DevOps, Software Evolution and Software Maintenance, BSc (Spring 2024)

Course code: BSDSESM1KU

May 22, 2024

Exam Assignment by:

Student	Email
Daria Damian	dard@itu.dk
David Zheng	jhou@itu.dk
Hallgrímur Jónas Jensson	hajj@itu.dk
Mathias E. L. Rasmussen	memr@itu.dk
Max-Emil Smith Thorius	maxt@itu.dk
Fujie Mei	fume@itu.dk

Contents

1 System's Perspective **3**
- 1.1 Design and architecture of the Minitwit system . . . 3
  - 1.1.1 Servers . . . 3
  - 1.1.2 Containers . . . 4
  - 1.1.3 App . . . 4
- 1.2 Dependencies of the Minitwit system . . . 5
  - 1.2.1 Monitoring . . . 5
  - 1.2.2 Logging . . . 6
  - 1.2.3 Development Dependencies . . . 7
  - 1.2.4 Runtime Dependencies . . . 7
- 1.3 Important interactions of subsystems . . . 7
- 1.4 Current state of the system . . . 10
  - 1.4.1 Static Analysis . . . 10
  - 1.4.2 Test coverage . . . 10
  - 1.4.3 Performance . . . 10
  - 1.4.4 Observability and Monitoring . . . 10

2 Process' perspective **11**
- 2.1 CI/CD Chain . . . 11
  - 2.1.1 Jobs . . . 13
- 2.2 Monitoring . . . 14
  - 2.2.1 API dashboard . . . 14
  - 2.2.2 Database dashboard . . . 15
- 2.3 Logging . . . 16
- 2.4 Security assessment . . . 17
- 2.5 Scaling strategy . . . 17

3 Lessons Learned Perspective **18**
- 3.1 Evolution and refactoring . . . 18
- 3.2 Operation . . . 18
- 3.3 Maintenance . . . 19

A Appendix **21**
- A.1 Artifacts constitutional to the project . . . 21

1 System's Perspective

1.1 Design and architecture of the Minitwit system

The following includes three abstractions of the system to understand the overall architecture better: the relations between the different servers, the relations between the different containers on the *webserver* hosting the main application, and the relations between the different packages in the main web application.

1.1.1 Servers

The system includes four servers serving different purposes, depicted in Figure 1.1. The *webserver* hosts the main application and the API reside, and the *dbserver* hosts the Postgres database, which is connected with GORM, an ORM package for Golang that facilitates the connection. Finally, the two worker servers are backup replicas of the *webserver* to help ensure its availability, connected to the *webserver* in a Docker Swarm environment with the 2 overlay networks: *ingress*, ensuring load balancing, and *minitwit-network*, allowing communication among the Docker daemons.

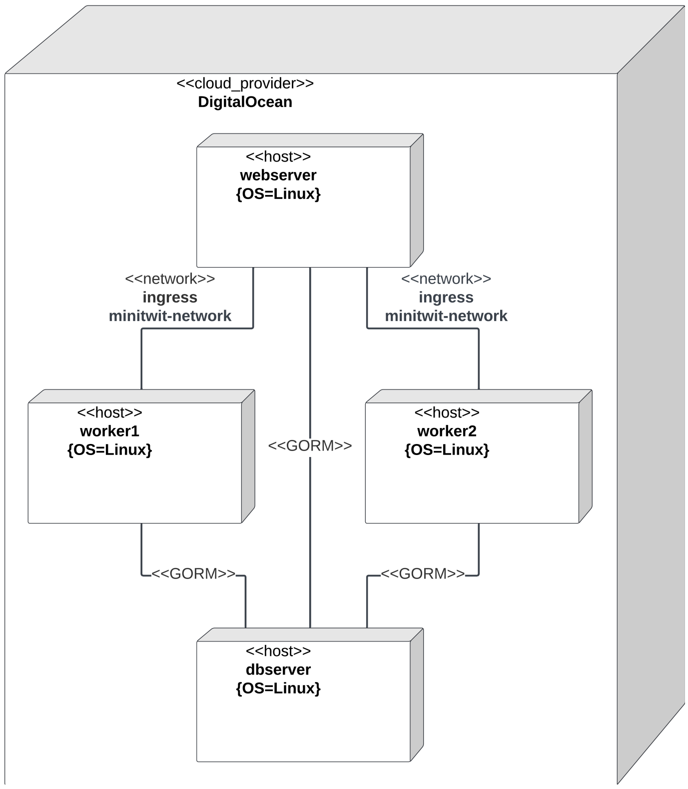

Figure 1.1: Diagram displaying the different servers in the architecture with relations.

1.1.2 Containers

Zooming in on the *webserver* reveals 2 different networks and 9 different containers running on this server. Figure 1.2 displays the containers running on the server.

