Eagles: MiniTwit

IT University of Copenhagen

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

---

Course Information:

Name: DevOps, Software Evolution and Software Maintenance, MSc (Spring 2024)
Code: KSDSESM1KU
Description: https://learnit.itu.dk/local/coursebase/view.php?ciid=1391
Teaching Staff:
Helge Pfeiffer ropf@itu.dk
Mircea Lungu mlun@itu.dk

Table of 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 The initial web application, which was outdated with the latest technologies and was not following any standard practices, was rewritten from the ground up and gradually equipped with automations and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 is able to process a high load of requests coming to the app.

The documentation of the architecture differentiates between the old and the new architecture. The old architecture was

Deployment views

Old architecture

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

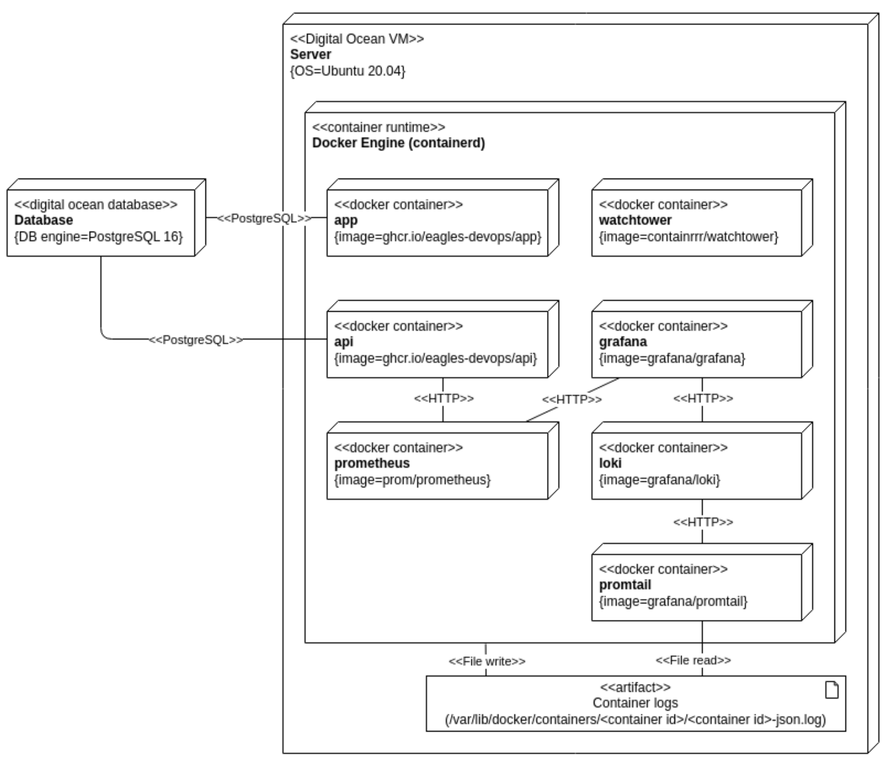
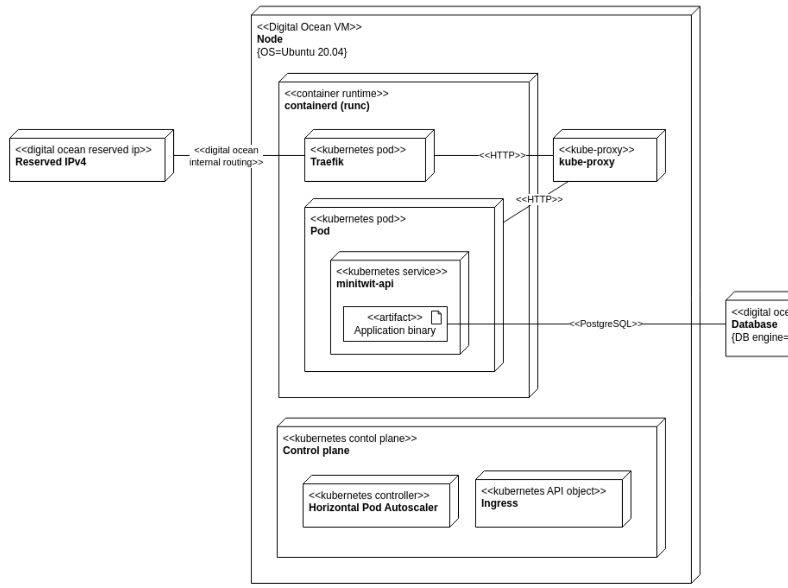


