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 Artifacts

Here are the artifacts of the project:

Artifact	Source
Repository	https://github.com/DevOps-CI-CDont/DevOps-CI-CDont
Frontpage	http://cicdont.live/
Kanban Board	https://github.com/orgs/DevOps-CI-CDont/projects/1
Grafana	http://cicdont.live:4000/
Kibana	http://cicdont.live:5601/

2 System's Perspective

2.1 Design & Architecture

The frontend is built using Next.js[14]. The data is fetched through a Go Gin[12] API interacting with the database through an ORM called GORM[13]. The service is deployed with Dockerized services. For logging & monitoring we run "EFK stack containers", Grafana & Prometheus. Finally, we also have a container for Nginx, acting as a reverse proxy and allowing subdomain configuration.

We chose Digital Ocean as our hosting provider, using their VM service "Droplets". Our top-level-architecture on DigitalOcean can be seen on Figure 1. The system runs on a single droplet. We're also billed for a hosted Postgres database on Digital Ocean that we use to store all data.

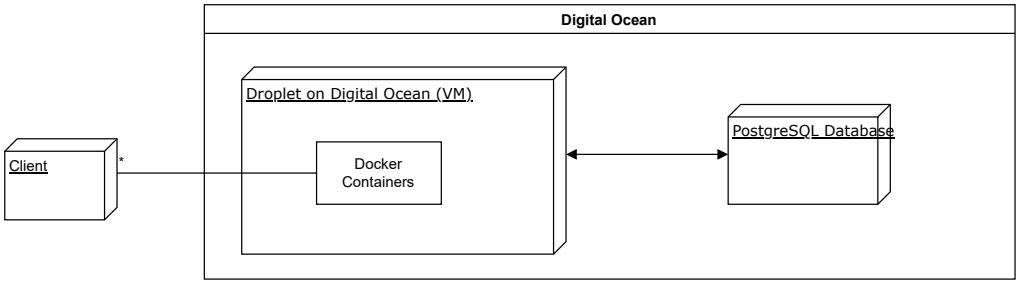

Figure 1: Digital Ocean infrastructure

An overview of the docker containers and their interactions are illustrated on Figure 2.

2.2 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.

Docker installation Our droplet installs docker through a script hosted at get.docker.com