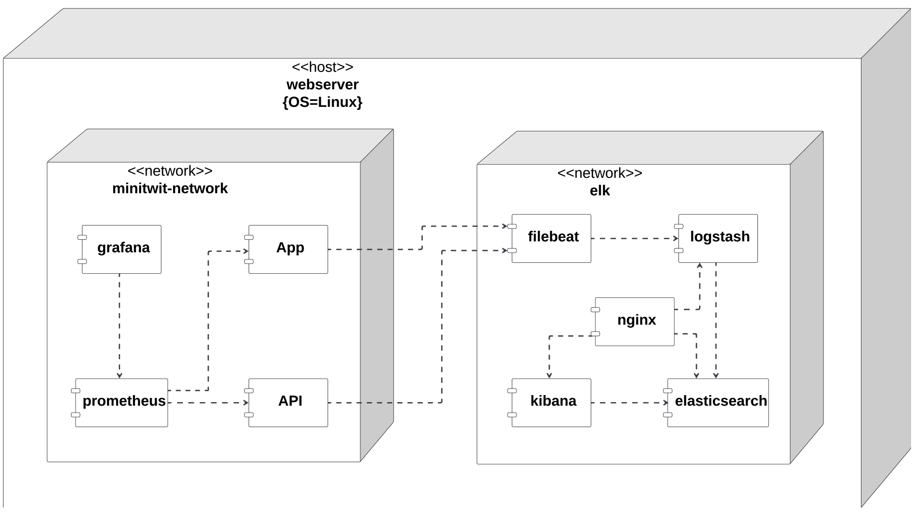

Figure 1.2: Component diagram emphasizing the different containers in the architecture with relations

1.1.3 App

The following describes the structure of the *App* container displayed in the services. It includes several packages, all located in the src directory. A package diagram that visualizes the app's structure is displayed in Figure 1.3. *main.go* is the executed file, which sets up the templates in the *Web* package, establishes database connection with the *Database* package, and sets up controllers in the *Controller* package to endpoints. The *Controller* package queries the database with the *Database* package, adds popup notifications with the *Flash* package, and creates ORM models with the*Models* package.

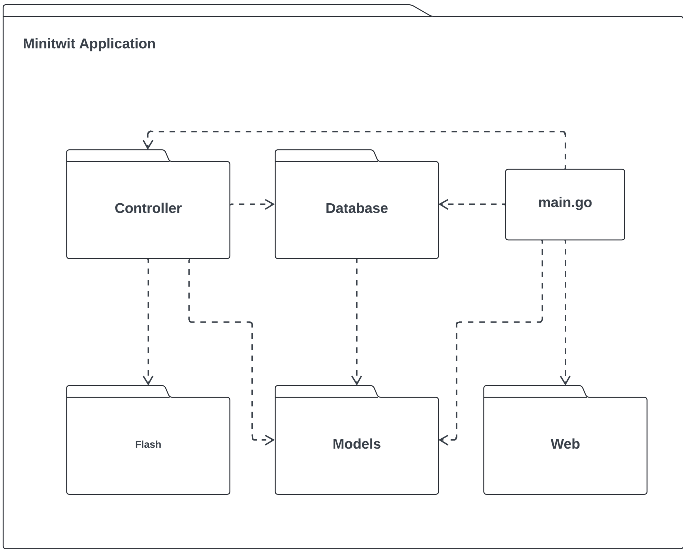

Figure 1.3: Package diagram of the main application running on the App service

1.2 Dependencies of the Minitwit system

This section will detail all the technologies, tools, and external services our system depends on, categorized for clarity.

1.2.1 Monitoring

The system is monitored with Grafana and Prometheus, with the stack depicted in Figure 1.4. Prometheus is the data collector, pulling metrics from the app and API. Grafana queries these metrics and visualizes relevant stats for maintenance. Furthermore, Grafana also connects to the database to retrieve information about the tables in the database.

