DevOps, Software Evolution and Software Maintenance

mkrh, mahf, jkau, lakj, ezpa

23 May 2024

1 Introduction

2 System's Perspective

2.1 Technology Stack

Our technology stack, as seen in figure 1, consists of Express.js and Node.js for both the front-end and back-end, utilizing JavaScript, Mocha, and WebdriverIO for end-to-end testing. Our database is managed by a PostgreSQL server and queried from the frontend with the ORM package Sequelize. It is hosted on a database cluster provided by Digital Ocean. The application itself is deployed on a Digital Ocean droplet. For monitoring and observability, we employ Prometheus for collecting metrics, and visualize the metrics using Grafana. Containerization and orchestration are handled by Docker and Docker Swarm, respectively. Continuous Integration and Continuous Deployment (CI/CD) pipelines are automated through GitHub Actions. Logging of MiniTwit is managed using the EFK (Elasticsearch, Filebeat, Kibana) stack. Currently, the logging is implemented with Sequalize, which sends the logs to the console. Filebeat gathers these logs from the console, as well as the and ships them to Elasticsearch. There it is indexed and stored, and Kibana provides a interface to search, analyze, and visualize the logs.

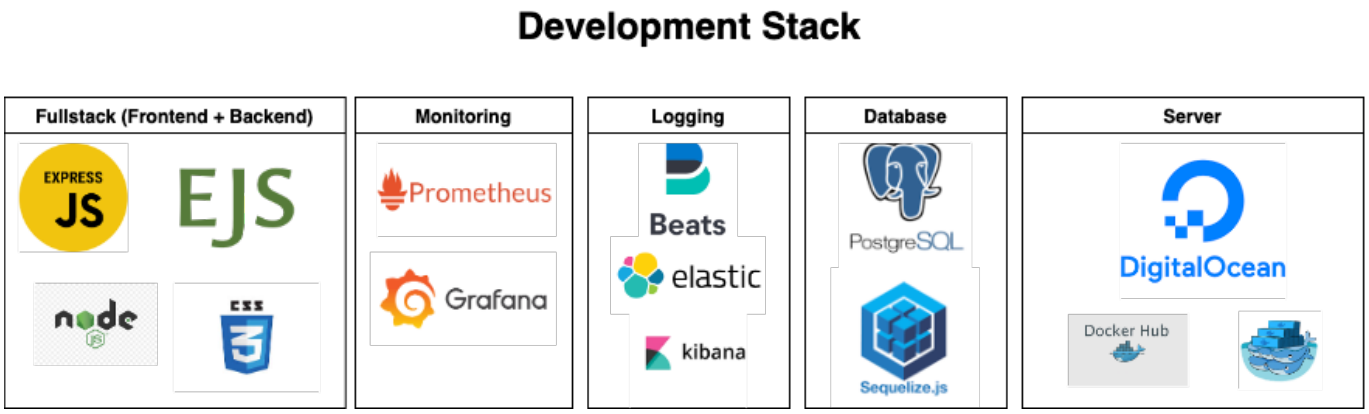

Figure 1: Overview of development stack, with each element grouped by use case.

2.1.1 Arguments for Choice of Technology Stack

Express.js, built on Node.js, is renowned for its non-blocking, event-driven architecture, which enhances performance and scalability and is well build and modern maintained programming language. By leveraging Express.js, we unify our technology stack under JavaScript for both front-end and back-end development which is neat for a simple webpage. Express.js integrates well with a wide array of modern development tools and services due to the extensive eco system of npm packages. Everything from PostgreSQL, Sequelize, WebdriverIO to small packages like express-flash (that made the refactor more simple).

Furthermore, JavaScript is the most used programming language in the world (**?**). When working in an interdisciplinary team with different backgrounds and skill sets, one should choose a language that all members can actively contribute to, even if they have no prior experience with the specific language. JavaScript

fulfills this criteria, with it being a widely known, high level programming language, and extensively documented. This allowed every member to contribute to the development of the app, whereas a lower level language with a higher barrier to entry would discourage collaboration.