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 built using

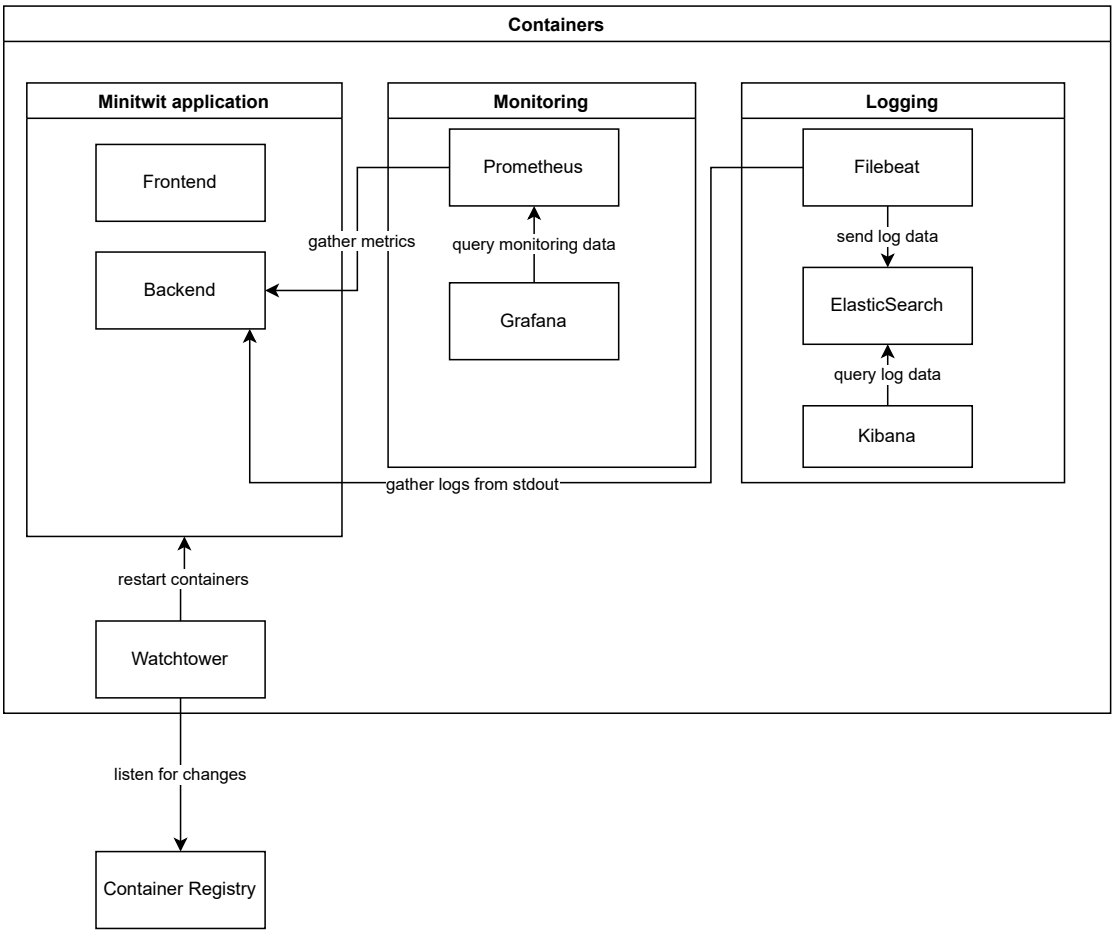

Figure 2: Container overview

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" (6.2) and "go.mod" (6.1) can be seen in the appendix.

2.3 Current state of system

To describe the state of the system, we have used SonarCloud's [4] service to review the code and quality of code. SonarCloud provides us with a series of reports, that indicate where possible problems. The results of the static code analysis is;[1]:

1. 0 bugs
2. 0 vulnerabilities

[1]Full report: https://sonarcloud.io/project/overview?id=DevOps-CI-CDont_DevOps-CI-CDont2

3. 51 code smells (naming convention breaches, some things could be constants)
4. 11 Security hotspots (eg. secrets in docker environment variables)

2.4 License & compatibility with direct dependencies

The repository is provided with an MIT license [2], which is an open-source license. Having an MIT license means we have to ensure all packages we use allow redistribution as is the case with our dependencies The list of dependencies can be seen in 6.2 and 6.1

2.5 Choice of technologies

2.5.1 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. Next.js also provides support for SEO optimizations, which is an important metric to be indexed and shown in Google searches. This is, however, not important for this project.
Having code splitting, divides the javascript 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. 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. The API routes folder removes many issues and privacy-related data, such as exposing environment 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.

2.5.2 API technology choice

We chose to use Go with Gin to make a REST-ful 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. 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, and just decided to use Gin as it seemed popular, and provides the API tooling we thought was needed.

2.5.3 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. We chose to have the database maintained by a provider because the time we felt the time we would have to spend managing a database ourselves could be spent on more important parts of the project instead.

2.5.4 Domain

We used Name.com to purchase `cicdont.live` because they have a discount for a free year through github student pack. They point to nameservers on DigitalOcean, who then redirects to our droplet's specific IP.

2.5.5 DigitalOcean

We chose DigitalOcean as our cloud provider simply because they provide $200 worth of credit for students, and it was suggested in the course.

2.5.6 Github Actions

GitHub was already used as a remote git repository and as an issue-tracking platform. As it was also suggested by the course, we found it to be an easy choice to also use their platform for Continuous Integration and Continuous deployment.

2.6 Virtualization choices

All the architecture that is needed to deploy the system, can be started from a single script in our repository. The script requires a secret environment variable(API keys from digital ocean) that will allow the creation of droplets on our Digital Ocean account. From there we use Digital Ocean's CLI tool "Doctl" to create a ubuntu22-10x64 with 4 GB of RAM Virtual Machine hosted in France. The script then installs docker on the droplet before accessing

our repository for configuration files through curl. Once all the necessary configuration files exist, we use docker-compose to run our service. Finally, we set up "iptables" that redirects http traffic (port 80) to the frontend (port 3000).

3 Process' Perspective

3.1 Developer interaction

The team met at every lecture and exercise session, where a plan was discussed on how to reach the goal of the following weeks exercises. This plan was then carried out between the developers. In order to work asynchronously, we also decided to use Discord as an online communication tool. The reason this was chosen, was due to the ease of creating a group channel, where all members could write their topics and update, whilst also being able to join voice calls and share individuals screens.
Another choice we could have made was to use Microsoft Teams, as this is also how the course provides news and discussions. We opted not to use teams, as all members were more comfortable with Discord and had accounts.

3.2 Organization of team

The team had to manage DevOps practices, Frontend and backend development. The team was not built with generalists, but rather specialists in the frontend and backend departments.

- **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), along 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.

3.3 Full stage & tool description of CI/CD Chains

3.3.1 Continuous integration

We built a continuous integration(CI) workflow[6] that would serve as a quality gate when pushing new changes. 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) "Runner" is spun up, by Github getting everything it needs as secrets through GitHub's built-in "Github secrets"[11].
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. DigitalOcean's command line tool "Doctl"[10] is installed, and our new image is pushed to a container registry on DigitalOcean.

