DevOps, Software Evolution & Software Maintenance

Course code: KSDSESM1KU
The Eagles (Group H)

Trond Pingel Anchær Rossing (trro@itu.dk)
Roman Zvoda (rozv@itu.dk)
Rasmus Balder Nordbjærg (rano@itu.dk)
Daniel Spandet Grønbjerg (dangr@itu.dk)
Jan Lishak (jlis@itu.dk)

May 21, 2024

Contents

1 Introduction

The project focuses on building and maintaining a mock version of Twitter called **MiniTwit** by applying various DevOps techniques such as automation, cloud deployment, scaling, maintainability, monitoring, testing, and others. The initial web application, which was outdated with the latest technologies and was not following any standard practices, was rewritten from the ground up and gradually equipped with automations and improvements so that it became able to process a high load of incoming requests to the app.

2 System perspective

2.1 Design and architecture

The documentation of the architecture differentiates between the old and the new architecture. The old architecture is deployed to a single Digital Ocean VM and spun up using docker compose. The new setup utilizes 2 digital ocean VMs hosting both worker nodes and the control plane.

2.1.1 Old architecture

The below diagram shows the deployment diagram for the different components in the old architecture. At the end of the project this setup is still in use however the system is in process of being migrated to the new architecture explained later in the report.

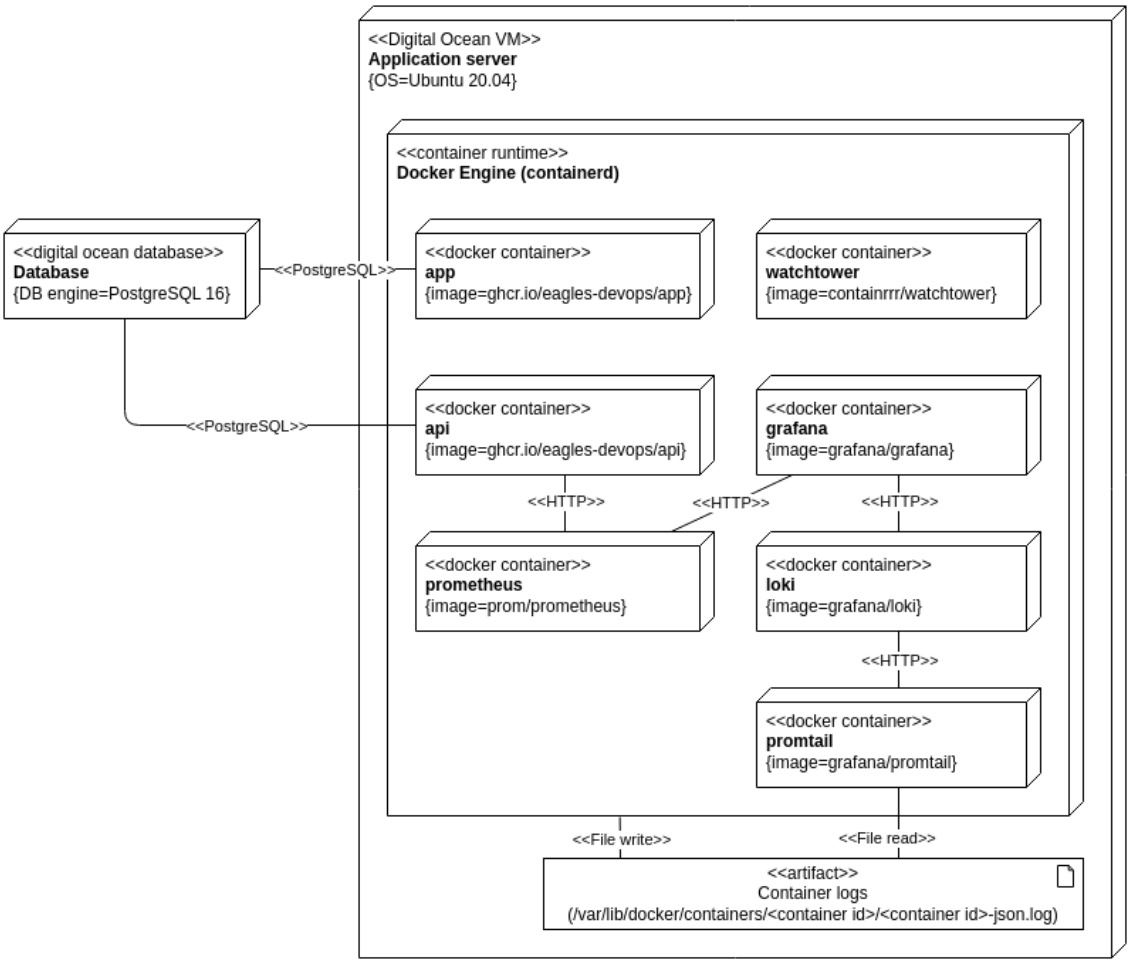

Figure 1: Deployment View: Old architecture

The 2 main infrastructure components of the system are the the database and the server. The database is a managed PostgreSQL database from Digital Ocean. The server is Digital Ocean droplet sporting 1GB 25GB SSD storage.

Running on the server is docker compose which contains the following services:

- app - The minitwit web app
- api - The minitwit API used by the simulator
- watchtower - A service that updates the app and api to the most recent container image
- prometheus - Scrapes the metric data exposed by the api
- promtail - Reads container logs from the host system
- loki - Receives logs pushed from promtail and indexes meta data
- grafana - The Grafana instance for displaying all logs and monitoring data. Pulls data from Loki and Prometheus.

The flow of an incoming user request is fairly simple and looks something like this:

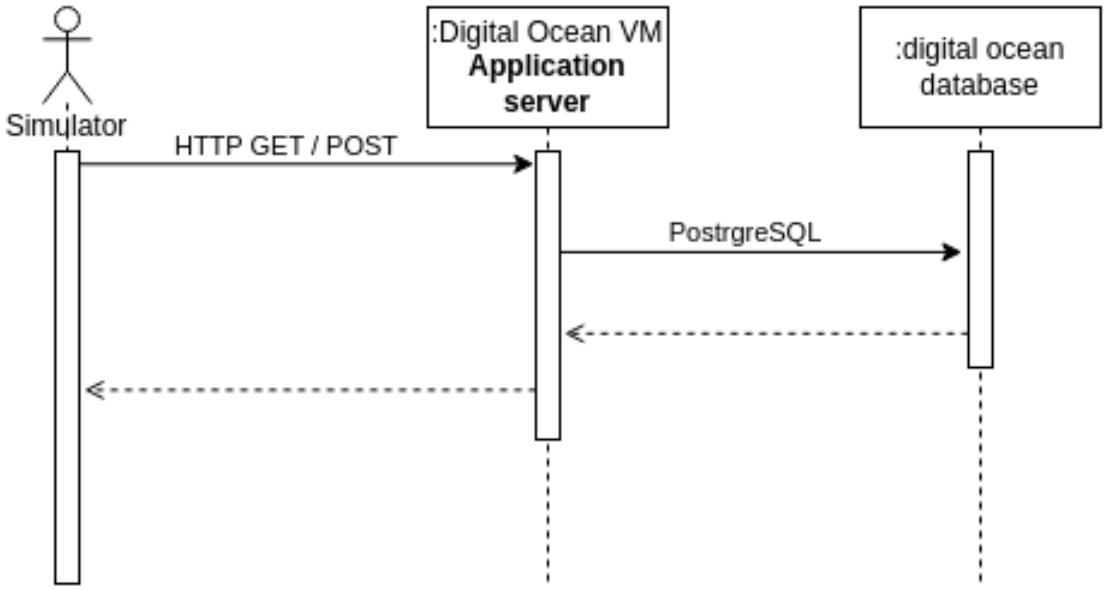

Figure 2: Sequence diagram: Old architecture

2.1.2 New architecture

Below diagram shows the new architecture hosted accross 2 Digital Ocean droplets, running Kubernetes:

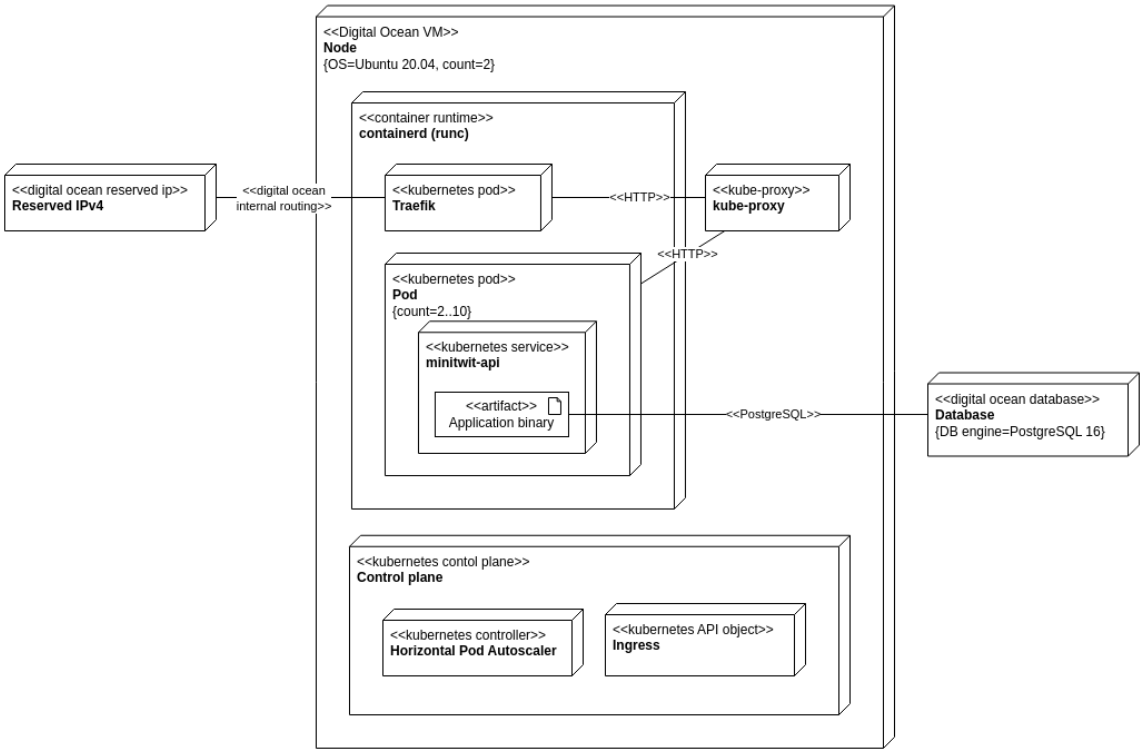

Figure 3: Deployment View: New architecture

The new architecture uses Traefik as a load balancer to route between 2..10 pods hosting the minitwit-api service. The ingress configuration defines the Traefik pod as the entry for incoming traffic. The Horizontal Pod Autoscaler (HPA) contains rules for when to increase/decrese the number of pods. The kube-proxy ensures routing between pods hosted in the cluster.

Generally the flow of a user request looks something like this:

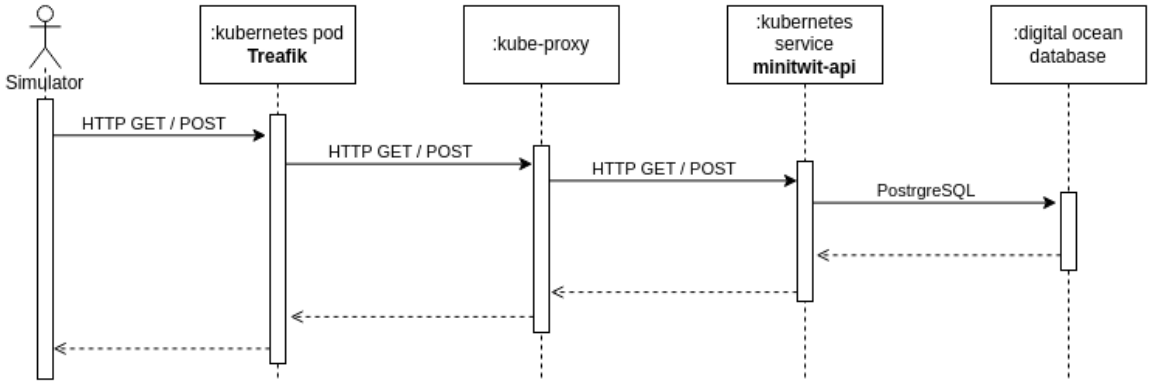

Figure 4: Sequence diagram: New architecture

The go library used for session handling uses securecookie for storing session data. This means that sessions data is encrypted at server-side but stored at client-side. This means that the session persists across pods as illustrated in below diagram:

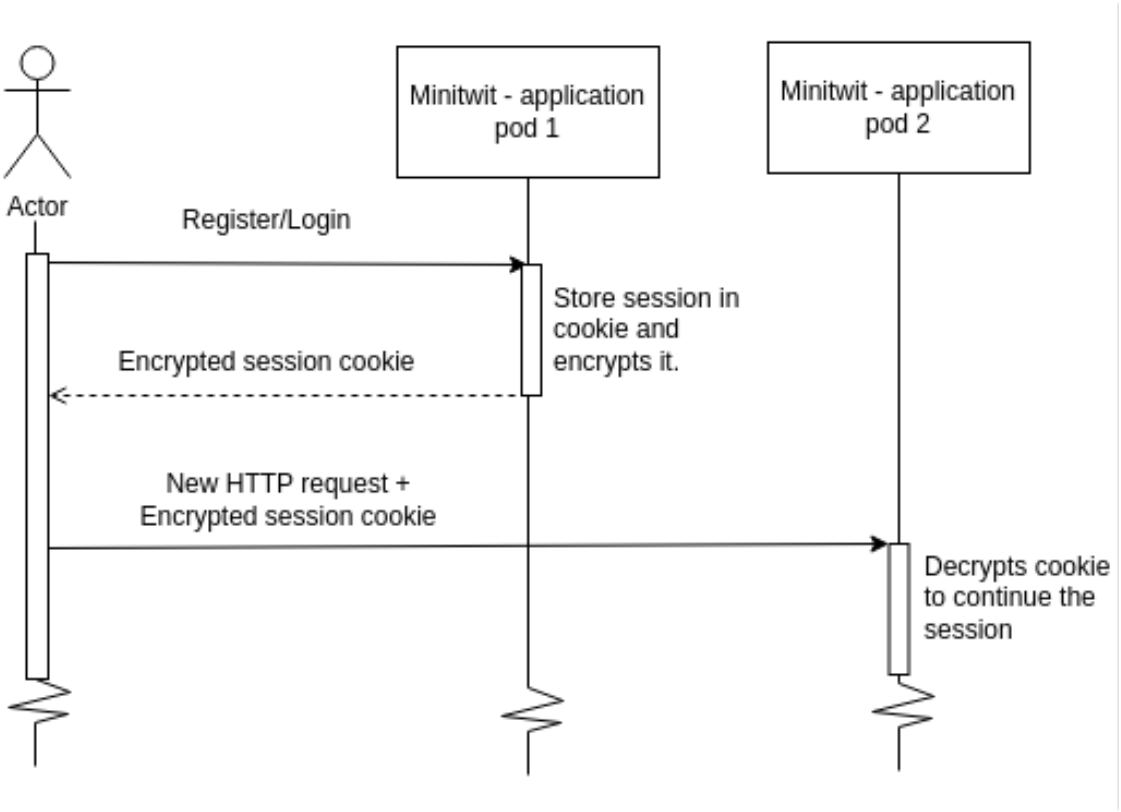

Figure 5: Sequence diagram: Cookie session

2.2 Repo structure

The diagrams in this section shows the repo structure in terms of files and folders. It is only showing files and folders that were deemed significant or interesting in understanding the repo.

