NW-PaGe
Northwest Pathogen Genomics Center of Excellence

Northwest Pathogen Genomics Center of Excellence

Policies, standards, and guidelines

2024-05-02

Frank Aragona
Washington State Department of Health
2024
Data Integration/Quality Assurance

Table des matières

Frank Aragona
Washington Department of Health, Data Integration/Quality Assurance
frank.aragona@doh.wa.gov

1 Introduction

This document details the policies and guidelines for the Northwest Pathogen Genomics Center of Excellence (NW-PaGe) Github Organization.

For more information, tutorials and code examples, please see the policies website here https://nw-page.github.io/standards/.

2 Security

Objectives

- Prevent sensitive information leaks to Github
- Set up guardrails, `.gitignore` , hooks
- Scrub private repos before they go public

If sensitive information is leaked and commited to the remote repo, then they will stay in the git history (and will require a lot of effort to remove them from the history). The following cannot be included in any repo **or any local commit!**:

Type	Examples
File Paths	• Network drives • Shared internal drives
Server Names	• ODBC Connections
Credentials	• SSH Keys • Tokens (REDCap, Azure, Github, etc) • Usernames • Passwords • Blob/bucket keys
Identifiable Information	• Addresses • Names • Any PHI

Prevent Credential Leaks with Env Variables

There are a number of ways to do this. We typically use a yaml file that can be filled out with personal credentials locally. The file will not be committed to the remote repo

2.1.a Create a private credentials file

The scripts use a `.yml` file that contains a list of API tokens, server names, and usernames/passwords specific to each individual user. There are two `.yml` files. One is a template (containing no actual passwords..) that exists in the repo and serves as a template so every individual user can keep up to date with new credential additions. The other is the individual `creds.yml` that is in the repo's `.gitignore`. This file will never exist in the repo and only exist locally (in the user's C drive).

2.1.b `creds.yml` details

The `.yml` file can work with multiple programming languages including R and Python. They are read in the same way and can be easily adjusted when adding new passwords or using them as configuration files.

They look like this:

```
# Default is needed to distinguish values.
# Leave a blank line (NO SPACES) as the last line in this file or
things will break
# Quotes aren't necessary, but can be used.
default:
  conn_list_wdrs:
    Driver: "SQL Server Native Client 11.0"
    Server:
    Database:
    Trusted_connection:
    ApplicationIntent:

  fulgent:
    username: <USERNAME>
    password: <PASSWORD>
```

You can have different variables assigned to unique lists, which allows for easy configuration. For example, the list starting with `default` has variables `conn_list_wdrs` and `fulgent`. You can have a different list of variables within the same file like this:

```
# Default is needed to distinguish values.
# Leave a blank line (NO SPACES) as the last line in this file or
things will break
# Quotes aren't necessary, but can be used.
default:
```

```
  conn_list_wdrs:
    Driver: "SQL Server Native Client 11.0"
    Server:
    Database:
    Trusted_connection:
    ApplicationIntent:

  fulgent:
    username: <USERNAME>
    password: <PASSWORD>

test:
  conn_list_wdrs:
    Driver: "SQL Server Native Client 11.0"
    Server:
    Database:
    Trusted_connection:
    ApplicationIntent:
```

Now there is a `test` list with its own variables. This lets us switch a set of variables within our scripts. `default` applies to the main credentials where `test` can distinguish which variables should be test or dev scripts specific. Notice below that you can now call the credentials from a `.yml` file into an R or Python script and the actual credentials will never exist in the code pushed to the repo.

```
# this script is in the repo, but credentials are hidden
library(yaml)

# read in the local credentials yaml file
creds <- yaml::read_yaml("path/to/local-credentials.yml")

# pull in the credentials
server_name <- creds$default$conn_list_wdrs$server
```

We can even get more specific and add an `if-else` statement to specify which credential we want to select. This can be helpful if we have a CI/CD pipeline and have a script automatically run on a task scheduler or cron job. We can call the credentials we want in the command line and have the command line code run in my task scheduler. That way we can use multiple different versions of the same script and have all of it be automated.

For example, the middle panel uses the `commandArgs()` to pull any arguments passed to the script in a shell/command line script. In the right panel, the shell script has `production` and `test` as second arguments. These are passed to the R script as `arg[2]` . Now we can use `arg[2]` in the if-else statement to conditionally select credentials and do it automatically.

```
args <- commandArgs(TRUE)

# this script is in the repo, but credentials are hidden
library(yaml)

# read in the local credentials yaml file
creds <- yaml::read_yaml("path/to/local-credentials.yml")

# pull in the credentials
if(args[2] == "production"){
  server_name <- creds$default$conn_list_wdrs$server
} else if(args[2] == "test"){
  server_name <- creds$test$conn_list_wdrs$server
}
```

```
# Run the production code
$ Rscript -e "source('path/script_in_repo.R');" production

# Run the test/dev code
$ Rscript -e "source('path/script_in_repo.R');" test
```

2.1.c Safe Guards - Prevent Accidental Leaks!

Once you have the credentials.yml template in your repo, make sure that nobody on your team (or anyone with write access..) is able to accidentally push changes to the template. We don't want someone's passwords or API tokens to exist in GitHub.

This link shows how to skip any changes made to the specific file https://stackoverflow.com/a/39776107. If someone makes local changes to the template, the changes will not show in their commit. It is a safe guard.

2.1.c.a For all individual users, run this code:

```
git update-index --skip-worktree creds_TEMPLATE.yml
```

This will tell your local git to ignore any changes made to `creds_TEMPLATE.yml` , but also allow it to exist in the repo (since `.gitignore` will prevent it from being in the repo)

2.1.c.b If you need to update the template file run this:

```
git update-index --no-skip-worktree creds_TEMPLATE.yml
```

This will allow changes to the template. **So when you need to update the template, use this code**

And to get a list of files that are "skipped", use this code:

```
git ls-files -v . | grep ^S
```

3 Security Guardrails

Using a `.gitignore` file for environmental variables/credentials is an excellent guardrail and promotes good coding habits, but we may also want additional guardrails such as hooks.

Hooks are processes that run in the background and can prevent code from being pushed if there is a security flaw. There are two hooks we could use for security; pre-commit hooks and pre-receive hooks

3.1 Pre-commit Hooks

Pre-commit hooks run a process locally when the user attempts to commit code to a git branch. Hooks have many uses. Here we can use them as a security guardrail to prevent accidental credential leaks in committed code. For example, if someone accidentally pushes a server name to the public repo, the hook will prevent that code from ever getting into the remote repo and will give the user a local error.

3.1.a Windows

1. Clone or download the zip from the AWS Git Secrets repo
2. Extract zip or `cd` to the repo
3. Open folder and right click install.ps1.
    a. Run in Power Shell
    b. Type Y to give permission

Alternatively, in the powershel terminal you can change directories `cd` to the repo and `.\install.ps1`

4. CD `cd` to a directory where you have the git repository you want to upload, either in PowerShell or R studio terminal

```
PS > cd path/to/repo/root
```

5. Run git secrets –install

```
git secrets --install
```

6. Make or copy the regex file called `secrets_key` containing the secret patterns into your folder.
    - This file should be given to you by your supervisor. It contains a regex of potential secrets
7. Make sure the file `secrets_key` is in your `.gitignore` . We can't push that to the remote repo.
8. Run `git secrets --add-provider -- cat ./secrets_key`

```
git secrets --add-provider -- cat ./secrets_key
```

You can also add prohibited patterns like this

```
# add a pattern
git secrets --add '[A-Z0-9]{20}'

# add a literal string, the + is escaped
git secrets --add --literal 'foo+bar'

# add an allowed pattern
git secrets --add -a 'allowed pattern'
```

9. Test Git history by running

```
git secrets --scan-history
```

10. If something gets flagged and you don't care about your history anymore: Delete .git folder and reinitialize repository

- I would take caution about this point. There might be better ways to clean your git history if you don't want to get rid of everything.

