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

Abstract

This report details the transformation of Maxitwit, initially a Python-Flask application, into a JavaScript-based platform utilizing Node.js, Express, and Pug. The transition was guided by DevOps principles, incorporating Docker Swarm and DigitalOcean for deployment and hosting. Key components include a PostgreSQL database and monitoring with Prometheus and Grafana. The refactoring emphasizes modern software practices and the project's adaptive strategies in architecture, deployment, and maintenance for enhanced performance and scalability.

1 System Perspective

1.1 Architecture

When we took over the minitwit application at the beginning of the course we started by evolving it away from Python using Flask and replacing it with Javascript using Node.js, Express and Pug. The group decided to do the rewrite in Javascript as all members were already familiar with it to varying degrees and because of the good ecosystem which offers tools for everything we need in this web application.

1.1.1 Description of Components

1.1.1.1 Frontend

The frontend of our maxitwit application consists of HTML and CSS which is being rendered using the Pug templating engine. The frontend handles user input and sends requests to the express server while also displaying all data it receives as response. The frontend is rendered on the server and there is no javascript running on the client to render the GUI.

1.1.1.2 Backend API

We decided to use Node.js as it is the most popular and mature runtime environment for building fast and scalable server side applications in Javascript.

Instead of writing the server side logic completely from scratch we decided to use the Express framework as it comes with a number of useful features for developing robust server-side applications. Using the Express framework we have a minimal yet flexible framework that provides middleware support, so middleware functions can be used to handle HTTP requests and responses, as well as Route Handling allowing us to define routes for a number of HTTP methods such as GET, POST, PUT, DELETE and the corresponding url patterns. Furthermore it offers a number of HTTP Utilities to simplify sending responses and accessing request data. Another useful feature for us is the static file serving provided by the framework which we use to serve our CSS styles. To render our HTML content dynamically Express also offer template engine support, in our case for Pug.

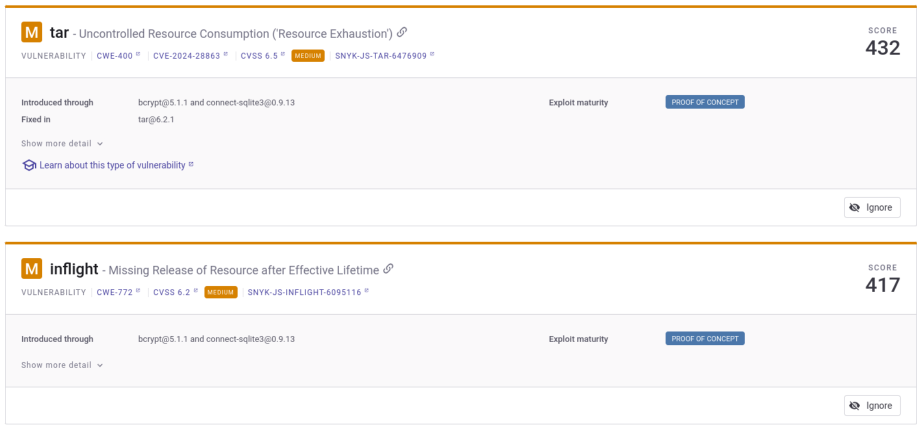
Finally the good support for Error Handling in the framework is essential when developing and maintaining complex application logic.

1.1.1.3 Database

When we first started working with the application it had an SQLite database for data storage which we used for the first weeks of the course. We then added the ORM, Prisma, which provided an abstraction layer over the database. We decided on Prisma as our ORM because of the efficient and clean communication it enabled us to have between the database and the backend API and because the schema driven approach would ensure data integrity. After adding the ORM we were tasked with migrating away from SQLite to another database for which we decided on PostgreSQL as all team members are familiar with it from the Introduction to Database Systems course, which we were all taking this semester. We also chose Postgres as it allows us to improve the perfromance of our application, especially as the database grew to a large data set and because we could easily scale PostgreSQL horizontally if we needed to.

1.2 Dependencies

We generated a dependency graph for our node dependencies.