Following is an overview of the top-level folders in the repo. The top-level folders are color coordinated which is also reflected in the following diagrams that dives deeper into each folder.

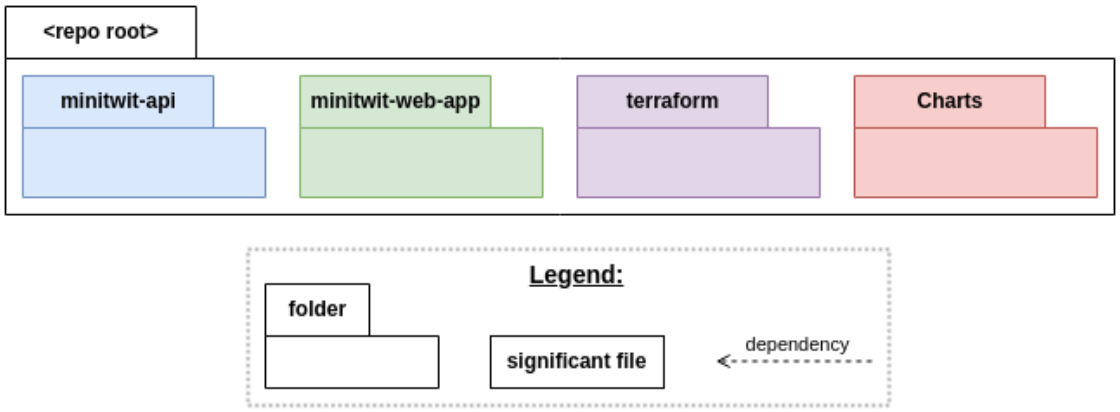

Figure 6: Module View: Repo overview

The *minitwit-api* folder contains go code that hosts the API.

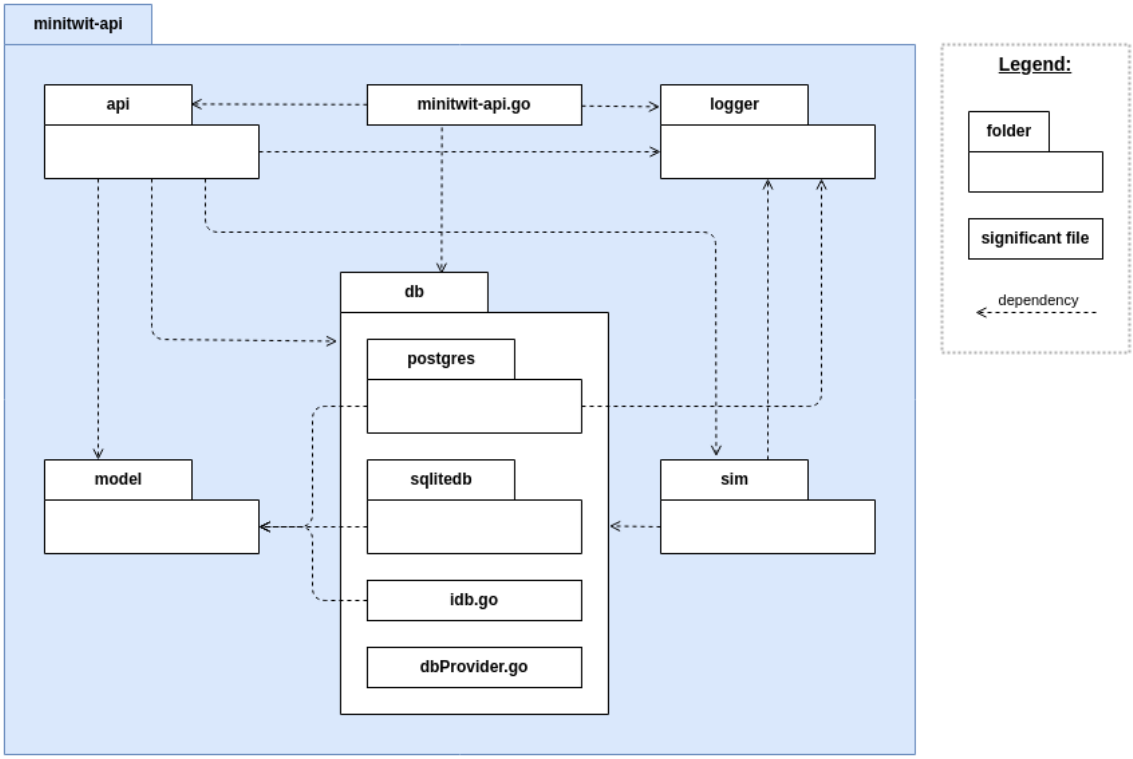

Figure 7: Module View: API

The *minitwit-web-app* contains go code for the web application.

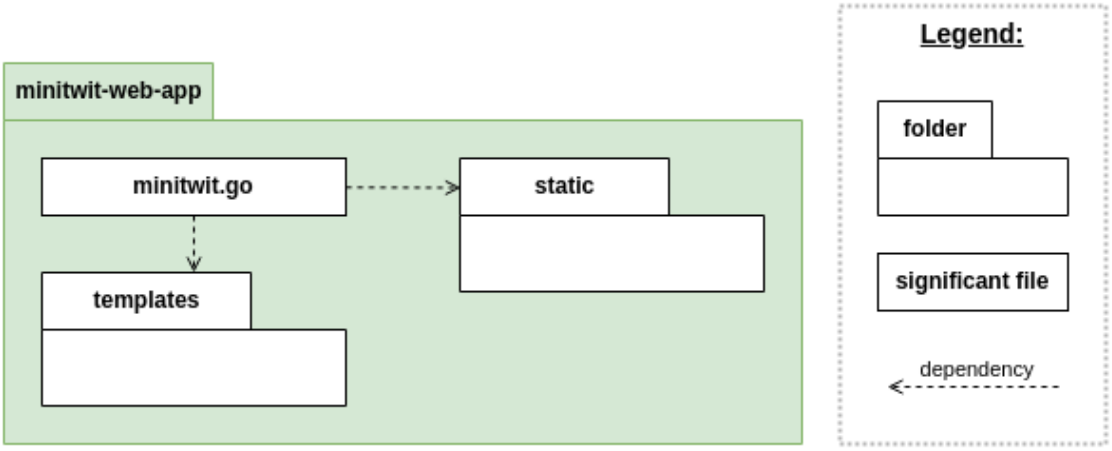

Figure 8: Module View: Web App

The *terraform* folder contains the project's terraform templates and the *bash* scripts to configure the VMs. The below diagram shows the folder mainly related to the old architecture which is why the *k3s* is not expanded.

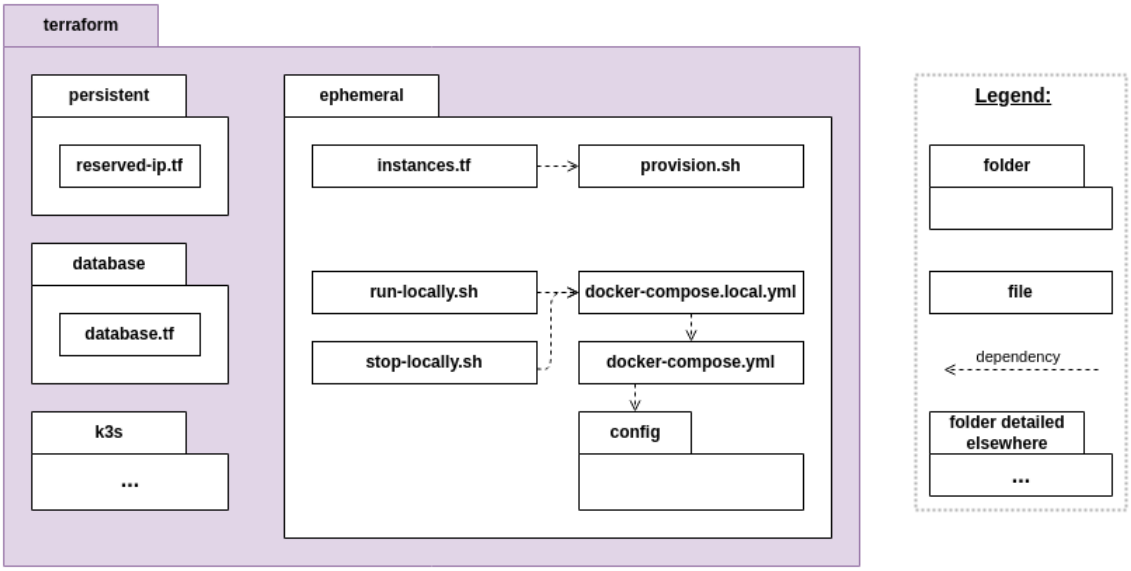

Figure 9: Module View: Terraform old

The below diagram shows the *terraform* and the *Charts* folders which were related to the Kubernetes setup. The *terraform/k3s* folder contains the setup for the VMs (nodes) and the *Charts* folder contains the helm chart for the Minitwit service

