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

1 System Perspective

1.1 Architecture

1.2 Dependencies

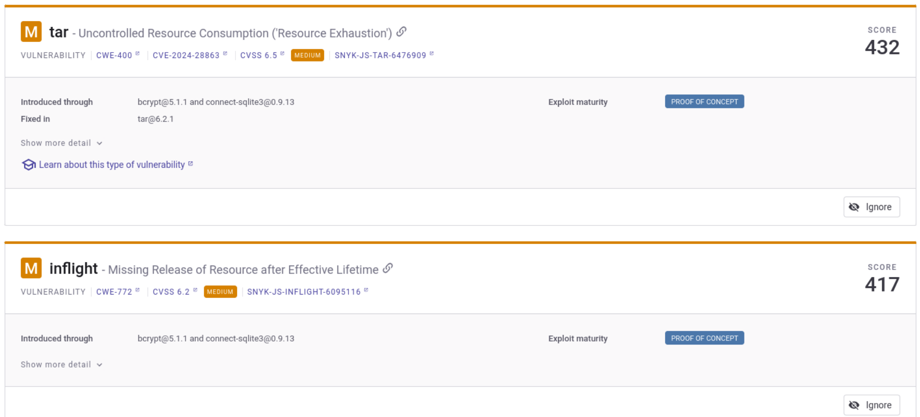


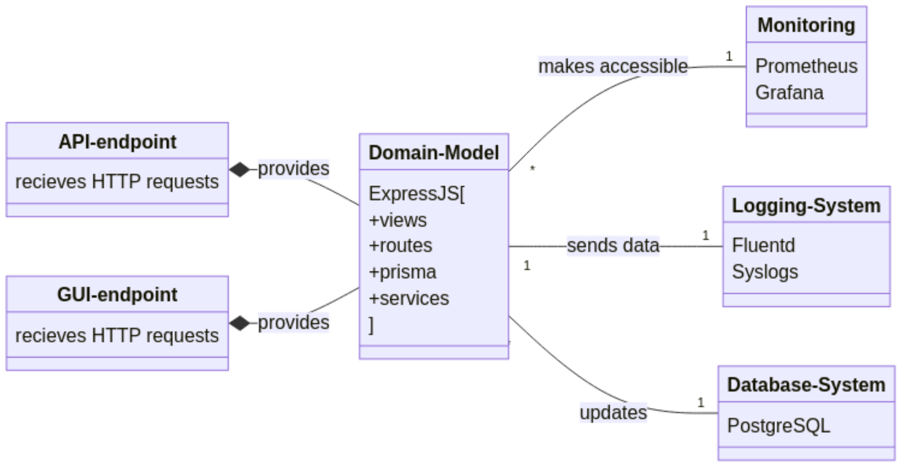
Figure 1: Snyk screenshot

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

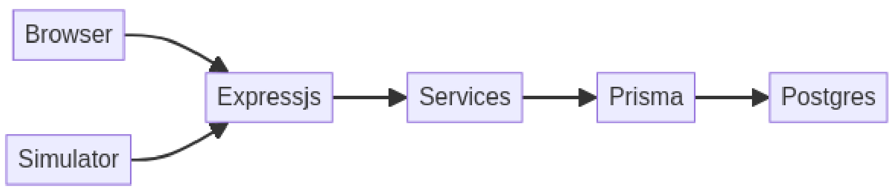
1.3.1 Module Viewpoint

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

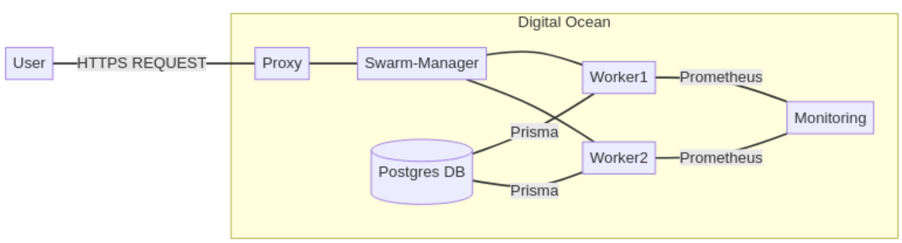


The above module viewpoint highlights how the expressjs application interacts with numerous systems with some being dependencies required for the running of the application, such as the postgres database, while others are tools meant for tasks such as monitoring and logging. What is not covered in this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