M tar - Uncontrolled Resource Consumption ('Resource Exhaustion')

VULNERABILITY | CWE-400 | CVE-2024-28863 | CVSS 6.5 MEDIUM | SNYK-JS-TAR-6476909

SCORE 432

Introduced through bcrypt@5.1.1 and connect-sqlite3@0.9.13

Fixed in tar@6.2.1

Exploit maturity PROOF OF CONCEPT

M inflight - Missing Release of Resource after Effective Lifetime

VULNERABILITY | CWE-772 | CVSS 6.2 MEDIUM | SNYK-JS-INFLIGHT-6095116

SCORE 417

Introduced through bcrypt@5.1.1 and connect-sqlite3@0.9.13

Exploit maturity PROOF OF CONCEPT

Figure 1: Snyk screenshot

For identifying and fixing vulnerabilities, we used Snyk, which provided us with detailed reports on a weekly basis. These potential vulnerabilities were categorized based on their severity and then addressed. However, not all of them have been resolved, such as inflight, which appears to no longer be maintained, and therefore, no current fix is available.

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.

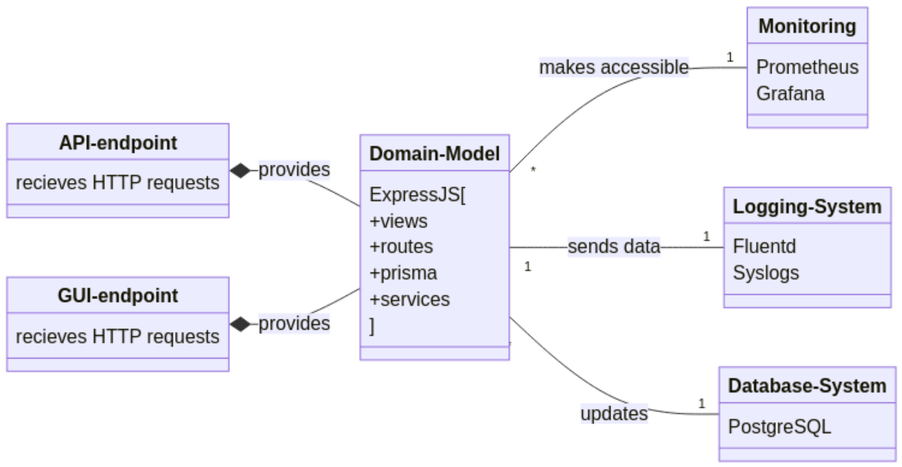


Figure 2: Module Viewpoint

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 Deployment Viewpoint

Our application is deployed on a Digital Ocean droplet. The droplet is running a Docker Swarm with one manager and two worker nodes. We use an Nginx reverse proxy to route the incoming requests and monitoring is also running in a separate droplet. In total we have 5 droplets and a database running on Digital Ocean.

We chose Digital Ocean because Github Education provides 200$ in credits for students, which was enough to cover the costs of the droplets and the database for the duration of the project.

1.4 Important interactions

The system can be interaceted with in two ways:

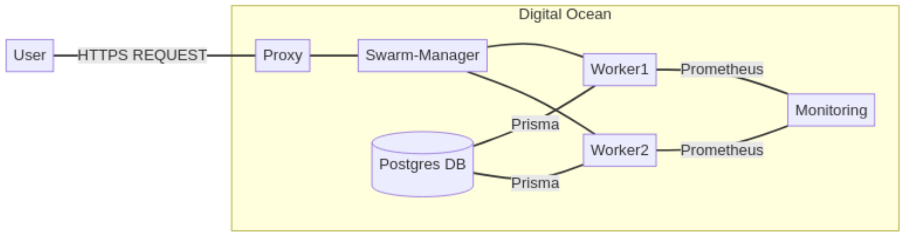


Figure 3: Deployment Viewpoint

- User Interface
- API for the simulator

The main interaction with the system is via an API, that is built for a simulator. The simulator sends HTTP requests to our endpoints to simulate a user registering, following, unfollowing and tweeting. The API uses Prisma to interact with the Postgres database. Prisma is an ORM that generates SQL queries based on the schema defined in the Prisma schema file.

