Version Control Walkthrough with



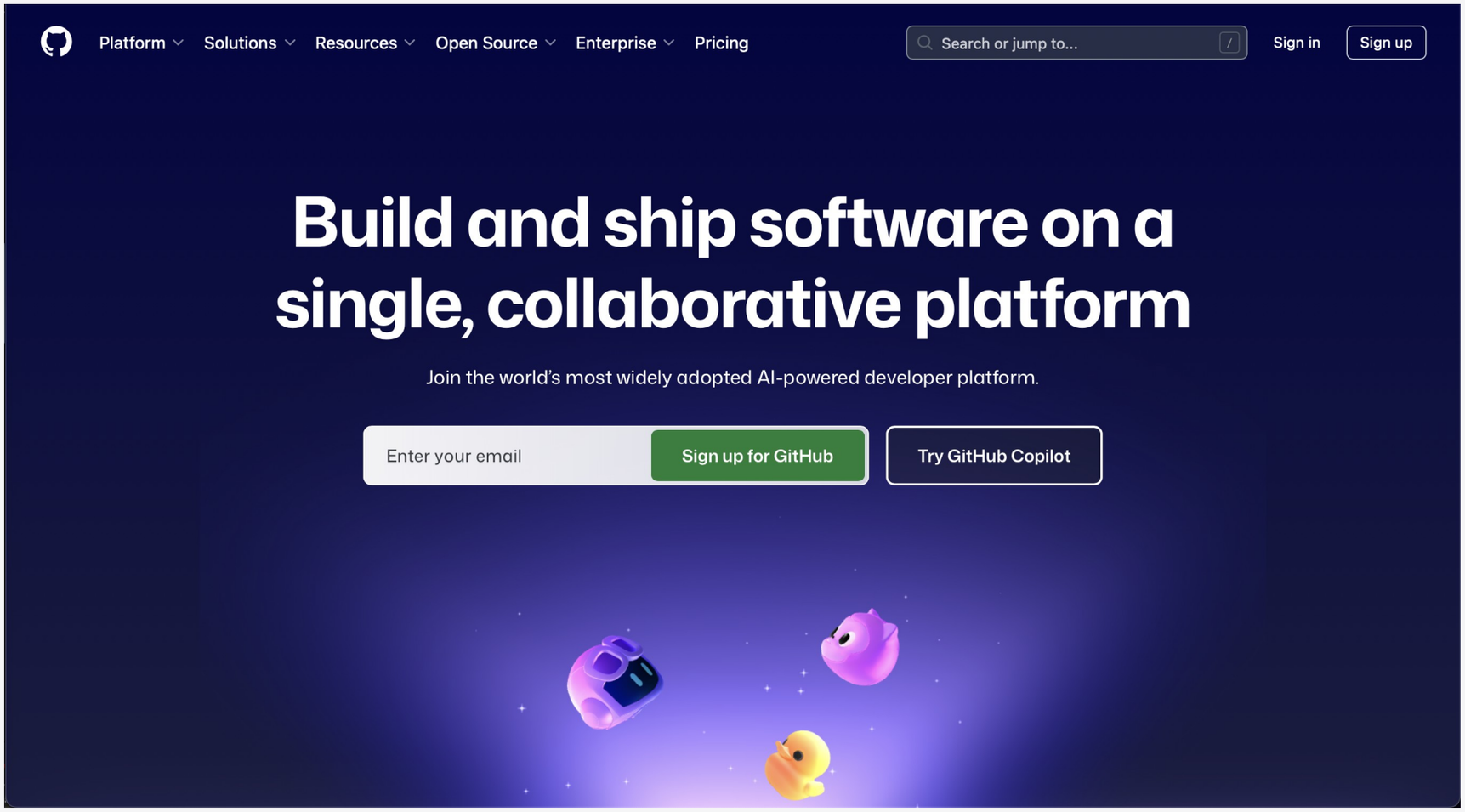
Ben Morgan (based on slides by Mark Slater)

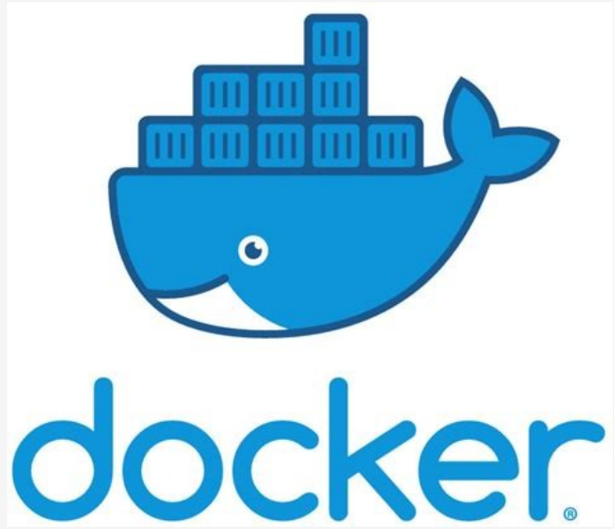
THE UNIVERSITY OF WARWICK

UNIVERSITY OF BIRMINGHAM

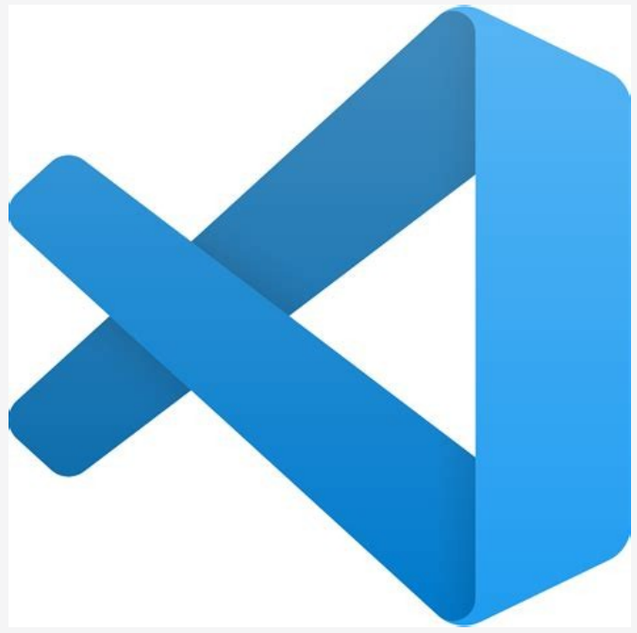
A Version Control Walkthrough with Git

- We haven't actually written anything yet, and this is the perfect time to get mpags-cipher under version control so you can use it through the whole course. We'll be using **git** as our version control system, with **github.com** acting as a central repository.
- We have created starting repositories that you can 'fork' (or copy) which means you have a clean repo to start each session from. You can then apply your changes to this and we can add comments, changes, etc. within github.
- Aims of the walkthough:
  - *Create a repo from the Github Assignment, get a working copy into Visual Studio Code, add files, commit changes and make tags*
  - *Show diffs and logs for the commits we've made*
  - *Push our local changes to the central repo, Pull changes from another repo*