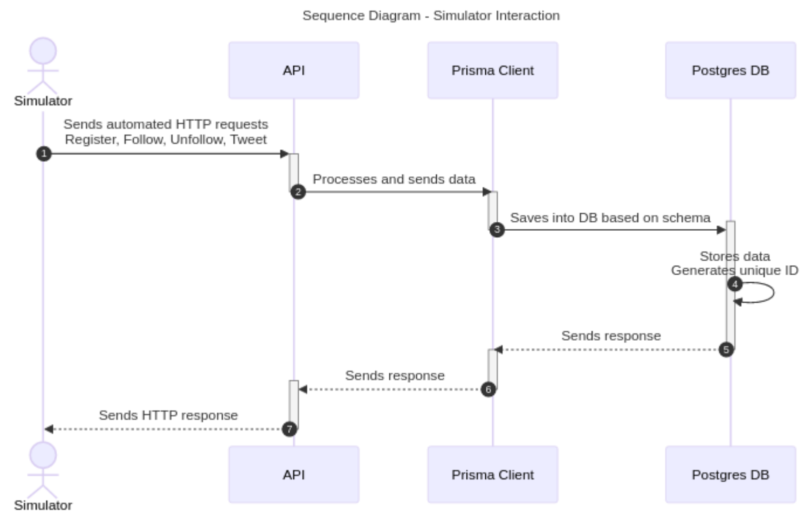


1.4 Important interactions

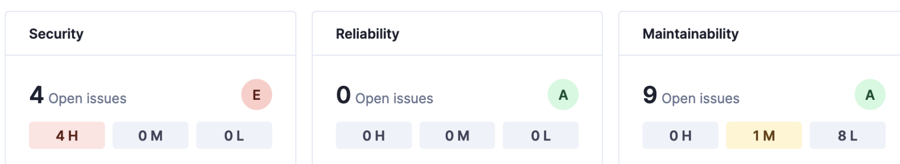
The system can be interaceted with in two ways:

- User Interface
- API for the simulator

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



1.5 Current State

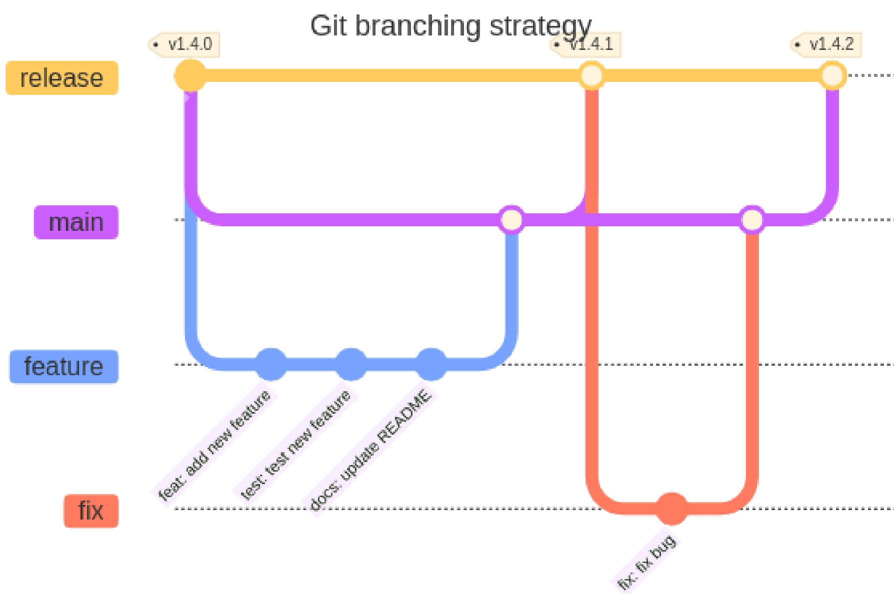


The application is practically fully functional, apart from a single outstanding bug. While the application has minimal technical debt, it relies on legacy code and dependencies to test the application (test suite and simulator).

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 pipline.

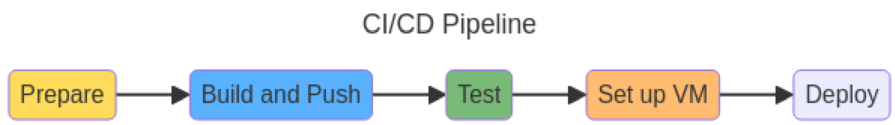
Succesfully merging a pull request to the release branch triggers the CD pipeline. Release tag is bumped according to the contents of the release, using the semantic versioning protocol.

