DevOps, Software Evolution & Software Maintenance

Course code: KSDSESM1KU
Spring 2022

By

Student	Email
Endrit Beqiri	enbe@itu.dk
Musab Kilic	muki@itu.dk
Dawid Woźniak	dawo@itu.dk
Holger Gott Christensen	hoch@itu.dk

---

1. System's perspective

1.1 Design

1.1.1 Application Stack

The initial MiniTwit application was written in Python 2. Due to us having to take over the application, we needed to decide on a programming language and web framework to use for the application rewrite.

To decide on a language we wanted to evaluate some current good choices and decided to test 4 different programming languages and their frameworks for the purpose of measuring their usefulness and suitability for our project.

Below we have summarised our main findings, with more explained in our decision log.

	Advantages	Downsides
Ruby & Sinatra	Minimalistic Ruby has a lot of documentation and a big community behind it	Not scalable Not flexible Needs many other libraries
Crystal & Kemal	Strongly typed	(Almost) no documentation Small community Low support

	Advantages	Downsides
Go & Gorilla	Good documentation and tutorials Large community Good support Scalable Strict typing	
Elixir & Phoenix	Performant Good documentations and tutorials Medium-sized community Functional programming language Easier to test	There are some issues with language support in IDE

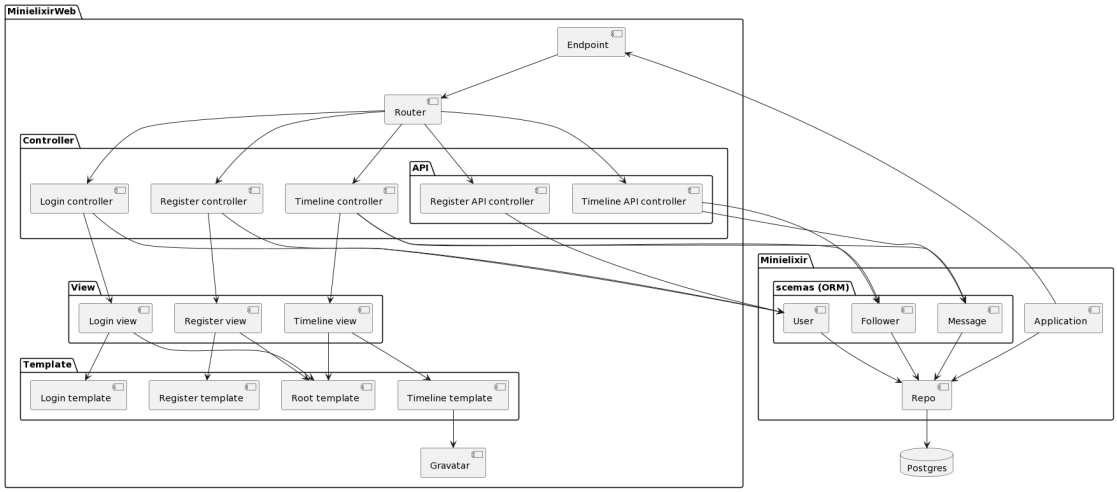
Our main choice was between Elixir and Go. When testing Go, we experienced problems with bugs that we had difficulty debugging. We chose Elixir based on testability and the support for the Phoenix web framework.

1.1.2 Choosing a database

The choice of programming language and framework affected the choice of database because of the ORM (Object Relational Mapper) framework Ecto. Ecto's recommendation is PostgreSQL and since we have developer experience with this database, it is open source, performant and has a large development community behind it, we decided it was the best choice.

1.1.3 System Design

Below we have created a diagram showing the module level design of the new MiniTwit application. This is modelled based on Elixir modules and are based on the MVC architecture.



(source)

The application can be partitioned into 2 main components: *MinitwitElixir* and *MinitwitElixirWeb*.