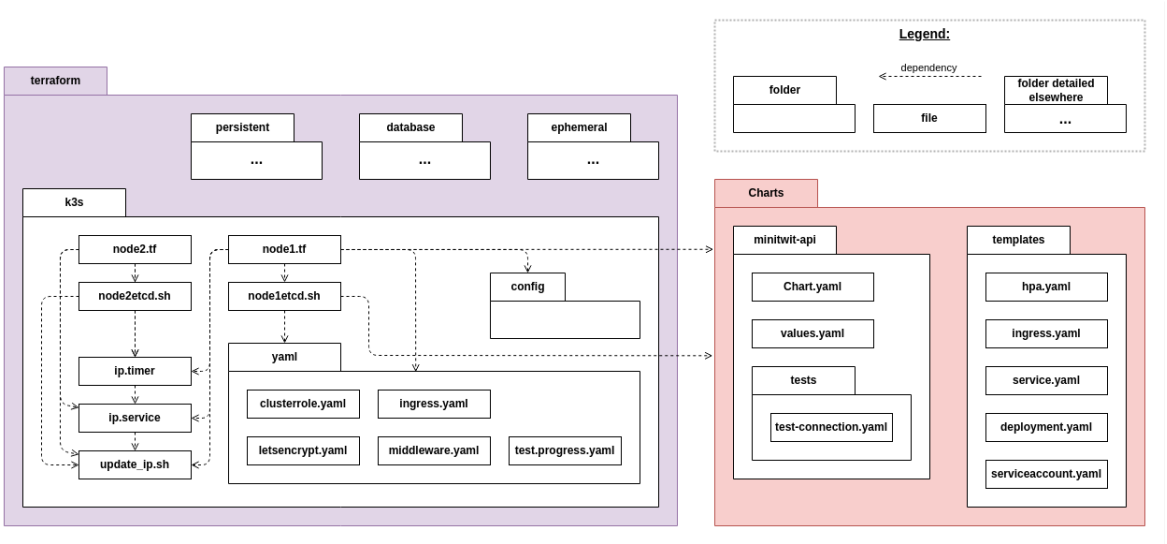

Figure 10: Module View: Terraform new K3S

2.3 K3S

The API is now running on a lightweight Kubernetes cluster - k3s. This cluster spans two server nodes. The cluster is spun up from scratch using terraform, and the infrastructure takes about an hour to spin up, as it needs to wait for dns propagation to be able to confirm domain ownership for the SSL certificate. Configuration and secrets are deployed in the cluster as part of the setup process.

When Kubernetes was chosen rather than an arguably easier option like docker swarm, it stems from Kubernetes being the industry standard. Docker swarm is nice for showing simple scaling of containers, but Kubernetes seemed like an interesting challenge. The complexity of Kubernetes has proven quite time consuming, and we didn't manage to move as many things as we would have liked to the cluster.

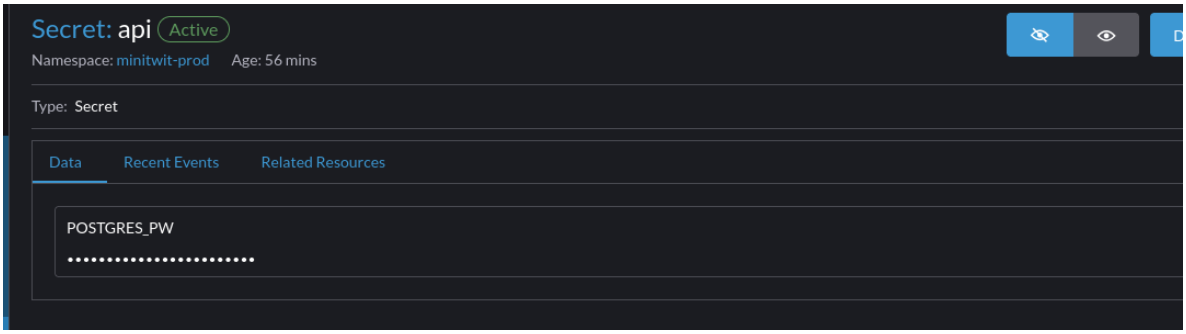

Figure 11: Api Secret

Rancher is running on top to provide a nice UI for management.

Let's Encrypt is used for SSL certificates and is automatically created/renewed for deployments.

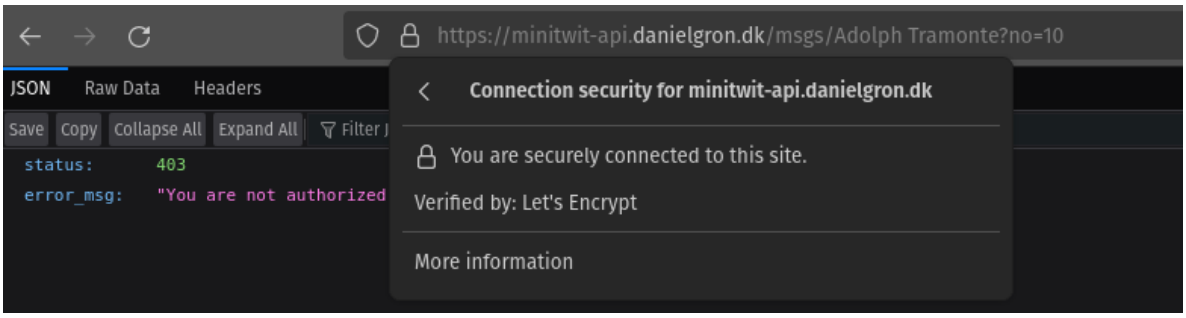

Figure 12: Https connection

2.4 Scaling and upgrades

When deployed on the Kubernetes cluster, the api will be deployed with 2 instances using blue/green deployment, which ensures no downtime when deploying. It uses a very basic health check that requires the database connection to be established. The ingress will not point to the new instance until this check passes.

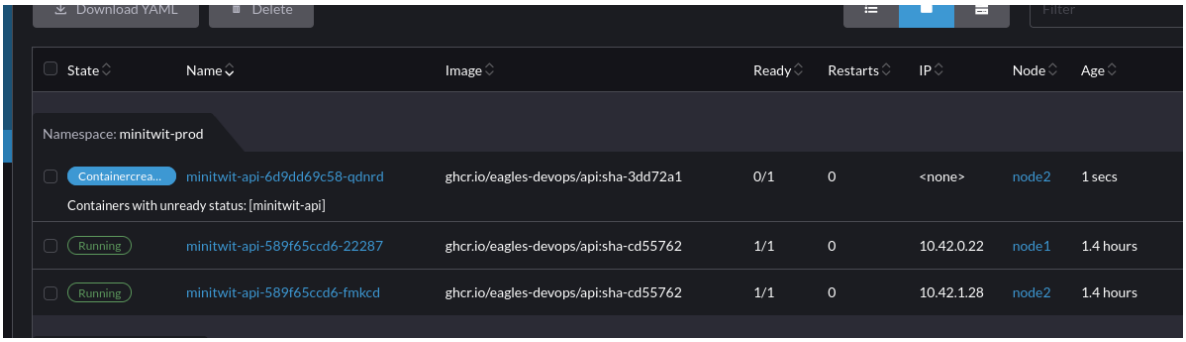

Figure 13: Blue/Green rollout

Once a container with the new image is up and running a coresponding pod based on the old image is terminated.