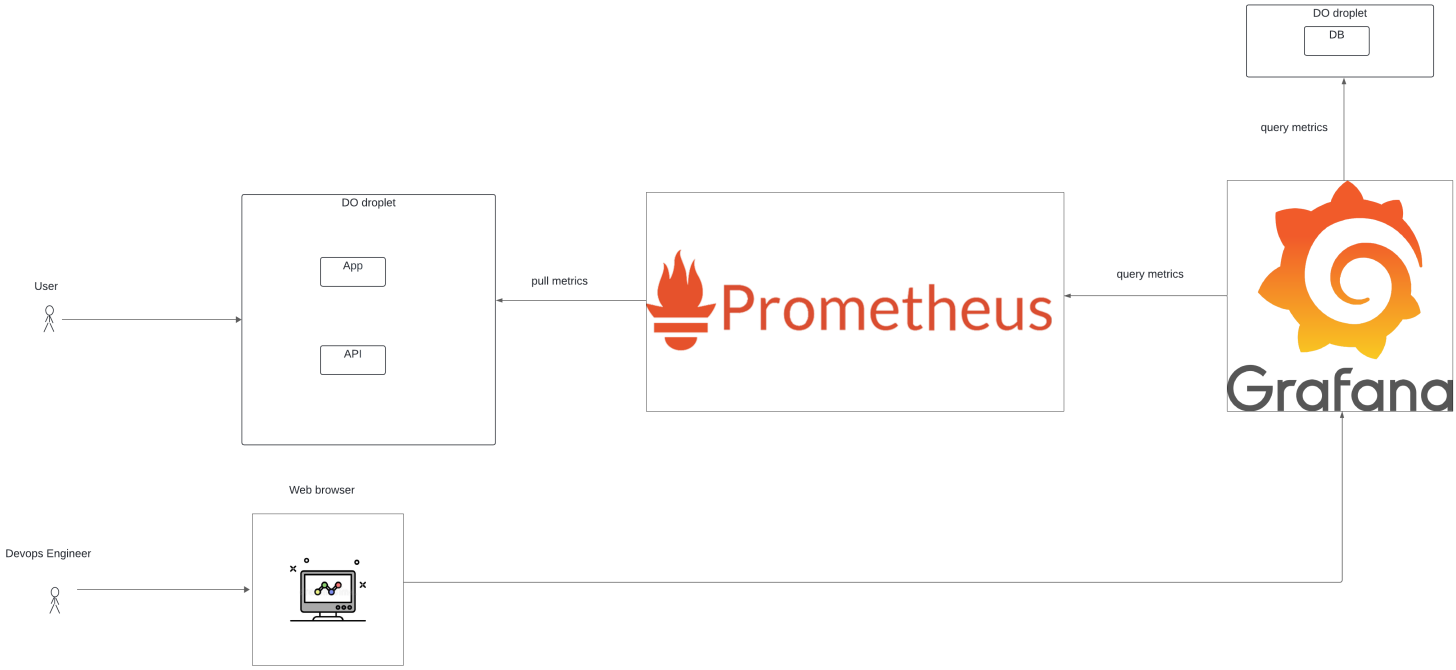

Figure 1.4: Monitoring stack

1.2.2 Logging

In Figure 1.5, you can see how our ELKB stack is structured to ship logs to Elasticsearch and further analyze them.

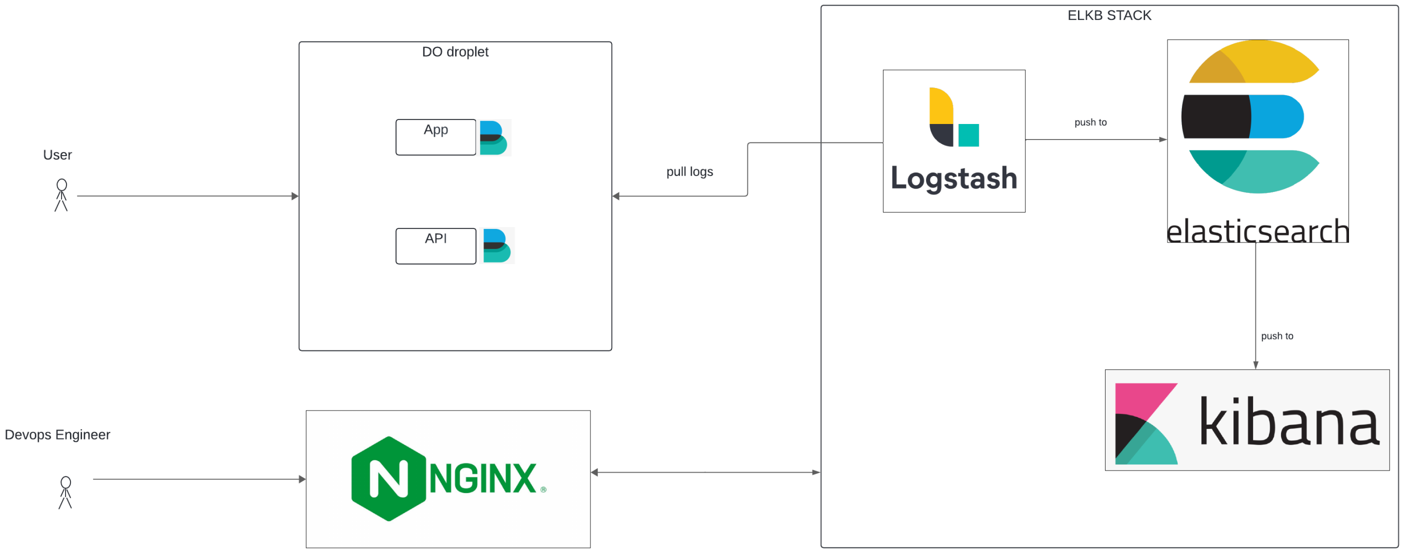

Figure 1.5: ELK stack

A label tag added to our *App* and *API* services enables Filebeat to auto-discover logs from those containers, which have that option enabled. Filebeat will collect logs only containing a timestamp, which are those generated by the logger of our code, and ship them to Logstash where they will be further processed to be outputted to Elasticsearch. Finally, the developer can analyze and visualize these logs through Kibana in the browser, where the stack will be reverse proxied by Nginx for authentication reasons.

1.2.3 Development Dependencies

We used Git and GitHub as the development version control and collaboration tools, and GitHub Organizations to manage the project with the "Projects" feature. GitHub Actions and Sonarcloud were used with workflows on pull requests to the *main* branch, automating the CI/CD process and analysis of code quality.

