Devops

Martin Meldgaard Holst, Hampuz Michael W. Iversen,
Nielz-Frederik Brinck, Emil Knudzen 69'er Niels' mor, Marcus Norton

May 17, 2021

Contents

1 System's Perspective **3**
- 1.1 Architecture and Design . . . 3
  - 1.1.1 Overview . . . 3
  - 1.1.2 Design choices . . . 5
- 1.2 Dependencies . . . 7
  - 1.2.1 Front-end and back-end low level diagrams . . . 7
  - 1.2.2 The big picture . . . 9
- 1.3 Describe the current state of your systems, for example using results of static analysis and quality assessment systems. . . . 9
- 1.4 Describe briefly, if the license that you have chosen for your project is actually compatible with the licenses of all your direct dependencies. . . . 9
- 1.5 Service Level Agreement . . . 10

2 Process' perspective **11**
- 2.1 Team organization and interaction . . . 11
- 2.2 A complete description of stages and tools included in the CI/CD chains. . . . 11
  - 2.2.1 Pull request . . . 11
  - 2.2.2 Push . . . 11
  - 2.2.3 Automating latex to pdf . . . 12
- 2.3 Organization of your repositor(ies) . . . 12
- 2.4 Applied branching strategy . . . 12
- 2.5 Applied development process and tools supporting it . . . 13
- 2.6 Monitoring and logging . . . 13
  - 2.6.1 Metrics collection . . . 13
  - 2.6.2 Log collection and aggregation . . . 15
  - 2.6.3 Acting . . . 16
- 2.7 Brief results of the security assessment . . . 16
- 2.8 Applied strategy for scaling and load balancing . . . 16
  - 2.8.1 Updating the swarm . . . 16

3 Lessons Learned Perspective **17**
- 3.1 evolution and refactoring . . . 17
- 3.2 operation . . . 17
- 3.3 maintenance . . . 17

4 Appendix **18**
- 4.1 docker-compose.yml . . . 18
- 4.2 Prometheus.yml . . . 21

1 System's Perspective

1.1 Architecture and Design

1.1.1 Overview

Below is a deployment diagram of the system:

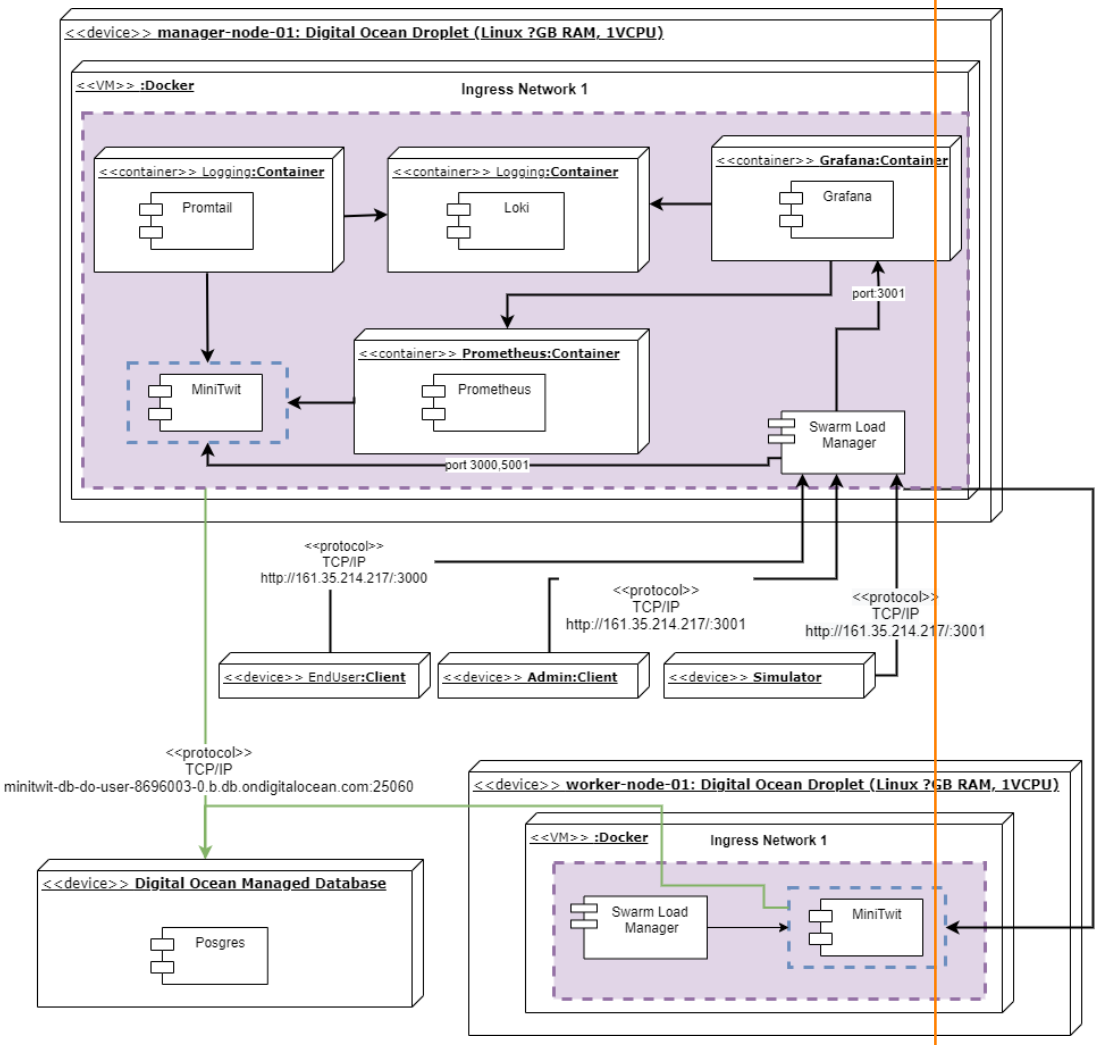

Figure 1: System Architecture

Green arrow: database connection

The MiniTwit application runs on two droplets on Digital Ocean. A managed database hosted by Digital Ocean is used to host a Postgres database. The

figuren skal opdateres med den nyeste version fra draw.io!

Vi skal lige få afklaret om vi bruger promtail eller ej!

production-worker droplet is running Docker swarm mode and is a swarm manager [2]. The *minitwit-webserver* droplet is a worker node within the swarm. The MiniTwit application itself is grouped into the Minitwit component. The swarm hosts the Minitwit component in four replicas. The swarm also hosts the monitoring and logging stack in one replica. The monitoring and logging stack consists of Promtail to perform log aggregation, Loki to store and enable log query, 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, which can be seen in appendix 4.1.
Furthermore, the swarm uses Dockerhub to pull all images for the services.
Below is a decomposition of the Minitwit component:

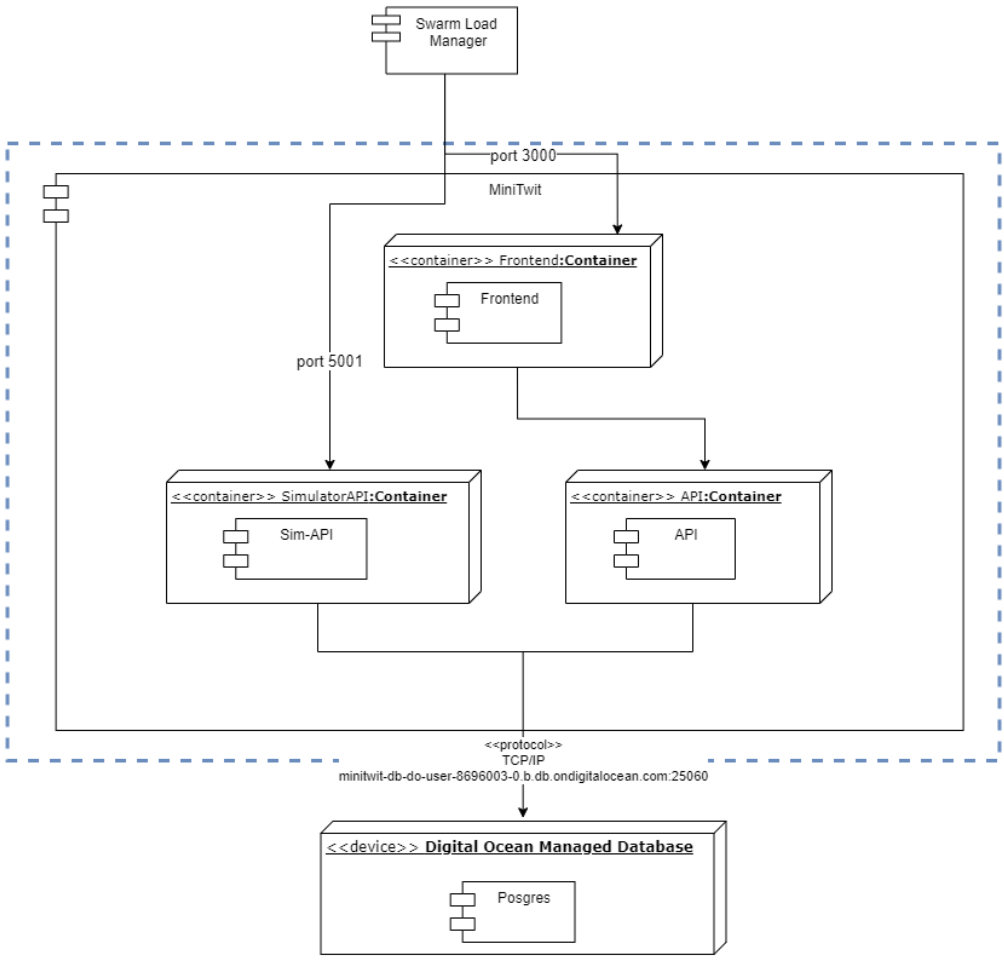

Figure 2: MiniTwit Component Decomposition

The Minitwit component is an abstraction to increase readability in the

UML drawing and is therefore not a single container hosted in the swarm. The component consists of the front-end, API for the simulator (Sim-Api) and the API for the front-end. The simulator uses the *Sim-API* service and the *Frontend* uses the *API* service.

1.1.2 Design choices

The *Frontend* service is written in React.js. Both the *Sim-API* and *API* are API's running on Node.js and exposes a RESTful API implemented with Express.js.
The rationale behind using javascript for both the front-end and back-end was that all of the team members were experienced with javascript. It is also a lightweight solution compared to a C# implementation which is much more complex in terms of sub modules and interfaces. Also, developing with javascript is quick as each change in the source code is re-compiled in run-time during development, making debugging very easy.

Below is a component diagram of the back-end illustrating the simplicity:

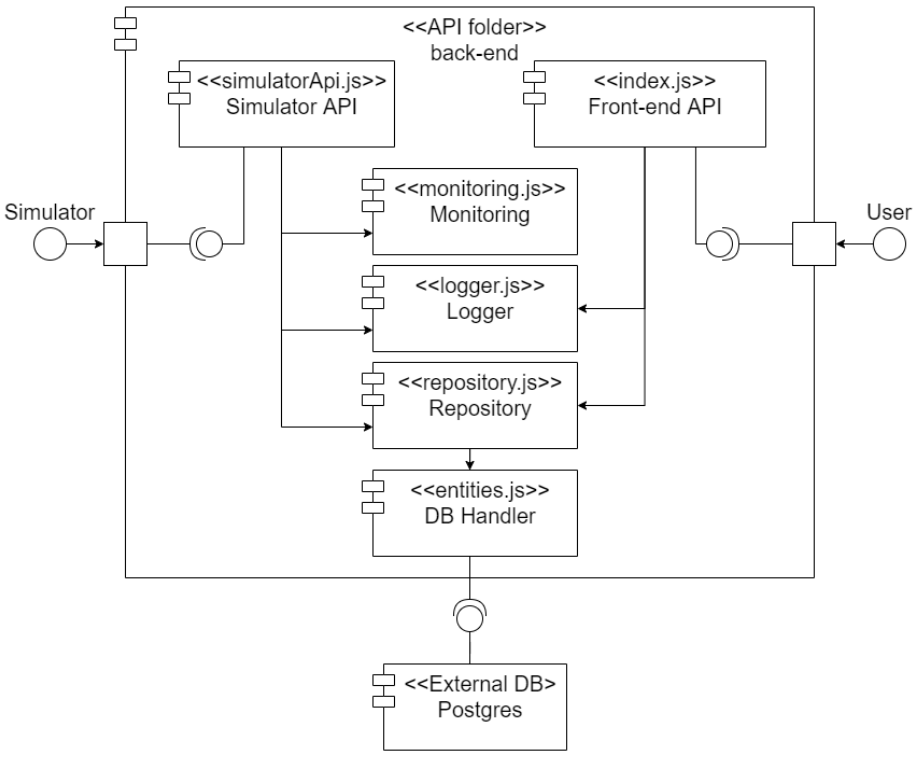

Figure 3: Back-end Component diagram

Each component in the back-end is one single file and there are no formal

interfaces. Both APIs implements the endpoints and uses the rest of the components as helpers to resolve each request.
The following list breifly describes each component:

1. Monitoring: Provides functionality to use metrics used for Prometheus
2. Logger: Provides logger objects to perform logging by using the *winston* and *express-winston* libraries.
3. Repository: Provides a Sequelize ORM object to query the database.
4. DB Handler: Provides a database object with the database connection and schema.

A Postgres database was chosen as all of the team members had previous experience using it and because it is a relational database, which fits the purpose of this project. The Sequelize library is used in the *DB Handler* as the database abstraction layer and was chosen because it is well documented [8] and has a large community on stackoverflow.

1.2 Dependencies

1.2.1 Front-end and back-end low level diagrams

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

1. Squares: files that the team developed
2. Circles: external libraries
3. Arrow: build-/compile-time dependency
4. Dashed arrow: run-time dependency

The front-end has the following dependencies:

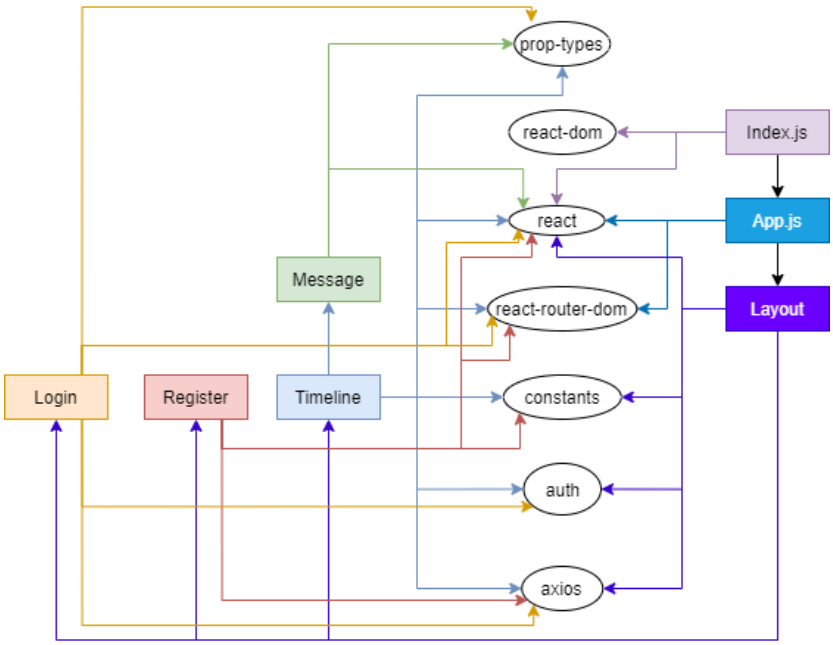

Figure 4: Front-end dependency diagram

The squares are here components. Colors have here been used to improve the readability by each component having their own color.
The source code for this diagram is located at the Minitwit/Frontend folder.