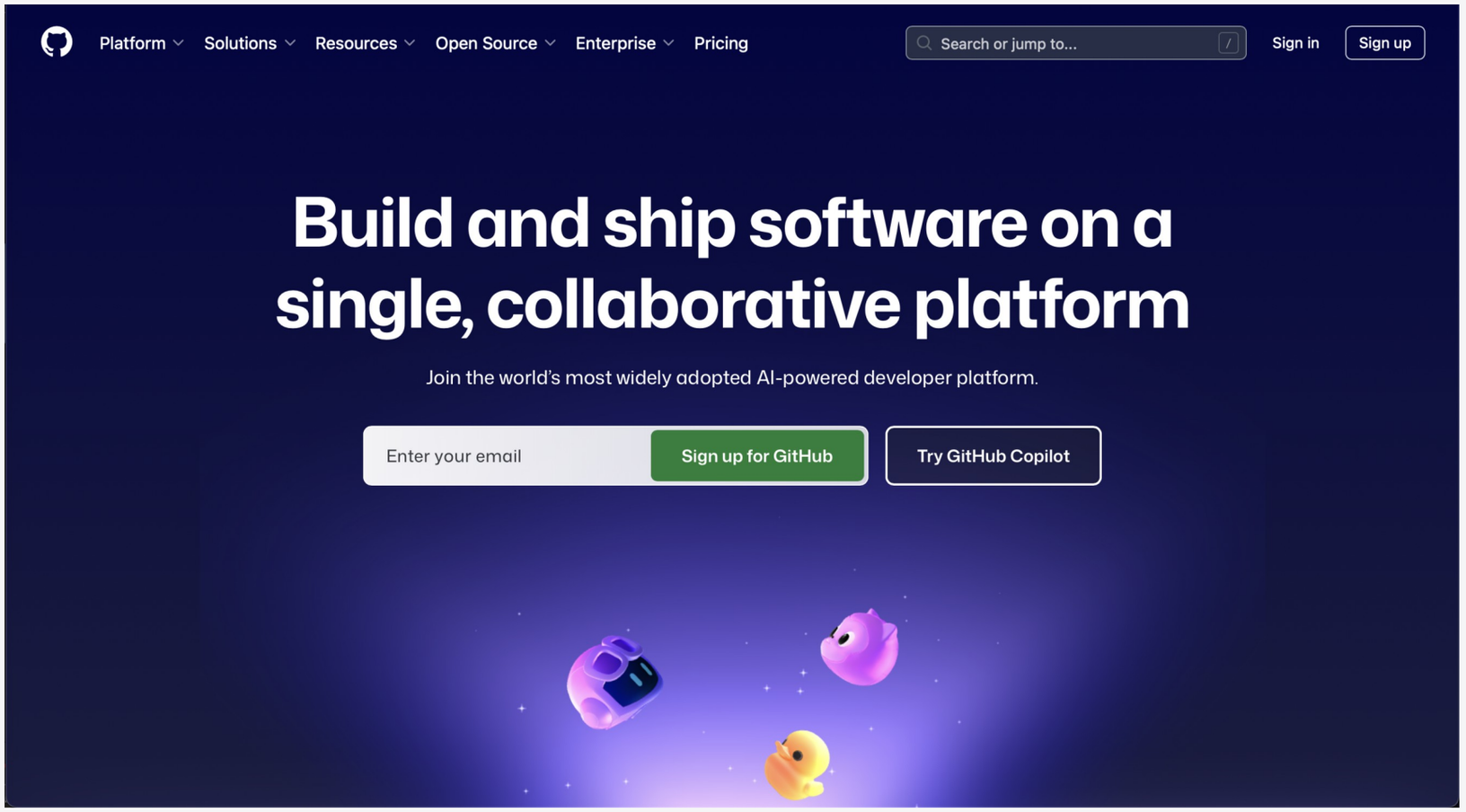




Tools you'll need

1: Creating a Github Account

Whilst git is completely distributed, to help with working in several locations and to supply solutions, you'll create a repository to be an authoritative one. This will be created for you when you go to the link for each assignment on the webpage for that session. However, before you can do anything in Github, you need to sign up for an account.



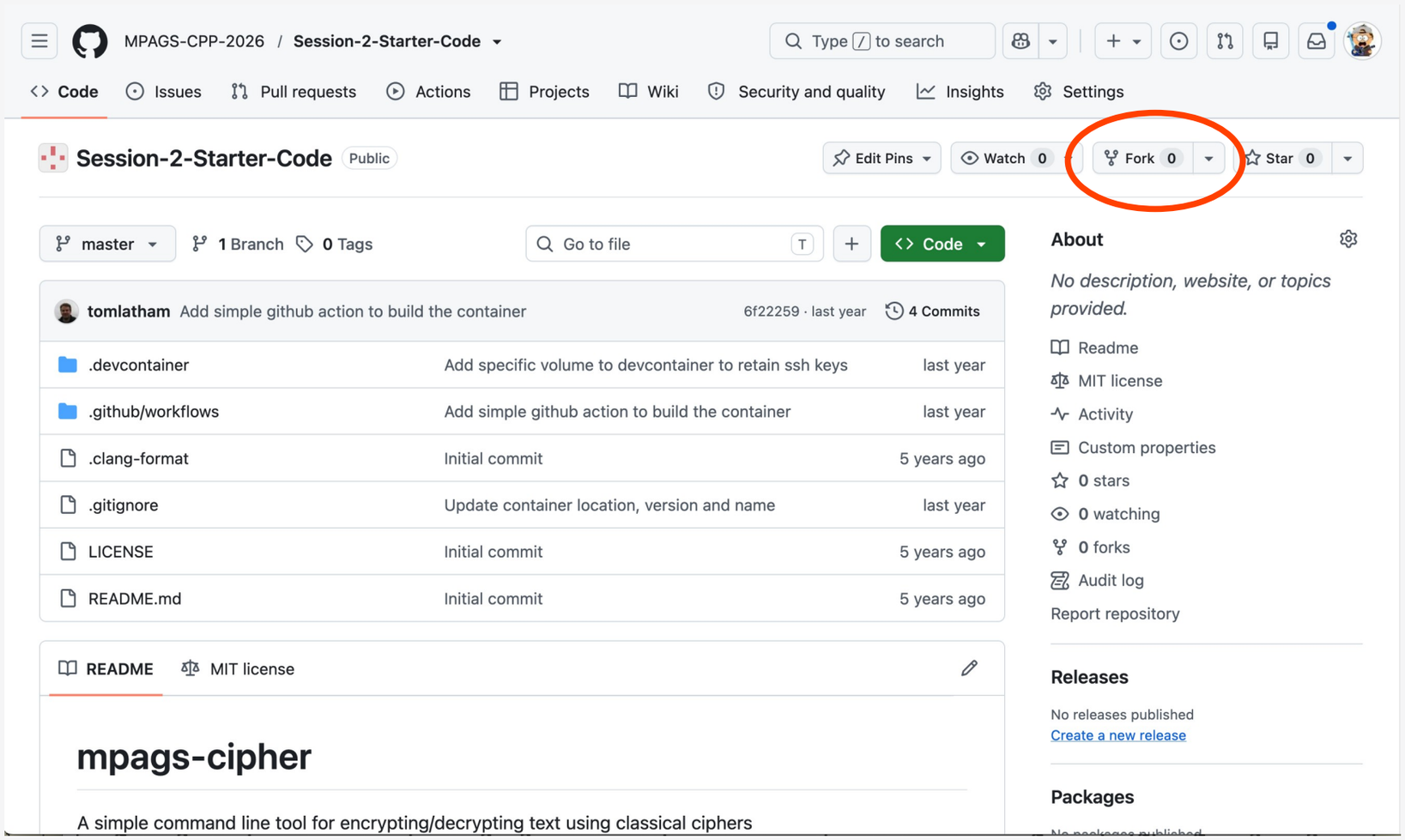
Notes

Other hosting services exist for git, though github is currently the most popular.

Similar hosting services exist for other VCSs.

2: Creating Your mpags-cipher Repository

You can create repositories in github in a number of ways: Creating one from scratch on Github, pushing an already existing local repo, forking an existing repo on Github, etc. For this course, you will 'fork' (similar to a copy) a base repo that you'll be working from on each session that has been created by us. This will also include the docker container definition that will provide all the tools we'll need. To create the repo for this session, go to the link on the course webpage, click 'fork' and 'Create Fork' on the following page to accept the defaults. This should create a new version of the repository in your personal github area



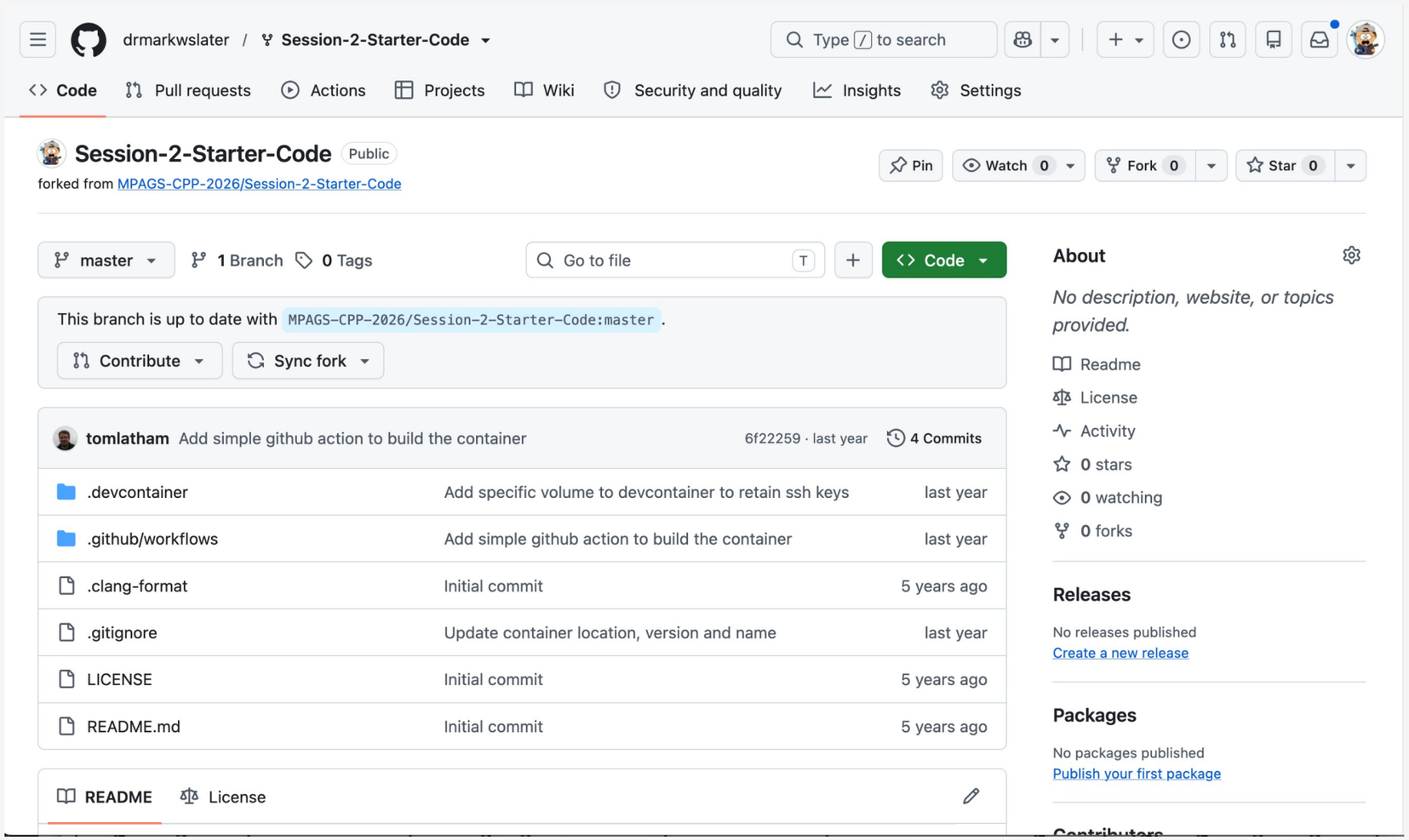
Notes

Though you will be creating a new repository for each session of the course, you are welcome to keep the previous repos as long as you wish to refer back to.