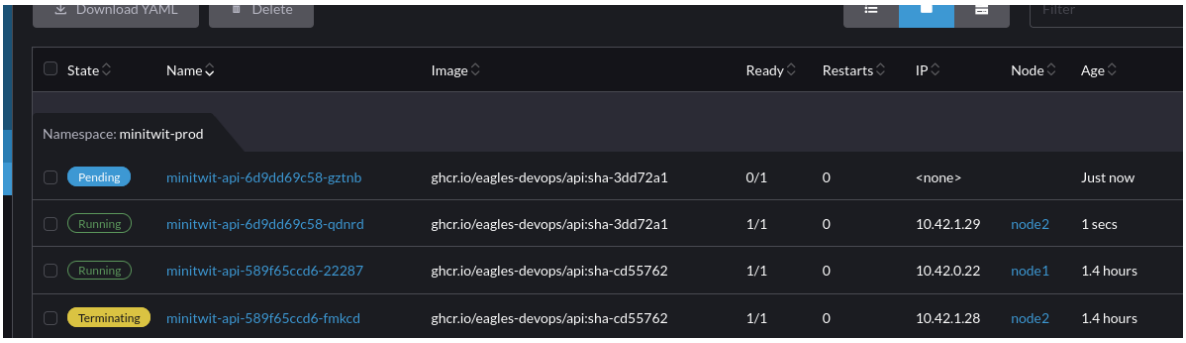

Figure 14: Container termination

Autoscaling is enabled for the deployment, meaning that if load for a pod exceeds the treshold set, an

extra pod is started. Currently the main bottleneck is the database, meaning the application itself does not experience a load that actually instantiates new pods.

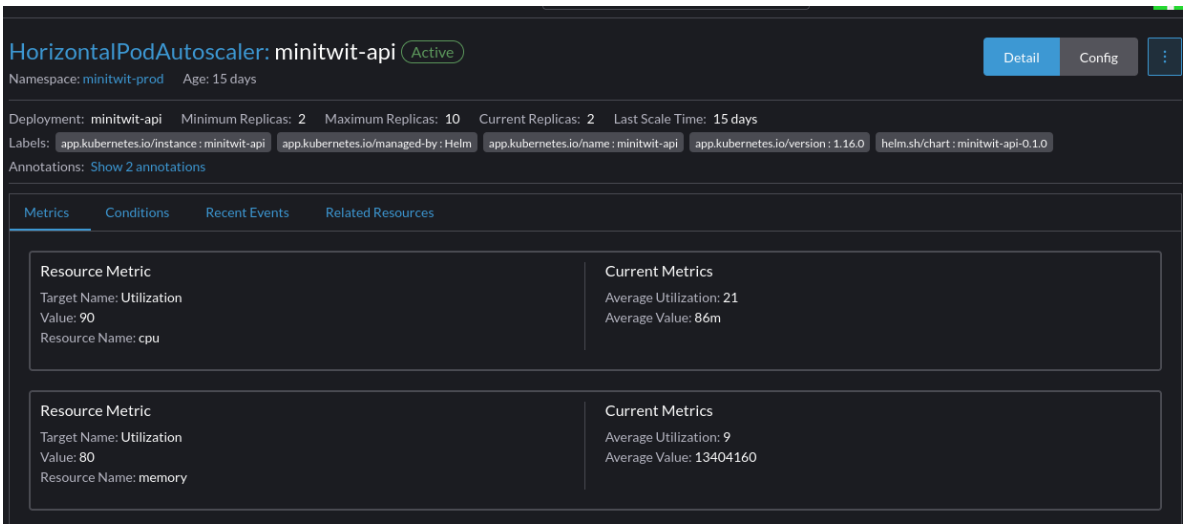

Figure 15: Horizontal autoscaler

However by adding artificial stress on the cpu the autoscaling can be demonstrated:

Figure 16: Autoscaled pods

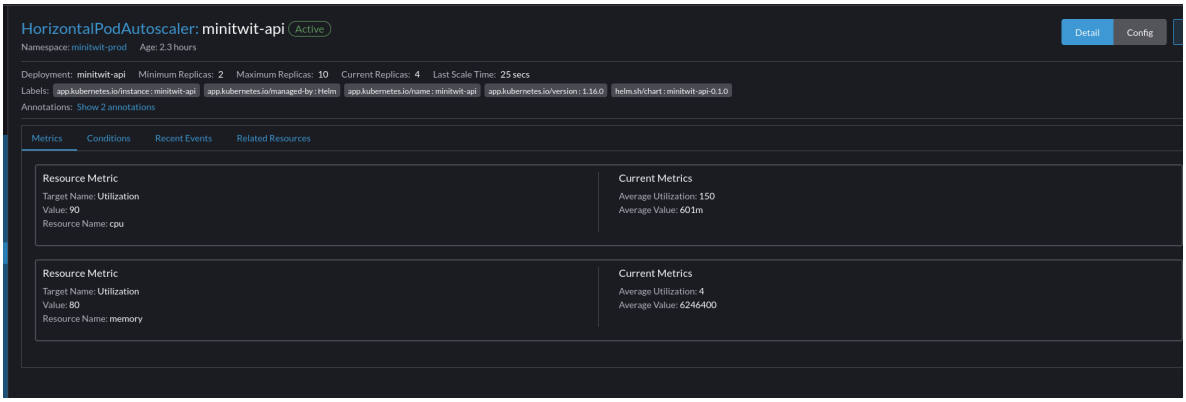

Figure 17: After autoscaling

2.5 State of the system

This section will break down the current state of the system looking through multiple components and their current status. Such approach allows us to provide sufficient report and locate which sections of the project require more work. Before, lets show some general data about the application to get an idea of the traffic. MiniTwit application has processed **14,5 million** request during its up-time with somewhere above 1 million of reported errors. This makes 6% error rate.

2.5.1 Code Quality Analysis

SonarQube and CodeClimate were used to determine our code quality. Based on the last provided analysis from SonarQube our code seems to be secure with no security concerns. in terms of reliability, the code is proven to have a stable code base where most of the issues are related to other datetime variable interpretation than SonarQube is advising to use. Maintainability sections show the most issues with 87 recorded. Our code has a lot of error print statements which can be changed into constants. This would make the maintability part of the code much easier.
To summarize; our code base would appreciate some minor adjustments but none of the aforementioned concerns create potential harm to the codes stability and readability.

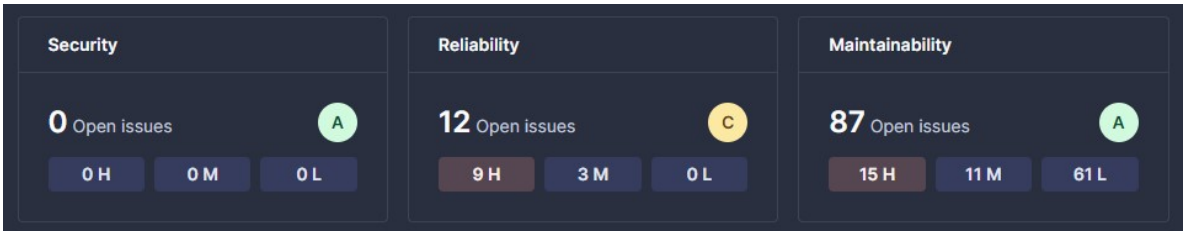

Figure 18: SonarQube general stats

2.5.2 Dependency scan

The project utilizes 100 dependencies based on the dependency report made by Snyk where there are 3 dependencies currently vulnerable to SQL injection. GitHub dependency report shows only 63 dependencies reporting similar issue regarding SQL injection vulnerability in some of the dependencies. GitHubs *dependabot* created PR with needed update of the vulnerable dependencies which should resolve the issues when merged into main.