Figure 1: Deployment View: Old architecture

The 2 main parts of the system are the the database and the server. The database is a managed PostgreSQL database from digital ocean. This was chosen for its

The components shown in the digram were all defined in a docker-compose file which was was to s

New architecture



Below diagram shows the new architecture

Old Setup

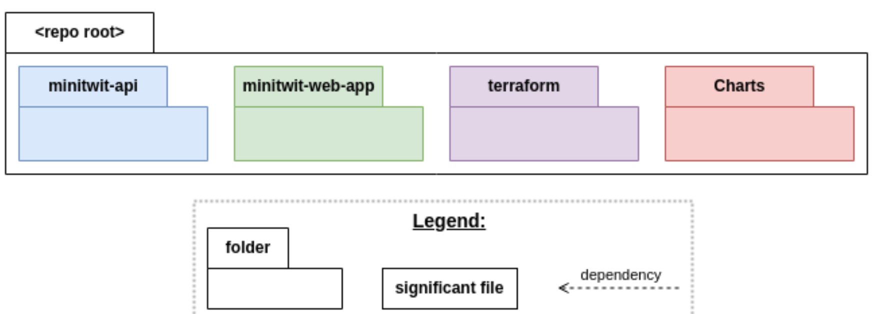


Figure 2: Module View: Repo overview

New setup

The new architecture was

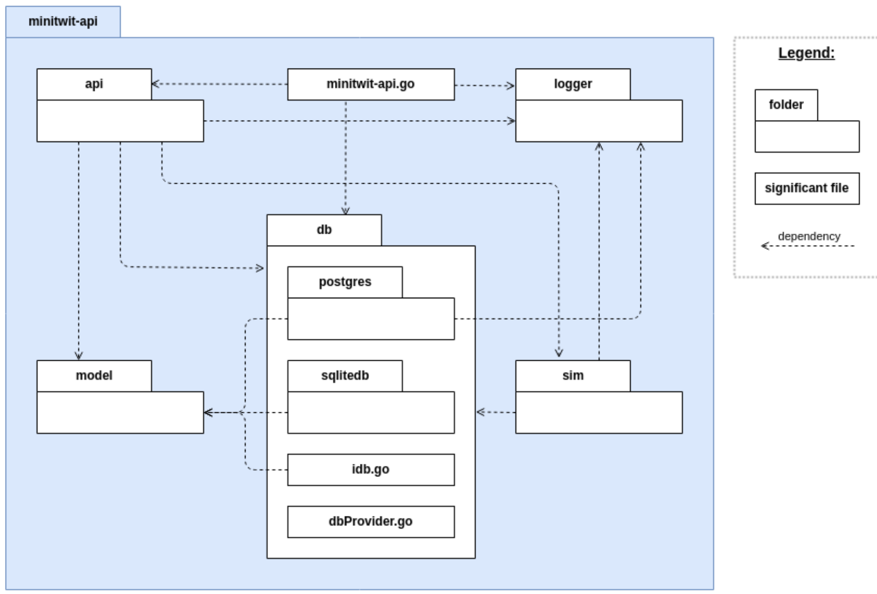


Figure 3: Module View: API

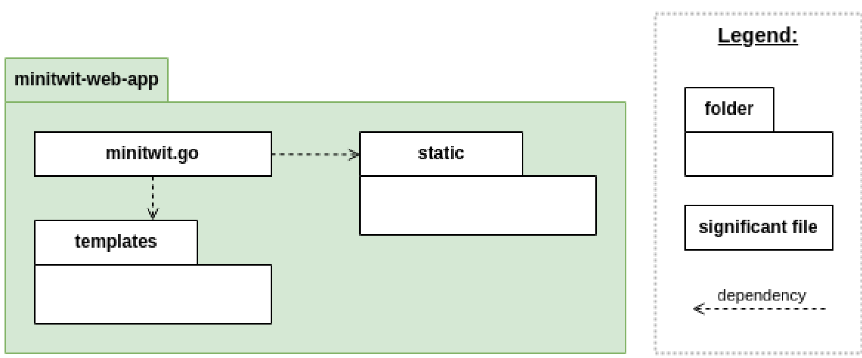


Figure 4: Module View: Web App

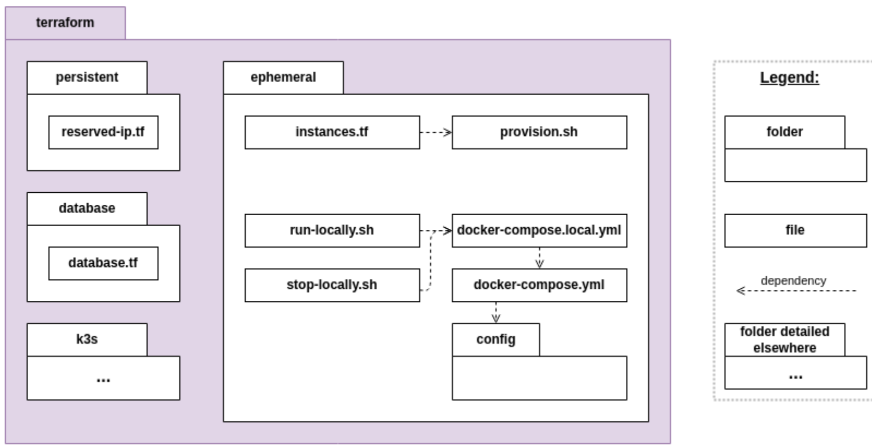


Figure 5: Module View: Terraform old

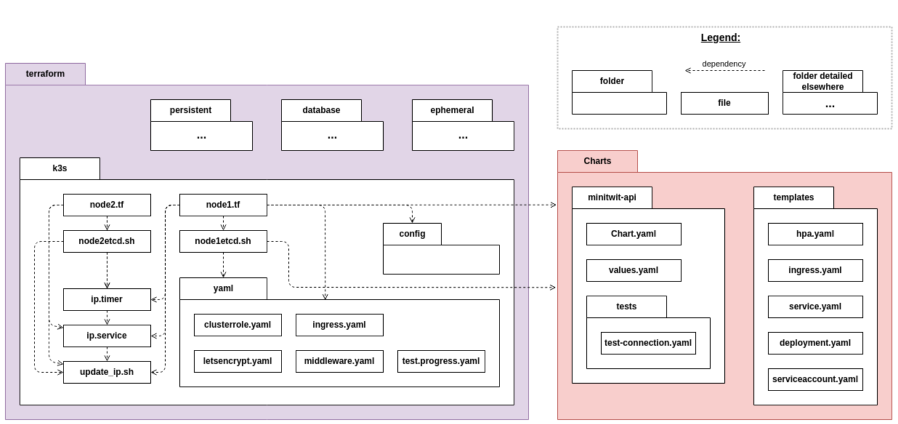


Figure 6: Module View: Terraform new K3S

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, multi-tenant log aggregation system inspired by Prometheus	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://pkg.go.dev/go.uber.org/zap@v1.27.0
SonarQube	open-source platform developed by SonarSource for continuous inspection of code quality	https://www.sonarsource.com/products/sonarqube/
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

State of the system

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

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. Reliability part of the code is prove to have a stable code base where most of the issues are related to other datetime variable interpretation as SonarQube is advicing 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 sumarize our code base would appriciate some minor adjustments but none of these crate a potential harm to our code stablity and readability.

