IT-University of Copenhagen

DevOps Minitwit - Group A

Martin Meldgaard Holst - (mmho@itu.dk),
Hampus Michael W. Iversen - (haiv@itu.dk),
Niels-Frederik Brinck - (nieb@itu.dk),
Emil Knudsen - (emkn@itu.dk),
Marcus Norton Quistgaard - (marq@itu.dk)

June 5, 2021

Course code: KSDSESM1KU

Contents

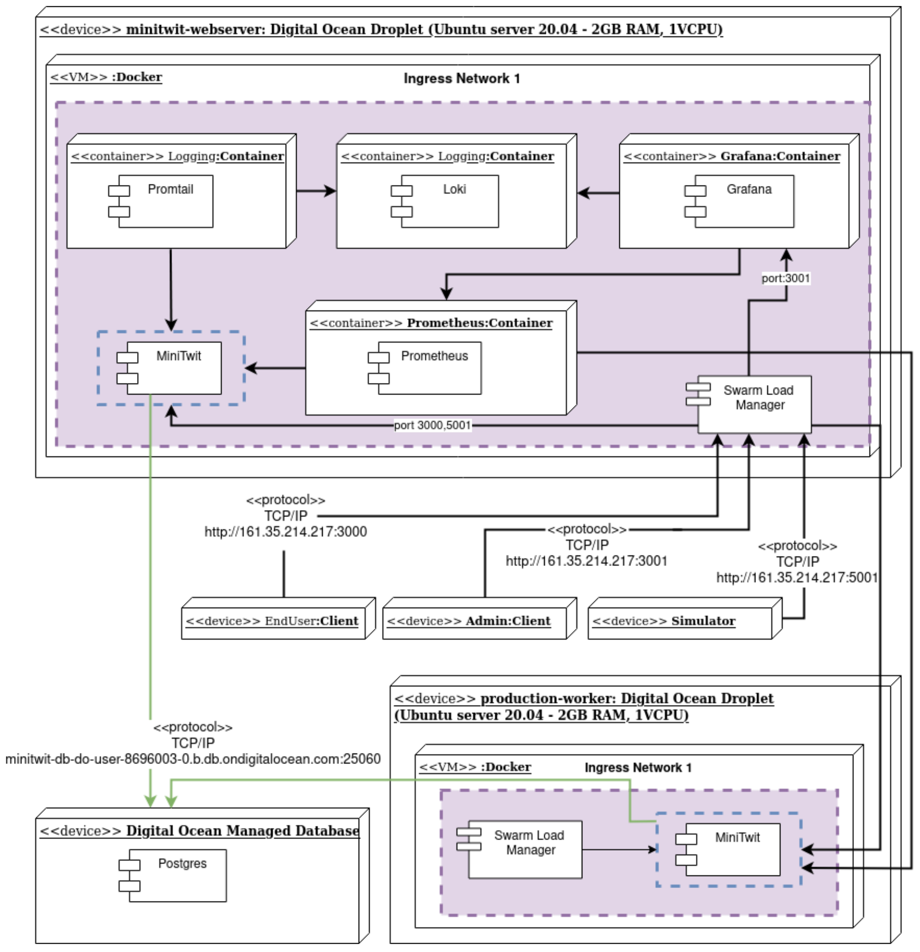
1 System's Perspective

In the sections below, the architecture and deployment specifications of the MiniTwit system will be covered.

1.1 Architecture and Design

1.1.1 Overview

Below is the deployment diagram of the complete system hosted with Docker on DigitalOcean:



<<device>> minitwit-webserver: Digital Ocean Droplet (Ubuntu server 20.04 - 2GB RAM, 1VCPU)
<<VM>> :Docker
Ingress Network 1
<<container>> Logging:Container
Promtail
<<container>> Logging:Container
Loki
<<container>> Grafana:Container
Grafana
port:3001
<<container>> Prometheus:Container
Prometheus
MiniTwit
Swarm Load Manager
port 3000,5001
<<protocol>>
TCP/IP
http://161.35.214.217:3000
<<protocol>>
TCP/IP
http://161.35.214.217:3001
<<protocol>>
TCP/IP
http://161.35.214.217:5001
<<device>> EndUser:Client
<<device>> Admin:Client
<<device>> Simulator
<<protocol>>
TCP/IP
minitwit-db-do-user-8696003-0.b.db.ondigitalocean.com:25060
<<device>> production-worker: Digital Ocean Droplet (Ubuntu server 20.04 - 2GB RAM, 1VCPU)
<<VM>> :Docker
Ingress Network 1
Swarm Load Manager
MiniTwit
<<device>> Digital Ocean Managed Database
Postgres

Figure 1: System Architecture (Click to see full size image)

Green arrow: database connection.
Blue dashed rectangle: containers abstracted into a component for readability.

The MiniTwit application runs on two droplets on Digital Ocean. A managed database, hosted on Digital Ocean, is hosting a Postgres database. The *minitwit-webserver* droplet is running in Docker swarm mode, and functions as a swarm manager [2]. The *production-worker* droplet is a worker node within this swarm. The swarm hosts the MiniTwit component in four replicas. Furthermore, the swarm hosts the monitoring and logging stack in a single replica. This stack consists of Promtail and Loki to fetch and aggregate logs, Prometheus to collect and perform metric querying, and finally Grafana to display the monitored and logged data. The swarm is run with a docker-compose.yml file, which can be seen in appendix 4.1.
Lastly, the swarm utilizes DockerHub to pull all images for the different services.

The MiniTwit component depicted, is an abstraction to increase readability in the UML drawing. It is therefore not a single container hosted in the swarm, but rather a component consisting of multiple services. Herein located; front-end, Sim-API for the simulator, and the API for the front-end. Figure 2 below is a decomposition of this MiniTwit component:

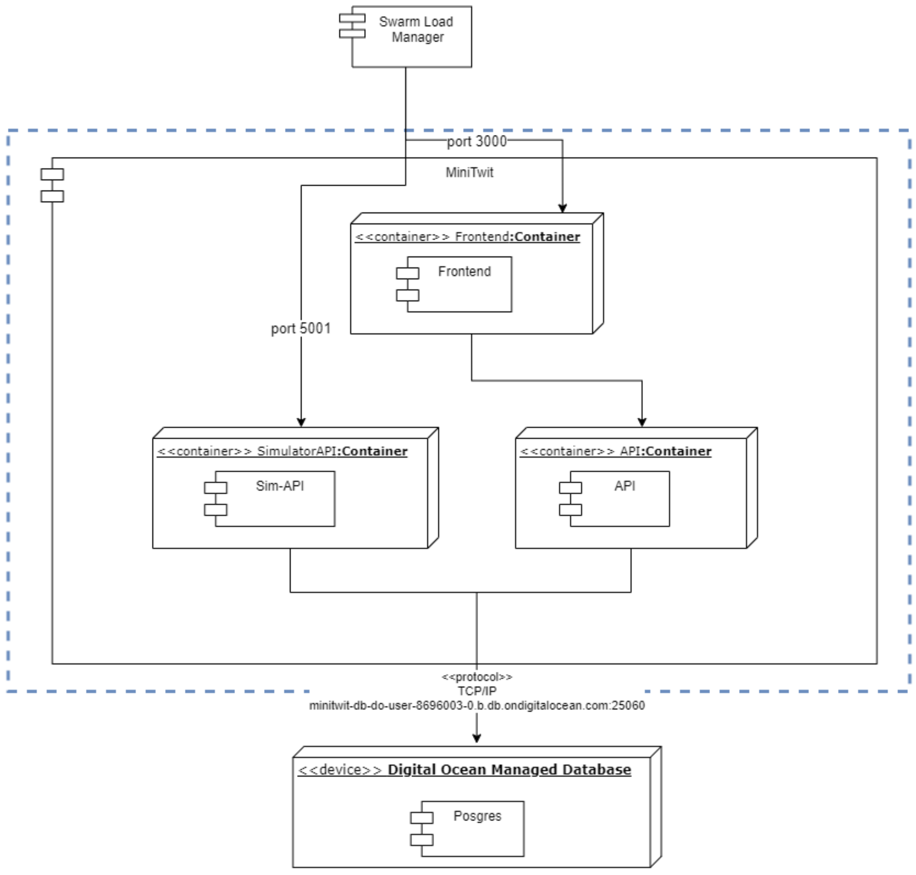

Swarm Load Manager
port 3000
MiniTwit
<<container>> Frontend:Container
Frontend
port 5001
<<container>> SimulatorAPI:Container
Sim-API
<<container>> API:Container
API
<<protocol>>
TCP/IP
minitwit-db-do-user-8696003-0.b.db.ondigitalocean.com:25060
<<device>> Digital Ocean Managed Database
Posgres