3. Security Analysis
Static code analysis already showed us code quality when it comes to security point of view. In this field our projects shows good score. The application does not expose any vulnerable secrets which can be used to access any parts of the system. Our vulnerable information such as login details, SSH keys and other information are stored either in GitHub secrets or we have an .env file which each member of the group store on their local machine. Sharing such sensitive information between developers is done via USB drive or sharing them through BitWarden. Moreover, when potential problems occur, GitHubs Advanced Security bot will create an alert and block the open PR.

4. Test coverage
Application has different sets of test - end-to-end, simulator tests, API tests as well as linters. Even with these tests in place we do not achieve 100% code coverage and some errors may slip through. Before every merge into main we did manual tests as well to catch bugs or other errors by hand. This method is not suitable in the long run. In the future, projects would require some time into making more test cases as well as different focus test sets.

2.5.3 Metrics, Logs and Dashboards

Application has monitoring running on Grafana utilizing Loki and Prometheus as the data sources. Monitoring an application can be a huge project itself when done properly and in detail. Currently our application monitors the basic data which we were using to estimate the application performance. We can divide the data into 2 sections. First one is focus on more technical parameters which help the developers to asses the current errors or any other potential problems. Main monitored factors: failed requests, request duration, database read/writes. Second section are data related to business which can be easily understood by non-tech person. This includes number of users registered, amount of requests, number of messages and overall application status.

Application and all of its functionalities work as expected also under higher load of upcoming requests to the server. Application uptime was satisfactory except one major outage happening from *25/03 01:30* till *28/03 20:30* caused by a memory issue error on the VM and was not spotted for few days. In a real life scenario such outage would be unacceptable and should trigger alerts and other security tools to inform the developers about downtime.

2.6 Dependencies

Name	Description	Link
Golang	Statically typed, compiled high-level programming language	https://go.dev/
crypto	Package supplying different cryptography libraries for golang	https://golang.org/x/crypto
net/http	Package used to make HTTP requests	https://pkg.go.dev/net/http
Gorm	easy to use ORM library for Golang	https://gorm.io/gorm
Grafana	Open source analytics and monitoring solution used for database	https://grafana.com/
Mimir	long term storage for grafana data	https://grafana.com/oss/mimir/
Loki	Loki is a horizontally scalable, highly available	https://grafana.com/oss/loki/
Prometheus	open-source software for monitoring webapps	https://github.com/prometheus/
xxhash	Golang implementation of the (fast) 64-bit xxHash algorithm	https://github.com/cespare/xxhash
Gorilla	Package that supplies different tools for developing web-applications in golang	https://github.com/gorilla
Gorilla/mux	Package for request routing	https://github.com/gorilla/mux
Docker	System for deployment, containerized applications and development	https://www.docker.com/
Kubernetes	open source system for automating deployment, scaling, and management of containerized applications.	https://kubernetes.io/
Rancher	open-source multi-cluster orchestration platform	https://www.rancher.com/
Letsencrypt	free, automated, and open certificate authority used for SSL certificates.	https://letsencrypt.org/
zap	Package for fast logging in golang	https://github.com/uber-go/zap
SonarQube	open-source platform developed by SonarSource for continuous inspection of code quality	https://www.sonarsource.com/
Codeclimate	system that helps incorporating fully-configurable static analysis and test coverage data into a development workflow.	https://codeclimate.com/
pgx	PostgreSQL driver with toolkit for GO.	https://github.com/jackc/pgx/v5
pq	postgres for Go's database package	https://github.com/lib/pq
go-sqlite3	sqlite3 driver for Golang	https://github.com/mattn/go-sqlite3
Logs	Visualize an entire stack, aggregate all logs into structured data, and query everything like a single database with SQL.	https://betterstack.com/logs

3 Process' perspective

3.1 CI-CD

3.1.1 Pull request tests

When an issue is resolved and ready to be merged into main, a pull request is opened with the code changes. Automated testing using GitHub Action is started right after the pull request is created. In the meanwhile the pull request is available to be reviewed by a member of the project. The prerequisites for a pull request to be merged is passing all tests, passing the quality gate of the static analysis tool and having at least on approval.

3.1.2 Deployment

When the pull request is completed, the changed are merged to main triggering the ci-cd workflow with the following stages:

- Build docker image
- Push docker image to registry
- Deploy to K3S with Helm

3.2 Other workflows

3.2.1 Automated releases

Minitwit is released every Thursday at 21:50 using automated releases. The Github Actions finds the latest tag, increments it and creates new Github release.

3.2.2 Assign issues to project

To keep our opened issues up to date with our Kanban board, a Github Action periodically checks for new cards and automatically creates issues for them.

3.2.3 Automated linting

There are three different linters each focusing on a different area of the codebase. Static analysis tool, docker files linter and source code checker.

3.2.4 Report PDF generation

Also as required there is a workflow for generating a PDF which takes all *.md* files using Pandoc from the report folder and combines them into a single file.

3.3 AI chat bots

AI tools such as ChatGPT, Gemini, Opus-3 were used during the development stage. These tools were useful in many different cases. Usage helped us solve the errors much faster by providing us with clarity of the error messages. Another benefits such as code refactoring, topic explanation, and providing us with new approaches/tool to implement for the given problem. Usage of LLMs sped up the work and saved us a lot of time.

3.4 Monitoring

3.4.1 Metrics

For monitoring we use Prometheus with Grafana. We do so by incrementing gauges or vectors whenever an event has successfully occurred. Currently the system is configured to monitor these values:

business related data - amount of users getting created - amount of new followers on the platform - amount of new messages posted - total amount of reads and writes made to the database between releases

developer oriented data: - amount of failed database read-writes - connection to the database - successful / failed HTTP requests

Monitoring these gives us an insight to the extend of traffic passing through our API. For ease of access to the monitored data and for visualization, the group uses Grafanas dashboards.

