file to the staging area, Git begins to track it, in Git parlance; it is now a `file to be committed`.

If we continued to modify the file in the working area, nothing would happen to the *staged* (files in the staging area are said to be *staged*) version of the file. If we wanted to overwrite the file in the staging area with a modified working copy, we would need to [add](#) it again.

Git still isn't properly tracking the `index.html` file; it knows we've done something to it, but we still haven't told it to put the file in its repository under full version control.

Now let's modify the `style.css` file, add the following to it:

```
STYLE.CSS
```

```
1 * {
2     margin: 0;
3     padding: 0;
4     box-sizing: border-box;
5     position: relative;
6 }
7
8 html {
9     background-color: #fbfaf6;          /* Set cream coloured page background */
10    color: #404030;
11    font-family: serif;
12    font-size: 26px;
13    text-rendering: optimizeLegibility;
14 }
15
16 body {
17     max-width: 1276px;
18     margin: 0 auto;
19     background-color: #fff;           /* make content area background white */
20     border-left: 1px solid #eddeded;
21     border-right: 1px solid #eddeded;
22 }
23
24 h1, h2, h3, h4, h5, h6 {           /* set standard headings */
25     font-family: sans-serif;
26     font-weight: normal;
27     font-size: 3rem;
28     padding: 2rem 5rem 2rem 5rem;
29 }
```

STYLE.CSS

```
30 h3 { font-size: 2.5rem; }
31
32 .cover-fig {
33     width: 50%; /* holder for cover image */
34     margin: 2rem auto;
35     padding: 0;
36 }
37 .cover-fig img {width: 100%;} /* format cover image */
38
39 p {
40     margin-bottom: 1.2rem; /* TEXT STYLE - paragraph */
41     padding: 0 5rem; /* *** THIS SETS PARAGRAPH SPACING *** */
42     line-height: 135%;
43 }
```

Code 2.2 style.css

The website actually looks like this, Figure 2.5 (*not bad for two minutes work*).

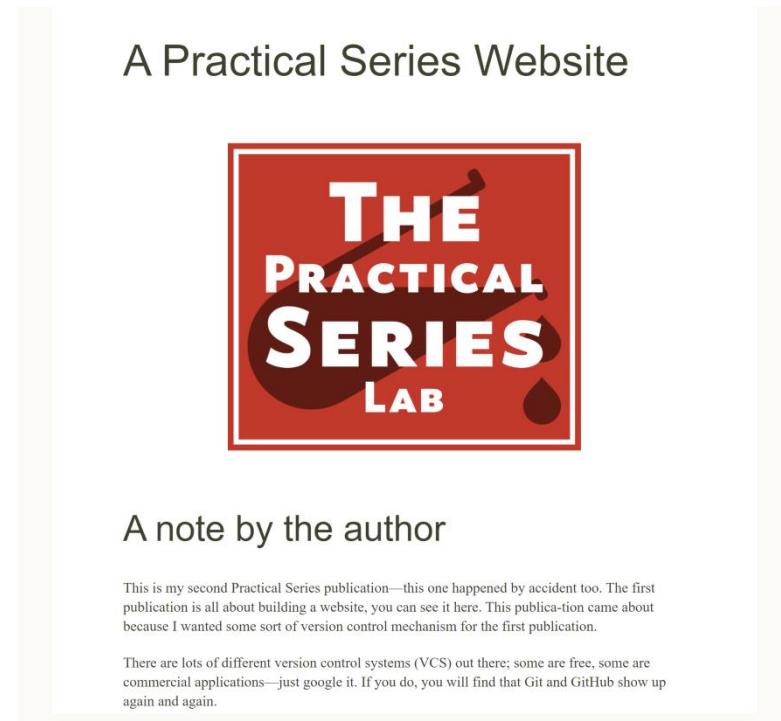


Figure 2.5 lab-01-website

Now, the `index.html` file is already in the staging area, the next thing to do is add the `style.css` and the `logo.png`. This is done with another `add` command.

Git command add all

```
$ git add
```

Our repository now looks like this:

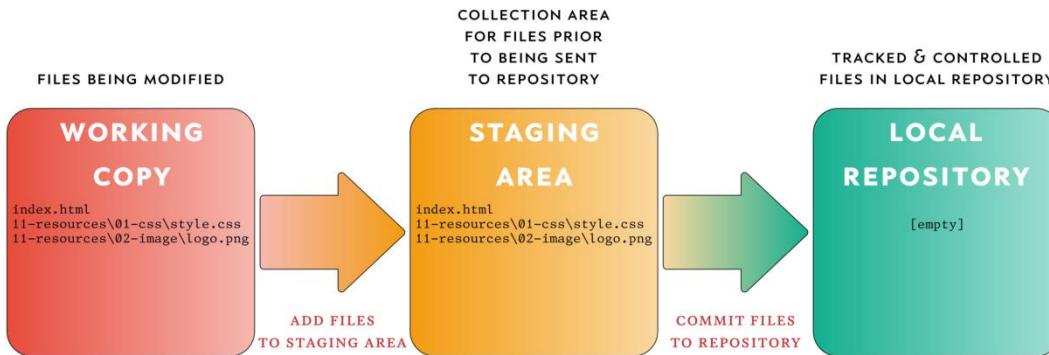


Figure 2.6 Repository with staged files

All the files are now in the staging area, the staging area acts as a collection area for files that we want to put into the local repository. It allows multiple files to be collected together and added to the local repository in one go. It means we can just have one message for the whole thing (we don't have to enter separate messages for each file).

Files are sent from the staging area to the local repository with a `commit` instruction. When a commit is executed, a specific message must be entered, in this case, the message is `initial commit` (with Git command line, the message can be entered as part of the command line syntax as shown in the sidebar).

Git command commit

```
$ git commit -m  
"initial commit"
```

Now we have this:

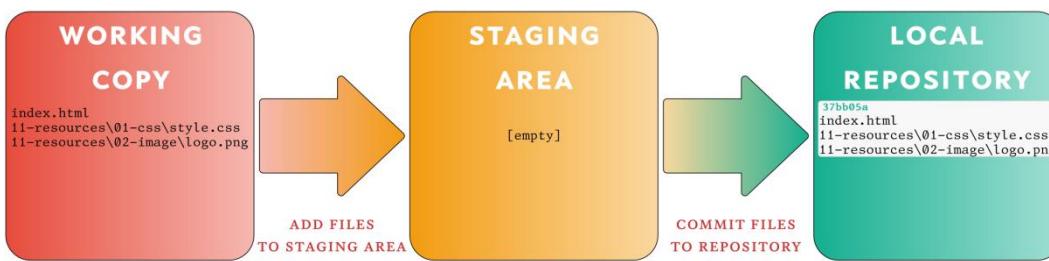


Figure 2.7 Repository with committed files

The staging area is empty (all its files have been *committed* to the local repository).

The working copy and the local repository now contain exactly the same versions of the file, in Git parlance, it would say `nothing to commit, working directory is clean`.

Let's make another modification to `index.html`. We'll delete the second paragraph:

```
INDEX.HTML
```

```
1 <html lang="en">                                <!-- Declare language -->
2   <head>                                         <!-- Start of head section -->
3     <meta charset="utf-8">                         <!-- Use unicode character set -->
4
5     <link rel="stylesheet" type="text/css" href="11-resources/01-css/style.css">
6
7     <title>PracticalSeries: Git Lab</title>
8   </head>
9
10  <body>
11    <h1>A Practical Series Website</h1>
12
13    <figure class="cover-fig">
14      
15    </figure>
16
17    <h3>A note by the author</h3>
18
19    <p>This is my second Practical Series publication--this one happened by accident
        too. The first publication is all about building a website, you can see it here.
        This publication came about because I wanted some sort of version control
        mechanism for the first publication.</p>
20
21  </body>
22 </html>
```

Code 2.3 Second modification to index.html

I've deleted lines 20 and 21 from the original file.

The website now looks like this:

A Practical Series Website



A note by the author

This is my second Practical Series publication—this one happened by accident too. The first publication is all about building a website, you can see it here. This publication came about because I wanted some sort of version control mechanism for the first publication.

Figure 2.8 Modified website

Let's also say that this is the only change we want to make. The next thing is to **add index.html** to the staging area:



Figure 2.9 Modified files staged

Git would now report the status of **index.html** as **modified** and **staged**.

And then we **commit** it with the message **index.html proof reading correction**.

And this time we have Figure 2.10:

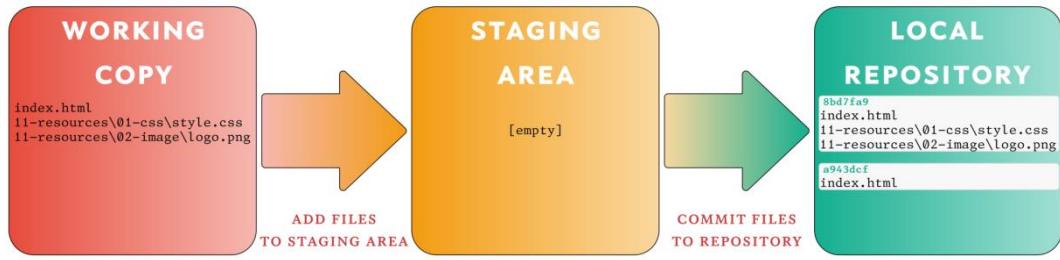


Figure 2.10 Second modification committed

There are now two changes stored in the local repository.

Get the idea?

2.2.2 Commit version numbers

You can see from this that we can keep changing things and we just get more entries in the local repository. Git knows what the whole project looks like at any point in time, in the above example. Git knows that there are three files in the project and it knows that the latest version of `style.css` and `logo.png` are from the first entry in the repository and that the latest version of `index.html` is in the second revision.

Calling these revisions first revision and second revision is ok if there is one person working on the project in a linear fashion. But Git is designed to cater for much more complicated arrangements—and it does it by numbering the changes in a very different way.

Look again at the local repository shown in Figure 2.10, I've reproduced it below:

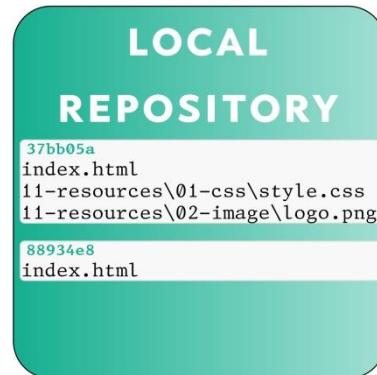


Figure 2.11 Local repository with two commits

It shows two entries, and each entry has a funny seven digit number (shown in green). The first commit has [\[37bb05a\]](#) and the second has [\[88934e8\]](#). These are effectively the version numbers. They are unique, but they are essentially just random numbers.

A NOTE ON COMMIT NUMBERS

These commit numbers are of course not random numbers. They are a checksum carried out of all the files in a commit, plus a header that contains other information (the commit numbers that immediately preceded this commit, plus some information about directory structures &c.).

A checksum is basically a function applied to the binary value of every byte in a file that gives a reproducible figure that can be used to check to see if two files are the same or to identify data corruption within a file.

The commit number used by Git is a checksum encode by using the SHA-1 algorithm (Secure Hash Algorithm 1). This produces a 20-byte (40 digit) hexadecimal number that uniquely identifies a commit. The commit number shown is just the first seven digits of the full commit number. This is usually enough to uniquely identify a commit (even on very large projects).

The first seven digits of a commit number gives 268 million unique values (the full 20 byte number has 1.5×10^{48} unique values; these values also only apply within a repository (two different repositories can have the same commit number, they don't interfere with each other)).

The chance of a duplicate 20 byte commit number is vanishingly small, even with just the first seven digits it won't happen on any project you are likely to be working on.

These commit numbers are referred to as either *hash* numbers or *SHA* (pronounced shar to rhyme with bar) numbers.

2.2.3 How to view commits

When a commit is made, there is effectively a snapshot of the complete project at that point. All the files in the project are available exactly as they were when the commit was made.

Any file in the project can be examined or reloaded from any commit (if it existed at the time of the commit). A series of commits form a regression path back through time, anything from the entire project to just a single file (or even part of a file) can be

restored to an earlier commit; similarly, once restored, it can be moved forward in time to a later commit.

To explain further, when I say that each commit is a snapshot of the entire project—I don’t mean that each commit stores every file in the project, it doesn’t. The information stored in a particular commit is the files that were added or modified by the commit, the commit also contains a link to any preceding commits and information about the directory structure at the time of the commit. This allows Git to determine exactly what the state of the entire project was at the time of the commit.

We don’t need to know the exact ins and outs of how Git manages its commits—as far as we are concerned **each commit is a snapshot of the entire project**.

2.2.4 Committing changes—best practice

The best practice is to commit often.

Some guidelines suggest you should only commit work that has reached some defined state (i.e. don’t commit half done work). *I don’t agree with this view.*

There is no harm in committing unfinished work (in fact there are benefits, *it keeps it safe*). Committing work because it’s home time is as good a reason as any for making a commit.

Where work is in progress and you make a commit, it’s often best to start your commit message with something that indicates this; I use `incremental build` this tells me that I made a commit, but that the particular commit was a work in progress.

Don’t be afraid to make commits (*don’t be afraid of commitment my son—well, just a little bit—like my old Dad used to say “don’t get married until you’re over thirty”. He also said “stay away from leggy blonds”—I screwed up on both counts*). The GitHub mantra is *commit early commit often—not sure what they have to say about leggy blonds, it doesn’t seem to feature in their documentation*.

2.3

Branching

In Git, branches are simply a group of commits (or even just a single commit).

Branches allow a project to be modified in sensible and controllable ways; let's think about the website example we did in the previous section: [lab-01-website](#).

If we look at what we have so far as a series of commits, we have this (Figure 2.12):

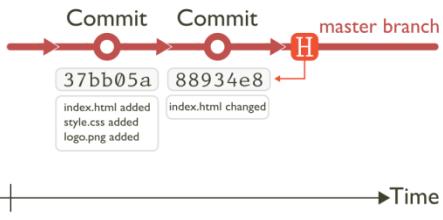


Figure 2.12 Commits on the master branch

We made our first commit [\[37bb05a\]](#) at a point in time, and sometime later we made our second commit [\[88934e8\]](#). These two commits already exist on a branch, the **master** branch. Now so far we haven't mentioned branches, but that doesn't mean we haven't been using one.

When we created ([initialised](#)) the repository in section 2.2.1, Git automatically created a branch for us (Git always has to have at least one branch). By default this branch is called the **master** branch; it is this branch that we have been working on all this time.

There is nothing special about the **master** branch, it is just the first branch created in the repository and by default Git always calls it “**master**”. It is perfectly possible to rename it or even delete it. But don't.

Virtually every project ever created with Git or GitHub will have a **master** branch. Not because it is special or particularly important, but because everybody just keeps it. In this sense it has had importance thrust upon it; the **master** branch has become the default branch for deployable code. This is just a convention you understand, there is absolutely nothing special about the **master** branch itself, but it is a convention I use and I explain it in the best practice section (§ 2.4.2 and Appendix B).

In Figure 2.12 I've added a **H** symbol; this is, in Git parlance, the **head**. The **head** (by default) points to the latest (most recent) commit on the currently active branch. In this case it is on the **master** branch (the only one we have) and points to commit **[88934e8]** which is the most recent commit.

So what do we do with branches?

Well let's suppose that our **index.html** page is finished and we're completely happy with it and its finished code is sitting at commit **[88934e8]**.

Now let's suppose that we want to add another page called **01-intro.html**. In my best practice model for branching (§ 2.4.2), the tested deployable code is always on the master branch. Any development work needs to be done on a new branch. In this case we will call it the **dev-01-intro**¹ branch.

Git command branch

```
$ git branch dev-01-intro
```

In Git we create a new branch with the **branch** command. All this does is create a new branch, nothing else has happened. We haven't moved to the new branch, we are still on the **master** branch and nothing has happened to any of the files in our working directory. Everything is just as it was.

Git command checkout

```
$ git checkout dev-01-intro
```

To change to the new branch we use the Git **checkout** command. This now switches us to the new branch. It looks like this, Figure 2.13:



Figure 2.13 A new branch in Git

Ok, so these all look like maps of the London underground.

Still nothing has really happened; we've activated the new branch so the **head** has moved to **dev-01-intro**. The latest commit is still **[88934e8]** and the **head** still points to this commit.

¹

I start development branches with **dev** for clarity; in later sections I abbreviate this to **d-**.

All this sort of makes sense; we've created a new branch, but we haven't changed any files or committed anything new to the repository. So let's make a change and see what happens.

We are now on the new **dev-01-branch**. Let's create a new **01-intro.html** file in the root folder (same place as **index.html**) and add the following code to it:

```
01-INTRO.HTML

1 <html lang="en">                                <!-- Declare language -->
2   <head>                                         <!-- Start of head section -->
3     <meta charset="utf-8">                         <!-- Use unicode character set -->
4
5     <link rel="stylesheet" type="text/css" href="11-resources/01-css/style.css">
6
7   <title>PracticalSeries: Git Lab - Introduction</title>
8 </head>
9
10 <body>
11   <h1>A Practical Series Website</h1>
12
13   <h3>Introduction</h3>
14
15   <p>This page is an introduction to the website and how to use it.</p>
16
17 </body>
18 </html>
```

Code 2.4 **dev-01-intro** branch – create new file

Add and commit the changes with the commit message New page **01-intro.html** added. In my case it is commit number [3541222].

Now we have Figure 2.14:

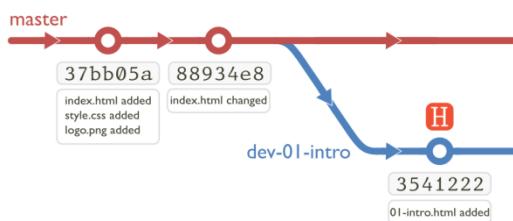


Figure 2.14 A new branch with first commit

The **head** has now moved to the new commit we just made—this is sensible, the **head** always points to the latest commit on the active branch.

I know what you're thinking, you're thinking what happens if I switch back to master?

Well let's have a look.

2.3.1 What happens when you switch branches

Pay attention now, this bit's important.

Our current branch **dev-01-intro** has an extra file in it, **01-intro.html**.

Git command checkout

```
$ git checkout master
```

This gives us Figure 2.15:

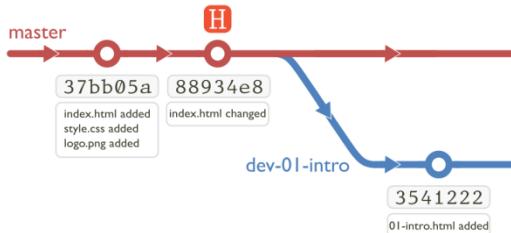


Figure 2.15 Switching branches

The **head** is now pointing to the last commit made on the **master** branch [**88934e8**] and there is no longer a file called **01-intro.html** in our working directory.

This is because the **master** branch doesn't know anything about the **dev-01-intro** branch or any commits that may have been made on it.

This is a comparison of what the two branches have in their working directories (Table 2.1):

MASTER BRANCH	DEV-01-INTRO BRANCH
<pre> graph LR master1[37bb05a
index.html added
style.css added
logo.png added] --> master2[88934e8
index.html changed] master2 --> dev01intro[dev-01-intro] dev01intro --> commit3[3541222
01-intro.html added] style="border: 1px solid black; padding: 2px; color: red;">H is positioned above commit 88934e8 </pre>	<pre> graph LR master1[37bb05a
index.html added
style.css added
logo.png added] --> master2[88934e8
index.html changed] master2 --> dev01intro[dev-01-intro] dev01intro --> commit3[3541222
01-intro.html added] style="border: 1px solid red; border-radius: 50%; width: 1em; height: 1em; display: inline-block; vertical-align: middle;"> is positioned above commit 3541222 </pre>
LATEST COMMIT (HEAD POSITION) [88934e8]	LATEST COMMIT (HEAD POSITION) [3541222]
WORKING FILES <code>index.html</code> <code>style.css</code> <code>logo.png</code>	WORKING FILES <code>index.html</code> <code>style.css</code> <code>logo.png</code> <code>01-intro-html</code>

Table 2.1 File and commit status on two different branches

So if we are on the `dev-01-intro` branch we have an extra file and if we switch back to `master` we lose it again.

What's the moral of all this? Well:

SWITCHING BRANCHES CHANGES THE FILES IN YOUR WORKING DIRECTORY

This is important, you'll remember it when you're on the wrong branch and you are wondering where all your files have gone. *Believe me you will be on the wrong branch more often than you think.*

2.3.2 Changing branches with uncommitted changes

Let's say we're back on the `dev-01-branch` and we've made some changes to `01-intro.html`, but we aren't ready to commit them to the local repository—i.e. we are in the process of modifying them and we're not ready to commit them yet.

Now let's say we want to go have a quick look on another branch. Well there is a problem, if we try to change branches with modifications that are in the working directory or staged area only, when we change branches, those modifications would be overwrit-

ten by whatever files are at the head of the branch we are changing to. We would lose the changes we had made in the previous branch *and that would be bad*.

Git won't let this happen:

**YOU CANNOT CHANGE BRANCHES IF YOU HAVE
UNCOMMITTED CHANGES IN YOUR WORKING COPY**

There is a way around this, it's called **stashing** files and it's essentially a temporary way of storing files that are a *work in progress* (WIP in Git terminology) without committing them—it essentially pushes the files onto a storage stack (*remember the Z80 push/pop assembler commands?—no?—never mind*). Neither Brackets nor GitHub support the stash and suffice it to say, I'm not a great fan of the stash either—*hence I don't cover it in these pages*.

We will always commit our changes before switching branches².

²

I'm going to say something here that some people won't agree with: "*I don't like stashing data*", I prefer to commit changes even if they're not finished—this is complete heresy to some people who think you only commit completed work—I'm not sure the GitHub people think this, GitHub doesn't have a stash facility and they recommend "*commit early and commit often*".

I don't stash unfinished work, I commit it. I commit it as an incremental build.
That's what engineers do.

2.4

More branches and merging

To demonstrate merging changes let's add another page to the `lab-01-website`. The `master` branch still has the latest version of the deployable code (although `dev-01-intro` is more advanced than the `master` branch, it is under development—new branches should always be based on the latest deployable code).

I want to add a new `02-about.html` page. Switch back to the `master` branch, create a new branch `dev-02-about` and switch to this new branch.

Now we have Figure 2.16:

Git commands

```
$ git checkout master  
$ git branch dev-02-about  
$ git checkout dev-02-about
```

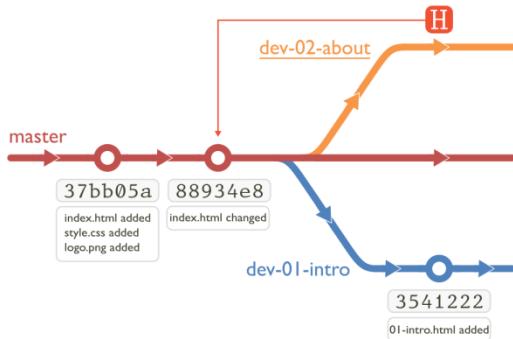


Figure 2.16 A third branch

This one's yellow (*Circle Line—to go with Central Line and Victoria*¹). The `head` is now on `dev-02-about`, but is pointing to the last commit on `master` [88934e8].

Time for another file let's add `02-about.html` in the root folder (same place as `index.html`) and add the following code to it:

¹

For anyone not from England, these are tube lines on the London Underground railway. It has an iconic map originally created by Mr Harry Beck, have a look [HERE](#).

02-ABOUT.HTML

```
1 <html lang="en">                                <!-- Declare language -->
2   <head>                                         <!-- Start of head section -->
3     <meta charset="utf-8">                      <!-- Use unicode character set -->
4
5     <link rel="stylesheet" type="text/css" href="11-resources/01-css/style.css">
6
7     <title>PracticalSeries: Git Lab - About us</title>
8   </head>
9
10  <body>
11    <h1>A Practical Series Website</h1>
12
13    <h3>About Us</h3>
14
15    <p>This page explains who we are and how we came to be doing this.</p>
16
17  </body>
18 </html>
```

Code 2.5 dev-02-about branch – create new file: 02-about.html

Add and commit the changes with the commit message New page 02-about.html added. In my case it is commit number [decb633].

Now we have Figure 2.17:

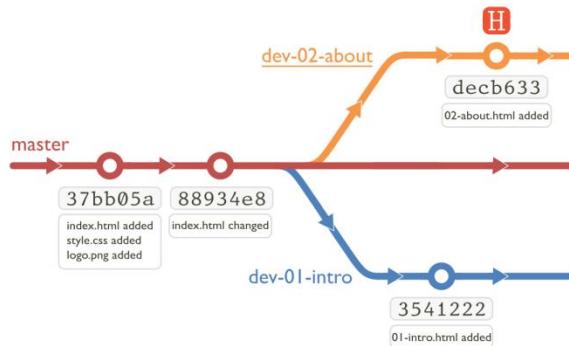


Figure 2.17 Second new branch with first commit

Git command checkout

```
$ git checkout dev-01-intro
```

Now switch to dev-01-intro (our first branch) and change the style.css file:

STYLE.CSS

```

1 * {
2   margin: 0;
3   padding: 0;
4   box-sizing: border-box;
5   position: relative;
6 }
7
8 html {
9   background-color: #fbfaf6;           /* Set cream coloured page background */
10  color: #404030;
11  font-family: serif;
12  font-size: 26px;
13  text-rendering: optimizeLegibility;
14 }
15
16 body {
17   max-width: 1276px;
18   margin: 0 auto;
19   background-color: #fff;             /* make content area background white */
20   border-left: 1px solid #eddeded;
21   border-right: 1px solid #eddeded;
22 }
23
24 h1, h2, h3, h4, h5, h6 {           /* set standard headings */
25   font-family: sans-serif;
26   font-weight: normal;
27   font-size: 3rem;
28   padding: 2rem 5rem 2rem 5rem;
29 }
30 h3 { font-size: 2.5rem; color: #4F81BD; }
31
32 .cover-fig {                      /* holder for cover image */
33   width: 50%;
34   margin: 2rem auto;
35   padding: 0;
36 }
37 .cover-fig img {width: 100%;}       /* format cover image */
38
39 p {                               /* TEXT STYLE - paragraph */
40   margin-bottom: 1.2rem;            /* *** THIS SETS PARAGRAPH SPACING *** */
41   padding: 0 5rem;
42   line-height: 135%;
43 }

```

Code 2.6 style.css modified in [dev-01-intro](#)

I've changed the colour of the `h3` element; it's blue like the Victoria Line. I've done this by modifying line 30; I've added a colour declaration (in bold):

```
30  h3 { font-size: 2.5rem; color: #4F81BD;}
```

It makes the page look like this if you're interested:



Figure 2.18 01-intro.html page

Let's commit the change with message `style.css – h3 colour changed to blue`. This gives me commit `[6454cdc]`. The London underground now looks like this:

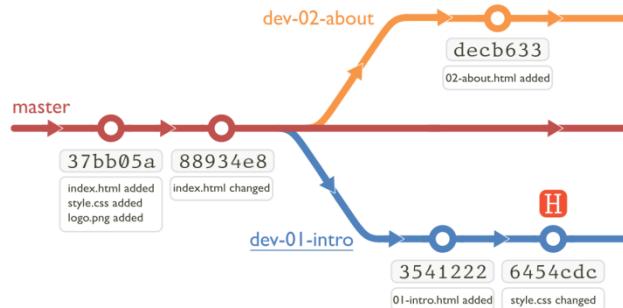


Figure 2.19 Second commit on dev-01-intro

Let's say that's it for `dev-01-intro`, we're happy with it and we want to deploy it, this means merging it back into the `master` branch. I want to do what is shown in Figure 2.20.

I want to put the two commits on the `dev-01-intro` branch onto the `master` branch.

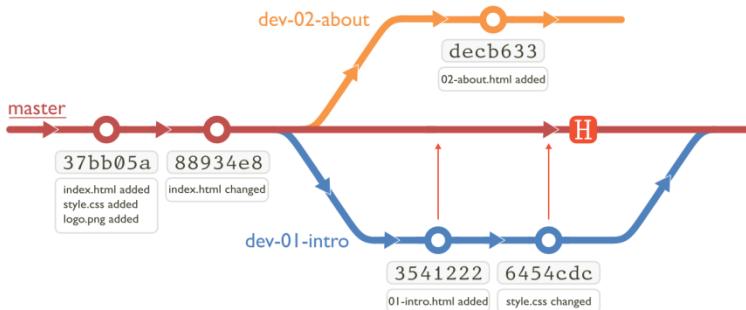


Figure 2.20 Merging one branch into another

And we can, we do it by changing to the branch that we want to receive the changes (the target branch), in this case it is the **master** branch. Then we use the **merge** command to bring in the changes from the source branch.

Git command merge

```
$ git checkout master
$ git merge dev-01-intro
```

This automatically creates a merge commit; this is just another commit with the message `Merge branch '<branch-name>'` in our case it is `Merge branch 'dev-01-intro'`. The merge commit is `[e659dad]`.

What we end up with is this:

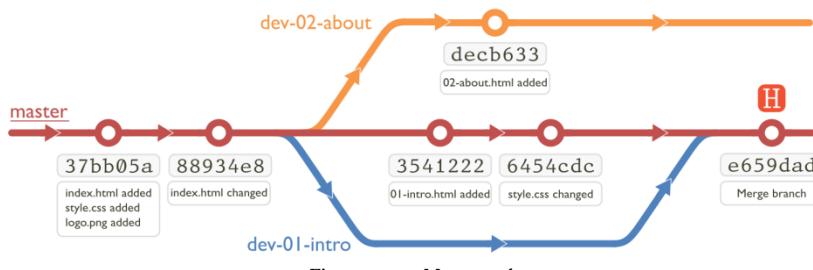


Figure 2.21 Merge result

The **dev-01-intro** branch is still there, but all its changes are now incorporated on to the **master** branch. We no longer need the **dev-01-intro** branch so we can delete it. *This is always the best thing to do with a merged branch.*

Git command `branch -d`

```
$ git branch -d dev-01-intro
```

The branch is deleted with the `branch -d` command. It leaves us with Figure 2.22.



Figure 2.22 Remove empty branch

Nearly done.

Git command `checkout`

```
$ git checkout dev-02-about
```

2.4.1 Conflict when merging

Finally, let's merge the `dev-02-about` branch, but first let's create a conflict situation.

Switch to the `dev-02-about` branch and modify the `style.css` file, this time we're going to modify exactly the same line as we did with the last change to `dev-01-intro`. But this time, we'll make the `h3` element red.

```
STYLE.CSS
```

```
1 * {  
2   margin: 0;  
3   padding: 0;  
4   box-sizing: border-box;  
5   position: relative;  
6 }  
...  
30 h3 { font-size: 2.5rem; color: #c0504d;}  
...
```

Code 2.7 conflicting modification to style.css

Again this is a modification to line 30 (in bold):

```
30 h3 { font-size: 2.5rem; color: #c0504d;}
```

This will conflict with line 30 on the `master` branch, this has:

```
30 h3 { font-size: 2.5rem; color: #4F81BD;}
```

The `dev-02-about` modification makes the `02-about.html` page look like this:



Figure 2.23 02-about.html page

Let's commit the change with message `style.css - h3 colour changed to red`. This gives me commit `[ce1e15a]`. The branches now look like this:

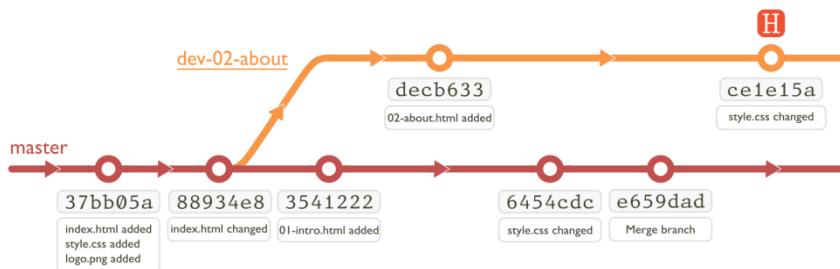


Figure 2.24 Second commit on `dev-02-about`

Now switch to the `master` branch and `merge` in the changes on `dev-02-about`.

This time when we try to merge, Git tells us there is a conflict and shows the two conflicting lines. Git is asking us to make a decision; it can't decide for itself what to do.

Git command `merge`

```
$ git checkout master  
$ git merge dev-02-about
```

We need to choose what we want to do, we can accept the `master` version, the `dev-02-about` version or we can choose to do something entirely different—we can also choose to abort the merge and think about things a bit longer.

In this case I'm going to choose the `master` version (the blue one). I do this by modifying the `style.css` file and committing the change. This will accept my resolution of the conflict and attach the commit message:

```
Merge branch 'dev-02-about'  
# Conflicts:  
# 11-resources/01-css/style.css
```

The commit for this merge is [5c9e772]. Our branches now look like this:

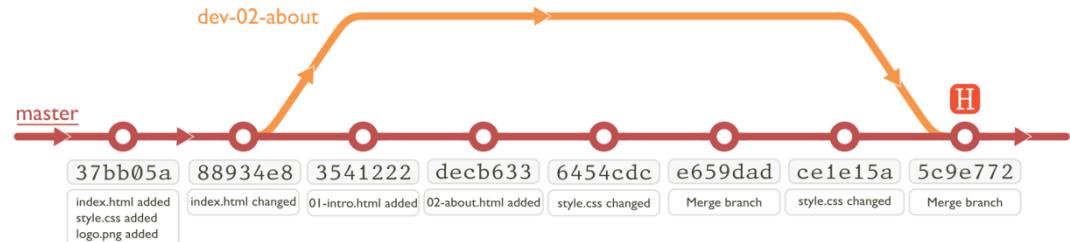


Figure 2.25 Merging the second branch after conflict resolution

Git command `branch -d`

```
$ git branch -d dev-02-about
```

Finally, the last thing to do is delete the empty branch. This just leaves the **master** branch with all the commits merged onto it. Like this Figure 2.26:



Figure 2.26 Final arrangement, all merged back to master branch

All the commits from the two branches that were created are now merged onto the master branch line.

In section 6 I go through this exact same operation using the Brackets-Git extension from within the Brackets text editor.

2.4.2 Branches and work flow—best practice

I tend to use a simple branching workflow (it is suitable for small teams). It is as follows:

I only have one main branch, the **master** branch. The **master** branch only ever contains deployable code. If development work takes place (for example developing a new web page) this always happens on a separate branch taken from the most recent commit point on the **master** branch (as we did with **dev-01-intro** and **dev-02-about**) in the previous examples. When work on a branch is complete and tested, the commits on that branch are merged back on to the **master** branch and the **development** branch is deleted.

Only merge complete and tested work back onto the **master** branch.

Each development branch must have a precise and limited scope (a single web page for example). It is better to have many branches each being limited to a particular topic rather than a single branch doing lots of things.

On any branch commit your work frequently and at any stage of development (even if it is a commit just because it's the end of the day and you're going home).

Appendix B has more details on how to manage this type of workflow and how to semantically name branches and tags for commit points.

2.5

Recovering an older commit with a reset

Recovering data from a previous commit point can be accomplished by using the [reset](#) command.

RESET—A WORD OF WARNING

Reset is the process of switching a project to a previous commit point.

**RESET IS A VERY CONFUSING ASPECT OF GIT,
IT'S A RIGHT BASTARD TO UNDERSTAND**

There are three types of [reset](#): *soft*, *mixed* and *hard*. The differences are to do with what happens to files in the working directory and staging area.

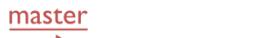
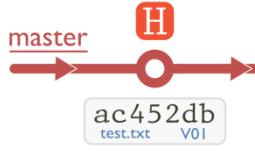
I will explain all of them, but before I do, I need to set up a project by way of an example that we can work with:

2.5.1 Setting up a project to explain the reset command

The easiest type of [reset](#) to understand is the *hard* reset; it resets a project back to an earlier commit and replaces all the files in the working directory with the files that were present at the time of the specified commit.

To explain this let's consider a simple project with just one file ([test.txt](#)). I will go through modifying the file and creating commits in a step-by-step way to demonstrate exactly what happens with a reset.

Let's assume we have a new project and we've just created a new file called [test.txt](#). This file is present in the working area only, we haven't added or committed the file yet (step 1). Next add the file to the staging area (step 2) and then commit it to the local repository (step 3):

ACTION	RESULT						
STEP 1	<p>Commit path</p>  <p>In a new repository create the file <code>test.txt</code> and add some text to it. This is now in the working area and is version V01 of the file.</p> <table border="1" data-bbox="504 557 938 624"> <tr> <th>Working</th> <th>Staged</th> <th>Head</th> </tr> <tr> <td>test.txt V01</td> <td></td> <td></td> </tr> </table>	Working	Staged	Head	test.txt V01		
Working	Staged	Head					
test.txt V01							
STEP 2	<p>Commit path</p>  <p>Add the file to put it in the staging area.</p> <table border="1" data-bbox="504 900 938 968"> <tr> <th>Working</th> <th>Staged</th> <th>Head</th> </tr> <tr> <td>test.txt V01</td> <td>test.txt V01</td> <td></td> </tr> </table>	Working	Staged	Head	test.txt V01	test.txt V01	
Working	Staged	Head					
test.txt V01	test.txt V01						
STEP 3	<p>Commit path</p>  <p>Now commit the file to the local repository.</p> <table border="1" data-bbox="504 1253 938 1320"> <tr> <th>Working</th> <th>Staged</th> <th>Head</th> </tr> <tr> <td>test.txt V01</td> <td>test.txt V01</td> <td>test.txt V01</td> </tr> </table>	Working	Staged	Head	test.txt V01	test.txt V01	test.txt V01
Working	Staged	Head					
test.txt V01	test.txt V01	test.txt V01					

This gives our first commit [\[ac452db\]](#).

Now let's modify the `test.txt` file (making it V02) and repeat the **add** and **commit** process to give our second commit [\[5493a7c\]](#) (step 4).

ACTION	RESULT						
STEP 4 Modify <code>text.txt</code> . It is now V02. Now add and commit the file to the local repository.	<p>Commit path</p> <p>The diagram shows a horizontal timeline with three circular nodes connected by arrows. The first node is labeled "master" above an arrow pointing right. Below it is a box containing "ac452db" and "test.txt V01". The second node is labeled "5493a7c" and "test.txt V02". The third node is labeled "H" (Head) above an arrow pointing right. Below it is a box containing "22af5b5" and "test.txt V03".</p> <p>Stored files</p> <table border="1"><thead><tr><th>Working</th><th>Staged</th><th>commit</th></tr></thead><tbody><tr><td>test.txt V02</td><td>test.txt V02</td><td>test.txt V02</td></tr></tbody></table>	Working	Staged	commit	test.txt V02	test.txt V02	test.txt V02
Working	Staged	commit					
test.txt V02	test.txt V02	test.txt V02					

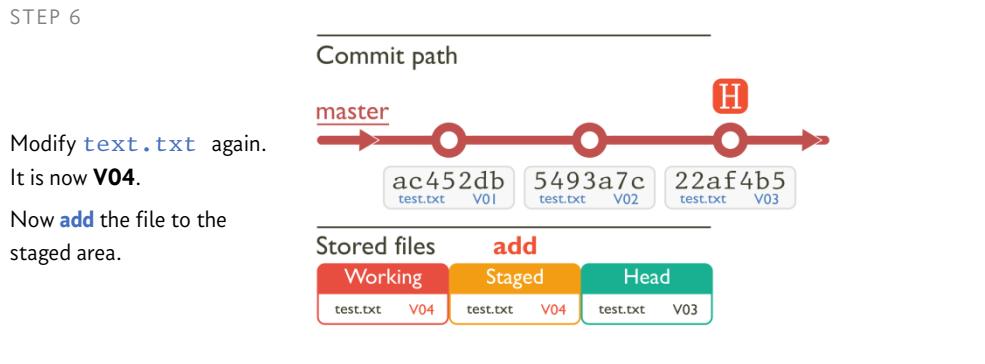
Now we do it all again for a third **commit** [22af5b5] and a file at V03 (step 5).

STEP 5	RESULT						
 Modify <code>text.txt</code> again. It is now V03. Now add and commit the file to the local repository to give 3 commits in total.	<p>Commit path</p> <p>The diagram shows a horizontal timeline with four circular nodes connected by arrows. The first node is labeled "master" above an arrow pointing right. Below it is a box containing "ac452db" and "test.txt V01". The second node is labeled "5493a7c" and "test.txt V02". The third node is labeled "22af5b5" and "test.txt V03". The fourth node is also labeled "22af5b5" and "test.txt V03". The Head is at the fourth node. Below it is a box containing "22af5b5" and "test.txt V03".</p> <p>Stored files</p> <table border="1"><thead><tr><th>Working</th><th>Staged</th><th>commit</th></tr></thead><tbody><tr><td>test.txt V03</td><td>test.txt V03</td><td>test.txt V03</td></tr></tbody></table>	Working	Staged	commit	test.txt V03	test.txt V03	test.txt V03
Working	Staged	commit					
test.txt V03	test.txt V03	test.txt V03					

Note: Some of you may be wondering why I show files in the staged area after a commit, when in Figure 2.7 to Figure 2.10 I showed it as empty. In practice, the staged area does still hold the files (I showed it as empty to make the explanation easier to understand).

The staged area always holds the files, but if they are the same as the committed files, Git considers it to be empty—it isn't, but there is nothing in there that requires action.

Finally, let's modify the working copy of `test.txt` (making it V04) and **add** it to the staging area (step 6). Do not **commit** these changes; this is work in progress and I will use it to show how the different types of reset work. We have this as the final step:



I've shown the modified and staged file version in red.

Now let's look at what a reset does.

RESET—A REASSURING POINT

The reset process can be destructive; it can overwrite data (it's one of the few things that Git does that can lose your data). That said:

A reset will never change or delete committed data.

Committed data is always safe, a commit point will never be deleted and you can always go back to it.

The worst a reset can do is overwrite changes in working or staged files.

I will start with a hard reset; like a hard Brexit, it's the option that makes most sense.

2.5.2 A hard reset

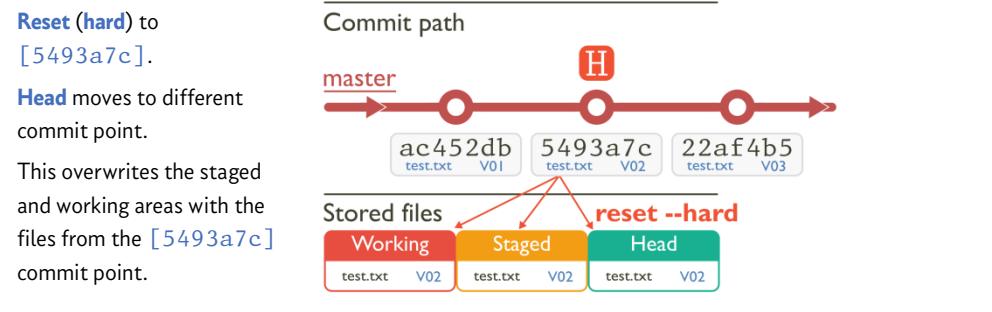
Ok, we now have a project with three commits in it. The **head** is on the **master** branch and is at the latest of those commits [\[22af5b5\]](#). All well and good, this is just exactly the same as the example we worked through in the previous section.

We've also got a modified working copy file that has also been staged but not committed (step 6).

A reset (any reset) moves the head to a different commit point (a bit like switching branches). **A hard reset overwrites the files in the working and staged areas with the files from the commit point that we are resetting to.**

Git command `reset`

```
$ git reset 5493a7c  
--hard
```



At this point we have lost the V04 changes to `test.txt` that were in the working and staged areas (step 5).

Note: This hard reset is a bit like changing branches; however, Git won't let you change branches if you have modified files in the working or staged area (because they would be lost, § 2.3.2). Git has no such qualms about a reset, you will lose any uncommitted changes that are in the working or staged areas when you do a hard reset.

As with branches, best thing to do is commit any modifications prior to a reset.

Git command `reset`

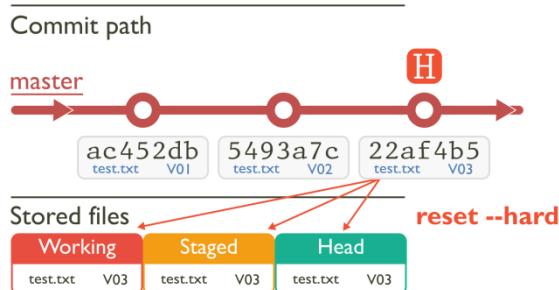
```
$ git reset 22af4b5  
--hard
```

If we now do another reset (without modifying anything) back to the most recent commit point [\[22af4b5\]](#) we just end up back where we were at step 5:

A second **Reset (hard)** to [22af4b5].

Moves the **Head** back to the most recent commit point.

This restores the project to the step 5 state above.



2.5.3 Making changes after a reset

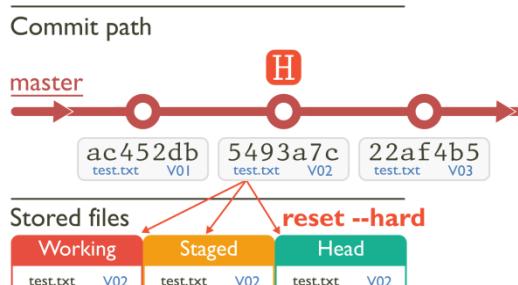
This is a bit like going back in time and accidentally killing your own father—it leaves things hanging.

Currently we are at step 5 and our commit history is:

[22af4b5]	Version V03
[5493a7c]	Version V02
[ac452db]	Version V01

Now **hard reset** back to commit point [5493a7c] just like we did in the previous section—we get exactly the same result:

Reset (hard) to [5493a7c].
Head move to different commit point.
This overwrites the staged and working areas with the files from the [5493a7c] commit point.



The history shows:

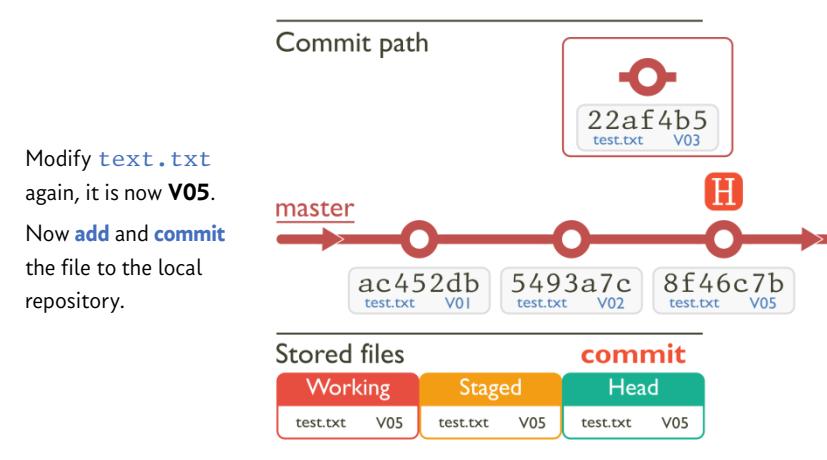
[5493a7c]	Version V02
[ac452db]	Version V01

It's not showing the [22af4b5] commit point.

This makes sense, the history only includes things up to the current **head** point.

We can still move back to the [22af4b5] commit point because we did in the previous section. The problem is *what if we make changes at this earlier commit point?*

Do it, modify `test.txt` this time we'll call it V05—*like the shampoo (I used V04 before remember)*—commit the changes, this is commit [8f46c7b].



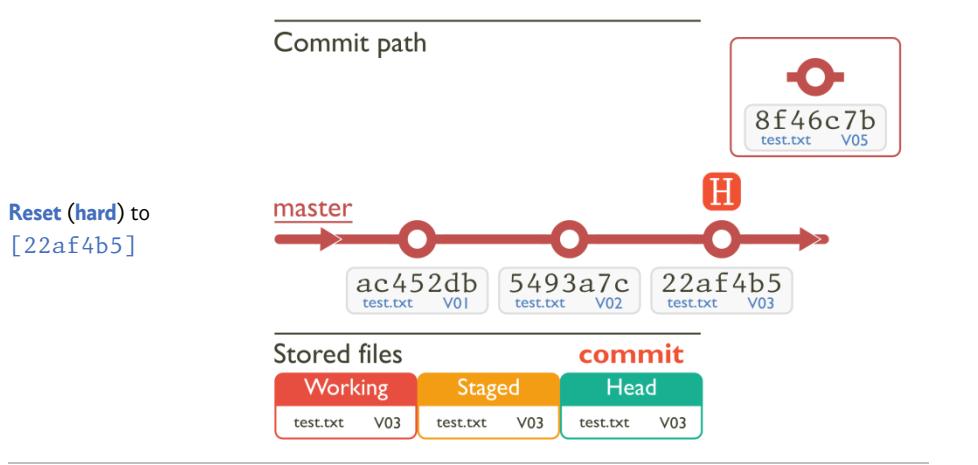
And now if we look at the history:

[8f46c7b]	Version V05
[5493a7c]	Version V02
[ac452db]	Version V01

Commit point [22af4b5] isn't there. A bit like Stalin, we've rewritten history.

The missing commit point is still there¹, it no longer fits into the chain of commits—it's just floating around in the Git repository.

It's perfectly possible to switch to the missing commit point [22af4b5]. If we were to reset to it [22af4b5] we would have:



The commit history being:

[22af4b5]	Version V03
[5493a7c]	Version V02
[ac452db]	Version V01

Confusing isn't it?

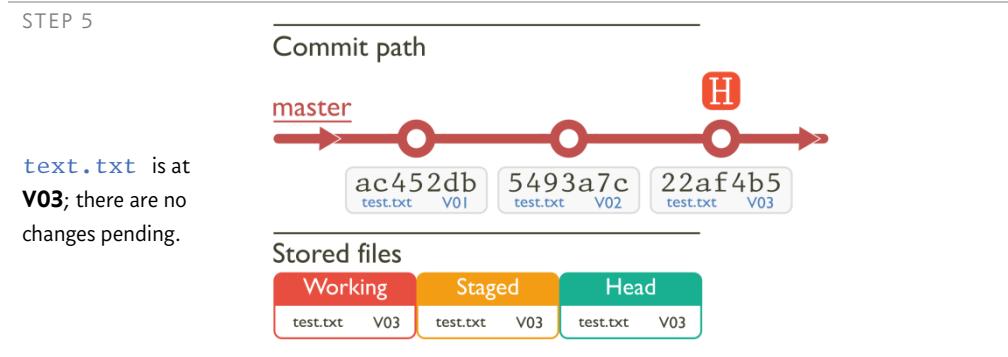
¹

You need to be careful with these floating commit points, they are deleted when the database is cleaned up (*pruned* in Git terminology—it's gardening, like branches).

2.5.4 Using resets—best practice

I don't like the idea of resetting to a previous commit point and then progressing on from there as if nothing had happened (effectively bypassing the later commits).

Let's assume that we've progressed from step 1 to step 5 in the previous example and we haven't done any resets (i.e. commit [8f46c7b] never happened). We have this:

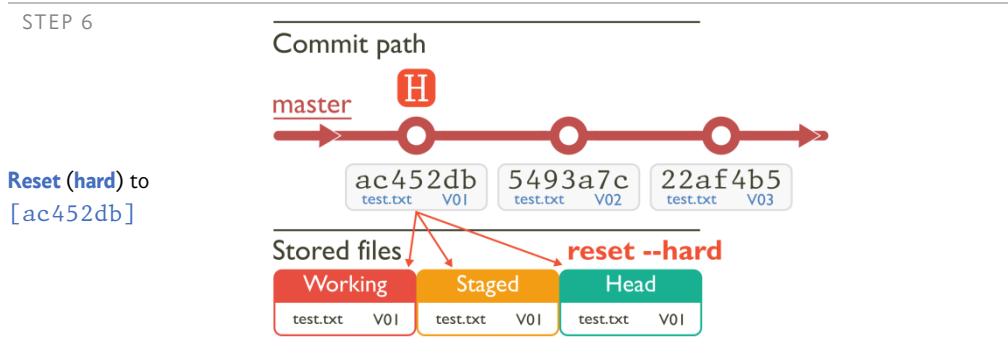


Let's also assume that there is some problem with **test.txt** and we realise that the version we need is actually version 01 with some slight modification. The best procedure to recover V01, modify it and commit the modifications is as follows:

Git command `reset`

```
$ git reset ac452db  
--hard
```

Commit or discard any pending changes (in our case there aren't any) and reset the project back to the V01 commit point with a **hard reset** [ac452db].

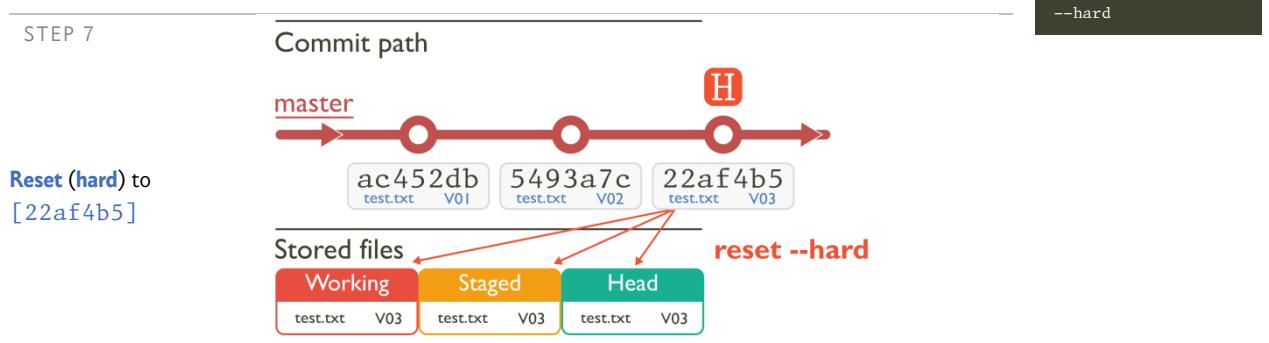


This places the V01 file in the working area.

Now either open the `test.txt` file and copy the contents or just copy the entire file to the clipboard.

We haven't changed any files at this point (just copied data).

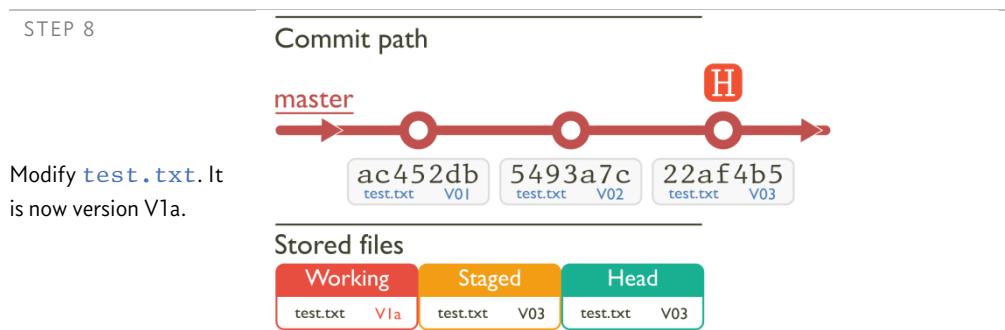
Now **hard reset** back to the latest commit in the chain `[22af4b5]`.



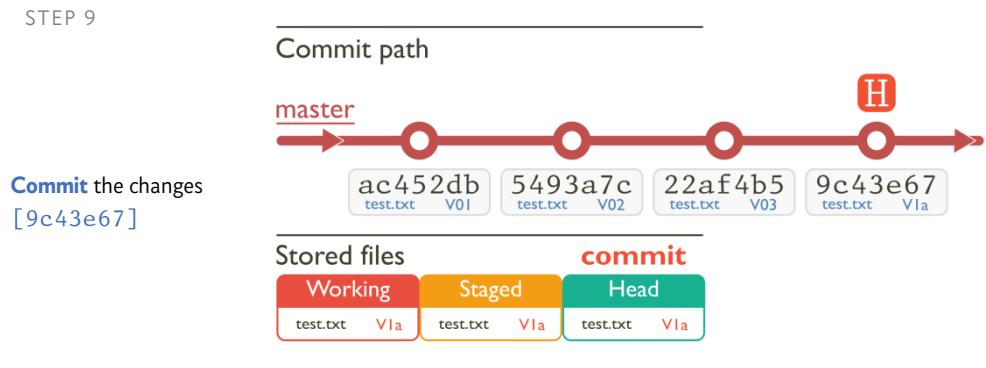
Next either open up the latest file, the V03 file, that is in the working area, delete everything and paste in the V01 copied contents from the clipboard, or just overwrite the file with the copied version.

Either way, the `test.txt` file in the working area now contains the V01 version that we copied from the step 6 **hard reset**.

Make any other modifications that are required and save the file (we will call this V1a for simplicity).



Now **add** and **commit** the changes with the commit message `test.txt via – based on v01 with modifications`. In this case it is commit [9c43e67].



The commit history for this is:

[9c43e67]	test.txt v1a – based on v01 with modifications
[22af4b5]	Version V03
[5493a7c]	Version V02
[ac452db]	Version V01

This I think keeps things clearer.

From an engineering point of view, it is not acceptable to just move the project back a couple of notches and then carry on as if nothing had happened—it breaks the traceability—and that's when they send you to gaol—it leaves the question: *what happens to the commit points that got skipped?* They're still there, someone could use them, it adds to the confusion—just record what you did to put it right and move on, even if that means copying files and data from an earlier commit point.

Just explain what you did and why.

It's the engineer's code—use it wisely young Skywalker.

2.5.5 A mixed reset

Let me say now, a **hard reset** is the only one to use.

Boy am I goanna get some emails—they'll all be from Linux people, they usually are².

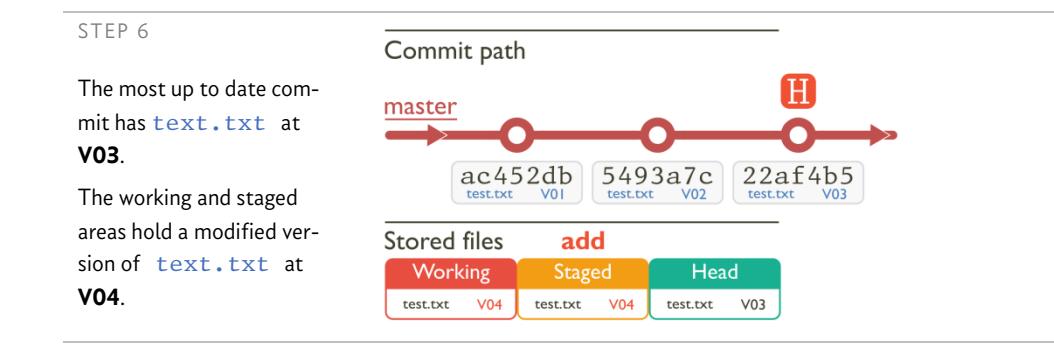
My rules for resetting:

- ① Always commit or delete modified files before resetting
- ② Always use a **hard reset**
- ③ Resetting a project should only be done to view the files as they were at that point in time, never to modify them
- ④ Never modify files at any commit point other than the latest (most recent) commit (on a given branch)

That said I will explain what the other types of reset do—I just haven't figured out what they are for.

The first one is a **mixed reset**. This is the default type of reset applied by Git.

Set up the project as it was in § 2.5.1 step 6. It looks like this:



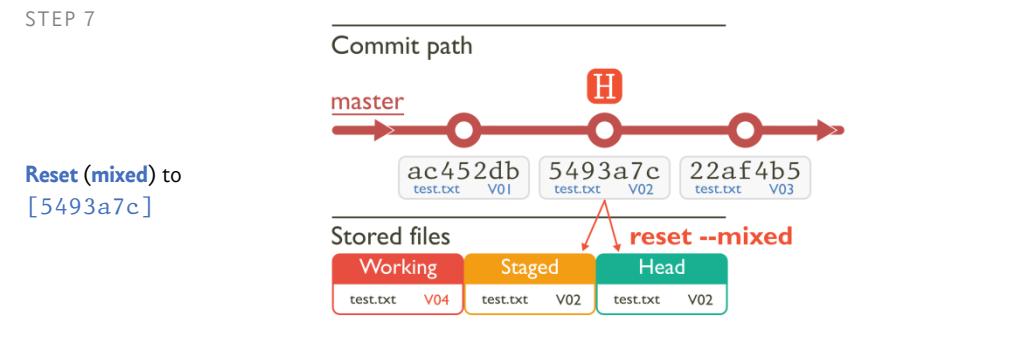
²

In my experience and judging by my inbox, Linux people are a bit like Jeremy Corbyn's supporters: entirely convinced of their own moral superiority and completely dismissive of any other argument. They have a stifling certitude, an implacable self-righteousness and they are always willing to be offended.

Git command `reset`

```
$ git reset 5493a7c  
--mixed
```

The **mixed reset** is a non-destructive reset. It does not overwrite the working copy but it does change the staging area and (obviously) the `head`. If we `reset (mixed)` to the V02 commit point [5493a7c] we would have:



It leaves the working file `test.txt` exactly as it was at version V04. It means we could stage and commit the V04 modification to a new commit based on the V02 commit point [5493a7c] we've just reset to if we wanted to. *But we wouldn't do that would we?*

Sound confusing?

Well it is. I appreciate it is non-destructive; it hasn't overwritten the V04 file in the working area. It's also pretty useless because I can't see any of the V02 files. None of the V02 files have been put in the working area and the working files are the only files that are visible or editable.

The `test.txt` file will be flagged as modified but not staged.

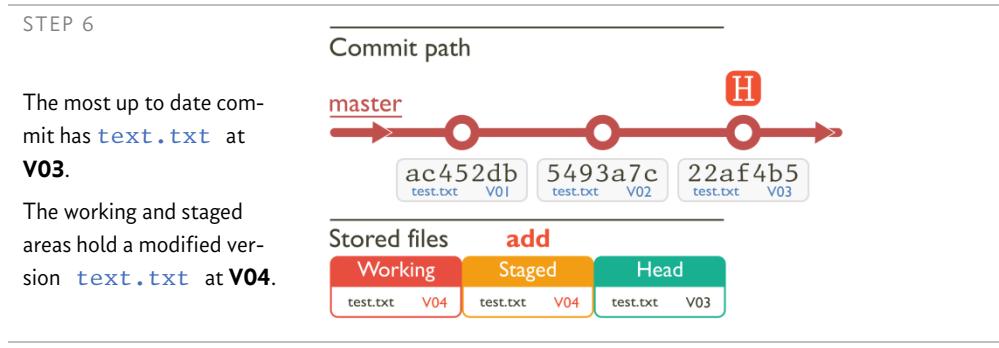
Again, I don't see a use for it. *I'll just wait for the Linux people to stop shouting and explain it to me.*

2.5.6 A soft reset

I refer you to the comments I made about the mixed reset in the previous section.

A soft rest does even less; all it does is move the **head**.

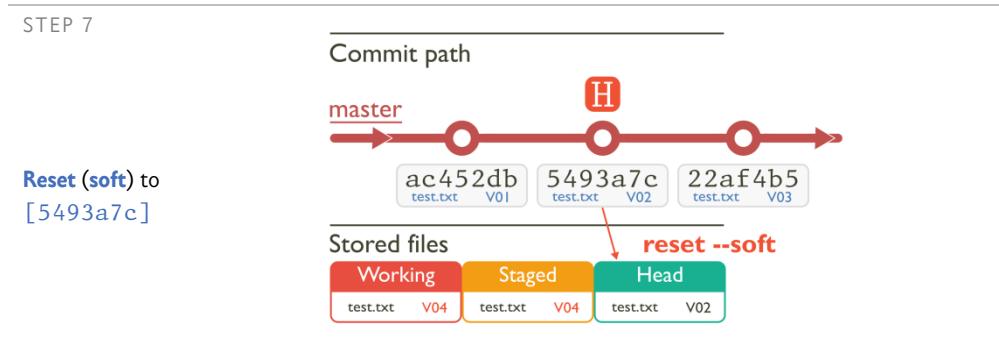
Again set up the project as it was in § 2.5.1 step 6. It looks like this:



Git command `reset`

```
$ git reset 5493a7c  
--soft
```

The **soft reset** is again a non-destructive reset. It does not overwrite the working copy or staged area; all it does is move the **head**. If we **reset (soft)** to the V02 commit point [5493a7c] we would have:



It leaves the working file `test.txt` exactly as it was at version V04 in both the working and staged area. It means we could commit the V04 modification to a new commit based on the V02 commit [5493a7c] we've just reset to if we wanted to.

The `test.txt` file will be flagged as modified and staged.

Again, I don't see a use for it.

2.5.7 Reset, a summary

My rules and notes for resetting are:

- ① A reset will never change or delete any committed data or commit points
- ② Always make a note of the latest commit number (hash)³
- ③ Always commit modified files before resetting
- ④ **Always use a hard reset**
- ⑤ Resetting a project should only be done to view the files as they were at that point in time, never to modify them
- ⑥ It's ok to copy files from an earlier commit (following a hard reset) and paste them into the latest commit point (ensure you explain it in the commit message)
- ⑦ Never modify files at a commit other than the latest (most recent) commit (on a given branch)

³

I say this because if you reset to an earlier commit, the later ones don't show up in the history and this makes it harder to get back to a later commit if you don't know its commit number. There are ways around this, see § 7.2.1).

2.6 Tagging

Tagging is used to give a meaningful name to a commit point.

Tags are usually used to indicate a particular state or release of a project.

The following are the rules for a tag:

- ① Each tag must be unique; two separate commits cannot have the same tag
- ② Tags can be any combination of letters and numbers (but not spaces) see § 2.6.4 for specific restrictions
- ③ Tags are not case sensitive
- ④ A particular commit can have more than one tag

Git supports two types of tag: a *lightweight* tag and an *annotated* tag.

2.6.1 Lightweight tags

Lightweight tags are just simple labels that are attached to particular commit. The lightweight tag does not support an associated message and does not record any user information about who created it. It is literally just a label attached to a commit.

Lightweight tags are created using the `tag` command within Git. If no commit point is specified, the tag will be applied to the current `head` commit.

Note: *Lightweight tags are the only tags supported by Brackets-Git.*

Git command tag

```
$ git tag v02 5493a7c
```

Git command `tag -a`

```
$ git tag -a v02  
5493a7c 'V02 release'
```

2.6.2 Annotated tags

Annotated tags do everything that lightweight tags do, but they contain additional information, they include a message (*hence annotated*) that is similar to a commit message, it includes information about the tag, e.g. `first public release`.

Annotated tags also record the username and email of the person creating the tag as well as the date of its creation.

Annotated tags are created using the `tag -a` command within Git. If no commit point is specified, the tag will be applied to the current `head` commit.

Annotated tags are stored as full objects in the Git database; that is to say they are part of the revision history.

2.6.3 Using tags

Once created a tag can be used in place of the commit number (*hash or SHA*). This is true of both lightweight and annotated tags.

In the above example, the commit point `[5493a7c]` has been given the tag name `V02`.

The Git command to do a hard reset to this commit point is `git reset 5493a7c --hard`. Now that the commit point has a tag, the command `git reset --hard V02` can be used instead.

I.e. when a commit is tagged, the tag name can be used to replace the commit number (*note the change to the order, the tag is at the end, the commit number is in the middle*).

2.6.4 Tag naming restrictions

Tags can be called pretty much anything you like; they can't contain spaces though; this is the full list of restrictions:

TAG NAMING RESTRICTIONS

Tags cannot begin or end with, or contain multiple consecutive `/` characters.

They cannot contain any of the following characters `\`, `?`, `~`, `^`, `:`, `*`, `[`, `@`.

They cannot contain a space.

They cannot end with a `.` or have two consecutive `..` anywhere within them.

In terms of my own conventions, I make the following restrictions:

- ① Use a dash instead of spaces
- ② Only use the characters [a-z], [A-Z], the numbers [0-9] and the dash/hyphen [-]

This is just my own preference you understand. Although it is a very safe one to use.

2.7

Ignoring files with `.gitignore`

Generally, there are files in a project that you want Git to ignore completely, typically these are files created by the operating system (`.DS_Store` on Macs, `Thumbs.db` on Windows &c.), temporary files, automatically generated log files &c.

With these sorts of files Git would normally flag them as **untracked** and would keep showing you them every time you did a status—eventually they would be accidentally added and committed to the local repository.

Git uses a file called `.gitignore` to tell it what files should be ignored completely. Such files are never mentioned in any status, never added to the staging area, or committed to the local repository.

The `.gitignore` file should be in the root of the working area of the project. In the `lab-01-website` we set up in § 2.2.1 it would be here:

`\lab-01-website\.\gitignore`

Same place as the `index.html` file.

The `.gitignore` file is a simple text file (*note it does not have an extension*) that lists the files to ignore. It accepts single and multiple character wildcards (?) and (*) respectively). It also accepts comments; anything after a # character to the end of the line is treated as a comment.

Here are some typical examples:

```
.GITIGNORE

# a comment everything after the # is treated as a comment
myfile.txt          # ignore the specific file myfile.txt wherever it occurs
*.log               # all files with a .log extension are ignored
!my.log              # BUT do track my.log (in exception to the previous line)
 myfile.txt          # ignore myfile.txt in the root, but not in any other folder
                      # or subfolder
 mywork/             # ignore anything in the mywork folder
 doc/*.txt           # ignore any .txt file in the doc folder, but not in any
                      # other folder or subfolder
```

Code 2.8 .gitignore typicals

Generally, both GitHub and Brackets-Git will create a [.gitignore](#) file by default when creating ([initialising](#)) a repository.

2.8 The remote repository

A remote repository is generally a copy of a local repository that exists on a server somewhere that other people also have access to.

There are two points to having a remote repository:

- ① It acts as a backup for the local repository
- ② It allows a team of people to collaborate on a Git project

2.8.1 Local and remote repositories: the differences

To state the bleeding obvious—local repositories are, well, local; they live on a specific computer used by a single project team member. The local repository is for the use of that specific person on that specific machine.

A remote repository on the other hand is hosted on a server somewhere and can be accessed by all the team members.

Local repositories have associated working and staging areas where work is developed staged and ultimately committed to the local repository.

Remote repositories do not have working and staging areas; it is just the bare repository.

2.8.2 Where does a remote repository live?

The answer to this is it could be on a networked server on a local network (within say, the office); or it could be stored on an internet based server (such as GitHub or Bitbucket).

In this publication I only consider a GitHub remote repository; this is the most common remote repository and it works very well. I explain how to use it in great detail in section §§ 9 and 10.

2.9

Moving data between local and remote repositories

Lots more London Underground maps.

Consider the [lab-01-website](#) we had at the end of section 2.4 (Figure 2.26) it looked like this:



Figure 2.27 lab-01-website—Final arrangement

This is the project currently stored in the local repository on our machine.

Now let's say we want to copy all this to a remote repository on GitHub (I explain how to set up a GitHub account in section 4.1).

We must first create an empty repository in a GitHub profile that will receive this local repository (again I explain how to do this in section 4.2). For simplicity I will assume that the remote repository will have the same name as the local repository i.e. [lab-01-website](#).

Note: There is no requirement for local and remote repositories to have the same name, but I find it less confusing if they do.

Finally, a secure communication link must be established between the local repository and the remote repository. This is referred to as a *secure shell link* (SSH) and I explain how to create and test this link in section 4.3.

Assuming these three conditions exist: the GitHub profile is set up, an empty remote repository is present and an SSH link is established. Then we can proceed.

Git command push

```
$ Git push origin  
master
```

2.9.1 Sending (*pushing*) data to a remote repository

A local repository is copied to a remote repository with the use of the **push** command. In Git terminology this is referred to a *pushing* your repository or sometimes *pushing upstream* (*it all sounds a bit rude*).

Now we have this:

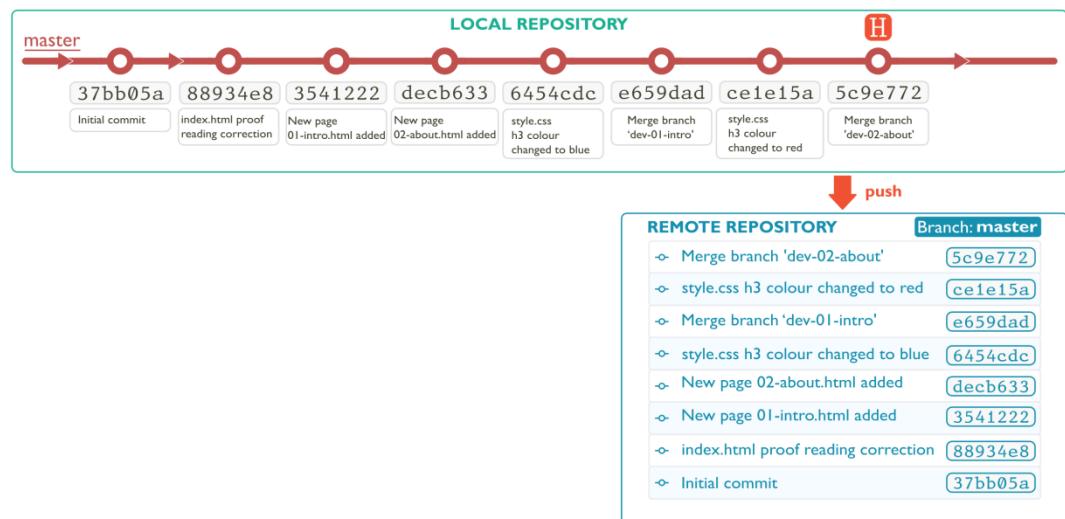


Figure 2.28 lab-01-website—push to remote repository

All the commit points on the master branch of the local repository have been sent to the remote repository.

A **push** only applies to a single branch. If no branch is specified, Git will assume you are pushing the currently active branch (where the **head** is).

2.9.2 Getting (*pulling*) data from a remote repository

Git command `pull`

Let's assume that some other user has added another commit to the remote repository. The remote repository now looks like Figure 2.29:

REMOTE REPOSITORY		Branch: master
↳	03-contact.html added	[ad457b6] NEW
↳	Merge branch 'dev-02-about'	[5c9e772]
↳	style.css h3 colour changed to red	[ce1e15a]
↳	Merge branch 'dev-01-intro'	[e659dad]
↳	style.css h3 colour changed to blue	[6454cdc]
↳	New page 02-about.html added	[decb633]
↳	New page 01-intro.html added	[3541222]
↳	index.html proof reading correction	[88934e8]
↳	Initial commit	[37bb05a]

Figure 2.29 Modified remote repository

There is a new commit at the top of the list [ad457b6].

To get the new commit from the remote repository into the local repository we use (*unsurprisingly*) a `pull` command. It works like this:

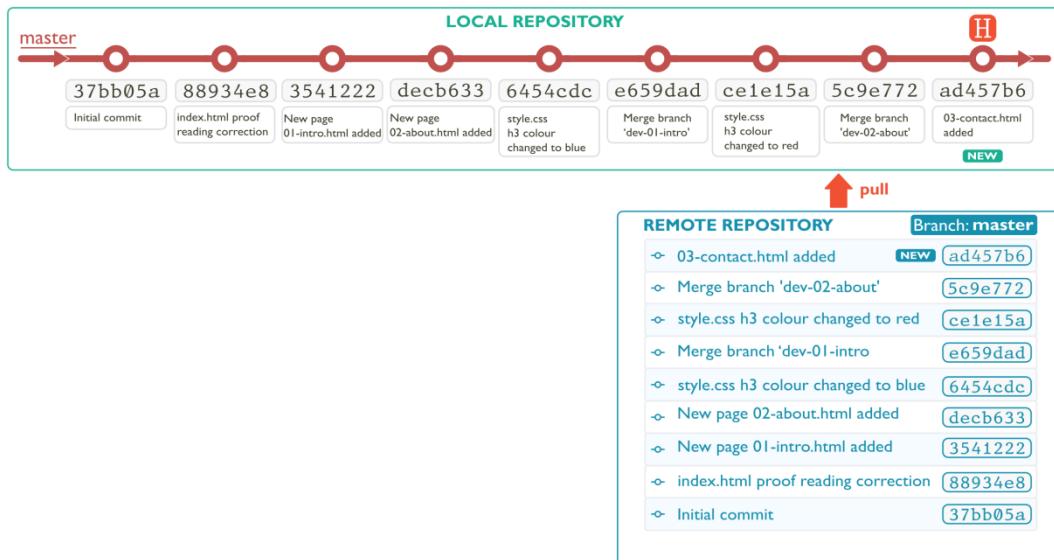


Figure 2.30 lab-01-website—pull from remote repository

Again the `pull` command only applies to a particular branch (or the currently active branch if none is specified).

2.9.3 **What happens if there is a conflict with the remote**

This happens when your local repository is out of date with the remote repository and is a perfectly normal situation.

Let's assume that you have pulled a copy of the repository in to your local repository and you are quite happily working away on it.

In the meantime some other thoughtless bastard also pulls a copy of the remote repository onto their local machine and they also start working on something.

Now let's assume that the other person finishes first¹ and pushes all his changes back to the remote repository.

At some point you will finish your work and try to **push** your changes back to the remote. You will be disappointed. Your changes will be rejected.

So, what to do?

Well in this situation, you have to pull the latest version from the remote repository and combine its changes into your local repository. This is very much like merging branches (§ 2.4).

Once you've pulled the remote repository and resolved any conflicts, you can now push everything back to the remote repository with your changes (*unless of course someone else has modified it again while you were doing all that—bastards*).

I work through a full example of this scenario in section 8.

¹

This seems to be the normal state of affairs for me; I'm usually the last to finish. I used to think it was because they gave me the hard stuff to do. In the end I realised that this wasn't the case—turns out I'm just slow.

2.9.4 Creating a local repository from an existing remote

This is a very common situation. There is a remote repository somewhere with a project in it that people have already been working on. You also want to work on it, so you need to copy the remote repository to a local repository on your machine.

It's easy. Let's say the `lab-01-website` repository exists on GitHub and you've been given user access to it and you want to copy it to your machine as a local repository.

In Git you would navigate to wherever you want the new repository folder to be located, in my case it would be here:

`D:\2500 Git Projects\`

Next use the `clone` command, this creates a new repository folder, initialises it as a local repository and copies all the data from the remote repository into it.

We now have a local repository linked to the remote repository. The new local repository is completely up to date and matches the remote repository exactly (at least it did at the point of creation, regular pull commands must be executed to keep it up to date).

There is a worked example of this, executed through brackets, in section 5.3.

2.9.5 A note on remote connection names

When dealing with a remote repository, that repository has a name (in the previous example it was `lab-01-website`). The local repository can have a completely different name, you can call it whatever you want (in my case, I always give the local and remote repositories the same name—this is just my preference, I think it avoids confusion).

The thing that joins these two repositories is a communication link, a URL (like a web address that points to the remote repository). In the previous example, the URL would be something like: `git@github.com:practicalseries-lab/lab-01-website.git`, which I'm sure you'll agree, would be a pain in the arse if you had to type it in every time you wanted to push data to the remote.

Git command `remote`

```
$ git remote add <con_name> <url>
```

To get round this Git gives the connection a shorter (but unique) name. By default, the first name it assigns is `origin`. There is nothing special about this name; you could assign any name you want. However, hardly anyone ever changes it (bit like the `master` branch). Most connections are called `origin`. I tend to stick with it.

2.9.6 Working with remotes—best practice

This is a short list:

- ① Use the same name for local and remote repositories
- ② Update the local repository frequently
 - Store local changes (`commit` changes)
 - Update the local repository (`pull` from remote)
- ③ Always update and resolve conflicts before pushing changes to the remote repository
 - Store local changes (`commit` changes)
 - Update the local repository (`pull` from remote)
 - Update the remote repository (`push` to remote)

3

INSTALLING GIT

How to install the Git version control software and associated programmes.

GIT IS A FREE, open source and distributed version control system for software and documentation projects.

The Git website says it's easy to use—that's not exactly been my experience; I found it difficult, counterintuitive and basically a right pain in the arse—It uses a command line interface. It's “*You are standing in an open field west of a white house*”¹ stuff. My experience is it's more like the “*you are in a maze of twisty little passages, all alike*”² bit of another game.

For some reason Git and its online partner in crime, GitHub seem to be the version control system of choice for every developer out there—no one has a bad word to say about it (*apart from me—this is going to go well for my inbox*). According to the world and its dog, it's way ahead of everything else.

Now I'm an engineer and I like version control, can't get enough of it, it's a way of life—it's what they teach you at engineering school. So I wanted to like Git (I don't like the name, *git* has a meaning in England: *git* [noun] *a contemptible, stupid or unpleasant person, usually a man—“You miserable git”*), I had a couple of half-hearted attempts and gave up. I eventually had a proper go at it when I started the Practical Series website and after a lot of buggering about, I think I've finally got the hang of it.

I'm recommending using Git and GitHub from within the Brackets text editor—this makes the whole thing easier and more manageable; however, before Brackets can be used to do this, Git must be installed on the local machine and this isn't quite as straight forward as it sounds (*Git was developed by the same people who made Linux, and keeping to my understanding of Linux standard operating procedures, it is by necessity, a right bastard to install*).

Once Git is installed, it has to be set up and configured with the correct defaults and third party applications that make it work properly—this means buggering about with the command line interface and delving into the bits of windows that are normally best left alone.

¹ You had to be a teenager in the 80s, it's from [ZORK](#), an old text adventure.

² It's even older [COLOSSAL CAVE ADVENTURE](#). You can still play it [ONLINE](#).

There is quite a lot to set up and get working and these are my instructions for doing so. I've gone through each step of the process in painful detail, some bits are reasonably complicated, not much of it is intuitive, and the commands needed are hardly user friendly—hence the detailed approach.

Note: *I exclusively use Windows as my operating system (see [HERE](#) for why), this means that all the instructions I give here are for the Windows version of the software (I don't cover the Mac OS or Linux versions).*

3.1

Downloading and installing Git

The first thing to do is install Git. It's available from [HTTPS://GIT-SCM.COM/](https://git-scm.com/) website. It looks like this:



Figure 3.1 Git website

Click the monitor screen to download the current version for your operating system. Going to [DOWNLOADS](#) will give more download options (*the monitor is ok for us*).

This will download the install file, in my case this was:

`git-2.13.3-64bit.exe`

Run the install file and select the following options (*there are a few of these*):

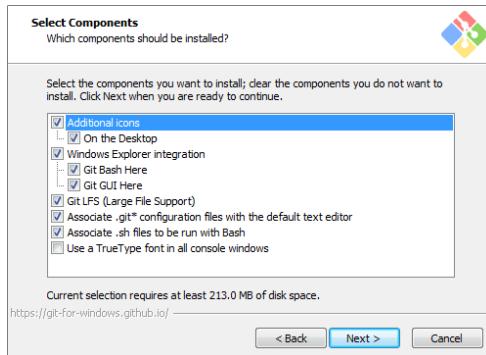


Figure 3.2 Git install—Components

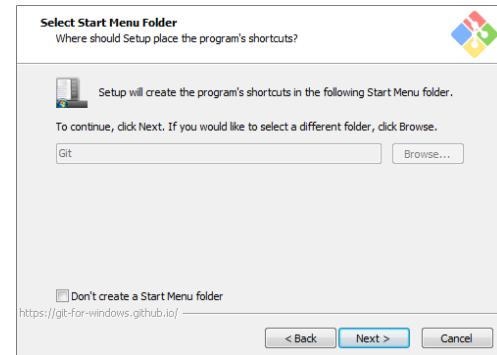


Figure 3.3 Git install—Start menu

The only additional option required on the first screen (Figure 3.2) is the **ADDITIONAL ICONS**, tick the box if you want a desktop icon (*I recommend you do, it comes in handy later*).

The second screen (Figure 3.3) can be left as it is, if you want a Start Menu folder (when you click the Windows start button), tick the box at the bottom (*I did*).

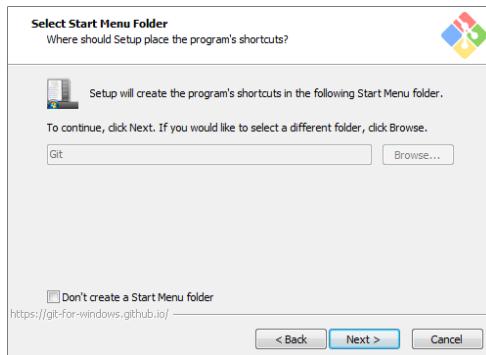


Figure 3.4 Git install—Path environment

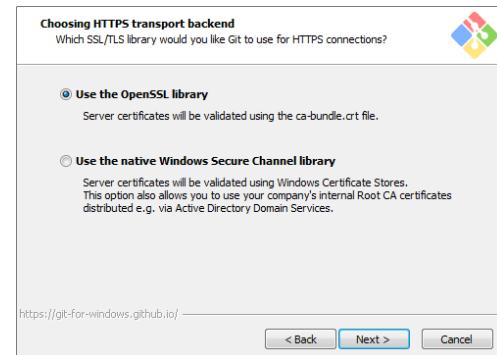


Figure 3.5 Git install—Choosing HTTPS transport

There are no changes to the next two screens (Figure 3.4 and Figure 3.5), leave the default settings selected.

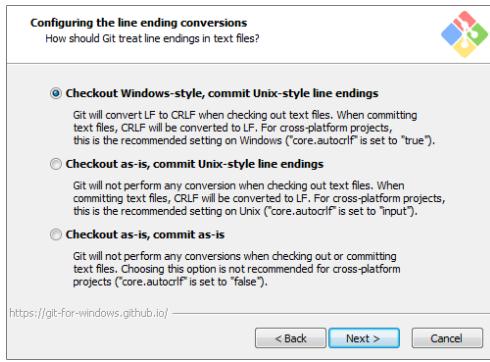


Figure 3.6 Git install—Line endings

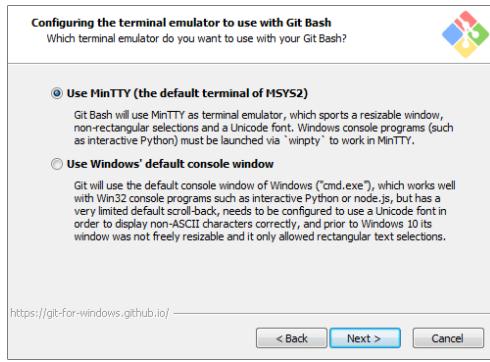


Figure 3.7 Git install—Terminal emulator

Again no changes on the next two (Figure 3.6 and Figure 3.7). I'm using Windows so I want the **WINDOWS-STYLE CHECKOUT**. I don't actually think it makes a great deal of difference. Keep the **USE MINTTY** for terminal emulation—*terminal emulation? Really? Who the hell still uses terminal emulators? It's not the 90s anymore.*

Finally, the last screen (Figure 3.8), no changes here either:

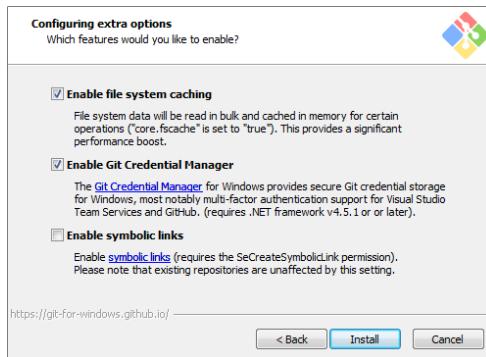


Figure 3.8 Extra options

After all this you will have a Git Bash icon on your desktop (*what's with these names—pretty sure that's a euphemism for something*):



Go on, click it—but brace yourself for the full **BBC MICRO MODEL B** experience:



Figure 3.9 Git Bash

I think the window title should be read MinGW not MING W64 (English again: **ming** [noun, slang] *an unpleasant smell*—“It’s started to ming”).

So there we are; it’s 1980 and this is **CEEFAX**, *can I get the weather?*

Now with most Windows programmes, you can click **FILE → NEW** and something happens, but not here. *What should I do next? How do I make it work? What did it say on its website? “Git is easy to learn”*—my hairy English arse it is.

I tried **XYZZY** and **PLUGH**¹—nothing; still, I haven’t been killed by a dwarf yet.

I’m starting to think they got the names right. *This program is a git, it is wank and it does smell bad.*

¹

If you don’t know, you’re too young to be reading this and I’m too old to explain—try **GOOGLE**.

3.2

Downloading everything else you need

Ok we've got Git, but that's just the start. We need a text editor and we need some sort of program for finding and merging differences in files. Then we need to set these as the default editor and merge tools in Git.

Taking these in turn:

3.2.1 A default text editor for Git

I'm a big fan of Brackets; I use it for all my web development stuff. I wax lyrical about it [HERE](#). The only problem is that it doesn't play nicely with the Git Bash command line; specifically it won't create a new file from the command line which is a bit of a problem for Git¹. My next favourite is [ATOM](#) but this also has problems with Git, It's difficult to set up as the default editor: under Windows, Git won't wait for Atom to close before continuing; again this is a problem for what we want to do.

So I used my third favourite [NOTE PAD++](#). I will still use Brackets for all the web development and I will also use it to manage Git and GitHub repositories. I will just use Notepad++ when creating or editing files directly from the Git Bash command line (that thing you can see in Figure 3.9)—hopefully this won't be too often, the whole point is to manage everything through Brackets and not have to bother with the command line stuff. Unfortunately, we need to use the command line stuff to get everything set up (that's what we're doing now).

I recommend you install both Brackets and Notepad++; I've already covered the Brackets installation on a previous posting: [GETTING AND INSTALLING BRACKETS](#). I look at installing Notepad++ below:

¹

I may be confusing things here; I'm using Brackets as the interface to Git. When I say Brackets doesn't work with Git, I mean as a command line editor in Git Bash. Brackets and its extensions manage Git in the Brackets development environment very well.

Notepad++ is available from the download page of the NOTEPAD-PLUS-PLUS.ORG website. I'm currently using version 7.4.2 (32 bit) for Windows. It is very easy to get, just click the big green download button on the download page (see Figure 3.10).



Figure 3.10 Notepad++ text editor download

This will download the [npp.7.4.2.installer.exe](#) file (at least it will if you are on a Windows system).

Run the file, there are a couple of install screens. On the first (Figure 3.11):

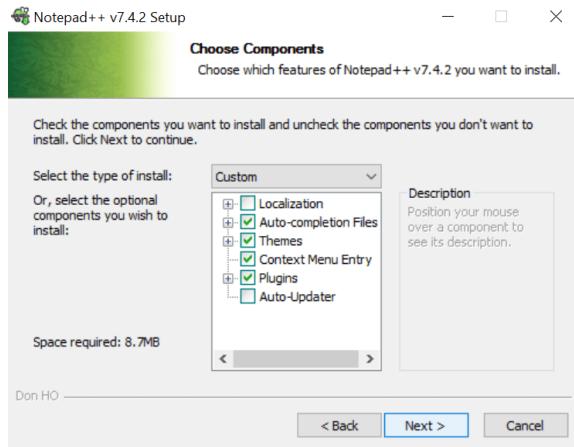


Figure 3.11 Notepad++ install components

I unchecked the **AUTO-UPDATER** option, this is just my preference, Notepad++ can be a bit bothersome with updates, it pesters all the time and it annoys me so I've disabled the option (*I'm a big boy now, I can choose when to update without being told*).

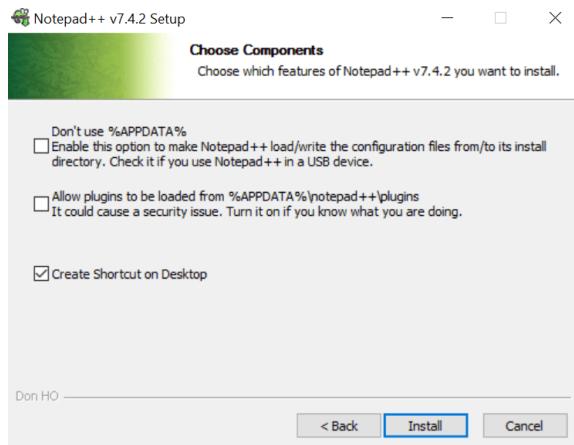


Figure 3.12 Notepad++ final options

Finally, on the next and last option screen (Figure 3.12), I've selected the **CREATE SHORTCUT ON DESKTOP** option (*again, this comes in handy later on*).

Click **INSTALL** and let it open afterwards. It looks like this Figure 3.13:

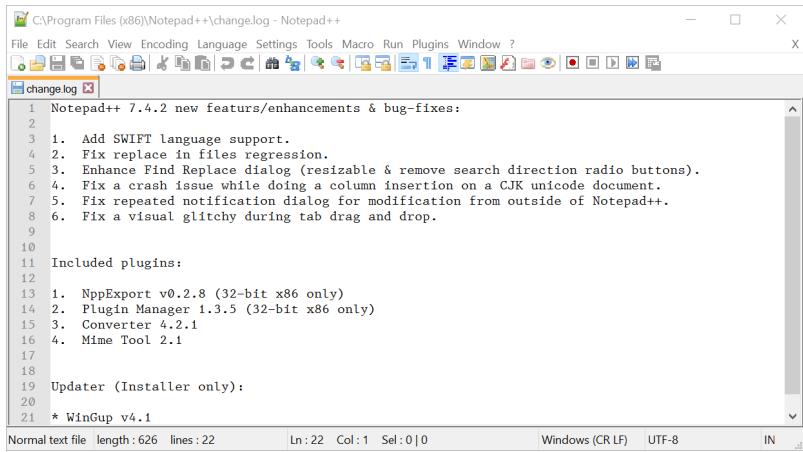


Figure 3.13 Notepad++

I don't like the default white colour theme and I've changed it to a darker one (this is just my choice, pick the one you like best, or leave it as it is).

To change the colour theme, click **SETTINGS → STYLE CONFIGURATOR**. This opens the Style Configuration dialogue box. At the top there is a **SELECT THEME** dropdown box that allows a whole range of themes to be adopted. The one I use is **OBSIDIAN**, select the one you want and click **SAVE AND CLOSE** to make the change.

Obsidian looks like this:

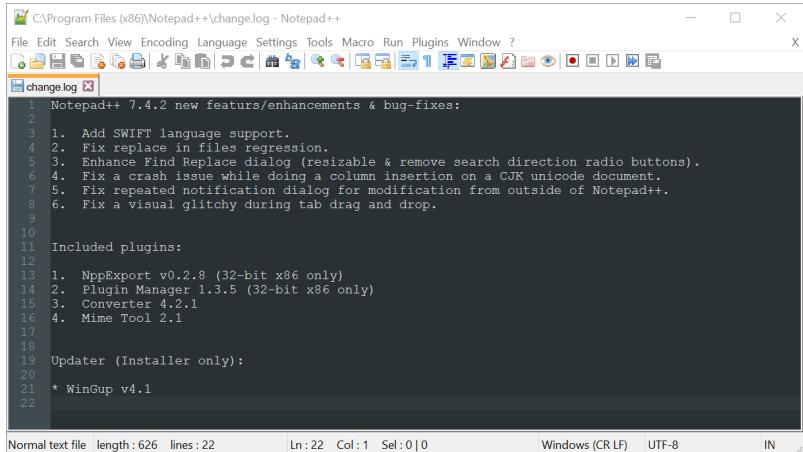


Figure 3.14 Notepad++ Obsidian theme

All the Notepad++ screen shots will have this theme selected.

3.2.2 A difference and merge tool

The next thing is a tool that highlights difference between two different versions of the same file. This is usually required if two people have made changes to the same part of file. The merge aspect allows the differences to be reconciled and merged back into a single file.

Generally, I don't like the idea of two people working on the same file at the same time, but it can happen and this is how Git deals with it. So we need a merge tool.

The one I use is P4Merge, this is a free package made by **PERFORCE** and was recommended to me—I find it works well and is easy to use; that said, I haven't tried any others (*there's a lot of them out there*).