11. Test on one of my projects to see if rebasing is a sustainable option
12. Make repo public
13. Will automatically scan on every commit and won't let it commit unless it's clean - Create a few files to show it working

> **Note**
>
> We can't use the "Non capture group" feature of regex. Meaning we can't use patterns like this in our regex: (?:abc) – see https://regexr.com IMPORTANT: Tab separate your regex expressions. Making new lines caused a bit of chaos and took really long to figure out. (you can use multiple tabs to separate them more visually)

3.1.b WSL/Linux

1. Clone the AWS Git Secrets repo
2. In the terminal, `cd` to the repo
3. Install the command:

```
sudo make install
```

4. `cd` to a directory where you have the git repository you want to push to

```
cd path/to/repo/root
```

5. Run git secrets –install

```
git secrets --install
```

6. Make or copy the regex file called `secrets_key` containing the secret patterns into your folder.
    - This file should be given to you by your supervisor. It contains a regex of potential secrets
7. Make sure the file `secrets_key` is in your `.gitignore`. We can't push that to the remote repo.

8. Run `git secrets --add-provider -- cat ./secrets_key`

```
git secrets --add-provider -- cat ./secrets_key
```

You can also add prohibited patterns like this

```
# add a pattern
git secrets --add '[A-Z0-9]{20}'

# add a literal string, the + is escaped
git secrets --add --literal 'foo+bar'

# add an allowed pattern
git secrets --add -a 'allowed pattern'
```

9. Test Git history by running

```
git secrets --scan-history
```

10. If something gets flagged and you don't care about your history anymore: Delete .git folder and reinitialize repository
    - I would take caution about this point. There might be better ways to clean your git history if you don't want to get rid of everything.
11. Test on one of my projects to see if rebasing is a sustainable option
12. Make repo public
13. Will automatically scan on every commit and won't let it commit unless it's clean - Create a few files to show it working

> **Note**
>
> We can't use the "Non capture group" feature of regex. Meaning we can't use patterns like this in our regex: (?:abc) – see https://regexr.com IMPORTANT: Tab separate your regex expressions. Making new lines caused a bit of chaos and took really long to figure out. (you can use multiple tabs to separate them more visually)

NOTE!!

- The REGEX strings used in the `secrets_key` file may be decieving
- Make sure to test that the regex flags what you want it to

- `git secrets --scan-history` may take a very long time to run

1. Check that the `secrets_key` regex is working by running the process on a repo that you know has secrets in it. For example, in a different folder, run all the pre-commit hook steps above and add a known "bad" string into the regex. For example, in the regex put `bad_string` and in a file in that folder put `bad_string` . When you scan it should get flagged.
2. If secret scanning is taking too long, you might want to check certain files first. I've found that HTML files take a *very* long time to scan for secrets. If you're using linux or macOS you should be able to scan certain files via grob like this `git secrets --scan path/to/files*` . If you're using windows it probably won't work. In that case, if you only want certain file extensions to be scanned, use this function in powershell below. Download the file here:

https://github.com/NW-PaGe/standards/blob/main/gh/secret_scanner.ps1

```
# Example Usage
# write this in the powershell terminal, adjust for the file type(s)
you want to scan - can be multiple types: $fileExtensions = @(".R",
".py")
# then execute this in the terminal: ScanFiles -FileExtensions
$fileExtensions

# It will give you an output of any secrets that are contained in
those files

Function ScanFiles{
  param (
      [string]$filePath = (Get-Location).Path,
      [string[]]$fileExtensions
)
  Get-ChildItem $filePath -recurse | Where-Object {$_.extension -
in $fileExtensions} |
  Foreach-Object {

      git secrets --scan $_.FullName

  }
}
```

Now, you can scan your secrets by copying and pasting the code into a powershell terminal like this:

```
$fileExtensions = @(".R",".py",".Rmd",".qmd")
ScanFiles -FileExtensions $fileExtensions
```

3.2 Pre-Recieve Hooks

These are still being investigated. They are remote hooks (not local like pre-commit hooks) that can be deployed throughout the Github organization. They can block certain commits from ever being pushed to the remote repo. They may make things unnecessarily complicated

3.3 Pushing Private Code to Public Repos

We may wish to take private codes and push them to a public repo. We need to make sure that the public code doesn't not contain sensitive or forbidden data/code, so cleaning up the private repo is important before pushing.

There are a few ways to do this, but the easiest way is to copy the clean private code to the public repo, that is, copy all the files you want to add publicly but **do not copy the `.git` folder**. If the private repo has a dirty git history we will not want that history in the public repo because the sensitive data will then be publicly available.

The private repository on the left still contains sensitive information in the git history. The public repository on the right has a clean git history because we copied only the current clean files from the private repo and did not attach its git history (which lives in the hidden `.git` folder)

4 Code Reviewers/Github Operations Team

With the guardrails above in place there should be few chances that credentials get pushed to a repo. However accidents may still happen. We want to make sure that anyone who opens up a repo in the Github organization adheres to the rules, has the proper credential/coding set-up, and installs their local pre-commit hooks properly.

It may be useful to have a team within the organization that helps with repo set-up. The team would help avoid a scenario where a person opens up a repo without reading this documentation and understanding the rules (and thus potentially breaking security rules).

This Github Operations Team could also be helpful in managing permissions for members in the organization. See the video below on how the company Qualcomm manages their Github organization https://www.youtube.com/embed/1T4HAPBFbb0?si=YRsUYXIxLPhdr41T and how they use a Github Operations Team to guide new members access/repo development

https://www.youtube.com/embed/1T4HAPBFbb0?si=YRsUYXIxLPhdr41T

5 Licensing

Summary

- Licenses prevent code theft and inappropriate redistribution of code.
- Review common open-source licenses
- License types vary depending on repo goals

6 General License Info

Below is a list of common open-source licenses.

There isn't a one size fits all license, so thankfully there are a variety of options. Here are two common ones:

7 GNU GPL licenses

a. These are the strong licenses
b. Prevents someone from taking our code and privatizing it (and making money off of it)
c. Someone can still use our code, they just need to ensure that what they're doing with it is open-source
d. "Copyright and license notices must be preserved."
e. "Contributors provide an express grant of patent rights. When a modified version is used to provide a service over a network, the complete source code of the modified version must be made available."