3: Your mpags-cipher Repository

After ‘forking’ the repository, github should send you to your version of the repo. You can also find it and your other personal repos in your personal github space by going to:

https://github.com/repos?q=owner%3A%40me



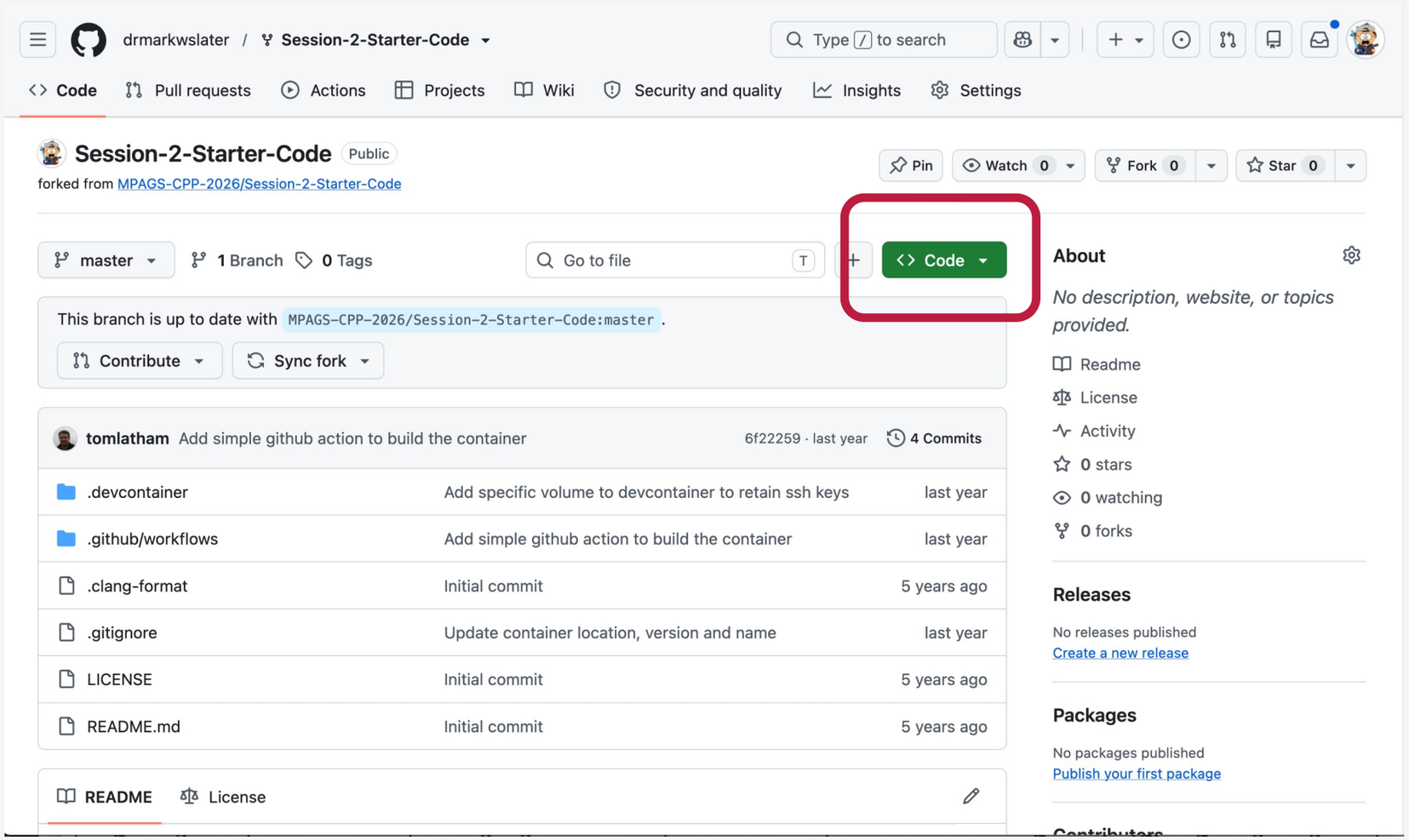
Notes

Though you don’t need to for this course, you can create repos from scratch within github or from the command line (using `git init`) and then push to github.

4: Accessing Your Repo with VS Code and Docker

To access your repo through the container, start Visual Studio Code and select 'View → Command Palette'. In the dialog that pops up, start typing 'clone repository', selecting 'Clone Repository in Named Container Volume'. In the next box, paste the URL of your repo, then choose a name for the volume (you can reuse this in future weeks), and finally accept the default folder name. Note that if you just want to clone a repository from the command line, use:

```
$ git clone <repourl> <localdir>
```



Notes

We'll use the **https** protocol to clone the repository as we haven't setup SSH keys yet and github has stopped supporting username+password authentication. The URL for cloning your GitHub repo can be found on the right hand side of its front page, as shown on the left. Click on the 'https' tab and copy the URL there.

5: Command Line Git: Getting Help

With the repo in place and the container started, we can now start working with git. In VSC, you can access the Terminal of the container by going to 'View → Terminal'. This will operate just as a normal Linux terminal – try doing the following to get help on git:

```
$ git help
```

```
[[macbook]$ git help
usage: git [-v | --version] [-h | --help] [-C <path>] [-c <name>=<value>]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--no-lazy-fetch]
           [--no-optional-locks] [--no-advice] [--bare] [--git-dir=<path>]
           [--work-tree=<path>] [--namespace=<name>] [--config-env=<name>=<envvar>]
           <command> [<args>]

These are common Git commands used in various situations:

start a working area (see also: git help tutorial)
   clone      Clone a repository into a new directory
   init       Create an empty Git repository or reinitialize an existing one

work on the current change (see also: git help everyday)
   add        Add file contents to the index
   mv         Move or rename a file, a directory, or a symlink
   restore    Restore working tree files
   rm         Remove files from the working tree and from the index

examine the history and state (see also: git help revisions)
   bisect     Use binary search to find the commit that introduced a bug
   diff       Show changes between commits, commit and working tree, etc
   grep       Print lines matching a pattern
   log        Show commit logs
   show       Show various types of objects
   status     Show the working tree status

grow, mark and tweak your common history
   backfill   Download missing objects in a partial clone
   branch     List, create, or delete branches
   commit     Record changes to the repository
   merge      Join two or more development histories together
   rebase     Reapply commits on top of another base tip
   reset      Reset current HEAD to the specified state
```

Notes

As the output of git help notes, to get help on a specific command, simply append its name to git help. It will open a man style page for the command (simply use 'q' to exit this).

Of course, also refer to text books and online resources such as git-scm.com.

6: Repository Structure

If not there already, change into your repository's directory and run `ls -larth` to see all the files. Apart from the files you'll have seen on the original GitHub site, there's an extra directory, `.git`. This is git's "database" holding the complete history of changes plus configuration information. It's this holding of the complete project history that allows the distributed version control

Note that in the images I have run: `alias ls='ls --color'` so directories are in blue

```
[macbook]$ ls -larth
total 32
drwx------  46 benmorgan  staff   1.4K 20 Oct 12:56 ..
-rw-r--r--   1 benmorgan  staff   3.3K 20 Oct 12:56 .clang-format
drwxr-xr-x   4 benmorgan  staff   128B 20 Oct 12:56 .devcontainer
drwxr-xr-x   4 benmorgan  staff   128B 20 Oct 12:56 .github
-rw-r--r--   1 benmorgan  staff   400B 20 Oct 12:56 .gitignore
-rw-r--r--   1 benmorgan  staff   1.0K 20 Oct 12:56 LICENSE
drwxr-xr-x   9 benmorgan  staff   288B 20 Oct 12:56 .
-rw-r--r--   1 benmorgan  staff    97B 20 Oct 12:56 README.md
drwxr-xr-x  12 benmorgan  staff   384B 20 Oct 12:56 .git
[macbook]$
```

Notes

As this directory holds all of your changes, be very careful not to delete it!

We won't dig into the content or structure of .git in this course. If you're interested in learning more about this, the main Git references cover it in detail.

7: Viewing Repository Status

As we add and edit files, it's useful to keep track of the repository status without changing anything. With git, simply use the **`status`** command to view the current repository status:

```
$ git status
```

```
[[macbook]$ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
[macbook]$
```

Notes

Of course at the moment, there's not much to report as we have not made any changes yet

Get into the habit of running git status regularly to see what you've changed.

