

Tooling

with Git, npm and Gulp

Contents

1	Introduction to Version Management	6
1.1	Is understanding version management important?	6
1.2	Why is version management an important skillset?	6
1.2.1	The nature of developing solutions with code	6
1.2.2	Approaches to version managing code	8
1.3	Introducing Git	8
1.4	Additional resources	9
2	Setup	10
2.1	Git software	10
2.2	Git repository host: E.g. GitHub	10
2.2.1	Sign up for an account on GitHub.com	10
2.2.2	Who are you?	11
2.2.3	SSH access to github.com not working?	11
2.2.4	Other setup	12
3	Git basics	14
3.1	Basic concepts	14
3.1.1	Repositories and commits	14
3.1.2	File states	15
3.1.3	.gitignore	16
3.2	Basic commands	16
3.2.1	Creating a new repository	16
3.2.2	Finding out the state of a file	17
3.2.3	Tracking our first file	19
3.2.4	Committing our first file	19
3.2.5	Subsequent commits	20
3.2.6	Changing a commit	21
3.2.7	Files with multiple statuses	21
3.2.8	Seeing your commit history	22
3.3	Additional resources	23

4	Moving between commits	24
4.1	Under the hood of Git	24
4.1.1	Git has integrity	24
4.1.2	Git only adds data	24
4.1.3	The HEAD	25
4.2	Commands for moving between commits	25
4.2.1	Commit hashes	25
4.2.2	Checking out older commits <code>git checkout</code>	26
4.2.3	Return to the end of your commits	26
4.2.4	Stashing unfinished work	27
4.3	Additional resources	27
5	Branches	28
5.1	What is a branch?	28
5.1.1	The default branch	29
5.1.2	Why does the default branch have different names?	29
5.2	Commands for working with branches	30
5.2.1	Creating a branch	30
5.2.2	Moving between branches	30
5.2.3	Seeing all your branches	31
5.2.4	Deleting a branch	31
5.3	Merging branches	31
5.3.1	Resolving merge conflicts	32
5.4	Additional resources	33
6	Remote repositories	34
6.1	Why use a remote repository?	34
6.1.1	Deploying code	34
6.2	Remote repo key concepts	36
6.2.1	Remote repo workflow	36
6.2.2	<code>origin</code>	37
6.2.3	Remote repo providers	37
6.3	Connecting local and remote repos together	38
6.3.1	Creating a remote repo from an existing local repo	38
6.3.2	Pulling down an existing repo with <code>git clone</code>	39
6.4	Branches and remote repos	40
6.4.1	Pushing a branch to remote repository	40
6.4.2	Pull requests	41
6.4.3	Push rejection	41
6.4.4	<code>git pull</code> merge conflict	42
6.4.5	Deleting old branches	42

7	Git workflows	43
7.1	Workflow considerations	43
7.1.1	Centralised workflow	44
7.1.2	Featured branch workflow	44
7.1.3	Gitflow workflow	44
7.1.4	Forking workflow	45
7.2	Additional resources	46
8	Links	47
8.1	Additional Resources	47
9	Git Glossary	48
10	npm basics	51
10.1	npm basics	51
10.2	package.json	51
10.3	Setting up a new project to use npm packages	52
10.4	Reproducing the packages from an existing project	53
10.5	.gitignore and npm	53
10.5.1	The bad option	53
10.5.2	The good option	53
10.6	Additional resources	54
11	gulp basics	55
11.1	Why use gulp?	55
11.1.1	What tasks would we like to automate?	55
11.2	gulp basics	55
11.2.1	Using gulp in a project	56
11.2.2	gulpfile.js	56
11.2.3	Minifying CSS files	57
11.2.4	Compiling SASS to CSS	59
11.2.5	Combining gulp tasks using series	60
11.2.6	Running tasks automatically using watch	60
11.2.7	Refreshing your browser when you make changes	61
11.2.8	Finding out what tasks are already defined in gulpfile.js	61
11.3	Managing build merge conflicts	62

How To Use This Document

Bits of text in **red** are links and should be clicked at every opportunity. Bits of text in **monospaced green** represent code. Most the other text is just text: you should probably read it.

Some sections are marked “Read-Only”: these are sections that are intended to be read through in your own time and will not have a corresponding lecture.

Copying and pasting code from a PDF can mess up indentation. For this reason large blocks of code will usually have a **View Code**  link above them. If you want to copy and paste the code you should follow the link and copy the file from GitHub.

Taking Notes

In earlier cohorts I experimented with giving out notes in an editable format. But I found that people would often unintentionally change the notes, which meant that the notes were then wrong. I’ve switched to using PDFs as they allow for the nicest formatting and are also immune from accidental changes.

Make sure you open the PDF in a PDF viewing app. If you open it in an app that converts it into some other format (e.g. Google Docs) you may well miss out on important formatting, which will make the notes harder to follow.

I make an effort to include all the necessary information in the notes, so you shouldn’t need to take any additional notes. However, I know that this doesn’t work for everyone. There are various tools that you can use to annotate PDFs:

- Preview (Mac)
- Edge (Windows)
- **Hypothes.is**
- Google Drive (*not* Google Docs)
- Dropbox

Do not use a word processor to take programming notes! (e.g. Google Docs, Word, Pages). Word processors have the nasty habit of converting double-quotes into “smart-quotes”. These can be almost impossible to spot in a text-editor, but will completely break your code.

The trouble with Git quotes is everyone has their own version

- Some T-Shirt

Chapter 1

Introduction to Version Management

1.1 Is understanding version management important?

Building and maintaining code can be a complicated process, especially when you have a complex code base and/or many people involved.

Knowing how to version manage a code base is an essential skillset for any professional developer.

It's worth the learning curve, trust us on this one!

1.2 Why is version management an important skillset?

1.2.1 The nature of developing solutions with code

Good code products are usually created **iteratively**. This means you tend to make many small improvements that layer on top of one another. A bit like an onion.

Iterative and incremental development

The basic idea behind this method is to develop a system through repeated cycles (iterative) and in smaller portions at a time (incremental), allowing software developers to take advantage of what was learned during development of earlier parts or versions of the system.

- Wikipedia

The part about building on what you have learned is really important. Coding is always a learning process, even when you are at senior level. It is normal to **get things wrong a few times before you get it right**, and in order to do that you often need to experiment with solutions.

And once built, code very rarely stands still. It will need continuous attention to manage it. Such as:

- Adding new features to respond to customer needs
- Fixing bugs
- Improving performance and efficiency
- Maintaining security

These tasks are likely to involve changes across lots of files, and we'll usually want to keep track of:

- What files have been changed
- When files were changed
- Why files were changed (ew feature, bug fix etc)

When working in teams, it's usually helpful to also know:

- Who created what code
- Who reviewed, approved and dealt with code conflicts

- What future changes are being worked on

There's a lot to keep on top of isn't there? Often a lot of this will be happening at the same time as well.

A strong approach to version management can help make all this stuff a whole bunch easier and ultimately allow you to spend more time coding and less on admin.

1.2.2 Approaches to version managing code

It is possible to version manage your code without tools, using manual approaches. But manual approaches are error prone and time consuming - boo!

Approaches to manual version management may include some or all of these:

- Duplicating specific files and renaming them
- Copying a whole folder or project and renaming it with a timestamp, eg taking a backup
- Creating a changelog for each file

But we're developers. And developers specialise in using and creating tools to make things quicker and easier. So let's look at what tools we can make use of.

1.3 Introducing Git

Git is a software tool that automates much of the version management process for us. It can:

- keep track of how files change over time (our code)
- can be used to version manage anything (text files, video, images)
- mainly used on command line, but GUIs are available

There are many version management tools out there, why do we use Git?

Each has its own quirks, and merits in its design.

Ultimately we use Git because:

- it is the most widely used source code management tool

- 33.3% of professional software developers reporting use Git or GitHub as their primary source control system
- excellent tooling support, supporting existing workflows and editors like VS-Code, Sublime

1.4 Additional resources

- [Wikipedia - iterative and incremental development](#)
- [15 BEST Version Control Software \(Source Code Management Tools\)](#)
- [Official git reference documentation](#)

Chapter 2

Setup

New computer? Get you.