After this job, our API is running on the "runner" and testing can begin. Testing happens in a new job, which also installs doctl, and then it pulls the "testing-image" the other job uploaded, runs the image and runs all go tests. Finally, a static code analysis job runs, that installs all dependencies and runs typescript type checking "tsc", "eslint", and "go vet". If any of these steps fail, the job fails and the workflow as a whole exits with a "Failure".

3.3.2 Continuous deployment

To support frequent delivery of new versions of Minitwit we also created a continuous deployment (CD) workflow on Github actions. The workflow runs any time the CI workflow completes successfully. The workflow then builds the backend and frontend images (using the necessary secrets stored in GitHub secrets), before installing doctl and pushing the images to the container registry.

The DigitalOcean Droplet needs to know when an update exists for one of the images. For this purpose, we use "Watchtower"[9]. Watchtower monitors running docker containers and watches for any updates for images used in running containers. If any images are updated watchtower will pull that update and restart the container using the new image. Watchtower is configured to look for image updates every 60 seconds. The combination of the previous workflow and watchtower means that any new code on the main branch (that passes the quality gates) also updates our "production" environment.

3.4 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. Monorepos are 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 microservices a polyrepo may seem the natural choice, however, the monorepo allowed us to create a unified and automated CI/CD pipeline, that can avoid many issues associated with polyrepos, such as not being in the same state [7].

3.5 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 trunk-based strategy [5]. This strategy works well for small teams, as it focuses on keeping the master in a deployable state. Commits to master are not frowned upon as the aim is that any fatal errors will be caught by CI. For bigger changes, feature branches can be made but they must be shortlived and rebased if development takes too long.

3.6 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 contained all the tasks that we found would be realistic/interesting tasks for the project if it were to continue.

3.7 Monitoring

For our monitoring setup we utilize Prometheus & Grafana. A Prometheus container scrapes our backend api for metrics on a fixed time interval, and the metrics are visualized by grafana which queries Prometheus. In Grafana we have a single main dashboard with two graphs, specifically graphs for memory usage (%) and endpoint counter, which can be seen on Figure 3.

For the endpoint counter we gather the metrics using middleware, that gets executed after every request. Here's how we use Gin to register endpoint usage:

```
...
Router.GET("/metrics", gin.WrapH(promHandler))
Router.GET("/mytimeline", getTimeline, incrementCounter(m, "/mytimeline"))
Router.GET("/public", getPublicTimeline, incrementCounter(m, "/public"))
...
```

The *incrementCounter* function is a helper function, which returns a "gin.Handlerfunc", which will increment the appropriate endpoint count once triggered e.g. "/mytimeline".

For recording the memory usage, we have a "goroutine" in the background which records the

current memory usage every 10 seconds. Prometheus scrapes our API for this information from the "/metrics" endpoint at a fixed interval.

Figure 3: The grafana dashboard after executing some requests to the /public, /login, and /register endpoints.

3.8 Logging

Our logging system consists of an EFK stack, i.e. elasticsearch, filebeat and kibana. Filebeat captures and ships the logs from standard out to elasticsearch, and kibana is the user interface that allows us to retrieve the logs. As we did not implement docker swarm/load balancing, there was no log aggregation from different sources.

3.9 Security assessment

The exercises for lecture 9 was to do a security assessment of the minitwit application. Our peer group sadly did not leave an assessment of our website, which is why we did it ourselves. For this, we used Skipfish [1]. Skipfish is a security reconnaissance tool, that gives an output of potential security risks. Running this tool on our website didn't bring any interesting results [2].

[2]Skipfish results: https://github.com/DevOps-CI-CDont/DevOps-CI-CDont/tree/main/CICdont3.html

A security assessment of our website:

1. **HTTPS**: The website does not support HTTPS. For a short moment, we had a SSL certificate set up through Cloudflare, such that all traffic was considered safe. This caused an issue for all the API calls, which were hosted on a port that was not on HTTPS, but rather HTTP. HTTPS states that you cannot make calls to non-secure HTTP sites, meaning that all our API calls failed. Due to time restrictions, we were unable to fix this issue, which is why the website is still hosted on HTTP.

2. **SQL injection**: The frontend handles user logins and signup and more through input fields, which are sent directly to the backend API. In the backend, strings are never executed directly against the database, instead, an ORM interface sits between the user and the database, this makes it impossible for the user to inject SQL.

3. **Packages**: When dealing with external packages and dependencies, we may be subject to a zero-day exploit. If there is discovered an XSS, prototype pollution or other kinds of attacks in a package, then we are vulnerable until a fix is created. For npm packages, we can run an audit, to check state of direct and indirect dependencies. Running an audit as of the handin date, there are no security risks with external packages.

4. **Authentication**: One of the biggest risks in our application is the authentication system. Currently, when you login in the frontend, the backend sends your user ID back which is then set as a `user_id` cookie. This is not optimal, as any user can login as any other user, by simply changing the `user_id` cookie. A better way of doing this would be to implement JSON Web Tokens (JWT) [3], but again, due to time restrictions, this was discarded.