8: Configuring Git

Before we start to use **`git`**, it's useful to configure it with the details to record in commit messages, an editor to use to write messages and to display changes using colour markup. The **`config`** command sets parameters, ‹key›, that **`git`** knows about to required values

```
$ git config --global <key> <value>
```

```
[[macbook]$ git config --global -l
user.name=Ben Morgan
user.email=ben.morgan@warwick.ac.uk
alias.mr=!sh -c 'git fetch $1 merge-requests/$2/head:mr-$1-$2 && git checkout mr-$1-$2' -
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
pull.ff=only
[macbook]$
```

Hints

Git can use the **`EDITOR`** environment variable rather than **`core.editor`**

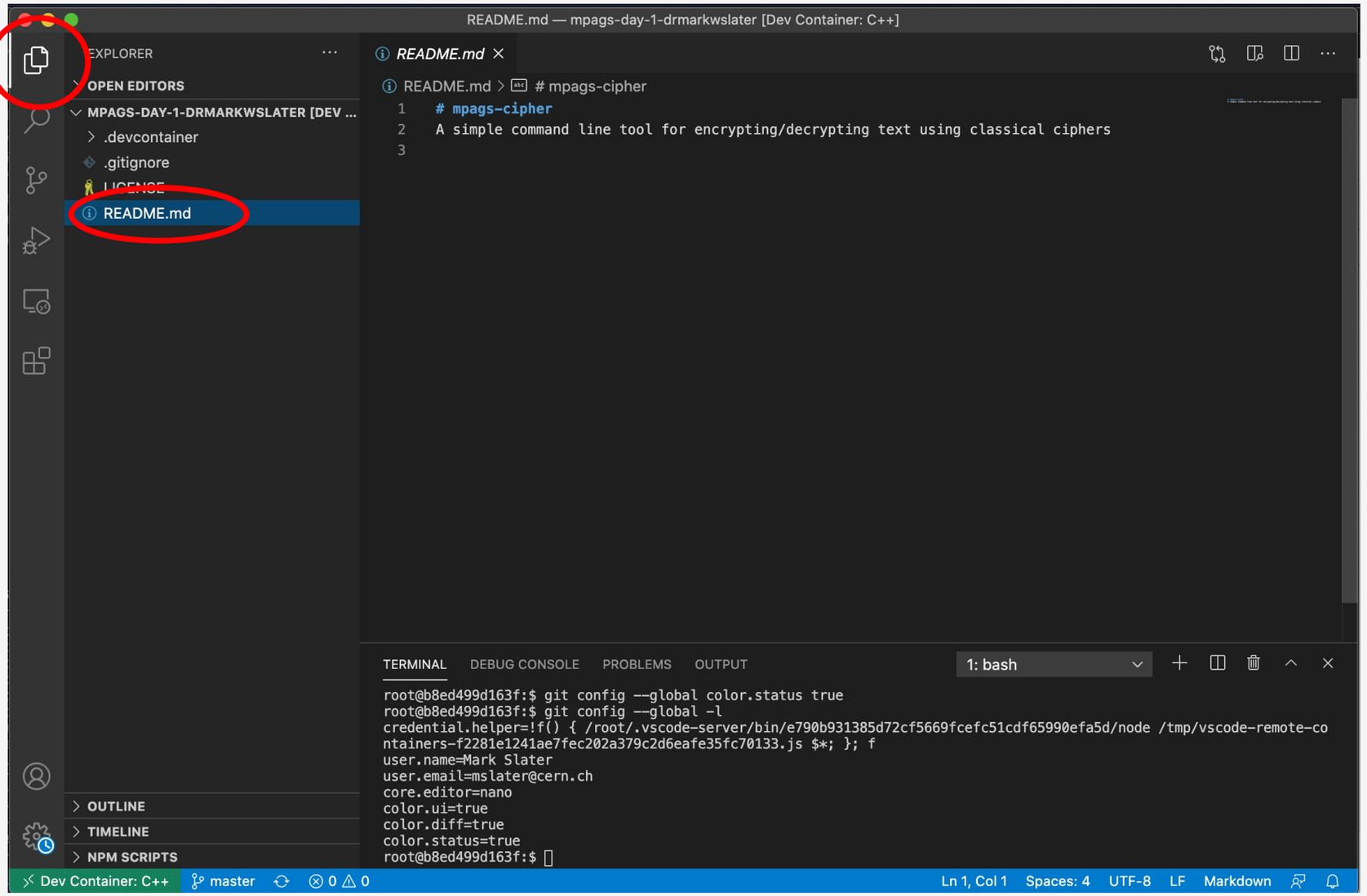
You can configure git options globally (**`--global`**) or locally in a repository (**`--local`**)

Also see the Git SCM Book and try out some options!

9: Improving the README file

We're going to start our project by improving the **`README`** for **`mpags-cipher`**. This is a file that sits in the top level of the project and provides some basic information about the project, how to install it, and other details such as author/copyright/license details. Our **`README`** is in plain text, using Markdown formatting. We use Markdown because it is human readable but easily convertible to other formats.

In the 'Explorer' tab, click on '**`README.md`**' to open it. Edit the file and then save it.



Notes

Add a bit more detail about the project, and create placeholder sections for "How to Install" and "Authors". Use the link above for a format guide

'*#*' marks major sections, and those with '*##*' are subsections. When we upload our project to github, we'll see how these are displayed.

10: Staging Changes for Commit

Having saved **`README.md`**, if you run **`git status`** again, git can see it's been changed. However, git marks these changes as "unstaged", i.e not yet ready to be committed to its database. To tell git we want to "stage" the changes, use the **`git add`** command on the file(s):

```
$ git add README.md
```

```
[[macbook]$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
[[macbook]$ git add README.md
[[macbook]$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   README.md

[macbook]$
```

Notes

"staged" files

- Ready to be commited

"unstaged" files

- Changed but not staged

"untracked" files

- Not tracked by git yet

"deleted" files

- Deleted by git and ready for removal

11: Committing Changes

You'll have noticed that when you ran **`git status`** after adding **`README.md`** it only says "*Changes to be committed*". Git stages changes before committing them to the repository. To store the changes we use the **`commit`** command with a message describing the changes:

```
$ git commit -m "Improve README"
```

```
[[macbook]$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   README.md

[[macbook]$ git commit -m "Improve README"
[master 88a91f8] Improve README
 1 file changed, 4 insertions(+)
[[macbook]$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
[macbook]$
```

Notes

The staging area is a place to queue up (or remove) changes before they are committed. This is useful when we start to deal with changes across multiple files

A "commit" is a snapshot of the repository. After the commit, **`status`** shows that we're ahead of where we cloned from (GitHub).

12: Making Further Changes

Now we've staged and committed **`README.md`**, we can continue to make changes. So edit your **`README.md`**, for example, add an empty section "Documentation". Save the file and run **`git status`** again. As before, git recognises we've made changes, but these are not yet staged. To actually update the repository, we run **`git add`** again to stage the change then **`git commit`** to update the repository. This cycle of staging and committing is the basic git workflow.

```
[[macbook]$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
[[macbook]$ git add README.md
[[macbook]$ git commit -m "Add section for documentation"
[master 33f6295] Add section for documentation
 1 file changed, 3 insertions(+)
[[macbook]$ git status
On branch master
Your branch is ahead of 'origin/master' by 2 commits.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
[macbook]$
```

Try This

Make a few more edits to **`README.md`** and use **`git add`** and **`git commit`** for each to get into the feel of staging and committing.

Remember to use **`git status`** regularly to see what's happening!

13: Unstaging Changes

So you've staged a change, and then you realise it either breaks something or you want to add something else. As you may have noticed, git status actually tells you what to do in this case. So, make a change, stage it up and then use **`git restore`** to unstage it:

```
$ git restore --staged README.md
```

```
[[macbook]$ git status
On branch master
Your branch is ahead of 'origin/master' by 2 commits.
  (use "git push" to publish your local commits)

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
[[macbook]$ git add README.md
[[macbook]$ git status
On branch master
Your branch is ahead of 'origin/master' by 2 commits.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   README.md

[[macbook]$ git restore --staged README.md
[[macbook]$ git status
On branch master
Your branch is ahead of 'origin/master' by 2 commits.
  (use "git push" to publish your local commits)

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   README.md

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

Notes

If you have a change staged then make further changes, just use **`git add`** to append these to the staging area.

Note that you need to do this if you've staged a file then made further changes to it.

14: Adding New Files

Staging/committing new files is the same as working with existing files. Create a new file named **`mpags-cipher.cpp`** with a single line "**`// mpags-cipher.cpp`**" and save it. Run **`git status`** again, and git recognises a new "Untracked" file, so just use **`git add`** to track it:

```
$ git add mpags-cipher.cpp
```

```
[[macbook]$ git status                                                          ]
On branch master
Your branch is ahead of 'origin/master' by 2 commits.
  (use "git push" to publish your local commits)

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        mpags-cipher.cpp

nothing added to commit but untracked files present (use "git add" to track)
[[macbook]$ git add mpags-cipher.cpp                                            ]
[[macbook]$ git status                                                          ]
On branch master
Your branch is ahead of 'origin/master' by 2 commits.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   mpags-cipher.cpp