Here is a quick setup guide to get you back working with Git.

2.1 Git software

You might already have Git installed, to check run:

```
git --version
```

If you get something like `git version 2.23.0` then you have Git installed.

If not, then head over to [Git's website](#) and download the tool.

2.2 Git repository host: E.g. GitHub

There are many places you can host your Git project (known as repositories).

On the Bootcamp we use GitHub.

2.2.1 Sign up for an account on GitHub.com

If you don't already have one, [sign up](#) for an account.

2.2.2 Who are you?

To avoid you needing to enter your GitHub username and password everytime Git interacts with GitHub we can setup SSH keys.

To check if these are in place:

```
ssh git@github.com
```

You should get something like this with `[username]` replaced with your username:

```
Hi [username]! You've successfully authenticated, but GitHub does not
↳ provide shell access.
```

2.2.3 SSH access to github.com not working?

Before you create some new SSH keys, worth checking if you already have some with:

```
tail ~/.ssh/id_rsa.pub
```

If you you get a screen of nonsense like this:

```
ssh-rsa
↳ AAAAB3NzaC1yc2EAAAADAQABAAQCAiMYAAHN7k2AS2ygLhxXoFFW68Cv+eQC0EkQka
AaPL7lgZppaoQoQ9Y/k6uky+JLd5BnXCyYV0d6APWWKxKkeE+0OodKdrPN394yBkYIQyOL0EqsJ9
CFK06Z1qbWWY/70rLUql0W232eoS8Ct6IHx/kw0d+ePYiGaXKssVwBb8IkdFQ4FZFv+QG7aZHCLW
e8wjT0DnQni11XDoG0Jh1Ea/V0MYCzSW71wmACd56HJmkwM4MB9RSrKUevDy14YnK/4Qr14WdDn4
vxCueX9ulFEkmPbjDUH8XUi3vGmecDUO+38sgg+BVbbonc23zR+FR9OH6mUPR4+4u3tA/2W8RmJN
P+8nMbVj8ttP18GIvp6FKHj6U14w5q+muJKn1Y32KWxhR1phhxACUmihwb9D4cTpcDgpC0QsLKsh
Jn8BUomMZ3K4Vx9CDyfKv/DABr/wQpoMOGnD1gOUkiBNOQTY0e9s+/o4x04izF+AkGWdSo+FrTEB
dH/v+DLNAJR1gG6DtmW1vfuy95d3jpwnT8d3tkEnw2/JDhPNGX3UZQKdzLHstCjG0NlhDXf+M4rq
dDSov9FZ00Kcm0jP04pl4Y93g1WFd3kpyLvqLh4GWvvy3D3MSovWNifzavSiw9g+rMSbwAh8uUTu
pHce7Pbodn7joNetQebKWzLXEZNxaaq+a75PVQ== your_email@email.com
```

If not (you get `No such file or directory`), then you'll need to generate a new keypair using [GitHub's suggested process](#):¹

```
ssh-keygen -t rsa -b 4096 -C "your_email@example.com"
```

Replace `your_email@example.com` with your own email or comment (this is what the `-C` flag means; comment).²

Once you've finished creating your keys, output the public part³ key with:

```
cat ~/.ssh/id_rsa.pub
```

Then add key to your GitHub account in [your account settings](#).

Finally, test your key with:

```
ssh git@github.com
```

2.2.4 Other setup

USING VS CODE AS YOUR DEFAULT EDITOR

Mac

In VS Code: `shift + cmd + P`

Type or paste `> Shell Command: Install 'Code' command in path` hit `[Enter]`

Windows

In VS Code: `shift + ctrl + P`

Type or paste `'> Shell Command: Install 'Code' command in path' hit '[Enter]'`

¹At the `Enter passphrase (empty for no passphrase)` step I'd suggest just hitting `[Enter]` providing no password.

²You often provide your email or name here as your SSH key may end up in a few places other people may find it. To prevent them removing it erroneously it helps them identify who's key it is and if it is still needed.

³the part ending `.pub`, the other part is the private part and should be treated like a password; never shared.

Make sure you selected Add to PATH during the installation.

All systems

From the command line, run

```
git config --global core.editor "code --wait"
```

SETTING COMMIT AUTHOR

So that Git knows who your commits are made by:

```
git config --global --edit
```

Then edit values as required and uncomment (by removing #).

Then, to save and close the file.

Chapter 3

Git basics

3.1 Basic concepts

Git won't help you manage your code unless you initialise it and tell it what's going on. It cannot organise your work for you like a magic fairy - you will always have to think about how to use its power to organise your code in a sensible way.

Getting good at this comes with experience and fluffing it up a few times.

3.1.1 Repositories and commits

When you begin work on a coding project you typically create a directory to house all your project files. Or you might use an existing one if someone else has already started work.

A repository

is a directory that Git has been told to monitor. It will watch all the files and sub-folders within that directory. It's usually referred to as a **repo**.

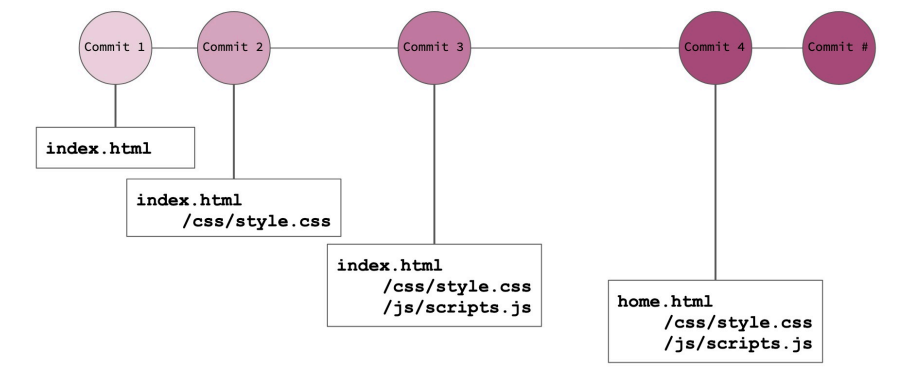
The basic building blocks of a git repo are called **commits**.

A commit

A snapshot or version of your code which you intentionally take at a moment in time.

Commits are named by the developer and a hash is automatically generated which looks like this `166348d5901829380ab5b44ffa09fea3a595f64d`.

By looking at the history of commits in a repository, another developer can look at the code and figure out the story of how it's gotten to where it is today.



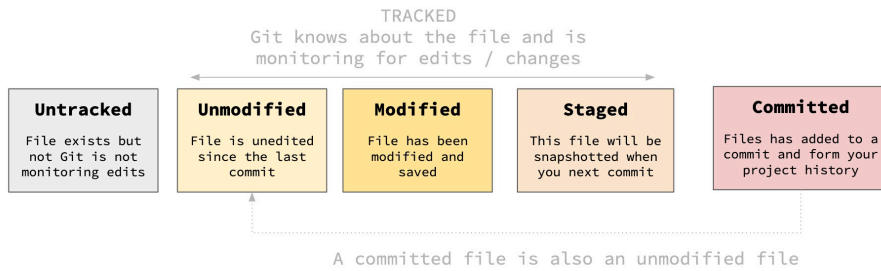
A commit history can tell the story of a project's code

3.1.2 File states

In Git's world, files can have different states. It is important to know what state a file is in as it will affect what Git does with it.

The main states are:

- **Untracked:** the file exists in the project directory but Git is unaware it exists, edits are not being watched/tracked. Files are usually only in this state when they are first created.
- **Unmodified:** Git knows about the file and is tracking the file for changes, but the file has not been edited since it was created or since the last commit.
- **Modified:** Git knows about the file and the file has been changed since it was created or since the last commit
- **Staged:** Git has been told to add the changes in this file to your next commit
- **Committed:** the file has been added to your last commit eg edits to it have been versioned/snapshotted. The file is now in the Unmodified state again.