There are a few hoops to jump through to get it. Start with the **PERFORCE** website (Figure 3.15):

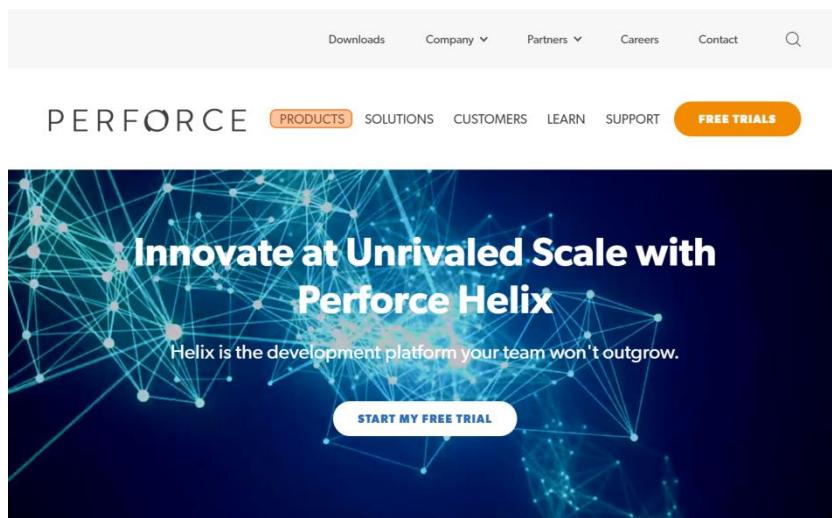


Figure 3.15 Perforce home page

Click the **PRODUCTS** button (highlighted) and on the dropdown click **HELIX APPS** (Figure 3.16):

Note: *This website changes a lot, if it looks different to this, just search for P4MERGE (magnifying glass at the top), you're looking for HELIX MERGE AND DIFF TOOLS (P4MERGE), this give's you something like Figure 3.18.*

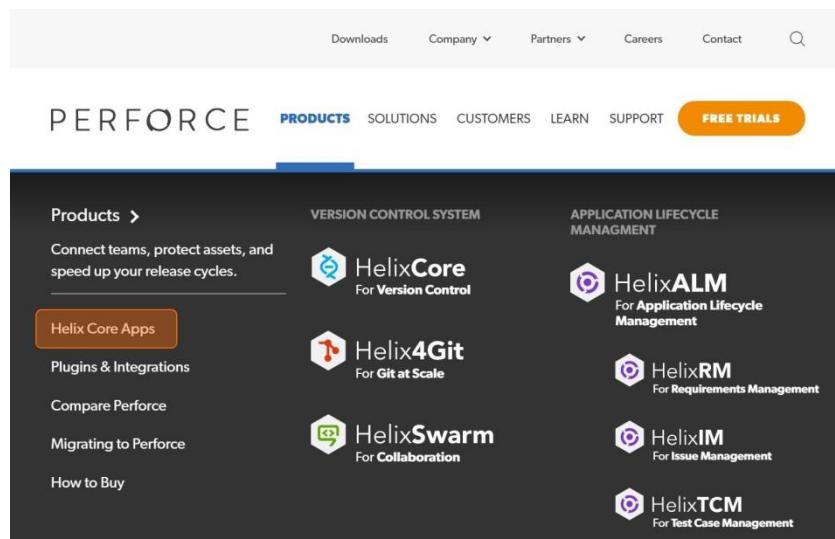


Figure 3.16 Perforce products—select Helix Apps

This gives the core apps page (Figure 3.17), click **MERGE AND DIFF TOOL** (highlighted):



Figure 3.17 Perforce products—select P4Merge tool

This gives the download page (Figure 3.18), click **DOWNLOAD NOW** (highlighted):

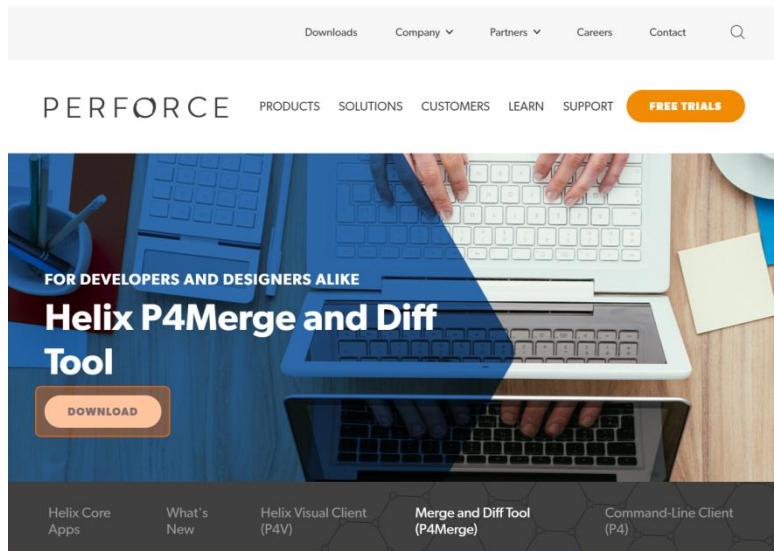


Figure 3.18 Perforce product—client download page

This is the bit we're after, the Visual Merge Tool (Figure 3.19):

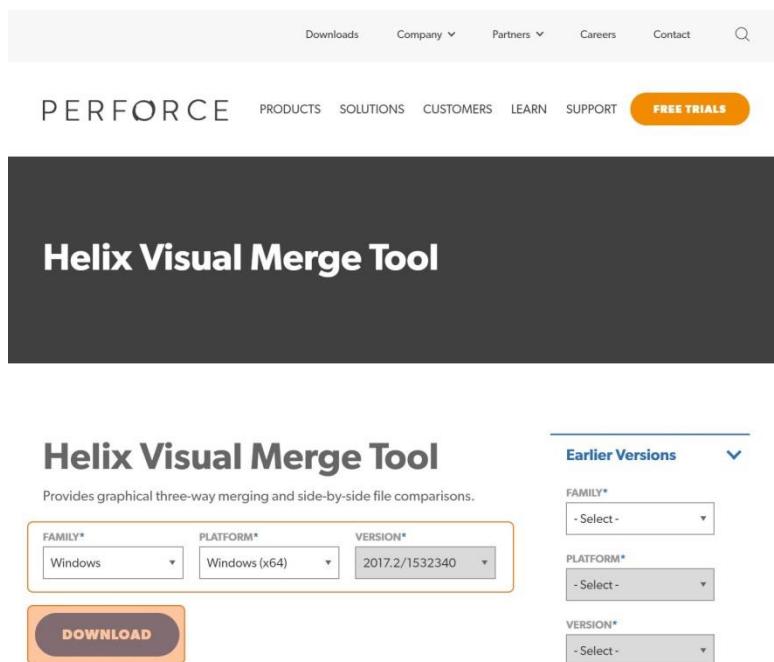


Figure 3.19 Perforce products—Helix download

Select the family and platform (in my case **WINDOWS** and **WINDOWS x64**) and click download. This gives the registration page (Figure 3.20):

The screenshot shows a registration form for the 'Download Helix Visual Merge Tool - Windows (x64) - 2017.2/1532340'. The form includes fields for First Name*, Last Name*, Email Address*, Role*, Team*, Company Name*, Customer or Free User* (with options for Customer and Free Small Team User), and Country*. There is also a 'Skip registration' link and an 'I AGREE' checkbox. The background features the Perforce Helix logo and navigation links.

Figure 3.20 Perforce products—Helix desktop & web apps

You don't have to register, tick the **I AGREE** box and click **SKIP REGISTRATION**. Finally, the download starts (*got there in the end*).

This will download the P4Merge installation file, in my case: **p4inst64.exe**.

Run the file, this gives the Select Features dialogue (Figure 3.21):

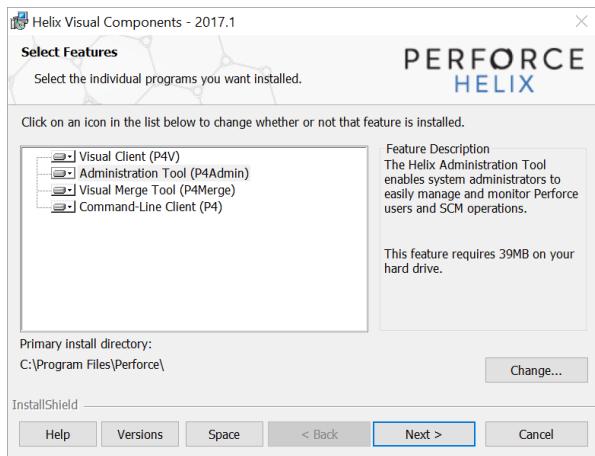


Figure 3.21 P4Merge—Installation features

The only thing we need is the **VISUAL MERGE TOOL (P4MERGE)**, deselect everything else (Figure 3.22):

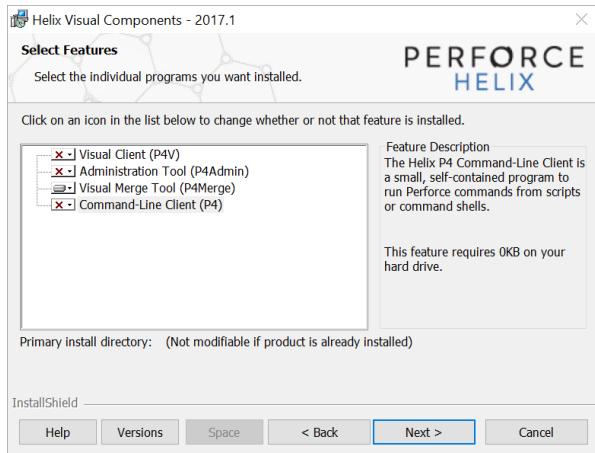


Figure 3.22 P4Merge—Required features

Click **NEXT** and then **START** to install the programme, there are no more option screens.

3.3

Changing the Git default directories

Back to the intuitive easy to learn Git Bash.

By default Git wants to use your user directory as its home directory. This is where it wants to put all its configuration files and repositories; it is the default location that is opened when you start Git Bash.

On my system it's here:

`C:\Users\Michael Gledhill\`

Now I don't want to use my user directory as the main location for Git, I want to use a special folder that I've set up to hold the Git repositories (Git repositories are where we store a software project). I actually want it to use this location:

`D:\2500 Git Projects`

This is just a local folder on my D drive (see § 2.2.1).

Each repository will be a sub directory of this parent directory ([2500 Git Projects](#)).

If you perfectly happy using your user directory as the default directory for Git, you can skip the next bit.

3.3.1 **Changing the default directory for Git**

There's two parts to this:

- ① Changing Git so it stores its configuration files in the new location
- ② Changing Git so it starts in the new location

Taking these in turn:

Changing where Git and Git Bash store the configuration files

Git has several configuration files:

It has a **master configuration** file called `gitconfig`. On a Windows machine it lives here:

`C:\Program Files\Git\mingw64\etc\gitconfig`

This is sometimes referred to as the **system** configuration file.

There is a **global configuration** file called `.gitconfig` (note the leading full stop). This is the one that gets used most and is the one we want to move. By default, on a Windows machine, it is located in:

`C:\Users<username>\.gitconfig`

Finally, each repository has its own **local configuration** file called `config`. This lives in the individual repository:

`...<repositoryname>\.git\config"`

But this is inside the `.git` folder and we don't go there (see § 2.2.1).

The order in which Git Bash applies these configuration files is:

- ① Master (system) configuration (`gitconfig`)
- ② Global configuration (`.gitconfig`)
- ③ Local configuration (`config`)

Effectively, the local configuration has priority because it is executed last and anything in there will override the others if there is a conflict.

By default the system configuration and local configuration are empty. Global configuration is where the changes are made and this is the file whose location I want to change.

To move the default location of the global configuration file we need to edit another of the Git initialisation files. This one is called `profile` and on a Windows machine it lives here:

```
C:\Program Files\Git\etc\profile
```

The directory in the middle really is called etc, I'm not abbreviating anything. Navigate to the file and edit it. It is a reasonably large file (Code 3.1).

To change the default (home) directory we need to add a line to the very end of the file:

```
HOME="D:\2500 Git Projects"
```

The syntax for this is:

```
HOME=path\to\home\folder
```

If any of the directory names contain a space, put double quotes around the whole thing as I did with:

```
HOME="D:\2500 Git Projects"
```

PROFILE

```
1 # To the extent possible under law, the author(s) have dedicated all
2 # copyright and related and neighboring rights to this software to the
3 # public domain worldwide. This software is distributed without any warranty.
4 # You should have received a copy of the CC0 Public Domain Dedication along
5 # with this software.
6 # If not, see <http://creativecommons.org/publicdomain/zero/1.0/>.
7
8
9 # System-wide profile file

. . .

192 echo
193 echo
194 echo
195 echo "#####
196 echo "#"
197 echo "#"
198 echo "#          C   A   U   T   I   O   N"
199 echo "#"
200 echo "#          This is first start of MSYS2."
201 echo "#      You MUST restart shell to apply necessary actions."
202 echo "#"
203 echo "#"
204 echo "#####
205 echo
206 echo
207 fi
208 unset MAYBE_FIRST_START
209
210
211 HOME="D:\2500 Git Projects"
212
```

Code 3.1 Git profile configuration file additions

Now we just need to move the existing `.gitconfig` (the one that was created when Git Bash was installed) to the new location.

In Windows Explorer navigate to:

`C:\Users\<username>\`

Where `<username>` is your user name on the machine. This is what mine looks like:

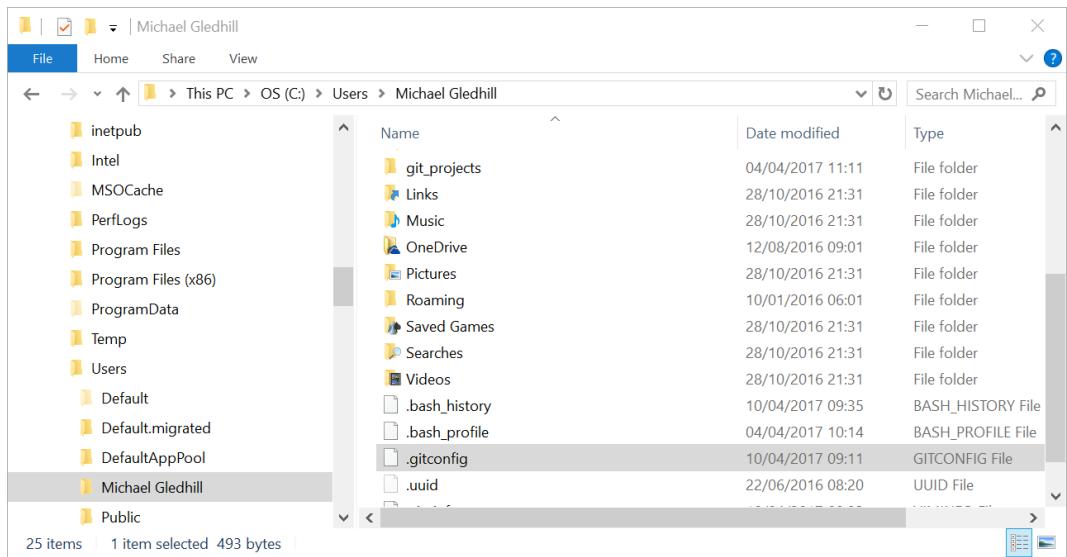


Figure 3.23 .gitconfig file—original location

Select `.gitconfig` and then **CUT** the file (**CTRL+X**), navigate to the new home directory—in my case this is:

D:\2500 Git Projects

And **PASTE** the cut file there (**CTRL+V**).

That's it.

Setting the start directory

This is much easier. On the desktop there will be Git Bash icon:



Right click this icon and select **PROPERTIES**, this opens the shortcut dialogue box (Figure 3.24):

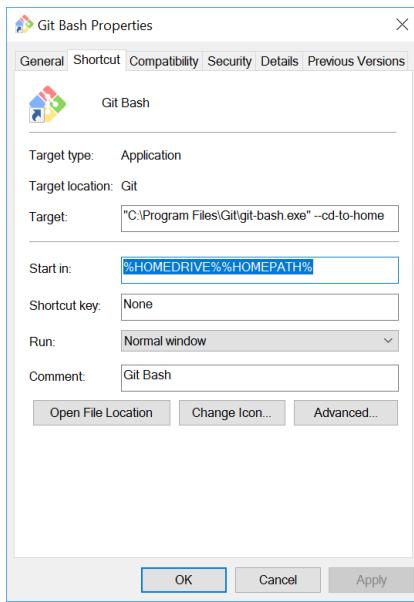


Figure 3.24 Git Bash shortcut dialogue box

Two things to change here:

First in the **START IN** box, enter the path to the new home directory. In my case this is:

"D:\2500 Git Projects"

Next, in the **TARGET** box remove the `--cd-to-home` entry at the end. The final thing should look like this (Figure 3.25):

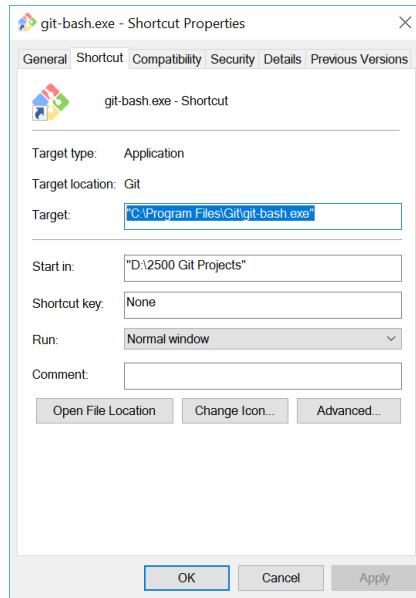


Figure 3.25 Modified Git Bash shortcut dialogue box

That's it, click ok and then click the icon to start Git Bash. This time it will start in your new home directory. Mine looks like this:

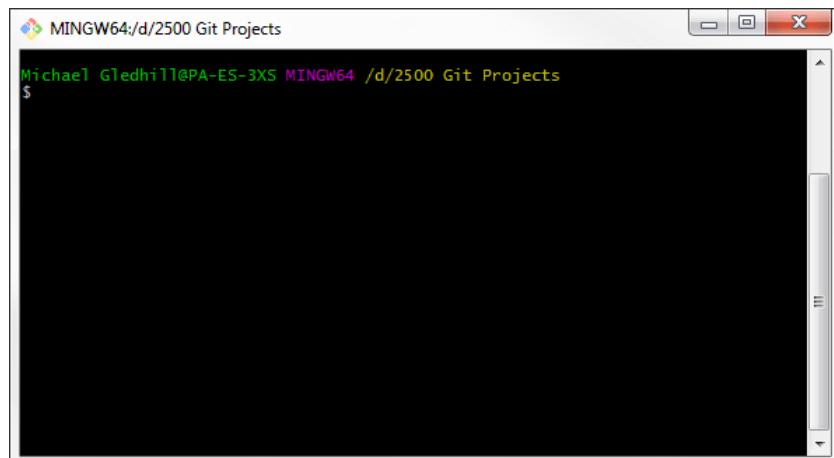


Figure 3.26 Git Bash with modified start folder

The bit in yellow is the path to the current directory (it's in the UNIX style with **D:** replaced with **/d** and obliques (**/**) in place of the reverse obliques (****) that Windows uses for path names—*but you get the idea*).

3.4

Changing the Git default configuration

Git Bash starts life, well, looking like a colourful version of MS-DOS 6.2; and it works in much the same way—*thank God it's so easy to learn.*

3.4.1 Configuring a Username and email address

Before we do anything else, we need to set ourselves up as a user on the local Git system. This means giving Git a user name and an email address.

The email address is purely an internal thing, Git will not use it as an actual email address, it will never send anything to it, try to verify it or pass it to any other party. It uses it to identify the user that has made a change to a file within a Git repository.

Note: *The username and email address supplied here don't need to match the username and email account used to set up your GitHub account (see § 4.1)—they don't need to, but in my case they do, it just make identifying who has made changes easier.*

Start Git Bash and at the \$ prompt enter the following commands:

```
$ git config --global user.name "your-username"
```

Don't actually enter `your-username`, that would be silly. Enter the username you want to use. My username is `practicalseries-lab` so I entered:

```
$ git config --global user.name "practicalseries-lab"
```

You won't get much of a response if you did it right, it just displays the \$ prompt on the next line.

Note: *Git Bash does this a lot—no news is good news.
If it hasn't complained, it's probably worked.
“You are in a maze of twisty little passages, all alike”.*

Next comes the email settings—in much the same way:

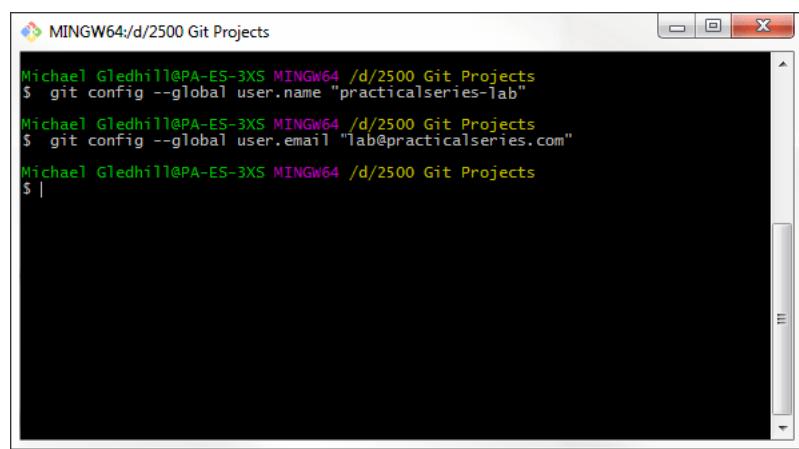
```
$ git config --global user.email "your-email"
```

In my case it's:

```
$ git config --global user.email "lab@practicalseries.com"
```

These commands are great, so blindingly obvious—so easy to learn don't you think?

This is what it looked like on my screen (Figure 3.27):



```
Michael Gledhill@PA-ES-3XS MINGW64 /d/2500 Git Projects
$ git config --global user.name "practicalseries-lab"
Michael Gledhill@PA-ES-3XS MINGW64 /d/2500 Git Projects
$ git config --global user.email "lab@practicalseries.com"
Michael Gledhill@PA-ES-3XS MINGW64 /d/2500 Git Projects
$ |
```

Figure 3.27 Configure Username and Email

How can I tell it's worked? You ask.

The answer is you can't, not yet, not until we've configured the text editor. That's next:

3.4.2 Configuring a default (core) text editor

I'm assuming you are going to use Notepad++ as the default text editor for Git Bash (this is the one I installed in section 3.2.1). Notepad++ works well with Git, some of the others (Brackets¹ for example) don't work so well.

Ok, you'll like this; it's a trip down memory lane. To make Notepad++ the default text editor for Git Bash we have to adjust the *Windows Environment Variable*. I think the last time I had to do this was when I was using WordPerfect 5.1 and Lotus 123. It was 1989, I was 23, MS-DOS was the operating system of choice, Margaret Thatcher was prime minister and my children hadn't been born—*happy times*.

Environment Variables (*can't believe I'm talking about this—it's like talking about my Grandmother's mangle or outside lavatories*), they still exist in Windows, though most programmes just set them up as part of the installation process or use the registry instead. In Windows, Environment Variables describe the computer environment within which programmes run; they define common names (the name of the computer for example) and default file extensions.

The best known Environment Variable is PATH, this contains a list of directory paths. If a user types a command without specifying the path, the operating system will search the current directory and then each of the paths listed in the PATH environment variable trying to find it. Git Bash relies on the PATH environment variable to find any programme we wish to execute—*of course it does, that's what programmes did in the eighties*.

Adding Notepad++ to the PATH Environment Variable

To make Notepad++ the default text editor for Git Bash, we must add the path to the notepad++ executable file to the PATH environment variable.

¹

Just to clarify; I'm using Brackets as the interface to Git. When I say Brackets doesn't work with Git, I mean as a command line editor in Git Bash. Brackets and its extensions manage Git in the Brackets development environment very well.

If you have a Notepad++ icon on your desk top, you can find the path by right clicking it and selecting **PROPERTIES**. This opens the Shortcut dialogue box (Figure 3.28).

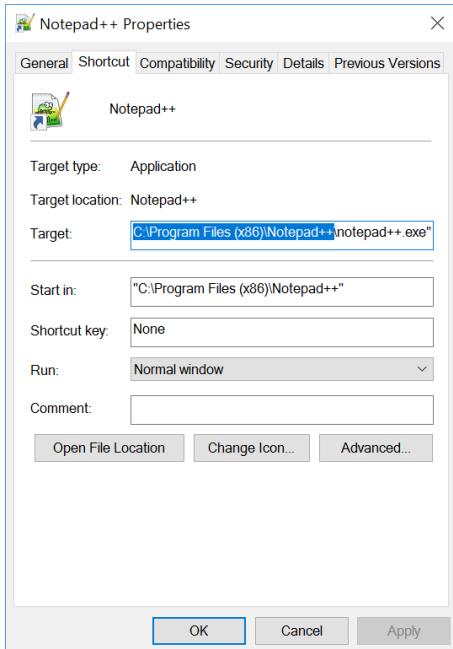


Figure 3.28 Notepad++ path

The information we need is in the **TARGET** box, it's everything up to the last reverse oblique (i.e. everything but the filename), highlighted in blue in the above figure.

In my case (and probably your case too if you didn't change where the programme was installed) it is:

C:\Program Files (x86)\Notepad++

Highlight it and copy it to the clipboard (**CTRL+C**).

Now to change the PATH Environment variable. To do this we need the system properties screen. On Windows 7, click the **START BUTTON** and then right click **COMPUTER** and select **PROPERTIES**. On Windows 10, right click the **START BUTTON** and select **SYSTEM**. Alternatively, use the shortcut **WIN+PAUSE/BREAK** (this is the Windows key and the pause/break key—top right of your keyboard), this works on either.

The system properties screen looks like this (Windows 7 left and Windows 10 right):

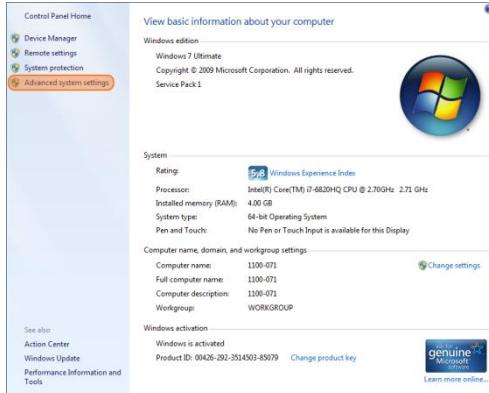


Figure 3.29 System properties Windows 7

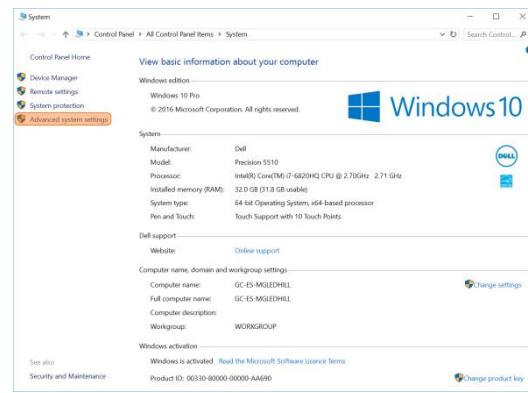


Figure 3.30 System properties Windows 10

In either case, click **ADVANCED SYSTEM SETTINGS** and then click **ENVIRONMENT VARIABLES** (this is much the same on Windows 7 and Windows 10), Figure 3.31.

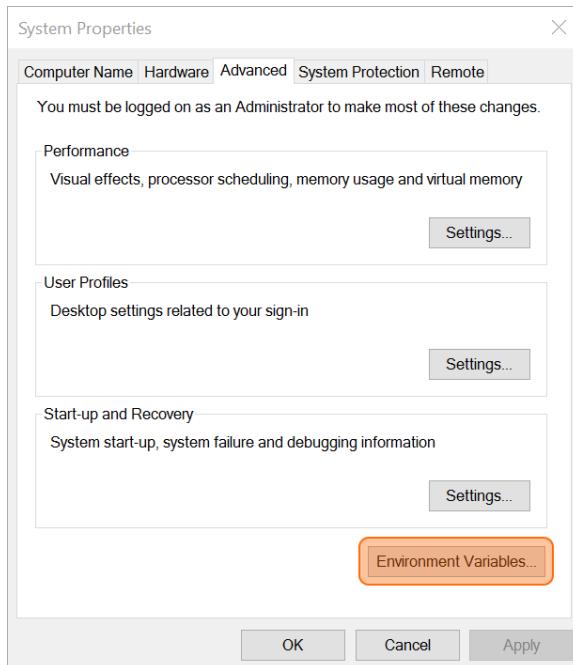


Figure 3.31 Select environment variables

This opens the Environment Variable dialogue box:

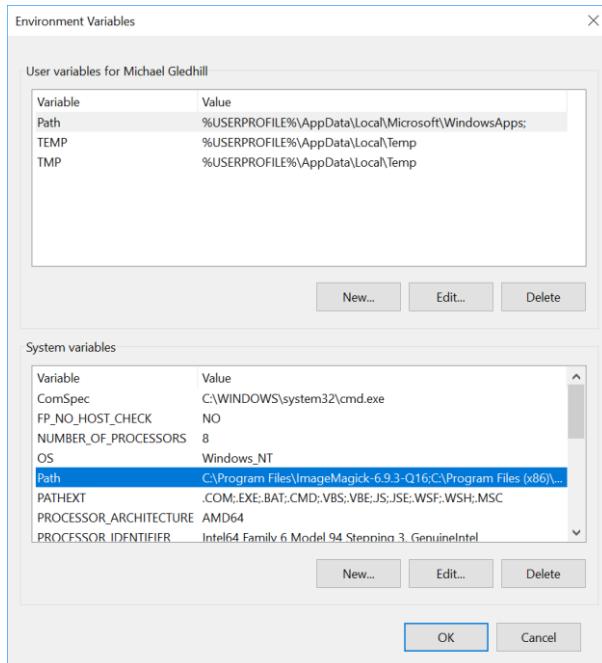


Figure 3.32 Select Path environment variable

Again this looks pretty much the same in Windows 7 and Windows 10. In the lower window (**SYSTEM VARIABLES**) scroll down until you find the **PATH** variable, highlighted in Figure 3.32.

Select the line and click **EDIT**—things are a little bit different now between Windows 7 and Windows 10. Windows 7 first:

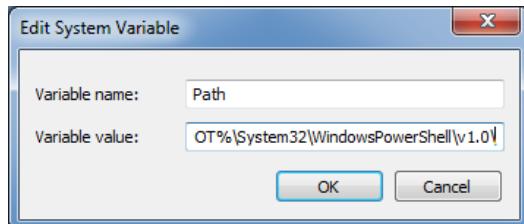


Figure 3.33 Windows 7 path variable

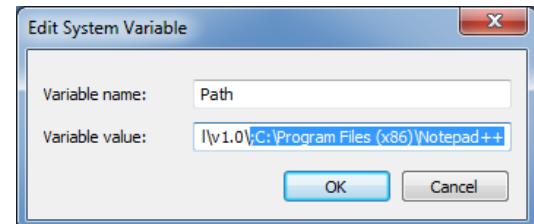


Figure 3.34 Windows 7 modified path variable

In Windows 7 this opens a simple edit dialogue box (Figure 3.33). Click inside the **VARIABLE VALUE** box and cursor right to the end of the line (*by default the box will be showing you the end of the line—there will be a lot of stuff in there*).

Add a semicolon (;) and then hit **CTRL+V** to past in the path copied from the NotePad++ shortcut properties (Figure 3.28). In my case (and probably yours) I added:

```
;C:\Program Files (x86)\Notepad++
```

Note the leading semicolon (this separates one path from another, see Figure 3.34).

That's it, click **OK** repeatedly and close the System Properties window.

In Windows 10, things are a bit easier. Clicking **EDIT** opens a slightly more intuitive edit dialogue box:

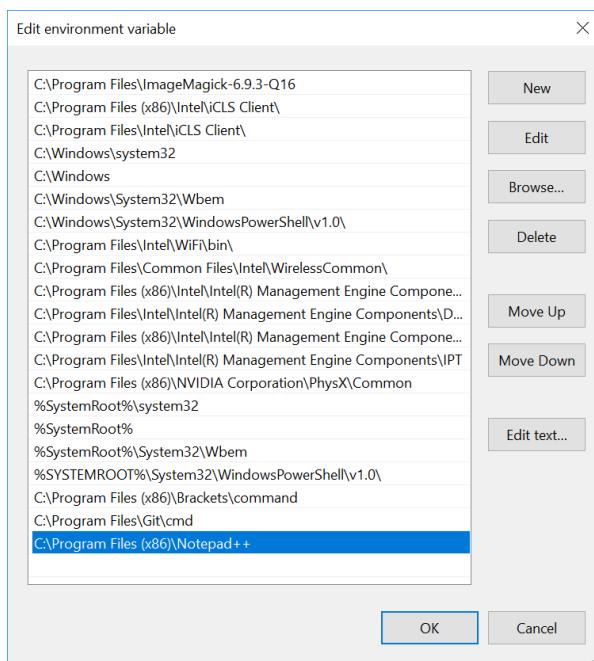


Figure 3.35 Windows 10 modified path variable

Click **NEW** and paste the link in the empty line at the bottom (Figure 3.35).

Again, repeatedly click **OK**, close the System Properties window and we're done.

Setting the Git Bash core text editor

Now if all that has worked, we can tell Git Bash to use Notepad++ as the default (core) text editor.

Start Git Bash and type the following easy to understand command:

```
$ git config --global core.editor "notepad++ -multiInst -nosession"
```

And that's it—*has it worked?* You ask.

Well if it has worked when you type the following, you should see the global configuration file open in Notepad++ (this is the global file edit command, *just go with it for now*):

```
$ git config --global -e
```

The file is the `.gitconfig` file I talked about earlier (§ 3.3.1). It looks like this:

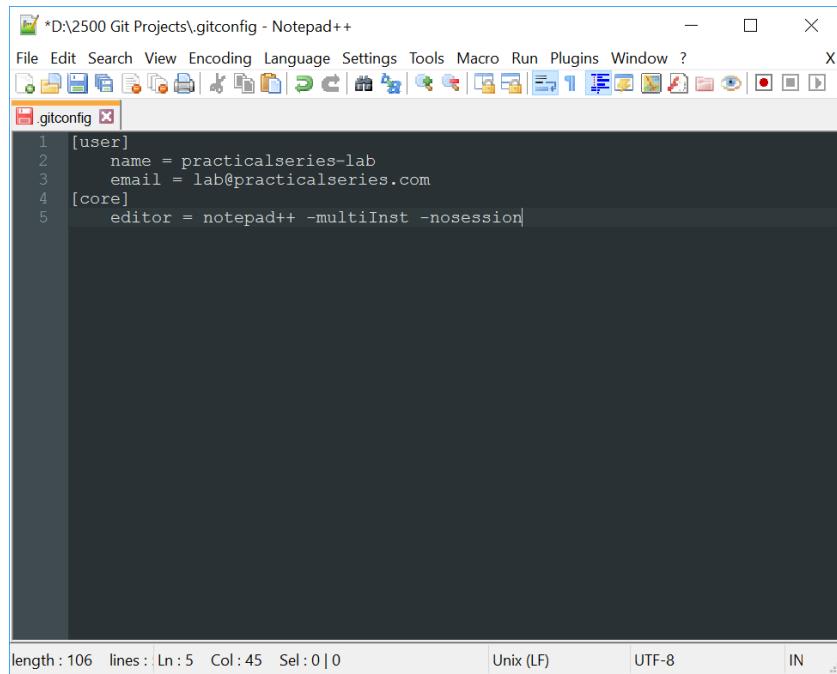


Figure 3.36 .gitconfig file in default text editor

Finally, we can see that the username and email is configured (§ 3.4.1) and that the core editor is set to Notepad++ (obviously, your file will have different user information).

While this file is open, there is something important to note:

If you look at your Git Bash window, it will look like something similar to Figure 3.37:

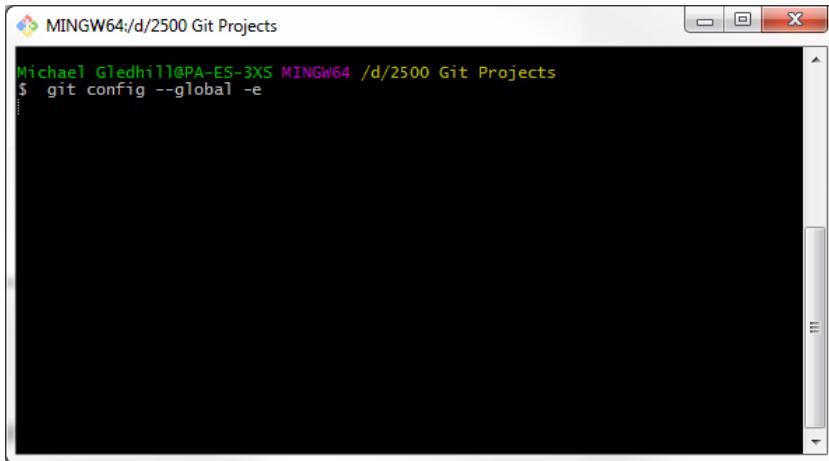


Figure 3.37 Git bash with open text editor

You can see that Git Bash has not returned with the \$ prompt, Git Bash is waiting for the text editor to close and this is exactly what we want it to do. Generally, Git Bash should wait for the triggered application to finish before it moves on².

Close Notepad++ and Git Bash will return to normal with a \$ prompt indicating it is waiting for your next instruction.

Now, I don't know about you, but this stuff doesn't seem that easy to learn to me. I'm not sure I agree with their statement "Git is easy to learn".

²

This is exactly what it didn't do with Atom, it just dropped through to the \$ prompt. I think this is a problem with Git (or Atom itself) running under Windows. It is for this reason that I'm not using Atom.

3.4.3 Configuring a default difference tool

Strictly speaking, this bit isn't necessary; however, it does improve the command line—we don't need it to operate Git through the Brackets interface. Skip it if you want to.

When Git detects a conflict in a file (this happens when two different users have modified the same section of a particular file and Git doesn't know which version to select, see § 2.4.1), it will try to display the conflict by using a difference or comparison tool. Git Bash has a rudimentary difference tool built into it, but it is limited and difficult to use.

To overcome this, we can configure Git Bash to use a third party difference tool, these generally have a graphical interface, accept the same arguments as the built in tool and are much easier to use.

In my case I use P4Merge, the thing we downloaded and installed in § 3.2.2.

To tell Git Bash to use P4Merge as its diff tool, enter the following commands:

```
$ git config --global diff.tool p4merge
$ git config --global difftool.p4merge.path "C:/Program
Files/Perforce/p4merge.exe"
$ git config --global difftool.prompt false
```

You can cut and paste the commands from the above list. In Git Bash use:

- **SHFT+INSERT** — insert selection from clipboard
- **CTRL+INSERT** — copy selection to clipboard

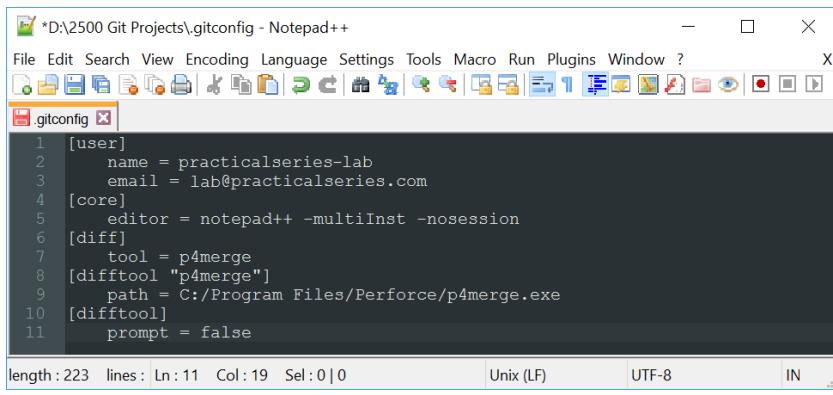
Note: In the second line, the path C:/Program Files/Perforce/p4merge.exe is the path to the P4Merge executable; it's probably the same on your machine if you didn't change the defaults. If you did you will have to type in the correct path for your PC.

Again note that in Git Bash, any Windows reverse oblique (\) in the path must be replaced by an oblique (/).

To see if all this has worked, look in the global configuration file by typing:

```
$ git config --global -e
```

This time you should have:



The screenshot shows a Notepad++ window with the title bar "D:\2500 Git Projects\.gitconfig - Notepad++". The menu bar includes File, Edit, Search, View, Encoding, Language, Settings, Tools, Macro, Run, Plugins, Window, and Help. The toolbar below the menu has various icons for file operations like Open, Save, Find, and Copy. The main editor area contains the following .gitconfig file content:

```
[user]
name = practicalseries-lab
email = lab@practicalseries.com
[core]
editor = notepad++ -multiInst -nosession
[diff]
tool = p4merge
[difftool "p4merge"]
path = C:/Program Files/Perforce/p4merge.exe
[difftool]
prompt = false
```

At the bottom of the window, status bars show "length: 223 lines: 11 Col: 19 Sel: 0 | 0", "Unix (LF)", "UTF-8", and "IN ..\\".

Figure 3.38 gitconfig file with diff tool configuration

3.4.4 Configuring a default merge tool

The merge tool is going to be P4Merge (same as the difference tool), this is because the package does both functions. Merging is the process of resolving a conflict displayed with the diff tool; normally, this is by selecting one modification over another, again § 2.4.1.

The commands for setting the default merge tool are identical to those used to set the default diff tool, but use the word `merge` instead of `diff`. Enter the following:

```
$ git config --global merge.tool p4merge
$ git config --global mergetool.p4merge.path "C:/Program
Files/Perforce/p4merge.exe"
$ git config --global mergetool.prompt false
```

Again to see the changes have been made, edit the global configuration file:

```
$ git config --global -e
```

```

*D:\2500 Git Projects\.gitconfig - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
D:\2500 Git Projects\*.gitconfig x
.gitconfig x
1 [user]
2     name = practicalseries-lab
3     email = lab@practicalseries.com
4 [core]
5     editor = notepad++ -multiInst -nosession
6 [diff]
7     tool = p4merge
8 [difftool "p4merge"]
9     path = C:/Program Files/Perforce/p4merge.exe
10 [difftool]
11    prompt = false
12 [merge]
13    tool = p4merge
14 [mergetool "p4merge"]
15    path = C:/Program Files/Perforce/p4merge.exe
16 [mergetool]
17    prompt = false

```

length : 343 lines : Ln : 17 Col : 19 Sel : 0 | 0 Unix (LF) UTF-8 IN

Figure 3.39 gitconfig file with diff and merge tool configuration

That's all the default files set up.

3.4.5 Defining an alias

There's one more thing to do. In Git bash, we can trigger Notepad++ just by typing its name at the \$ prompt:

```
$ notepad++
```

But this is rather a lot to type each time. There is a fairly common abbreviation for Notepad++ and that is npp, better to just be able to type [npp](#). To do this, we must assign an alias to [notepad++](#).

Aliases are assigned in another of the Git Bash configuration files, this time it is a file called [.bash_profile](#).

We'll edit it from within Git Bash itself type:

```
$ notepad++ ~/.bash_profile
```

This will open or create the [.bash_profile](#) file in Notepad++ (it will be empty).

Add the following line:

```
1 alias npp='notepad++ -multiInst -nosession'
```

Save the file, close and restart Git Bash.

Now type:

```
$ npp
```

This will now open Notepad++.

The alias works by assigning a name ([npp](#) in this case) to a command and its arguments.

4

GITHUB AND SSH LINKING

Create a GitHub account and link it to a local repository using an SSH key.

IN THE PREVIOUS SECTION we installed and set up the Git and Git Bash local version control system.

In this section we will create an online GitHub account, create a remote repository and then link this remote repository to our local Git application. In the next section (§ 5), we will expand this to make a local clone of a remote repository using Brackets and the Brackets-Git extension.

The link between the two repositories will use *secure shell* protocol (SSH) and I will explain in this section, how to generate the SSH security keys and how to deploy them.

4.1

Creating a GitHub account

Creating a GitHub account is easy; go to the [GITHUB.COM](#) web page and click the big [SIGN UP FOR GITHUB](#) button (Figure 4.1):

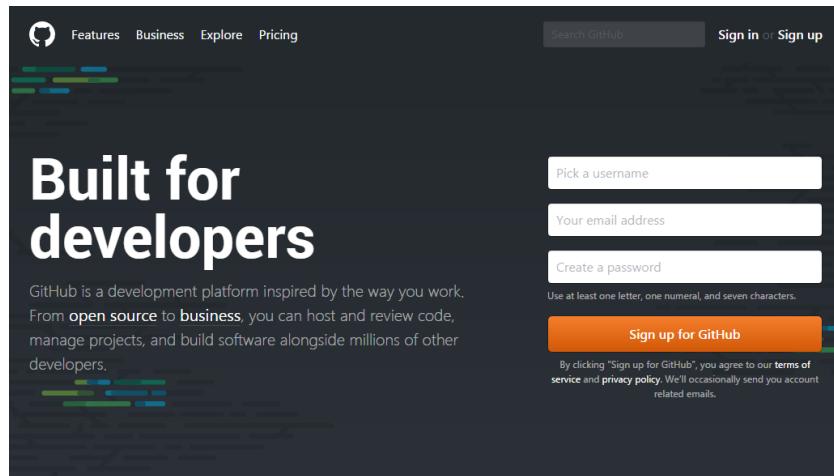


Figure 4.1 Signup for GitHub

Next thing is to enter a username, email address and password (Figure 4.2).

These don't have to be the same as those used on the local version of Git. I.e. those we set up in the previous section (§ 3.4.1), but in my case they are.

The username must be unique amongst all the GitHub accounts, if it is you will get a green tick after it (as shown). Similarly, the email address must not have been used for any other GitHub account.

The email address must be valid and you will have to verify it later.

The password must be a minimum of seven characters and contain at least one letter and number (it will accept punctuation characters too).

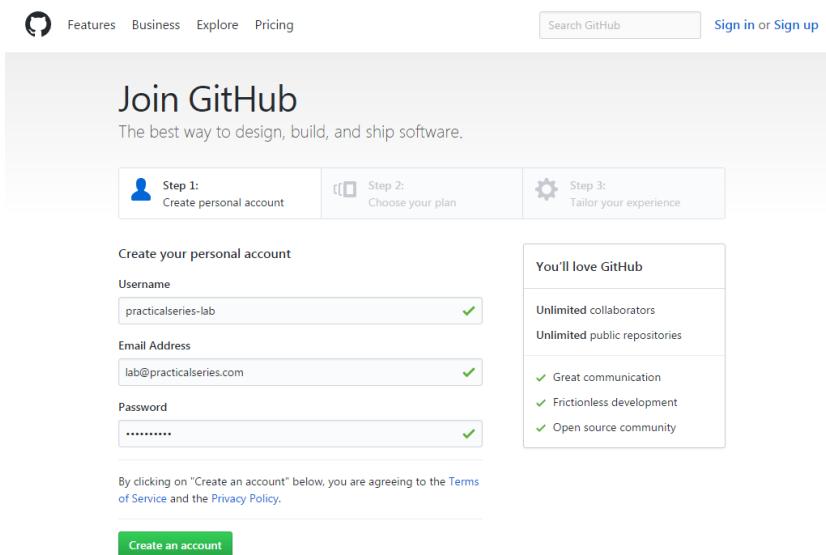


Figure 4.2 Create a personal account

Click **CREATE AN ACCOUNT**.

The next thing is to choose the type of account you want (Figure 4.3):

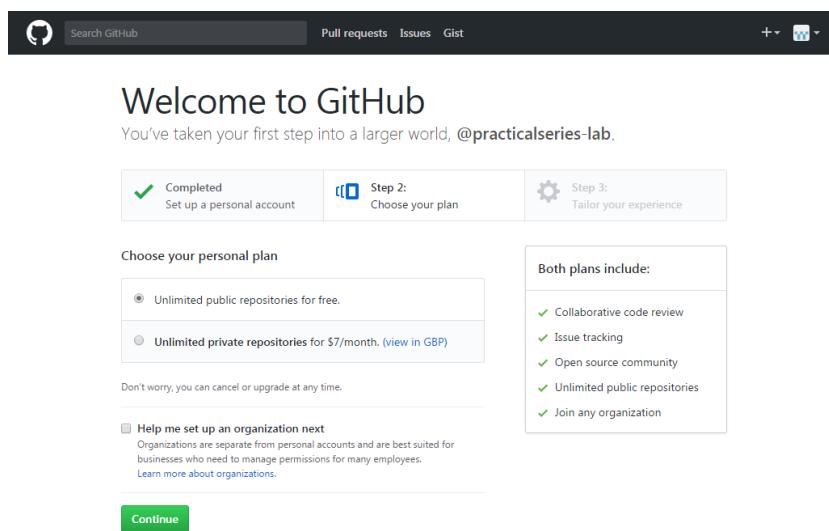


Figure 4.3 Choose your plan

GitHub accounts are free, but only if you are willing to just have public repositories. Public repositories can be seen by anyone, anyone can access the repository and copy anything that is in there. It is entirely public.

If you want the facility to have private repositories, you must pay for the privilege. Private repositories can only be viewed by you and people you give permission to.

For the time being go with the free option. You can upgrade at any time.

Click **CONTINUE**.

The next screen just asks for some personal information, supply whatever you want and click **SUBMIT** or click **SKIP THIS STEP** if you don't want to supply any information.

The screenshot shows the GitHub Experience page. At the top, there's a navigation bar with the GitHub logo, a search bar, and links for 'Pull requests', 'Issues', and 'Gist'. To the right are icons for '+' and 'W'. Below the bar, the title 'Welcome to GitHub' is displayed, followed by the subtext 'You'll find endless opportunities to learn, code, and create, @practicalseries-lab.' A progress bar indicates 'Completed Set up a personal account' (green checkmark), 'Step 2: Choose your plan' (blue icon), and 'Step 3: Tailor your experience' (blue gear icon). The main content area asks about programming experience with radio buttons for 'Very experienced', 'Somewhat experienced', and 'Totally new to programming'. It also asks about plans for GitHub use with checkboxes for 'Research', 'School projects', 'Project Management', 'Design', 'Development', and 'Other (please specify)'. It then asks about self-description with radio buttons for 'I'm a student', 'I'm a hobbyist', and 'I'm a professional', plus an 'Other (please specify)' option. A text input field for interests is present with placeholder text 'e.g. tutorials, android, ruby, web-development, machine-learning, open-source'. At the bottom are 'Submit' and 'skip this step' buttons.

Figure 4.4 Experience page

After this you will be taken to the newsfeed (*landing*) page for your new account. Mine looked like this:

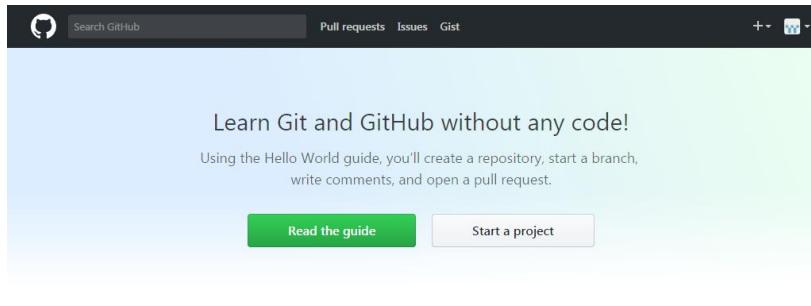


Figure 4.5 Newsfeed page

The next thing to do is check your emails and verify the email address you supplied. GitHub will have sent an email to the address you gave. This was mine:

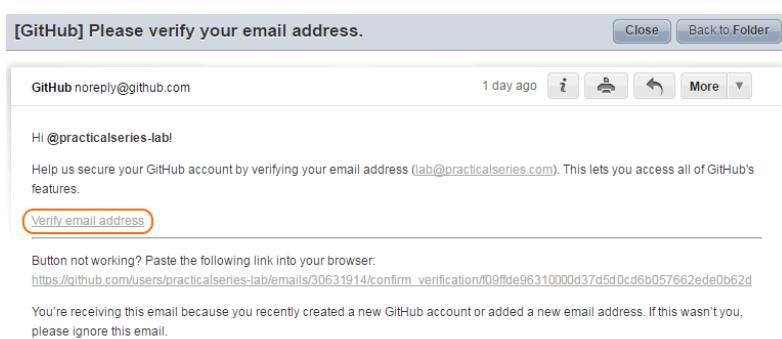


Figure 4.6 Email verification

Click the **VERIFY EMAIL ADDRESS** link and this will take you back to your profile settings page. My profile page is shown in Figure 4.8.

Note: *The profile settings page can be accessed by clicking the dropdown next to the small space invader icon at the top left and selecting settings (Figure 4.7):*

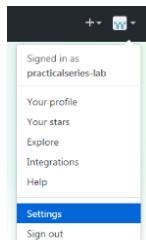


Figure 4.7 Account settings

The screenshot shows the GitHub profile settings page for a user named "Practical Series Lab".

- Public profile:**
 - Name:** Practical Series Lab
 - Profile picture:** A red square placeholder image with the letters "PS" on it.
 - Bio:** This is the Practical Series of Publications demo area; it holds various examples that show how to manage Git repositories from within the Brackets text editor.
 - URL:** http://practicalseries.com/
 - Company:** @practicalseries
 - Location:** Chester, Great Britain
 - Update profile:** A button to save changes.
 - Privacy:** We store your personal data in the United States. See our [privacy policy](#) for more information.
- Contributions:**
 - Include private contributions on my profile:** A checkbox that, when checked, allows credit for work on private repositories.
 - Update contributions:** A button to refresh contribution statistics.
- GitHub Developer Program:**
 - Information:** Building an application, service, or tool that integrates with GitHub? Join the GitHub Developer Program, or read more about it at our [Developer site](#).
 - Developer Site:** Check out the [Developer site](#) for guides, our API reference, and other resources for building applications that integrate with GitHub. Make sure your contact information is up-to-date below. Thanks for being a member!
- Jobs profile:**
 - Available for hire:** A checkbox that, when checked, indicates the user is available for hire.
 - Save jobs profile:** A button to save changes.

Figure 4.8 GitHub profile settings

If you have not yet verified your email address (see above), GitHub will complain and you will have a warning at the top of your profile settings.

Populate this page as you will. If you upload a picture, it will replace the space invader icon in the top right. If you do change your picture, you must also click **UPDATE PROFILE** to make the change permanent.

There's not much more to say, click through the options on the left sidebar to set things up as you require.

Don't do anything with [SSH AND GPG KEYS](#), I discuss these in the next section.

One thing you may have noticed on the [practicalseries-labs](#) profile page is the [@practicalseries](#) entry under [COMPANY](#); this is a way of listing other users, I also have the GitHub account [PRACTICALSERIES](#). I also have my own account [MGLEDHILL](#) (*that's me, top left, handsome chap*)—you will find the repository for the development of this website publication under my [MGLEDHILL](#) profile.

4.2

Creating a first repository

I'm going to create a very basic repository in GitHub and then establish a *secure shell* (SSH) link to it from the Git Bash application on my local machine. This will allow me to securely create a local copy of the repository on my machine, in Git terminology, this is called *cloning* (see §§ 2.9.4 and 5.3).

Let's create a GitHub repository. In my case I'm going to call it `lab-1st-repo`.

Go to the GitHub site and log in to the account you created earlier. This will take you to your newsfeed page. Mine still looks like this:

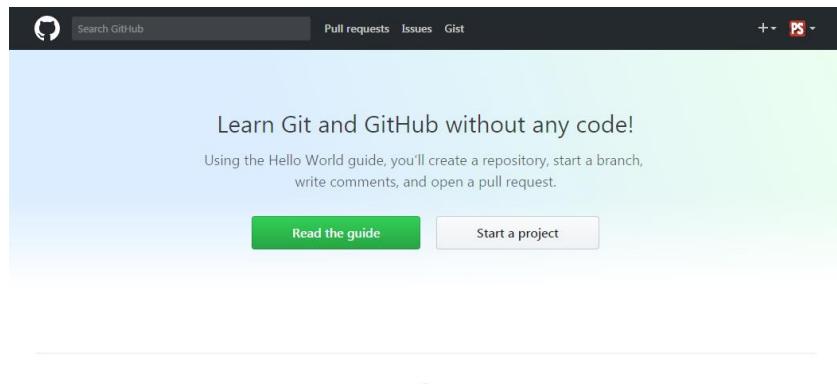


Figure 4.9 GitHub newsfeed page

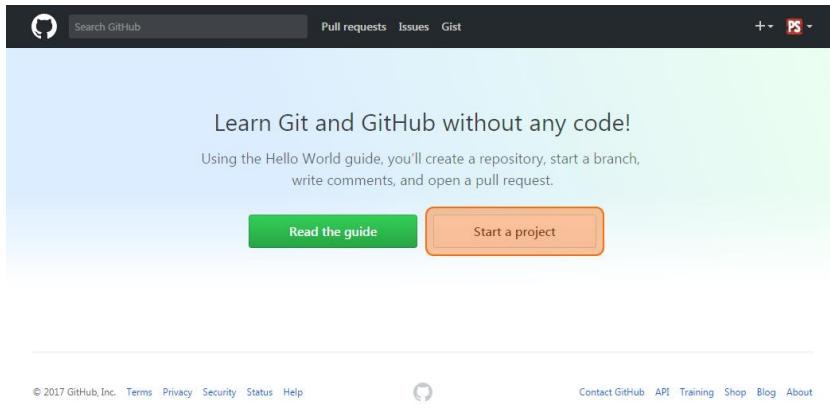
You can always reach this page by clicking the *Octocat* icon at the top left, this:



The first thing to do is create a new repository to hold our example project. I'm going to call it:

lab-1st-repo¹

To do this, click the **START PROJECT** button (highlighted).



© 2017 GitHub, Inc. Terms Privacy Security Status Help Contact GitHub API Training Shop Blog About

Figure 4.10 Start new project

This will open the Create new repository page. I'm going to select the following options (Table 4.1 and Figure 4.11):

PROPERTY	VALUE
Repository name	lab-1st-rep
Description	A first repository to demonstrate SSH connections
Public/private	Public
Initialise with README	Ticked (selected)
Add .gitignore	None
Add a license	None

Table 4.1 lab-1st-repo settings

¹

Reverting back to my schooldays here, *lab*: short for *laboravi in exemplum*: "A worked example"—a disproportionate amount of my schooldays were spent learning Latin

Repository names only have to be unique within your profile; two different profiles can have repositories with the same name.

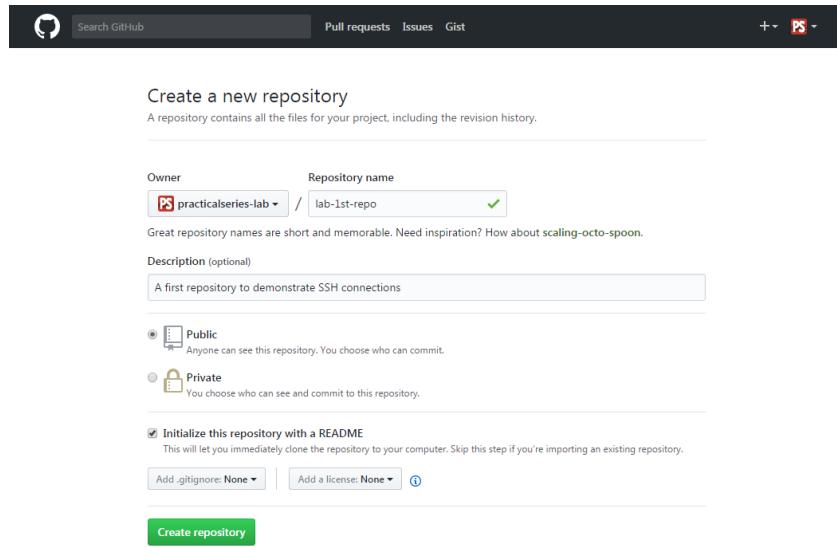


Figure 4.11 GitHub—Create a new repository

Click **CREATE REPOSITORY** and it will automatically open the repository page:

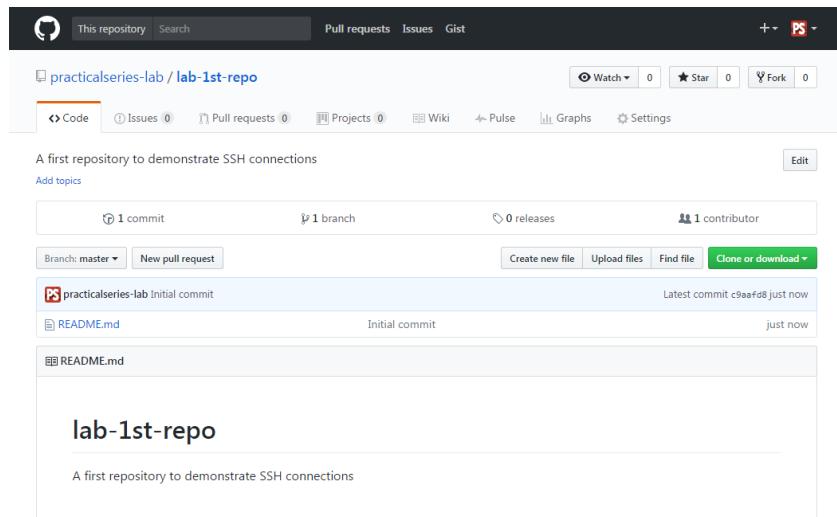


Figure 4.12 GitHub—lab-1st -repo repository page

And that's it; we've created a remote repository in GitHub.

It's got just one file: [README .md](#).

4.3

Setting up an SSH key on the local machine

Ok, we've got our first remote repository in GitHub, now we want to link our local version of Git to this repository and we do this by using a *secure shell* (SSH) protocol connection.

SSH is a secure way of connecting a machine to a remote server; in our case connecting a local machine to a GitHub profile.

SSH uses a pair of connected keys (these are text files that contain a unique and long string of characters) one key is the private key, this will live on the local machine and the other is the public key, this will be given to the server. When the local machine tries to connect to the server, the private key on the local machine is compared with the public key on the server; if the two are a pair, the server grants access to the local machine without the need for a password.

The SSH keys are long enough to be very secure, and are nearly impossible to crack with a brute force attack.

4.3.1 Creating an SSH key pair

Start Git Bash (*yes we're back here again*).

This will start in the default directory; earlier I set this to be:

D:\2500 Git Projects

You probably have a different directory, but I'll run through it on my machine.

This is what I have (Figure 4.13):

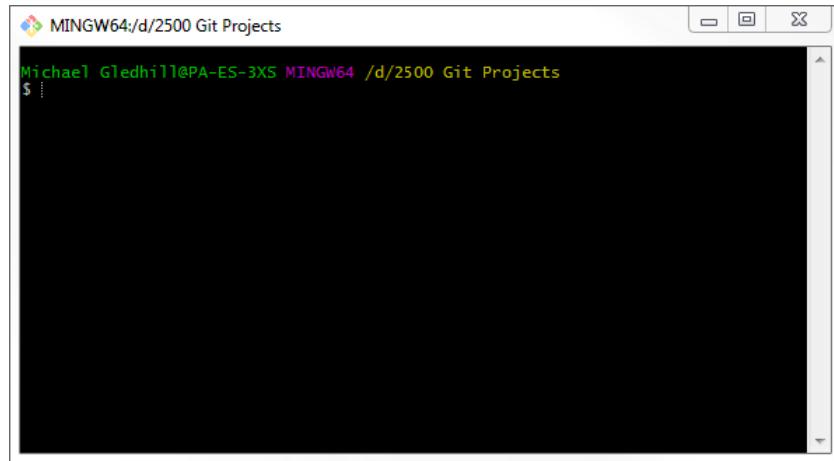


Figure 4.13 Git Bash home directory

Now, while this is the directory we want for our repositories, it is not the place to put the SSH keys, these have to go in the user directory for the current user. In my case, this is (if you remember from § 3.3):

C:\Users\Michael Gledhill\

The easiest way to do this is to, open a Windows command prompt ([WIN+R](#) then type [cmd](#) in the [OPEN](#) box). This will open a Windows command prompt and it will be set automatically to the correct directory. Mine looks like Figure 4.14:

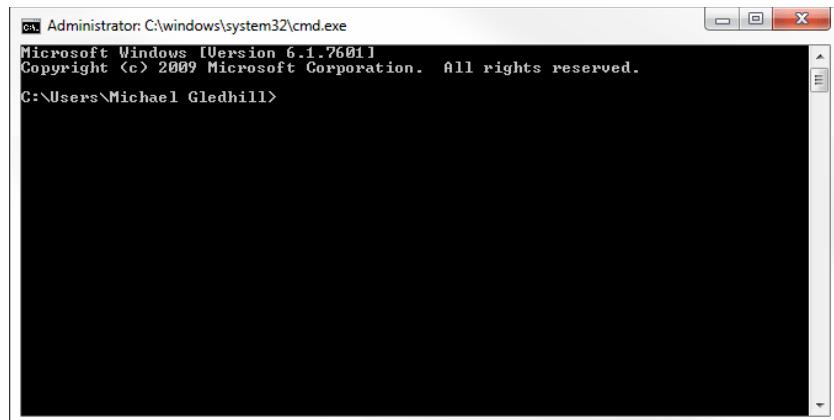


Figure 4.14 Windows command prompt and home directory

Yours will be different but will be similar (probably something like this):

```
C:\Users\<username>\
```

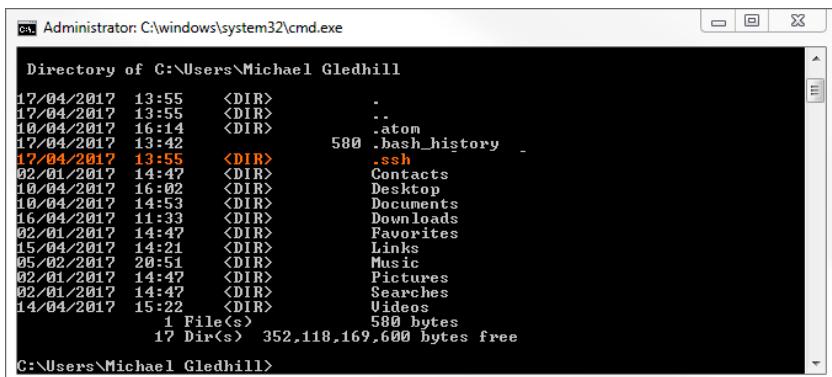
We need to create a directory to store the SSH keys. This directory should be called `.ssh` (note the leading full stop). To create the directory, enter the following command in the Windows command prompt (*obviously, don't enter the C:\Users\Michael Gledhill> bit, just the stuff after it*):

```
C:\Users\Michael Gledhill> mkdir .ssh
```

To see if its worked, enter:

```
C:\Users\Michael Gledhill> dir
```

This will list the directories in the user folder; I have this (I've highlighted the new folder in orange `.ssh`, Figure 4.15):



```
Administrator: C:\windows\system32\cmd.exe
Directory of C:\Users\Michael Gledhill
17/04/2017 13:55    <DIR>      .
17/04/2017 13:55    <DIR>      ..
10/04/2017 16:14    <DIR>      .atom
17/04/2017 13:42    580 .bash_history
17/04/2017 13:55    <DIR>      .ssh
02/01/2017 14:47    <DIR>      Contacts
01/04/2017 16:02    <DIR>      Desktop
10/04/2017 14:53    <DIR>      Documents
16/04/2017 11:33    <DIR>      Downloads
02/01/2017 14:47    <DIR>      Favorites
15/04/2017 14:21    <DIR>      Links
05/02/2017 20:51    <DIR>      Music
02/01/2017 14:47    <DIR>      Pictures
02/01/2017 14:47    <DIR>      Searches
14/04/2017 15:22    <DIR>      Videos
               1 File(s)      580 bytes
               17 Dir(s)   352,118,169,600 bytes free
C:\Users\Michael Gledhill>
```

Figure 4.15 .ssh directory in the users folder

That's all we need, close the Windows command prompt by clicking the cross at the top.

Note: You can't do this from Windows Explorer, Explorer won't let you create a file starting with a full stop (thinks it hasn't got a filename).

Go back to the Git Bash window we opened earlier.

The next thing is to use the SSH keygen command to create the keys. Enter the following:

```
$ ssh-keygen -t rsa -C "your_email@something.com"
```

Use the email address you associated with your GitHub account. In my case I used lab@practicalseries.com. So I entered the command:

```
$ ssh-keygen -t rsa -C "lab@practicalseries.com"
```

I'll go through what happens step by step. Enter the above command and hit return. You will get the following response (or something similar):

```
Michael Gledhill@PA-ES-3XS MINGW64 /d/2500 Git Projects
$ ssh-keygen -t rsa -C "lab@practicalseries.com"
Generating public/private rsa key pair.
Enter file in which to save the key (/c/Users/Michael Gledhill/.ssh/id_rsa): |
```

It's saying it wants to save the file in the `.ssh` directory we just created and that the keys will be in files called `id_rsa`. Hit return:

```
Michael Gledhill@PA-ES-3XS MINGW64 /d/2500 Git Projects
$ ssh-keygen -t rsa -C "lab@practicalseries.com"
Generating public/private rsa key pair.
Enter file in which to save the key (/c/Users/Michael Gledhill/.ssh/id_rsa):
Enter passphrase (empty for no passphrase): |
```

Now it's asking for a passphrase, this is an additional layer of security. If you enter a passphrase¹, you will be prompted for it whenever you connect to the remote repository. I find this too annoying, so I don't use a passphrase.

¹

An SSH passphrase is a phrase that is known only to you and is used to encrypt your private SSH key, a passphrase should be at least 15 characters in length, contain upper and lower case letters, numbers and punctuation. Something like: `H3ct0r & H3nry aRE Stup1D d0g5`. It needs to be something you can remember.

If a passphrase is not used, anyone who has access to your computer or your private SSH key can gain access to your repositories. If a passphrase is used, then the private key is further encrypted and is useless without the passphrase.

If you want to use a passphrase enter it now, hit return and enter it again. If you don't want a passphrase just press return twice. Now you will get something like this:

```
Michael Gledhill@PA-ES-3X5 MINGW64 /d/2500 Git Projects
$ ssh-keygen -t rsa -C "lab@practicalseries.com"
Generating public/private rsa key pair.
Enter file in which to save the key (/c/Users/Michael Gledhill/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /c/Users/Michael Gledhill/.ssh/id_rsa.
Your public key has been saved in /c/Users/Michael Gledhill/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:DePDTaXEW4p8X+5PHRCP9BbwvUvfE1QLgQhGIKYmlrs lab@practicalseries.com
The key's randomart image is:
+---[RSA 2048]---+
|   o .o+.oo o=.=++|
| + . .  o++. O = |
|= . = . .o.o. + =.|
+ .   ooo^ . +o |
.     .S.oo .o+ |
.       ... .o+ |
E         . ...
.       ...
+---[SHA256]---+
```

Figure 4.16 .ssh public and private key creation

What this means is that the private key has been saved in the file `id-rsa` and the public key in the file `id_rsa.pub`. Both files are in the `.ssh` directory we created earlier.

The *key finger print* is a shortened version of the public key and is can be used to identify the key itself—a bit like the seven digit short hash key rather than the full 40 digit hash for a commit.

The *randomart* image is a mechanism that allows two keys to be compared by eye (rather than comparing a long string of numbers and letters).

Note: This isn't actually my SSH key—I'm not that stupid, this is one I made just for demonstration purposes.

The only bit of this we need is the path to the public key, it's highlighted in Figure 4.17:

```

Michael Gledhill@PA-ES-3XS MINGW64 /d/2500 Git Projects
$ ssh-keygen -t rsa -C "lab@practicalseries.com"
Generating public/private rsa key pair.
Enter file in which to save the key (/c/Users/Michael Gledhill/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /c/Users/Michael Gledhill/.ssh/id_rsa.
Your public key has been saved in /c/Users/Michael Gledhill/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:DePDTaXEw4p8X+5PHRCP9BbWvUvfE1QLgQhGIKYmlrs lab@practicalseries.com
The key's randomart image is:
+---[RSA 2048]---+
|   o .o+.oo o=++|
| + . . o++. O =|
|= . . .o.o. + =.|
+ .     ooo^ . +o|
| .      .5.oo  .o+|
| .      . . . .o+|
| E      . . . . |
| .      .. . . |
+---[SHA256]---+

```

Figure 4.17 Public key including its path

We need to open this file in a text editor, the easiest way is to highlight the path in the Git Bash window and copy it (either right click and select **COPY** or hit **CTRL+INSERT**).

Enter the following command (*obviously with your own data*):

```
$ npp "/c/Users/Michael Gledhill/.ssh/id_rsa.pub"
```

Enter **npp "** then paste in the copied path and filename (right click and select **PASTE** or hit **SHFT+INSERT**) followed by a closing quote **"**, like this:

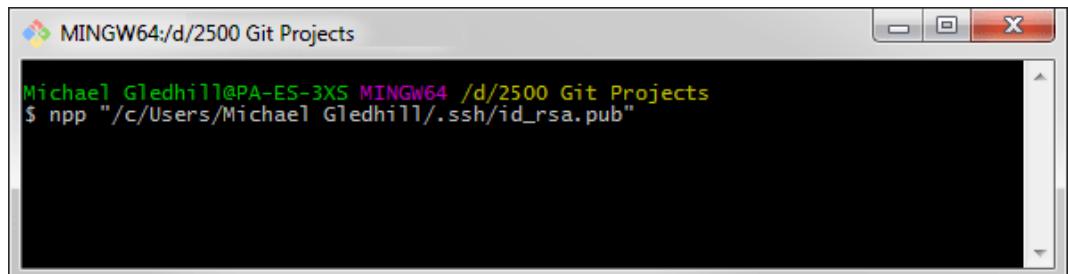


Figure 4.18 Open the public key in Notepad++

Alternatively navigate to the path in Windows Explorer, right click the file and select [EDIT WITH NOTEPAD++](#).

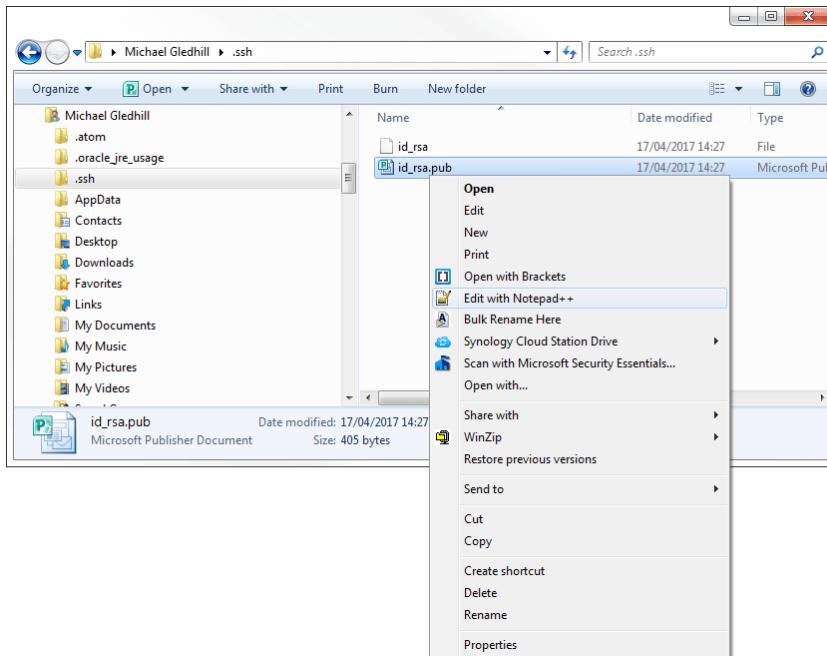


Figure 4.19 Open public key from Windows Explorer

Either way, the file will open in Notepad++. It will look like this:

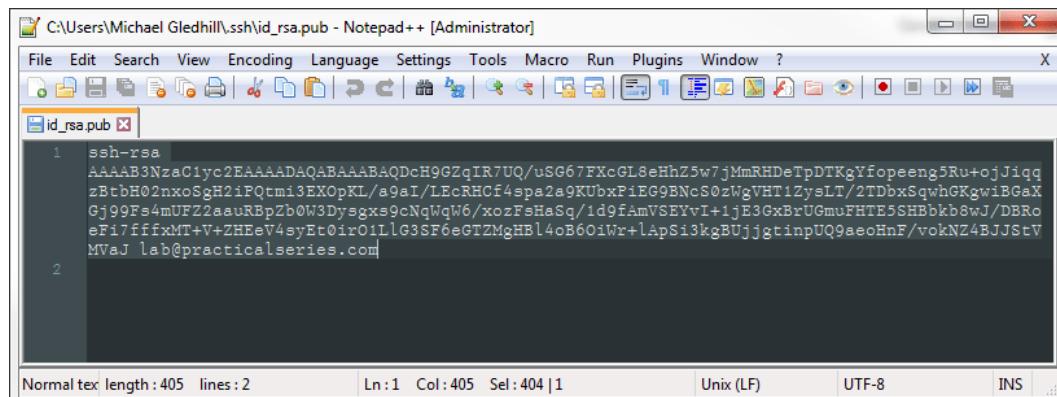


Figure 4.20 The SSH public key in Notepad++

If you can't see it all, if it disappears off the right hand side of the screen, you need to turn on word wrap, click the icon (or **VIEW → WORDWRAP**, about a quarter of the way down).

Select all the text, from the first `s` of `SSH` to the last character of the email address and copy it to the clip board (`EDIT → COPY`, or `CTRL+C`). Now you can close the file.

4.3.2 Adding the public key to GitHub

Go to GitHub and sign in. This will take you to the newsfeed page, this is mine (Figure 4.21):

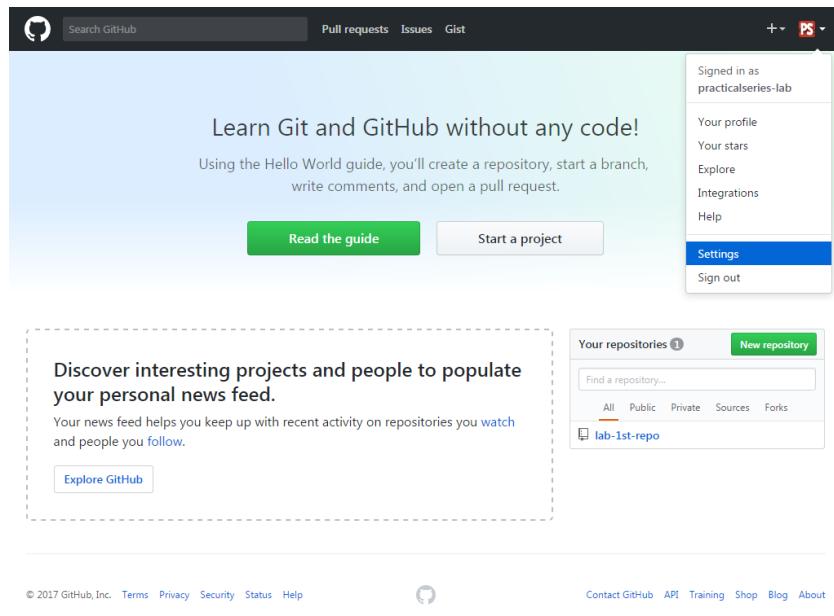


Figure 4.21 GitHub newsfeed page

Click the profile icon at the top right and select `SETTINGS` from the dropdown (again Figure 4.21).

This opens the profile page, again mine looks like this:

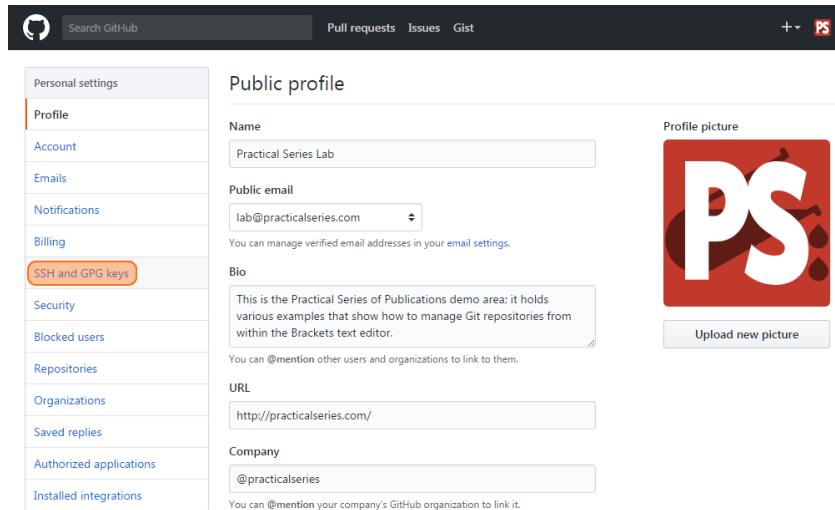


Figure 4.22 GitHub profile settings page

Now click **SSH AND GPG KEYS**, Figure 4.23:

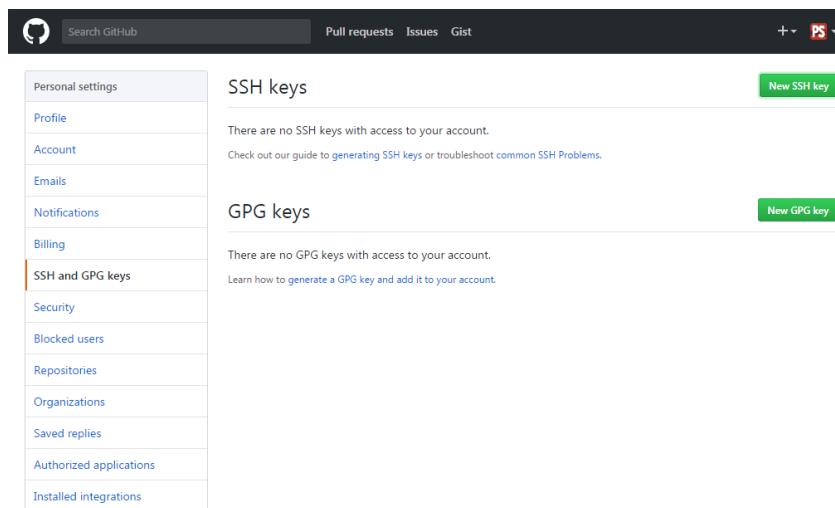


Figure 4.23 GitHub SSH and GPG keys page

Click the green **NEW SSH KEY** button at the top to expand the selection box (Figure 4.24):

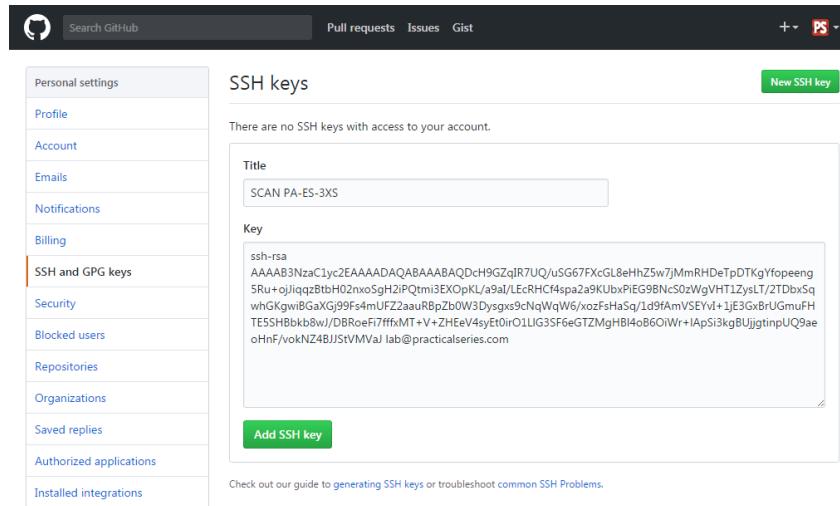


Figure 4.24 GitHub completed SSH and GPG keys page

Click inside the **KEY** area and paste in the public key data copied to the clip board (right click inside it and select **PASTE** or **CTRL+V**).

In the **TITLE** box enter something that identifies the machine the key came from. It doesn't have to be the actual machine's name, it's just so you can identify where the key originated and the machine it is installed on.

That's it, click **ADD SSH KEY**; you will now have to re-enter your GitHub password to confirm the addition (Figure 4.25):

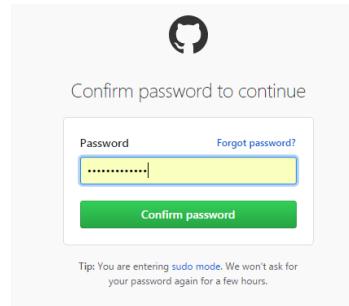


Figure 4.25 GitHub confirm entry

This will take you back to the SSH and GPG keys page and show the new entry (Figure 4.26).

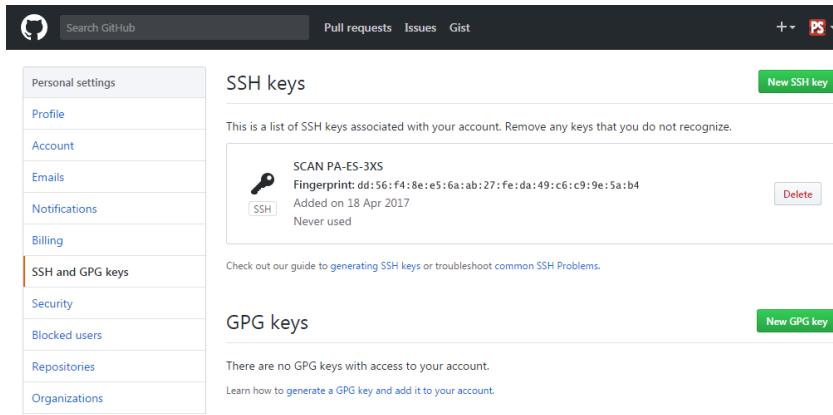


Figure 4.26 The completed SSH key entry

If you intend to use more than one computer to access the GitHub account, you must make a new key on each additional computer and add the key for that computer to GitHub in the same way we have done here.

4.3.3 Testing the SSH connection

Ok, we've set it up; how do we know it's working?—*well back to Git bash*.

Close Git Bash and restart it—this just reinitialises everything.

Now enter the command:

```
$ ssh -T git@github.com
```

This will generate the following response (your finger print will be different, *obviously*):

```
Michael_Gledhill@PA-ES-3XS MINGW64 /d/2500 Git Projects
$ ssh -T git@github.com
The authenticity of host 'github.com (192.30.253.113)' can't be established.
RSA key fingerprint is SHA256:DePDTaXEw4p8X+5PHRCP9BbWvUvfE1QLgQhGIKYmlrs.
Are you sure you want to continue connecting (yes/no)?
```

Figure 4.27 Testing the connection

It's just warning you that it is about to connect to a server and are you sure you want to continue. Well we do, type **YES** and hit **RETURN**.

Now you will get something like this:

```
Michael Gledhill@PA-ES-3XS MINGW64 /d/2500 Git Projects
$ ssh -T git@github.com
The authenticity of host 'github.com (192.30.253.113)' can't be established.
RSA key fingerprint is SHA256:nThbg6kXUpJWG17E1IGOcspRomTxdCARLviKw6E5SY8.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'github.com,192.30.253.113' (RSA) to the list of known
hosts.
Hi practicalseries-lab! You've successfully authenticated, but GitHub does not
provide shell access.

Michael Gledhill@PA-ES-3XS MINGW64 /d/2500 Git Projects
$
```

Figure 4.28 The connection is permanently established

This is good news, `github.com` has been permanently added to the list of known hosts and (in my case) the `practicalseries-lab` user has been authenticated by GitHub. The `github does not provide shell access` warning is fine, we don't expect it to.

If you now go back to the SSH and GPG keys page on GitHub (if you still have it open from before, just refresh it, F5) it. The list of SSH keys will now have changed from `NEVER USED` at the bottom, to `LAST USED WITHIN THE LAST DAY`. The key will also have gone green.



Figure 4.29 Verification at GitHub that the connection is active

So that's it, everything is now linked together and is working properly.

See, easy.

5

GIT INSIDE BRACKETS

How to use Git and GitHub from within the Brackets development environment.

BRACKETS—this is my text editor of choice for creating websites. I discuss why and how to install Brackets on the [PRACTICAL SERIES](#) website (the instructions are summarised in Appendix A). Brackets uses extensions, these are “plugins” that add extra functionality to Brackets, I install some general purpose ones to make web development easier (I list these and the mechanism of installing them [HERE](#) and again in Appendix A.2).

I recommend you install the above extensions; it will make the following examples easier.

In this section I will cover making a new repository in GitHub, making a local copy (*clone*) on a PC and then using Brackets to manage this local copy.

The process will require the installation of another Brackets extension called Brackets-Git, this provides the interface between Brackets and the Git version control system.

This is the main point of this document, this is the bit I want to explain: controlling Git and GitHub from within a Brackets project—all the other things we’ve done up to now, we’ve done just to make this bit work.

5.1

Installing Brackets-Git

The first thing to do is install the Brackets-Git extension in Brackets. Start Brackets and open the Extension Manager by clicking the Lego brick on the right hand sidebar:

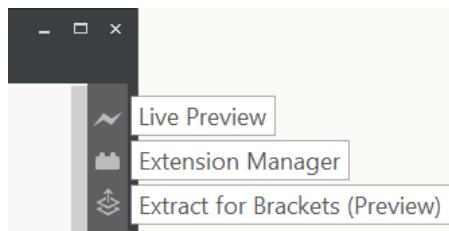


Figure 5.1 Brackets sidebar

In the Extensions Manager, with the **AVAILABLE** tab selected, type **brackets-git** into the search box; the results will be:

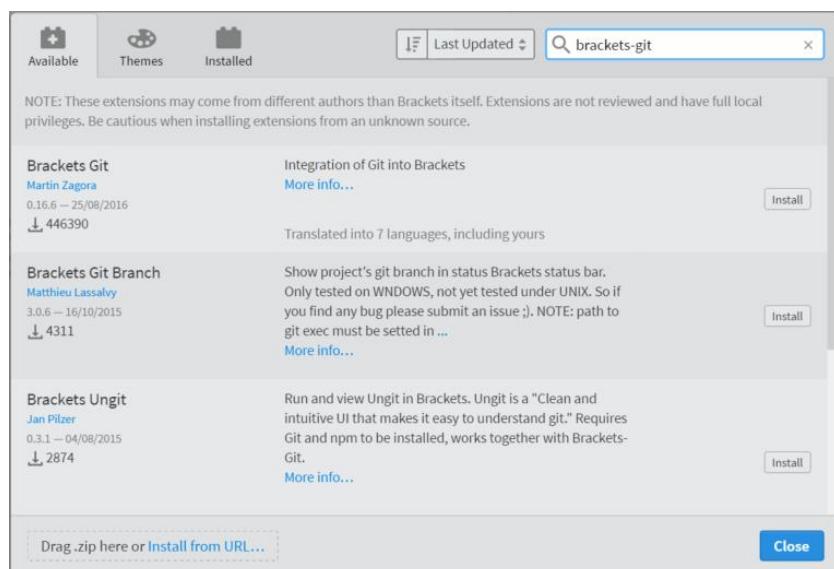


Figure 5.2 Search Brackets extensions

One of the extensions in the resultant list will be called Brackets-Git (by Martin Zagora), in my case it is top of the list. Click the **INSTALL** button next to it.

The installation process has a Git Settings screen with several tabs. The only things I changed were on the first screen (Figure 5.3), I activated **ENABLE ADVANCED FEATURES**; this activates the **reset** to commit functionality of Git within Brackets (see § 7.2). I also selected the **SHOW TERMINAL ICON IN TOOL BAR**, this allows the Git Bash terminal to be started from the sidebar, it puts an icon () in the right sidebar.

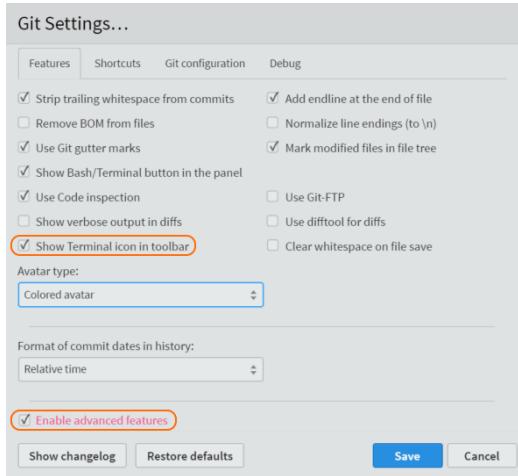


Figure 5.3 Brackets-Git install—Features

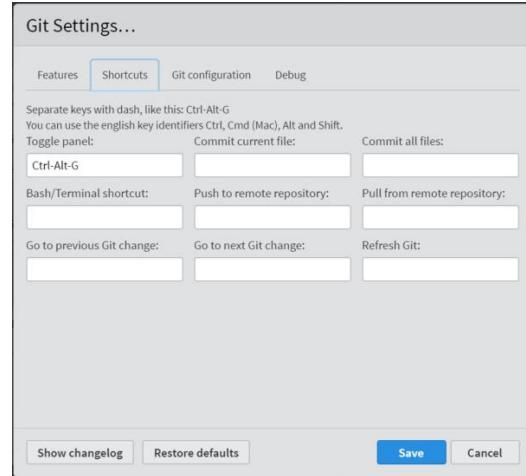


Figure 5.4 Brackets-Git install—Shortcuts

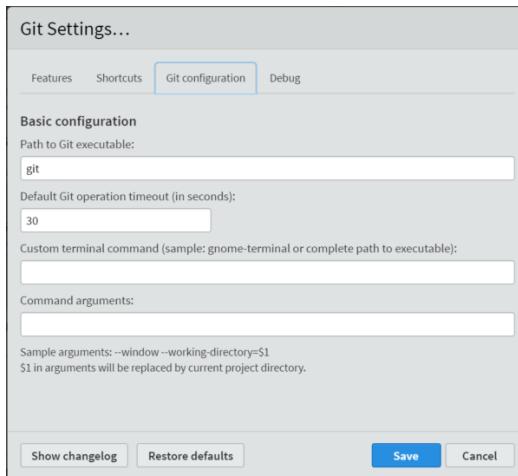


Figure 5.5 Brackets-Git install—Git Config

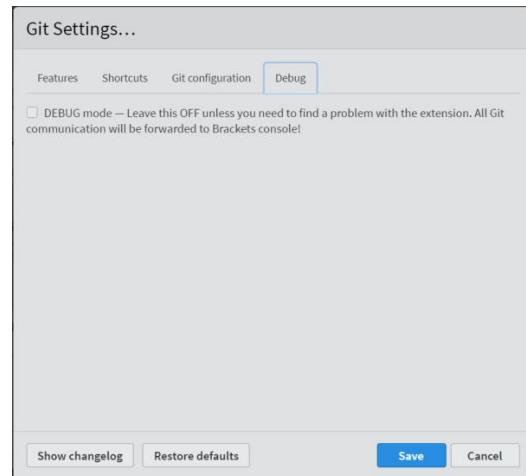


Figure 5.6 Brackets-Git install—Debug

Click **SAVE** and allow Brackets to restart.

One other extension that is useful with Brackets and Git is the Markdown Preview extension (GitHub uses a slightly customised form of Markdown for things like [README.md](#) files, the Markdown Preview shows how these files will look on a GitHub page).

To get the Markdown Preview extension, reopen the extension manager and search for “markdown preview”. The one you are looking for is by Lois Begue:

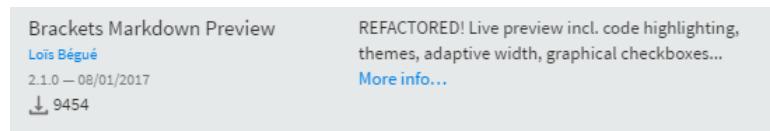


Figure 5.7 Markdown Preview extension

Brackets will now have a Brackets-Git icon and a Markdown Preview icon on the right sidebar (Figure 5.8), it will also have a Git Bash terminal icon if you selected that particular option:

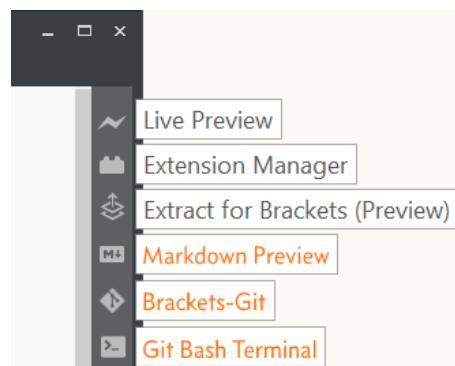


Figure 5.8 Brackets-Git extension icon

The Git Settings (Figure 5.3-Figure 5.6) can be accessed at any time by clicking [FILE → GIT SETTINGS](#).