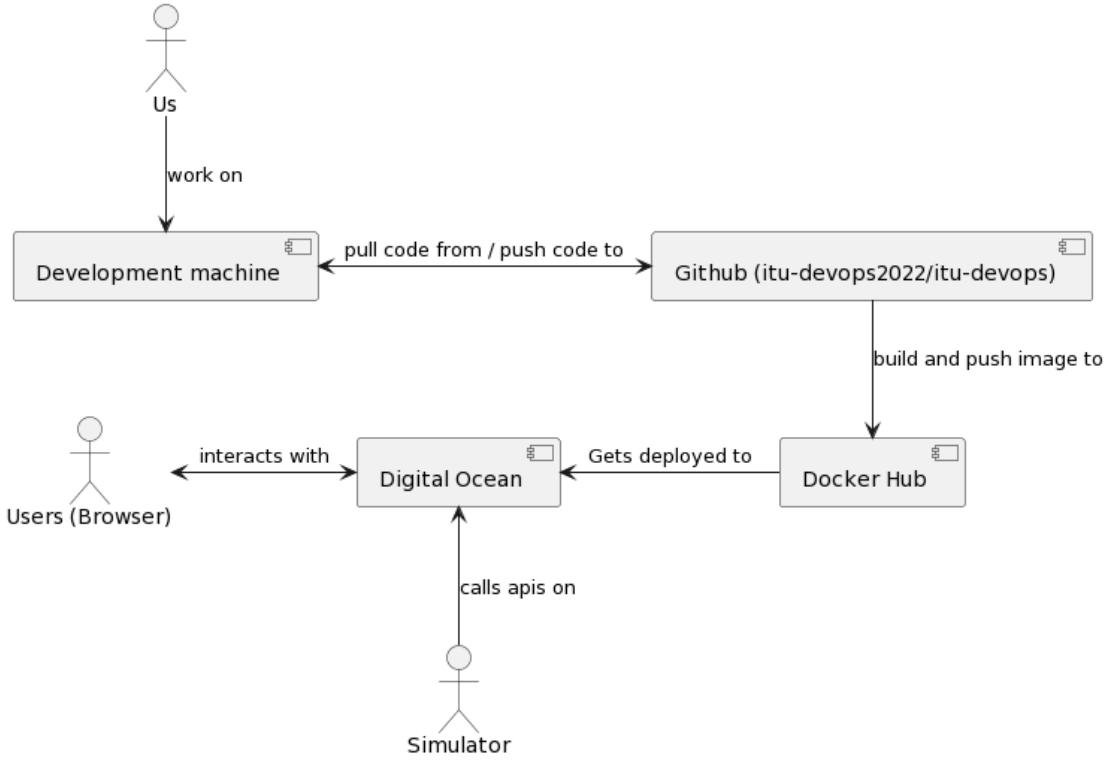
MinitwitElixir contains the main application file that gets called on program startup. This sets up the database connection, initiates the metrics and starts the web framework. It also contains the model of the application storing schemas for the users, followers and messages. These are used by the ORM to access items in the database through the Repo interface.

MinitwitElixirWeb has mainly two responsibilities. The first is the functionality of networking. This consist of the Endpoint and Router. The Endpoint is the entry point for user requests and configures the pipeline where the requests go through before being processed in the application logic. The Router receives the requests via the Endpoint and maps the path in the request header to the appropriate handler.
The second functionality is the application logic stored in the different controllers. We have two different groupings of controllers. Those who handle api requests and those who handle browser requests. The API controller implements the logic and returns the specified json. The other controllers depends on a view and a templating language created by Phoenix to serve the html response sent to the browser.

1.2 Architecture

1.2.1 Architecture overview

Below we have created a high level overview of the architecture of our Minitwit system.



(source)

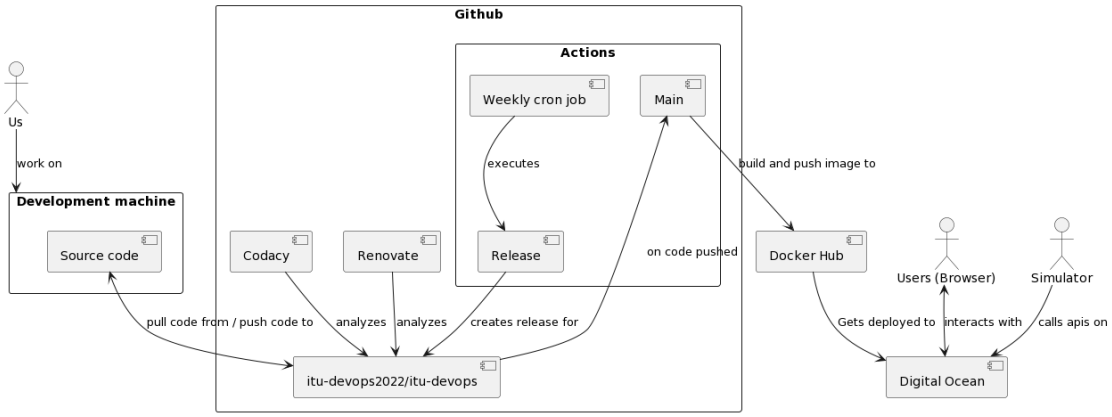
The system comprises of several different components and these will be specified more thoroughly in the upcoming sections. We as developers work on our machines and push code from our local to the GitHub hosted git repository. Whenever code is pushed to the master branch we build a docker image and deploy this to our Digital Ocean droplets. The user interacts with the application inside the docker container within the droplet. The simulator interacts in the same way with the application.

1.2.2 VCS

Our initial action was to integrate the code base to a VCS. We chose Git and GitHub since it is by far the most prominent VCS. Furthermore, we decided to migrate the repository to an organisation because it gives us more flexibility when setting up the CI/CD for the repository.