The main states a file can be in according to Git

3.1.3 .gitignore

The `.gitignore` file allows you to specify any files or directories you do not want tracked inside your project repo.

On complex projects this is invaluable.

It uses globbing patterns to match against file names. Examples are as follows:

```
// By default, patterns match files in any directory
Debug.log

// The * is a wildcard that matches zero or more characters
*.log

// You can prepend a pattern with a double asterisk to match
↳ directories anywhere in the repository
**.log

//Prepending a slash matches files only in the repository root
/debug.log
```

[Globbing patterns Atlassian gitignore article](#)

3.2 Basic commands

3.2.1 Creating a new repository

Navigate to the directory you wish to get Git watching and type:

```
git init
```

This command creates a hidden local repository folder `.git`. This is where Git will store all its information about your project; which files there are, how they have changed over time, etc.

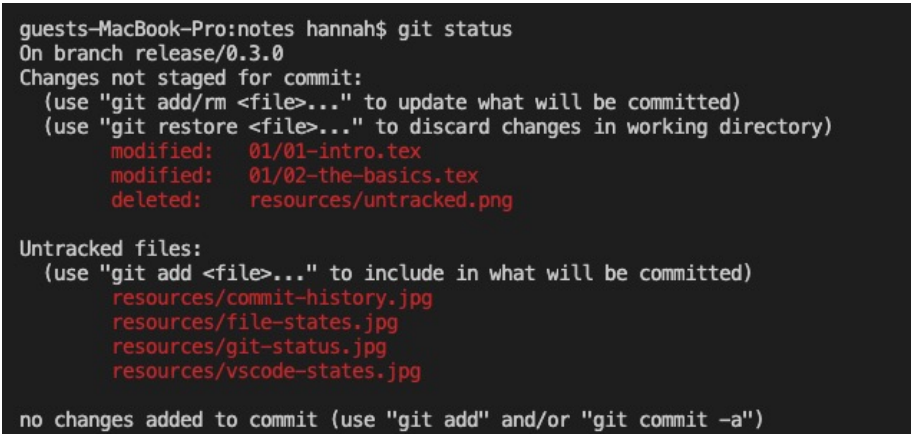
Avoid having a Git repository within a Git repository. Eg you initialise Git in a parent directory and then also initialise Git in a subdirectory of the parent. Git will get in a total muddle and you'll get all sorts of funny side affects.

Anyway, after running `git init` successfully Git is ready to go, but is not yet tracking (version managing) anything.

3.2.2 Finding out the state of a file

The main command you will use to find the states of files in your repo, and what is going on in your repo generally:

```
git status
```

A terminal window with a dark background showing the output of the 'git status' command. The output is as follows:

```
guests-MacBook-Pro:notes hannah$ git status
On branch release/0.3.0
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   01/01-intro.tex
    modified:   01/02-the-basics.tex
    deleted:    resources/untracked.png

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    resources/commit-history.jpg
    resources/file-states.jpg
    resources/git-status.jpg
    resources/vscode-states.jpg

no changes added to commit (use "git add" and/or "git commit -a")
```

An example output from running `git status`

You can also pass `git status` a flag, that gives a slightly different output.

```
git status -s
```

```

guests-MacBook-Pro:notes hannah$ git status -s
 M 01/01-intro.tex
 M 01/02-the-basics.tex
 D resources/untracked.png
 ?? resources/commit-history.jpg
 ?? resources/file-states.jpg
 ?? resources/git-status.jpg
 ?? resources/vscode-states.jpg

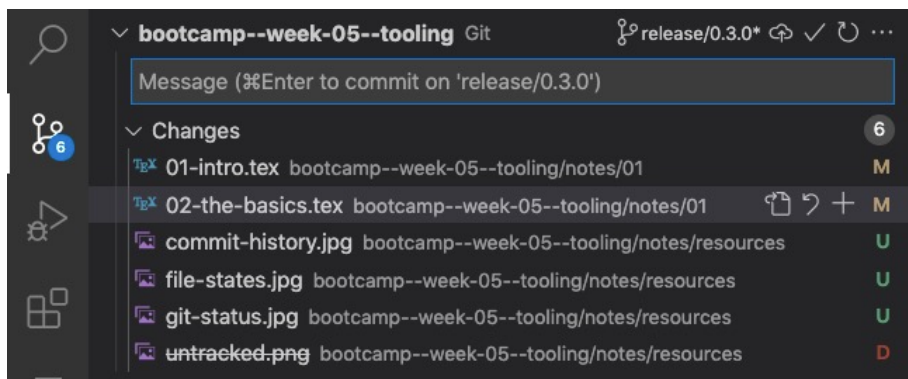
```

An example output from running `git status -s`

The status codes are as follows:

- ?? = untracked
- ' ' = unmodified (eg it's not showing up in the list)
- M = modified
- A = added
- D = deleted
- R = renamed
- C = copied
- U = updated but unmerged

VSCode also has some handy tools to help you track the status of your files too, and you will notice the same status flags on the right of each file name.



The inbuilt Git tools in VSCode are really useful

3.2.3 Tracking our first file

New files added to your project directory start off in an untracked state. They need to be tracked so Git watches them for changes.

A file only needs to be tracked once and then Git will remember its existence forever - handy!

Tracking a file and staging a file use the same command - more on staging files in a bit.

```
git add {filename}
```

A few git add tricks

```
// add multiple files in one command
git add css/file1.css js/file2.js file3.html

// add all files in a folder
git add img/*

// add all of type
git add *.scss

// use carefully.
// a shorthand for staging all modified files, including hidden
→ files like .gitignore
git add .

// use carefully.
// as git add . but does not stage hidden files
git add *
```

3.2.4 Committing our first file

Now we've used `git add` to track a file and we've edited the file, we need to commit it.

```
git commit -m "my first commit"
```

Commits need to have a name and it is very important that you take the time to write good commit messages. Another person will need to read the message and fathom what changes to the code’s behaviour are expected.

	COMMENT	DATE
○	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
○	ENABLED CONFIG FILE PARSING	9 HOURS AGO
○	MISC BUGFIXES	5 HOURS AGO
○	CODE ADDITIONS/EDITS	4 HOURS AGO
○	MORE CODE	4 HOURS AGO
○	HERE HAVE CODE	4 HOURS AGO
○	AAAAAAAAA	3 HOURS AGO
○	ADKFJSLKDFJSDKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAAAAAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

Don't be like this person

3.2.5 Subsequent commits

Most developers will regularly commit through their development time on a project. For files that are already tracked (remember you only need to track them once when you first create them), you will need to ensure they are staged so Git knows to add them to your commit.

Confusingly, staging a file uses the same command as tracking a file for the first time.

```
git add {filename}
```

So you will find yourself using `git add` a lot! There are some shortcuts, so that you can stage and commit modified (and therefore tracked) files in one go:

git commit arguments

m = Message associated with the changes in the commit

a = All changed tracked files included in the commit

Example usage, allowing the **git add** step to be skipped.

```
git commit -am "my second commit"
```

However, caution should be taken when you use this command. There will be times when you may not want to add all your modified files to a commit. You may have changed other files for testing or for other reasons.

You must always be thoughtful of what files you are committing. If in doubt, take your time and run **git status** a whole ton of times to make sure you know what Git is doing (or about to do!).

3.2.6 Changing a commit

If you wish to amend the last commit you made, e.g. to change the commit message, then use:

```
git commit --amend
```

3.2.7 Files with multiple statuses

It is possible to edit a file, stage that change, then make a further change.

The file will then be in both a staged state (first edit) and a modified state (further edits since first edits were staged).

git status -s can help us decypher what has gone on here.

