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 *pushing the git repository* to the servers of the cloud provider Heroku. It is pretty simple to release software in Heroku at least compared to many other types of hosting setups but it still contains risks: nothing prevents us from accidentally pushing 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 on the 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 PC 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).

Exercises 11.10-11.12.

Before going to the below exercises, you should setup your application in Heroku environment like the one we did in part 3.

In contrast to part 3 now we *do not push the code* to Heroku ourselves, we let the Github Actions workflow do that for us!

Ensure now that you have Heroku CLI installed and login to Heroku using the CLI with `heroku login`.

Create a new app in Heroku using the CLI: `heroku create --region eu {your_app_name}`, pick a region close to your own location!

Generate an API token for your Heroku profile using command `heroku authorizations:create`, and save the credentials to a local file but *do not push those to GitHub*!

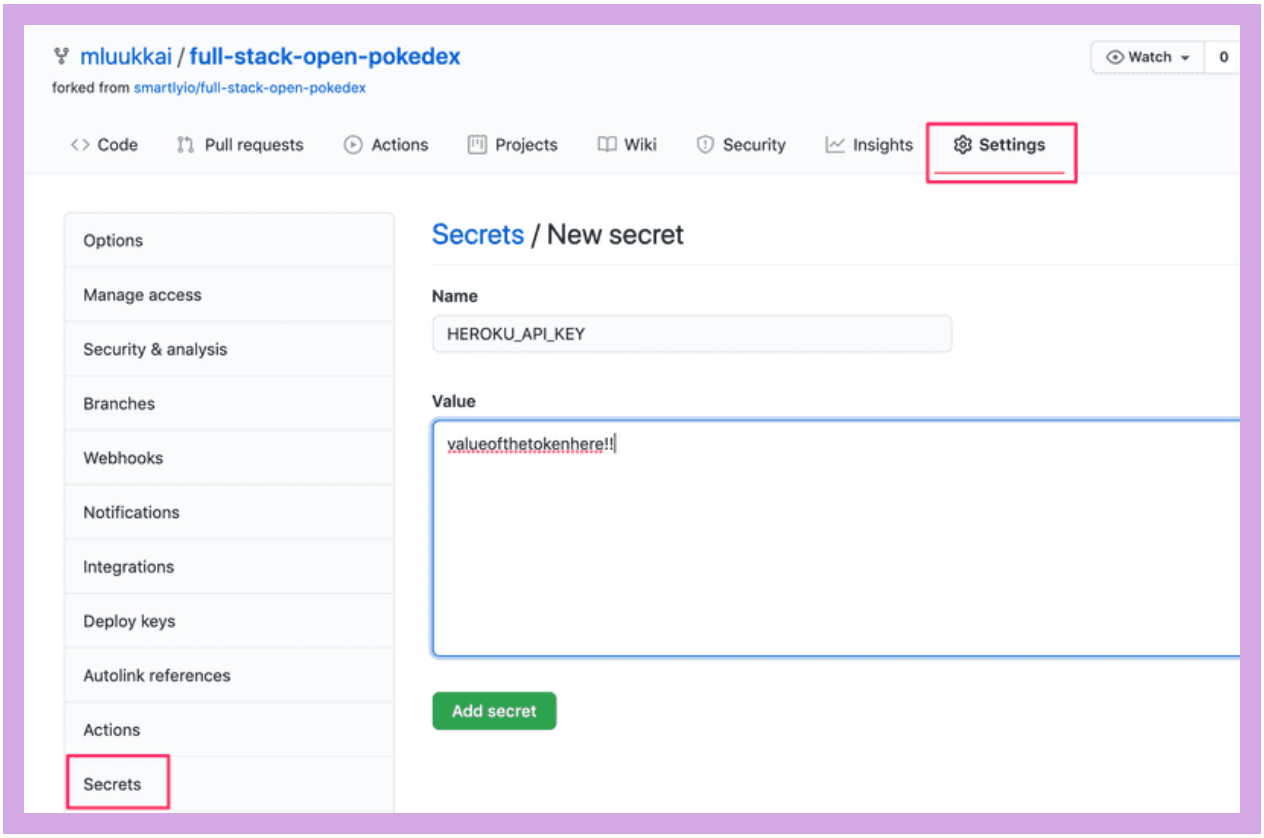
You'll need the token soon for your deployment workflow. See more information at about Heroku tokens here.

11.10 Deploying your application to Heroku

Extend the workflow with a step to deploy your application to Heroku.

The below assumes that you use the ready-made Heroku deploy action AkhileshNS/heroku-deploy that has been developed by the community.