Below we have expanded the architecture view to show a more detailed view on the components on GitHub. We have the code stored in a repository and have some GitHub bots analyse the code and check code quality

and dependencies. When the code is pushed to the master branch we run a GitHub Actions pipeline. The Actions are described in more detail in the CI/CD section below.



(source)

1.2.3 CI/CD Pipelines

The technologies to choose between were either self-hosted solutions such as Jenkins or TeamCity, or CI/CD as a service solutions such as Travis CI or Github Actions. In order to not have additional artefacts to maintain, we chose CI/CD as a service. However, we had difficulty choosing between Travis CI and Github Actions. Travis CI is an older solution and therefore more mature compared to Github Actions. However, the latter is free, provides pre-built configuration pipelines, and we are already using GitHub.

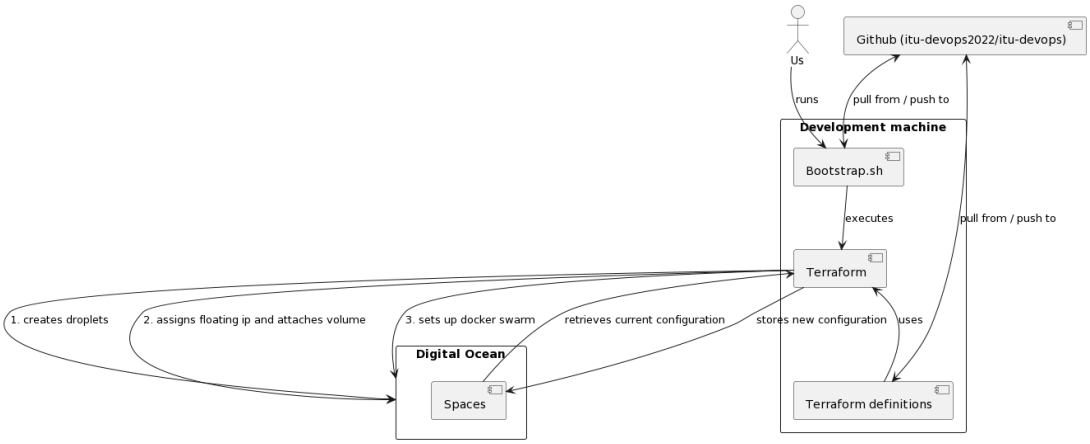
We started with a travis pipeline because of maturity, but the tokens ran out quite quickly. Travis is quite an expensive solution and we decided to migrate to Github Actions.

We have two different pipelines. The first pipeline starts on every push to the master branch. This pipeline includes in order unit testing, building and pushing an image to DockerHub registry, pushing to the release branch, and finally deploying by SSH to the virtual machine and running the most recent docker container. The second pipeline is scheduled to run every Sunday to make a weekly release.

1.2.4 Infrastructure as code

We use Terraform for Infrastructure as code. We want to be able to store a configuration as code so we can put it in version control, easily reproduce the production environment and not have to manually configure the servers. We don’t want to be stuck at using a specific cloud vender, and Terraform makes it easier to change provider. The Terraform definitions also serve as documentation as to the structure of the application making it easy for all developers in the team to configure the system.

Below we have a diagram showing the structure of Terraform what depends on each other.

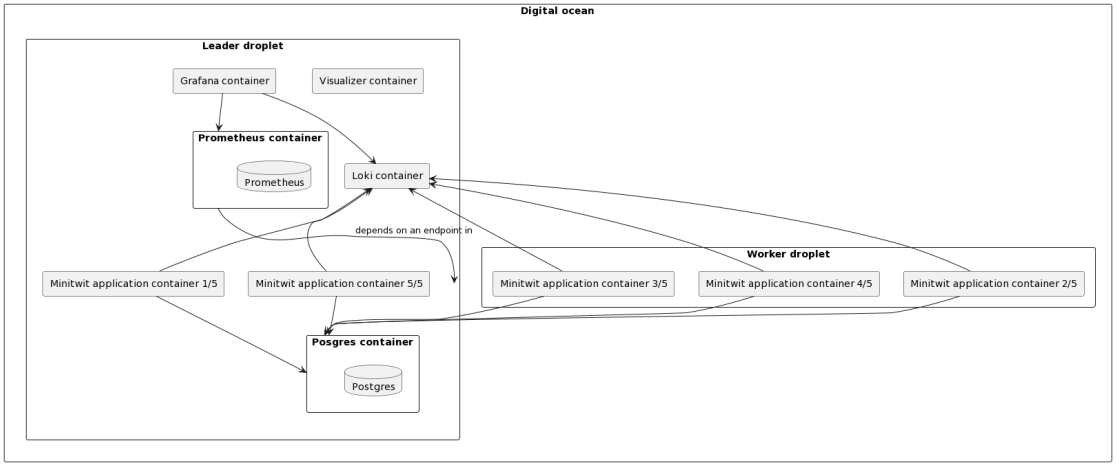


(source)

Earlier we used Vagrant for automating the process of setting up virtual machines. Terraform and Vagrant are different tools with different purposes, however, they share the common feature of setting up virtual machine. Terraform describes the infrastructure as code configuring service provider (and their corresponding virtual machines), whereas, Vagrant manages virtual machines(link).