2.1.2 Arguments for choice of Virtualization Techniques and Deployment Targets

Docker is a great choice for making environments portable across different systems and machines. It encapsulates the dependencies needed for development and the use of Dockerhub allows to centralize the repository for the docker images. Furthermore, we implement Docker Swarm so that we can scale our service horizontally across multiple hosts. When coupled with Terraform, it enables us to easily scale the amount of droplets for the swarm based on demand. Docker was voted #1 Most-used developer tool by Stack-Overflow 2023 Developer Survey (**?**) and therefor familiarising ourselves with it can improve our CV and work opportunities. The choice of Digital Ocean as a cloud provider was mainly based on recommendations and cost. Cloud providers can be very expansive and Digital Ocean provided us with a 200 USD free spending cap since we are students.

3 Process' Perspective

The following diagram describes the Process Development Landscape:

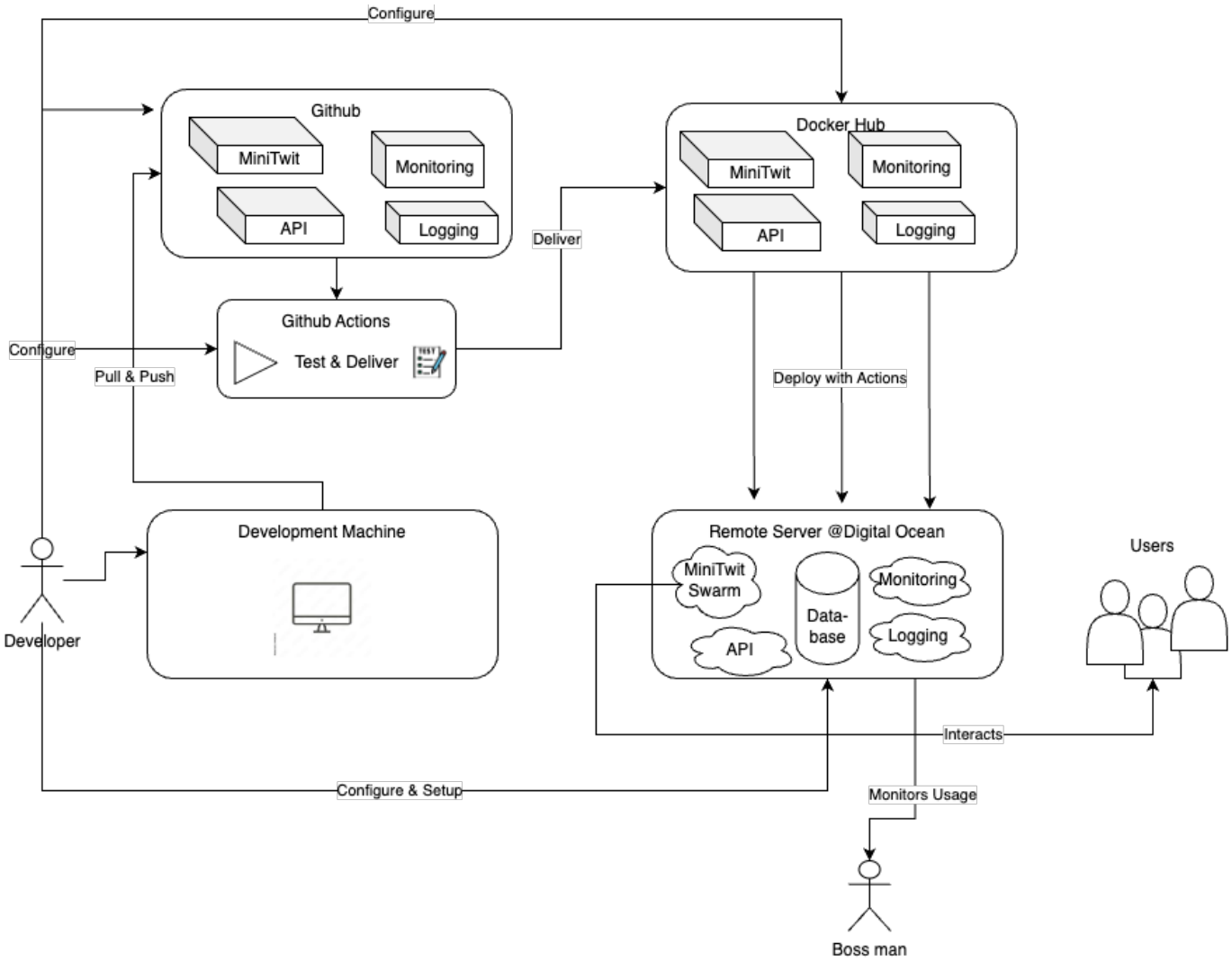

Figure 2: Process' Landscape

3.1 Implementing CICD with GitHub Actions

A complete description of stages and tools included in the CI/CD chains, including deployment and release of your systems - Remember to log and provide good arguments for the choice