8 MIT license

a. I think this is the most commonly used one
b. "short and simple permissive license… only requiring preservation of copyright and license notices"

c. "Licensed works, modifications, and larger works may be distributed under different terms and without source code."
d. Someone could basically do whatever they want with the code.
e. Nextstain/ncov repo is currently using this

And here are a couple of youtube videos that were helping in explaining licensing

https://www.youtube.com/embed/rbQg9DY_4y0?si=OvU9vLBHX43dTIcA

https://www.youtube.com/embed/ndORMSnb2nw?si=tkUzjwZYWKfrLTEU

9 Policies

Objectives

- Ensure that all repos in the org have the required documents
- Set policy rules at the Organization level
- Repos need to have reproducible code
- Repos need to have documentation

In the Github Organization we may require all repositories to contain certain documents. For example, we want to make sure that every repo has a `CODE OF CONDUCT` document that is a general policy applied throughout the organization.

Here's a list of required documents:

`README`

`README` files are instructions or documentation on how to use your software. It should give a quick introduction to the repo and instructions on how to install or run the code.

`CODE_OF_CONDUCT`

A Code of Conduct can let a user know what the rules of the organization are and how any wrongful behavior will be addressed. The document will provide the "standards for how to engage in a community"

`CONTRIBUTING.md`

This file should appear in the issue tab in a repo. It lets a user know how they can contribute to the project and if they need to sign any forms before contributing. Some larger organizations require that a person knows what they are contribut-

ing to and they must sign a form acknowledging that any software/code contributions to the project will be used and cannot be retracted by the user. The code submitted may also be used to develop processes but the organization will not pay the individual contributor (since this is open-source, we only look for open-source contributions)

`LICENSE`

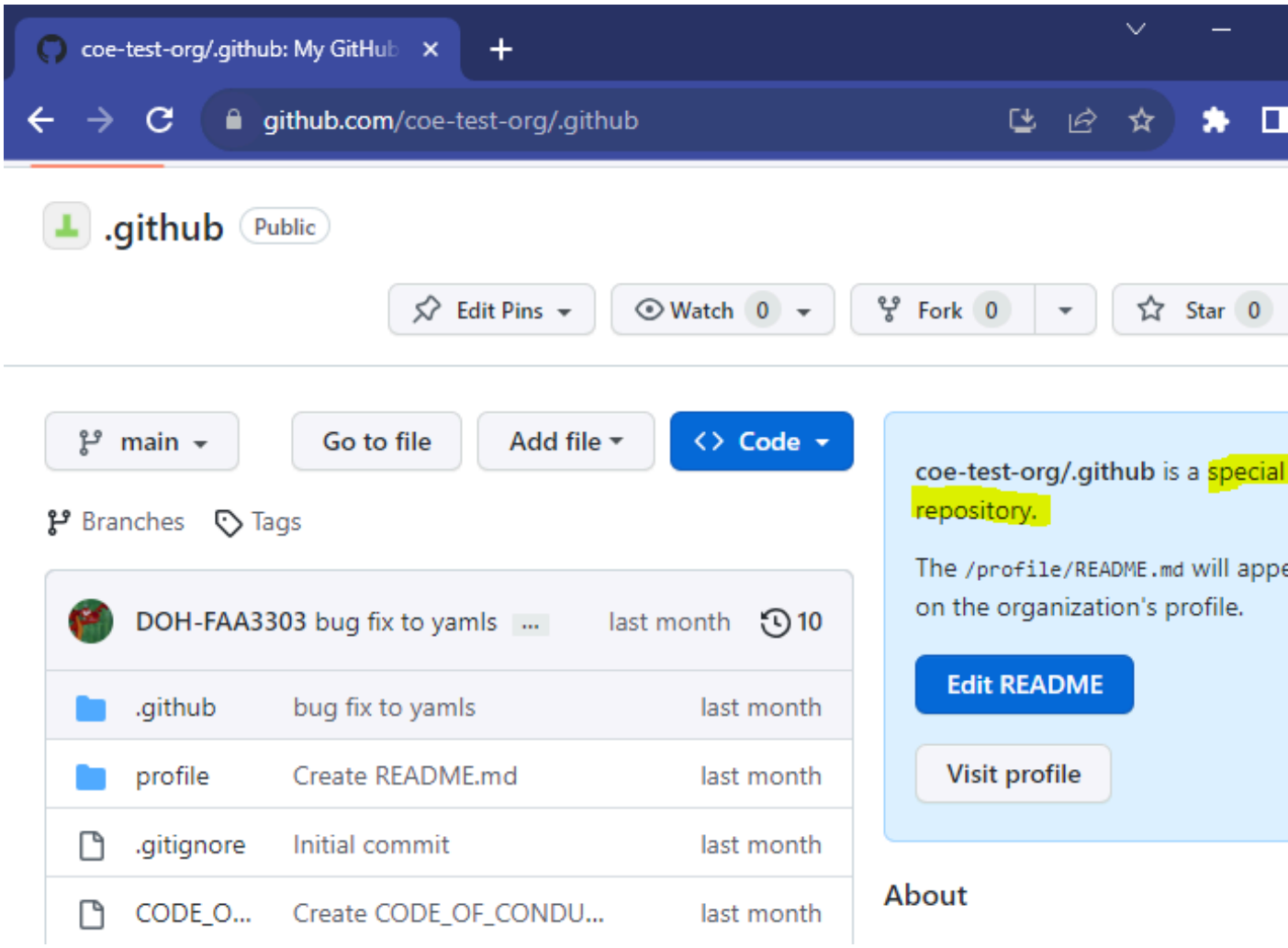
These should be and are set at the repo level. There will be many different licenses to choose from that will depend on the specific repo. More on that here.

10 Set Policy Rules at Org Level

Policy rules may include requiring certain documents in each repo or requiring that a person sign every commit.

