

```
{() => fs}
```



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 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 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 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  
})
```

[copy](#)

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
```

[copy](#)

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"]
```

[copy](#)

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
```

[copy](#)

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.

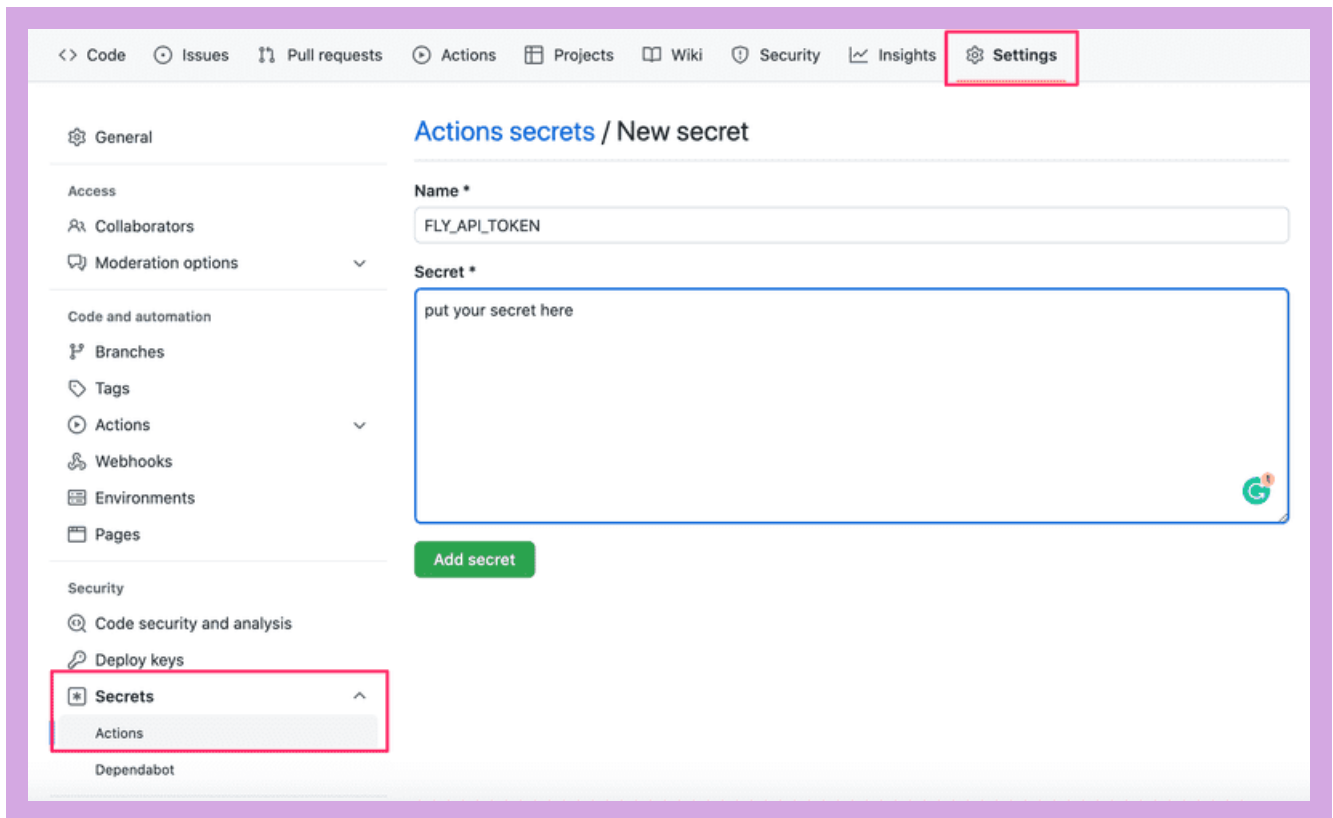
Before moving to the next exercise, make sure that the manual deployment with the command `flyctl deploy` works!

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`:



Now the workflow can access the token value as follows:

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

[copy](#)

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

simple_deployment_pipeline

succeeded 1 minute ago in 2m 33s

Search logs

Run flyctl deploy --remote-only

```

137
138 Updating existing machines in 'pdex' with rolling strategy
139 > [1/2] Updating 48e5996b76eed8 [app]
140 > [1/2] Updating 48e5996b76eed8 [app]
141 > [1/2] Waiting for 48e5996b76eed8 [app] to have state: started
142 > [1/2] Machine 48e5996b76eed8 [app] has state: started
143 > [1/2] Checking that 48e5996b76eed8 [app] is up and running
144 > [1/2] Waiting for 48e5996b76eed8 [app] to become healthy: 1/1
145
146 ✓ [1/2] Machine 48e5996b76eed8 [app] update succeeded
147 > [2/2] Updating 5683dd7db03228 [app]
148 > [2/2] Updating 5683dd7db03228 [app]
149 > [2/2] Waiting for 5683dd7db03228 [app] to have state: started
150 > [2/2] Machine 5683dd7db03228 [app] has state: started
151 > [2/2] Checking that 5683dd7db03228 [app] is up and running
152 > [2/2] Waiting for 5683dd7db03228 [app] to become healthy: 0/1
153
154 > [2/2] Waiting for 5683dd7db03228 [app] to become healthy: 1/1
155
156 ✓ [2/2] Machine 5683dd7db03228 [app] update succeeded
157
158 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

copy

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