The back-end has the following dependencies:

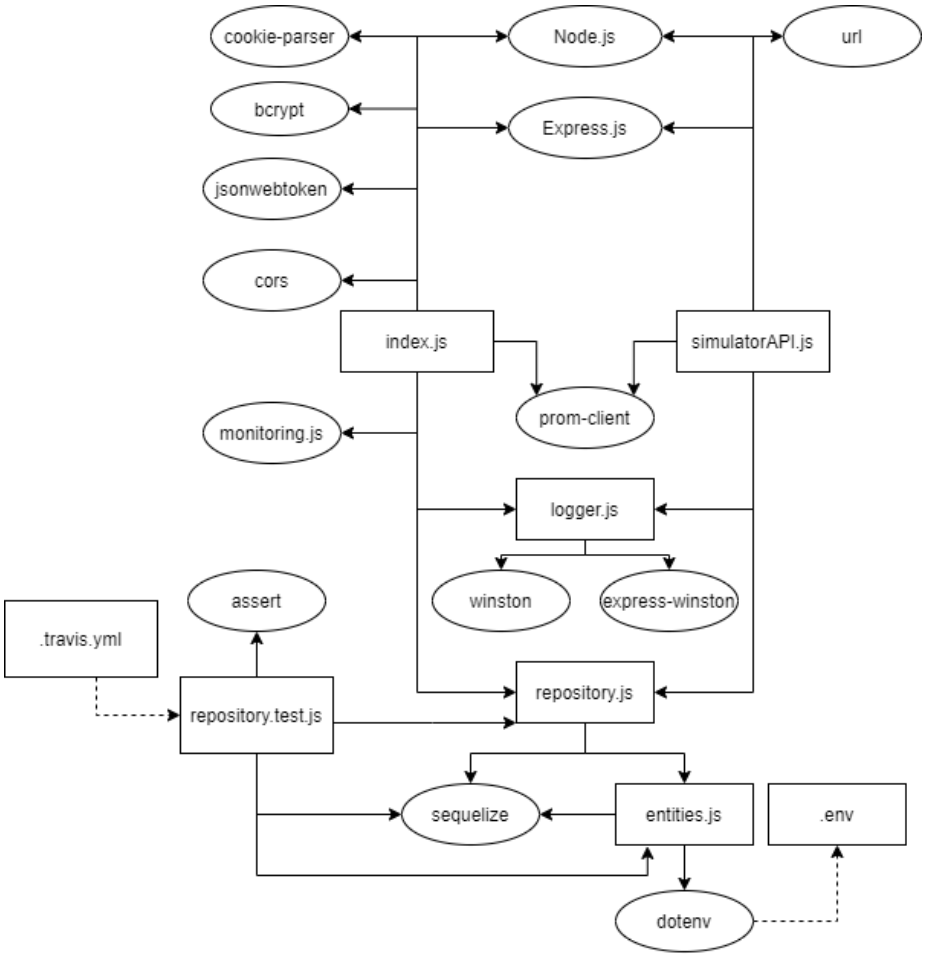

Figure 5: Back-end dependency diagram

The source code for this diagram is located at the Minitwit/API folder. 'index.js' runs the api service and 'simulatorAPI.js' runs the sim-api service. All dependencies in the front-end and back-end uses the last release of the current major release. E.g. the *bcrypt* dependency is ˆ5.0.0 and will use future releases up until before the next major release: v6.0.0.