10.1 Document Requirements with `.github` Repos

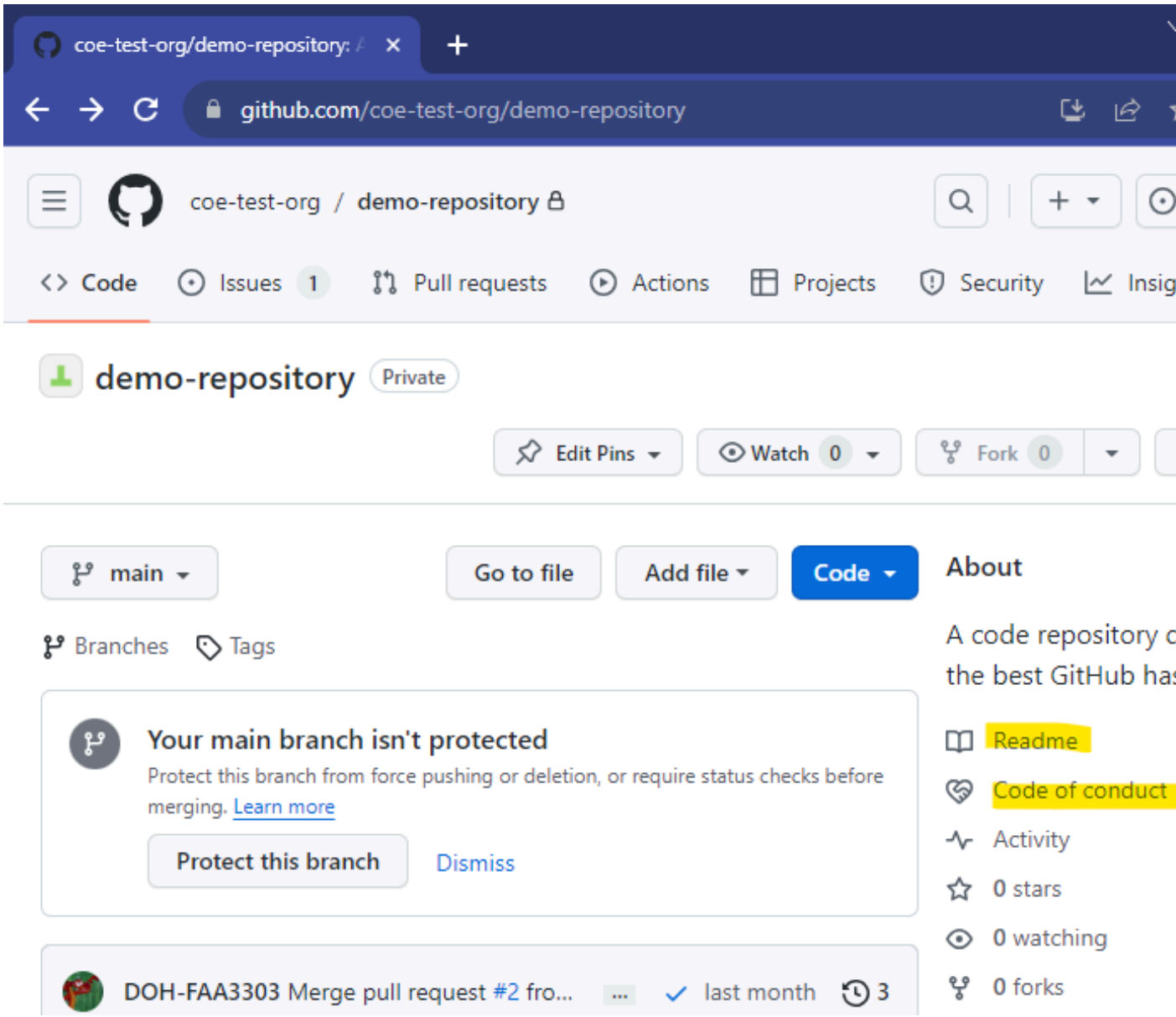
You can set most policy rules and create documents for each repo at the organization level by using a special `.github` repo. Dot files and dot folders have special functionality in some software. For Github, the `.github` folder defines workflows for things like Github Actions in a repo. A `.github` *repository* on the other hand defines *organization level* rules and templates.



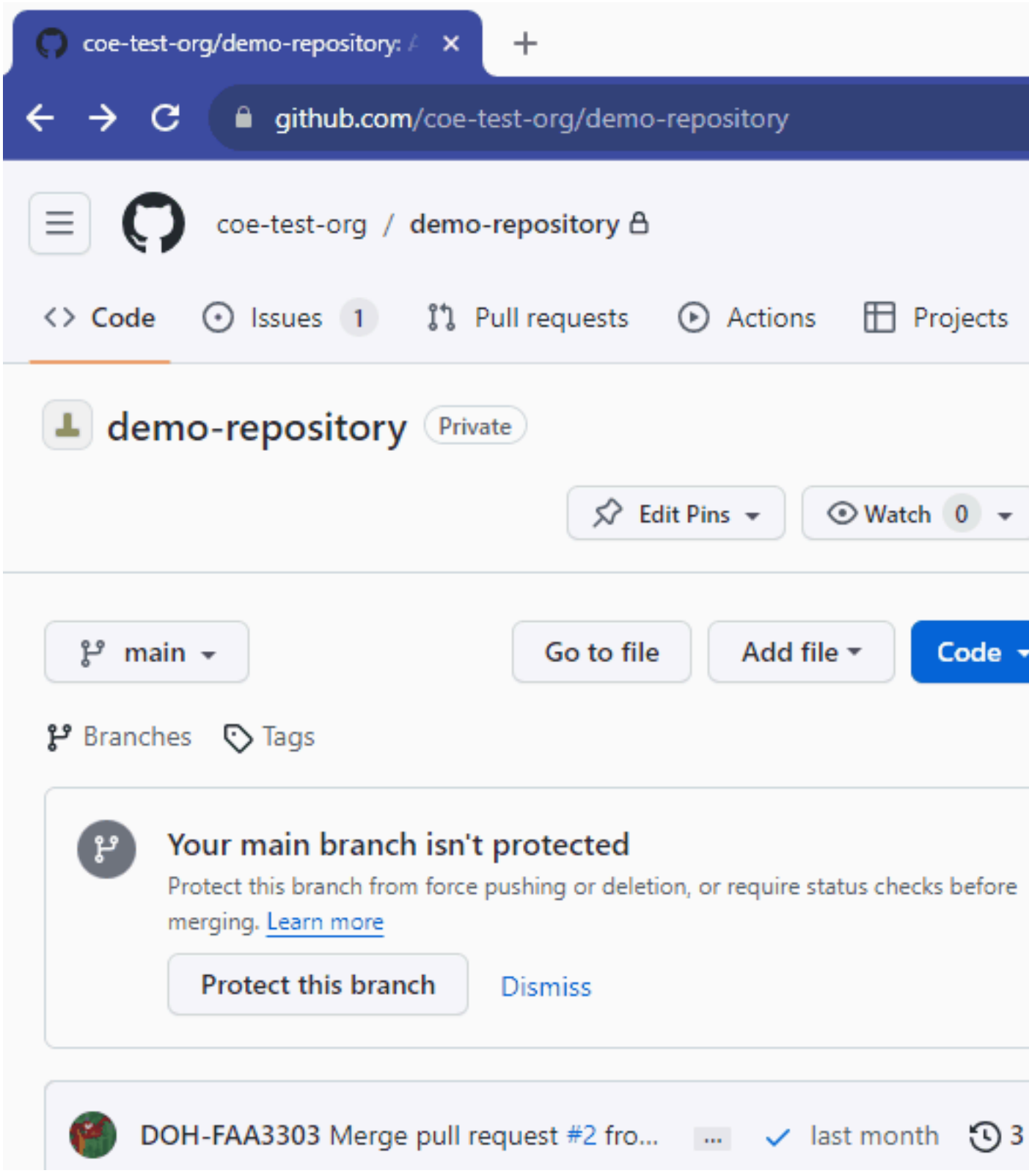
In order to write and set these policies at the organization level we can put them at the root of the `.github` repository and edit them there.

```
$ tree /f
C:.
|   .gitignore
|   CODE_OF_CONDUCT.md
|   CONTRIBUTING.md
|   LICENSE
|   README.md
|
├───.github
└───profile
        README.md
```

Take a look above. I have the required documents/policies at the root of the `.github` repo directory. Now if I open up any given repo in the organization I will find a link to those files:



If you click on the `CODE_OF_CONDUCT` link it will take you right to the `.github` repo and open the `CODE_OF_CONDUCT.md` file there:



Now you can set organization level policies from the `.github` repo and they will automatically populate in *all* existing and new repositories *unless there are repo specific policies in place*. If a repo already has its own policies they will not be overwritten.

11 Set Templates at the Org Level

Aside from policy documents, you can make templates at the organization level. Two commonly used templates are issue templates and discussion templates.

In the public repos there may be end users that may have limited experience using Github. If they want to submit an issue or ask a question they get lost. Templates can help them form a question or idea. Templates can also help standardize how issues and discussions are maintained throughout the organization.