The Brackets-Git extension itself has its own GitHub repository and can be seen [HERE](#).

5.2

Creating an empty GitHub repository

In this section I will do the following:

- ① Create an empty remote repository using GitHub
- ② Make a local copy (*clone*) of the repository using Brackets-Git
- ③ Populate the local repository with an example project
- ④ Update the GitHub repository by *pushing* the local changes to it

Here we go:

5.2.1 Create a new repository using GitHub

Go to the **GITHUB** website and login to your account. This will take you to your newsfeed page, mine looks like Figure 5.9.

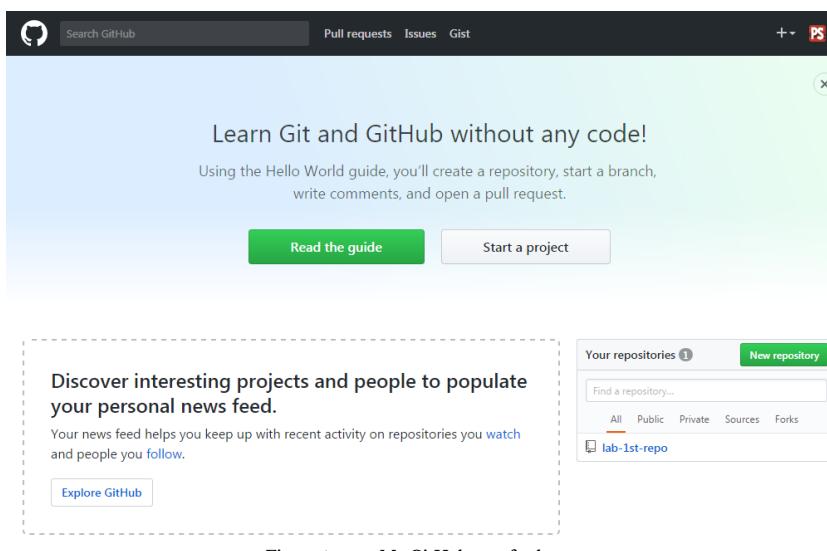


Figure 5.9 My GitHub newsfeed page

We already have a repository ([lab-1st-repo](#)) that we made in the last section, but I'm going to make a new one. I'm going to call it:

`lab-brackets-git`

To do this, click either the **START PROJECT** button or the **NEW REPOSITORY** button (they both do the same).

Either option will move you to the Create a new repository page. I'm going to select the following options (Table 5.1 and Figure 5.10):

PROPERTY	VALUE
Repository name	lab-brackets-git
Description	A repository to demonstrate how to use the brackets-git extension for the Brackets text editor
Public/private	Public
Initialise with README	Ticked (selected)
Add .gitignore	None
Add a license	None

Table 5.1 lab-brackets-git settings

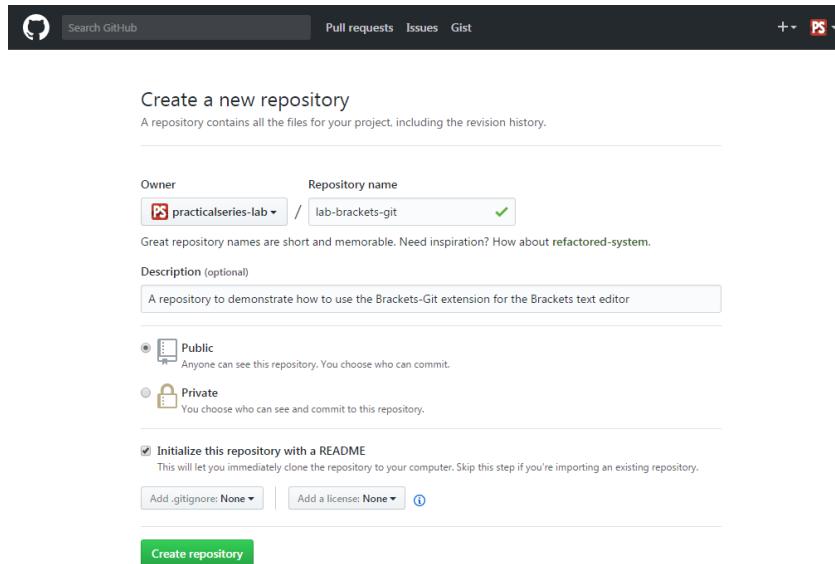


Figure 5.10 GitHub—Create a new repository

Click **CREATE REPOSITORY** and, just like last time, it will automatically open the repository page:

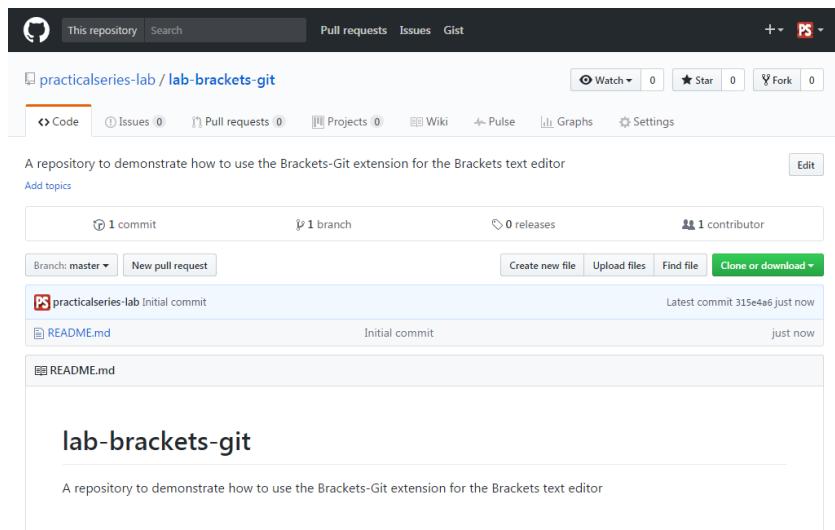


Figure 5.11 GitHub—lab-brackets-git repository page

5.3

Cloning the GitHub repository with Brackets

Before we create a local copy of the GitHub repository (a *clone* in Git terminology § 2.9.4), we need somewhere to put it on our local machine. In my case, all the Git repositories are stored in the directory:

D:\2500 Git Projects

When we use Brackets-Git to clone the repository from GitHub, we must have an empty directory to put it into. In my case I use a local directory with the same name as the GitHub repository¹. Hence I create an empty directory called `lab-brackets-git` within the `2500 Git Projects` folder:

D:\2500 Git Projects\lab-brackets-git

The next thing we need is the SSH link for this repository. In GitHub, navigate to the repository home page (Figure 5.11), click **CLONE OR DOWNLOAD** to open the **CLONE WITH** dialogue box. If it doesn't say **CLONE WITH SSH**, click the **USE SSH** link at the top right of the box (where **USE HTTPS** is in Figure 5.12).

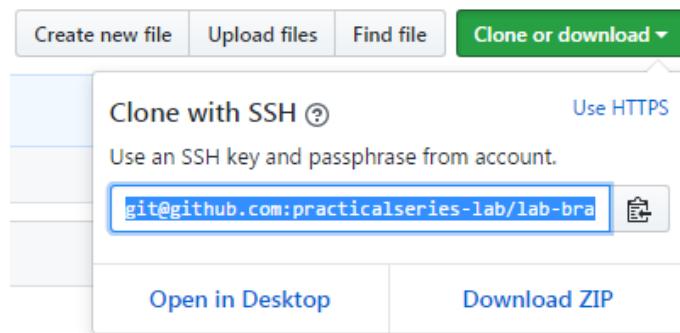


Figure 5.12 GitHub—Clone a repository

¹

The directory can be any name at all, but I find it avoids confusion if I give it the same name as the GitHub repository.

Select the text that starts `git@github.com...` and copy it (**CTRL+C**), or click the paste to clipboard icon .

Next start Brackets. Click **FILE → OPEN FOLDER** and navigate to where you want to create the local repository, in my case this is:

D:\2500 Git Projects\lab-brackets-git

Now click the Brackets-Git icon  on the right hand sidebar, the icon will turn blue and the Git interface window will open in the bottom pane of the Brackets window (Figure 5.13):

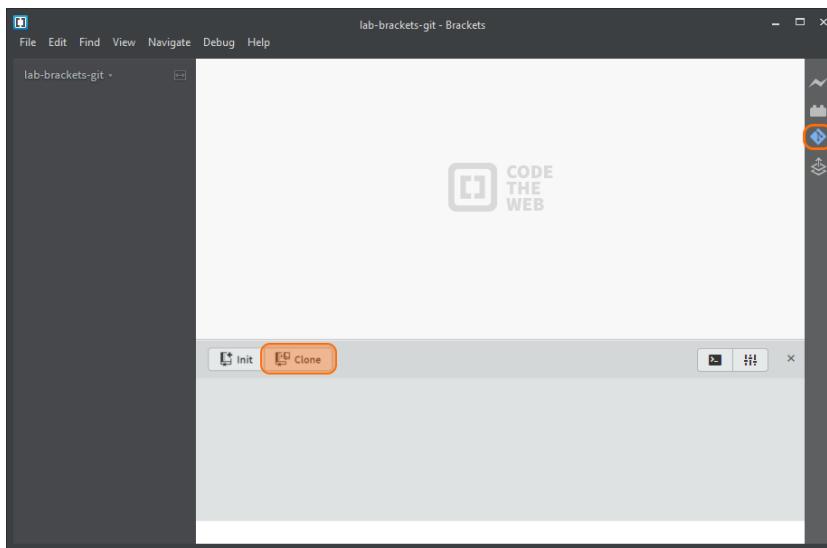


Figure 5.13 Git window in Brackets

Click the **CLONE** button and in the resulting dialog box, paste in the information copied from the **CLONE OR DOWNLOAD** box in GitHub (Figure 5.14).

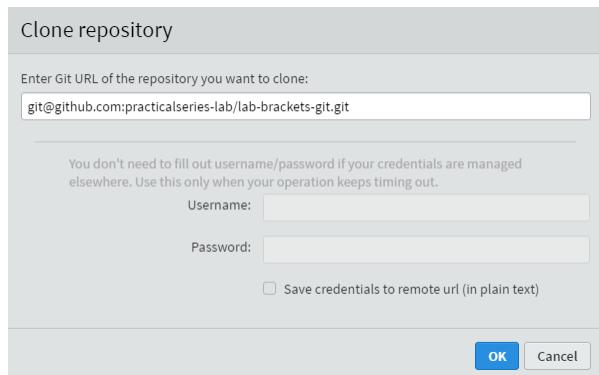


Figure 5.14 Brackets-Git clone repository dialogue box

Click **OK** to start the cloning process (this takes a few seconds) after which, Brackets will display the contents of the repository (Figure 5.15).

Note: *Brackets may ask you for your username and email address (in my case, practicalseries-lab and lab@practicalseries.com). It depends if this is the first connection to this repository or if you have previously logged in to a different GitHub account.*

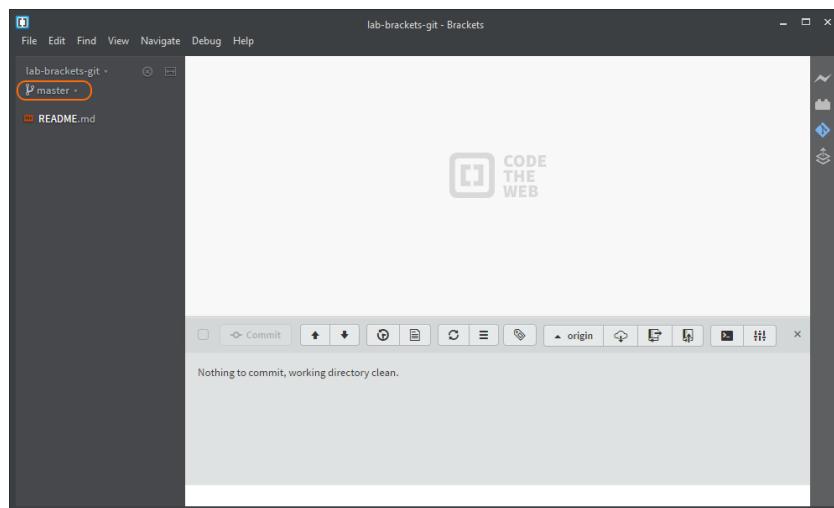


Figure 5.15 Brackets with open Git repository

The clue to this being a repository rather than just some folder is the branch icon on the left hand sidebar (Figure 5.15, highlighted in orange). This shows we are in a Git repository on the master branch.

5.4

Changes commits and push from Brackets-Git

Ok, we've cloned the GitHub [lab-brackets-git](#) repository to a local directory and we're managing the repository with Brackets using the Brackets-Git extension.

The next thing we will do is modify the [README.md](#) file, add some folders (containing images), commit the changes to the [master](#) branch (the only branch we have at the moment) and then push all the changes back to the GitHub remote repository.

5.4.1 Creating a new folder in the repository

Make sure the [lab-brackets-git](#) project is open in Brackets; it should look like Figure 5.15.

Now right click in the left sidebar and select [NEW FOLDER](#) (Figure 5.16):

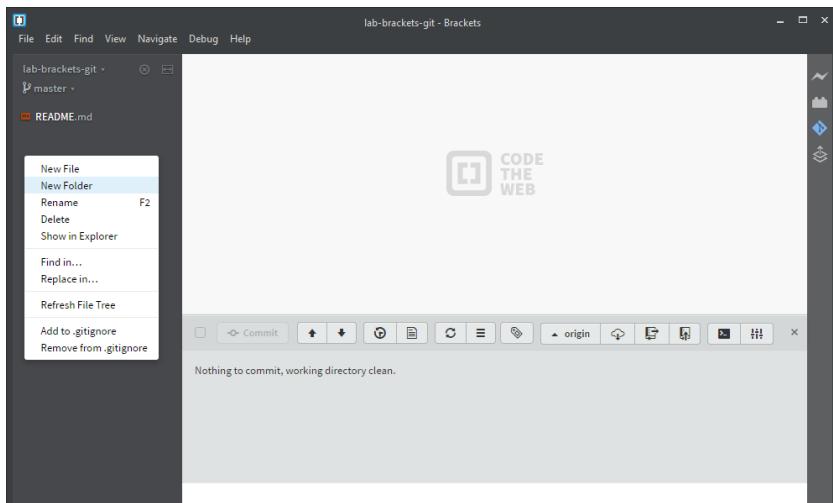


Figure 5.16 Create a new folder

Call the new folder [11-resources](#). Now create another folder inside this new folder called [02-images](#) (i.e. `\lab-brackets-git\11-resources\02-images`). This is just my convention for folders on a website—if you’re interested in the thinking behind it, I explain it [HERE](#).

Externally from Brackets, copy the `cover.png` image (available here [↓](#)) into the new `02-image` folder.

Back in Brackets the system should have detected the change and it will be showing it in the file tree (make sure the `11-resources` and within it, the `02-image` folders are expanded; *just click them if they're not*). It should look like this, Figure 5.17:

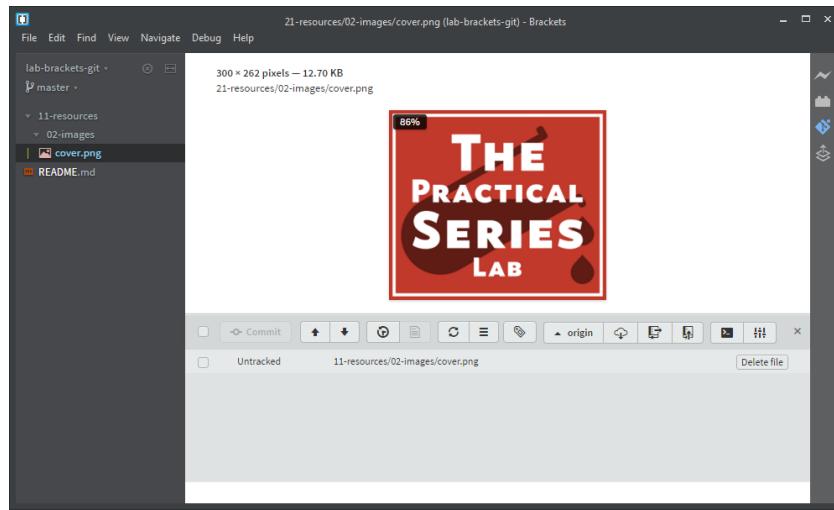


Figure 5.17 Examine the image

If Brackets hasn't automatically refreshed (*sometimes it doesn't with networked files*), right click the left sidebar and select `REFRESH FILE TREE` or just hit `F5`. The Brackets-Git pane (light grey at the bottom) is now showing an untracked file. In my case:

Untracked `11-resources/02-images/cover.png`

This is good, it means that Git has detected the file but doesn't yet know what to do with it (it's a new file as far as the repository is concerned).

5.4.2 Modifying a file

Now let's modify the `README.md` file that GitHub created for us.

GitHub likes `README.md` files; it will automatically display the contents of it on the repository home page. The `.md` extension means the file is written in the *markdown* language.

Markdown is a simple text formatting language (markdown is actually a mark-up language—*go figure*), it uses standard keyboard characters to display headings, emphasis, lists &c. even basic HTML. GitHub uses a slightly enhanced version (called *GitHub Flavoured Markdown*) that can display code fragments and link to issues and users within GitHub. I give a summary of markdown and Git Flavoured Markdown in Appendix C).

In Brackets, double click the `README.md` file to open it.

Replace the default text with the following:

```
 README.MD

1 # A PracticalSeries Publication
2
3 <p align="center">
4   
5 </p>
6
7 The **Practical Series of publications** is a website resource for web developers and
8 engineers. It contains a number of online publications designed to help and explain how
9 to build a website, how to use version control and how to write engineering software for
10 control systems.
11 ## lab-brackets-git: How to use Brackets-Git
12
13 This is a demonstration (teaching) repository that explains how to manage a GitHub repos-
14itory from within the Brackets text editor using the Brackets-Git extension.
```

Code 5.1 README.md file contents

Click **CTRL+S** to save the file.

If you have the markdown preview activated , the `README.md` will be displayed as it will appear on GitHub at the bottom of the screen (Figure 5.18):

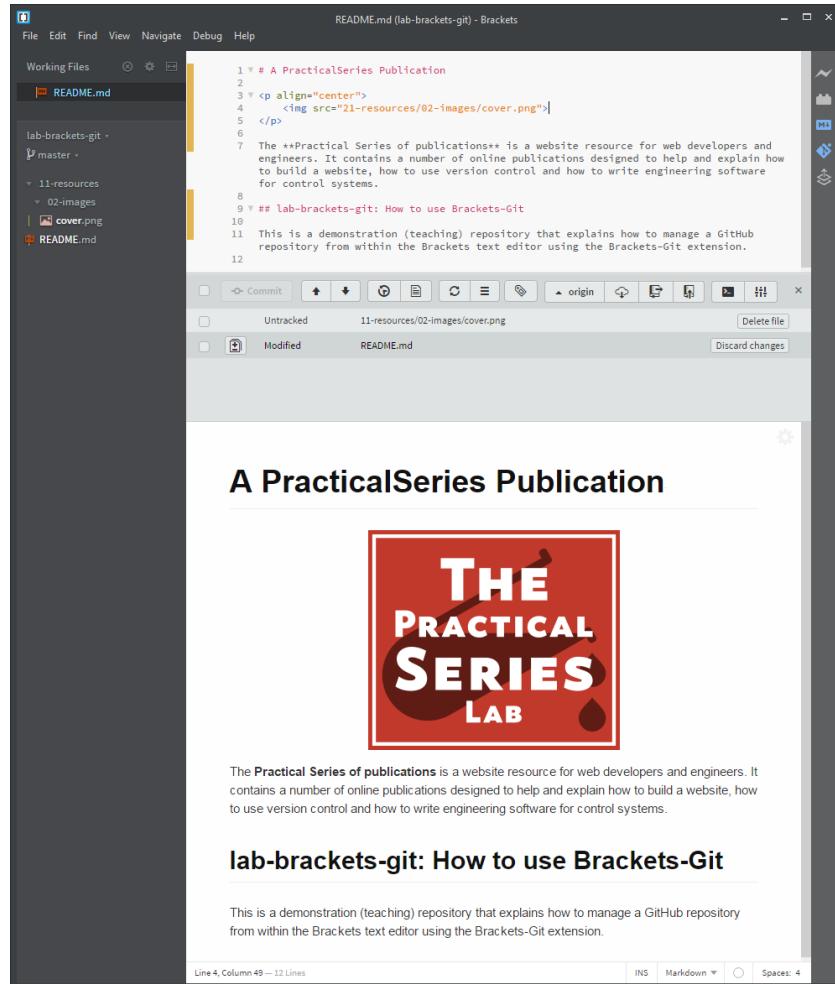


Figure 5.18 Modify the README.md file

In the Git pane, we now have two files listed that need attention:

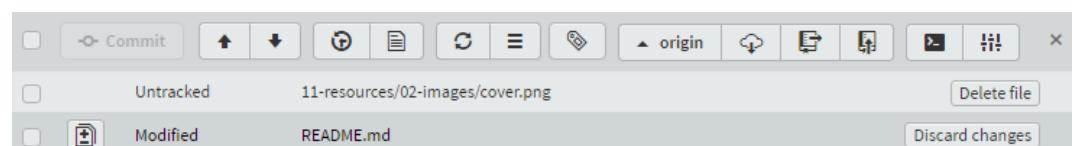


Figure 5.19 New and modified files

5.4.3 Committing the changes

The next thing to do is commit the changes (this accepts the changes and puts them in the local repository).

To do this, tick the boxes to the left of each file (or the single tick box at the top to select all) and click the **COMMIT** button, it looks like this (I've highlighted the boxes and buttons):

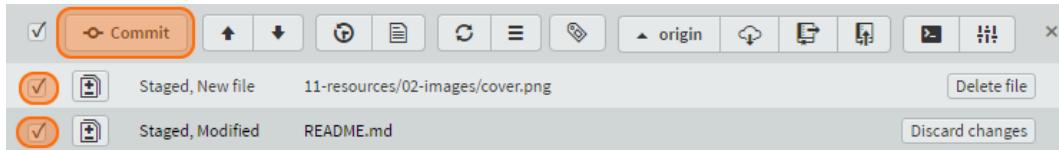


Figure 5.20 Staged files

The Git commit dialogue box will open showing the changes (deletions in red, additions in green). Enter the following commit message:

Cover image added and README updated from default

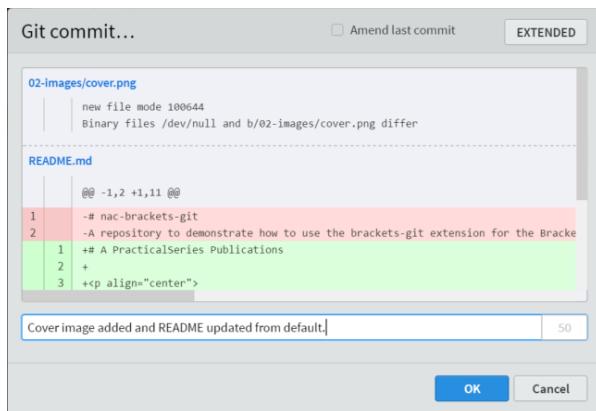


Figure 5.21 Git commit dialogue box

Click **OK** to accept the changes. The Git pane will now show :

Nothing to commit, working directory clean

This is exactly what we want; we have successfully modified the working files and committed them to the local repository.

5.4.4 Pushing the changes back to GitHub

The next thing is to update the remote repository on GitHub to match our local repository. This in GitHub terminology is called a *push*.

It's easy to do from Brackets, just press the **GIT PUSH** button:



The number in brackets indicates the number of commits that will be pushed to GitHub (in our case, just one).

This opens the Push to remote dialogue box:

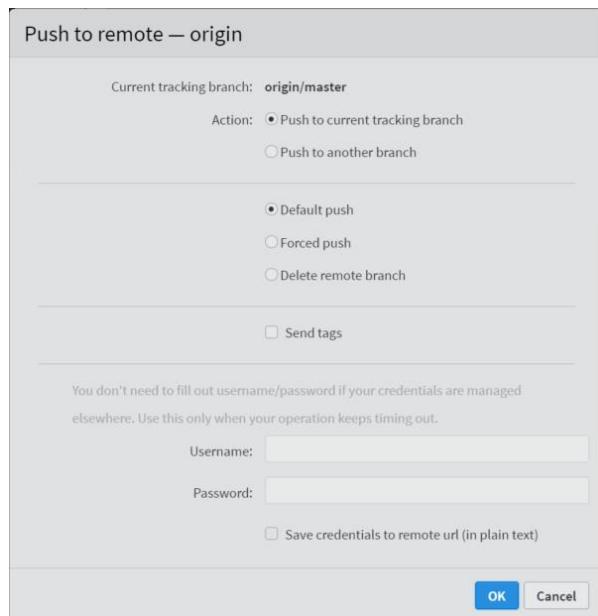


Figure 5.22 Git push to remote dialogue box

Just click **OK**.

Note: If you have previously logged in to a different GitHub account, you will be prompted to enter the username and email address of the repository you are trying to push to (in my case, [practicalseries-lab](#) and lab@practicalseries.com).

After this you should get a successful Git push response (just click **OK** to close it).

Now go back to the GitHub website and navigate to the [lab-brackets-git](#) page.

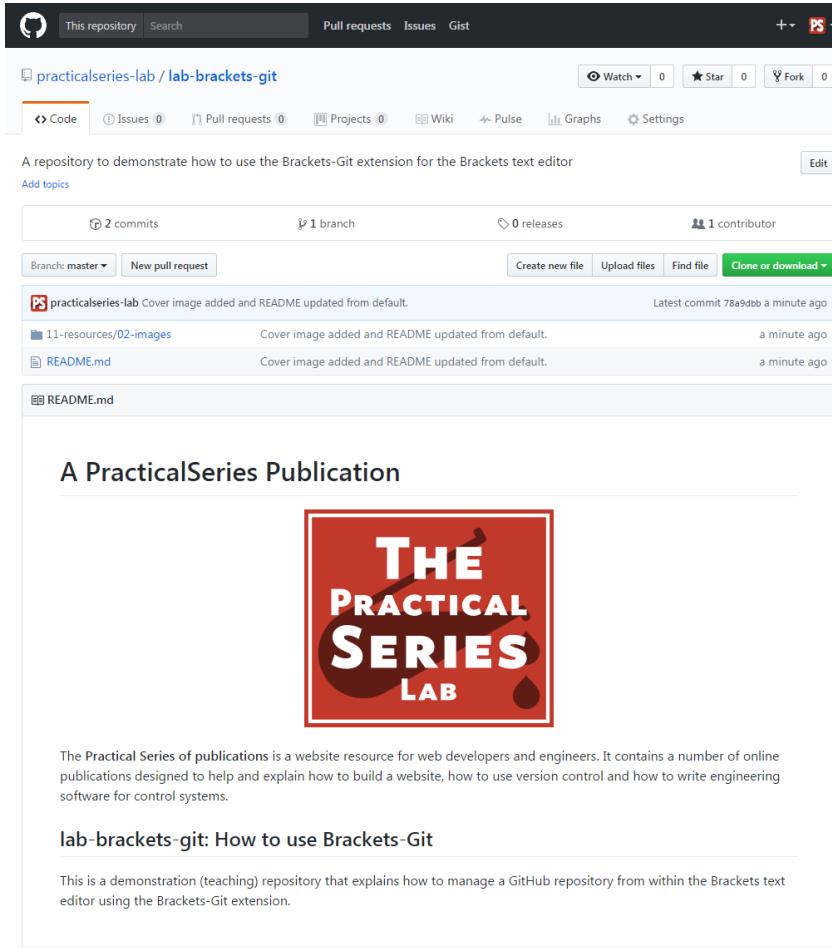


Figure 5.23 Modified page in GitHub

Here we can see the changes we have made, the [README](#) file is being displayed with the changes we made in Brackets, there is a [11-resources/02-image](#) folder and the [cover.png](#) image is being displayed on the [README.md](#) section.

If you're seeing this, it's worked.

6

GIT & BRACKETS A WORKED EXAMPLE

A worked, practical example of using Git within Brackets.

IN THIS SECTION, I use the Brackets-Git extension to create the [lab-01-website](#) discussed in a theoretical manner in Section 2.2.

I will go through this process as a fully worked example demonstrating branching, merging and conflict resolution.

Broadly, I will use Brackets-Git to do the following:

- ① Create a new repository on a local machine
- ② Create the initial website and add it to the repository
- ③ Create new branches to develop additional web pages
- ④ Merge these branches back into the master branch
- ⑤ Explain how to resolve conflicts
- ⑥ Tagging commit points

This section deals entirely with a local repository on a single PC. In subsequent sections I expand this website and incorporate it into a remote repository to demonstrate how Brackets-Git can manage remote as well as local repositories.

Some of what I do here is similar to what I did in the previous chapter—that however was just an introduction. This is a fully worked example and covers Brackets-Git in much more detail.

6.1

Creating the local repository

Let's start with a very simple example, a single page website with a picture; I'm going to call it `lab-01-website`. This is the same website I discussed in a theoretical fashion in Section 2.2. I'm now going to build it for real using Brackets¹.

On my machine I keep all my Git repositories under a single directory, that directory is on my D: drive and is called `2500 Git Projects`. Like this:

D:\2500 Git Projects

I need a new empty directory to keep the repository in. Using Windows Explorer I've created a `lab-01-website` folder under the `2500 Git Projects` directory, thus:

D:\2500 Git Projects\lab-01-website

It looks like this in Windows Explorer (Figure 6.1):

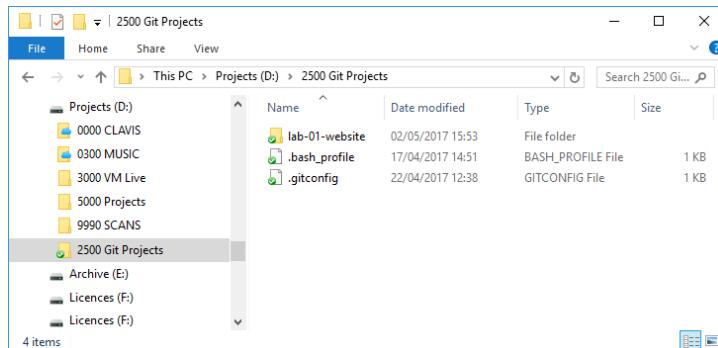


Figure 6.1 A new folder for the repository

¹

I'm just going to refer to it as Brackets rather than Bracket-Git or Brackets with Git. When I say Brackets, I mean Brackets equipped with the Brackets-Git extension installed (see § 5.1).

The next thing is to open the new folder in Brackets, the easiest way is to just right click the folder in Windows Explorer and select **OPEN AS A BRACKETS PROJECT**:

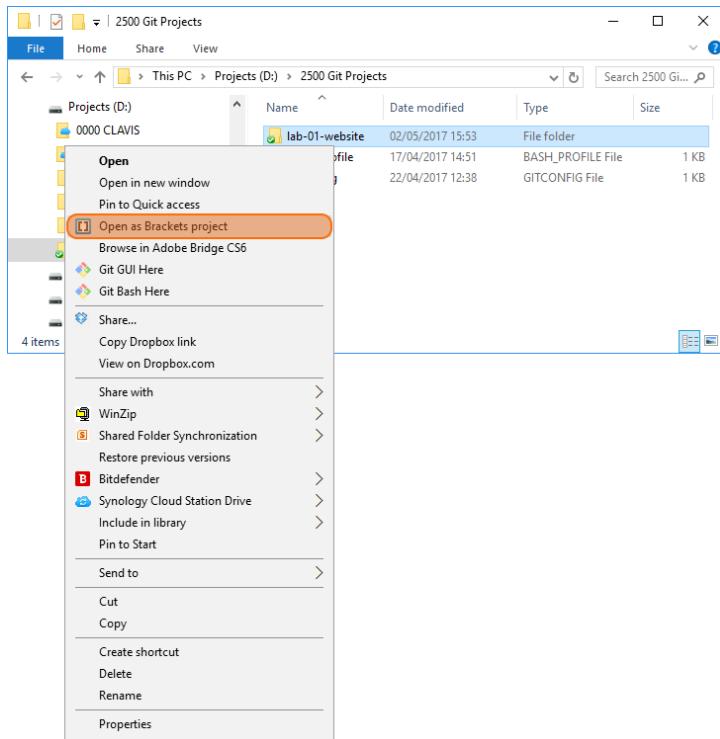


Figure 6.2 Open the new folder in Brackets

This can also be done from within Brackets, click **FILE → OPEN FOLDER...** and navigate to the newly created **lab-01-website** folder and click **SELECT FOLDER**.

Either way, you will have an empty folder open in Brackets. It should look like this:

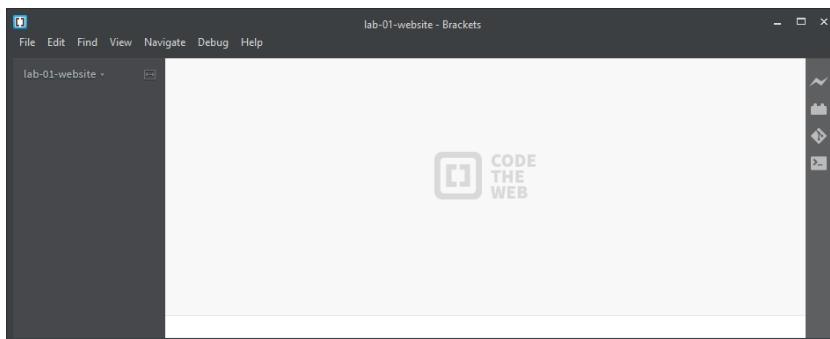


Figure 6.3 lab-01-website project in Brackets

At this point, all we've done is create an empty folder and open it in Brackets. The next thing is to [initialise](#) the folder and make it a Git repository.

Click the Brackets-Git icon  on the right hand sidebar, the icon will turn blue and the Git interface window will open in the bottom pane of the Brackets window (Figure 5.13), I call this the *Git pane*:

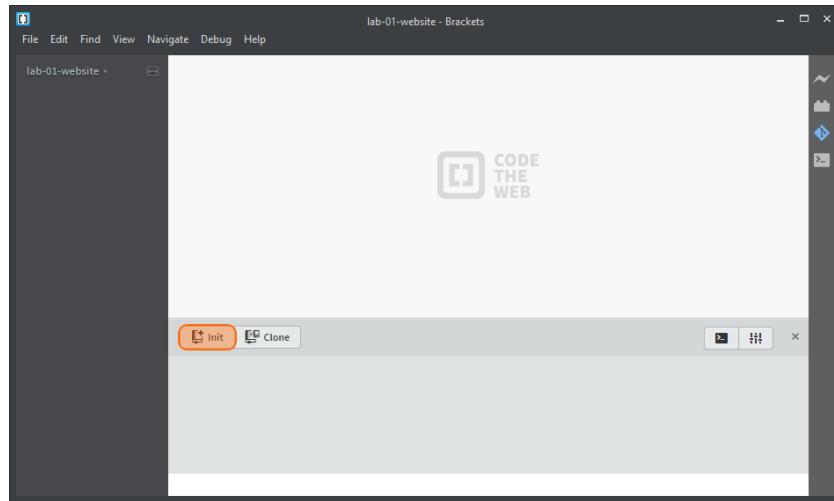


Figure 6.4 Initialise the repository

Click the [INIT](#) button and this will initialise the repository. It will do two things: it will create the `.git` folder inside the `lab-01-website` making this a Git repository. It also creates a `.gitignore` file.

The `.git` folder can be seen by navigating to `lab-01-website`:

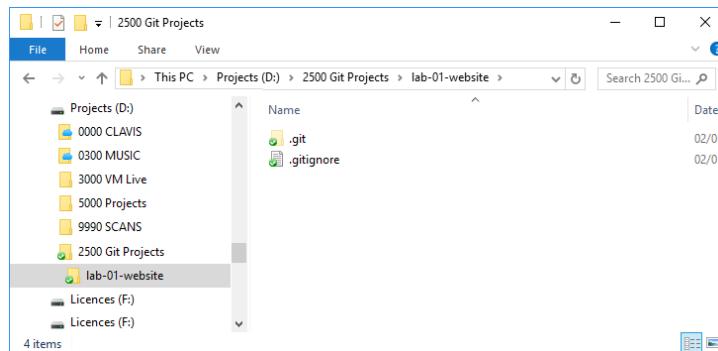


Figure 6.5 The local repository in the new folder

Let's look at what we have in Brackets (Figure 6.6):

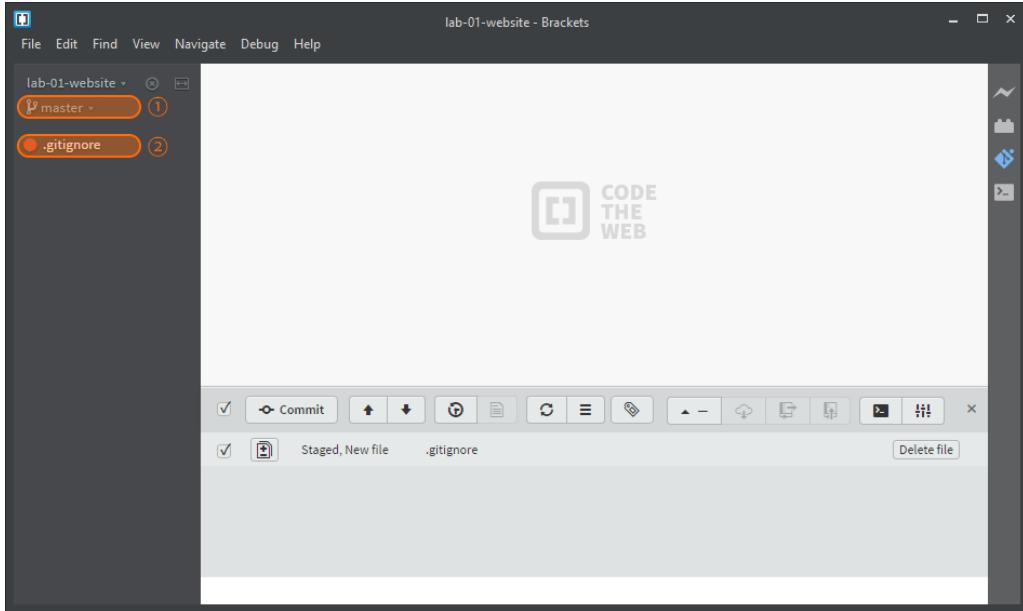


Figure 6.6 A newly created repository in Brackets

There is a couple of things to notice. Firstly a local repository has been created and the active branch is **master** (point ① in Figure 6.6).

Secondly, the new `.gitignore` file is showing in Brackets (point ② in Figure 6.6).

The `.gitignore` file is used to exclude certain files from the Git repository (see § 2.7). Brackets has also detected the presence of the file and wants to add it to the repository. It is flagged as *staged* in the Git pane:

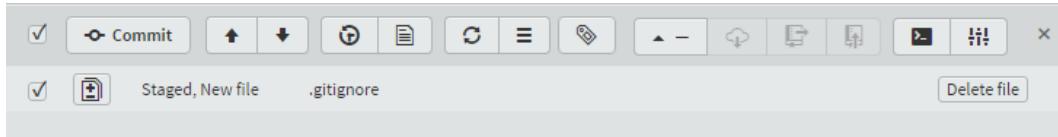


Figure 6.7 Staged files

Don't commit the changes yet, I want to do some work before we commit anything (I go through commits in § 6.3).

So there we are, we've created a new Git repository (*and, best of all, we haven't had to use the command line*).

6.2

Creating the folder structure and initial files

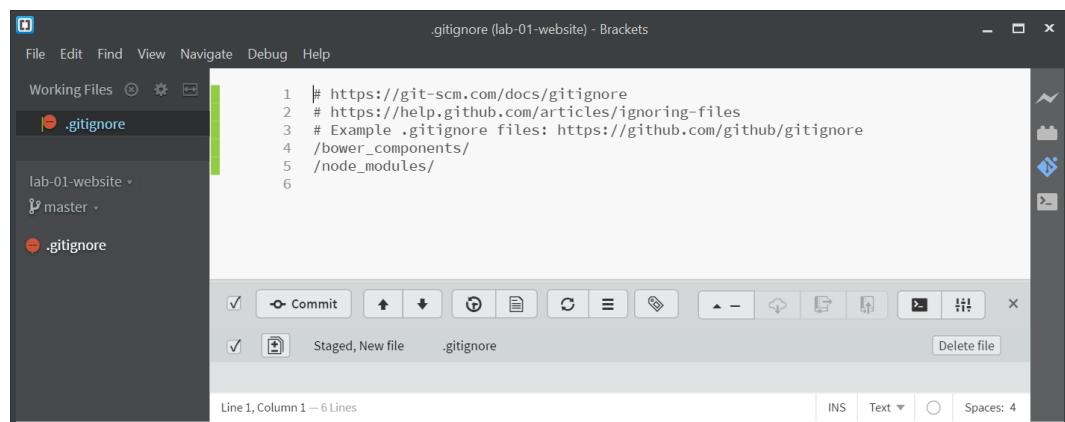
In this section, I'm going to create the project folder structure, add the first set of files for the project to these folders and then make them ready for a first commit to store it all in the local repository.

This is essentially the initial build of the project.

6.2.1 The `.gitignore` file

First thing, since it's there—let's change the `.gitignore` file.

Double click the `.gitignore` file in the left hand pane to open it in Brackets:



```
# https://git-scm.com/docs/gitignore
# https://help.github.com/articles/ignoring-files
# Example .gitignore files: https://github.com/github/gitignore
/bower_components/
/node_modules/
```

Figure 6.8 .gitignore in Brackets

Leave the stuff that's in there (The Brackets-Git extension added these things when it created the file. They don't do any harm, *but I don't exactly know what they do either*) and add the following (lines 6 onwards):

```

.GITIGNORE

1 # https://git-scm.com/docs/gitignore
2 # https://help.github.com/articles/ignoring-files
3 # Example .gitignore files: https://github.com/github/gitignore
4 /bower_components/
5 /node_modules/
6
7 # Windows OS Files
8 Thumbs.db
9 Desktop.ini
10
11 # Mac OS Files
12 .DS_Store

```

Code 6.1 .gitignore modifications

Save the file ([FILE → SAVE](#) or [CTRL+S](#) or, if you have [AUTOSAVE](#), just click outside the Brackets window). The Git pane will recognise the change and will now show the file as [untracked](#) (it will no longer be in the staging area).

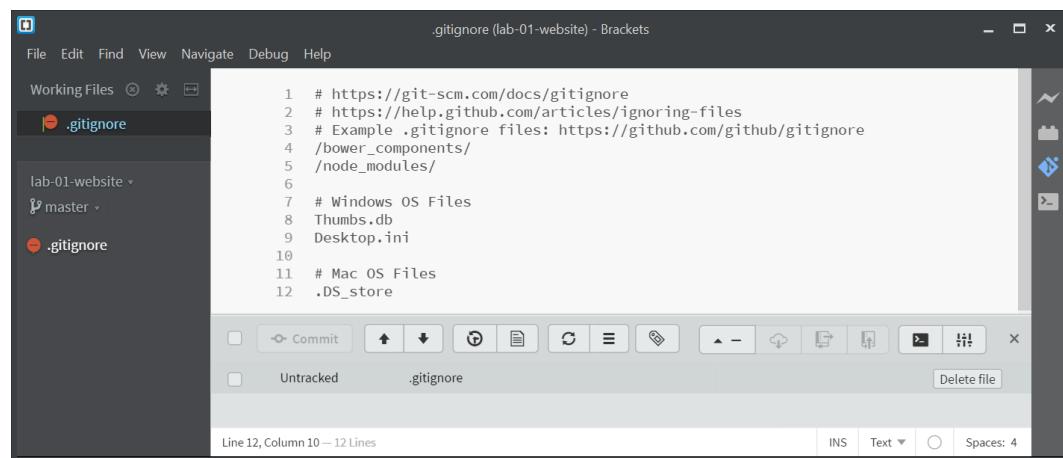


Figure 6.9 File status change Brackets

The little green line that shows next to the file in the left hand file tree of Brackets indicates that the file has been modified (doesn't match the file in the repository). The green line will disappear when we commit the changes and everything matches.

6.2.2 Creating a folder structure

Now let's add some folders and images to the project. Git handles folders quite well¹. The folder structure I'm looking for is shown in Figure 6.10.

Note: *I'm not showing the .git folder.*

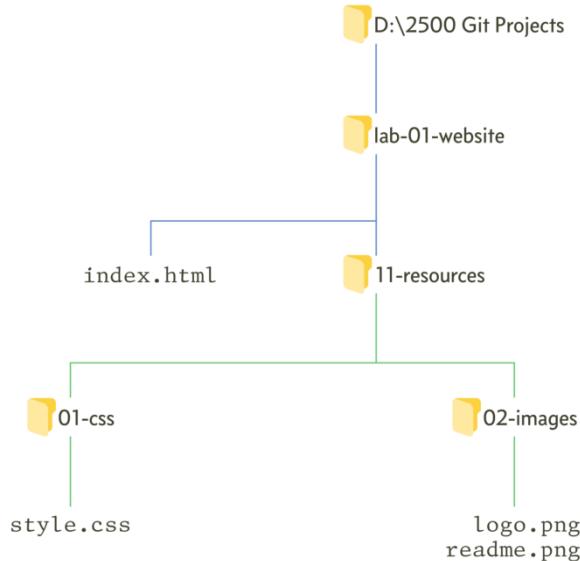


Figure 6.10 lab-01-website Folder structure

To create the first folder (`11-resources`) right click in the file tree (left pane of Brackets) and select **NEW FOLDER** (Figure 6.11).

¹

Git will always keep files in the correct folder and different folders can hold files of the same name—it all works as you would expect. Git doesn't however, track empty folders; empty folders are ignored by Git and won't be in the repository. This leads to dummy files, placeholders (*lieutenants if you will—in England it is pronounced “leff-tenant”, sorry Americans, you've been saying it wrong all these years*) being used to make Git correctly store empty folders. There is an informal convention that these placeholder files are called `.gitkeep`. It is a convention I use.

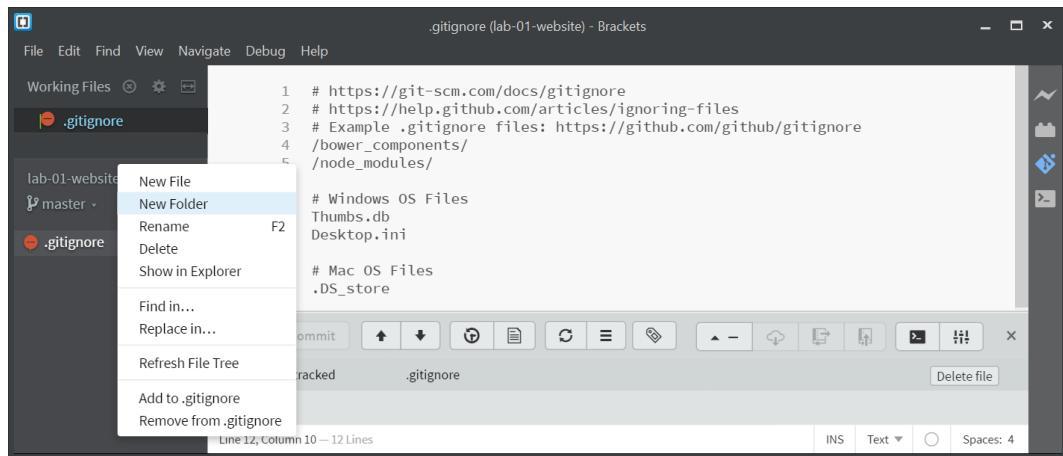


Figure 6.11 Brackets, create folder

Call the new folder **11-resources** (it will be highlighted in the left hand tree view).

Now right click the new **11-resources** folder and create another new folder within it, call this one **01-css**. Then repeat the process (right click **11-resources** again) and create a second folder called **02-images**. The final thing should look like this:

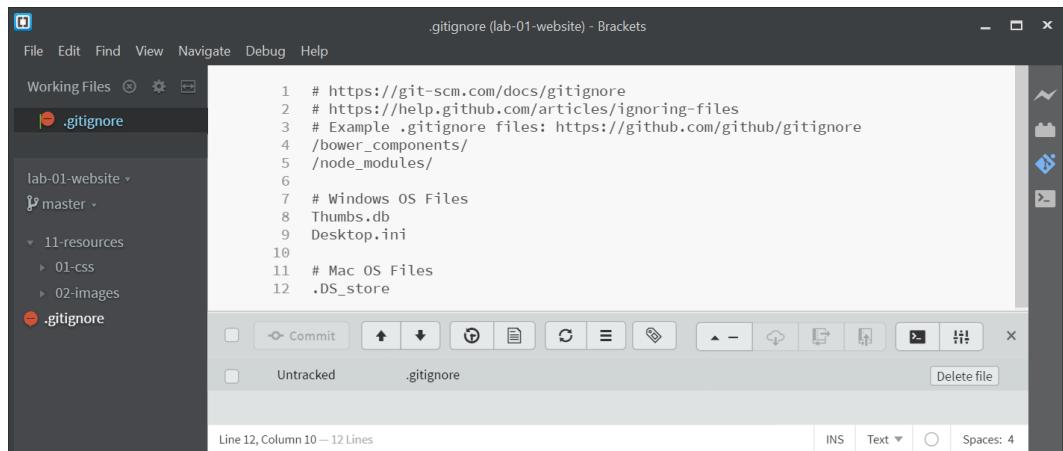


Figure 6.12 Brackets, final folder structure

Note: Although the new folders are showing in Brackets, they are not listed in the Git pane; this is because they are empty. We'll fill them soon.

That's the folder structure in place. Next thing is to add some files.

6.2.3 Adding images to the project

The first thing is the two images (`logo.png` and `readme.png`). They look like this:



Figure 6.13 logo.png



Figure 6.14 readme.png

These two files can be downloaded here [↓](#). Copy and paste them into the `02-images` folder (the easiest way is with Windows Explorer).

Brackets should automatically recognise that two new files are present and update the left hand file tree accordingly. If it doesn't (sometimes it is slow with networked drives) right click the left sidebar and select **REFRESH FILE TREE** or just hit **F5**.

Either way it should look like this (click the little arrow heads at the side of the folder names in the left hand file tree to expand them), Figure 6.15:

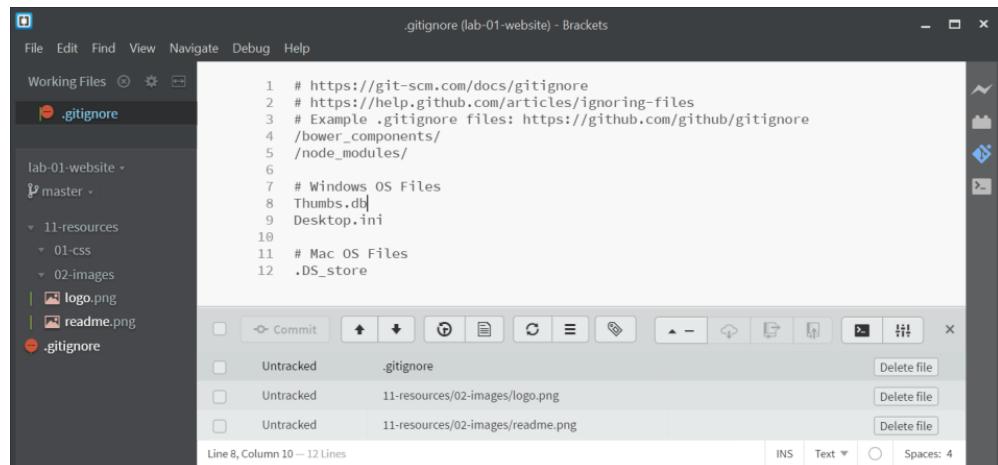


Figure 6.15 Brackets, with image files

We can also see that the two files have been detected by Git and are correctly flagged as **untracked**.

6.2.4 The `README.md` file

I haven't really discussed the `README.md` file. I mentioned it briefly in section 5.4 and made some changes by way of example. The `README.md` file was introduced with GitHub. GitHub likes readme files. It likes them to such an extent that if one lives in the root folder it will be displayed on the home screen for that repository.

Now I know that this is a local repository and isn't on GitHub; but I'm still going to create a `README.md` file. I think they're a good idea and we will be putting this project into a remote GitHub repository in a subsequent section (section 8).

So what is a `README.md` file? Well it's a file that describes what the project is and how to use it.

The `README.md` file is a Markdown file (*ironically this is actually a version of a mark-up language*) hence the `.md` extension. It is widely used for README files. It allows the text in the file to be formatted by using simple character combinations:

MARKDOWN	RESULT
<code># This is an h1 heading</code>	<h1>This is h1 heading</h1>
<code>## This is a h2 heading</code>	<h2>This is a h2 heading</h2>
<code>##### This is a h6 heading</code>	<h6>This is a h6 heading</h6>
<code>*Text in italics*</code>	<i>Text in italics</i>
<code>**Text in bold**</code>	Text in bold
<code>1. Item 1</code>	1. Item 1
<code>2. Item 2</code>	2. Item 2
<code>3. Item 3</code>	3. Item 3
<code> * Item 3a</code>	○ Item 3a
<code> * Item 3b</code>	○ Item 3b

Table 6.1 Basic markdown

GitHub uses a slightly more advanced version of Markdown, this is called *Git Flavoured Markdown* and allows images, code fragments and some directly encoded HTML.

Appendix C gives a summary and some examples of both Markdown and Git Flavoured Markdown.

The `README.md` should contain certain specific things:

- A title
- An Introduction (what the project is for)
- A Table of Contents (TOC)
- Instructions for use
- Installation instruction (if required)
- Links to other documentation (if not all contained here)
- List of contributors
- Licence (unless listed in another document)

Let's create the readme file. It needs to be in the root of the project folder.

In Brackets, right click the bottom of the file tree and click new file and then rename the new file as `README.md`².

Double click the new file and add the following code to it:

²

The `README.md` file is by convention written with README in capitals (I think to make it noticeable). There is no actual requirement for this; GitHub will quite happily use `readme.md` instead. If both are present, it will display `README.md` in preference to `readme.md`.

README.MD

```
1 # A PracticalSeries Publication
2
3 <p align="center">
4   
5 </p>
6
7 The **Practical Series of publications** is a website resource for web developers and
8 engineers. It contains a number of online publications designed to help and explain how
9 to build a website, how to use version control and how to write engineering software for
10 control systems.
11 This particular repository is designed as an example project to demonstrate how to build
12 a Git and GitHub repository using the Brackets-Git extension for the Brackets text
13 editor.
14 The full set of PracticalSeries publications is available at
15 [practicalseries.com] (http://practicalseries.com "Practical Series Website").
16
17 ## How to use this repository
18 This repository is a worked example demonstrating how to build a version control project
19 using Git and GitHub from within the Brackets text editor.
20 This repository is intended to be used with the accompanying documentation
21 [practicalseries Git and GitHub] (http://practicalseries.com/0021-git-vcs/index.html
22 "Practical Series - Git and GitHub").
23
24 ## Contributors
25 This repository was constructed by [Michael Gledhill] (https://github.com/practicalseries-lab "Michael Gledhill").
26
27 ## Licence
28 This is simply a demonstration repository, the contents are free to use by anyone who
29 wishes to do so.
```

Code 6.2 README.md file contents

Again when you save the file, Git will detect it and it will appear in the Git pane marked as **untracked**.

If you have the Markdown Preview activated , the **README.md** will be displayed as it will appear on GitHub at the bottom of the screen (Figure 6.16):

File Edit Find View Navigate Debug Help

Working Files

- .gitignore
- README.md

lab-01-website

master

- 11-resources
 - 01-css
 - 02-images
 - logo.png
 - readme.png
- .gitignore
- README.md

```

1 v # A PracticalSeries Publication
2
3 v <p align="center">
4   
5 </p>
6
7 The **Practical Series of publications** is a website resource for web
developers and engineers. It contains a number of online publications designed
to help and explain how to build a website, how to use version control and how
to write engineering software for control systems.
8 This particular repository is designed as an example project to demonstrate how
to build a Git and GitHub repository using the Brackets-Git extension for the
Brackets text editor.
9 The full set of PracticalSeries publications is available at
\[practicalseries.com\]\(http://practicalseries.com "Practical Series Website"\).
10
11 v ## How to use this repository
12 This repository is a worked example demonstrating how to build a version control
project using Git and GitHub from within the Brackets text editor.
13 This repository is intended to be used with the accompanying documentation
\[practicalseries Git and GitHub\]\(http://practicalseries.com/0021-git-vcs/index.html "Practical Series - Git and GitHub"\).
14
15 v ## Contributors
16 This repository was constructed by \[Michael Gledhill\]\(https://github.com/mgledhill "Michael Gledhill"\).
17
18 v ## Licence
19 This is simply a demonstration repository, the contents are free to use by
anyone who wishes to do so.

```

Commit

<input type="checkbox"/>	Untracked	.gitignore	<input type="button" value="Delete file"/>
<input type="checkbox"/>	Untracked	11-resources/02-images/logo.png	<input type="button" value="Delete file"/>
<input type="checkbox"/>	Untracked	11-resources/02-images/readme.png	<input type="button" value="Delete file"/>
<input type="checkbox"/>	Untracked	README.md	<input type="button" value="Delete file"/>

A PracticalSeries Publication

Figure 6.16 README file in Brackets

The actual [README](#) file is shown in full in Figure 6.17:

A PracticalSeries Publication



The **Practical Series of publications** is a website resource for web developers and engineers. It contains a number of online publications designed to help and explain how to build a website, how to use version control and how to write engineering software for control systems. This particular repository is designed as an example project to demonstrate how to build a Git and GitHub repository using the Brackets-Git extension for the Brackets text editor. The full set of PracticalSeries publications is available at practicalseries.com.

How to use this repository

This repository is a worked example demonstrating how to build a version control project using Git and GitHub from within the Brackets text editor. This repository is intended to be used with the accompanying documentation [practicalseries Git and GitHub](#).

Contributors

This repository was constructed by [Michael Gledhill](#).

Licence

This is simply a demonstration repository, the contents are free to use by anyone who wishes to do so.

Figure 6.17 README file in full

That's it for the [README .md](#) file.

6.2.5 The `index.html` file

Nearly there, just two more files to go.

First `index.html`. Create a new file in the root directory (same place as `.gitignore` and `README.md`). Call it `index.html` and add the following code:

```
INDEX.HTML

1 <html lang="en">                                <!-- Declare language -->
2   <head>                                         <!-- Start of head section -->
3     <meta charset="utf-8">                         <!-- Use Unicode character set -->
4
5     <link rel="stylesheet" type="text/css" href="11-resources/01-css/style.css">
6
7     <title>PracticalSeries: Git Lab</title>
8   </head>
9
10  <body>
11    <h1>A Practical Series Website</h1>
12
13    <figure class="cover-fig">
14      
15    </figure>
16
17    <h3>A note by the author</h3>
18
19    <p>This is my second Practical Series publication--this one happened by accident
20      too. The first publication is all about building a website, you can see it here.
21      This publication came about because I wanted some sort of version control
22      mechanism for the first publication.</p>
23
24    <p>There are lots of different version control systems (VCS) out there; some are
25      free, some are commercial applications just google it. If you do, you will find
26      that Git and GitHub show up again and again.</p>
27
28  </body>
29 </html>
```

Code 6.3 index.html

Save the file and again confirm that it is recognised by Brackets, that it is in the Git pane and is marked as **untracked** (Figure 6.18):

The screenshot shows the Brackets IDE interface with the file 'index.html' open. The left sidebar displays a file tree for a repository named 'lab-01-website' with a single commit 'master'. The commit details show files: .gitignore, README.md, and index.html. The right pane contains the code editor with the following content:

```

1 <html lang="en">
2   <head>
3     <meta charset="utf-8">
4   </head>
5   <link rel="stylesheet" type="text/css" href="11-resources/01-
css/style.css">
6
7   <title>PracticalSeries: Git Lab</title>
8 </head>
9
10  <body>
11    <h1>A Practical Series Website</h1>
12
13    <figure class="cover-fig">
14      
15    </figure>
16
17    <h3>A note by the author</h3>
18
19    <p>This is my second Practical Series publication--this one happened by
accident too. The first publication is all about building a website, you
can see it here. This publication came about because I wanted some sort
of version control mechanism for the first publication.</p>
20
21    <p>There are lots of different version control systems (VCS) out there;
some are free, some are commercial applications just google it. If you
do, you will find that Git and GitHub show up again and again.</p>
22
23  </body>
24 </html>

```

The bottom status bar indicates 'Line 24, Column 8 — 24 Lines'.

Figure 6.18 index.html file in Brackets

6.2.6 The `style.css` file

Finally, the `style.css` file.

The `style.css` file needs to be created in the `11-resources\01-css\` folder. If the folder isn't visible, expand the folder by clicking the small arrow next to the folder name in the left hand file tree view. Once you can see the `01-css` folder right click it and select **NEW FILE**. Rename the new file `style.css`. Add the following code:

```

1 * {
2     margin: 0;
3     padding: 0;
4     box-sizing: border-box;
5     position: relative;
6 }
7
8 html {
9     background-color: #fbfaf6; /* Set cream coloured page background */
10    color: #404030;
11    font-family: serif;
12    font-size: 26px;
13    text-rendering: optimizeLegibility;
14 }
15
16 body {
17     max-width: 1276px;
18     margin: 0 auto;
19     background-color: #fff; /* make content area background white */
20     border-left: 1px solid #eddede;
21     border-right: 1px solid #eddede;
22 }
23
24 h1, h2, h3, h4, h5, h6 { /* set standard headings */
25     font-family: sans-serif;
26     font-weight: normal;
27     font-size: 3rem;
28     padding: 2rem 5rem 2rem 5rem;
29 }
30 h3 { font-size: 2.5rem; }
31
32 .cover-fig { /* holder for cover image */
33     width: 50%;
34     margin: 2rem auto;
35     padding: 0;
36 }
37 .cover-fig img {width: 100%;} /* format cover image */
38
39 p { /* TEXT STYLE - paragraph */
40     margin-bottom: 1.2rem; /* *** THIS SETS PARAGRAPH SPACING *** */
41     padding: 0 5rem;
42     line-height: 135%;
43 }

```

Code 6.4 style.css

Save the file and again confirm that it is recognised by Git. Git now has six **untracked** files (Table 6.2 and Figure 6.19); all files are displayed in the Brackets file tree:

FILE (AND PATH)	STATUS
.gitignore	Untracked
11-resources/01-css/style.css	Untracked
11-resources/02-images/logo.png	Untracked
11-resources/02-images/readme.png	Untracked
README.md	Untracked
index.html	Untracked

Table 6.2 List of all files

The screenshot shows the Brackets IDE interface. The left sidebar displays a file tree for a project named 'lab-01-website' with a single branch 'master'. Inside 'master', there's a folder '11-resources' containing '01-css' and '02-images', along with files '.gitignore', 'index.html', and 'README.md'. The main editor window shows the content of 'style.css':

```

1 * {
2   margin: 0;
3   padding: 0;
4   box-sizing: border-box;
5   position: relative;
6 }
7
8 html {
9   background-color: #fbfaf6; /* Set cream coloured page background */
10  color: #404030;
11  font-family: serif;
12  font-size: 26px;
13  text-rendering: optimizeLegibility;
14 }
15
16 body {
17   max-width: 1276px;
18   margin: 0 auto;
19   background-color: #fff; /* make content area background white */
20   border-left: 1px solid #eddede;
21   border-right: 1px solid #eddede;
22 }
23
24 h1, h2, h3, h4, h5, h6 { /* set standard headings */
25   font-family: sans-serif;
26   font-weight: normal;
27   font-size: 3rem;
28   padding: 2rem 5rem 2rem 5rem;
29 }
30 h3 { font-size: 2.5rem; }
31

```

Below the editor, a list of untracked files is shown in the 'Working Files' panel:

- .gitignore
- 11-resources/01-css/style.css
- 11-resources/02-images/logo.png
- 11-resources/02-images/readme.png
- README.md
- index.html

Each item has a checkbox and a 'Delete file' button next to it. The status bar at the bottom indicates 'Line 26, Column 9 — 43 Lines'.

Figure 6.19 style.css and all six untracked files

These files are now ready to be staged and committed. *That's next.*

6.3 The first commit

We've reached the point where we have six files that we are ready to **commit** to the repository. These are the files listed in Table 6.2. All are listed as **untracked**. This is apparent by looking at the Git pane in Brackets:

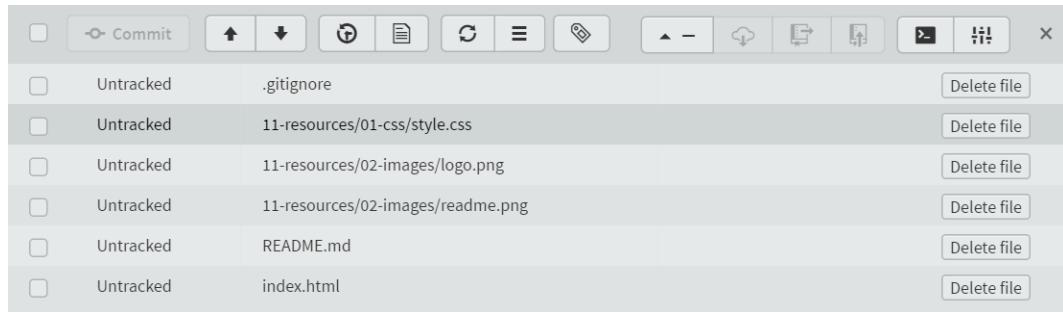


Figure 6.20 All six untracked files

When we worked through this example in Section 2.2, the first thing we did after creating them was add them to the staging area—this was a process in its own right with Git and similar to the commit process itself (it used the **add** command to stage the files in a similar way to using the **commit** command to add them to the local repository).

6.3.1 Staging the files

With Brackets it is a bit easier, to stage the files just click the tick box next to the file in the Git pane. If I do this for the **.gitignore** file its status will change to **staged**.

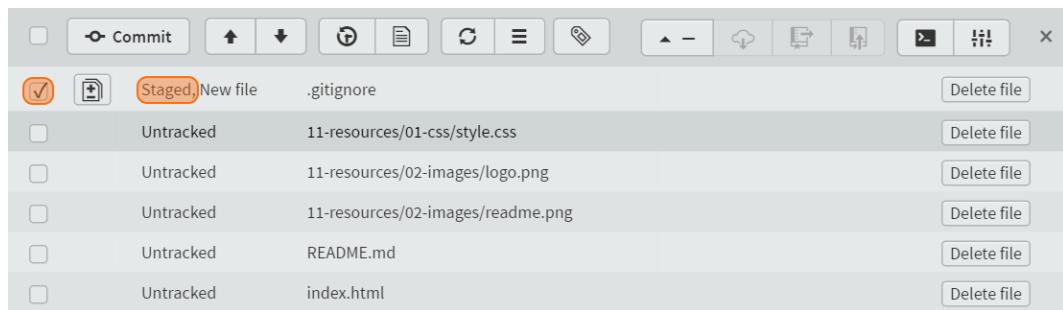
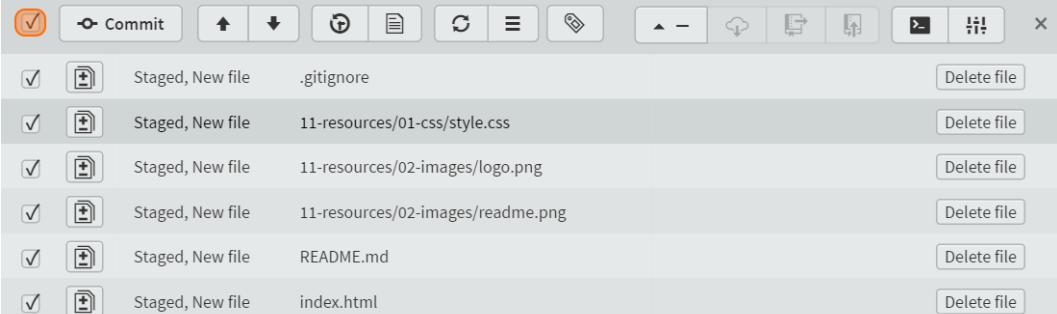


Figure 6.21 A staged file in Brackets

We actually want to stage all six files, this can be done individually or with the common tick box at the top (next to the commit button), this will select all the files in the list. It looks like this:



The screenshot shows the Brackets IDE interface with a list of files in the center pane. At the top, there is a toolbar with various icons and a 'Commit' button. Below the toolbar, a list of files is displayed, each with a checked checkbox in the first column, indicating they are staged. The files listed are: '.gitignore', '11-resources/01-css/style.css', '11-resources/02-images/logo.png', '11-resources/02-images/readme.png', 'README.md', and 'index.html'. To the right of each file name, there is a 'Delete file' button.

<input checked="" type="checkbox"/>	 Commit	Staged, New file	.gitignore
<input checked="" type="checkbox"/>	 Commit	Staged, New file	11-resources/01-css/style.css
<input checked="" type="checkbox"/>	 Commit	Staged, New file	11-resources/02-images/logo.png
<input checked="" type="checkbox"/>	 Commit	Staged, New file	11-resources/02-images/readme.png
<input checked="" type="checkbox"/>	 Commit	Staged, New file	README.md
<input checked="" type="checkbox"/>	 Commit	Staged, New file	index.html

Figure 6.22 Stage all the files in Brackets

The **diff** icon , if clicked will open a window showing the changes made to the associated file, it's a bit meaningless here, all the files are new so everything in them is a difference.

To *unstage* a file just untick the box. If you modify a staged file, it will automatically be unstaged and you will have to tick the box again.

The next thing is to commit the files.

6.3.2 Making the first commit

Ok, we've got all six files staged and we want to commit them. It's easy, just press the **COMMIT** button . Don't worry, you still get a chance to cancel out.

Note: *Only staged files will be committed, those with a tick in the box.*

This opens the Git commit dialogue box:

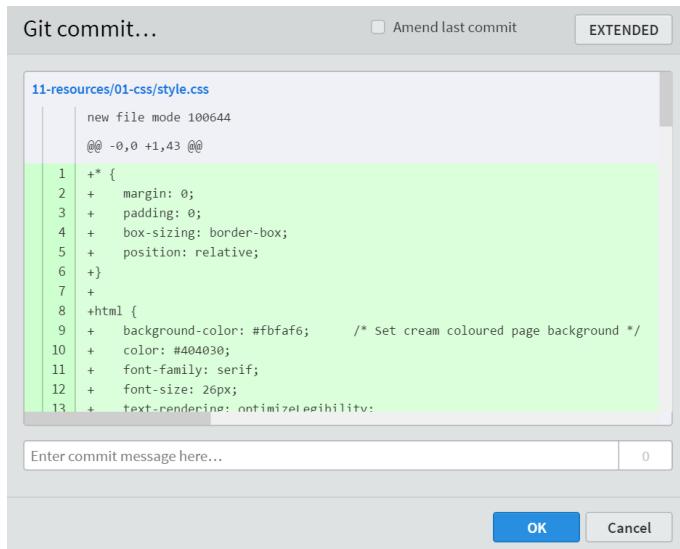


Figure 6.23 Brackets Git commit dialogue box

Let's look at what we've got.

The most obvious thing is the bit in the middle (the green bit), this shows all the changes to each of the staged files, scroll down to see all the changes in each file one after the other. This bit is here to make sure you know what you are about to commit (*I suspect most people are like me and don't bother looking*).

The thing this dialogue box is really asking for is the bit in the box underneath this. *The commit message*. The commit message is very important; it explains the reason for the change. I have my own style for commit messages; I explain it in Appendix B, feel free to use it or use whatever style you prefer.

This dialogue box allows the commit message to be entered as either a single line (how it looks in Figure 6.23) or if you click the EXTENDED button at the top **EXTENDED** you can enter multiple lines:

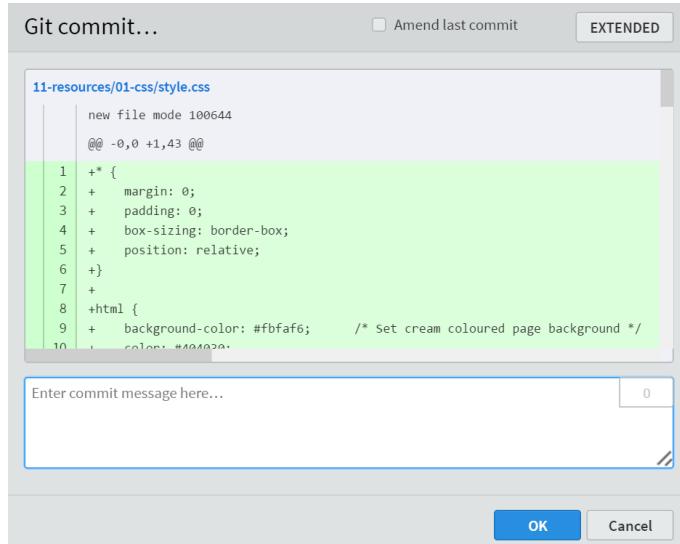


Figure 6.24 Brackets Git extended commit message box

This is my message:

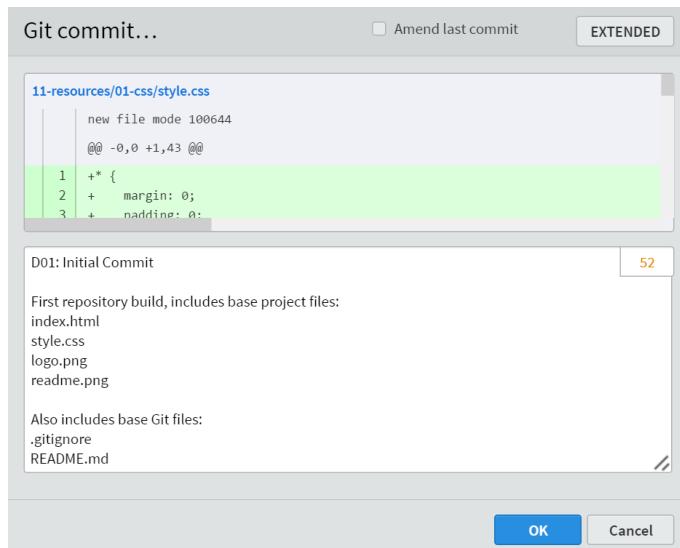


Figure 6.25 Initial commit message

The golden rule of commit messages is that the first line should be less than 50 characters long¹ (*that's what the number is in the box at the top right of the message area—unfortunately it doesn't really work when using the extended box, it counts everything*). If entering a multiline message, make sure there is a blank line after the first line.

Commit the changes by pressing the **OK** button (*this is the point of no return*).

A note on the first commit with Brackets

Usually, with the first commit, Brackets will ask you to change the username and email for the repository. This is because we haven't set them yet for this project (it didn't ask us when we initialised the project and for some reason it doesn't just adopt the ones we setup in the Git installation² § 3.4.1). So we have to give it the information. In my case my username is `practicalseries-lab` and email is `lab@practicalseries.com`. Enter your own (*obviously*).

The entry boxes look like Figure 6.26:

¹

There are some other basic rules for commit messages that apply to all messages (and not just to my particular style of message). These are:

- 1 The first line should be less than 50 characters.
It can be more, but best practice is not to make it too long otherwise it gets cut off in the display area
- 2 If using an extended commit message, always leave a blank line after the first line—this ensures the first line is treated as a title for the commit
- 3 When using extended comments, keep each line short (75 characters max) otherwise these can also be cut off

²

I did a bit of digging with this; I think Brackets want's to have the username and email in the local configuration file (`config`), see § 3.3.1, this is inside the `.git` folder (*where we never go*). Remember we put the username and email in the `.gitconfig` file in home directory. If the information is not in the `config` file Brackets will ask for it.

There is some sense to this, it means Brackets can switch repositories and adopt the username and email for the new project automatically.

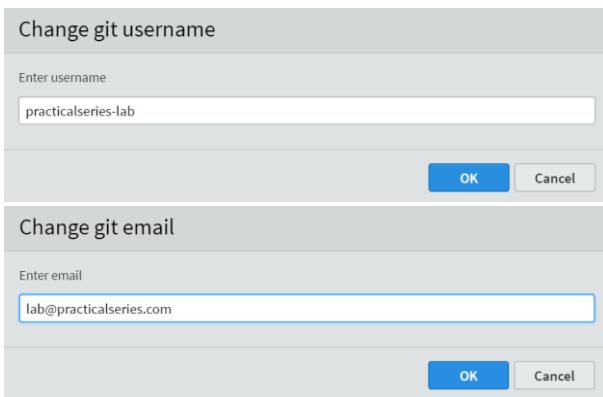


Figure 6.26 Enter username and email for the repository

Enter the information and click **OK** each time.

Unfortunately, the commit **will not** have been made, it takes you back to the staging area and you have to click the **COMMIT** button again—don't worry, you won't have to re-enter the commit message, it will still be there. Just click **OK**.

That's it we've committed the changes to the repository.

Brackets now looks like this:

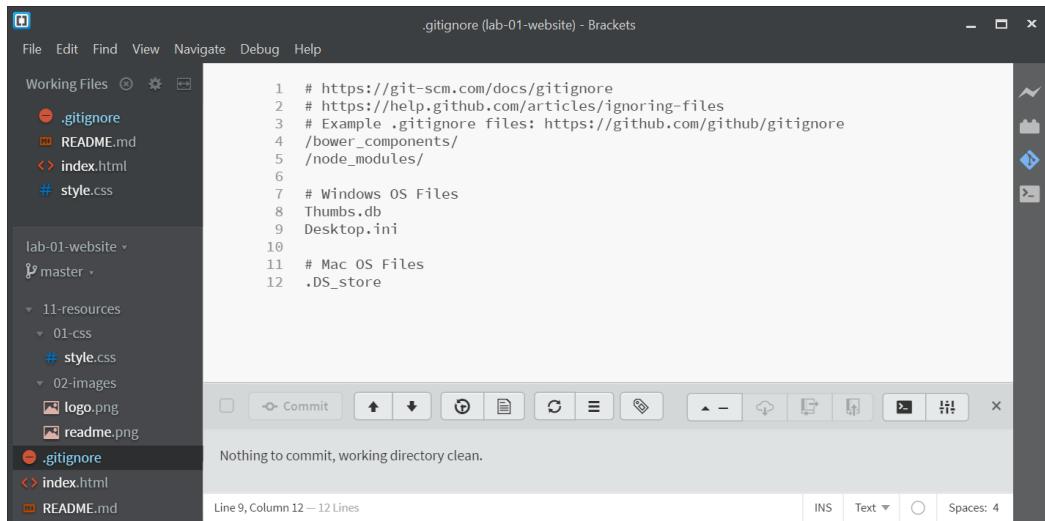


Figure 6.27 Brackets, post commit

Brackets is now reporting **Nothing to commit, working directory clean.**

6.3.3 Viewing the commit history

Seeing the commit history is easy, just click the  (SHOW HISTORY) button. It opens the current history in the Git pane:



Figure 6.28 Brackets, post commit detail

There are a couple of things to see here, first is the commit number on the right, in my case it is [8ce2e8e]. The second is the commit message, it shows the first line of the commit message we entered in the previous section.

It also shows who made the commit ([practicalseries-lab](#)) and when (if you don't like the *2 hours ago* style, it can be changed, see § 7.1.1). The white P on an orange background is generated by Brackets, it is just the first letter of the username on a coloured background—it allows commits made by different users to be readily identified.

Clicking anywhere on a commit line in the Git pane, opens a larger commit information pane above it. I.e.:

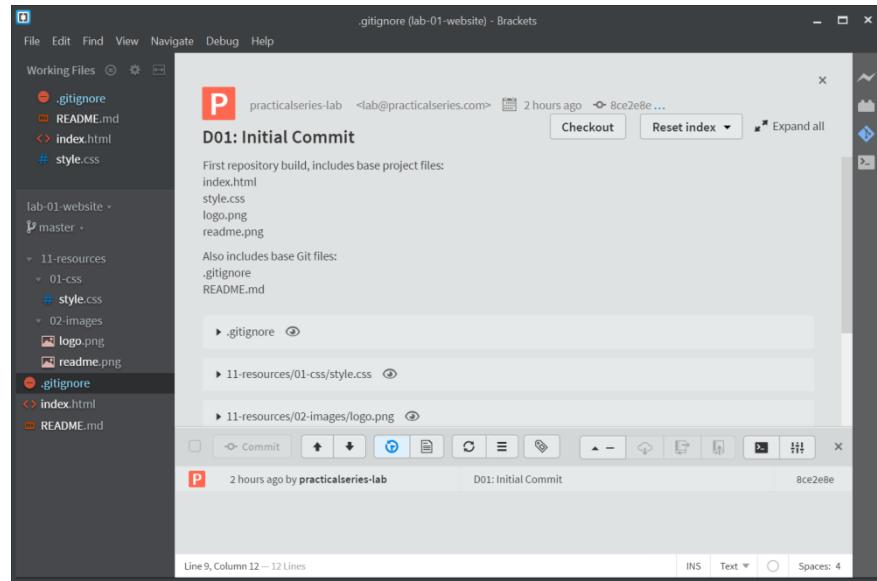


Figure 6.29 Commit information

Let's concentrate on the information pane, make it a bit bigger:

The screenshot shows a GitHub commit details page for a repository named 'practicalseries-lab'. The commit was made by 'lab@practicalseries.com' 2 hours ago, with a commit hash of '8ce2e8e ...'. The commit title is 'D01: Initial Commit'. The commit message states: 'First repository build, includes base project files: index.html, style.css, logo.png, readme.png'. It also mentions 'Also includes base Git files: .gitignore, README.md'. The main content of the commit is the .gitignore file, which contains the following code:

```
new file mode 100644
@@ -0,0 +1,12 @@
1  +# https://git-scm.com/docs/gitignore
2  +# https://help.github.com/articles/ignoring-files
3  +# Example .gitignore files: https://github.com/github/gitignore
4  +/bower_components/
5  +/node_modules/
6  +
7  +# Windows OS Files
8  +Thumbs.db
9  +Desktop.ini
10 +
11 +# Mac OS Files
12 +.DS_Store
\ No newline at end of file
```

Below the commit content, there is a list of tracked files:

- 11-resources/01-css/style.css
- 11-resources/02-images/logo.png
- 11-resources/02-images/readme.png
- README.md
- index.html

Figure 6.30 Commit information in full

Let's start at the top; it shows the Brackets generated icon for the user (P) followed by the username, email address and the time of the commit.

The next thing is the commit number, mine is:

 8ce2e8e

This is the seven digit short form of the commit number. If you click the three blue dots following it, you get the full 40 digit hash (SHA-1) number (see § 2.2.2 for an explanation). In my case this is it:

 8ce2e8e4db2de57c87a8c0613d60aa847e28c48f

I'm going to ignore the **CHECKOUT** and **RESET INDEX** buttons; these are to do with moving to earlier commit points and I cover this in section 7 (If you can't see the buttons, you probably haven't activated one of the options, again it's in section 7).

Next is the commit message in full, the first line of the commit message is shown as a heading.

Underneath this is a list of all the files that were in the commit. If you click the small arrow on the left of the file name, it expands the file showing the changes that were made at the time of the commit. In Figure 6.30 I've expanded the `.gitignore` file. If you click the **EXPAND ALL** button  **Expand all** at the top, it expands all files in the commit.

Finally, the little eye symbol  This opens the current version of the file in the Brackets editor (if the file is already opened in Brackets, then the button doesn't do anything).

6.3.4 Tagging the commit

I'm going to tag the commits as I go along (*you don't have to do this if you don't want to*) I'm doing it because I want to demonstrate tagging and it's easier to demonstrate if we do it as we go along—it is also useful for identifying commit points.

I'm using the tagging arrangement listed in Appendix B. Brackets will let you tag the latest commit point (it won't let you tag earlier commits³—*hence I'm doing it as we go along*).

To tag the latest commit point, click the tag icon in the Git pane . This opens a simple tag dialogue box:

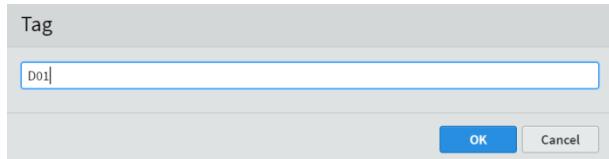


Figure 6.31 Tag dialogue box

I'm going to call the first tag D01. Enter it in the box and click OK.

This will automatically activate the history view in the Git pane; this will show the new tag against the latest commit point:

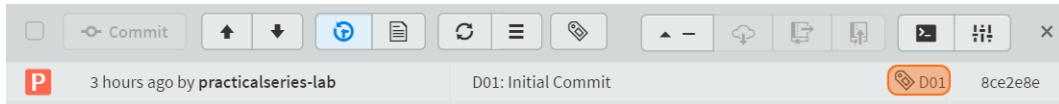


Figure 6.32 A tagged commit point

3

Brackets won't let you tag anything but the current commit. Git however will, the Git command line terminal Git Bash can be used to assign a tag to any previous commit (see § 9.6).

6.3.5 The current workflow

This is another London Underground diagram, just like MORNINGTON CRESCENT. It shows where we are in the current project as a series of branches and commits. It's very simple at the minute:

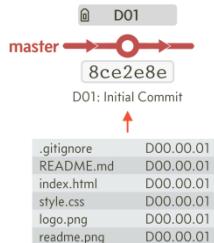


Figure 6.33 Workflow (D01)

I'll expand this diagram as we go along.

6.4

Adding another commit

Before we look at branches, I want to make some small modifications to `index.html` and add it as another commit. Just so we've got two commit points to look at. This again is following the previous example in Section 2.2.

Modify `index.html`, remove the second paragraph by deleting lines 21 and 22 from the original file:

```
INDEX.HTML

1 <html lang="en">                                <!-- Declare language -->
2   <head>                                         <!-- Start of head section -->
3     <meta charset="utf-8">                      <!-- Use unicode character set -->
4
5     <link rel="stylesheet" type="text/css" href="11-resources/01-css/style.css">
6
7     <title>PracticalSeries: Git Lab</title>
8   </head>
9
10  <body>
11    <h1>A Practical Series Website</h1>
12
13    <figure class="cover-fig">
14      
15    </figure>
16
17    <h3>A note by the author</h3>
18
19    <p>This is my second Practical Series publication—this one happened by accident
        too. The first publication is all about building a website, you can see it here.
        This publication came about because I wanted some sort of version control
        mechanism for the first publication.</p>
20
21  </body>
22 </html>
```

Code 6.5 Modification to index.html

Save the file and it will show in the Git pane as **modified**.

Tick the box to stage the file.

The screenshot shows the Brackets IDE interface with the file 'index.html' open. The left sidebar displays the project structure for 'lab-01-website' with branches 'master' and '01-css'. The 'Working Files' panel shows files like '.gitignore', 'README.md', 'index.html', and 'style.css'. The main editor area contains the HTML code for 'index.html', which includes a title, a figure with a logo, and a note about the publication. A red bar highlights a deleted line of code at line 20. The bottom status bar indicates 'Line 20, Column 1 — 23 Lines'.

```
index.html (lab-01-website) - Brackets
File Edit Find View Navigate Debug Help
Working Files
.gitignore README.md index.html style.css
lab-01-website
master
11-resources
01-css
02-images
logo.png
readme.png
.gitignore index.html README.md
1 <html lang="en">
2   <head>
3     <meta charset="utf-8">
4
5     <link rel="stylesheet" type="text/css" href="11-resources/01-
css/style.css">
6
7   </head>
8
9   <body>
10    <h1>A Practical Series Website</h1>
11
12    <figure class="cover-fig">
13      
14    </figure>
15
16    <h3>A note by the author</h3>
17
18    <p>This is my second Practical Series publication—this one happened by
19      accident too. The first publication is all about building a website,
20      you can see it here. This publication came about because I wanted some
21      sort of version control mechanism for the first publication.</p>
22
23  </body>
24 </html>
```

Figure 6.34 Modified and staged index.html

Just one thing to note, there is a small red (*pink?—I'm a man, not good with colours*) bar showing in the `index.html` file. This shows where something has been deleted. A green bar shows where something has been added and a yellow bar (*might be amber or orange*) shows a line that has been modified.

Commit the change with the following commit message:

P01: First Publication

Index.html modified, proofreading correction.

It looks like this:

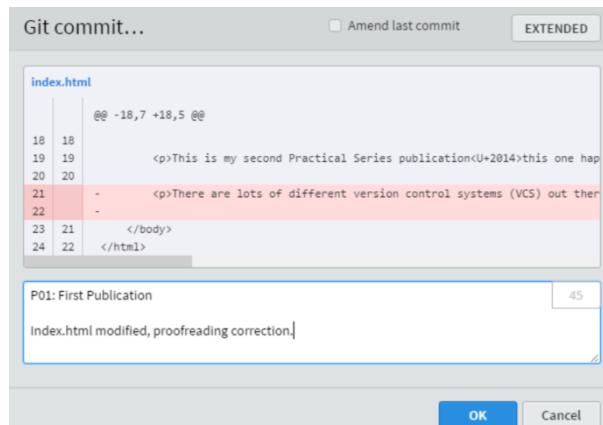


Figure 6.35 Commit dialogue box

This time (*because there is only one file*), the Commit dialogue box shows the changes to `index.html`. It shows what has been deleted (*in pink, that's definitely pink*).

Click the history button to see the two commits.



Figure 6.36 Second commit in history

This time the commit is [\[25c1410\]](#). Clicking this commit shows the commit information screen:

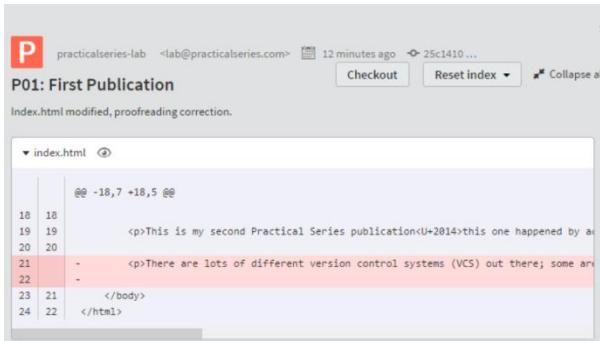


Figure 6.37 Second commit information

Let's also tag this new commit with the label [P01](#).

Again click the tag icon in the Git pane  . This opens the simple tag dialogue box:

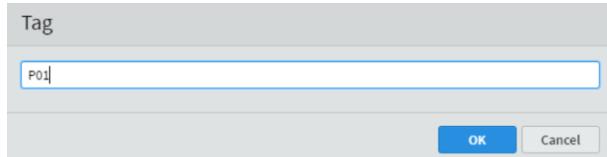


Figure 6.38 Tag dialogue box

Giving us this:



Figure 6.39 Commits with tags

And we now have this workflow:

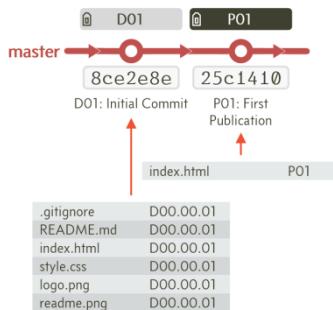


Figure 6.40 Workflow with two commits

6.4.1 Viewing the file history

We used the commit history to see a full list of commits made within the project (§ 6.3.3 and above). Now not every commit affects every file; of the two commits in the project, only `index.html` is affected by both, all the other files were affected by just the first commit.

We can see this by using the *file history*. To demonstrate this open the file history in the Git pane by clicking the  button. The result depends on which file is selected in the file tree at the left hand side. I have this:

The screenshot shows the Brackets IDE interface. The left sidebar displays a file tree for a repository named 'lab-01-website' with a single commit 'master'. The main editor area shows the content of 'index.html'. The bottom right corner of the editor has a small 'git' icon. Below the editor is a commit history panel. The commit history for 'index.html' shows two entries:

Author	Commit Message	Date	SHA
P	38 minutes ago by practicalseries-lab	P01: First Publication	25c1410
P	a day ago by practicalseries-lab	D01: Initial Commit	8ce2e8e

At the bottom of the screen, there is a status bar with the text "Line 20, Column 1 — 23 Lines".

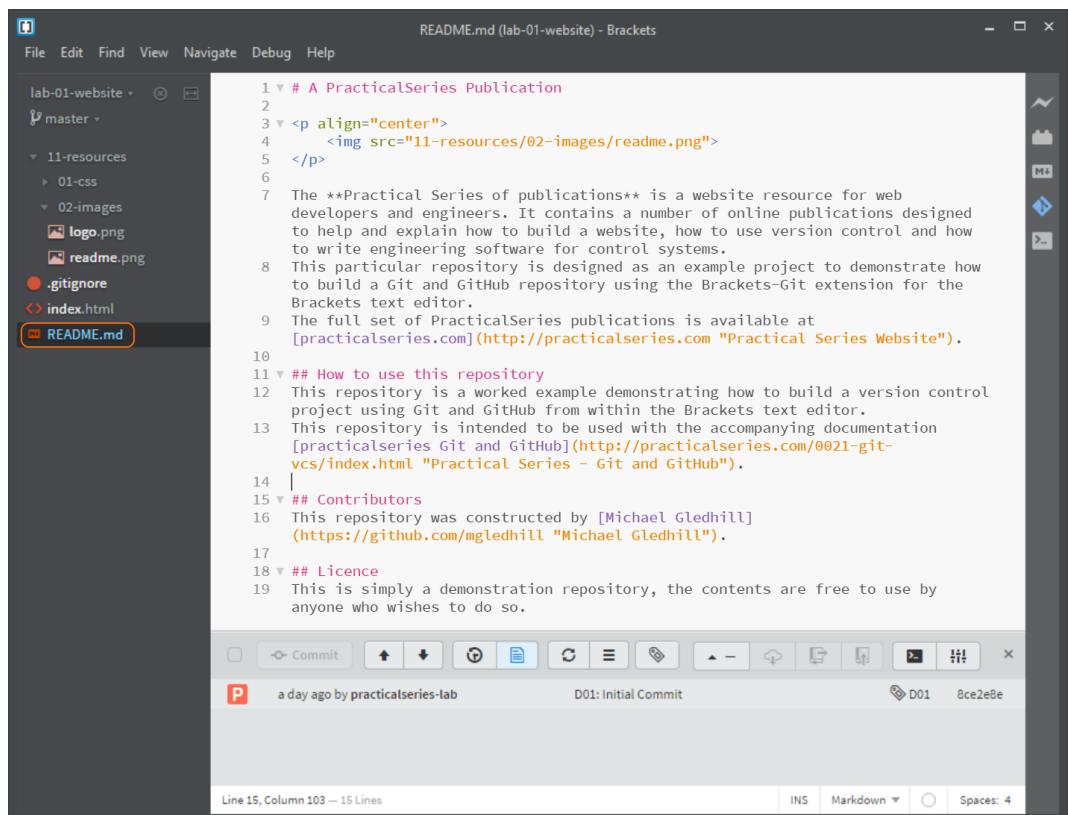
Figure 6.41 index.html in file history

The history shown is for the selected file in the file tree (the selected file is shown in blue), I've highlighted where the selection is in orange in Figure 6.41.

The `index.html` file is affected by two commits; if I select the `README.md` file I get Figure 6.42.

The `README.md` file is only associated with one commit, the first one.

Clicking the commit line opens the commit information screen in just the same way as the commit history.



