{() => fs}

Fullstack > Part 11 > Introduction to CI/CD

a Introduction to CI/CD

During this part, you will build a robust *deployment pipeline* to a ready made example project starting in exercise 11.2. You will fork the example project and that will create you a personal copy of the repository. In the last two exercises, you will build another deployment pipeline for some of *your own* previously created apps!

There are 21 exercises in this part, and you need to complete *each* exercise for completing the course. Exercises are submitted via the submissions system just like in the previous parts, but unlike parts 0 to 7, the submission goes to a different "course instance".

This part will rely on many concepts covered in the previous parts of the course. It is recommended that you finish at least parts 0 to 5 before starting this part.

Unlike the other parts of this course, you do not write many lines of code in this part, it is much more about configuration. Debugging code might be hard but debugging configurations is way harder, so in this part, you need lots of patience and discipline!

Getting software to production

Writing software is all well and good but nothing exists in a vacuum. Eventually, we'll need to deploy the software to production, i.e. give it to the real users. After that we need to maintain it, release new versions, and work with other people to expand that software.

We've already used GitHub to store our source code, but what happens when we work within a team with more developers?

Many problems may arise when several developers are involved. The software might work just fine on *my computer*, but maybe some of the other developers are using a different operating system or different library versions. It is not uncommon that a code works just fine in one developer's machine but another developer can not even get it started. This is often called the "works on my machine" problem.

There are also more involved problems. If two developers are both working on changes and they haven't decided on a way to deploy to production, whose changes get deployed? How would it be possible to prevent one developer's changes from overwriting another's?

In this part, we'll cover ways to work together and build and deploy software in a strictly defined way so that it's clear *exactly* what will happen under any given circumstance.

Some useful terms

In this part we'll be using some terms you may not be familiar with or you may not have a good understanding of. We'll discuss some of these terms here. Even if you are familiar with the terms, give this section a read so when we use the terms in this part, we're on the same page.

Branches

Git allows multiple copies, streams, or versions of the code to co-exist without overwriting each other. When you first create a repository, you will be looking at the main branch (usually in Git, we call this *main* or *master*, but that does vary in older projects). This is fine if there's only one developer for a project and that developer only works on one feature at a time.

Branches are useful when this environment becomes more complex. In this context, each developer can have one or more branches. Each branch is effectively a copy of the main branch with some changes that make it diverge from it. Once the feature or change in the branch is ready it can be *merged* back into the main branch, effectively making that feature or change part of the main software. In this way, each developer can work on their own set of changes and not affect any other developer until the changes are ready.

But once one developer has merged their changes into the main branch, what happens to the other developers' branches? They are now diverging from an older copy of the main branch. How will the developer on the later branch know if their changes are compatible with the current state of the main branch? That is one of the fundamental questions we will be trying to answer in this part.

You can read more about branches e.g. from here.

Pull request

In GitHub merging a branch back to the main branch of software is quite often happening using a mechanism called pull request, where the developer who has done some changes is requesting the changes to be merged to the main branch. Once the pull request, or PR as it's often called, is made or *opened*, another developer checks that all is ok and *merges* the PR.

If you have proposed changes to the material of this course, you have already made a pull request!

Build

The term "build" has different meanings in different languages. In some interpreted languages such as Python or Ruby, there is actually no need for a build step at all.

In general when we talk about building we mean preparing software to run on the platform where it's intended to run. This might mean, for example, that if you've written your application in TypeScript, and you intend to run it on Node, then the build step might be transpiling the TypeScript into JavaScript.

This step is much more complicated (and required) in compiled languages such as C and Rust where the code needs to be compiled into an executable.

In part 7 we had a look at Webpack that is the current de facto tool for building a production version of a React or any other frontend JavaScript or TypeScript codebase.

Deploy

Deployment refers to putting the software where it needs to be for the end-user to use it. In the case of libraries, this may simply mean pushing an npm package to a package archive (such as npmjs.com) where other users can find it and include it in their software.

Deploying a service (such as a web app) can vary in complexity. In part 3 our deployment workflow involved running some scripts manually and pushing the repository code to Fly.io or Render hosting service.

In this part, we'll develop a simple "deployment pipeline" that deploys each commit of your code automatically to Fly.io or Render *if* the committed code does not break anything.

Deployments can be significantly more complex, especially if we add requirements such as "the software must be available at all times during the deployment" (zero downtime deployments) or if we have to take things like database migrations into account. We won't cover complex deployments like those in this part but it's important to know that they exist.

What is CI?

The strict definition of CI (Continuous Integration) and the way the term is used in the industry may sometimes be different. One influential but quite early (written already in 2006) discussion of the topic is in Martin Fowler's blog.

Strictly speaking, CI refers to merging developer changes to the main branch often, Wikipedia even helpfully suggests: "several times a day". This is usually true but when we refer to CI in industry, we're quite often talking about what happens after the actual merge happens.

We'd likely want to do some of these steps:

- Lint: to keep our code clean and maintainable
- Build: put all of our code together into runnable software bundle
- Test: to ensure we don't break existing features
- Package: Put it all together in an easily movable batch

- Deploy: Make it available to the world

We'll discuss each of these steps (and when they're suitable) in more detail later. What is important to remember is that this process should be strictly defined.

Usually, strict definitions act as a constraint on creativity/development speed. This, however, should usually not be true for CI. This strictness should be set up in such a way as to allow for easier development and working together. Using a good CI system (such as GitHub Actions that we'll cover in this part) will allow us to do this all automagically.

Packaging and Deployment as a part of CI

It may be worthwhile to note that packaging and especially deployment are sometimes not considered to fall under the umbrella of CI. We'll add them in here because in the real world it makes sense to lump it all together. This is partly because they make sense in the context of the flow and pipeline (I want to get my code to users) and partially because these are in fact the most likely point of failure.

The packaging is often an area where issues crop up in CI as this isn't something that's usually tested locally. It makes sense to test the packaging of a project during the CI workflow even if we don't do anything with the resulting package. With some workflows, we may even be testing the already built packages. This assures us that we have tested the code in the same form as what will be deployed to production.

What about deployment then? We'll talk about consistency and repeatability at length in the coming sections but we'll mention here that we want a process that always looks the same, whether we're running tests on a development branch or the main branch. In fact, the process may *literally* be the same with only a check at the end to determine if we are on the main branch and need to do a deployment. In this context, it makes sense to include deployment in the CI process since we'll be maintaining it at the same time we work on CI.

Is this CD thing related?

The terms *Continuous Delivery* and *Continuous Deployment* (both of which have the acronym CD) are often used when one talks about CI that also takes care of deployments. We won't bore you with the exact definition (you can use e.g. Wikipedia or another Martin Fowler blog post) but in general, we refer to CD as the practice where the main branch is kept deployable at all times. In general, this is also frequently coupled with automated deployments triggered from merges into the main branch.

What about the murky area between CI and CD? If we, for example, have tests that must be run before any new code can be merged to the main branch, is this CI because we're making frequent merges to the main branch, or is it CD because we're making sure that the main branch is always deployable?

So, some concepts frequently cross the line between CI and CD and, as we discussed above, deployment sometimes makes sense to consider CD as part of CI. This is why you'll often see references to CI/CD to describe the entire process. We'll use the terms "CI" and "CI/CD" interchangeably in this part.

Why is it important?

Above we talked about the "works on my machine" problem and the deployment of multiple changes, but what about other issues. What if Alice committed directly to the main branch? What if Bob used a branch but didn't bother to run tests before merging? What if Charlie tries to build the software for production but does so with the wrong parameters?

With the use of continuous integration and systematic ways of working, we can avoid these.

- We can disallow commits directly to the main branch
- We can have our CI process run on all Pull Requests (PRs) against the main branch and allow merges only when our desired conditions are met e.g. tests pass
- We can build our packages for production in the known environment of the CI system

There are other advantages to extending this setup:

- If we use CI/CD with deployment every time there is a merge to the main branch, then we know that it will always work in production
- If we only allow merges when the branch is up to date with the main branch, then we can be sure that different developers don't overwrite each other's changes

Note that in this part we are assuming that the *main* branch contains the code that is running in production. There are numerous different workflows one can use with Git, e.g. in some cases, it may be a specific *release branch* that contains the code that is running in production.

Important principles

It's important to remember that CI/CD is not the goal. The goal is better, faster software development with fewer preventable bugs and better team cooperation.

To that end, CI should always be configured to the task at hand and the project itself. The end goal should be kept in mind at all times. You can think of CI as the answer to these questions:

- How to make sure that tests run on all code that will be deployed?
- How to make sure that the main branch is deployable at all times?
- How to ensure that builds will be consistent and will always work on the platform it'd be deploying to?
- How to make sure that the changes don't overwrite each other?
- How to make deployments happen at the click of a button or automatically when one merges to the main branch?

There even exists scientific evidence on the numerous benefits the usage of CI/CD has. According to a large study reported in the book Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations, the use of CI/CD correlate heavily with organizational success (e.g. improves profitability and product quality, increases market share, shortens the time to market). CI/CD even makes developers happier by reducing their burnout rate. The results summarized in the book are also reported in scientific articles such as this.

Documented behavior

There's an old joke that a bug is just an "undocumented feature". We'd like to avoid that. We'd like to avoid any situations where we don't know the exact outcome. For example, if we depend on a label on a PR to define whether something is a "major", "minor" or "patch" release (we'll cover the meanings of those terms later), then it's important that we know what happens if a developer forgets to put a label on their PR. What if they put a label on after the build/test process has started? What happens if the developer changes the label mid-way through, which one is the one that actually releases?

It's possible to cover all cases you can think of and still have gaps where the developer will do something "creative" that you didn't think of, so it's important to have the process fail safely in this case.

For example, if we have the case mentioned above where the label changes midway through the build. If we didn't think of this beforehand, it might be best to fail the build and alert the user if something we weren't expecting happened. The alternative, where we deploy the wrong type of version anyway, could result in bigger problems, so failing and notifying the developer is the safest way out of this situation.

Know the same thing happens every time

We might have the best tests imaginable for our software, tests that catch every possible issue. That's great, but they're useless if we don't run them on the code before it's deployed.

We need to guarantee that the tests will run and we need to be sure that they run against the code that will actually be deployed. For example, it's no use if the tests are *only* run against Alice's branch if they would fail after merging to the main branch. We're deploying from the main branch so we need to make sure that the tests are run against a copy of the main branch with Alice's changes merged in.

This brings us to a critical concept. We need to make sure that the same thing happens every time. Or rather that the required tasks are all performed and in the right order.

Code always kept deployable

Having code that's always deployable makes life easier. This is especially true when the main branch contains the code running in the production environment. For example, if a bug is found and it needs to be fixed, you can pull a copy of the main branch (knowing it is the code running in production), fix the bug, and make a pull request back to the main branch. This is relatively straight forward.

If, on the other hand, the main branch and production are very different and the main branch is not deployable, then you would have to find out what code *is* running in production, pull a copy of that, fix the bug, figure out a way to push it back, then work out how to deploy that specific commit. That's not great and would have to be a completely different workflow from a normal deployment.

Knowing what code is deployed (sha sum/version)

It's often important to know what is actually running in production. Ideally, as we discussed above, we'd have the main branch running in production. This is not always possible. Sometimes we intend to have the main branch in production but a build fails, sometimes we batch together several changes and want to have them all deployed at once.

What we need in these cases (and is a good idea in general) is to know exactly *what code is running in production*. Sometimes this can be done with a version number, sometimes it's useful to have the commit SHA sum (uniquely identifying hash of that particular commit in git) attached to the code. We will discuss versioning further a bit later in this part.

It is even more useful if we combine the version information with a history of all releases. If, for example, we found out that a particular commit has introduced a bug, we can find out exactly when that was released and how many users were affected. This is especially useful when that bug has written bad data to the database. We'd now be able to track where that bad data went based on the time.

Types of CI setup

To meet some of the requirements listed above, we want to dedicate a separate server for running the tasks in continuous integration. Having a separate server for the purpose minimizes the risk that something else interferes with the CI/CD process and causes it to be unpredictable.

There are two options: host our own server or use a cloud service.

Jenkins (and other self-hosted setups)

Among the self-hosted options, Jenkins is the most popular. It's extremely flexible and there are plugins for almost anything (except that one thing you want to do). This is a great option for many applications, using a self-hosted setup means that the entire environment is under your control, the number of resources can be controlled, secrets (we'll elaborate a little more on security in later sections of this part) are never exposed to anyone else and you can do anything you want on the hardware.

Unfortunately, there is also a downside. Jenkins is quite complicated to set up. It's very flexible but that means that there's often quite a bit of boilerplate/template code involved to get builds working. With Jenkins specifically, it also means that CI/CD must be set up with Jenkins' own domain-specific language. There are also the risks of hardware failures which can be an issue if the setup sees heavy use.

With self-hosted options, the billing is usually based on the hardware. You pay for the server. What you do on the server doesn't change the billing.

GitHub Actions and other cloud-based solutions

In a cloud-hosted setup, the setup of the environment is not something you need to worry about. It's there, all you need to do is tell it what to do. Doing that usually involves putting a file in your repository and then telling the CI system to read the file (or to check your repository for that particular file).

The actual CI config for the cloud-based options is often a little simpler, at least if you stay within what is considered "normal" usage. If you want to do something a little bit more special, then cloud-based options may become more limited, or you may find it difficult to do that one specific task for which the cloud platform just isn't built for.

In this part, we'll look at a fairly normal use case. The more complicated setups might, for example, make use of specific hardware resources, e.g. a GPU.

Aside from the configuration issue mentioned above, there are often resource limitations on cloud-based platforms. In a self-hosted setup, if a build is slow, you can just get a bigger server and throw more resources at it. In cloud-based options, this may not be possible. For example, in GitHub Actions, the nodes your builds will run on have 2 vCPUs and 8GB of RAM.

Cloud-based options are also usually billed by build time which is something to consider.

Why pick one over the other

In general, if you have a small to medium software project that doesn't have any special requirements (e.g. a need for a graphics card to run tests), a cloud-based solution is probably best. The configuration is simple and you don't need to go to the hassle or expense of setting up your own system. For smaller projects especially, this should be cheaper.

For larger projects where more resources are needed or in larger companies where there are multiple teams and projects to take advantage of it, a self-hosted CI setup is probably the way to go.

Why use GitHub Actions for this course

For this course, we'll use GitHub Actions. It is an obvious choice since we're using GitHub anyway. We can get a robust CI solution working immediately without any hassle of setting up a server or configuring a third-party cloud-based service.

Besides being easy to take into use, GitHub Actions is a good choice in other respects. It might be the best cloud-based solution at the moment. It has gained lots of popularity since its initial release in November 2019.

Exercise 11.1

Before getting our hands dirty with setting up the CI/CD pipeline let us reflect a bit on what we have read.

11.1 Warming up

Think about a hypothetical situation where we have an application being worked on by a team of about 6 people. The application is in active development and will be released soon.

Let us assume that the application is coded with some other language than JavaScript/TypeScript, e.g. in Python, Java, or Ruby. You can freely pick the language. This might even be a language you do not know much yourself.

Write a short text, say 200-300 words, where you answer or discuss some of the points below. You can check the length with https://wordcounter.net/. Save your answer to the file named *exercise1.md* in the root of the repository that you shall create in exercise 11.2.

The points to discuss:

- Some common steps in a CI setup include *linting*, *testing*, and *building*. What are the specific tools for taking care of these steps in the ecosystem of the language you picked? You can search for the answers by Google.
- What alternatives are there to set up the CI besides Jenkins and GitHub Actions? Again, you can ask Google!
- Would this setup be better in a self-hosted or a cloud-based environment? Why? What information would you need to make that decision?

Remember that there are no 'right' answers to the above!

Propose changes to material

Part 10
Previous part

Part 11b
Next part

About course

Course contents

FAQ

Partners

Challenge

UNIVERSITY OF HELSINKI

HOUSTON

{() => fs}

b Getting started with GitHub Actions

Before we start playing with GitHub Actions, let's have a look at what they are and how do they work.

GitHub Actions work on a basis of workflows. A workflow is a series of jobs that are run when a certain triggering event happens. The jobs that are run then themselves contain instructions for what GitHub Actions should do.

A typical execution of a workflow looks like this:

- Triggering event happens (for example, there is a push to the main branch).
- The workflow with that trigger is executed.
- Cleanup

Basic needs

In general, to have CI operate on a repository, we need a few things:

- A repository (obviously)
- Some definition of what the CI needs to do: This can be in the form of a specific file inside the repository or it can be defined in the CI system
- The CI needs to be aware that the repository (and the configuration file within it) exist
- The CI needs to be able to access the repository
- The CI needs permissions to perform the actions it is supposed to be able to do: For example, if the CI needs to be able to deploy to a production environment, it needs *credentials* for that environment.

That's the traditional model at least, we'll see in a minute how GitHub Actions short-circuit some of these steps or rather make it such that you don't have to worry about them!

GitHub Actions have a great advantage over self-hosted solutions: the repository is hosted with the CI provider. In other words, GitHub provides both the repository and the CI platform. This means that if we've enabled actions for a repository, GitHub is already aware of the fact that we have workflows defined and what those definitions look like.

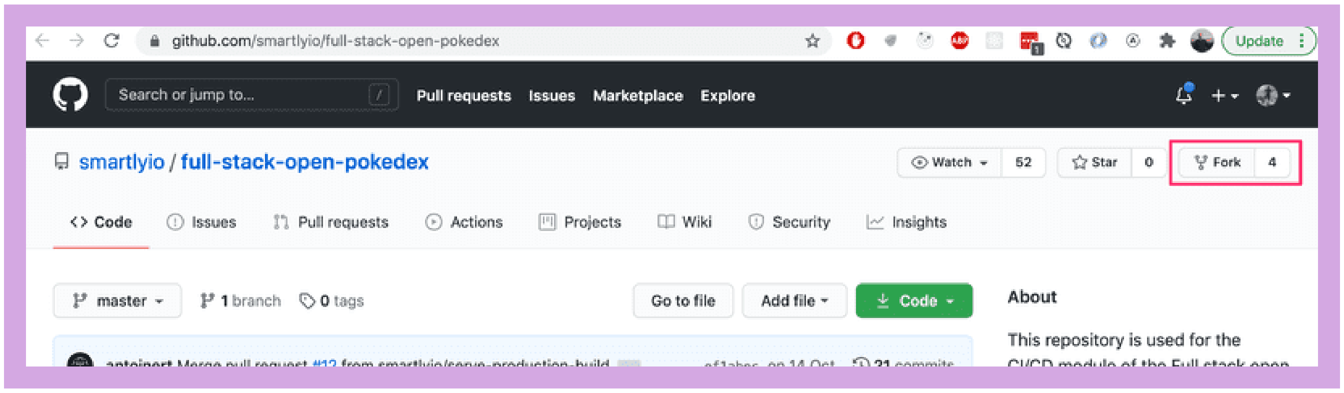
Exercise 11.2.

In most exercises of this part, we are building a CI/CD pipeline for a small project found in this example project repository.

11.2 The example project

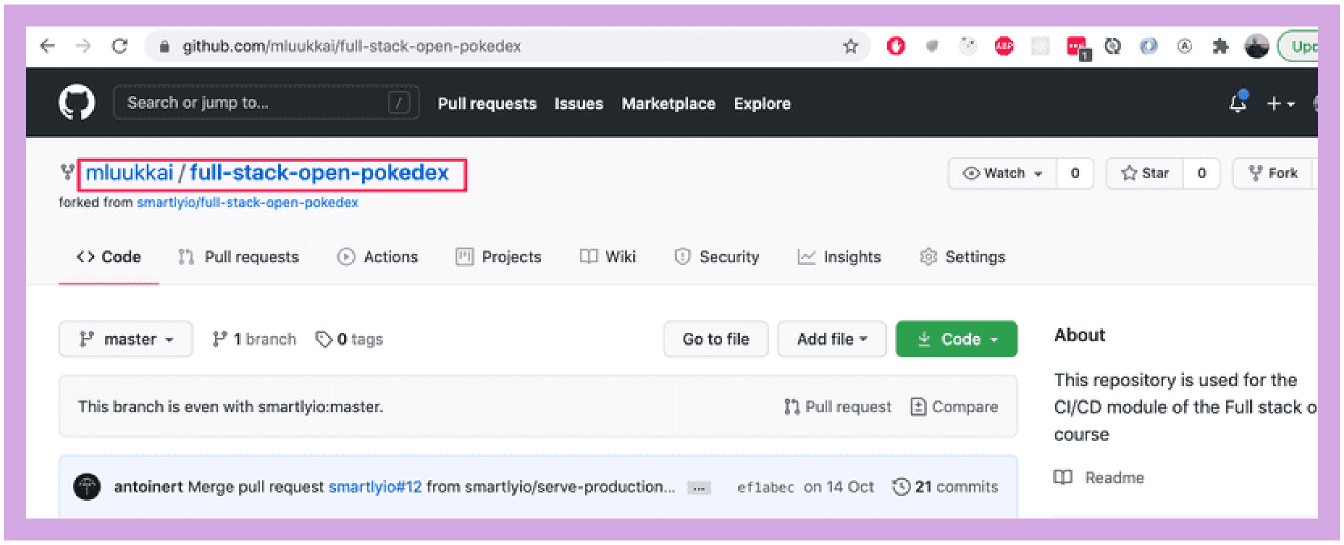
The first thing you'll want to do is to fork the example repository under your name. What it essentially does is it creates a copy of the repository under your GitHub user profile for your use.