Figure 2: MiniTwit Component Decomposition

1.1.2 Design Choices

The *Frontend* service is written in React.js. Both the *Sim-API* and *API* runs on *Node.js* and exposes a RESTful API implemented with *Express.js*.
The rationale behind using JavaScript was mainly due to its lightweight approach compared to e.g. C#. Furthermore, the team wanted to have experience with a new language and frameworks. Lastly, developing with JavaScript is quick and eases the development process through libraries like Nodemon, which allows for recompilation at run-time. The component diagram of the back-end can be seen below in figure 3:

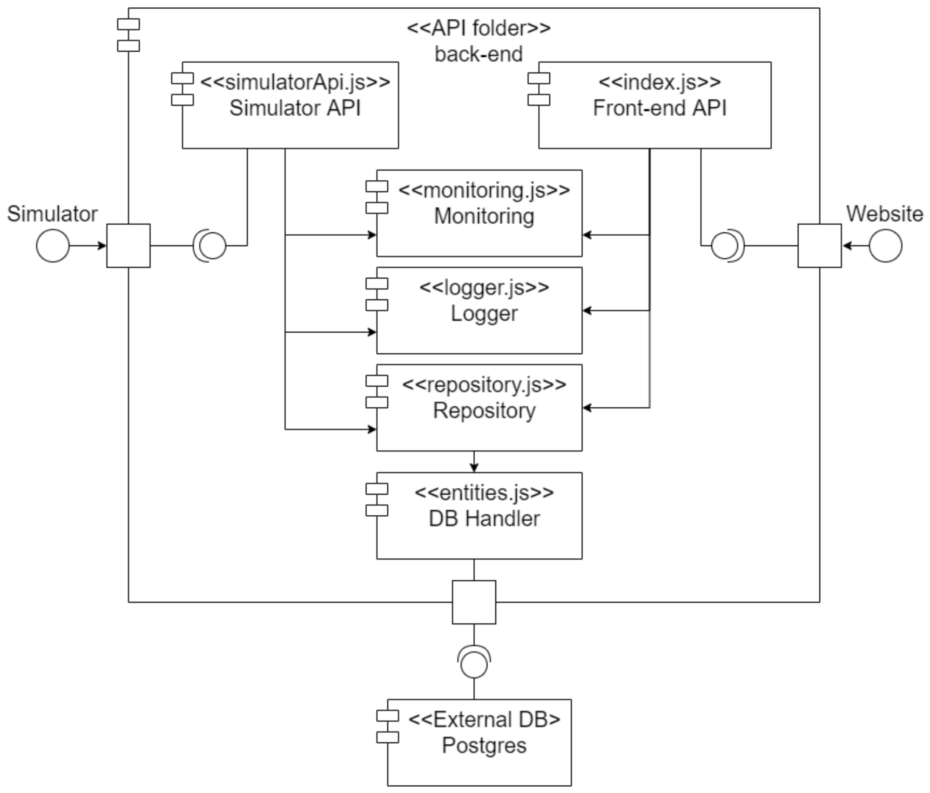


Figure 3: Back-end Component diagram

Each component in the back-end is represented by a single file. Both APIs implement the same endpoints, and uses the rest of the components as helpers to resolve each request.
The following list briefly describes each component:

1. **Monitoring:** Provides functionality to collect metrics used by Prometheus
2. **Logger:** Provides logger objects to perform logging by utilizing the *winston* and *express-winston* libraries.
3. **Repository:** Provides a repository object, which uses Sequelize to query the database.
4. **DB Handler:** Provides a Sequelize ORM object to query the database.

A Postgres database was chosen due to the team's previous experience, and because of its relational nature, which fits the purpose of this project. The Sequelize library is used in the *DB Handler* as the database abstraction layer. It was chosen because it is well documented [9], and has a large community on Stackoverflow.

1.2 Dependencies

This section will cover all the dependencies in the system on all layers of abstraction.

1.2.1 Front-end and Back-end Low Level Diagrams

The following is a description of the syntax used in the next two diagrams:

- Squares: files developed by the team
- Circles: external libraries
- Arrow: build-/compile-time dependency
- Dashed arrow: run-time dependency

The Front-end has the following Dependencies:

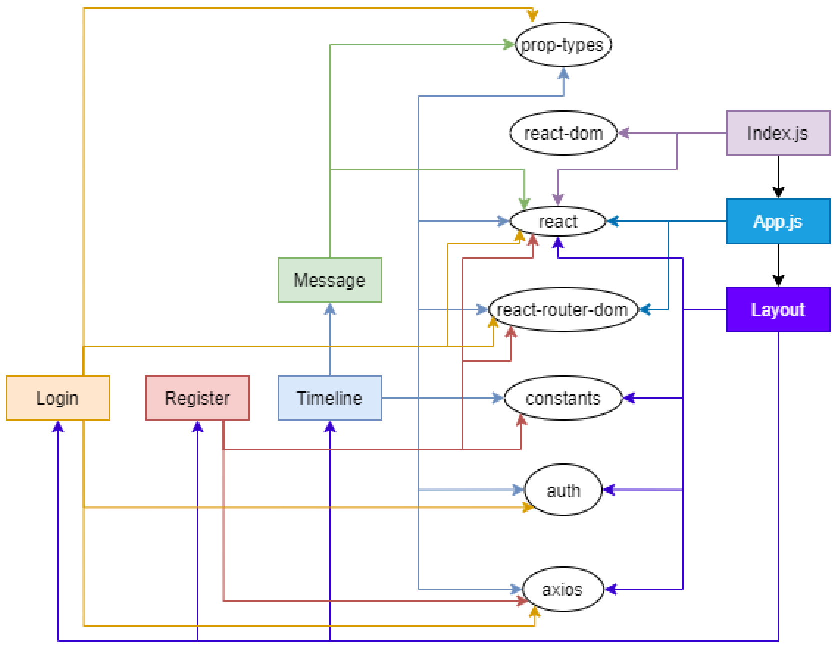


Figure 4: Front-end dependency diagram

The squares represent React components. Colors are used to improve readability.
The source code represented in this diagram is located in the Minitwit/Frontend folder. The exact versions of all frontend dependencies can be seen in appendix 4.3

The back-end has the following dependencies:

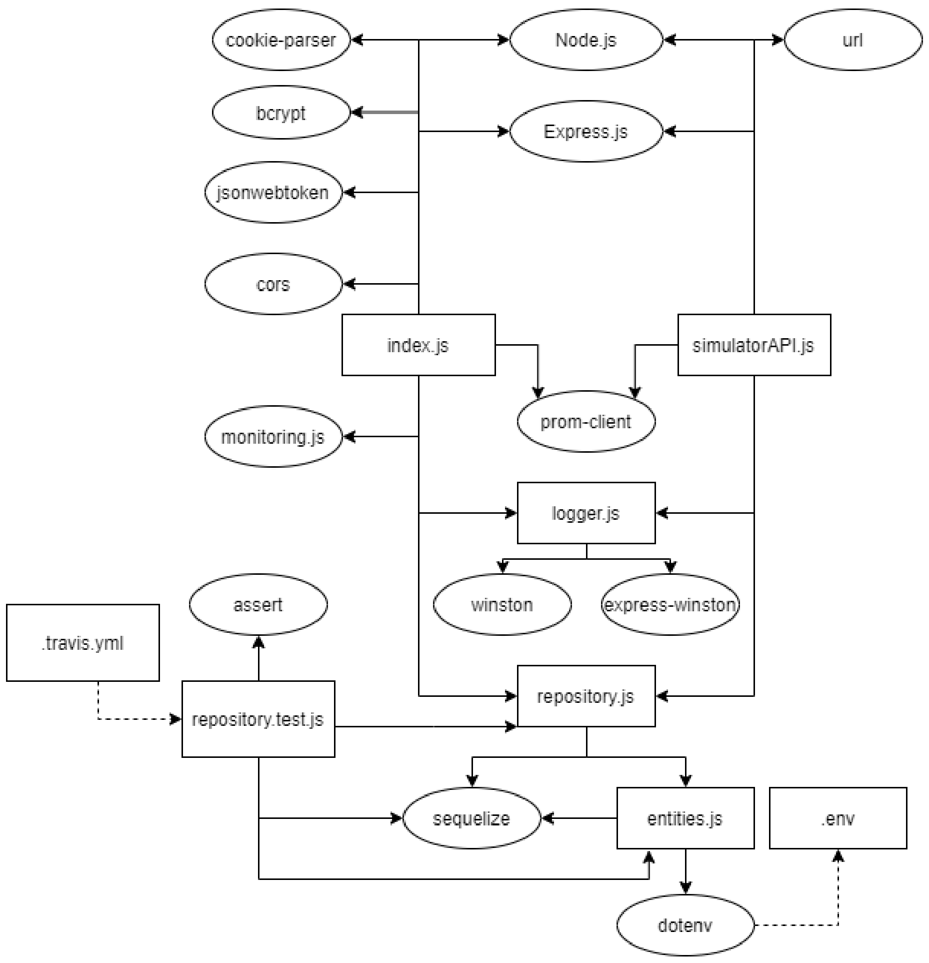


Figure 5: Back-end dependency diagram

The source code represented in this diagram is located in the Minitwit/API folder. The API used when accessing the system through the webpage is handled by the *index.js* file, while the simulator uses the *simulatorAPI.js* file.
All dependencies in the front-end and back-end use the latest major release, e.g. the *bcrypt* dependency uses ˆ5.0.0, meaning all releases up until the next major release: v6.0.0.
The exact versions of all backend dependencies can be seen in appendix 4.4

1.2.2 The Whole System

The following describes the syntax used in the diagram:

- Full line arrow: dependency
- Stippled line arrow: retired dependency
- Colored line arrow: for readability purpose only

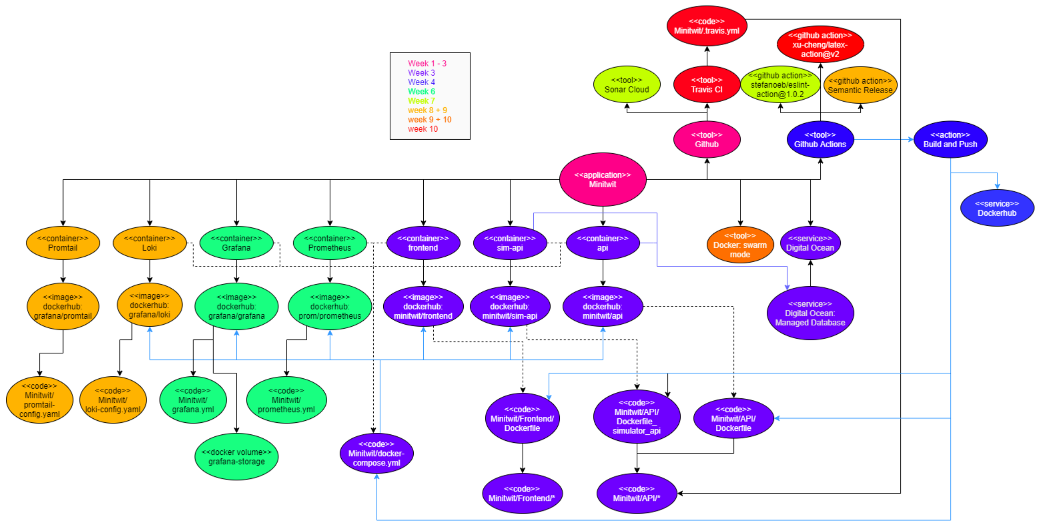

Week 1 - 3
Week 3
Week 4
Week 6
Week 7
week 8 + 9
week 9 + 10
week 10
<<code>> Minitwit/.travis.yml
<<github action>> xu-cheng/latex-action@v2
<<tool>> Sonar Cloud
<<tool>> Travis CI
<<github action>> stefanoeb/eslint-action@1.0.2
<<github action>> Semantic Release
<<tool>> Github
<<tool>> Github Actions
<<action>> Build and Push
<<application>> Minitwit
<<service>> Dockerhub
<<container>> Promtail
<<container>> Loki
<<container>> Grafana
<<container>> Prometheus
<<container>> frontend
<<container>> sim-api
<<container>> api
<<tool>> Docker: swarm mode
<<service>> Digital Ocean
<<image>> dockerhub: grafana/promtail
<<image>> dockerhub: grafana/loki
<<image>> dockerhub: grafana/grafana
<<image>> dockerhub: prom/prometheus
<<image>> dockerhub: minitwit/frontend
<<image>> dockerhub: minitwit/sim-api
<<image>> dockerhub: minitwit/api
<<service>> Digital Ocean: Managed Database
<<code>> Minitwit/ promtail-config.yaml
<<code>> Minitwit/ loki-config.yaml
<<code>> Minitwit/ grafana.yml
<<code>> Minitwit/ prometheus.yml
<<code>> Minitwit/Frontend/ Dockerfile
<<code>> Minitwit/API/ Dockerfile_ simulator_api
<<code>> Minitwit/API/ Dockerfile
<<docker volume>> grafana-storage
<<code>> Minitwit/docker-compose.yml
<<code>> Minitwit/Frontend/*
<<code>> Minitwit/API/*

Figure 6: System dependencies. Click here for full size image

The diagram displays the dependencies of the project at a high level of abstraction. The various dependencies are color-coded, such that the color corresponds to the week, the dependency was introduced.
The dependencies of the front-end and back-end are omitted as they are described in figure 4 and 5.