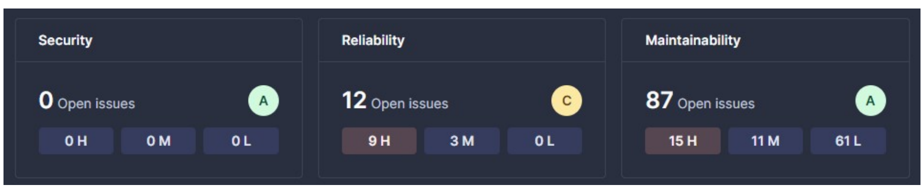


Figure 7: SonarQube general stats

2. Dependency scan
Projects utilizes 100 dependencies based from the dependency report made by Snyk where there are 3 dependensies currently vulnerable towards SQl injection. GitHub dependency report shows only 63 dependencies reporting similar issue regarding SQL injection vulnerability in some of the dependencies. GitHubs dependa bot created PR with needed update of the vulnerable dependencies which should solve the issue 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. 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 store either in GitHub secrets or we have an .enf file which each of us have on their local machine. Sharing such sensitive information between developer is done via USB drive or sharing them through BitWarden. Moreoever, when potential problem occurs GitHubs Advanced Security bot will create an alert and block open PR.

4. Test coverage
Application has a different sets of test - end-to-end, simulator tests, API tests

as well as linters. Even with these test in place we do not have a 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 for a long run. In the future projects would require some time into making more tets cases as well as different focus test sets.

5. Metrics, Logs and Dashboards
Application has a monitoring running on Grafana utilizing Loki and Mimir 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, amout of requests, number of messages and overall appliation status.

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

AI chat bots AI tools such as ChatGPT, Gemini, Opus-3 were used during the development stage. These tools were usefull 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 fo time.

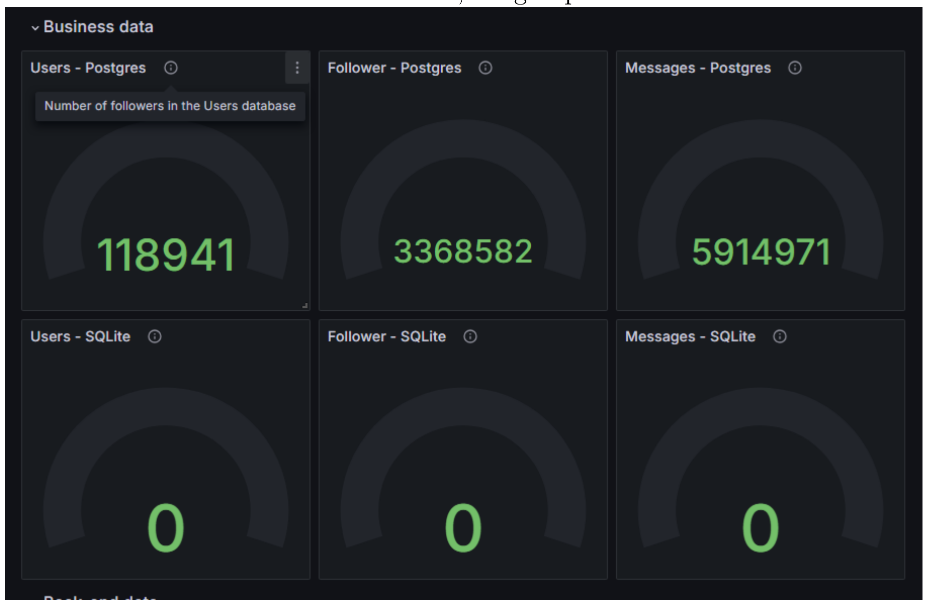
Logging

How do you monitor your systems and what precicely do you monitor?