To fork the repository, you can click on the Fork button in the top-right area of the repository view next to the Star button:



Once you've clicked on the Fork button, GitHub will start the creation of a new repository called `{github_username}/full-stack-open-pokedex` .

Once the process has been finished, you should be redirected to your brand-new repository:


github.com/mluukkai/full-stack-open-pokedex
Search or jump to...
Pull requests
Issues
Marketplace
Explore
mluukkai / full-stack-open-pokedex
forked from smartlyio/full-stack-open-pokedex
Watch 0
Star 0
Fork
Code
Pull requests
Actions
Projects
Wiki
Security
Insights
Settings
master
1 branch
0 tags
Go to file
Add file
Code
About
This repository is used for the CI/CD module of the Full stack o
course
This branch is even with smartlyio:master.
Pull request
Compare
antoinert Merge pull request smartlyio#12 from smartlyio/serve-production...
ef1abec on 14 Oct
21 commits
Readme

Clone the project now to your machine. As always, when starting with a new code, the most obvious place to look first is the file `package.json`

***NOTE** since the project is already a bit old, you need Node 16 to work with it!*

Try now the following:

- install dependencies (by running `npm install`)
- start the code in development mode
- run tests
- lint the code

You might notice that the project contains some broken tests and linting errors. **Just leave them as they are for now.** We will get around those later in the exercises.

NOTE the tests of the project have been made with Jest. The course material in part 5 uses Vitest. From the usage point of view, the libraries have barely any difference.

As you might remember from part 3, the React code *should not* be run in development mode once it is deployed in production. Try now the following

- create a production *build* of the project
- run the production version locally

Also for these two tasks, there are ready-made npm scripts in the project!

Study the structure of the project for a while. As you notice both the frontend and the backend code are now in the same repository. In earlier parts of the course we had a separate repository for both, but having those in the same repository makes things much simpler when setting up a CI environment.

In contrast to most projects in this course, the frontend code *does not use* Vite but it has a relatively simple Webpack configuration that takes care of creating the development environment and creating the production bundle.

Getting started with workflows

The core component of creating CI/CD pipelines with GitHub Actions is something called a Workflow. Workflows are process flows that you can set up in your repository to run automated tasks such as building, testing, linting, releasing, and deploying to name a few! The hierarchy of a workflow looks as follows:

Workflow

- Job
  - Step
  - Step
- Job
  - Step

Each workflow must specify at least one Job, which contains a set of Steps to perform individual tasks. The jobs will be run in parallel and the steps in each job will be executed sequentially.

Steps can vary from running a custom command to using pre-defined actions, thus the name GitHub Actions. You can create customized actions or use any actions published by the community, which are plenty, but let's get back to that later!

For GitHub to recognize your workflows, they must be specified in `.github/workflows` folder in your repository. Each Workflow is its own separate file which needs to be configured using the `YAML` data-serialization language.

YAML is a recursive acronym for "YAML Ain't Markup Language". As the name might hint its goal is to be human-readable and it is commonly used for configuration files. You will notice below that it is indeed very easy to understand!

Notice that indentations are important in YAML. You can learn more about the syntax here.

A basic workflow contains three elements in a YAML document. These three elements are:

- name: Yep, you guessed it, the name of the workflow
- (on) triggers: The events that trigger the workflow to be executed
- jobs: The separate jobs that the workflow will execute (a basic workflow might contain only one job).

A simple workflow definition looks like this:

```yaml
name: Hello World!

on:
  push:
    branches:
      - main

jobs:
  hello_world_job:
    runs-on: ubuntu-20.04
    steps:
      - name: Say hello
        run: |
          echo "Hello World!"
```

There is one job named *hello_world_job*, it will be run in a virtual environment with Ubuntu 20.04. The job has just one step named "Say hello", which will run the `echo "Hello World!"` command in the shell.

So you may ask, when does GitHub trigger a workflow to be started? There are plenty of options to choose from, but generally speaking, you can configure a workflow to start once:

- An *event on GitHub* occurs such as when someone pushes a commit to a repository or when an issue or pull request is created
- A *scheduled event*, that is specified using the cron-syntax, happens
- An *external event* occurs, for example, a command is performed in an external application such as Slack or Discord messaging app

To learn more about which events can be used to trigger workflows, please refer to GitHub Action's documentation.

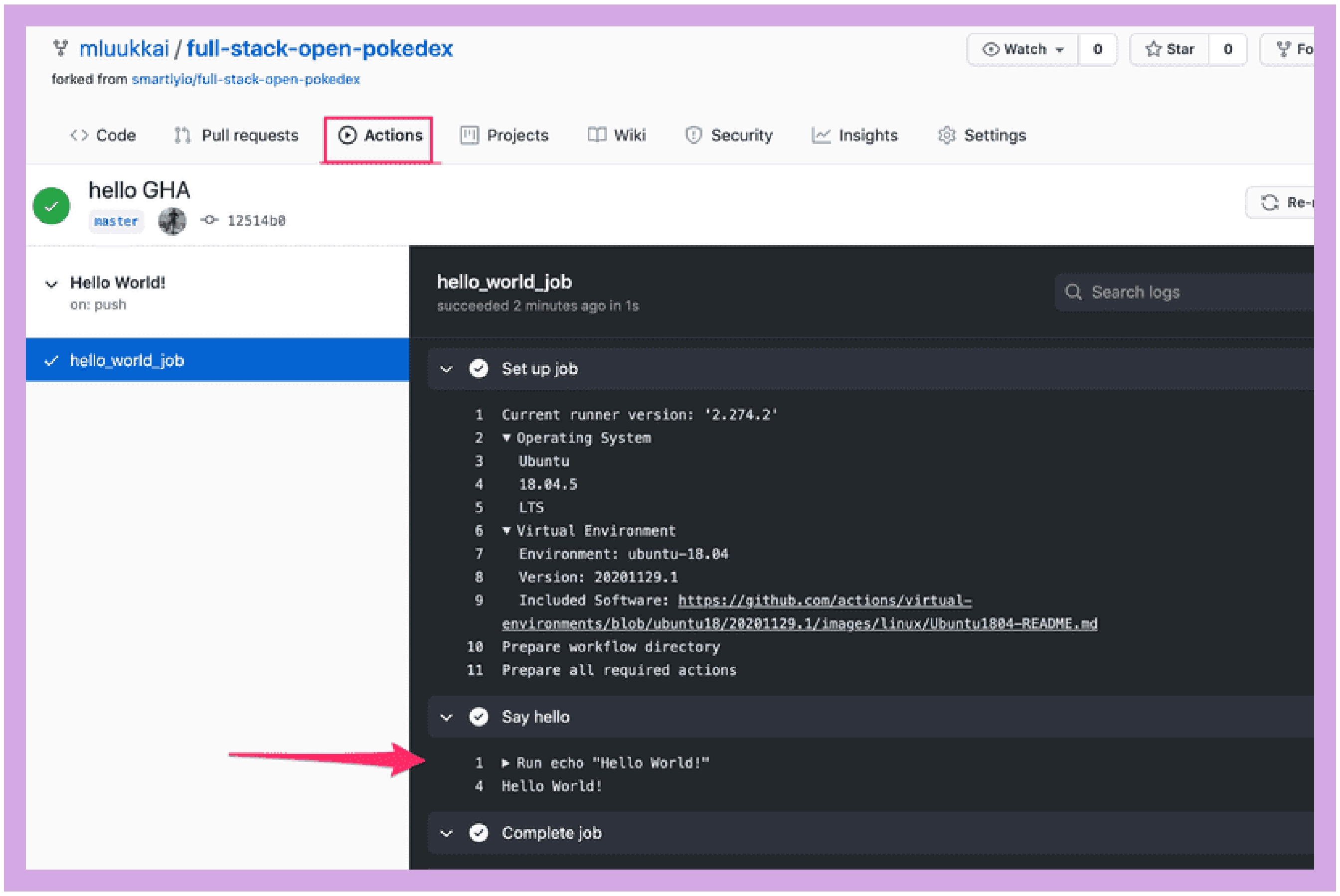
Exercises 11.3-11.4.

To tie this all together, let us now get GitHub Actions up and running in the example project!

11.3 Hello world!

Create a new Workflow that outputs "Hello World!" to the user. For the setup, you should create the directory `.github/workflows` and a file `hello.yml` to your repository.

To see what your GitHub Action workflow has done, you can navigate to the **Actions** tab in GitHub where you should see the workflows in your repository and the steps they implement. The output of your Hello World workflow should look something like this with a properly configured workflow.



hello_world_job
succeeded 2 minutes ago in 1s

```
Current runner version: '2.274.2'
▼ Operating System
  Ubuntu
  18.04.5
  LTS
▼ Virtual Environment
  Environment: ubuntu-18.04
  Version: 20201129.1
  Included Software: https://github.com/actions/virtual-environments/blob/ubuntu18/20201129.1/images/linux/Ubuntu1804-README.md
Prepare workflow directory
Prepare all required actions
```

```
▶ Run echo "Hello World!"
Hello World!
```

You should see the "Hello World!" message as an output. If that's the case then you have successfully gone through all the necessary steps. You have your first GitHub Actions workflow active!

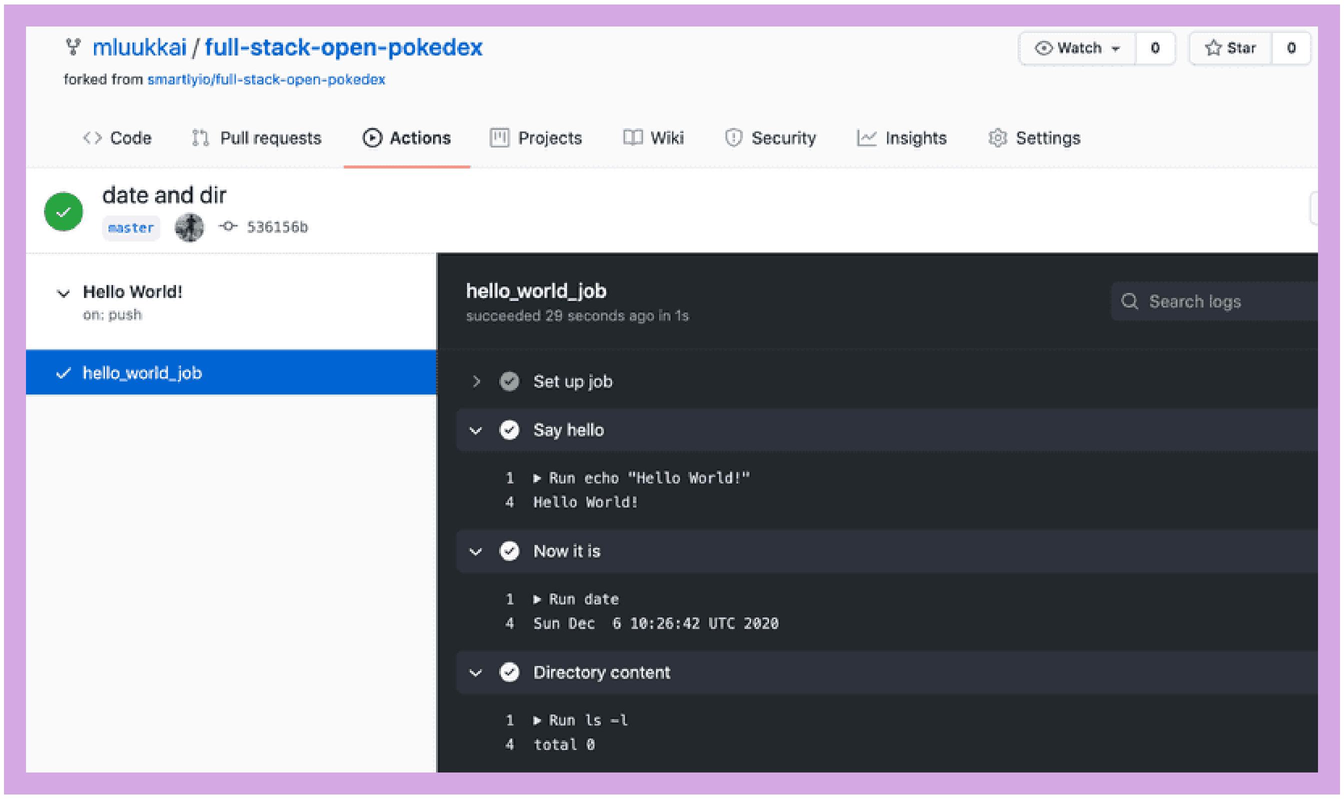
Note that GitHub Actions also informs you on the exact environment (operating system, and its `setup`) where your workflow is run. This is important since if something surprising happens, it makes debugging so much easier if you can reproduce all the steps in your machine!

11.4 Date and directory contents

Extend the workflow with steps that print the date and current directory content in the long format.

Both of these are easy steps, and just running commands `date` and `ls` will do the trick.

Your workflow should now look like this



```
hello_world_job
succeeded 29 seconds ago in 1s

Set up job
Say hello
1  ▶ Run echo "Hello World!"
4  Hello World!
Now it is
1  ▶ Run date
4  Sun Dec  6 10:26:42 UTC 2020
Directory content
1  ▶ Run ls -l
4  total 0
```

As the output of the command `ls -l` shows, by default, the virtual environment that runs our workflow *does not* have any code!

Setting up lint, test and build steps

After completing the first exercises, you should have a simple but pretty useless workflow set up. Let's make our workflow do something useful.

Let's implement a GitHub Action that will lint the code. If the checks don't pass, GitHub Actions will show a red status.

At the start, the workflow that we will save to file `pipeline.yml` looks like this:

```yaml
name: Deployment pipeline

on:
  push:
    branches:
      - main

jobs:
```

Before we can run a command to lint the code, we have to perform a couple of actions to set up the environment of the job.

Setting up the environment

Setting up the environment is an important task while configuring a pipeline. We're going to use an `ubuntu-20.04` virtual environment because this is the version of Ubuntu we're going to be running in production.

It is important to replicate the same environment in CI as in production as closely as possible, to avoid situations where the same code works differently in CI and production, which would effectively defeat the purpose of using CI.

Next, we list the steps in the "build" job that the CI would need to perform. As we noticed in the last exercise, by default the virtual environment does not have any code in it, so we need to *checkout the code* from the repository.

This is an easy step:

```
name: Deployment pipeline

on:
  push:
    branches:
      - main

jobs:
  simple_deployment_pipeline:
    runs-on: ubuntu-20.04
    steps:
      - uses: actions/checkout@v4
```

The uses keyword tells the workflow to run a specific *action*. An action is a reusable piece of code, like a function. Actions can be defined in your repository in a separate file or you can use the ones available in public repositories.

Here we're using a public action actions/checkout and we specify a version (`@v4`) to avoid potential breaking changes if the action gets updated. The `checkout` action does what the name implies: it checkouts the project source code from Git.

Secondly, as the application is written in JavaScript, Node.js must be set up to be able to utilize the commands that are specified in `package.json` . To set up Node.js, actions/setup-node action can be used. Version `20` is selected because it is the version the application is using in the production environment.

```
# name and trigger not shown anymore...

jobs:
  simple_deployment_pipeline:
    runs-on: ubuntu-20.04
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: '20'
```

As we can see, the `with` keyword is used to give a "parameter" to the action. Here the parameter specifies the version of Node.js we want to use.

Lastly, the dependencies of the application must be installed. Just like on your own machine we execute `npm install`. The steps in the job should now look something like

```
jobs:
  simple_deployment_pipeline:
    runs-on: ubuntu-20.04
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: '20'
      - name: Install dependencies
        run: npm install
```

Now the environment should be completely ready for the job to run actual important tasks in!

Lint

After the environment has been set up we can run all the scripts from `package.json` like we would on our own machine. To lint the code all you have to do is add a step to run the `npm run eslint` command.

```
jobs:
  simple_deployment_pipeline:
    runs-on: ubuntu-20.04
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: '20'
      - name: Install dependencies
        run: npm install
      - name: Check style
        run: npm run eslint
```

Note that the `name` of a step is optional, if you define a step as follows

```
- run: npm run eslint
```

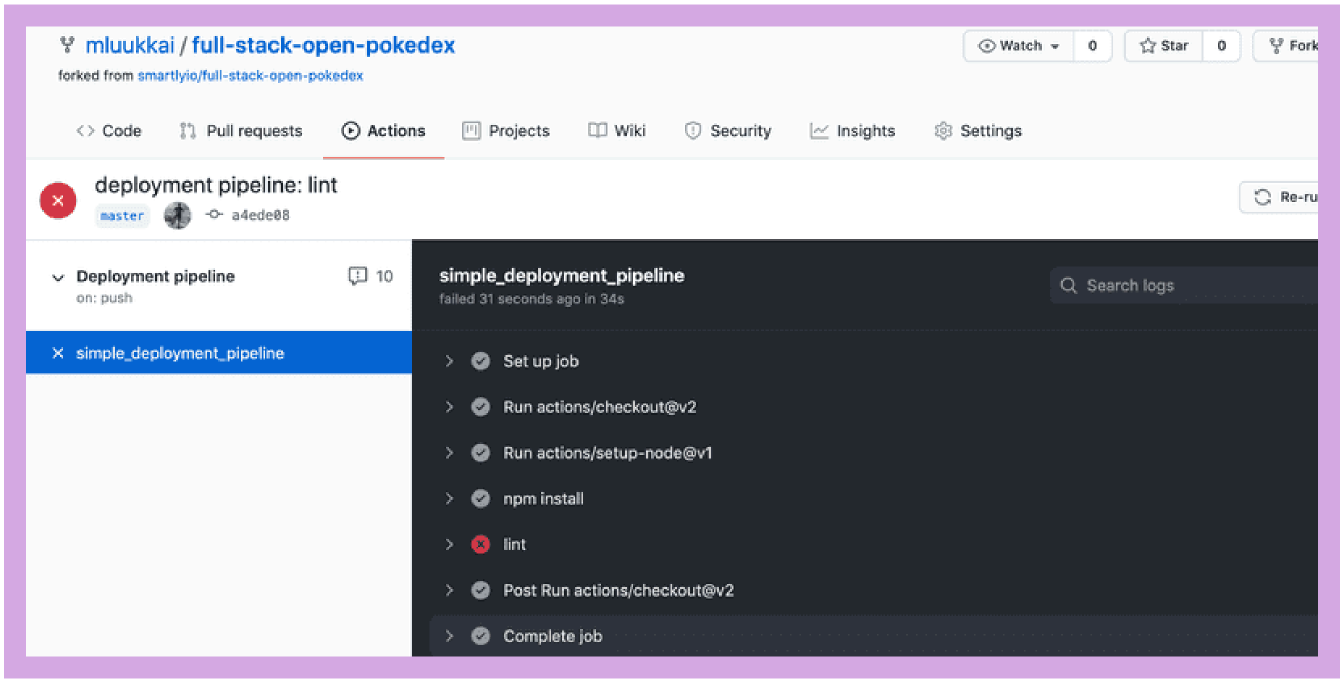
the command that is run is used as the default name.

Exercises 11.5.-11.9.