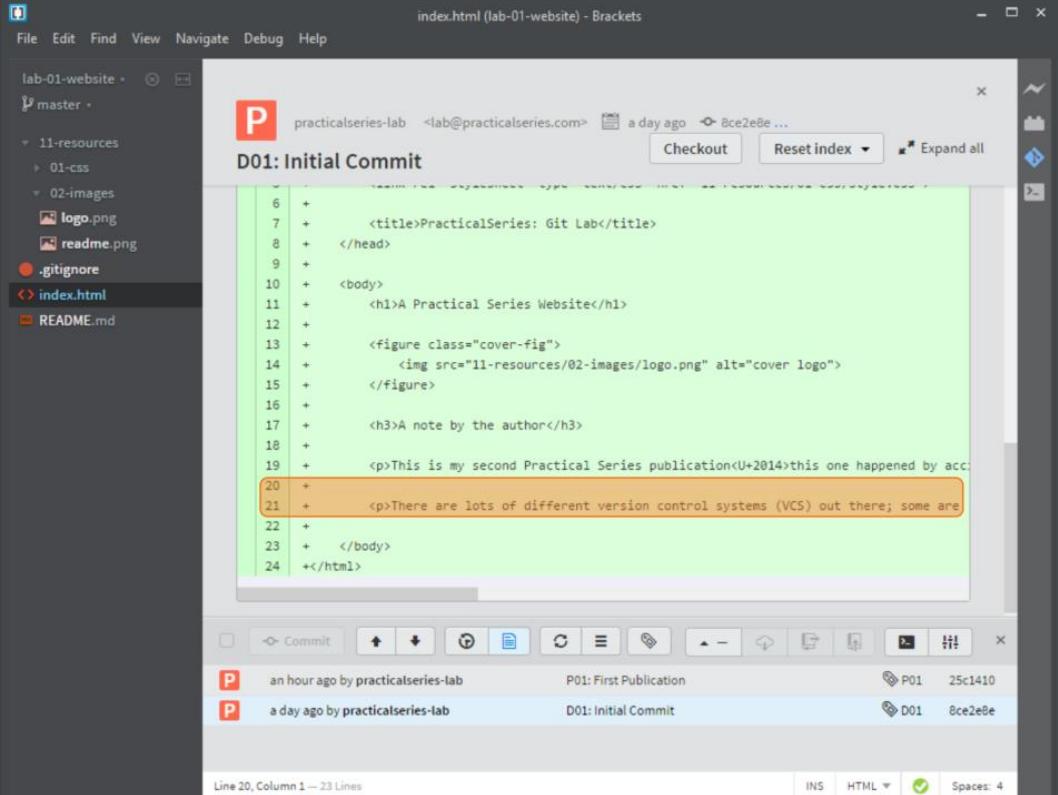
The screenshot shows the Brackets text editor interface. The title bar reads "README.md (lab-01-website) - Brackets". The left sidebar displays a file tree for the "lab-01-website" repository, including ".gitignore", "index.html", and "README.md". The main editor area contains the content of the README.md file, which describes the Practical Series of publications. The file history panel at the bottom shows a single commit: "a day ago by practicalseries-lab" with the message "D01: Initial Commit". The status bar at the bottom indicates "Line 15, Column 103 — 15 Lines", "Markdown", and "Spaces: 4".

```
1 v # A PracticalSeries Publication
2
3 v <p align="center">
4   
5 </p>
6
7 The **Practical Series of publications** is a website resource for web
8 developers and engineers. It contains a number of online publications designed
9 to help and explain how to build a website, how to use version control and how
10 to write engineering software for control systems.
11 This particular repository is designed as an example project to demonstrate how
12 to build a Git and GitHub repository using the Brackets-Git extension for the
13 Brackets text editor.
14
15 v ## How to use this repository
16 This repository is a worked example demonstrating how to build a version control
17 project using Git and GitHub from within the Brackets text editor.
18 This repository is intended to be used with the accompanying documentation
19 [practicalseries Git and GitHub](http://practicalseries.com/0021-git-
vcs/index.html "Practical Series – Git and GitHub").
20
21 v ## Contributors
22 This repository was constructed by [Michael Gledhill]
23 (https://github.com/mgledhill "Michael Gledhill").
24
25 v ## Licence
26 This is simply a demonstration repository, the contents are free to use by
27 anyone who wishes to do so.
```

Figure 6.42 README.md in file history

6.4.2 Viewing the contents of a file at a previous commit

Go back to viewing the file history for the `index.html` file (Figure 6.41) and click on the first commit `[8ce2e8e]`. This will open the commit information for `index.html` at that first commit, it looks like Figure 6.43.



The screenshot shows the Brackets IDE interface. The left sidebar displays a file tree for a repository named 'lab-01-website' with branches 'master' and 'lab-01-website'. The 'index.html' file is selected. The main editor window shows the content of 'index.html' with the title 'PracticalSeries: Git Lab'. Below the code, the commit history for 'index.html' is displayed, showing two commits:

- P an hour ago by practicalseries-lab P01: First Publication P01 25c1410
- P a day ago by practicalseries-lab D01: Initial Commit D01 8ce2e8e

The commit 'D01: Initial Commit' is expanded, showing the code as it was at that point. Lines 20 and 21, which contain the deleted text 'There are lots of different version control systems (VCS) out there; some are', are highlighted in orange.

Figure 6.43 index.html at the first commit point

This is showing `index.html` as it was at the first commit, you can see the lines we deleted (line 20 and 21) are still there (I've highlighted them in orange).

6.5 Branches

To demonstrate branches, I'm going to add another page to the website project. This will be an introduction page called `01-intro.html`.

This new page will be added to the project on a new branch. Currently we only have the `master` branch and everything we've done so far has been done on this `master` branch.

6.5.1 Adding a branch in Brackets

This is easy. Just click the small arrow to the right of the master branch in the left hand file tree and select `CREATE NEW BRANCH...` (Figure 6.44):

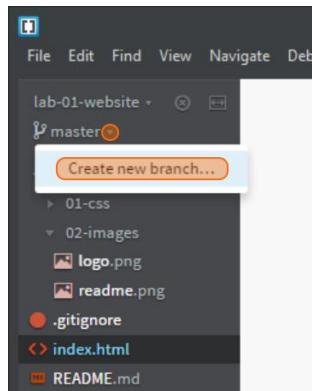


Figure 6.44 Brackets—create new branch

This opens the Create new branch dialogue box:

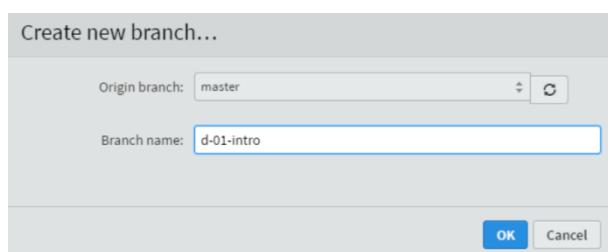
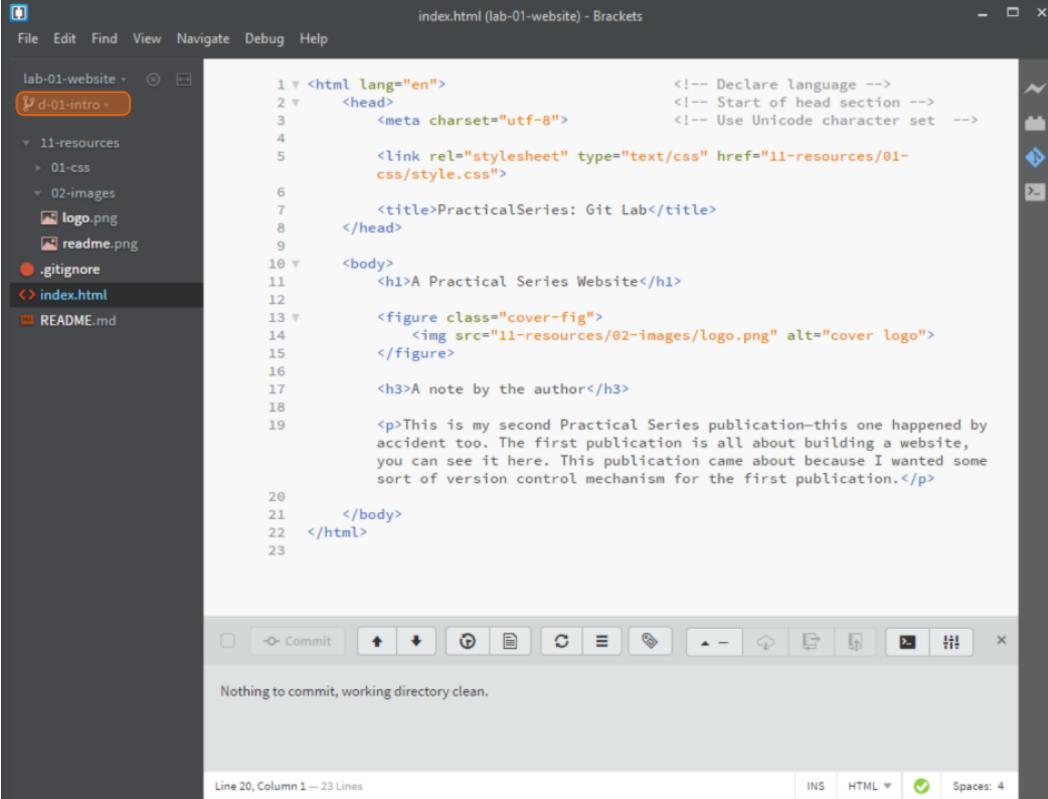


Figure 6.45 Brackets—create new branch dialogue box

If we had more than one branch, we could choose which branch to split from ([ORIGIN BRANCH](#)). In our case we only have the **master** branch and that's all we can choose.

Enter the name for the new branch; call it **d-01-intro**.

Brackets will automatically switch to the new branch (Figure 6.46):



The screenshot shows the Brackets IDE interface. The file tree on the left has a branch named 'd-01-intro' highlighted in orange. The main editor window displays the content of 'index.html'. The code is as follows:

```
1 <html lang="en">          <!-- Declare language -->
2   <head>                  <!-- Start of head section -->
3     <meta charset="utf-8">    <!-- Use Unicode character set -->
4
5     <link rel="stylesheet" type="text/css" href="11-resources/01-
css/style.css">
6
7     <title>PracticalSeries: Git Lab</title>
8   </head>
9
10  <body>
11    <h1>A Practical Series Website</h1>
12
13    <figure class="cover-fig">
14      
15    </figure>
16
17    <h3>A note by the author</h3>
18
19    <p>This is my second Practical Series publication—this one happened by
accident too. The first publication is all about building a website,
you can see it here. This publication came about because I wanted some
sort of version control mechanism for the first publication.</p>
20
21  </body>
22 </html>
```

The status bar at the bottom indicates 'Nothing to commit, working directory clean.' The bottom right corner shows 'INS', 'HTML', and 'Spaces: 4'.

Figure 6.46 Brackets—new branch

I've highlighted the new branch in the file tree (highlighted in orange). Notice also, that although we are on a new branch, nothing has changed, Brackets is still reporting [Nothing to commit, working directory clean](#).

The workflow diagram now has a branch—*woohoo* (*These diagrams are now reminding me of the opening sequence to Dad's Army*):

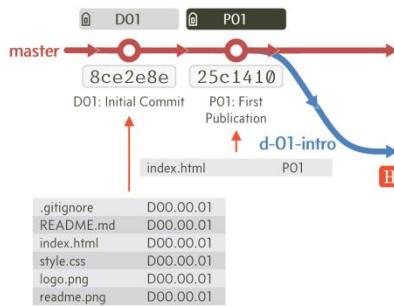


Figure 6.47 Workflow with the **d-01-intro** branch

6.5.2 Making a commit on a new branch

Make sure the new branch is selected (**d-01-intro** is showing as the active branch in the file tree).

Now add a new file (just as we did before in § 6.2). Right click the file tree select **NEW FILE** and rename the file **01-intro.html**.

Add the following code to it.

```

01-INTRO.HTML

1 <html lang="en">
2   <head>
3     <meta charset="utf-8">
4
5     <link rel="stylesheet" type="text/css" href="11-resources/01-css/style.css">
6
7     <title>PracticalSeries: Git Lab – Introduction</title>
8   </head>
9
10  <body>
11    <h1>A Practical Series Website</h1>
12
13    <h3>Introduction</h3>
14
15    <p>This page is an introduction to the website and how to use it.</p>
16
17  </body>
18 </html>

```

Code 6.6 **d-01-intro** branch—01-intro.html new file

This will show as **untracked** in the Git pane. Tick the box, click **COMMIT** and enter the following commit message:

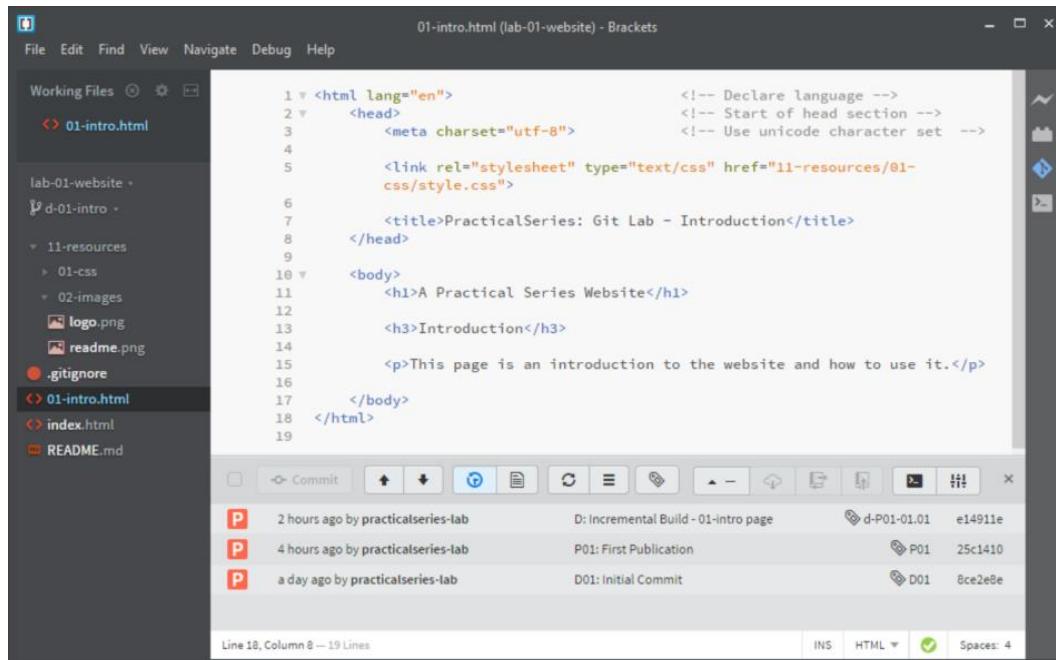
D: Incremental Build - 01-intro page

01-intro.html added.

I get commit [e14911e].

Give the commit the tag d-P01-01.01 and we're done.

This is what I have in Brackets (commit history view):



The screenshot shows the Brackets IDE interface with the following details:

- File Menu:** File, Edit, Find, View, Navigate, Debug, Help.
- Working Files:** Shows files like 01-intro.html, .gitignore, index.html, and README.md.
- Project Structure:** lab-01-website, d-01-intro, and 11-resources.
- Code Editor:** Displays the content of 01-intro.html, which includes HTML code for a practical series website introduction.
- Commit History:** Shows three commits in the bottom panel:
 - 2 hours ago by practicalseries-lab: D: Incremental Build - 01-intro page (tagged d-P01-01.01, commit e14911e)
 - 4 hours ago by practicalseries-lab: P01: First Publication (tagged P01, commit 25c1410)
 - a day ago by practicalseries-lab: D01: Initial Commit (tagged D01, commit 8ce2e0e)
- Status Bar:** Line 10, Column 8 — 19 Lines, INS, HTML, Spaces: 4.

Figure 6.48 First commit on the new branch

And the workflow is now:

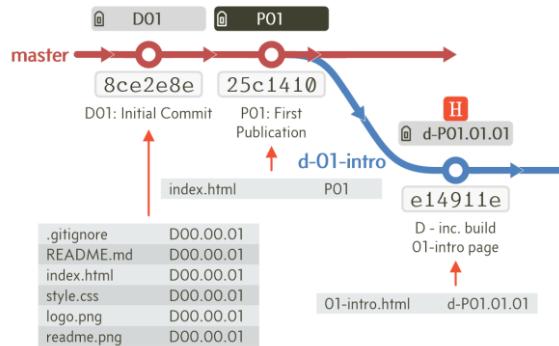


Figure 6.49 Workflow

6.6 Switching branches

This is a small but important point.

If I'm on the **d-01-intro** branch I have these files in the file tree and the following commits:

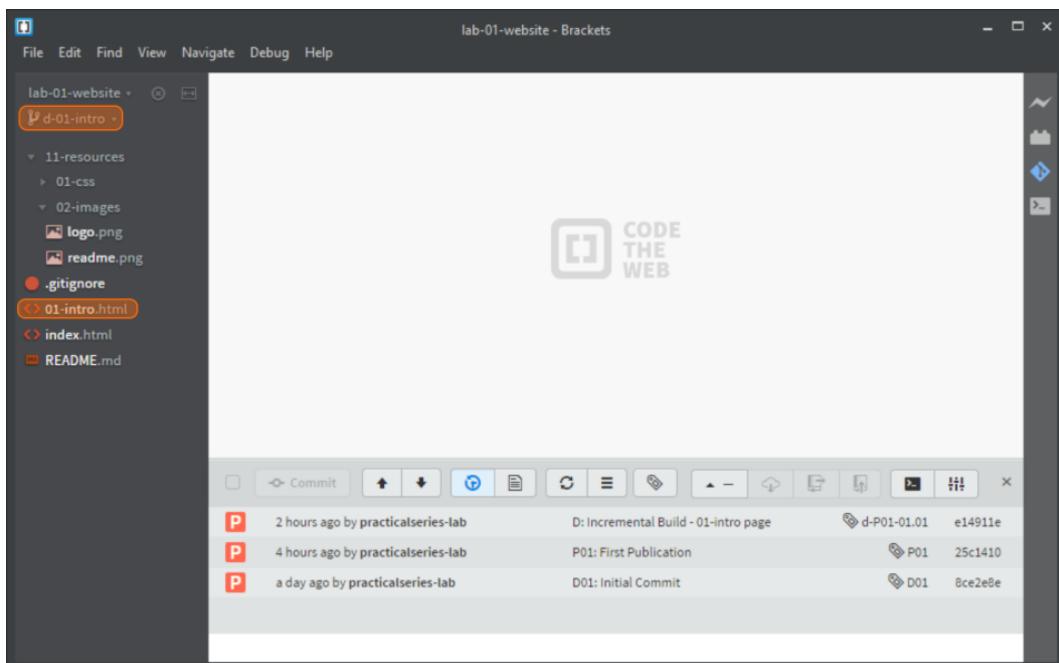


Figure 6.50 Files and commits on the **d-01-intro** branch

I clearly have the **01-intro.html** file and there are three commits.

Change back to the **master** branch (by clicking the small arrow next to the branch in the file tree and selecting **master**, Figure 6.51).

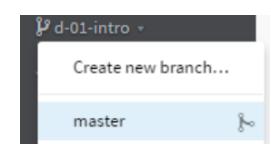


Figure 6.51 Brackets—switch branches

Now I get:

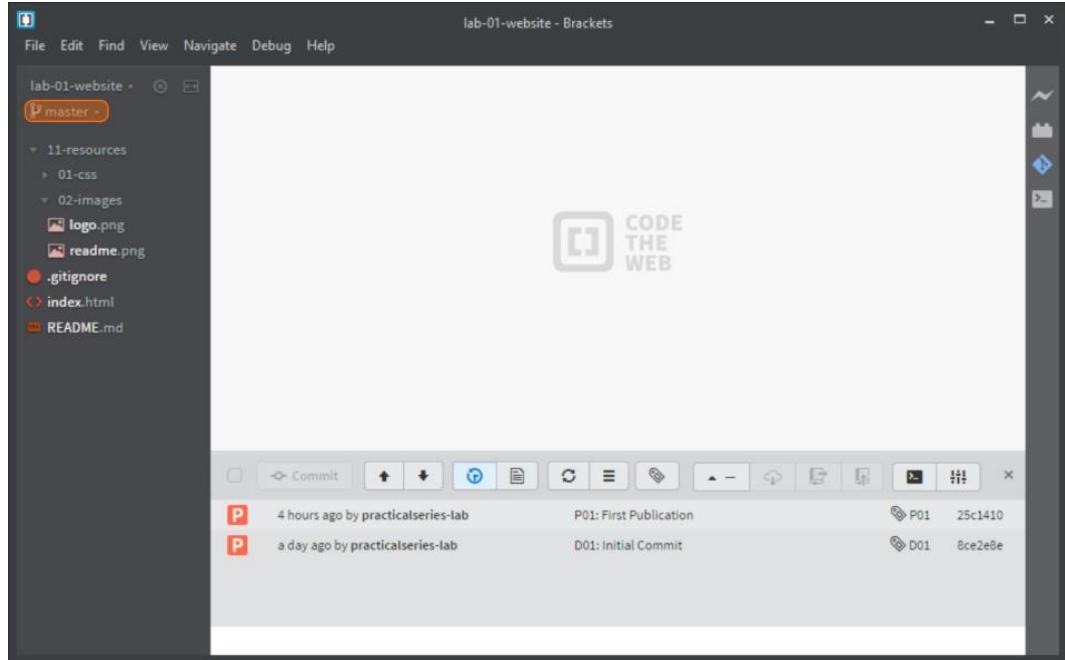


Figure 6.52 Files and commits on the `master` branch

The `01-intro.html` file is no longer there and there are only two commits.

To reiterate what I said in section 2.3.1:

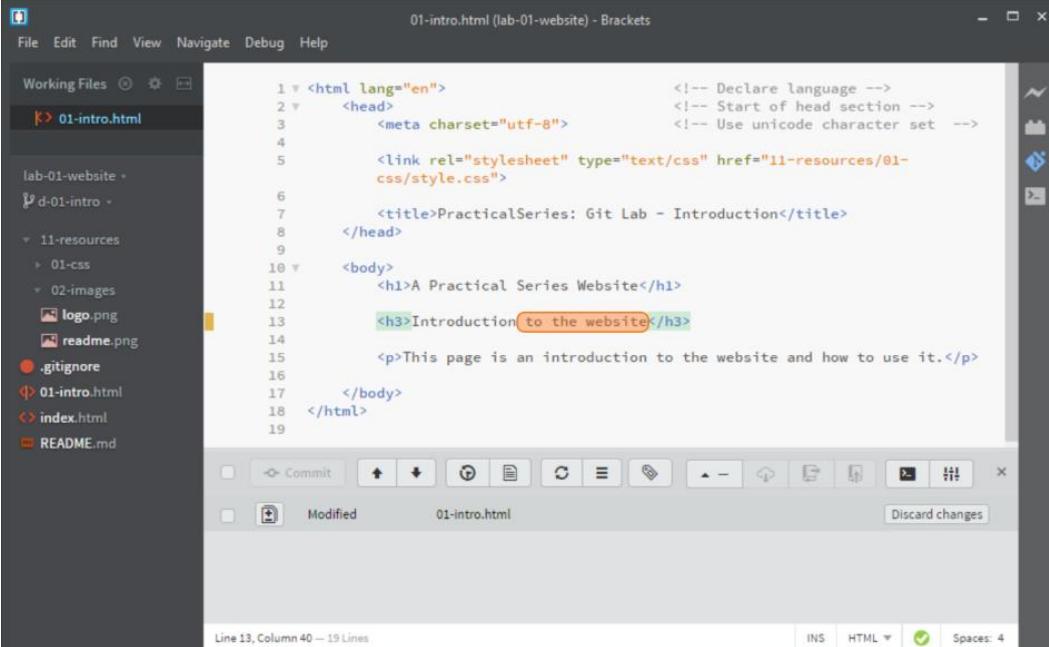
**SWITCHING BRANCHES CHANGES THE FILES
IN YOUR WORKING DIRECTORY**

This is important. Like I said “*you’ll remember it when you’re on the wrong branch wondering where all your files have gone*”.

6.6.1 Switching branches with uncommitted changes

I said before, “*you can’t switch branches if you have uncommitted changes on that branch*”.

Switch back to the **d-01-intro** branch and open the **01-intro.html** file. Add some text into the **<h3>Introduction</h3>** line (line 13) and save the file. The file status will now be modified:



The screenshot shows the Brackets IDE interface. The title bar reads "01-intro.html (lab-01-website) - Brackets". The menu bar includes File, Edit, Find, View, Navigate, Debug, and Help. The left sidebar shows a project structure with "Working Files" containing "01-intro.html", and a folder "lab-01-website" with subfolders "11-resources", "01-css", "02-images", and files ".gitignore", "01-intro.html", "index.html", and "README.md". The main editor area displays the HTML code for "01-intro.html". Line 13 contains the text "**<h3>Introduction to the website</h3>**". The status bar at the bottom indicates "Line 13, Column 40 — 19 Lines", "INS", "HTML", and "Spaces: 4". A toolbar below the editor includes icons for Commit, Undo, Redo, Save, and others. A message bar at the bottom right says "Modified 01-intro.html".

Figure 6.53 Modified file on d-01-intro

If I now try to change back to the master branch I get the following message:

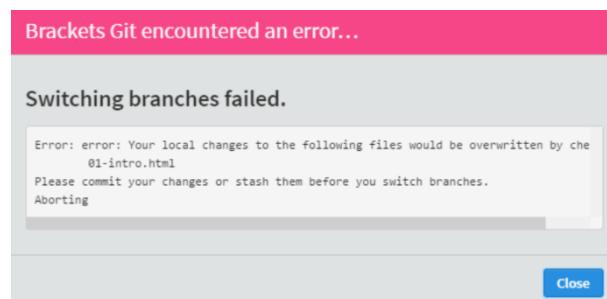


Figure 6.54 Brackets switch branch error

Brackets won’t let you change branches if you have uncommitted changes.

Again this confirms what I said in section 2.3.1:

**YOU CANNOT CHANGE BRANCHES IF YOU HAVE
UNCOMMITTED CHANGES IN YOUR WORKING COPY**

Always commit changes before switching branches

See the footnote in § 2.3.1 for the reasons why.

6.6.2 Discarding the latest changes

I made the modification to `01-intro.html` to prove the above point; I don't really want to make these changes. However, I've saved the file so *how do I go back?*

The answer is simple, click the **MORE ACTIONS** button  and select **DISCARD ALL CHANGES SINCE LAST COMMIT**.

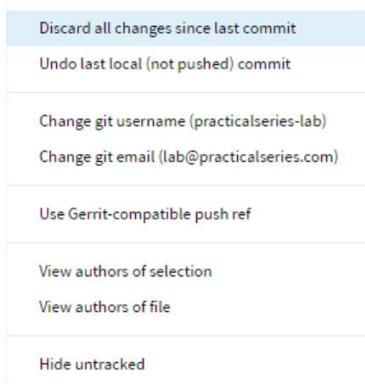


Figure 6.55 Brackets—discard changes

Click **OK**, the file will go back to how it was and Git will show:

Nothing to commit, working directory clean.

6.7

Merging, conflicts and deleting branches

I'm going to demonstrate merging branches in exactly the same way I did theoretically in section 2.4.

I'll start by adding another page to the [lab-01-website](#). The **master** branch still has the latest version of the deployable code (although **d-01-intro** is more advanced than the **master** branch, it is under development—new branches should always be based on the latest deployable code, see Appendix B).

I'm going to add a new [`02-about.html`](#) page and I'm going to do so by creating a new branch from the [`\[25c1410\]`](#) commit point on the **master** branch. Switch back to the **master** branch, create a new branch **d-02-about** and switch to this new branch.

Now we have Figure 6.56:

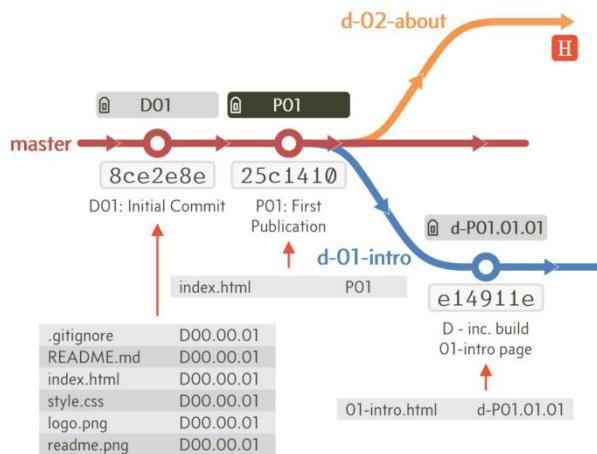


Figure 6.56 A third branch

Time for another file. In the new **d-02-about** branch add [`02-about.html`](#) in the root folder (same place as [`index.html`](#)) and add the following code to it:

```

02-ABOUT.HTML

1 <html lang="en">
2   <head>
3     <meta charset="utf-8">
4
5     <link rel="stylesheet" type="text/css" href="11-resources/01-css/style.css">
6
7     <title>PracticalSeries: Git Lab - About us</title>
8   </head>
9
10  <body>
11    <h1>A Practical Series Website</h1>
12
13    <h3>About Us</h3>
14
15    <p>This page explains who we are and how we came to be doing this.</p>
16
17  </body>
18 </html>

```

Code 6.7 d-02-about branch—create new file: 02-about.html

Save the file, and in the Git pane tick the box to stage the file:

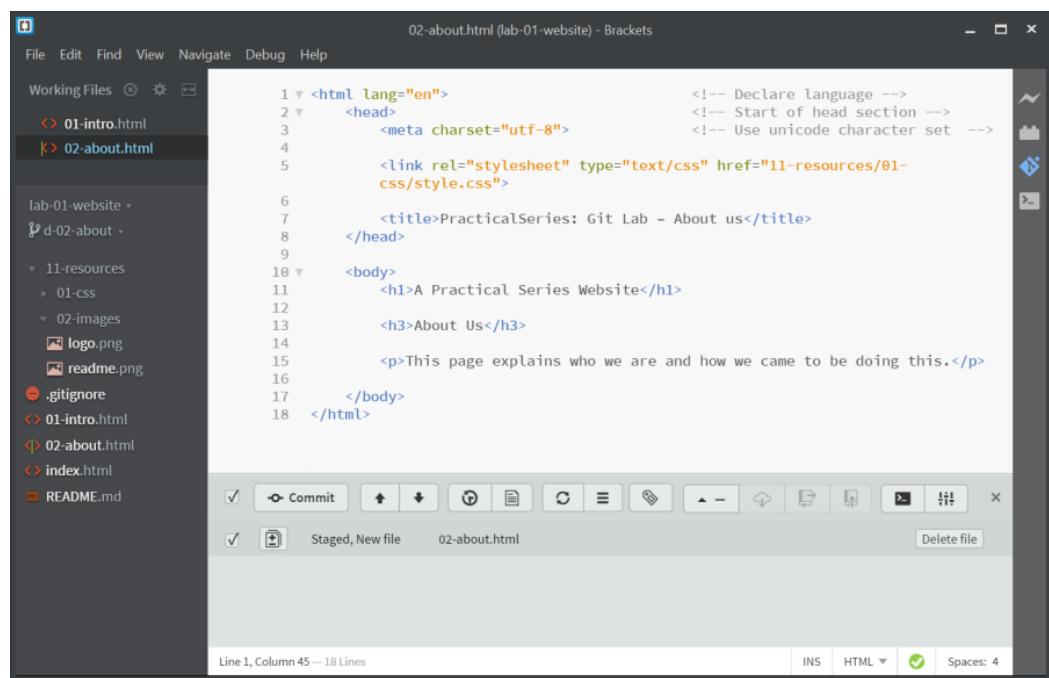


Figure 6.57 02-about.html

Commit the changes with the commit message:

D: Incremental Build - 02-about page

02-about.html added.

Give the new commit point the tag [d-P01.02.01](#). In my case the commit point is [\[3f15582\]](#).

I now have four commits:



Figure 6.58 Current commits

The workflow is:

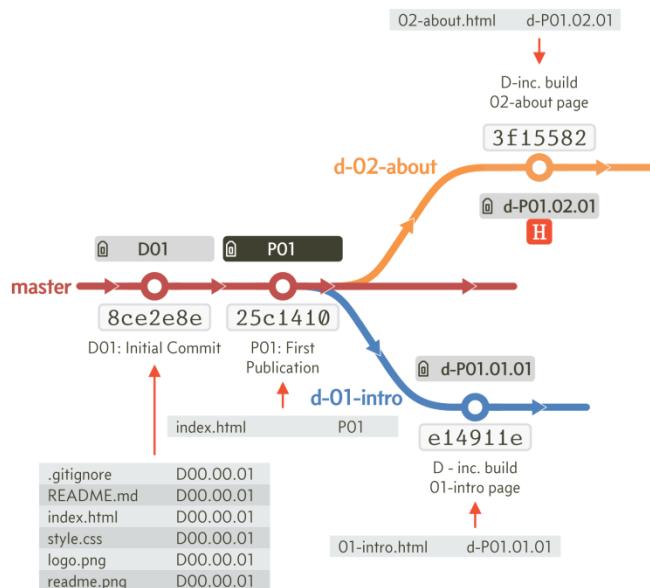


Figure 6.59 Second new branch with first commit

I'm going to do some jumping about between branches here to set up the merge situation. First switch to **d-01-intro** and change the **style.css** file:

```
STYLE.CSS
```

```
24 h1, h2, h3, h4, h5, h6 { /* set standard headings */
25   font-family: sans-serif;
26   font-weight: normal;
27   font-size: 3rem;
28   padding: 2rem 5rem 2rem 5rem;
29 }
30 h3 { font-size: 2.5rem; color: #4F81BD; }
31
```

Code 6.8 style.css modified in **d-01-intro**

The only thing I've done is to change the colour of the **h3** element in line 30; I've made it blue by adding a colour declaration (in bold below):

```
30 h3 { font-size: 2.5rem; color: #4F81BD; }
```

Save the file, tick the stage box and **commit** the change to **style.css** with the commit message:

```
S: Staged - 01-intro page
Style.css modified <h3> colour changed to blue.
```

It makes the page look like this:

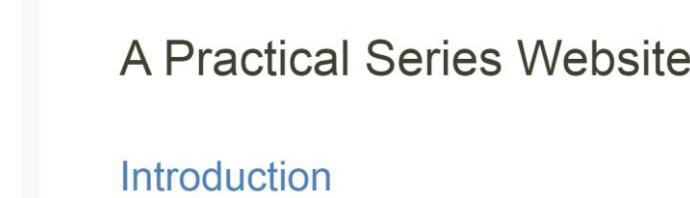


Figure 6.60 01-intro.html page

My commit point is [\[f5aeb76\]](#). Tag the commit with the label **s-P01.01.P02**. This will be the last commit on this branch (*hence the tag, see Appendix B*).

We are now at this point:

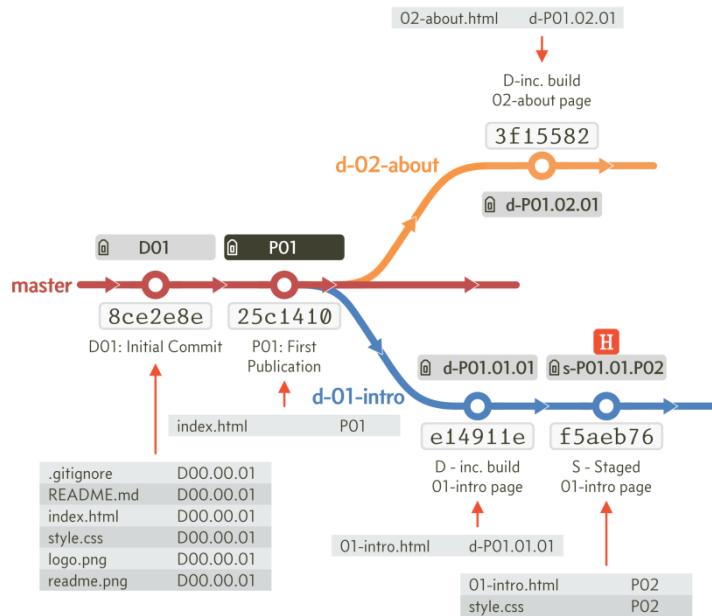


Figure 6.61 Second commit on **d-01-intro**

The next thing is to merge the two commits on the **d-01-intro** branch onto the **master** branch, like Figure 6.62:

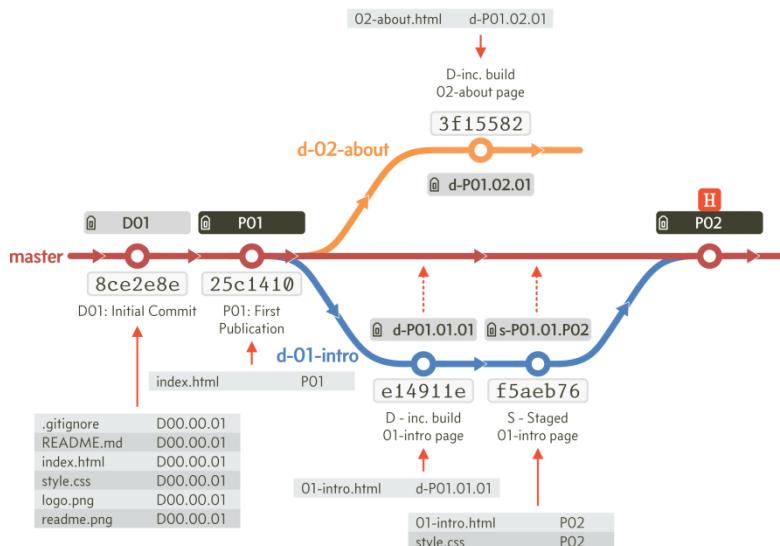


Figure 6.62 Merging one branch into another

6.7.1 Merging a branch with Brackets

This is something new, we haven't merged a branch with Brackets yet, this is how you do it.

I want to merge the [d-01-intro](#) branch back onto the [master](#) branch.

The first thing is to **switch to the receiving branch**. In this case we need to be on the [master](#) branch. To do that; click the small arrow next to the [d-01-intro](#) branch in the file tree and in the dropdown select [master](#):

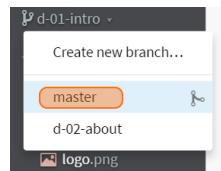


Figure 6.63 Switch to [master](#) branch

Back on the [master](#) branch, if you open the history, you will only see the first two commits:

A screenshot of the Brackets interface showing the commit history for the 'master' branch. The commit history pane at the bottom shows two entries:

- P01: First Publication (a day ago by practicalseries-lab) - Commit ID: 25c1410
- D01: Initial Commit (2 days ago by practicalseries-lab) - Commit ID: 8ce2e8e

The commit history pane also includes buttons for 'Commit', 'File', 'Edit', and 'Delete'.

Figure 6.64 [master](#) branch commit history

Also, the `01-intro.html` file and the `02-about.html` files are missing (these don't exist on the `master` branch). I'm showing the `style.css` file in Figure 6.64, you can see here that the `master` branch version doesn't have the `h3` changes.

Once again:

SWITCHING BRANCHES CHANGES THE FILES IN YOUR WORKING DIRECTORY

Ok, now to do the merge.

Click the small arrow next to the `master` branch in the file tree.

I want to merge in the changes on the `d-01-intro` branch—to do this, click the merge icon  next to `d-01-intro` in the dropdown menu (Figure 6.65):

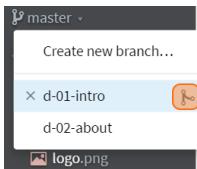


Figure 6.65 Brackets—merge branches

This will open the merge branch dialogue box:

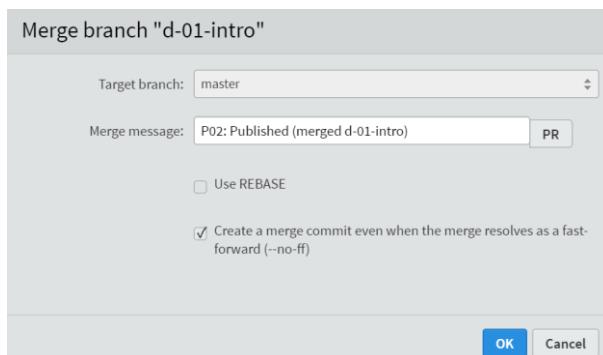


Figure 6.66 Brackets—merge branch dialogue box

The first line **TARGET BRANCH**, can't be changed; it is set to this because we were on the `master` branch when we made the merge.

When I make the merge (by clicking **OK**), Brackets will create a new commit point on the **master** branch. This is called a *merge commit*.

Merge commits, like every other commit, need a commit message and that is what the next line is. The **MERGE MESSAGE** box is asking for a one line commit message, enter:

```
P02: Published (merged d-01-intro).
```

The **PR** button next to the message puts in an automatic *pull request* message. Pull request is a GitHub thing, I haven't covered this yet (see § 10.1.2). For the minute don't push the button—you *will just have to delete the text it puts in the box if you do*.

The next two tick boxes:

REBASE—rebase allows us to effectively re-write history. I don't like rebase and I recommend you don't use it. I explain it as a topic and why I don't like it in Appendix E.1. **Leave this box unticked.**

Tick the final box **CREATE A MERGE COMMIT EVEN WHEN THE MERGE RESOLVES AS A FAST-FORWARD**. A fast-forward merge is basically a merge with no conflicts. I still want a commit point after the merge, even if there were no problems. **Tick this box**. That's it, click **OK**. This will generate a merge result dialogue box:



Figure 6.67 Brackets—merge result dialogue box

Ok, we changed two files (**01-intro.html** and **style.css**) and I'm willing to bet there were 19 additions and one deletion. Click **CLOSE**.

The **master** branch is still active; it now has a new commit, the merge commit, **[abf1121]** in my case. Give this the tag **P02** and open the commit history:

```

11-resources/01-css/style.css (lab-01-website) - Brackets
File Edit Find View Navigate Debug Help
Working Files
# style.css
lab-01-website
  master
    11-resources
      01-css
        # style.css
      02-images
        logo.png
        readme.png
      .gitignore
      01-intro.html
      index.html
      README.md
Line 30, Column 1 — Selected 41 columns — 44 Lines
INS CSS Spaces: 4

```

P 26 minutes ago by practicalseries-lab P02: Published (merged d-01-intro) abf1121
P an hour ago by practicalseries-lab S: Staged - 01-intro page f5aeb76
P a day ago by practicalseries-lab D: Incremental Build - 01-intro page e14911e
P a day ago by practicalseries-lab P01: First Publication 25c1410
P 2 days ago by practicalseries-lab D01: Initial Commit 8ce2e8e

Figure 6.68 Brackets—master branch after merge

The **master** branch now has five commits on it. The last one being the merge commit. The workflow now looks like this:

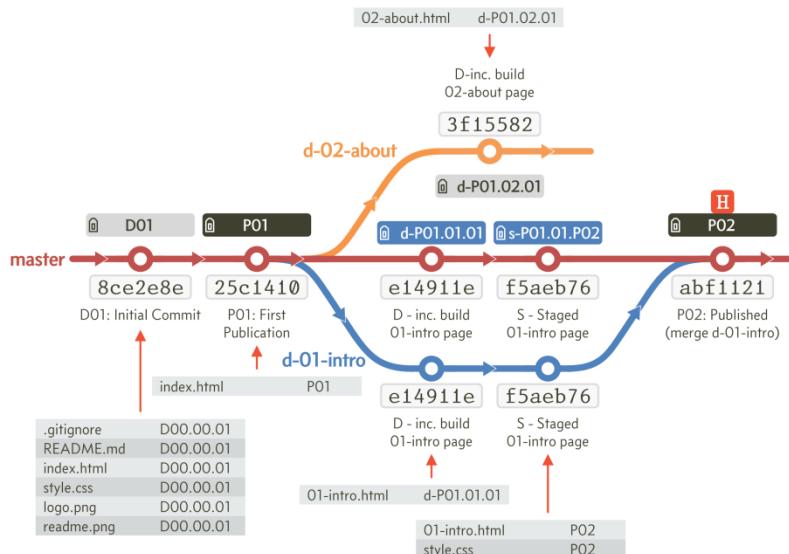


Figure 6.69 Workflow after d-01-intro merge

At this point, everything on the **d-01-intro** branch has been merged on to the **master** branch—**d-01-intro** still exists and it still has the two commit points on it (you can see this if you change to the branch and view the commit history). However, the branch doesn't contain anything we need—the commits have been merged on to the **master** branch.

The best thing to do now is delete the **d-01-intro** branch. It's served its purpose and we don't need it anymore (*we can always recreate it if we need to work on 01-intro again*).

6.7.2 Deleting a branch with Brackets

This is easy; **make sure that the branch you want to delete is not active**. In my case I'm on the **master** branch and I want to delete **d-01-intro** so this is ok.

Again click the small arrow next to the **master** branch in the file tree.

This time I want to delete the **d-01-intro** branch—to do this, click the delete icon  next to **d-01-intro** (on the left) in the dropdown menu (Figure 6.70):

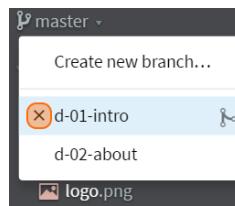


Figure 6.70 Brackets—delete a branch link

This will open an **ARE YOU SURE? DELETE LOCAL BRANCH** dialogue box, click **OK** and the branch is gone.

This gives the workflow of Figure 6.71.

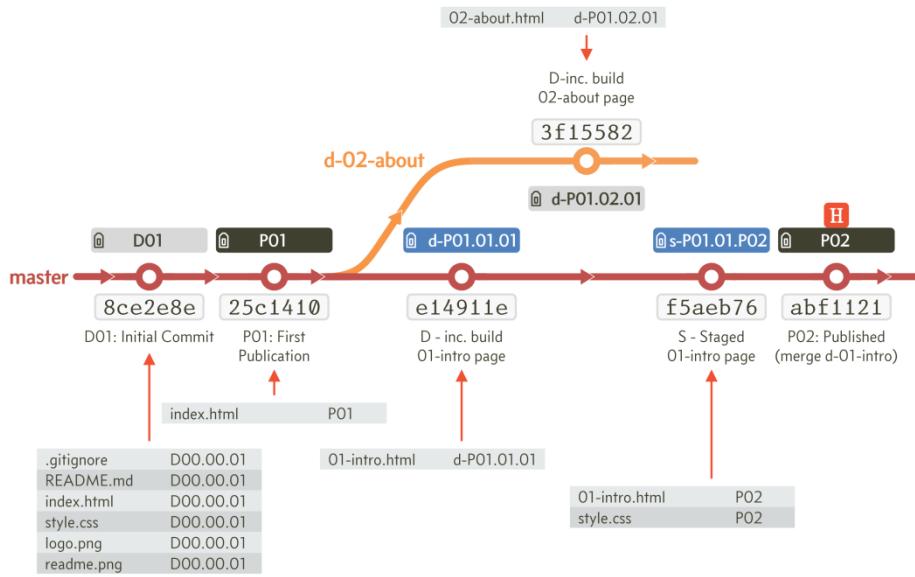


Figure 6.71 Brackets—delete a branch workflow

6.7.3 Merging a branch with a Conflict

Last bit.

I'm going to create a conflict on the **d-02-about** branch and then try to merge it back into the **master** branch (just like the theoretical arrangement in § 2.4.1).

Switch back to the **d-02-about** branch. I'm going to modify exactly the same line in the **style.css** as I did with the last change to **d-01-intro**. But this time, we'll make the **h3** element red.

```
STYLE.CSS
24 h1, h2, h3, h4, h5, h6 { /* set standard headings */
25   font-family: sans-serif;
26   font-weight: normal;
27   font-size: 3rem;
28   padding: 2rem 5rem 2rem 5rem;
29 }
30 h3 { font-size: 2.5rem; color: #c0504d; }
31
```

Code 6.9 style.css modified in **d-02-about**

Again this is a modification to line 30 (in bold below):

```
30 | h3 { font-size: 2.5rem; color: #c0504d;}
```

This will conflict with line 30 on the master branch, this has:

```
30 | h3 { font-size: 2.5rem; color: #4F81BD;}
```

The **d-02-about** modification makes the **02-about** page look like this:



Figure 6.72 02-about.html page

Save the file, tick the stage box and **commit** the change to **style.css** with the commit message:

```
S: Staged - 02-about page
```

```
Style.css modified <h3> colour changed to red.
```

My commit point is [\[28a335c\]](#). Tag the commit with the label **s-P01.02.P03**. This will be the last commit on this branch.

We now have:

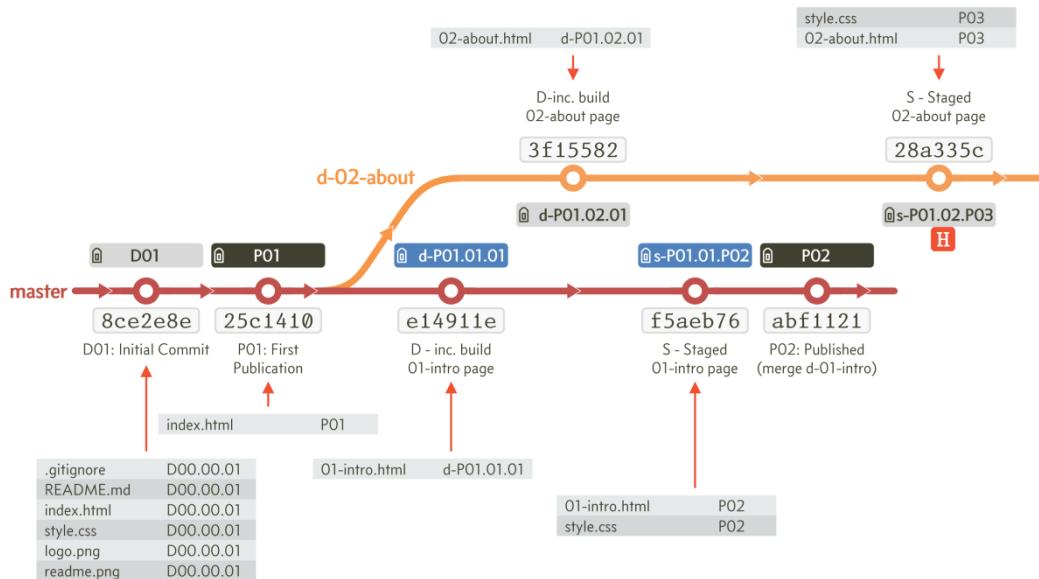


Figure 6.73 Second commit on **d-02-about**

Now switch to the **master** branch and merge in the changes on **d-02-about** (in just the same way as § 6.7.1). Click the merge icon next to **d-02-about** in the dropdown menu (Figure 6.74):

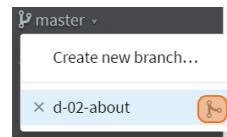


Figure 6.74 Brackets—merge **d-02-about**

Again we have the merge branch dialogue box:

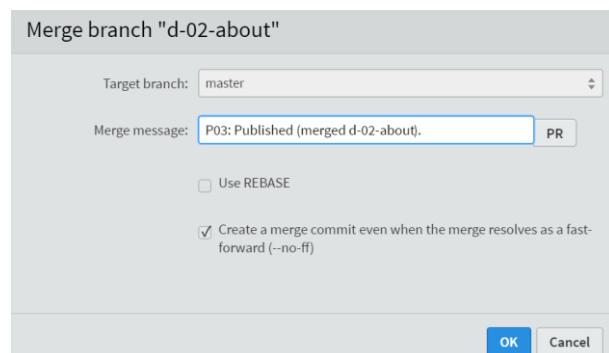


Figure 6.75 Brackets—merge dialogue box

This time enter the merge message:

P03: Published (merged d-02-about).

And click [OK](#).

This bit is confusing—bear with me.

This opens an empty Merge result box:

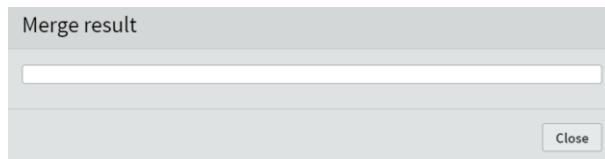


Figure 6.76 Brackets—empty merge dialogue box (conflict)

It's empty because the merge hasn't happened (*yet*). Just close the box, (*you can't enter anything on the line—as much as it looks as if you should be able to*). Click [CLOSE](#). What we want is behind it.

That's the confusing bit over—just the strange merge result box.

Things look a bit different now. This is the result:

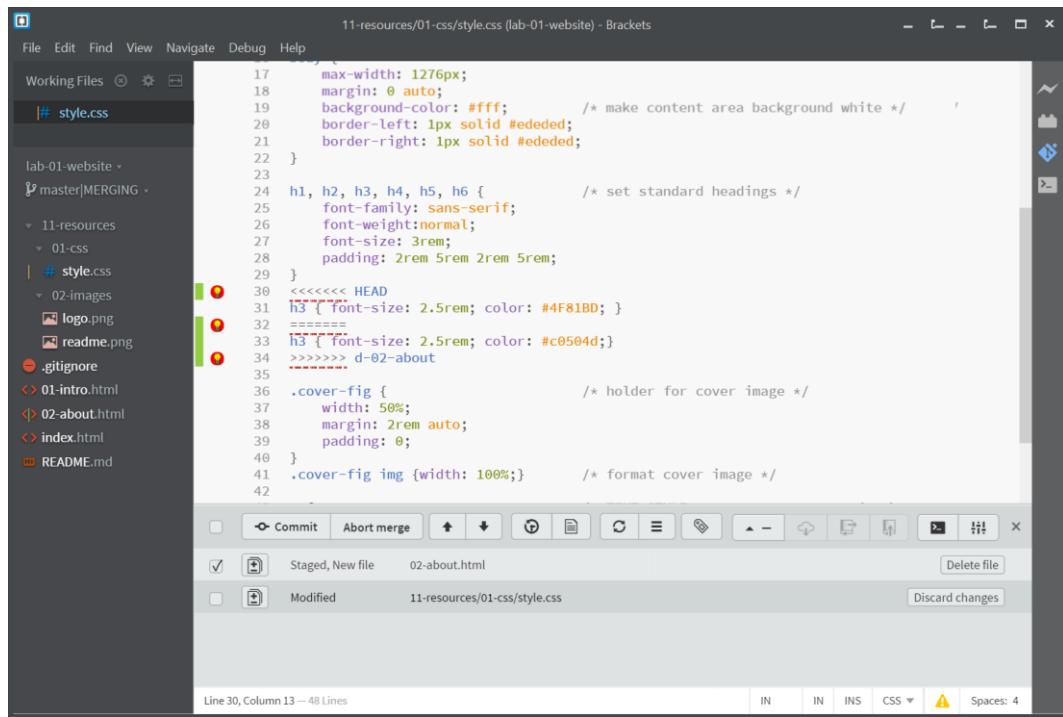


Figure 6.77 Brackets—with a merge conflict

First thing is we now have an **ABORT MERGE** button in the Git pane. Click that and we go back to where we were before we tried to merge the files (i.e. where we were in Figure 6.73).

Brackets is trying to tell us there is a conflict in the `style.css` file. It's opened the file in the editor and is showing the lines with a problem. It looks like this:

```
STYLE.CSS
30 <<<<< HEAD
31 h3 { font-size: 2.5rem; color: #4F81BD; }
32 =====
33 h3 { font-size: 2.5rem; color: #c0504d; }
34 >>>> d-02-about
```

Code 6.10 style.css with a conflict

It starts with a line of less than signs:

```
<<<<< HEAD
```

the `HEAD` tells us that what follows is from the current `head` (which is on the `master` branch).

The line that follows is the code as it is on the `master` branch:

```
h3 { font-size: 2.5rem; color: #4F81BD; }
```

Then we have a row of equal signs, this is just a divider:

```
=====
```

The last two lines show the code as it is on the merging branch: `d-02-about`:

```
h3 { font-size: 2.5rem; color: #c0504d; }
>>>>> d-02-about
```

The greater than signs identify the merging branch (`d-02-about`).

Brackets is telling us it can't decide what to do and it wants us to manually change the file to the correct version.

I could select either version and just delete the other one, or I could enter something completely different. In this case, I'm going to put the file back to how it was at the start (without specifying any colour).

Make the modifications:

```
STYLE.CSS
28     padding: 2rem 5rem 2rem 5rem;
29 }
30 h3 { font-size: 2.5rem; }
31 .cover-fimg /* holder for cover image */
32
```

Code 6.11 Resolved style.css

Note: You must delete the lines with the less than, greater than and equals signs too.

Mine looks like this after the modifications, Figure 6.78:

The screenshot shows the Brackets IDE interface. The left sidebar displays a file tree for a project named 'lab-01-website'. The 'Working Files' section shows files like 'style.css', 'index.html', and 'README.md'. The main editor area contains the CSS code for 'style.css', specifically lines 17 through 42. The bottom status bar indicates 'Line 30, Column 1 — Selected 25 columns — 44 Lines'. The bottom toolbar includes buttons for 'Commit', 'Abort merge', and other Git-related functions.

```
17 max-width: 1276px;
18 margin: 0 auto;
19 background-color: #fff; /* make content area background white */
20 border-left: 1px solid #eddede;
21 border-right: 1px solid #eddede;
22 }
23
24 ▼ h1, h2, h3, h4, h5, h6 { /* set standard headings */
25   font-family: sans-serif;
26   font-weight: normal;
27   font-size: 3rem;
28   padding: 2rem 5rem 2rem 5rem;
29 }
30 h3 { font-size: 2.5rem; }
31
32 ▼ .cover-fig { /* holder for cover image */
33   width: 50%;
34   margin: 2rem auto;
35   padding: 0;
36 }
37 .cover-fig img {width: 100%; } /* format cover image */
38
39 ▼ p { /* TEXT STYLE - paragraph */
40   margin-bottom: 1.2rem; /* *** THIS SETS PARAGRAPH SPACING *** */
41   padding: 0 5rem;
42   line-height: 135%;
```

Figure 6.78 Brackets—modified file after conflict

Save `style.css` and make sure it is staged (tick the box in the Git pane).

Now click **COMMIT**.

This opens a normal commit dialogue box. Its message will have the first line we entered initially.

P03: Published (merged d-02-about).

It will also have the information that a conflict occurred in `style.css`.

Mine looks like this:

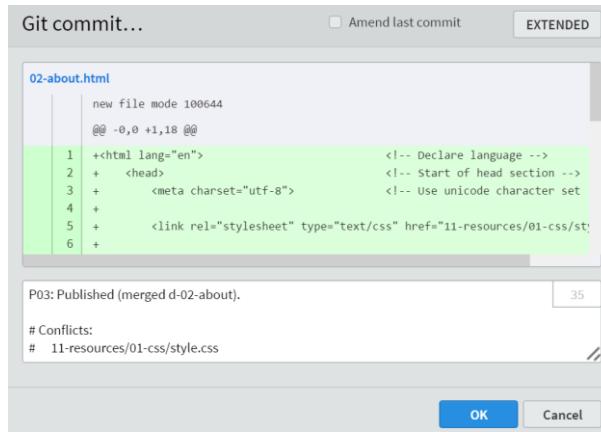


Figure 6.79 Brackets—Post conflict commit

Just click **OK**. Brackets will automatically change to the **master** branch. In my case the new commit is [\[07fe437\]](#). Tag the new commit **P03** and view the history:

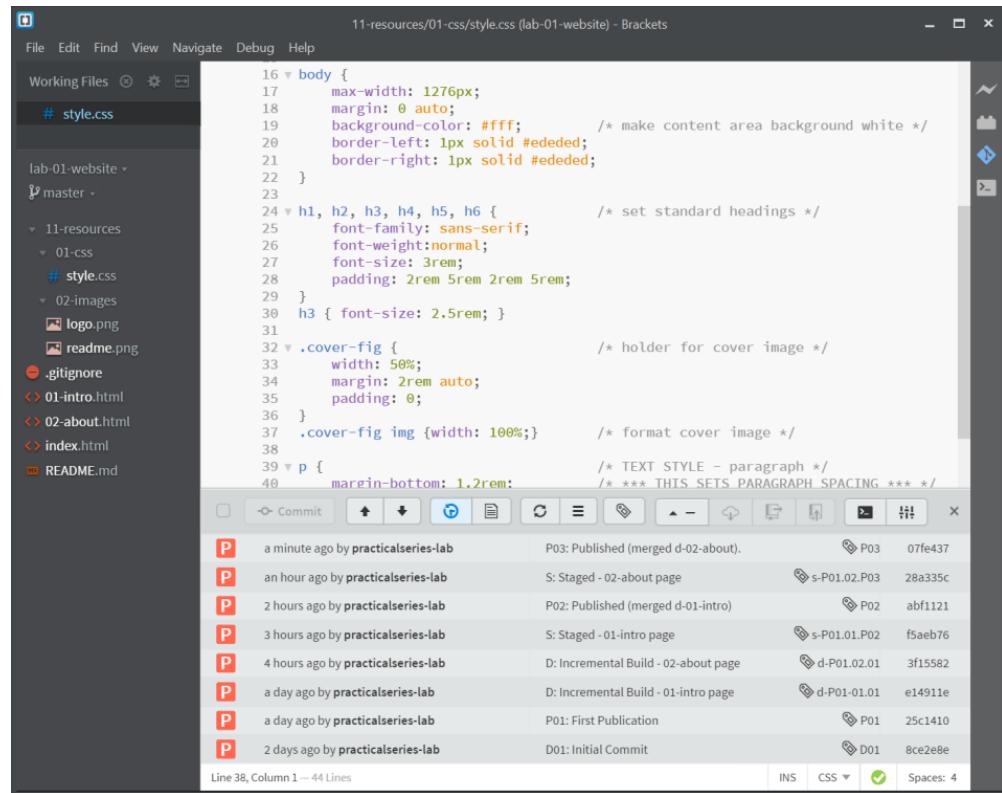


Figure 6.80 Brackets—**master** branch after commit

The workflow is now:

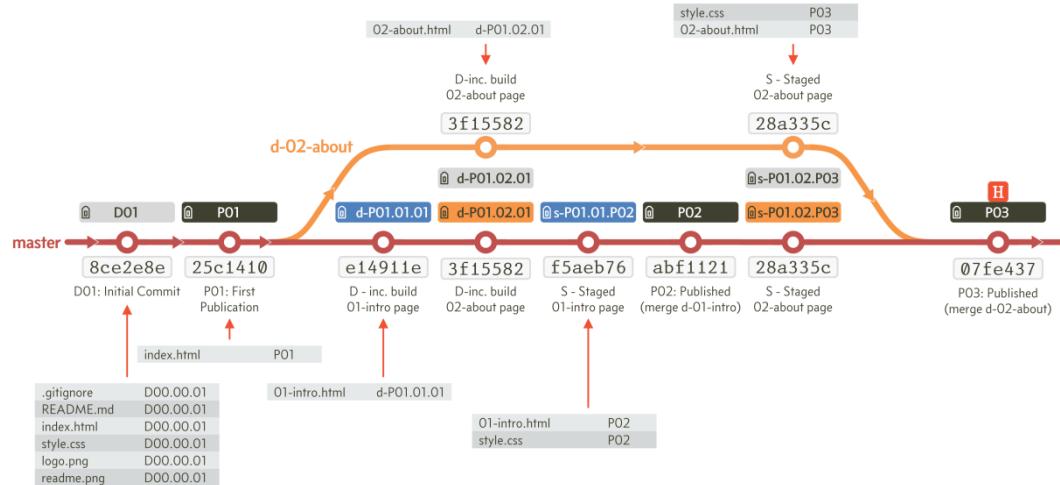


Figure 6.81 Merging the second branch after conflict resolution

Finally, the last thing to do is delete the redundant branch. Again click the small arrow next to the **master** branch in the file tree, click the delete icon next to **d-02-about** in the dropdown menu.

We're done. The final workflow is all on the **master** branch, Figure 6.82:

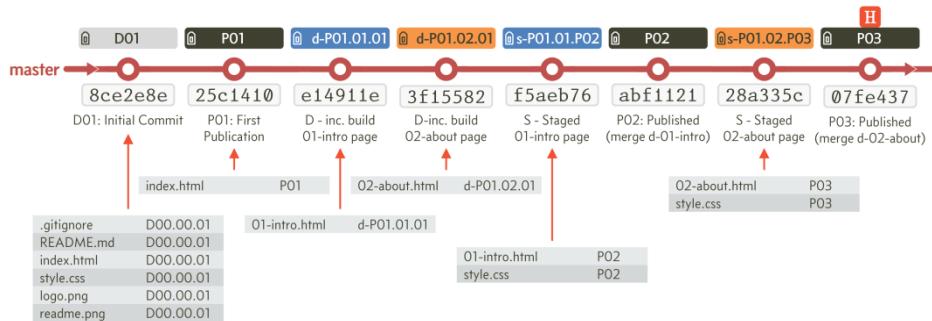


Figure 6.82 Final arrangement, all merged back to master branch

All the commits from the two branches that were created are now merged onto the **master** branch line.

7

REGRESSION WITH BRACKETS

How to use Brackets to regress to an earlier point in a project and copy the files from that commit point.

BEING ABLE TO RECOVER INFORMATION from an earlier point in a project is an important aspect of any version control system.

In section 2.5 I explained the process of *resetting* a project back to an earlier commit point; I also explained my best practice for doing so.

Brackets has a reset facility that allows this process to take place within the Brackets environment (albeit with some limitations).

Brackets also supports a *checkout* function that performs a similar action but without the restrictions of the reset function.

Git also supports this *checkout an earlier commit* function.

I cover all these options in this section with the aid of the [lab-01-website](#) project developed in the previous section.

7.1

Brackets—enabling the reset and checkout functions

The reset and checkout functions are normally available when looking at the commit history of a project; however the functions must be enabled in the Bracket-Git settings.

Open the [Lab-01-website](#) project in Brackets (this is the project we developed in section 6) and activate the Git pane by clicking its icon in the right side bar (↗).

To access the Brackets-Git settings, either click the settings button in the Git pane , or select [FILE → GIT SETTINGS....](#). They open the following settings dialogue box.

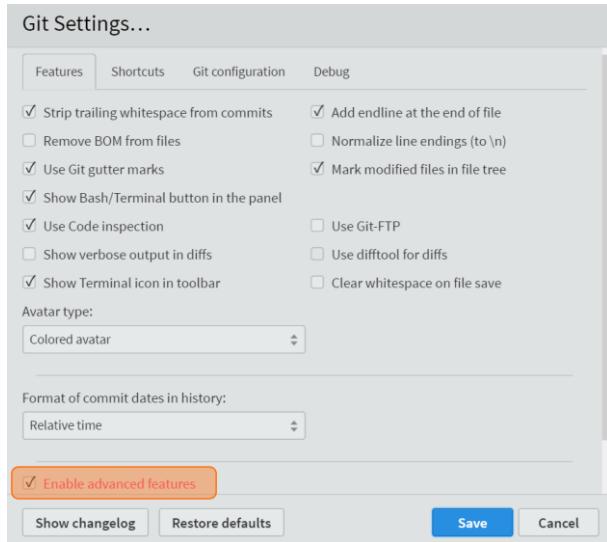


Figure 7.1 Brackets-Git settings dialogue box

The thing we need is at the bottom. Make sure [ENABLE ADVANCED FEATURES](#) is active (ticked). These advanced features are the very reset and checkout functions we're looking for. That's it, click [SAVE](#) and if you made any changes it will prompt to restart Brackets. Let it.

To see if it's worked, click the commit history button in the Git pane  and then click on any commit line to open the commit information screen (see § 6.3.3 for detailed instruction). You should have something like this:

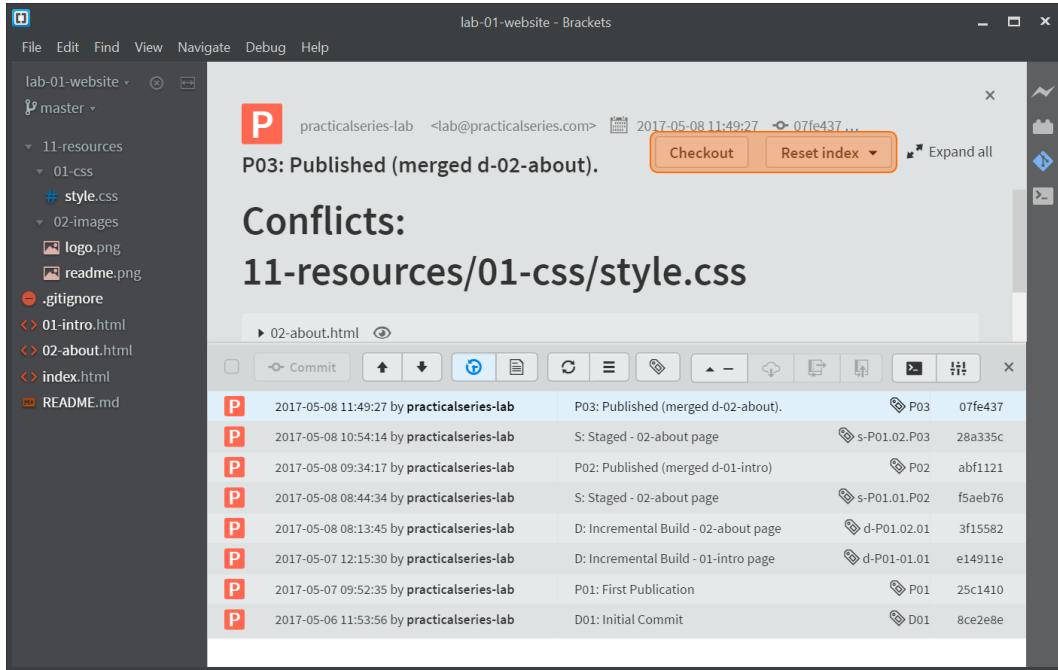


Figure 7.2 Bracket-Git reset and checkout buttons

The buttons are at the top (highlighted in Figure 7.2).

If you were to untick the [ENABLE ADVANCED FEATURES](#), the buttons would disappear.

7.1.1 Date and time settings

One other thing I've done here is change how the time and date of commits are displayed, in the previous section they were shown as relative dates (i.e. *commit made 3 hours ago*). I'm going to change this and use absolute times and dates for clarity (*if I use the relative ones, it keeps changing to mark the lapse of time*).

I'm going to use the standard **ISO 8601** date format (*this is what engineers always use*).

YYYY-MM-DD HH:mm:ss

To change how the date and time are displayed; reopen the same settings page (Figure 7.3):

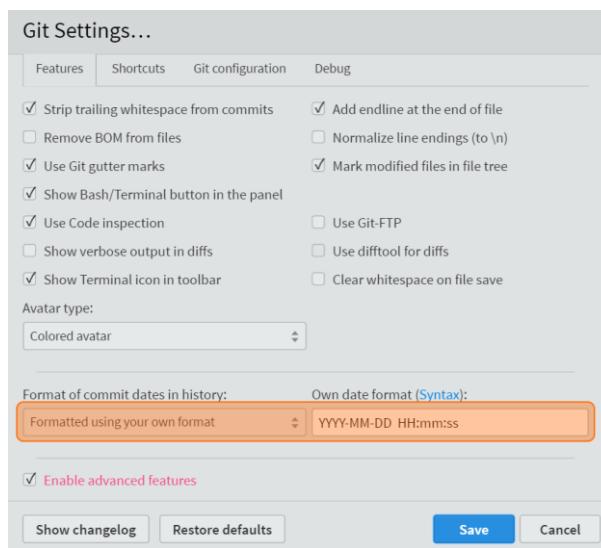


Figure 7.3 Bracket-Git Commit history date format

Change the **FORMAT OF COMMIT DATES IN HISTORY** to **FORMATTED USING YOUR OWN FORMAT**. Enter the format you want in the **OWN DATE FORMAT** box (if you click the blue **SYNTAX**, it will open a web page showing all the possible formats).

You can see the format I've used (the uppercase **HH** for hours shows it as a two digit 24 hour clock value).

7.2

Regressing to an earlier commit with Reset

Brackets supports two mechanisms for moving the project back (*regressing*) to an earlier commit point.

The mechanisms are called:

- Reset index
- Checkout

Now I discussed how the reset works in section 2.5. The Reset index above is this type of reset (it can do a *hard*, *mixed* or *soft* reset) and it work in exactly the same way I described in that section.

The reset index function in the Brackets environment has one or two restrictions (*restrictions is perhaps the wrong word, shortcomings might be better*) that can make things a little difficult.

For this reason I recommend:

**ONLY EVER USE THE CHECKOUT FUNCTION
TO ACCESS PREVIOUS COMMITS**

I cover checkouts in the next section. For now I will explain the Reset index function and its *shortcomings*.

7.2.1 Resetting to an earlier commit point

Currently the project looks like this:

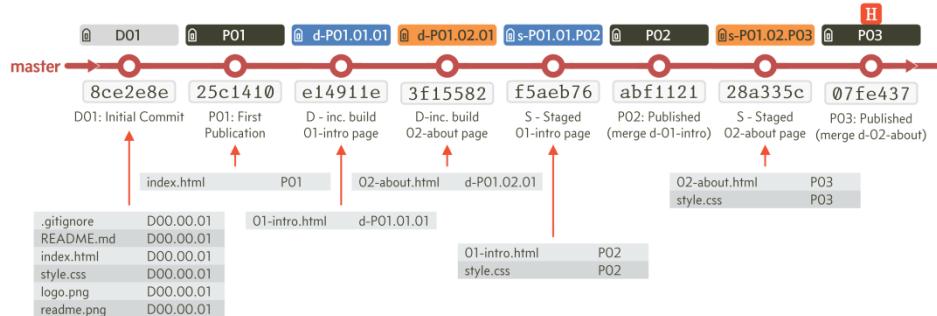


Figure 7.4 Final arrangement, all merged back to master branch

There is a single **master** branch with eight commits and the **head** is at the most recent commit, in my case [\[07fe437\]](#).

Looking at this in the Brackets commit history (click) I have:

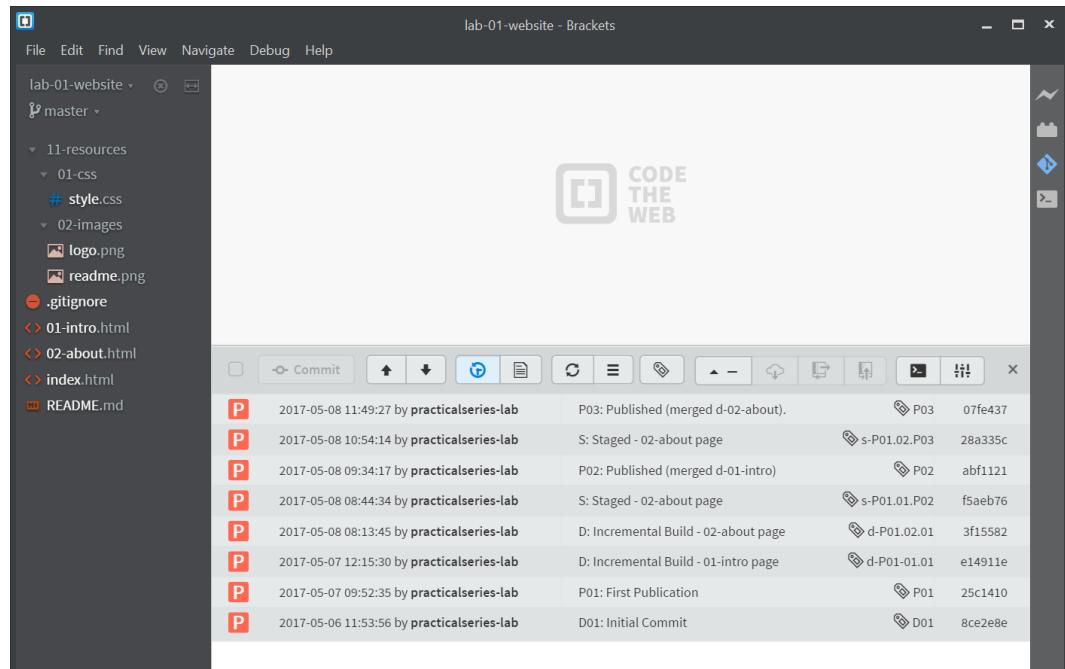


Figure 7.5 Brackets commit history

Note: At this point I suggest you take a screen shot of your commit history —you will regret it if you don't—as you will see it's quite handy knowing the commit numbers.

There are a few things to note here: firstly every commit point has a tag (*this isn't necessarily vital, but it allows me to indicate exactly which commit point I'm referring to in the following text*).

Secondly we have eight files:

LIST OF FILES AT COMMIT 07FE437	TAG: P03
index.html	
01-intro.html	
02-about.html	
README.md	
.gitignore	
11-resources\01-css\style.css	
11-resources\02-images\logo.png	
11-resources\02-images\readme.png	

Table 7.1 File list at commit point P03 [07FE437]

If I close the commit history Git is reporting:

Nothing to commit, working directory clean

So there are no changes pending (the files in the working area and staging all match the committed files).

So let's do a reset, I'm going to go right back to the first P01 commit point [25c1410] in my list.

To do this, open the commit history (Figure 7.5) and click anywhere on the P01 commit line [25c1410]. This will open the commit information screen:

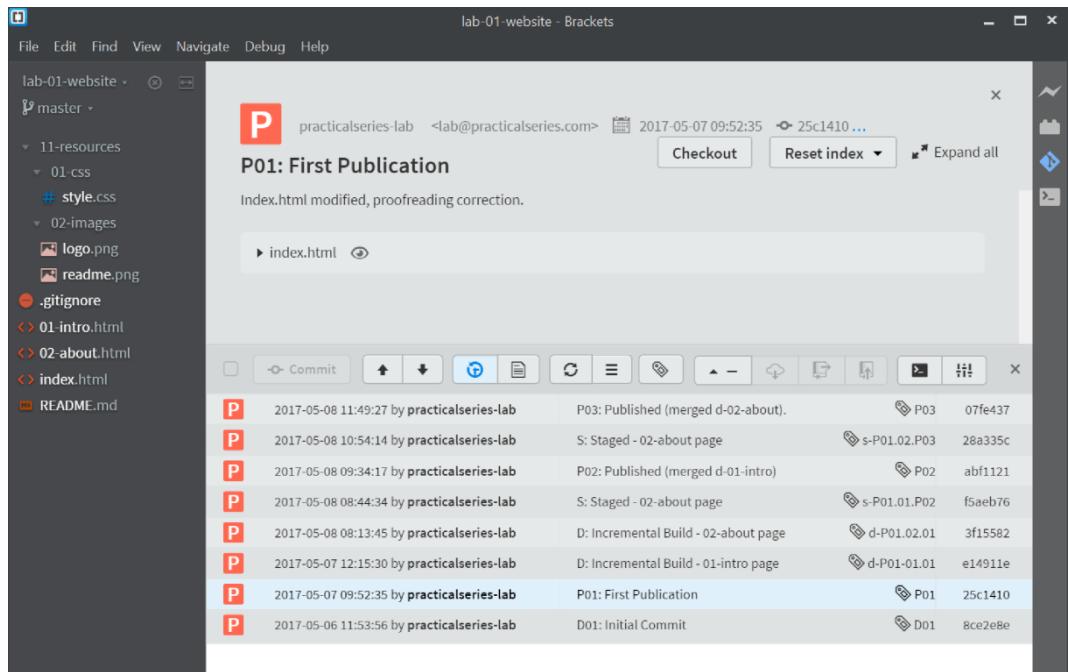


Figure 7.6 P01 commit point information screen

Now click the `RESET INDEX` button `Reset index ▾`. This will open the reset type dropdown menu:

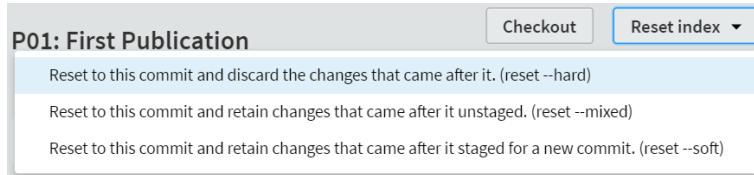


Figure 7.7 Reset type menu

As I said in section 2.5, the only type of reset to do is a hard reset:

THE BEST RESET IS A HARD RESET

So let's do that, click the top line (`RESET --HARD`).

Now we get an “are you sure?” box:

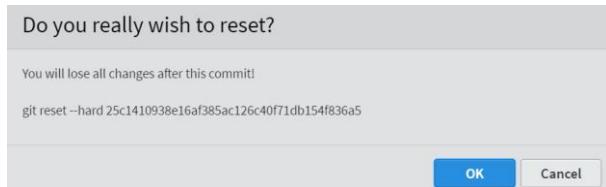


Figure 7.8 Reset “are you sure?” box

Click [OK](#).

Now we've done it.

Open the commit history:

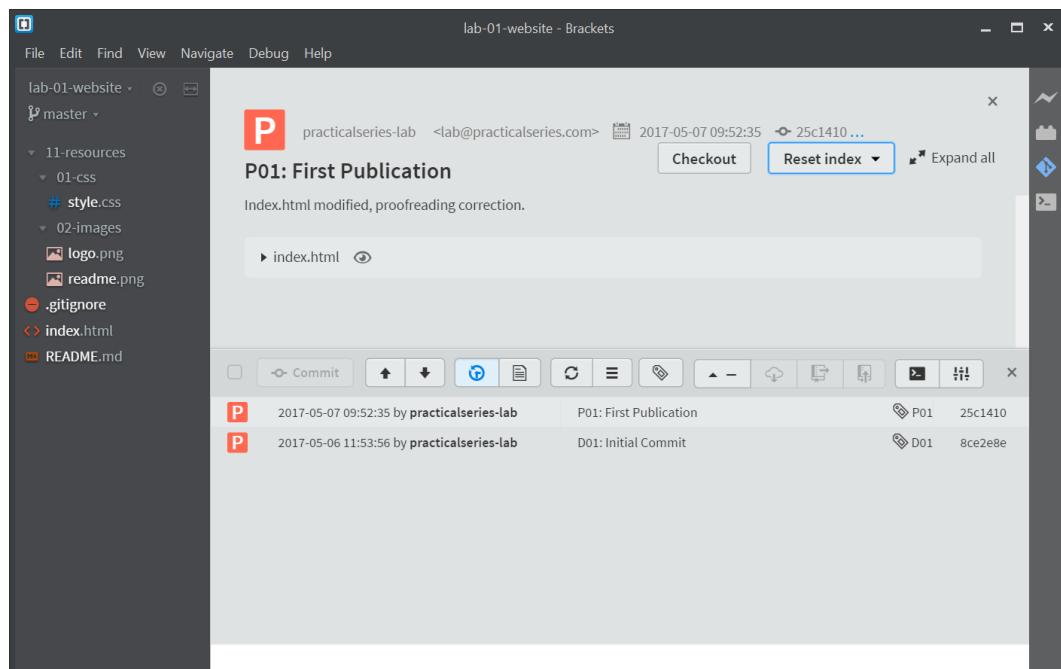


Figure 7.9 Commit history after the reset

Well, we're still on the master branch. But we now only have two commit points ([D01](#) and [P01](#)).

The files have changed too; we now only have these files:

LIST OF FILES AT COMMIT 25C1410	TAG: P01
index.html	
README.md	
.gitignore	
11-resources\01-css\style.css	
11-resources\02-images\logo.png	
11-resources\02-images\readme.png	

Table 7.2 File list after reset to P01 [25c1410]

The files that were added after have disappeared:

01-intro.html
02-about.html

Also if we were to look inside the `lab-01-website` folder with Windows Explorer we would only see `index.html`, `README.md` and `.gitignore`.

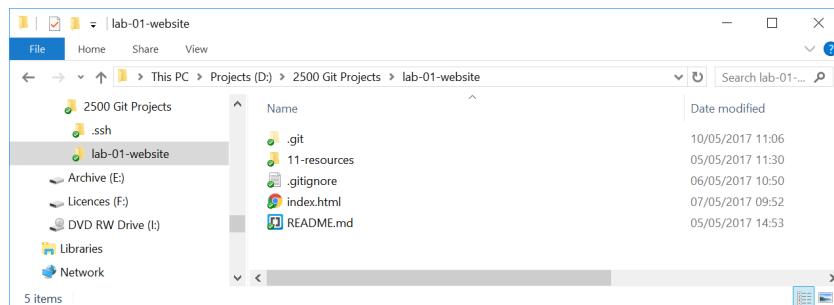


Figure 7.10 The working directory after the reset

The project really has moved back in time to the earlier commit points, if I opened any of the project files, they would be exactly as they were at the time of the commit.

Using the best practice approach to resetting I discussed in section 2.5.4, this would allow us to copy any or all of the files from this commit point and, after moving back to the most recent commit point, paste them into the working area and make a new commit with the old files from the earlier commit (i.e. replace the most up to date file with an older file).

This leads to the question: “*how do we get back to the most recent commit?*”

Errr...

Well you can’t. Not just like that. Not from within Brackets.

It’s like we went back in time and the Tardis broke down and now were stuck with the bloody Victorians (*and possibly Peter Capaldi, which would be worse*). Actually, it’s more like being stuck in the eighties with Zork.

And fortunately, in the eighties we had access to the command line; and boy can that thing can bail us out of some shit. (*I don’t know how we managed without it—I take back everything I said*).

We can start the Git Bash command line terminal emulator from Brackets; in the Git Pane click the terminal icon , if you ticked the **SHOW TERMINAL ICON IN TOOLBAR** option in the Git settings (Figure 7.1) you can also start the terminal emulator from the right sidebar icon: .

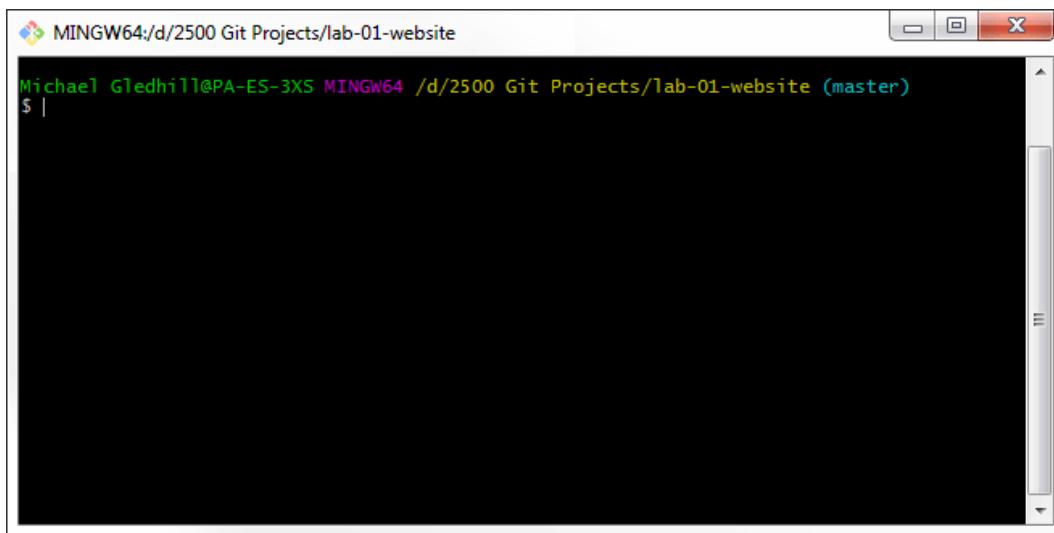


Figure 7.11 Git terminal emulator

I hope you kept a record of all the commits.

We've reset the project back to commit point [25c1410]; this can be seen from the history in Figure 7.9.

Now the latest commit, is commit [07fe437]; this can be seen in the original history before the reset, Figure 7.5.

So in my case, I want to move the **head** back to commit point [07fe437] to put everything back how it was. I have to do this in the command line. Enter the following:

```
$ git reset 07fe437 --hard
```

Obviously, you need to enter your commit number. It gives me this:

```
Michael Gledhill@PA-ES-3XS MINGW64 /d/2500 Git Projects/lab-01-website (master)
$ git reset 07fe437 --hard
HEAD is now at 07fe437 P03: Published (merged d-02-about).

Michael Gledhill@PA-ES-3XS MINGW64 /d/2500 Git Projects/lab-01-website (master)
$ ...
```

Figure 7.12 Hard reset using the command line

And if I now look at the history in Brackets, I'm back where I started:

P	2017-05-08 11:49:27 by practicalseries-lab	P03: Published (merged d-02-about).	P03	07fe437
P	2017-05-08 10:54:14 by practicalseries-lab	S: Staged - 02-about page	s-P01.02.P03	28a335c
P	2017-05-08 09:34:17 by practicalseries-lab	P02: Published (merged d-01-intro)	P02	abf1121
P	2017-05-08 08:44:34 by practicalseries-lab	S: Staged - 02-about page	s-P01.01.P02	f5aeb76
P	2017-05-08 08:13:45 by practicalseries-lab	D: Incremental Build - 02-about page	d-P01.02.01	3f15582
P	2017-05-07 12:15:30 by practicalseries-lab	D: Incremental Build - 01-intro page	d-P01-01.01	e14911e
P	2017-05-07 09:52:35 by practicalseries-lab	P01: First Publication	P01	25c1410
P	2017-05-06 11:53:56 by practicalseries-lab	D01: Initial Commit	D01	8ce2e8e

Figure 7.13 Brackets commit history

Phew...

What if I can't remember my original commit?

What did I say at the start of page 243? “*Take a screen shot of your commits*” I said, “*you'll regret it if you don't*”—sound familiar?

Ok, there is a couple of ways round this: firstly, you can reset to a tag.

In the above example I got back to the most recent commit point by resetting to a commit number:

```
$ git reset 07fe437 --hard
```

The syntax for this is:

```
$ git reset <commit number> --<type of reset>
```

The order is `commit number` followed by `type of reset`.

In my case the commit point `[07fe437]` also has the tag `P03` (you can see this in Figure 7.13).

To reset to the same point using a tag, the command is:

```
$ git reset --hard P03
```

The syntax being:

```
$ git reset --<type of reset> <tag name>
```

Yep, the order is `type of reset` followed by the `tag name`.

That's not at all confusing is it? But there we are, what can I say?—Linux people.

Ah, but what if it the commit doesn't have a tag and I can't remember the commit number?

Well, Git can always list all of the commits.

Do a hard reset back to the `P01` commit point (same as I did above).

You will just have the two commit points of Figure 7.9.

Open the Git command line terminal and type:

```
$ gitk --all
```

You should get something that looks like this:

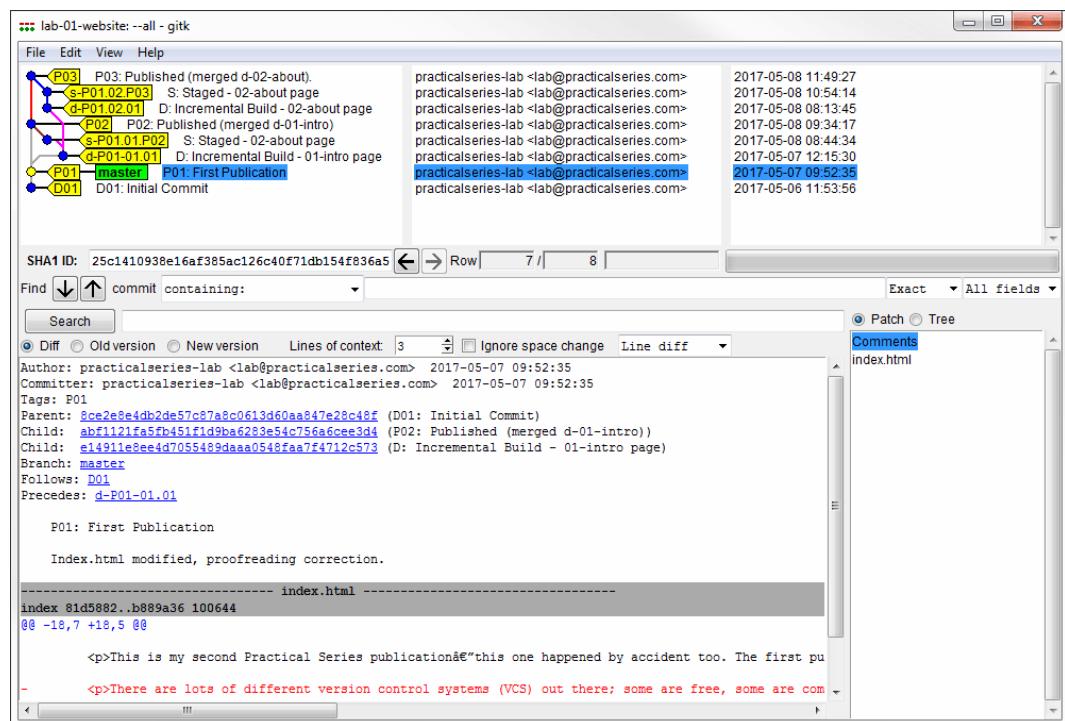


Figure 7.14 GITK, a commit history viewer

Wow—WTF is that? You ask.

`gitk` is the commit history viewer that was installed when you install Git. We invoked it when we entered `gitk` in the command line. The `--all` tells `gitk` to show all commit points.

The important bit is the little diagram at the top. This is currently showing information about the current `head` point (`P01` in blue in Figure 7.14).

We want the commit number of the most recent commit; this is at the top of the diagram. To get the commit number click the commit message next to the commit (it starts ‘P03: published’):

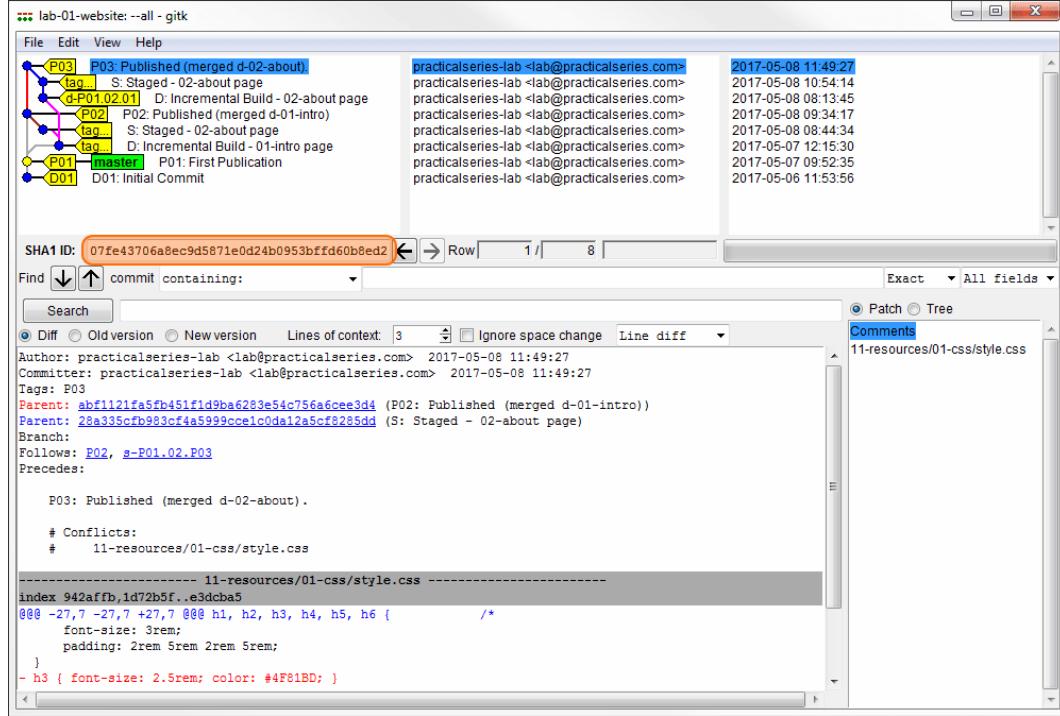


Figure 7.15 GITK, find a commit number

The commit number of the selected commit point is shown in the [SHA-1](#) field just under the branch diagram (highlighted in orange in Figure 7.15). This field can be selected (wholly or in part) and copied to the clipboard, remember you only need to copy the first seven digits of the commit hash number (*it will work if you copy all of it too*).

Copy what you need, close [gitk](#) and enter the reset command in the command line:

```
$ git reset <copied hash> --hard
```

OK, that’s resetting—it’s a pain in the arse.

The better way, as I said at the start, is to use the [CHECKOUT](#) function. I cover this in the next section.

7.3

Regressing to an earlier commit with Checkout

I can't stress strongly enough, if you want to look at an earlier commit point do it with the checkout function.