3.10 Applied strategy for scaling and load-balancing

Our scaling has only been vertical: increasing resources for CPU, RAM, and Disk on a single Virtual Machine (Droplet). It has also been manual, meaning we have to notice resource usage on the droplet and go into DigitalOcean to upscale resources (during which the Droplet must be turned off). We had configured some resource alerts in DigitalOcean (Memory Usage percent and CPU utilization percent, sending an email if either is above 85% for 15 minutes or longer).
We didn't implement Docker Swarm as a scaling/load-balancing measure.

3.11 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.

4 Lessons Learned Perspective

4.1 Maintenance burden with one Virtual Machine

We started off with our cloud infrastructure consisting of a single Virtual Machine. We have learned how much has to be managed to be able to run a Linux Server with plenty of dockerized services available to the world.
Some notable tasks this has included:

1. iptables configuration
2. resource management
   (a) Upgrading, RAM, CPU, Disk space
   (b) Configuring Cronjobs to prune docker images (clean up disk space)
3. Managing secrets and .env values on the machine.
4. Authorizing SSH access for all group members.
5. Configuring watchtower to detect new images from our CD-chain, and then restarting relevant services on the droplet.

Toward the end of the project, we developed a script to deploy a Droplet with all the important configurations being automatically set up on a Virtual Machine. So in principle, we can super-quickly deploy our architecture as it was.

4.2 DevOps style

We have had some channels of relatively quick feedback on problems in operation, such as resource alerts, uptime alerts, the error status page from Helge - which have all contributed to prompt us as developers to fix our errors quickly.
In software, there are a **lot** of operational tasks that one could spend a lot of time on if doing so manually. We have quickly seen the value of testing, building, and deploying automatically. It has been a valuable experience to build Github Actions workflows to serve as automation with quality gate steps (static code analysis + tests), taking steps towards ensuring "bad code" doesn't reach production. When you have good quality gates and a lot of automation, deployment is not scary.

4.3 Go Memory issue

For a big part of the project, our virtual machine would continually increase its memory usage until it eventually crashed, because a docker container for our APIs kept growing in memory usage. We created this issue for it: https://github.com/DevOps-CI-CDont/Dev

Ops-CI-CDont/issues/58, where Silas continually updated with comments on diagnosing the reason for the problem.
This issue meant that we would miss all requests from the simulator until we realized it had crashed and we could start it up again. The issue turned out to be a common pitfall in Go (specifically in the "net/http" package in go). Every time we received a request from the simulator we sent an API request to our backend, however, we didn't close the corresponding "Response Body" as we assumed that was done automatically in any garbage-collected language. This meant that response bodies that were stored in RAM kept racking up until there was no more RAM to store them in [8]. The solution was to simply defer a call to response.body.close(), such that once we were done with any HTTP response the memory would be freed again. This commit fixed the problem: https://github.com/DevOps-CI-CDont/DevOps-CI-CDont/commit/2cdcb30858dd25fe5851d41f8c5547749718820d.

4.4 Refactoring an old project

When starting on a old project with legacy code, there are a lot of things to be aware of. We want to keep all the same functionality as the old project provided, luckily there was already a good test suite that could be used to see what input leads to some expected output. Also, the whole original minitwit was written in Python, which we could all understand fairly well. One of the lessons we learned in this process, is that it can be tough to refactor code, whilst learning a new language, as we did with Go. Go was completely unfamiliar to us, meaning we did not know a lot of the pitfalls that the language had.
Furthermore in Next.js, even though this was not necessarily a new language to us (JavaScript), we tried implementing SSR (a Next.js feature) which is popular for optimizing SEO, without experience in this framework. All of these factors contributed to experiencing time sinks, that we would not have had if using languages/frameworks that we were more familiar with.

5 References

[1] Google Inc. *Skipfish*. Accessed: May 23, 2023. 2012. URL: https://code.google.com/archive/p/skipfish/.

[2] *MIT License*. https://opensource.org/licenses/MIT/. Accessed: May 23, 2023. 2013.

[3] Auth0. *JSON Web Tokens (JWT)*. Accessed: May 23, 2023. Auth0. 2021. URL: https://auth0.com/docs/secure/tokens/json-web-tokens.

[4] SonarSource. *SonarCloud*. Accessed: May 23, 2023. SonarSource. 2021. URL: https://sonarcloud.io/.

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

[6] 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.

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

[8] CloudImmunity. *Gotchas and Common Mistakes in Go (Golang)*. n.d. URL: http://devs.cloudimmunity.com/gotchas-and-common-mistakes-in-go-golang/index.html#close_http_resp_body.

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

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

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

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

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

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

6 Appendix

6.1 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
)
```

6.2 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"
  }
}
```