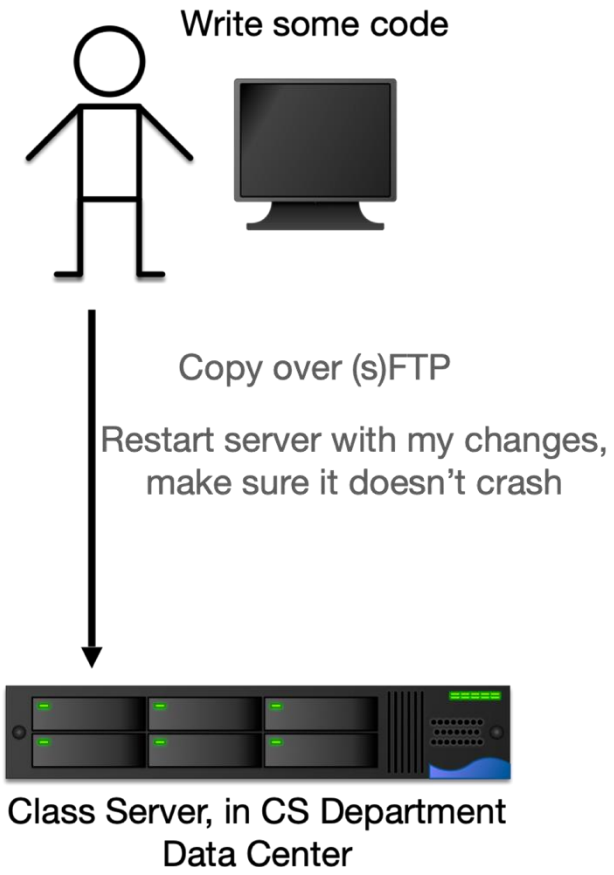
CS 4530: Fundamentals of Software Engineering
Lesson 5.2
Continuous Deployment

Rob Simmons

Khoury College of Computer Sciences

How to (re)deploy a web app?

- Very old school: copy over SFTP, restart the server
- Also very old school: SSH in and edit the code on the server directly
- Slightly less terrifying: have the code in a git repository, SSH in, update the repo, restart server



Upgrade steps

These are instructions for upgrading from Hometown `v4.0.14+1.1.1`.

As always, make sure you have backups of the database before performing any upgrades. A postgres backup command would look something like this: `pg_dump -Fc -U postgres mastodon_production > name_of_the_backup.dump`

- `git remote update && git checkout v4.0.15+hometown-1.1.1`
- Install dependencies: `bundle install` and `yarn install --frozen-lockfile`
- Restart all Mastodon processes

https://github.com/hometown-fork/hometown/releases

This is… not agile.

Agile Goal: *continuous deployment*

- Whenever we commit to the main branch of our repository, we want to update the service running to users

Continuous Deployment for Static Sites

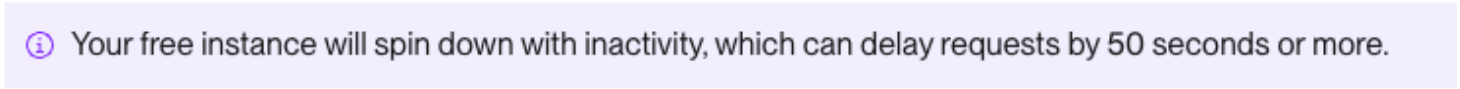
1. GitHub reports that the main branch is updated
2. Spin up a VM that checks out the repo
3. Build the files for the website
4. Send those files to a PaaS that serves it to users
5. Shut down VM

- Render.com will do this for free!
- GitHub Pages will too — that’s how the course site works

Continuous Deployment for Web Services

1. *Last version of the web server is puttering away in a VM*
2. GitHub reports that the main branch is updated
3. Spin up a *new* VM that checks out the repo
4. Do any build steps that need to happen to get the web server ready to run
5. *Redirect network traffic from old VM to new VM*
6. Shut down *old* VM. Zero downtime!