Structuring the format of issues and discussions can make the author and the end-user's lives easier.

In the `.github` repo I made a *folder* called `.github`. This is a special folder that can hold Github Action workflows and more, as mentioned above.

In the `.github` *folder* I have a folder called `DISCUSSION_TEMPLATE` and another called `ISSUE_TEMPLATE`. These are special folders that Github recognizes as discussion and issue folders that will set templates at the repo (or in this case the org) level.

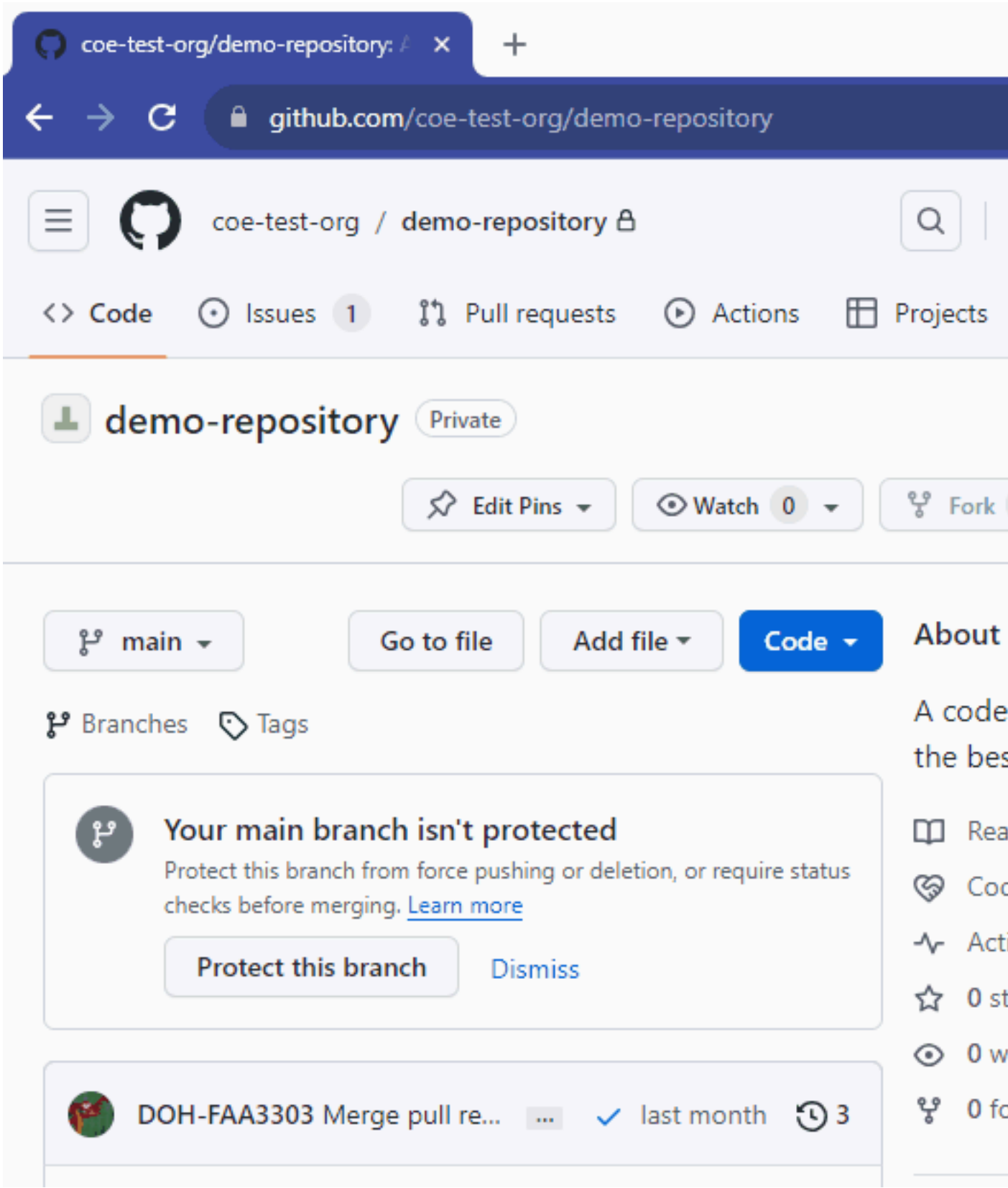
```
$ tree /f
C:.
│   .gitignore
│   CODE_OF_CONDUCT.md
│   CONTRIBUTING.md
│   LICENSE
│   README.md
│
├───.github
│   │   pull_request_template.md
│   │
│   ├───DISCUSSION_TEMPLATE
│   │       feature-requests.yml
│   │       q-a.yml
│   │       show-and-tell.yml
│   │
│   └───ISSUE_TEMPLATE
│           bug_report.yml
│           config.yml
│           feature_request.yml
│
└───profile
        README.md
```

Each Folder has `.yml` files in it that are basically Github instructions on how to format issues and discussions.

For example, in the `ISSUE_TEMPLATE` folder I have a `.yml` file called `bug_report.yml`. This file contains the structure for how someone can report a bug.

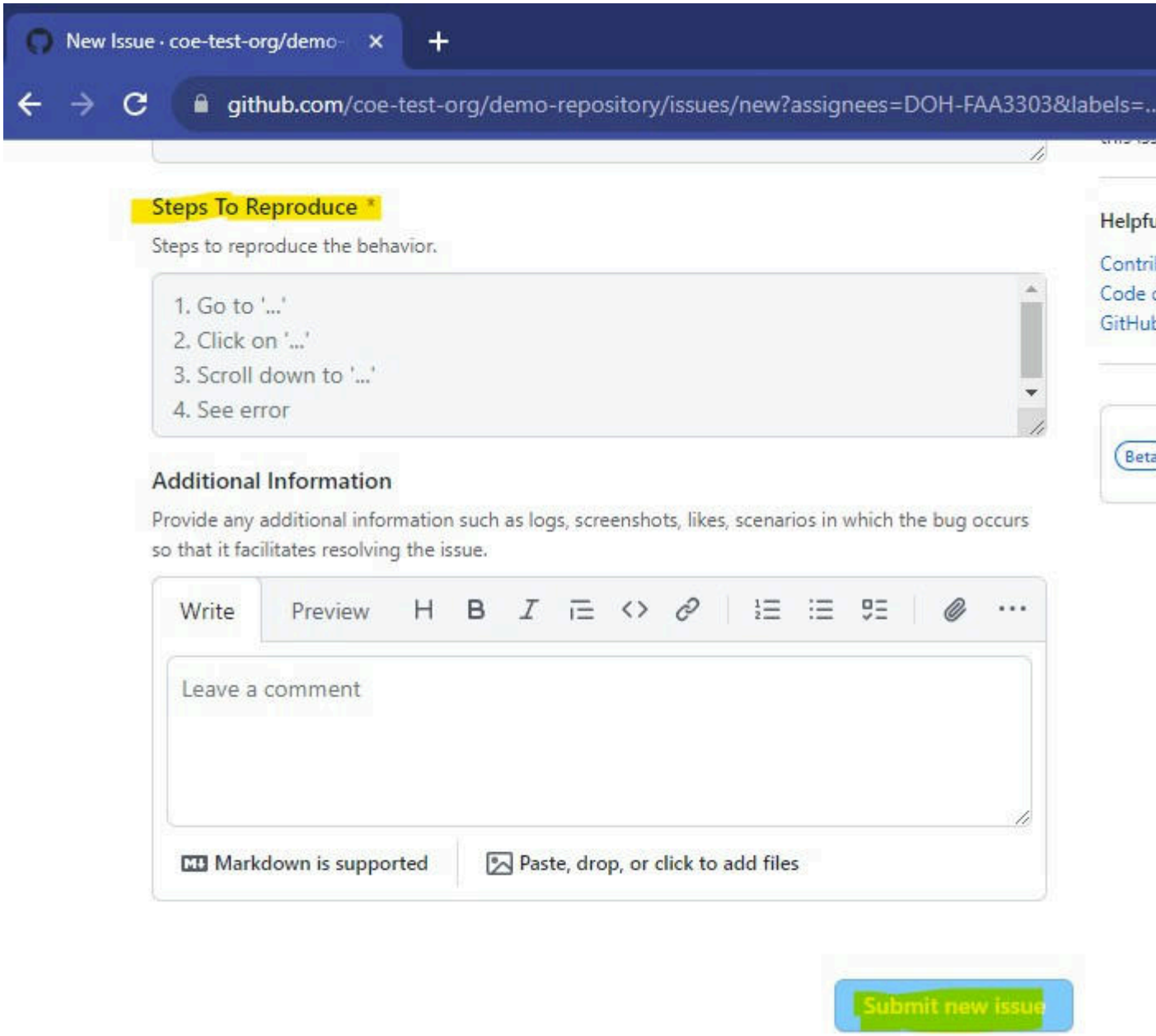
```yaml
name: Bug Report
description: File a bug report here
title: "[BUG]: "
labels: ["bug"]
assignees: ["DOH-FAA3303"]
body:
  type: markdown
    attributes:
      value: |
        Thanks for taking the time to fill out this bug report
       Make sure there aren't any open/closed issues for this topic
```