1.3 The Current State of The System

SonarCloud provides an overview of the general code quality for the MiniTwit application as seen in figure 7 below:

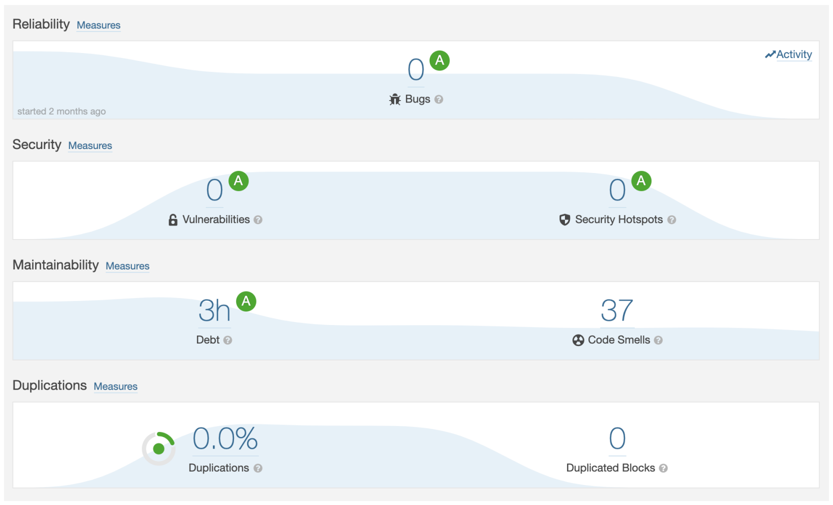


Figure 7: SonarCloud overview of code quality

From this overview, it is seen that the state of the system is as desired, with room for improvement on the maintainability measures. For this project, SonarCloud shows an estimated technical debt of 3 hours distributed between 37 code smells. The code smells mainly comprise of outcommented code, which is solved simply by removing it. SonarCloud estimates the time consumption to solve each of these code smells to be 5 minutes, which makes the technical debt seem higher than what it is in reality:

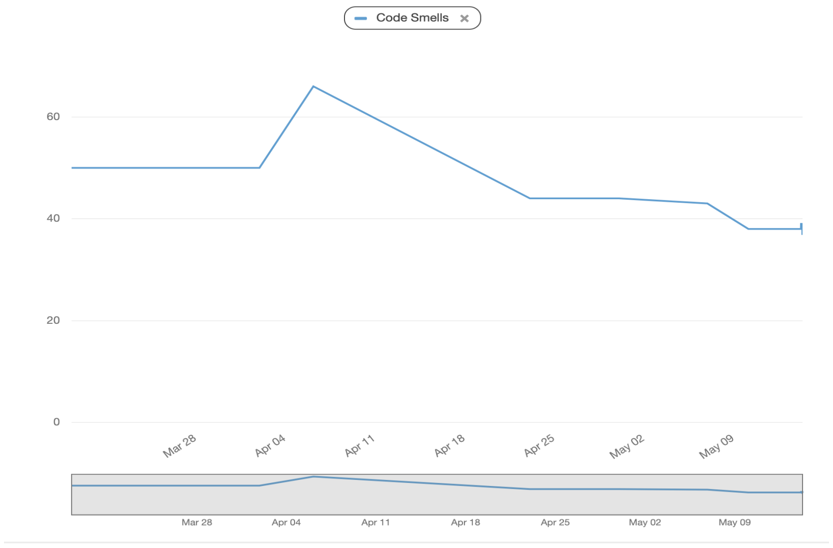


Figure 8: Amount of code smells per date

As seen in figure 8, the number of code smells has steadily decreased since adding ESLint to the pipeline, which will be elaborated in section 2.2. To secure less code smell in the future, ESLint could have been configured to be more strict.

1.4 License

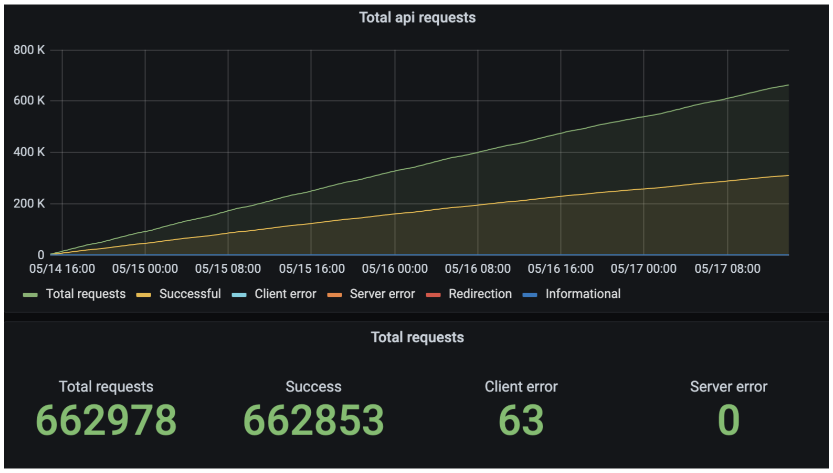
The ScanCode toolkit[7] was used to scan for all licenses within the MiniTwit repository. The scan here outputted several permissive licenses, where the most relevant were MIT and Apache-2.0. Furthermore, the most important finding was that no version of the GPL license was present. This is important, as the GPL is a copyleft license that would require MiniTwit to be a free open-source license[12]. Hence, since no copyleft licenses such as GPL are in the project, MiniTwit has a variety of licenses to choose from.
The group chose the permissive MIT license to provide everyone with the free right to use the API, as well as keeping the developer team free from being liable for any claims made against any possible damage caused by the API.

1.5 Service Level Agreement

The SLA can be found here.
The SLA mentions that the API guarantees an up-time of 99.99%. This metric is calculated by the following equation:

$$UpTimePercentage = \frac{TotalTransactionAttempts - FailedTransactions}{TotalTransactionAttempts}$$

Due to an error causing stored monitored data to be wiped prior to the the 14th. of May, the following data from 14/05 - 17/05 was used for the calculation:



Total requests	Success	Client error	Server error
662978	662853	63	0

Figure 9: Total requests 14/05 - 17/05

We here chose to count all of the client errors as Failed Transactions as the client errors were caused by users not being registered in the MiniTwit application due to a server error. Hence, the 99.99% is derived from the following:

$$99.99\% = \frac{(662978 - 63)}{662978}$$

Furthermore, the SLA states a response time on POST and GET messages. It here states a POST response time varying from ~0.0250s to ~1.02s and a maximum GET response time of 100ms. Both of these are derived from the following figure: 14

2 Process' perspective

2.1 Team Organization and Interaction

The team created a Discord channel to function as the main communication channel throughout the project. Here, the team met each Monday after the lecture to discuss what work had been completed since last week, what work had to be completed by the end of the current week, and who should do which tasks.
The team used Jira's Kanban Board feature[14] to gain a better overview of the various tasks. The Kanban consisted of the following columns: "To Do", "In Progress", and "Done". Hence, the team could move the tasks to the corresponding status of its progress, such that the state was visible to the rest of the team members.

2.2 CI/CD Pipeline and Tools

The CI/CD pipeline uses GitHub actions, Travis CI, ESLint, SonarCloud, and Semantic Release. GitHub actions are run on both the Develop and main branch when a pull request or push to the repository is made.

ESLint is used through the GitHub action *stefanoeb/eslint-action@1.0.2* [3] and is performing static code analysis by checking line indentation, unused variables, props passed to React components, etc. Travis CI is integrated into the MiniTwit repository and is used to run all unit tests within the MiniTwit/API folder [13]. SonarCloud is also used to perform static code analysis and checks code smells such as code duplication exceeding 4%. It also scans for potential security vulnerabilities [10].

2.2.1 Pull Request

On a pull request, the Travis CI, SonarCloud, and ESLint GitHub action are all being run. If they all pass the developers are allowed to perform the merge.

