

# DevOps, Software Evolution & Software Maintenance

Group P - Maxitwit

Andreas Andrä-Fredsted - aandr@itu.dk  
Bence Luzsinszky - bluz@itu.dk  
Christian Emil Nielsen - cemn@itu.dk  
Michel Moritz Thies - mithi@itu.dk  
Róbert Sluka - rslu@itu.dk

2024

# Contents

<b>1</b>	<b>System Perspective</b>	<b>3</b>
1.1	Architecture . . . . .	3
1.2	Dependencies . . . . .	3
1.3	Viewpoints . . . . .	3
1.3.1	Module Viewpoint . . . . .	3
1.3.2	Components Viewpoint . . . . .	3
1.3.3	Deployment Viewpoint . . . . .	4
1.4	Important interactions . . . . .	4
1.5	Current State . . . . .	4
<b>2</b>	<b>Process Perspective</b>	<b>5</b>
2.1	Branching strategy . . . . .	5
2.2	Commit hooks . . . . .	5
2.3	CI/CD pipeline . . . . .	6
2.4	Monitoring . . . . .	7
2.5	Security Assesment . . . . .	7
2.6	Scaling strategy . . . . .	7
<b>3</b>	<b>Lessons Learned</b>	<b>7</b>
3.1	Evolution and refactoring . . . . .	7
3.1.1	Implementation of Logging . . . . .	7
3.2	Operation . . . . .	8
3.3	Maintenance . . . . .	8
3.3.1	Issues with monitoring . . . . .	8
3.3.2	Maintaining a performant DB . . . . .	8

# 1 System Perspective

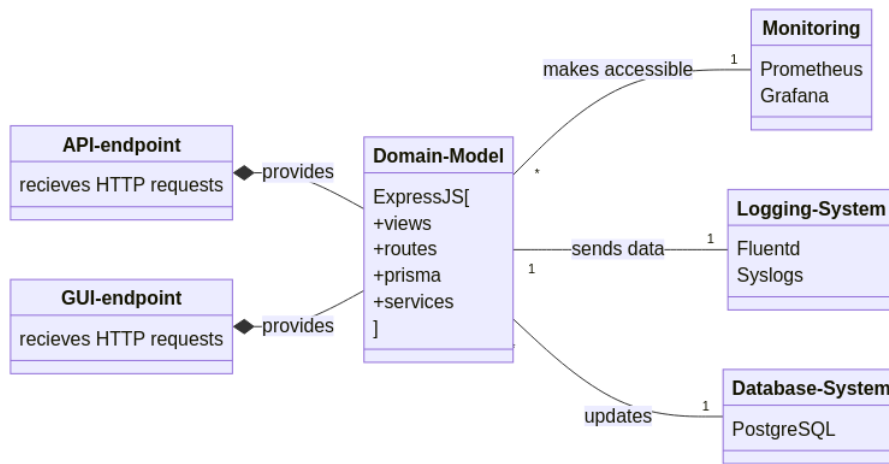
## 1.1 Architecture

## 1.2 Dependencies

## 1.3 Viewpoints

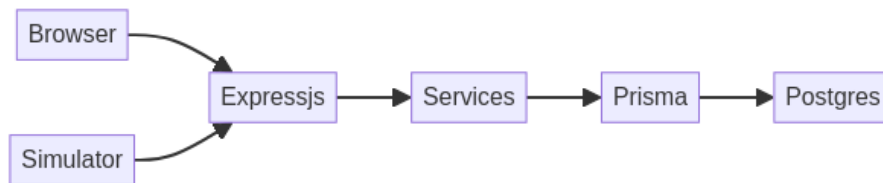
### 1.3.1 Module Viewpoint

To effectively capture this, the following class diagram presents the components of the web-app mapped to their respective dependencies.

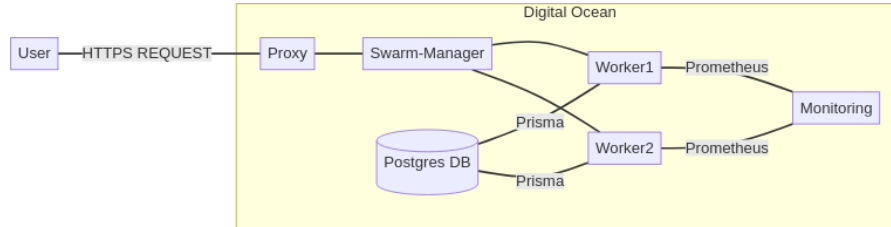


The above module viewpoint highlights how the expressjs application interacts with numerous systems with some being dependencies required for the running of the application, such as the postgres database, while others are tools meant for tasks such as monitoring and logging. What is not covered in this illustration is the framework in which the application is run and managed, which is covered in the following viewpoints.