For monitoring we use Prometheus with Grafana. We do so by incrementing gauges or vectors whenever an event has succesfully occured. In the system we monitor a multitude of things, for business data we log: - We monitor the amount of users getting created. - The amount of new followers on the platform - Amount of new messages posted - The total amount of reads and writes made to the database between releases. Besides incrementing counters we also monitor back-end data: - The amount of failed database read-writes - Whether there is a connection to the database - Succesful HTTP requests

Monitoring these gives us an insight to the extend of traffic passing through our

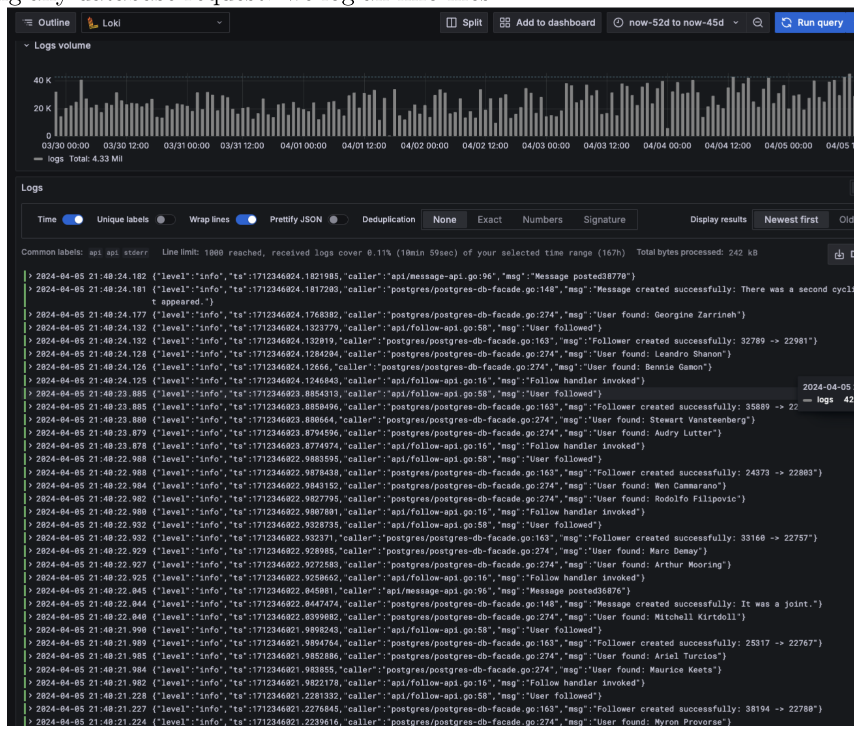
API. For ease of access to the monitored data and for visualization, the group uses

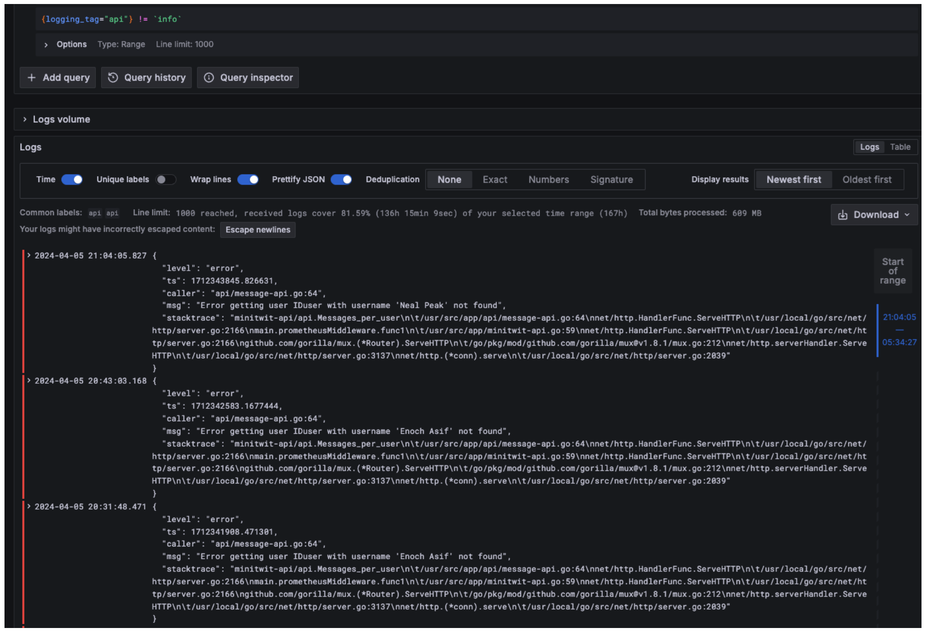


Grafanas dashboards, see
//

What do you log in your systems and how do you aggregate logs?