Now, when someone clicks on the `Issues` tab in a repo in this organization they will be met with the `Bug Report` template:



Notice that in the template you can create text areas and pre-fill those areas with suggestions. You can even require that someone fills out those areas before they can submit the issue:

```yaml
- type: textarea
    id: steps-to-reproduce
    attributes:
      label: Steps To Reproduce
      description: Steps to reproduce the behavior.
      placeholder: |
        1. Go to '...'
        2. Click on '...'
        3. Scroll down to '...'
        4. See error
    validations:
      required: true
```



11.1 Commit Sign-Off Requirement - Github Apps

We may want to require authors or reviewers to sign-off on commits to a repo. This is sometimes established in projects to "ensure that copyrighted code not released under an appropriate free software (open source) license is not included in the kernel."

You can install a Github App in the organization and it will be applied to all repos. The DCO App (Developer Certificate of Origin) is popular and lightweight. To install it in the organization, click on Configure and it will give you the option to configure it with the organization of choice.

12 IaC

Infrastructure as Code (IaC) can be helpful when managing administration tasks or writing hooks at the org level.

13 Reproducibility

Objectives

- Data and Code Democratization
- Github Codespaces
- Package reproducibility with virtual environments
- Github Releases
- Documentation

14 Data and Code Democratization

Data and code in our repositories need to be accessible to end users and developers. There should be no bottlenecks or difficulties with installing software, executing code, finding documentation, and using test datasets.

The goal is for any user to run code without needing to install anything on their personal machine and run your code with minimal set up. This may not be possible in every scenario, but there are tools available in Github to make this possible for the majority of our repos.

15 Github Codespaces

Github Codespaces are virtual machines (VMs) owned by Github that are connected to each repository. They let a user open the repo in a browser IDE (Inte-

grated Development Environment) and execute the code in that environment. There is no set up or installation necessary for them.

The VMs are free for up to 60 hours a month of use and there are more hours added for Github users with paid memberships. 60 hours/month should be plenty for our purposes. Users are responsible for their own Codespace, so if they go over the limit they will be responsible for adding more hours and paying for the service.

15.1 Open a Codespace

At the root of the repo, click on the **Code** drop down button

1. On the right there is a tab called Codespaces.
2. Click the + sign and a Codespace will launch

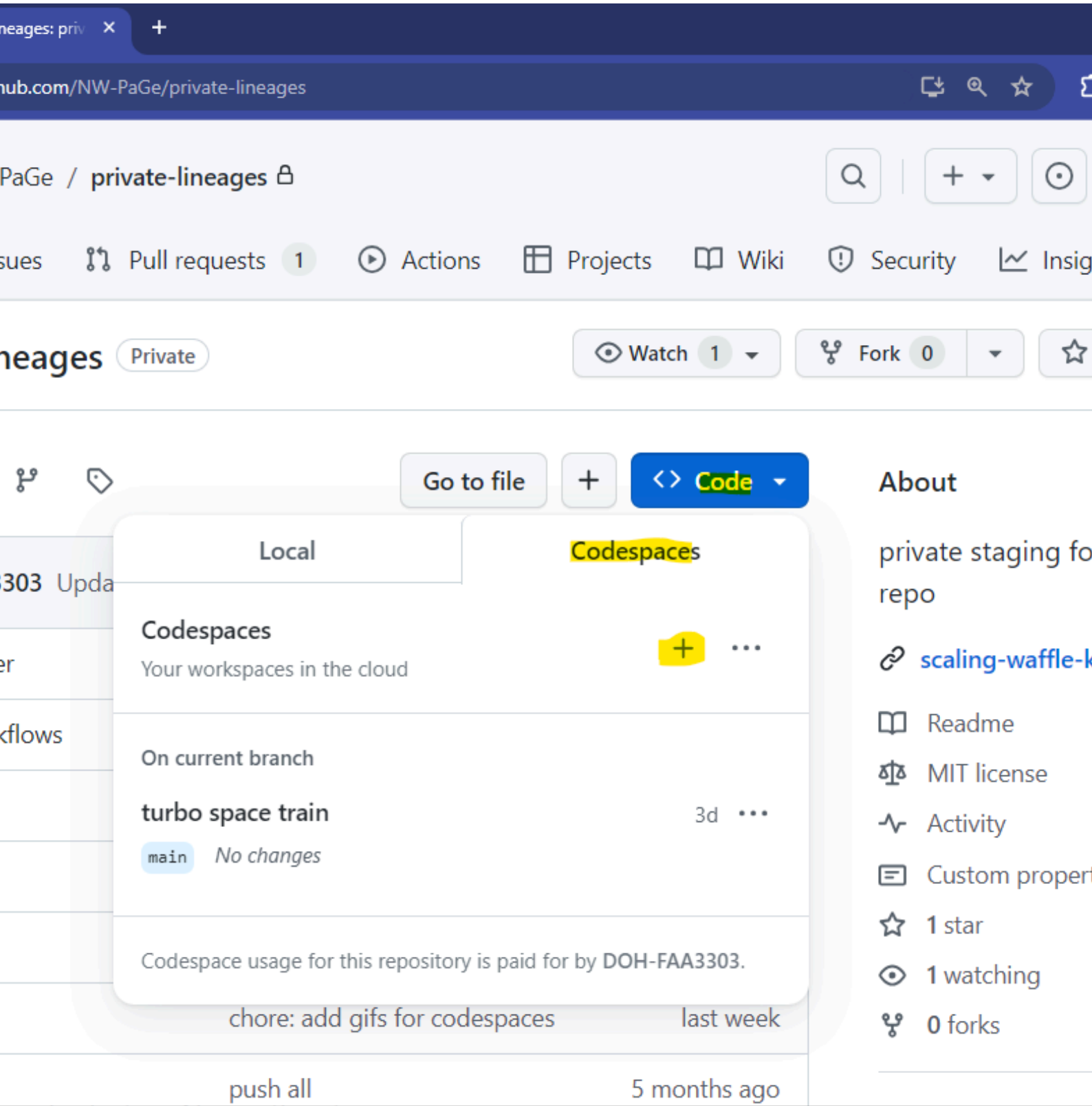


Figure 1 — open up a codespace

This will open up a VS Code window in your browser. There are also options to open up a Jupyter Notebook or Jetbrains IDE (Pycharm). You can also install an