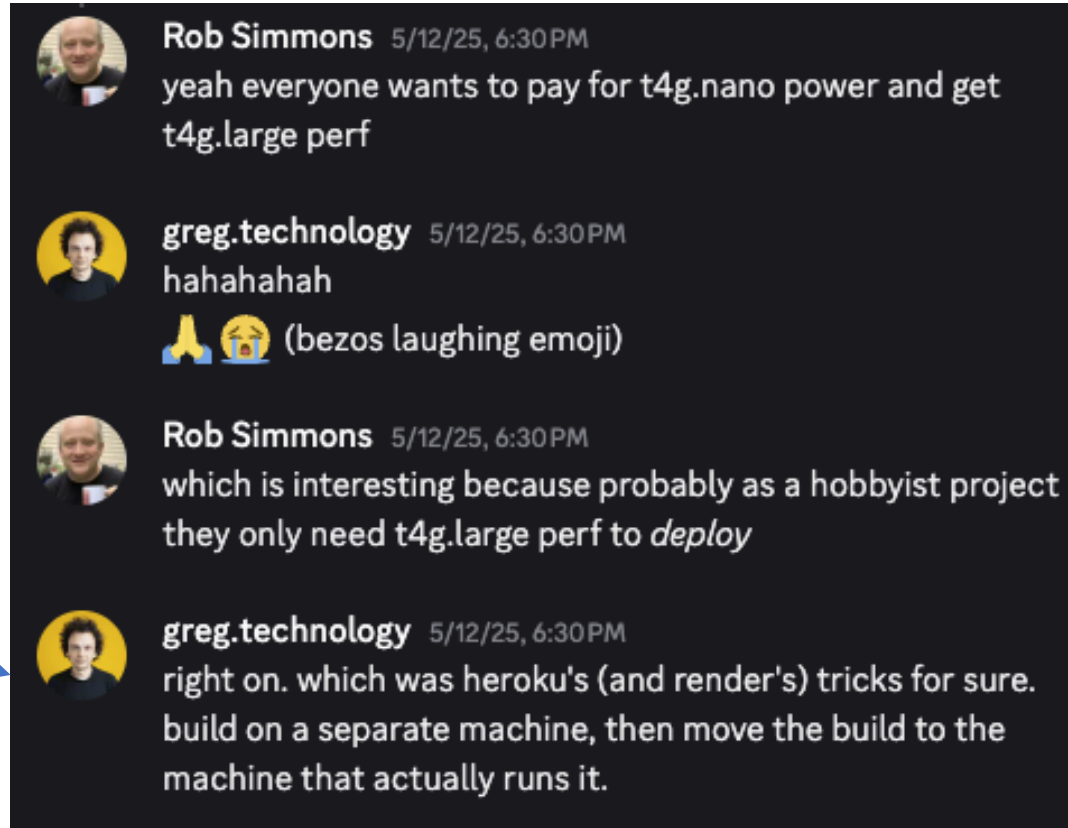
- Heroku pioneered this as a business model in 2007, Render.com will do this for free, kinda!



- Not actually VMs, but *containers*

Virtual Machines to Containers

- Each VM contains a full operating system
- What if each application could run in the same (overall) operating system? Why have multiple copies?
- Advantages to smaller apps:
  - Faster to copy (and hence provision)
  - Consume less storage (base OS images are usually 3-10GB)
  - Quite easy and fast to change resources available to a container



Containers run layered images, reducing storage space

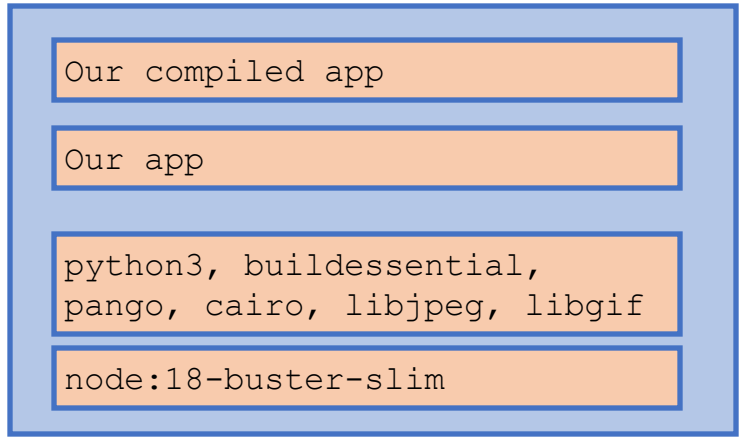
- Images are defined programmatically as a series of “build steps” (e.g. Dockerfile)
- Each step in the build becomes a “layer”
- Built layers can be shared and cached
- To run a container, the layers are linked together with an “overlay” filesystem

```
FROM node:18-buster-slim
RUN apt-get update && apt-get install python3
build-essential libpango1.0-dev libcairo2-dev
libjpeg-dev libgif-dev -y

RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app
COPY ./ /usr/src/app

RUN npm ci
RUN npm run build
CMD [ "npm", "start" ]
```