I'm going to start from the same point I used for the reset in the previous section:

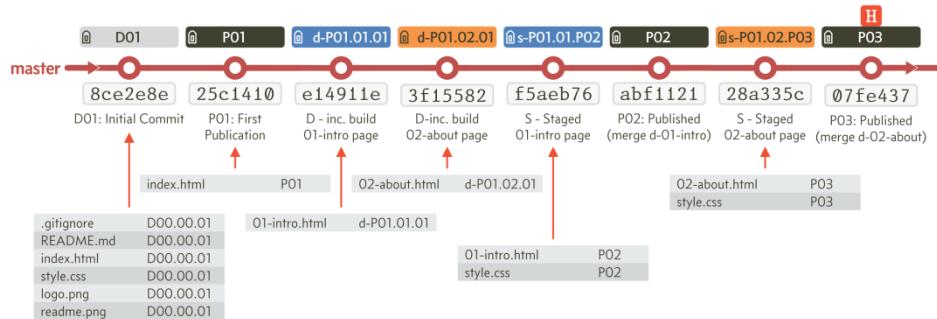


Figure 7.16 Final arrangement, all merged back to master branch

There is a single **master** branch with eight commits and the **head** is at the most recent commit, in my case [\[07fe437\]](#). In Brackets, the history is:

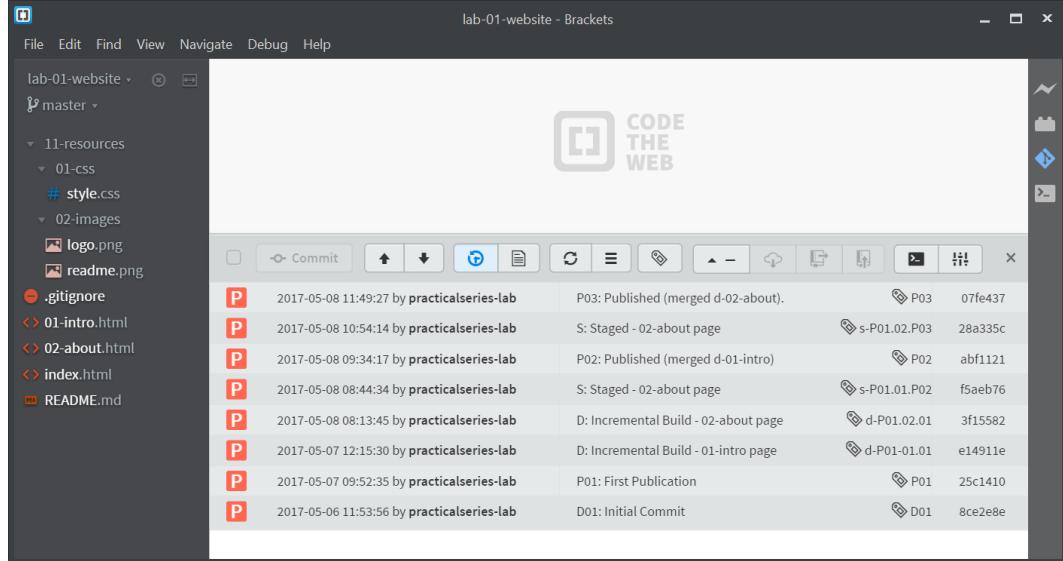


Figure 7.17 Brackets commit history

And we have the following files:

LIST OF FILES AT COMMIT 07FE437	TAG: P03
index.html	
01-intro.html	
02-about.html	
README.md	
.gitignore	
11-resources\01-css\style.css	
11-resources\02-images\logo.png	
11-resources\02-images\readme.png	

Table 7.3 File list at commit point P03 [07FE437]

Finally, Git is reporting:

Nothing to commit, working directory clean

So let's do a checkout, again I'm going back to the first P01 commit point, [25c1410] in my list.

To do this, open the commit history (Figure 7.17) and click anywhere on the P01 commit line [25c1410]. This will open the commit information screen:

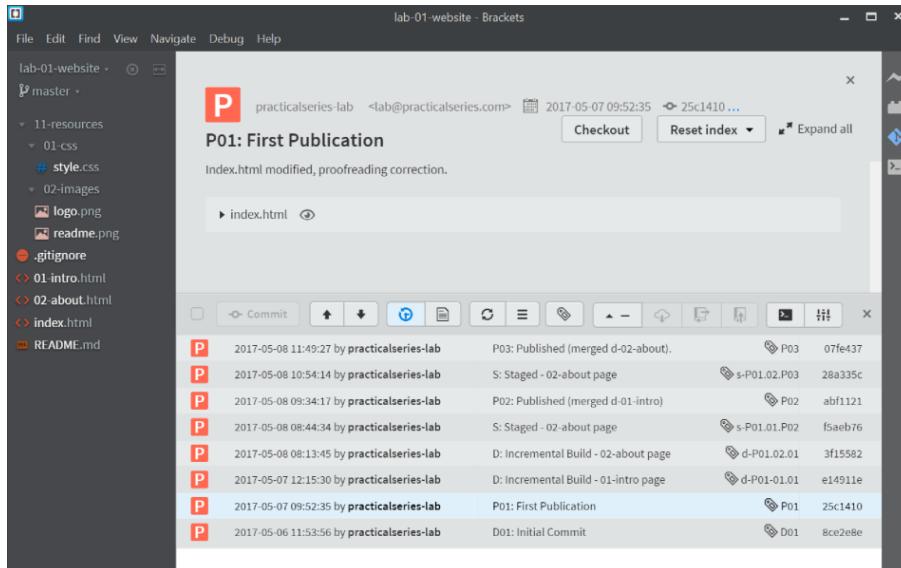


Figure 7.18 P01 commit point information screen

This time click the **CHECKOUT** button  . This will open the checkout warning dialogue box:

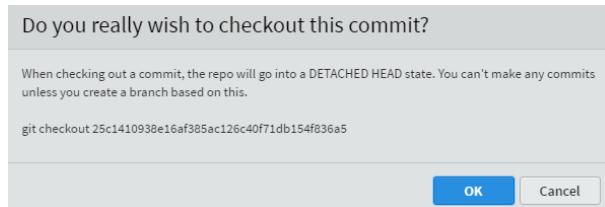


Figure 7.19 Checkout function warning

This is telling us that the project is going into a **detached head** state and that we can't make any further **commits**. This is ok, we don't want to make any commits—we just want to have a look. Click **OK**.

Open the commit history, and let's see what we have:

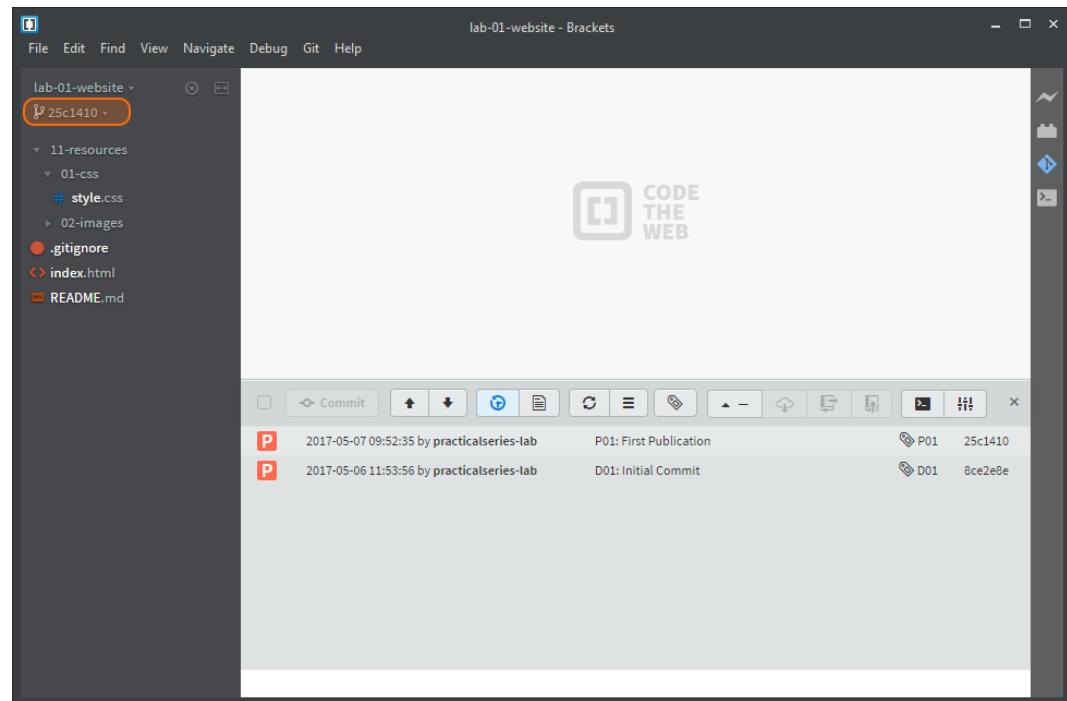


Figure 7.20 Commit history after the checkout

Right, things are a bit different this time. We have just the two commit points ([D01](#) and [P01](#)) exactly as we had with the reset, but this time we are not on the **master** branch.

We are now on a branch called **25c1410**. The **master** branch is still there, click the down arrow next to the **25c1410** branch and you will see it:

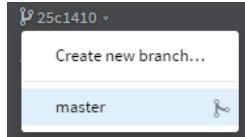


Figure 7.21 Branches present after a checkout

We could switch back to it, and we will; but first let's look around.

The files have changed to match the earlier commit point (just like with the reset); we now have these files:

LIST OF FILES AT COMMIT 25C1410	TAG: P01
index.html	
README.md	
.gitignore	
11-resources\01-css\style.css	
11-resources\02-images\logo.png	
11-resources\02-images\readme.png	

Table 7.4 File list after reset to [P01 \[25c1410\]](#)

These files have disappeared:

[01-intro.html](#)
[02-about.html](#)

Again if we were to look inside the [lab-01-website](#) folder with Windows Explorer we would only see [index.html](#), [README.md](#) and [.gitignore](#).

Just like it was with the reset.

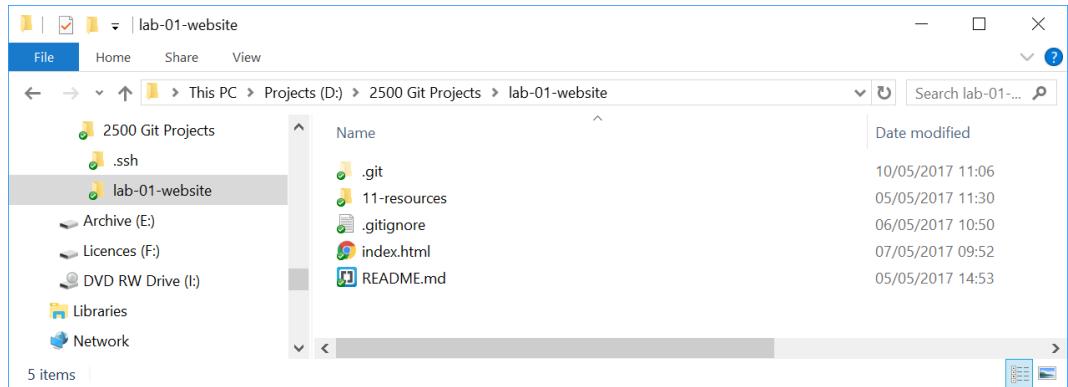


Figure 7.22 The working directory after the checkout

So, the project has moved back in time to the earlier commit points in exactly the same way as it did with the reset and if I opened any of the project files, they would be exactly as they were at the time of the [\[25c1410\]](#) commit.

The best practice approach to resetting (§ 2.5.4), is that if we want to reinstate the files at an earlier commit, we copy any or all of the files from this checked out commit point and, after moving back (*as in back to the future*) to the most recent commit point, paste them into the working area and make a new commit with the old files from the earlier commit (i.e. replace the most up to date files with older files).

This, if you remember is where the reset fell down, we couldn't easily get back to the most recent commit point (there was a lot of messing about with the command line).

Things are much easier with the checkout function.

The most recent commit point (in my case [\[07fe437\]](#)) is the **head** of the **master** branch. To get back, just switch to the **master** branch.

Click the arrow next to the [25c1410](#) branch and in the dropdown click **MASTER**:

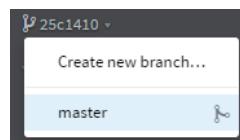
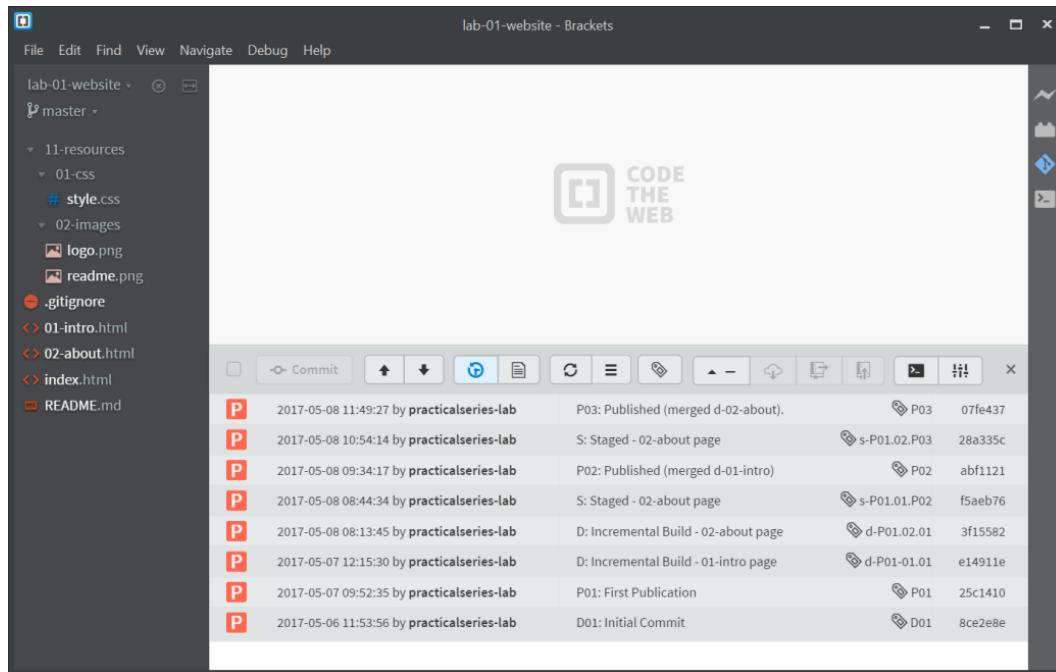


Figure 7.23 Change branch present after a checkout

This instantly takes us back to where we were; open the commit history and everything is back where it was—easy:



The screenshot shows the Brackets IDE interface with the title bar "lab-01-website - Brackets". The left sidebar displays the file structure of the "lab-01-website" repository, including the "master" branch, "11-resources" folder, "01-css" folder containing "style.css", "02-images" folder containing "logo.png" and "readme.png", ".gitignore", "01-intro.html", "02-about.html", "index.html", and "README.md". The main panel shows the "CODE THE WEB" logo. Below the logo is a commit history table with the following data:

Date	Author	Message	SHA
2017-05-08 11:49:27	practicalseries-lab	P03: Published (merged d-02-about)	07fe437
2017-05-08 10:54:14	practicalseries-lab	S: Staged - 02-about page	s-P01.02.P03 28a335c
2017-05-08 09:34:17	practicalseries-lab	P02: Published (merged d-01-intro)	P02 abf1121
2017-05-08 08:44:34	practicalseries-lab	S: Staged - 02-about page	s-P01.01.P02 f5aeb76
2017-05-08 08:13:45	practicalseries-lab	D: Incremental Build - 02-about page	d-P01.02.01 3f15582
2017-05-07 12:15:30	practicalseries-lab	D: Incremental Build - 01-intro page	d-P01-01.01 e149110
2017-05-07 09:52:35	practicalseries-lab	P01: First Publication	P01 25c1410
2017-05-06 11:53:56	practicalseries-lab	D01: Initial Commit	D01 8ce2e8e

Figure 7.24 Returning to the most recent commit

This time if you click the arrow next to the **master** branch, there will not be any other branches listed, the **25c1410** branch has disappeared.

The **25c1410** branch that was created by the checkout is just a temporary branch—it disappears when we switch off it to any other branch.

The checkout is a much better way of viewing earlier commits.

**ONLY EVER USE THE CHECKOUT FUNCTION
TO ACCESS PREVIOUS COMMITS**

8

BRACKETS AND REMOTE REPOSITORIES

How to use Brackets to link with and manage a remote repository.

IN THIS SECTION I cover linking the [lab-01-website](#) project to a remote GitHub repository and show how to push, pull and merge changes between a local repository and its remote counterpart.

I cover:

- ① Creating an empty remote repository with GitHub
- ② Pushing a local repository up to the GitHub remote repository
- ③ Understanding how branches work with a remote repository
- ④ Incorporating remote changes into the local repository
- ⑤ Deleting both local and remote branches

Section 4 covered how to create a GitHub profile and how to link it to a local machine using an SSH link. I'm assuming in this section that you have done this with your own account and will be able to follow the steps accordingly.

I'm also assuming that you have the [lab-01-website](#) repository (I set this up in section 6; this will form the basis of the remote repository).

8.1

Extending the current project

Currently the project looks like this:

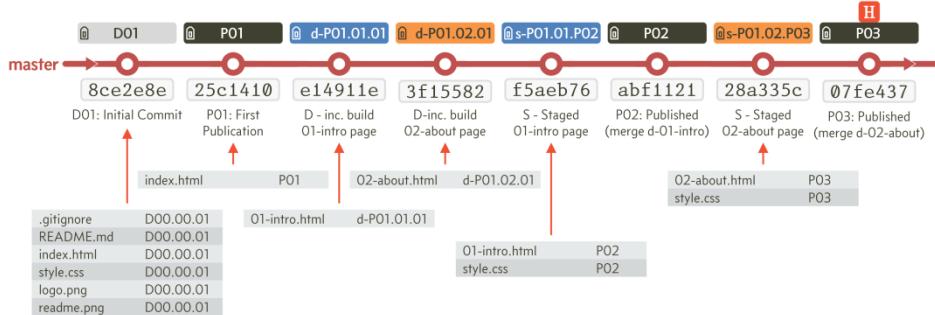


Figure 8.1 Final arrangement, all merged back to master branch

There is a single **master** branch with eight commits and the **head** is at the most recent commit, in my case [\[07fe437\]](#). In Brackets, the commit history is:

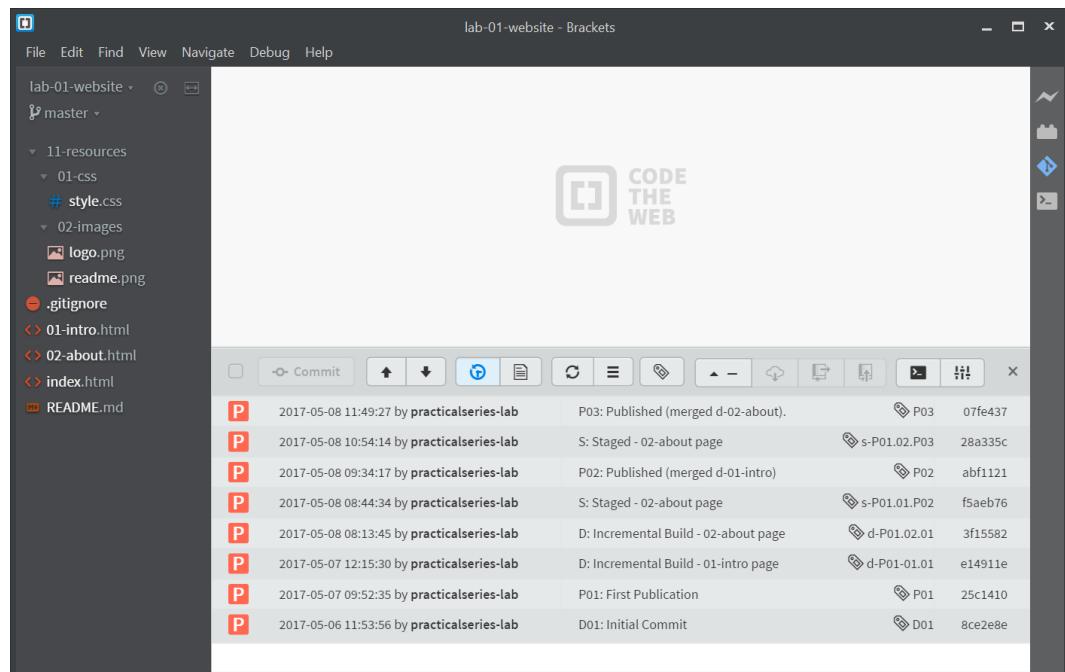


Figure 8.2 Brackets commit history

The repository has the following eight files:

LIST OF FILES AT COMMIT 07FE437	TAG: P03
index.html	
01-intro.html	
02-about.html	
README.md	
.gitignore	
11-resources\01-css\style.css	
11-resources\02-images\logo.png	
11-resources\02-images\readme.png	

Table 8.1 File list at commit point P03 [07FE437]

Before we start working with the remote repository, I want to create a new branch and commit point in the local project—this is to demonstrate how GitHub handles branches.

I'm going to add a new `03-contact.html` page and I'm going to do so by creating a new branch from the most recent commit point [07fe437] on the `master` branch. From the `master` branch, create a new `d-03-contact` branch and switch to it (see § 6.5 for instructions about creating and changing branches).

Now we have Figure 8.3, the head is on the new branch, but there are no further commits yet:

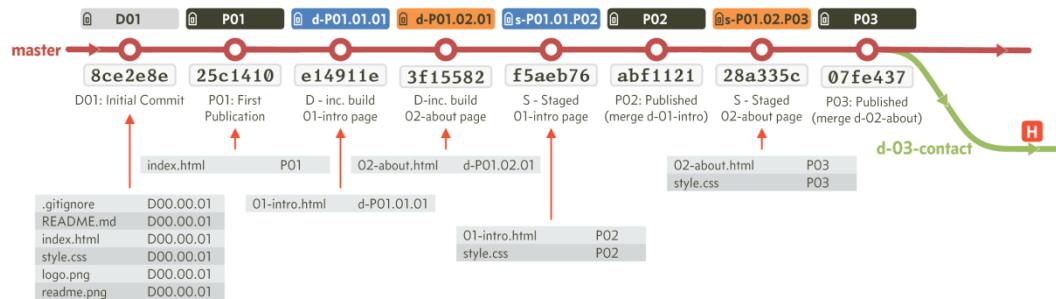


Figure 8.3 The `d-03-contact` branch

Time for another file; in the new `d-03-contact` branch create a file `03-contact.html` in the root folder (same place as `index.html`) and add the following code to it (This is just like we did in § 6.5.2):

```

03-CONTACT.HTML

1 <html lang="en">
2   <head>
3     <meta charset="utf-8">
4
5     <link rel="stylesheet" type="text/css" href="11-resources/01-css/style.css">
6
7     <title>PracticalSeries: Git Lab – Contact us</title>
8   </head>
9
10  <body>
11    <h1>A Practical Series Website</h1>
12
13    <h3>Contact Us</h3>
14
15    <p>This page explains how to email Practical Series.</p>
16
17  </body>
18 </html>

```

Code 8.1 d-03-contact branch—create new file: 03-contact.html

Save the file and in the Git pane tick the box to stage the file:

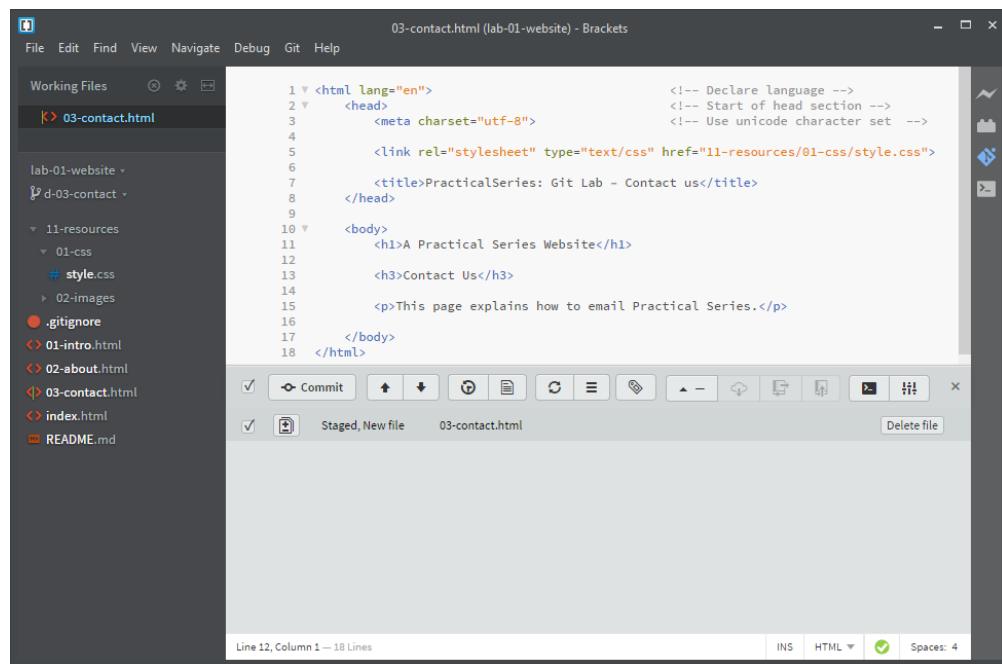


Figure 8.4 03-contact.html

Commit the changes with the commit message:

D: Incremental Build - 03-contact page

03-contact added.

Give the new commit point the tag [d-P03.03.01](#). In my case the commit point is [\[8329b67\]](#).

I now have nine commits:

P	2017-05-13 09:12:04 by practicalseries-lab	D: Incremental Build - 03-contact page	d-P03-03-01	8329b67
P	2017-05-08 11:49:27 by practicalseries-lab	P03: Published (merged d-02-about).	P03	07fe437
P	2017-05-08 10:54:14 by practicalseries-lab	S: Staged - 02-about page	s-P01.02.P03	28a335c
P	2017-05-08 09:34:17 by practicalseries-lab	P02: Published (merged d-01-intro)	P02	abf1121
P	2017-05-08 08:44:34 by practicalseries-lab	S: Staged - 02-about page	s-P01.01.P02	f5aeb76
P	2017-05-08 08:13:45 by practicalseries-lab	D: Incremental Build - 02-about page	d-P01.02.01	3f15582
P	2017-05-07 12:15:30 by practicalseries-lab	D: Incremental Build - 01-intro page	d-P01-01.01	e14911e
P	2017-05-07 09:52:35 by practicalseries-lab	P01: First Publication	P01	25c1410
P	2017-05-06 11:53:56 by practicalseries-lab	D01: Initial Commit	D01	8ce2e8e

Figure 8.5 Current commits

The workflow is:

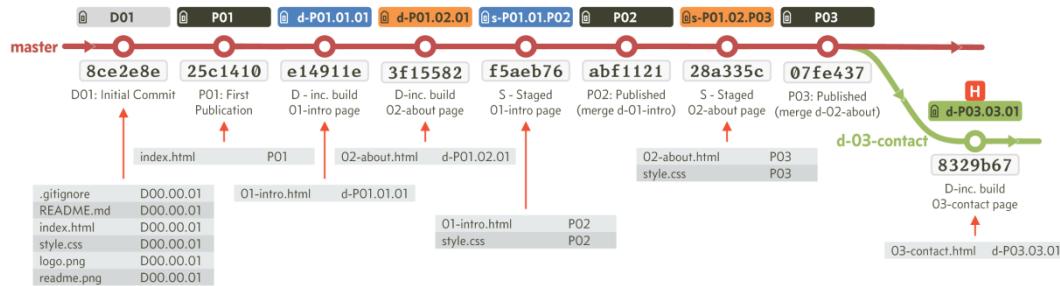


Figure 8.6 New branch with first commit

The last thing is to switch back to the **master** branch, the **master** branch is still the only deployable branch and it is this branch that I want to upload to the remote repository first.

Click the arrow next to the **d-03-contact** branch and select **master** from the dropdown.

This gives the following: back on the **master** branch there are only eight commits and the new file **03-contact.html** is not in the file list (this is on the **d-03-contact** branch and not visible from the **master** branch):

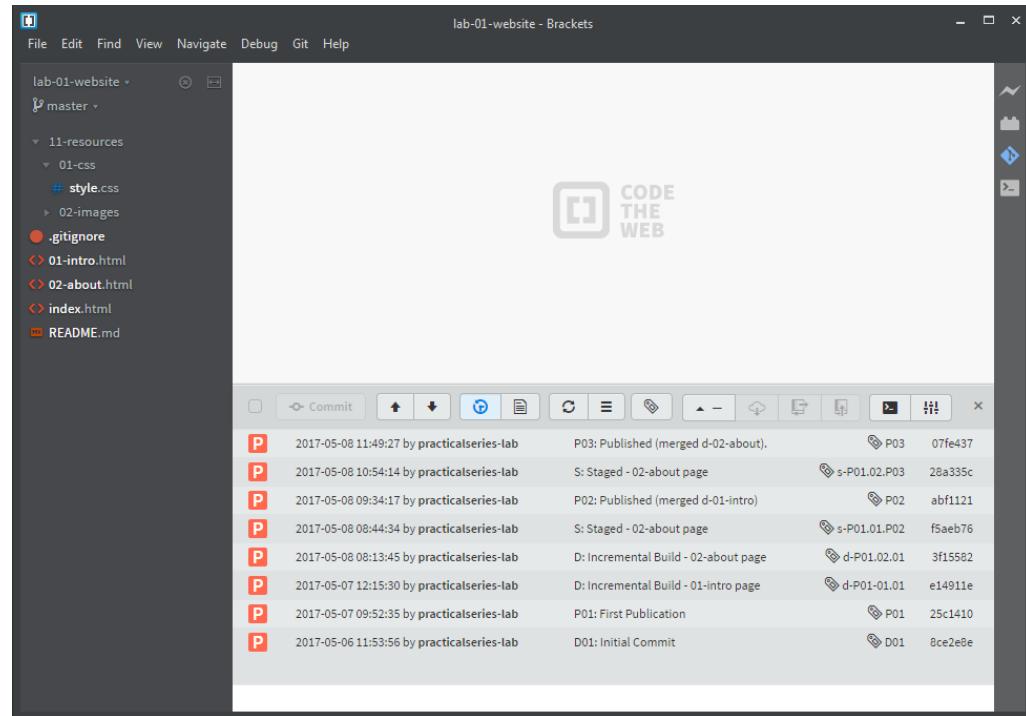


Figure 8.7 Local repository **master** branch

The work flow is now (just the same but with the **head** on the **master** branch):

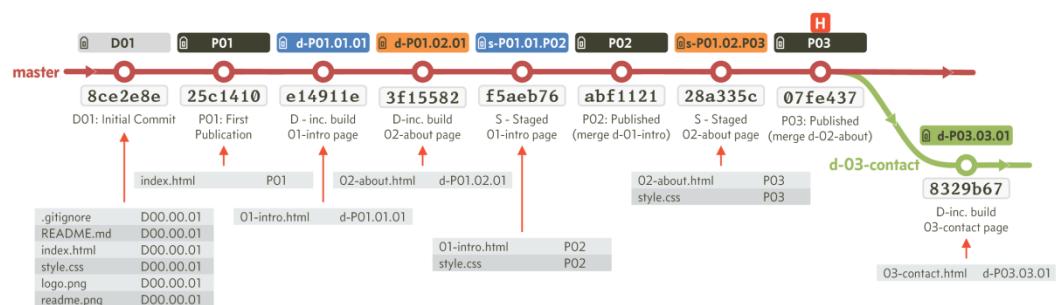


Figure 8.8 Project workflow—**head** on **master** branch

Leave it like this for now while we get the remote repository established.

8.2

Creating the remote repository

The first thing is to create an empty repository in GitHub:

Back to [GitHub](#), log into your account. This is mine:

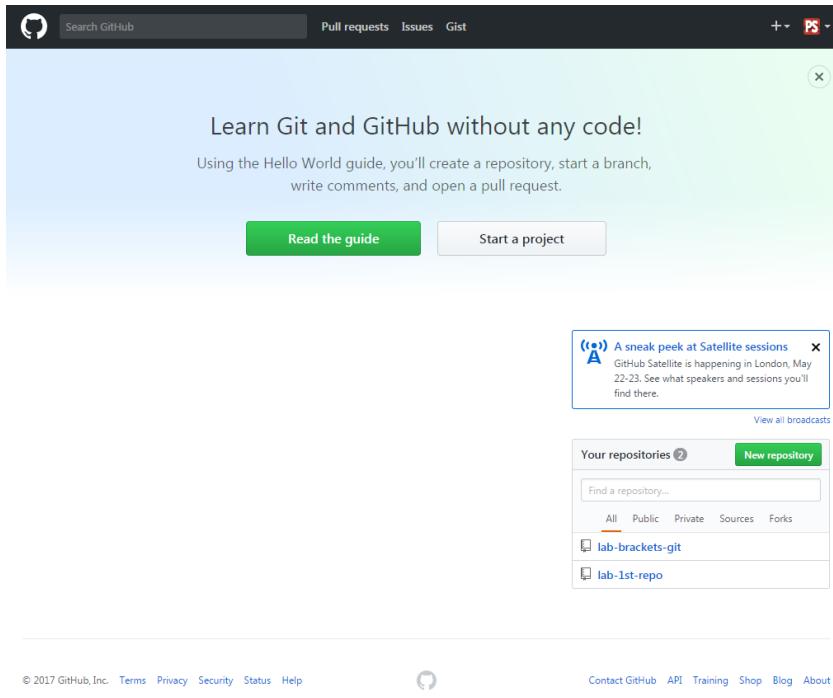


Figure 8.9 GitHub profile main page

There are currently two repositories: [lab-1st-repo](#) that we made in section 4.2 and [lab-brackets-git](#) that we made in section 5.2.

We're now going to make a third repository for the [lab-01-website](#) that is currently in our local repository.

I'm going to give the remote repository the same name as the local repository, i.e.:

[lab-01-website](#)

It is not necessary for the names to be the same; I do it only to avoid confusion.

To create the new remote repository in GitHub, click either the **START PROJECT** button or the **NEW REPOSITORY** button (they both do the same).

Either option will open the Create new repository page. I'm going to select the following options (Table 8.2 and Figure 8.10):

PROPERTY	VALUE
Repository name	lab-01-website
Description	A simple website repository used as an example to demonstrate how the Brackets-Git extension works
Public/private	Public
Initialise with README	Unticked (not selected)
Add .gitignore	None
Add a license	None

Table 8.2 lab-01-website repository properties

The screenshot shows the GitHub 'Create a new repository' interface. At the top, there's a navigation bar with icons for user profile, search, pull requests, issues, and gist. Below it is the title 'Create a new repository' with a subtitle 'A repository contains all the files for your project, including the revision history.' The main form has the following fields:

- Owner:** practicalseries-lab (dropdown menu)
- Repository name:** lab-01-website (text input field with a green checkmark)
- Description (optional):** (empty text input field)
- Visibility:** A radio button for **Public** is selected, with a note: 'Anyone can see this repository. You choose who can commit.' A radio button for **Private** is also present, with a note: 'You choose who can see and commit to this repository.'
- Initialization:** A checkbox for 'Initialize this repository with a README' is unchecked, with a note: 'This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.'
- Additional Options:** Buttons for 'Add .gitignore: None' and 'Add a license: None'.
- Create repository:** A green button at the bottom.

Figure 8.10 GitHub—Create the lab-01-website repository

Click **CREATE REPOSITORY** and it will automatically open the repository page:

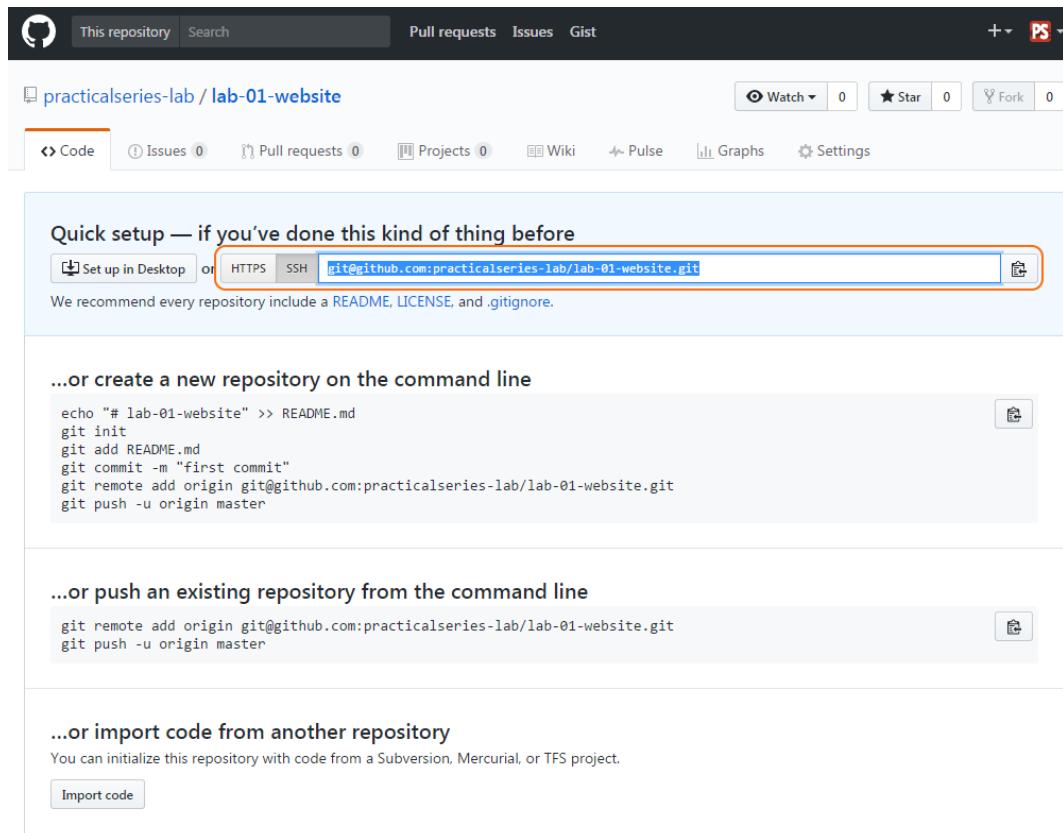


Figure 8.11 GitHub—lab-01-website repository page

This gives us a completely empty repository—GitHub is a bit *complainy* at this point (*ok, I might have made that word up*). It knows the repository is completely empty and it is telling us how to put something in it.

That's next.

Make a note of, or copy the SSH URL from the GitHub repository page (highlighted in orange above), you will need this for what follows.

8.3

Pushing a local repository to a remote

I'm going to assume that you've already set up the SSH link between your computer and your GitHub profile (the instructions for how to do this are in § 4).

If you have, all we need is the SSH URL from the GitHub repository page (highlighted in orange in Figure 8.11). In my case it is:

```
git@github.com:practicalseries-lab/lab-01-website.git
```

Yours will be different, copy it from your account.

8.3.1 Linking Brackets to the remote repository

The SSH URL is all we need to make the link—back to Brackets.

We have to tell Brackets about the remote repository. There is a button for this on the Git pane; it looks like this  it says **PICK PREFERRED REMOTE** if you hover the mouse over it. I've highlighted its position in Brackets in Figure 8.12:

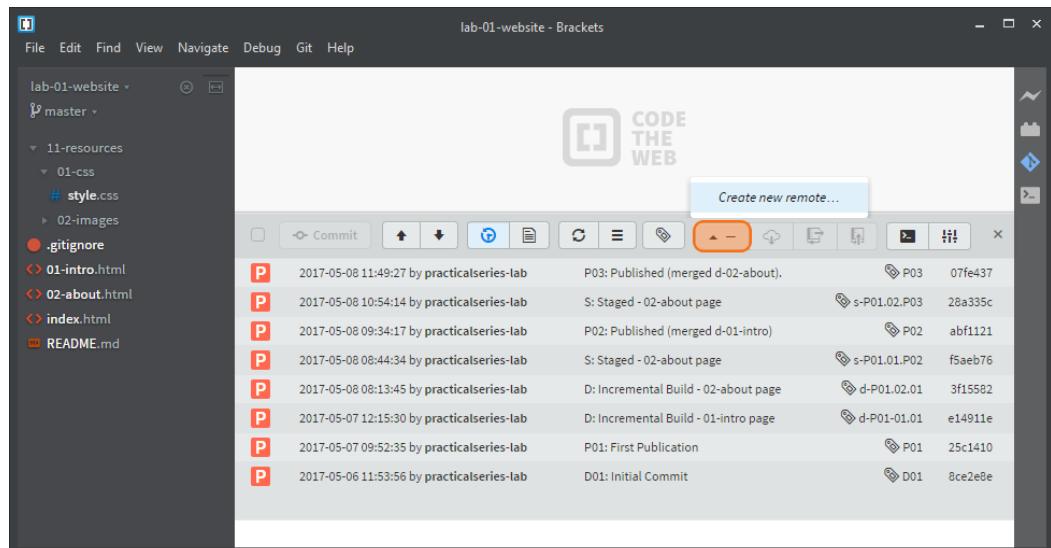


Figure 8.12 Brackets—pick remote repository button

Click it and select **CREATE NEW REMOTE** from the dropdown (shown above). This opens the create new remote dialogue box:

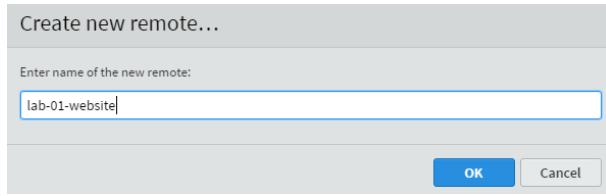


Figure 8.13 Create new remote dialogue box 1

This is asking for a name for the remote, call it **lab-01-website**. Click **OK**.

Note: The name I've used here lab-01-website, is the name of the remote link, this is usually called origin—I discussed this in § 2.9.5. It doesn't matter; you can use origin if you want.

Git uses origin by default, Brackets doesn't presume and you will always have to enter a name—hence I use the name of the repository.

Now it wants the information we copied from the new repository in GitHub.

Paste the information in and click **OK**.

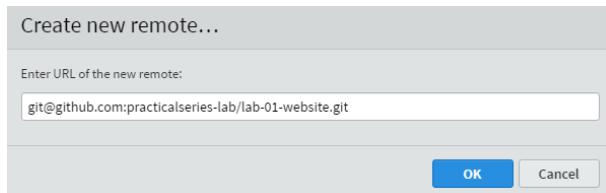


Figure 8.14 Create new remote dialogue box 2

This activates the remote **FETCH**, **PULL** and **PUSH** buttons on the Git pane:

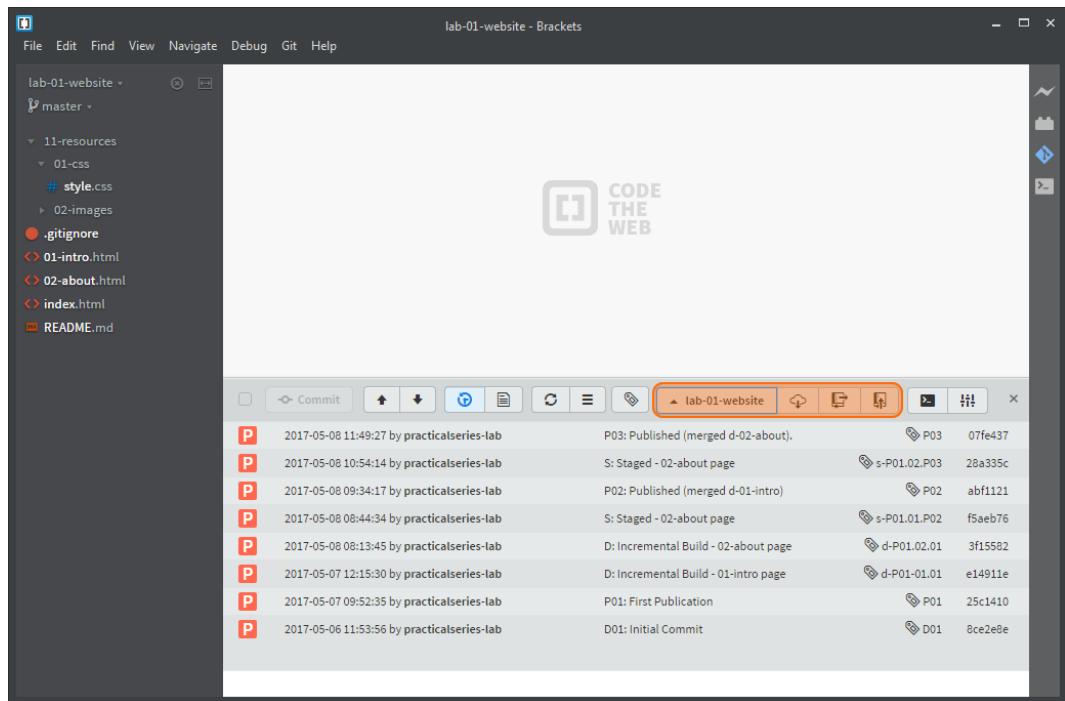


Figure 8.15 Brackets—remote repository buttons

8.3.2 Pushing the local repository to the remote repository

At this point we've set up the link to the remote repository, but nothing else has happened we haven't done anything with the remote repository; we haven't even connected to it yet. We've just given the connection details to Brackets.

The next thing to do is send (*push*) the local repository up to the remote.

Make sure you are on the master branch and click the **GIT PUSH** button on the Git pane: .

This opens the push to remote dialogue box:

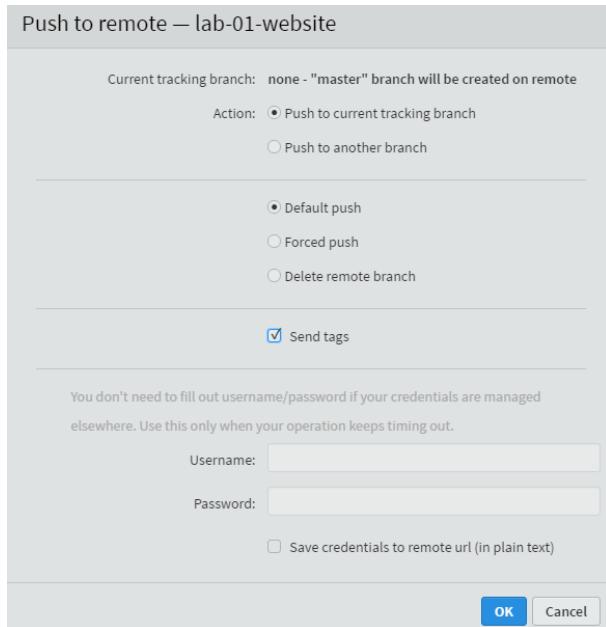


Figure 8.16 Brackets—push to remote dialogue box

We've seen this screen before in section 5.4.4 with the [lab-brackets-git](#) repository. Let's look at it in a bit more detail.

The first line **CURRENT TRACKING BRANCH** tells us it will create a new **master** branch on the remote. The **ACTION** is to push to this branch (*this is what we want; it would be confusing to push the local master branch to a different remote branch*). We also want to do a **DEFAULT PUSH**. There is no reason to do anything else.

The only thing we need to change is the **SEND TAGS** tick box, tick it; we want to send our tags to the remote repository.

There is no username or password to enter, we are using SSH as the link protocol and that does not need a username and password (see section 4.3).

Click **OK**.

A couple of things happen next; first you (briefly) get a screen showing the progress of the operation:

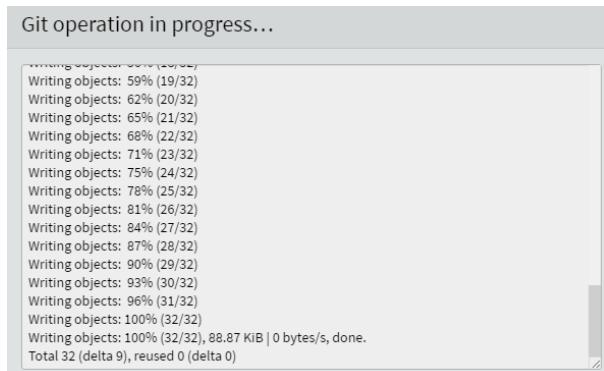


Figure 8.17 Brackets—push to remote dialogue box, operation in progress

And after this the response screen:

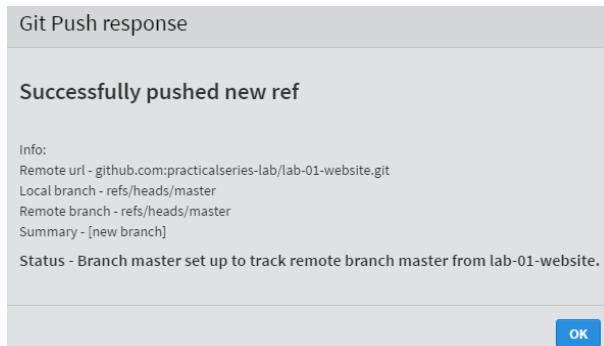


Figure 8.18 Brackets—push to remote dialogue box, result

Well it looks ok, it says “successfully” so that’s good. It also says the **LOCAL BRANCH MASTER IS TRACKING THE REMOTE BRANCH MASTER**. This just means it will report if there are changes to the remote **master** branch that have not been incorporated into our local repository.

Everything seems ok so close the dialogue box.

Now go back to GitHub and navigate to the [lab-01-website](#) repository (if you already have it open, refresh the page by pressing **F5**). It looks a bit different.

Mine looks like Figure 8.19:

The screenshot shows a GitHub repository page. At the top, there's a navigation bar with links for 'This repository', 'Search', 'Pull requests', 'Issues', and 'Gist'. On the right side of the top bar, there are buttons for 'Watch' (0), 'Star' (0), 'Fork' (0), and a profile icon labeled 'PS'. Below the top bar, the repository name 'practicalseries-lab / lab-01-website' is displayed, along with a 'Code' tab and other tabs for 'Issues' (0), 'Pull requests' (0), 'Projects' (0), 'Wiki', 'Pulse', 'Graphs', and 'Settings'. A message below the tabs says 'A simple website repository used as an example to demonstrate how the Brackets-Git extension works'. There are buttons for 'Edit' and 'Add topics'. Below this, a summary bar shows '8 commits', '1 branch', '9 releases', and '1 contributor'. A dropdown menu for 'Branch: master' and a 'New pull request' button are also present. The main content area displays a list of commits, showing files like '11-resources', '.gitignore', '01-intro.html', '02-about.html', 'README.md', and 'index.html' with their respective commit messages and dates. Below the commit list is a section titled 'A PracticalSeries Publication' containing a logo for 'THE PRACTICAL SERIES LAB'.

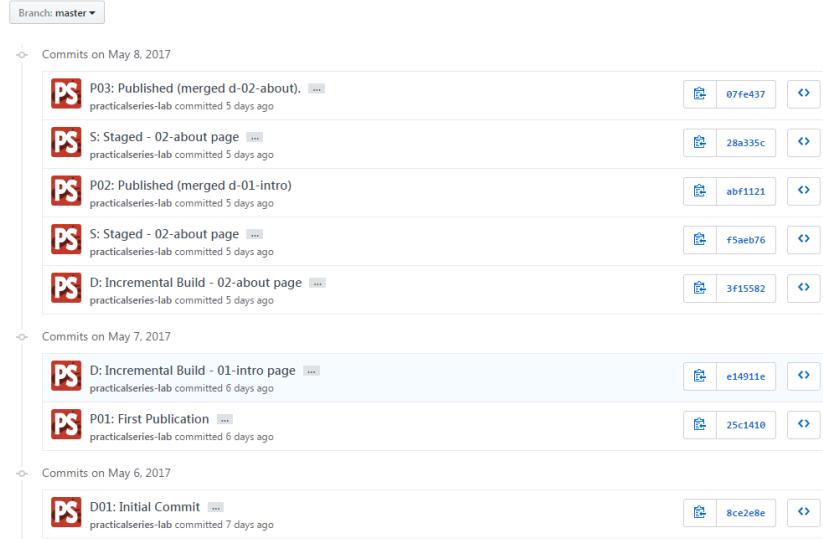
Figure 8.19 GitHub—lab-01-website after push

I don't want to run through too many of the GitHub features here, I do that in section 9 and 10; but there are somethings we need to look at. They're all in the tab bar:



Figure 8.20 GitHub—tab bar

Starting on the left at the top. It says we have 8 commits. This is because we have only pushed the **master** branch and that only had eight commits on it (Figure 8.15). If you click the commits tab  it lists all the commits on the branch:



Date	Commit Message	Author	Hash
May 8, 2017	P03: Published (merged d-02-about).	practicalseries-lab committed 5 days ago	07fe437
May 8, 2017	S: Staged - 02-about page	practicalseries-lab committed 5 days ago	28a335c
May 8, 2017	P02: Published (merged d-01-intro)	practicalseries-lab committed 5 days ago	abf1121
May 8, 2017	S: Staged - 02-about page	practicalseries-lab committed 5 days ago	f5aeb76
May 8, 2017	D: Incremental Build - 02-about page	practicalseries-lab committed 5 days ago	3f15582
May 7, 2017	D: Incremental Build - 01-intro page	practicalseries-lab committed 6 days ago	e14911e
May 7, 2017	P01: First Publication	practicalseries-lab committed 6 days ago	25c1410
May 6, 2017	D01: Initial Commit	practicalseries-lab committed 7 days ago	8ce2e8e

Figure 8.21 GitHub—commits tab

These match the commits on the **master** branch workflow (exactly the same hash numbers).

Go back to the main repository screen (click the [/ LAB-01-WEBSITE](#) at the top of the page under the black bar with the Octocat icon).

Looking at the branches tab  it says there is one branch, this is the **master** branch. The local repository has two branches, but the remote repository only has one. *Why?*

The answer is that when we do a push from Brackets (or indeed from the Git command line) it only pushes a single branch—if no specific branch is given, it pushes the currently active branch. In our case this was the **master** branch; and that is why there is only one branch in the remote repository, we haven't pushed the second branch yet.

Click the branch tab, it just shows just the single **master** branch.

The screenshot shows the GitHub branches tab interface. At the top, there are tabs for Overview, Yours, Active, Stale, and All branches. The All branches tab is selected. A search bar labeled "Search branches..." is present. Below the tabs, a section titled "Default branch" shows a single entry: "master Updated 5 days ago by practicalseries-lab". To the right of this entry are buttons for "Default" and "Change default branch".

Figure 8.22 GitHub—branches tab

Next is the releases tab , this lists the tags we passed to the remote repository:

The screenshot shows the GitHub releases tab interface. At the top, there are tabs for Releases and Tags. The Releases tab is selected. A button for "Draft a new release" is visible. Below the tabs, a list of releases is shown, each with a timestamp, a tag name, and download links for zip and tar.gz formats. The releases listed are:

Published	Tag	Commit	Downloads
5 days ago	P03	07fe437	zip tar.gz
5 days ago	s-P01.02.P03	28a335c	zip tar.gz
5 days ago	P02	abf1121	zip tar.gz
5 days ago	s-P01.01.P02	f5aeb76	zip tar.gz
5 days ago	d-P01.02.01	3f15582	zip tar.gz
6 days ago	d-P01-01.01	e14911e	zip tar.gz
6 days ago	P01	25c1410	zip tar.gz
7 days ago	D01	8ce2e8e	zip tar.gz

Figure 8.23 GitHub—releases tab

These are all the tags that were entered as we created the local repository.

Note: There is a slight discrepancy between what Brackets (and Git) call a tag and what GitHub calls a tag (you can see our tags are classed as both tags and releases in GitHub, click where it says TAGS next to the blue RELEASES button), you will see the same list. Bear with it for now, I explain it more fully in § 9.6.

Finally, the contributor tab . There is only one contributor, me:

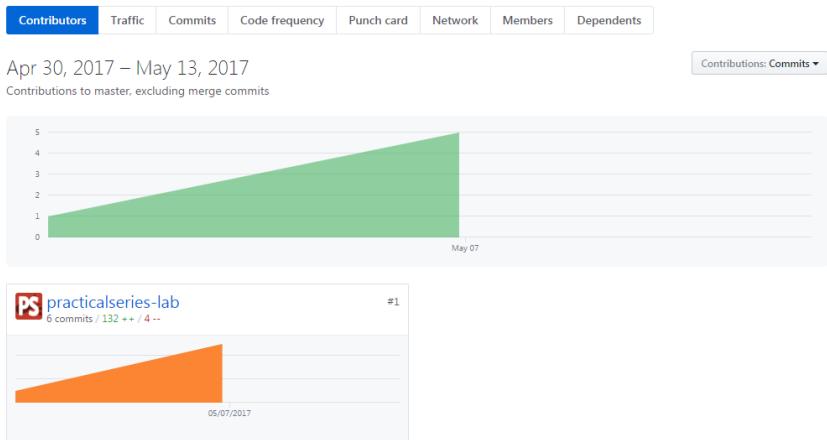


Figure 8.24 GitHub—contributor tab

The contributor page has a lot of statistics and information in it—most of it is self-explanatory and we don't need to go through it here.

There is one final thing, go back to the repository home page and look at the coloured bar under the tabs, Figure 8.20. It's red and purple. Click it:



Figure 8.25 GitHub—code breakdown

It shows a breakdown of the code within the repository 66% HTML, 34% CSS. HTML is shown in red; CSS in purple—other languages have other colours. Click the bar again to get back to the tabs.

8.3.3 Pushing another branch to the remote

Back to Brackets. Let's push the **d-03-contact** branch up to the remote repository.

First switch to **d-03-contact**, click the small arrow next to the **master** branch and select **d-03-contact** from the dropdown. It should look something like this:

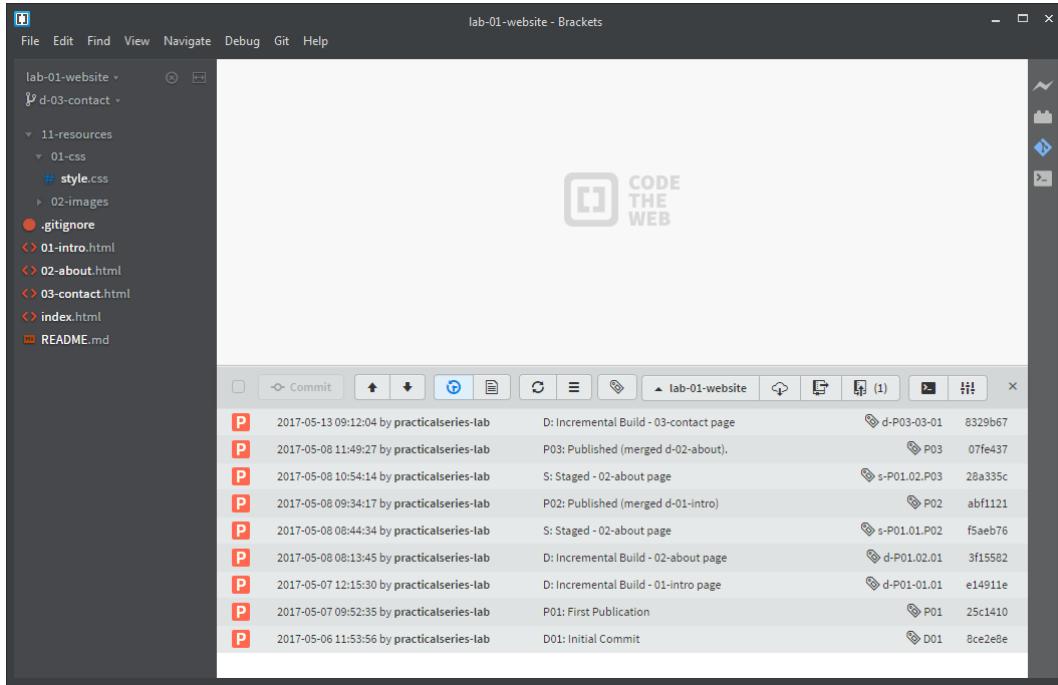


Figure 8.26 Brackets—switch to the **d-03-contact** branch

This time we have a number alongside the **GIT PUSH** button , this number means that the current branch **d-03-contact** has one extra commit that needs to be pushed to the remote repository; indeed there are nine commits listed in the commit history (Figure 8.26) and we know that the remote repository only has eight commits (Figure 8.21).

Click the **GIT PUSH** button:

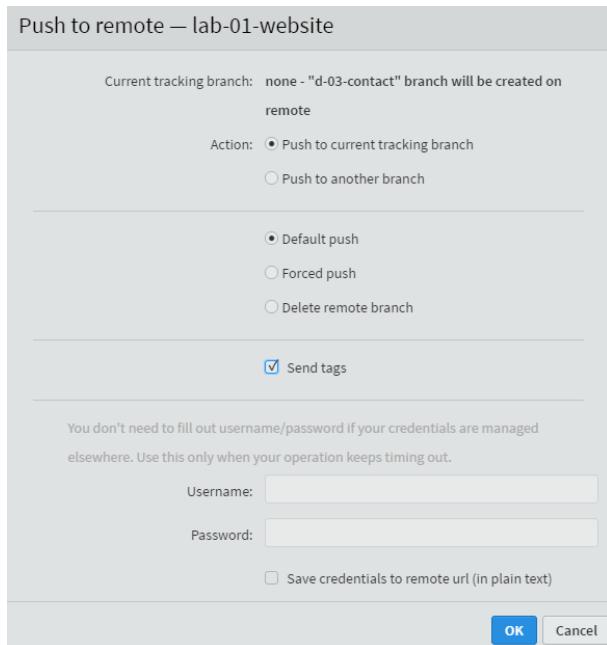


Figure 8.27 Brackets—push the **d-03-contact** branch

In the push to remote dialogue box make sure the **SEND TAGS** option is ticked and click **OK**.

Again we get the successful response screen:

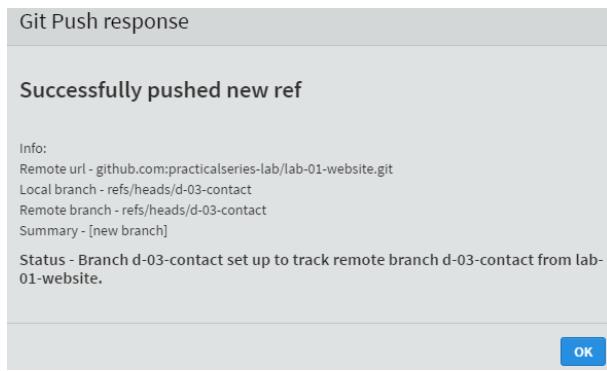


Figure 8.28 Brackets—push response

This time we are told the branch **d-03-contact** is now being tracked.

Let's go look at in GitHub. You should have something like this:

The screenshot shows a GitHub repository page for 'practicalseries-lab / lab-01-website'. At the top, there are buttons for 'This repository' and 'Search', and tabs for 'Pull requests', 'Issues', and 'Gist'. On the right, there are 'Watch' (0), 'Star' (0), 'Fork' (0) buttons, and a 'PS' icon. Below the header, there are navigation links for 'Code', 'Issues (0)', 'Pull requests (0)', 'Projects (0)', 'Wiki', 'Pulse', 'Graphs', and 'Settings'. A summary bar shows '8 commits', '2 branches', '9 releases', and '1 contributor'. Under 'Your recently pushed branches:', the 'd-03-contact' branch is highlighted in yellow, indicating it was just pushed. A 'Compare & pull request' button is next to it. Below this, a list of commits shows the history of the repository, with the most recent commit being 'Published (merged d-02-about)' by 'practicalseries-lab' on '5 days ago'. The 'master' branch is also listed in the commit history.

Figure 8.29 GitHub—after the addition of a second branch

It's telling us that the new branch **d-03-contact** has been added (*pushed*), that's the bit in yellow. We can also see that there are two branches.

It's only showing eight commits though—this is because we are still on the **master** branch. You can see the current branch; it's listed in the **BRANCH** button: `Branch: master`.

To change the active branch, click the **BRANCH** button:

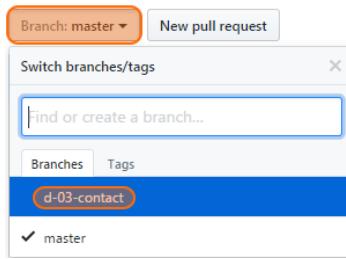


Figure 8.30 GitHub—switch branches

Select the branch you want from the list, **d-03-contact** in this case.

This gives us:

The screenshot shows a GitHub repository page for 'practicalseries-lab / lab-01-website'. The repository has 9 commits, 2 branches, 9 releases, and 1 contributor. A yellow box highlights the 'd-03-contact' branch, which is 14 minutes old. The commit list shows several files being published, including '11-resources', '.gitignore', '01-intro.html', '02-about.html', '03-contact.html', 'README.md', and 'index.html'. The latest commit, '8329b67', is from 7 hours ago. A green button at the bottom right says 'Clone & download'.

Figure 8.31 GitHub—on the `d-03-contact` branch

The `03-contact.html` file is there and the latest commit `[8329b67]` is also shown (in the pale blue line about halfway down) so it looks like it's worked.

8.4

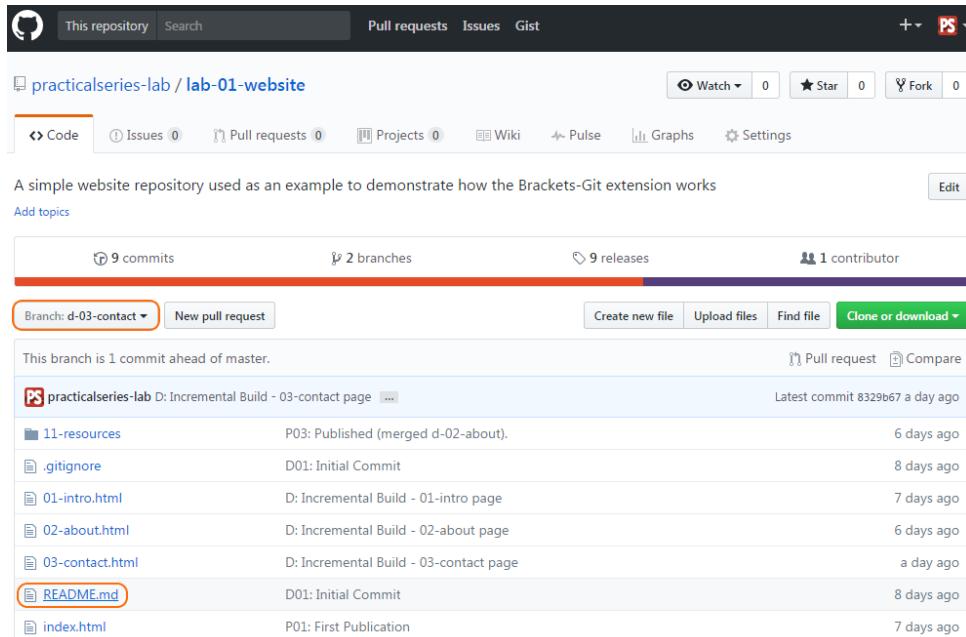
Pulling remote changes into the local repository

This is probably the most common situation that arises when working with remote repositories (*not so much in my case, the Practical Series website has so far been a solo effort—I don't always play well with others—I usually do things my own way and I'm too old to change now—most people get used to me in the end, back me or sack me that's my motto*).

Where the repository is a team effort, the remote repository quite often changes and moves ahead of your local copy. The remote repository should always be considered the master repository.

To illustrate the point, let's make a change to the **d-03-contact** branch on the remote repository and see how we get this back into our local repository using Brackets.

To GitHub then. Open the **lab-01-website** repository in GitHub and switch to the **d-03-contact** branch. It should look like this:



A screenshot of a GitHub repository page. The repository name is `practicalseries-lab / lab-01-website`. The `d-03-contact` branch is selected. Key statistics shown are 9 commits, 2 branches, 9 releases, and 1 contributor. The README.md file is highlighted with an orange border. The page also includes links for creating new files, uploading files, finding files, and cloning or downloading the repository.

Figure 8.32 GitHub—README.md on the **d-03-contact** branch

Make sure **d-03-contact** is selected in the **BRANCH** button. Now click the **README.md** file (this is the file we are going to change); it's highlighted in Figure 8.32. This will show a preview of the **README.md** file:

The screenshot shows a GitHub repository page for 'practicalseries-lab / lab-01-website'. The 'Code' tab is selected. In the center, there is a preview of the 'README.md' file with the content '# A PracticalSeries Publication'. At the bottom right of the preview area, there is a pencil icon, which is highlighted with a red box.

Figure 8.33 GitHub—README.md preview page

To edit the file click the pencil symbol (highlighted). This now puts us on the edit page (Figure 8.34):

The screenshot shows the GitHub edit page for the 'README.md' file in the 'lab-01-website' repository. The 'Edit file' tab is selected. The code editor displays the following Markdown content:

```

1 # A PracticalSeries Publication
2
3 <p align="center">
4   
5 </p>
6
7 The **Practical Series of publications** is a website resource for web developers and engineers. It contains a number of online
publications designed to help and explain how to build a website, how to use version control and how to write engineering software for
control systems.
8 This particular repository is designed as an example project to demonstrate how to build a Git and GitHub repository using the Brackets-
Git extension for the Brackets text editor.
9 The full set of PracticalSeries publications is available at practicalseries.com(http://practicalseries.com "Practical Series Website").
10
11 ## How to use this repository
12 This repository is a worked example demonstrating how to build a version control project using Git and GitHub from within the Brackets
text editor.
13 This repository is intended to be used with the accompanying documentation practicalseries Git and GitHub
(http://practicalseries.com/0021-git-vcs/index.html "Practical Series - Git and GitHub").
14
15 ## Contributors
16 This repository was constructed by Michael Gledhill(https://github.com/mgledhill "Michael Gledhill").
17
18 ## Licence
19 This is simply a demonstration repository, the contents are free to use by anyone who wishes to do so.

```

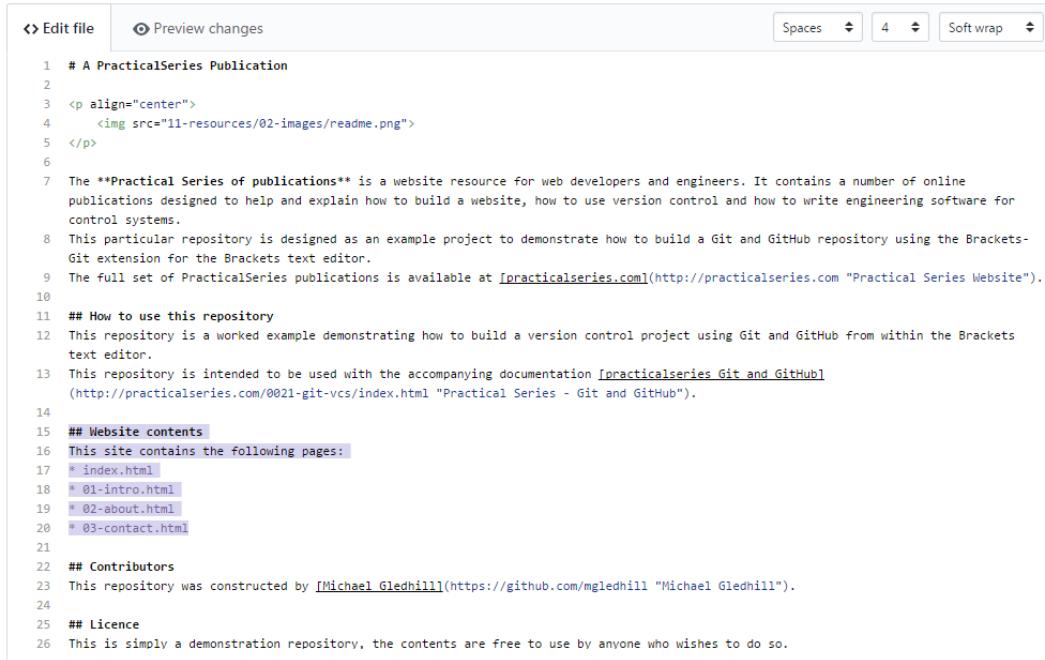
Figure 8.34 GitHub—edit README.md

Let's add some more text to the file, add the following:

```
14
15 ## Website contents
16 This site contains the following pages:
17 * index.html
18 * 01-intro.html
19 * 02-about.html
20 * 03-contact.html
```

Code 8.2 README.md modification

It looks like this on GitHub:



The screenshot shows a GitHub repository's README.md file in edit mode. The 'Preview changes' tab is selected. The code block contains several sections of text, with the first section from line 14 onwards highlighted in blue, indicating it is new content being added to the file.

```
1 # A PracticalSeries Publication
2
3 <p align="center">
4   
5 </p>
6
7 The **Practical Series of publications** is a website resource for web developers and engineers. It contains a number of online
publications designed to help and explain how to build a website, how to use version control and how to write engineering software for
control systems.
8 This particular repository is designed as an example project to demonstrate how to build a Git and GitHub repository using the Brackets-
Git extension for the Brackets text editor.
9 The full set of PracticalSeries publications is available at practicalseries.com(http://practicalseries.com "Practical Series Website").
10
11 ## How to use this repository
12 This repository is a worked example demonstrating how to build a version control project using Git and GitHub from within the Brackets
text editor.
13 This repository is intended to be used with the accompanying documentation practicalseries\_Git\_and\_Github
(http://practicalseries.com/0021-git-vcs/index.html "Practical Series - Git and GitHub").
14
15 ## Website contents
16 This site contains the following pages:
17 * index.html
18 * 01-intro.html
19 * 02-about.html
20 * 03-contact.html
21
22 ## Contributors
23 This repository was constructed by Michael Gledhill(https://github.com/mgledhill "Michael Gledhill").
24
25 ## Licence
26 This is simply a demonstration repository, the contents are free to use by anyone who wishes to do so.
```

Figure 8.35 GitHub—README.md modifications (highlighted)

If you click the PREVIEW CHANGES tab, it will show how the page looks:

A PracticalSeries Publication



The Practical Series of publications is a website resource for web developers and engineers. It contains a number of online publications designed to help and explain how to build a website, how to use version control and how to write engineering software for control systems. This particular repository is designed as an example project to demonstrate how to build a Git and GitHub repository using the Brackets-Git extension for the Brackets text editor. The full set of PracticalSeries publications is available at practicalseries.com.

How to use this repository

This repository is a worked example demonstrating how to build a version control project using Git and GitHub from within the Brackets text editor. This repository is intended to be used with the accompanying documentation [practicalseries Git](#) and [GitHub](#).

Website contents

This site contains the following pages:

- index.html
- 01-intro.html
- 02-about.html
- 03-contact.html

Contributors

This repository was constructed by [Michael Gledhill](#).

Licence

This is simply a demonstration repository, the contents are free to use by anyone who wishes to do so.

Figure 8.36 GitHub—README.md modifications preview

To commit the change we need to complete the boxes at the bottom by adding the commit message:

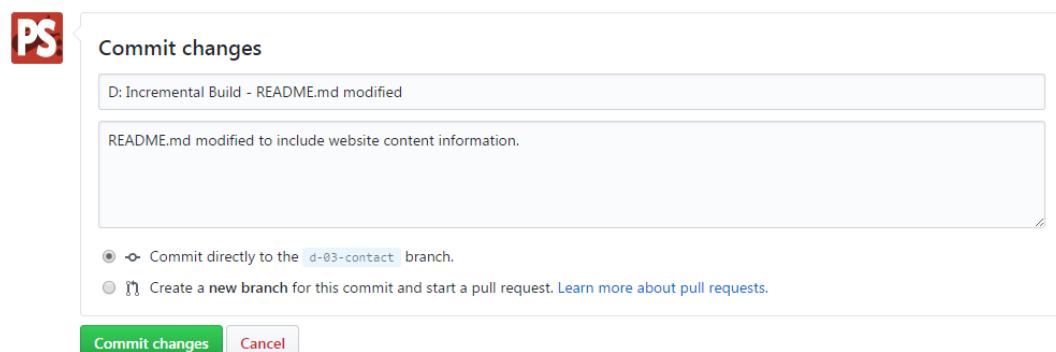


Figure 8.37 GitHub—README.md commit message

Enter the commit message as shown above and click the green **COMMIT CHANGES** button.

Now go back to the repository home page by clicking the `LAB-01-WEBSITE` link next to the branch button (highlighted below):

Branch: d-03-contact **lab-01-website** / README.md

Figure 8.38 GitHub—go to repository home page

The home page now looks like this (make sure the `d-03-contact` branch is selected):

This repository

Search

Pull requests Issues Gist

Watch 0 Star 0 Fork 0

practicalseries-lab / lab-01-website

A simple website repository used as an example to demonstrate how the Brackets-Git extension works

10 commits 2 branches 9 releases 1 contributor

Your recently pushed branches:

d-03-contact (3 minutes ago)

Create new file Upload files Find file Clone or download

Branch: d-03-contact New pull request

This branch is 2 commits ahead of master.

4da998f D: Incremental Build - README.md modified (5 minutes ago)

11-resources P03: Published (merged d-02-about). 6 days ago

.gitignore D01: Initial Commit 8 days ago

01-intro.html D: Incremental Build - 01-intro page 7 days ago

02-about.html D: Incremental Build - 02-about page 6 days ago

03-contact.html D: Incremental Build - 03-contact page a day ago

README.md D: Incremental Build - README.md modified 5 minutes ago

index.html P01: First Publication 7 days ago

Figure 8.39 GitHub—new commit on `d-03-contact` branch

There is a new commit, it is `[4da998f]` and the workflow is:

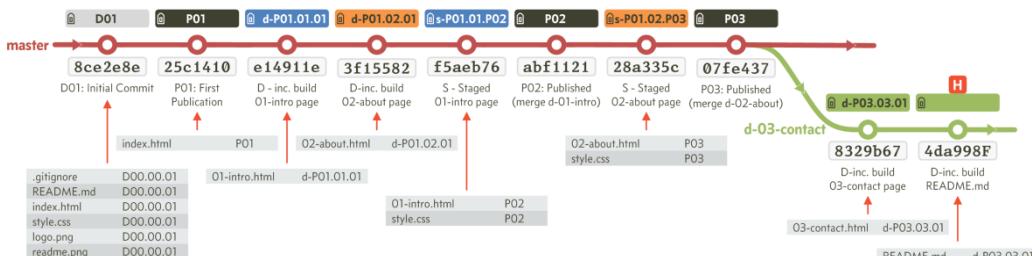


Figure 8.40 Remote repository workflow

To summarise, the remote repository now has an extra commit on the `d-03-contact` branch:

Back to Brackets. My local repository looks like this in Brackets (I'm on the **d-03-contact** branch):

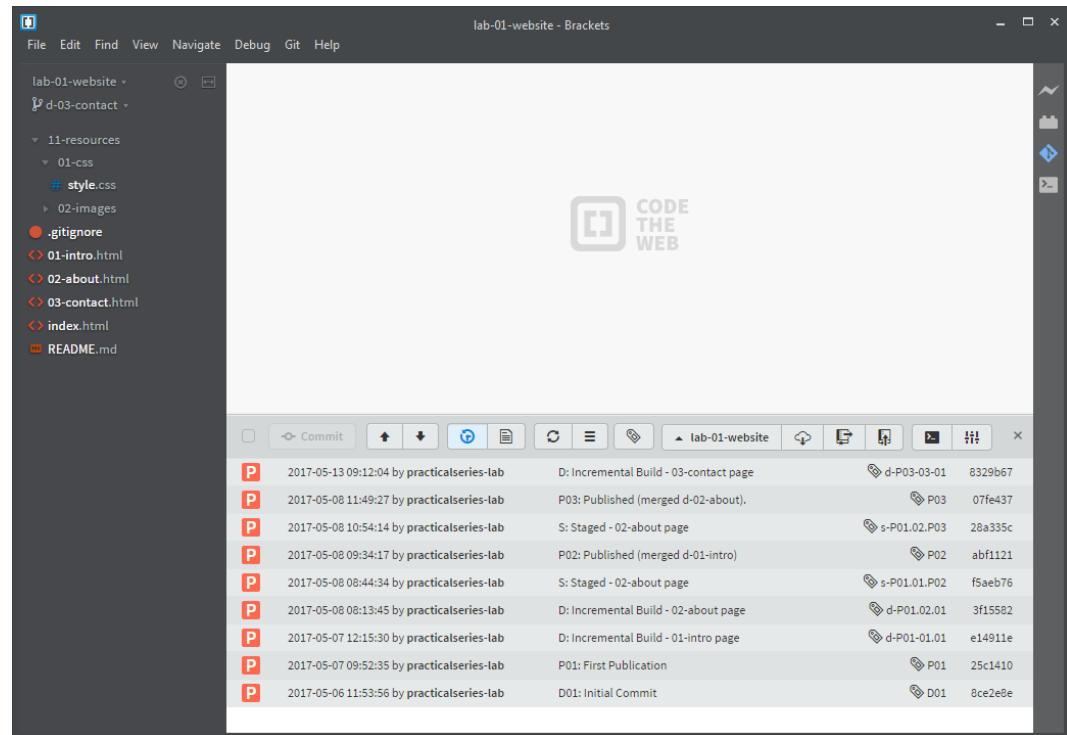


Figure 8.41 Brackets—**d-03-contact** commit history

In Brackets, there is nothing to indicate that it knows of any changes to the remote repository.

To refresh Brackets with data from the remote repository click the **FETCH** button .

This will open (briefly) the fetch in progress screen:

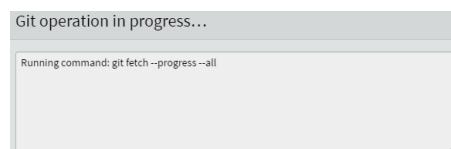


Figure 8.42 Brackets—Fetch in progress

This refreshes the Git pane counters:

P	2017-05-13 09:12:04 by practicalseries-lab	D: Incremental Build - 03-contact page	d-P03-03-01	8329b67
P	2017-05-08 11:49:27 by practicalseries-lab	P03: Published (merged d-02-about).	P03	07fe437
P	2017-05-08 10:54:14 by practicalseries-lab	S: Staged - 02-about page	s-P01.02.P03	28a335c
P	2017-05-08 09:34:17 by practicalseries-lab	P02: Published (merged d-01-intro)	P02	abf1121
P	2017-05-08 08:44:34 by practicalseries-lab	S: Staged - 02-about page	s-P01.01.P02	f5aeb76
P	2017-05-08 08:13:45 by practicalseries-lab	D: Incremental Build - 02-about page	d-P01.02.01	3f15582
P	2017-05-07 12:15:30 by practicalseries-lab	D: Incremental Build - 01-intro page	d-P01-01.01	e14911e
P	2017-05-07 09:52:35 by practicalseries-lab	P01: First Publication	P01	25c1410
P	2017-05-06 11:53:56 by practicalseries-lab	D01: Initial Commit	D01	8ce2e8e

Figure 8.43 Bracket—updated counters

At this point, nothing has actually changed; the changes made to `README.md` in the remote repository are still not present on the local repository. All we've done is ask Brackets to check the status of the remote repository.

Brackets has detected that the remote repository has an additional commit, hence the number 1 in the **GIT PULL** button:

Let's copy (*pull* in Git terminology) the remote repository into the local repository. Click the **GIT PULL** button: . This opens the pull dialogue box:

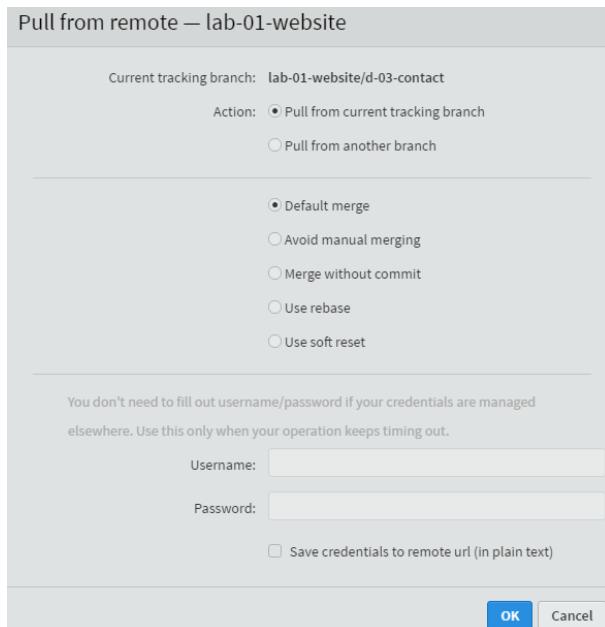


Figure 8.44 Bracket—pull dialogue box

There is nothing to change here, click **OK**. This give the response screen.

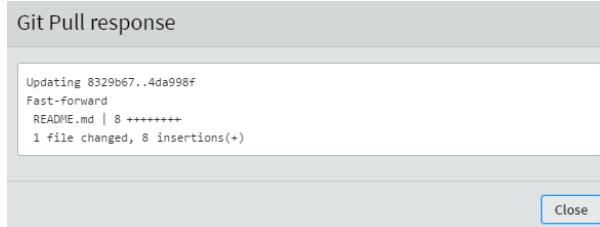


Figure 8.45 Brackets—pull response dialogue box

Click **CLOSE**, and look at the commit history, in my case I have Figure 8.46:

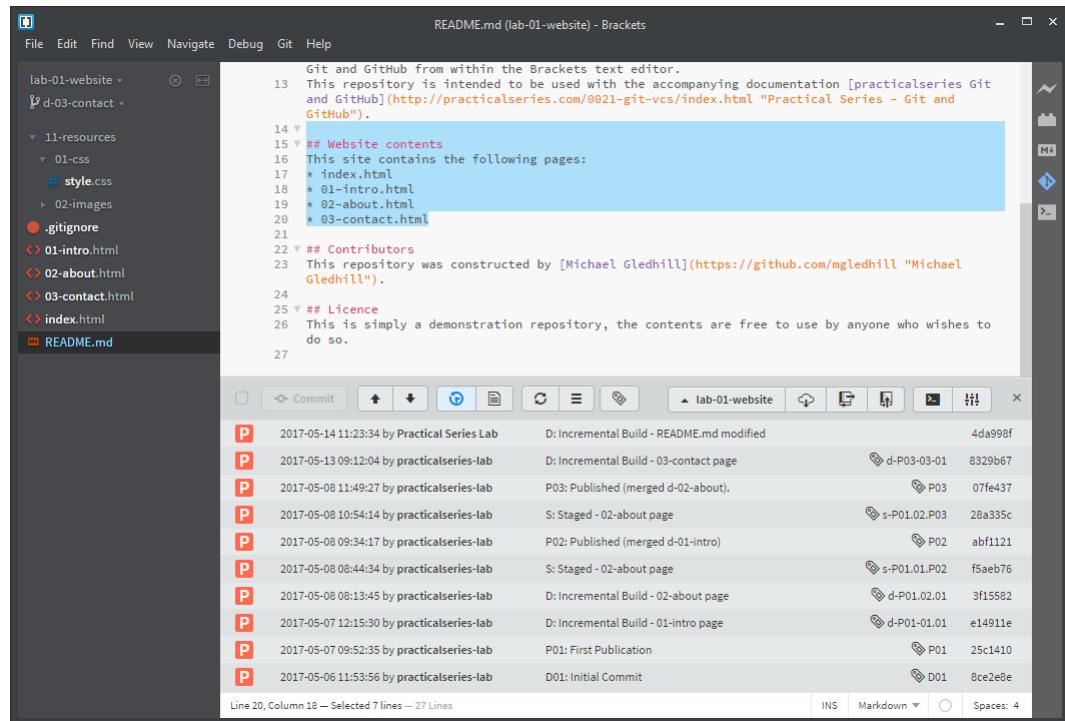


Figure 8.46 Brackets—after the pull

The new commit from the remote repository is visible [4da998f], and the changes made to `README.md` are visible in the local working copy.

The only thing that is missing is a tag for the new commit. This is because I didn't assign one in GitHub. I cover this in section 9.6.

In Brackets, give the new commit point [4da998f] the tag `d-P03.03.02`.

Do a git push  and in the dialogue box tick the **SEND TAGS** box (just like Figure 8.27).

The report will say that there were no changes to be made, and this is true (*we haven't changed the code or added a commit point*); however it will push the new tag and if you look at that branch in GitHub it will have 10 releases.

8.5

Conflicts between local and remote repositories

This is a re-enactment of what we did in § 6.7.3 where we made conflicting changes to the same line of the `style.css` file. In that case it was merging separate branches; here it will be merging changes between the remote and local repositories.

The `style.css` file currently is (in both the remote and local repositories):

```
STYLE.CSS
```

```
24 h1, h2, h3, h4, h5, h6 {           /* set standard headings */
25   font-family: sans-serif;
26   font-weight: normal;
27   font-size: 3rem;
28   padding: 2rem 5rem 2rem 5rem;
29 }
30 h3 { font-size: 2.5rem; }
31
32 .cover-faq {                         /* holder for cover image */
33   width: 50%;
34   margin: 2rem auto;
35   padding: 0;
36 }
```

Code 8.3 style.css **d-03-contact** local repository

Firstly let's modify the file in the local repository.

Make sure you are on the **d-03-contact** branch, open `style.css` and add a colour declaration to the `h3` element (this is exactly the same change we made in § 6.7.3):

```
30 h3 { font-size: 2.5rem; color: #c0504d; }
```

The `h3` colour is changed to red. Save the change and commit it with the commit message:

D: Incremental build - 03-contact page

Style.css modified <h3> colour changed to red.

My commit point is [f52a6a5]. Tag the commit with the label d-P03.03.03.

The Git pane is showing one change that needs to be pushed to the remote:

P	Date	Message	Status	Hash
P	2017-05-14 14:43:05	by practicalseries-lab	D: Incremental build - 03-contact page	d-P03.03.03 f52a6a5
P	2017-05-14 11:23:34	by Practical Series Lab	D: Incremental Build - README.md modified	d-P03.03.02 4da998f
P	2017-05-13 09:12:04	by practicalseries-lab	D: Incremental Build - 03-contact page	d-P03-03-01 8329b67
P	2017-05-08 11:49:27	by practicalseries-lab	P03: Published (merged d-02-about).	P03 07fe437
P	2017-05-08 10:54:14	by practicalseries-lab	S: Staged - 02-about page	s-P01.02.P03 28a335c
P	2017-05-08 09:34:17	by practicalseries-lab	P02: Published (merged d-01-intro)	P02 abf1121
P	2017-05-08 08:44:34	by practicalseries-lab	S: Staged - 02-about page	s-P01.01.P02 f5aeb76
P	2017-05-08 08:13:45	by practicalseries-lab	D: Incremental Build - 02-about page	d-P01.02.01 3f15582
P	2017-05-07 12:15:30	by practicalseries-lab	D: Incremental Build - 01-intro page	d-P01-01.01 e14911e
P	2017-05-07 09:52:35	by practicalseries-lab	P01: First Publication	P01 25c1410
P	2017-05-06 11:53:56	by practicalseries-lab	D01: Initial Commit	D01 8ce2e8e

Figure 8.47 Bracket—style.css change in local repository

Now let's do the same in GitHub. Open the lab-01-website repository, select the d-03-contact branch and click the 11-resources folder to open it, then click 01-css and finally style.css. This will open a preview of the file:

```
practicalseries-lab P03: Published (merged d-02-about).
07fe437 6 days ago
1 contributor
44 lines (38 sloc) | 1.03 KB
Raw Blame History ⚙️ 🗑️

1   * {
2     margin: 0;
3     padding: 0;
4     box-sizing: border-box;
5     position: relative;
6   }
7
8   html {
9     background-color: #fbfaf6; /* Set cream coloured page background */
10    color: #404030;
11    font-family: serif;
12    font-size: 26px;
13    text-rendering: optimizeLegibility;
14  }
15
16  body {
17    max-width: 1276px;
18    margin: 0 auto;
19    background-color: #fff; /* make content area background white */
20    border-left: 1px solid #eddede;
21    border-right: 1px solid #eddede;
22  }
23
24  h1, h2, h3, h4, h5, h6 { /* set standard headings */
25    font-family: sans-serif;
26    font-weight: normal;
27    font-size: 3rem;
28    padding: 2rem 5rem 2rem 5rem;
29  }
30  h3 { font-size: 2.5rem; }
31
```

Figure 8.48 GitHub—style.css preview in remote repository

Click the pencil to edit the file:

Change line 30 to make it blue:

```
30  h3 { font-size: 2.5rem; color: #4F81BD; }
```

And add this commit message:

D: Incremental build - 03-contact page

Style.css modified <h3> colour changed to blue.

It looks like this:

The screenshot shows the GitHub interface for editing a file named 'style.css' in a repository. The code editor displays the following CSS:

```
6 }
7
8 html {
9     background-color: #fbfa#6;      /* Set cream coloured page background */
10    color: #404030;
11    font-family: serif;
12    font-size: 26px;
13    text-rendering: optimizeLegibility;
14 }
15
16 body {
17     max-width: 1276px;
18     margin: 0 auto;
19     background-color: #fff;        /* make content area background white */
20     border-left: 1px solid #e0e0e0;
21     border-right: 1px solid #e0e0e0;
22 }
23
24 h1, h2, h3, h4, h5, h6 {          /* set standard headings */
25     font-family: sans-serif;
26     font-weight: normal;
27     font-size: 3rem;
28     padding: 2rem 5rem 2rem 5rem;
29 }
30 h3 { font-size: 2.5rem; color: #4F81BD; } // Line 30, highlighted in yellow
31
32 .cover-fig {                      /* holder for cover image */
33     width: 50%;
34     margin: 2rem auto;
35     padding: 0;
36 }
37 .cover-fig img {width: 100%;}       /* format cover image */
38
39 p {                                /* TEXT STYLE - paragraph */
40     margin-bottom: 1.2rem;           /* *** THIS SETS PARAGRAPH SPACING *** */
41     padding: 0 5rem;
42     line-height: 135%;
43 }
```

Below the code editor is a 'Commit changes' dialog box. It contains the commit message 'D: Incremental build - 03-contact page' and the detailed message 'Style.css modified <h3> colour changed to blue.' There are two radio button options at the bottom: one for committing directly to the branch and another for creating a new branch. At the very bottom are 'Commit changes' and 'Cancel' buttons.

Figure 8.49 GitHub—style.css edit in remote repository

Click the **COMMIT CHANGES** button. On the repository home page the new commit is [\[80fe002\]](#).

The workflows for the local and remote repositories are now (I'm only showing the **d-03-contact** branch):

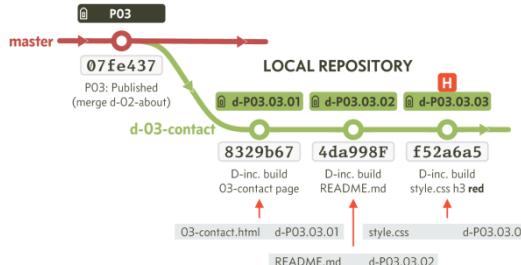


Figure 8.50 Local repository workflow

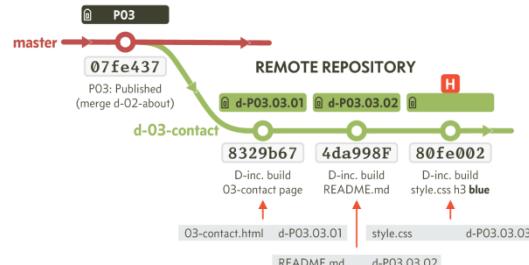


Figure 8.51 Remote repository workflow

Let's refresh the counters in Brackets, click the **FETCH** button

It gives this:

P	2017-05-14 14:43:05 by practicalseries-lab	D: Incremental build - 03-contact page		d-P03.03.03	f52a6a5
P	2017-05-14 11:23:34 by Practical Series Lab	D: Incremental Build - README.md modified		d-P03.03.02	4da998f
P	2017-05-13 09:12:04 by practicalseries-lab	D: Incremental Build - 03-contact page		d-P03.03.01	8329b67
P	2017-05-08 11:49:27 by practicalseries-lab	P03: Published (merged d-02-about).			P03 07fe437
P	2017-05-08 10:54:14 by practicalseries-lab	S: Staged - 02-about page		s-P01.02.P03	28a335c
P	2017-05-08 09:34:17 by practicalseries-lab	P02: Published (merged d-01-intro)		P02	abf1121
P	2017-05-08 08:44:34 by practicalseries-lab	S: Staged - 02-about page		s-P01.01.P02	f5aeb76
P	2017-05-08 08:13:45 by practicalseries-lab	D: Incremental Build - 02-about page		d-P01.02.01	3f15582
P	2017-05-07 12:15:30 by practicalseries-lab	D: Incremental Build - 01-intro page		d-P01.01.01	e14911e
P	2017-05-07 09:52:35 by practicalseries-lab	P01: First Publication		P01	25c1410
P	2017-05-06 11:53:56 by practicalseries-lab	D01: Initial Commit		D01	8ce2e8e

Figure 8.52 Brackets—Refresh counters

Not surprisingly we are showing one change to pull and one to push.