We log every error that happens during any database request. we log an info message each time some request happens. This enables us to trace all the steps made through the API to (partially) rebuild the database in case of loss of data. We log every error that happens during any



database request.


These are written through *zap*. The setup is such that the individual error logs are collected by logtail. Logtail then sends it to a Loki database that handles aggregation of the logs. The logs are visible through the Grafana Dashboard

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.

Lessons learned

Biggest Issues

In the initial stages of the development, whole team was working on refactoring the old code into new one. This created multiple instances of merge conflicts, different coding approaches.

Reflection

After 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. # Conlusion MiniTwit web applciation has performed with satisfactory results. All parth of the systems have been stable and functional during the whole development stage. Team has created an effective way of deployment and coordination which played a significant role towards the app performance. Application has encountered only one major downtime issue besides this no other major problems, error have been envountered which could affect the state.

Appendix

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

```
C:\Transfer\Dokumenty\DANSKO\ITU\Games\2ndSemester\DevOps\MiniTwit\MiniTwit>snyk test ./minitwit-api

Testing ./minitwit-api...

X High severity vulnerability found in github.com/jackc/pgx/v5/pgproto3
  Description: SQL Injection
  Info: https://security.snyk.io/vuln/SNYK-GOLANG-GITHUBCOMJACKCPGXV5PGPROTO3-6371510
  Introduced through: gorm.io/driver/postgres@1.5.7
  From: gorm.io/driver/postgres@1.5.7 > github.com/jackc/pgx/v5/stdlib@5.4.3 > github.com/jackc/pgx/v5/pgconn@5.4.3 > github.com/jackc/pgx/v5/pgproto3@5.4.3
  Fixed in: 5.5.4

X High severity vulnerability found in github.com/jackc/pgx/v5/pgconn
  Description: SQL Injection
  Info: https://security.snyk.io/vuln/SNYK-GOLANG-GITHUBCOMJACKCPGXV5PGCONN-6371509
  Introduced through: gorm.io/driver/postgres@1.5.7
  From: gorm.io/driver/postgres@1.5.7 > github.com/jackc/pgx/v5/pgconn@5.4.3
  From: gorm.io/driver/postgres@1.5.7 > github.com/jackc/pgx/v5/stdlib@5.4.3 > github.com/jackc/pgx/v5/pgconn@5.4.3
  From: gorm.io/driver/postgres@1.5.7 > github.com/jackc/pgx/v5/stdlib@5.4.3 > github.com/jackc/pgx/v5@5.4.3 > github.com/jackc/pgx/v5/pgconn@5.4.3
  and 1 more...
  Fixed in: 5.5.4

X High severity vulnerability found in github.com/jackc/pgx/v5/internal/sanitize
  Description: SQL Injection
  Info: https://security.snyk.io/vuln/SNYK-GOLANG-GITHUBCOMJACKCPGXV5INTERNALSANITIZE-6371505
  Introduced through: gorm.io/driver/postgres@1.5.7
  From: gorm.io/driver/postgres@1.5.7 > github.com/jackc/pgx/v5/stdlib@5.4.3 > github.com/jackc/pgx/v5@5.4.3 > github.com/jackc/pgx/v5/internal/sanitize@5.4.3
  Fixed in: 5.5.4



Organization:      minitwit
Package manager:   gomodules
Target file:       go.mod
Project name:      minitwit-api
Open source:       no
Project path:      ./minitwit-api
Licenses:          enabled

Tested 100 dependencies for known issues, found 3 issues, 6 vulnerable paths.


C:\Transfer\Dokumenty\DANSKO\ITU\Games\2ndSemester\DevOps\MiniTwit\MiniTwit>snyk test ./minitwit-web-app

Testing ./minitwit-web-app...

Organization:      minitwit
Package manager:   gomodules
Target file:       go.mod
Project name:      minitwit
Open source:       no
Project path:      ./minitwit-web-app
Licenses:          enabled

✓Tested 9 dependencies for known issues, no vulnerable paths found.

Next steps:
- Run `snyk monitor` to be notified about new related vulnerabilities.
- Run `snyk test` as part of your CI/test.
```