11.5 Linting workflow

Implement or *copy-paste* the "Lint" workflow and commit it to the repository. Use a new *yml* file for this workflow, you may call it e.g. *pipeline.yml*.

Push your code and navigate to "Actions" tab and click on your newly created workflow on the left. You should see that the workflow run has failed:

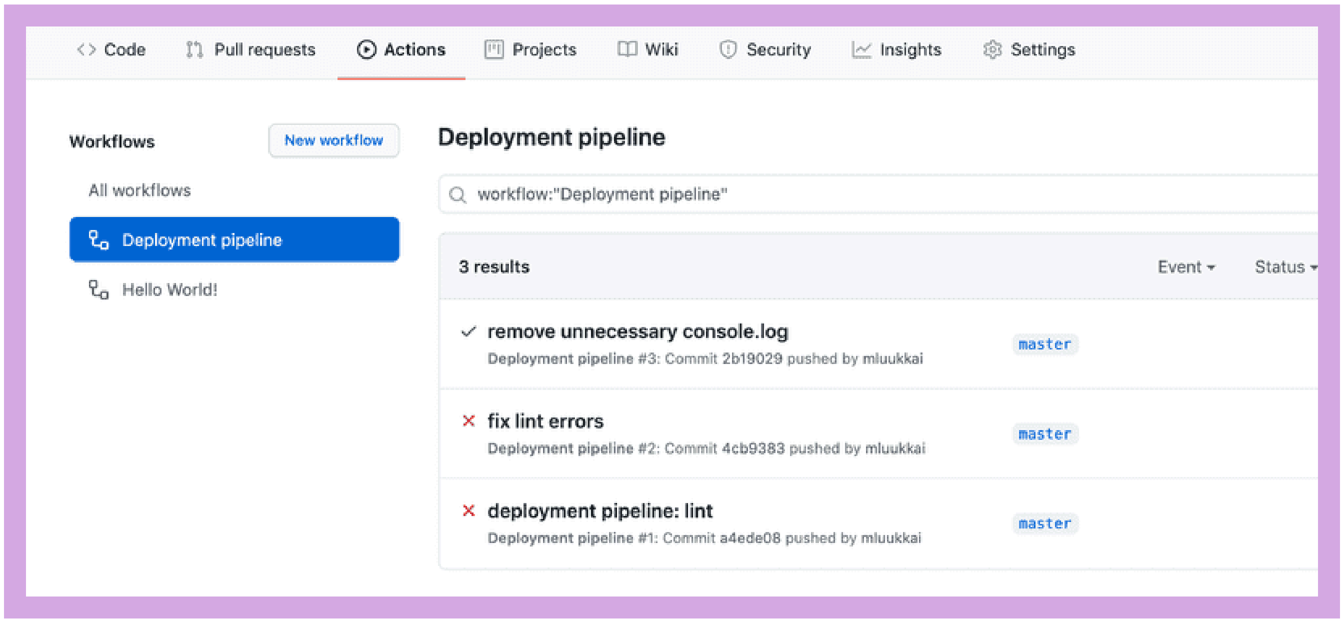

mluukkai / full-stack-open-pokedex
forked from smartlyio/full-stack-open-pokedex
Watch 0
Star 0
Fork
Code
Pull requests
Actions
Projects
Wiki
Security
Insights
Settings
deployment pipeline: lint
master
a4ede08
Re-ru
Deployment pipeline
on: push
10
simple_deployment_pipeline
simple_deployment_pipeline
failed 31 seconds ago in 34s
Search logs
Set up job
Run actions/checkout@v2
Run actions/setup-node@v1
npm install
lint
Post Run actions/checkout@v2
Complete job

11.6 Fix the code

There are some issues with the code that you will need to fix. Open up the workflow logs and investigate what is wrong.

A couple of hints. One of the errors is best to be fixed by specifying proper *env* for linting, see here how it can be done . One of the complaints concerning `console.log` statement could be taken care of by simply silencing the rule for that specific line. Ask google how to do it.

Make the necessary changes to the source code so that the lint workflow passes. Once you commit new code the workflow will run again and you will see updated output where all is green again:



Deployment pipeline

workflow:"Deployment pipeline"

3 results

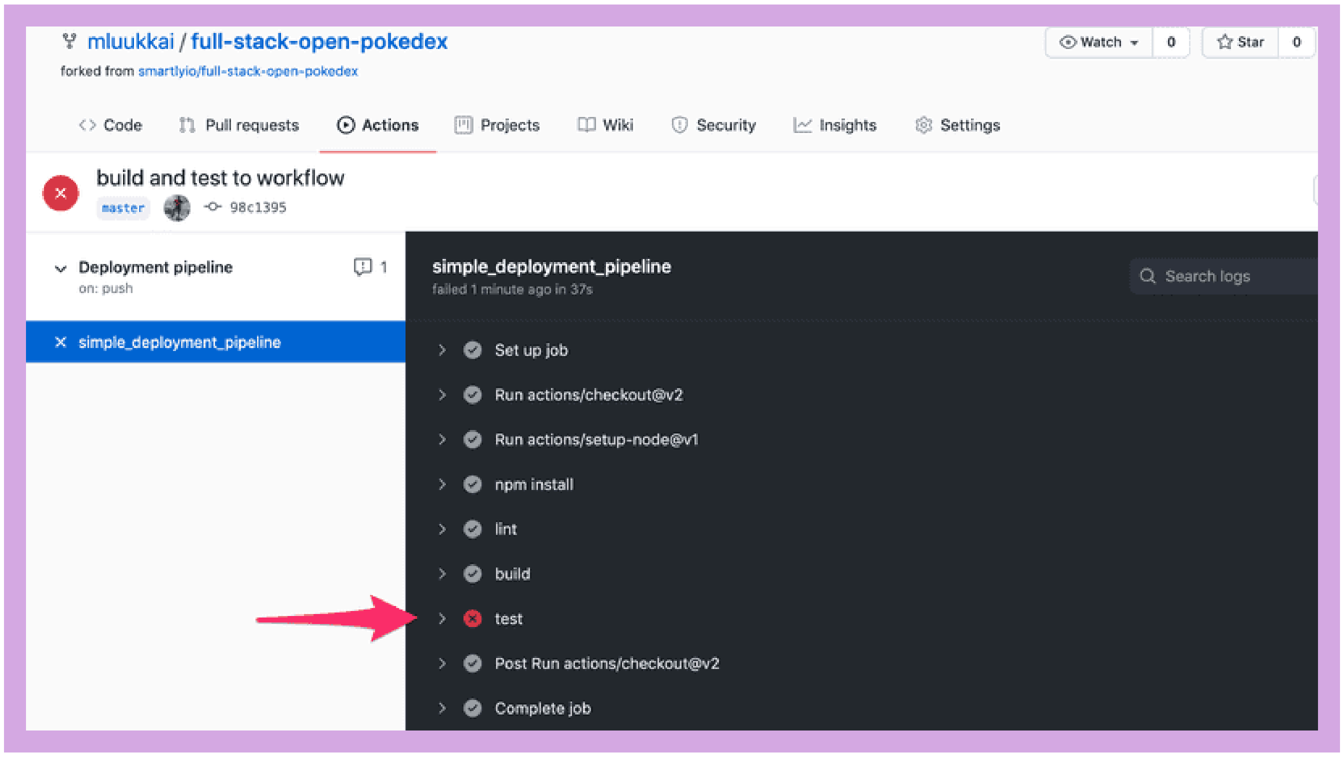
✓ remove unnecessary console.log
Deployment pipeline #3: Commit 2b19029 pushed by mluukkai — master

✗ fix lint errors
Deployment pipeline #2: Commit 4cb9383 pushed by mluukkai — master

✗ deployment pipeline: lint
Deployment pipeline #1: Commit a4ede08 pushed by mluukkai — master

11.7 Building and testing

Let's expand on the previous workflow that currently does the linting of the code. Edit the workflow and similarly to the lint command add commands for build and test. After this step outcome should look like this


mluukkai / full-stack-open-pokedex
forked from smartlyio/full-stack-open-pokedex
Watch 0
Star 0
Code
Pull requests
Actions
Projects
Wiki
Security
Insights
Settings
build and test to workflow
master
98c1395
Deployment pipeline
on: push
1
simple_deployment_pipeline
Search logs

simple_deployment_pipeline
failed 1 minute ago in 37s

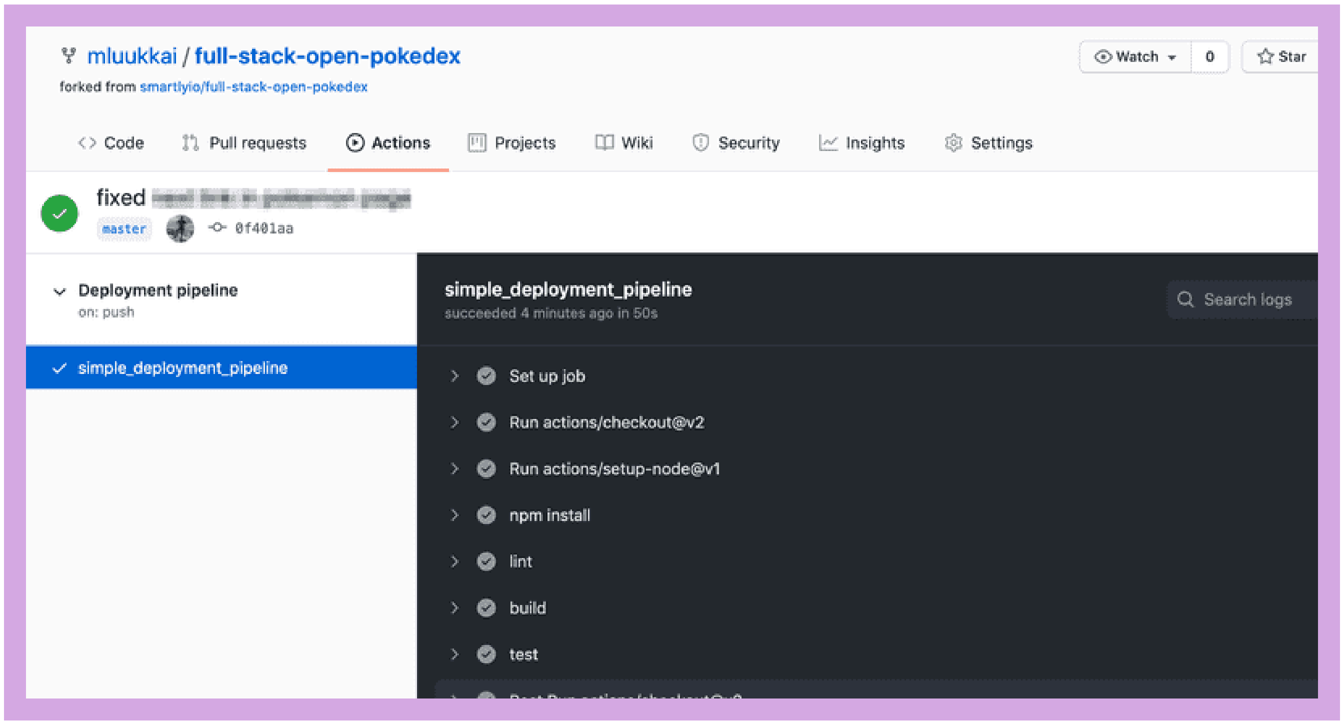
- Set up job
- Run actions/checkout@v2
- Run actions/setup-node@v1
- npm install
- lint
- build
- test
- Post Run actions/checkout@v2
- Complete job

As you might have guessed, there are some problems in code...

11.8 Back to green

Investigate which test fails and fix the issue in the code (do not change the tests).

Once you have fixed all the issues and the Pokedex is bug-free, the workflow run will succeed and show green!



```
simple_deployment_pipeline
succeeded 4 minutes ago in 50s
Set up job
Run actions/checkout@v2
Run actions/setup-node@v1
npm install
lint
build
test
```

11.9 Simple end-to-end tests

The current set of tests uses Jest to ensure that the React components work as intended. This is essentially the same thing that is done in the section Testing React apps of part 5 with Vitest.

Testing components in isolation is quite useful but that still does not ensure that the system as a whole works as we wish. To have more confidence about this, let us write a couple of really simple end-to-end tests similarly we did in section part 5. You could use Playwright or Cypress for the tests.

No matter which you choose, you should extend Jest-definition in package.json to prevent Jest from trying to run the e2e-tests. Assuming that directory `e2e-tests` is used for e2e-tests, the definition is:

```
{
  // ...
  "jest": {
    "testEnvironment": "jsdom",
    "testPathIgnorePatterns": ["e2e-tests"]
  }
}
```

Playwright

Set Playwright up (you'll find here all the info you need) to your repository. Note that in contrast to part 5, you should now install Playwright to the same project with the rest of the code!

Use this test first:

```
const { test, describe, expect, beforeEach } = require('@playwright/test')

describe('Pokedex', () => {
  test('front page can be opened', async ({ page }) => {
     await page.goto('')
    await expect(page.getByText('ivysaur')).toBeVisible()
    await expect(page.getByText('Pokémon and Pokémon character names are trademarks of 
Nintendo.')).toBeVisible()
  })
})
```

Note is that although the page renders the Pokemon names with an initial capital letter, the names are actually written with lowercase letters in the source, so you should test for `ivysaur` instead of `Ivysaur` !

Define a npm script `test:e2e` for running the e2e tests from the command line.

Remember that the Playwright tests *assume that the application is up and running* when you run the test! Instead of starting the app manually, you should now configure a *Playwright development server* to start the app while tests are executed, see here how that can be done.

Ensure that the test passes locally.

Once the end-to-end test works in your machine, include it in the GitHub Action workflow. That should be pretty easy by following this.

Cypress

Set Cypress up (you'll find here all the info you need) and use this test first:

```
describe('Pokedex', function() {
  it('front page can be opened', function() {
    cy.visit('http://localhost:5000')
    cy.contains('ivysaur')
    cy.contains('Pokémon and Pokémon character names are trademarks of Nintendo.')
  })
})
```

Define a npm script `test:e2e` for running the e2e tests from the command line.

Note is that although the page renders the Pokemon names with an initial capital letter, the names are actually written with lowercase letters in the source, so you should test for `ivysaur` instead of `Ivysaur` !

Ensure that the test passes locally. Remember that the Cypress tests `assume that the application is up and running` when you run the test! If you have forgotten the details, please see part 5 how to get up and running with Cypress.

Once the end-to-end test works in your machine, include it in the GitHub Action workflow. By far the easiest way to do that is to use the ready-made action cypress-io/github-action. The step that suits us is the following:

```yaml
- name: e2e tests
  uses: cypress-io/github-action@v5
  with:
    command: npm run test:e2e
    start: npm run start-prod
    wait-on: http://localhost:5000
```

Three options are used: command specifies how to run Cypress tests, start gives npm script that starts the server, and wait-on says that before the tests are run, the server should have started on url http://localhost:5000.

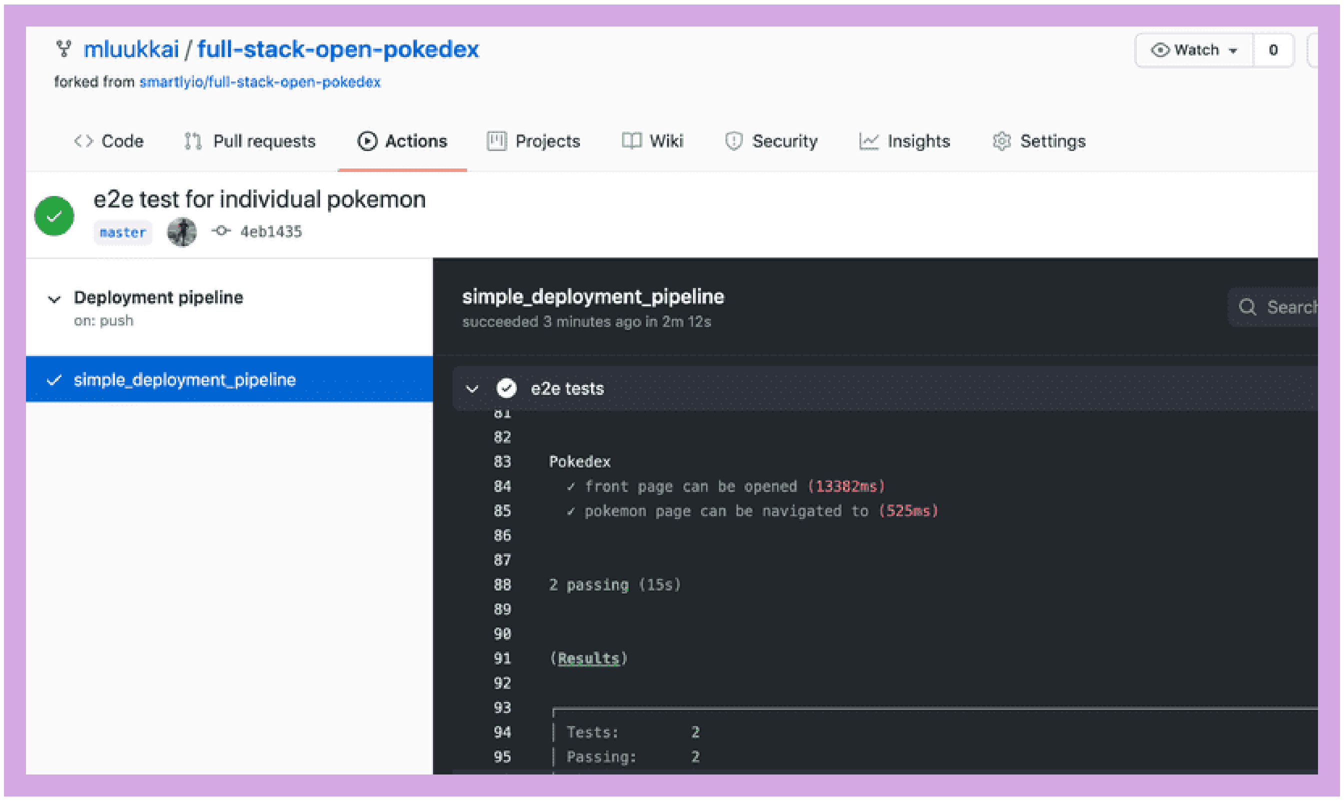
Note that you need to build the app in GitHub Actions before it can be started in production mode!

Once the pipeline works...

Once you are sure that the pipeline works, *write another test* that ensures that one can navigate from the main page to the page of a particular Pokemon, e.g. *ivysaur*. The test does not need to be a complex one, just check that when you navigate to a link, the page has some proper content, such as the string *chlorophyll* in the case of *ivysaur*.

Note the Pokemon abilities are written with lowercase letters in the source code (the capitalization is done in CSS), so *do not* test for *Chlorophyll* but rather *chlorophyll*.

The end result should be something like this


mluukkai / full-stack-open-pokedex
forked from smartlyio/full-stack-open-pokedex
Watch 0
Code
Pull requests
Actions
Projects
Wiki
Security
Insights
Settings
e2e test for individual pokemon
master
4eb1435
Deployment pipeline
on: push
simple_deployment_pipeline
simple_deployment_pipeline
succeeded 3 minutes ago in 2m 12s
Search
e2e tests

```
Pokedex
  ✓ front page can be opened (13382ms)
  ✓ pokemon page can be navigated to (525ms)


2 passing (15s)


(Results)

┌───────────────────
│ Tests:        2
│ Passing:      2
```

End-to-end tests are nice since they give us confidence that software works from the end user's perspective. The price we have to pay is the slower feedback time. Now executing the whole workflow takes quite much longer.

Propose changes to material

Part 11a
Previous part

Part 11c
Next part

About course

Course contents

FAQ

Partners

Challenge

UNIVERSITY OF HELSINKI

HOUSTON

{() => fs}

Fullstack > Part 11 > Deployment

c Deployment

Having written a nice application it's time to think about how we're going to deploy it to the use of real users.

In part 3 of this course, we did this by simply running a single command from terminal to get the code up and running the servers of the cloud provider Fly.io or Render.

It is pretty simple to release software in Fly.io and Render at least compared to many other types of hosting setups but it still contains risks: nothing prevents us from accidentally releasing broken code to production.

Next, we're going to look at the principles of making a deployment safely and some of the principles of deploying software on both a small and large scale.

Anything that can go wrong...