```

copy

```
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

copy

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}`)
})
```

copy

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')
})
```

copy

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 either *canary* or *bluegreen*. These strategies ensure that only an app with a healthy state gets deployed.

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

The screenshot shows a GitHub Actions workflow run for a job named 'simple_deployment_pipeline'. The run failed 30 minutes ago in 7m 8s. The failed step is 'Run flyctl deploy --remote-only'. The log output shows the deployment process for machine 48e5996b76eed8, which failed due to a timeout reaching the 'started' state. The error message is: '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'. The final log line states: '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')
})
```

copy

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
```

[copy](#)

```
echo "Build script"
```

```
# add the commands here
```

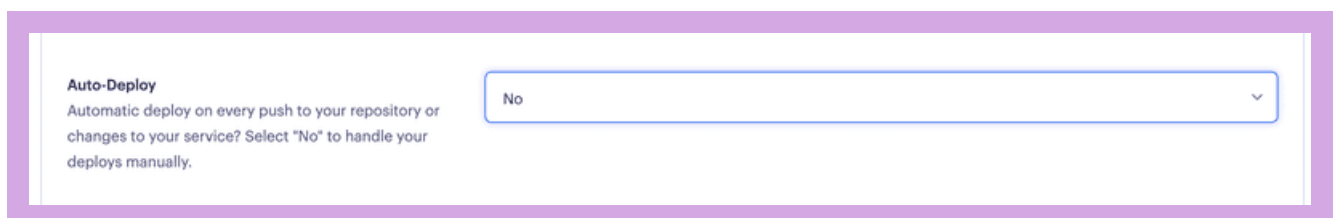
Give it execution permissions (Google or see e.g. [this](#) to find out how) and ensure that you can run it from the command line:

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

[copy](#)

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-randomcharactershere`

[copy](#)

Deployment with deploy hook

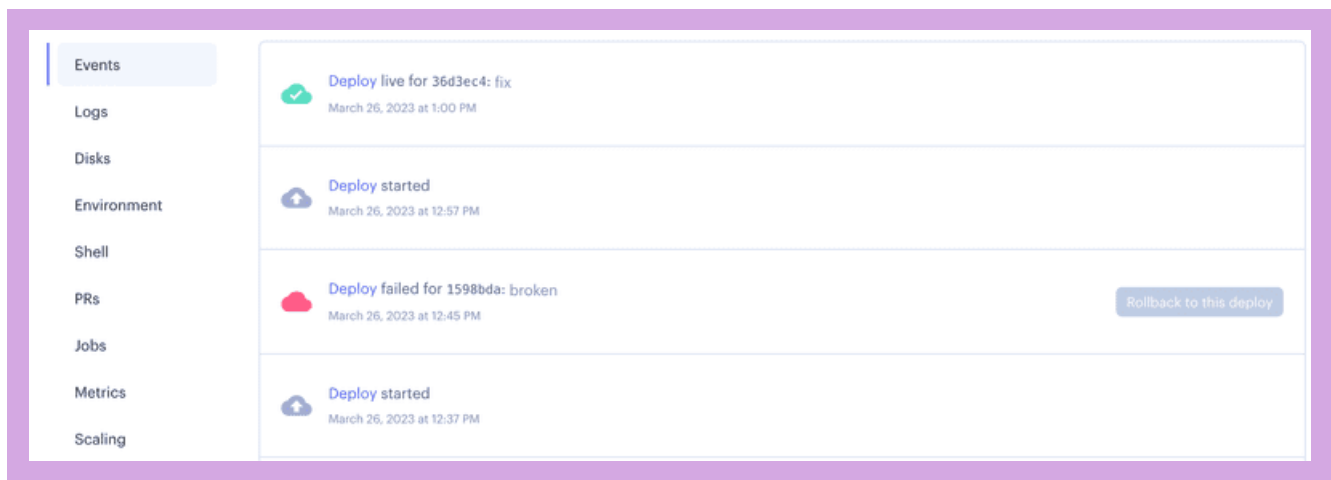
Alternative, and perhaps a more reliable option is to use Render Deploy Hook which is a private URL to trigger the deployment. You can get it from your app settings:

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

[copy](#)

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



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')
})
```

copy

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
```

copy

```
if (true) throw('error... ')\nres.send('ok')\n})
```

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](#)[About course](#)[Course contents](#)[FAQ](#)[Partners](#)[Challenge](#)



UNIVERSITY OF HELSINKI