We used Terraform as the IaC, allowing for a simple IaC setup, with a *tfvars* file to include environment variables efficiently. All processes in the system are containerized with Docker to work without any dependency issues, using mainly docker-compose to orchestrate the services.

1.2.4 Runtime Dependencies

The application is hosted in a Linux server inside a DigitalOcean droplet, where it is containerized with Docker. The application and API connect to a Postgres database with GORM, a Golang ORM framework. The database can easily be replaced due to a volume connected to the container storing the data. The application and API are built with Golang and utilize the *Gin* dependency, an HTTP web application package.

1.3 Important interactions of subsystems

Information Flow of the Web Application

The diagram in Figure 1.6 showcases the system traversal of submitting a message. When a user submits a message, the system makes a POST request to our web server, which routes the request to the controller, where it performs all the logic. Afterwards, the proper status code is returned.

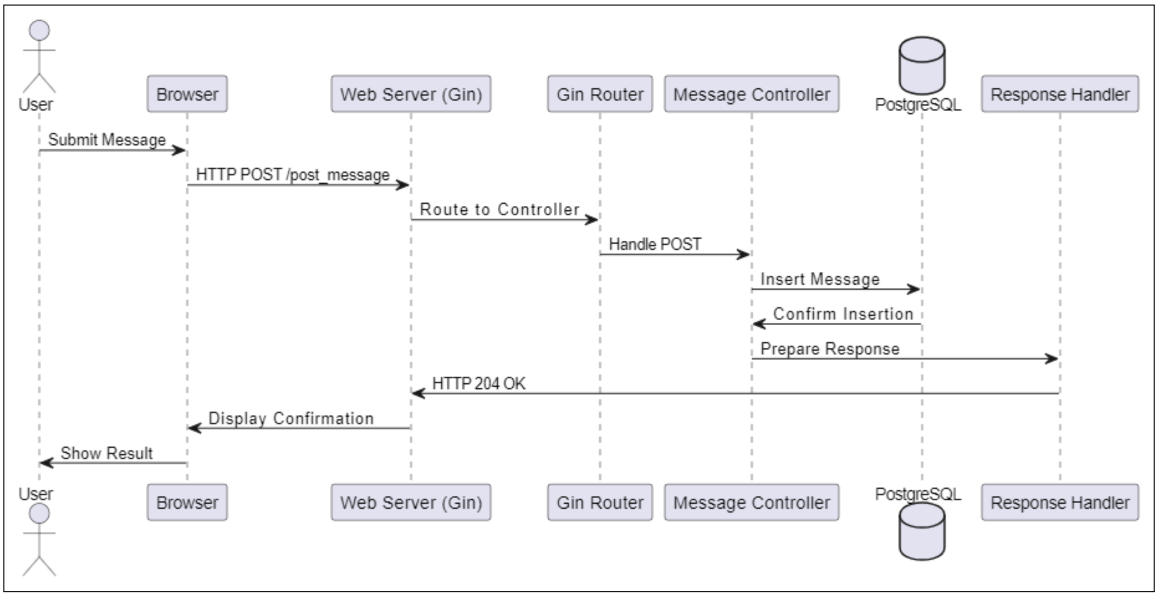

Figure 1.6: Sequence diagram of information flow of submitting a message in the application.

Flow of API Endpoints

Figure 1.7 displays how the simulator traverses the register endpoint, including exception handling of unfilled register form and username already registered.

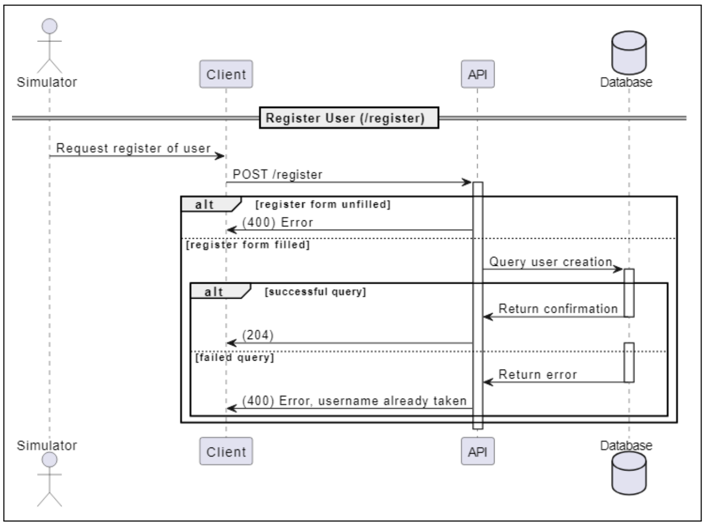

Figure 1.7: Sequence diagram displaying the traversal of the `POST /register` endpoint of the API.