We'd like to define some rules about how our deployment process should work but before that, we have to look at some constraints of reality.

One phrasing of Murphy's Law holds that: "Anything that can go wrong will go wrong."

It's important to remember this when we plan out our deployment system. Some of the things we'll need to consider could include:

- What if my computer crashes or hangs during deployment?
- I'm connected to the server and deploying over the internet, what happens if my internet connection dies?
- What happens if any specific instruction in my deployment script/system fails?

- What happens if, for whatever reason, my software doesn't work as expected on the server I'm deploying to? Can I roll back to a previous version?
- What happens if a user does an HTTP request to our software just before we do deployment (we didn't have time to send a response to the user)?

These are just a small selection of what can go wrong during a deployment, or rather, things that we should plan for. Regardless of what happens, our deployment system should **never** leave our software in a broken state. We should also always know (or be easily able to find out) what state a deployment is in.

Another important rule to remember when it comes to deployments (and CI in general) is: "Silent failures are **very** bad!"

This doesn't mean that failures need to be shown to the users of the software, it means we need to be aware if anything goes wrong. If we are aware of a problem, we can fix it. If the deployment system doesn't give any errors but fails, we may end up in a state where we believe we have fixed a critical bug but the deployment failed, leaving the bug in our production environment and us unaware of the situation.

What does a good deployment system do?

Defining definitive rules or requirements for a deployment system is difficult, let's try anyway:

- Our deployment system should be able to fail gracefully at **any** step of the deployment.
- Our deployment system should **never** leave our software in a broken state.
- Our deployment system should let us know when a failure has happened. It's more important to notify about failure than about success.
- Our deployment system should allow us to roll back to a previous deployment
  - Preferably this rollback should be easier to do and less prone to failure than a full deployment
  - Of course, the best option would be an automatic rollback in case of deployment failures
- Our deployment system should handle the situation where a user makes an HTTP request just before/during a deployment.
- Our deployment system should make sure that the software we are deploying meets the requirements we have set for this (e.g. don't deploy if tests haven't been run).

Let's define some things we **want** in this hypothetical deployment system too:

- We would like it to be fast
- We'd like to have no downtime during the deployment (this is distinct from the requirement we have for handling user requests just before/during the deployment).

Next we will have three sets of exercises for automating the deployment with GitHub Actions, one for Fly.io, another one for Render. The process of deployment is always specific to the particular cloud provider, so you can also do both the exercise sets if you want to see the differences on how these services work with respect to deployments.

Has the app been deployed?

Since we are not making any real changes to the app, it might be a bit hard to see if the app deployment really works. Let us create a dummy endpoint in the app that makes it possible to do some code changes and to ensure that the deployed version has really changed:

```
app.get('/version', (req, res) => {
  res.send('1') // change this string to ensure a new version deployed
})
```

Exercises 11.10-11.12. (Fly.io)

If you rather want to use other hosting options, there is an alternative set of exercises for Render.

11.10 Deploying your application to Fly.io

Setup your application in Fly.io hosting service like the one we did in part 3.

In contrast to part 3, in this part we *do not deploy the code* to Fly.io ourselves (with the command *flyctl deploy*), we let the GitHub Actions workflow do that for us.

Before going to the automated deployment, we shall ensure in this exercise that the app can be deployed manually.

So, create a new app in Fly.io. After that generate a Fly.io API token with the command

```
flyctl auth token
```

You'll need the token soon for your deployment workflow so save it somewhere (but do not commit that to GitHub)!

As said, before setting up the deployment pipeline in the next exercise we will now ensure that a manual deployment with the command *flyctl deploy* works.

A couple of changes are needed.

The configuration file *fly.toml* should be modified to include the following:

```
[env]
  PORT = "3000" # add this where PORT matches the internal_port below

[processes]
  app = "node app.js" # add this

[http_service]
  internal_port = 3000
  force_https = true
  auto_stop_machines = true
  auto_start_machines = true
  min_machines_running = 0
  processes = ["app"]
```

In processes we define the command that starts the application. Without this change Fly.io just starts the React dev server and that causes it to shut down since the app itself does not start up. We will also set up the PORT to be passed to the app as an environment variable.

We also need to alter the file `.dockerignore` a bit, the next line should be removed:

```
dist
```

If the line is not removed, the product build of the frontend does not get downloaded to the Fly.io server.

Deployment should now work `if` the production build exists in the local machine, that is, the command `npm build` is run.

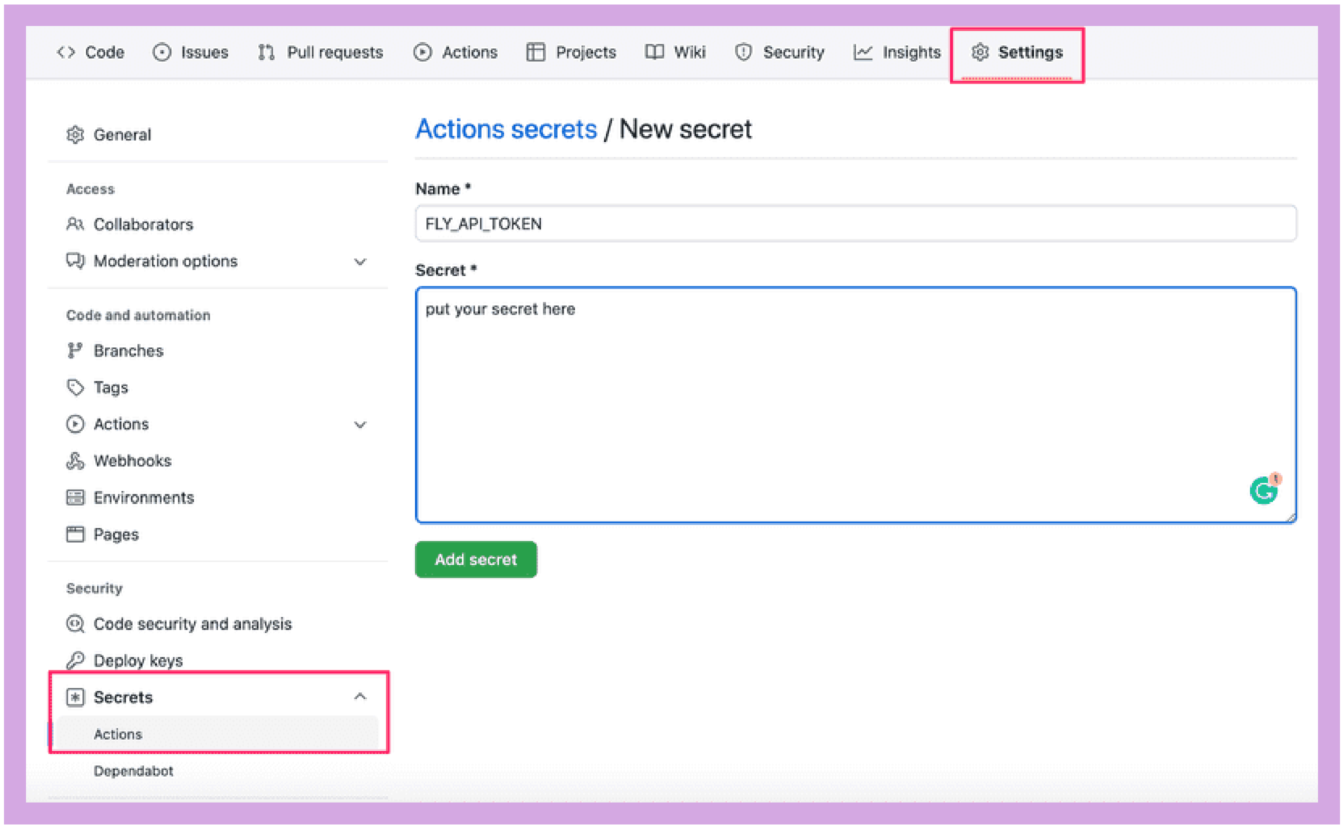
Before moving to the next exercise, make sure that the manual deployment with the command *flyctl deploy* works!

11.11 Automatic deployments

Extend the workflow with a step to deploy your application to Fly.io by following the advice given here.

Note that the GitHub Action should create the production build (with `npm run build`) before the deployment step!

You need the authorization token that you just created for the deployment. The proper way to pass it's value to GitHub Actions is to use `Repository secrets`:



Actions secrets / New secret

Name *

FLY_API_TOKEN

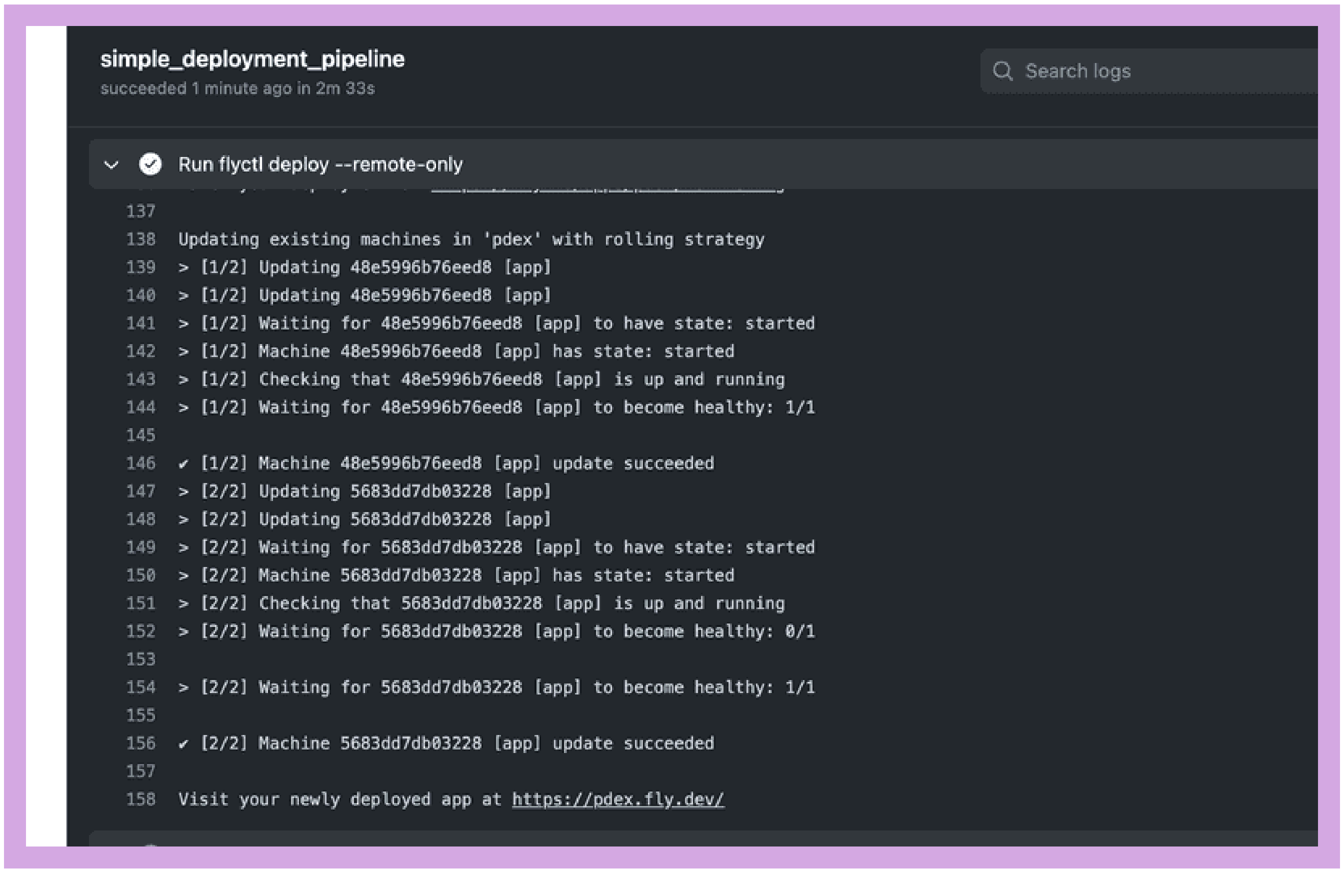
Secret *

put your secret here

Now the workflow can access the token value as follows:

```
${{secrets.FLY_API_TOKEN}}
```

If all goes well, your workflow log should look a bit like this:


simple_deployment_pipeline
succeeded 1 minute ago in 2m 33s

Run flyctl deploy --remote-only

```

Updating existing machines in 'pdex' with rolling strategy
> [1/2] Updating 48e5996b76eed8 [app]
> [1/2] Updating 48e5996b76eed8 [app]
> [1/2] Waiting for 48e5996b76eed8 [app] to have state: started
> [1/2] Machine 48e5996b76eed8 [app] has state: started
> [1/2] Checking that 48e5996b76eed8 [app] is up and running
> [1/2] Waiting for 48e5996b76eed8 [app] to become healthy: 1/1

✔ [1/2] Machine 48e5996b76eed8 [app] update succeeded
> [2/2] Updating 5683dd7db03228 [app]
> [2/2] Updating 5683dd7db03228 [app]
> [2/2] Waiting for 5683dd7db03228 [app] to have state: started
> [2/2] Machine 5683dd7db03228 [app] has state: started
> [2/2] Checking that 5683dd7db03228 [app] is up and running
> [2/2] Waiting for 5683dd7db03228 [app] to become healthy: 0/1

> [2/2] Waiting for 5683dd7db03228 [app] to become healthy: 1/1

✔ [2/2] Machine 5683dd7db03228 [app] update succeeded

Visit your newly deployed app at https://pdex.fly.dev/
```

Remember that it is always essential to keep an eye on what is happening in server logs when playing around with product deployments, so use `flyctl logs` early and use it often. No, use it all the time!

11.12 Health check

Each deployment in Fly.io creates a release. Releases can be checked from the command line:

```
$ flyctl releases
VERSION STATUS    DESCRIPTION USER           DATE
v18     complete  Release     mluukkai@iki.fi 16h56m ago
v17     complete  Release     mluukkai@iki.fi 17h3m ago
v16     complete  Release     mluukkai@iki.fi 21h22m ago
v15     complete  Release     mluukkai@iki.fi 21h25m ago
v14     complete  Release     mluukkai@iki.fi 21h34m ago
```

It is essential to ensure that a deployment ends up in a *succeeding* release, where the app is in healthy functional state. Fortunately, Fly.io has several configuration options that take care of the application health check.

If we change the app as follows, it fails to start:

```
app.listen(PORT, () => {
  this_causes_error
  // eslint-disable-next-line no-console
  console.log(`server started on port ${PORT}`)
})
```

In this case, the deployment fails:

```
$ flyctl releases
VERSION STATUS      DESCRIPTION USER            DATE
v19     failed      Release     mluukkai@iki.fi 3m52s ago
v18     complete    Release     mluukkai@iki.fi 16h56m ago
v17     complete    Release     mluukkai@iki.fi 17h3m ago
v16     complete    Release     mluukkai@iki.fi 21h22m ago
v15     complete    Release     mluukkai@iki.fi 21h25m ago
v14     complete    Release     mluukkai@iki.fi 21h34m ago
```

The app however stays up and running, Fly.io does not replace the functioning version (v18) with the broken one (v19).

Let us consider the following change

```
// start app in a wrong port
app.listen(PORT + 1, () => {
  // eslint-disable-next-line no-console
  console.log(`server started on port ${PORT}`)
})
```

Now the app starts but it is connected to the wrong port, so the service will not be functional. Fly.io thinks this is a successful deployment, so it deploys the app in a broken state.

One possibility to prevent broken deployments is to use an HTTP-level check defined in section http_service.http_checks. This type of check can be used to ensure that the app for real is in a functional state.

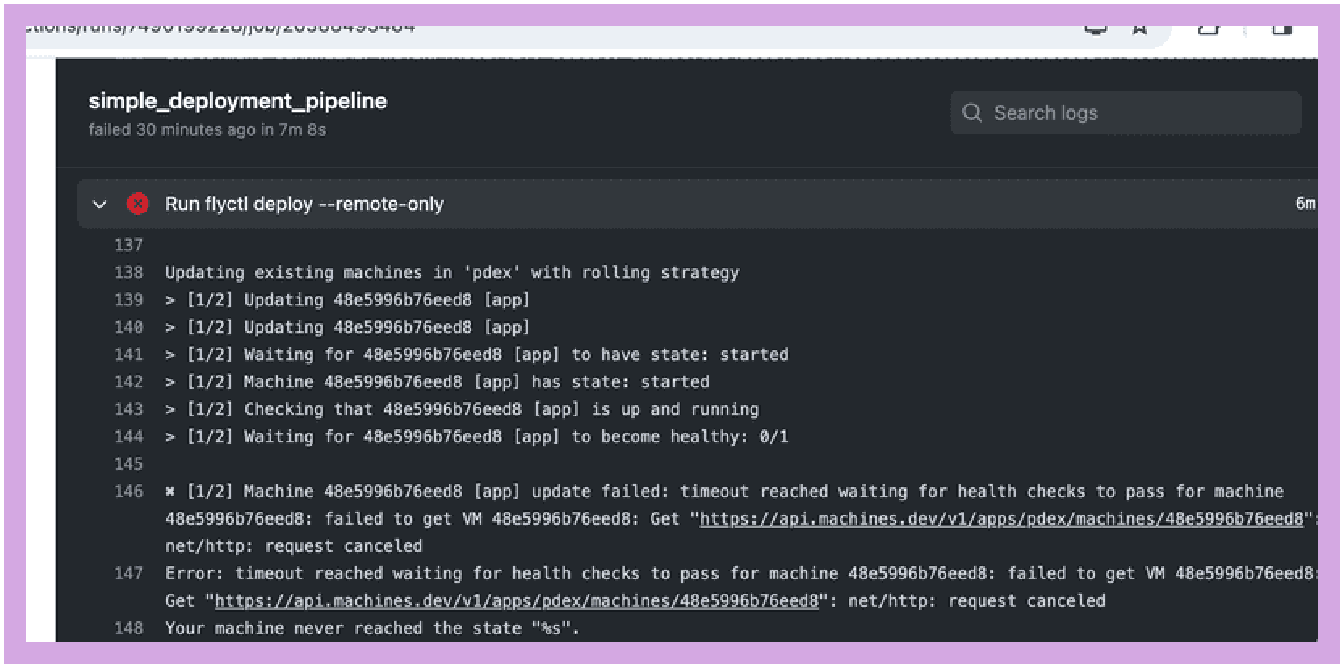
Add a simple endpoint for doing an application health check to the backend. You may e.g. copy this code:

```
app.get('/health', (req, res) => {
  res.send('ok')
})
```

Configure then an HTTP check that ensures the health of the deployments based on the HTTP request to the defined health check endpoint.

You also need to set the deployment strategy (in the file `fly.toml`) of the app to be *canary*. These strategies ensure that only an app with a healthy state gets deployed.

