DevOps 2025

IT University of Copenhagen

Team Sad People

People

- *Gábor Tódor* - gato@itu.dk
- *Zalán Kálny* - zaka@itu.dk
- *Nicklas Koch Rasmussen* - nicra@itu.dk
- *Nicolai Grymer* - ngry@itu.dk
- *Sebastian Andersen* - seaa@itu.dk

Subject

- **School:** *IT-University of Copenhagen*
- **Course manager:** *Helge Pfeiffer* - ropf@itu.dk
- **Course Code:** KSDSESM1KU

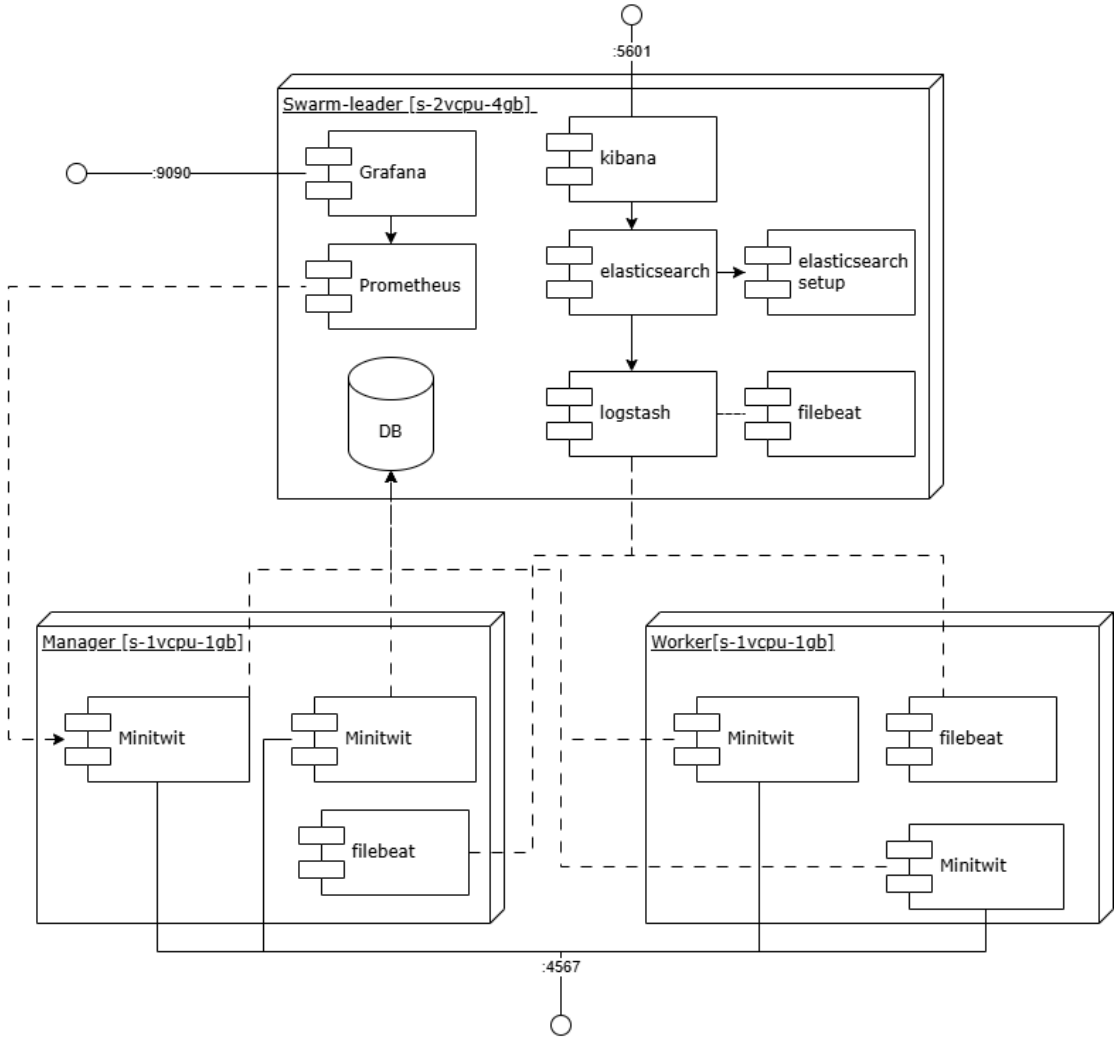
Introduction

System

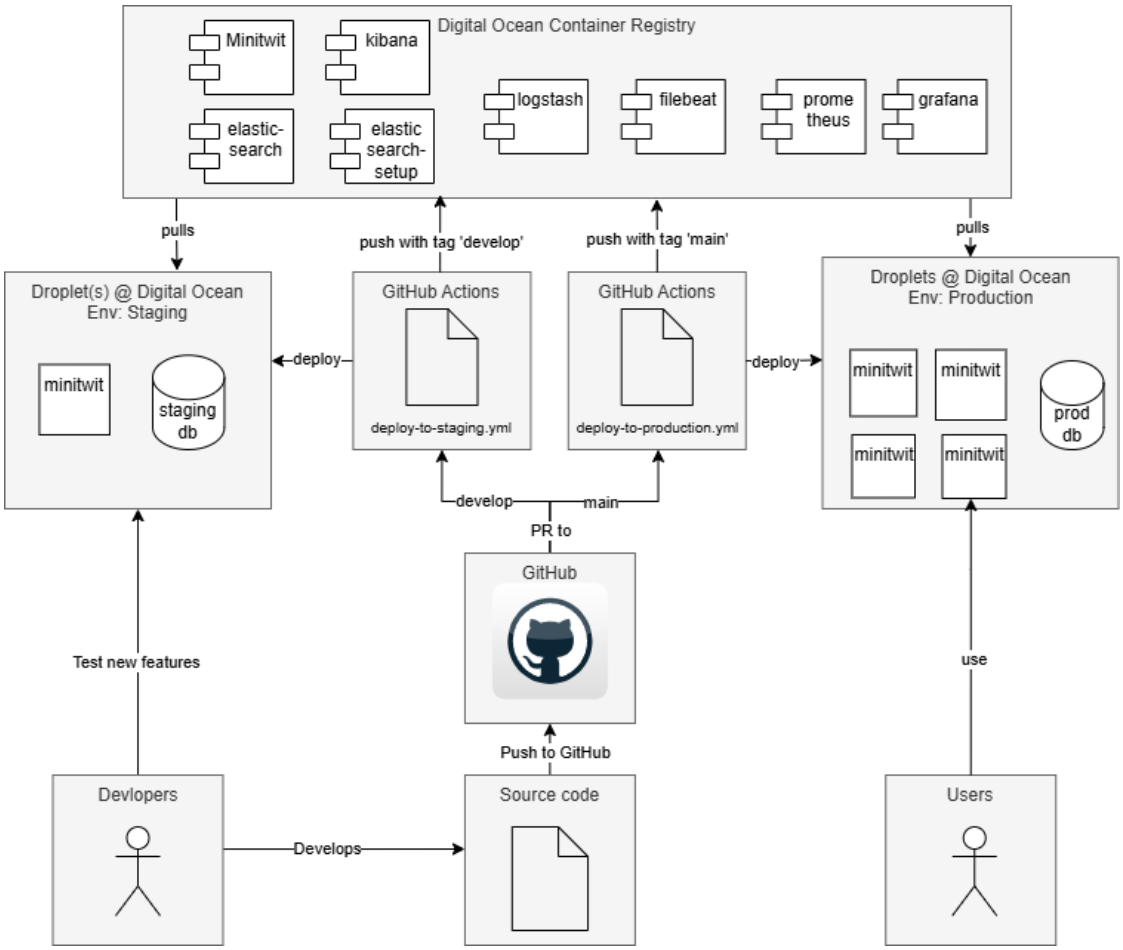
A description and illustration of the:

[seb/nick] Design and architecture of your *ITU-MiniTwit* systems

The current achitecture of the minitwit system.