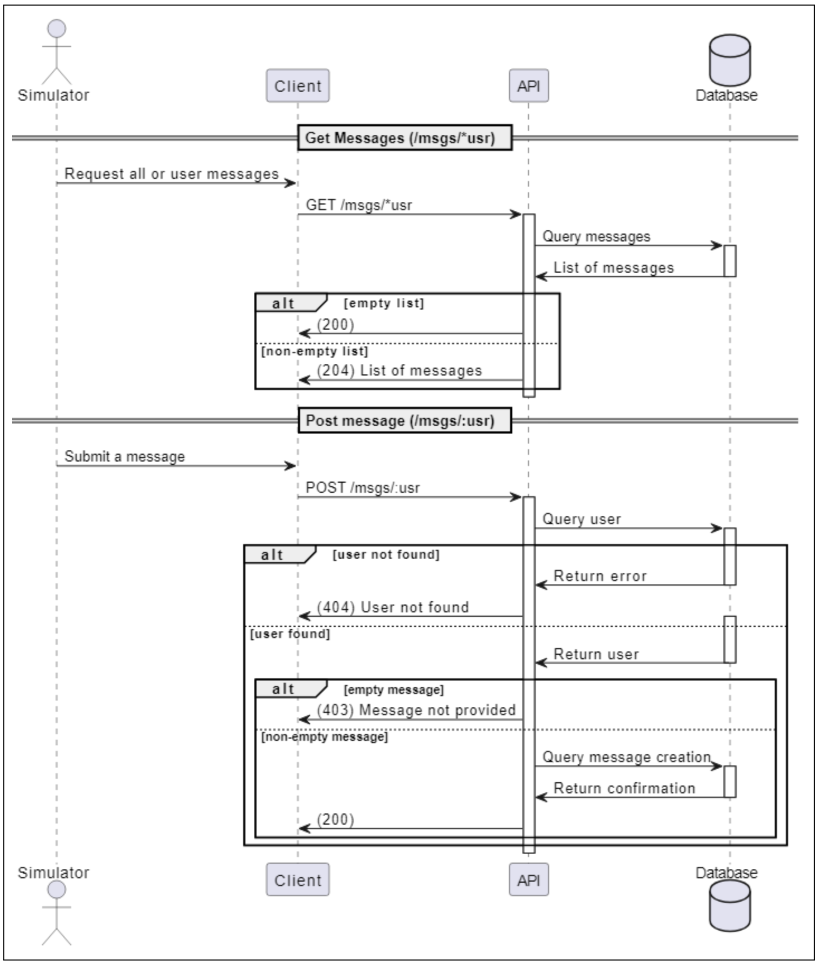

Figure 1.8: Sequence diagram displaying the traversal of the `/msgs` endpoints, including `GET /msgs/*usr` and `POST /msgs/:usr`, of the API.

Figure 1.8 displays the traversal of the `/msgs` endpoints, including error handling with unregistered users and empty messages. An important distinction is between the `usr`, where `*` indicates an optional user (fetching all messages if not provided), and `:` indicates a required user.

Figure 1.9 depicts the traversal of the `/fllws` endpoints, error handling at unregistered users, insufficient JSON format for the ORM, and if the query fails. Unsuccessful follow will return 400, and unsuccessful unfollow will return 403.