1.2.5 Containerisation & Virtual Machine Provisioning

Since the application stack has several dependencies, we need a mechanism that can abstract these dependencies so that running & testing the application will not become environment dependent. For this reason, we use Docker to containerise our application allowing it to run and be tested cross-platform. A registry in DockerHub is used so that when deploying the application on a remote machine it will be able to get the container image and run it. We chose Docker instead of other container technologies because it is the most popular in the industry.



(source)

Above we have a diagram of the current structure of our digital ocean instance. This is also depicting the current location of application containers, although this can change because we have the application deployed in a docker swarm. We have two droplets, a leader and a worker. The leader contains the Grafana instance, the Loki container, the Prometheus database and the Postgres database. These are constrained to only be on the leader droplet in the swarm. Both the leader and the worker contains application instances, and requests made are distributed round robin style to all instances.

Using a docker swarm also gives us a load balancer and automatic startup on failure. We also get rolling updates.

1.3 Dependencies

1.3.1 Application level dependencies

The Elixir programming language depends on the Erlang VM. It is distributed on multiple platforms and depends on the specific system level dependencies. Furthermore we depend on Elixir packages. Some are first level dependencies such as Phoenix and Ecto, where others are second level dependents of other packages. In total we have 59 Elixir package dependencies.

1.3.2 System dependencies

Our application has a number of system level dependencies. We use docker to handle these and provide reproducible builds. We are dependent on the Linux (ubuntu) system used by the Elixir image hosted on DockerHub. We deploy our own image based on that, so DockerHub is another dependency. The docker image bundles all the system dependencies to run the application. We are dependent on the Postgres and 3 other Docker Images.

1.3.3 Monitoring and logging dependencies

For monitoring we depend on the time-series database Prometheus. To serve the Prometheus scraper we depend on the Elixir package PromEx. For logging we depend on Loki and an open source Elixir package called LokiLogger. To visualise the monitoring and logging we depend on Grafana.

1.3.4 Database dependencies

We depend on the PostgreSQL image hosted on DockerHub which in turn has a lot of dependencies. To connect to the database we depend on the ORM Elixir package Ecto.

1.3.5 Version control dependencies

For version control we depend on GitHub and git as its underlying VCS technology. We depend on GitHub Actions for the CI/CD pipeline which tests, build, releases, pushes Docker images to DockerHub and deploys our application.

1.3.6 DigitalOcean & Terraform

For infrastructure we depend on Digital Ocean and their droplets. We also depend on volumes and other technologies provided by Digital Ocean. For infrastructure configuration we depend on Terraform, which in our case depends on a Digital Ocean Space for storing the current state of the infrastructure.

1.4 Important interactions of subsystems

While some of the high level interactions are described in the sections above, we have chosen two to go into more depth with.

1.4.1 Database subsystem

The most central subsystem in our application is that of database interactions. The interactions between the MiniTwit application and the Postgres database works through the Ecto Elixir package. Ecto is an ORM that abstracts away the implementation of the database. We write queries by composing Ecto functions. This makes sure security attacks such as SQL injections are not possible. Ecto provides a Repo class where we

can retrieve users, messages and who follows whom from the database. This is also used when inserting new users and messages.

1.4.2 Metrics subsystem

As described in the metrics dependencies section we depend on the Telemetry Elixir implementation. The metrics are updated for every requests relevant for business such as users registered, message posts, authorisation among others. We use Prometheus as pull-based monitoring and configured it to pull metrics from the application server every 15 seconds. It pulls the metrics by creating an API request in the /metrics endpoint.
The default implementation of telemetry resets all metrics when the application closes. We decided to implement custom logic persisting the current metrics to the Postgres database. Prometheus stores the metrics at specific points of time and we are able to connect to this and visualise the metrics using Grafana.

1.5 Current state of the systems

1.5.1 Codacy

We have added Codacy as a static analysis tool to get suggestions for our code quality and security vulnerabilities. Most of the critical issues in our code are regarding codestyle, e.g. lines being too long. These are not concerning since the code fits in the editor on our computers and are not detrimental to the understanding of our code. The most issues was found in CSS files about styling. We do not consider these issues and have not had any focus on CSS during the project. Overall, Codacy have given us a B for software quality.