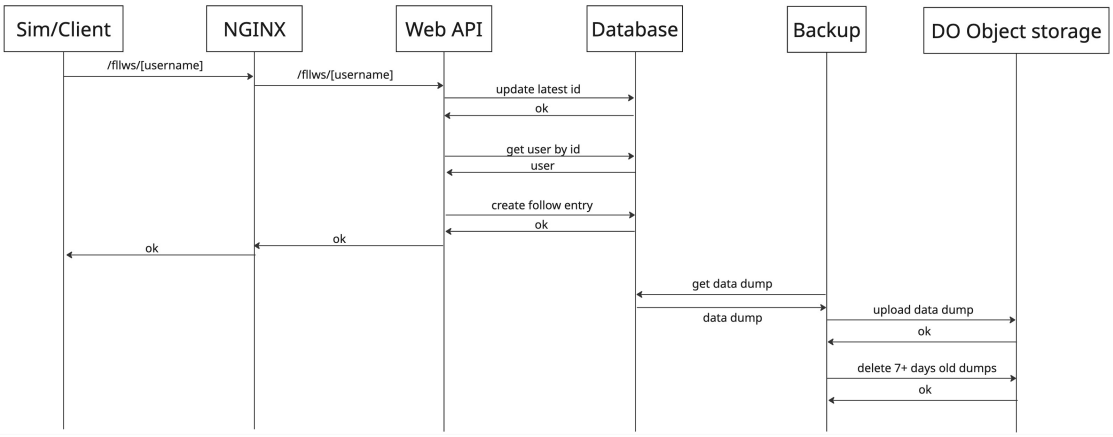
The current deployment flow.



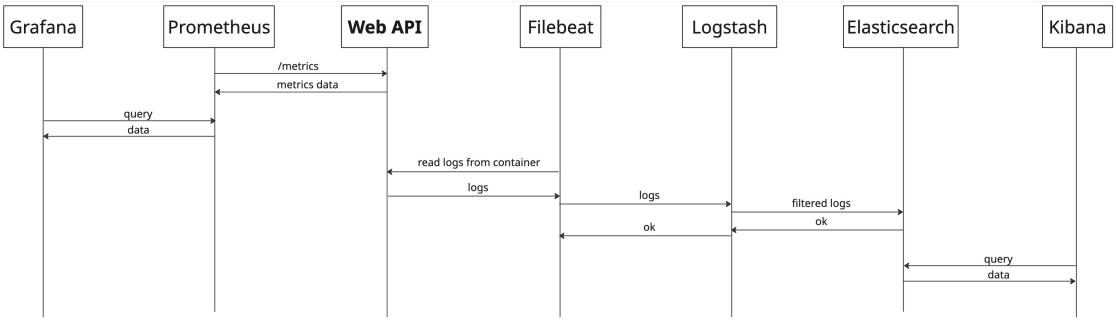
[Nic] All dependencies of your *ITU-MiniTwit* systems on all levels of abstraction and development stages. That is, list and briefly describe all technologies and tools you applied and depend on.

[Nic] Important interactions of subsystems.

Both the simulator and client contact the same API application, so both sequence diagrams look identical. The following sequence diagram uses the simulator request endpoint `/fllws/[username]` as the baseline. The following sequence diagrams does not take Docker Swarm into account, as the underlying communication is hidden.



For monitoring and logging, we have also included a sequence diagram to show how they interact with each other.



[G] Describe the current state of your systems, for example using results of static analysis and quality assessments.

At the current state of the project all major functionalities are implemented and out on production with only a few minor issues remaining, which are listed on the Github repository's Issues page here.

Throughout development we also used static analysis tools such as SonarQube and CodeClimate to ensure code quality. The quality assessment reported by SonarQube can be seen on the image below:

We managed to avoid/solve *Security* and *Reliablity* issues, and the remaining *Maintainability* issues mainly included "multiple string literals" problems, which we deemed non-crucial. Code duplication was also kept at minimal, coming in at 7.9% for the the entire codebase (*Note: source code provided by the course, such as the flag_tool or the simulator was excluded from the quality assessment*).

[ALL] MSc students should argue for the choice of technologies and decisions for at least all cases for which we asked you to do so in the tasks at the end of each session.

Process

This perspective should clarify how code or other artifacts come from idea into the running system and everything that happens on the way.

In particular, the following descriptions should be included:

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

In the following section, we will discuss the CI/CD pipeline of our system, and for this, we discuss two key branches: `main` and `develop`. The `main` branch includes the code running on our production environment, and `develop` branch includes the code running on our staging environment. For the sake of communication, we will simply address these branches by `production` and `staging`.