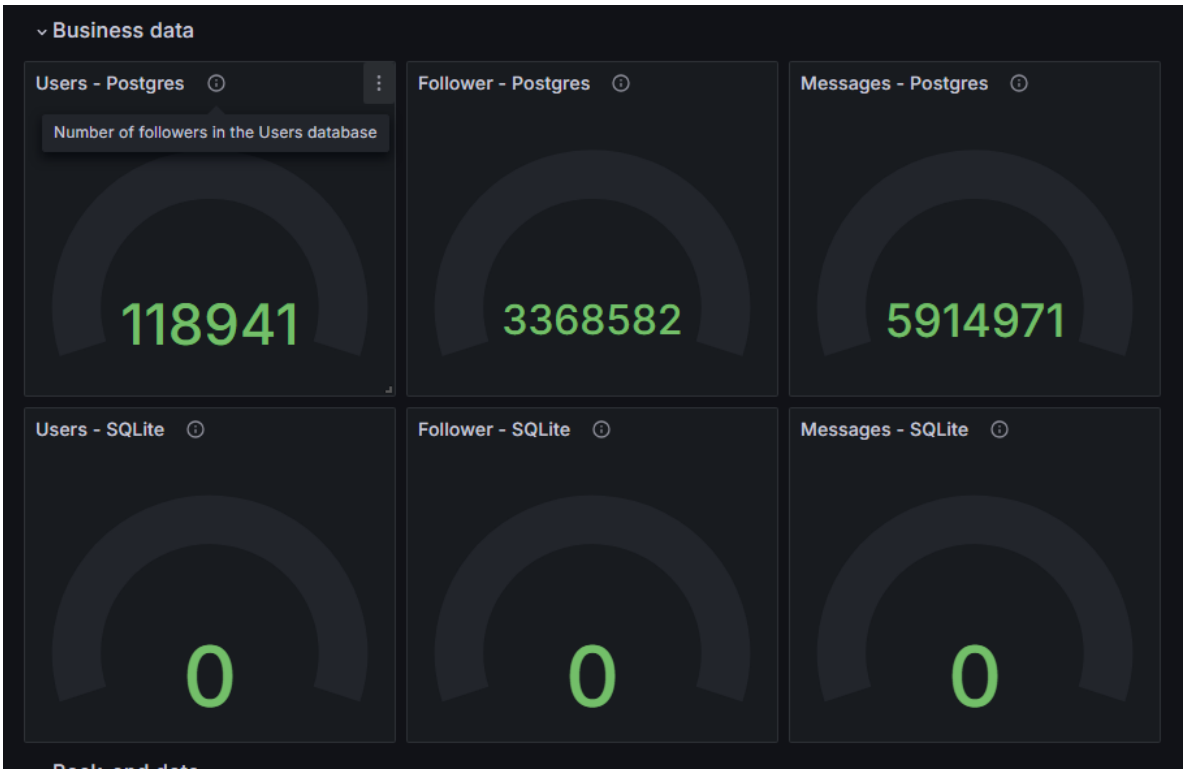

Figure 19: Grafana Business data monitoring

3.4.2 Logs

For each action in our system a log entry is created. There are different categories of logs such as info, warn and error. Most of our logs are infos, however if a process fails it will be marked as an error which allows us to easily filter and find issues.

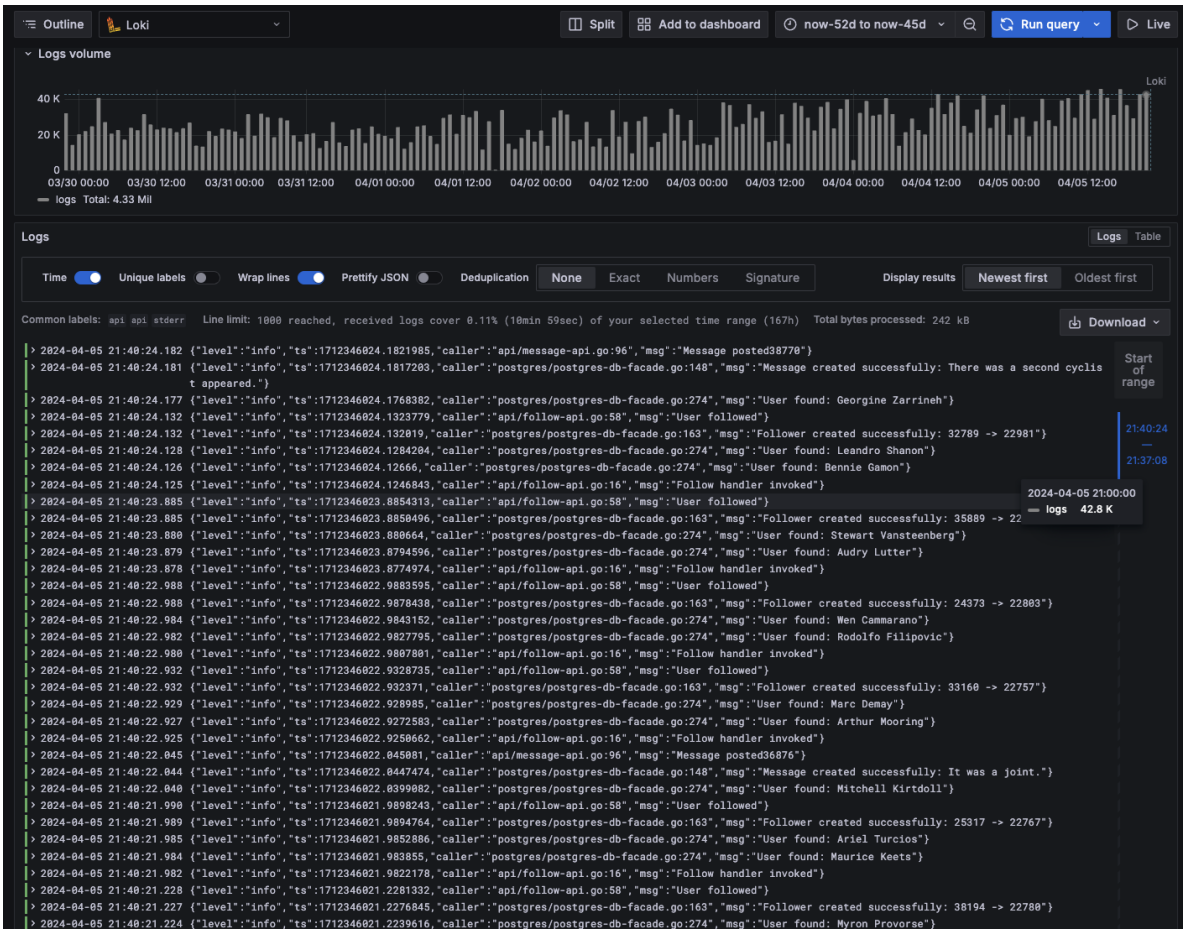

Figure 20: Info logs

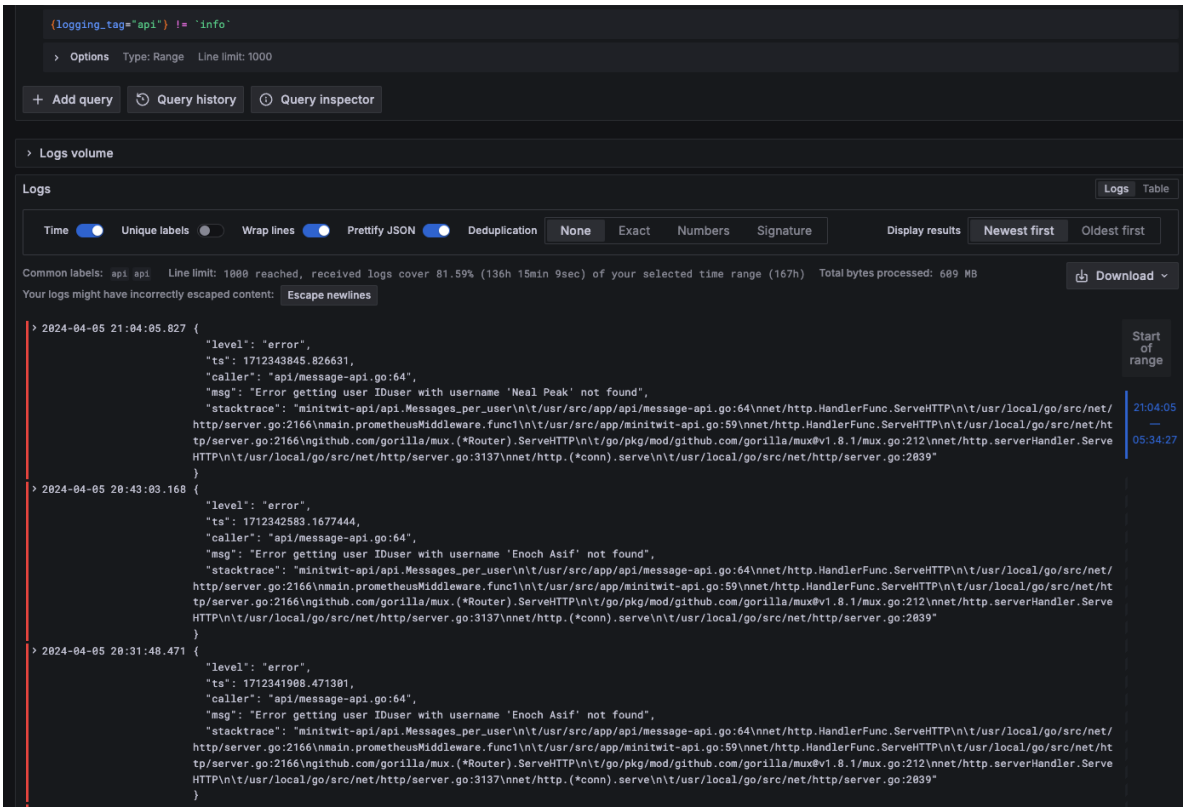

Figure 21: error logs

The logs are first created using ZAP library that that uses a common json format. Both API and App writes logs to standard output and error output. Since all our component run in docker it is easy to collect all our logs at a single place. We use Promtail that is connected to the docker engine which periodically reads logs and ships them to Loki. Grafana uses Loki as source and provides us with an option to execute queries on the logs.