### 1.3.2 Components Viewpoint



### 1.3.3 Deployment Viewpoint

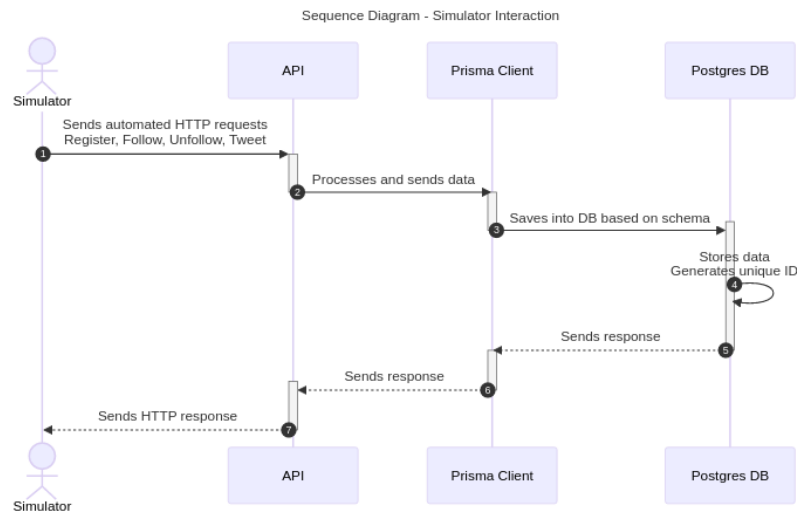


### 1.4 Important interactions

The system can be interacted with in two ways:

- User Interface
- API for the simulator

A user (or the simulator) can register, follow/unfollow other users and send tweets.



### 1.5 Current State

Security	Reliability	Maintainability
<b>4</b> Open issues <span style="color: red;">E</span>	<b>0</b> Open issues <span style="color: green;">A</span>	<b>9</b> Open issues <span style="color: green;">A</span>
<div>4 H</div> <div>0 M</div> <div>0 L</div>	<div>0 H</div> <div>0 M</div> <div>0 L</div>	<div>0 H</div> <div>1 M</div> <div>8 L</div>

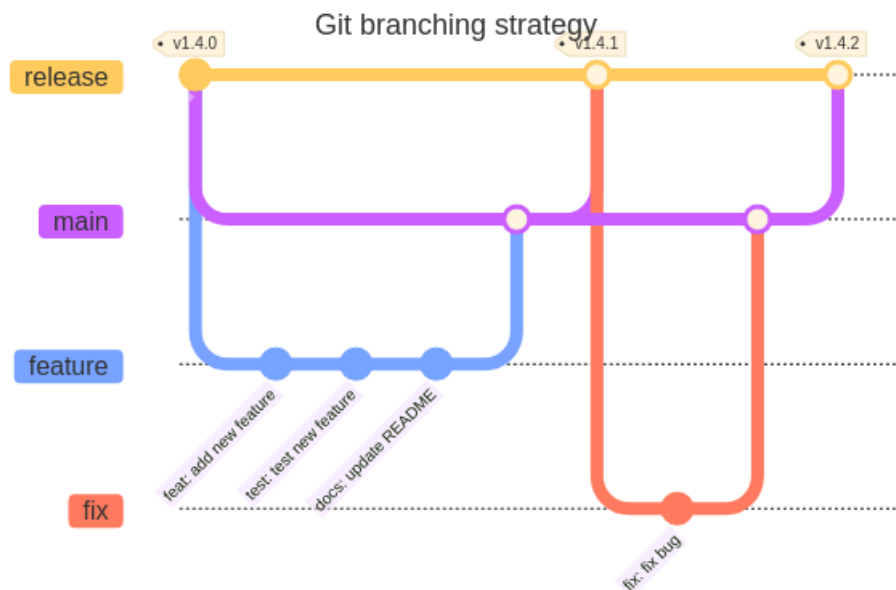
The application is practically fully functional, apart from a single outstanding

bug. While the application has [minimal technical debt](#), it relies on legacy code and dependencies to test the application (test suite and simulator).

## 2 Process Perspective

Why: ExpressJS, Prisma, Postgres

### 2.1 Branching strategy



The chosen branching strategy loosely follows the [Gitflow](#) workflow. We chose to omit hotfix branches and merge the concept of a main/develop branch for simplicity. Committing to main or release is not allowed only pull requests.

Opening a pull request from a feature branch to main triggers the CI pipeline.

Successfully merging a pull request to the release branch triggers the CD pipeline. Release tag is bumped according to the contents of the release, using the [semantic versioning](#) protocol.

### 2.2 Commit hooks

A pre-commit hook was added in [d40fcba](#) to lint and enforce commit messages and to follow the [semantic versioning](#) protocol. A [CLI-tool](#) was also [added](#) to aid developers write commit messages that follows the chosen protocol. Effectively standardizing a common development process, improving our process quality and readability of the git log.

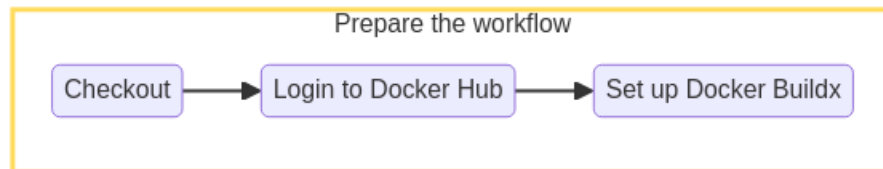
## 2.3 CI/CD pipeline

Our CI/CD pipeline is based on **Github Actions**. We have a [deploy.yml](#) file that is automatically triggered when new data is pushed to the **release branch**.