We use GitHub for handling our repository and tracking the process with their issue system. We use a branching strategy, where features written in issues are worked on in `feature`-branches. Once ready, they are then merged into `staging` and then into `production`. This enables us to test and deploy the feature before production, at the cost of slightly longer delivery times. This means that for features to make it through to production, it includes three phases:

1. We work on the issue using a `feature`-branch. Developers work on and finalize the feature on this branch.
2. Once ready, a pull-request is created to merge the `feature`-branch into `staging`, where tests, linting, static code analysis and a fellow team member, must pass or approve the request, before being able to merge it into staging.
3. Once deployed to the staging environment, if the staging environment sees no failures and passes a manual test, a pull-request into `production` is made. Once approved by tests, linting, static code analysis and a fellow team member, the feature is pushed into main.

Automated Testing and Quality Gates

Pull-requests as well as pushing to staging and production, include several tests that are performed using workflows that trigger a GitHub action, which builds a Docker container with which these tests can be performed. On top of the web API container, an associated PostgreSQL database is instantiated, to perform E2E and simulation tests.

- Unit tests are performed using Ruby Rack
- E2E tests are performed using Playwright
- Simulation tests are performed by instantiating a new environment, and using Python to perform requests
- Static code analysis using SonarQube, which requires ≤3.0% code duplication in the Ruby application.

GitHub branch protection rules ensure that developers follow this workflow. Concretely it prevents users from merging directly into the `develop` and `main` branch.

On top of the above, Ruby and Docker code is formatted and linted on push to any branch. This is done using the GitHub action modules `standardrb/standard-ruby-action@v1` and `hadolint/hadolint` respectively.

Build and Deployment Process

We deploy using GitHub Actions, which builds containers and uploads them to Digital Ocean's container registry. We also upload a new version of the Ruby application, and if there are updates to the configs for either our monitoring or logging stack, we also push a new version of that. If tests pass, then we automatically continue to deploy. Currently we differentiate between containers designated for the staging and production environment by assigning them such a tag.

The deployment process involves SSH'ing into the manager node of the Docker Swarm, that is running as droplets (virtual machines) on Digital Ocean, and running a deploy.sh script, which simply pulls the newest version of the stack from the container registry.

On pushes to main, we automatically create a new release, which includes bumping the application with a new minor-update, meaning 1.0.0 turns into 1.1.0. If we wish to introduce a patch or major update, we can specify in the commit message.

We orchestrate the containers using Docker Swarm, and given the size of our application, we currently follow the direct deployment rollout strategy, where we simply push a new version to all worker nodes at once. This is a point for improvement.

Environment Management and Infrastructure

We have manually set up instances using Digital Ocean's interface, but have prepared a Terraform script for setting up a new environment in the future. For artifact management, we use Digital Ocean's container registry, where we only differentiate between container versions using staging and production tags.

To distribute secrets that GitHub Actions can access, we set up GitHub secrets to keep an access key to Digital Ocean on which we deploy our application.

Rollback Strategy

To roll back, it would require manually SSH'ing into the server and modifying the compose script to depend on a specific container in Digital Ocean's registry. This is definitely a weak point, making it time consuming to rollback and represents an area for future improvement.

Monitoring and Observability

Once the feature is successfully integrated into the production codebase, we use Prometheus and Grafana to monitor the application, ensuring that the feature introduces no error, and that operation levels remain the same. In case of noticeable changes, we use Kibana to navigate logs to help diagnose the problem. Kibana queries Elasticsearch, which receives logs from Logstash, who in turn accesses log-files using Filebeat.

Choice of Architecture & Technologies

We chose to have a staging environment, as it helped us understand how to properly integrate features and architecture changes before proceeding to do so on production. Given this projects work included a lot of architecture change and adding new technologies, this helped us immensely in preventing down-time by ensuring that the config worked on a deployed environment.

We chose Github as it is a well establish standard for git and code control. On top, it granted us access to Github Actions, which is a great tool for building workflows to establish a CI/CD pipeline. Github Action was a great choice, as it integrated well into the Github environment.

Using Digital Oceans container registry became our choice given it's integration with the DigitalOcean platform. Only after using it, did we realize that the biggest upsides are primary for users of digital oceans other deployment tools, that are not running on rented VMs. Given the price of DO's container registry, we would most likely migrate to Docker Hub, if the project had continued.

We chose SonarQube for static analysis as it gave us an ability to understand code duplication while being simple to integrate into our CD/CI pipeline.

[Z/G] How do you monitor your systems and what precisely do you monitor?

Monitoring is configured in our system using Prometheus and Grafana. Prometheus handles time-series based raw metric collection, Grafana handles metric visualization.