[[macbook]$ git commit -m "Add placeholder for main program"                    ]
[master a5a571e] Add placeholder for main program
 1 file changed, 1 insertion(+)
 create mode 100644 mpags-cipher.cpp
[macbook]$
```

Try This

Once staged, git recognises **`mpags-cipher.cpp`** as a "**`new file`**". However, the next commit step is identical, so just commit as normal!

Commits can contain both new files and modifications to existing ones. Note that you can run **`git add`** with multiple files at once.

15: Removing Files

Files can also be removed, but note that git's **`rm`** command removes the file(s) from both the repository and local disk! Git regards a deletion as a change, so **`rm`** also stages the deletion for commit (though the physical file has been deleted). Try removing the **`mpags-cipher.cpp`** file:

```
$ git rm mpags-cipher.cpp
```

```
[[macbook]$ git status
 On branch master
 Your branch is ahead of 'origin/master' by 3 commits.
   (use "git push" to publish your local commits)

 nothing to commit, working tree clean
[[macbook]$ ls
 LICENSE          mpags-cipher.cpp README.md
[[macbook]$ git rm mpags-cipher.cpp
 rm 'mpags-cipher.cpp'
[[macbook]$ git status
 On branch master
 Your branch is ahead of 'origin/master' by 3 commits.
   (use "git push" to publish your local commits)

 Changes to be committed:
   (use "git restore --staged <file>..." to unstage)
         deleted:    mpags-cipher.cpp

[[macbook]$ ls
 LICENSE   README.md
[[macbook]$ git commit -m "Remove main program temporarily"
 [master 32bcffa] Remove main program temporarily
  1 file changed, 1 deletion(-)
  delete mode 100644 mpags-cipher.cpp
[[macbook]$ ls
 LICENSE   README.md
 [macbook]$
```

Try This

After using **`git rm`**, make the commit.

As with all changes, you can stage deletions along with additions and modifications.

Like **`git add`**, **`git rm`** can be run with several files at once.

16: Viewing Logs

We've now made a few commits to our repository, so how do we go back and see what we've done and why? Just use the `log` command!

```
$ git log
```

```
commit 32bcffaac492921d3d99bfbdf267f3a026aa9c54 (HEAD -> master)
Author: Ben Morgan <ben.morgan@warwick.ac.uk>
Date:   Mon Oct 20 13:22:37 2025 +0100

    Remove main program temporarily

commit a5a571ed5911ca71ab868eff95eb84a9ff262f74
Author: Ben Morgan <ben.morgan@warwick.ac.uk>
Date:   Mon Oct 20 13:21:22 2025 +0100

    Add placeholder for main program

commit 33f62954b8d7d370ca93497ebd2f092e80ad08ed
Author: Ben Morgan <ben.morgan@warwick.ac.uk>
Date:   Mon Oct 20 13:15:06 2025 +0100

    Add section for documentation

commit 88a91f86fe835ce1c3b084e3744efb75c88d650b
Author: Ben Morgan <ben.morgan@warwick.ac.uk>
Date:   Mon Oct 20 13:13:41 2025 +0100

    Improve README

commit e18699535d82da3d57524d4f0898fbf907f2bff8 (origin/master, origin/HEAD)
Author: github-classroom[bot] <66690702+github-classroom[bot]@users.noreply.github.com>
Date:   Mon Oct 20 11:49:49 2025 +0000

    Setting up GitHub Classroom Feedback

commit 8c4637fcc34431dcb05535894e7eecc914cf8575 (origin/feedback)
Author: github-classroom[bot] <66690702+github-classroom[bot]@users.noreply.github.com>
Date:   Mon Oct 20 11:49:48 2025 +0000

:
```

Try This

Plain log displays everything! To get the N most recent commits, use `git log -nN`.

You can also use `git log --summary` to get a more detailed overview, though it doesn't show much as we've only worked with one file.

17: Viewing Changes

The basic log command shows the timeline of changes, but not what changed. To see what actually changed between commits, we can use **`git log -p`** or the **`diff`** command. Without any arguments, it shows a diff between the last commit and any **unstaged** changes:

```
$ git diff
```

```
[macbook]$ git diff
diff --git a/README.md b/README.md
index 198b9ce..f9bfd3e 100644
--- a/README.md
+++ b/README.md
@@ -4,6 +4,7 @@ A simple command line tool for encrypting/decrypting text using classical c
ipher
 # How to install

 # Documentation
+Takes input and runs a variety of classical ciphers to produce output.

 # Authors
-
+- Ben Morgan
[macbook]$
```

Notes

Git shows difference using the standard diff format for additions/removals. On the left, additions are in green, removals in red.

Depending on the default configuration, the diff may be output to a pager, in which case use ‘q’ to quit.

18: Changes between Commits

As you'll have seen in using `git log`, git labels commits using a 40 character hash code. You can use these labels to view differences between any two commits, though because hashes are unique, you don't have to type out 80 characters, e.g:

```
$ git diff <from> <to>
```

```
[[macbook]$ git diff 88a9 a5a5
diff --git a/README.md b/README.md
index ae70fac..198b9ce 100644
--- a/README.md
+++ b/README.md
@@ -3,4 +3,7 @@ A simple command line tool for encrypting/decrypting text using classical c
ipher

 # How to install

+# Documentation
+
 # Authors
+
diff --git a/mpags-cipher.cpp b/mpags-cipher.cpp
new file mode 100644
index 0000000..c1d9f8c
--- /dev/null
+++ b/mpags-cipher.cpp
@@ -0,0 +1 @@
+// mpags-cipher.cpp
[macbook]$
```

Try This

The commit specifier needs to contain enough characters to uniquely identify the commit. Note that your hashes will differ!

The arguments to git diff can take a variety of forms. See `man gitrevisions` for more details, or the more helpful Git SCM Book! Try some of these out.

19: Writing Good Commit Messages

Our edits so far have been simple and confined to one file. In these cases, a single line commit message using **`git commit -m "commit message"`** is completely sufficient (e.g. "Fixed typographic errors"). As we start to make more involved commits involving several files, then we need to provide more detail. Because of the way git works with patches and email, it tends to recommend the specific style of commit message listed below.

```
# Contributing
## Writing Good Commit Messages
Capitalized, short (50 chars or less) summary

More detailed explanatory text, if necessary.  Wrap it to about 72
characters or so.  In some contexts, the first line is treated as the
subject of an email and the rest of the text as the body.  The blank
line separating the summary from the body is critical (unless you omit
the body entirely); tools like rebase can get confused if you run the
two together.

Write your commit message in the imperative: "Fix bug" and not "Fixed bug"
or "Fixes bug."  This convention matches up with commit messages generated
by commands like git merge and git revert.

Further paragraphs come after blank lines.

- Bullet points are okay, too

- Typically a hyphen or asterisk is used for the bullet, followed by a
  single space, with blank lines in between, but conventions vary here

- Use a hanging indent

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch Day1Branch
# Your branch is ahead of 'origin/Day1Branch' by 5 commits.
#   (use "git push" to publish your local commits)
#
-- INSERT --
```

Why?

There's a good example in the text on the left (taken from a post by Tim Pope).

If you "fixed a bug" you should say which bug, and how it was fixed. You might also say (and include in the commit) that a test has been added to check for the bug in the future.

20: The .gitignore File

When you run `git status`, git will report any files it doesn't track ("untracked"). In some cases we'll have files that we don't want git to track, for example files generated by the build or text editor temporaries, but we may accidentally add them to the repository (e.g. by `git add .`).

Git uses the .gitignore file in our repository (provided by upstream) to determine what to ignore. This contains a list of filename patterns that git should ignore and already contains patterns for C++ compiled objects. Have a look at the .gitignore provided in the repository and make sure it looks like below:

.gitignore

```
1   # Lines beginning with a '#' are comments
2   # Text Editor Temp Files
3   *~
4   *.swp
5
6   # Mac Filesystem
7   .DS_Store?
8
9   # Compiled Object files
10  *.slo
11  *.lo
12  *.o
13  *.obj
14
15  # Precompiled Headers
16  *.gch
17  *.pch
18
19  # Compiled Dynamic libraries
20  *.so
21  *.dylib
22  *.dll
23
```

Notes

You can also have a global ignores file. You could have a file named `.global_gitignores` in your `HOME` directory. Git can be made aware of this file by setting the `core.excludesfile` variable to point to it in the global `git config`

21: Tagging

We've seen that in git, commits are described by a 40 character hash. At certain points in development, we'll want to mark a commit as a usable, stable piece of work. The hashes aren't an easy way of marking these points, so instead we create a "tag". Current tags are listed via:

```
$ git tag
```

```
[[macbook]$ git status
On branch master
Your branch is ahead of 'origin/master' by 5 commits.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
[[macbook]$ git tag
[[macbook]$ git tag -a v0.1.0 -m "mpags-cipher 0.1.0: End of git walkthrough"
[[macbook]$ git tag
v0.1.0
[[macbook]$ git show v0.1.0
tag v0.1.0
Tagger: Ben Morgan <ben.morgan@warwick.ac.uk>
Date:   Mon Oct 20 13:29:20 2025 +0100

mpags-cipher 0.1.0: End of git walkthrough

commit 529b18e0de9e8342b8159c700dfe6777b70c04a8 (HEAD -> master, tag: v0.1.0)
Author: Ben Morgan <ben.morgan@warwick.ac.uk>
Date:   Mon Oct 20 13:28:42 2025 +0100