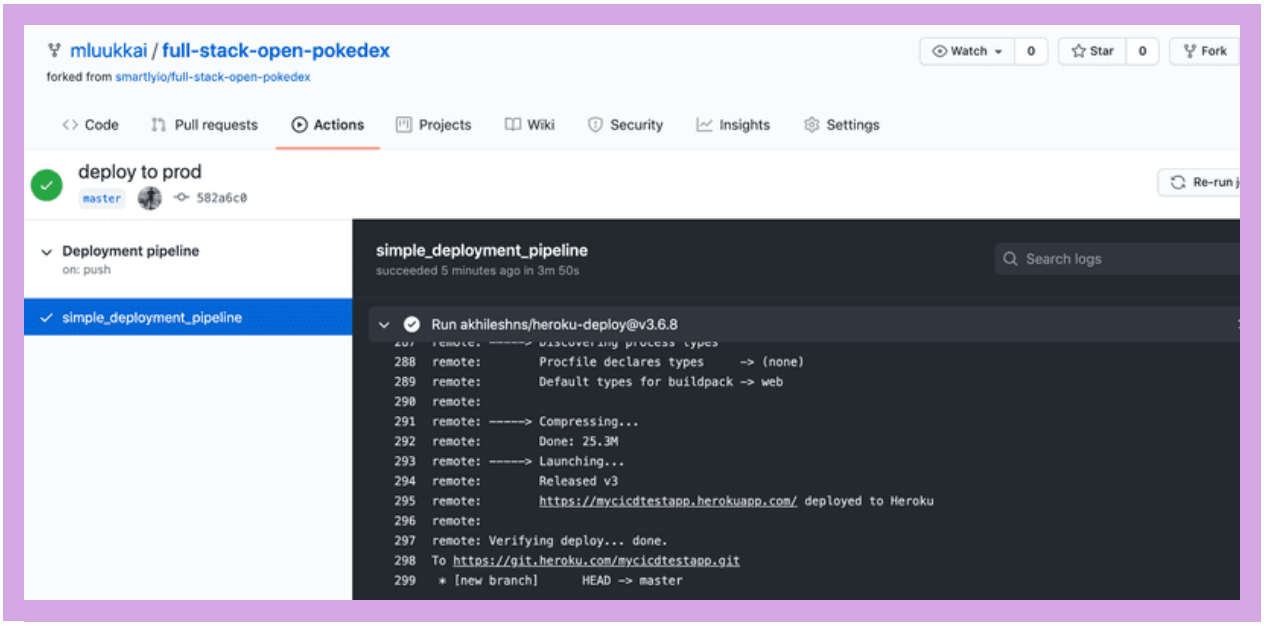
You need the autorization token that you just created for the deployment. The proper way to pass it's value to GitHub Actions is to use repository secrets:



Now the workflow can access the token value as follows:

```
${{secrets.HEROKU_API_KEY}}
```

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



You can then try the app with a browser, but most likely you run into a problem. If we read carefully the section 'Application to the Internet' in part 3 we notice that Heroku assumes that the repository has a file called *Procfile* that tells Heroku how to start the application.

So, add a proper Procfile and ensure that the application starts properly.

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 `heroku logs` early and use it often. No, use it all the time!

11.11 Health check

Before moving on let us expand the workflow with one more step, a check that ensures that the application is up and running after the deployment.

Actually a separate workflow step is not needed, since the action deploy-to-heroku contains an option that takes care of it.

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')
})
```

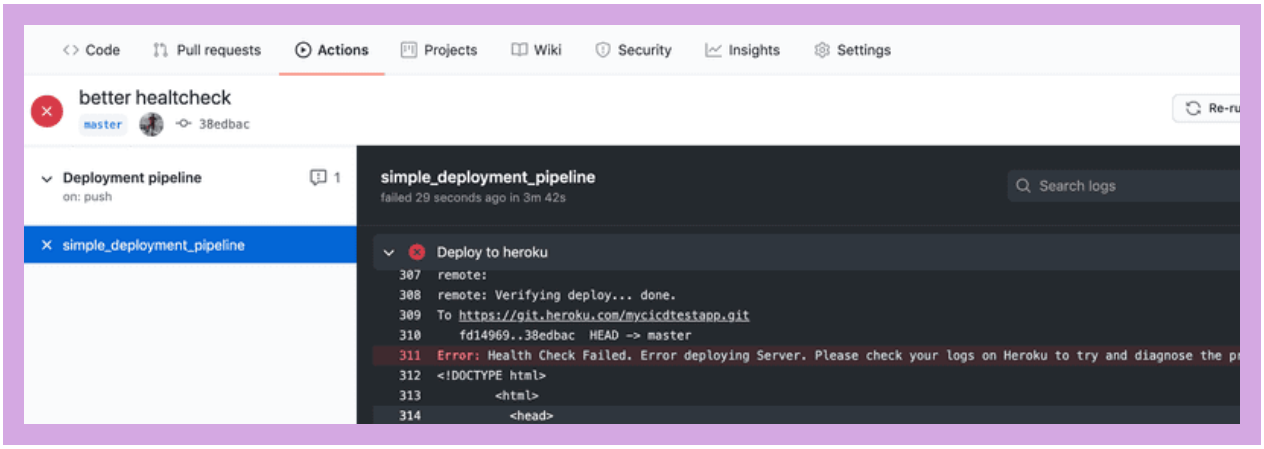
It might also be a good idea to have a dummy endpoint in the app that makes it possible to do some code changes and to ensure that the deployed version has really changed:

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