How it works in our system

Our Minitwit application uses an existing Ruby client library of Prometheus, and exposes raw metrics on the `/metrics` endpoint. The Prometheus service periodically scrapes the data from this endpoint of the application. In Grafana, these collected metrics are visualized using highly customizable dashboard panels. In each panel, metrics from Prometheus are queried at regular intervals using PromQL queries. We also set up email alerting for certain panels, this way we can be notified when certain conditions, thresholds etc. are met.

Panels configured on our Grafana dashboard

- **HTTP response count by status codes:** Time-series. Shows the number of HTTP responses during the last minute at any given point of the time range, grouped by status codes.

- **HTTP error response count:** Time-series. Shows the number of HTTP client- and server-side error responses during the last minute at any given point of the time range. Email alerting is also set up for this panel, when the server-side (5XX) error count during the last minute hits a certain threshold.
- **Latency percentiles:** Time-series. Shows the median, 95th and 99th percentiles of latency (request duration) in milliseconds at any given point of the time range.
- **Average latency:** Gauge. Shows the average latency in milliseconds over the given time range.
- **Total registered users:** Stat. Shows the total number of registered users in the Minitwit application.

As seen in the list above, aside from the *Total registered users* panel, we mainly do infrastructure monitoring in our system. We also planned to include more meaningful application-specific monitoring too such as the number of new users/new posts made in the last X minutes; due to time constraints, however, we did not implement these.

[Z/G] What do you log in your systems and how do you aggregate logs?

Logging is configured in our system using the ELFK stack: Elasticsearch, Logstash, Filebeat and Kibana.

How it works in our system and the way we aggregate logs

First, Filebeat handles log shipping by reading and collecting logs from each of our Docker containers' log files. These logs are then forwarded to Logstash, which processes and transforms the log data as needed. Logstash then sends the processed logs to Elasticsearch where they are indexed and stored for efficient querying. Finally, the aggregated logs are visualized using Kibana.

The reason we also used Filebeat for log shipping is because it is much more lightweight than Logstash. Traditionally, Logstash is the log aggregator which collects, transforms and forwards logs for further processing, but since we have multiple physical nodes in our system, each node would require one Logstash instance running on them. Instead, each node has a Filebeat instance running which handles log shipping for the containers running on that node.

What we log in our system

Filebeat forwards all logs, from all Docker containers in the system. In Logstash, we filter based on the logging levels (filtering/parsing is specific to each service's logging format). We try to parse each log record to extract the logging level; if the parsing was successful, all *DEBUG*- and *INFO*-level messages are excluded, everything else is forwarded to Elasticsearch for indexing. Additionally, Logstash also drops many unneeded fields in each log record, so that the number of indexed fields will stay relatively small.

[Nic] Brief results of the security assessment and brief description of how did you harden the security of your system based on the analysis.

By running through the OWASP Top 10 list on security assessment, we have done the following analysis:

- A01:2021-Broken Access Control In the system only two levels of access control exist in the system. Either you are a user, who can post, follow and unfollow, or you access as a public user. For user-specific

endpoints, we have not found any vulnerabilities. CORS settings however, allow anyone to access the API. This misconfiguration allows malicious websites to make authenticated requests to the API on behalf of logged-in users.

- A02:2021-Cryptographic We've upgraded from HTTP to HTTPS, but still expose the port of the application, meaning IP:PORT still gives users access to the service in non-encrypted ways, such that network eavesdroppers can capture username and passwords. The hashing algorithm has been upgraded from MD5 to SHA256, but unfortunately without salting, allowing attackers who gain access to the database to easily crack passwords with rainbow tables or brute force attacks. Lastly, the simulator protection-key is hard-coded which means anyone with access to the public github repo, can essentially bypass that security measure.
- A03:2021-Injection We use the ORM Ruby Sequel, which includes sanitization of input before constructing SQL statements. Developers can create raw SQL statements, but we have opted not to do this given the impracticality and security risks.

A04:2021-Insecure Design Given the tiny feature set, we could not find anything particularly noteworthy about the design.

A05:2021-Security Misconfiguration After experiencing a ransomware attack, requiring bitcoin for our data, we closed ports and changed the default password to prevent future attacks. Similarly, we discovered that `ufw` was disabled by the end of the course, which exposes all services to the web. Lastly, we are aware that CORS settings are overly permissive as elaborated in A01.