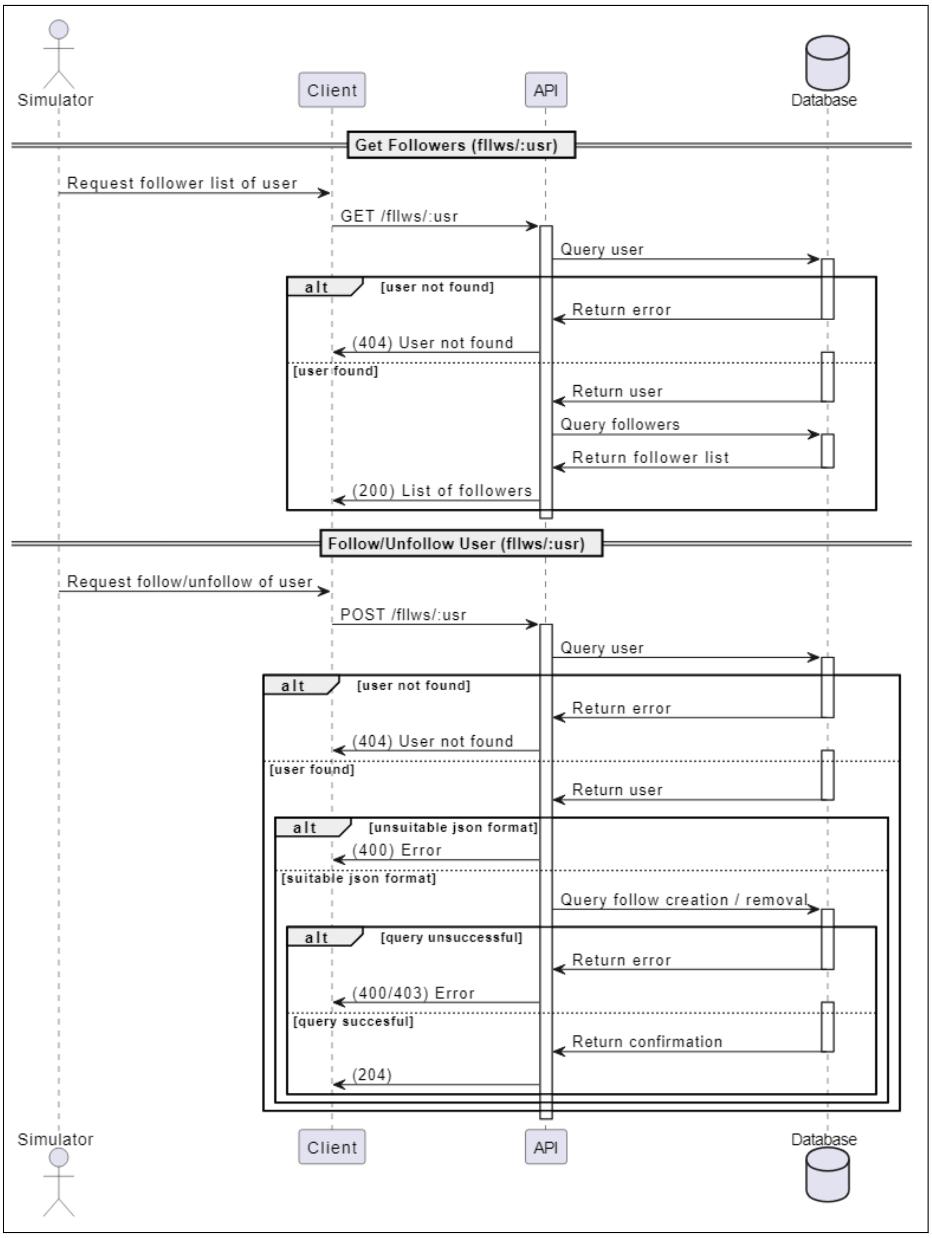

Figure 1.9: Sequence diagram displaying the traversal of the `/fllws` endpoints, including `GET fllws/:usr` and `GET /fllws/:usr`, of the API.

1.4 Current state of the system

All artifacts of the system are linked in Appendix A.

1.4.1 Static Analysis

We use SonarCloud for static analysis, performing scans automatically on pull requests. It identifies code smells, security and maintainability issues etc. Current issues are most low to moderate severity, mostly concerning maintainability, and should only require minor updates. Dependency health is good. The only security issues identified concern the use of recursive copying in a Dockerfile.

1.4.2 Test coverage

In the build stage of our CI/CD pipeline we test all the API endpoints, except for the "root", "version" and "metrics" endpoints with both positive and negative test cases, ensuring that the API should work according to the users expectations. Unit tests have not been implemented.

1.4.3 Performance

From our Grafana Dashboard, we can see that requests to our API currently take $\sim 9\,\text{ms}$ for the 99th percentile of requests. Unfortunately, the app webpage takes ~ 6 seconds to process requests, an issue that warrants further inspection.

1.4.4 Observability and Monitoring

In our logs, we have not detected any unusual activity. Monitoring and logging is further discussed in Sections 2.2 and 2.3.

2 Process' perspective

2.1 CI/CD Chain

We implemented a GitHub Actions workflow to automate the process of testing, building and deploying the most recent version of Minitwit, set to execute on each pull request to the *main* branch of our GitHub repository. The workflow is separated into three jobs: *BuildAndTest*, *Deploy*, and *Release*, which are codependent and executed sequentially,

A comprehensive deployment diagram describing the build and deployment part of the CI/CD applied with GitHub Actions is displayed in Figure 2.1.