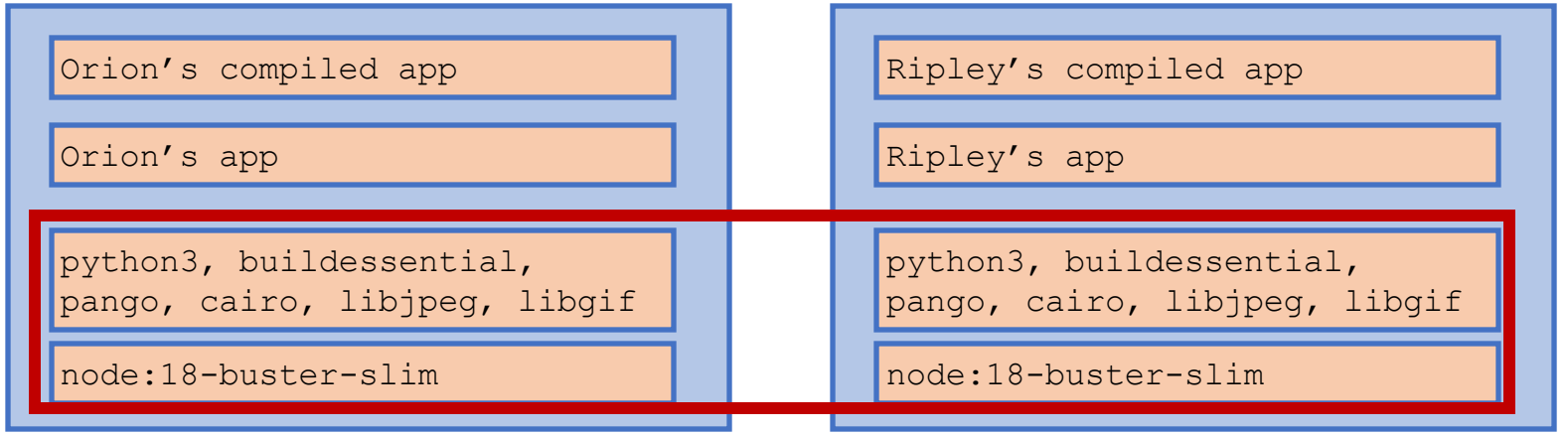
Example image specification (Dockerfile)



Example image, with layers shown

Containers run layered images, reducing storage space

- Many images may share the same lower layers (e.g. OS, NodeJS, some system dependencies)
- Layers are shared between images
- Multi-tenancy: N running containers only require one copy of each layer (they are read-only)