A06:2021-Vulnerable and Outdated Components Our system has very weak password checking, which allow users to create easily hackable accounts. Simultaneously, weak email validation and not sending a confirmation email makes it particularly easy for bots to create users. In fact, 99.9% of our activity is from a single bot.

On the developers side, we did not require 2FA to log into DigitalOcean, bringing our level of security down to the weakest login-type of the five team members. And technically, Dependabot has been suggesting a Ruby update from `3.3.7` to `3.4.4`, which have been postponed multiple times.

A08:2021-Software and Data Integrity We have not been able to identify any issues regarding this.

A09:2021-Security Logging and Monitoring Failures We experienced a log overflow causing our production service to fail. This failure did not cause any warnings, causing three days of downtime for our application. We will elaborate on how we fixed this when reflecting on system operation.

A10:2021-Server-Side Request Forgery We have not been able to identify any issues regarding this.

[Seb/Nick] Applied strategy for scaling and upgrades.

We have used both horizontal and vertical scaling.

For the logging and monitoring it was nessesary to scale vertical here we scaled from one 1gb-CPU(s-1vcpu-1gb) to two 2GB CPU (s-2vcpu-4gb) to handle the workload associated with monitoring and logging.

We increased the number of node/droplets from 1 to 3 To increase availability we upgraded from docker compose to docker swarm with a docker stack deployment containing multiple replicas of the minitwit

application.

[Nic] In case you have used AI-assistants during your project briefly explain which system(s) you used during the project and reflect how it supported or hindered your process.

This project included the use of both chatbots and in-editor help using copilot. These were provided by OpenAI and Anthrophic.

Copilot increased speed by solving minor problems through it's line-for-line help. To increase the precision, prompt-like comments would be added prior to the line of interest, or specific prompts would be used to concretely specify the desired change.

Chatbots on the other hand involved four primary types of prompts:

- Elaboration: Please explain X technology
- Comparing: What is the difference between X and Y technology
- Creation: I want to X
- Solving: I want X, but instead Y happens.

Elaboration and comparison were primarily used at the planning stage of implementing new technologies, or for developers unfamiliar with existing technologies already implemented.

Creation is used throughout the implementation of technologies or features, but the scale of the issues attempting to address diminishes over time, as the feature or technology becomes more intergrated into the system, and required changes are smaller. As more code is added to the codebase, solving unwanted behavior becomes more important, and makes out large parts of prompts.

Additional reflection on use of chatbots, we found that Claude 3.7 Sonnet provided better code-based responses as well as understanding misconfigurations and bugs. It gives detailed descriptions of different variables and potential flaws in the code and configs. This is measured against ChatGPT o1.

Reflection

Evolution and refactoring

Finishing a sprint and adding a new feature

[Nic] Database migrating from SQLite to PostgreSQL to PostgreSQL

The migration from SQLite to Postgresql happened at a stage, where no active users was using our platform (The simulator was yet to start). This meant that we could safely upgrade without having to move over data, which would have otherwise been a hassle given the SQL dissimilarities.

Given the educational purpose of the project, we later sought out an opportunity to perform a database migration. Such an opportunity arose when database optimization became a necessity. Before optimizing, we deemed it necessary to introduce an ORM, which would improve the developer experience as well as

migration experience going forward. Given the new database structure introduced by the ORM, although quite similar, we needed to migrate from one database with one schema, to another database with another schema. The approach taken involved extracting data from one postgres instance in the shape of SQL Insertion statement, which we then manipulated to fit the new data scheme, and then simply ran the sql insertion statements in the new database.

As soon as the migration was done, we switched to the new application image, meaning we now served requests from the new database. This approach involved having 5 minutes of forgotten data, and 3 seconds of lost availability. We found this price and strategy reasonable, although the 5 minutes of lost data, could have had serious impact on the business. As we will later discuss, we found that using logical replication, proved to be a much nicer approach to copying data.

[Nic] Transition from docker compose to docker swarm (networking problems).

Transitioning onto multiple machines with docker swarm came with multiple obstacles.

[Seb/Nick] Docker compose versioning problem (moving to stack).