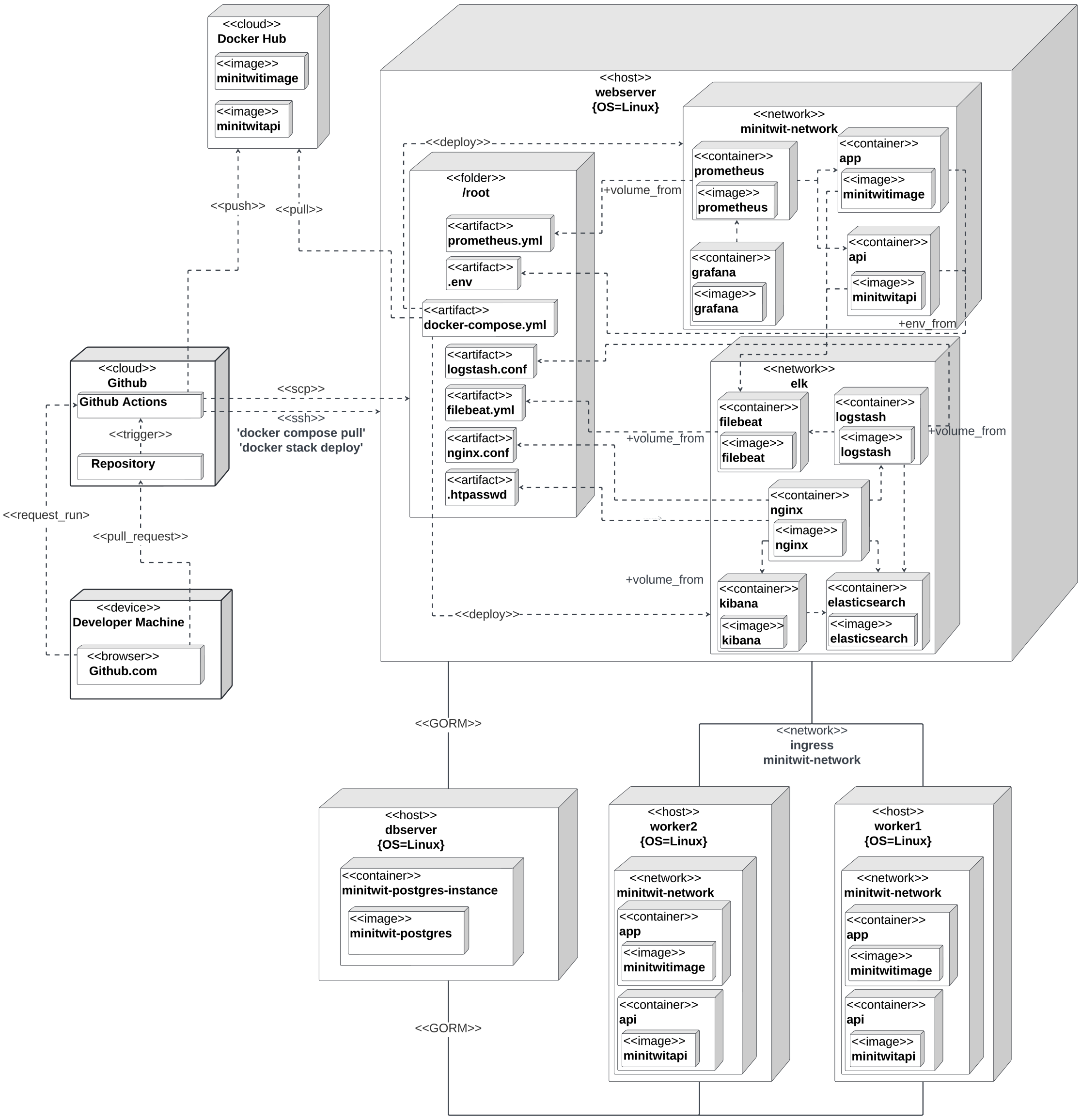

Figure 2.1: Deployment Diagram of deploying with a pull request or manually on GitHub Actions.

2.1.1 Jobs

BuildAndTest

All relevant images are built and pushed to DockerHub and then tested to ensure the system works. The key steps are:

1. **Checkout**
   - Uses `actions/checkout@v2` to fetch the codebase from the repository.
2. **Environment Setup**
   - Creates a `.env` file with database configurations sourced from GitHub secrets.
3. **Docker Operations**
   - Logs into Docker Hub using credentials from GitHub secrets to push built images, including:
     - The application image
     - API image
     - A test database image
4. **Python Setup and Dependency Installation**
   - Configures the Python environment and installs necessary dependencies.
5. **Testing**
   - Executes integration tests by setting up the application and its dependencies in Docker containers.
   - Conducts API tests and application-specific tests.

Deploy

The **Deploy** job is activated on successfully completing the **BuildAndTest** job. It deploys the application to the remote server where it is hosted. The steps are:

1. **Checkout**
   - Fetches the codebase from the repository, uses `actions/checkout@v2` similar to the first job.
2. **SSH Configuration**
   - Prepares SSH keys for connection to the deployment server.
3. **Deployment Execution**
   - Updates environmental configurations and transfers necessary files to the server using SCP.
   - Pulls the latest Docker images from the Docker Compose file.
   - Deploys the images using Docker Stack, updating the running application on all servers in the Docker swarm.

Release

Manages software versioning and public release.

1. **Version Calculation:**
   - Executes a script to determine the version number for automated version tracking.
2. **GitHub Release Creation:**
   - Uses `actions/create-release@v1` to create a formal release on GitHub.
   - Tags the release with the new version number and provides release notes.

2.2 Monitoring

We use Prometheus to collect and integrate metrics with Grafana to visualize key metrics live. Additionally, Grafana queries the database to visualize the data stored. We have two dashboards, one monitoring the API and the other the database. Both dashboards are passive monitoring, sniffing the simulated users' HTTP requests[1]. They are also both considered proactive monitoring by measuring application performance[2], and both are considered whitebox[3] since they are applied inside the codebase. DigitalOcean also includes infrastructure monitoring with real-time bandwidth, CPU usage, and disk I/O graphs, as depicted in Figure 2.2.