```
})
```

Look now from the documentation how to include the health check in the deployment step. Use the created endpoint for the health check url. You most likely need also the *checkstring* option to get the check working.

Ensure that Actions notices if a deployment breaks your application. You may simulate this e.g. by writing a wrong startup command to Procfile:



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

11.12. Rollback

If the deployment results in a broken application, the best thing to do is to *roll back* to the previous release. Luckily Heroku makes this pretty easy. Every deployment to Heroku results in a release. You can see your application's releases with the command `heroku releases`:

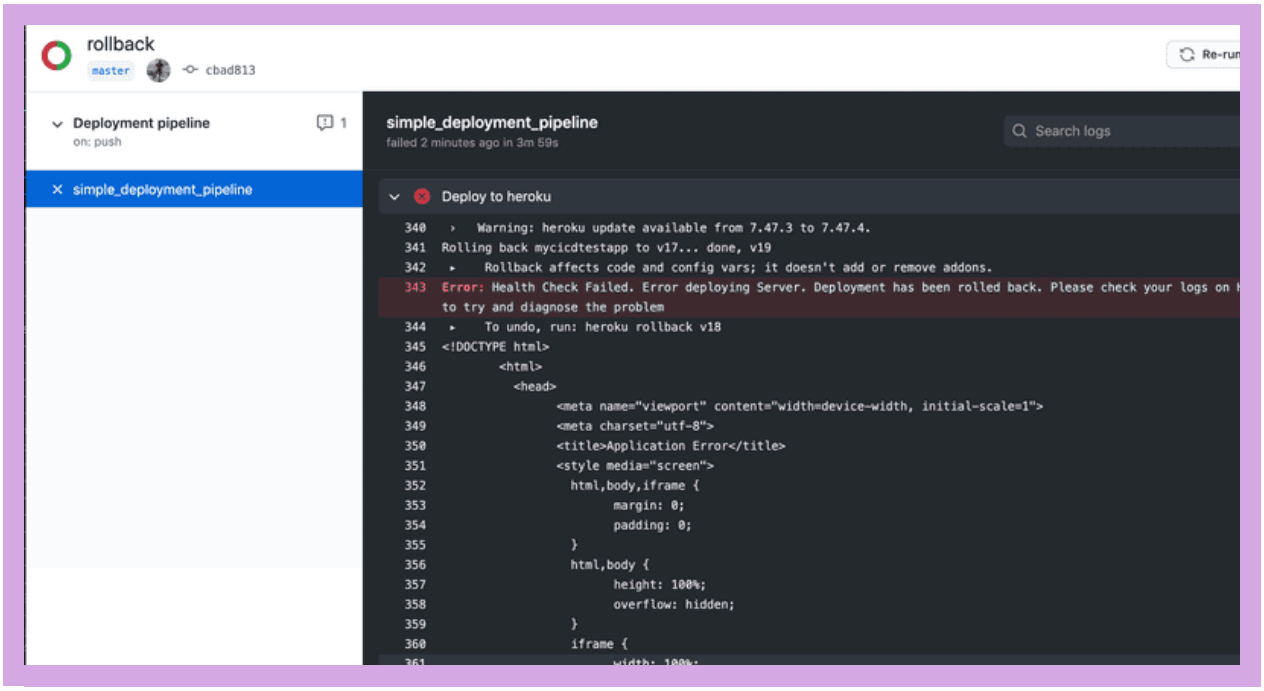
```
$ heroku releases
=== cicdtest222 Releases - Current: v29
v29  Deploy 7fff7150  mluukkai@iki.fi  2020/12/05 18:22:32 +0200
v28  Deploy 764c37d4  mluukkai@iki.fi  2020/12/05 18:09:04 +0200
v27  Deploy 1467d514  mluukkai@iki.fi  2020/12/05 16:28:52 +0200
v26  Deploy ec0ea68b  mluukkai@iki.fi  2020/12/05 15:39:31 +0200
v25  Deploy a8a88aff  mluukkai@iki.fi  2020/12/05 15:34:55 +0200
v24  Deploy d0f3ae58  mluukkai@iki.fi  2020/12/05 15:31:45 +0200
