# CICDont DevOps exam report

Benjamin Kjær, Janus Jónatan Hannesarson, Oliver Jørgensen & Silas Nielsen

IT University of Copenhagen, Copenhagen, Denmark
bekj, januh, ojoe, sipn

Date 25/05/2023

Course code: KSDSESM1KU

# Contents

# 1 System's Perspective

Refferences?

## 1.1 Design & Architecture

Frontend is built using Next.js[9]. The data is fetched through a Go Gin[7] API interacting with the database through an ORM called GORM[8]. The service is deployed through Docker with the backend and frontend being separate containers. To provide insights into the system we also deploy "supporting" containers like EFK stack, Grafana & Prometheus. Finally, we also have a container for Nginx responsible for providing a network configuration that allows us to have an SSL certificate along with a domain address.

We chose Digital Ocean as our hosting provider using their VM service "Droplets". The system consists of 3 droplets, 1 manager node, and 2 worker nodes working together in a docker swarm setup . We also pay for a hosted postgres database on Digital Ocean that we use to store all messages, follower relations and users.

Make sure we actuall
this before turnin

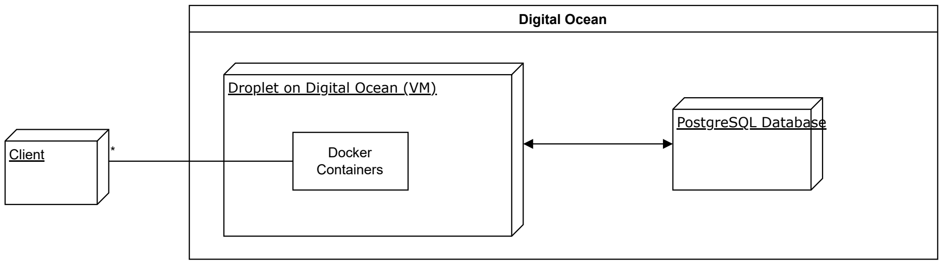

Figure 1: Digital Ocean infrastructure

If we do swarm, then
date image

## 1.2 Dependencies on all levels of abstraction

**Hosting** depends on Virtual Machines (droplets) & Database(postgres) managed by DigitalOcean.

**CI/CD chain** depends on Github Actions as a service Github runs.

**Remote code repository** is hosted on Github.

**Project dependencies:** Both the frontend and backend solutions rely on external dependencies and libraries in order to work. It is not feasible to construct every single package from the ground up, which is why we chose this approach. Since the frontend is build using

Next.js we use npm as a package manager, which constructs a "package.json" that handles the dependencies. For the go backend, we also rely on external packages, that are all stored in a "go.mod" file. Both the "package.json" (5.2) and "go.mod" (5.1) can be seen in the appendix.

## 1.3 Important interactions of subsystems

ere

## 1.4 Current state of system

SonarCloud result: `https://sonarcloud.io/project/overview?id=DevOps-CI-CDont_DevOps-CI-CDont2`

1. 1 bug ("an unnecessary expression in an if statement")
2. 0 vulnerabilities
3. 51 code smells (naming convention breaches, some things could be constants)
4. 11 Security hotspots (eg. secrets in docker environment variables)

## 1.5 License & compatibility with direct dependencies

## 1.6 Weekly Tasks

The weekly tasked, were based on the exercises that were given in the lecture.

## 1.7 Choice of technologies

### 1.7.1 Frontend

We chose Next.js as our frontend service, due to it's many build in features, such as:

1. - Code splitting
2. - Routing support
3. - Server Side Rendering
4. - Layouts and Components
5. - API routes
6. - Build in optimizations
7. - Support for middleware

Furthermore, we have added Tailwindcss as a CSS utility class for easy styling. Having Server Side Rendering (SSR) isn't necessary for the Minitwit application, but it allows us to fetch data and build a page on the server, which gives a much better score on Google Lighthouse and also removes Cumulative Layout Shift (CLS), which gives a better user experience.

Having code splitting, divides the js build files into smaller files, making the size smaller and therefore 'lighter' to fetch (Making the application faster).

Next.js handles routing and API routes in the pages directory (Next.js 13 = app directory). This removes the overhead of having to create a react router that points to the different page components. Having a dedicated API routes folder, allows us to create API routes, that can fetch our backend data with ease. The API routes folder removes many issues and privacy related data, such as enviornment variables.

The Minitwit application has a user authentication system. With Next.js we can create a middleware that removes unauthenticated users from accessing pages, they're not allowed to view.