So what do we do? Do we push or do we pull?

Well, the remote repository is the master repository so we pull its information into our repository and merge it together before we push it back with our changes.

The rule is:

**ALWAYS PULL CHANGES FROM THE REMOTE
BEFORE PUSHING CHANGES BACK TO IT**

So let's do that. Click the **GIT PULL** button . This opens the pull dialogue box:

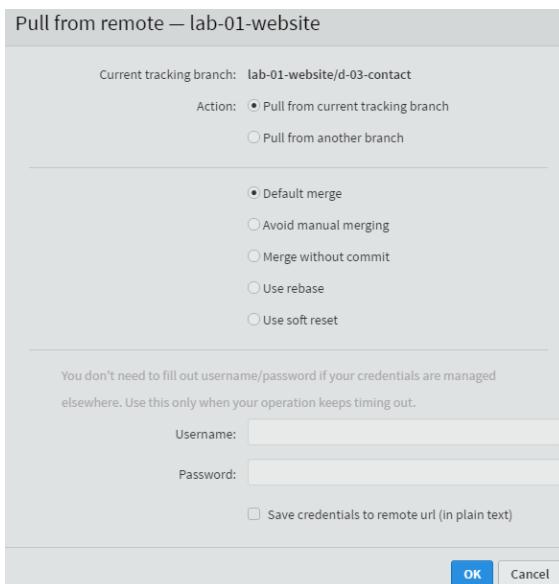


Figure 8.53 Brackets—pull dialogue box

Again leave all the defaults and click **OK**. This time the response is a conflict:

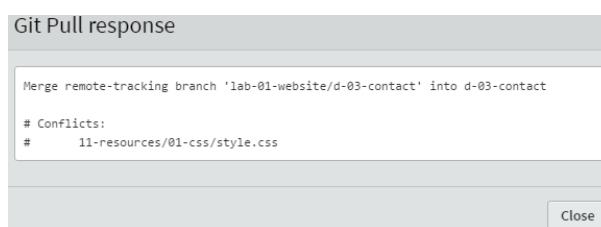


Figure 8.54 Brackets—pull response dialogue box

It tells us that we have a conflict in `style.css` (as we would expect).

Click **CLOSE**, and Brackets will have opened a merged version of `style.css` in the editor showing the conflict. This process is now identical to resolving the merge conflict we had in § 6.7.3 and Figure 6.77:

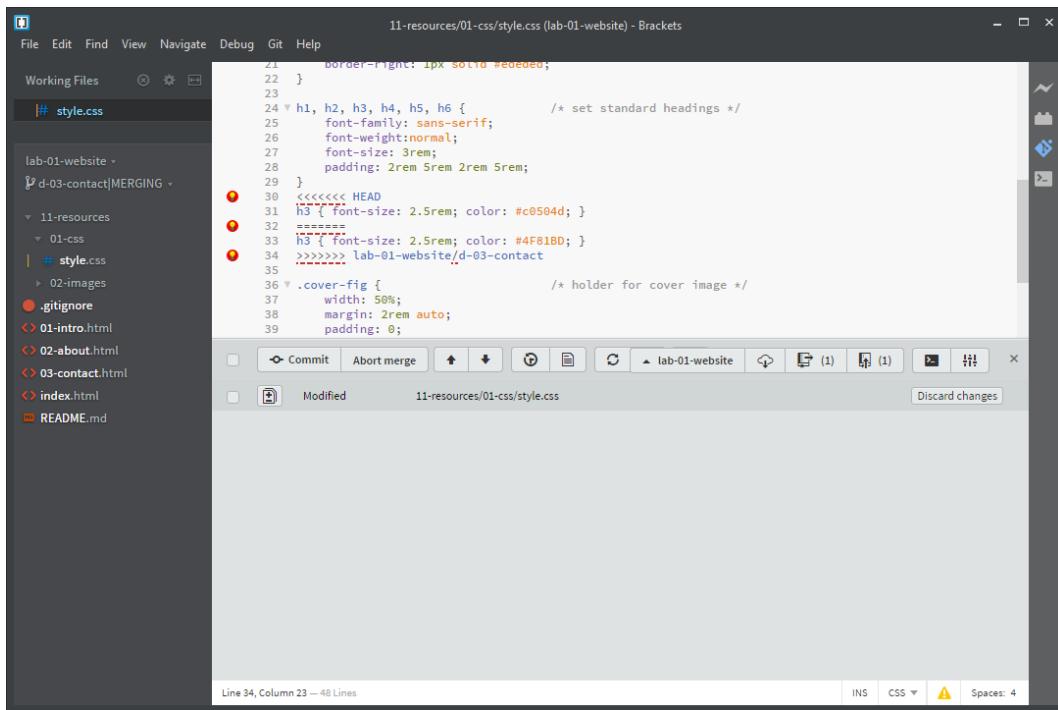


Figure 8.55 Brackets—with a merge conflict

Again we have an **ABORT MERGE** button in the Git pane. Click that and we go back to where we were before we tried to merge the repositories.

Brackets is telling us there is a conflict in the `style.css` file in just the same way as before, but this time the conflict is between the local `head` on `d-03-contact` and the remote `d-03-contact` branch on the remote `lab-01-website` repository:

```

30 <<<<< HEAD
31 h3 { font-size: 2.5rem; color: #c0504d; }
32 =====
33 h3 { font-size: 2.5rem; color: #4F81BD; }
34 >>>>> lab-01-website/d-03-contact

```

Code 8.4 style.css with a conflict

It starts with a line of less than signs:

<<<<< HEAD

The `HEAD` tells us that what follows is from the current local `head` (which is on the `d-03-contact` branch).

The line that follows it is how the code is on the local branch (the red version):

`h3 { font-size: 2.5rem; color: #c0504d; }`

Then we have a row of equal signs, this is just a divider:

=====

The last two lines are how the code is on the remote repository branch `d-03-contact` (this is the blue version):

```

h3 { font-size: 2.5rem; color: #4F81BD; }
>>>>> lab-01-website/d-03-contact

```

The greater than signs show the merging repository, the `lab-01-website` is what we called the link to the remote repository (Figure 8.13) the `/d-03-contact` identifies the remote branch.

Again, Brackets is telling us it can't decide what to do and it wants us to manually change the file to the correct version.

This time I'm going to keep the local (red) version.

Make the modifications:

```
STYLE.CSS
```

```
28     padding: 2rem 5rem 2rem 5rem;
29 }
30 h3 { font-size: 2.5rem; color: #c0504d; }
31
32 .cover-fig { /* holder for cover image */
```

Code 8.5 Resolved style.css

Note: You must delete the lines with the less than, greater than and equals signs too.

Mine looks like this after the modifications, Figure 8.56:

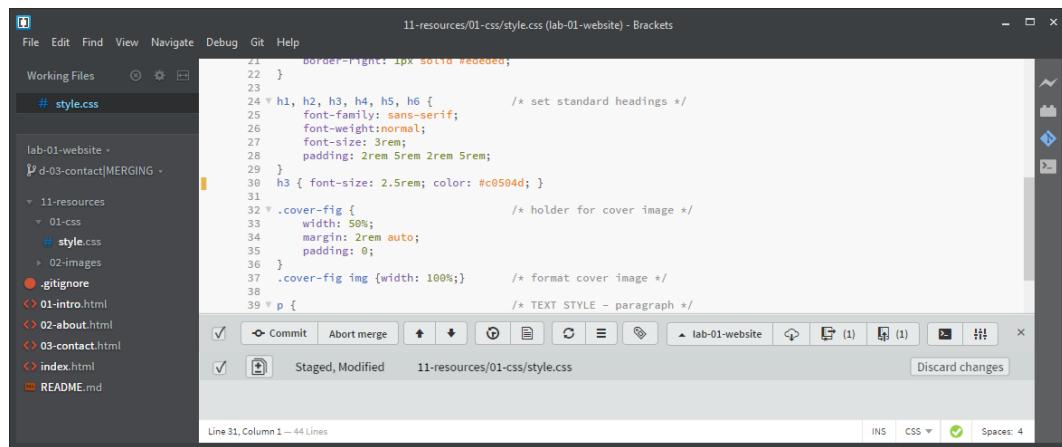


Figure 8.56 Brackets—modified file after remote/local conflict

Save **style.css** and make sure it is staged (tick the box in the Git pane).

Now click **COMMIT**.

This will open a merge commit message dialogue box:

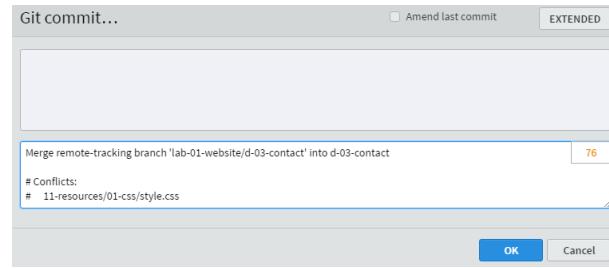


Figure 8.57 Brackets—merge commit dialogue box

Just go with the default message, click **OK** and return to Brackets:

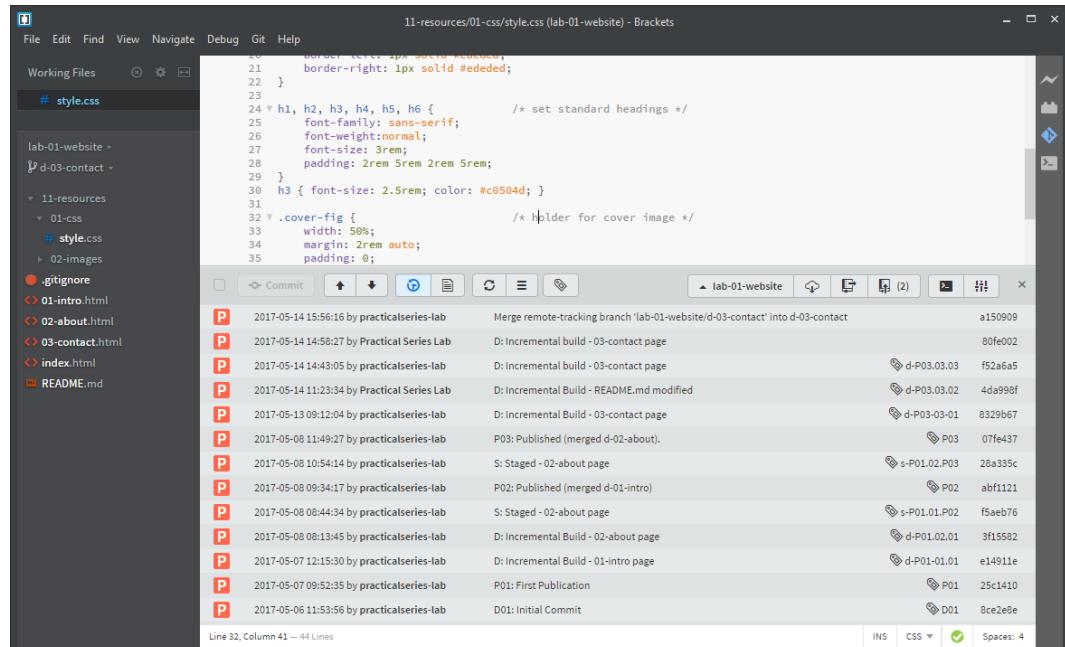


Figure 8.58 Brackets—after the local/remote merge

So what's happened?

Well, the local repository now has its original commit with the modified `style.css` (red) [`f52a6a5`]; it also has the commit for the modified `style.css` (blue) from the remote repository [`80fe002`]; and finally, it has a new merge commit [`a150909`] that resolves the conflict between the remote and local repositories (makes it red again).

Give the new commit point the tag `s-P03.03.P04`.

The local work flow is:

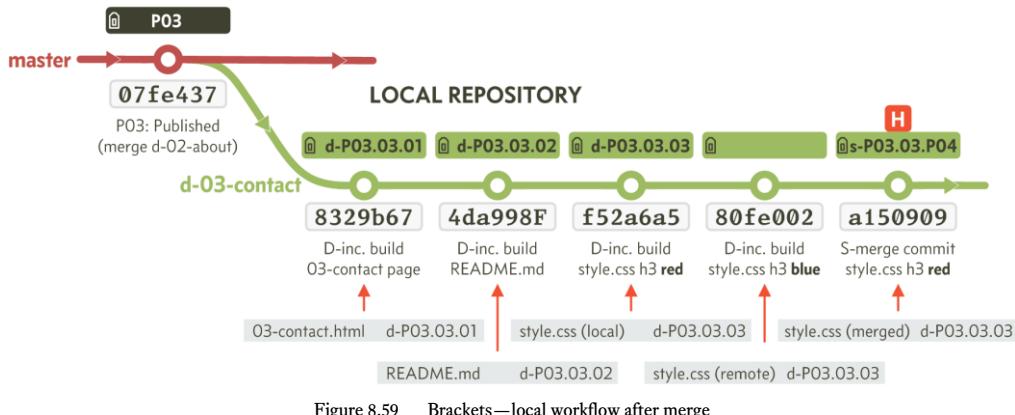


Figure 8.59 Brackets—local workflow after merge

Brackets is now reporting that there are two commits to push back to the remote repository: (2). The two commits are the original `style.css` modification [\[f52a6a5\]](#) and the latest merge commit [\[a150909\]](#).

Let's do the push to complete the thing. Click the `GIT PUSH` button (2).

Tick the `SEND TAGS` box on the push to remote dialogue box and click `OK`, and click `OK` again to close the response box.

Now to GitHub. Refresh the repository and make sure the `d-03-commit` branch is active. I have this:

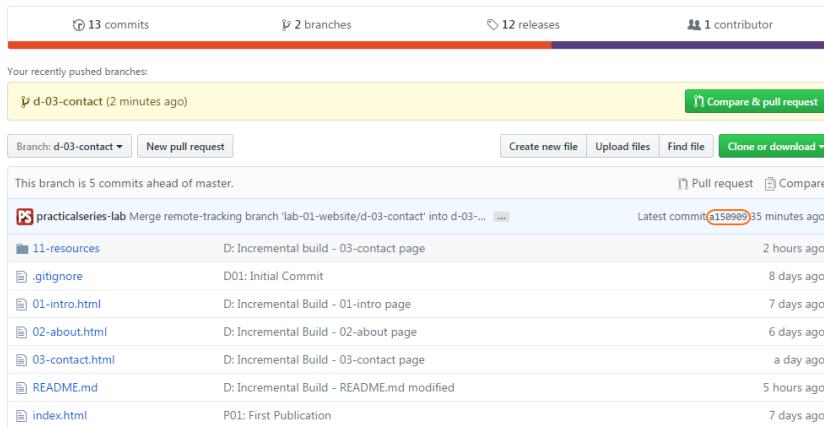


Figure 8.60 GitHub—local/remote merged repositories

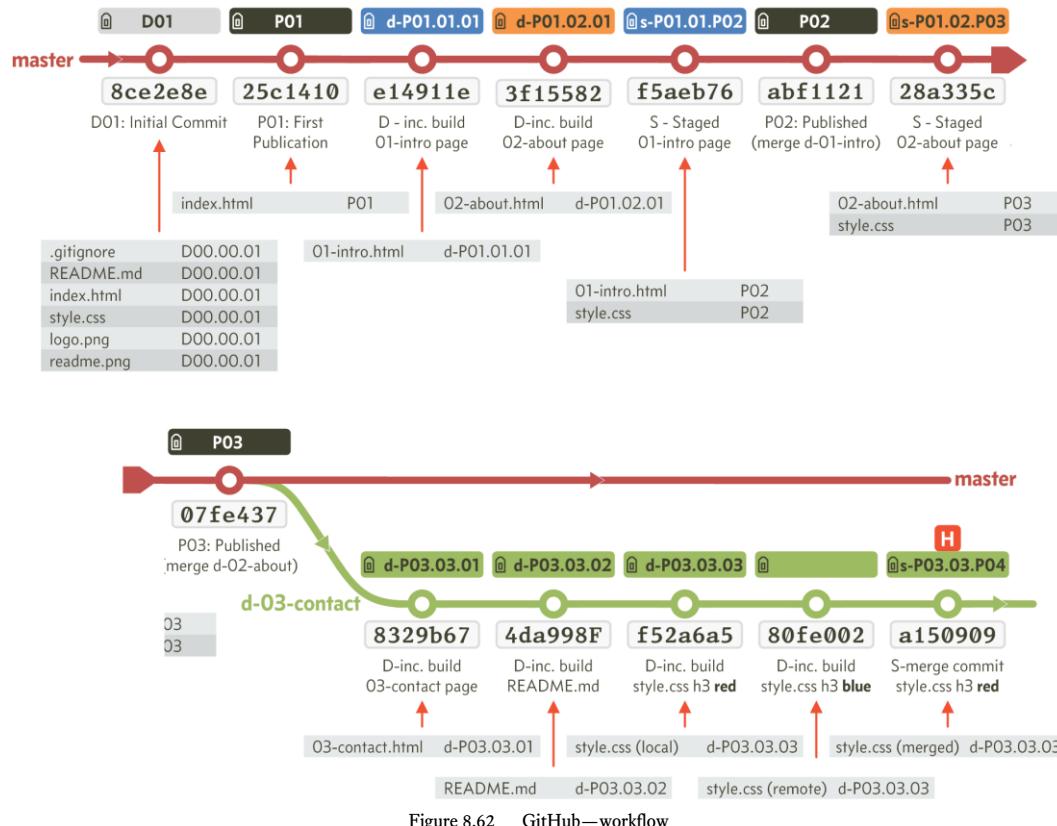
GitHub is reporting 13 commits and that the latest commit is [a150909]. Go to the commits screen, everything is there:

The screenshot shows the GitHub repository page for `practicalseries-lab / lab-01-website`. The navigation bar at the top includes links for This repository, Search, Pull requests, Issues, Gist, Watch (0), Star (0), Fork (0), and a settings icon. Below the navigation bar, there are tabs for Code, Issues (0), Pull requests (0), Projects (0), Wiki, Pulse, Graphs, and Settings. A dropdown menu for the branch shows `d-03-contact`. The main content area displays a list of commits grouped by date:

- Commits on May 14, 2017:**
 - Merge remote-tracking branch 'lab-01-website/d-03-contact' into d-03-contact ... (practicalseries-lab committed 40 minutes ago) - Commit ID: a150909
 - D: Incremental build - 03-contact page ... (practicalseries-lab committed on GitHub 2 hours ago) - Commit ID: 80fe002
 - D: Incremental build - 03-contact page ... (practicalseries-lab committed 2 hours ago) - Commit ID: f52a6a5
 - D: Incremental Build - README.md modified ... (practicalseries-lab committed on GitHub 5 hours ago) - Commit ID: 4da99bf
- Commits on May 13, 2017:**
 - D: Incremental Build - 03-contact page ... (practicalseries-lab committed a day ago) - Commit ID: 8329b67
- Commits on May 8, 2017:**
 - P03: Published (merged d-02-about). (practicalseries-lab committed 6 days ago) - Commit ID: 07fe437
 - S: Staged - 02-about page ... (practicalseries-lab committed 6 days ago) - Commit ID: 28a335c
 - P02: Published (merged d-01-intro) (practicalseries-lab committed 6 days ago) - Commit ID: abf1121
 - S: Staged - 02-about page ... (practicalseries-lab committed 6 days ago) - Commit ID: f5aeb76
 - D: Incremental Build - 02-about page ... (practicalseries-lab committed 6 days ago) - Commit ID: 3f15582
- Commits on May 7, 2017:**
 - D: Incremental Build - 01-intro page ... (practicalseries-lab committed 7 days ago) - Commit ID: e14911e
 - P01: First Publication ... (practicalseries-lab committed 7 days ago) - Commit ID: 25c1410
- Commits on May 6, 2017:**
 - D01: Initial Commit ... (practicalseries-lab committed 8 days ago) - Commit ID: 8ce2e8e

Figure 8.61 GitHub—all commits

The final workflow is:



8.6

Deleting remote branches

There are currently two branches, the **master** branch and **d-03-contact**. Let's assume that we've finished everything on the **d-03-contact** branch and we want to merge it back on to the **master** branch and then delete **d-03-contact**.

The current situation is that everything is up to date, the local and remote repositories match (this is because I just pushed all the changes to the remote at the end of the previous section).

To make sure there are no differences between the remote and local, we can do a fetch to check, click the **FETCH** button .

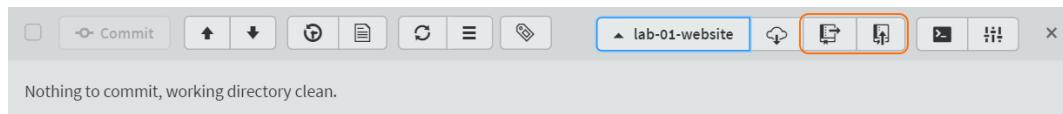


Figure 8.63 Brackets—check local and remote are in synch

The fact that there are no numbers in the **GIT PULL** and **GIT PUSH** buttons (highlighted) shows that there are no differences.

Next thing is to merge **d-03-contact** into the **master** branch locally. This is an identical process to that in § 6.7.1.

To merge branches, we must be on the receiving branch (the **master** branch). In Brackets switch to the **master** branch, click the arrow next to the selected branch in the left file tree pane:

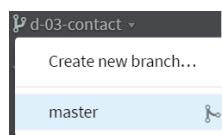


Figure 8.64 Brackets—switch back to **master** branch

You should now be on the **master** branch; the **03-contact.html** file will no longer be visible in the file tree:

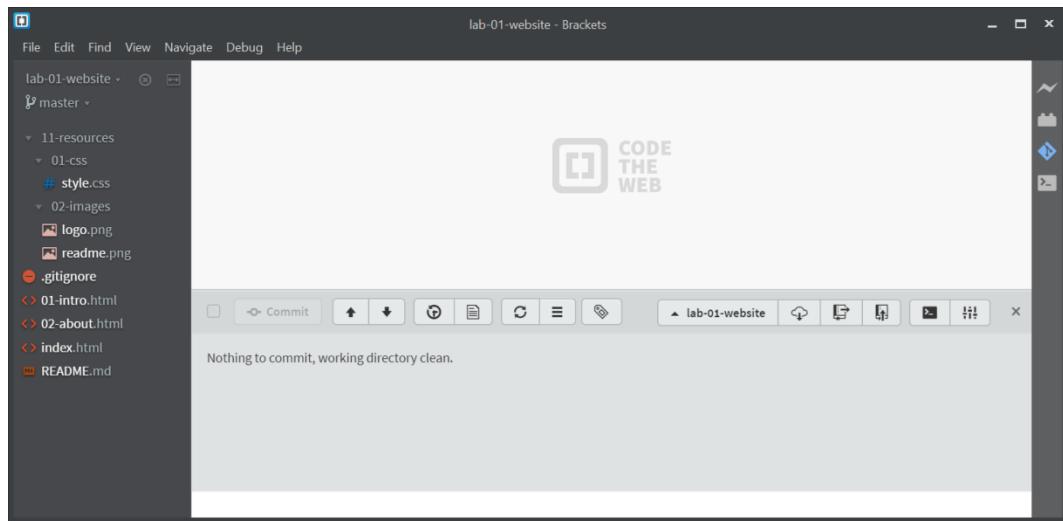


Figure 8.65 Brackets—on the **master** branch

To merge the branches, again click the arrow next to the **master** branch in the file tree and this time click the merge icon next to **d-03-contact**:

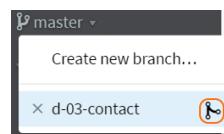


Figure 8.66 Brackets—merge **d-03-contact** branch

In the merge dialogue box, enter the merge message shown below:

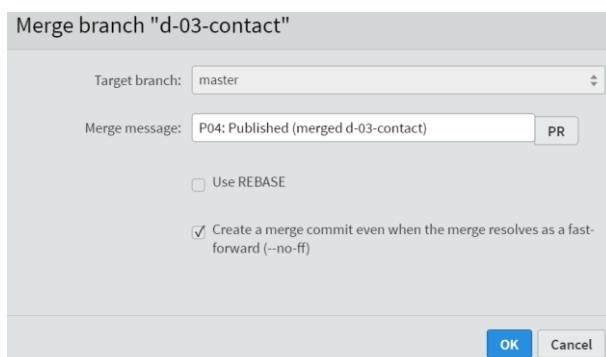
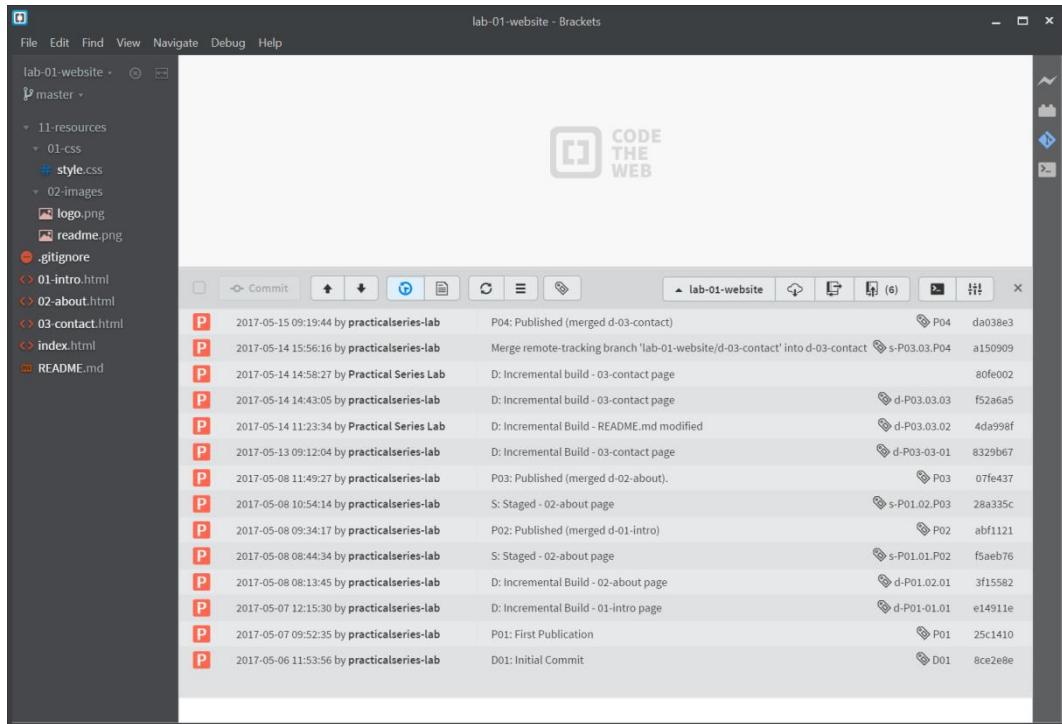


Figure 8.67 Brackets—merge dialogue box

Click **OK** and **CLOSE** the merge result box (there are no conflicts).

My new commit point is **[da038e3]**, tag the new commit point with tag: **P04**.

Examining the commit history now shows:

A screenshot of the Brackets IDE interface. The title bar says "lab-01-website - Brackets". The left sidebar shows a file tree for "lab-01-website" with branches "master" and "d-03-contact". The "master" branch contains files: 11-resources, 01-css (with "style.css" checked), 02-images (with "logo.png" and "readme.png"), .gitignore, 01-intro.html, 02-about.html, 03-contact.html, index.html, and README.md. The main workspace shows a logo for "CODE THE WEB". Below the workspace is a commit history panel titled "Commit". It lists 19 commits, all from the "master" branch, with the last 6 being new. The commits are: P 2017-05-15 09:19:44 by practicalseries-lab (P04: Published (merged d-03-contact)), P 2017-05-14 15:56:16 by practicalseries-lab (Merge remote-tracking branch 'lab-01-website/d-03-contact' into d-03-contact (s-P03.03.P04)), P 2017-05-14 14:58:27 by Practical Series Lab (D: Incremental build - 03-contact page), P 2017-05-14 14:43:05 by practicalseries-lab (D: Incremental build - 03-contact page), P 2017-05-14 11:23:34 by Practical Series Lab (D: Incremental Build - README.md modified), P 2017-05-13 09:12:04 by practicalseries-lab (D: Incremental Build - 03-contact page), P 2017-05-08 11:49:27 by practicalseries-lab (P03: Published (merged d-02-about). (s-P03.03.P03)), P 2017-05-08 10:54:14 by practicalseries-lab (S: Staged - 02-about page), P 2017-05-08 09:34:17 by practicalseries-lab (P02: Published (merged d-01-intro) (s-P01.02.P02)), P 2017-05-08 08:44:34 by practicalseries-lab (S: Staged - 02-about page), P 2017-05-08 08:13:45 by practicalseries-lab (D: Incremental Build - 02-about page (d-P01.02.01)), P 2017-05-07 12:15:30 by practicalseries-lab (D: Incremental Build - 01-intro page (d-P01-01.01)), P 2017-05-07 09:52:35 by practicalseries-lab (P01: First Publish (P01)), and P 2017-05-06 11:53:56 by practicalseries-lab (D01: Initial Commit (D01)).

Author	Date	Message	Type	SHA
practicalseries-lab	2017-05-15 09:19:44	P04: Published (merged d-03-contact)	P	da308e3
practicalseries-lab	2017-05-14 15:56:16	Merge remote-tracking branch 'lab-01-website/d-03-contact' into d-03-contact (s-P03.03.P04)	P	a150909
Practical Series Lab	2017-05-14 14:58:27	D: Incremental build - 03-contact page	P	80fe002
practicalseries-lab	2017-05-14 14:43:05	D: Incremental build - 03-contact page	P	f52a6a5
Practical Series Lab	2017-05-14 11:23:34	D: Incremental Build - README.md modified	P	4da998f
practicalseries-lab	2017-05-13 09:12:04	D: Incremental Build - 03-contact page	P	8329b67
practicalseries-lab	2017-05-08 11:49:27	P03: Published (merged d-02-about). (s-P03.03.P03)	P	07fe437
practicalseries-lab	2017-05-08 10:54:14	S: Staged - 02-about page	P	28a335c
practicalseries-lab	2017-05-08 09:34:17	P02: Published (merged d-01-intro) (s-P01.02.P02)	P	abf1121
practicalseries-lab	2017-05-08 08:44:34	S: Staged - 02-about page	P	f5ae676
practicalseries-lab	2017-05-08 08:13:45	D: Incremental Build - 02-about page (d-P01.02.01)	P	3f15582
practicalseries-lab	2017-05-07 12:15:30	D: Incremental Build - 01-intro page (d-P01-01.01)	P	e14911e
practicalseries-lab	2017-05-07 09:52:35	P01: First Publish (P01)	P	25c1410
practicalseries-lab	2017-05-06 11:53:56	D01: Initial Commit (D01)	P	8ce2e8e

Figure 8.68 Brackets—final commit history

Everything is back on the **master** branch (locally at least) and Brackets tells us there are six commits to push to the **master** branch on the remote repository.

This is because all those commits that were on the **d-03-contact** branch are now on the **master** branch (locally), but the remote repository knows nothing about them yet—hence six commits to push.

Make the push, click the **GIT PUSH** button: .

As always, in the push dialogue box make sure the **SEND TAGS** box is ticked, leave everything else and click **OK**.

Close the response box and open the remote repository in GitHub. I now have 14 commits in total on the master branch, matching the local repository (Figure 8.69):

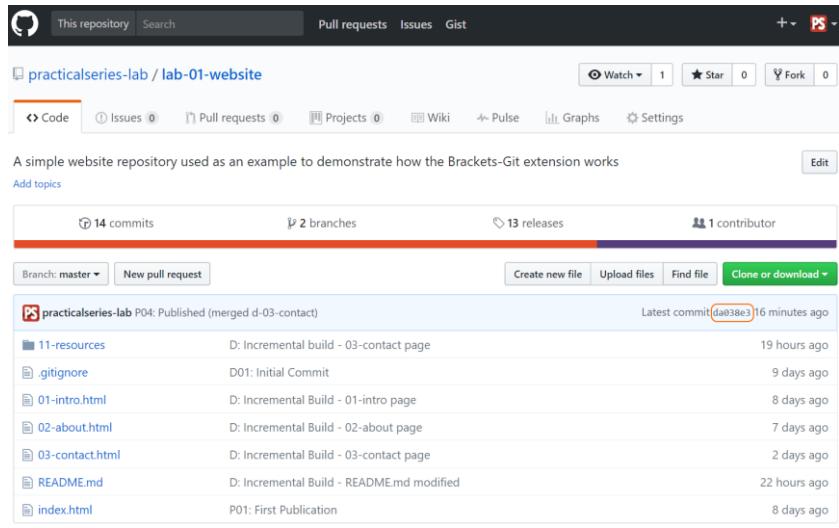


Figure 8.69 GitHub—updated **master** branch

It is also showing the most recent P04 commit point [[da038e3](#)].

So that's all good. Only thing now is to delete the **d-03-contact** branch, both locally and remotely.

8.6.1 Deleting the remote branch

There is one golden rule for deleting a branch in both a local and a remote repository:

ALWAYS DELETE THE REMOTE BRANCH FIRST

In Brackets switch to the **d-03-contact** branch. It will look like this.

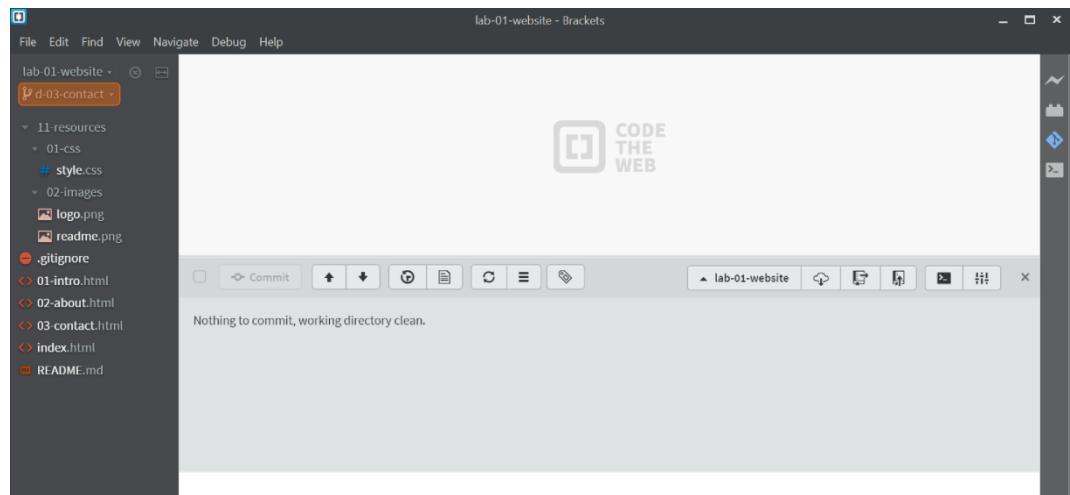


Figure 8.70 Brackets—on the redundant **d-03-contact** branch

Make sure you are on **d-03-contact (highlighted).**

You can only delete the remote version of the branch that is active locally. We don't want to delete the **master** branch.

Now click the **GIT PUSH** button: . This opens the push dialogue box:

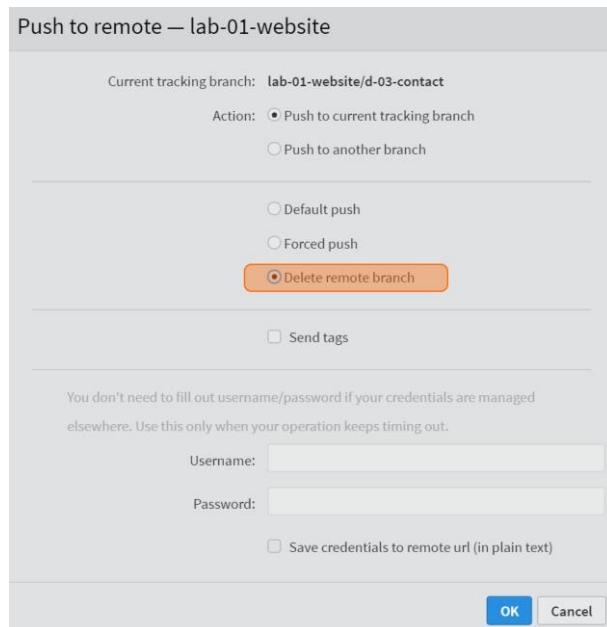


Figure 8.71 Brackets—delete remote branch

This time select **DELETE REMOTE BRANCH** (highlighted), leave everything else and click **OK**. This give the following response:

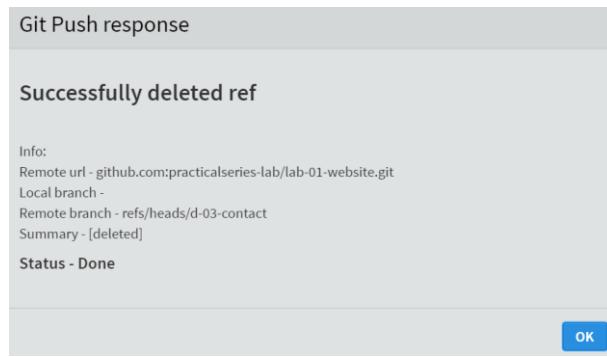


Figure 8.72 Brackets—delete remote branch response

Looks like it's done it. Click **OK**.

Refresh the GitHub repository:

The screenshot shows a GitHub repository page for 'practicalseries-lab / lab-01-website'. The repository has 14 commits, 1 branch, 13 releases, and 1 contributor. A dropdown menu is open under 'Branch: master' with the option 'd-03-contact' selected. Below the dropdown, a list of files and their commit history is shown, including '02-about.html', '03-contact.html', 'README.md', and 'index.html'. The commit history for '03-contact.html' includes 'Initial Commit', 'Incremental Build - 01-intro page', 'D: Incremental Build - 02-about page', 'D: Incremental Build - 03-contact page', 'D: Incremental Build - README.md modified', and 'P01: First Publication'.

Figure 8.73 GitHub—after d-03-contact deletion

First thing: in the tab bar it shows just one branch (highlighted). Secondly, clicking the **BRANCH** button there is only the **master** branch in the dropdown.

We've deleted **d-03-contact** from the remote repository.

8.6.2 Deleting the local branch

This is just what we did in § 6.7.2.

In Brackets, switch back to the **master** branch. Click the small arrow next to the **master** branch in the file tree:

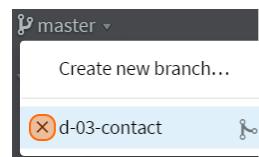


Figure 8.74 Brackets—delete the local branch

Click the cross next to the **d-03-contact** branch in the dropdown. Click **OK** in the “are you sure?” dialogue box.

That's it; the **d-03-contact** branch is now deleted from both the local and remote repositories.

This is the final workflow—both locally and remotely:

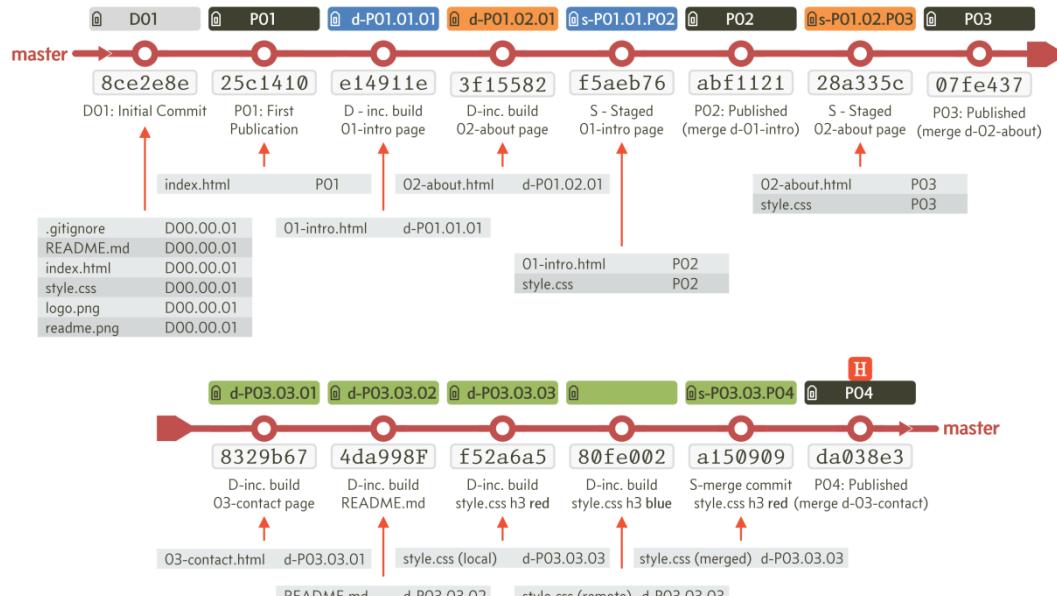


Figure 8.75 final workflow

9

GITHUB

How to use GitHub to manage a remote repository.

THIS SECTION concentrates on the GitHub website and how to use it.

My first impression with GitHub was that it was quite difficult to understand and navigate. This was partly because it is, but mostly (at the time) because I didn't understand the concepts behind Git. Hopefully I've addressed the Git version control mechanisms and concepts in the previous sections so that just leaves GitHub itself. That's what this section is for.

In this section I cover the basics of GitHub:

- ① Examining files and folders
- ② Editing files directly on GitHub
- ③ Creating new files
- ④ Renaming and deleting files
- ⑤ Examining commits and the commit list
- ⑥ Examining and recovering earlier commit points
- ⑦ Creating branches
- ⑧ Comparing and pull requests
- ⑨ Merging and Handling conflicts
- ⑩ Tags and releases

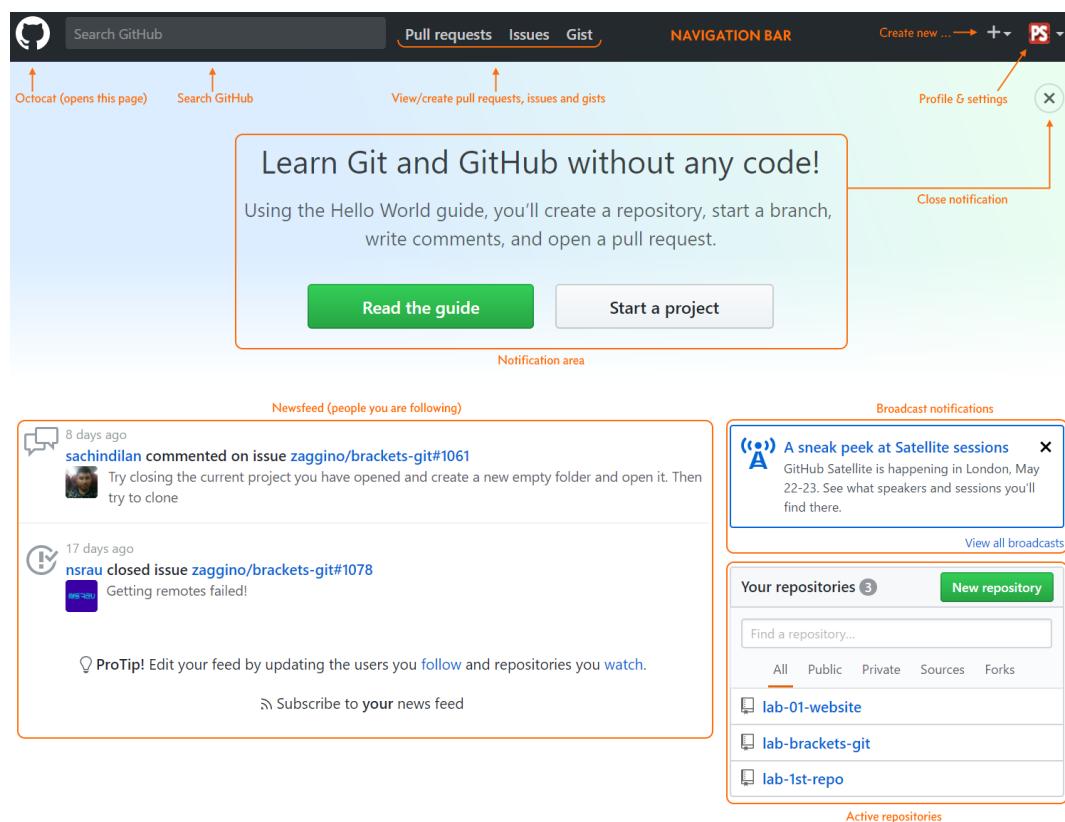
9.1

GitHub—the basic pages

Let's have a look around the GitHub website. Log on to your GitHub profile. This will open your profile landing page—this is universally called the newsfeed page.

9.1.1 GitHub profile—newsfeed page

My newsfeed page looks like this:



© 2017 GitHub, Inc. Terms Privacy Security Status Help



Contact GitHub API Training Shop Blog About

Figure 9.1 GitHub—newsfeed page

We've seen this page in the earlier sections, but I didn't really explain it—let's look through what we have:

Starting at the top with the navigation bar:

- **OCTOCAT ICON**—always takes you back to the newsfeed page (this page)
- **SEARCH GITHUB**—searches GitHub for repositories and entries in README files
- **PULLS AND ISSUES**—I look at these in sections 10
- **CREATE NEW**—create new repositories &c.
- **YOUR PROFILE**—view your profile and settings

The main area:

The top of the main area shows **NOTIFICATIONS**, the one that is currently on the page “**LEARN GIT...**” is the default notification you get when you create a profile—I tend to close the notification area (click the cross indicated).

BROADCASTS, are blogs and articles that GitHub thinks you might be interested in (again I close this box).

NEWSFEED, this shows posts and activity by any user you might be following (*watching* in the creepy terminology of GitHub). If you view any GitHub user profile, you will be given the option to follow that user; similarly, if you view a particular repository, you can watch that repository. In either case, any relevant postings or activity will show on your newsfeed page.

Note: A word of caution, once you follow people or repositories you will receive entries on your newsfeed page, you cannot get rid of these entries (even if you stop following that profile or repository).

Finally, the **REPOSITORIES**; this is a list of your repositories and any other repositories that you may have copied (*forked* in GitHub's vulgar parlance). This is a quick way to access your repositories or create a new one (see § 5.2.1).

After I've simplified things, the **PracticalSeries-lab** newsfeed page looks like this:

The screenshot shows the GitHub newsfeed for the user 'PracticalSeries-lab'. At the top, there is a navigation bar with links for 'Pull requests', 'Issues', and 'Gist'. On the right side of the header, there are buttons for creating a new repository ('New repository') and switching between public and private repositories ('Public' is selected). Below the header, there are two entries in the newsfeed:

- sachindilan commented on issue zaggino/brackets-git#1061**
Try closing the current project you have opened and create a new empty folder and open it. Then try to clone
- nrau closed issue zaggino/brackets-git#1078**
Getting remotes failed!

On the right side of the screen, there is a sidebar titled 'Your repositories' which lists three repositories:

- lab-01-website
- lab-brackets-git
- lab-1st-repo

At the bottom of the page, there is a 'ProTip!' section with the text 'Edit your feed by updating the users you follow and repositories you watch.' and a link to 'Subscribe to your news feed'.

© 2017 GitHub, Inc. [Terms](#) [Privacy](#) [Security](#) [Status](#) [Help](#) [Contact GitHub](#) [API](#) [Training](#) [Shop](#) [Blog](#) [About](#)

Figure 9.2 GitHub—tidied up newsfeed page

Note: The entries in my newsfeed are from the Brackets-Git repository, I'm following it.

By and large, the newsfeed page is really only useful as a quick way to access your repositories.

Let's look at a repository home page, click on the [lab-01-website](#) link in the **YOUR REPOSITORIES** section:

9.1.2 A repository home page

Clicking any repository link opens the home page for that repository; this is the [lab-01-website](#) home page:

The screenshot shows the GitHub repository home page for 'practicalseries-lab / lab-01-website'. The top navigation bar includes links for 'This repository', 'Search', 'Pull requests', 'Issues', 'Gist', and 'NAVIGATION BAR'. Collaboration buttons for 'Watch' (1), 'Star' (0), and 'Fork' (0) are also present. Below the header, the repository path 'practicalseries-lab / lab-01-website' is displayed. The main content area includes sections for 'Code' (Issues 0, Pull requests 0, Projects 0, Wiki, Pulse, Graphs, Settings), 'PROJECT TABS' (Code, Issues 0, Pull requests 0, Projects 0, Wiki, Pulse, Graphs, Settings), and 'A simple website repository used as an example to demonstrate how the Brackets-Git extension works'. A 'Repository description and edit' button is available. Below this, a summary bar shows 14 commits, 1 branch, 13 releases, and 1 contributor. Control buttons include 'Branch: master', 'New pull request', 'Create new file', 'Upload files', 'Find file', and 'Clone or download'. The 'LATEST COMMIT (HEAD INFORMATION)' section lists commits for files like 11-resources, .gitignore, 01-intro.html, 02-about.html, 03-contact.html, README.md, and index.html, with dates ranging from 3 days ago to 11 days ago. The 'PROJECT CONTENT AREA' section contains a large image of a book cover titled 'THE PRACTICAL SERIES LAB'. The 'README.md FILE CONTENTS' section displays the text: 'The Practical Series of publications is a website resource for web developers and engineers. It contains a number of online publications designed to help and explain how to build a website, how to use version control and how to write engineering software for control systems. This particular repository is designed as an example project to demonstrate how to build a Git and GitHub repository using the Brackets-Git extension for the Brackets text editor. The full set of PracticalSeries publications is available at [practicalseries.com](#).'. The footer includes links for 'Contact GitHub', 'API', 'Training', 'Shop', 'Blog', and 'About', along with copyright information for 2017 GitHub, Inc.

Figure 9.3 GitHub—repository home page

Starting at the top, the **NAVIGATION BAR**, this is identical to that in § 9.1.1.

The **USERNAME** link and the **REPOSITORY** link navigate to the profile newsfeed page (Figure 9.1) and the repository home page (this page) respectively. These are quick links to return to the landing page for either the user or the repository—they are visible from most pages in a repository.

The **PROJECT TABS** allow the user to view the various aspects of the repository; these are covered in the following sections.

Beneath the **PROJECT TABS** is the **PROJECT DESCRIPTION** (entered when the project was created). This description can be modified by clicking the **EDIT** button. The **ADD TOPICS** link can be used to add keywords to the description; these keywords are used to make the repository more discoverable in GitHub searches.

Following this is the **STATUS BAR** (I covered some aspects of this in § 8.3.2), I go through the more useful elements of this bar in this section and the next.

Next we have the **CONTROL BUTTONS**, these allow various actions to be taken within the repository (change or create branches, create files &c.).

The **HEAD INFORMATION** shows the **LATEST COMMIT** on the current branch, when it was made, who made it and the first line of the commit message. This is the pale blue band across the middle of the page.

Underneath this is the current repository directory (**PROJECT CONTENT AREA**). This allows files and folders to be opened and edited.

Finally, there is a preview of the **README.MD** file in all its technicolour glory.

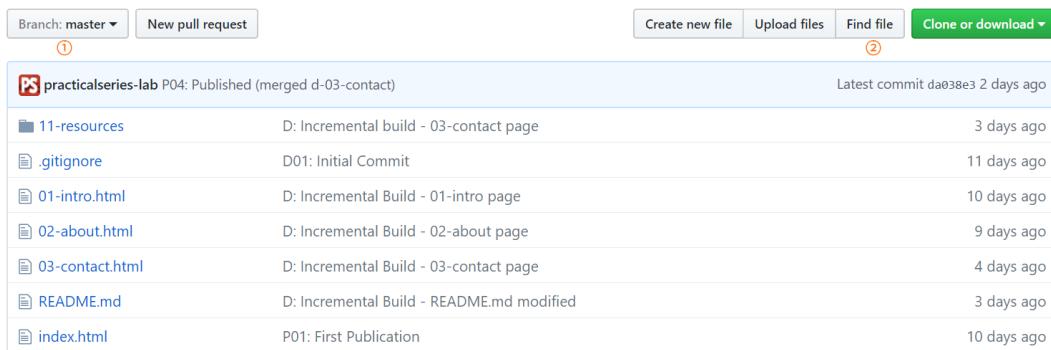
Some of this stuff we've already used and some of it is new. I'm not going to go through absolutely all of it (*there is some stuff in there I've never even looked at*), but I will cover the main bits in some detail.

9.2

GitHub—finding and viewing files

This is a more methodical view of working within the GitHub environment; we've looked at some aspects of this in the previous sections, but not in as much detail.

The basic view of the project (repository) contents is that of Figure 9.3 and below:



A screenshot of a GitHub repository page showing the contents of the master branch. The top navigation bar includes buttons for 'Branch: master' (with a dropdown arrow), 'New pull request', 'Create new file', 'Upload files', 'Find file', and 'Clone or download'. The main area displays a list of files and their details. A commit message from 'P04: Published (merged d-03-contact)' is shown at the top right. Below it is a list of files with their last modification times:

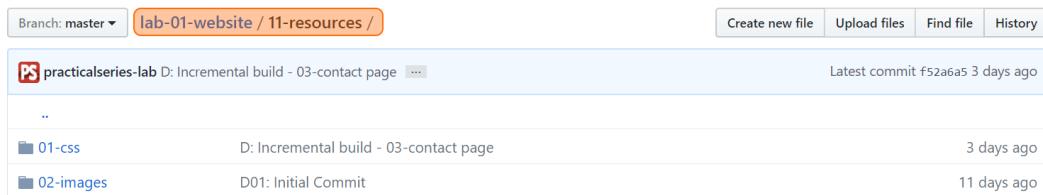
File	Description	Last Modified
11-resources	D: Incremental build - 03-contact page	3 days ago
.gitignore	D01: Initial Commit	11 days ago
01-intro.html	D: Incremental Build - 01-intro page	10 days ago
02-about.html	D: Incremental Build - 02-about page	9 days ago
03-contact.html	D: Incremental Build - 03-contact page	4 days ago
README.md	D: Incremental Build - README.md modified	3 days ago
index.html	P01: First Publication	10 days ago

Figure 9.4 GitHub—Repository contents

This view shows the contents of the repository for the currently selected branch, in this case the **master** branch (this can be seen in the **BRANCH** button point ① above).

This view just shows the root folder and its contents, files are given the symbol  and folders the symbol . The most recent commit comment associated with the file or folder is displayed in the centre of the area and the time of its last modification displayed on the far right (in relative time).

This is similar to a Windows Explorer view, if you click the **11-resources** link, it will drill down into that folder:



A screenshot of a GitHub repository page showing the contents of the 11-resources directory within the master branch. The top navigation bar includes buttons for 'Branch: master' (with a dropdown arrow), 'lab-01-website / 11-resources /' (highlighted in orange), 'Create new file', 'Upload files', 'Find file', and 'History'. The main area displays a list of files with their details. A commit message from 'D: Incremental build - 03-contact page' is shown at the top right. Below it is a list of files with their last modification times:

File	Description	Last Modified
01-css	D: Incremental build - 03-contact page	3 days ago
02-images	D01: Initial Commit	11 days ago

Figure 9.5 GitHub—opening a sub-directory

Once you have drilled down to a sub-directory, the link back to the root level is available by clicking the `lab-01-website` link next to the branch button (highlighted in Figure 9.5).

As you drill down further, the parent folders become clickable links in their own right in the same area.

It is also possible to go up a directory level by clicking the two full stops (..) that are listed as the first line of the **CONTENT AREA** (always just below the pale blue **HEAD BAR**).

Note: As you click down through the directories, the latest commit changes to reflect the latest commit in that directory, in Figure 9.4 (root) the latest commit is [da038e3]; in Figure 9.5 (`11-resources` folder) this changes to [f52a6a5].

9.2.1 File view

Go back to the root level of the project (Figure 9.4); now click the **FIND FILE** button, point ② above (an alternative is to press the **T** key, this does the same).

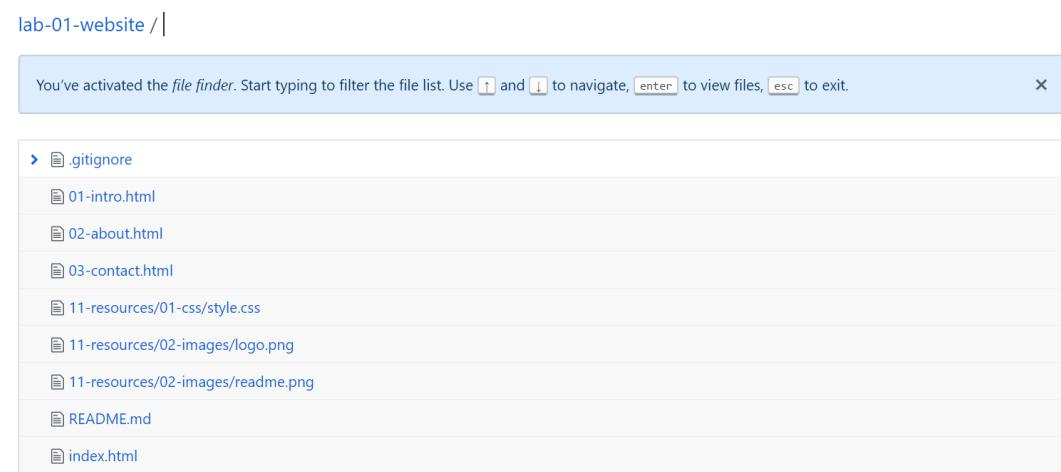


Figure 9.6 GitHub—file view mode

This shows a list of all the files (along with the path) within the repository.

The cursor is by default positioned after the `lab-01-website /`, if you start typing something it will filter the list by that search (type `0`, and you will have all the files and folders that start with a zero). Just press `ESC` to return back to the normal view.

The file view can be useful, it saves you having to drill down through multiple folders (with a page load each time) and it means you don't have to know which folder a file is in. It does however, become a bit cumbersome if you have a lot of files.

9.2.2 Viewing the contents of a file

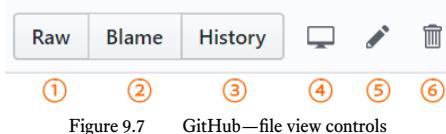
Viewing the contents of a file is easy; just click its name in either the content area or the file view and it will load the file in a viewer.

To view `style.css`, either drill down to it `11-resources/01-css/style.css` or find it in file view and just click its name.

This will open a view of the file contents; it will look like Figure 9.8.

The view shows all the lines in the CSS file, these are semantically coloured (see § 9.2.3).

There are also three buttons and three icons at the top of the file contents area:



The three buttons are used to examine different views of the file. The icons do things to the file. I look at each in turn.

Branch: master ▾ lab-01-website / 11-resources / 01-css / style.css Find file Copy path

practicalseries-lab D: Incremental build - 03-contact page f52a6a5 3 days ago

1 contributor

44 lines (38 sloc) | 1.05 KB

```

1  * {
2    margin: 0;
3    padding: 0;
4    box-sizing: border-box;
5    position: relative;
6  }
7
8  html {
9    background-color: #fbfaf6;      /* Set cream coloured page background */
10   color: #404030;
11   font-family: serif;
12   font-size: 26px;
13   text-rendering: optimizeLegibility;
14 }
15
16 body {
17   max-width: 1276px;
18   margin: 0 auto;
19   background-color: #fff;        /* make content area background white */
20   border-left: 1px solid #eddede;
21   border-right: 1px solid #eddede;
22 }
23
24 h1, h2, h3, h4, h5, h6 {          /* set standard headings */
25   font-family: sans-serif;
26   font-weight: normal;
27   font-size: 3rem;
28   padding: 2rem 5rem 2rem 5rem;
29 }
30 h3 { font-size: 2.5rem; color: #c0504d; }
31
32 .cover-fig {                      /* holder for cover image */
33   width: 50%;
34   margin: 2rem auto;
35   padding: 0;
36 }
37 .cover-fig img { width: 100%; }     /* format cover image */
38
39 p {                                /* TEXT STYLE - paragraph */
40   margin-bottom: 1.2rem;
41   padding: 0 5rem;
42   line-height: 135%;
43 }
```

Raw Blame History

① ② ③ ④ ⑤ ⑥

Figure 9.8 GitHub—viewing the contents of a file

The **RAW** button

The **RAW** button ① shows a view of the file contents without any formatting. It looks like this:

```
* {
    margin: 0;
    padding: 0;
    box-sizing: border-box;
    position: relative;
}

html {
    background-color: #fbfaf6;      /* Set cream coloured page background */
    color: #404030;
    font-family: serif;
    font-size: 26px;
    text-rendering: optimizeLegibility;
}

body {
    max-width: 1276px;
    margin: 0 auto;
    background-color: #fff;        /* make content area background white */
    border-left: 1px solid #eddede;
    border-right: 1px solid #eddede;
}

h1, h2, h3, h4, h5, h6 {           /* set standard headings */
    font-family: sans-serif;
    font-weight: normal;
    font-size: 3rem;
    padding: 2rem 5rem 2rem 5rem;
}
h3 { font-size: 2.5rem; color: #c0504d; }

.cover-fig {                      /* holder for cover image */
    width: 50%;
    margin: 2rem auto;
    padding: 0;
}
.cover-fig img {width: 100%;}       /* format cover image */

p {                                /* TEXT STYLE - paragraph */
    margin-bottom: 1.2rem;
    padding: 0 5rem;
    line-height: 135%;
}
```

Figure 9.9 GitHub—file raw view

The **HISTORY** button

The history button ③ shows all the commits associated with the file (this is analogous to the file history in Brackets, § 6.4.1).

History for [lab-01-website / 11-resources / 01-css / style.css](#)

- o Commits on May 14, 2017
 -  D: Incremental build - 03-contact page ... practicalseries-lab committed 3 days ago [copy] [f52a6a5] [diff]
- o Commits on May 8, 2017
 -  P03: Published (merged d-02-about). ... practicalseries-lab committed 9 days ago [copy] [07fe437] [diff]
 -  S: Staged - 02-about page ... practicalseries-lab committed 9 days ago [copy] [28a335c] [diff]
 -  S: Staged - 02-about page ... practicalseries-lab committed 9 days ago [copy] [f5aeb76] [diff]
- o Commits on May 6, 2017
 -  D01: Initial Commit ... practicalseries-lab committed 11 days ago [copy] [8ce2e8e] [diff]

Figure 9.10 GitHub—file history view

This is a commit view and I discuss how this works in § 9.4 where I talk about commits. You get the idea; it's a list of all the commits that affected the file.

The **BLAME** button

This is *blame* as in *whose fault is it?* It's button ②, and it gives a more detailed version of the file history; it shows how the contents of the file have changed. Here it is:

```
* {
    margin: 0;
    padding: 0;
    box-sizing: border-box;
    position: relative;
}

html {
    background-color: #fbfaf6; /* Set cream coloured page background */
    color: #404030;
    font-family: serif;
    font-size: 26px;
    text-rendering: optimizeLegibility;
}

body {
    max-width: 1276px;
    margin: 0 auto;
    background-color: #fff; /* make content area background white */
    border-left: 1px solid #ebeded;
    border-right: 1px solid #ebeded;
}

h1, h2, h3, h4, h5, h6 { /* set standard headings */
    font-family: sans-serif;
    font-weight: normal;
    font-size: 3rem;
    padding: 2rem 5rem 2rem 5rem;
}

h3 { font-size: 2.5rem; color: #c0504d; }

.cover-fig { /* holder for cover image */
    width: 50%;
    margin: 2rem auto;
    padding: 0;
}
.cover-fig img {width: 100%;} /* format cover image */

p { /* TEXT STYLE - paragraph */
    margin-bottom: 1.2rem; /* *** THIS SETS PARAGRAPH SPACING *** */
    padding: 0 5rem;
    line-height: 135%;
}
```

Figure 9.11 GitHub—file blame view

This is a listing of the file contents and its showing that lines 1 to 29 are all associated with the first commit (D01 commit), lines 31 to 43 are the same. Line 30 however is

listed as an incremental build. This is from when we were changing files to generate conflicts.

Clicking the  icon will move the view of the file back to that commit (it shows the version of the file at the time of that commit), it will show the commits associated with each line of the file at that time. It allows changes to be viewed in turn all the way back to the first commit of the file.

The colour of the central vertical bar gives some indication of how recent the file was modified (newest yellow, oldest dark red).

I get the idea; it makes it possible to view every change to any file through the commit history. In practice, I've never used it.

The **GITHUB DESKTOP** icon

This is the **GITHUB DESKTOP** icon () point ④ on the list.

GitHub has a desktop application (a GUI frontend for Git and GitHub). I don't use it, I use Brackets instead. I can offer no opinion on it, I haven't tried it—from the reviews I've read, it has certain restrictions (I don't know what these are); but then again so does Brackets.

The **EDIT** and **DELETE** file icons

The final two icons edit () and delete () points ⑤ and ⑥ respectively allow the file to be edited (as we did in § 8.4) or (not surprisingly) will delete the file.

Both these actions require a commit to be made in the remote repository and I cover them fully in the next section (§ 9.3).

9.2.3 Viewing other types of files

The file we looked at in the previous section was a CSS file, essentially this is a text file and it was displayed as such in Figure 9.8. GitHub knows it's a CSS file and is able to colour it semantically¹ to make it easier to read. It does this for all the common web languages HTML, JavaScript jQuery &c.

Let's view a different type of file, open the [README .md](#) file:

The screenshot shows the GitHub interface for the file `README.md` in the `lab-01-website` repository. The page header includes the branch `master`, the repository name `lab-01-website / README.md`, and standard GitHub navigation buttons for `Find file` and `Copy path`. Below the header, a commit card for `practicalseries-lab` is shown, indicating a modification to `README.md` 3 days ago. The main content area displays the Markdown file's text, which includes a title "A PracticalSeries Publication" and an image of a red book cover titled "THE PRACTICAL SERIES LAB". The text below the image describes the Practical Series of publications as a website resource for web developers and engineers, mentioning online publications, version control, and engineering software for control systems. It also links to practicalseries.com. The page features several sections: "How to use this repository", "Website contents" (listing files like `index.html`, `01-intro.html`, etc.), "Contributors" (listing Michael Gledhill), and a "Licence" section stating the repository is a demonstration project. A note at the bottom indicates the contents are free to use.

Figure 9.12 GitHub—file preview

¹

Symantec colouring or highlighting is the technique of colouring elements of the source code to reflect the category of the item, in CSS this might be rules, declarations, class names, element names units and arguments—they are all coloured differently to make the code more readable.

GitHub doesn't show the source code behind the file, it interprets it and renders it according to the rules of the mark-up language.

Similarly, if you open an image GitHub renders the image for you to see, here is `logo.png`:

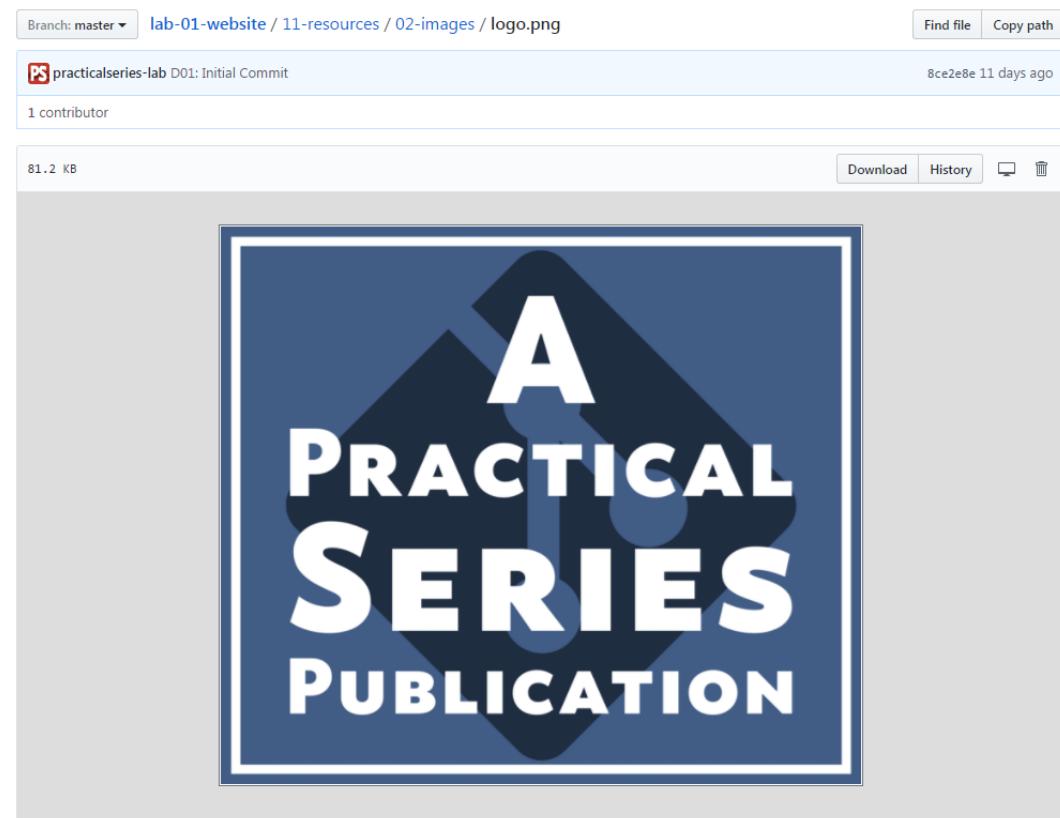


Figure 9.13 GitHub—image file view

GitHub tries to do the right thing when you view any file in the repository.

9.3

GitHub—creating, modifying, deleting and committing files

More of the basics, some of this we've done before and a lot of it is fairly obvious; I'm just going through it in a more structured manner.

There is one thing to say about creating and editing files within GitHub: **don't do it**.

It is generally better to modify the files locally (in Brackets) and push the changes up to the remote GitHub repository. This is the best practice method for using Git and GitHub.

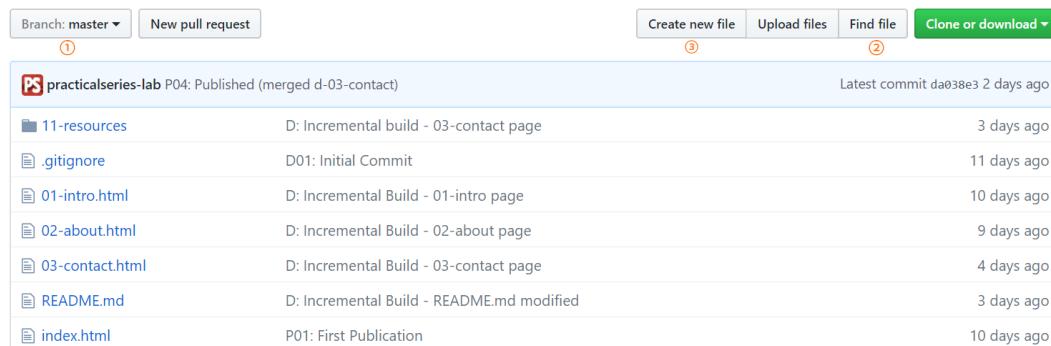
That said, virtually everybody will at some point modify files in GitHub and this is how you do it.

9.3.1 Creating new folders and files

Let's go through the process of creating a new folder and then creating a new file inside it.

Against all my rules, but for the sake of simplicity I'm going to do this directly on the **master** branch in [the lab-01-website repository](#).

Go to the repository home page on GitHub, mine looks like this:



The screenshot shows a GitHub repository page for 'practicalseries-lab'. At the top, there are buttons for 'Branch: master' (with a dropdown arrow), 'New pull request', 'Create new file' (pointed to by a circled ③), 'Upload files', 'Find file', and 'Clone or download'. Below the header, the repository name is 'practicalseries-lab P04: Published (merged d-03-contact)' and the latest commit is 'da038e3 2 days ago'. A table lists the repository contents:

File	Commit Message	Time Ago
11-resources	D: Incremental build - 03-contact page	3 days ago
.gitignore	D01: Initial Commit	11 days ago
01-intro.html	D: Incremental Build - 01-intro page	10 days ago
02-about.html	D: Incremental Build - 02-about page	9 days ago
03-contact.html	D: Incremental Build - 03-contact page	4 days ago
README.md	D: Incremental Build - README.md modified	3 days ago
index.html	P01: First Publication	10 days ago

Figure 9.14 GitHub—Repository contents

Click the [CREATE NEW FILE](#) button (point ③ above).

It looks like Figure 9.15:

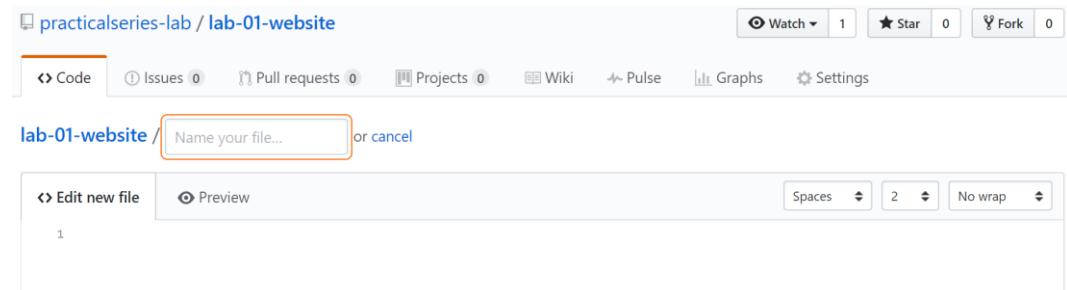


Figure 9.15 GitHub—create file page

It's asking for a file name. If I just give it a name `content.txt` for example, it will create the file in the root directory i.e. `lab-01-website/content.txt`.

Now I want to put the file in a new folder *so how do I create the folder to put it in?* The answer is just start typing the folder name and file name in the highlighted box. Separate the file name from the directory name with an oblique character (`/`). I want to create a file called `content.txt` and I want it to live in a new directory called `docs`.

To do this start typing `docs` in the name your file box:

lab-01-website / or cancel

Now hit the oblique key (`/`) to separate the folder from the file name, the moment you do, it will recognise `docs` is a directory and move it outside the box:

lab-01-website / or cancel

Continue typing the filename (`content.txt`) and when you are done hit return (*clever isn't it*).

Now add some text into the body of the file, I've added the following five lines:

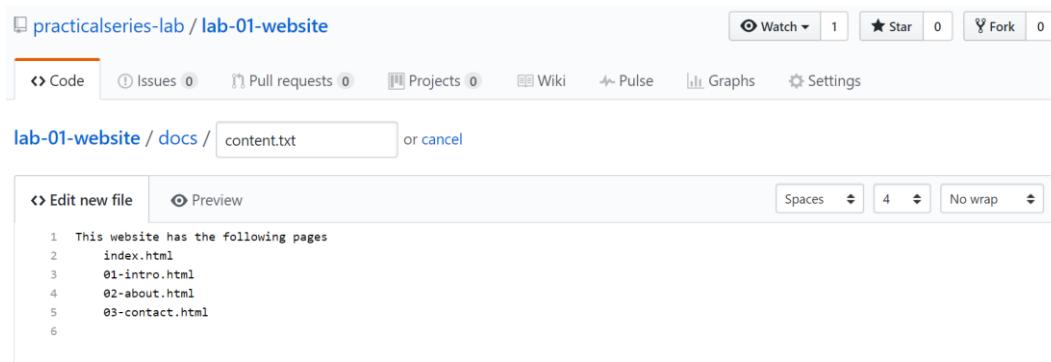


Figure 9.16 GitHub—add text to the file

The buttons at the top: **SPACES**, **NUMBER**, **NO WRAP** show how the page is edited and displayed. The **NO WRAP** button can be changed to cause lines to wrap in the display (generally, for long lines set this to **SOFT WRAP**). The **SPACES** button determines how tabs are constructed, either from the tab character or by spaces, if spaces then the number of spaces specified in the next (**NUMBER**) button will be inserted whenever the tab key is pressed.

Generally, for text files, leave it set to **SPACES** and change the number to **4**, this keeps it in line with the way Brackets handles such files.

Now we need to commit the file, this is just what we did in § 8.4.

9.3.2 Committing files

Scroll down to the bottom of the page (to the commit box) and enter the following commit message:

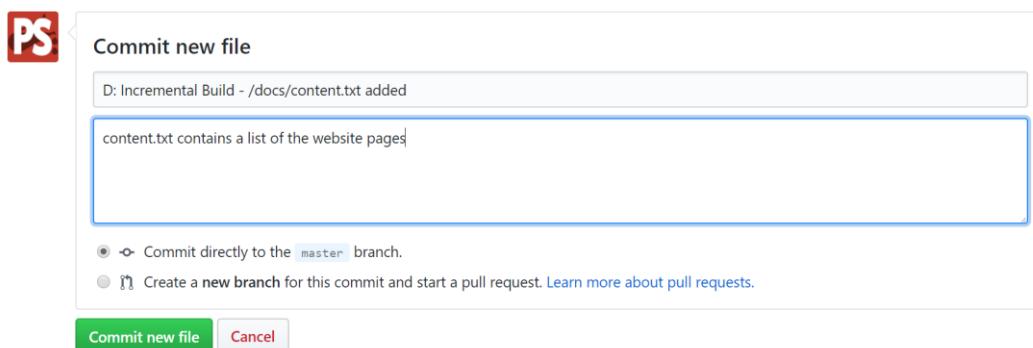


Figure 9.17 GitHub—commit the file

Click the **COMMIT NEW FILE** button to add the file to the repository this will take you to the newly created **docs** folder and show you the new file and commit number [64478f4] in my case:

The screenshot shows a GitHub repository page for 'practicalseries-lab / lab-01-website'. The 'docs' folder is selected in the sidebar. A new file, 'content.txt', has been added. The commit message is 'D: Incremental Build - /docs/content.txt added'. The commit was made by 'practicalseries-lab' and is dated 'a minute ago'. The commit number is 64478f4.

Figure 9.18 GitHub—new file and commit number

Go back to the repository home page:

The screenshot shows the GitHub repository home page for 'practicalseries-lab / lab-01-website'. It displays 15 commits, 1 branch, 13 releases, and 1 contributor. The 'docs' folder is now listed under '11-resources'. The latest commit is 64478f4, made 3 minutes ago. Other commits include 'Initial Commit' and modifications to 'index.html' and 'README.md'.

Figure 9.19 GitHub—repository home page after new file created

The new **docs** folder is there and the commit number has changed to [64478f4].

9.3.3 Uploading a file

Let's add another file to the `docs` folder, this time by uploading it from a PC. The file in question is a short word document explaining my version control numbering mechanism. Get it here ↴. It's called:

[PS1002-5-2121-091 R01.00 PS Version Numbering.docx](#)

Boy, do I like numbering things.

Back to GitHub, from the [lab-01-website](#) repository home page, navigate to the `docs` folder (by clicking it):

The screenshot shows the GitHub repository page for 'practicalseries-lab / lab-01-website'. The 'Code' tab is selected. In the top right, there are buttons for 'Watch' (1), 'Star' (0), 'Fork' (0), and 'Settings'. Below the tabs, it says 'Branch: master' and shows a commit history for 'lab-01-website / docs /'. One commit is highlighted: 'PS practicalseries-lab committed on GitHub D: Incremental Build - /docs/content.txt added' with a timestamp of 'Latest commit 64478f4 28 minutes ago'. At the bottom of the commit list, there is a link to 'content.txt' and a note 'D: Incremental Build - /docs/content.txt added' with a timestamp of '28 minutes ago'. A prominent orange button labeled 'Upload files' is highlighted with a red box.

Figure 9.20 GitHub—upload file

Click the **UPLOAD FILE** button (highlighted):

This gets you:

[lab-01-website / docs](#)

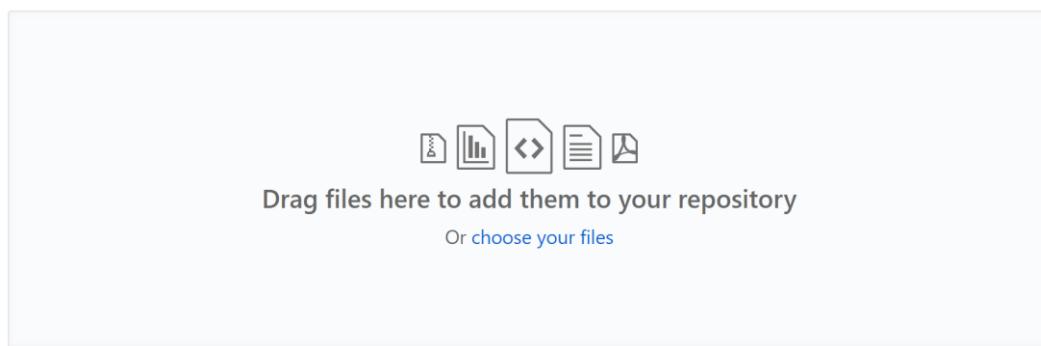


Figure 9.21 GitHub—upload file window

Either drag your file into the main area or click **CHOOSE YOUR FILE** and navigate with Windows Explorer to where your file is, I did the latter:

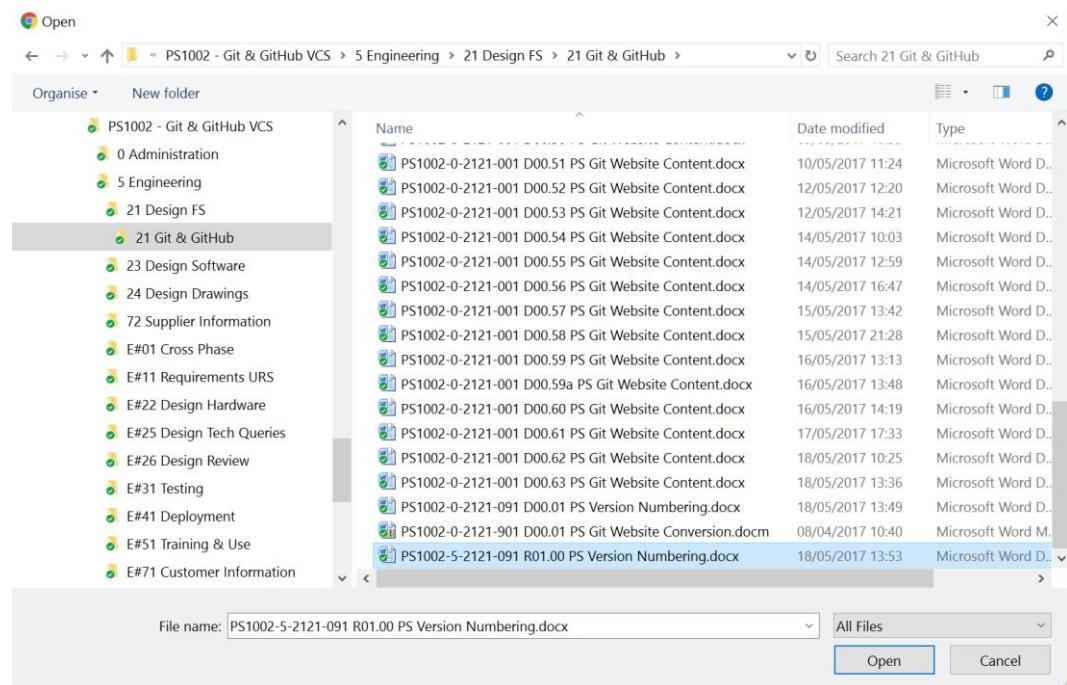


Figure 9.22 GitHub—upload file dialogue box

Select the file and click **OPEN**.

This uploads the file to GitHub, but doesn't add it to the repository yet.

To add the file to the repository, we must complete the commit message box at the bottom of the screen. I've added the message:

P05: Published – Version Numbering doc added

PS1002-5-2121-091 R01.00 PS Version Numbering.docx added to /docs

It all looks like this:

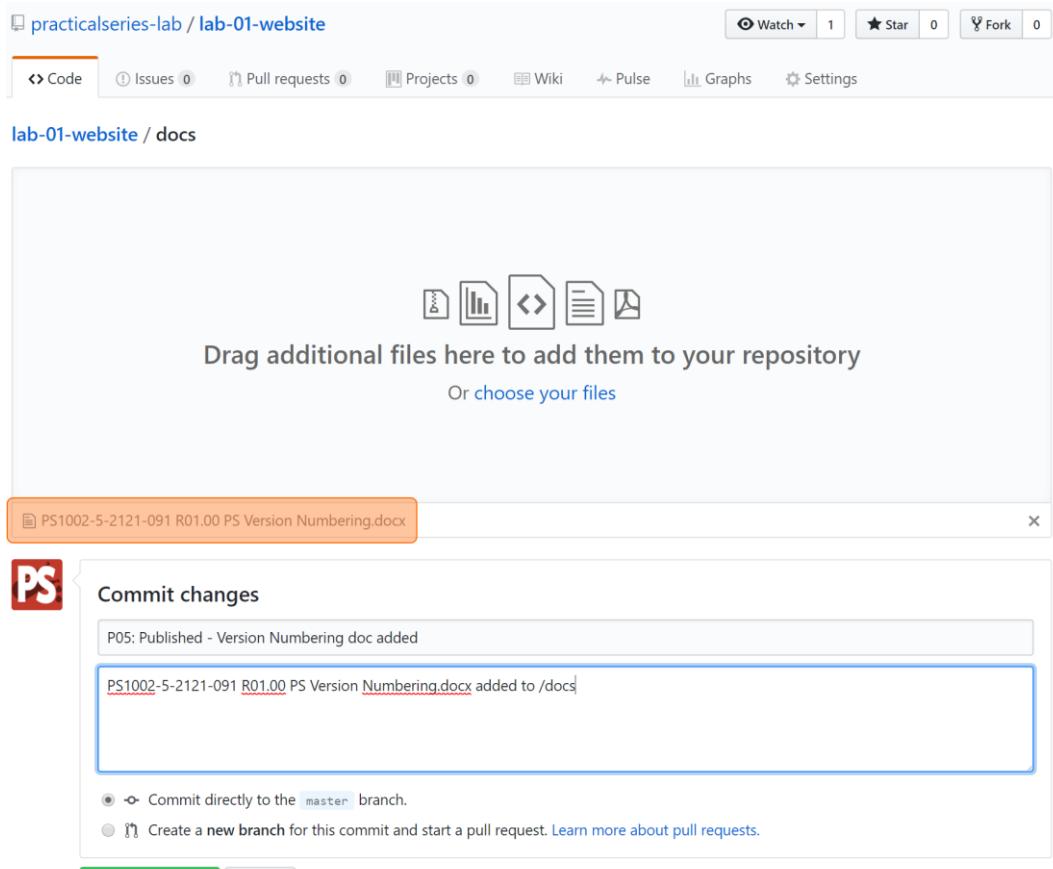


Figure 9.23 GitHub—upload commit message

You can see the new file at the bottom of the drag window (highlighted).

Click the **COMMIT CHANGES** button to add the file to the repository home page and will show you the new file and commit number, **[100bd32]** in my case.

Navigate to the **docs** folder (by clicking it):

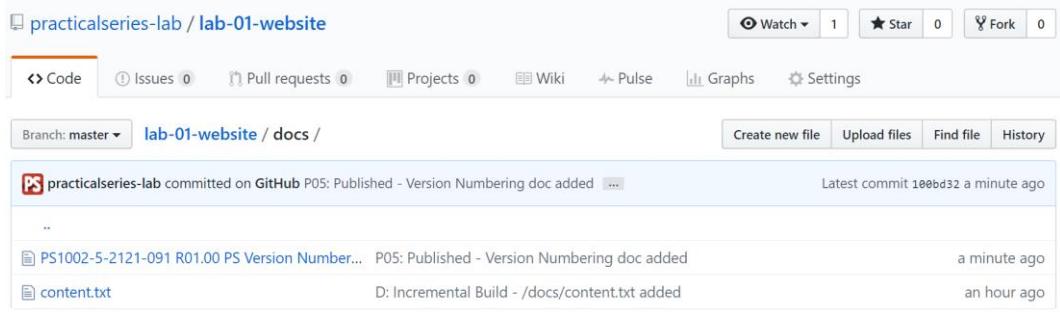


Figure 9.24 GitHub—uploaded file in the docs folder

All pretty straight forward.

One last thing; the file we just uploaded is a Word file, a `.docx` file, now GitHub knows about text file based source code (HTML, CSS, JS &c.) and it has an editor that can handle them. It does not however, have an editor for Word files, if you double click the new file to open it, GitHub will respond with:

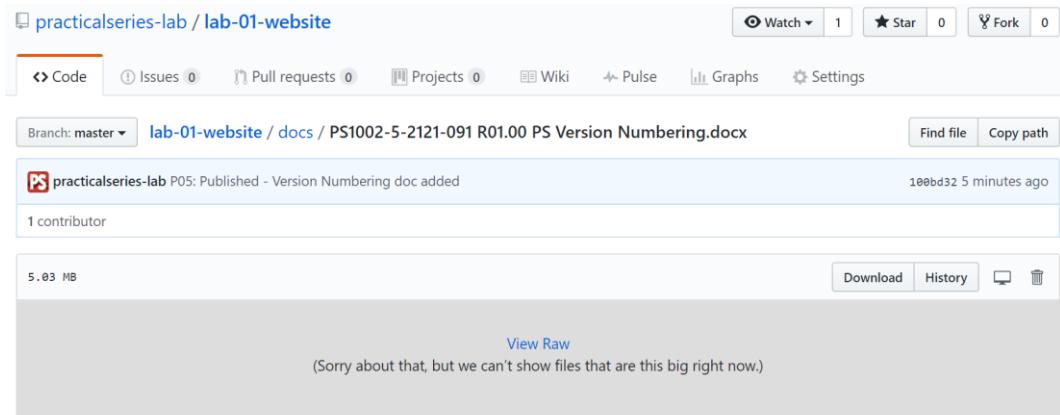


Figure 9.25 GitHub—try to open a Word file

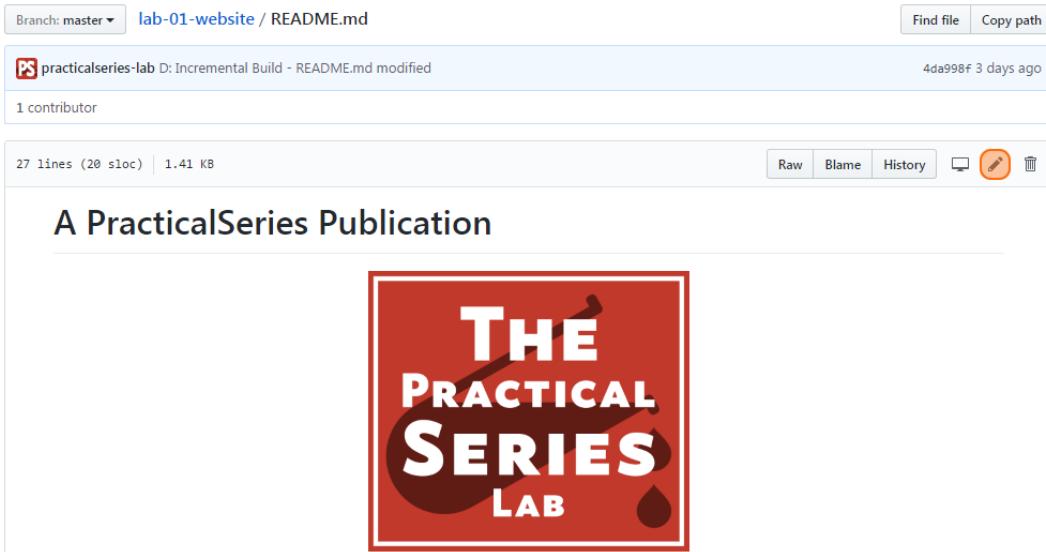
This is understandable. If you need to modify the file, do it in Word and upload it again.

Note: If you click `VIEW RAW`, it will just download the file.

9.3.4 Editing a file

We've done this before, but for completeness I'll cover it again here.

Go to the repository home page and click the `README.md` file, this opens the file as a preview (just like § 9.2.3). It looks like this:



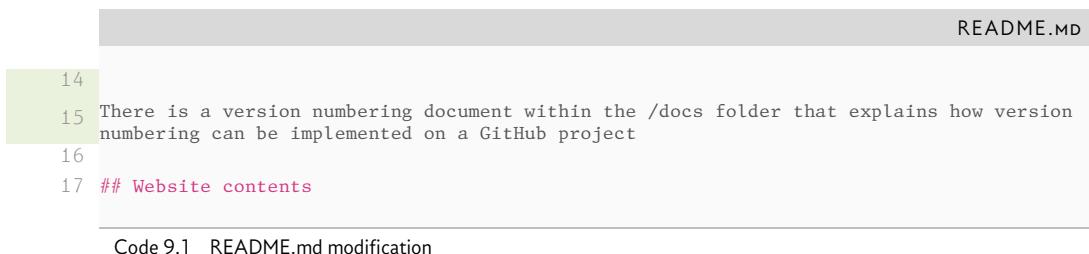
A screenshot of a GitHub repository page for 'lab-01-website'. The top navigation bar shows 'Branch: master' and the file path 'lab-01-website / README.md'. Below the header, there is a summary card with the PS icon, the repository name 'practicalseries-lab', the branch 'D: Incremental Build - README.md modified', the commit hash '4da998f', and the date '3 days ago'. It also indicates '1 contributor'. The main content area shows the file's content: 'A PracticalSeries Publication' followed by a logo graphic for 'THE PRACTICAL SERIES LAB'. At the bottom of the preview area, there are links for 'Raw', 'Blame', 'History', and an orange 'Edit' icon.

Figure 9.26 GitHub—README.md preview

Click the edit icon (📝) to open the file editor.

Because we have opened a markdown file (with extension `.md`), we have the option of previewing the file (i.e. see how it will appear when it is rendered as markdown, like it is in Figure 9.26). To preview the file click the preview tab at the top, it has the 🌐 symbol.

In the editor, add the following lines:



```
14
15 There is a version numbering document within the /docs folder that explains how version
    numbering can be implemented on a GitHub project
16
17 ## Website contents
```

Code 9.1 README.md modification

It looks like this on the website (I've highlighted the new lines):

```
13 This repository is intended to be used with the accompanying documentation [practicalseries Git and GitHub]  
(http://practicalseries.com/0021-git-vcs/index.html "Practical Series - Git and GitHub").  
14  
15 There is a version numbering document within the /docs folder that explains how version numbering can be implemented on a GitHub project.  
16  
17 ## Website contents  
18 This site contains the following pages:  
19 * index.html  
20 * 01-intro.html  
21 * 02-about.html  
22 * 03-contact.html  
23 |  
24 ## Contributors  
25 This repository was constructed by [Michael Gledhill](https://github.com/mgledhill) "Michael Gledhill".  
26  
27 ## Licence  
28 This is simply a demonstration repository, the contents are free to use by anyone who wishes to do so.  
29
```

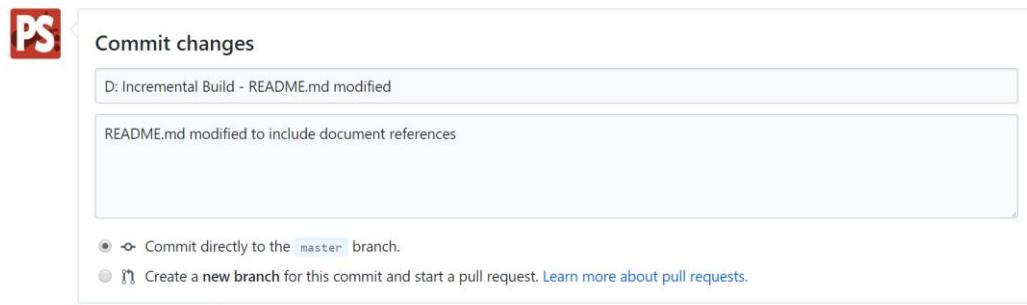


Figure 9.27 GitHub—commit the edited file

Add the commit comments as shown and click **COMMIT CHANGES** to add the modified file to the repository.

This takes us back to the file preview page and shows the new commit number [[5644edc](#)].

You can see the changed lines at the bottom of the page (highlighted):

practicalseries-lab / lab-01-website

Code Issues Pull requests Projects Wiki Pulse Graphs Settings

Branch: master lab-01-website / README.md Find file Copy path

PS practicalseries-lab D: Incremental Build - README.md modified 5644edc a minute ago

1 contributor

29 lines (21 sloc) | 1.55 KB Raw Blame History

A PracticalSeries Publication



The Practical Series of publications is a website resource for web developers and engineers. It contains a number of online publications designed to help and explain how to build a website, how to use version control and how to write engineering software for control systems. This particular repository is designed as an example project to demonstrate how to build a Git and GitHub repository using the Brackets-Git extension for the Brackets text editor. The full set of PracticalSeries publications is available at practicalseries.com.