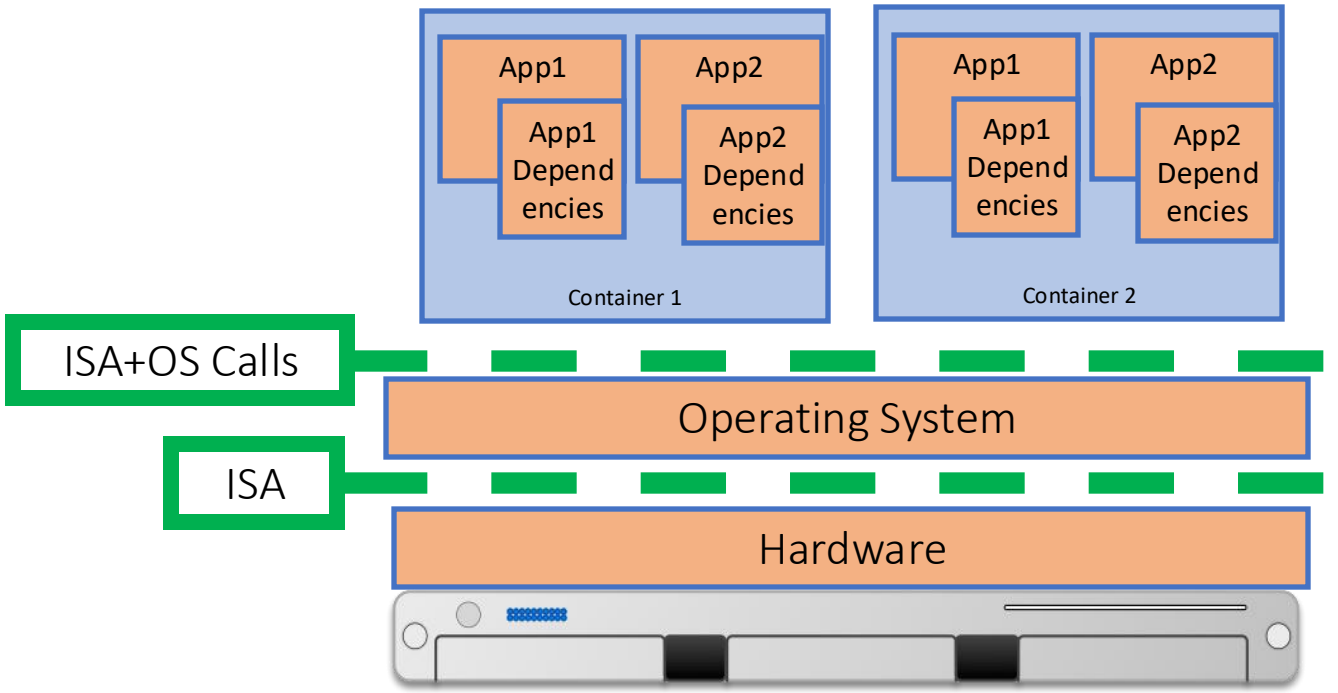
Two images, sharing two layers

A container contains your apps and all their dependencies

- Each application is encapsulated in a “lightweight container,” includes:
  - System libraries (e.g. glibc)
  - External dependencies (e.g. nodejs)
- “Lightweight” in that container images are smaller than VM images - multi tenant containers run in the OS
- Cloud providers offer “containers as a service” (Amazon ECS Fargate, Azure Kubernetes, Google Kubernetes)

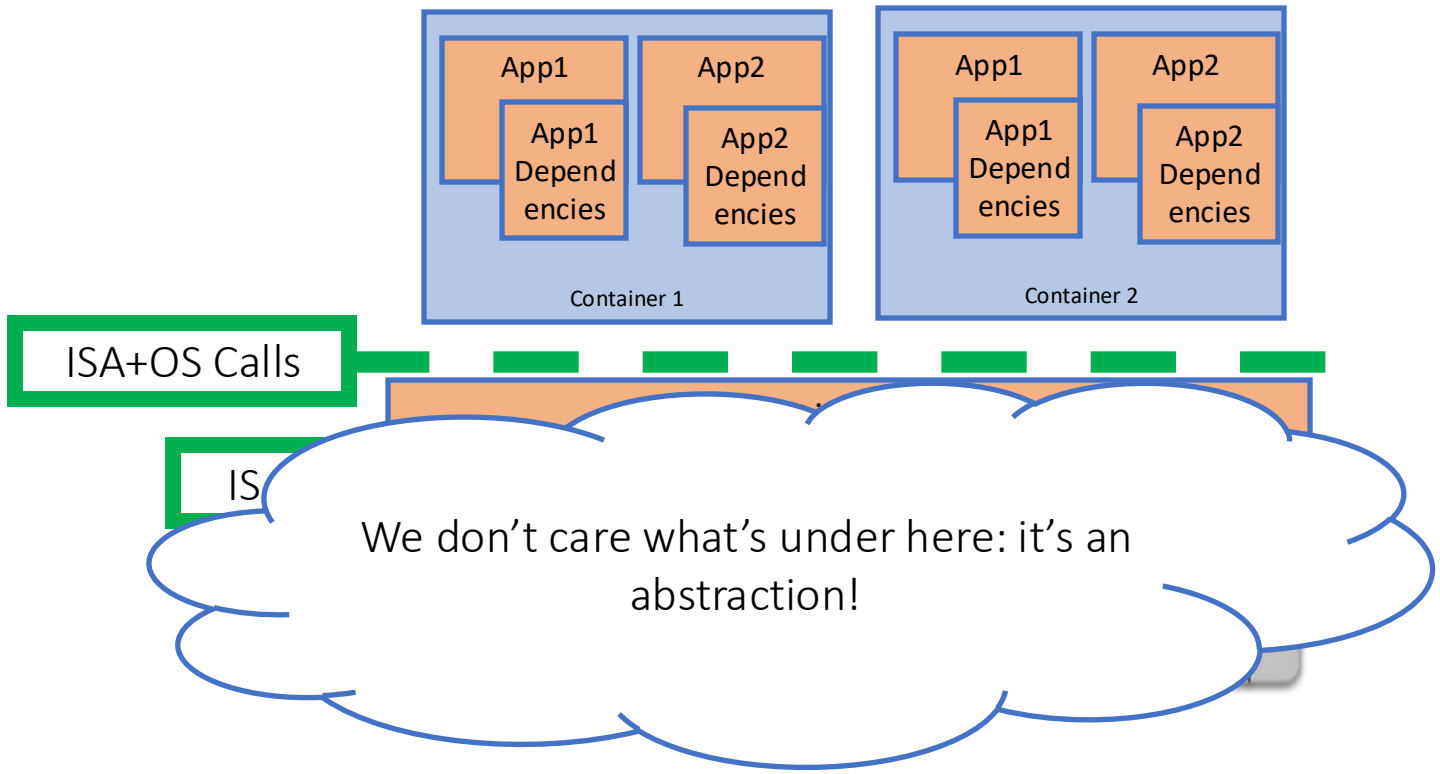
A container contains your apps and all their dependencies