Rstudio IDE into the codespace. It will look something like this - note that the repository is already linked and checked out into the codespace:

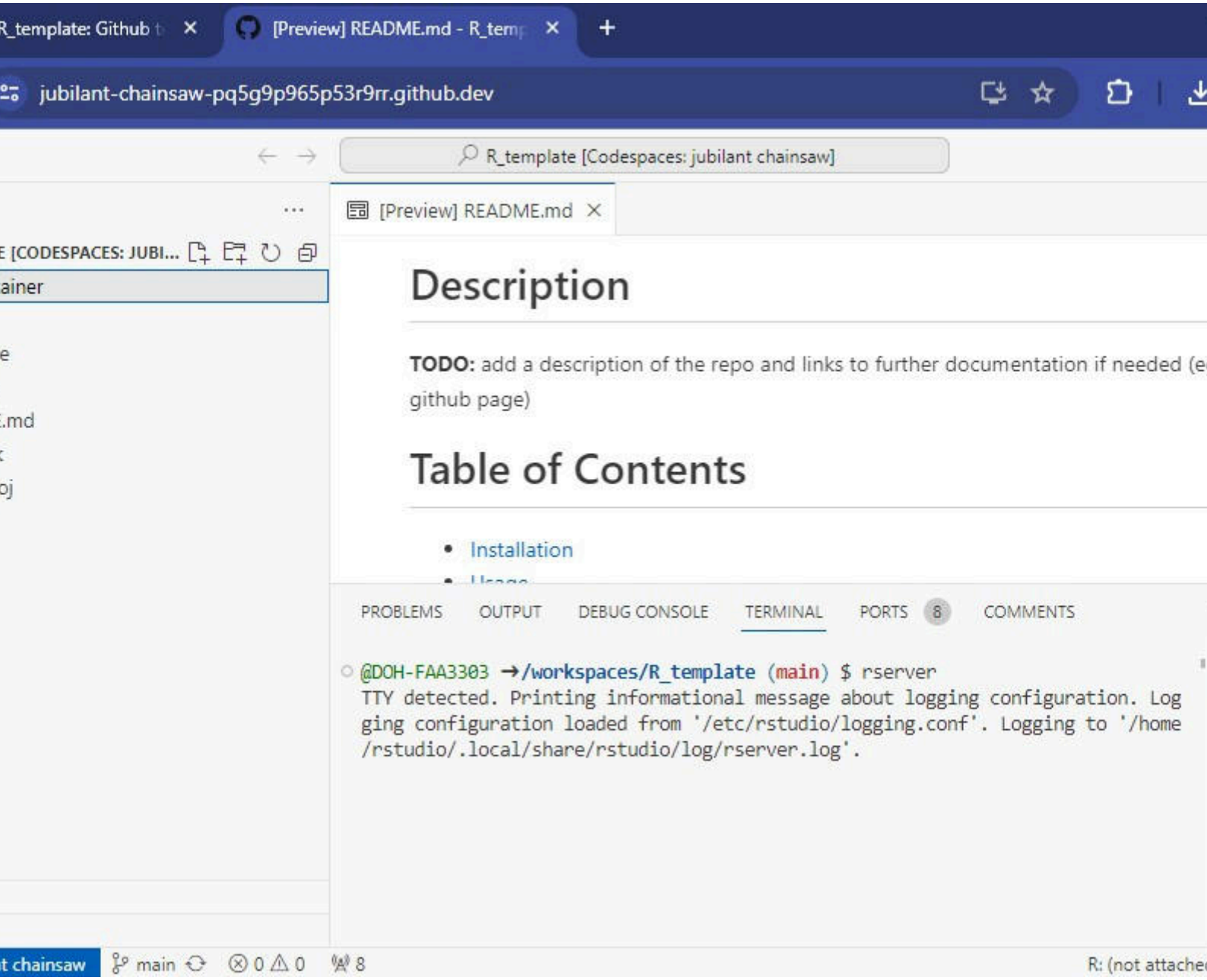


Figure 2 — VS Code IDE in Github Codespaces

Here you can install most software. You can also customize the Codespace so that whenever someone opens one in your repo it will come with software pre-installed. More on that in the devcontainers section