### 1.7.2 API technology choice

We chose to use Go with Gin to make a RESTful API because: It is a new language for all group members, so we can learn it together. We thought it would be an interesting language to build an API in. It is a pretty modern language, that is said to be built for concurrency, it's fast. *"The story goes that Google engineers designed Go while waiting for other programs to compile. Their frustration at their toolset forced them to rethink system programming from the ground up, creating a lean, mean, and compiled solution that allows for massive multithreading, concurrency, and performance under pressure."* - <https://stackoverflow.blog/2020/11/02/go-golang-learn-fast-programming-languages/> We first expected to use Go with Gorilla Mux, but we found out that that library has been archived - and so decided not to use Mux.

### 1.7.3 Database

At first we "inherited a local" database setup (a db file inside backend/tmp), and we didn't change that when we started containerizing the application. This was a very bad setup for real production data, as the database would be lost when the container was restarted.
We have since changed the database to a hosted PostgreSQL database via DigitalOcean.

### 1.7.4 Domain

We used Name.com to purchase `cicdont.live` for a year, for free.

Skal der være flere te
nology choices her?
"MSc students remem
to argue for the choic
of technologies and d
sions for at least all o
for which we asked yo
to do so in the tasks
the end of each sessio

# 2 Process' Perspective

## 2.1 Developer interaction

## 2.2 Organization of team

The team handled both Agile processes, DevOps practices, Frontend and backend development, therefore a suitable name for the team would be a "Full-Stack DevOps Team". The team was not build by generalists, but rather specialists in the frontend and backend departments. As DevOps is the main intended learning outcome of the course, all team members were responsible for this practice.

- **Agile:** As the course required new implementations and requirements every week, the team is organized in an agile manner, always ready to adapt and respond to changes.
- **DevOps:** Other than migrating the old "minitwit" application into a modern framework, most of the work thereafter was done in devops practices, including monitoring, CI/CD pipelines and more. All developers were responsible for working and implementing the changing feature requirements, provided by the course.
- **Frontend:** 1 of the team members was assigned to the frontend, redesigning the entire application to use a modern frontend framework (Next.js), a long with updating dependencies and improving the UI and UX of the application.
- **Backend:** 3 of the team members worked on the backend, migrating the application from a Python flask API into a Go:GIN application.

## 2.3 Full stage & tool description of CI/CD Chains

### 2.3.1 Continuous integration

We built a continuous integration(CI) workflow[2] that would serve as a quality gate when pushing new changes. The aim was to define a set of tests that if ever failed we would not want to publish those changes to production. As the main focus of Minitwit was to handle requests from the simulator our tests only related to the backend and no end-to-end tests exist.

When a push or pull-request is accepted into the main branch, a virtual machine (VM) is spun up containing environment variables needed for establishing a connection to the database, these secrets are fed through GitHub's built-in "github secrets"[6]. Once the machine is ready it builds a test-specific Dockerfile that mimics our real Dockerfile but instead connects to a test database rather than production. We then also install DigitalOcean's command line tool "Doctl"[5], and push our new image to the Digital Oceans Container registry.

With this job completed we now have our API running on our virtual machine and we can start testing. Testing happens in a new job so again we spin up a virtual machine that installs doctl but this time it instead pulls the image the other job uploaded and then runs the image before making a call to "go test" running all go tests defined in the repository. Finally, we also have a static code analysis job running that installs all dependencies before running a typescript type check "tsc", eslint, and go vet. If any of these steps fails the job fails and the workflow as a whole exists with a "Failure".

### 2.3.2 Continuous deployment

To support frequent delivery of new versions of mini twist we also created a continuous deployment(CD) workflow on Github actions. The workflow runs any time the CI workflow completes and the conclusion is "Success", meaning that all jobs succeeded without fail. The workflow then starts a VM that builds the backend and frontend images (along with the necessary secrets stored in github secrets), before installing doctl and pushing the images to the container registry.

But just pushing them to the registry is not enough as we need to communicate to our digital ocean droplet that a change was made to one of the images. To achieve this we utilize "Watchtower"[4]. Watchtower monitors running docker containers and watches for any changes made to the images that initially started those containers, if any images are updated watchtower will restart the container using the new image. Watchtower is set up on our project to look for a change on any of our images every 60 seconds. The combination of the CD and watchtower means that any time we push code(that passes the quality gate defined in CI) to the main branch it also updates our hosted production server.

## 2.4 Organization of repository

We decided to structure the repository as a monorepo. We had 2 projects in the repo concerning a frontend project in Next.js and a backend project in Go. The reason we decided for this structure, is that a monorepo allows us to sync releases from one main repository. Meaning every time there is a release, then all of the code is at the same state. Furthermore a monorepo is useful when the codebase is not too large, meaning we do not have thousands of files and or packages in the codebase, slowing down each pull, commit and push.

When building micro services a polyrepo can seem to be the natural choice, however what the monorepo allowed us to do was creating a unified and automated build and deploy pipeline, that can mitigate many issues associated with polyrepos, such as not being at the same state [3].