1.5.2 Renovate

We added renovate to analyse our dependencies and remind us if any new versions exist. It should also alert us if there is any security problems in the dependencies.

1.5.3 Snyk and CodeScene

There was only found low level issues, all connected to dependencies of Postgres.

1.6 License compatibility with dependencies of the project

We chose the Apache 2.0 licence for our MiniTwit application. We chose this because it is a permissive license, while also protecting us from people putting in copyrighted code into our repository. It has no copy left requirement, so it allows others to use our code for proprietary use.

We ran the ScanCode toolkit to analyse the different licenses used in our project (result). The tool reported that we mainly depend on code that is under the MIT, Apache 2.0 and ISC license. These are all compatible with the license we have chosen.

2. Process' perspective

2.1 Team organization

We organize our team into one group. All members work on all aspects of the project - frontend, backend and DevOps. We all share knowledge and have a full system picture in our minds.

Our primary channel of asynchronous communication is a Discord server. Thanks to **UptimeRobot** we are notified the latest 15 minutes after our website is down. If we work collectively on some important aspect of the website, we usually meet in person.

2.2 Organization of our repositories

Our project contains only one repository put in the GitHub organization called **itu-devops2022**:

- itu-devops

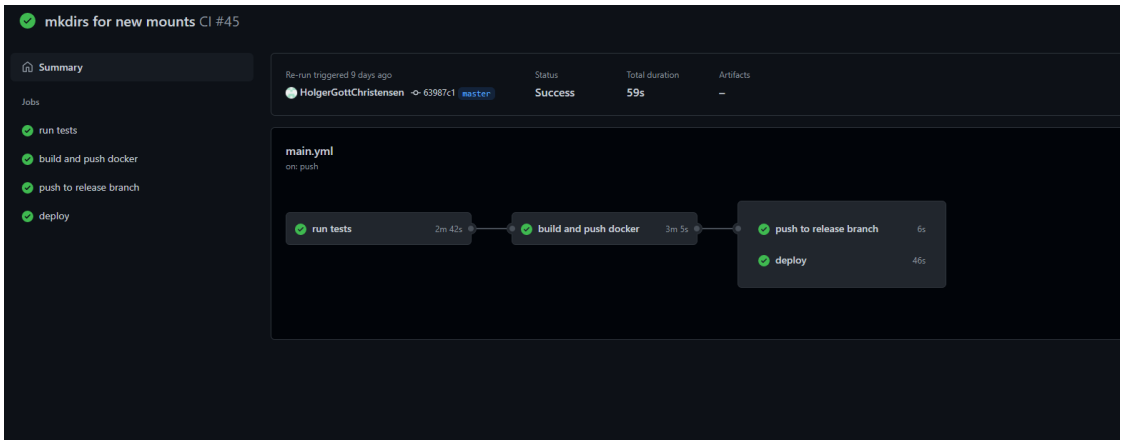
We made the conceptual decision to store everything in one repository to avoid confusion, forking and breaking our project into too many pieces.

Our main repo contains all website parts. This approach has many advantages:

- Avoiding of creation complicated links between different repositories to create easily a local environment and test interdisciplinary changes,
- New changes reviewers can see all changes in one place,
- The project builds are easier to maintain (one stage build - one build result).

2.3 CI/CD chains

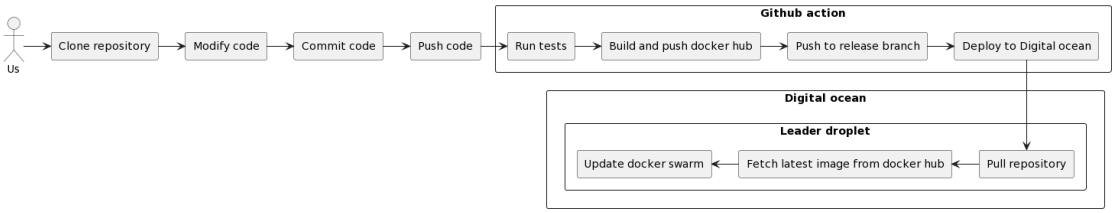
Our CI/CD process is integrated directly with our repository and DigitalOcean through **GitHub Action**.



Essentially, we have four steps. We run our automated tests first, then we build and push our docker to DigitalOcean. Finally, we add our changes to the release branch (automatically created) and trigger re-deploy on DigitalOcean.

We also use **GitHub Action** to automatically create a release. They are created every 7 days without human interaction.

Below is a short overview of the process from making a change in code to it being in production.



(source)

2.4 Branching strategy

In one of the first project weeks, we've created a few rules to our Distributed Workflow. In this section, we review our agreement with reality.