It is important to note that we, due to time constraints, did not migrate our logs when moving to Kubernetes. The old logs and any new logs are hosted on the old production droplet.

4 Lessons learned

4.1 Biggest Issues

In the initial stages of the development, whole team was working on refactoring the old code into new one. At the early stages of the development we have decided to split the code bases into API section and web-app section. These 2 folders do share the code and need to be updated separately due to duplicity code. This proved to be an issue at later stages since we need to update both code bases with the same code twice.

Another issue which was found at the end stage of the development phase was slow data loading for the UI. Reason for this is not optimized query which was comparing each single message with the user ID to make connection. This should potentially be solved by using JOIN in the query and therefore makes the process much faster.

4.2 Reflection

The group could have spend more time on dividing big tasks into smaller tasks, as to minimize merge conflicts and large pull requests. With each group member having varying schedules, knowledge sharing fast was at times an issue, but this was resolved by planning joint meetings between either the parties that held the knowledge and the ones who needed it, or with the entire group.

In the initial stages of the project, the group had some difficulties understanding each-other within the team. After multiple conflict-resolution meetings improvements were made. After a while we managed to get a good communication flow and organized planning. Team members took tasks which they felt comfortable with but also wanted to gain new knowledge and improve. Code reviews were taken seriously which helped us to improve and reflect on the code before it was pushed into the main. Breaking down tasks and setting deadlines for them helped us to keep the whole project on track.

Once the rules were set and understood by all members of the group, we were able to work more productively and efficiently.

4.3 Technology choices

4.3.1 Programming Language

- **Choice:** GO
- **Considered:** Java, C#
- Compiles to a single binary which makes it easy to deploy
- Low memory consumption, runs well on slower VMs
- language forces proper error handling and safe code
- has detailed documentation

4.3.2 Software Artifacts

- **Choice:** Docker
- **Considered:** VMs, Linux Packages, LXC, Go Packages
- Lower overhead compared to VMs.
- Supported on most Linux distributions regardless of package managers.
- Containers isolate the environment from the host system.
- Support for using different language compared to language specific artifacts.
- Support micro-services in our case allow us to run API and app with Docker Compose.
- Community support.

4.3.3 CI/CD Pipelines Tool

- **Choice:** GitHub Actions
- **Considered:** Jenkins, GitLab CI/CD, Bamboo
- Already integrated into code repository of our choice (Github).
- Minimal setup, compared to tools such as Jenkins.
- Runs on cloud without need of provisioning.
- Modern and easy to use UI.
- team members previous experience

4.3.4 Artifact Registry

- **Choice:** GitHub Container Registry
- **Considered:** DockerHub, GitHub Packages
- We switched from DockerHub, because we were only able to use an individual DockerHub account unless we were willing to pay for an organization.
- We chose GitHub Container Registry since it allowed us to publish container images directly in the GitHub organization. It also did not require us to use PATs since we could use the GITHUB_TOKEN from the action itself.

4.3.5 Monitoring

- **Choice:** Loki, Promtail and Grafana
- **Considered:** ELK stack
- Loki has lower memory usage
- lightweight and easy to deploy
- easy to manage

4.3.6 Infrastructure Automation Platforms

- **Choice:** Terraform
- **Previously Used:** Vagrant
- There is currently larger community behind Terraform than Vagrant.
- Less unexpected behavior we experienced compared to using Vagrant.

5 Conclusion

MiniTwit web application has performed with satisfactory results. All parts of the systems have been stable and functional during the whole development stage. The team has created an effective way of deployment and coordination which played a significant role towards the app performance. The application has encountered only one major downtime issue. Besides this we had only minor problems that were promptly fixed.

6 Appendix

1. Dependency scan made with Snyk. Tested both API app and Web app.

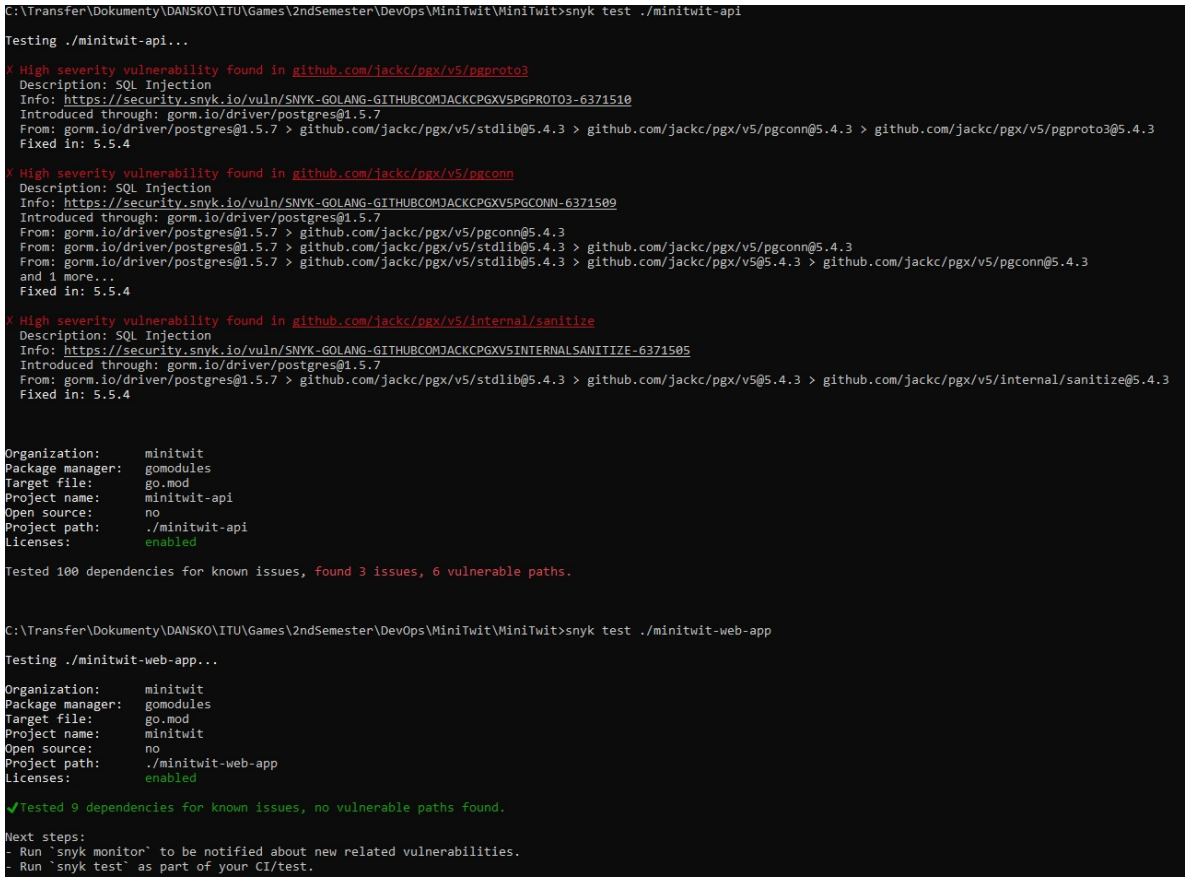

Figure 22: Dependency scan