## 2.5 Applied branching strategy

At the project kick-off we did not have a branching strategy, meaning all development was done on a "Dev" branch, that all developers were working on. This did not cause a lot of issues, as all commits and pushes were short in terms on modified lines of code, leading to no merge conflicts.

Later in the project, we developed a dedicated branching strategy, following the GitHub Flow strategy [1]. This strategy is dedicated to small teams, having a main, develop, feature and hotfix branch. All developers branch directly from the main branch, such that the main branch which remains stable. All branches are then isolated to work on and can then be merged into main again. The reason we chose this over the GitFlow strategy, is that the GitHub Flow strategy does the same but without release branches. We chose to have all of our releases directly from the main branch, when all work was done. This kept the main branch in a constant deployable state, supporting our CI and CD processes.

## 2.6 Applied development process

We used Github's built-in issue tracking system, eventually also using those issues inside a Kanban-like board via a Github "Project" for the Github organization we had. At the end, we had 4 columns in the board (Todo, In Progress, Done and Unprioritized). Unprioritized recommended all the tasks that we found would be realistic tasks for the project if it were to continue, that weren't prioritized for the course.

## 2.7 Monitoring

Prometheus & Grafana ...
DigitalOcean resource alerts and uptime (`http://cicdont.live/public`)

## 2.8 Logging

Standard out from all containers...

## 2.9 Security assessment

## 2.10 Applied strategy for scaling and load balancing

Before trying to use Docker Swarm, our only scaling had been vertical: increasing resources for CPU, RAM, and Disk on a single Virtual Machine (Droplet).

### 2.11 AI assistants

We have used Github Copilot via its VSCode extension. A couple of areas where it was memorably useful:

1. writing out repetitive patterns in the API (eg. error handling in Go, checking for certain parameters/cookies).
2. writing fetch requests from the frontend (it seems to understand well how to use the endpoints that are described in the same repository).
3. writing utility functions (Copilot is especially effective if given a perfectly descriptive function name, even better if standard terminology is used)

One challenge of using Copilot with an unfamiliar language (such as Go was to all of us), is that it can be really hard to tell if a suggestion is correct. Even if something seems to work as intended, it is still important to understand the code.

Another "AI Assistant" we have used is ChatGPT. A couple of areas where it was memorably useful:

1. Suggesting and explaining nginx configurations
2. Debugging CORS errors, suggesting fixes with middleware in Go backend.
3. Debugging memory usage problems.
4. Generating commands for iptables configuration.

The conversational design is quite nice to be able to "tweak" its suggestions. It often spits out large blocks of code that aren't quite what you want, but it can fix that if told what to tweak.
A downside is that it may *hallucinate* commands, flags, and functions that seem very plausible and one may think "How great! That command is exactly what I want", and then it doesn't exist.
When asked to use the functionality of a library, it will tend to use it in the way that there is most content like on the internet - this means the newest "best practices" aren't as likely to be used as whatever is still most common in the training data.

## 3 Lessons Learned Perspective

ware as a Service can
orth it :)

## 4 References

[1] Rowan Haddad. *What Are the Best Git Branching Strategies*. https://www.flagship.io/git-branching-strategies/. 2022.

[2] Benjamin Kjær, Janus Jónatan Hannesarson, Oliver Jørgensen & Silas Nielsen. *Continuous integration*. https://github.com/DevOps-CI-CDont/DevOps-CI-CDont/blob/main/.github/workflows/CI.yml. 2023.

[3] Ron Powell. *Benefits and challenges of monorepo development practices*. https://circleci.com/blog/monorepo-dev-practices/. 2023.

[4] containrrr. *Watchtower*. https://github.com/containrrr/watchtower. GitHub repository. n.d.

[5] DigitalOcean. *doctl Command-Line Interface (CLI)*. DigitalOcean. n.d. URL: https://docs.digitalocean.com/reference/doctl/.

[6] GitHub. *Encrypted Secrets*. GitHub. n.d. URL: https://docs.github.com/en/actions/security-guides/encrypted-secrets.

[7] *Gin Web Framework*. https://gin-gonic.com/. Accessed: May 18, 2023.

[8] *GORM*. https://gorm.io/. Accessed: May 18, 2023.

[9] *Next.js*. https://nextjs.org/. Accessed: May 18, 2023.