of CI/CD system, i.e., why do you choose your solution instead of any other? For our CI/CD pipeline we use Github Actions. A study shows that GitHub Actions has become the most popular CI service, surpassing older, popular options like Jenkins and Travis CI (**?**). The authors provide several reasons for this shift. First and foremost is the seamless integration with GitHub. GitHub Actions benefits from its native integration with GitHub repositories. This close integration simplifies setup, maintenance, and overall workflow, making it a superior choice compared to other CI services like Travis CI or Jenkins, which might require additional configuration for GitHub projects. This seamless integration ensures that changes made in the repository are immediately reflected in the CI/CD process. Secondly, GitHub Actions offers predefined workflows and a wide array of reusable actions, significantly simplifying the setup of CI/CD pipelines. This feature allows us to quickly implement robust pipelines without needing to build everything from scratch. This approach not only speeds up the development process but also reduces the potential for errors, as these actions are maintained by the community and are regularly updated. Thirdly, cost-effectiveness plays a major role as well. GitHub Actions offers a free tier, which gives an advantage for open-source projects and smaller teams like ours.
The paper argues that these three points are a big contributor to the rise of GitHub Actions in the CICD space. Moreover, with more users adopting GitHub Actions, there are more forum posts, blog posts, guides, and tutorials available, increasing our odds of finding solutions to problems we encounter or detailed guides to specific workflows we need to implement.

3.2 How do you monitor your systems and what precisely do you monitor?

For system monitoring, we use *Prometheus* and *Grafana*. These services are separated from the rest of MiniTwit into their own repository[1]. We use Prometheus for collecting metrics from both *MiniTwit* and *MiniTwit-api*. To this end, both applications expose a `/metrics` endpoint that exposes metrics for Prometheus to scrape. Prometheus regularly scrapes all of MiniTwit for these metrics as defined in the `prometheus.yml` file. It is configured for both collecting metrics from the production system as well as a local instance.

Grafana is used to aggregate and visualize data collected by Prometheus. It integrates natively with Prometheus as a datasource, which is automatically configured through Grafana's `datasources.yml` configuration file. A dashboard is also automatically setup with some of the metrics Prometheus collects.

With this Prometheus and Grafana stack, we collect real-time information about MiniTwit's current traffic and state, as seen in figure 3. Specifically, we currently monitor CPU usage, the rate of traffic and the response time distribution, all of which are shown for both the app and the api. These metrics are collected and monitored, as they give an indication of the overall system state, if it is overloaded, and if it is experiencing slow response times; all of which are crucial to control in a production environment.

Figure 3: Grafana

How do we deploy it? Why not with rest of MiniTwit?

3.3 What do you log in your systems and how do you aggregate logs?

With our EFK stack we plan to log every single interaction a user might have with the website, as well as all the logs provided by docker. The sequalize.js file logs each user interaction to the console. Provided we

[1] `https://github.com/group-o-minitwit-2024/MiniTwit-monitoring`

had the time, Winston[2] would have been our preferred logging implementation, as it allows for additional logging capabilities. There are mainly two improvements for using Winston over Sequalize: Winston have 7 logging levels, ranging from 0 to 6 and the severity of all levels is assumed to be numerically ascending from most important to least important. This conforms to the severity ordering specified by RFC5424 (**?**). Secondly, Winston allows for the logs to be saved to a specified file, instead of only writing to console. In our `filebeat.yml` file, we specify a process called *add_docker_metadata*, which annotates each event with relevant metadata from Docker containers[3]. This ensures that all logs are enriched with context about their source.
Logs collected by Filebeat are sent to Elasticsearch, where they are indexed and stored. Using Kibana, we create two index patterns: one for logs from the website and another for logs from Docker. This setup allows us to efficiently search, analyze, and visualize logs from different sources in a centralized location. By aggregating logs in this manner, we can monitor system performance, troubleshoot issues, and gain insights into user behavior.