Ensure that GitHub Actions notices if a deployment breaks your application:



simple_deployment_pipeline
failed 30 minutes ago in 7m 8s

Run flyctl deploy --remote-only

```
Updating existing machines in 'pdex' with rolling strategy
> [1/2] Updating 48e5996b76eed8 [app]
> [1/2] Updating 48e5996b76eed8 [app]
> [1/2] Waiting for 48e5996b76eed8 [app] to have state: started
> [1/2] Machine 48e5996b76eed8 [app] has state: started
> [1/2] Checking that 48e5996b76eed8 [app] is up and running
> [1/2] Waiting for 48e5996b76eed8 [app] to become healthy: 0/1

✖ [1/2] Machine 48e5996b76eed8 [app] update failed: timeout reached waiting for health checks to pass for machine
48e5996b76eed8: failed to get VM 48e5996b76eed8: Get "https://api.machines.dev/v1/apps/pdex/machines/48e5996b76eed8"
net/http: request canceled
Error: timeout reached waiting for health checks to pass for machine 48e5996b76eed8: failed to get VM 48e5996b76eed8
Get "https://api.machines.dev/v1/apps/pdex/machines/48e5996b76eed8": net/http: request canceled
Your machine never reached the state "%s".
```

You may simulate this e.g. as follows:

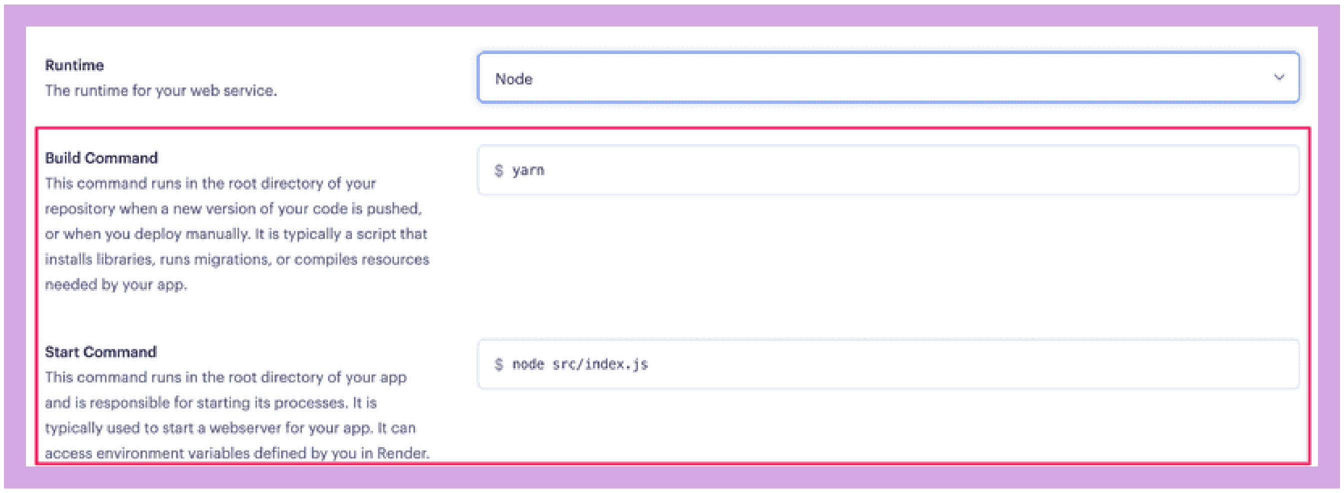
```
app.get('/health', (req, res) => {
  // eslint-disable-next-line no-constant-condition
  if (true) throw('error...  ')
  res.send('ok')
})
```

Exercises 11.10-11.12. (Render)

If you rather want to use other hosting options, there is an alternative set of exercises for Fly.io.

11.10 Deploying your application to Render

Set up your application in Render. The setup is now not quite as straightforward as in part 3. You have to carefully think about what should go to these settings:


Runtime
The runtime for your web service.
Node

Build Command
This command runs in the root directory of your repository when a new version of your code is pushed, or when you deploy manually. It is typically a script that installs libraries, runs migrations, or compiles resources needed by your app.
$ yarn

Start Command
This command runs in the root directory of your app and is responsible for starting its processes. It is typically used to start a webserver for your app. It can access environment variables defined by you in Render.
$ node src/index.js

If you need to run several commands in the build or start command, you may use a simple shell script for that.

Create eg. a file *build_step.sh* with the following content:

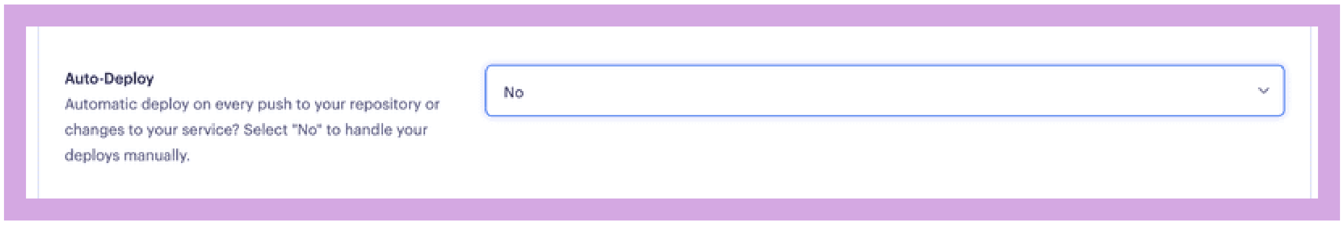
```
#!/bin/bash

echo "Build script"

# add the commands here
```

Give it execution permissions (Google or see e.g. this to find out how) and ensure that you can run it from the command line:

```
$ ./build_step.sh
Build script
```

Other option is to use a Pre deploy command, with that you may run one additional command before the deployment starts.

You also need to open the *Advanced settings* and turn the auto-deploy off since we want to control the deployment in the GitHub Actions:


Auto-Deploy
Automatic deploy on every push to your repository or changes to your service? Select "No" to handle your deploys manually.
No

Ensure now that you get the app up and running. Use the *Manual deploy*.

Most likely things will fail at the start, so remember to keep the *Logs* open all the time.

11.11 Automatic deployments

Next step is to automate the deployment. There are two options, a ready-made custom action or the use of the Render deploy hook.

Deployment with custom action

Go to GitHub Actions marketplace and search for action for our purposes. You might search with *render deploy*. There are several actions to choose from. You can pick any. Quite often the best choice is the one with the most stars. It is also a good idea to look if the action is actively maintained (time of the last release) and does it has many open issues or pull requests.

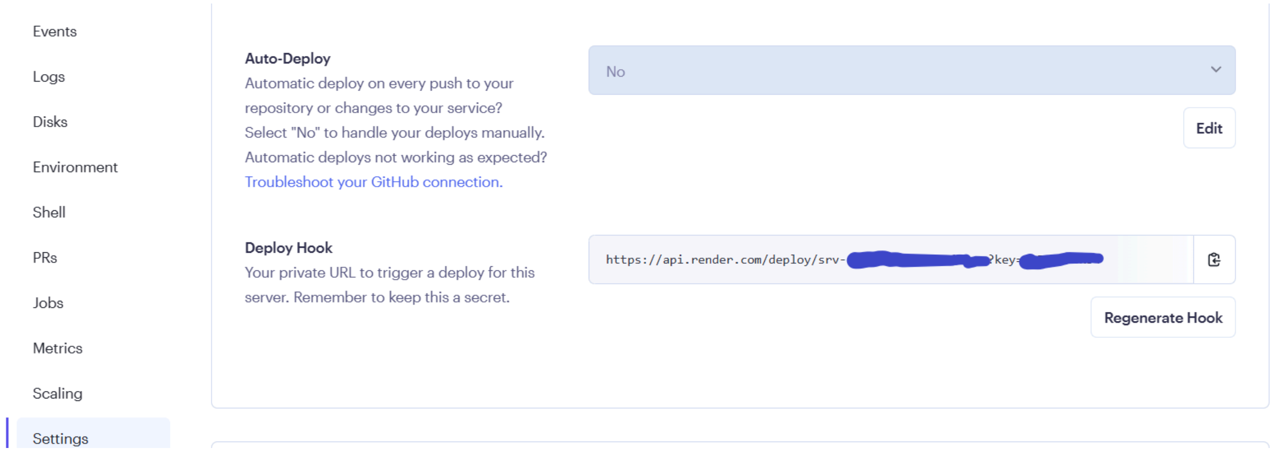
Warning: for some reason, the most starred option render-action was very unreliable when the part was updated (16th Jan 2024), so better avoid that. If you end up with too much problems, the deploy hook might be a better option!

Set up the action to your workflow and ensure that every commit that passes all the checks results in a new deployment. Note that you need Render API key and the app service id for the deployment. See here how the API key is generated. You can get the service id from the URL of the Render dashboard of your app. The end of the URL (starting with `srv-`) is the id:

```
https://dashboard.render.com/web/srv-randomcharachtershere
```

Deployment with deploy hook

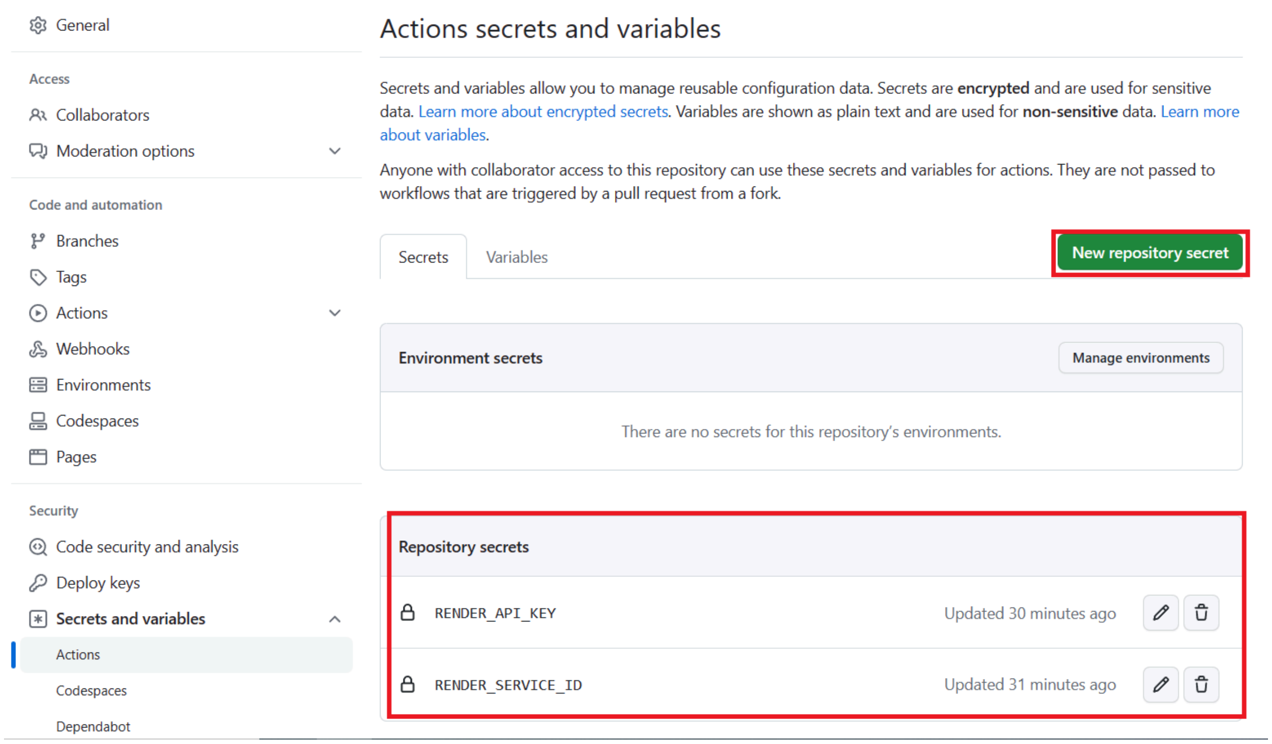
Alternative, and perhaps a more reliable option is to use Render Deploy Hook which is a private URL to trigger the deployment. You can get it from your app settings:



Auto-Deploy
Automatic deploy on every push to your repository or changes to your service? Select "No" to handle your deploys manually. Automatic deploys not working as expected? Troubleshoot your GitHub connection.
No

Deploy Hook
Your private URL to trigger a deploy for this server. Remember to keep this a secret.
`https://api.render.com/deploy/srv-[redacted]?key=[redacted]`

DON'T USE the plain URL in your pipeline. Instead create GitHub secrets for your key and service id:



Actions secrets and variables

Secrets and variables allow you to manage reusable configuration data. Secrets are **encrypted** and are used for sensitive data. Learn more about encrypted secrets. Variables are shown as plain text and are used for **non-sensitive** data. Learn more about variables.

Anyone with collaborator access to this repository can use these secrets and variables for actions. They are not passed to workflows that are triggered by a pull request from a fork.

Environment secrets

There are no secrets for this repository's environments.

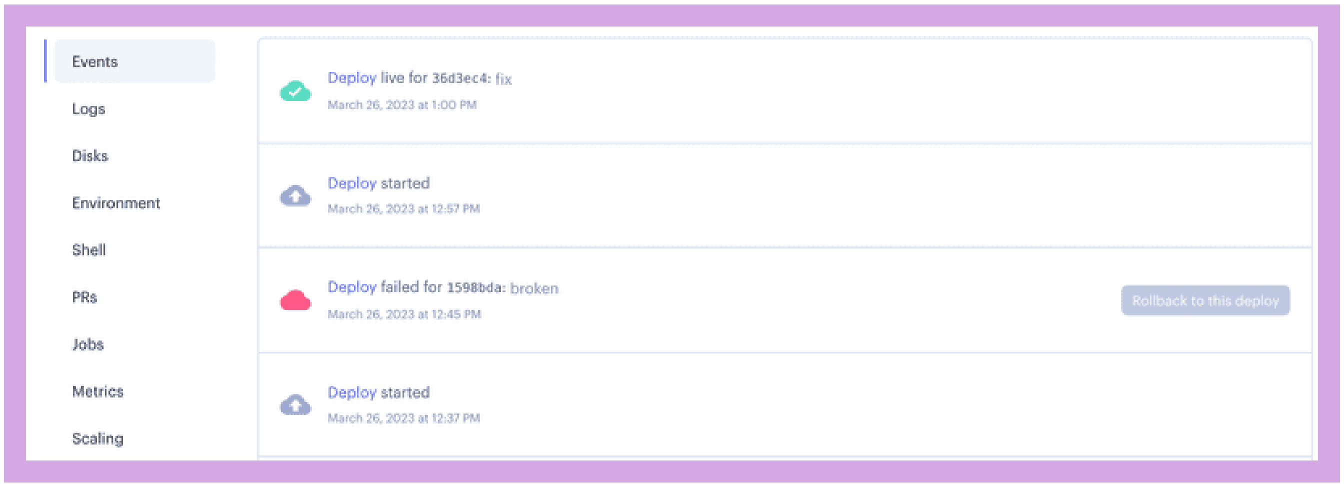
Repository secrets

RENDER_API_KEY	Updated 30 minutes ago
RENDER_SERVICE_ID	Updated 31 minutes ago

Then you can use them like this:

```
- name: Trigger deployment
  run: curl https://api.render.com/deploy/srv-${{ secrets.RENDER_SERVICE_ID }}?key=${{
secrets.RENDER_API_KEY }}
```

The deployment takes some time. See the events tab of the Render dashboard to see when the new deployment is ready:



Deploy live for 36d3ec4: fix
March 26, 2023 at 1:00 PM

Deploy started
March 26, 2023 at 12:57 PM

Deploy failed for 1598bda: broken
March 26, 2023 at 12:45 PM

Deploy started
March 26, 2023 at 12:37 PM

11.12 Health check

All tests pass and the new version of the app gets automatically deployed to Render so everything seems to be in order. But does the app really work? Besides the checks done in the deployment pipeline, it is extremely beneficial to have also some "application level" health checks ensuring that the app for real is in a functional state.

The zero downtime deploys in Render should ensure that your app stays functional all the time! For some reason, this property did not always work as promised when this part was updated (16th Jan 2024). The reason might be the use of a free account.

Add a simple endpoint for doing an application health check to the backend. You may e.g. copy this code:

```
app.get('/health', (req, res) => {
  res.send('ok')
})
```

Commit the code and push it to GitHub. Ensure that you can access the health check endpoint of your app.

Configure now a *Health Check Path* to your app. The configuration is done in the settings tab of the Render dashboard.

Make a change in your code, push it to GitHub, and ensure that the deployment succeeds.

Note that you can see the log of deployment by clicking the most recent deployment in the events tab.

When you are set up with the health check, simulate a broken deployment by changing the code as follows:

```
app.get('/health', (req, res) => {
  // eslint-disable-next-line no-constant-condition
  if (true) throw('error...  ')
  res.send('ok')
})
```

Push the code to GitHub and ensure that a broken version does not get deployed and the previous version of the app keeps running.

Before moving on, fix your deployment and ensure that the application works again as intended.

Propose changes to material

Part 11b
Previous part

Part 11d
Next part

HOUSTON

{() => fs}

Fullstack > Part 11 > Keeping green

d Keeping green

Your main branch of the code should always remain *green*. Being green means that all the steps of your build pipeline should complete successfully: the project should build successfully, tests should run without errors, and the linter shouldn't have anything to complain about, etc.

Why is this important? You will likely deploy your code to production specifically from your main branch. Any failures in the main branch would mean that new features cannot be deployed to production until the issue is sorted out. Sometimes you will discover a nasty bug in production that was not caught by the CI/CD pipeline. In these cases, you want to be able to roll the production environment back to a previous commit in a safe manner.

How do you keep your main branch green then? Avoid committing any changes directly to the main branch. Instead, commit your code on a branch based on the freshest possible version of the main branch. Once you think the branch is ready to be merged into the main you create a GitHub Pull Request (also referred to as PR).

Working with Pull Requests

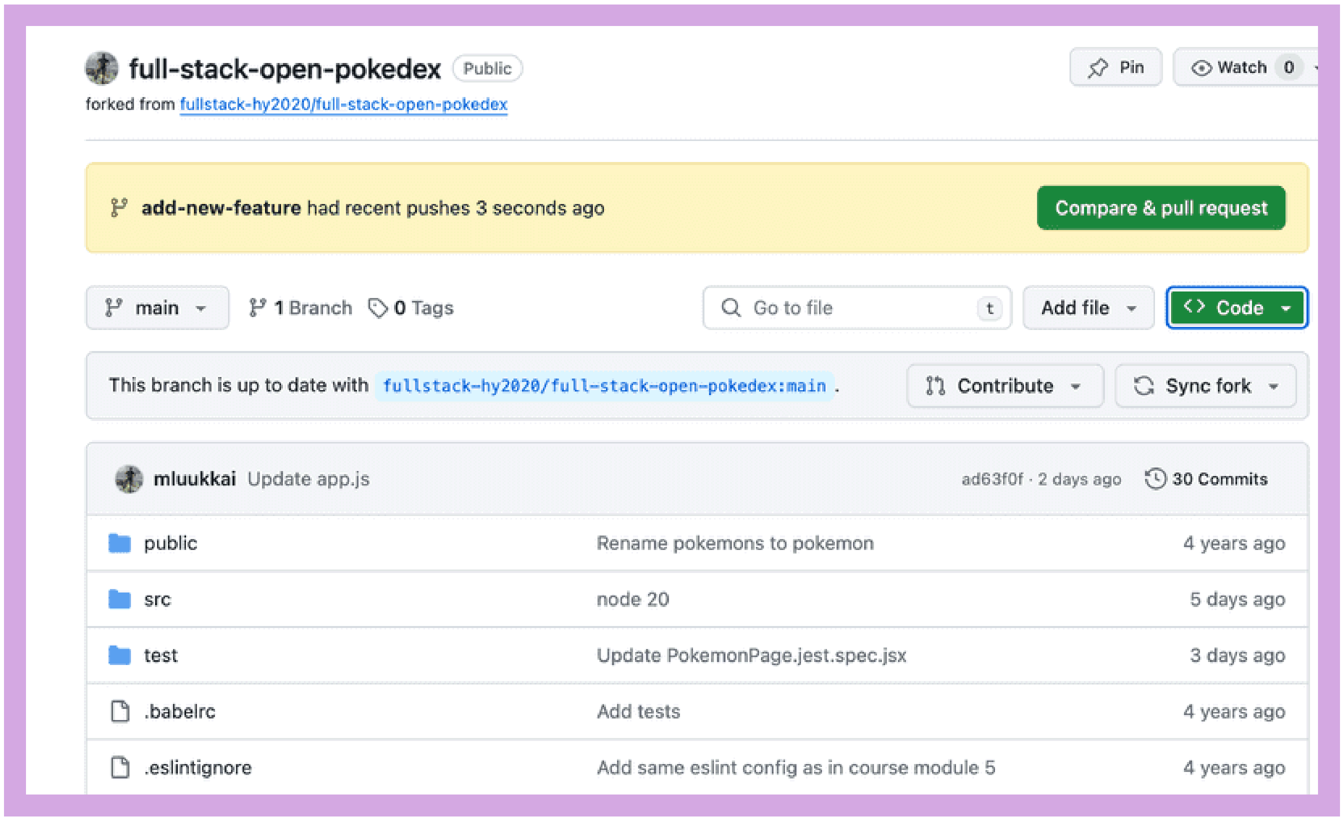
Pull requests are a core part of the collaboration process when working on any software project with at least two contributors. When making changes to a project you checkout a new branch locally, make and commit your changes, push the branch to the remote repository (in our case to GitHub) and create a pull request for someone to review your changes before those can be merged into the main branch.

There are several reasons why using pull requests and getting your code reviewed by at least one other person is always a good idea.