2.2.2 Push

On a push to the main or Develop branch, the *Build and Push* GitHub action is run. The jobs build the Frontend, API, and Sim-API Docker images and push them to Dockerhub. Once all three jobs have finished, a fourth job is started, which SSHs into the swarm manager droplet on Digital Ocean. This job updates the corresponding running services in the swarm by using the *docker service update –image name-of-image* command. The SSH action is *appleboy/ssh-action@master*[11]. Below is an image from the GitHub action log showing the four jobs:

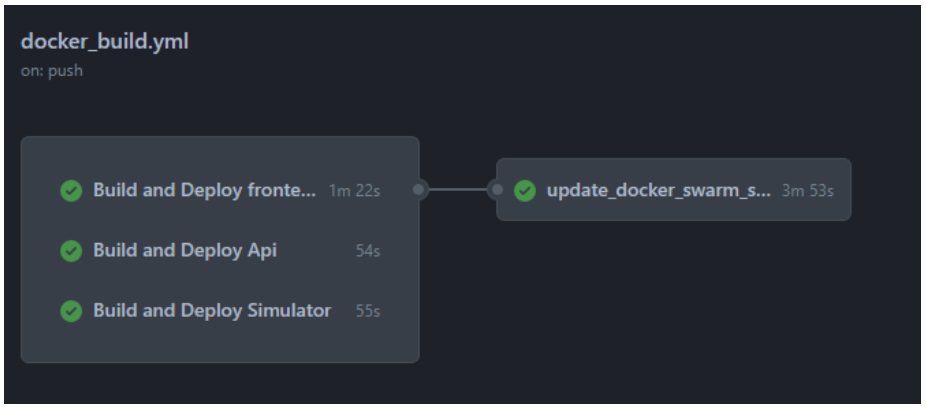

docker_build.yml
on: push
Build and Deploy fronte... 1m 22s
Build and Deploy Api 54s
Build and Deploy Simulator 55s
update_docker_swarm_s... 3m 53s

Figure 10: Build and deploy action dependency graph

Furthermore, a GitHub action with the name *Create Release* running *Semantic Release* will act on any push to the main branch with a commit name with the prefix *fix*, *feat* or *BREAKING CHANGE*. If the commit's name fulfills the prefix, the *Semantic Release* will create a new release on the main branch [8].

A last action named *Latex to PDF* is run in the *latex_to_pdf.yml* file, which compiles the *main.tex* file in the Minitwit/report/directory folder. This action compiles the *main.tex* file using the GitHub action *xu-cheng/latex-action@v2* [5] and outputs it as *report.pdf*. It then commits and pushes the new file to the main branch in the same directory.

2.3 Organization of Repository

The MiniTwit application was built using a mono-repository on GitHub. The structure in the repository consisted of a sub-folder for the API (API for the front end and the Simulator API), a sub-folder for the Frontend React Application, and a sub-folder for the report written in Latex.

2.4 Applied Branching Strategy

The team applied the task-branching strategy[4] together with having both a Main and Develop branch. The Develop branch was initially a mirror of the Main branch and is where new functionality is developed and tested. Once a task is created on Jira and a team member has been assigned to develop it, the developer will create a branch with the task name off of the Develop branch. Once completed and tested successfully, the branch is merged into Develop. Finally, Develop was merged into Main once a week. However, this was not happening weekly in the first weeks of the project due to the team being behind schedule.

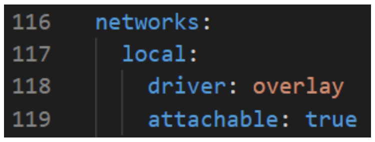
2.5 Applied Development Process and Tools Supporting It

The team tried to use pair-programming in an informal manner as much as possible, as opposed to assigning each member with their own task. Pair programming was preferred to allow most possible members to get hands-on experience with the various DevOps themes and to enhance code quality. To facilitate pair-programming, the team utilized Discord's screen sharing functionality as well as Visual Studio Code's Live Share functionality[6]. While screen sharing functionality is self-explanatory, the Live Share functionality provided a way for multiple developers to navigate the documents simultaneously.

2.6 Monitoring and Logging

2.6.1 Metrics Collection

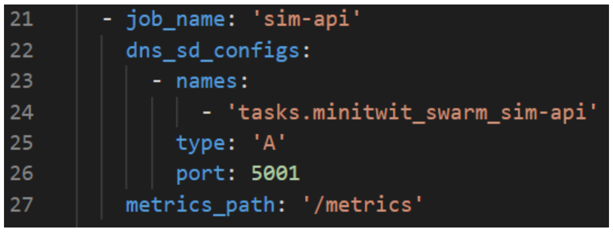
The Prometheus container uses the *local* network which is configured to use an overlay driver in the *docker-compose.yml*:



```
networks:
  local:
    driver: overlay
    attachable: true
```

Figure 11: code snippet from Minitwit/docker-compose.yml

See appendix 4.1 for the full compose file.
The services to be scraped are also running on the *local* network.
This network configuration allows Prometheus to retrieve the virtual IP addresses of all of the api and sim-api service replicas running in the Docker swarm.
Prometheus's scraping job is configured in the *Minitwit/prometheus.yml*:



```
  - job_name: 'sim-api'
    dns_sd_configs:
      - names:
        - 'tasks.minitwit_swarm_sim-api'
        type: 'A'
        port: 5001
    metrics_path: '/metrics'
```

Figure 12: code snippet from Minitwit/prometheus.yml

See appendix 4.2 for the full configuration file.
Here, the sim-api job scrapes all sim-api tasks/replicas by executing a DNS query to retrieve their IP and pull the metrics with the specified *metrics_path* endpoint.

As of the current setup, only the sim-api service is being scraped and not the api service. The monitoring setup uses a pull based monitoring setup where Prometheus pulls the metrics from the services and Grafana pulls the metrics from Prometheus.

Figure 5 shows that both the API and sim-API use the prom-client library to passively monitor endpoint requests and responses by ”sniffing”. Prom-client is used for the whitebox monitoring as part of the API’s middleware to monitor the total requests and responses and the individual endpoints and response codes.

As seen in the figure 13, the Grafana dashboard displays graphs for the total amount of requests and responses as well as more precise counters for the requests and responses of each endpoint in the last hour. Furthermore, figure 14 shows the average response time of some of the endpoints monitored to keep track of bottlenecks.