2.2 Commit hooks

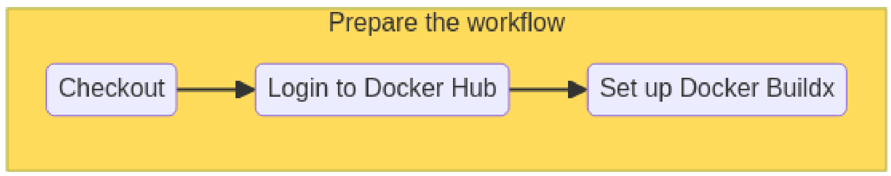
A pre-commit hook was added in d40fcba to lint and enforce commit messages and to follow the semantic versioning protocol. A CLI-tool was also added to aid developers write commit messages that follows the chosen protocol. Effectively standardizing a common development process, improving our process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

2.3 CI/CD pipline

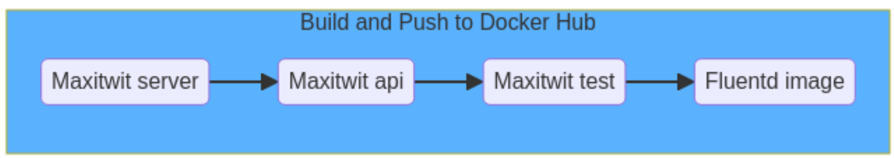
Our CI/CD pipleine is based on **Github Actions**. We have a deploy.yml file that is automatically triggered when new data is pushed to the **release branch**.



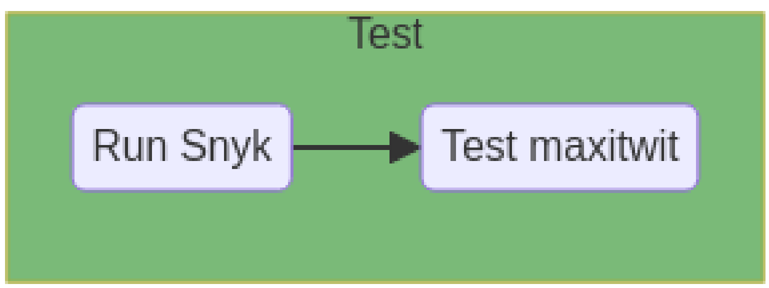
We prepare the workflow by checking out to our release branch, logging in to Docker Hub and setting up Docker Buildx so it can build the images.



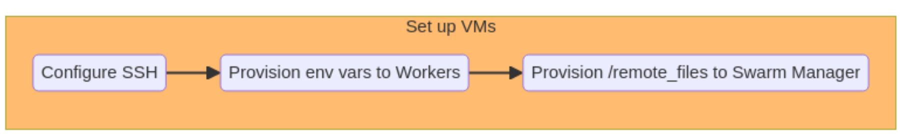
Th workflow builds our images and pushes them to Docker Hub.



The workflow runs snyk to check for vulerabilities, then 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 SCPd 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 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.

2.5 Security Assesement

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 enterying 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 implementation of the logging system proved difficult, especially as the system was prepared for scaling using docker swarm. Originally, a simple syslogs setup inside a droplet was created which was managed by the npm packaged winston and morgan. This solution proved inscalable in a docker swarm framework, as there would be no centralized logging. Thus, we attempted to expand on the system by adding a fluentd container to each droplet, which would recieve the logs from the winston npm package and send them all to a centralized storage droplet running elasticsearch and kibana. This however failed as the Elasticsearch integration kept crashing due to memory issues. To still provide centralized logs, we defaulted to have fluentd send logfiles to the droplets running the load balancers, which would store them in a /logs folder. Reflecting on this experience, had we from the beginning worked on implementing a scalable logging system, the amount of refactoring and experiential learning required for the implementation of the EFK-stack would have been diminished. In other words, it shows how technical debt can hinder the scaling of software solutions in practice.

3.2 Operation

- database migration
- system crashed due to failing fluentd container 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 certain logs, which led to the system rebooting. The logs of this crash are lined [here]. Such an issue would have been difficult to foresee, as it was isolated to a specific subset of events occuring in tandem. 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. The larger issue at hand was that 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. Thus, even though unlikely, the independant failure of multiple systems should be expected and guarded against. In our case, further manual testing of the website on a regular basis was deemed sufficient, however, it was discussed whether a shell script could be created to run get requests against the Api could be created, to have a continuous, reliant, status of the webapp.

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 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.