    Some more README improvements

diff --git a/README.md b/README.md
index 198b9ce..f9bfd3e 100644
--- a/README.md
+++ b/README.md
@@ -4,6 +4,7 @@ A simple command line tool for encrypting/decrypting text using classical c
ipher
 # How to install

 # Documentation
+Takes input and runs a variety of classical ciphers to produce output.

 # Authors
```

Try This

Follow the steps on the left to tag your repository,

Tags can have any name, but git projects tend to use 'vMAJOR.MINOR.PATCH' for version numbers. "Annotated" tags are the best to use to begin with, as they can take extra info about the tag. Use show to see this info.

22: Sharing Changes between Repositories

Whilst we've made commits to our repository, these are all local as we work on a copy of the repository. If you go back to the GitHub page for your project and refresh the browser, you'll see that this is still in its original state. The distributed nature of git means it can track a set of repositories, which we can view with the remote command:

```
$ git remote -v
```

```
[[macbook]$ git status
 On branch master
 nothing to commit, working tree clean
[[macbook]$ git remote -v
 origin  https://github.com/MPAGS-CPP-2025/mpags-session-2-drbenmorgan.git (fetch)
 origin  https://github.com/MPAGS-CPP-2025/mpags-session-2-drbenmorgan.git (push)
 [macbook]$
```

Notes

Because we cloned from Github, the default "origin" remote points to it.

The "-v" flag gets git to show the full URLs.

As we'll see, we can have multiple remotes.

23: Preparing to Send your Repository to github

We now want to send our changes to github. However, to do this we need to generate an **ssh key** consisting of a **public** part and a **private** part. You send the **public** part to github and keep the **private** part on your machine and using using the combination, github can authenticate you. To generate the key, run the following:

```
$ ssh-keygen -t ed25519
```

```
root@98c7fce9651d:/workspaces# ssh-keygen -t ed25519
Generating public/private ed25519 key pair.
Enter file in which to save the key (/root/.ssh/id_ed25519):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /root/.ssh/id_ed25519.
Your public key has been saved in /root/.ssh/id_ed25519.pub.
The key fingerprint is:
SHA256:dul/fwRP5xGbob+3icHjKl1KH9zwHWy3XUklLOGi9+U root@98c7fce9651d
The key's randomart image is:
+--[ED25519 256]--+
|            .o. o|
|           .. .= |
|          . ..+ *|
|         . o o.O=|
|        S + . B=X|
|       . + o.* =*|
|          + =+E..|
|         . +..=.+|
|          ..o+ +=|
+----[SHA256]-----+
root@98c7fce9651d:/workspaces#
```

Notes

During this process you will be asked to provide a passphrase. This will ‘unlock’ the ssh key when you push to github.

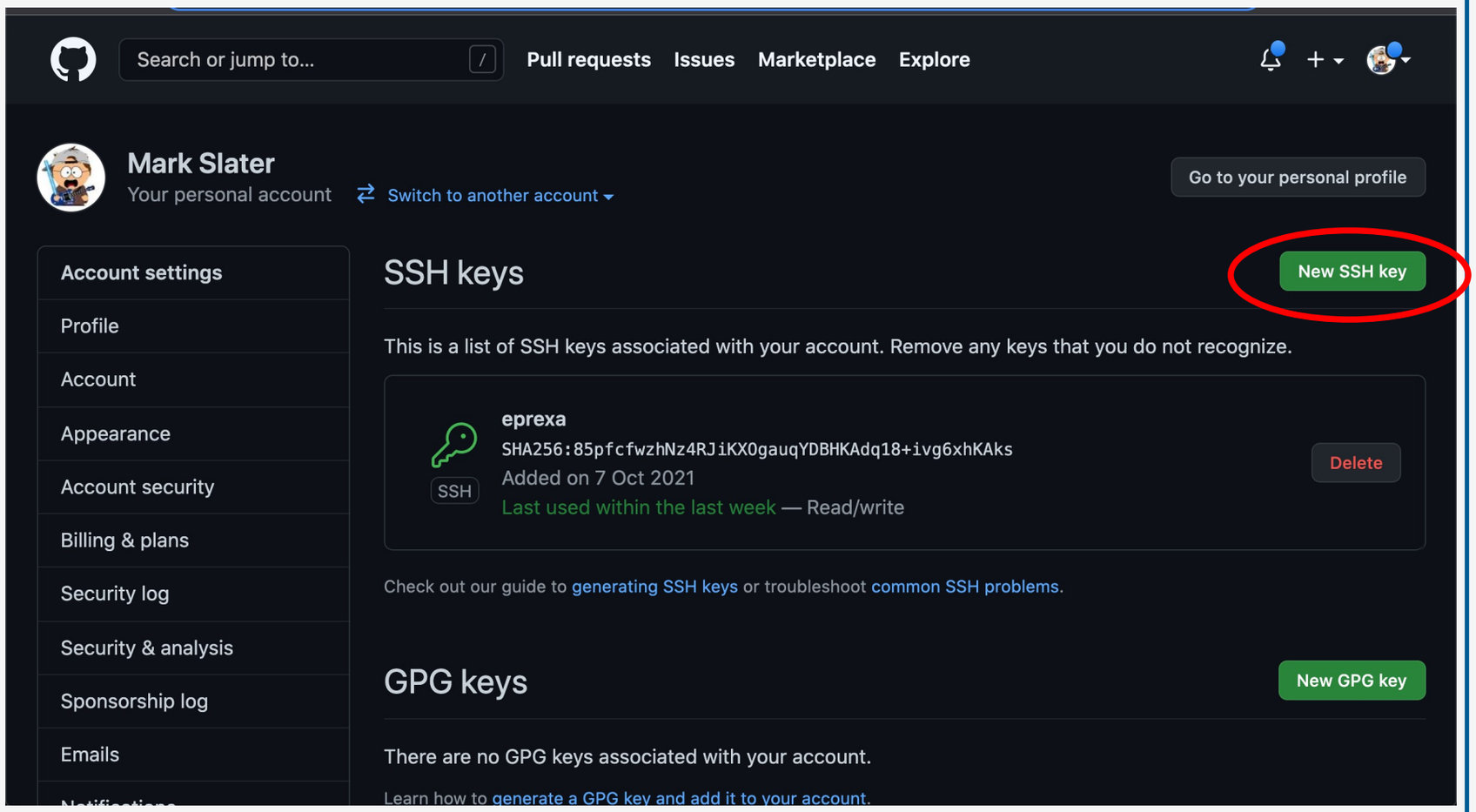
Though this is not sent over the internet and is useless without the private part of the key, you should still ensure it’s not easily guessable!

24: Adding the Public SSH Key Part to Github

You now have two additional files: `~/.ssh/id_ed25519` and `~/.ssh/id_ed25519.pub`. The second of these is the **public** part which you can freely give out – **DON'T GIVE OUT THE PRIVATE PART!** To provide this to github, go to:

https://github.com/settings/keys

And copy and paste the contents of `~/.ssh/id_ed25519.pub` file as well as giving it a title.



Notes

You can store many keys on github so if you are using several machines, you can either copy the private and public parts to the different machines or generate different keys for each.

As set up, the Docker image you're using *should* retain and set up your key between sessions.

25: Changing the remote URL

We need to make one more change – when we downloaded the repo we used 'https' but we want to **push** our changes using **ssh**. To change this, we will change the URL or the 'origin' remote:

```
$ git remote set-url origin <remote-url>
```

The URL you need can be found by just replacing the 'https://github.com/' part to 'git@github.com:', e.g.

https://github.com/MPAGS-CPP-2026/Session-2-Starter-Code.git → git@github.com:MPAGS-CPP-2026/Session-2-Starter-Code.git

```
[[macbook]$ git status
 On branch master
 nothing to commit, working tree clean
[[macbook]$ git remote -v
 origin  https://github.com/MPAGS-CPP-2025/mpags-session-2-drbenmorgan.git (fetch)
 origin  https://github.com/MPAGS-CPP-2025/mpags-session-2-drbenmorgan.git (push)
[[macbook]$ git remote set-url origin git@github.com:MPAGS-CPP-2025/mpags-session-2-drbenmor]
 gan.git
[[macbook]$ git remote -v
 origin  git@github.com:MPAGS-CPP-2025/mpags-session-2-drbenmorgan.git (fetch)
 origin  git@github.com:MPAGS-CPP-2025/mpags-session-2-drbenmorgan.git (push)
[macbook]$
```

Notes

You can also find the URL from the gihub interface for your repository. Go to 'Code' and select 'SSH'. The URL can be copied from there.

26: Pushing your Repository to Github

We're now ready to to send our changes to github. To do this, we use **`git push`**. This can be supplied with the name of the remote to push to ('origin' usually), and the "refspec" or branch we want to share. In our case though, because we've changed the remote, we first need to set the upstream (origin and the current branch - master):

```
$ git push -u origin master
```

```
[[macbook]$ git remote -v
origin  git@github.com:MPAGS-CPP-2025/mpags-session-2-drbenmorgan.git (fetch)
origin  git@github.com:MPAGS-CPP-2025/mpags-session-2-drbenmorgan.git (push)
[[macbook]$ git push
fatal: The current branch master has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin master

To have this happen automatically for branches without a tracking
upstream, see 'push.autoSetupRemote' in 'git help config'.