v23  Deploy a348f651  mluukkai@iki.fi  2020/12/05 15:28:19 +0200
v22  Deploy 254d24d4  mluukkai@iki.fi  2020/12/05 14:27:33 +0200
v21  Deploy 950f5403  mluukkai@iki.fi  2020/12/05 14:24:44 +0200
v20  Deploy 9d51da28  mluukkai@iki.fi  2020/12/05 14:22:20 +0200
```

One can quickly do a rollback to a release with just a single command from commandline.

What is even better, is that the action deploy-to-heroku can take care of the rollback for us!

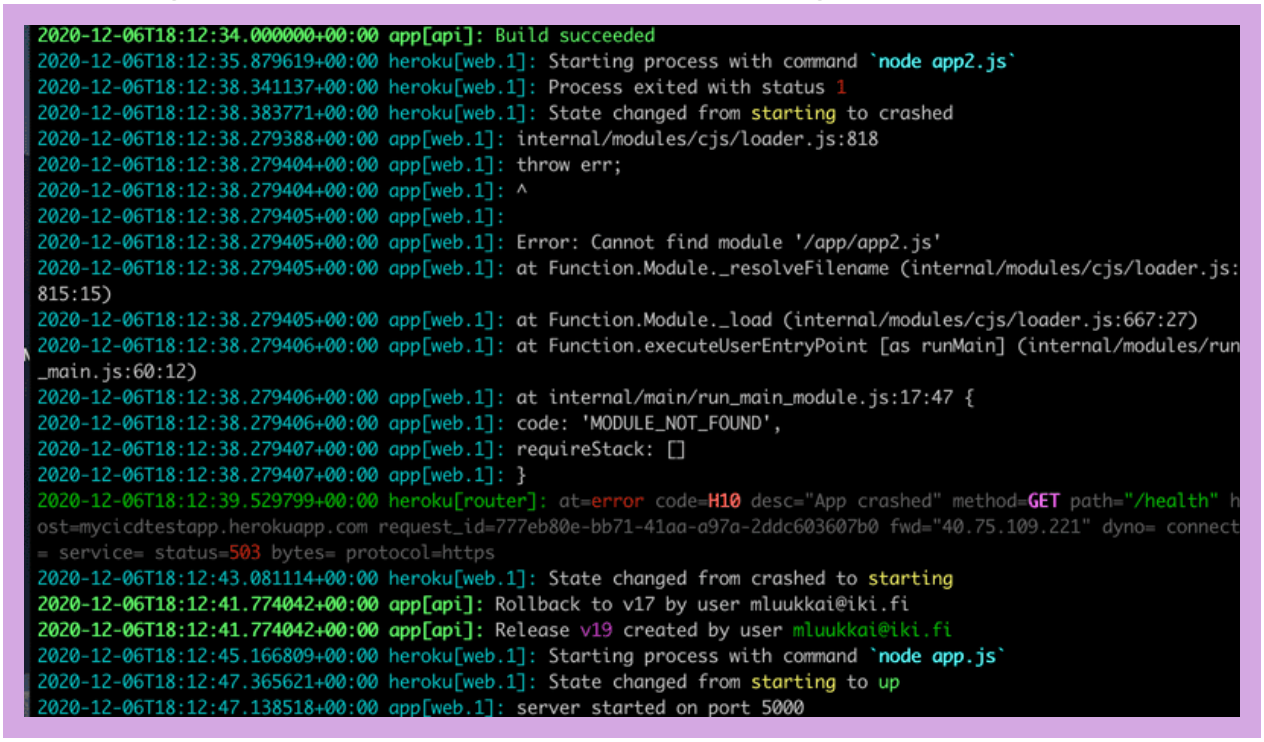
So read again the documentation and modify the workflow to prevent a broken deployment

altogether. You can again simulate a broken deployment with breaking the Procfile:



Ensure that the application stays still operational despite a broken deployment.

Note that despite the automatic rollback operation, the build fails and when this happens in real life it is *essential* to find what caused the problem and fix it quickly. As usual, the best place to start finding out the cause of the problem is to study Heroku logs:



Propose changes to material

Part 11b
Previous part

Part 11d
Next part

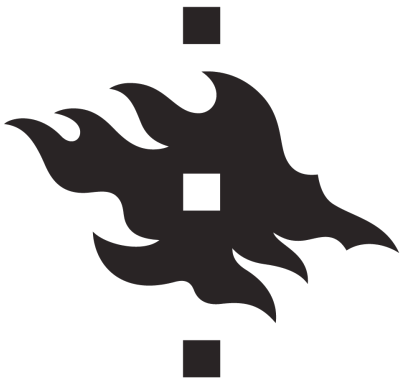
About course

Course contents

FAQ

Partners

Challenge



UNIVERSITY OF HELSINKI