1.2.2 The big picture

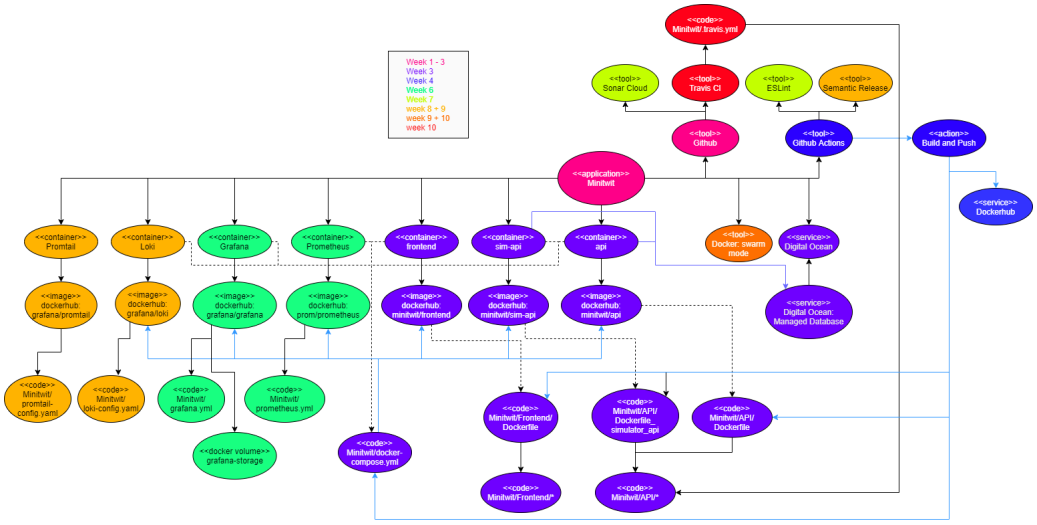

Figure 6: System dependencies. Click here for full size image

The following describes the UML syntax:

1. Full line arrow: dependency
2. Stippled line arrow: retired dependency
3. Colored line arrow: for readability purpose only

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 Describe the current state of your systems, for example using results of static analysis and quality assessment systems.

Skal laves

1.4 Describe briefly, if the license that you have chosen for your project is actually compatible with the licenses of all your direct dependencies.

The ScanCode toolkit[6] were 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 were present. This is important, as the GPL is a copyleft license which would require MiniTwit to be a free open-source license[11]. 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 in order to provide everyone with the free right to use the API, as well as keeping the developer team free from being liable of any claims made against any possible damage caused by the API.

1.5 Service Level Agreement

The SLA mentions that the API guarantees an uptime of 100%
Vores SLA "https://github.com/Niels-Frederik/MiniTwit/blob/main/API/SLA.md"
TA's omtale vdr. SLA: "https://github.com/LazyOpsDev/Minitwit/blob/develop/report/report.md"

Læs i SLA'en hvordan uptime udregnes. Der skal skrives her hvordan dataen konkret ser ud (se TAøs rapport), jeg gik i stå fordi vi bare skriver "failure" under nogen af dem, og ikke om det er client eller server fejl. Tænker Yarl har lavet det?

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[13] 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 A complete description of stages and tools included in the CI/CD chains.

Need better styling

The CI/CD pipeline uses Github actions, Travis CI, ESLint, Sonar Cloud, and Semantic Release. Github actions is 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 [12]. Sonar Cloud is also used to perform static code analysis and checks code smells such as if code duplication exceeds 4%. It also scans for potential security vulnerabilities [9].

2.2.1 Pull request

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

2.2.2 Push

On a push to the main or Develop branch, a *Build and Push* Github action is run. The jobs builds the Frontend, API and Sim-Api Docker images and pushes them to Dockerhub. Once all three jobs have finished, a fourth job is started which ssh's into the swarm manager droplet on Digital Ocean and updates the corresponding running services in the swarm using the *docker service update –image name-of-image* command. The ssh action is *appleboy/ssh-action@master*[10]. Below is an image taken from the Github action log showing the four jobs:

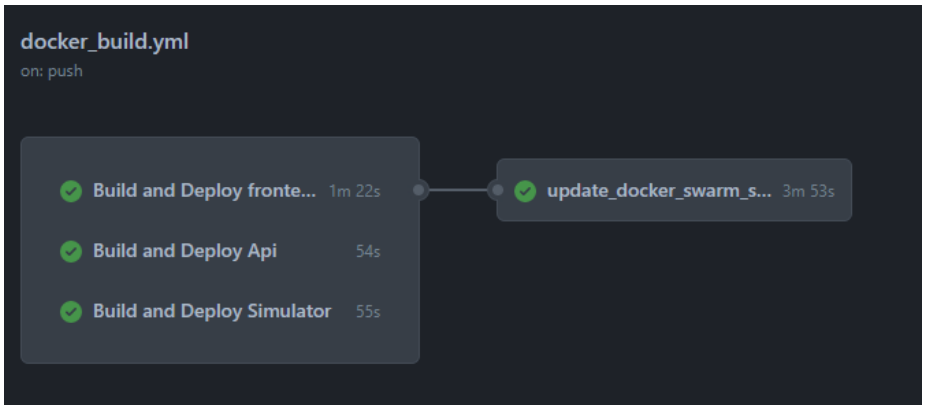

Figure 7: Build and deploy action successfully run

Furthermore, a Github action with the name *Create Release* running *Semantic Release* will act on any push to the master 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 [7].