   ## K3S The API is now running on a leightweight 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.

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

Letsencrypt is used for SSL certificates and is automatically created/renewed for deployments.

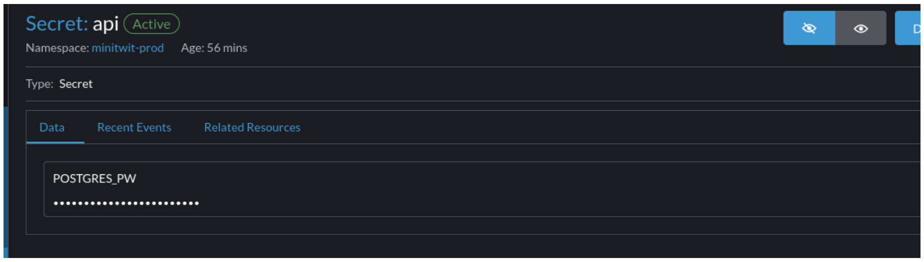


Figure 8: secret

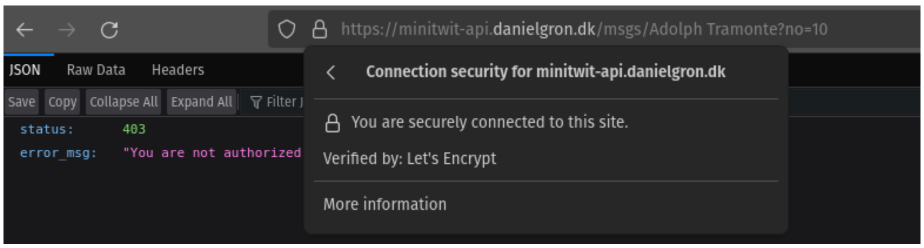


Figure 9: ssl

CI-CD

PR tests

A complete description of stages and tools included in the CI/CD chains, including deployment and release of your systems.

When code changes related to an issue are ready to be merged in to main, a pull request is opened, and unit- and integration tests are then run as a GitHub Action. These - along with a review, and passing the quality gate of the static analysis tool, are a prerequisit for the pull request to be completed.

Deployment

When the pull request is completed, the changed are merged to main triggering the ci-cd workflow. - Build docker image - Push docker image to registry - Deploy to K3S with Helm

Other workflow

We have a few other workflows as part of our setup.

As required a workflow generating a PDF top the report directory has been created.

A manually triggered workflow for running linters exists in linters.yaml.

Lastly a workflow for automating a weekly release is running at the end of each week. At this point it would make more sense to consider each automated deploy a release instead, but the automated release can be seen as a weekly snapshot. ## 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 deployting. 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.