It's output:

```
M README.md
MM index.html
```

```
A  css/styles.css
M  js/scripts.js
?? LICENSE.txt
```

- ? Left column: Staged changes column
- ? Right column: Unstaged changes column

README.md

File has been edited but those modifications haven't been staged with `git add README.md` or committed with `git commit ...` yet.

index.html

File has been edited, those modifications staged, and then has had further modifications made.

css/styles.css

File has been added into project (it's new) and a `git add css/styles.css` has been run to add the file into staging. The next `git commit ...` would add this file to the history.

js/scripts.js

File has been added edited and those modifications staged with `git add js/scripts.js`.

LICENSE.txt

Untracked file. Possibly new, but has never had `git add LICENSE.txt` run to start Git tracking it.

3.2.8 Seeing your commit history

To see a list of your commits use:

```
git log
```

For a shorter list of recent commits use:

```
git reflog
```

3.3 Additional resources

- [GDS Git style guide](#)
- [How to write a good Git commit message](#)
- [5 tips for a better commit message](#)
- [Globbing patterns](#)
- [Atlassian gitignore article](#)

Chapter 4

Moving between commits

There are times when we may need to view or even work from the files in our older commits. Perhaps because we need to compare the code to fix something broken, or refer to older code to see how something worked before or to add it back in.

Understanding how Git saves your changes can help you learn how to move between different versions more quickly and why something might be happening if you have unexpected results.

4.1 Under the hood of Git

4.1.1 Git has integrity

Git has a clever way of knowing if files have changed in any way, or if the files match an earlier snapshot.

- Everything is check-summed before storage
- Snapshots are referred to by that checksum
- Impossible to change the contents of any file or directory without Git knowing about it because it will change the checksum
- Checksumming is SHA-1 hash, producing 40-character string e.g:
24b9da6552252987aa493b52f8696cd6d3b00373

4.1.2 Git only adds data

...mostly.

It is hard to get the system to do anything that is not undoable or to make it erase data in any way.

Actions in Git (nearly) only add data to the Git database.

It is therefore a great safety net for trying things out, and rolling back if needed.

After you commit a snapshot it is very difficult to lose work, especially if you regularly push to a remote repository.

The only way you can lose or mess up changes is if you haven't pushed your work yet (see working with the remote), or use some more niche commands and arguments like `--force` (more on this later).

4.1.3 The HEAD

The HEAD (yes it's in capitals on purpose!)

The active commit you are currently working from.

Usually it is the last commit you made, but it can be an older commit if you moved to that.

When we move between commits we change the commit the HEAD refers to.

4.2 Commands for moving between commits

4.2.1 Commit hashes

To move to a different commit, we need to find its hash. The hash is automatically generated by Git and looks like this `166348d5901829380ab5b44ffa09fea3a595f64d`.

This is the checksum value that we referred to earlier.

We can use the `git log` or `git reflog` command to get a list of the hashes. `git reflog` gives a shortened version of the hashes but they still work the same as the long ones.

4.2.2 Checking out older commits `git checkout`

We can move to an older commit using:

```
git checkout {commit hash}
```

eg.

```
git checkout 166348d5901829380ab5b44ffa09fea3a595f64d
```

or

```
git checkout 166348d
```

Note that if you have unsaved changes or files you have not committed you will be prompted to save and commit before you move.

4.2.3 Return to the end of your commits

To return to the very last commit you made (the tip of your series of commits), you can use:

```
git checkout master
```

or

```
git switch -
```

Don't use these commands blindly, especially as you get more advanced with Git and start using different branches. More to follow.

Another option is to use `git reset`. This does a very similar thing and will return you to the tip (last commit) of whatever branch you are working on.

```
git reset --hard HEAD
```

Using the `--hard` flag resets the index and working tree. Any changes to tracked files in the working tree since your last commit are discarded.

```
git reset --soft HEAD
```

Using the `--soft` flag leaves all your changed files eg "changes to be committed" but moves your tree so you are working from your last commit.

4.2.4 Stashing unfinished work

`git stash` temporarily shelves (or stashes) changes you've made to your working copy so you can work on something else, and then come back and re-apply them later on.

Stashing is handy if you need to quickly switch context and work on something else, but you're mid-way through a code change and aren't quite ready to commit.

The downside with stashing is remembering that you stashed something. I nearly always forget what I stashed where so use this with caution. Like all things Git, you still have to know what's going on, a tool like Git doesn't solve all your workflow problems!

[Atlassian article on stashing](#)

4.3 Additional resources

- [Wikipedia: checksum](#)
- [Git for Computer Scientists \(Quick introduction to git internals for people who are not scared by words like Directed Acyclic Graph\)](#)
- [How to reset, revert and return to previous states in Git](#)
- [Atlassian tutorial on git revert](#)
- [Atlassian article on stashing](#)
- [Useful tricks you might not know about Git stash](#)

Chapter 5

Branches

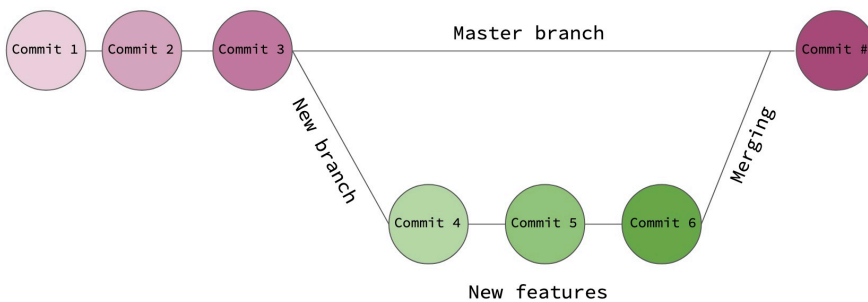
5.1 What is a branch?

Branches are an essential feature of Git to get your head around and professional developers will make heavy use of them.

A branch

A single development stream, or split in the project, allowing you to try things out or develop features without affecting other work.

Branching means you diverge from the main line of development and continue to code in a separate stream of commits. Some people might refer to a branch as a parallel code world.



The main concepts of branches

The power of branches are incredibly useful for:

- Feature development
- Bug fixes
- Trial and experiment

5.1.1 The default branch

When you run `git init` Git automatically creates your first branch, known as the default branch.

The default branch is typically considered the version of your code that is production ready. In fact an identical copy of your default branch is usually what's running on a live site or in a live app. More on Git workflows later.

The default branch will have a name. When you use Git in the terminal the default branch is called `master`.

But when you create a new repo in GitHub, the default branch is called `main`.

The default branch name can be changed by running `git branch -M new-branch-name` from the default branch. Note that you need to have made at least one commit in the default branch to be able to rename it.

5.1.2 Why does the default branch have different names?

In summer 2020, the Black Lives Matters movement prompted the tech community as a whole to review its practices to change the systems that cause inequalities across society.

One of the outcomes of this consideration, was feedback that the default branch should not be called `master` as many considered the word an oppressive term for black communities.

As a result, some tech communities have begun transitioning away from `master` as the default name.

You can read more about GitHub's decision here: [Github to remove master-slave terminology from its platform](#).

5.2 Commands for working with branches

5.2.1 Creating a branch

We can create a new branch and call it whatever we like using `git branch`.

Take care to think about which branch you are deviating from. If you are on the master branch when you run this command your new branch will be an exact copy of the master branch, complete with the full commit history.

It is usually best practice to ensure you are working from the master branch before you create a new branch, because master is usually considered the canonical/primary version of your code and should be bug free.

```
// Move to the master branch first
git checkout master

// Now create your new branch
git branch {branch-name}

eg.
git branch new-branch
```

When you create a branch you don't automatically start working on it, you'll need to use the moving between branches command.

5.2.2 Moving between branches

You can move between branches using `git checkout`.

```
git checkout {branch-name}
```

Creating and moving to a new branch all at once

```
// add multiple files in one command
```

```
git checkout -b {branch-name}
```

eg.

```
git checkout -b new-branch
```

5.2.3 Seeing all your branches

It's very easy to forget the name of the branches you have, or if you start work on existing project it's useful to be able to see what's already being work on.

```
git branch -a
```

5.2.4 Deleting a branch

Once you have finished working on a branch (probably because you have merged it into master) it is good practice to get rid of it, so your git repo stays tidy and managable.

```
git branch -D {branch-name}
```

You can't run this command if you have already checked out the branch, eg you are already in it. It a good idea to run this command from the master branch.