## 5 Appendix

### 5.1 Go.mod

```
module minitwit-backend/init

go 1.20

require github.com/gin-gonic/gin v1.8.2

require (
	github.com/beorn7/perks v1.0.1 // indirect
	github.com/cespare/xxhash/v2 v2.1.2 // indirect
	github.com/davecgh/go-spew v1.1.1 // indirect
	github.com/gin-contrib/cors v1.4.0 // indirect
	github.com/golang/protobuf v1.5.2 // indirect
	github.com/jackc/pgpassfile v1.0.0 // indirect
	github.com/jackc/pgservicefile v0.0.0-20221227161230-091c0ba34f0a // indirect
	github.com/jackc/pgx/v5 v5.3.1 // indirect
	github.com/jinzhu/inflection v1.0.0 // indirect
	github.com/jinzhu/now v1.1.5 // indirect
	github.com/joho/godotenv v1.5.1 // indirect
	github.com/lib/pq v1.10.7 // indirect
	github.com/matttproud/golang_protobuf_extensions v1.0.1 // indirect
	github.com/pmezard/go-difflib v1.0.0 // indirect
	github.com/prometheus/client_model v0.3.0 // indirect
	github.com/prometheus/common v0.37.0 // indirect
	github.com/prometheus/procfs v0.8.0 // indirect
	github.com/steinfletcher/apitest v1.5.14 // indirect
	github.com/stretchr/testify v1.8.1 // indirect
	gopkg.in/yaml.v3 v3.0.1 // indirect
	gorm.io/driver/postgres v1.5.0 // indirect
	gorm.io/gorm v1.24.7-0.20230306060331-85eaf9eeda11 // indirect
)

require (
	github.com/gin-contrib/sse v0.1.0 // indirect
	github.com/go-playground/locales v0.14.0 // indirect
	github.com/go-playground/universal-translator v0.18.0 // indirect
```

```
        github.com/go-playground/validator/v10 v10.11.1 // indirect
        github.com/goccy/go-json v0.9.11 // indirect
        github.com/json-iterator/go v1.1.12 // indirect
        github.com/leodido/go-urn v1.2.1 // indirect
        github.com/mackerelio/go-osstat v0.2.4
        github.com/mattn/go-isatty v0.0.16 // indirect
        github.com/mattn/go-sqlite3 v1.14.16
        github.com/modern-go/concurrent v0.0.0-20180306012644-bacd9c7ef1dd // indirec
        github.com/modern-go/reflect2 v1.0.2 // indirect
        github.com/pelletier/go-toml/v2 v2.0.6 // indirect
        github.com/prometheus/client_golang v1.14.0
        github.com/sirupsen/logrus v1.9.0
        github.com/ugorji/go/codec v1.2.7 // indirect
        golang.org/x/crypto v0.7.0 // indirect
        golang.org/x/net v0.8.0 // indirect
        golang.org/x/sys v0.6.0 // indirect
        golang.org/x/text v0.8.0 // indirect
        google.golang.org/protobuf v1.28.1 // indirect
        gopkg.in/yaml.v2 v2.4.0 // indirect
        gorm.io/plugin/dbresolver v1.4.1
)
```

### 5.2 Package.json

```
    {
  "name": "minitwit-frontend",
  "version": "2.0.0",
  "private": true,
  "scripts": {
    "dev": "next dev",
    "devnextjs": "next dev",
    "build": "next build",
    "start": "next start",
    "precommit": "npm run tsc && npm run lint && npm run build",
    "tsc": "tsc --noEmit",
    "tsc:skip": "tsc --noEmit --skipLibCheck",
    "lint": "next lint",
    "lint:fix": "eslint src --fix && npm run format",
    "lint:strict": "eslint src",
    "format:check": "prettier -c .",
    "prepare": "cd .. && cd .. && husky install itu-minitwit/frontend/.husky",
    "pretest": "npm run lint",
    "format": "prettier --loglevel warn --write \"**/*.{ts,tsx,css,md}\"",
    "posttest": "npm run format"
  },
  "dependencies": {
    "@heroicons/react": "^2.0.16",
    "@next/font": "13.1.6",
    "@types/react-dom": "18.0.10",
    "axios": "^1.3.5",
    "clsx": "^1.2.1",
    "cookie": "^0.5.0",
    "dotenv": "^16.0.3",
    "eslint-config-next": "13.1.6",
    "lint-staged": "^13.2.0",
    "next": "13.1.6",
    "next-themes": "^0.2.1",
    "react": "18.2.0",
    "react-cookie": "^4.1.1",
    "react-dom": "18.2.0",
    "react-query": "^3.39.3",
    "zod": "^3.21.4",
    "zustand": "^4.3.3"
  },
  "devDependencies": {
    "@types/node": "18.14.6",
    "@types/react": "18.0.28",
    "@typescript-eslint/eslint-plugin": "^5.55.0",
    "@typescript-eslint/parser": "^5.55.0",
    "autoprefixer": "^10.4.13",
    "concurrently": "^7.6.0",
    "eslint": "^8.36.0",
    "husky": "^8.0.0",
    "postcss": "^8.4.21",
    "prettier": "^2.8.4",
    "tailwindcss": "^3.2.6",
    "typescript": "4.9.5"
  }
}
```