We chose prisma because it is a modern ORM that is easy to use and has a lot of features that make it easy to interact with the database.

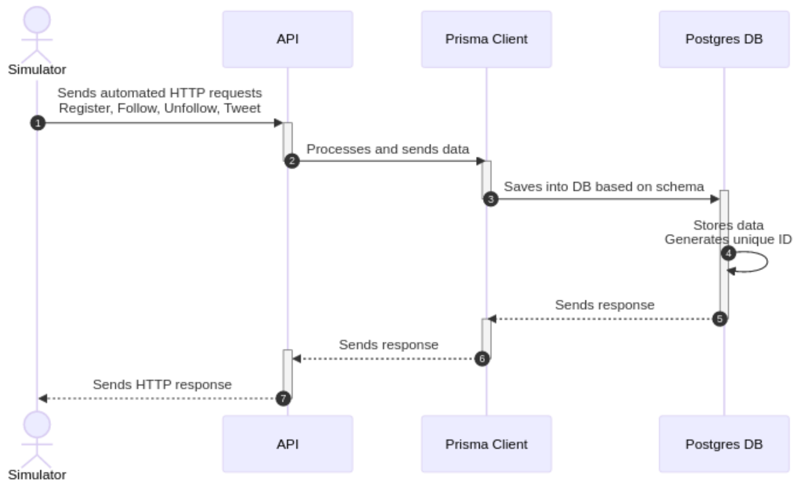
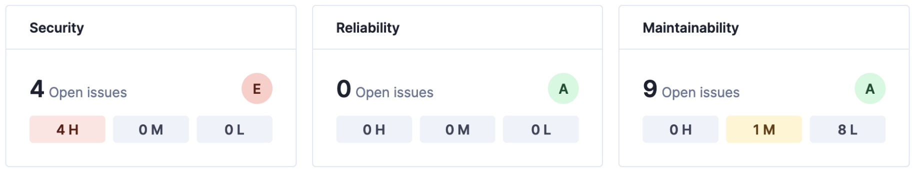


Figure 4: Simulator Interaction

1.5 Current State

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). The project


Security

4 Open issues E

4 H 0 M 0 L

Reliability

0 Open issues A

0 H 0 M 0 L

Maintainability

9 Open issues A

0 H 1 M 8 L

Figure 5: Sonarcloud screenshot

has a couple of outstanding PR's that fixes the most relevant cwe's. Overall, the quality of the code base is high, with minimal technical debt.

2 Process Perspective

2.1 Monitoring

Inside the application droplets, prometheus has a volume storing it's state, such that the cd pipeline would not reset monitoring. We use Prometheus to generate metrics for our monitoring and Grafana to visualize them. We made this decision because with this setup we can easily make relevant and informative dashboards representing the state of our system.

2.2 Database Migration

During the semester we had the task to migrate from SQLite to a database of our choice. We chose Postgres to supplement our studies in Introdutcion to Database Systems course that we are having paralelly.

2.3 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 pipline.

Succesfully 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.

2.4 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.