5.3 Merging branches

Merging

Combining two branches together eg. your experimental branch back into master.

Merging creates a unified history and applies the changes of the other branch into your current working branch.

Merging is where Git starts to become really really useful. But it can also become a little bit of a headache especially if you have changes to the same parts of a file

in both branches.

It is usually when you come to run `git merge` that you can tell if you have broken your work down into sensible features and if your approach to using Git is sensible or not.

```
// Start from the branch you wish to bring the changes into
git checkout {receiving-branch}
```

```
git merge {branch-name}
```

5.3.1 Resolving merge conflicts

If the two branches you're trying to merge both changed the same part of the same file, Git won't know what your code means and therefore what should replace what.

When such a situation occurs, it stops right before the merge commit so that you can resolve the conflicts manually. This is called a merge conflict.

A merge conflict

An event that occurs when Git is unable to automatically resolve differences in code between two commits.

When you encounter a merge conflict, running `git status` shows you which files need to be resolved.

Git will helpfully **edit the content** of the affected files with visual indicators that mark both sides of the conflicted content. These visual markers are:

- <<<<<<
- =====
- >>>>>>

Generally the content before the ===== marker is the receiving branch and the part after is the merging branch.

Its helpful to search a project for these indicators during a merge to find where conflicts need to be resolved.

5.4 Additional resources

- [Atlassian Git merge](#)

Chapter 6

Remote repositories

6.1 Why use a remote repository?

Most operations are local, they are things that only affect your computer. Such as:

- rolling back to early snapshot (checking out an earlier version of your files)
- creating a branch
- creating a new commit
- merging branches

Working locally reduces requests over network, increases speed/efficiency and allows you to work offline. This can be very helpful a lot of the time.

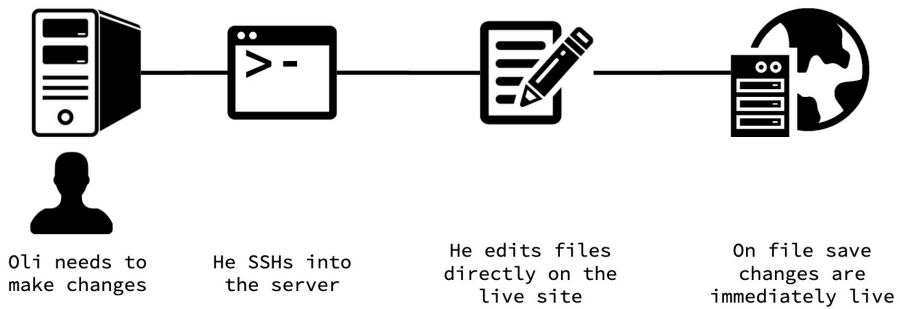
But we also need to be able to:

- collaborate with others
- make automated deployments of code to various environments
- backup our code (losing code is sooo sad)

6.1.1 Deploying code

There are three common approaches to deploying code to a production environment.

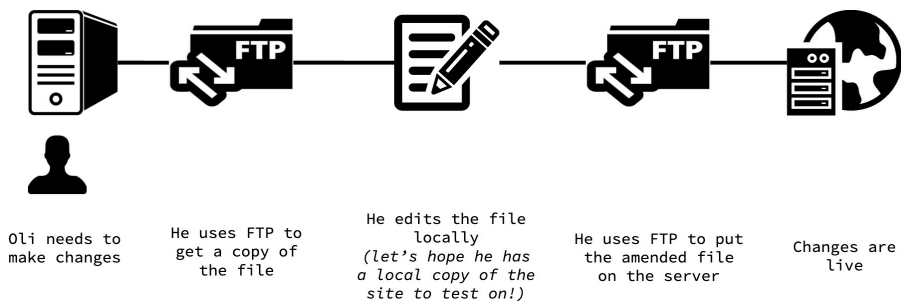
- Working directly on files on a server (VERY bad!)
- Deploying using FTP (pretty bad)
- Deploying with version management (good, we like)



Working directly on files on a server

Drawbacks:

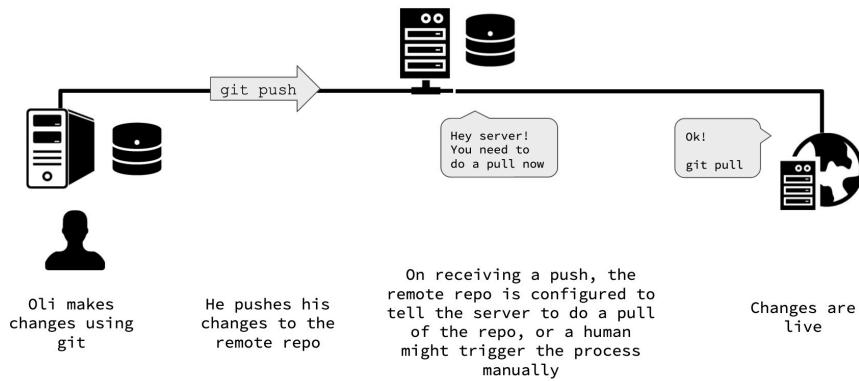
- No ability to automatically rollback changes
- Requires you to keep backups and different versions of files manually
- Working with multiple contributors is tricky
- High risk that you break stuff in production



Deploy using FTP

Drawbacks are all of the above plus:

- Slow, uploading and downloading each time
- Error prone



Deploying with version management

Advantages:

- Each version of a project is a commit and can be reverted to easily
- Automatically keeps track of which files have changed and need to be uploaded
- Changes have an audit log, who made what change when
- Automated deployments also have an audit log

6.2 Remote repo key concepts

6.2.1 Remote repo workflow

When we are working with remote repos, here's the series of steps that are usually happening over and over again.

- You **pull** other people's work from the remote repo
- You make changes to it (in a new or existing branch)
- You commit work
- You **push** your work
- You resolve conflicts if necessary

git pull

Retrieve / download new commits from the remote repo and attempt to merge them into your branch. Can lead to merge conflicts.

git push

Send / upload your commits to the remote repo so others can use your work

6.2.2 origin

Origin is the name of the remote repository where you want to publish your commits, as defined by you locally. This means someone else might connect to the same repo on their local machine but reference it with a different name to you.

By convention the default remote repository is named `origin`, but you can work with several remotes (with different names) at the same time. It's not usually necessary to do this though as a lot the time it is just one remote repo for a project.

This is helpful because the other way of telling your machine where the remote repo is by using the URL. This can be quite long like:

```
https://github.com/develop-me/bootcamp--week-05--tooling.git
```

or

```
git@github.com:develop-me/bootcamp--week-05--tooling.git
```

Typing a long name like that every time you want to push and pull from a repo will be a right pain. So the name, like `origin` gives us a shortcut way of referring to it.

6.2.3 Remote repo providers

There are lots of companies that provide remote repos. A few popular ones are:

- GitHub
- GitLab

- BitBucket
- Beanstalk

The services these providers offer is built on the workings of Git. However there may be additional services they offer such as project management tools, automated deployments and user management.

A good example of a difference between Git and GitHub is the naming of the default branch. On the command line, Git names the default branch `master`. But the default branch is named `main` in GitHub.