- Even a seasoned developer can often overlook some issues in their code: we all know of the tunnel vision effect.

- A reviewer can have a different perspective and offer a different point of view.
- After reading through your changes, at least one other developer will be familiar with the changes you've made.
- Using PRs allows you to automatically run all tasks in your CI pipeline before the code gets to the main branch. GitHub Actions provides a trigger for pull requests.

You can configure your GitHub repository in such a way that pull requests cannot be merged until they are approved.



forked from fullstack-hy2020/full-stack-open-pokedex

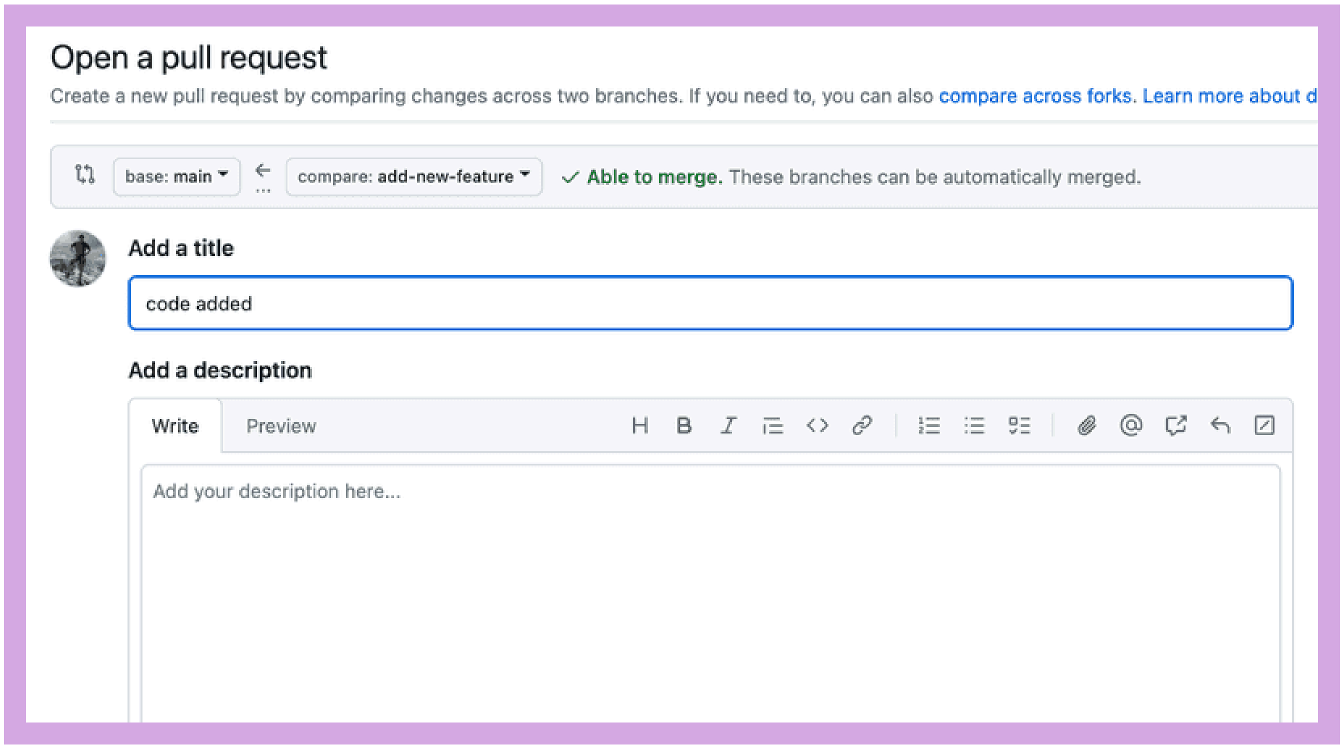
add-new-feature had recent pushes 3 seconds ago

This branch is up to date with `fullstack-hy2020/full-stack-open-pokedex:main` .

mluukkai Update app.js ad63f0f · 2 days ago

public	Rename pokemons to pokemon	4 years ago
src	node 20	5 days ago
test	Update PokemonPage.jest.spec.jsx	3 days ago
.babelrc	Add tests	4 years ago
.eslintignore	Add same eslint config as in course module 5	4 years ago

To open a new pull request, open your branch in GitHub and click on the green "Compare & pull request" button at the top. You will be presented with a form where you can fill in the pull request description.


Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also compare across forks. Learn more about d

base: main
compare: add-new-feature
✓ Able to merge. These branches can be automatically merged.
Add a title
code added
Add a description
Write
Preview
Add your description here...

GitHub's pull request interface presents a description and the discussion interface. At the bottom, it displays all the CI checks (in our case each of our Github Actions) that are configured to run for each PR and the statuses of these checks. A green board is what you aim for! You can click on Details of each check to view details and run logs.

All the workflows we looked at so far were triggered by commits to the main branch. To make the workflow run for each pull request we would have to update the trigger part of the workflow. We use the "pull_request" trigger for branch "main" (our main branch) and limit the trigger to events "opened" and "synchronize". Basically, this means, that the workflow will run when a PR into the main branch is opened or updated.

So let us change events that trigger of the workflow as follows:

```
on:
  push:
    branches:
      - main
  pull_request:
    branches: [main]
    types: [opened, synchronize]
```

We shall soon make it impossible to push the code directly to the main branch, but in the meantime, let us still run the workflow also for all the possible direct pushes to the main branch.

Exercises 11.13-11.14.

Our workflow is doing a nice job of ensuring good code quality, but since it is run on commits to the main branch, it's catching the problems too late!

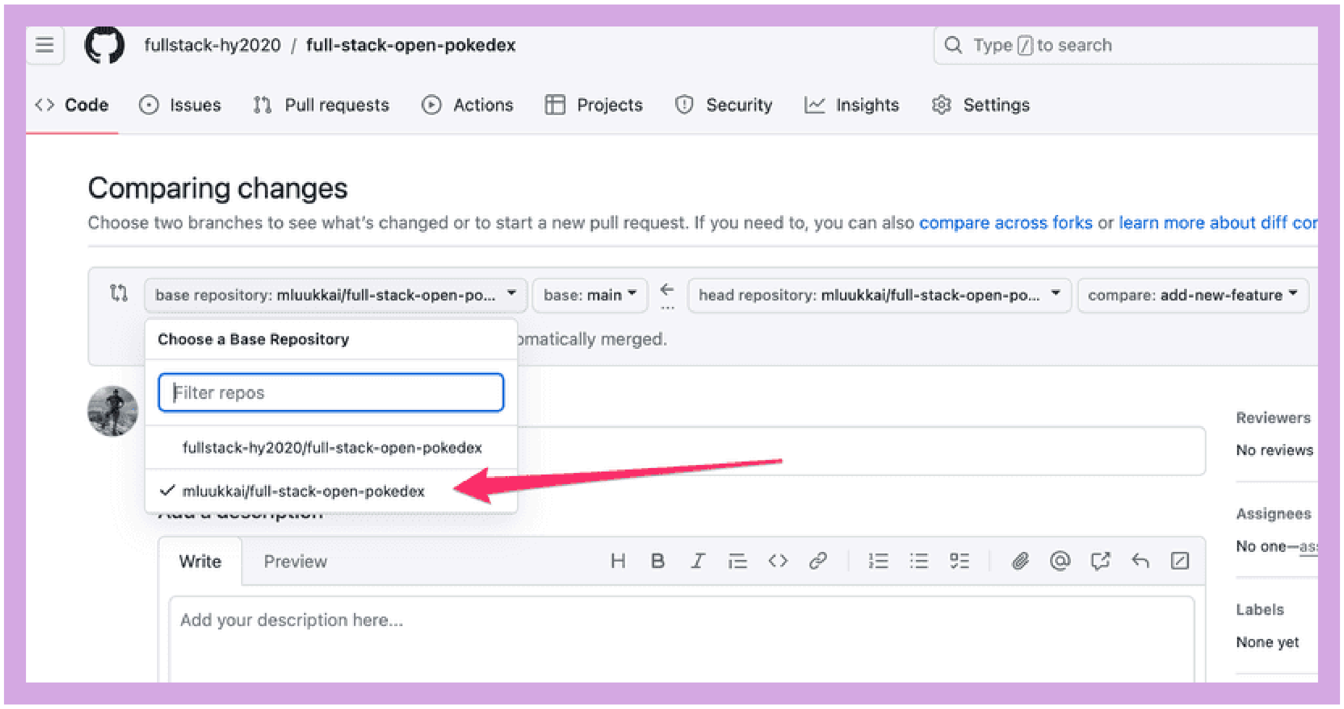
11.13 Pull request

Update the trigger of the existing workflow as suggested above to run on new pull requests to your main branch.

Create a new branch, commit your changes, and open a pull request to your main branch.

If you have not worked with branches before, check e.g. this tutorial to get started.

Note that when you open the pull request, make sure that you select here your *own* repository as the destination *base repository*. By default, the selection is the original repository by https://github.com/fullstack-hy2020 and you **do not want** to do that:



Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also compare across forks or learn more about diff cor

base repository: mluukkai/full-stack-open-po... base: main head repository: mluukkai/full-stack-open-po... compare: add-new-feature

Choose a Base Repository

Filter repos

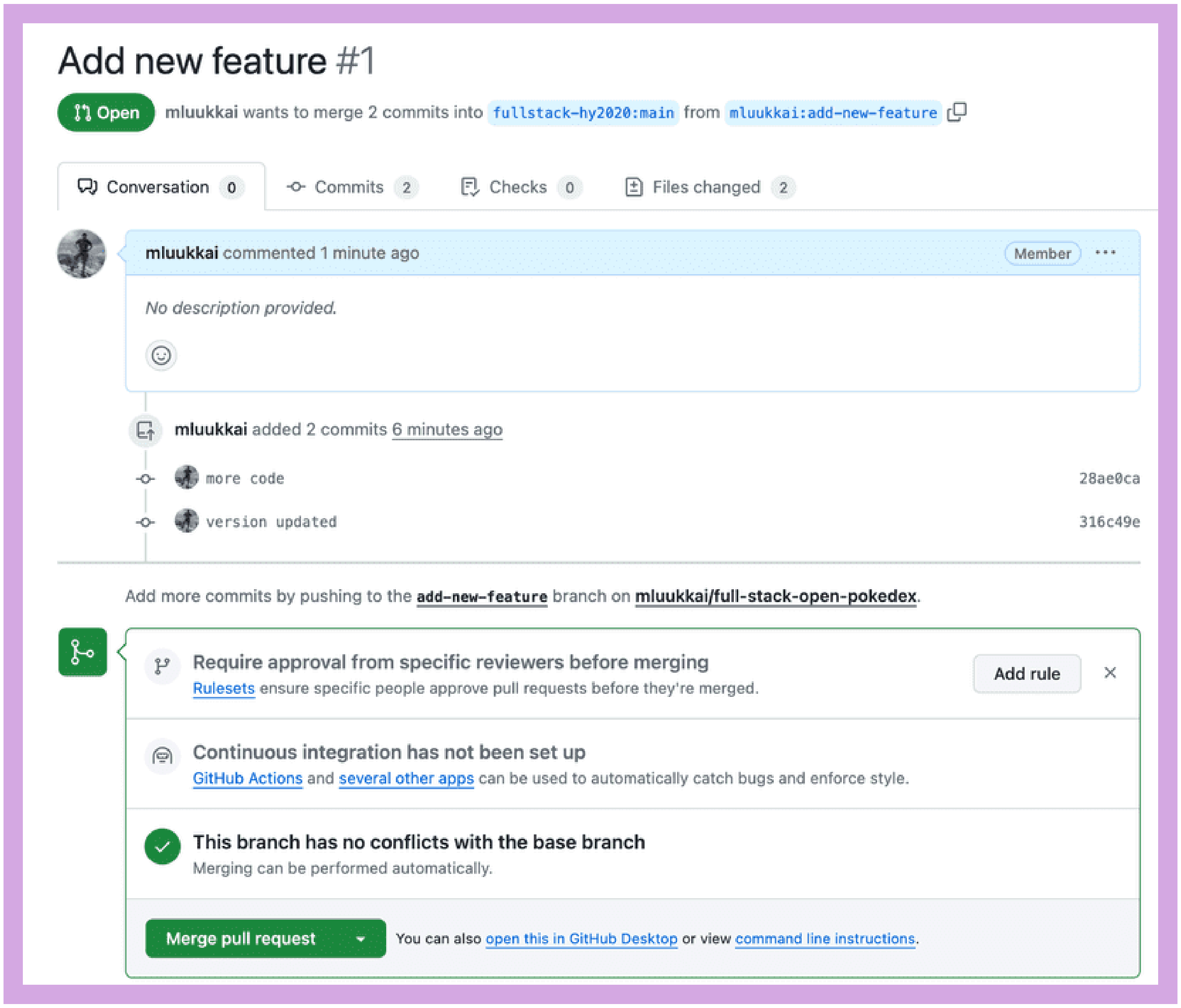
fullstack-hy2020/full-stack-open-pokedex

✓ mluukkai/full-stack-open-pokedex

omatically merged.

Add your description here...

In the "Conversation" tab of the pull request you should see your latest commit(s) and the yellow status for checks in progress:


Add new feature #1

Open mluukkai wants to merge 2 commits into fullstack-hy2020:main from mluukkai:add-new-feature

Conversation 0 Commits 2 Checks 0 Files changed 2

mluukkai commented 1 minute ago Member

No description provided.

mluukkai added 2 commits 6 minutes ago

more code 28ae0ca

version updated 316c49e

Add more commits by pushing to the add-new-feature branch on mluukkai/full-stack-open-pokedex.

Require approval from specific reviewers before merging
Rulesets ensure specific people approve pull requests before they're merged.

Add rule

Continuous integration has not been set up
GitHub Actions and several other apps can be used to automatically catch bugs and enforce style.

This branch has no conflicts with the base branch
Merging can be performed automatically.

Merge pull request

You can also open this in GitHub Desktop or view command line instructions.

Once the checks have been run, the status should turn to green. Make sure all the checks pass. Do not merge your branch yet, there's still one more thing we need to improve on our pipeline.

11.14 Run deployment step only for the main branch

All looks good, but there is actually a pretty serious problem with the current workflow. All the steps, including the deployment, are run also for pull requests. This is surely something we do not want!

Fortunately, there is an easy solution for the problem! We can add an `if` condition to the deployment step, which ensures that the step is executed only when the code is being merged or pushed to the main branch.

The workflow context gives various kinds of information about the code the workflow is run.

The relevant information is found in GitHub context, the field *event_name* tells us what is the "name" of the event that triggered the workflow. When a pull request is merged, the name of the event is somehow paradoxically *push*, the same event that happens when pushing the code to the repository. Thus, we get the desired behavior by adding the following condition to the step that deploys the code:

```
if: ${{ github.event_name == 'push' }}
```

Push some more code to your branch, and ensure that the deployment step *is not executed* anymore. Then merge the branch to the main branch and make sure that the deployment happens.

Versioning

The most important purpose of versioning is to uniquely identify the software we're running and the code associated with it.

The ordering of versions is also an important piece of information. For example, if the current release has broken critical functionality and we need to identify the *previous version* of the software so that we can roll back the release back to a stable state.

Semantic Versioning and Hash Versioning

How an application is versioned is sometimes called a versioning strategy. We'll look at and compare two such strategies.

The first one is semantic versioning, where a version is in the form `{major}.{minor}.{patch}`. For example, if the version is `1.2.3`, it has `1` as the major version, `2` is the minor version, and `3` is the patch version.

In general, changes that fix the functionality without changing how the application works from the outside are `patch` changes, changes that make small changes to functionality (as viewed from the outside) are `minor` changes and changes that completely change the application (or major functionality changes) are `major` changes. The definitions of each of these terms can vary from project to project.

For example, npm-libraries are following the semantic versioning. At the time of writing this text (16th March 2023) the most recent version of React is 18.2.0, so the major version is 18 and the minor version is 2.

Hash versioning (also sometimes known as SHA versioning) is quite different. The version "number" in hash versioning is a hash (that looks like a random string) derived from the contents of the repository and the changes introduced in the commit that created the version. In Git, this is already done for you as the commit hash that is unique for any change set.

Hash versioning is almost always used in conjunction with automation. It's a pain (and error-prone) to copy 32 character long version numbers around to make sure that everything is correctly deployed.

But what does the version point to?

Determining what code belongs to a given version is important and the way this is achieved is again quite different between semantic and hash versioning. In hash versioning (at least in Git) it's as simple as looking up the commit based on the hash. This will let us know exactly what code is deployed with a specific version.

It's a little more complicated when using semantic versioning and there are several ways to approach the problem. These boil down to three possible approaches: something in the code itself, something in the repo or repo metadata, something completely outside the repo.

While we won't cover the last option on the list (since that's a rabbit hole all on its own), it's worth mentioning that this can be as simple as a spreadsheet that lists the Semantic Version and the commit it points to.

For the two repository based approaches, the approach with something in the code usually boils down to a version number in a file and the repo/metadata approach usually relies on tags or (in the case of GitHub) releases. In the case of tags or releases, this is relatively simple, the tag or release points to a commit, the code in that commit is the code in the release.

Version order

In semantic versioning, even if we have version bumps of different types (major, minor, or patch) it's still quite easy to put the releases in order: 1.3.7 comes before 2.0.0 which itself comes before 2.1.5 which comes before 2.2.0. A list of releases (conveniently provided by a package manager or GitHub) is still needed to know what the last version is but it's easier to look at that list and discuss it: It's easier to say "We need to roll back to 3.2.4" than to try communicate a hash in person.

That's not to say that hashes are inconvenient: if you know which commit caused the particular problem, it's easy enough to look back through a Git history and get the hash of the previous commit. But if you have two hashes, say `d052aa41edfb4a7671c974c5901f4abe1c2db071` and `12c6f6738a18154cb1cef7cf0607a681f72eaff3` , you really can not say which came earlier in history, you need something more, such as the Git log that reveals the ordering.

Comparing the Two

We've already touched on some of the advantages and disadvantages of the two versioning methods discussed above but it's perhaps useful to address where they'd each likely be used.

Semantic Versioning works well when deploying services where the version number could be of significance or might actually be looked at. As an example, think of the JavaScript libraries that you're using. If you're using version 3.4.6 of a particular library, and there's an update available to 3.4.8, if the library uses semantic versioning, you could (hopefully) safely assume that you're ok to upgrade without breaking anything. If the version jumps to 4.0.1 then maybe it's not such a safe upgrade.

Hash versioning is very useful where most commits are being built into artifacts (e.g. runnable binaries or Docker images) that are themselves uploaded or stored. As an example, if your testing requires building your package into an artifact, uploading it to a server, and running tests against it, it would be convenient to have hash versioning as it would prevent accidents.

As an example think that you're working on version 3.2.2 and you have a failing test, you fix the failure and push the commit but as you're working in your branch, you're not going to update the version number. Without hash versioning, the artifact name may not change. If there's an error in uploading the artifact, maybe the tests run again with the older artifact (since it's still there and has the same name) and you get the wrong test results. If the artifact is versioned with the hash, then the version number `must` change on every commit and this means that if the upload fails, there will be an error since the artifact you told the tests to run against does not exist.

Having an error happen when something goes wrong is almost always preferable to having a problem silently ignored in CI.

Best of Both Worlds

From the comparison above, it would seem that the semantic versioning makes sense for releasing software while hash-based versioning (or artifact naming) makes more sense during development. This doesn't necessarily cause a conflict.

Think of it this way: versioning boils down to a technique that points to a specific commit and says "We'll give this point a name, it's name will be 3.5.5". Nothing is preventing us from also referring to the same commit by its hash.

There is a catch. We discussed at the beginning of this part that we always have to know exactly what is happening with our code, for example, we need to be sure that we have tested the code we want to deploy. Having two parallel versioning (or naming) conventions can make this a little more difficult.

For example, when we have a project that uses hash-based artifact builds for testing, it's always possible to track the result of every build, lint, and test to a specific commit and developers know the state their code is in. This is all automated and transparent to the developers. They never need to be aware of the fact that the CI system is using the commit hash underneath to name build and test artifacts. When the developers merge their code to the main branch, again the CI takes over. This time, it will build and test all the code and give it a semantic version number all in one go. It attaches the version number to the relevant commit with a Git tag.