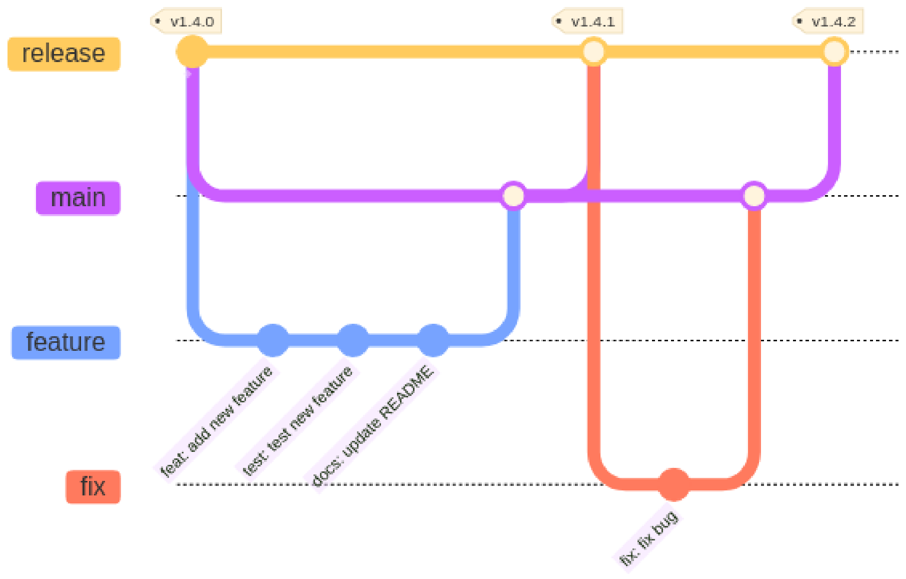


Figure 6: Git Branching Strategy

2.5 CI/CD pipline

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

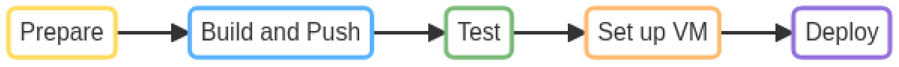
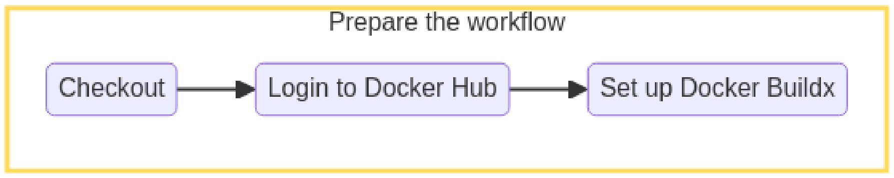
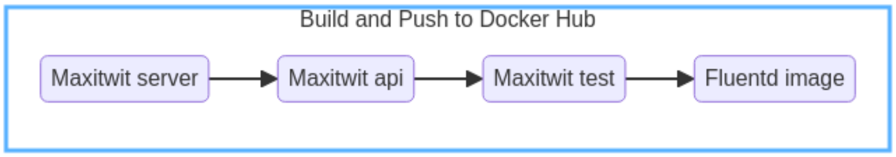


Figure 7: CI/CD Pipeline

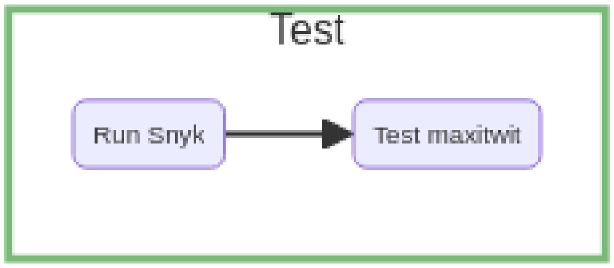
We prepare the workflow by checking out to our release branch, logging in to Docker Hub and setting up Docker Buildx so the workflow can build the images.



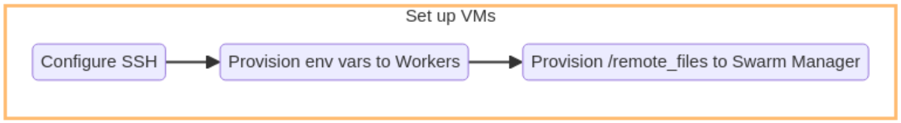
Th workflow builds our images and pushes them to Docker Hub.



Snyk is run to check for vulerabilities. 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.


Set up VMs
Configure SSH
Provision env vars to Workers
Provision /remote_files to Swarm Manager

Finally we SSH onto the Swarm Manager and run the deploy.sh script to pull and build the new images.

2.6 Monitoring

We use Prometheus and Grafana for monitoring. There are multiple metrics set up in our backend, that are sent to /metrics enpoint 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.

2.7 Logging

The Logging system started out as a simple logger using the `winston` npm package for the logging-client and the `morgan` npm package as middleware to interface with express.js. To make logging system scale in a distributed context, the logger was reconfigured to send the gathered logs to a fluentd instance listening on port 24224, which then send the logs to be stored in the same droplet containing the load balancer.