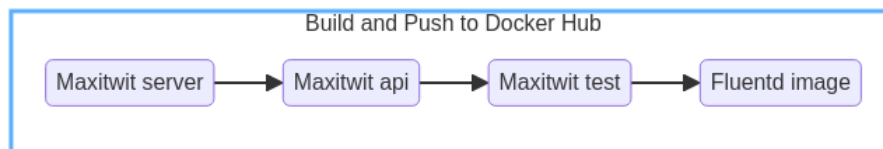
CI/CD Pipeline



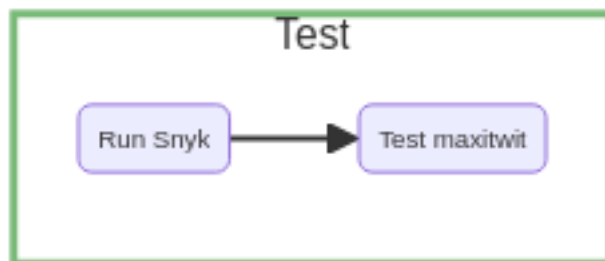
We prepare the workflow by checking out to our release branch, logging in to Docker Hub and setting up Docker Buildx so the workflow can build the images.



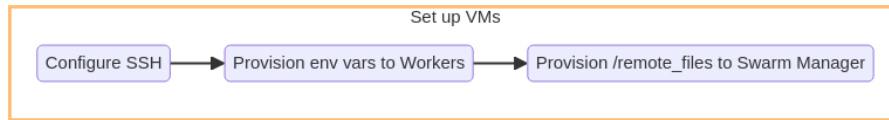
The workflow builds our images and pushes them to Docker Hub.



Snyk is run to check for vulnerabilities. After the workflow builds our images and runs our tests suite against them.



The environment variables stored in GitHub Actions Secrets are given to the workers and the most recent [/remote\\_files](#) are copied with SCP to the Swarm Manager.



Finally we SSH onto the Swarm Manager and run the [deploy.sh](#) script to pull and build the new images.

## 2.4 Monitoring

We use Prometheus and Grafana for [monitoring](#). There are multiple metrics set up in our backend, that are sent to /metrics endpoint on our both our [GUI](#) and the [API](#). Prometheus scrapes these endpoints and Grafana visualizes the data.

We set up a separate Droplet on DigitalOcean for monitoring, because we had issues with its resource consumption. The monitoring droplet runs Prometheus and Grafana, and scrapes the metrics from the Worker nodes of the Docker swarm.

## 2.5 Security Assesment

- TODO sentence about our pipelines using root users which violates [PloP](#)

According to the documentation that can be found [Restrictions to ssh](#), we are aware that setting the flag for StrictHostKeyChecking to “no”, might result in malicious parties being able to access the super user console of our system. Setting it to yes would prevent third parties from entering our system and only known hosts would be able to.

## 2.6 Scaling strategy

We used Docker Swarm for horizontal scaling. The strategy is defined in [compose.yml](#). One manager node is responsible for the load balancing and the health checks of two worker nodes. Worker nodes we have 6 replicas of the service running. We update our system with rolling upgrades. The replicas are updated 2 at a time, with 10s interval between the updates. The health of the service is monitored every 10s. If the service fails, it will be restarted with a maximum of 2 attempts.

# 3 Lessons Learned

## 3.1 Evolution and refactoring

### 3.1.1 Implementation of Logging

The evolution to the docker swarm architecture made the reconfiguration of subsystems a necessity. Specifically the reimplementaion of logs proved difficult,

as we had to sync logs across workers. To achieve this, it was attempted to have fluentd containers in each worker gather logs and send them to a separate digital ocean droplet with elasticsearch and kibana running. This however proved infeasible within the constraints of this class, as elasticsearch kept crashing due to the limited resources provided to it on the droplet. Thus, we defaulted to store logs in a /logs folder on the droplet also containing the load balancer. Overall, this proved a learning experience for how a change to the tech-stack can make other segments obsolete, thus increasing the amount of refactoring required for a change to be feasible. In this specific case, a migration to the EFK-stack before the implementation of the docker swarm would have allowed for a more seamless and robust evolution of the application.

## **3.2 Operation**

## **3.3 Maintenance**

### **3.3.1 Issues with monitoring**

Our inbuilt metrics for prometheus turned out to be [very resource demanding](#). So much that building the Prometheus container instantly started using 100% CPU and RAM of our droplet. This was solved by reducing the unnecessary metrics and moving the Monitoring to its own droplet.

### **3.3.2 Maintaining a performant DB**

We noticed the performance of the public timeline endpoint getting slower as the database grew. To remedy this, we [wrote a shell script](#) to query the performance table of our production database to [identify which relations needed indices](#).