In the case above, the software we release is tested because the CI system makes sure that tests are run on the code it is about to tag. It would not be incorrect to say that the project uses semantic versioning and simply ignore that the CI system tests individual developer branches/PRs with a hash-based naming system. We do this because the version we care about (the one that is released) is given a semantic version.

Exercises 11.15-11.16.

Let's extend our workflow so that it will automatically increase (bump) the version when a pull request is merged into the main branch and tag the release with the version number. We will use an open source action developed by a third party: anothrNick/github-tag-action.

11.15 Adding versioning

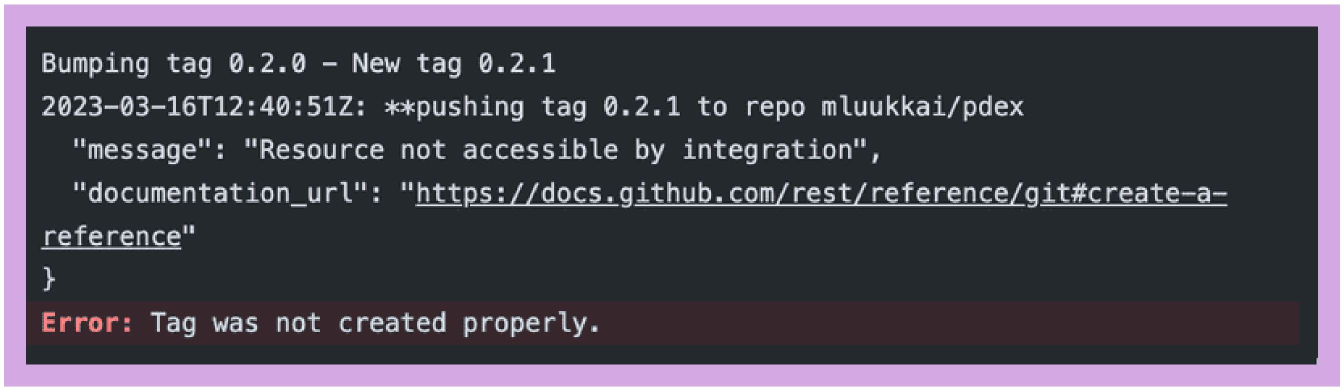
We will extend our workflow with one more step:

```yaml
- name: Bump version and push tag
  uses: anothrNick/github-tag-action@1.64.0
  env:
    GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
```

Note: you should use the most recent version of the action, see here if a more recent version is available.

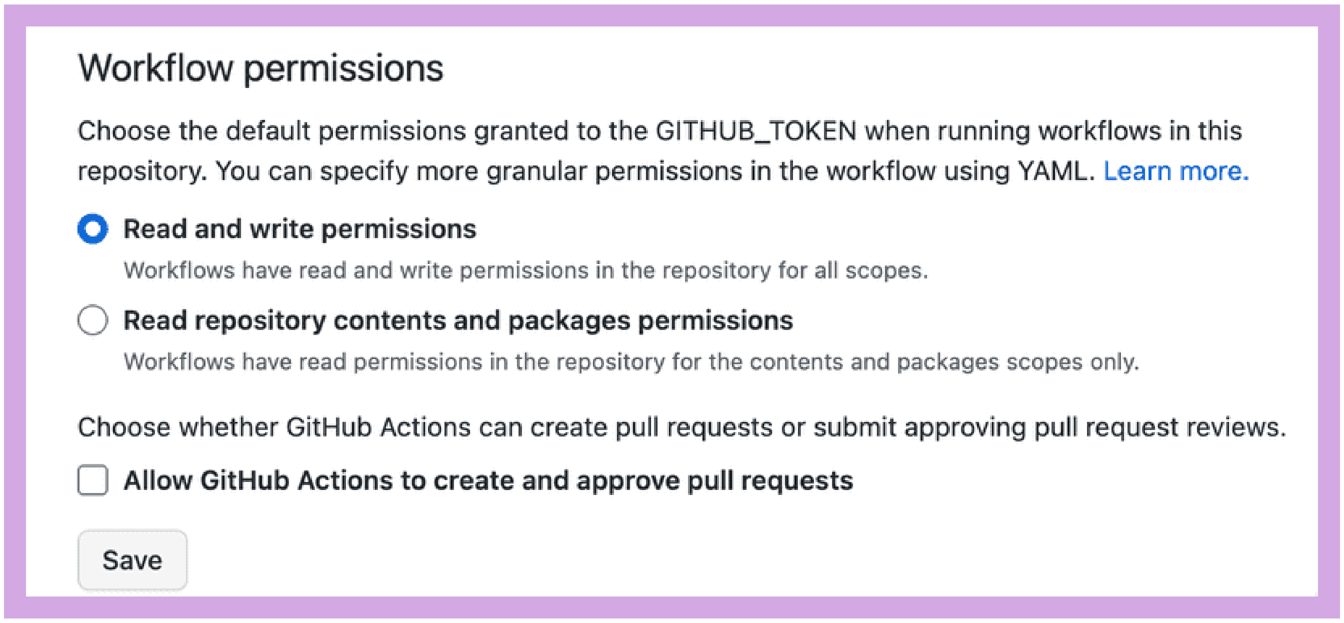
We're passing an environmental variable `secrets.GITHUB_TOKEN` to the action. As it is third party action, it needs the token for authentication in your repository. You can read more here about authentication in GitHub Actions.

You may end up having this error message



```
Bumping tag 0.2.0 - New tag 0.2.1
2023-03-16T12:40:51Z: **pushing tag 0.2.1 to repo mluukkai/pdex
  "message": "Resource not accessible by integration",
  "documentation_url": "https://docs.github.com/rest/reference/git#create-a-reference"
}
Error: Tag was not created properly.
```

The most likely cause for this is that your token has no write access to your repo. Go to your repository settings, select actions/general, and ensure that your token has *read and write permissions*:



Workflow permissions

Choose the default permissions granted to the GITHUB_TOKEN when running workflows in this repository. You can specify more granular permissions in the workflow using YAML. Learn more.

[✓ Read and write permissions]
Workflows have read and write permissions in the repository for all scopes.

Read repository contents and packages permissions
Workflows have read permissions in the repository for the contents and packages scopes only.

Choose whether GitHub Actions can create pull requests or submit approving pull request reviews.

☐ Allow GitHub Actions to create and approve pull requests

The anothrNick/github-tag-action action accepts some environment variables that modify the way the action tags your releases. You can look at these in the README and see what suits your needs.

As you can see from the documentation by default your releases will receive a `minor` bump, meaning that the middle number will be incremented.

Modify the configuration above so that each new version is by default a `patch` bump in the version number, so that by default, the last number is increased.

Remember that we want only to bump the version when the change happens to the main branch! So add a similar `if` condition to prevent version bumps on pull request as was done in Exercise 11.14 to prevent deployment on pull request related events.

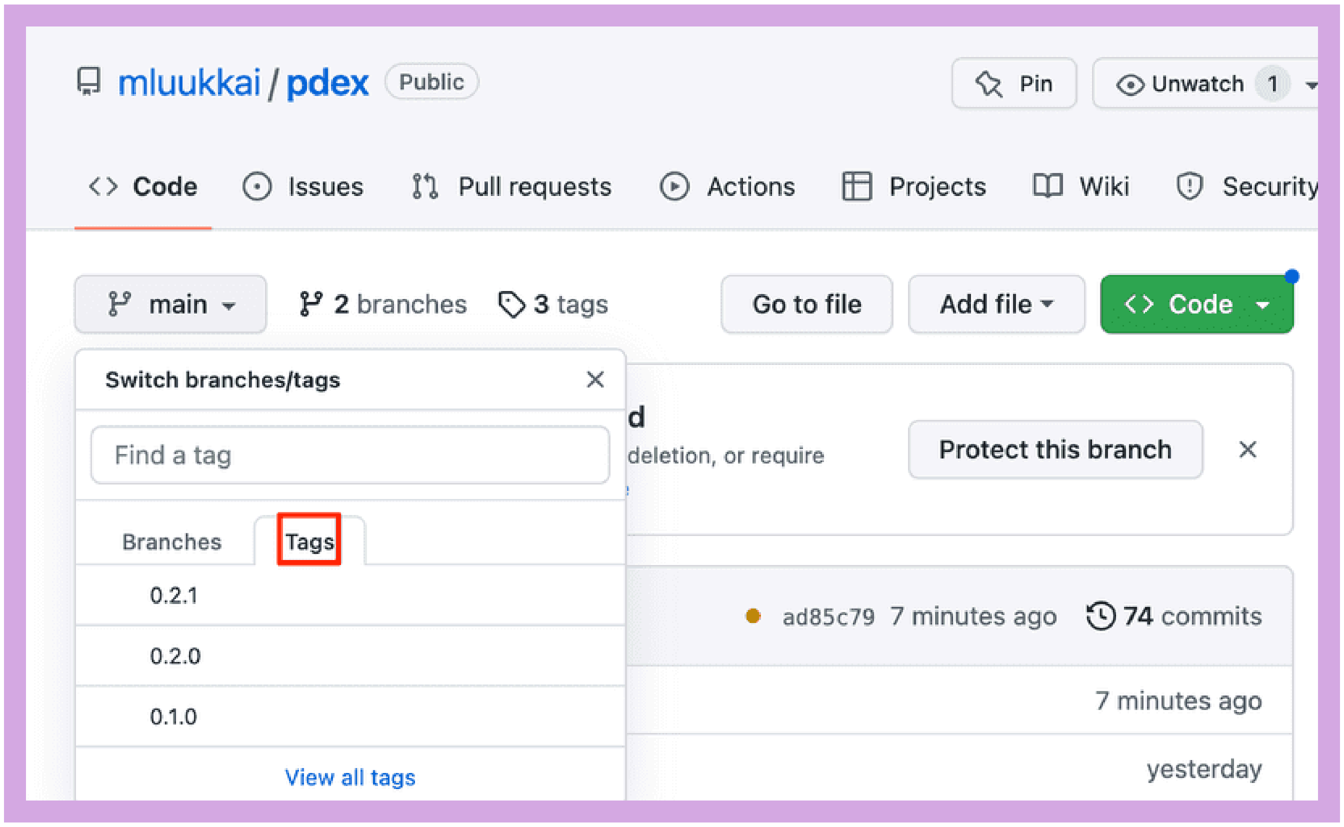
Complete now the workflow. Do not just add it as another step, but configure it as a separate job that depends on the job that takes care of linting, testing and deployment. So change your workflow definition as follows:

```
name: Deployment pipeline

on:
  push:
    branches:
      - main
  pull_request:
    branches: [main]
    types: [opened, synchronize]

jobs:
  simple_deployment_pipeline:
    runs-on: ubuntu-20.04
    steps:
      // steps here
  tag_release:
    needs: [simple_deployment_pipeline]
    runs-on: ubuntu-20.04
    steps:
      // steps here
```

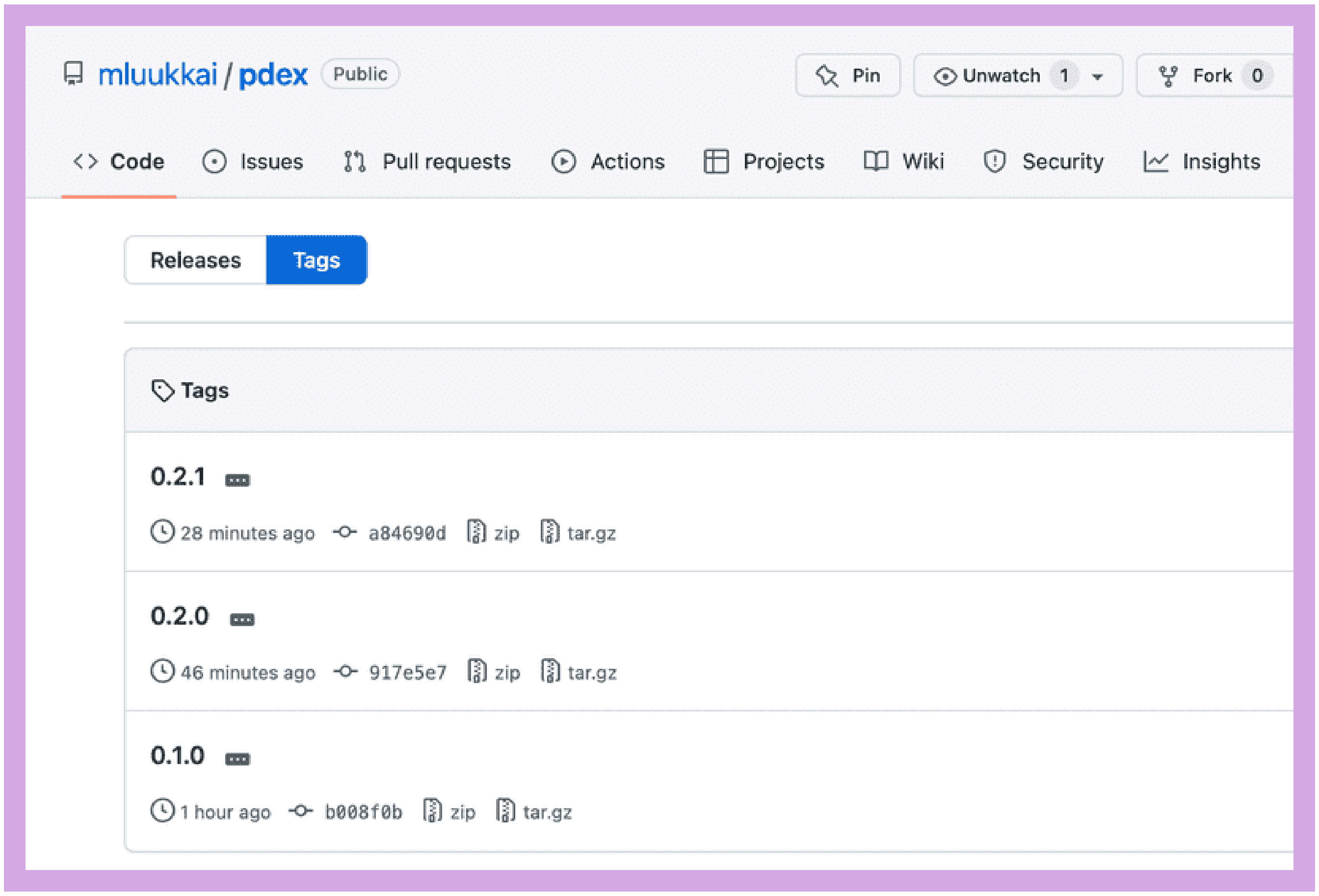
As was mentioned earlier jobs of a workflow are executed in parallel but since we want the linting, testing and deployment to be done first, we set a dependency that the *tag_release* waits the another job to execute first since we do not want to tag the release unless it passes tests and is deployed.

If you're uncertain of the configuration, you can set `DRY_RUN` to `true`, which will make the action output the next version number without creating or tagging the release!

Once the workflow runs successfully, the repository mentions that there are some *tags*:



By clicking *view all tags*, you can see all the tags listed:



If needed, you can navigate to the view of a single tag that shows eg. what is the GitHub commit corresponding to the tag.

11.16 Skipping a commit for tagging and deployment

In general, the more often you deploy the main branch to production, the better. However, there might be some valid reasons sometimes to skip a particular commit or a merged pull request to become tagged and released to production.

Modify your setup so that if a commit message in a pull request contains `#skip`, the merge will not be deployed to production and it is not tagged with a version number.

Hints:

The easiest way to implement this is to alter the `if` conditions of the relevant steps. Similarly to exercise 11-14 you can get the relevant information from the GitHub context of the workflow.

You might take this as a starting point:

```yaml
name: Testing stuff

on:
  push:
    branches:
      - main

jobs:
  a_test_job:
    runs-on: ubuntu-20.04
    steps:
      - uses: actions/checkout@v4
      - name: github context
        env:
          GITHUB_CONTEXT: ${{ toJson(github) }}
        run: echo "$GITHUB_CONTEXT"
      - name: commits
        env:
          COMMITS: ${{ toJson(github.event.commits) }}
        run: echo "$COMMITS"
      - name: commit messages
        env:
          COMMIT_MESSAGES: ${{ toJson(github.event.commits.*.message) }}
        run: echo "$COMMIT_MESSAGES"
```

See what gets printed in the workflow log!

Note that you can access the commits and commit messages *only when pushing or merging to the main branch*, so for pull requests the `github.event.commits` is empty. It is anyway not needed, since we want to skip the step altogether for pull requests.

You most likely need functions contains and join for your if condition.

Developing workflows is not easy, and quite often the only option is trial and error. It might actually be advisable to have a separate repository for getting the configuration right, and when it is done, to copy the right configurations to the actual repository.

It would also be possible to install a tool such as act that makes it possible to run your workflows locally. Unless you end up using more involved use cases like creating your own custom actions, going through the burden of setting up a tool such as act is most likely not worth the trouble.

A note about using third-party actions

When using a third-party action such that *github-tag-action* it might be a good idea to specify the used version with hash instead of using a version number. The reason for this is that the version number, that is implemented with a Git tag can in principle be *moved*. So today's version 1.61.0 might be a different code that is at next week the version 1.61.0!

However, the code in a commit with a particular hash does not change in any circumstances, so if we want to be 100% sure about the code we use, it is safest to use the hash.

Version 1.61.0 of the action corresponds to a commit with hash `8c8163ef62cf9c4677c8e800f36270af27930f42`, so we might want to change our configuration as follows:

```yaml
    - name: Bump version and push tag
      uses: anothrNick/github-tag-action@8c8163ef62cf9c4677c8e800f36270af27930f42
      env:
        GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
```

When we use actions provided by GitHub we trust them not to mess with version tags and to thoroughly test their code.

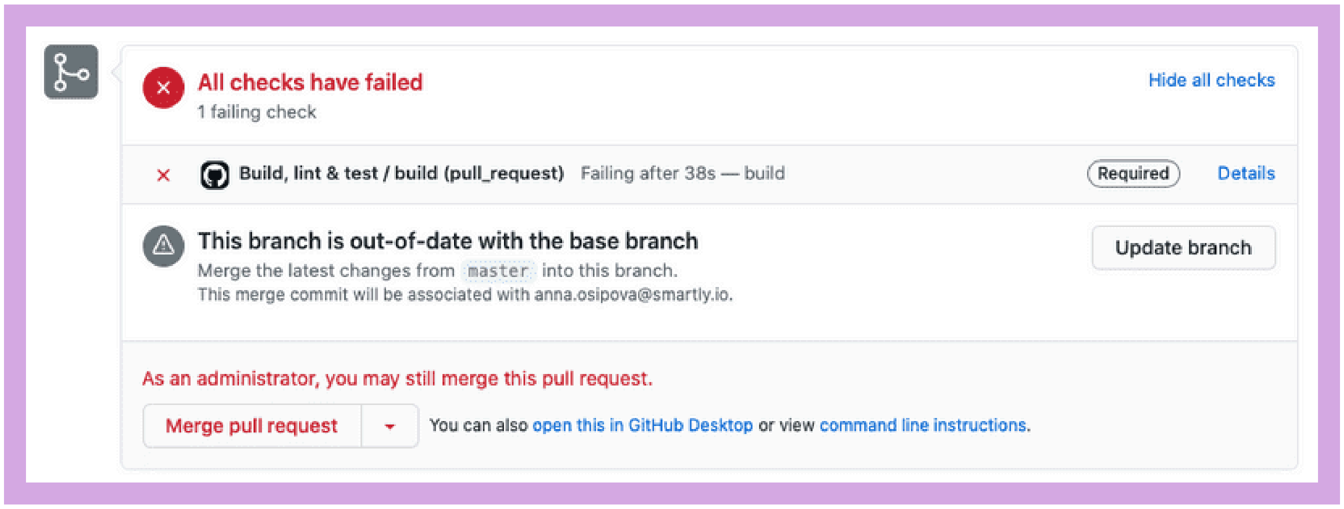
In the case of third-party actions, the code might end up being buggy or even malicious. Even when the author of the open-source code does not have the intention of doing something bad, they might end up leaving their credentials on a post-it note in a cafe, and then who knows what might happen.

By pointing to the hash of a specific commit we can be sure that the code we use when running the workflow will not change because changing the underlying commit and its contents would also change the hash.