Our distributed development workflow is the centralized workflow. The main branch of this repo is **master**. We agreed noone can push directly to this branch. In reality, if we worked together and we were sure our changes worked correctly, we pushed directly to master with the oral approval of other project members. We opened a pull request to change code if we were working asynchronously. Some pull requests were also opened by **Renovate bot** it helped us to keep our dependencies up to date.

Any contributor to the project could create a branch and propose the change. The name of the branch and commits should be meaningful. The most convenient way to name branches turned out to be names of the feature we worked on. Some branch name examples are *loki*, *logging* or *minitwit-tests*.

We agreed to use the **squash and merge** strategy. It wasn't the best choice. Usually, our pull requests have more than one commit and wanted to keep this history when we merge to master. Therefore, we used mainly the strategy **creating a merge commit**.

2.5 Development process and tools supporting it

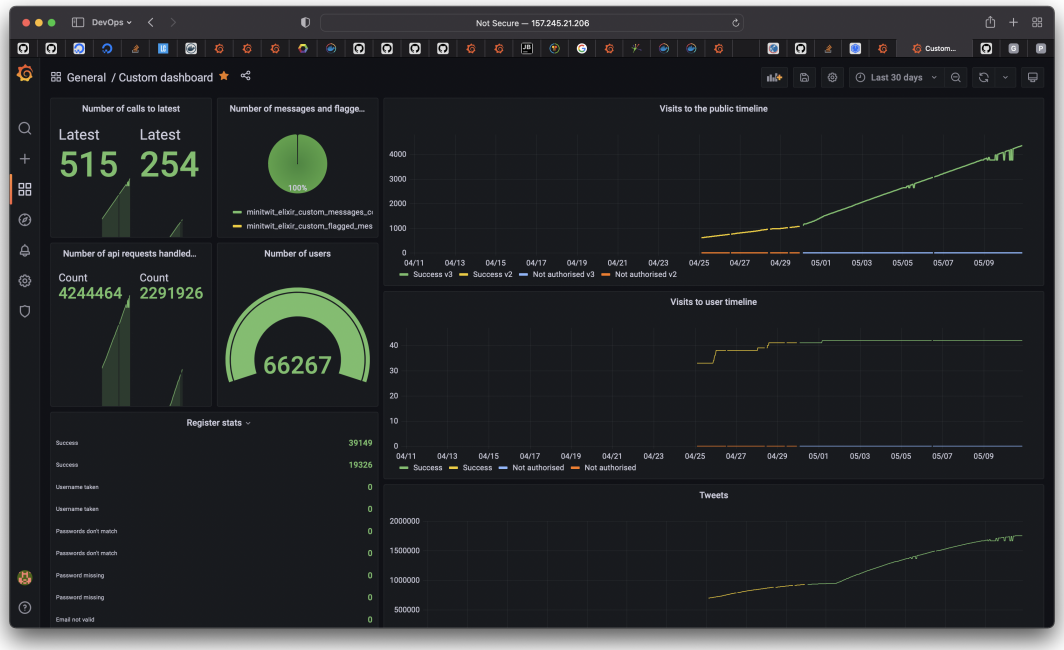
As our organization board, documentation place and the way to track progress on open tasks, we used features embedded in GitHub.

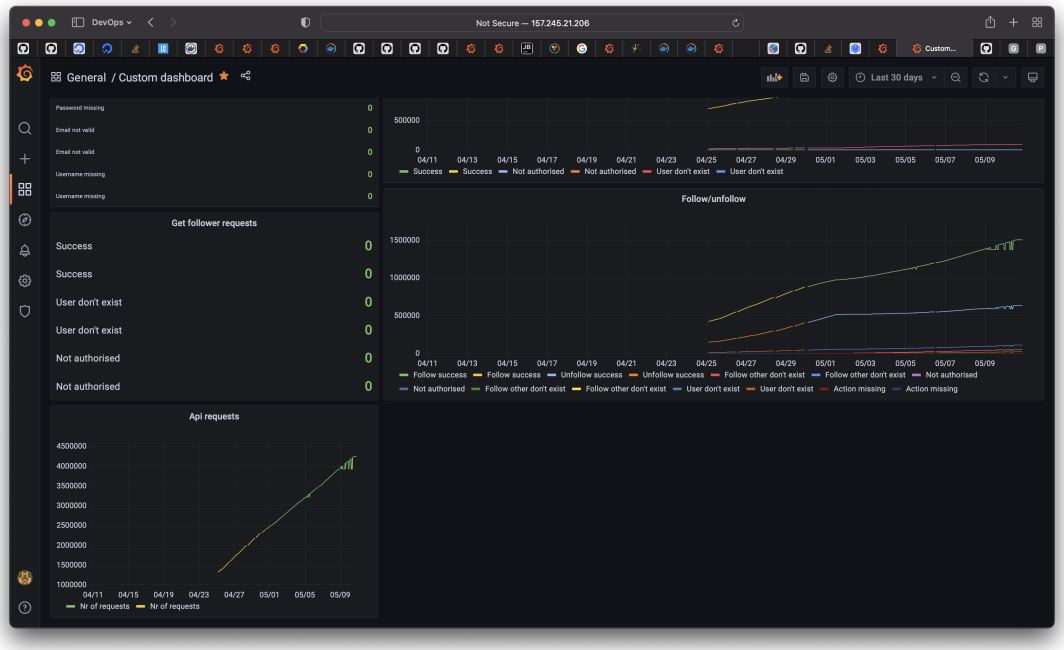
- We use **issues** to store tasks not specificed by weekly assignments.
- We use **Wiki** to keep our documentation, including decision and week logs.