Firstly, the docker compose version that supports `docker stack deploy` is a legacy version of docker, and there is a difference in the features and syntax supported which caused some problems. The `docker stack` does not take `build`, `container_name` and `depend_on` into consideration, and also handles unnamed volumes differently. Therefore, we had to rewrite the compose scripts to make them compatible with docker stack. One of the biggest change that also resulted from this is that from this point on, we had to build the configuration files of each service into Docker images, and publish them to our DigitalOcean registry. This came with several complications, such as automatically building images in our workflows during deployment, and ensuring correct versioning/tagging for each image, since publishing all images in every deployment would take a lot of resources and time. This change also solved another one of our problems: previously, we manually copied all configuration files onto the server using `scp`, which made every deployment error prone.

Secondly, the swarm nodes were able to communicate with each other, but self-instantiated virtual networks defined in the docker-compose file, did not propagate to worker nodes, leaving application containers unable to contact the database, and prometheus unable to collect monitoring events. To accommodate the issue, we destroyed and redeployed new virtual machines, and this time used the VPC IP address to define the IP address of the manager node. This meant that workers are referring to the manager using the virtual network layer, and solved the communication issue.

[G/Z] Logging issues (ELK logging resource heavy + too many fields + all fields were indexed, Filebeat on all swarm nodes)

The initial deployment of our logging stack was quite problematic, as Elasticsearch turned out to be very resource heavy, consuming almost ~60-80% CPU at times (before introducing logging, it was ~10% at peak load), and also taking all available RAM. We tackled this in two ways:

- We scaled the droplets vertically, giving them more resources, as previously discussed. This was necessary because the stack has larger minimal resource requirements than what we had.
- We introduced Logstash filters, which dramatically decreased the number of fields indexed by Elasticsearch, lowering its resource consumption greatly.

We also encountered another issue with Filebeat, after switching to Docker Swarm. We found that one Filebeat instance needs to run on each node, because every instance needs direct access to read the containers' log files. This was solved by introducing global replication for not only the Minitwit application, but for Filebeat as well.

[Seb/Nick] Large amount of features cloging up in staging (Impossible to migrate to production)

Implementing new features fully sometimes took longer than a week. Due to the development workflow described earlier the staging environment was used as a development/testing environment. This resulted in features gate keeped by ohter not fully implemented features.

Some of the new features required changing the other features, such as the migration from docker compose to docker stack required a full rewrite of the docker compose scripts.

This made it impossible to migrate the changes to production and delayed the release of the full implementation of the logging and monitoring.

Operation

Keep the system running

[Nic] Database logical replication resulting in db crash

Migrating from docker compose to the docker swarm included the use of postgres feature: Logical replication, which allows postgres instances to live sync data from running postgres instance to the other. This feature is typically used to keep a hot stand-in database ready. In our case, it meant we would actively sync data from the active production database, onto the new production database, and allow us to switch from one stack to the other with zero downtime, as the stand-in database would become the new default.

Unfortunately, after switching a few days later, the pub/sub mechanism of logical replication in Postgres accidentally corrupted a tracking file, meaning the postgres would immediately crash on start. This problem was accomodated by immediately running `pg_resetwal` on startup to reset the corrputed file, and then unsubscriping from the expired subscription. The subscription does not provide any value at this point, as we swapped from the old to the new production machine, and the old one has been turned off.

[G/Z] Log overflow problem. Access denied to machine. Massive clutch

After quite some time into development, our droplet became overwhelmed by the large volume of logs being generated by the different containers. This log overflow consumed all available disk space on our droplet, resulting in the droplet becoming inoperable, also shutting down our whole system. We did not notice this issue for a few days as Grafana also shut down, being unable to send alerts to us. After noticing the issue, we were also denied SSH access into the droplet. We had to use DigitalOcean's recovery console to regain access to the server, delete unnecessary log files, and restore normal operation. Learning from this incident, we introduced log rotation in our docker compose file which limits the maximum number and size of generated log files.

[Nic] Backup strategy (cron job every three hours)

Although it's great to solve problems on your own, sometimes other have done a great job already. And this proved to be the case for backing up a Postgres database. Simply adding the `eeshugerman/postgres-backup-s3:15` container image to the manager node and configuring environment variables, we successfully setup a cron job that automatically backs up daily, and sends the backup to DO's space storage, which is S3 compatible. Using the exact same script, it also includes functions that easily allow restoring from a previous backup. The latter is a crucial step, for when things are burning. The container likewise provide clean-up functionality, such that only 7 days of backups are kept.