Therefore `Git != GitHub`

6.3 Connecting local and remote repos together

6.3.1 Creating a remote repo from an existing local repo

The steps (with no code) are as follows:

- Set up project directory locally
- Start version managing your directory and making commits
- Create a new remote repo (in GitHub for example)
- Check the naming of the default branch in local and remote
- Change your local default branch name if necessary
- Add the remote repository (use `git@github.com` link)
- Push changes

First commit is in your local repo

1. Create a directory locally and initialise Git within it by running `git init`
2. Make your first commit locally `git add filename` then `git commit -m "First commit in repo"` - this is the first commit
3. Create your project repo on GitHub - DO NOT choose to "Initialize this repository with a README", keep the repo blank to start with

4. Check the names of the default branches match in your local repo and remote repo. You can use `git branch -a` to list your local branches
5. Rename your local branch if necessary using `git branch -M main`. Change `main` to match the name of your remote's default branch.
6. Use `git remote add origin repository-URL` to connect your local and remote repos together.
7. Push your first commit to the remote using `git push -u origin main`. Change `main` to match the name of your remote's default branch.
8. Push subsequent commits using just `git push`

6.3.2 Pulling down an existing repo with `git clone`

A great shortcut for setting up a new project directory and connecting with a remote repo that already exists.

```
git clone {repository URL}
```

It is equivalent to running all these commands:

```
mkdir {directory to create}  
cd {directory to create}  
git init  
git remote add origin {repository URL}  
git fetch  
git checkout master
```

Note that when you run this command, it creates a new directory and gives it the same name as your repo. All the files from your repo go inside this new directory. So make sure you run the command from the right place or you might end up with extra level of directory that gets in your way.

If you want to customise the name of the directory you can do so like this:

```
git clone {repository URL} {custom-directory-name}
```

`git fetch` vs. `git pull`

What's the difference?

`git pull` is in effect a `git fetch` then a `git checkout`. It goes to the remote and fetches any new commits. It will check you out to the last commit. It will also attempt to do a merge if your commit histories are different - this can result in a `merge conflict`.

6.4 Branches and remote repos

When working with branches locally you'll often want to get those branches onto the remote.

This allows others to pull your work down and code review it, run and test it or build upon your progress. Another key advantage is make sure any work you have done to date is backed up and not just on your local machine.

6.4.1 Pushing a branch to remote repository

First create the branch locally and then link it to a branch on your remote repo.

The commands to follow are:

```
# optionally going back to master branch at the beginning
git checkout master
```

```
# create your feature branch
git branch my-feature-branch
```

```
# switch to working on that branch
git checkout my-feature-branch
```

```
# [do work] = make your code changes and commit them
git commit -am "commit message"
```

```
# create branch on the remote and push commits up to it
git push origin my-feature-branch
```

```
git branch -set-upstream-to=origin/master master
```

6.4.2 Pull requests

A **pull request** is a request to merge a branch with another branch.

These are usually created through using the tools your online repo provider offers in your browser. They can be resolved on your local machine or often using tools the Git repo provider offers in the browser. The principles of dealing with merge conflicts in remote repos is the same as local repos.

They are an important feature of team working and will be an integral part of a process to discuss and review submitted code. Every team will handle this slightly differently depending on the team's culture.

A pull request can create a **merge conflict**.

6.4.3 Push rejection

Someone has pushed some code to the same branch that you're trying to contribute to.

That means the branch's HEAD has moved forwards in the remote.

You need to get the other person's code in your branch using 'git pull'. This will sync your commit histories and then allow you to contribute your changes to the tip of the branch.

The error on the command line usually looks like this:

```
To git@github.com:develop-me/bdf.git
! [rejected]          master -> master (non-fast-forward)
error: failed to push some refs to 'git@github.com:develop-me/bdf.git'

hint: Updates were rejected because the tip of your current branch is
↪ behind its remote counterpart. Merge the remote changes (e.g. 'git
↪ pull') before pushing again.

hint: See the 'Note about fast-forwards' in 'git push --help' for
↪ details.
```

6.4.4 `git pull` merge conflict

You're trying to get some new code in your branch, but there's a conflict that needs resolving. This often follows on from when you get a push rejection.

You won't be able to `git push` until its sorted.

The error on the command line usually looks like this:

```
From github.com:develop-me/simple
```

```
* branch                master      -> FETCH_HEAD
```

```
Auto-merging README.md
```

```
CONFLICT (add/add): Merge conflict in README.md
```

```
Automatic merge failed; fix conflicts and then commit the result.
```

You need to look at the conflicts manually and figure out what to keep and how the final file should look - just like a regular merge.

6.4.5 Deleting old branches

It is good practice to delete a branch from the remote after it has been merged into another branch and work on it has completely finished.

```
git checkout master and pull
git branch -d my-feature-branch
git branch -a
git remote prune origin
```

Chapter 7

Git workflows

7.1 Workflow considerations

There are lots of things to consider when designing a workflow for a project.

Who?

- Who's going to work on what? Task delegation/ownership
- Can the project work be chopped up into chunks/tasks?
- Is there a clear delegation based on team roles? Front-end, back-end, tech lead

Timescales

- What is the project timeline?
- What are the milestones?
- Are certain tasks dependant on others?
- When are things going live?

How?

- Is that work going to be reviewed? How?
- How is it going to be tested?
- How is it going merged and deployed?
- Are we releasing in phases?
- Do we quickly release new code to address bugs in the live system?

These considerations help to discern which workflow is best for your project.

7.1.1 Centralised workflow

Everyone works on master and commits all changes directly to master. This is a tricky way to work in teams.

We tried an exercise doing it, I reckon most of you absolutely hated it.

It can work if you are the only person working on a project.

7.1.2 Featured branch workflow

All feature development takes place in a dedicated branch instead of the master branch, with branches merged back into master when ready.

Adding feature branches to your development process is an easy way to encourage collaboration and streamline communication. It gives other developers or lead developer the opportunity to sign off on a feature before it gets integrated into the official project.

It's easy for multiple developers to work on a particular feature without disturbing the main codebase.

`master` will never contain broken code, good for continuous integration environments.

7.1.3 Gitflow workflow

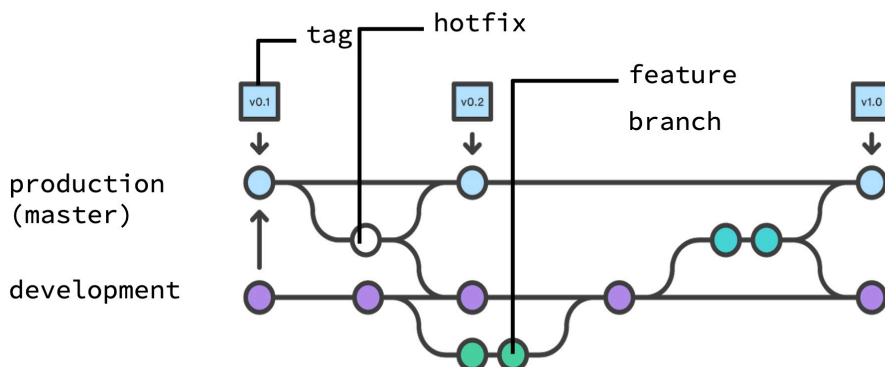
This, or a variation of it, is quite common in larger projects. Here's a great article about it: [A successful Git branching model](#).

All development should take place on the `develop` branch instead of `master`.

New **features** should reside in their own branch and instead of branching off `master`, features branch from `develop` as their parent branch.

When a feature is complete, it gets merged back into `develop` and should never interact directly with `master`.

Instead features end up going live because `develop` merges into `master` using a **release**.



An illustration of the key Gitflow principles

Because merging into `master` is a very deliberate action, this strategy has the advantage of always having a `master` branch that represents a fully-working version of the code that's ready to be used.

For website development it is also common for the `master` branch to represent the **production** version of the site - the code that's up on the live website - and for the `develop` branch to represent the **staging** version of the website - the in-progress test version.

Gitflow is not well suited to all projects. Some companies use a “deploy early and often” model, where new features are quickly added to the live site, but often only turned on for a subset of users (using “feature flags”) - trying to use Gitflow with this deployment model becomes messy very quickly.

7.1.4 Forking workflow

Instead of using a single server-side repository to act as the “central” codebase, it gives every developer a server-side repository.

Contributions can be integrated without the need for everybody to push to a single central repository.

Developers push to their own server-side repositories with only the project maintainer can push to the official repository.

Allows the maintainer to accept commits from any developer without giving them write access to the official codebase.