As we mentioned earlier, we used also **GitHub Action**, **Discord**, **Renovate bot** and **UptimeRobot** to make our development easier.

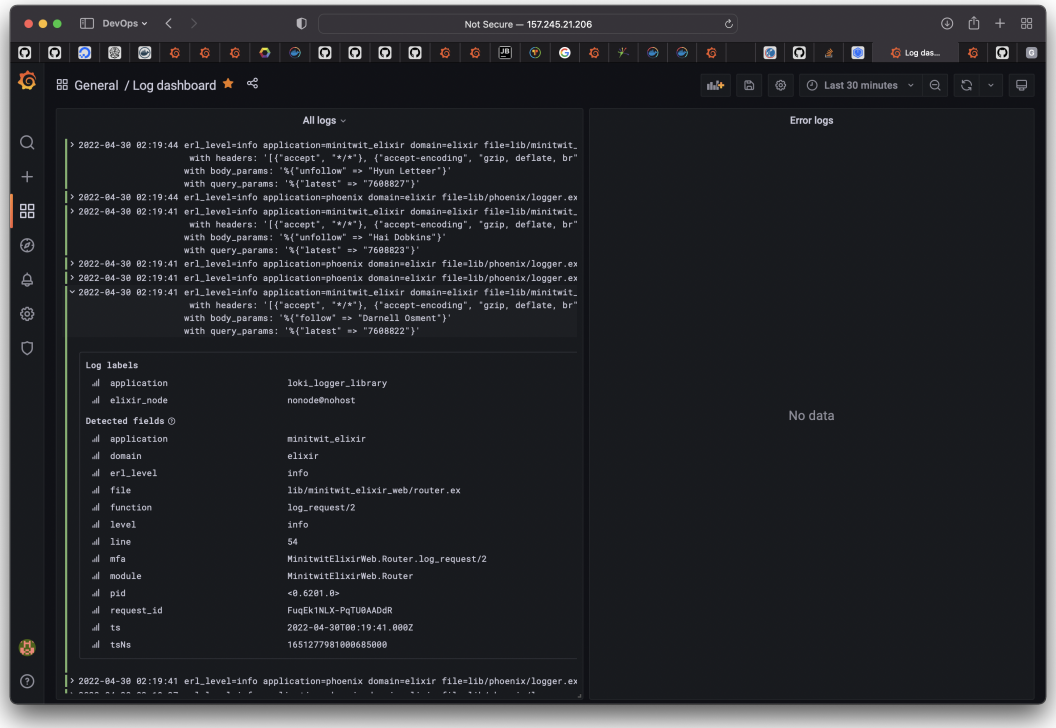
2.6 Monitoring and logs

In MiniTwit we are monitoring all user requests to the website and api. We keep count of e.g. how many users we have, how many failed user registrations and varius other metrics.





We are logging all requests to the service, but automatically exclude sensitive data like passwords and session tokens. We store all headers and the request body in the log.



2.7 Brief results of the security assessment

We performed our security assessment according to industry standards. The best brief our report summary is the risk assessment matrix.

Problem	Privacy Risk	Security Risk	Availability Risk	Recovery Risk	Estimated Total Risk
Lack of SSL Certificate	High	High	None	None	Medium
DDoS Mitigation Services	Low	Low	High	High	Medium
Password policy	Medium	Medium	None	None	Low
Backup policy	Low	Low	Low	High	Low

2.8 Strategy for scaling and load balancing

We have two different scaling strategies; horizontal and vertical scaling. Our architecture allows us to use both. We can change the size of the droplets in using our Terraform definitions, resulting in vertical scaling. We can also add more worker droplets and add them to our swarm and deploy application to them.

One limitation is that we always store the database on the leader node. This means that all database requests needs to go through this and means vertical scaling is most suitable for the leader droplet, while the worker droplets will be better scaled using horizontal scaling.

3. Lessons Learned Perspective

3.1 Evolution and refactoring

As we mentioned before, we decided to use non-popular technical stack to refactor our project. It has turned out to be problematic in sense that there are not that much support and feature framework choices. We also improved our communication over time using a dedicated server and useful bots.