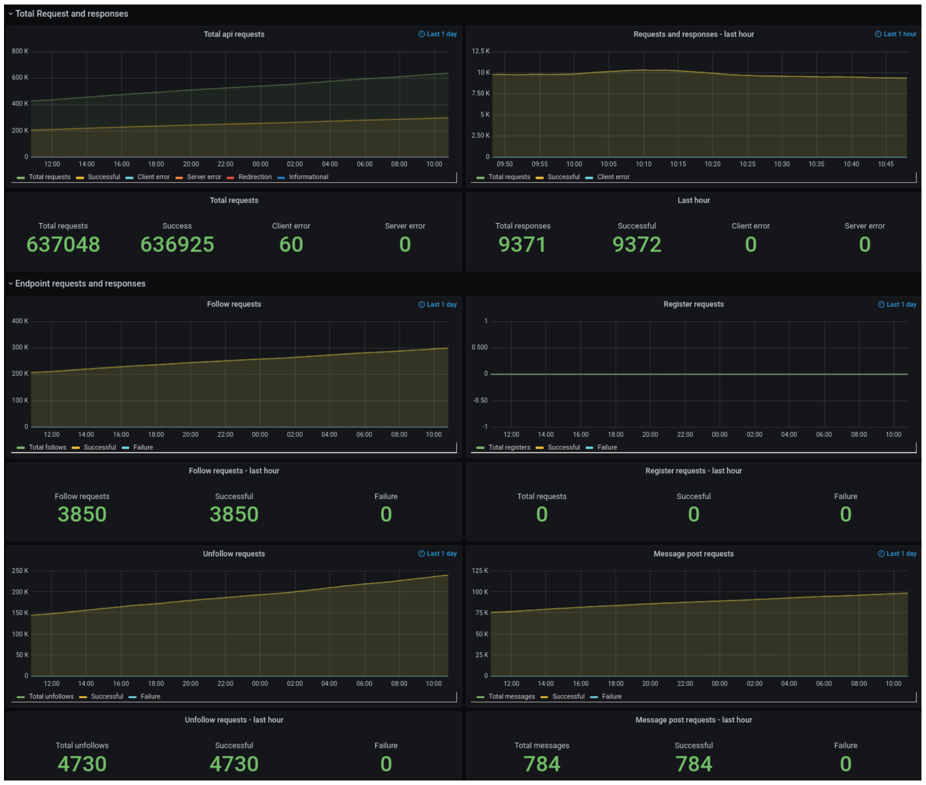


Figure 13: Request/response monitoring. Click here for full size image

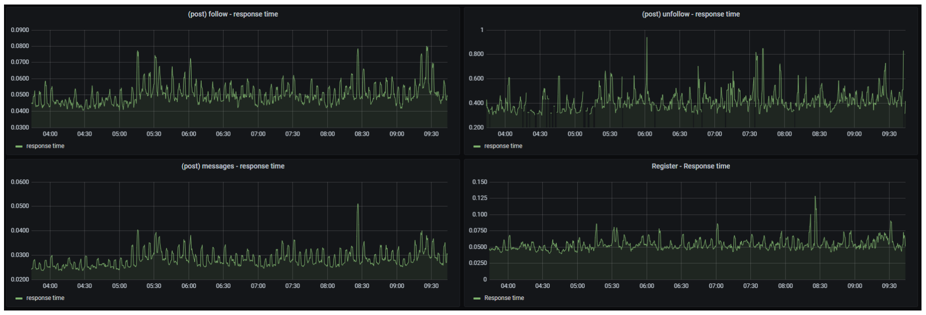


Figure 14: Average response time monitoring. Click here for full size image

The dashboard allows for a quick overview of user activity as well as average response times. This makes it easy to monitor the number of errors compared to requests, the type of errors, the endpoints which fail, and potential bottlenecks.

2.6.2 Log Collection and Aggregation

The system uses a Loki/Promtail/Grafana stack for logging. A Loki driver is installed on both the nodes in the docker swarm service with the following command:

```
docker plugin install grafana/loki-docker-driver:latest
--alias loki --grant-all-permissions
```

Furthermore, the */etc/docker/daemon.json* file on both nodes is modified, which allows for all docker containers to redirect their logs to the Loki endpoint via the Loki driver.

```
    {
    "debug" : true,
    "log-driver": "loki",
    "log-opts": {
        "loki-url": "http://161.35.214.217/loki/api/v1/push"
    }
}
```

This allows for aggregating logs from all nodes in the docker swarm stack, which the Grafana dashboard can query and visualize. Lastly, a Promtail container is deployed to access locally stored logs. Promtail is not currently utilized on the Grafana logging dashboard but can be useful for accessing specific logs.

This stack is chosen for many different reasons:

- Requires significantly less memory compared to ELK/EFK stack
- Utilizing Grafana for visualization, which is already used by Prometheus

- Easy setup and integration with docker swarm

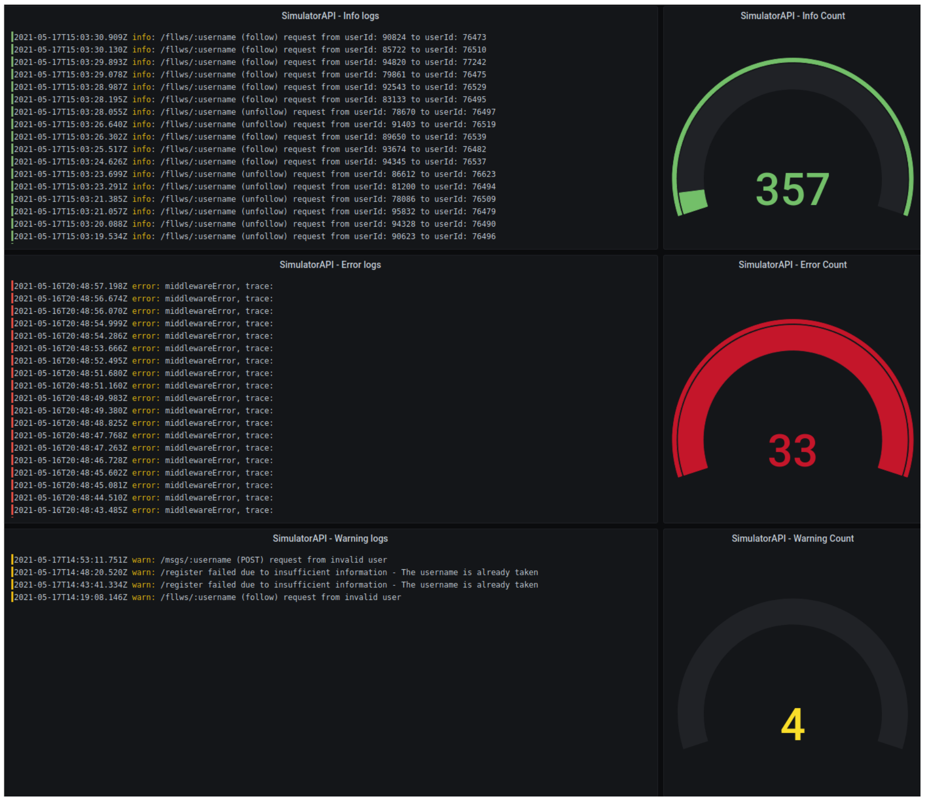
Furthermore, the collection of logs have utilized the Winston and Express-Winston library in order to provide an easier way to maintain how logs are stored and formatted.[15] As an example, the following CustomLogFormat has been created:

```
const customLogFormat = printf(({message, level, timestamp}) => {
    return `${timestamp} ${level}: ${message}`
});

const customLogger = createLogger({
    format: combine(timestamp(), customLogFormat),
    transports: [new transports.Console()],
});
```

This provides a log format to be used which shows the time-stamp of the log, the logging level, and the log message. Hence, the logs get a common syntax, and if a change is needed in the syntax it happens in a single place.

This also allows for splitting the logs into 3 different levels; information, warnings, and errors. It was chosen to log requests to each endpoint, either resulting in a successful log, a warning, or an error. For the dashboard it was chosen to visualize the information logs, error logs, and warning logs independently, to provide an easy overview of processes happening in the system. This dashboard can be seen in figure 15 below.



```
2021-05-17T15:03:30.909Z info: /fllws/:username (follow) request from userId: 90824 to userId: 76473
2021-05-17T15:03:30.130Z info: /fllws/:username (follow) request from userId: 85722 to userId: 76510
2021-05-17T15:03:29.893Z info: /fllws/:username (follow) request from userId: 94820 to userId: 77242
2021-05-17T15:03:29.078Z info: /fllws/:username (follow) request from userId: 79861 to userId: 76475
2021-05-17T15:03:28.987Z info: /fllws/:username (follow) request from userId: 92543 to userId: 76529
2021-05-17T15:03:28.195Z info: /fllws/:username (follow) request from userId: 83133 to userId: 76495
2021-05-17T15:03:28.055Z info: /fllws/:username (unfollow) request from userId: 78670 to userId: 76497
2021-05-17T15:03:26.640Z info: /fllws/:username (unfollow) request from userId: 91403 to userId: 76519
2021-05-17T15:03:26.302Z info: /fllws/:username (follow) request from userId: 89650 to userId: 76539
2021-05-17T15:03:25.517Z info: /fllws/:username (follow) request from userId: 93674 to userId: 76482
2021-05-17T15:03:24.626Z info: /fllws/:username (follow) request from userId: 94345 to userId: 76537
2021-05-17T15:03:23.699Z info: /fllws/:username (unfollow) request from userId: 86612 to userId: 76623
2021-05-17T15:03:23.291Z info: /fllws/:username (unfollow) request from userId: 81200 to userId: 76494
2021-05-17T15:03:21.385Z info: /fllws/:username (unfollow) request from userId: 78086 to userId: 76509
2021-05-17T15:03:21.057Z info: /fllws/:username (unfollow) request from userId: 95832 to userId: 76479
2021-05-17T15:03:20.088Z info: /fllws/:username (unfollow) request from userId: 94328 to userId: 76490
2021-05-17T15:03:19.534Z info: /fllws/:username (unfollow) request from userId: 90623 to userId: 76496
```