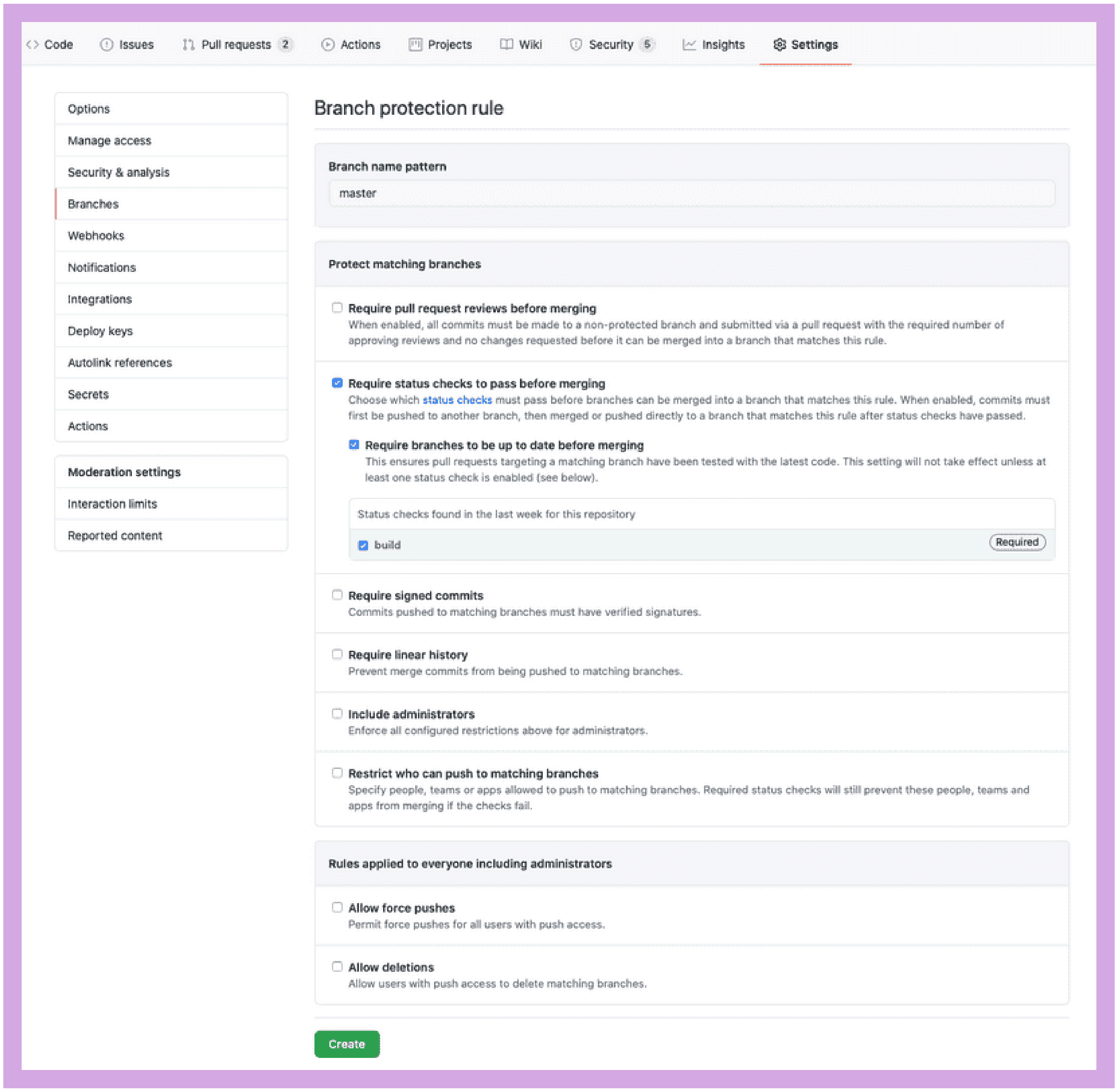
Keep the main branch protected

GitHub allows you to set up protected branches. It is important to protect your most important branch that should never be broken: *main*. In repository settings, you can choose between several levels of protection. We will not go over all of the protection options, you can learn more about them in GitHub documentation. Requiring pull request approval when merging into the main branch is one of the options we mentioned earlier.

From CI point of view, the most important protection is requiring status checks to pass before a PR can be merged into the main branch. This means that if you have set up GitHub Actions to run e.g. linting and testing tasks, then until all the lint errors are fixed and all the tests pass the PR cannot be merged. Because you are the administrator of your repository, you will see an option to override the restriction. However, non-administrators will not have this option.


All checks have failed
1 failing check
Build, lint & test / build (pull_request) Failing after 38s — build
This branch is out-of-date with the base branch
Merge the latest changes from master into this branch.
This merge commit will be associated with anna.osipova@smartly.io.
As an administrator, you may still merge this pull request.
You can also open this in GitHub Desktop or view command line instructions.

To set up protection for your main branch, navigate to repository "Settings" from the top menu inside the repository. In the left-side menu select "Branches". Click "Add rule" button next to "Branch protection rules". Type a branch name pattern ("main" will do nicely) and select the protection you would want to set up. At least "Require status checks to pass before merging" is necessary for you to fully utilize the power of GitHub Actions. Under it, you should also check "Require branches to be up to date before merging" and select all of the status checks that should pass before a PR can be merged.



Branch protection rule

Branch name pattern

master

Protect matching branches

- [ ] Require pull request reviews before merging
  When enabled, all commits must be made to a non-protected branch and submitted via a pull request with the required number of approving reviews and no changes requested before it can be merged into a branch that matches this rule.
- [x] Require status checks to pass before merging
  Choose which status checks must pass before branches can be merged into a branch that matches this rule. When enabled, commits must first be pushed to another branch, then merged or pushed directly to a branch that matches this rule after status checks have passed.
  - [x] Require branches to be up to date before merging
    This ensures pull requests targeting a matching branch have been tested with the latest code. This setting will not take effect unless at least one status check is enabled (see below).

  Status checks found in the last week for this repository
  - [x] build Required
- [ ] Require signed commits
  Commits pushed to matching branches must have verified signatures.
- [ ] Require linear history
  Prevent merge commits from being pushed to matching branches.
- [ ] Include administrators
  Enforce all configured restrictions above for administrators.
- [ ] Restrict who can push to matching branches
  Specify people, teams or apps allowed to push to matching branches. Required status checks will still prevent these people, teams and apps from merging if the checks fail.

Rules applied to everyone including administrators

- [ ] Allow force pushes
  Permit force pushes for all users with push access.
- [ ] Allow deletions
  Allow users with push access to delete matching branches.

Exercise 11.17

11.17 Adding protection to your main branch

Add protection to your *main* branch.

You should protect it to:

- Require all pull request to be approved before merging
- Require all status checks to pass before merging

Propose changes to material

Part 11c
Previous part

Part 11e
Next part

About course

Course contents

FAQ

Partners

Challenge

UNIVERSITY OF HELSINKI

HOUSTON

{() => fs}

Fullstack > Part 11 > Expanding Further

e Expanding Further

This part focuses on building a simple, effective, and robust CI system that helps developers to work together, maintain code quality, and deploy safely. What more could one possibly want? In the real world, there are more fingers in the pie than just developers and users. Even if that weren't true, even for developers, there's a lot more value to be gained from CI systems than just the things above.

Visibility and Understanding

In all but the smallest companies, decisions on what to develop are not made exclusively by developers. The term 'stakeholder' is often used to refer to people, both inside and outside the development team, who may have some interest in keeping an eye on the progress of the development. To this end, there are often integrations between Git and whatever project management/bug tracking software the team is using.

A common use of this is to have some reference to the tracking system in Git pull requests or commits. This way, for example, when you're working on issue number 123, you might name your pull request `BUG-123: Fix user copy issue` and the bug tracking system would notice the first part of the PR name and automatically move the issue to `Done` when the PR is merged.

Notifications

When the CI process finishes quickly, it can be convenient to just watch it execute and wait for the result. As projects become more complex, so too does the process of building and testing the code. This can quickly lead to a situation where it takes long enough to generate the build result that a developer may want to begin working on another task. This in turn leads to a forgotten build.

This is especially problematic if we're talking about merging PRs that may affect another developer's work, either causing problems or delays for them. This can also lead to a situation where you think you've deployed something but haven't actually finished a deployment, this can lead to miscommunication with teammates and customers (e.g. "Go ahead and try that again, the bug should be fixed").

There are several solutions to this problem ranging from simple notifications to more complicated processes that simply merge passing code if certain conditions are met. We're going to discuss notifications as a simple solution since it's the one that interferes with the team workflow the least.

By default, GitHub Actions sends an email on a build failure. This can be changed to send notifications regardless of build status and can also be configured to alert you on the GitHub web interface. Great. But what if we want more. What if for whatever reason this doesn't work for our use case.

There are integrations for example to various messaging applications such as Slack or Discord, to send notifications. These integrations still decide what to send and when to send it based on logic from GitHub.

Exercise 11.18

We have set up a channel *fullstack_webhook* to the course Discord group at https://study.cs.helsinki.fi/discord/join/fullstack for testing a messaging integration.

Register now to Discord if you have not already done that. You will also need a *Discord webhook* in this exercise. You find the webhook in the pinned message of the channel *fullstack_webhook*. Please do not commit the webhook to GitHub!

11.18 Build success/failure notification action

You can find quite a few third-party actions from GitHub Action Marketplace by using the search phrase discord. Pick one for this exercise. My choice was discord-webhook-notify since it has quite many stars and decent documentation.

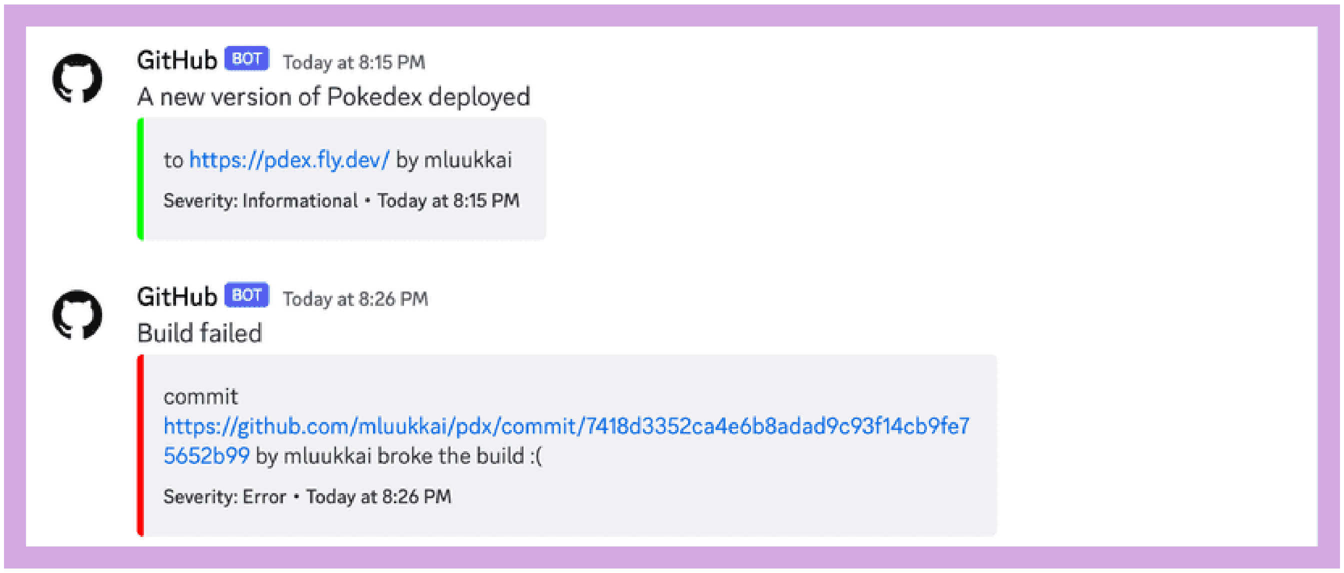
Setup the action so that it gives two types of notifications:

- A success indication if a new version gets deployed
- An error indication if a build fails

In the case of an error, the notification should be a bit more verbose to help developers find quickly which is the commit that caused it.

See here how to check the job status!

Your notifications may look like the following:


GitHub BOT Today at 8:15 PM
A new version of Pokedex deployed

to https://pdex.fly.dev/ by mluukkai
Severity: Informational • Today at 8:15 PM

GitHub BOT Today at 8:26 PM
Build failed

commit https://github.com/mluukkai/pdx/commit/7418d3352ca4e6b8adad9c93f14cb9fe75652b99 by mluukkai broke the build :(
Severity: Error • Today at 8:26 PM

Metrics

In the previous section, we mentioned that as projects get more complicated, so too, do their builds, and the duration of the builds increases. That's obviously not ideal: The longer the feedback loop, the slower the development.

While there are things that can be done about this increase in build times, it's useful to have a better view of the overall picture. It's useful to know how long a build took a few months ago versus how long it takes now. Was the progression linear or did it suddenly jump? Knowing what caused the increase in build time can be very useful in helping to solve it. If the build time increased linearly from 5 minutes to 10 minutes over the last year, maybe we can expect it to take another few months to get to 15 minutes and we have an idea of how much value there is in spending time speeding up the CI process.

Metrics can either be self-reported (also called 'push' metrics, where each build reports how long it took) or the data can be fetched from the API afterward (sometimes called 'pull' metrics). The risk with self-reporting is that the self-reporting itself takes time and may have a significant impact on "total time taken for all builds".

This data can be sent to a time-series database or to an archive of another type. There are plenty of cloud services where you can easily aggregate the metrics, one good option is Datadog.

Periodic tasks

There are often periodic tasks that need to be done in a software development team. Some of these can be automated with commonly available tools and some you will need to automate yourself.

The former category includes things like checking packages for security vulnerabilities. Several tools can already do this for you. Some of these tools would even be free for certain types (e.g. open source) projects. GitHub provides one such tool, Dependabot.

Words of advice to consider: If your budget allows it, it's almost always better to use a tool that already does the job than to roll your own solution. If security isn't the industry you're aiming for, for example, use Dependabot to check for security vulnerabilities instead of making your own tool.

What about the tasks that don't have a tool? You can automate these yourself with GitHub Actions too. GitHub Actions provides a scheduled trigger that can be used to execute a task at a particular time.

Exercises 11.19-11.21

11.19 Periodic health check

We are pretty confident now that our pipeline prevents bad code from being deployed. However, there are many sources of errors. If our application would e.g. depend on a database that would for some reason become unavailable, our application would most likely crash. That's why it would be a good idea to set up *a periodic health check* that would regularly do an HTTP GET request to our server. We quite often refer to this kind of request as a *ping*.

It is possible to schedule GitHub actions to happen regularly.

Use now the action url-health-check or any other alternative and schedule a periodic health check ping to your deployed software. Try to simulate a situation where your application breaks down and ensure that the check detects the problem. Write this periodic workflow to an own file.

Note that unfortunately it takes quite long until GitHub Actions starts the scheduled workflow for the first time. For me, it took nearly one hour. So it might be a good idea to get the check working firstly by triggering the workflow with Git push. When you are sure that the check is properly working, then switch to a scheduled trigger.

Note also that once you get this working, it is best to drop the ping frequency (to max once in 24 hours) or disable the rule altogether since otherwise your health check may consume all your monthly free hours.

11.20 Your own pipeline

Build a similar CI/CD-pipeline for some of your own applications. Some of the good candidates are the phonebook app that was built in parts 2 and 3 of the course, or the blogapp built in parts 4 and 5, or the Redux anecdotes built in part 6. You may also use some app of your own for this exercise.

You most likely need to do some restructuring to get all the pieces together. A logical first step is to store both the frontend and backend code in the same repository. This is not a requirement but it is recommended since it makes things much more simple.

One possible repository structure would be to have the backend at the root of the repository and the frontend as a subdirectory. You can also "copy paste" the structure of the example app of this part or try out the example app mentioned in part 7.

It is perhaps best to create a new repository for this exercise and simply copy and paste the old code there. In real life, you most likely would do this all in the old repository but now "a fresh start" makes things easier.

This is a long and perhaps quite a tough exercise, but this kind of situation where you have a "legacy code" and you need to build proper deployment pipeline is quite common in real life!

Obviously, this exercise is not done in the same repository as the previous exercises. Since you can return only one repository to the submission system, put a link of the *other* repository to the one you fill into the submission form.

11.21 Protect your main branch and ask for pull request

Protect the main branch of the repository where you did the previous exercise. This time prevent also the administrators from merging the code without a review.

Do a pull request and ask GitHub user mluukkai to review your code. Once the review is done, merge your code to the main branch. Note that the reviewer needs to be a collaborator in the repository. Ping us in Discord to get the review, and to include the collaboration invite link to the message.

Please note what was written above, include the link to `the collaboration invite` in the ping, not the link to the pull request.

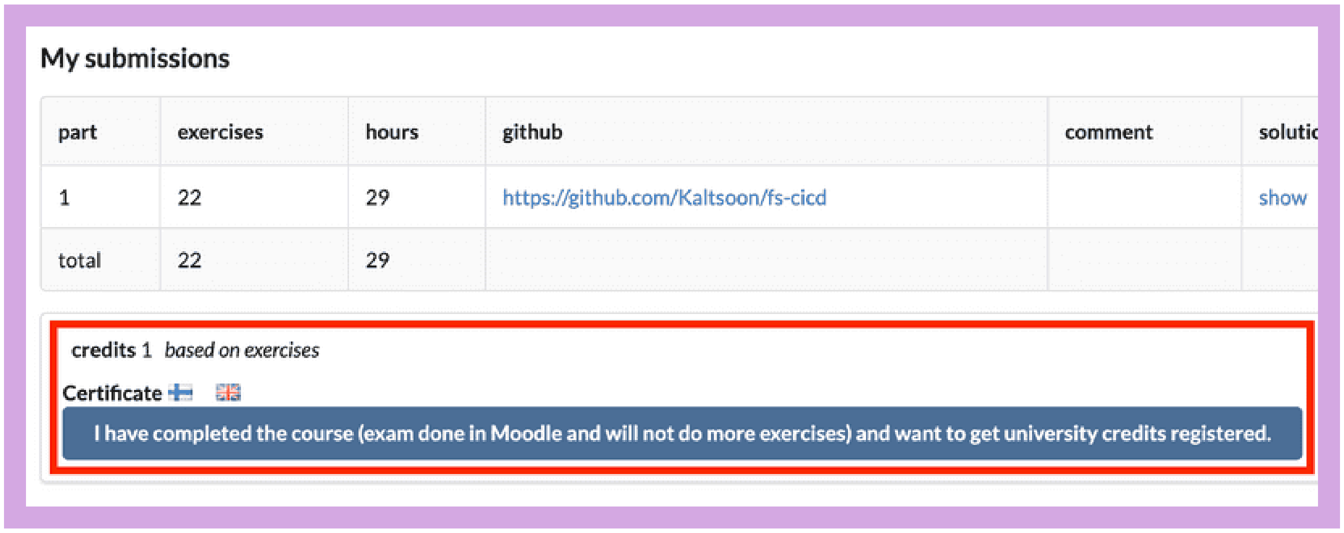
Then you are done!

Submitting exercises and getting the credits

Exercises of this part are submitted via the submissions system just like in the previous parts, but unlike parts 0 to 7, the submission goes to different "course instance". Remember that you have to finish *all the exercises* to pass this part!

Your solutions are in two repositories (pokedex and your own project), and since you can return only one repository to the submission system, put a link of the *other* repository to the one you fill into the submission form!

Once you have completed the exercises and want to get the credits, let us know through the exercise submission system that you have completed the course:



My submissions

part	exercises	hours	github	comment	solutic
1	22	29	https://github.com/Kaltsoon/fs-cicd		show
total	22	29			

credits 1 *based on exercises*

Certificate

I have completed the course (exam done in Moodle and will not do more exercises) and want to get university credits registered.

Note that you need a registration to the corresponding course part for getting the credits registered, see here for more information.

You can download the certificate for completing this part by clicking one of the flag icons. The flag icon corresponds to the certificate's language.

Propose changes to material

Part 11d
Previous part

Part 12
Next part

About course

Course contents

FAQ

Partners

Challenge

UNIVERSITY OF HELSINKI

HOUSTON