Maintenance

Keep system up to date and fix bugs

[Seb/Nick] MAYBE NOT NECCESSAY TO DISCUSS THIS Stale ReadMe.md throughout project

[Seb/Nick] Returning wrong statuscode (Misalignment with simulation)

We implemented the simulator test in our testing workflow. It runs to ensure that the endpoints are available, works and return the correct status codes. After we had implemented the simulator test they failed and we realised that one of our enpoints was misaligned the specification. The endpoint returned the wrong status code. By implementing the simulator tests we discovered the issue in a very early stage.

[Nic] Upgrading to NGINX, setting up ufw, moving to domain

Upgrading to NGINX, we learned multiple things about running a system. First being that having a staging environment to learn how to run command in the correct order proved great to build a shell script that immediately upgrades the production service. Second, that although we had configured our ufw with all the right ports, actually the service was disabled, which it is by default. Only by realizing that the 443 port was open although no specified, did we realize that ufw needs to be actively activated. From this we gathered, that it is important to double check firewall and other security measures, to ensure they're configured properly. Luckily, our database was not exposed by PORT from the docker network, and therefore inaccessible, but having full access to other ports may have exposed other vulnerabilities on the machine.

[Nic] Simulator IP protection stopped sim access (causing errors)

Refactoring simulator requests to only be accepted from a single IP address helped us prevent other malicious users from interfering with the active simulation requests. However, when the new update was pushed, unfortunately the IP protection feature also protected us from the actual simulation IP. Although this worked perfect locally, moving it onto production showed that the feature declined all sim api requests. From this we learned about the importance of being able to quickly roll back an update. During this experience, we found that we did not have a previous versioned container ready to roll back to, and instead had to allow all IP's which was possibly by passing in `ACCEPTED_IPS=*`, which essentially disabled the IP protection.

Style of work

Reflect and describe what was the "DevOps" style of your work.

[Seb/Nick] Reflect on the workflow. Extensive Friday meeting. Split work into three groups

Each Friday after the lectures we met up to have an extensive meeting about the current state of the project. First we shared what features we had worked on the past week, what kind of problems we had faced and how we had solved them. We did this to keep everyone up to date with all the different new technologies and featues implemented.

Second we discussed the content of the lecture and inspected what new features to implement in the next week. Then we discussed *how* to implement the new features and the pros and cons of the different options. For each tasks we created a new issue on GitHub.

Third we discussed *when* to implement the new issues. The group members had different schedules and varying capacity due to other commitments such as handins for other courses. We took this into considaration when we delegated the work. We typically worked in three teams:

1. Nicolaj
2. Gabor and Zalan
3. Sebastian and Nicklas

These Friday meetings worked very well for us, as we all had a very busy schedule. These meetings allowed us to delegate the work, inspect the progess, adapt the plan and be up to date in terms of the implementation details. While still going in depth into the subjects.

[Seb/Nick] MERGE THIS INTO PROCESS SECTION Development environemnt: local => branch => staging => production

As explain in the Proceess section we wen developing a new fea

When developing new features you branch off `develop` then implement the changes and test them **locally** via the local docker development environment `dovker-compose.dev.yml`. Then changes are pushed to a remote branch so another person can continue working on the changes. When the feature/tasks is completed a pull request is created. When the changes are approved they merge into `develop` and trigger a new deployment to the staging environment. If the changes work in the staging environment a pull request from `develop` into `main` can be created. Once the pull request is approved a new release and deployment to production is triggered.

That's it folks!

[Seb/Nick] Repo settings. Workflows on merge. Require 1 team member on pull requests.

This section is also described in the Process Section

To support and enforce the development workflow of new features as explained in Process Section we have setup branch protection rules via Github. For the `main` and `develop` branch the rules are:

1. No direct merge into protected branch.
2. Changes must be approved by at least team member
3. Workflows and test must pass

This ensures that all changes to the protected branches have been approved and tested.

[Seb/Nick] Development environemnt: local => branch => staging => production

As explain in the Proceess section we wen developing a new fea

When developing new features you branch off `develop` then implement the changes and test them **locally** via the local docker development environment `dovker-compose.dev.yml`. Then changes are pushed to a remote branch so another person can continue working on the changes. When the feature/tasks is completed a pull request is created. When the changes are approved they merge into `develop` and trigger a new deployment to the staging environment. If the changes work in the staging environment a pull request from `develop` into `main` can be created. Once the pull request is approved a new release and deployment to production is triggered.