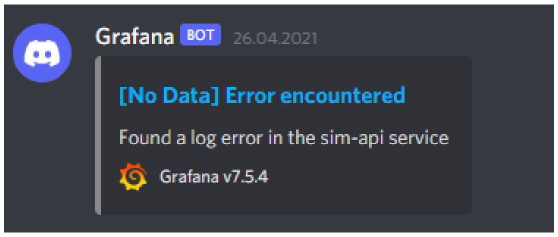
```
2021-05-16T20:48:57.198Z error: middlewareError, trace:
2021-05-16T20:48:56.674Z error: middlewareError, trace:
2021-05-16T20:48:56.070Z error: middlewareError, trace:
2021-05-16T20:48:54.999Z error: middlewareError, trace:
2021-05-16T20:48:54.286Z error: middlewareError, trace:
2021-05-16T20:48:53.666Z error: middlewareError, trace:
2021-05-16T20:48:52.495Z error: middlewareError, trace:
2021-05-16T20:48:51.680Z error: middlewareError, trace:
2021-05-16T20:48:51.160Z error: middlewareError, trace:
2021-05-16T20:48:49.983Z error: middlewareError, trace:
2021-05-16T20:48:49.380Z error: middlewareError, trace:
2021-05-16T20:48:48.825Z error: middlewareError, trace:
2021-05-16T20:48:47.768Z error: middlewareError, trace:
2021-05-16T20:48:47.263Z error: middlewareError, trace:
2021-05-16T20:48:46.728Z error: middlewareError, trace:
2021-05-16T20:48:45.602Z error: middlewareError, trace:
2021-05-16T20:48:45.081Z error: middlewareError, trace:
2021-05-16T20:48:44.510Z error: middlewareError, trace:
2021-05-16T20:48:43.485Z error: middlewareError, trace:
```

```
2021-05-17T14:53:11.751Z warn: /msgs/:username (POST) request from invalid user
2021-05-17T14:48:20.520Z warn: /register failed due to insufficient information - The username is already taken
2021-05-17T14:43:41.334Z warn: /register failed due to insufficient information - The username is already taken
2021-05-17T14:19:08.146Z warn: /fllws/:username (follow) request from invalid user
```

Figure 15: Logging dashboard from Grafana. Click here for full size image

2.6.3 Alerts

To allow for a quick response to possible errors, an alerting rule has been implemented in Grafana. This is integrated with the team's Discord channel, meaning when an error is identified, a message is sent to the discord server. Figure 16 below shows an example of an alert sent by Grafana to the team's discord server:


Grafana BOT 26.04.2021

[No Data] Error encountered

Found a log error in the sim-api service

Grafana v7.5.4

Figure 16: Grafana alert in discord

2.7 Brief Results of the Security Assessment

Analyzing our system in regards to the OWASP top 10 web application security risks, we find that we break point 10: Insufficient logging and monitoring. This is also discussed in section 3.1. Furthermore, the security of the system could be improved by implementing backups. At the current state, if the database is hacked all data would be gone. Lastly, some secrets have not been properly protected. In the GitHub history, some old secrets can be found and possibly exploited. These secrets were also at times communicated through unsafe communication channels such as Discord.

A basic pentesting of the system has been performed. NMAP was used to investigate open ports but showed no exploitable weaknesses. Port 22 was running OpenSSH 8.2p1, which could allow for a brute force password attack. However, multiple steps were taken to prevent this, meaning it was no threat. These included limiting SSH access to the use of RSA keys and using a timeout after 6 failed attempts. Both DIRB and DIRBUSTER were run to check for directories or files, but none were found. Lastly, the site was checked for cross-site scripting and SQL injection attacks, but no vulnerabilities were found.

2.8 Applied Strategy for Scaling and Load Balancing

As already mentioned, the system is hosted in Docker swarm mode with two nodes running on their own droplets in the network. The docker swarm scales horizontally, as more worker nodes running on their own physical machine can join the swarm if the swarm was to be scaled.

The default swarm setup implements load balancing out of the box. The load balancer decides which of the running instances, of the requested service, the request should end up at[1]. This is seen in figure 1 where an incoming request to the load balancer on the *minitwit-webserver* droplet can be redirected to a container running within the *Minitwit* component of the *production-worker* droplet.

2.8.1 Updating the Swarm

The swarm is configured to use rolling updates with *–update-order* set to *start-first.* For each service, an updated version is instantiated and monitored for 5 seconds before it is decided to be running successfully. Once running, the old instance of the service is shut down. This rolling update strategy is run for each service for each replica. See appendix 4.1 for the full docker-compose.yml.
The start-first approach was chosen to guarantee that at least the desired amount of replicas are always run during updates.

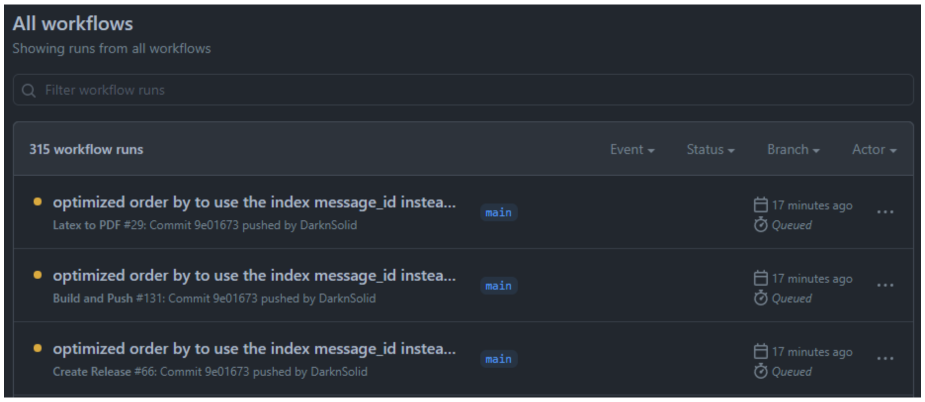
3 Lessons Learned Perspective

3.1 Monitoring

Monitoring proved very useful, and especially monitoring the response time of various endpoints allowed us to determine bottlenecks. This showed us one endpoint being significantly slower than the rest and allowed us to optimize the query, thereby decreasing response time from 25 seconds to 150ms, a 100x factor decrease.
Due to the lack of infrastructure monitoring, identifying the state of our system was more complicated than necessary. Setting up infrastructure monitoring would have allowed us to get a quick and easy overview of the various containers' status. Instead, we were required to SSH into the different droplets to obtain this information. Furthermore, aggregating system information, such as CPU, RAM, and disk usage, from the droplets and database would have been beneficial. This would allow us to monitor the status on the server itself, thereby resulting in a single access point for all monitoring and logging. This could also be integrated with the previously mentioned alerts 2.6.3, which would allow us to react to potential threats, e.g. too high RAM or CPU usage.

3.2 Reliant on GitHub Actions

Due to our CI/CD pipeline being fully dependent on GitHub actions, we experienced a significant delay when this service was down. An example of this is shown below in figure 17, where the workflow had been started and 17 minutes later still hasn't finished due to it being queued by GitHub.



315 workflow runs

optimized order by to use the index message_id instea... main 17 minutes ago Queued
Latex to PDF #29: Commit 9e01673 pushed by DarknSolid

optimized order by to use the index message_id instea... main 17 minutes ago Queued
Build and Push #131: Commit 9e01673 pushed by DarknSolid

optimized order by to use the index message_id instea... main 17 minutes ago Queued
Create Release #66: Commit 9e01673 pushed by DarknSolid

Figure 17: Github action workflow queues

This could potentially be avoided by implementing a secondary backup pipeline with e.g. Travis or Vagrant or by having our own VM to run the CI/CD pipeline.