How to use this repository

This repository is a worked example demonstrating how to build a version control project using Git and GitHub from within the Brackets text editor. This repository is intended to be used with the accompanying documentation [practicalseries Git and GitHub](#).

There is a version numbering document within the /docs folder that explains how version numbering can be implemented on a GitHub project.

Figure 9.28 GitHub—final edited file

9.3.5 Renaming a file

To demonstrate this, let's rename the `content.txt` file that we created in § 9.3.1; let's change its name to `page-list.txt`.

This is easy.

Open the file in GitHub (from the repository home page click the `docs` folder and then click `content.txt`).

Now edit the file by clicking the pencil symbol (✏️), just like we did for `README.md` in the last section. You should have this:

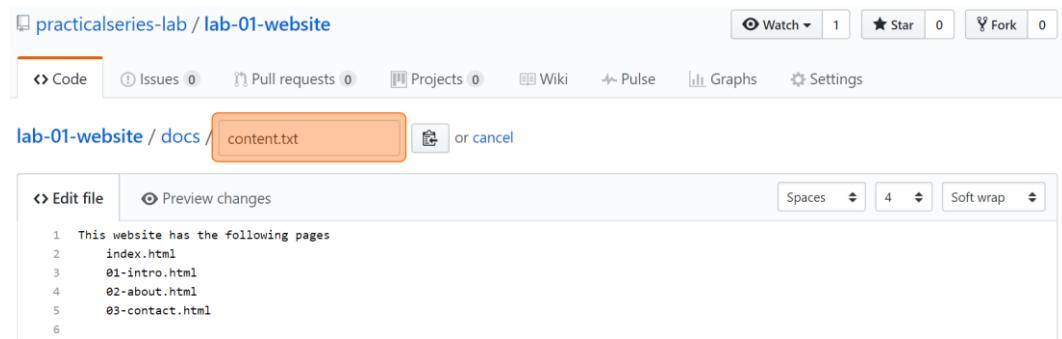


Figure 9.29 GitHub—rename a file

To rename the file just change its name in the box highlighted. Change it to `page-list.txt`.

You could also change the directory here by adding a new folder name followed by an oblique followed by the new filename (this is similar to creating a new folder in § 9.3.1—it's the same mechanism).

Rename the file and then add the commit message shown in Figure 9.30:

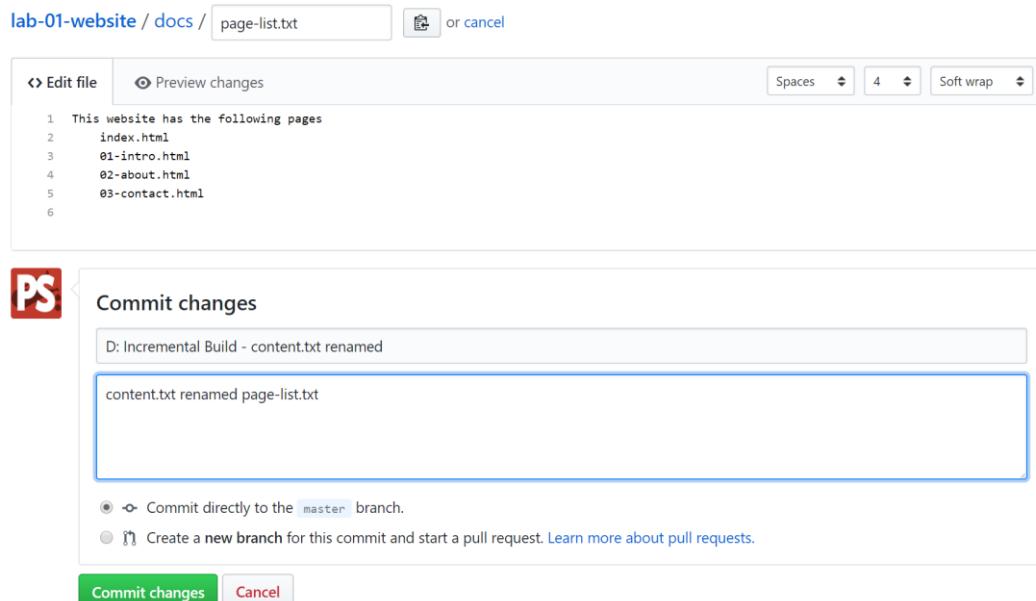


Figure 9.30 GitHub—rename a file commit message

Click **COMMIT CHANGES** and the filename has been changed; there is a new commit point [[09fc21b](#)].

Note: *There are restrictions to what you can rename—specifically it is not possible to rename files that GitHub cannot edit; the `docs` file in § 9.3.3 cannot be edited and consequently cannot be renamed (it can be deleted and re-uploaded with a different name though).*

9.3.6 Deleting a file

To demonstrate let's get rid of what is now the `page-list.txt` file—it's not much use.

Open the file in GitHub (from the repository home page click the `docs` folder and then click `page-list.txt`).

You should have this:

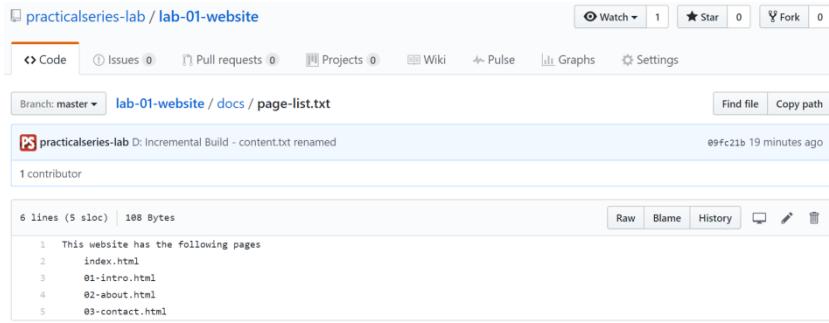


Figure 9.31 GitHub—delete a file

Now delete the file by clicking the rubbish bin symbol (trash). This takes you to the commit changes page, add the commit message:

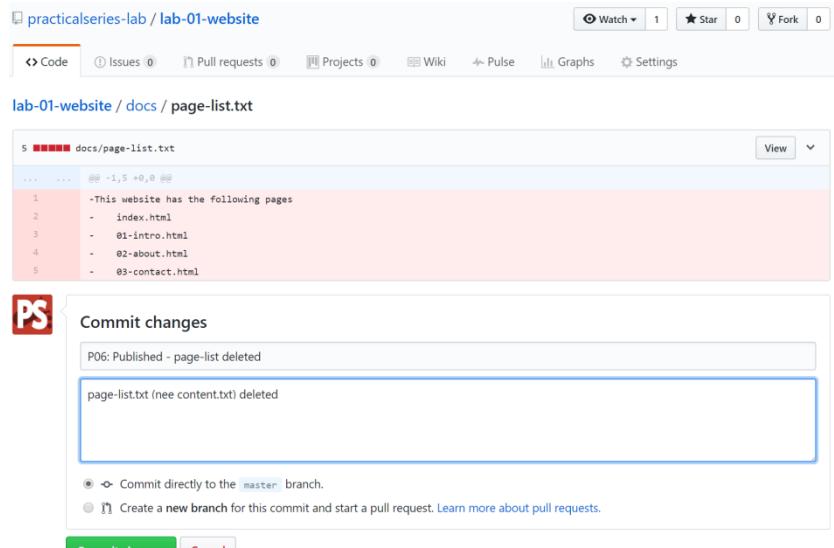


Figure 9.32 GitHub—delete a file commit message

Click **COMMIT CHANGES** and the file will be deleted; there is a new commit point [[19de35d](#)].

Note:

You can't delete folders from GitHub, the only way is to link the remote repository to a local repository, delete it there and then push the change back to GitHub.

9.4

GitHub—navigating commits and regression

Examining different commit points is very easy in GitHub (easier than it is with Brackets and definitely easier than with the Git Bash command line).

Before we start with moving back to earlier commits, let's just have a look at some features of the repository home screen that are relevant to all commit points—it will help you identify where you are:

The screenshot shows the GitHub repository page for 'practicalseries-lab / lab-01-website'. At the top, there are buttons for Watch (1), Star (0), Fork (0), and a search bar. Below the header, there are tabs for Code, Issues (0), Pull requests (0), Projects (0), Wiki, Pulse, Graphs, and Settings. A sub-header indicates it's a simple website repository used as an example to demonstrate how the Brackets-Git extension works. It shows 19 commits, 1 branch, 13 releases, and 1 contributor. The master branch is selected. There are buttons for New pull request, Create new file, Upload files, Find file, and Clone or download. The main area displays a list of commits, each with a file icon, the file name, a commit message, and a timestamp. The most recent commit is highlighted with a blue background and labeled 'Latest commit 19de35d 16 hours ago'. A point ① is placed over this commit, and a point ② is placed over the three dots next to the commit message.

File	Commit Message	Time Ago
11-resources	D: Incremental build - 03-contact page	5 days ago
docs	P06: Published - page-list deleted	16 hours ago
.gitignore	D01: Initial Commit	13 days ago
01-intro.html	D: Incremental Build - 01-intro page	12 days ago
02-about.html	D: Incremental Build - 02-about page	11 days ago
03-contact.html	D: Incremental Build - 03-contact page	6 days ago
README.md	D: Incremental Build - README.md modified	17 hours ago
index.html	P01: First Publication	12 days ago

Figure 9.33 GitHub—home page

Ok, the current commit is shown in the blue bar point ①, this is the home page so the commit number shown is the most recent commit in the whole project. The first line of the commit message and who made the commit is also shown, point ②. If you click the three dots ③ next to the commit message it will display the whole commit message:

The screenshot shows a detailed view of a GitHub commit message. The commit is from 'practicalseries-lab' on 'GitHub' and is labeled 'P06: Published - page-list deleted'. The timestamp is 'Latest commit 19de35d 17 hours ago'. Below the commit message, it says 'page-list.txt (nee content.txt) deleted'. A point ① is placed over the commit message, and a point ② is placed over the three dots next to the commit message.

Figure 9.34 GitHub—view full commit message

The **CLONE OR DOWNLOAD** button ③ is a feature of every commit point. It allows the entire project to be downloaded exactly as it was at that particular commit point.

Click **CLONE OR DOWNLOAD** to open the dropdown menu:

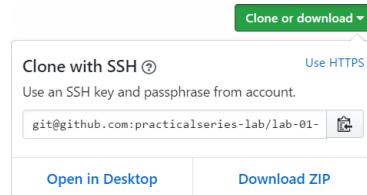


Figure 9.35 GitHub—clone and download

We've used this before, the URL in the middle is what we used to link Brackets to GitHub (§ 5.3), the bits we haven't looked at are the two buttons at the bottom **OPEN IN DESKTOP** and **DOWNLOAD ZIP**. The first **OPEN IN DESKTOP** opens the project in the desktop version of GitHub, this is the same desktop application I discuss on page 328, we're not using the application and we won't be pushing that button.

The **DOWNLOAD ZIP** is very useful. If you click it, it will download a zip file of the entire project exactly as it was at the time of the commit. This includes the full folder structure with all the files stored in their correct positions within the structure. The downloaded file is called `<repository name>-master.zip`; in my case `lab-01-website-master.zip`. If I unzip the file I get this:

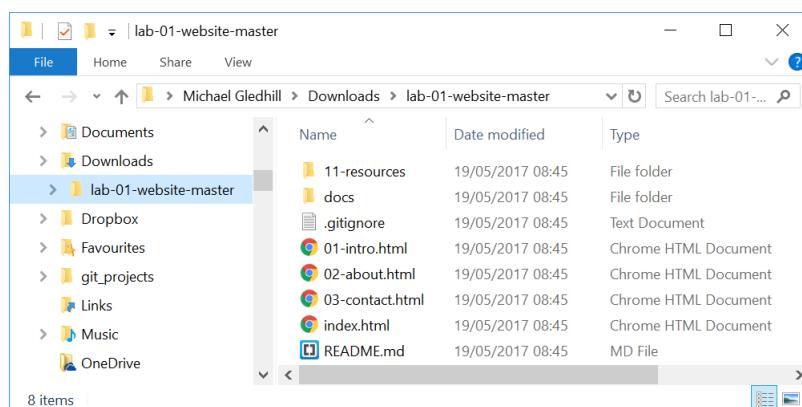


Figure 9.36 GitHub—zip file contents

Now this is a very useful facility, it gives a snapshot of the entire project at any commit; and it is very easy to use.

9.4.1 The commit history

We had a quick look at the commit history in § 8.3.2, to access it click the **COMMITS** tab in the status bar, point ④ in Figure 9.33. It opens the commit history page (I've only shown part of it here):

The screenshot shows the GitHub commit history for the **master** branch. At the top left, there is a dropdown menu labeled "Branch: master". Below it, two sections of commits are listed: "Commits on May 18, 2017" and "Commits on May 15, 2017". Each commit entry includes a small profile icon, the commit message, the author, the date, a copy-to-clipboard button, the commit hash, and a diff button.

- May 18, 2017:**
 - P06: Published - page-list deleted ... practicalseries-lab committed on GitHub 17 hours ago (hash: 19de35d)
 - D: Incremental Build - content.txt renamed ... practicalseries-lab committed on GitHub 18 hours ago (hash: 09fc21b)
 - D: Incremental Build - README.md modified ... practicalseries-lab committed on GitHub 18 hours ago (hash: 5644edc)
 - P05: Published - Version Numbering doc added ... practicalseries-lab committed on GitHub 19 hours ago (hash: 180bd32)
 - D: Incremental Build - /docs/content.txt added ... practicalseries-lab committed on GitHub 20 hours ago (hash: 64478f4)
- May 15, 2017:**
 - P04: Published (merged d-03-contact) practicalseries-lab committed 4 days ago (hash: da038e3)

Figure 9.37 GitHub—commit history page

Up at the top, we can see that this is the commit history for the **master** branch (I currently only have the **master** branch—if I had more branches I would be able to select them with the **BRANCH** button and see the commit history for that branch).

Again I can see the first line of the commit message for each commit, who made it and when; clicking the three dots after commit message will show the full commit message (just like on the home page).

On the left we have three buttons:



Figure 9.38 GitHub—commit page buttons

The first button, the clipboard (📋) copies the full 40 digit commit (*hash*) number to the clipboard.

The second button (with the seven digit commit number in it) takes you to an information page for that commit (this is similar to the commit information page in Brackets, § 6.3.3. It looks like this:

D: Incremental Build - README.md modified

README.md modified to include document references

master

PS practicalseries-lab committed on GitHub 18 hours ago

Showing 1 changed file with 2 additions and 0 deletions.

Unified Split

2 README.md

```

@@ -11,6 +11,8 @@
 11 11 ## How to use this repository
 12 12 This repository is a worked example demonstrating how to build a version control project using Git and GitHub from within the
 13 13 Brackets text editor.
 14 14 +
 15 15 +There is a version numbering document within the /docs folder that explains how version numbering can be implemented on a GitHub
 16 16 project.
 17 17 ## Website contents
 18 18 This site contains the following pages:

```

0 comments on commit 5644edc

Lock conversation

Write Preview

Leave a comment

Attach files by dragging & dropping, selecting them, or pasting from the clipboard.

Styling with Markdown is supported

Comment on this commit

Figure 9.39 GitHub—commit information page

The browse file button at the top does exactly the same as clicking the icon in Figure 9.38 and I cover this below in the next section.

Looking around the screen, the bit in the middle shows the changes to the file, lines 14 and 15 were added (that's why they are in green). This is showing the file in the unified mode, see the **UNIFIED** button is pressed (just under the blue bar on the right). If you change to split view, you see the version of the file at this commit alongside the version prior to this commit:

The screenshot shows a GitHub commit information split view for a file named README.md. The interface is divided into two main sections: the left side shows the original code and the right side shows the modified code. The commit message at the top indicates changes from line 11 to 18. The right side shows the addition of a new section '## Version numbering' with a detailed explanation. The GitHub UI includes standard navigation buttons like back, forward, and search.

```

@@ -11,6 +11,8 @@ The full set of PracticalSeries publications is available at [practicalseries.co
11 ## How to use this repository
12 This repository is a worked example demonstrating how to build a
version control project using Git and GitHub from within the Brackets
text editor.
13 This repository is intended to be used with the accompanying
documentation [practicalseries Git and GitHub]
(http://practicalseries.com/0021-git-vcs/index.html "Practical Series -
Git and GitHub").
14 +
15 +There is a version numbering document within the /docs folder that
explains how version numbering can be implemented on a GitHub project.
16 ## Website contents
17 This site contains the following pages:
18

```

Figure 9.40 GitHub—commit information split view

The three buttons just above the file show the file in different ways: the first () shows the source code, the second () shows a preview and the **VIEW** button displays the whole file in preview mode.

Making additional commit comments

GitHub allows additional commit comments to be made on any commit in the repository; these can even be attached to an individual line in the source code. Go back to the unified view of the file: and hover the mouse over the number for line 15:

The screenshot shows the unified view of the README.md file on GitHub. A cursor is hovering over the line number 15, which triggers a small blue plus sign icon. Clicking this icon opens an inline message box where a user can type a comment. The commit message at the top indicates changes from line 11 to 18. The right side shows the addition of a new section '## Version numbering' with a detailed explanation.

```

@@ -11,6 +11,8 @@ The full set of PracticalSeries publications is available at [practicalseries.co
11 11 ## How to use this repository
12 12 This repository is a worked example demonstrating how to build a
version control project using Git and GitHub from within the Brackets
text editor.
13 13 This repository is intended to be used with the accompanying documentation [practicalseries Git and GitHub]
(http://practicalseries.com/0021-git-vcs/index.html "Practical Series -
Git and GitHub").
14 +
15 +There is a version numbering document within the /docs folder that
explains how version numbering can be implemented on a GitHub
project.
16 16 ## Website contents
17 17 This site contains the following pages:
18

```

Figure 9.41 GitHub—inline commit comment

It will produce a small blue plus sign, click the plus sign and it will open an inline message box, add the message shown:

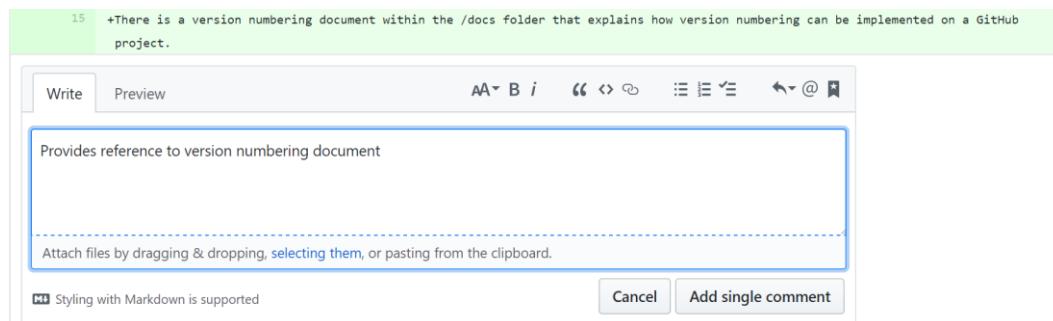


Figure 9.42 GitHub—enter inline comment

Click the **ADD SINGLE COMMENT** button and it will show in the source code file:

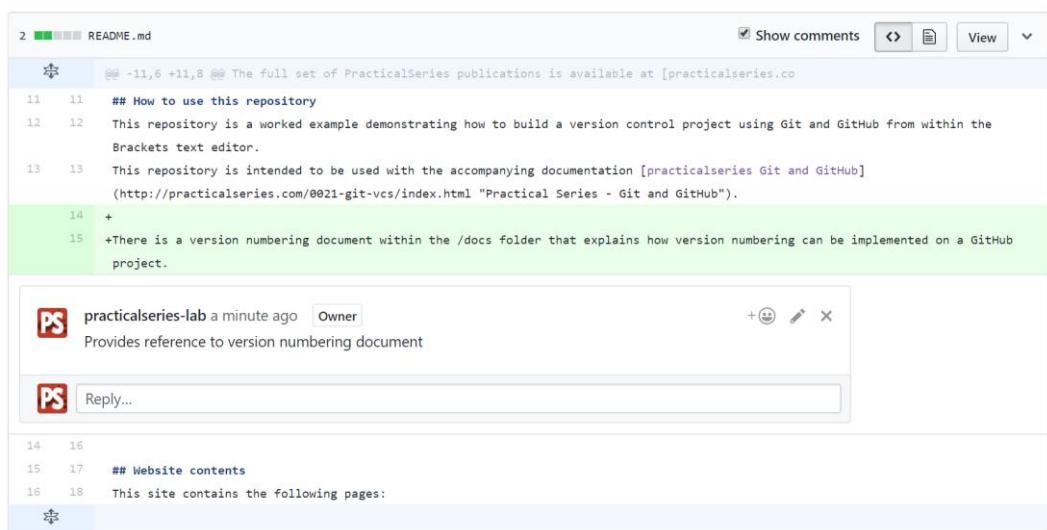


Figure 9.43 GitHub—inline commit comment – reply, edit and delete

A reply can be given to the message (type it in the reply area), the message itself can be edited (pencil symbol) or deleted (cross) or an emoticon can be added if you are a teenager (smiley face symbol).

Note:

There is just one rule for emoticons:

Don't. Ever.

Engineers in particular are not at home to emoticons.



As well as inline comments, an additional general comment for the whole commit can be added in the box at the bottom of the screen:

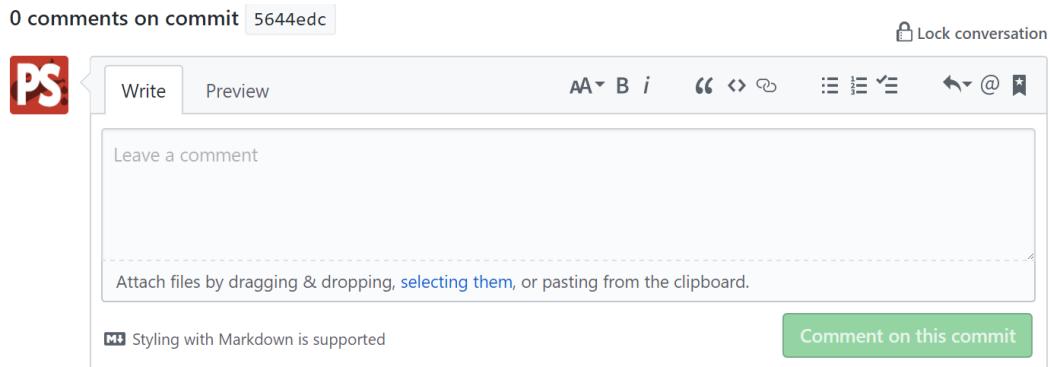


Figure 9.44 GitHub—general commit comment

Enter whatever message you want in the box and click the [COMMENT ON THIS COMMIT](#) button.

Note: Both inline and general commit comments support markdown (like the `README.md` file).

If a commit comment is present (either inline or general), this will be indicated when viewing the commit history, thus:

Commits on May 18, 2017	
	P06: Published - page-list deleted ... practicalseries-lab committed on GitHub 2 days ago
	D: Incremental Build - content.txt renamed ... practicalseries-lab committed on GitHub 2 days ago
	D: Incremental Build - README.md modified ... practicalseries-lab committed on GitHub 2 days ago
	P05: Published - Version Numbering doc added ... practicalseries-lab committed on GitHub 2 days ago

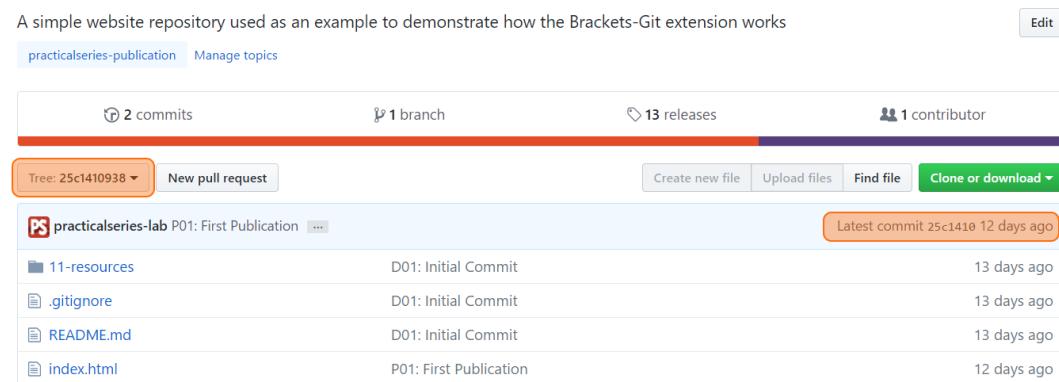
Figure 9.45 GitHub—commit comment indicator

I've highlighted the commit comment indicator in orange. The number indicates how many comments have been made.

9.4.2 Recovering data from an earlier commit

Back to the commit history page (Figure 9.37). The final button that we haven't looked at yet is the **BROWSE REPOSITORY AT THIS POINT IN HISTORY** button the  icon.

Click it for one of the early commits (the 2nd one, P01) in my case [\[25c1410\]](#). It looks like this:



A screenshot of a GitHub repository page. At the top, it says "A simple website repository used as an example to demonstrate how the Brackets-Git extension works". Below that are buttons for "Edit", "practicalseries-publication", and "Manage topics". A red bar highlights the "2 commits" button. Below the bar, there are four summary statistics: "2 commits", "1 branch", "13 releases", and "1 contributor". A dropdown menu shows "Tree: 25c1410938" and "New pull request". To the right are buttons for "Create new file", "Upload files", "Find file", and "Clone or download". A blue box highlights the "Clone or download" button. Below these are the commit details:

File	Commit Message	Date
11-resources	D01: Initial Commit	13 days ago
.gitignore	D01: Initial Commit	13 days ago
README.md	D01: Initial Commit	13 days ago
index.html	P01: First Publication	12 days ago

Latest commit 25c1410 12 days ago

Figure 9.46 GitHub—moving to an earlier commit point

The important thing is that we are now looking at the project at an earlier commit point, this can be seen in the blue bar and in what was the branch button (highlighted), in my case commit point [\[25c1410\]](#). The branch button is now called a **TREE** button, this is the similar to the way we did a checkout in Brackets (§ 7.3).

The **master** branch is available on the **TREE** button dropdown and selecting it will move back to the most recent commit (again just like the checkout function in Brackets § 7.3).

We can also see that there are fewer files available, the [01-intro](#), [02-about](#) and [03_contact](#) HTML files are missing.

Clicking any of the files will open it in the state it was at the time of the commit; similarly clicking **CLONE OR DOWNLOAD** will allow a zipped copy of the project at the time of the commit (complete with the correct folder structure) to be downloaded.

The zipped file will be given the name of the repository followed by the full 40 digit commit number.

In my case this is:

[lab-01-website-25c1410938e16af385ac126c40f71db154f836a5.zip](#)

Unzipping the file gives:

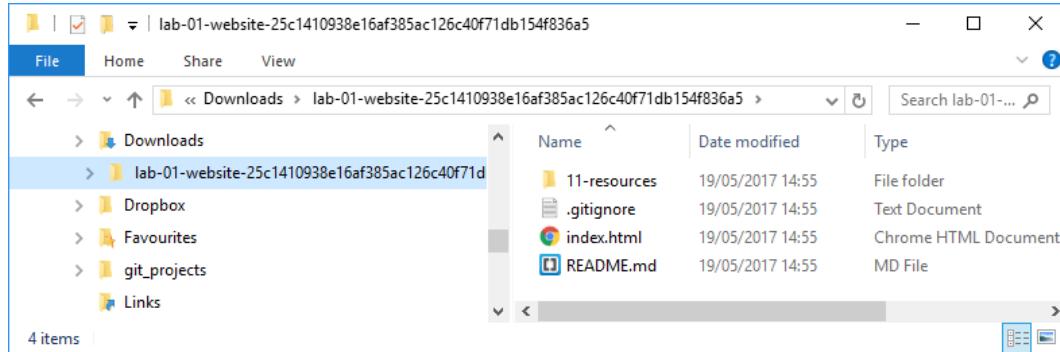


Figure 9.47 GitHub—earlier commit point zip file contents

Again the zip file contains the files and folders in exactly the state they were at the time of the commit.

This is a very convenient way of recovering data from an earlier commit point.

When we did this in Brackets with the reset we had a problem of getting back to the most recent commit. This is not a problem here, to get back to the most recent version of the project, return to the repository home page by clicking the `lab-01-website` name at the top (highlighted below):



Figure 9.48 GitHub—return to most recent commit point

9.5

GitHub—branches. Merging and conflicts

GitHub provides facilities for creating, merging and deleting branches. It also has the capability of resolving conflicts that occur in the merging process.

Just to clarify a point, the merging process on GitHub is started by triggering a *pull request*. You may have seen pull requests mentioned on the GitHub website (and in virtually every piece of documentation that goes with it); there is a button and status for it on the repository home page.

The pull request originates in the communal working environment and it can be interpreted as a *request to pull modifications into the repository*. I cover it in terms of merging branches in this section (§ 9.5.3) and I cover it as a communal coding activity in section 10.

Where people refer to a pull request, they are generally referring to a process of merging modifications back into the repository.

First things:

9.5.1 Creating a new branch

By way of example, I will add a new legal page to the website ([04-legal.html](#)) in much the same way I did with other pages in previous examples. I will do this by creating a new branch adding the page to the new branch and then merging it back into the **master** branch.

The easiest way to create a new branch is from the repository home page. Open the [lab-01-website](#) home page:

The screenshot shows the GitHub repository home page for 'practicalseries-lab'. At the top, there are statistics: 19 commits, 1 branch, 13 releases, and 1 contributor. Below this, a navigation bar includes 'Branch: master ▾' (which is highlighted with a red box), 'New pull request', 'Create new file', 'Upload files', 'Find file', and 'Clone or download ▾'. The main area displays a list of commits. The first commit is from 'PS practicalseries-lab' on 'GitHub P06: Published - page-list deleted' at 'Latest commit 19de35d 2 days ago'. Other commits include '11-resources' (6 days ago), 'docs' (2 days ago), '.gitignore' (14 days ago), '01-intro.html' (13 days ago), '02-about.html' (12 days ago), '03-contact.html' (7 days ago), 'README.md' (2 days ago), and 'index.html' (13 days ago).

Figure 9.49 GitHub—repository home page

Click the **BRANCH** button to open the branch dropdown box (Figure 9.50):

The screenshot shows the GitHub branch dropdown box. It has a header with 'Branch: master ▾' and 'New pull request'. Below this is a search bar labeled 'Find or create a branch...'. Underneath the search bar are tabs for 'Branches' and 'Tags', with 'Branches' currently selected. A blue checkmark next to 'master' indicates it is the active branch.

Figure 9.50 GitHub—branch dropdown box unpopulated

The screenshot shows the GitHub branch dropdown box after a new branch is being created. The search bar now contains 'd-04-legal'. Below the search bar, there are tabs for 'Branches' and 'Tags'. A large blue button at the bottom says '>Create branch: d-04-legal from 'master''. There is also a small 'X' icon in the top right corner of the dropdown box.

Figure 9.51 GitHub—branch dropdown box create branch

In the **FIND OR CREATE A BRANCH** box start typing the name of the branch to be created, in this case it is **d-04-legal**.

If the branch already exists, GitHub will try to switch to it (hence the find), in our case the branch does not exist and GitHub is telling us it will create a new branch called **d-04-legal** from the latest commit on the **master** branch (the blue bit at the bottom, Figure 9.51). Hit **RETURN** to create the branch; GitHub will open the branch page for **d-04-legal**:

A simple website repository used as an example to demonstrate how the Brackets-Git extension works

practicalseries-publication Manage topics

19 commits 2 branches 13 releases 1 contributor

Branch: d-04-legal New pull request Create new file Upload files Find file Clone or download

This branch is even with master.

	practicalseries-lab committed on GitHub P06: Published - page-list deleted	... Latest commit 19de35d 2 days ago
	11-resources D: Incremental build - 03-contact page	6 days ago
	docs P06: Published - page-list deleted	2 days ago
	.gitignore D01: Initial Commit	14 days ago
	01-intro.html D: Incremental Build - 01-intro page	13 days ago
	02-about.html D: Incremental Build - 02-about page	12 days ago
	03-contact.html D: Incremental Build - 03-contact page	7 days ago
	README.md D: Incremental Build - README.md modified	2 days ago
	index.html P01: First Publication	13 days ago

Figure 9.52 GitHub—**d-04-legal** home page

The **BRANCH** button is now telling us we are on a new branch called **d-04-legal**. The commit number has not changed (we are still pointing to the latest commit on the **master** branch, see Figure 9.49).

Note: *Although we are now on a new branch, the default branch (GitHub has a default branch) is still the **master** branch. This means that if you return to the repository home page, GitHub will transfer you back to the **master** branch (the default branch) and you will have to reselect the new branch if you want it.*

It will catch you out (at least it does me) and you will find yourself on the wrong branch. It is possible to change the default branch, see § 9.5.6.

That's it; we've created a new branch and switched to it.

9.5.2 Creating a file on the new branch

From the **d-04-legal** branch page, click the **CREATE NEW FILE** button.

Call the new file **04-legal.html** and add the following code:

```
04-LEGAL.HTML

1 <html lang="en">                                <!-- Declare language -->
2   <head>                                         <!-- Start of head section -->
3     <meta charset="utf-8">                      <!-- Use unicode character set -->
4
5     <link rel="stylesheet" type="text/css" href="11-resources/01-css/style.css">
6
7   <title>PracticalSeries: Git Lab - legal</title>
8 </head>
9
10 <body>
11   <h1>A Practical Series Website</h1>
12
13   <h3> Legal Declaration </h3>
14
15   <p>This page contains the legal information for the website.</p>
16
17 </body>
18 </html>
```

Code 9.2 **d-04-legal** branch—create new file: 04-legal.html

And give it the commit message:

D: Incremental Build - 04-legal page

04-legal.html added.

It should all look like Figure 9.53:

The screenshot shows the GitHub interface for a repository named 'lab-01-website'. In the code editor, a new file '04-legal.html' is being created. The code content is:

```

1 <html lang="en">
2   <head>
3     <meta charset="utf-8">
4
5     <link rel="stylesheet" type="text/css" href="11-resources/01-css/style.css">
6
7     <title>PracticalSeries: Git Lab - legal</title>
8   </head>
9
10  <body>
11    <h1>A Practical Series Website</h1>
12
13    <h3> Legal Declaration </h3>
14
15    <p>This page contains the legal information for the website.</p>
16
17  </body>
18 </html>

```

In the commit dialog, the message 'D: Incremental Build - 04 legal page' is entered, and the file '04-legal.html' is listed as added. There are two options for committing:

- Commit directly to the `d-04-legal` branch.
- Create a new branch for this commit and start a pull request. [Learn more about pull requests.](#)

At the bottom are 'Commit new file' and 'Cancel' buttons.

Figure 9.53 GitHub—add a page to the `d-04-legal` branch

Click **COMMIT NEW FILE** to add the file to the repository. This takes us back to the branch page Figure 9.54.

The new commit point is, in my case [\[9c7954b\]](#); the new file [04-legal.html](#) is also visible.

A simple website repository used as an example to demonstrate how the Brackets-Git extension works

practicalseries-publication Manage topics

20 commits 2 branches 13 releases 1 contributor

Your recently pushed branches:

d-04-legal (less than a minute ago) Compare & pull request

Branch: d-04-legal New pull request Create new file Upload files Find file Clone or download ▾

This branch is 1 commit ahead of master.

PS practicalseries-lab committed on GitHub D: Incremental Build - 04 legal page ... Latest commit 9c7954b a minute ago

File	Commit Message	Time Ago
11-resources	D: Incremental build - 03-contact page	6 days ago
docs	P06: Published - page-list deleted	2 days ago
.gitignore	D01: Initial Commit	14 days ago
01-intro.html	D: Incremental Build - 01-intro page	13 days ago
02-about.html	D: Incremental Build - 02-about page	12 days ago
03-contact.html	D: Incremental Build - 03-contact page	7 days ago
04-legal.html	D: Incremental Build - 04 legal page	a minute ago
README.md	D: Incremental Build - README.md modified	2 days ago
index.html	P01: First Publication	13 days ago

Figure 9.54 GitHub—d-04-legal branch page with new file

The yellow bar informs us that the **d-04-legal** branch has just been pushed to the repository—technically it hasn't been pushed, we did in GitHub directly in the remote repository.

There is also a message just above the blue commit line; it says this “*branch is 1 commit ahead of the master*”, this all makes sense.

Wherever you look there are also prompts for **PULL REQUESTS**. Let's look at that next.

9.5.3 Merging branches with a [PULL REQUEST](#)

GitHub uses (by and large) public repositories—all the repositories we've looked at so far have been public repositories. This means that anyone can see them and can copy them.

People can even copy them with the intention of developing them further (this process has the rather vulgar name of *forking*¹ and I discuss how it works in § 10.1). Now, it is generally difficult to stop people doing this and GitHub (I think) actively encourages it (it is possible to block users, but this is intended only for abusive users and not just to stop people looking—it would be difficult to block every user of GitHub).

Clearly, though we don't want every Tom, Dick or Harry modifying our code—hence the pull request. This allows some other user to take your code, modify it in their own account—and, if they think it's good enough, send you a pull request.

The pull request alerts you to the fact that someone has made a modification and they are *requesting* that you *pull* their modification in to the repository. Only the owner (or those users given write permission to the repository) can grant this pull request and merge the modification into the repository.

Now in our example here, I (the owner of the repository) made a change to a particular branch; there has been no forking of the repository, no one has copied it and there is no communal working—just me adding a file. So why, you may wonder, is it banging on about pull requests?

The answer is it uses the same process no matter where the change originated (be it some other user who had copied, *forked*, the repository or me, the owner of the repository, making a change). So we use a pull request to merge the files.

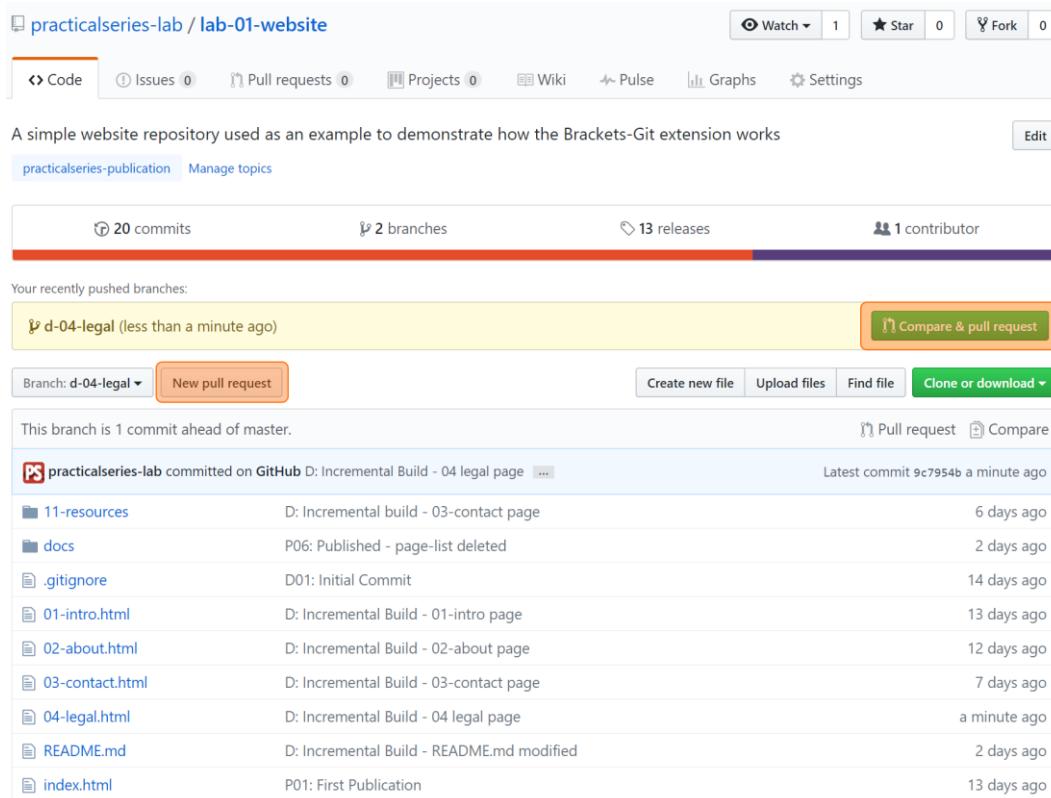
Let's do it.

¹

I generally don't like the names that Git and GitHub use, I think them all coarse, crude and vulgar; but then I'm English and we're taught good manners from an early age.

Creating a pull request

From the branch page click either the **COMPARE AND PULL REQUEST** button in the yellow bar or the **NEW PULL REQUEST** button next to the **BRANCH** button. They both do exactly the same:



The screenshot shows a GitHub repository page for 'practicalseries-lab / lab-01-website'. The top navigation bar includes 'Watch 1', 'Star 0', 'Fork 0', and tabs for 'Code', 'Issues 0', 'Pull requests 0', 'Projects 0', 'Wiki', 'Pulse', 'Graphs', and 'Settings'. Below the navigation is a brief description: 'A simple website repository used as an example to demonstrate how the Brackets-Git extension works'. A 'Edit' button is on the right. Underneath, there's a 'practicalseries-publication' topic and a 'Manage topics' link. A summary bar shows '20 commits', '2 branches', '13 releases', and '1 contributor'. A note says 'Your recently pushed branches:' followed by a list: 'd-04-legal (less than a minute ago)'. A prominent orange button labeled 'Compare & pull request' is highlighted with a red box. Below this, a dropdown menu shows 'Branch: d-04-legal' and a 'New pull request' button, also highlighted with a red box. A toolbar at the bottom includes 'Create new file', 'Upload files', 'Find file', and 'Clone or download'. At the very bottom, it says 'This branch is 1 commit ahead of master.' with 'Pull request' and 'Compare' buttons.

Figure 9.55 GitHub—**d-04-legal** initiate a merge (pull request)

This takes us to the open pull request page Figure 9.56 (I haven't shown all of the **04-legal.html** file, the stuff in green—there was too much of it):

Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#).

The screenshot shows the GitHub interface for opening a pull request. At the top, it says "base: master" and "compare: d-04-legal". A green checkmark indicates "Able to merge". The main area shows a "Write" tab selected, with a preview window showing the file "04-legal.html" has been added. To the right, there are sections for "Reviewers", "Assignees", "Labels", "Projects", and "Milestone", all currently set to "None yet". A "Create pull request" button is at the bottom. Below this, the pull request details are shown: 1 commit, 1 file changed, 0 commit comments, and 1 contributor. The commit log shows a single commit from "practicalseries-lab" on May 20, 2017, titled "D: Incremental Build - 04 legal page". The diff view shows the addition of "04-legal.html" with 18 additions and 0 deletions. The diff content includes the declaration of language as English, the start of the head section, and the use of UTF-8 encoding. A note below the diff says "No commit comments for this range".

Figure 9.56 GitHub—open a pull request page

The important bit is at the top:

This screenshot shows the "pull request details" page for the same pull request as Figure 9.56. It displays the base branch ("master"), compare branch ("d-04-legal"), and the fact that the branches are "Able to merge". The commit log and diff view are identical to the previous figure, showing one commit from "practicalseries-lab" on May 20, 2017, titled "D: Incremental Build - 04 legal page". The diff view shows the addition of "04-legal.html" with 18 additions and 0 deletions.

Figure 9.57 GitHub—pull request details

The base branch (**master**) is the branch to which the changes will be merged (the receiving branch as it were), the compare branch (**d-04-legal**) is the branch we are trying to merge from.

The bit following this, the **Able to merge** message is important, it tells us that there are no conflicts between the two branches (the branches can merge without issue).

There is not really anything else to add, the bit underneath, next to the user logo allows a message to be entered, by default it uses the commit message that was entered when the change was committed—add whatever additional information you require (in a pull request from another user, this would be where they explain what the change is and why it would be useful).

I'm just going to leave it with the default content. Click **CREATE PULL REQUEST**:

Reviewing the pull request

After creating a pull request, we're taken to the pull request status page:

D: Incremental Build - 04 legal page #1

Open practicalseries-lab wants to merge 1 commit into `master` from `d-04-legal`

Conversation 0 Commits 1 Files changed 1 +18 -0

PS practicalseries-lab commented just now
04-legal.html added

PS D: Incremental Build - 04 legal page ... 9c7954b

Add more commits by pushing to the `d-04-legal` branch on [practicalseries-lab/lab-01-website](#).

This branch has no conflicts with the base branch
Merging can be performed automatically.

Merge pull request You can also open this in GitHub Desktop or view command line instructions.

Write Preview AA B i Leave a comment Attach files by dragging & dropping, selecting them, or pasting from the clipboard. Styling with Markdown is supported

Close pull request Comment

Notifications Unsubscribe You're receiving notifications because you authored the thread.

1 participant PS

Figure 9.58 GitHub—pull request status page

At this point absolutely nothing has happened.

PULL REQUESTS

Pull requests are just that, a request absolutely nothing happens to the contents of the repository when a pull request is generated.

A pull request has been generated, but nothing in the repository has changed.

Making a pull request takes us to the pull request status page (Figure 9.58). The first thing to note is at the top (highlighted), we are in the pull request tab (we haven't been here before). This is where all the open pull requests (wherever they were generated) are listed.

We just have one (hence the one in the tab), the one we generated in the last section.

There are three things we can do with a pull request:

- ① Reject it (hit the **CLOSE PULL REQUEST** button)
- ② Ask a question (enter text in the box and hit the **COMMENT** button)
- ③ Accept it (hit the **MERGE PULL REQUEST** button)

Looking at these in turn:

Rejecting a pull request

To reject a pull request click the **CLOSE PULL REQUEST** button, this is listed a point ① in Figure 9.58. Closing a pull request does not delete it, it will be listed under closed pull requests, but it will stop it bothering you and it will not require any further interaction.

Note: It is not possible to delete pull requests, they hang around as closed pull requests, they are not generally visible, but you can search for them.

A note of caution: closing pull requests can upset some people, some people don't handle rejection well—best to let them down gently—as someone once said “never argue with strangers on the internet”.

Starting a pull request conversation

Next: the conversation. If you notice, we are actually in a sub-tab called **CONVERSATION**, it's just under the green open image (below the D: Incremental Build line).

If you're not sure about the pull request you can ask questions or make a statement “*good idea, I'll implement later*”—or in my case “*who are you? Stop bothering me. Go away*”; type what you want in the box at the bottom and click the **COMMENT** button (both labelled point ② in Figure 9.58) and the message will be added to the pull request listing.

Accepting a pull request

The final option is to accept the changes. Click the **MERGE PULL REQUEST** button point ③ in Figure 9.58.

Note: Only users who are able to commit changes to the repository can do this.

The **MERGE PULL REQUEST** button has various options:

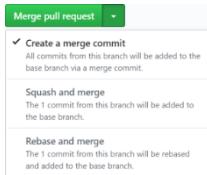


Figure 9.59 GitHub—pull request merge options

We want the default option, **CREATE A MERGE COMMIT**. This is in keeping with my idea of the best practice for merging branches. Generally, the top option is the correct one (and it certainly won't do any harm).

Clicking **MERGE PULL REQUEST** automatically generates a merge commit message (you can alter it if you want):

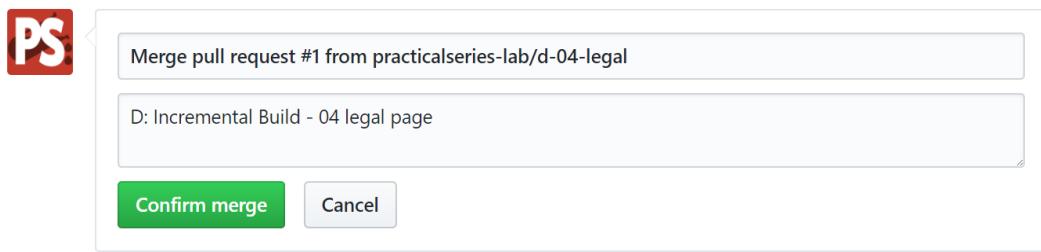


Figure 9.60 GitHub—pull request merge commit message

I'm going with the default, click **CONFIRM MERGE**.

This still leaves us on the pull request status page:

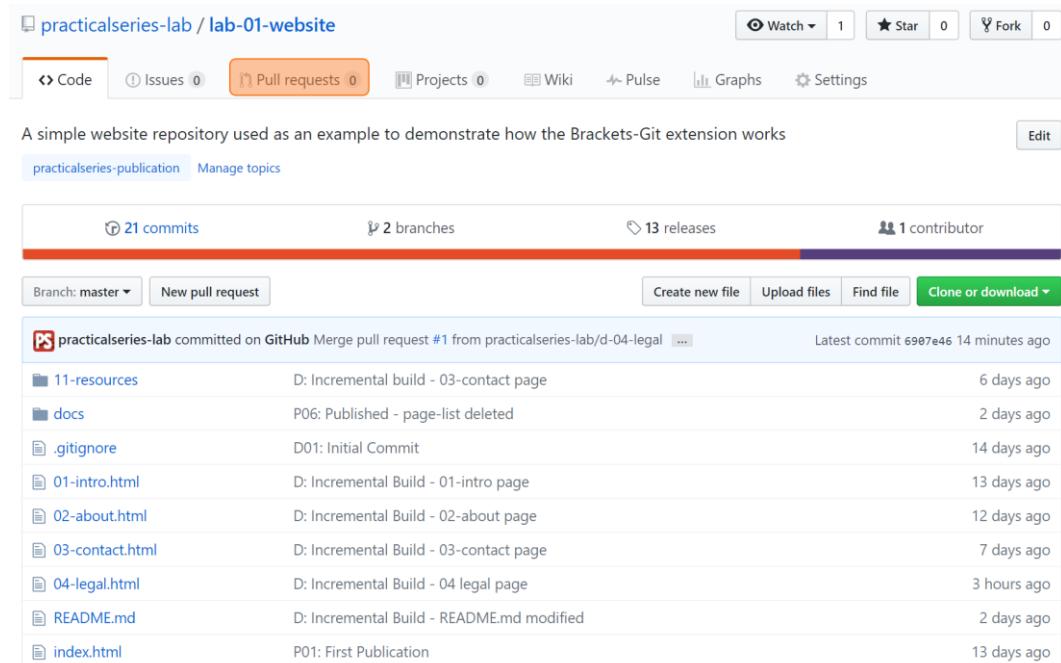
Figure 9.61 GitHub—pull request status (post merge)

We can now see the merged pull request in the pull request chain, it is telling us (next to the purple merge icon ) that the new commit is [6907e46]. At this stage the commit has been made and is in the repository.

However, GitHub (unlike Git and Brackets) has an undo button, the **REVERT** button will undo the change and take us back to the previous state (Figure 9.58). With GitHub we get to change our mind.

9.5.4 Examining previous pull requests

We've merged a branch by using a pull request (merged the **d-04-legal** branch on to the **master** branch), we can see this if we open the repository home page; it shows the latest version:



A simple website repository used as an example to demonstrate how the Brackets-Git extension works

Branch: master ▾ New pull request Create new file Upload files Find file Clone or download ▾

File	Commit Message	Time Ago
11-resources	D: Incremental build - 03-contact page	6 days ago
docs	P06: Published - page-list deleted	2 days ago
.gitignore	D01: Initial Commit	14 days ago
01-intro.html	D: Incremental Build - 01-intro page	13 days ago
02-about.html	D: Incremental Build - 02-about page	12 days ago
03-contact.html	D: Incremental Build - 03-contact page	7 days ago
04-legal.html	D: Incremental Build - 04 legal page	3 hours ago
README.md	D: Incremental Build - README.md modified	2 days ago
index.html	P01: First Publication	13 days ago

Figure 9.62 GitHub—repository home page (post merge)

The **master** branch is now selected, the new **04-legal.html** page is there and the latest commit is [6907e46]. There are still two branches (**BRANCHES** tab), this is because we haven't deleted **d-04-legal**. The **PULL REQUEST** tab is also showing no requests.

Click the **PULL REQUEST** tab to open the pull request page:

The screenshot shows the GitHub interface for the repository 'practicalseries-lab / lab-01-website'. The 'Pull requests' tab is selected. A search bar at the top contains the query 'is:pr is:open'. Below the search bar, there are buttons for 'Labels' and 'Milestones'. A green button labeled 'New pull request' is visible on the right. A filter bar shows '0 Open' and '1 Closed' (which is highlighted with an orange border). The main content area displays a message: 'There aren't any open pull requests.' followed by a note: 'Use the links above to find what you're looking for, or try a new search query. The Filters menu is also super helpful for quickly finding issues most relevant to you.'

Figure 9.63 GitHub—pull request page (post merge)

It is showing that there are no open commit requests and one closed request, the one we just made.

Clicking the closed tab (highlighted) will show the current status of the closed pull requests:

This screenshot shows the same GitHub interface as Figure 9.63, but with a different search query in the search bar: 'is:pr is:closed'. The '1 Closed' filter is still highlighted with an orange border. The main content area now lists a single closed pull request: 'D: Incremental Build - 04 legal page' with the note '#1 by practicalseries-lab was merged 26 minutes ago'.

Figure 9.64 GitHub—closed pull requests

If you click the pull request itself (where it says D: Incremental Build) it will show you the details of the pull request, this is the same as Figure 9.61. The **REVERT** option is still there if you want to undo the pull request.

9.5.5 Pull request to merge a branch with a conflict

The **d-04-legal** branch still exists, go back to the repository home page (Figure 9.62), click the branch button and select **d-04-LEGAL** from the drop down to switch to that branch.

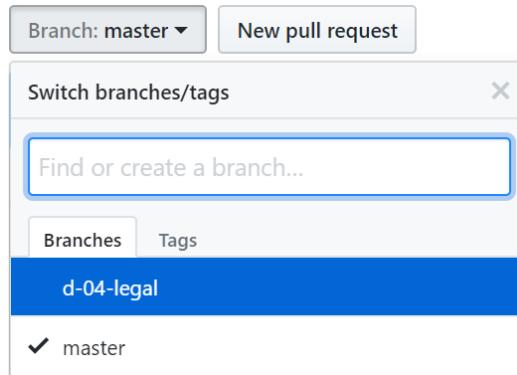


Figure 9.65 GitHub—switch to **d-04-legal** branch

Open the **11-resources** folder and then the **01-css** folder and finally open the **style.css** file:

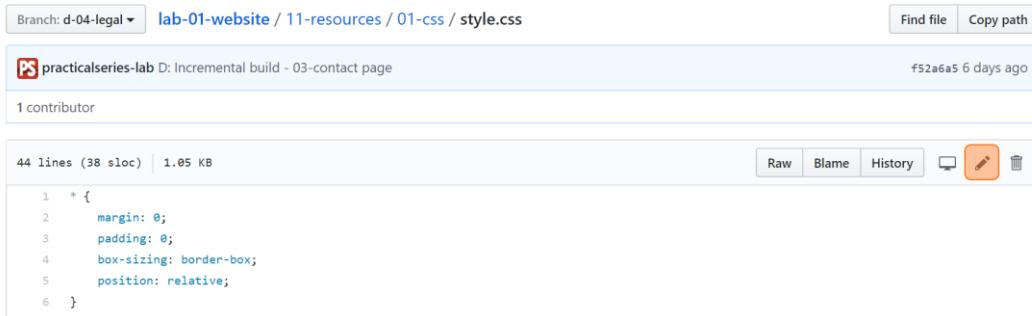


Figure 9.66 GitHub—style.css preview

Click the pencil symbol (✎) to edit the file and modify line 30 as follows:

```
30 h3 { font-size: 2.5rem; color: #404030;}
```

It gives the **h3** heading a grey colour.

The edited file looks like this, enter the commit message as shown:

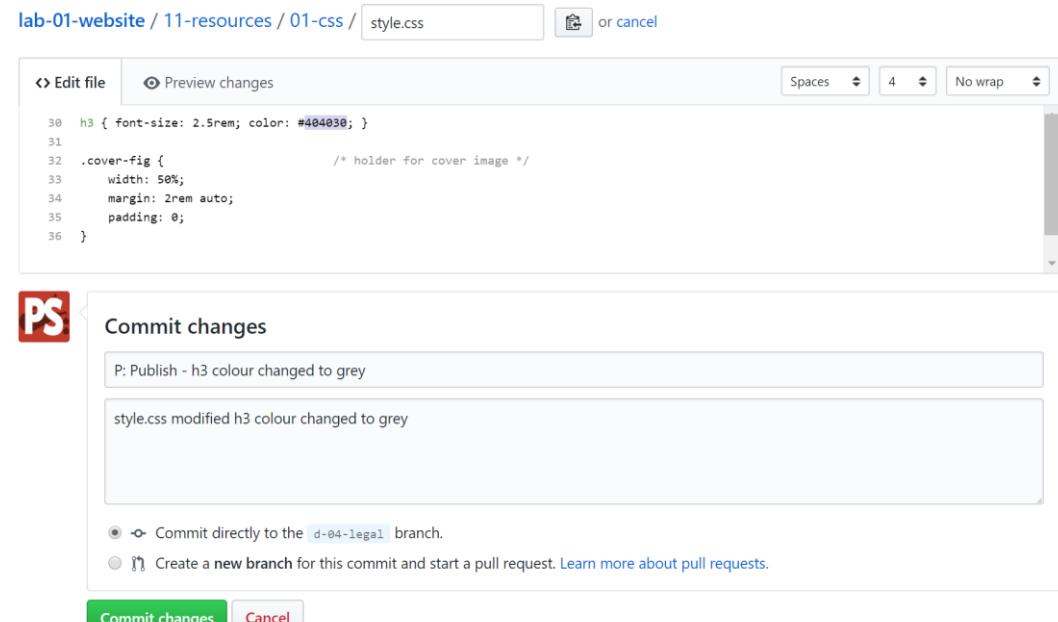


Figure 9.67 GitHub—style.css edited

Click **COMMIT CHANGES** and then go back to the **d-04-legal** branch page. It will once again inform us that **d-04-legal** is one commit ahead of **master**:



Figure 9.68 GitHub—style.css edited on d-04-legal

The next thing is to modify the **master** branch in just the same way to make a conflict (i.e. change the **style.css** file).

Go back to the repository home page and make sure you are on the **master** branch.

Drill down to the **style.css** file open it and edit it (in just the same way we did above for the **d-04-legal** branch).

This time make line 30 read:



The `h3` element is completely black, it looks like this:

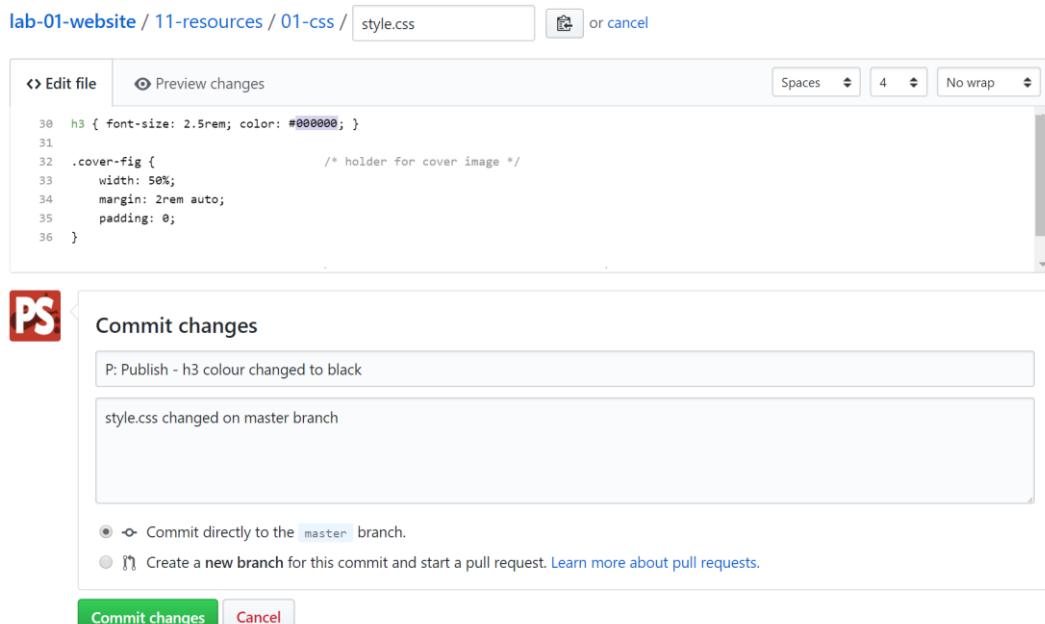


Figure 9.69 GitHub—style.css edited on master

Enter the commit message and **COMMIT CHANGES**.

Go back to the repository home page:

Your recently pushed branches:

Branch	Commit Message	Time Ago
d-04-legal	(18 minutes ago)	

Branch: master ▾ New pull request Create new file Upload files Find file Clone or download ▾

practicalseries-lab committed on GitHub P06: Published - page-list deleted ... Latest commit 19de35d 2 days ago

File	Description	Time Ago
11-resources	D: Incremental build - 03-contact page	6 days ago
docs	P06: Published - page-list deleted	2 days ago
.gitignore	D01: Initial Commit	14 days ago
01-intro.html	D: Incremental Build - 01-intro page	13 days ago
02-about.html	D: Incremental Build - 02-about page	12 days ago
03-contact.html	D: Incremental Build - 03-contact page	7 days ago
README.md	D: Incremental Build - README.md modified	2 days ago
index.html	P01: First Publication	13 days ago

Figure 9.70 GitHub—repository home page

Now we've change `style.css` on both branches and they have conflicting changes.

Let's try to merge them together by creating a pull request. Click either the **COMPARE AND PULL REQUEST** button or the **NEW PULL REQUEST** button (they do the same thing). This time we get:

Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also compare across forks.

The screenshot shows the GitHub pull request creation interface. At the top, there are dropdown menus for 'base: master' and 'compare: d-04-legal'. A red error message 'Can't automatically merge' is displayed. Below the form, the pull request summary is shown: 'P: Publish - h3 colour changed to grey', '1 commit', '1 file changed', '0 commit comments', and '1 contributor'. On the right, there are sections for 'Reviewers', 'Assignees', 'Labels', 'Projects', and 'Milestone', all currently empty. The main area shows the code diff for 'style.css' with a conflict at line 30. The bottom section displays the message 'No commit comments for this range'.

Figure 9.71 GitHub—open pull request with a conflict

Not looking so good this time.

The top bit is the same; we're merging from **d-04-legal** to the **master** branch.

It's the bit after that that is the problem × **Can't automatically merge**. But it does go on to say that we can still create the pull request; it just means we have to sort it out later.

The bit at the bottom shows the conflict (in case it can be fixed and the pull request redone). In our case I want to leave it, I want the conflict situation. Click the **CREATE PULL REQUEST** button and let's see what happens:

Again we go to the pull request status page and this time we have:

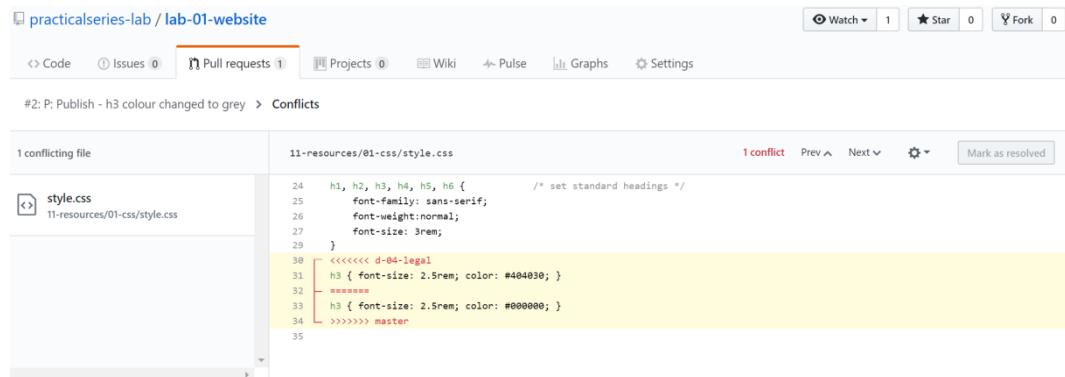
The screenshot shows a GitHub pull request page for a pull request titled "P: Publish - h3 colour changed to grey #2". The pull request is merging 1 commit from the "d-04-legal" branch into the "master" branch. The commit message is "style.css modified h3 colour changed to grey". A comment from "practicalseries-lab" states "style.css modified h3 colour changed to grey". Below the commit, there is a note: "Add more commits by pushing to the d-04-legal branch on practicalseries-lab/lab-01-website." A warning message in a box says: "This branch has conflicts that must be resolved. Use the web editor or the command line to resolve conflicts." It lists "Conflicting files" as "11-resources/01-css/style.css". Below this, there is a "Merge pull request" button and a note: "You can also open this in GitHub Desktop or view command line instructions." At the bottom, there is a comment section with a "Write" tab, a preview area, and a "Comment" button. On the right side, there are settings for "Reviewers", "Assignees", "Labels", "Projects", "Milestone", and "Notifications". The notifications section says "You're receiving notifications because you authored the thread." and has a "Unsubscribe" button. There is also a "Lock conversation" link.

Figure 9.72 GitHub—pull request with a conflict

The [MERGE PULL REQUEST](#) button is greyed out, but now we have a [RESOLVE CONFLICT](#) button.

As before, we could if we wanted, just reject the pull request by clicking the [CLOSE PULL REQUEST](#) button. We could also ask questions with the [COMMENT](#) button (just like last time).

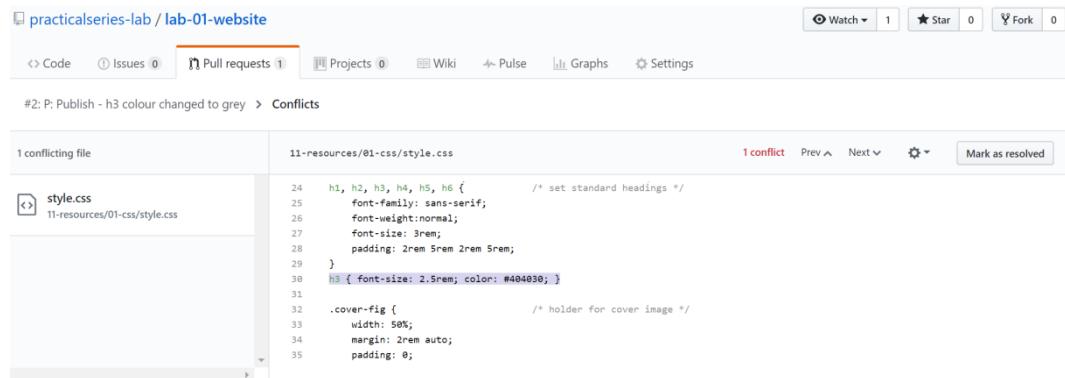
What we're really interested in is resolving the conflict, click the [RESOLVE CONFLICT](#) button—*does it look familiar?*



A screenshot of a GitHub pull request interface. The top navigation bar shows the repository 'practicalseries-lab / lab-01-website' and has tabs for 'Code', 'Issues 0', 'Pull requests 1', 'Projects 0', 'Wiki', 'Pulse', 'Graphs', and 'Settings'. The 'Pull requests' tab is active, showing a single pull request with the title '#2: P: Publish - h3 colour changed to grey'. Below the title, it says 'Conflicts'. A sidebar on the left lists '1 conflicting file' named 'style.css'. The main area shows the '11-resources/01-css/style.css' file with code lines 24 through 35. Lines 30, 31, 32, and 33 are highlighted with a yellow background, indicating a conflict. Line 30 contains the text 'h3 { font-size: 2.5rem; color: #404030; }'. Line 31 contains '====='. Line 32 contains '====='. Line 33 contains '====='. Line 34 contains '>>>> master'. On the right side of the code editor, there are buttons for '1 conflict', 'Prev ⌂', 'Next ⌂', and 'Mark as resolved'.

Figure 9.73 GitHub—merge with a conflict

It wants us to make the correction (just like § 6.7.3 & § 8.5). This time I'm going to keep the first one (grey). Modify the file to make it look like this:



A screenshot of a GitHub pull request interface, identical to Figure 9.73 but with a modified file. The 'style.css' file now contains the following code:

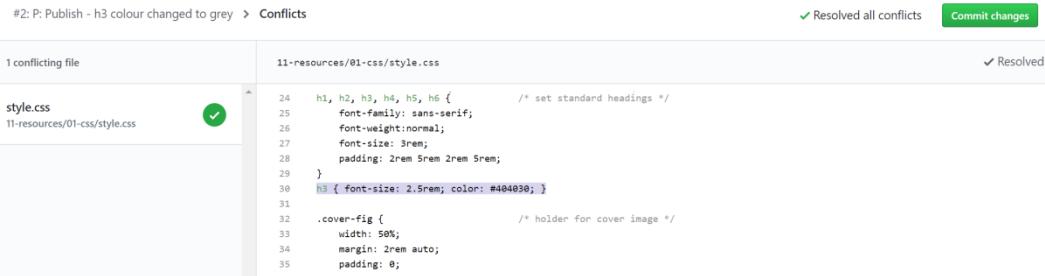
```
h1, h2, h3, h4, h5, h6 { /* set standard headings */  
    font-family: sans-serif;  
    font-weight: normal;  
    font-size: 3rem;  
}  
  
h3 { font-size: 2.5rem; color: #404030; }  
=====  
h3 { font-size: 2.5rem; color: #000000; }  
=====  
>>>> master
```

The conflict markers from Figure 9.73 are no longer present. The 'Mark as resolved' button is visible at the top right of the code editor.

Figure 9.74 GitHub—modified file to resolve conflict

Click the [MARK AS RESOLVED](#) button.

Now we get:



A screenshot of a GitHub commit page. At the top, it says "#2: P: Publish - h3 colour changed to grey > Conflicts". Below this, there's a table showing a single conflicting file, "style.css". The file content is as follows:

```
11-resources/01-css/style.css
24  h1, h2, h3, h4, h5, h6 { /* set standard headings */
25    font-family: sans-serif;
26    font-weight: normal;
27    font-size: 3rem;
28    padding: 2rem 5rem 2rem 5rem;
29  }
30  h3 { font-size: 2.5rem; color: #404030; }
31
32  .cover-fig { /* holder for cover image */
33    width: 50%;
34    margin: 2rem auto;
35    padding: 0;
```

At the bottom right of the commit page, there are two buttons: "Resolved all conflicts" and "Commit changes".

Figure 9.75 GitHub—commit modified file

Click the commit changes button and it takes us back to the pull request status page but this time it looks like it did in § 9.5.3, we can merge the changes:

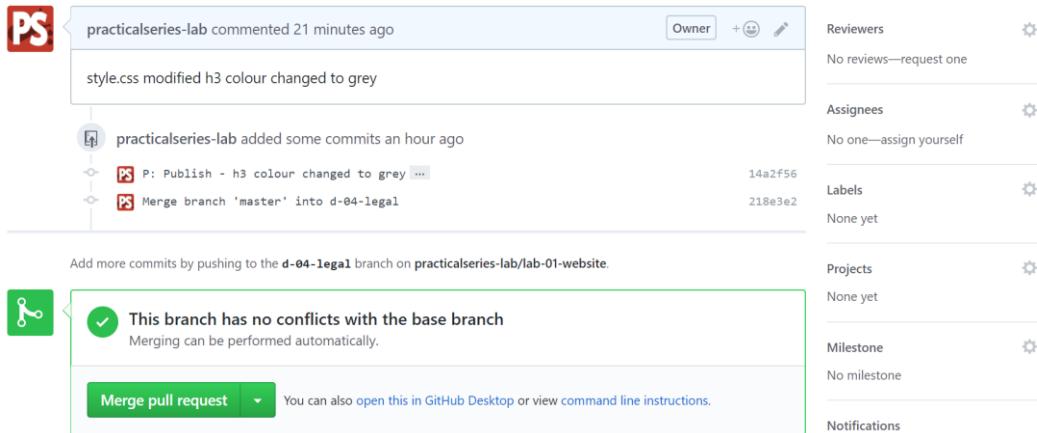


Figure 9.76 GitHub—pull request status page after conflict resolution

We do exactly the same as we did with the original merge, click **MERGE PULL REQUEST** and in the message box, accept the default message by clicking **CONFIRM MERGE**.

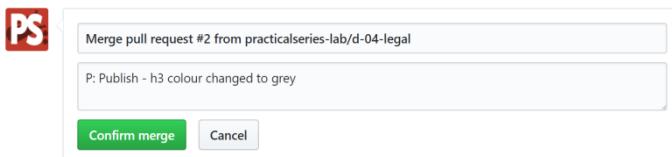


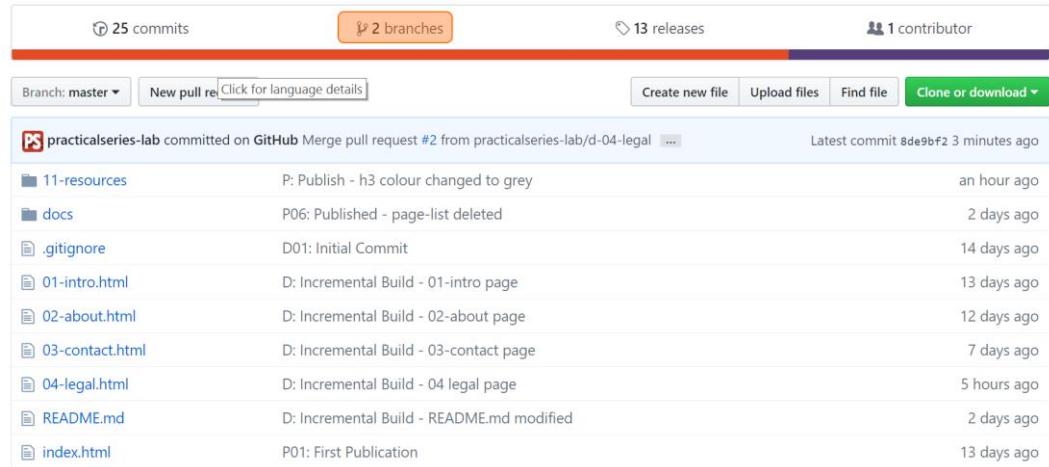
Figure 9.77 GitHub—confirm merge after conflict resolution

And that's it, it's committed. It will leave you on the pull request status page showing the closed pull request. All done.

9.5.6 Deleting a branch

The final thing with branches. Deleting the branch.

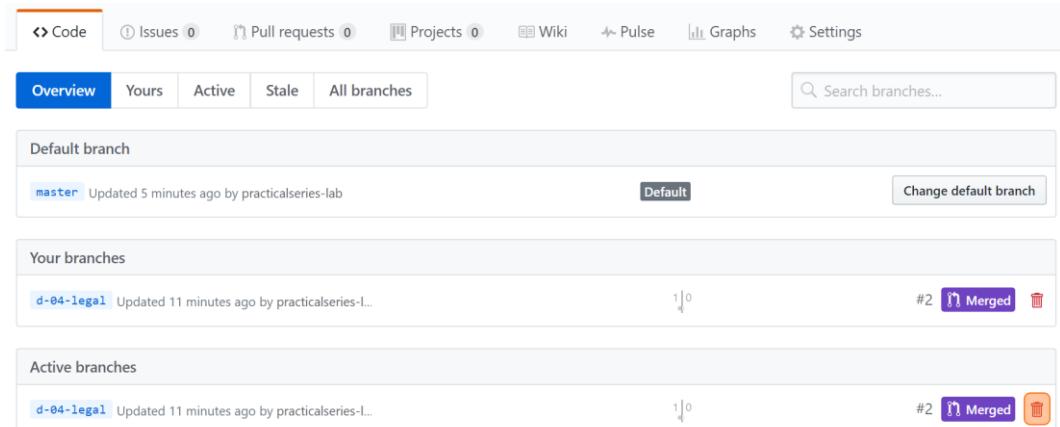
Go to the repository home screen and click the branches tab (highlighted):



A screenshot of a GitHub repository's main page. At the top, there are four summary metrics: 25 commits, 2 branches (which is highlighted with an orange border), 13 releases, and 1 contributor. Below these are several navigation buttons: Branch: master (with a dropdown arrow), New pull request, Click for language details, Create new file, Upload files, Find file, and Clone or download. The main content area shows a list of files and their commit history. A specific commit for '11-resources' is highlighted with a red box, showing the message 'P: Publish - h3 colour changed to grey' and the timestamp 'an hour ago'. Other commits listed include 'docs', '.gitignore', '01-intro.html', '02-about.html', '03-contact.html', '04-legal.html', 'README.md', and 'index.html'.

Figure 9.78 GitHub—select the branches tab

It opens the branches page:



A screenshot of the GitHub branches page. At the top, there are navigation links: Code, Issues 0, Pull requests 0, Projects 0, Wiki, Pulse, Graphs, and Settings. Below these are tabs: Overview (which is highlighted in blue), Yours, Active, Stale, and All branches. A search bar labeled 'Search branches...' is also present. The main content is divided into sections: 'Default branch' (showing 'master' with the status 'Updated 5 minutes ago by practicalseries-lab' and a 'Default' button), 'Your branches' (showing 'd-04-legal' with the status 'Updated 11 minutes ago by practicalseries-lab...', a merge icon, and a trash bin icon), and 'Active branches' (showing 'd-04-legal' with the same status and merge/trash icons). The 'd-04-legal' branch in the 'Your branches' section has a trash bin icon next to it, indicating it is ready to be deleted.

Figure 9.79 GitHub—delete a branch from the branches tab

To delete the branch just click the rubbish bin, it will do it straight away (there is no “are you sure box?”). It does however give you the option to restore it afterwards.

Note: This is also where you can change the default branch, click the button.

9.6

GitHub—tags and releases

We've used tags quite extensively in the local repository using Brackets to create them. We've also seen that we can push these tags up to the GitHub repository (§ 8.3.2). They're there now, you can see them.

From the repository home page, click the releases tab:



Figure 9.80 GitHub—select the releases tab

This takes us to the release page:

A screenshot of the GitHub releases page for the repository 'practicalseries-lab / lab-01-website'. The page has a header with repository information, a navigation bar with 'Code', 'Issues 0', 'Pull requests 0', 'Projects 0', 'Wiki', 'Pulse', 'Graphs', and 'Settings'. Below this, there are two tabs: 'Releases' (which is selected) and 'Tags'. On the right, there is a button 'Draft a new release'. The main content area lists three releases: 1. 'P04' (5 days ago), which contains two assets: 'da038e3.zip' and 'da038e3.tar.gz'. 2. 's-P03.03.P04' (6 days ago), which contains two assets: 'a150909.zip' and 'a150909.tar.gz'. 3. 'd-P03.03.03' (6 days ago), which contains two assets: 'f52a6a5.zip' and 'f52a6a5.tar.gz'. Each release entry includes a dropdown arrow and three dots for more options.

Figure 9.81 GitHub—releases page

This lists all the tags we entered in the course of the previous examples using Brackets (I've only shown the first three, there are 13 altogether).

Now these are releases, not tags (see the blue button). If you click the tags button you get something very similar:

Created	Tag	Commit Hash	Actions
5 days ago	P04	da038e3	zip tar.gz
6 days ago	s-P03.03.P04	a150909	zip tar.gz
6 days ago	d-P03.03.03	f52a6a5	zip tar.gz

Figure 9.82 GitHub—tags page

All the tags we created in Brackets show up in GitHub as both tags and release.

GitHub doesn't really bother about tags—it accepts them because that is what is available in Git (and consequently Brackets). But it really wants us to use releases. GitHub doesn't allow us to create tags directly, they have to be added by adding a release (although it can delete them directly).

So what's the difference?

9.6.1 Tags and releases, what's the difference?

Well a tag is just a unique symbolic name attached to a particular commit point. Git supports two types of tag (§ 6.3.4) a lightweight tag (without a message) and an annotated tag (with a message). In Brackets we only have lightweight tags.

GitHub can display both. In Figure 9.82 above, the tags are all lightweight, I can tell this because it tells me there is **NO RELEASE NOTES** for any of the tags, this release note is what Git refers to as the tag message attached to the annotated tag. If we added a release note it would become an annotated tag.

A release on the other hand is a GitHub object, it is still attached to a particular commit point, but this time it allows us to specify that a particular commit has a particular release status.

GitHub converts all our tags to releases when we push them up from the local repository and that is why they are listed as both releases and tags.

9.6.2 Viewing releases (and tags)

Let's have a closer look at the releases page (I'm choosing release over tags, but the points in this section apply equally to both), this is an expanded version of Figure 9.81:

The screenshot shows the GitHub Releases page with the 'Releases' tab selected. There are ten releases listed, each with a timestamp, a tag name, and four download links (zip, tar.gz, etc.). The fourth release, 'd-P03-03-01', is expanded, revealing its commit message: 'D: Incremental Build - 03-contact page 03-contact added.' Below the commit message are four numbered points: ①, ②, ③, and ④. Point ② is circled in orange and points to the expanded commit message. Point ③ is circled in orange and points to the commit point '8329b67'. Point ④ is circled in orange and points to the tag name 'd-P03-03-01'. Points ① and ② are also circled in orange.

Release	Tag	Commit	Zip	Tar.GZ
5 days ago	P04	da038e3	[zip]	[tar.gz]
6 days ago	s-P03.03.P04	a150909	[zip]	[tar.gz]
6 days ago	d-P03.03.03	f52a6a5	[zip]	[tar.gz]
6 days ago	d-P03.03.02	4da998f	[zip]	[tar.gz]
7 days ago	d-P03-03-01	8329b67	[zip]	[tar.gz]
12 days ago	P03	87fe437	[zip]	[tar.gz]
12 days ago	s-P01.02.P03	28a335c	[zip]	[tar.gz]
12 days ago	P02	abf1121	[zip]	[tar.gz]
12 days ago	s-P01.01.P02	f5aeb76	[zip]	[tar.gz]
12 days ago	d-P01.02.01	3f15582	[zip]	[tar.gz]

Figure 9.83 GitHub—releases page, details

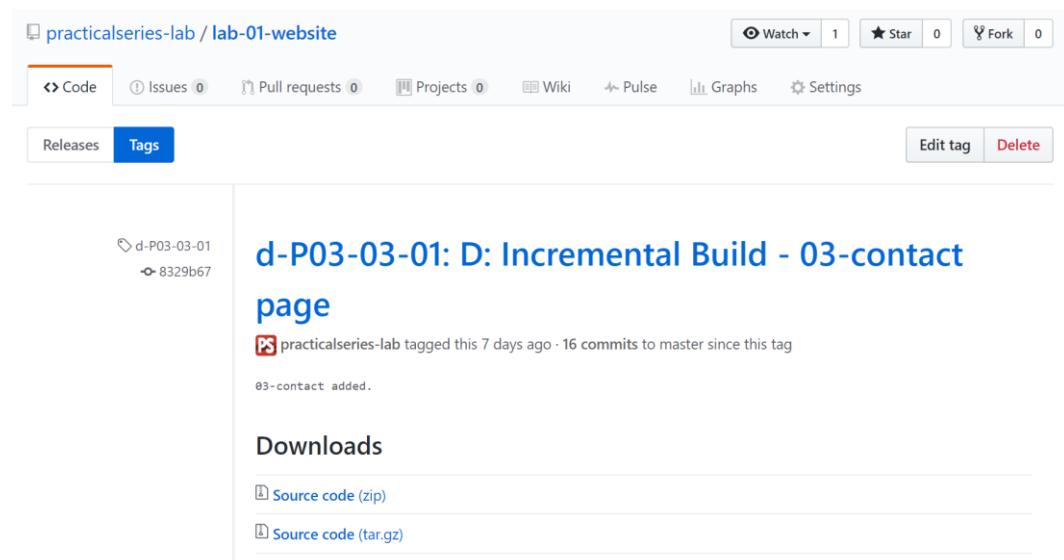
This shows the ten most recent releases (tags). Each release has four different link groups. The most simple is point ②, this just expands the commit message underneath the tag name (if the tag has an annotation or release note, it will show this instead).

Point ③ is the commit point to which the release (tag) is attached. If you click this it will take you to that commit point (just like the commit history § 9.4.2).

Point ④, the [ZIP](#) and [TAR.GZ](#)¹ buttons download a copy of the repository exactly as it was at the time of the commit with which the release is associated, this is exactly the same as downloading it from the commit history (except the file is called `<repository name>-<release name>.zip or .tar.gz`).

This is a quick way to download the contents of a previous commit point; they're all listed in the same place.

Finally point ① (*perhaps I should have numbered them differently*). This takes us to the detailed tag information page (note: this is the underlying tag part of the release). It looks like this:



The screenshot shows a GitHub tag page for the repository `practicalseries-lab / lab-01-website`. The tag is `d-P03-03-01`, which corresponds to commit `8329b67`. The page includes a summary of 16 commits since the tag was created. Below the tag summary, there's a section titled "Downloads" containing links to download the source code as a zip file or a tar.gz file.

Figure 9.84 GitHub—tag page, details

Clicking either the tag or the commit number in the left column will take you to that commit (again, just like the commit history § 9.4.2). The **DOWNLOADS** area just downloads the zip and tar files again.

¹

Tar.gz is a compressed archive (TAR is from UNIX, it stands for tape archive)—similar to zip files, but common within Linux and Unix environments. The .gz extension indicates that it is a compressed file. Tar.gz files are often referred to as tarballs (tar from Tape ARchive and ball from the collective tar ball as in sticks things together—*what can I say? Linux people*).

The **EDIT** button allows a release message to be added or if it already exists, edited—same as the add release notes in Figure 9.82.

The **DELETE** button, not surprisingly, deletes the tag. It pulls up an “*are you sure?*” box:

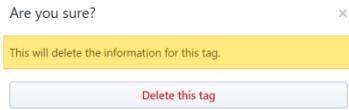
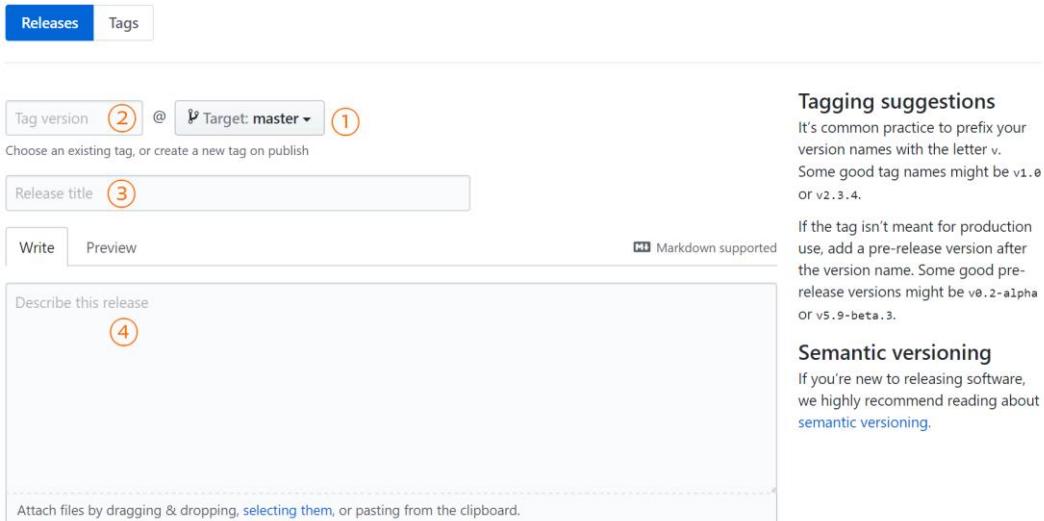


Figure 9.85 GitHub—delete tag

9.6.3 Creating a release

Go back to the releases page (Figure 9.81) and click the **DRAFT A NEW RELEASE** button:



Releases Tags

Tag version (2) @ Target: master (1)
Choose an existing tag, or create a new tag on publish

Release title (3)

Write Preview

Markdown supported

Describe this release (4)

Attach files by dragging & dropping, selecting them, or pasting from the clipboard.

Tagging suggestions
It's common practice to prefix your version names with the letter v.
Some good tag names might be v1.0 or v2.3.4.
If the tag isn't meant for production use, add a pre-release version after the version name. Some good pre-release versions might be v0.2-alpha or v5.9-beta.3.

Semantic versioning
If you're new to releasing software, we highly recommend reading about semantic versioning.

Figure 9.86 GitHub—create a release

The main thing here is point ①, this is currently pointing to the **head** of the **master** branch [8de9bf2], it just says **TARGET MASTER** but that's what it means (the **head** of the selected branch).

It's possible to select lots of things from this button:

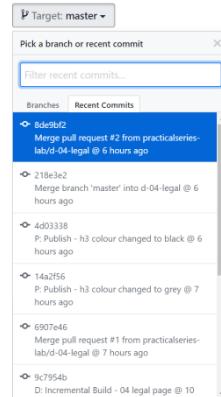


Figure 9.87 GitHub—create release, select target

If you click the **RECENT COMMITS** tab it gives a list of all the commit points in the project. This allows any commit point to be given a release (and a tag).

Leave it set to the top commit [8de9bf2], we'll tag the latest commit point.

The next thing is point ②, this is the tag name, call it P07, it should look like this:

P07 @ Target: master

Excellent! This tag will be created from the target when you publish this release.

P07: Published

Write Preview Markdown supported

04- legal added (branch d-04-legal merged)

Attach files by dragging & dropping, selecting them, or pasting from the clipboard.

Attach binaries by dropping them here or selecting them.

This is a pre-release
We'll point out that this release is identified as non-production ready.

Publish release **Save draft**

Figure 9.88 GitHub—create a release details

A release title can be added, point ③, and a description, point ④; both are optional.

There is also a **PRE-RELEASE** tick box, this again is optional, it affects how GitHub shows the release, if it is a pre-release (this one is), tick the box.

The finished article should look like Figure 9.88. Click the **PUBLISH RELEASE** button to complete the operation.

This takes you back to the releases page and it now looks like this:

The screenshot shows the GitHub Releases page. At the top, there are tabs for 'Releases' (which is selected) and 'Tags'. In the top right corner, there is a button labeled 'Draft a new release'. Below the tabs, there is a section for a new release point. It has a 'Pre-release' checkbox checked, a title 'P07: Published', and a subtitle 'practicalseries-lab released this 6 hours ago'. The commit hash is '8de9bf2'. The notes say '04- legal added (branch d-04-legal merged)'. Below this, there is a 'Downloads' section with links for 'Source code (zip)' and 'Source code (tar.gz)'. At the bottom of the page, there is another release point 'P04' from 6 days ago, which includes a commit hash 'da038e3' and download links for 'zip' and 'tar.gz'.

Figure 9.89 GitHub—release page with new release point

It looks a bit more impressive than the ones we had before.

Go to the tags page and you will see we've added the release as a tag too:

The screenshot shows the GitHub Tags page. At the top, there are tabs for 'Releases' and 'Tags' (which is selected). Below the tabs, there are three entries: 1. A release point 'P07' from 6 hours ago with commit hash '8de9bf2' and download links for 'zip' and 'tar.gz'. To the right, there are buttons for 'Edit release notes' and 'Release notes'. 2. A release point 'P04' from 6 days ago with commit hash 'da038e3' and download links for 'zip' and 'tar.gz'. To the right, there is a link 'Add release notes (No release notes)'. 3. A tag 's-P03.03.P04' from 6 days ago with commit hash 'a150909' and download links for 'zip' and 'tar.gz'. To the right, there is a link 'Add release notes (No release notes)'. The tags 'P07' and 's-P03.03.P04' are highlighted in blue.

Figure 9.90 GitHub—release page with new release point

There it is, if you pull this into Brackets, the new tag will be there too.

9.6.4 Pulling releases back into Brackets

This isn't strictly speaking to do with GitHub, but if we pull all the changes we've made back into Brackets we get this:

The screenshot shows the Brackets IDE interface. The main area displays the code for `04-legal.html`. The commit history panel on the right lists numerous commits from the GitHub repository, with the most recent one at the top:

Commit	Message	Author	Date
P	Merge pull request #2 from practicalseries-lab/d-04-legal	P07	8de0bf2
P	Merge branch 'master' into d-04-legal		218e3e2
P	P: Publish - h3 colour changed to black		4d03338
P	P: Publish - h3 colour changed to grey		14a2f56
P	Merge pull request #1 from practicalseries-lab/d-04-legal		6907e46
P	D: Incremental Build - 04 legal page		9c7954b
P	P06: Published - page-list deleted		19de35d
P	D: Incremental Build - content.txt renamed		09fc21b
P	D: Incremental Build - README.md modified		5644edc
P	P05: Published - Version Numbering doc added		100bd32
P	D: Incremental Build - /docs/content.txt added		6447f4
P	P04: Published (merged d-03-contact)	P04	da030e3
P	Merge remote-tracking branch 'lab-01-website/d-03-contact' into d-03-contact	s-P03.04	a150909
P	D: Incremental build - 03-contact page	temp	80fe002
P	D: Incremental build - 03-contact page	d-P03.03	f52a6a5
P	D: Incremental Build - README.md modified	d-P03.03.02	4da998f
P	D: Incremental Build - 03-contact page	d-P03.03.01	8329b67
P	P03: Published (merged d-02-about).	P03	07fe437
P	S: Staged - 02-about page	s-P01.02.P03	28a335c
P	P02: Published (merged d-01-intro)	P02	abf1121
P	S: Staged - 02-about page	s-P01.01.P02	f5aeb76
P	D: Incremental Build - 02-about page	d-P01.02.01	3f15582
P	D: Incremental Build - 01-intro page	d-P01.01.01	e14911e
P	P01: First Publication	P01	25c1410
P	D01: Initial Commit	D01	8ce2e6e

Figure 9.91 Pull GitHub releases back into Brackets

There it is, top of the list—**P07**, it appears as a tag.

10

COLLABORATIVE WORKING

GitHub and working with other people's repositories.

GITHUB IS DESIGNED for collaborative working, a team of people can work on the same project at the same time and GitHub will manage everything, resolve conflicts and keep it all up to date—just like a good version control system should.

Now mostly when people work together as a team, the team is established, people are given responsibility for things and the work gets done (*and you make a neat gun*¹). There is usually someone managing the team (let's call them a lead engineer) telling people what to do and assigning privileges and access to users of the project; and this is how it is meant to be. This is how we do things at work (*we have cake too*), and we do make neat guns.

GitHub lets us do this too; GitHub allows different users to be given access to a repository, it also allows organisations to be set up for just this purpose —an organisation is effectively a repository (or several repositories) that is managed by a group of users.

This is all good and sensible (“*right and proper*” as my old Dad would have said) and just what you would expect. These collaborative repositories can be accessed by users as local repositories on their own machine (just as we did in section 8) and it all works very well.

Where I have a problem is with the open source aspects of GitHub (*here comes the Linux lynch mob*), is that anyone can take a copy of your repository and work on it, it's called *forking* (*I really, really hate these names*). Not only can they work on it, they can then send it back to you (with a pull request, just like we used in § 9.5.3) and ask you to incorporate it back into your repository.

I know it is possible to have private repositories (*and I have them*), but I also have public repositories and I make these public because I think they might be useful. Now I don't mind people copying this work, to some extent that's what it's there for. I just think it would be polite to be asked first. It seems a bit rude taking a copy of someone's work without so much as a by your leave—what's worse is posting it back to them, telling them you've fixed all their shortcomings and would they be good enough to incorporate your changes, and then complaining when you don't—*bloody do-gooders*.

¹

Sorry—it sounded like Portal.

Now I know the younger generation with their Facebook and Twitter tend to live in each other's pockets and like to share things (*in a way that I very much do not*).

But I'm a tired old Yorkshire man; I'm a bit grumpy and not tainted with emotion. In my day, the only thing young people shared was herpes.

So while I'm willing to share my software—people should remember I do it by and large for nothing, and I do it in my own time. I'm open to criticism and for people to help, but I would like this to be polite criticism. I don't like being shouted at (*Linux people I'm talking about you here*) and there is a bit too much of it on GitHub and I think *forking* is a symptom of this.

Well there's my rant.

I will explain forking, and I have no problem with it between repositories that you have access to; however, if you're copying a complete stranger's repository maybe you should ask first.