2.2.3 Automating latex to pdf

vi mangler GitHub action der konverterer Latex-rapporten til PDF

2.3 Organization of your repositor(ies)

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[5]. While screen sharing functionality is self-explanatory, the Live Share functionality provided a way for multiple developers to work on 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:

```
116   networks:
117     local:
118       driver: overlay
119       attachable: true
```

Figure 8: 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 resolve 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:

```
21    - job_name: 'sim-api'
22      dns_sd_configs:
23        - names:
24            - 'tasks.minitwit_swarm_sim-api'
25          type: 'A'
26          port: 5001
27      metrics_path: '/metrics'
```

Figure 9: 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 resolve their ip and then uses the specified *metrics_path* endpoint to pull the metrics.
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 where Prometheus pulls the metrics from the services and Grafana pulls the metrics from Prometheus.

kan vi nå at fikse dette?

As seen in figure 5, both the api and sim-api uses the prom-client library to passively monitor endpoint requests and responses by "sniffing". Prom-client is used for the whitebox monitoring as part of both of the api's middleware to monitor the total requests and responses and the individual endpoints and response codes.

As seen in the figure 10, the Grafana dashboard display 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.

Figure 10: Request/response monitoring. Click here for full size image

No infrastructure monitoring
The dashboard allows for a quick overview of user activity,

Se om vi kan lave application monitoring og/eller infrastructure monitoring: response time kunne være fedt

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 us to aggregate logs from all nodes in the docker swarm stack, which is sent to the Grafana dashboard for querying and visualization. Lastly, a Promtail container is deployed to load locally stored logs into Grafana.

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 library in order to provide an easier way to maintain how logs are stored and formatted.[14] As an example, the following so-called 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 so-called "log level", and the log message. Hence, the logs get a common syntax, and if a change is needed in the syntax it happens at a single place.

For the dashboard we chose to visualize
Currently no log rotation is implemented

Færdiggør nedenstående (hvem end der startede det)

2.6.3 Acting

2.7 Brief results of the security assessment

Running NMAP showed the following open ports and services:
None of these have any exploitable weaknesses.

Færdiggør NMAP (hvem end der startede det)

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 were to be scaled.

With swarm comes load balancing out of the box. The ingress network exposes the services on the swarm via the routing mesh. 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 2 where an incoming request to the load balancer on the *production-worker* droplet can be redirected to a container running within the *Minitwit* component of the *minitwit-webserver* 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 evolution and refactoring

3.2 operation

3.3 maintenance

4 Appendix

4.1 docker-compose.yml

```yaml
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:
```

```
43        - "3000:80"
    depends_on:
45        - api
          - grafana
47        - prometheus
    networks:
49        - local
    deploy:
51        placement:
            max_replicas_per_node: 10
53        replicas: 4
          update_config:
55          parallelism: 1
            delay: 20s
57          order: start-first

59  prometheus:
    image: prom/prometheus:latest
61  container_name: prometheus
    volumes:
63        - ./prometheus.yml:/etc/prometheus/
 prometheus.yml
    ports:
65        - "9090:9090"
    networks:
67        - local


69

    grafana:
71  image: grafana/grafana:latest
    container_name: grafana
73  volumes:
          - grafana-storage:/var/lib/grafana
75  environment:
          - GF_SECURITY_ADMIN_USER=devops21
77        - GF_SECURITY_ADMIN_PASSWORD=-lEtMEIn-
    ports:
79        - "3001:3000"
    user: "104"
81  networks:
          - local
83        - loki

85  loki:
    image: grafana/loki:2.0.0
87  ports:
```

```
      - "3100:3100"
    command: -config.file=/etc/loki/local-config.yaml
    networks:
      - loki
      - 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

  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

```
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

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

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

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

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

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

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

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

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

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

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