Distributed workflow that provides a flexible way for large, organic teams (including untrusted third-parties) to collaborate securely.

This also makes it an ideal workflow for open source projects.

Existing Projects

If you're coming to a project that already exists, it's important to **make sure you use the workflow that's already been established**. You can often ascertain this just by looking at the branch names being used. It's also common for this information to be provided in a `CONTRIBUTING.md` file in the repo.

If you're not sure which workflow is being used then be sure to ask whoever created the repository.

7.2 Additional resources

- [Atlassian comparing workflows](#)
- [A successful Git branching model](#)

Chapter 8

Links

8.1 Additional Resources

- [The Git Book](#)
- [Smashing Magazine: Make Life Easier When Using Git](#)
- [Ohshitgit](#)
- [Markdown cheatsheet](#)

Chapter 9

Git Glossary

- **auto-merge:** A remote change to a file that can be automatically combined with your version, merging those changes in.
- **branch:** A single development stream, or split in the project, allowing you to try things out or develop features without affecting other work.
- **checkout:** Switch your project directory to a certain version of the project, replaces version managed files with the versions from this point in time.
- **conflict** A situation where there are overlapping changes in a file, this situation Git will warn you and you'll need to fix the conflict yourself.
- **commit:** Create a point in time snapshot/version of the current state of the project files.
- **fetch:** Fetching file versions and information from central repository server, e.g. GitHub
- **the HEAD:** The active commit you are currently working from.
- **merge:** Merging two branches together, for example, your experimental branch back into master once to create a unified history. Applies the changes of the other branch into your current working branch.
- **merge conflict:** An event that occurs when Git is unable to automatically resolve differences in code between two commits.
- **origin:** The name of the remote repository where you want to publish your commits, as defined by you locally.
- **repository / repo:** Project files and a versioning database, in our example this is hosted on GitHub, but can be hosted on any Git server or your local machine.

- **pull** Pulling down from the central project repository and updating the branch you are working on.
- **pull request** A request to merge one branch with another branch usually originating from a remote repo.
- **push** Push your commits to the central project repository to allow other people to pull and checkout your changes.
- **push rejection** Someone has pushed some code to the same branch that you're trying to contribute to. You won't be able to push your commit until you have done a successful 'git pull'

Introducing the hide and seek champion since 1958 ;

- Some T-Shirt

Chapter 10

npm basics

10.1 npm basics

npm is a package manager for JavaScript.

A package

Packages are discrete chunks of code that can be used by someone else to fulfill a purpose.

Packages are incredibly useful for:

- Making use of the code written by others - this is smart, never shy away from using well-written and maintained code to save you time
- Openly sharing your work with others to use
- Manage packages privately, eg for your team to use

You can find packages through the npm database at npmjs.com.

10.2 package.json

`package.json` is a file you always need in a project directory, usually in the project root, which holds meta-data relevant to your project. Think of it as a kind of manifest.

More specifically it details your project dependencies - the packages your project needs (eg is dependent on) to run and the minimum version numbers required.

This makes any project that uses npm reproducible by someone else. You can just share the `package.json` file and have all the details about what is needed.

10.3 Setting up a new project to use npm packages

Running `npm init` on the command line guides you through the process of creating a basic `package.json` file.

It will ask you key questions about the metadata of your projects through a command line questionnaire.

You can also run `npm init -yes` which skips the questionnaire and is a lot faster.

After that you can install any npm packages into your project through using `npm install package-name`.

This will download the specified package and insert it into a new directory called `node_modules`. If `node_modules` does not already exist in the root of your project, npm will create it for you.

`npm install` also updates your `package.json` file details of the package you just added. This is very useful for helping you maintain a reproducible setup.

You will also notice that a new file called `package-lock.json` appears. This auto-generates for any operation where npm changes either `package.json` or `node_modules`. It describes the exact tree generated, with detailed version numbers and can be helpful in the ensuring subsequent installs are identical.

An npm install shortcut

You can install multiple npm packages at once by using

```
npm install {package1} {package2} {package3}
```

10.4 Reproducing the packages from an existing project

By sharing `package.json` with another developer, they can use `npm install` (or even faster to type `npm i`) to download all the dependencies listed.

This avoids you having to look through `package.json` and install each dependency listed one at a time - which is of course incredibly error prone.

10.5 .gitignore and npm

You might have noticed that when you download a package, it usually consists of lots of files, like hundreds of files, maybe even thousands of files. When you first install a package and run `git status`, Git will report that all these files are unstaged.

All these files will get in your way of seeing the actual files you are changing.

You have two choices: a good option and a bad option.

10.5.1 The bad option

The bad option is track all these files and add them to your git repo. Why is this bad?

It's bad because Git is really meant to help you version control the files you are actively developing, and if you are just using someone's package as is (highly likely) you won't be editing the files within the package.

So including them in your Git repo adds no value because we can already get these files in any project by getting hold of `package.json` and running `npm i`. See previous chapter.

They just kinda get in the way of your Git repo.

10.5.2 The good option

Tell me the good option I hear you cry!

The good option is to use `.gitignore`. You can refer back to the Git Basics section for more info.

So edit your `.gitignore` file to include

```
node_modules
```

and watch Git forget all about them and make your working repo a much easier place to work within.

Remember, is good practice to include your `.gitignore` file in your repo.

10.6 Additional resources

- [package-lock.json documentation](#)

Chapter 11

gulp basics

11.1 Why use gulp?

gulp is a toolkit for automating painful or time-consuming tasks in your development and build workflow.

It's not the only toolkit available. There's also [Webpack](#), [Grunt](#), [Yarn](#) and a whole heap more.

We teach gulp because it is fairly quick to learn (some might say, we'll take your judgement on that though!), commonly used and pretty quick to run tasks with.

11.1.1 What tasks would we like to automate?

- Compiling SASS to CSS
- Minifying CSS and JS files
- Refreshing your browser when you save a file in your project directory
- Convert spaces to tabs or vice versa
- Transpiling ES6 to ES5 code

11.2 gulp basics

Gulp uses JavaScript and is based on node.js.

It **reads files as streams** and **pipes the streams** to different tasks. Those tasks will then modify the files, or build news ones depending on the tasks defined.

11.2.1 Using gulp in a project

gulp is just another package you can include in a project using npm.

```
npm gulp install
```

As before installing an npm package will update your `package.json` file.

However just installing gulp as a plugin won't do much. Gulp needs additional packages (called **plugins**) to perform tasks.

You will also need to create a file called `gulpfile.js` to make use of these plugins and define the tasks you want to run.

When you want to run a task you will type something like this into your terminal:

```
gulp minify-css
```

11.2.2 `gulpfile.js`

This is the beating heart of you rgulp set-up. The is usually split into two distinct sections.

The top part is where you declare the packages needed to use in order to run your tasks. This means including gulp itself and also any plugins it needs to. It'll look something like this:

```
let gulp      = require( 'gulp' );  
let cleanCSS  = require( 'gulp-clean-css' );  
let rename    = require( 'gulp-rename' );
```

The rest of your `gulpfile.js` will contain the definitions of the tasks you want to run. Read on for specific examples of using these.

11.2.3 Minifying CSS files

Minifying CSS files (and also JS files) is something we do to improve the performance on our websites and apps.

A minified file is a file where we have stripped out all the code formatting to make it more human readable. So this means removing the spaces, line returns and comments. The file is still perfectly readable to the browser so it will run just the same.

The advantage is the reduction in file size. We like reducing filesize because it means it is quicker to send to a user over the network and the knock-on effect of that is it leads to faster page loads times. And fast is good!

When we use minified files in a project, we need to keep the original files so we can continue to maintain the code. Minified files are really really difficult to work with.

We end up with two sets of CSS files. We will load the minified version into the project and keep the original files, known as the **source files**.

Usually we put these two different files in different directories to make it clearer which file is for what. For example we might have `/src` (short for source) and `/build` directories in our project.