10.1 Forking

I need two GitHub user accounts to demonstrate this, and fortunately I have them, I have the [practicalseries-lab](#) account and I also have a [michaelgledhill](#) account.

I'm going to logon to the [michaelgledhill](#) account and copy the [lab-01-website](#) repository into a new repository in that account; this is a process that is referred to as *forking* in GitHub terminology.

This is the [michaelgledhill](#) home page, you can tell, it's got my picture at the top:

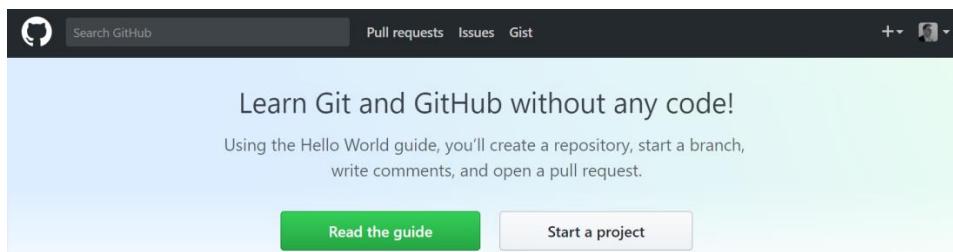


Figure 10.1 The michaelgledhill newsfeed page

I can search for any repository in GitHub by typing all or part of its name into the [SEARCH GITHUB](#) box at the top.

If I type [lab-01-website](#) I get this:

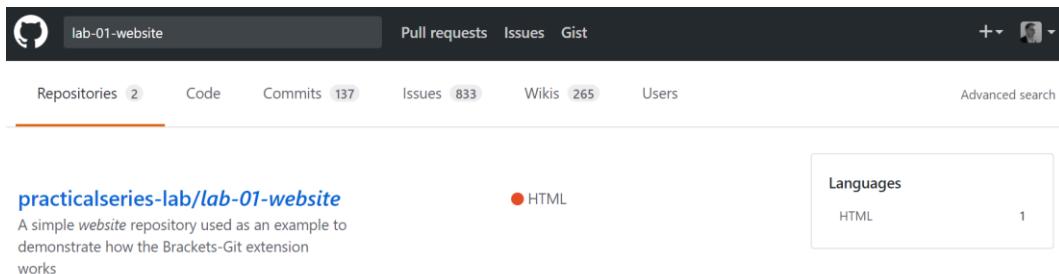


Figure 10.2 Search for lab-01-website

If I click [PRACTICALSERIES-LAB/LAB-01-WEBSITE](#) link it takes me to the repository home page:

The screenshot shows a GitHub repository page for the repository `practicalseries-lab / lab-01-website`. The top navigation bar includes links for "This repository", "Search", "Pull requests", "Issues", and "Gist". On the right side of the header are buttons for "Watch" (with 1 watch), "Star" (0 stars), "Fork" (0 forks), and a profile picture. Below the header, there are tabs for "Code", "Issues 0", "Pull requests 0", "Projects 0", "Wiki", "Pulse", and "Graphs". A descriptive text below the tabs reads: "A simple website repository used as an example to demonstrate how the Brackets-Git extension works". A blue link labeled "practicalseries-publication" is visible. Below this, a summary box shows "25 commits", "1 branch", "14 releases", and "1 contributor". A dropdown menu shows "Branch: master" and a "New pull request" button. To the right are buttons for "Create new file", "Upload files", "Find file", and a green "Clone or download" button. The main content area displays a list of commits, each with a file icon, a commit message, and a timestamp. The commits listed are:

File	Message	Time Ago
11-resources	P: Publish - h3 colour changed to grey	21 hours ago
docs	P06: Published - page-list deleted	3 days ago
.gitignore	D01: Initial Commit	15 days ago
01-intro.html	D: Incremental Build - 01-intro page	14 days ago
02-about.html	D: Incremental Build - 02-about page	13 days ago
03-contact.html	D: Incremental Build - 03-contact page	8 days ago
04-legal.html	D: Incremental Build - 04 legal page	a day ago
README.md	D: Incremental Build - README.md modified	3 days ago
index.html	P01: First Publication	14 days ago

Figure 10.3 The lab-01-website website viewed from another user profile

Now this looks just the same as it did when we were logged in as `practicalseries-lab`; the only difference can be seen at the top, it's got my picture instead of the practical series logo.

There is a difference if you try to modify a file, if I click the `README.md` file and then try to edit it, I get the following message:

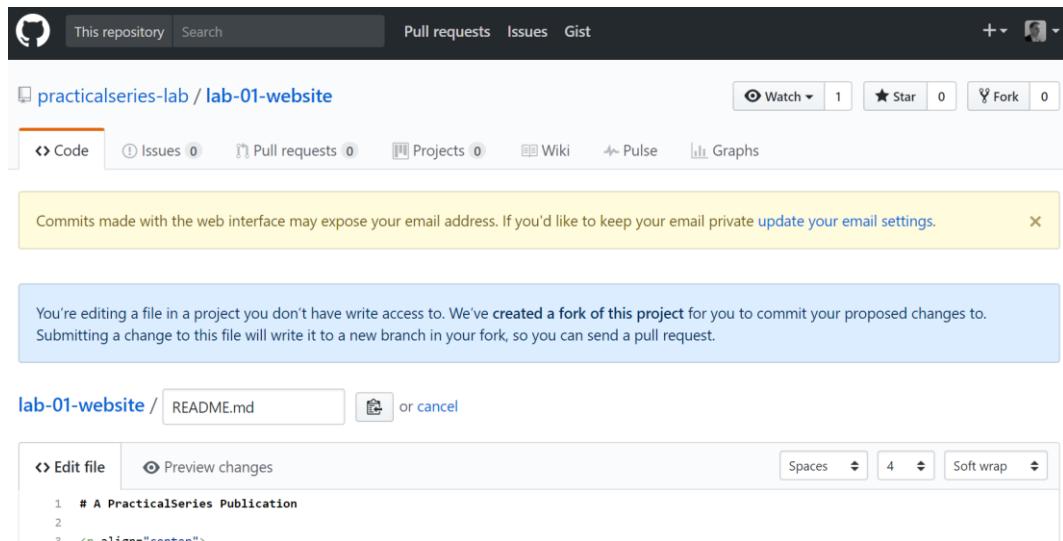


Figure 10.4 Editing a file in another users repository

It tells me “I’m trying to edit a file I don’t have write access to”. It also says it’s “created a *fork* for this project”. And indeed it has done so, this is one way of forking a repository—but this is not the usual way of doing it.

10.1.1 Forking (copying) a repository

Go back to the repository [practicalseries-lab](#), [lab-01-website](#) repository home page (Figure 10.3).

The easiest way to fork (create a copy of the project in a different profile) is to just click the **FORK** button at the top. If you hover the mouse over the button you get this:

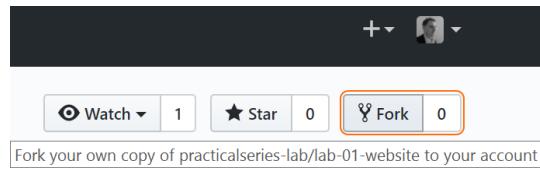


Figure 10.5 Fork a repository

It tells me it will make a copy of the repository in my account (the [michaelgledhill](#) account).

If I click the **FORK** button, point ① in Figure 10.3 I get this:

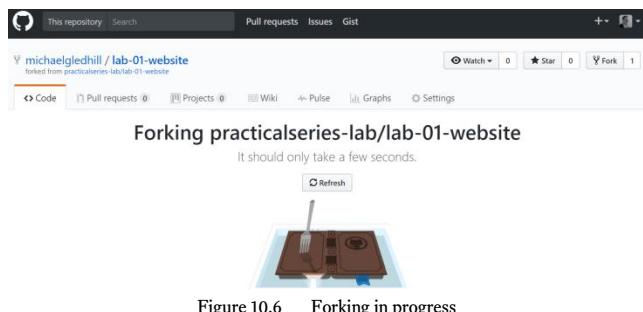


Figure 10.6 Forking in progress

This screen hangs around for a few seconds¹.

It then takes me to the home page for the repository, but in the `michaelgledhill` account (not `practicalseries-lab`). I.e. it is the home page of the copy, not the original.

The screenshot shows the GitHub repository homepage for `michaelgledhill / lab-01-website`. The header is identical to Figure 10.6. The main content area includes a brief description: 'A simple website repository used as an example to demonstrate how the Brackets-Git extension works', an 'Edit' button, and a 'Add topics' link. Below this are summary statistics: '25 commits', '1 branch', '14 releases', and '1 contributor'. There are buttons for 'Branch: master', 'New pull request', 'Create new file', 'Upload files', 'Find file', and 'Clone or download'. The commit list shows the following entries:

File/Commit	Description	Time Ago
<code>11-resources</code>	P: Publish - h3 colour changed to grey	22 hours ago
<code>docs</code>	P06: Published - page-list deleted	3 days ago
<code>.gitignore</code>	D01: Initial Commit	15 days ago
<code>01-intro.html</code>	D: Incremental Build - 01-intro page	14 days ago
<code>02-about.html</code>	D: Incremental Build - 02-about page	13 days ago
<code>03-contact.html</code>	D: Incremental Build - 03-contact page	8 days ago
<code>04-legal.html</code>	D: Incremental Build - 04 legal page	a day ago
<code>README.md</code>	D: Incremental Build - README.md modified	3 days ago
<code>index.html</code>	P01: First Publication	14 days ago

Figure 10.7 The forked repository

¹

I think this is a false delay screen, it implies it's copying the repository and it takes a few seconds; when I tried to edit the file, it forked the repository instantly.

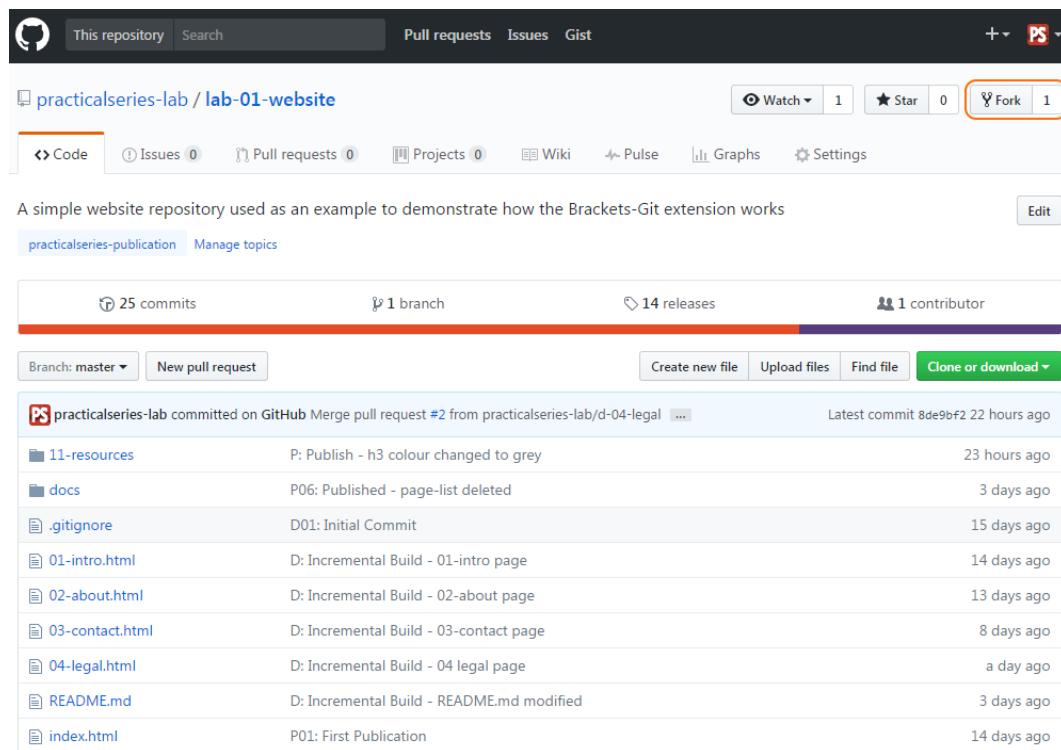
The clue to this being a copy is at the top, it says:

michaelgledhill / lab-01-website
forked from practicalseries-lab/lab-01-website

It is in the `michaelgledhill` account, not the `practicalseries-lab` account, indeed the next line tells me it's been forked from that account.

So there we are, I've forked (*copied*) a repository from one GitHub profile to another.

Let's have a quick look from the `practicalseries-lab` side of things. If I sign in to the `practicalseries-lab` account and go to the `lab-01-website` repository home page I have this:



A simple website repository used as an example to demonstrate how the Brackets-Git extension works

practicalseries-publication Manage topics

Branch: master ▾ New pull request

Create new file Upload files Find file Clone or download ▾

File	Commit Message	Time
11-resources	P: Publish - h3 colour changed to grey	23 hours ago
docs	P06: Published - page-list deleted	3 days ago
.gitignore	D01: Initial Commit	15 days ago
01-intro.html	D: Incremental Build - 01-intro page	14 days ago
02-about.html	D: Incremental Build - 02-about page	13 days ago
03-contact.html	D: Incremental Build - 03-contact page	8 days ago
04-legal.html	D: Incremental Build - 04 legal page	a day ago
README.md	D: Incremental Build - README.md modified	3 days ago
index.html	P01: First Publication	14 days ago

Figure 10.8 The original repository

I can see that someone has forked the repository (the `1` highlighted).

I can even find out who, if I click the number next to the **FORK** button, it takes me to a network diagram of the repository:

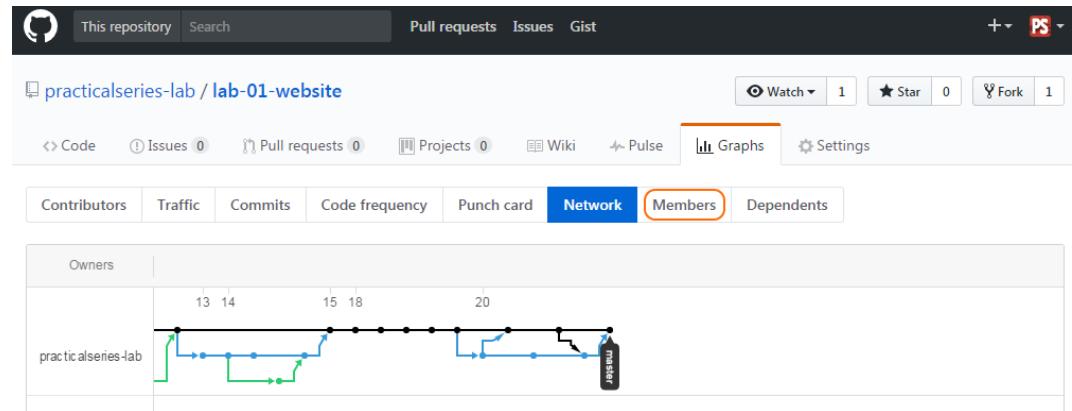


Figure 10.9 The original repository network

It shows a kind of network of the repository, it shows where branches have been created and commits made—it's a poor man's version of the London Underground diagrams I use in my examples. It's not particularly useful.

Click the members tab (highlighted):

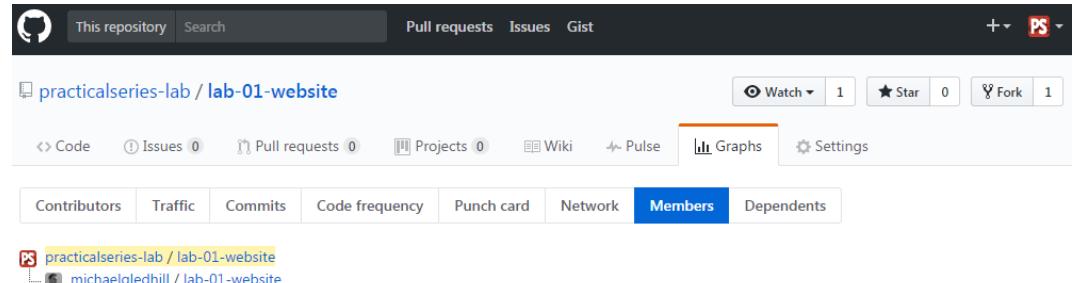


Figure 10.10 The original repository members

This shows who has made a fork of the repository, just me in this case.

10.1.2 Creating a pull request on a forked repository

Just look at that title, *creating a pull request on a forked repository*—It's gibberish—it would mean absolutely nothing to a normal person.

Back to the forked repository in the [michaelgledhill](#) profile, it looks like Figure 10.7.

This forked repository is for all intents and purpose just a repository in the [michaelgledhill](#) profile, it could be cloned to a local repository (just like we did in § 5.3) and managed through Brackets. The difference here is that GitHub knows it was copied from some other repository.

If we modify our copy of the repository, we can send our changes back to the original in the form of a *pull request* that the owner of the original repository can merge back into the original (or reject by closing the pull request). This is exactly the same process we went through with pull requests when merging branches.

I'll run through it, I'm going to modify the [README.md](#) file in the forked repository; I'm just going to put a new line in at the top:

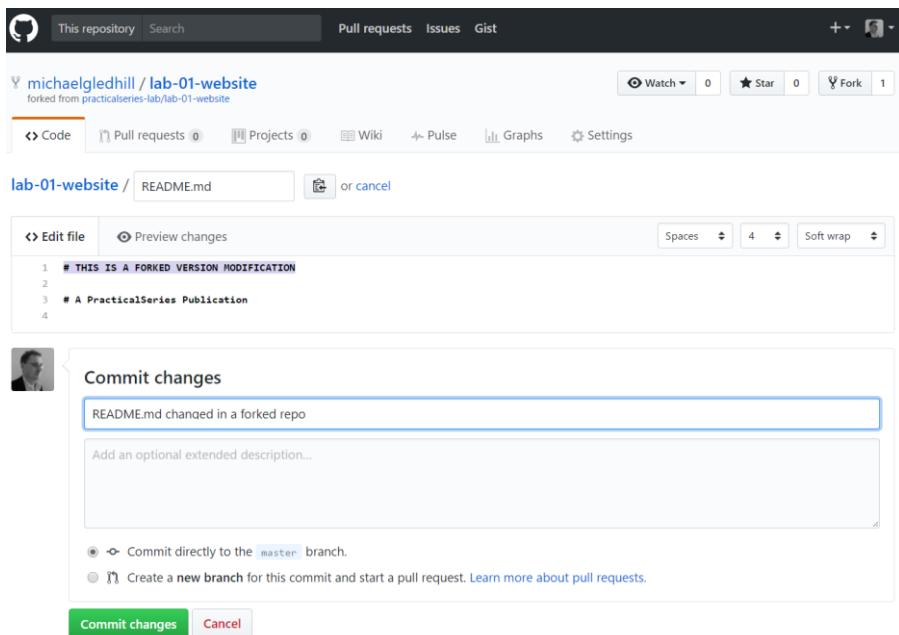


Figure 10.11 modifying a forked repository

Add the commit message and click **COMMIT CHANGES**.

Go back to the forked repository home page (the change will be visible at the bottom):

The screenshot shows a GitHub repository page for 'michaelgledhill / lab-01-website'. At the top, it says 'forked from practicalseries-lab/lab-01-website'. Below the header, there are tabs for 'Code', 'Pull requests (0)', 'Projects (0)', 'Wiki', 'Pulse', 'Graphs', and 'Settings'. A summary bar indicates 26 commits, 1 branch, 14 releases, and 1 contributor. A 'New pull request' button is highlighted. The main area lists commits, with the latest being a modification to 'README.md' by 'michaelgledhill' 3 minutes ago. Below the commit list, a large red box highlights the text 'THIS IS A FORKED VERSION MODIFICATION' and 'A PracticalSeries Publication'. A logo for 'THE' is also visible.

File	Commit Message	Time Ago
11-resources	P: Publish - h3 colour changed to grey	2 days ago
docs	P06: Published - page-list deleted	4 days ago
.gitignore	D01: Initial Commit	16 days ago
01-intro.html	D: Incremental Build - 01-intro page	15 days ago
02-about.html	D: Incremental Build - 02-about page	14 days ago
03-contact.html	D: Incremental Build - 03-contact page	9 days ago
04-legal.html	D: Incremental Build - 04 legal page	2 days ago
README.md	README.md changed in a forked repo	3 minutes ago
index.html	P01: First Publication	15 days ago

Figure 10.12 Forked repository after the modification

Now I've modified my forked copy of the repository, I can send this change back to the original repository by creating a pull request (highlighted), this is the same process as merging two branches (§ 9.5.3). Click the **NEW PULL REQUEST** button:

The screenshot shows a GitHub repository page for 'practicalseries-lab / lab-01-website'. At the top, there are navigation links for 'This repository', 'Search', 'Pull requests', 'Issues', 'Marketplace', and 'Gist'. On the right, there are buttons for 'Watch' (1), 'Star' (0), 'Fork' (1), and a user profile icon. Below the header, the repository name 'practicalseries-lab / lab-01-website' is displayed, along with a 'Code' tab and other navigation links for 'Issues' (0), 'Pull requests' (0), 'Projects' (0), 'Wiki', 'Pulse', and 'Graphs'. A message says 'Choose two branches to see what's changed or to start a new pull request. If you need to, you can also compare across forks.' Below this, a dropdown menu shows 'base fork: practicalseries-lab/lab-01-website...', 'base: master', '...', 'head fork: michaeldledhill/lab-01-website...', and 'compare: master'. A green checkmark indicates 'Able to merge. These branches can be automatically merged.' A 'Create pull request' button is visible. The main content area shows a commit history for May 22, 2017, by 'michaeldledhill' changing 'README.md' in a forked repository. The commit hash is 17b9868. Below the commit details, it says 'Showing 1 changed file with 2 additions and 0 deletions.' A diff viewer shows the changes in 'Unified' mode, highlighting the addition of '# THIS IS A FORKED VERSION MODIFICATION' and '# A PracticalSeries Publication'. There are buttons for 'Unified' and 'Split' view modes.

Figure 10.13 Forked repository create a pull request

The important bit here is the section I've highlighted; it shows that the pull request has a base (receiving) repository set to the original `practicalseries-lab/lab-01-website master` branch; the other side is the forked repository. It also says the files can be merged without conflict.

I'll create the pull request by clicking the **CREATE PULL REQUEST** button.

Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#).

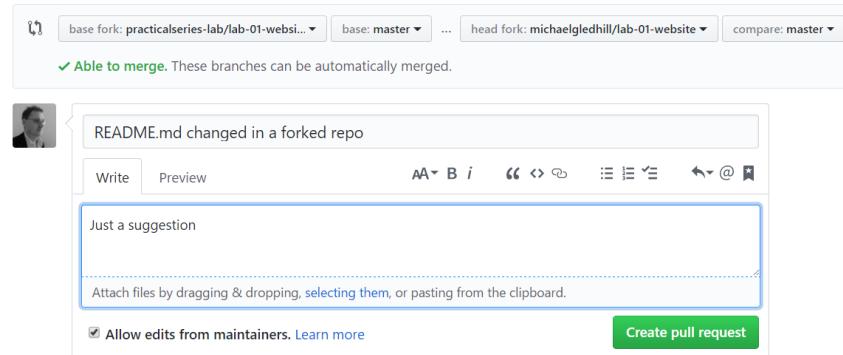


Figure 10.14 Forked repository open the pull request

This time I've added a comment in the box. I click the green button to create the pull request.

Now I get this:

The screenshot shows the GitHub pull request page for the commit 'README.md changed in a forked repo'. The pull request is titled 'README.md changed in a forked repo #3'. It has been opened by 'michaelgledhill' and merged into 'practicalseries-lab:master' from 'michaelgledhill:master'. The commit message is 'Just a suggestion'. On the right side, there are sections for 'Reviewers' (no reviews), 'Assignees' (no one assigned), 'Labels' (none yet), 'Projects' (none yet), 'Milestone' (no milestone), 'Notifications' (you're receiving notifications because you authored the thread), and '1 participant' (the user's profile picture). The main body of the pull request shows the commit message and the GitHub UI for leaving a comment or closing the pull request.

Figure 10.15 Forked repository pull request

There are a couple of things here, the first is that **I can't merge the pull request**, the line in the middle says “*only those with write access to the repository can merge pull requests*” and I don't have write access to the original.

There are two things that I can do from here; I can close the pull request myself by clicking the **CLOSE PULL REQUEST** button (highlighted). By closing it (if say, I'd made a mistake) I would stop the pull request bothering the owner of the original repository.

I can also send further comments if I want to.

Let's look at it from the [practicalseries-lab](#) side (the owner of [lab-01-website](#)):

The screenshot shows a GitHub repository page for [practicalseries-lab / lab-01-website](#). The top navigation bar includes links for Pull requests, Issues, Marketplace, and Gist. Below the header, there are buttons for Watch (1), Star (0), Fork (1), and a dropdown menu. The main content area displays a simple website repository used as an example to demonstrate how the Brackets-Git extension works. It shows 25 commits, 1 branch, 14 releases, and 1 contributor. A pull request summary indicates that practicalseries-lab committed on GitHub Merge pull request #2 from practicalseries-lab/d-04-legal. The list of commits includes:

- 11-resources: P: Publish - h3 colour changed to grey (2 days ago)
- docs: P06: Published - page-list deleted (4 days ago)
- .gitignore: D01: Initial Commit (16 days ago)
- 01-intro.html: D: Incremental Build - 01-intro page (15 days ago)
- 02-about.html: D: Incremental Build - 02-about page (14 days ago)
- 03-contact.html: D: Incremental Build - 03-contact page (9 days ago)
- 04-legal.html: D: Incremental Build - 04 legal page (2 days ago)
- README.md: D: Incremental Build - README.md modified (4 days ago)
- index.html: P01: First Publication (15 days ago)

Figure 10.16 Viewing the pull request from the original repository

I can see that there is 1 pull request. If I click the tab I can view all the pull requests:

The screenshot shows the pull request list for a forked repository. The interface includes filters for Open and Closed pull requests, and sorting options. One pull request is listed:

- README.md changed in a forked repo**: #3 opened 12 minutes ago by michaelgledhill

Figure 10.17 The pull request list

Clicking on the pull request itself, opens the full details of the request:

The screenshot shows a GitHub pull request page for a repository named 'practicalseries-lab'. The pull request is titled 'README.md changed in a forked repo #3'. The status is 'Open' and it shows one commit from 'michaelgledhill' merging into 'master' from 'michaelgledhill:master'. The commit message is 'Just a suggestion'. Below the commit, there is a note: 'Add more commits by pushing to the master branch on michaelgledhill/lab-01-website.' A green box highlights the merge status: 'This branch has no conflicts with the base branch' and 'Merging can be performed automatically.' A large text input field contains the message 'No, I don't want to do this.' at the top, followed by a placeholder 'Attach files by dragging & dropping, selecting them, or pasting from the clipboard.' At the bottom, there are buttons for 'Close and comment' and 'Comment'.

Figure 10.18 The pull request page

This time I can merge the pull request (I'm the owner of the repository), I can also just **CLOSE** the request or I can generate a **COMMENT**.

This time I'm going to close the request (not implement it), but I will do it with the message:

No, I don't want to do this.

I've added the message to the box and I close the request by clicking the **CLOSE AND COMMENT** button.

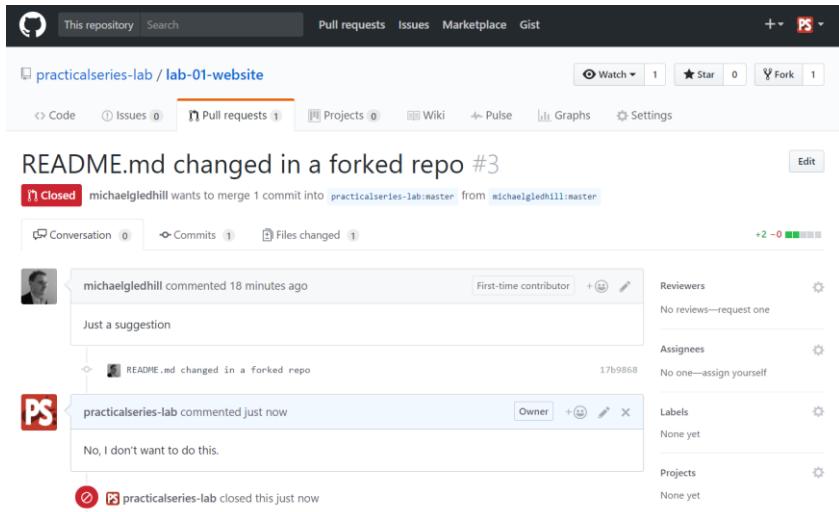


Figure 10.19 Closed pull request from the owner's side

If I go back to the [michaelgledhill](#) profile, I don't see anything that tells me what has happened, there are no messages and there is nothing in the pull request tab.

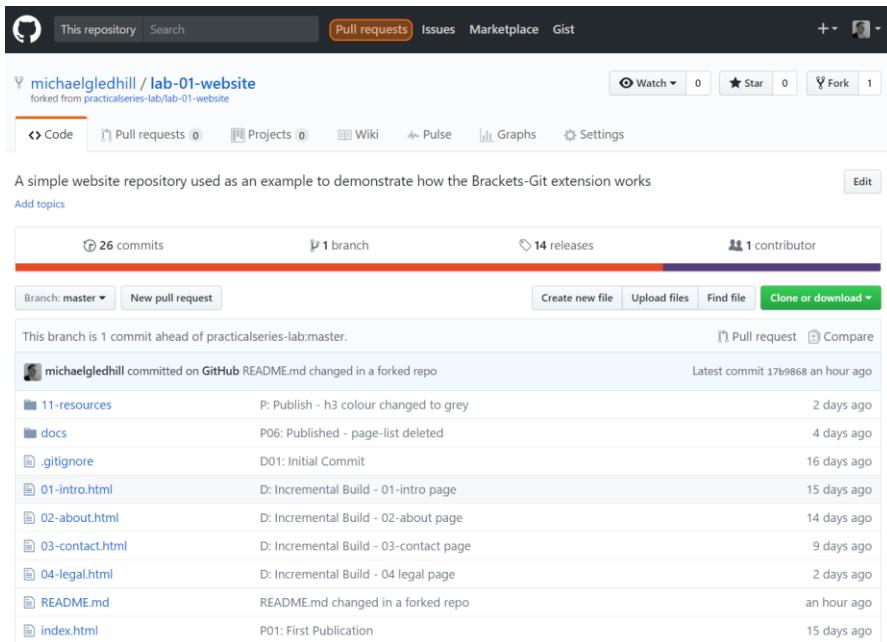


Figure 10.20 Closed pull on the forked repository home page

The [PULL REQUESTS](#) tab only shows pull requests that are made to this repository (if say I added a branch and wanted to merge it back in).

To view pull requests that have been made by this user, click the **PULL REQUEST** link in the black bar at the top (highlighted), it gives the following:

The screenshot shows a GitHub search interface. The search bar contains the query "is:open is:pr author:michaelgledhill". Below the search bar, there are filters: "Created" (selected), "Assigned", "Mentioned", and "Review requests". The results table shows 0 Open and 1 Closed pull request. The closed pull request is described as "#3 by michaelgledhill was closed 17 minutes ago".

Figure 10.21 A user's created pull request page

This page by default shows all the open pull requests I've created. It also tells me that I have **1 CLOSED** request. If I click this I get a list of all the closed pull requests:

The screenshot shows a list of closed pull requests for the repository "practicalseries-lab/lab-01-website". There is one closed pull request, #3, which was closed 17 minutes ago. The message for the pull request is "README.md changed in a forked repo".

Figure 10.22 A list of closed pull requests

Finally, if I click the particular request I see the closed message:

The screenshot shows the detailed view of the closed pull request #3. The title is "README.md changed in a forked repo". The status is "Closed" by "michaelgledhill". The conversation shows a comment from "michaelgledhill" saying "Just a suggestion" and a reply from "practicalseries-lab" saying "No, I don't want to do this.". The pull request has 0 commits and 0 files changed. On the right side, there are sections for Reviewers, Assignees, Labels, and Projects, all of which are currently empty.

Figure 10.23 The specific closed pull requests

This is exactly the same page as Figure 10.19. It opens the pull request page from the [practicalseries-lab](#) repository.

If instead of rejecting the pull request (I clicked **CLOSE AND COMMENT**, Figure 10.18), I wanted to merge the changes into my repository, I would click the **MERGE PULL REQUEST** button (Figure 10.18) and this would merge the changes into the **master** branch of the **practicalseries-lab/lab-01-website** repository and this would be exactly the same process I covered in section 9.5.3 (if there were no conflicts) or section 9.5.5 (if there were conflicts).

10.1.3 Synchronising a forked repository (the limitations)

Ok, this is going to be easy.

The question is:

How can I keep my forked repository in synch with the original?

And the answer is:

You can't—not from GitHub

So that's that.

You should be able to, it's a bit like keeping a local copy of a repository in synch with a remote repository, and you should be able to do the equivalent of a pull to get the latest changes in to the forked copy.

But you can't, GitHub just doesn't have the facility.

There is a way around it by using a local repository as an intermediary, but it's convoluted and confusing and I don't cover it here.

Since I'm not a great fan of forking in the first place; I can't get too worked up over this—it does seem to be a deficiency with GitHub though.

10.2

Issues and milestones

GitHub has a rudimentary issue tracking mechanism built into it. It allows issues to be categorised with *labels* and assigned a completion date (*milestone*).

I will demonstrate the use of issues from the [practicalseries-lab](#) profile using the [lab-01-website](#) repository.

10.2.1 Creating issue labels

From the [lab-01-website](#) repository home page, click the **ISSUES** tab:

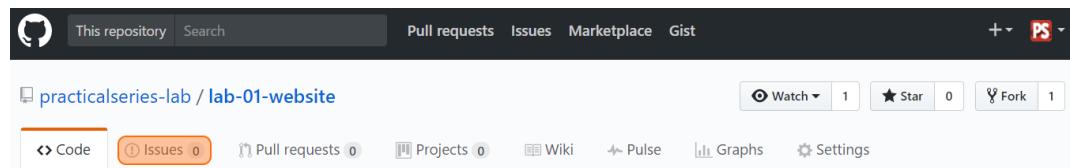


Figure 10.24 Repository home page issues tab

This takes us to the open issues page:

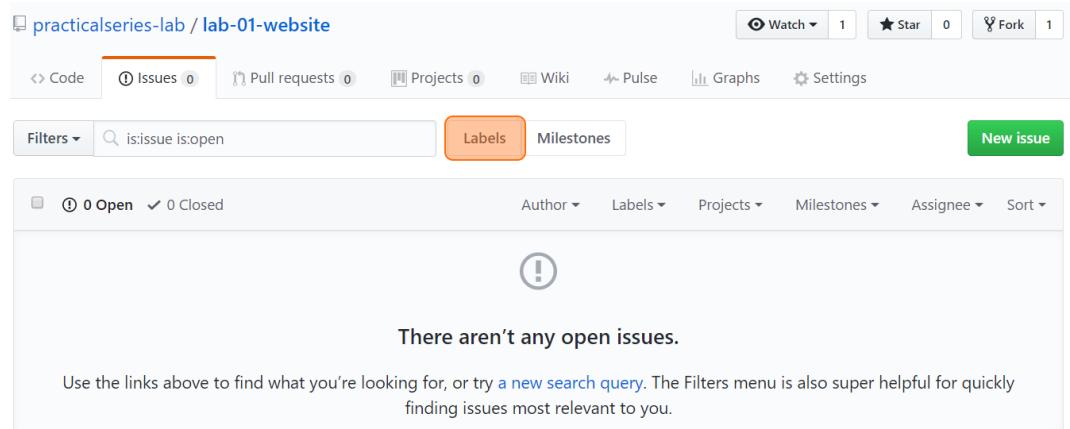


Figure 10.25 Open issues

To create, edit and delete labels click the **LABELS** button (highlighted above):

7 labels			
Sort ▾			
bug	0 open issues		
duplicate	0 open issues		
enhancement	0 open issues		
help wanted	0 open issues		
invalid	0 open issues		
question	0 open issues		
wontfix	0 open issues		

Figure 10.26 Label edit page

Clicking the **DELETE** button on the right hand side will delete the associated label (there is a confirmation box).

The **EDIT** button allows the text and colour of the label to be changed.

The **NEW LABEL** button creates a new label; use it to create a **proofreading** label:

Figure 10.27 Create a label

Enter the required label name and click **CREATE LABEL** to add it to the list.

10.2.2 Creating milestones

Milestones just mark specific dates with an identifying name that can then be attached to an issue.

To create a milestone, click the **MILESTONE** button next to the **LABEL** button on either the issues page (Figure 10.25) or from the label page (Figure 10.26).

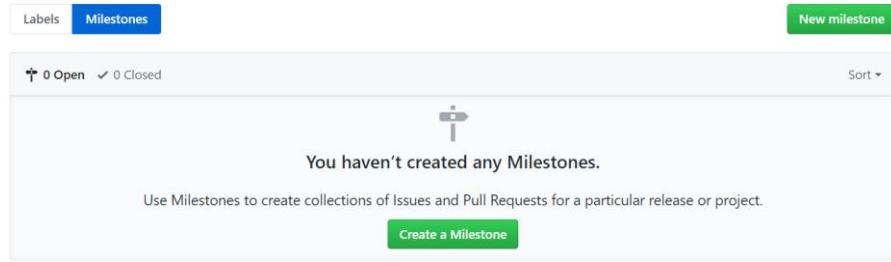


Figure 10.28 Milestone page

Click the **NEW MILESTONE** button to add a milestone:

New milestone

Create a new milestone to help organize your issues and pull requests. Learn more about [milestones and issues](#).

This screenshot shows the 'Create milestone' form. It includes fields for 'Title' (containing 'Release 01'), 'Description' (containing 'Projected first release date for the website. All pages to be complete and tested by this date.'), and a 'Due Date (optional)' calendar. The calendar shows July 2017 with the 14th selected. A green 'Create milestone' button is located at the bottom right.

Figure 10.29 Create milestone page

Add a title ([Release 01](#) in this case)—*note: milestone titles must be unique*, enter a description and give it a date from the calendar. Click **CREATE MILESTONE** to create it. This will show the new milestone on the milestone page:

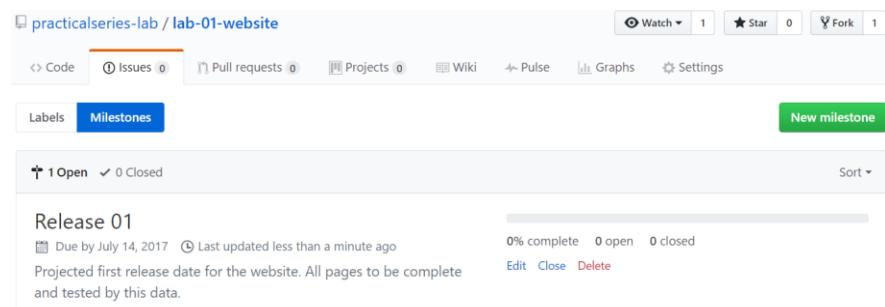


Figure 10.30 Populated milestone page

10.2.3 Creating an issue

Click the **ISSUES** tab to open the issues page, Figure 10.25.

Click the **NEW ISSUE** button to open the create issue page:

The screenshot shows the GitHub interface for creating a new issue. At the top, the repository name 'practicalseries-lab / lab-01-website' is visible along with various status indicators like 'Watch 1', 'Star 0', and 'Fork 1'. The 'Issues' tab is selected. The main area is titled 'README - add getting started section' and contains a text input field with the placeholder 'Add a getting started section to the README.md file.' Below the input field is a note: 'Attach files by dragging & dropping, selecting them, or pasting from the clipboard.' On the right side, there are four sections: 'Assignees' (with a cogwheel icon), 'Labels' (with a cogwheel icon, showing 'Proofreading' is selected), 'Projects' (with a cogwheel icon, showing 'None yet'), and 'Milestone' (with a cogwheel icon, showing 'Release 01'). At the bottom right is a green 'Submit new issue' button.

Figure 10.31 Create issue page

This is fairly straight forward, give the issue a title and add any comment you feel is required. I gave it the title **README.md – add getting started section**.

To assign a user to the issue and to assign labels and milestones just click the relevant cogwheel (highlighted) and choose what you want from the dropdown list.

To create the issue, click the **SUBMIT NEW ISSUE** button. This will take you back to the issue page (Figure 10.32):

practicalseries-lab / lab-01-website

Code Issues 1 Pull requests 0 Projects 0 Wiki Pulse Graphs Settings

README - add getting started section #4

Open practicalseries-lab opened this issue just now · 0 comments

practicalseries-lab commented just now

Add a getting started section to the README.md file.

practicalseries-lab added the **Proofreading** label just now

practicalseries-lab added this to the **Release 01** milestone just now

practicalseries-lab self-assigned this just now

Assignees practicalseries-lab

Labels **Proofreading**

Projects None yet

Milestone **Release 01**

Figure 10.32 Open issues page

This is just a record of what we did when we created the issue.

There is one thing of note here; the `#number` at the top (highlighted), this is the issue number and it's already at 4—“*but why, this is the first issue?*” you say.

The answer is: because pull requests are also part of the issue numbering mechanism and there have already been three of these (one for the simple branch merge, one from the branch merge with conflict and one for the forked repository pull request).

This number is useful; we can use it to close the issue from a commit.

10.2.4 Closing an issue directly

Any issue can be closed from the issues page:

The screenshot shows a GitHub repository page for 'practicalseries-lab / lab-01-website'. The 'Issues' tab is selected, showing one open issue. The issue details page for '#4 README - add getting started section' is displayed. A comment from 'practicalseries-lab' states: 'Add a getting started section to the README.md file.' Below this comment, there are three more actions: adding a 'Proofreading' label, adding it to the 'Release 01' milestone, and self-assigning. On the right side of the issue page, there are sections for Assignees (practicalseries-lab), Labels (Proofreading), Projects (None yet), and Milestone (Release 01). The Notifications section indicates that the user is receiving notifications because they were assigned. At the bottom of the comment section, there is a 'Close and comment' button.

Figure 10.33 Directly closing an issue

Just give a reason in the comment section and click the **CLOSE AND COMMENT** button. This will close the issue.

10.2.5 Closing an issue from a commit

Let's assume we didn't close the comment in the previous section and it's still open; let's also say we want to make the modification suggested (add a getting started section).

Go back to the repository home page, open and edit the `README.md` file and add the following:

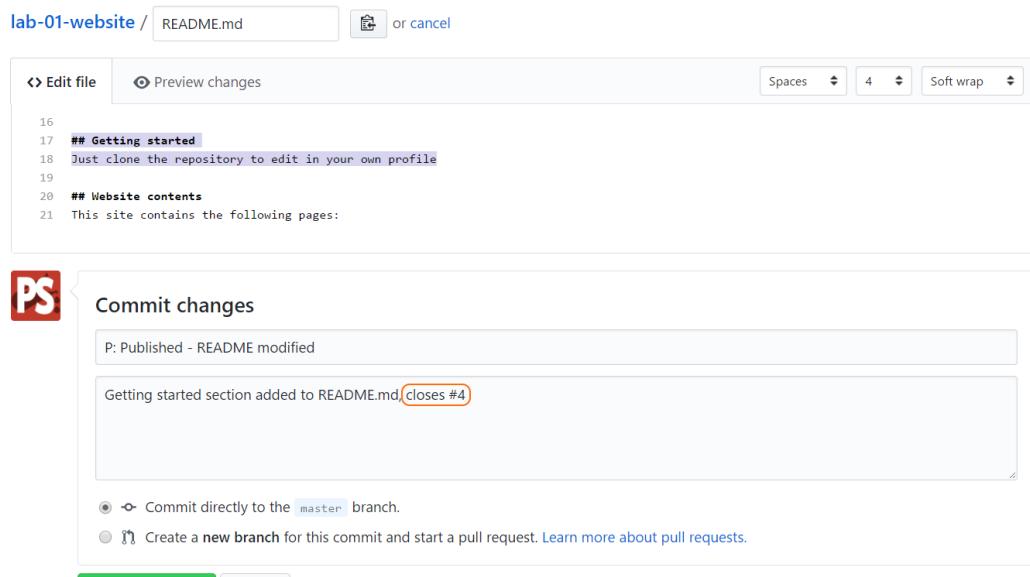


Figure 10.34 Closing an issue from a commit

I've added lines 17, 18 and 19 to the `README.md` file.

I've also added a commit message and an additional comment.

The thing to notice is that in the commit message I've added the text `closes #4`. This is an inline command and it tells GitHub to close the `#4` issue, this was the issue we created in Figure 10.32, the number is at the top, highlighted.

Click `COMMIT CHANGES`.

Go back to the issues page; the issue will now be closed so go to the `CLOSED` issues tab and click the issue itself:

The screenshot shows a GitHub issue page for a repository named 'practicalseries-lab / lab-01-website'. The issue is titled 'README - add getting started section #4'. It is marked as 'Closed' by 'practicalseries-lab' 31 minutes ago with 0 comments. The issue description is 'Add a getting started section to the README.md file.' The issue has a 'Proofreading' label, is assigned to 'practicalseries-lab', and is part of the 'Release 01' milestone. It is also self-assigned.

Figure 10.35 The closed issue

The issue is closed, and the commit number that closed it is shown: [780b895].

Note also that the milestone has been completed, this is because the only issue associated with the milestone is complete (hence the milestone is too).

The inline command `closes` is not the only command that can close an issue, the following all do the same:

- `close`
- `closes`
- `closed`
- `fix`
- `fixes`
- `fixed`
- `resolve`
- `resolves`
- `resolved`

AND FINALLY...

Contacts, acknowledgements, and legal stuff.

CONTACT MICHAEL GLEDHILL

You can reach me by email. I invite questions, corrections, constructive criticism and complaints (polite ones) with the following caveats:

- ① I do have a day job (*surprising isn't it*), I will respond to all polite emails but not necessarily instantly.
- ② I can't offer detailed engineering advice about specific problems (*e.g. why does that valve blow all the fuses when I try to open it*), but I will offer pearls of wisdom about less specific software issues.
- ③ I don't know anything about car engines or kettles so please don't ask.
- ④ For the more emotionally tainted, my dogs, Hector and Henry, are Salukis and while it may be off topic I have a soft spot for them and I may be willing to discuss them if pushed.
- ⑤ If your email comes down to "*I think your website's rubbish, I won't pay for it but I do want to shout at you for a while about your outrageous shortcomings*" then please, there is no need to trouble yourself; you've already said everything by not paying.

So if you're happy with that, you can reach me here: MG@PRACTICALSERIES.COM.

ACKNOWLEDGEMENTS

Thanks to everyone who has paid for this website. Your contributions are appreciated.

Thanks to the dogs: Hector (left) and Henry (right) their advice was invaluable.



Most of the images in the website are my own or of my creation (which probably explains the poor quality).

The fonts used were created by Mr Matthew Butterick and are Equity (serif font), Concourse (sans serif font) and Triplicate (monospace font). If you want to use them you can buy them [HERE](#).

The website uses various third-party software and files, I hope I've adequately and correctly made notification of them on the website and in the source code—I do so again here:

FILE	AUTHOR(S)	PURPOSE
NORMALISE.CSS (4.1.1)	Nicolas Gallagher Jonathan Neal	Conditions various HTML tags so that they are rendered consistently in different browsers
JQUERY.MIN.JS (3.1.0)	jQuery foundation (library hosted by Google)	jQuery library used to run the jQuery scripts
WAYPOINTS.MIN.JS (4.0.0)	Caleb Troughton	Used to detect when to switch to fixed navigation bar (has many other uses)
HYPHENATOR.JS (5.1.0)	Mathias Nater	Dynamically hyphenates text within a web page
RUN-PRETTIFY.JS (04-03-2013)	Google	Used to display code fragments within an HTML page
LIGHTBOX.JS (2.8.2)	Lokesh Dhakar	Used to overlay images on top of the current page
MATHJAX.JS (2.6.1)	MathJax	Renders AsciiMath, TeX/LaTeX and MathML code as standard mathematical notation

Third-party software running on the website

Normalise, Hyphenator, Lightbox and Waypoints are available under the GitHub MIT licence (reproduced below) and jQuery, MathJax and Prettify are available under the Apache 2.0 licence (also reproduced below):

GITHUB MIT LICENCE

The MIT License (MIT)

Copyright © [Authors as listed in the above table]

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Github Massachusetts Institute of Technology (MIT) Licence

APACHE 2.0 LICENCE & REFERENCE

GOOGLE LICENCE

Copyright [2015] [google]

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at:

[HTTP://WWW.APACHE.ORG/LICENSES/LICENSE-2.0](http://www.apache.org/licenses/LICENSE-2.0)

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Apache Licence

COLOPHON



This website is part of the Practical Series of publications. It was designed and published in the United Kingdom by Michael Gledhill.
It was first published in April 2017.

The website is printed mainly in the Equity typeface (serif), with headings, tables and annotation in Concourse (sans serif), and code fragments in TriPLICATE (monospaced).

The fonts have been created by Mr Matthew Butterick and are available [HERE](#).

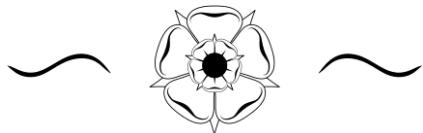


LEGAL

© 2017 Michael Gledhill. All rights reserved.

The downloadable source code and examples, I make available to all who wish to use them. The text and images within the website may not be publicly reproduced without my written permission. In the unlikely event that you do wish to do so, please send requests to me at: MG@PRACTICALSERIES.COM.

I have not been compensated to recommend any software, websites or products mentioned on these pages or used in the production of this website.



APPENDICES

The appendices are a collection of useful information that doesn't fit directly in the main body of the text; there are five of them and they broadly cover:

APPENDIX	PURPOSE
A	Downloading Brackets and some useful extensions
B	A best practice guide for version numbering
C	A guide to markdown syntax (general) and Git Flavoured markdown
D	Git command line command summary
E	Advanced topics: 1. Understanding Rebase

Currently there is only one entry in the advanced topics section and that is a discussion of how to use the Git [rebase](#) function. I will add further topics to this section as I think of them.

A

INSTALLING BRACKETS

How to install Brackets and its extensions.

A.1

The Brackets text editor

Brackets is my text editor of choice.

I used Brackets to write the entire HTML, CSS and jQuery code within this website.

Brackets lets you add extensions that do certain things: auto-save, additional menu functions, automatically add browser prefixes &c.

A.1.1 Getting and installing Brackets

Brackets is available from the BRACKETS.IO website. I'm currently using version 1.9. It's pretty easy to get; just click the big blue download button (highlighted in orange in Figure A.1).

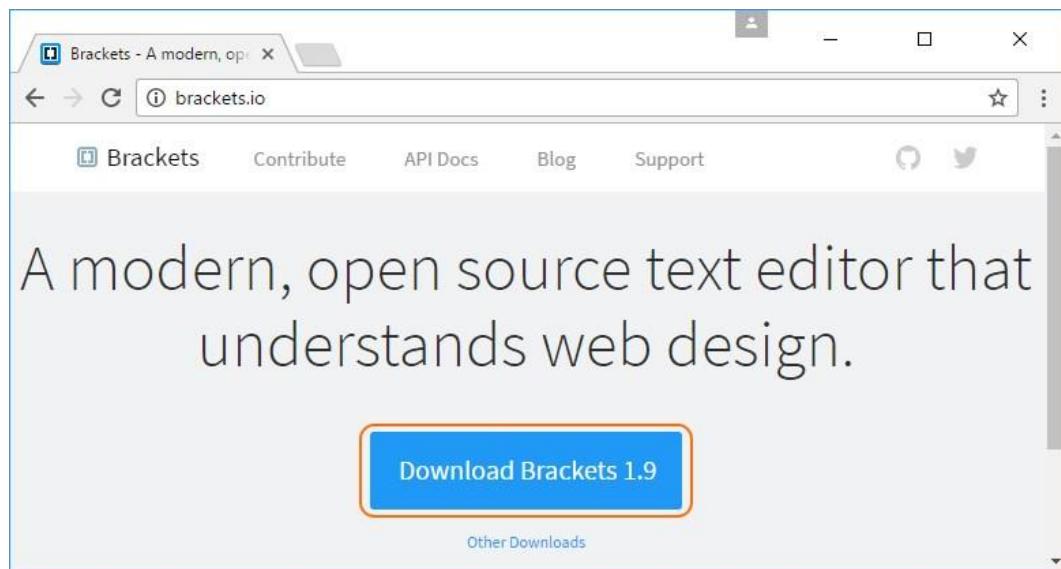


Figure A.1 Brackets text editor download

This will download the [brackets.release.1.9.msi](#) file.

Run this file and leave all the default options selected (Figure A.2):

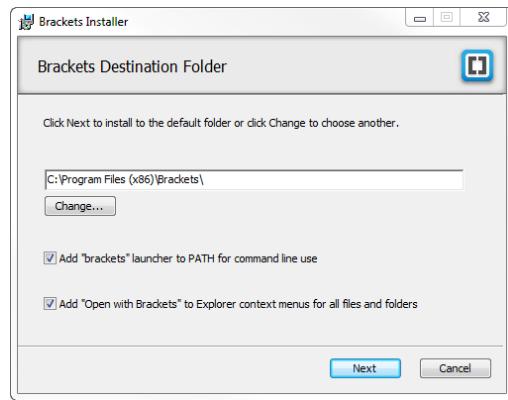


Figure A.2 Brackets text editor install options

After the installation is complete, start Brackets; it will look something like Figure A.3:

```
1 <!DOCTYPE html>
2 <html>
3
4   <head>
5     <meta charset="utf-8">
6     <meta http-equiv="X-UA-Compatible"
7       content="IE=edge">
8     <title>GETTING STARTED WITH BRACKETS</title>
9     <meta name="description" content="An
10       interactive getting started guide for
11       Brackets.">
12     <link rel="stylesheet" href="main.css">
13   </head>
14   <body>
15
16   <!--
17     MADE WITH <3 AND JAVASCRIPT
18   -->
19
20   <p>
21     Welcome to Brackets, a modern open-source
22       code editor that understands web design.
23       It's a lightweight,
24       yet powerful, code editor that blends
25       visual tools into the editor so you get
26       the right amount of help
27       when you want it.
28   </p>
29
30   <!--
31     WHAT IS BRACKETS?
32   -->
33   <p>
34     <em>Brackets is a different type of
35       editor.</em>
36     Brackets has some unique features like
37       Quick Edit, Live Preview and others that
38       you may not find in other
39       editors. Brackets is written in
40       JavaScript, HTML and CSS. That means that
41       most of you using Brackets
42       have the skills necessary to modify and
43       extend the editor. In fact, we use
44       Brackets every day to build
```

Figure A.3 Brackets first use

A.2

Adding extensions to Brackets

Right, we've installed Brackets and now we can install some *extensions*. Extensions are "plugins" that add extra functionality to Brackets, two come pre-installed with Brackets, the first is the *live preview function* (this is the lightning bolt icon in the right hand side bar, Figure A.4).

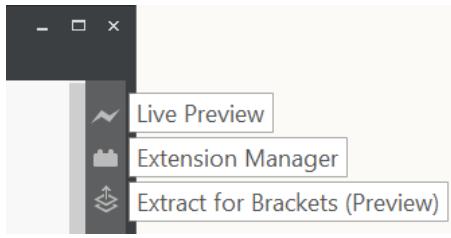


Figure A.4 Brackets sidebar

The live preview opens a new Chrome window and directs it to the current web page being edited in Brackets, the browser preview is always up to date, any changes made to the file in Brackets are instantly reloaded in the live preview (no page reloads are required).

The live preview updates the chrome browser in real time whenever changes are made to HTML or CSS files, changes to other file types (Java script for example) cause the preview page to be reloaded when the file is saved.

The second pre-installed extension is "*Extract for Brackets (preview)*" (it's the two overlapping diamonds with the arrow in the middle—right hand sidebar, bottom icon Figure A.4). This allows content to be imported from another Adobe product: Photoshop CC.

I don't use Photoshop (*it's one of the least intuitive and hard to use programmes I've come across*) and therefore I don't use this extension.

Note: On earlier releases, it was possible to install Brackets without the Extract extension (I don't see this option now in the latest version—it used to be accessible from the "other downloads" link below the blue button). It is possible to remove the Extracts extension using the extensions manager (see below).

Brackets has a fairly extensive library of extensions available to it. Most of these are written by third-part developers (rather than Adobe) and I've installed several that I think are useful.

These are:

EXTENSION	SEARCH FOR	FUNCTION
Autoprefixer	Autoprefix	Automatically adds browser prefixes to CSS code (all those -webkit things that nobody understands)
Autosave files on Window Blur	Autosave	Automatically saves all the files open in Brackets whenever you click out of the Brackets window
Brackets icons	Icons	Adds little icons to the side of each file in the file tree showing its type (HTML, CSS, JS &c.)
Copy as HTML	Copy HTML	Copies code to the clipboard and preserves the colour coding used by Brackets (I used it to copy code from Brackets into documents)
Interactive Linter	Interactive	Analyses the HTML, CSS and Java script code and highlights errors in the syntax and structure—a process referred to as “linting” ¹
Pop-up menu Brackets	Pop-up	Adds case conversion functionality to the pop-up menu that appears when text is right clicked in the Brackets window. It adds uppercase, lowercase and camel case ² conversions

Table A.1 My selection of Brackets extensions

¹

Linting: (present participle of the verb **to lint**—yes I know, I think *lint* is the stuff you get off old jumpers too; and it's definitely not a verb) is the process of automatically checking software for errors—this process is usually carried out by another software programme called (*would you believe it*) a “linter”. Apparently it started with C code and there is a whole Wikipedia article on it [HERE](#).

²

Camel case is the practice of joining words together and capitalising the start of each word (CamelCase for example), its more formal name is *medial capitals*.

To add extensions to Brackets click the Extensions Manager icon (it's the second icon in the right hand side bar; it looks like a Lego brick, Figure A.4). This opens the extension manager dialogue box (Figure A.5):

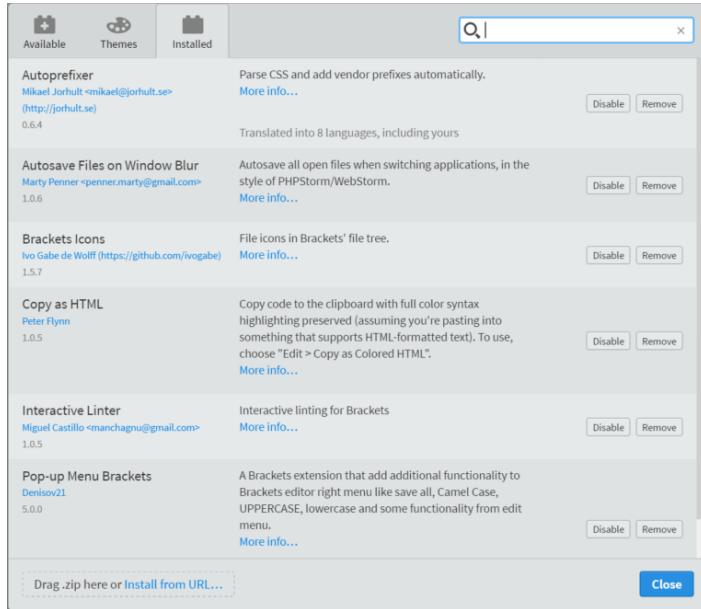


Figure A.5 Brackets extension manager

To find the extensions, click the **AVAILABLE** tab and enter the “*search for*” terms listed in Table A.1 in the search box at the top; select the actual extension from those presented by the search and click **INSTALL** on the right hand side.

Note: *It is my experience that Brackets should be closed and re-started after each extension is installed. It can tie itself in knots if you don't.*

The *Copy as HTML* extension is not required for coding (the others all make coding easier apart from the icon thing, that just makes it look nice), I use it purely to copy code from Brackets and paste it into other documents (it preserves all the syntax colour coding). It's how I got the code fragments from Brackets into this document.

Uninstalling an extension is much the same, click the **INSTALLED** tab and then click the **REMOVE** button next to the extension you want to uninstall. It will require Brackets to close and re-open. This is how to uninstall the Extracts extension if you don't want it.

B

A REVISION NUMBERING MECHANISM

A revision numbering strategy for a website development project under the Git and GitHub version control systems.

THIS SECTION describes a revision numbering strategy for a website development project under the Git and GitHub version control systems.

Git and GitHub use commit numbers derived from the checksum of files being added to the repository. These appear at best to be seemingly random seven digit hexadecimal numbers. They do not represent a meaningful number that is useful for team members trying to identify a revision path.

Git and GitHub allow any commit point to have an associated tag, this is entirely at the discretion of the user and (*other than the requirement of being unique*) can be anything at all.

This allows each commit point to be tagged with a more meaningful (*semantic*) version number. Something that makes sense to humans.

The following is a proposed mechanism for just such a workflow and tagging arrangement.

B.1

Workflow arrangements

The workflow consists of a single main branch, the **master** branch.

The **master** branch (after some initial development work to establish it) will only contain either finished publishable work or a completed and released website.

Publishable work is a section of the website that is complete in itself—it does not indicate that the whole website is finished, just that the section in question is complete, tested and deployable.

A website release indicates that the whole website is finished. Release R01 will be active when the entire website and all its pages is published for the first time.

Development work can take place at any time and will always take place on a separate branch. Development branches always spur from some published or released point on the master branch.

A development branch must have a very restricted scope. E.g. a single section of the website (or just a page).

Generally, a development branch will contain all the things associated with that section: **html**, **js** and **css** files as well as associated images and data.

When the development is complete and tested, it will be merged back to the **master** branch, the merge point will be a published or released commit (Figure B.1):

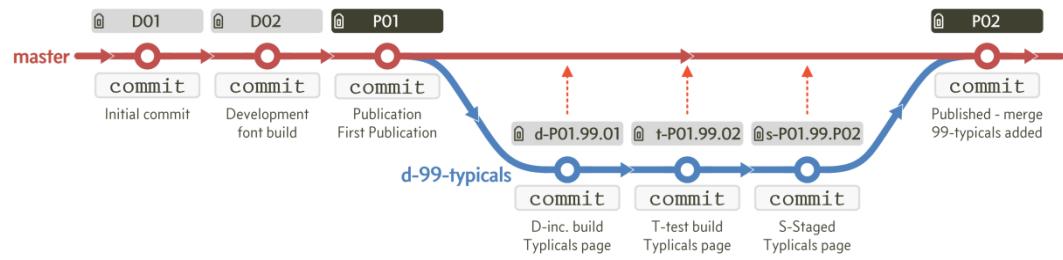


Figure B.1 master branch workflow

B.2

Master branch revision states

The project progresses through various different states along the **master** branch.

Each state is given a letter that represents the condition of a particular commit point.

STATE	EXAMPLE	MEANING
D	D02	Development The site is entirely under development and no part of it has been published
P	P04	Published (partial) Partial publication—certain pages of the website are finished and have been published
R	R01	Released Released—all pages are complete and the whole site is finished

Table B.1 Major revision point

Master branch states have the following format:

SNN

Where **S** is the state letter (Table B.1):

- D — Development
- P — Published (partial)
- R — Released

NN is a number; this starts at 01 for each particular state and is incremented by one for each subsequent issue.

E.g. D01 → D02 → D03 → P01 → P02 → P03 → R01 → R02 &c.

B.3

Development branch revision states

Development can take place along the **master** branch until some suitable (publishable) point is reached.

At this point the site has been published, the site might not be complete (with the Practical Series Web Development site I tended to write a section, test it and then publish that section), but no further development work can take place on the **master** branch.

This leads to the website being developed and published in stages; however, any code on the **master** branch at a published or released commit point must be finished, fully tested, working and deployable.

Development work always takes place on a separate **development** branch. Each **development** branch is taken from the latest published or released version of the software on the **master** branch. The name given to a **development** branch is:

`d-XX-name`

Where `d-` indicates the branch is for development (all my branches are development branches and always start with `d-`).

`XX` refers to the number of the web page or section being developed. For example, the PS Web Development is split into sections and each section is given a two digit number. It starts like this:

- `00-index` and common files—css/fonts &c.
- `01-introduction`
- `02-fairfax`
- `03-grid`

In Figure B.1 the first development branch is called `d-99-typicals`. This is a page that contains an example of all the possible sections, headings, tables, images and advanced functions such as code fragments, formulae and lightbox imagery.

It is a good starting point for any other page that may be required and it makes sense to develop this page first.

All the development work takes place on the `development` branch. There will be multiple commits made on this branch; these will mostly be incremental builds (an incremental build is just a point at which the work was committed, the work wasn't finished; I may just have committed the code at that point because it was the end of the day).

Each development commit on a `development` branch is tagged, and the tag has the format:

`d-SNN.XX.YY`

Where `SNN` is the state of the commit point on the `master` branch when the `development` branch was created (i.e. the parent commit point to which the branch is linked).

`XX` refers to the number of the web page or section being developed (this is the same as the `XX` in the branch name, see above).

`YY` is a number starting at 01 and is incremented by one for each commit along the branch.

Once the code has been developed, it progresses to a test stage; this takes place along the same branch. The commit tag changes however, it now becomes:

`t-SNN.XX.YY`

It is now preceded by a `t-` instead of a `d-`. The number `YY` continues at the next number following the last development commit when the test stage starts. This allows tested code to drop back into development. The following sequence illustrates this:

`d-P01.02.01 → t-P01.02.02 → d-P01.02.03 → t-P01.02.04 → s-P01.02.P02`

Finally, at the end of testing, the code is finished and is ready to be merged back into the **master** branch. At this point, a new commit point is made on the branch. This is referred to as a *Staged* commit. It again has a tag and this time the tag format is:

s-SNN.XX.SNN₊k

The **s-** indicates a staged commit. **SNN.XX** is exactly the same as each other commit on the branch.

SNN₊k indicates the new merge commit point on the master branch that will be created when the development branch is merged onto the master branch. I.e. the point at which the branch re-joins the **master**.

Normally, this will just be the **SNN** value plus one (it may be more if other branches are involved, it will always be the next publication or release commit number on the **master** branch, see § B.4).

In Figure B.2, the development branch **d-99-typicalas** split from the **master** branch at commit point **P01** (**SNN** is **P01**). After the development process, the code is ready to merge back onto the **master** branch. The next publication commit point on the **master** branch will be **P02** (**NN** goes up by 1). So the last staged commit on the development branch will be **s-P01.99.P02**.

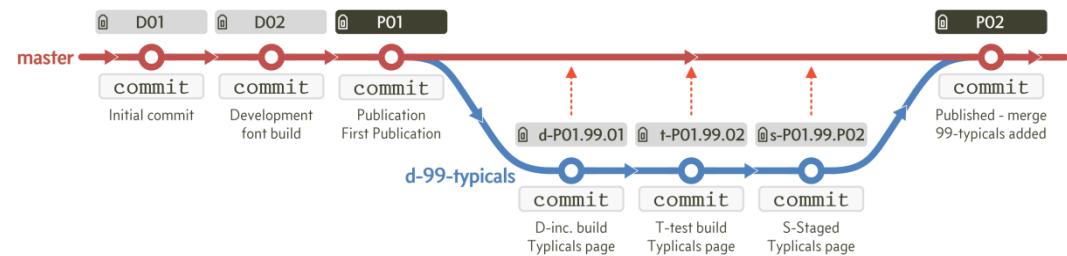


Figure B.2 development branch workflow

The final commit on the development branch, the *staged* commit (**s-P01.99.P02** in this case) is done to allow the software versions of each file to be changed to match the forthcoming **master** branch commit (**P02** in this case). The code itself shouldn't change, just its revision status.

There is only ever one staged commit on a development branch.

B.4

Parallel development branches

It is perfectly possible to have two simultaneous development branches:

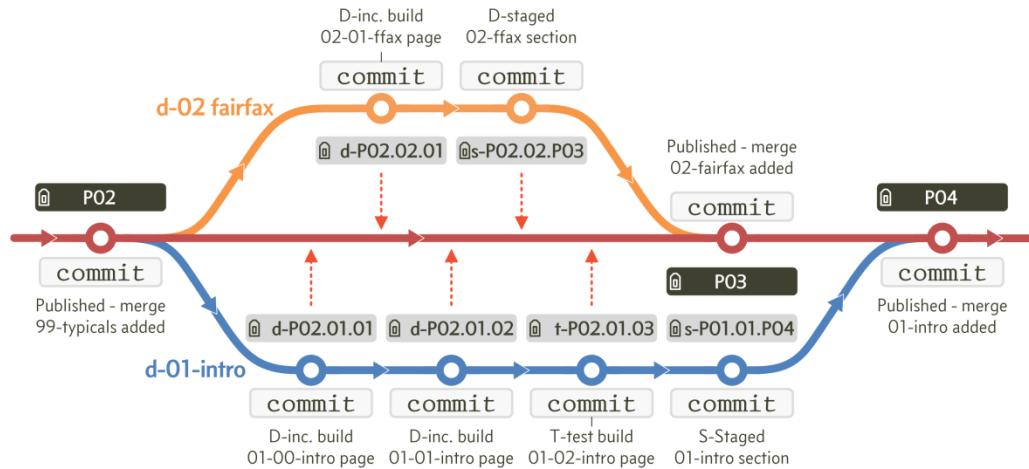


Figure B.3 Parallel development branches

Note: *With parallel branches, it does not matter what order the branches are made or what order they collapse, here **d-01** is created first, but **d-02** is merged back into the **master** before it.*

This type of arrangement can appear slightly confusing when all the branches are merged back onto the master branch, it looks like this (I've coloured the tags with the appropriate branch colour):

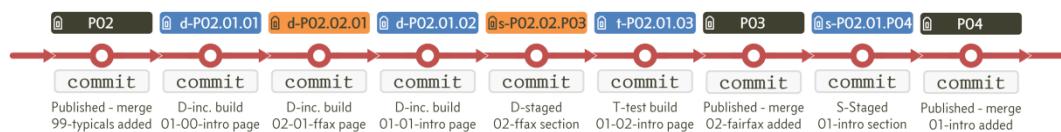


Figure B.4 Merged parallel development branches

It makes sense if you think about it; commits are listed in order of the time they were applied.

B.5 Nested development branches

It is possible to have a development branch from another development branch (referred to as nesting). Nested branches always merge back onto their parent branch:

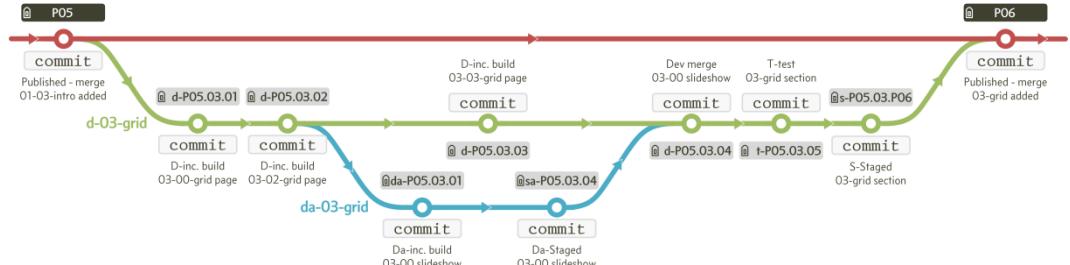


Figure B.5 Nested development branches

The second development branch is called `da-<as parent branch>`. The third would be called `db-<as parent branch>, da, db, dc &c.`

Each commit on the nested branch has the format:

`d/t/sL-SNN.XX.ZZ`

Where `L` is the branch identifier (`a, b, c, &c.`).

`SNN-XX` is identical to the parent branch (`master` branch commit and web page or section number).

`ZZ` is a number starting at 01 and is incremented by one for each commit along the nested branch.

Nested branches can have test revisions (`ta, tb, tc, &c.`) and a staged commit (`sa, sb, sc &c.`).

The staged commit reflects the merged commit number on the parent branch. There is only one staged commit per nested branch.

B.6

A note on numbering

All the numbering I've shown in this section has been two digits [d-P01.03.05](#). This clearly limits any revision, page or section number to a maximum value of 99.

This needn't be the case; it is perfectly possible to use three digit numbers [d-P001.003.005](#), indeed this could be selectively adapted: [d-P01.0103.005](#), you can do whatever you like.

I think the two digit version is enough for me. If I've got more than 99 commits on a development branch I'm probably doing it wrong.

I can see the advantage in having a four digit middle number [d-P01.0103.05](#), it would allow me to resolve to individual web pages; however, I nearly always develop a full section at a time. I write it in Word and then encode it in HTML.

I suggest you start with the two digit version and see how you get on.

B.7 A note on commit messages

Commit messages should have a short (less than 50 characters) first line. In Brackets, where extended commits are used, always leave a blank line after the first line (this ensures the first line is treated as a title).

Always include the status of the commit in the title:

- D — Development
- P — Published (partial)
- R — Released

Generally keep the remaining lines short (less than 75 characters).

Within the commit message list every file that was altered in the commit (list them separately on each line).

When closing issues with commits (see § 10.2.5) provide a separate line within the commit message indicating which issue was closed and why (don't put all the close command in the title line); e.g.:

```
P: Published - Minor corrections

index.html modified      - closes #4
styel.css modified       - closes #5 and closes #6
01-intro modified        - closes #8
```

C

MARKDOWN

Markdown and Git Flavoured Markdown.

C.1

General markdown syntax

C.1.1 Headings

SYNTAX	APPEARANCE IN GITHUB
Normal text	Normal text
# h1 heading	<h1>h1 heading</h1>
## h2 heading	<h2>h2 heading</h2>
### h3 heading	<h3>h3 heading</h3>
#### h4 heading	<h4>h4 heading</h4>
##### h5 heading	<h5>h5 heading</h5>
###### h6 heading	<h6>h6 heading</h6>
Normal text	Normal text

C.1.2 Block quotes

SYNTAX	APPEARANCE IN GITHUB
Normal text	Normal text
> Yet beautiful and bright > He stood > As born to rule the storm	Yet beautiful and bright He stood As born to rule the storm

C.1.3 Emphasis

SYNTAX	APPEARANCE IN GITHUB
Normal text	Normal text
This is italics	<i>This is italics</i>
This is also italics	<i>This is also italics</i>
This is bold	This is bold
__This is also bold__	This is also bold
They *can **be** combined*	They <i>can be combined</i>
This *is **also combined***	<i>This is also combined</i>

C.1.4 Ordered lists

SYNTAX	APPEARANCE IN GITHUB
Normal text	Normal text
1. First entry 2. Second entry 3. Third entry	1. First entry 2. Second entry 3. Third entry
1. First entry 2. Second entry 1. First sub-entry 2. Second sub-entry 3. Third entry	1. First entry 2. Second entry i. First sub-entry ii. Second sub-entry 3. Third entry

C.1.5 Unordered lists

SYNTAX	APPEARANCE IN GITHUB
Normal text	Normal text
* First entry * Second entry * Third entry	<ul style="list-style-type: none">• First entry• Second entry• Third entry

SYNTAX	APPEARANCE IN GITHUB
* First entry * First sub-entry * Second sub-entry * Second entry * Third entry	<ul style="list-style-type: none">• First entry<ul style="list-style-type: none">◦ First sub-entry◦ Second sub-entry• Second entry• Third entry

C.1.6 Combined lists

SYNTAX	APPEARANCE IN GITHUB
1. First entry 2. Second entry * First sub-entry * Second sub-entry 3. Third entry	<ol style="list-style-type: none">1. First entry2. Second entry<ul style="list-style-type: none">◦ First sub-entry◦ Second sub-entry3. Third entry
* First entry * Second entry 1. First sub-entry 2. Second sub-entry * Third entry	<ul style="list-style-type: none">• First entry• Second entry<ul style="list-style-type: none">i. First sub-entryii. Second sub-entry• Third entry

C.1.7

Images & links

![Logo] (/11-resources/02-images/readme.png)

![Alt text] ([URL](#))



[GitHub] (<http://github.com>)

[GitHub](#)

[Link text] ([URL](#))

Images can also be added by using basic HTML¹, this allow the size and position of the image to be adjusted:

```
<p align="center">
    
</p>
```



Note: *README.md files support SVG images if they are referenced as a HTML link (e.g. <http://practicalseries.com/1002-vcs/11-resources/02-images/02-build-status/build-badge.svg>).*

If they are given as a relative address: /11-resources/02-images/02-build-status/build-badge.svg, they will not be displayed.

The reason appears to be some security issue “cross site scripting vulnerabilities”—whatever that means?

¹

In the case of markdown, basic HTML is everything in HTML except the style attribute.

C.2

Git flavoured markdown syntax

GitHub flavoured markdown can use all the general markdown syntax and all of the following additions:

C.2.1 Code fragment

SYNTAX	APPEARANCE IN GITHUB
Normal text	Normal text
```	
<p align="left">Any text</p>	<p align="left">Any text</p>
```	
```html	
<p align="left">Semantic colouring</p>	<p align="left">Semantic colouring</p>
```	
Inline code ```git init``` fragment	Inline code <code>git init</code> fragment

The three ` `` ` characters are back-ticks **ALT+96**; follow these with a language identifier to give semantic colouring. See [HERE](#) for a list of language identifiers.

C.2.2 Task lists

SYNTAX	APPEARANCE IN GITHUB
Normal text	Normal text
- [x] Completed task	<input checked="" type="checkbox"/> Completed task
- [] Open task	<input type="checkbox"/> Open task
- [x] Completed task	<input checked="" type="checkbox"/> Completed task
- [] Open task	<input type="checkbox"/> Open task

C.2.3 Tables

SYNTAX	APPEARANCE IN GITHUB						
Normal text	Normal text						
Header 1 Header 2 ----- ----- Cell A1 Cell B1 Cell A2 Cell B2	<table border="1"><thead><tr><th>Header 1</th><th>Header 2</th></tr></thead><tbody><tr><td>Cell A1</td><td>Cell B1</td></tr><tr><td>Cell A2</td><td>Cell B2</td></tr></tbody></table>	Header 1	Header 2	Cell A1	Cell B1	Cell A2	Cell B2
Header 1	Header 2						
Cell A1	Cell B1						
Cell A2	Cell B2						

Note there is a space before and after the bar character (|).

To split a heading onto two lines use
 where you want the break to occur.

C.2.4 Special GitHub references (issues and users)

SYNTAX	APPEARANCE IN GITHUB
Normal text	Normal text
@mgledhill link to a user	<u>@mgledhill</u> link to a user
#1 issue to be resolved	#1 issue to be resolved

C.2.5 Escape characters

Escape characters are used to display those characters that would otherwise have special meaning in markdown (putting asterisks around a word for example). Escaping a character is done by putting a reverse oblique (\) in front of it:

SYNTAX	APPEARANCE IN GITHUB
Normal text	Normal text
asterisk around a phrase	*asterisk around a phrase*

The following characters can all be escaped:

\, \` , *, _, \#, \+, \-, \., \!, \[, \], \{, \}, \(, \)

D

GIT AND COMMAND LINE

A summary of Git and command line instructions.

D.1 Git command line summary

CREATE A REPOSITORY

```
$ git init
```

Create a new local repository

```
$ git clone git@github.com:<repo-name>
```

Clone an existing repository from GitHub

STATUS AND HISTORY

```
$ git status
```

Show the status of a repository

```
$ git diff
```

Show changes in tracked files

```
$ git log
```

Show all commits, starting with newest in verbose form

```
$ git log --oneline
```

Show all commits—one commit per line

```
$ git log --oneline --graph
```

Show all commits—one commit per line with branch graph

```
$ git log -p <file>
```

Show changes for a specific file

```
$ git blame <file>
```

Who change what and when in a specific file

GIT ADD AND COMMIT

```
$ git add .
```

Add all current changes to the staging area

```
$ git add <file>
```

Add all current changes in a file to the staging area

```
$ git commit
```

Commit all staged changes (will prompt for a message)

```
$ git commit -m "commit message"
```

Commit all staged changes with a message

```
$ git commit -a
```

Stage and commit all changes (will prompt for a message)

```
$ git commit -am "commit message"
```

Stage and commit all changes with a message

BRANCHES

```
$ git branch -av
```

List all branches

```
$ git branch <new-branch>
```

Create a new branch based on the current `head`

```
$ git checkout <branch>
```

Switch to a different branch

```
$ git branch -d <branch>
```

Delete a branch

```
$ git checkout --track <remote/branch>
```

Create a new tracking branch based on a remote branch

TAGS

```
$ git tag --list
```

List all tags

```
$ git tag <tag-name>
```

Tag the current commit point with a lightweight tag

```
$ git tag -a <tag-name> "annotation"
```

Tag the current commit point with an annotated tag

```
$ git tag -d <tag-name>
```

Delete a tag

```
$ git tag <tag-name> <commit-hash>
```

Tag a specific commit hash

RESET AND CHECKOUT

```
$ git reset --hard HEAD
```

Discard all local changes in your working directory

```
$ git reset --hard <commit>
```

Reset to a specific commit point (discards working changes)

```
$ git reset --mixed <commit>
```

Reset to a specific commit point (preserving unstaged changes)

```
$ git reset --soft <commit>
```

Reset to a specific commit point (preserving uncommitted changes)

```
$ git reset <tag-name> --hard
```

Reset to a specific tag (discards working changes)

```
$ checkout -b <newbranch> <commit>
```

Create a new branch based on a previous commit point

MERGE

```
$ git merge <branch>
```

Merge <branch> in to the current branch

```
$ git mergetool
```

Start the configured merge tool

UPDATE AND PUBLISH

```
$ git remote add <connection-name> <url>
```

Add new remote repository, named <connection-name>

```
$ git remote -v
```

List all configured remotes

```
$ git remote show <remote>
```

Show specific information about a remote

```
$ git fetch <remote>
```

Download all changes from <remote>, but don't integrate into head

```
$ git pull
```

Download changes and directly merge into current branch

```
$ git pull <remote> <branch>
```

Download changes from specific repository and branch, directly

```
$ git push
```

Publish local changes on current branch to the current remote

```
$ git push <remote> <branch>
```

Publish local changes to specific repository and branch

```
$ git branch -dr <remote/branch>
```

Delete a branch on the remote

D.2

Git Bash instruction summary

DIRECTORIES AND TERMINAL

\$ `clear`

Clears the terminal window

\$ `pwd`

Display path of current working directory

\$ `cd <directory>`

Change directory to <directory>

\$ `cd ..`

Change to parent directory

\$ `rm -r <directory>`

Delete <directory>, use `-rf` to force delete <directory>

\$ `ls`

List directory contents (`ls` can be replaced with `dir`)

\$ `ls -la`

List detailed directory contents, including hidden files
(`ls` can be replaced with `dir`)

\$ `mkdir <directory>`

Create new <directory>

SEARCH

\$ `find <dir> -name "<file>"`

Find all files named <file> inside <dir> (use wildcards `*`, `?` to search for parts of filenames)

\$ `grep "<text>" <file>`

Output all occurrences of <text> inside <file> (add `-i` for case-insensitivity)

\$ `grep -rl "<text>" <dir>`

Search for all files containing <text> inside <dir>

FILES

\$ `rm <file>`

Delete <file>

\$ `rm -f <file>`

Force delete <file> (if it is in use)

\$ `mv <file-old> <file-new>`

Rename <file-old> to <file-new>

\$ `mv <file> <directory>`

Move <file> to <directory> (will overwrite an existing file)

\$ `cp <file> <directory>`

Copy <file> to <directory> (will overwrite an existing file)

\$ `cp -r <directory1> <directory2>`

Copy <directory1> and its contents to <directory2> (will overwrite)

\$ `touch <file>`

Create file

OUTPUT

\$ `cat <file>`

Output the contents of <file>

\$ `less <file>`

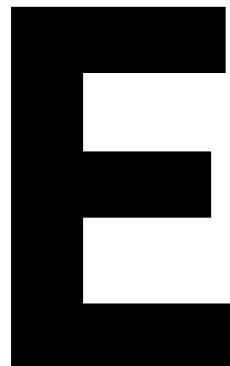
Output the contents of <file> with pagination

\$ `head <file>`

Output the first 10 lines of <file>

\$ `<cmd> > <file>`

Direct the output of <cmd> into <file>
use `>>` instead to append the output of <cmd> to <file>



GIT ADVANCED TOPICS

Some advanced topics and information.

E.1 Rebase

Rebase is another way of merging branches together. It does so without creating a merge commit and people like it because it gives a less complicated workflow (*it doesn't have all the merge commits*).

It gets talked about a lot in Git polite society; generally because it is quite difficult to come to terms with.

I'm going to say right from the start that:

I don't like REBASE—I recommend you don't use it

Rebase is a little bit dangerous, it can leave unattached commit points floating around in the repository—just like resetting to an earlier commit and carrying on from there (I covered this in § 2.5.3). It can also change a commit and effectively create a replacement for it.

From an engineering point of view neither of these are good things, they spoil the traceability (someone could always use an unattached commit).

E.1.1 **Rebase—the rules**

There are some rules for using the rebase. The first one is an absolute; you will read about it wherever rebase is discussed, it is this:

REBASE—THE GOLDEN RULE

Never use rebase with a branch that is available on a public repository.

If you want to merge two branches together using rebase and either branch is available in a remote repository (if, for example, it is stored on GitHub) then never use rebase.

Secondly, my rule:

REBASE—THE GLEDHILL RULE

Just don't use rebase

Rebase is confusing; it changes what you think is a commit point to something else. Where remote repositories are concerned, this can get you into a lot of trouble with your colleagues—they will wonder where their commit point has gone and then they will shout at you.

All that said, this is how rebase works and how to use it:

E.1.2 Understanding rebase

To demonstrate the rebase, I will construct a simple repository with just one file in it.

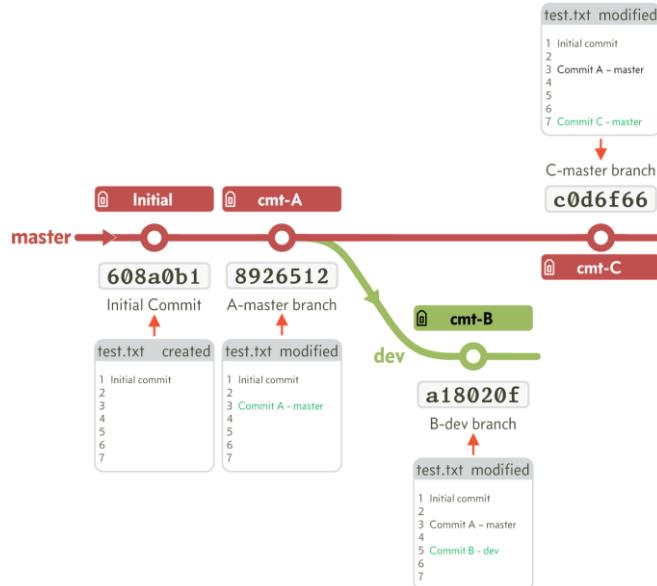


Figure E.1 A simple workflow with two branches

I created a file `test.txt` with seven lines, the first line being `Initial commit`. It was created on the **master** branch and committed with commit number `[608a0b1]`.

The idea is that I will add extra text to the file at each commit and I will do this in two branches. I will then combine the branches using the `rebase` function.

After the initial commit, I make a second commit on the **master** branch. In this commit I modified line 3 of `test.txt` and added the text `Commit A-master branch`. This commit point has the number `[8926512]` and is tagged `cmt-A`.

Next I made a new **dev** branch from the **master** branch at the `cmt-A` commit point.

On the new **dev** branch I made a third commit, this time modifying line 5 to include the text `Commit B-dev branch`. This third commit point has the number `[a18020f]` and is tagged `cmt-B`.

Finally on the **master** branch I made a fourth commit, this time modifying line 7 to include the text `Commit C-master branch`. This fourth commit point has the number `[c0d6f66]` and is tagged `cmt-C`.

The full arrangement of commits and the contents of `text.txt` at each point can be seen in Figure E.1.

In Brackets the **master** branch looks like this:

```
test.txt (lab-02-rebase) - Brackets
File Edit Find View Navigate Debug Git Help
Working Files test.txt
lab-02-rebase master
.gitignore test.txt

1 Initial commit
2
3 Commit A - master branch
4
5
6
7 Commit C - master branch

P 2017-05-25 11:39:58 by practicalseries-lab C - master branch
P 2017-05-25 11:36:42 by practicalseries-lab A - master branch
P 2017-05-25 11:35:08 by practicalseries-lab Initial commit

Line 15, Column 1 — 17 Lines
INS Text Spaces: 4
```

Figure E.2 **master** branch

The **dev** branch looks like this:

```
test.txt (lab-02-rebase) - Brackets
File Edit Find View Navigate Debug Git Help
Working Files test.txt
lab-02-rebase dev
.gitignore test.txt

1 Initial commit
2
3 Commit A - master branch
4
5 Commit B - dev branch
6
7

P 2017-05-25 11:37:31 by practicalseries-lab B - dev branch
P 2017-05-25 11:36:42 by practicalseries-lab A - master branch
P 2017-05-25 11:35:08 by practicalseries-lab Initial commit

Line 15, Column 1 — 17 Lines
INS Text Spaces: 4
```

Figure E.3 **dev** branch

To summarise the **master** branch is missing commit B and the **dev** branch is missing commit C.

The next thing is the rebase itself. When we did a branch merge (back in section 6.7.1), we selected the **master** branch and merged the other branch into it.

A rebase is the other way around; we start the rebase on the other branch (the **dev** branch in this case), the rebase effectively moves the point at which the *rebasing* branch deviated from the parent branch (**master**) to the end of the chain, it wants to do this:

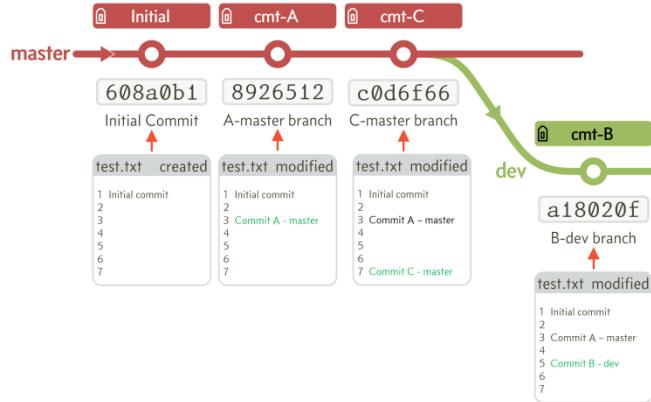


Figure E.4 Theoretical workflow after a rebase

Essentially, the rebase *unplugs* the branch that is being rebased, and moves it to the tip of the other branch; *in theory that is*.

If we merged **dev** back into **master** at this point we would have:

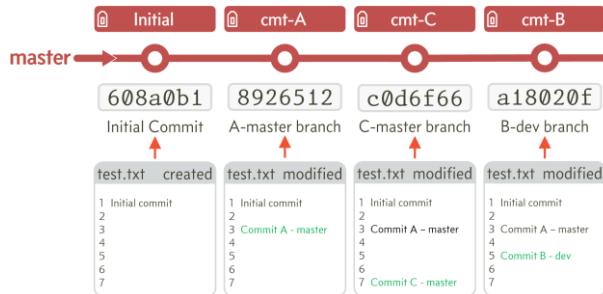


Figure E.5 Theoretical workflow after a rebase and merge

The trouble is, that isn't exactly what happens.

Here is the proper explanation:

The B commit was originally tied to the A commit (that's where the branches originally diverged). It means that moving the B commit to the end can't be just a cut and paste. A rebase sequentially takes all the commits from the other branch and replicates (*reapplies*) them to the destination branch (`dev` in our case). This process has two main problems:

- By reapplying commits Git creates new ones. Those new commits, even if they bring in the same set of changes, will be treated as completely different commits and given new commit numbers.
- The rebase, when it reapplies commits, creates new ones, but does not destroy the old ones. It means that even after a rebase, the old commits will still be in the repository, and are still available.

What actually happens is this:

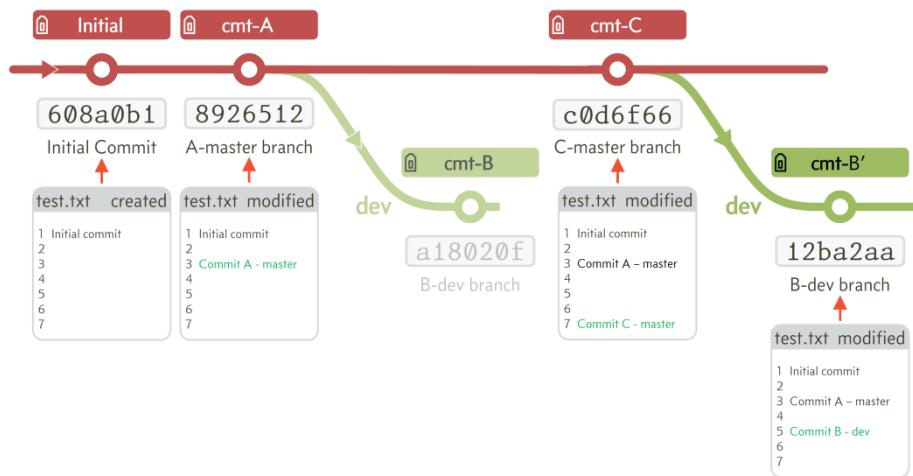


Figure E.6 The actual workflow after a rebase

The `dev` branch has a completely new commit (I've called it `cmt-B'` that's B *prime*); it even has a new commit number [`12ba2aa`].

The problem for me is that the previous commit (or commits if there was more than one on the `dev` branch) is not destroyed. It is still there, (it is not that easy to access, it

would need a `reset` to find it), but someone could use it (particularly if they reset to a tag).

Once the rebase is complete, merging the `dev` branch with `master` gives (after a rebase, the merge will always work, it will be a fast-forward merge and needn't create a merge commit):

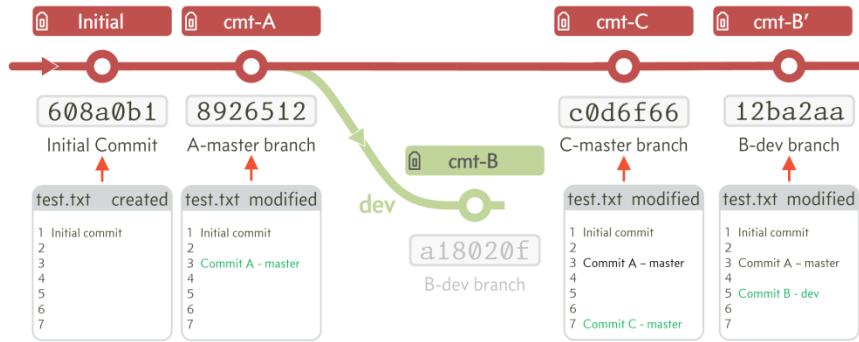


Figure E.7 The actual workflow after a rebase and merge

The final arrangement in Brackets (and the content of `test.txt`) is:

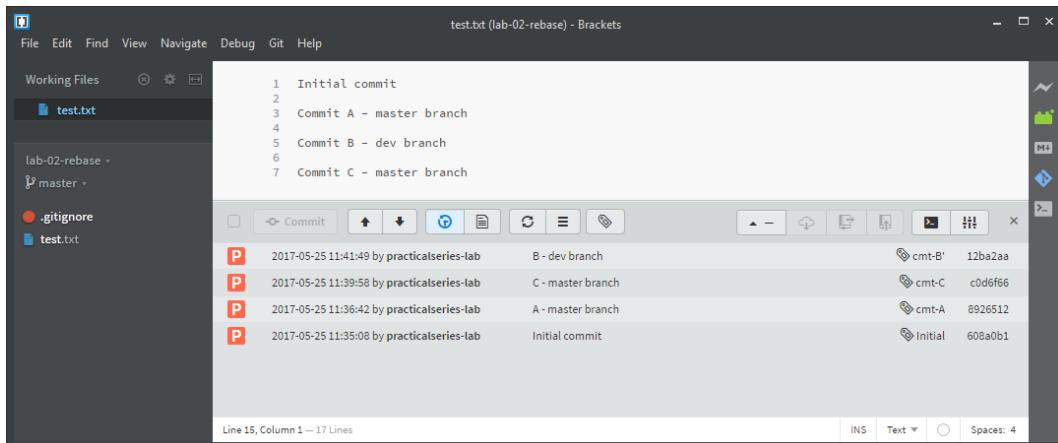


Figure E.8 The actual workflow after a rebase and merge in Brackets

I think it is confusing—think long and hard before using it.