Fluentd specifically was chosen over other similar alternatives such as Logstash for it's provided flexibility and integration with other services, as the decision whether to integrate logging into elasticsearch had not been made at the time. Thus, Fluentd provided a scalable solution that could fit with multiple evolution paths.

2.8 Security Assesment

A severe vulnerability we found is that many of our containerized services executed process as root. This included images that ran in our CI/CD pipeline. This is a security risk because it violates PloP.

According to the documentation that can be found Restricitons 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.

NPM was used to manage and audit dependencies with security vulnerabilities with `npm audit`. It was a challenge to upgrade certain dependencies, either because they were bundled or because they create cyclic dependencies. We generated a dependency graph for our dependencies.

2.9 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, each node having 6 replicas. 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.

2.10 AI and LLM's

LLM's were very useful tools in the refactoring process. We found that AI tools work best when you can provide extensive context as a prompt. However, sometimes gathering all the context needed for a prompt was wasted cognitive load if they didn't provide a useful response. Especially when debugging niche interactions between system components, LLM's were not very helpful.

3 Lessons Learned

3.1 Evolution and refactoring

3.1.1 State in a Load Balanced System

A hindrance to the application running in a distributed environment, such as Docker Swarm, is the configuration of the session handling. Currently, it is done using the express-session npm package, which was set up to use a locally stored sqlite database. This means that users would get their sessions dropped/logged out if their requests got directed to a node in the swarm that did not contain the database. To fix this issue, we discussed ways to manage session-handling using our managed postgres database to handle user sessions instead. This would however require refactoring of the session-handling to use a foreign database.

3.1.2 Implementation of Logging

The implementation of the logging system proved difficult, as storing logs locally in the application did not scale in a docker swarm network. We attempted to expand on the system by adding a fluentd container as a global service in the swarm so it would run on each node in the swarm. The service would recieve the logs from the containers and then send them all to a centralized storage droplet running elasticsearch and kibana. This proved infeasable given the hardware specification of Elasticsearch, as it kept crashing due to memory issues. To provide centralized logs, we defaulted to have fluentd send logfiles to the droplets running the load balancers. Overall, choices made early on in the project made the refactoring required for a centralized logging system too expansive.

3.1.3 Database Migration

The Database Migration task proved a challenge, as many issues arose in the process, such as namespaces and types not being compatible with Postgresql. Specifically, the TIMESTAMP type in sqlite proved difficult, as postgres stores timestamps as integers.

Firstly, we tried to modify the sql dump using different regex, `sed` commands and bash scripts, and then using an ssh connection to run the script against the postgresql droplet. This proved fatal however, as the script had not finished running after five hours due to each insert statement requiring a new connection. Furthermore, the process failed due to conflicting id's between the insert statements and postgres.

Finally, a solution was found in the shape of a pythonscript, which represented insert statements as classes, where each attribute in the insert statement was modified in the constructor of the class to match the postgresql schema. The script then aggregated insert statements, allowing us to run 1000 insert statements per connection. Thus, the migration was completed in five minutes.

This experience showed us that even with abstraction layers, such as prisma, unique issues related to our migration occured which necessitated the development of a specific solution.

3.2 Operation

During the last week of the simulator being active, our application crashed which we ended up not noticing. The reason for the crash, which became clear when inspecting the docker logs, was that a misconfiguration in Fluentd stopped the API- and GUI- containers from running, thereby bringing the entire application to a standstill. The issue seemed to be that Fluentd was not configured to deal with some of the formats generated by the logging-client certain logs. Furthermore, it was trivial to solve when we became aware of it, as it only required a slight modification in how logs were matched and transported out of fluentd. Our monitoring system failed to inform us of this crash, which was caused by Prometheus having crashed around the same time. Thus, a set of systems set up to monitor and log the system had failed with no relation to eachother, allowing for the issue to go unnoticed.

3.3 Maintenance

3.3.1 Issues with monitoring

Our inbuilt metrics for prometheus turned out to be very resource demanding. Starting the Prometheus container would instantly max out system resource usage in the droplet. This was solved by reducing the unnecesarry 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.