- You might put several apps in a single container, together with their dependencies
- Might have only one copy of shared dependencies



XaaS: Containers as a Service

- Vendor supplies an on-demand instance of an operating system
  - e.g.: Linux version NN
- Vendor is free to implement that instance in a way that optimizes costs across many clients.

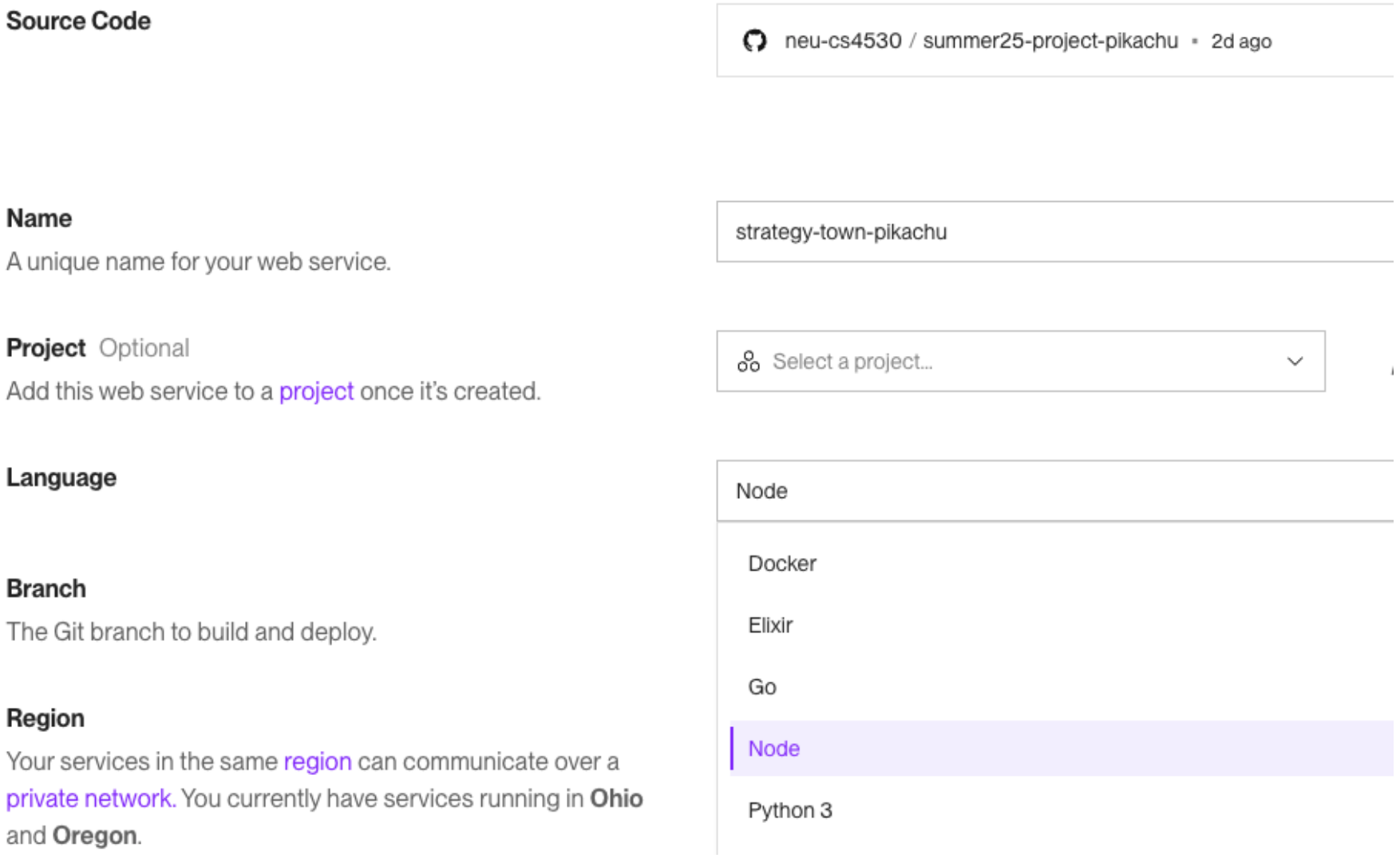


Tradeoffs between VMs and Containers

- Performance is comparable
- Each VM has a copy of the OS and libraries
  - Higher resource overhead
  - Slower to provision
  - Support for wider variety of OS’s
- Containers are “lightweight”
  - Lower resource overhead
  - Faster to provision
  - Potential for compatibility issues, especially with older software

Continuous Deployment on Render

- For apps doing basic things with JS (or Go, or Python…) Render can hide the fact you're using containers behind some friendly web-form config



Continuous Deployment on Disco

https://disco.cloud
(Greg and Antoine are very friendly!)

- The raspberry pi in my utility closet does all the steps that Render does, basically, with Disco!

./Dockerfile

```
# This file is one way of describing how to build
# and run strategy.town
FROM node:24-slim
WORKDIR /code

# start with dependencies to enjoy caching
COPY ./package.json /code/package.json
COPY ./package-lock.json /code/package-lock.json
COPY ./shared/package.json /code/shared/package.json
COPY ./server/package.json /code/server/package.json
COPY ./client/package.json /code/client/package.json
RUN npm ci

# copy rest and build
COPY ./ /code/.
RUN npm run build -w=client
CMD ["npm", "run", "start", "-w=server"]
```

./disco.json

```
{
 "version": "1.0",
 "services": {
  "web": { "port": 8000 }
 }
}
```

<- tells disco that this is a web service that expects connections on port 8000

<- **Docker** is the prevailing container platform; the Dockerfile describes how to build the container the app is built in and runs in

Continuous Deployment on NU's Private Cloud