[[macbook]$ git push -u origin master
[Enter passphrase for key '/Users/benmorgan/.ssh/github':
Enumerating objects: 15, done.
Counting objects: 100% (15/15), done.
Delta compression using up to 8 threads
Compressing objects: 100% (12/12), done.
Writing objects: 100% (13/13), 1.29 KiB | 1.29 MiB/s, done.
Total 13 (delta 7), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (7/7), completed with 1 local object.
To github.com:MPAGS-CPP-2025/mpags-session-2-drbenmorgan.git
   e186995..529b18e  master -> master
branch 'master' set up to track 'origin/master'.
[macbook]$
```

Notes

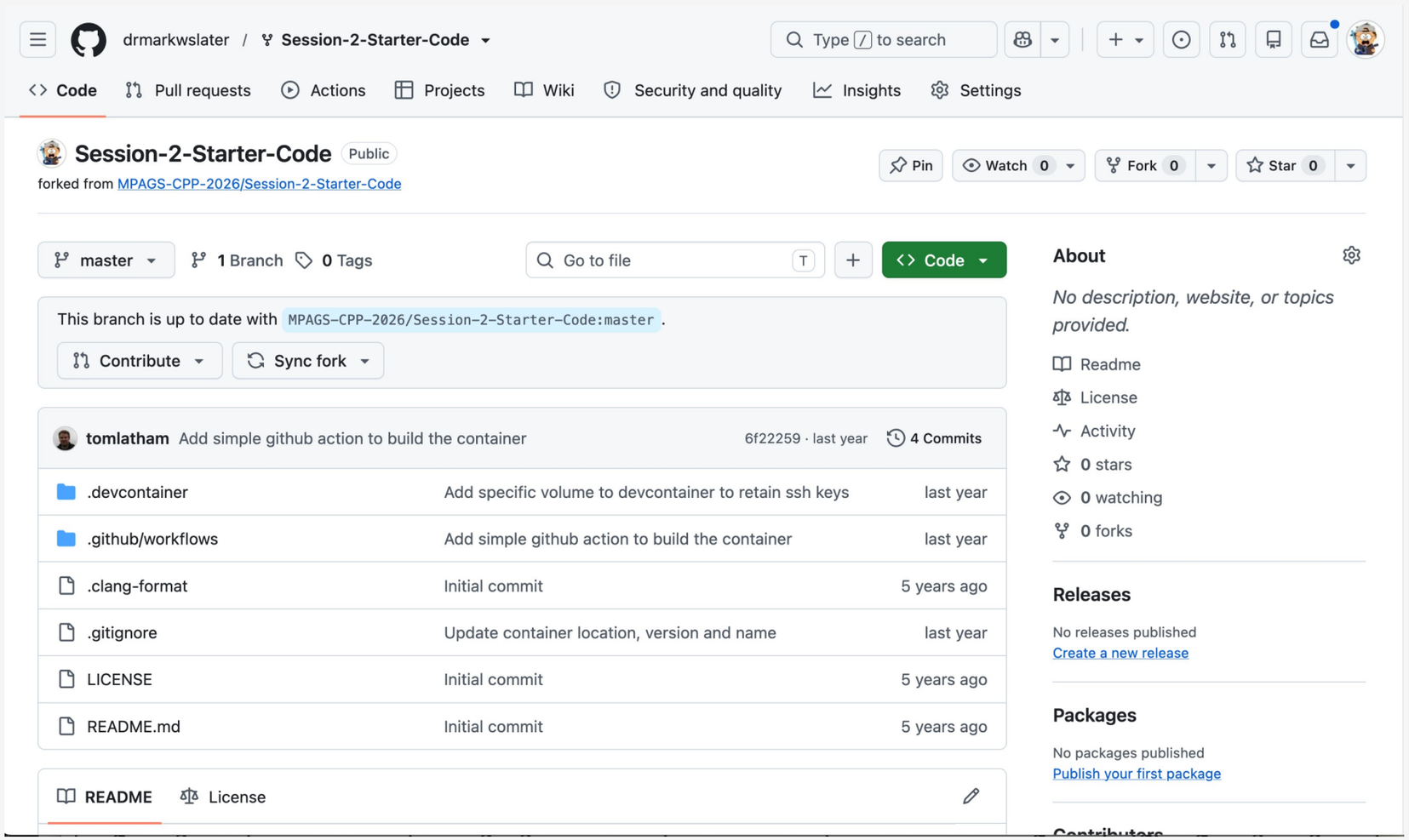
We won't cover Branching, but they can be thought of as separate sequences of commits. They're used to partition development, such as implementing new functionality, without interfering with others.

After this, you should be able to do plain **`git push`**

27: Viewing Changes on GitHub

After running `git push`, go back to your browser and refresh the page for your repository. You should now see that it's updated with the commits you've made up to the point you pushed.

It provides a very nice interface for browsing changes, so explore the viewing options and see how these map to the git command line arguments.



Notes

Now we can see the advantage of using Markdown format, as Github has rendered it nicely for us.

We could have used other markup styles as well.

28: Pushing Tags

Like any other repository "refspec", tags can be pushed to a remote repository. However, **`push`** does not **`push`** tags by default (you can see this as Github does not list your tag yet under "releases"). To do this, we have to either specify the tag name or use the **`--tags`** argument:

```
$ git push origin v0.1.0
```

```
[[macbook]$ git tag
 v0.1.0
[[macbook]$ git push origin v0.1.0
[Enter passphrase for key '/Users/benmorgan/.ssh/github':
 Enumerating objects: 1, done.
 Counting objects: 100% (1/1), done.
 Writing objects: 100% (1/1), 188 bytes | 188.00 KiB/s, done.
 Total 1 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To github.com:MPAGS-CPP-2025/mpags-session-2-drbenmorgan.git
 * [new tag]         v0.1.0 -> v0.1.0
[macbook]$
```

Notes

You should always push tags so they appear when others pull from your remote repo (including you!).

If you look on your github repository, you should see a '1' next to the "releases". Clicking on this will take you an interface where you can download a source archive for your code at the tag!

29a: Setting up to Pull Changes from Github

At present we're only working with a single copy of our GitHub repo. Later you may obtain other copies, e.g. on your Laptop or another Location, so commits will get pushed to Github that other copies don't yet have. To update the current copy of the repo, we'll use git pull, but first we need to configure how it'll integrate these changes:

```
$ git config --global pull.ff only
```

```
[[macbook]$ git pull
[Enter passphrase for key '/Users/benmorgan/.ssh/github':
From github.com:MPAGS-CPP-2025/mpags-session-2-drbenmorgan
 * [new branch]      feedback   -> origin/feedback
Already up to date.
[macbook]$
```

Why?

Note that git pull is two steps: i) Fetch changes, ii) reconcile changes.

By requiring a "Fast Forward" merge, we're telling git to only integrate the changes if it can reconcile them without having to do a "merge". We won't go into the details of how it decides this, but by requiring fast-forwards, we retain a clearer picture of the commit history.

29b: Actually Pulling Changes from Github

At present we're only working with a single copy of our GitHub repo. Later you may obtain other copies, e.g. on your Laptop or another Location, so commits will get pushed to Github that other copies don't yet have. To update the current copy of the repo, we can use git pull:

```
$ git pull origin
```

```
[[macbook]$ git pull origin
[Enter passphrase for key '/Users/benmorgan/.ssh/github':
 remote: Enumerating objects: 5, done.
 remote: Counting objects: 100% (5/5), done.
 remote: Compressing objects: 100% (3/3), done.
 remote: Total 3 (delta 2), reused 0 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (3/3), 1009 bytes | 201.00 KiB/s, done.
From github.com:MPAGS-CPP-2025/mpags-session-2-drbenmorgan
   529b18e..b96c257  master     -> origin/master
Updating 529b18e..b96c257
Fast-forward
 README.md | 3 +++
 1 file changed, 3 insertions(+)
[macbook]$
```

Try This

Using the web editor on github.com, make a change to your README file by adding a fresh line at the end. Return to your main copy, then `git pull` to get those changes. Git will report what changes have been made.

Note that git pull is two steps: i) Fetch changes, ii) Merge changes.

30a: Git Conflicts

Git is very smart at merging content changes in files, but it is not infallible - e.g. if a single word has changed on the same line, which one should be preferred? When conflicts occurs, git will warn us about them, and git status can be used to review them. **Make an edit to README.md on github and then alter the same line in a different way in your local copy of the repo**. After committing, pull the changes from the remote repository. `git pull` will say that we can't be fast-forwarded (per our settings).