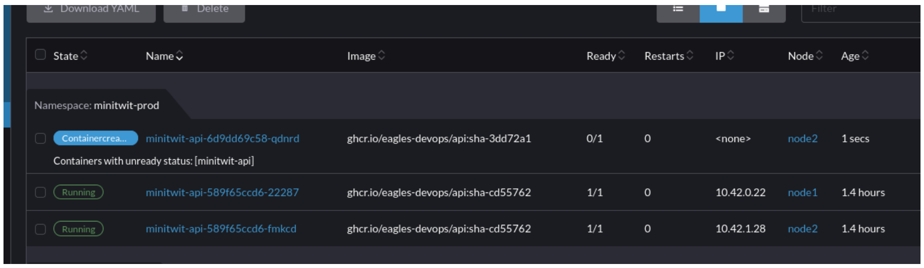


Figure 10: bluegreen

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

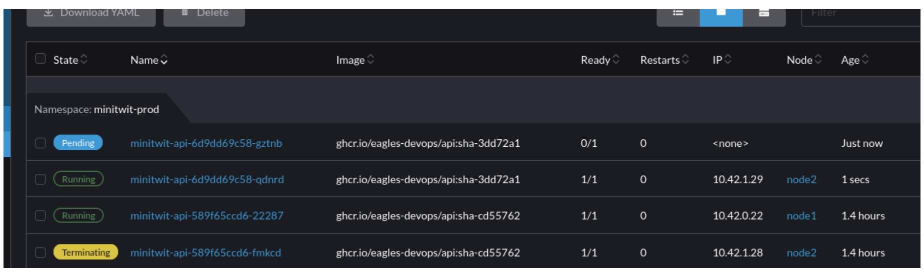


Figure 11: bluegreen

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.

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

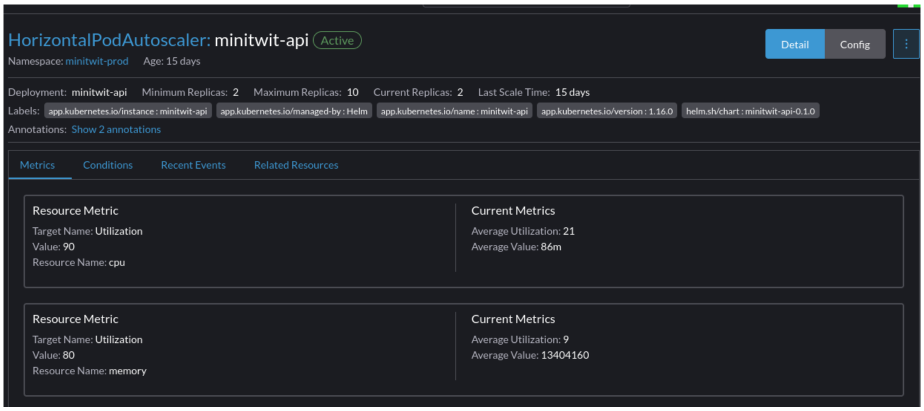


Figure 12: bluegreen

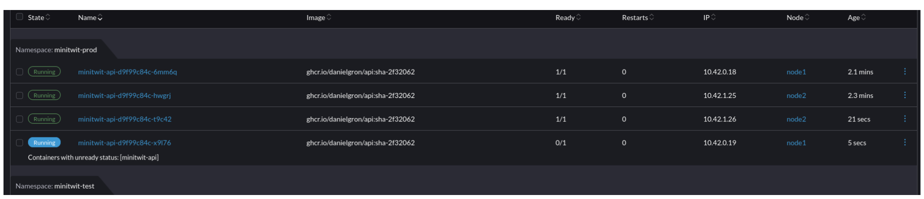


Figure 13: bluegreen

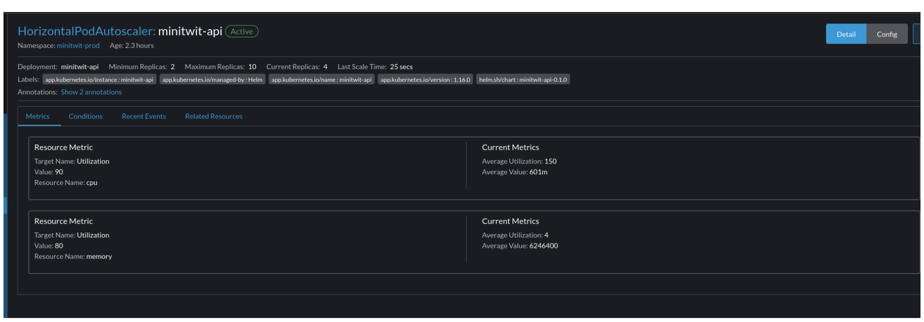


Figure 14: bluegreen