./Dockerfile

```
# This file is one way of describing how to build
# and run strategy.town
FROM node:24-slim
WORKDIR /code

# start with dependencies to enjoy caching
COPY ./package.json /code/package.json
COPY ./package-lock.json /code/package-lock.json
COPY ./shared/package.json /code/shared/package.json
COPY ./server/package.json /code/server/package.json
COPY ./client/package.json /code/client/package.json
RUN npm ci

# copy rest and build
COPY ./ /code/.
RUN npm run build -w=client
CMD ["npm", "run", "start", "-w=server"]
```

./helm/templates/ingress.yaml

```
{{- if .Values.ingress.enabled -}}
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: {{ include "cs4530-project.fullname" . }}
  labels:
    {{- include "cs4530-project.labels" . | nindent 4 }}
  annotations:
    nginx.org/websocket-services: {{ include "cs4530-project.fullname" $ }}
  {{- with .Values.ingress.annotations }}
    {{- toYaml . | nindent 4 }}
  {{- end }}
spec:
  {{- with .Values.ingress.className }}
  ingressClassName: {{ . }}
  {{- end }}
  {{- if .Values.ingress.tls }}
  tls:
    {{- range .Values.ingress.tls }}
    - hosts:
        {{- range .hosts }}
        - {{ . | quote }}
        {{- end }}
      secretName: {{ .secretName }}
    {{- end }}
  {{- end }}
  rules:
    {{- range .Values.ingress.hosts }}
    - host: {{ .host | quote }}
      http:
        paths:
          {{- range .paths }}
          - path: {{ .path }}
            {{- with .pathType }}
            pathType: {{ . }}
            {{- end }}
```

<- no clue tbh, and there are like a dozen of these files

Continuous Deployment

Common elements:

- Dependencies
  - “Do normal node stuff” (Render) vs Dockerfile (Render, Disco, NU)
- Configuration
  - Web-based config (Render), config file (Disco), something else (NU)
- Secrets and environment variables (i.e. MONGODB_CONNECTION_STRING)
  - **Don’t check your secrets in to your repo!**
  - Web-based config (Render, Disco)
  - GitHub secrets that get passed to the container-running machines (NU)