3.4 Brief results of the security assessment and brief description of how did you harden the security of your system based on the analysis

3.5 Applied strategy for scaling and upgrades

Docker swarm with terraform

3.6 Role of AI in the development process

3.7 Risk Identification

1) Identify assets (e.g. web application).
2) Identify threat sources (e.g. SQL injection)
3) Construct risk scenarios (e.g. Attacker performs SQL injection on web application to download sensitive user data

4 Risk Assessment

4.1 Risk Identification and Analysis

On the next page is a table summarising risk assesments. We prioritize the risks based on their combined likelihood and impact. We also include the mitigation strategies in the table to make it easier to get an overview. The table is in ranked order, from highest priority to lowest. The highest priority risks are those with high likelihood and high impact, such as SQL Injection Attacks, Data Breaches, and Software Vulnerability Exploits.We base our findings on Sonarcloud's produced rapport on our application as well as researched security threats and mitigation s

We use these definitions:

Likelihood: Certain, Likely, Possible, Unlikely, Rare

Impact: Insignificant, Negligible, Marginal, Critical, Catastrophic

[2] `https://github.com/winstonjs/winston`
[3] `https://www.elastic.co/guide/en/beats/filebeat/current/add-docker-metadata.html`

Table 1: Risk Assessment

Risk	**Likelihood**	**Impact**	**Mitigation Strategy**
SQL Injection Attack	Likely	catastrhopic	We use the ORM feature; Sequelize to avoid raw queries. We should regularly update dependencie. With our monitoring we should try to review the database logs and performance metrics for any suspicious access patterns and unusual spikes
Data Breach	Possible	Catastrophic	We encrypt sensitive data like passwords and use HTTPS.We also follow REST API prnincples. Generally we shoud strive to enforce strong authentication and authorization practices. We should also consider implementing security monitoring tools to detect and respond to breaches in close to real-time
DDoS Attack	Possible	Critical	We scale using Docker Swarm to try to distribute the system load. Again here, montitoring for suspicous patterns and the like is something we should include in our strategy. We can for instance set up alerts for sudden increases in traffic
Data Loss	Unlikely	Catastrophic	There are backups with the PostgreSQL database. There is also redundancy in Digital Ocean droplets. Looking ahead we could implement disaster recovery plans
Software Vulnerability Exploits	Likely	Critical	ll First of all we should make sure that software components are regularly updated. Dedicated vulnerability / security scans as CI GitHub Actions should also be considered. Altough current like Sonarcloud do uncover som vulnerabilites
Resource Exhaustion (CPU/Memory)	Rare	Critical	We monitor system performance with Prometheus. We also scale resources using Docker Swarm which should help optimize the application for performance
Unauthorized Access	Unlikely	Marginal	We try to have a strong password policy. Ideally we could implement multi-factor authentication. Monitoring is also relevant here.

5 Lessons Learned Perspective

The biggest lessons learned relate to the overall structuring and development of a larger collaborative project. While we attained significant hands-on experience with new tools and technologies, we found more value in learning to work together in a team with widely varied skill sets and backgrounds. When someone initially developed features early in the project, we would spend great time knowledge sharing and ensuring everyone

understood, what was going on. However, as we got further into the project, less knowledge was shared, while no documentation was written, leading to a decrease in productivity, as it became an increasingly daunting task to develop on the evergrowing complex system. This issue influenced the teams productivity both in terms of maintenance and evolution, since it was hard to work on elements without any knowledge. This is reflected in many of our GitHub Issues being left open for months at a time, preventing an agile workflow. One example of this is seen in the issue *"Race condition with regards to database initialization and app execution in compose #37"*. This maintenance task was left open for approximately 2 months, despite its relative simplicity. The biggest factor as to why this was not solved sooner is the lack of knowledge sharing, which turned a simple maintenance task into a whole self study on the infrastructure and networking of the app. This is just one example, but has been a problem all throughout the project.