3.3 Testing Environment

Throughout the course, the importance of having a testing environment was learned. When working with features not directly related to the code itself, such as GitHub actions, Docker, logging, etc. this was especially relevant. Due to these features being dependent on a fully deployed system, they were not testable in a local environment. Furthermore, as these techniques were new to us, getting them to work (especially the CI/CD pipeline) often required a lot of testing which could result in downtime. Thereby, the testing environment would suffer from downtime rather than the production environment.
An example of this was the implementation of logging. This required modifying various docker settings and installing plugins. Being able to test this in a development environment with no consequences made the process much easier. An implementation of such an environment was attempted but did not work optimally due to various reasons. The most important was due to the fact that the simulator only made requests to the production environment. As testing many of our features required some activity on the server itself, it was difficult to see the actual impact of the changes made. This could have been resolved by running a similar simulator targeting the testing environment.

3.4 Static Code Analysis

We added ESLint late in the project after having refactored MiniTwit to the JavaScript framework. This revealed a large number of linting errors, which could have been avoided if linting was implemented from the start. Thereby the project could have started with fewer code smells and linting errors and the accumulation of technical debt throughout the first weeks could have been avoided.

3.5 License

We learned the importance of keeping track of the licenses used throughout a project. More specifically, we learned that to avoid a project being forced into being open-source you have to look out for libraries using copyleft licenses such as the GPL.

4 Appendix

4.1 docker-compose.yml

```
version: '3.9'

services:
  sim-api:
    container_name: sim-api
    image: index.docker.io/minitwit/simulator
    ports:
      - "5001:5001"
    networks:
      - local
    deploy:
      replicas: 4
      placement:
        max_replicas_per_node: 10
      update_config:
        parallelism: 1
        delay: 20s
        order: start-first

  api:
    container_name: api
    image: index.docker.io/minitwit/api
    ports:
      - "5000:5000"
    depends_on:
      - prometheus
      - grafana
    networks:
      - local
    deploy:
      placement:
        max_replicas_per_node: 10
      replicas: 4
      update_config:
        parallelism: 1
        delay: 20s
        order: start-first

  frontend:
    container_name: frontend
    image: index.docker.io/minitwit/frontend
    ports:
        - "3000:80"
      depends_on:
        - api
        - grafana
        - prometheus
      networks:
        - local
      deploy:
        placement:
          max_replicas_per_node: 10
        replicas: 4
        update_config:
          parallelism: 1
          delay: 20s
          order: start-first

  prometheus:
    image: prom/prometheus:latest
    container_name: prometheus
    volumes:
      - ./prometheus.yml:/etc/prometheus/
   prometheus.yml
    ports:
      - "9090:9090"
    networks:
      - local


  grafana:
    image: grafana/grafana:latest
    container_name: grafana
    volumes:
      - grafana-storage:/var/lib/grafana
    environment:
      - GF_SECURITY_ADMIN_USER=devops21
      - GF_SECURITY_ADMIN_PASSWORD=-lEtMEIn-
    ports:
      - "3001:3000"
    user: "104"
    networks:
      - local
      - loki

  loki:
    image: grafana/loki:2.0.0
    ports:
      - "3100:3100"
    command: -config.file=/etc/loki/local-config.
  yaml
    networks:
      - loki
      - local

  promtail:
    image: grafana/promtail:2.0.0
    command: -config.file=/etc/promtail/config.yml
    networks:
      - loki

networks:
  local:
    driver: overlay
    attachable: true
  loki:

volumes:
  grafana-storage:
```

docker-compose.yml

4.2 Prometheus.yml

```yaml
global:
  scrape_interval:     15s # By default, scrape targets every 15 seconds.
  evaluation_interval: 15s # Evaluate rules every 15 seconds.

  # Attach these extra labels to all timeseries collected by this Prometheus instance.
  external_labels:
    monitor: 'codelab-monitor'

rule_files:
  #- 'prometheus.rules.yml'

scrape_configs:
  - job_name: 'prometheus'

    # Override the global default and scrape targets from this job every 5 seconds.
    scrape_interval: 5s

    static_configs:
      - targets: ['prometheus:9090']

  - job_name: 'sim-api'
    dns_sd_configs:
      - names:
          - 'tasks.minitwit_swarm_sim-api'
        type: 'A'
        port: 5001
    metrics_path: '/metrics'

  # - job_name:       'minittwit-app'

  #   # Override the global default and scrape targets from this job every 5 seconds.
  #   scrape_interval: 5s

  #   static_configs:
  #     - targets: ['sim-api:5001']
  #       labels:
  #         group: 'production'
```

prometheus.yml

4.3 front-end dependencies

- "axios": "ˆ0.21.1",
- "js-cookie": "ˆ2.2.1",
- "react": "ˆ17.0.1",
- "react-dom": "ˆ17.0.1",
- "react-router-dom": "ˆ5.2.0",
- "react-scripts": "4.0.2",
- "web-vitals": "ˆ1.1.0"

4.4 back-end dependencies

- "bcrypt": "ˆ5.0.0",
- "cookie-parser": "ˆ1.4.5",
- "cors": "ˆ2.8.5",
- "cross-env": "ˆ7.0.3",
- "dotenv": "ˆ8.2.0",
- "express": "ˆ4.17.1",
- "express-winston": "ˆ4.1.0",
- "jsonwebtoken": "ˆ8.5.1",
- "mocha": "ˆ8.3.2",
- "nodemon": "ˆ2.0.7",
- "pg": "ˆ8.5.1",
- "prom-client": "ˆ13.1.0",
- "sanitize-html": "ˆ2.3.3",
- "sequelize": "ˆ6.5.0",
- "sqlite3": "ˆ5.0.2",
- "winston": "ˆ3.3.3"

References

[1] *Docker ingress network.* URL: `https://docs.docker.com/engine/swarm/ingress/`. (accessed: 07.05.2021).

[2] *Docker swarm overview.* URL: `https://docs.docker.com/engine/swarm/`. (accessed: 07.05.2021).

[3] *ESLint.* URL: `https://github.com/stefanoeb/eslint-action`. (accessed: 10.05.2021).

[4] *Feature branching your way to greatness.* URL: `https://www.atlassian.com/agile/software-development/branching`. (accessed: 07.05.2021).

[5] *LaTeX github action.* URL: `https://github.com/marketplace/actions/github-action-for-latex`. (accessed: 17.05.2021).

[6] *Live Share Extension Pack.* URL: `https://marketplace.visualstudio.com/items?itemName=MS-vsliveshare.vsliveshare-pack`. (accessed: 07.05.2021).

[7] *scancode documentation.* URL: `https://scancode-toolkit.readthedocs.io/en/latest/getting-started/home.html#what-does-scancode-toolkit-do`. (accessed: 13.05.2021).

[8] *semantic-release.* URL: `https://github.com/semantic-release/semantic-release`. (accessed: 12.05.2021).

[9] *Sequelize documentation.* URL: `https://sequelize.org/master/`. (accessed: 14.05.2021).

[10] *Sonar Cloud.* URL: `https://sonarcloud.io/`. (accessed: 10.05.2021).

[11] *SSH Actino.* URL: `https://github.com/appleboy/ssh-action`. (accessed: 10.05.2021).

[12] *Top 10 GPL License Questions Answered.* URL: `https://www.whitesourcesoftware.com/resources/blog/top-10-gpl-license-questions-answered/`. (accessed: 13.05.2021).

[13] *Travis CI.* URL: `https://docs.travis-ci.com/user/tutorial/`. (accessed: 10.05.2021).

[14] *What is a Kanban Board?* URL: `https://www.atlassian.com/agile/kanban/boards`. (accessed: 07.05.2021).

[15] *Winston.* URL: `https://github.com/winstonjs/winston#adding-custom-transports`. (accessed: 13.05.2021).