3.2 Operation

3.2.1 Containerisation

Using containerisation during development and testing has showed to reduce a lot of setup time. It enables us to easily run the application regardless of the operating system. Using Docker Compose, we can locally run the entire system that is to be deployed to cloud services locally. In other projects we have experienced how setting up several technologies manually can be cumbersome. From this perspective Docker was a useful tool.

3.2.2 DigitalOcean, Vagrant & Terraform

We chose DigitalOcean over other services e.g. AWS as it presents the best free offer in relation to offered features.
Vagrant took a lot of time to setup and our setup with it was flaky at best. We could not ssh into our machine using vagrant consistently, even when we could manually.
Terraform is a great tool and we will gladly use it again. It made the whole experience of setting up machines and configuring them seamless and even when updating the infrastructure “it just worked™". We have not tried to change what cloud provider we use with Terraform, but it seems to be easy to change in the future.

3.2.3 VCS & CI/CD Pipelines

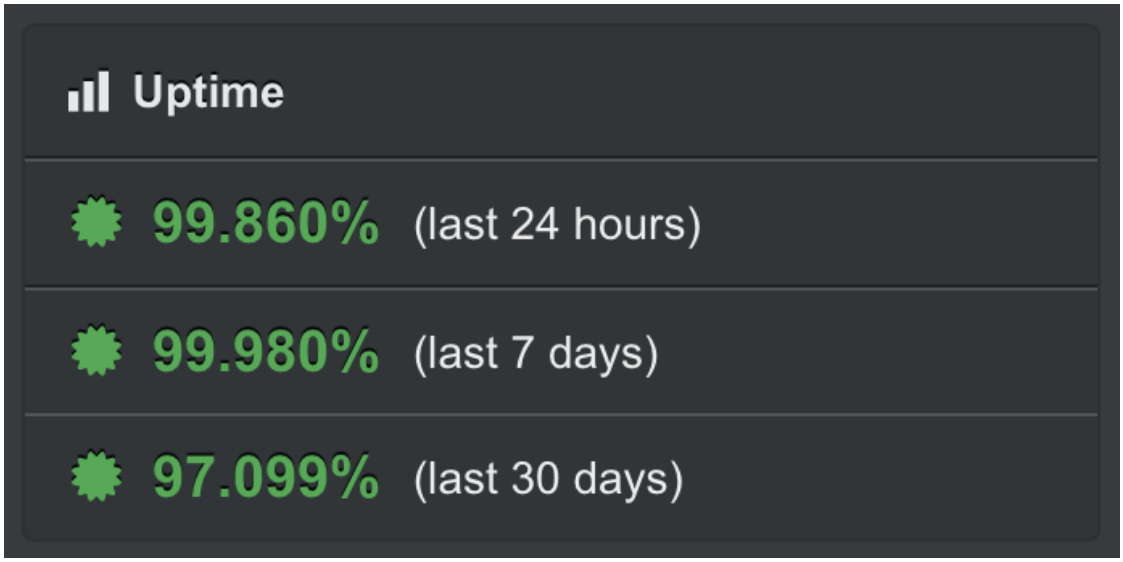
One of our biggest learnings was that when experimenting with infrastructure and configuration even when we had set up a CI pipeline, we had to cancel a lot of builds because the CI pipeline was slow and it got in the way. Every time we push to the repo to test a change in configuration the pipeline wanted to be build.

The pipeline catched some code level bugs and catched some configuration issues with docker not building. These were useful and meant we could be more sure when deploying things that they worked before pushing to production. The CI pipeline also automated the task of doing release which was another useful feature.

3.2.4 Uptime monitoring

At the start of the simulator running we did not detect that the website had crashed. This caused a lot of users not to be registered and added 99% of the errors reported in the course graphs along with us loosing the original IP address submitted for the simulator.

Because we did not at that time have any monitoring we didn’t detect that the service was down. We did not want this to happen again and added a simple uptime detection service to check if the website was reachable. This was one of the most useful tools we had. At the end of the simulator running 99.8% uptime in the latest 24 hours and 97.0% uptime the latest 30 days.



3.3 Maintenance

3.3.1 Monitoring

In school projects, we often find reasons for changes in the application by testing it individually and usability testing. Consequently, this does not provide a realistic environment in which the application will operate. Applying monitoring and simulation revealed how the former technique would not discover defects in the application.

3.3.2 Logging

The integration of logging into our system gave us insight about “the trap of mainstream”. Since the ELK stack was the most popular choice for logging, we blindly tried to integrate this into our system. However, the futile attempts to solve the performance issue of ElasticSearch indicated that another technology would be more appropriate in our case.
We learned that instead of comparing logging to debugging, logging complements the use of debugging.