```
[[macbook]$ git add README.md
[[macbook]$ git commit -m "Demonstrate conflicts"
 [master f445d60] Demonstrate conflicts
  1 file changed, 1 insertion(+), 1 deletion(-)
[[macbook]$ git status
 On branch master
 Your branch is ahead of 'origin/master' by 1 commit.
   (use "git push" to publish your local commits)

 nothing to commit, working tree clean
[[macbook]$ git pull
[Enter passphrase for key '/Users/benmorgan/.ssh/github':
 remote: Enumerating objects: 5, done.
 remote: Counting objects: 100% (5/5), done.
 remote: Compressing objects: 100% (3/3), done.
 remote: Total 3 (delta 2), reused 0 (delta 0), pack-reused 0 (from 0)
 Unpacking objects: 100% (3/3), 1009 bytes | 336.00 KiB/s, done.
 From github.com:MPAGS-CPP-2025/mpags-session-2-drbenmorgan
    b96c257..1c93819  master     -> origin/master
 hint: Diverging branches can't be fast-forwarded, you need to either:
 hint:
 hint:   git merge --no-ff
 hint:
 hint: or:
 hint:
 hint:   git rebase
 hint:
 hint: Disable this message with "git config set advice.diverging false"
 fatal: Not possible to fast-forward, aborting.
 [macbook]$
```

Notes

Conflicts are most common in collaborative development, but can also happen in your own work, so it's worth learning how to resolve them!

Here, git has fetched the changes from the upstream repo, it just can't do anything with them until we tell it what to do.

30b: Git Conflicts

Here we'll chose the first option, to merge without fast-forwarding, we already have the changes locally so we just run **`git merge --no-ff`** as suggested. **`git status`** will say there are conflicts that can't be automatically dealt with. We need to edit the file(s) to resolve the conflict, then add/commit just as we did for any other edit.

What has actually happened here is that Git will create a "merge commit", recording that we've brought two lines of development together once we've resolved the conflict. See the **Git Documentation for more details**

```
[[macbook]$ git status
On branch master
Your branch and 'origin/master' have diverged,
and have 1 and 1 different commits each, respectively.
  (use "git pull" if you want to integrate the remote branch with yours)

nothing to commit, working tree clean
[[macbook]$ git merge --no-ff
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
[[macbook]$ git status
On branch master
Your branch and 'origin/master' have diverged,
and have 1 and 1 different commits each, respectively.
  (use "git pull" if you want to integrate the remote branch with yours)

You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
        both modified:   README.md

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

Notes

We don't cover the rebase option here. You may be required to use this in certain projects, but the process of resolving conflicts is much the same.

NB: Even if Git can FF (or do a merge), still good to check your changes make sense in the context of the changes merged in from upstream. Git doesn't know C++!

31: Viewing Conflicts

Git marks conflicts in files using a special markup block showing the conflicting content

```
<<<<<<< HEAD
local content
=======
remote content
>>>>>>> refspec
```

The HEAD block shows our local content.

'=======' divides the sections, and after this is shown the conflicting content. This ends with the "refspec" of the commit causing the conflict

README.md > # <<<<<<< HEAD

```
1   # mpags-cipher
2   A simple command line tool for encrypting/decrypting text using
    classical ciphers
3
4   # Author
5   Mark Slater
6
7   # Documentation
8   Takes input and runs a variety of classical ciphers to produce output
9
10  # License
    Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
11  <<<<<<< HEAD (Current Change)
12  License info is included in the contained LICENSE file
13  =======
14  See included LICENSE file in repo
15  >>>>>>> 977e8fc5a21d582f65f7e262a388bff00bb4c646 (Incoming Change)
16
```

Notes

There may be more than one conflict block in a file!

This is where having a good syntax aware text editor helps. Add-ons are usually available specifically to highlight and handle git syntax like the conflict block. Visual Studio Code highlight them by default.

32: Resolving Conflicts

The content of the conflict has to be resolved manually, and is up to you (it may be a simple merge, choosing one or the other, or more complex). Once you've done this, remove the git markup (`<<<<<<< HEAD`, `=======` and `>>>>>>> refspec`) and save the file. Using `git status` will still show it as unmerged, but we can now use `git add` to stage it, followed by `git commit` to commit it and all the other changes brought in. Finally, push the changes to your repository!

```
[[macbook]$ git add README.md                                                  ]
[[macbook]$ git commit -m "Resolved conflict"                                  ]
 [master aaaba86] Resolved conflict
[[macbook]$ git commit                                                         ]
 On branch master
 Your branch is ahead of 'origin/master' by 2 commits.
   (use "git push" to publish your local commits)

 nothing to commit, working tree clean
 [macbook]$
```

Notes

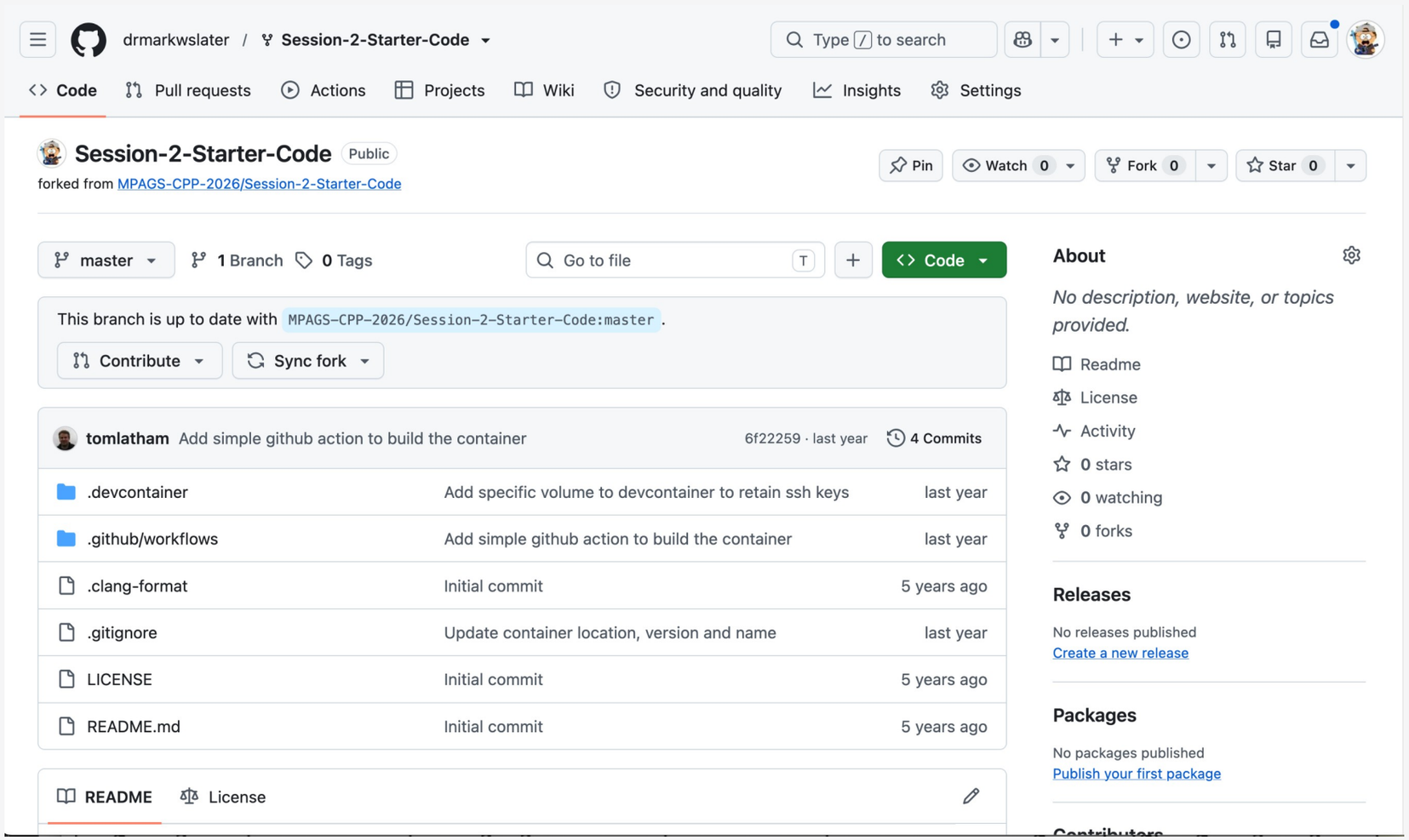
Don't be frightened of conflicts, git provides all the tools to help you resolve them!

Though you are unlikely to encounter conflicts in this course as you will be using your own repo, it's good to know how to deal with them.

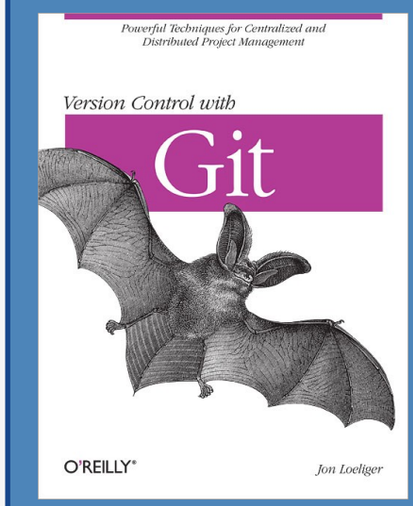
33: And we're done

That about covers the basic usage of git and github. All of the techniques are applicable to other VCSs you may work with, of particular importance being **the writing of good commit messages** so you (and your collaborators) know not only what changes were done, but why!

Through the course, remember to commit your work regularly when you have got something working (NEVER commit code that doesn't work for you!). Push to GitHub regularly. Use tags to mark feature/task completion, again, the tag should work!



Don't Forget Resources



git

github
SOCIAL CODING

Code Reviews

- Through forking our initial repos, you will get a repository for each session.
- **Once you're happy with your code from the end of each session, do the following:**
  - **Go to the front page of your repository and select 'Contribute' and 'Open Pull Request'**
  - **You can add more info or comments in the description here**
  - **Click 'Create Pull Request' to create the PR and email us**
  - **we can review the code changes you've made to the base repo, leaving comments and suggesting changes, as would be done in a real code review**