Another lesson learned is that we should prioritize the integration of new features. When working, we had a tendency to try to perfect every little detail of an implementation before merging, leading to long development times, and minimal feedback. This had the consequence of work being invisible, and when something was finally done, there would be a huge batch of code that needs merging with the master branch and to be deployed. This is contradictory to the principle of flow (**?**), where work should be visible and often integrated. It also meant that there was no feedback flow, leading to stagnating development. One example of this is with our integration of *Docker Swarm* and *Terraform*. Both of these individually are huge steps which have major ramifications for the existing code and infrastructure. However, both was developed on one branch and merged in the same pull request (see pull request *"Docker Swarm and Infrastructure as code #64"*), leading to a longer than needed development time, a huge bulk of code, and a lot of minor details that needed to be tuned in production. In retrospect, it would have been better to either do Docker Swarm first, or do both of them in parallel but separated with smaller incremental developments.

Lastly, write about the fear of fucking up hindering development!!! Use example of indeces on prod database (I will write tomorrow, but fml I don't want to work anymore :))))

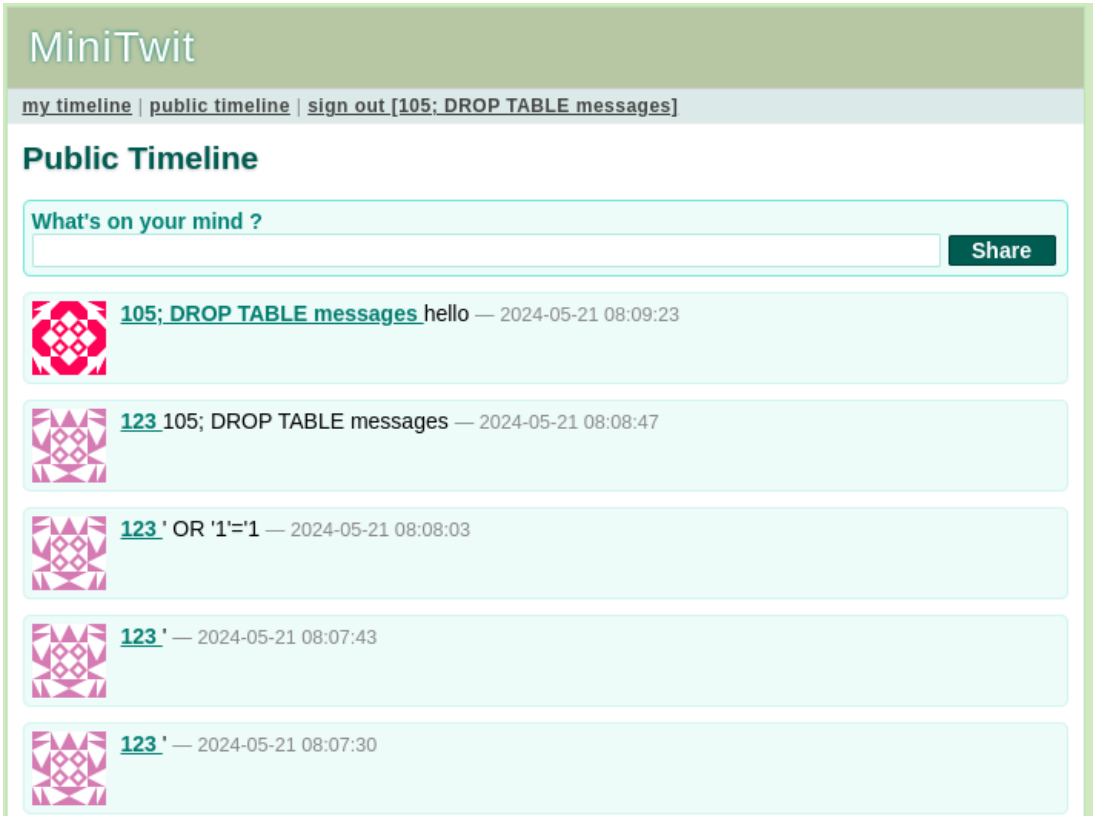

Figure 4: Caption

6 Appendix