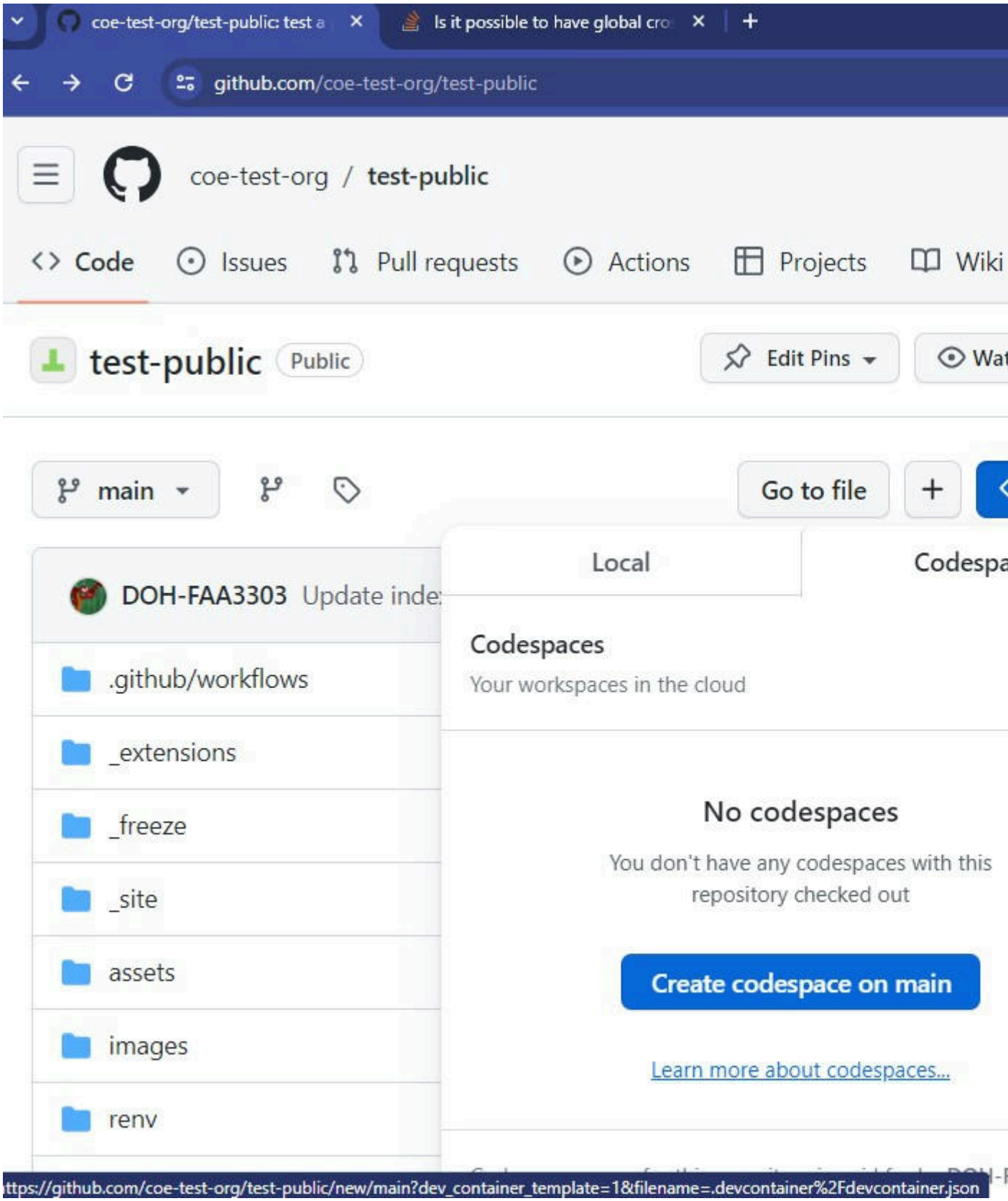
15.2 Devcontainers

Devcontainers are a way to install software into a Codespace so that whenever a user opens up the Codespace they won't need to install anything themselves. Making a container can be a little tricky, so we've made Github templates that

have devcontainers already made. See templates. There are R, Python, and general default templates. These containers will install R, Rstudio, Python, and all the packages in the repo's virtual environments (venv, conda, pip, renv, etc) so that the user can run all the code in your repo within a couple minutes.

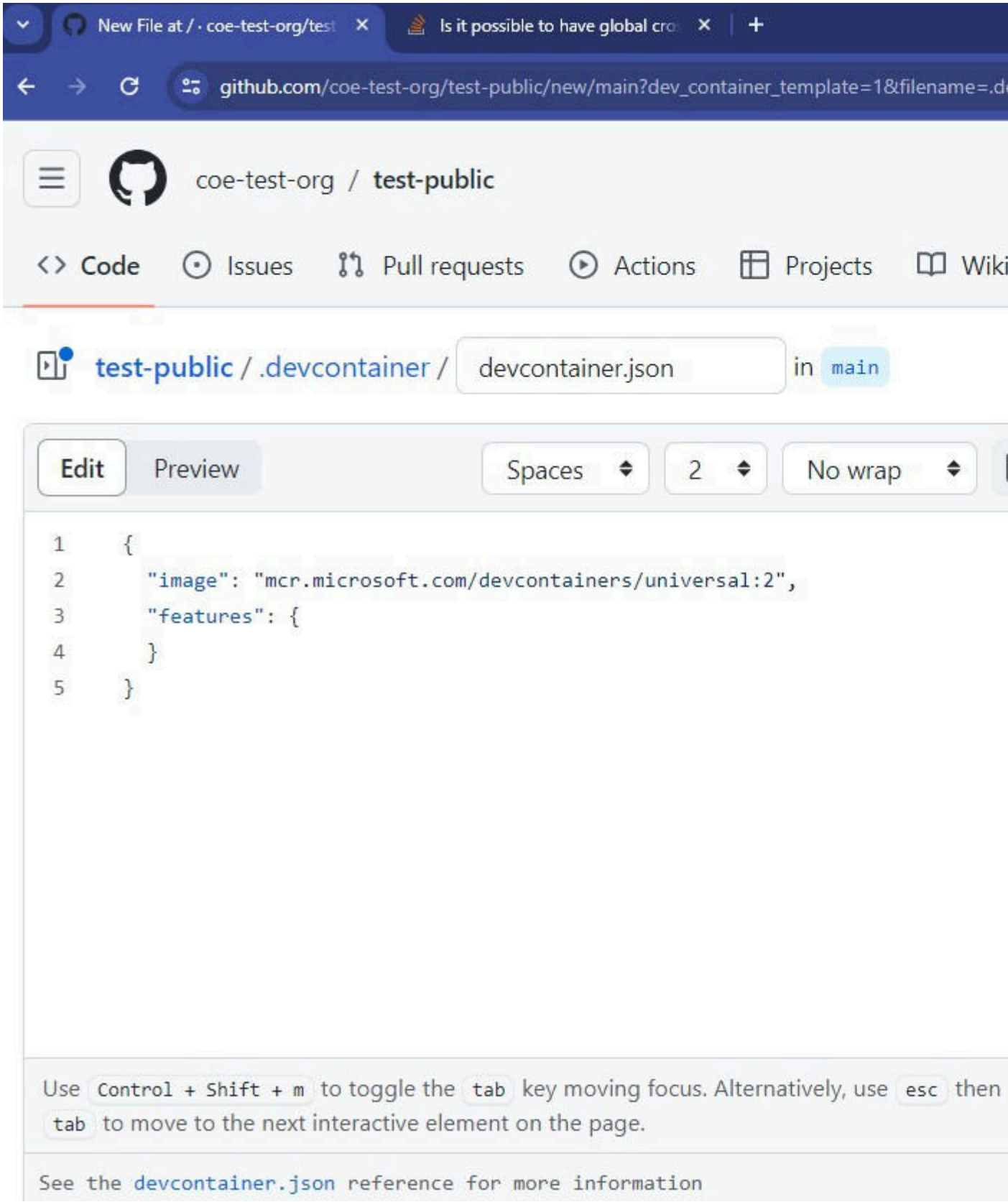
To set up a devcontainer for yourself;

1. Click on `Code > Codespaces > Configure dev container



2. This will make a folder named `.devcontainer` at the root of your repo
3. In that folder it will make a file named `devcontainer.json`

4. On the right there is a searchable marketplace for software to add to your container



5. Each one comes with instructions on how to add the software to the `.devcontainer.json`

For more information about Codespaces, see the guides here

16 Virtual Environments

Virtual Environments are another great way to make sure aspects of your repo are reproducible. They are commonly used to record package versions that the code/project uses. For more on virtual environments, please see the venv guide.

17 Github Releases

Github Releases save code snapshots, versions, and changelogs of your repo. They are a great way for end users and developers to use different versions of their code and visualize changes that happened with each version. Please see the Github Releases guide for more information.

18 Documentation

Your code should be well documented so that end users (and developers) can understand what code is doing, how to install the software, and the utility of the project.

In general, you should have a README.md file in your repo that explains at least a high level summary of the code in the repo and what it does, how to install the code, outputs, and how to contribute to the repo. In addition, it may be a good idea to make a Github Page (a static website hosted in your repo) that explains the code in more detail. See the documentation guides here.

Having a Github Page is necessary if you have a package. Consider using software like `pkgdown` for R or `quartodoc` for Python (or other related software that helps link code to your documentation automatically). See more about package documentation here.