Here's some CSS code before minification:

```
.card-list {
  width: 85vw;
  margin: 0 auto;
  display: grid;
  grid-template-columns: 1fr 1fr 1fr 1fr;
  grid-gap: 20px;
}

.ui-helper-hidden {
  display: none;
}

.ui-helper-hidden-accessible {
  border: 0;
  height: 1px;
  margin: -1px;
```

```
overflow: hidden;
padding: 0;
position: absolute;
width: 1px;
clip: rect(0 0 0 0);
}
```

And here it is after being minified (you might need to zoom in on the pdf to make this out):

```
.card-list{width:85vw;margin:0 auto;display:grid;grid-template-columns:1fr 1fr 1fr 1fr;grid-gap:20px}.ui-helper-hidden{display:none}.ui
```

Minified CSS in the code editor is displayed all on one line. Here's what I could see without scrolling.

We can write a gulp task to do this minification for us. Here's some code to do that:

```
1.  gulp.task( 'minify-css', () => {
2.    return gulp.src( 'css/style.css' )
3.      .pipe( cleanCSS( {compatibility: 'ie8'} ) )
4.      .pipe( rename( {suffix: '.min'} ) )
5.      .pipe( gulp.dest( './css/' ) );
6.  } );
```

We would then run this task using `gulp minify-css` in the terminal because that is the name of the task we defined in the line 1 of the code.

On line 2 we define which file we want to **pipe in** in this case `css/style.css`. Note that the path to the files is relative to the root of the project where your **gulpfile.json** should be situated.

On line 3 we are saying take the piped file and run the `cleanCSS` function over it. The `cleanCSS` function comes from the `clean-CSS` plugin we required at the top of the file.

On line 4 we rename the file and add a suffix (end) of `.min` to the file.

And line 5 we tell gulp where we want the resulting file to be put. In this case in the `css` directory.

11.2.4 Compiling SASS to CSS

SASS is a language that we can use to write CSS. It is way more fun writing CSS with SASS as it gives us more features to help us make our code DRY and maintainable.

For more info head to the SASS documentation, which is beautifully written. The [SASS guide](#) is a great read, and pretty short, to cover the basics. For more in depth help head to the [SASS documentation](#).

SASS files end with `.scss` (superset of CSS, uses curly brace syntax) or `.sass` (indented syntax) and cannot be read by a browser. They must be **compiled** / **preprocessed** into CSS.

There are lots of ways you can do this, gulp is but one of many options. You could even just use the npm sass package. But we like to use gulp as we can combine the compilation of your SASS with other tasks. More on this in a bit.

To make a SASS task run with gulp, we will need to use the `gulp-sass` plugin as well as others. You will need to include it at the top of your `gulpfile.js` like this:

```
let gulp      = require( 'gulp' );
let rename    = require( 'gulp-rename' );
let sass      = require( 'gulp-sass' );
```

It's the last line that is the new addition in the code example above.

The task itself would look like this:

```
gulp.task('sass', function () {
  return gulp.src('./scss/styles.scss')
    .pipe(sass())
    .pipe(rename('styles.css'))
    .pipe(gulp.dest('./css/'));
});
```

We would then run this task using `gulp sass` in the terminal.

11.2.5 Combining gulp tasks using `series`

So far we've just looked at running gulp tasks one at a time. Most of the time we'll have several tasks we want to run over our files at once.

For example compiling our SASS to CSS, and then minifying the resulting CSS straight after so it's production ready.

Gulp allows us to combine several different tasks and use only one command to run them as a set. Handy!

`gulp.series` allows us to do that. This is an in-built function of gulp so you don't need to include anything else in the top of your `gulpfile.js` to make this work.

Don't forget as before,
Assuming we have two tasks already defined in our `gulpfile.js` called `sass` and `minify-sass` we could combine them together by adding this code to the end of our `gulpfile.js`:

```
gulp.task( 'minify-sass', gulp.series( 'sass', 'minify-css' ) );
```

Then on the command line we can run `minify-sass` and watch both tasks happen one after the other.

11.2.6 Running tasks automatically using `watch`

When we are developing, we'll be constantly amending our files and testing our work to make sure we've got it right. That means you might end up rebuilding your files every few minutes.

Having to go back to your terminal and rerun your commands manually each time is a bit of a drag.

Instead we can get gulp to **watch our files** and perform changes every time we save files. This is an incredible time saver for your workflow.

`gulp.watch` allows us to do that. This is an in-built function of gulp so you don't need to include anything else in the top of your `gulpfile.js` to make this work.

You can add the following code to your `gulpfile.js` to get gulp watching your file:

```
gulp.task( 'watch', function () {  
  return gulp.watch( 'scss/**/*.scss',  
    gulp.task( 'minify-sass' ) );  
} );
```

A word of caution! If you edit your **gulpfile.js** whilst it is running, in this case watching a task, it will keep running your old version.

You need to stop gulp running using **Ctrl + C** or **Apple + C** and then restart it to see your new changes take affect.

11.2.7 Refreshing your browser when you make changes

Another brilliantly useful task you can get gulp to run is a browser refresh task.

This allows you to get your browser to automatically refresh and show you your changed output. It's definite timesaver when you are working hard on a project.

A good gulp plugin for this is called [browser-sync](#).

If you need some help with implementing this have a look at our [code sample in the git repo](#).

11.2.8 Finding out what tasks are already defined in `gulpfile.js`

If you are given a `gulpfile.js` to use in a project it's usually helpful to figure out what tasks have already been defined.

You can use `gulp --tasks` to give you an output. That output usually looks something like this:

```

guests-MacBook-Pro:sample-answers hannah$ gulp --tasks
[14:40:20] Tasks for ~/Dropbox/Apps/local/developme/sample-answers/gulpfile.js
[14:40:20] └─ minify-js
[14:40:20] └─ minify-css
[14:40:20] └─ sass
[14:40:20] └─ minify-sass
[14:40:20]     └─ <series>
[14:40:20]         └─ sass
[14:40:20]             └─ minify-css
[14:40:20] └─ watch-css
[14:40:20] └─ watch-js
[14:40:20] └─ watch-all
[14:40:20]     └─ <parallel>
[14:40:20]         └─ watch-css
[14:40:20]             └─ watch-js

```

A summary of all the tasks contained in a gulpfile

11.3 Managing build merge conflicts

When gulp, or any other task runner, is involved in creating build files you will find you frequently get merge conflicts with the built files.

Taking the example of compiling SASS into CSS and then minifying it, the resulting minified CSS is likely to cause conflicts in pretty much every merge. Because a minified file is essentially one line of CSS, Git will trigger a merge conflict because from it's point of view the same lines of code have been edited.

Trying to resolve a merge conflict in a minified file is a pretty horrendous job for any developer. So prone to mistakes and problems. We just wouldn't really expect someone to have to resolve this by hand.

Instead, a common practice is that when you are resolving the merge conflict you just accept all the incoming changes for the CSS file and commit that. Or the original changes - it actually doesn't really matter. Once you have completed the merge you then regenerate the CSS from the source files and add that as a new commit.

It is important to note that resolving merge conflicts on the source files, eg the original files is still really important and you need to pay attention. But the final build/compiled files don't require too much thought as you just regenerate them as a separate commit.

There are other approaches such as never committing the final build files and have those generated on the fly when code is submitted. But those require different skills to implement Continuous Improvement (CI) or Continuous Deploymet (CD) pipelines.

As with so many things in the world of development, there are lots of ways to approach the same problems. **What is right and wrong is usually a matter of context and team culture** so our advice is to keep it practical. Get something working that solves some immediate problems for you and your team first. You can always iterate and improve any solution later down the line as you try it out and learn where the pain points are.

Colophon

Created using T_EX

Fonts

- **Feijoa** by Klim Type Foundry
- **Avenir Next** by Adrian Frutiger, Akira Kobayashi & Akaki Razmadze
- **Fira Mono** by Carrois Apostrophe

Written by Oli Ward and Hannah Smith



October 29, 2020