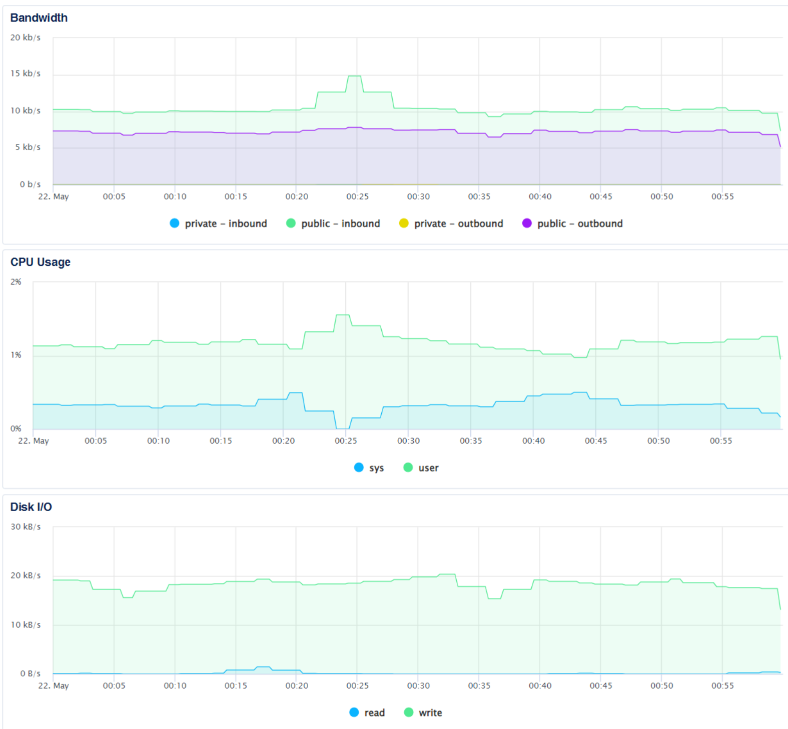

Figure 2.2: DigitalOcean Monitoring

2.2.1 API dashboard

The API dashboard mainly focuses on determining which HTTP requests are incoming from the API, prioritized due to the simulator using the API. The dashboard collects data from Prometheus. The monitoring uses an application monitoring tactic, monitoring how well the API responds to requests and which requests occur[4]. The metrics from Prometheus are pulled from the code, which pushes all metrics into an API endpoint, so it is somewhat of a hybrid monitoring. Figure 2.3 displays the API monitoring dashboard.

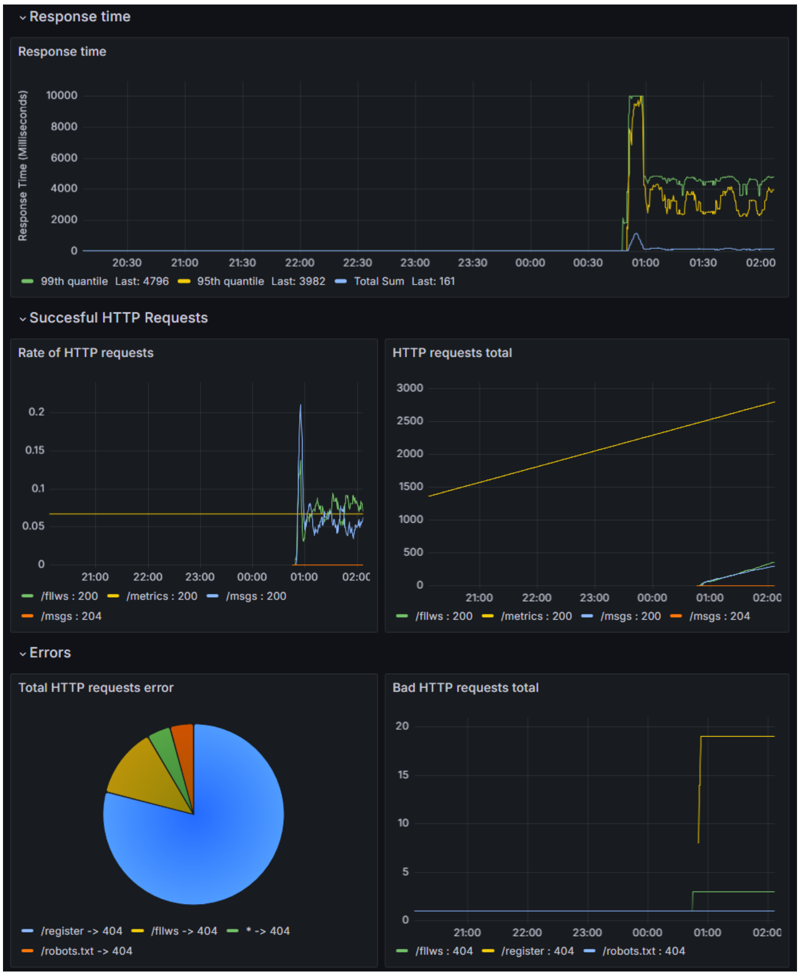

Figure 2.3: API Dashboard in Grafana

2.2.2 Database dashboard

The database dashboard visualizes stats about what is currently obtained in the database by querying the database directly from Grafana. The database monitoring acts like a 'monitoring the business' tactic by visualizing information about the number of users and messages there are and revealing distributions that may be useful for business [4]. The database statistics are considered pull-based since the database is queried to and, therefore, pulled from. Figure 2.4 displays the database monitoring dashboard.

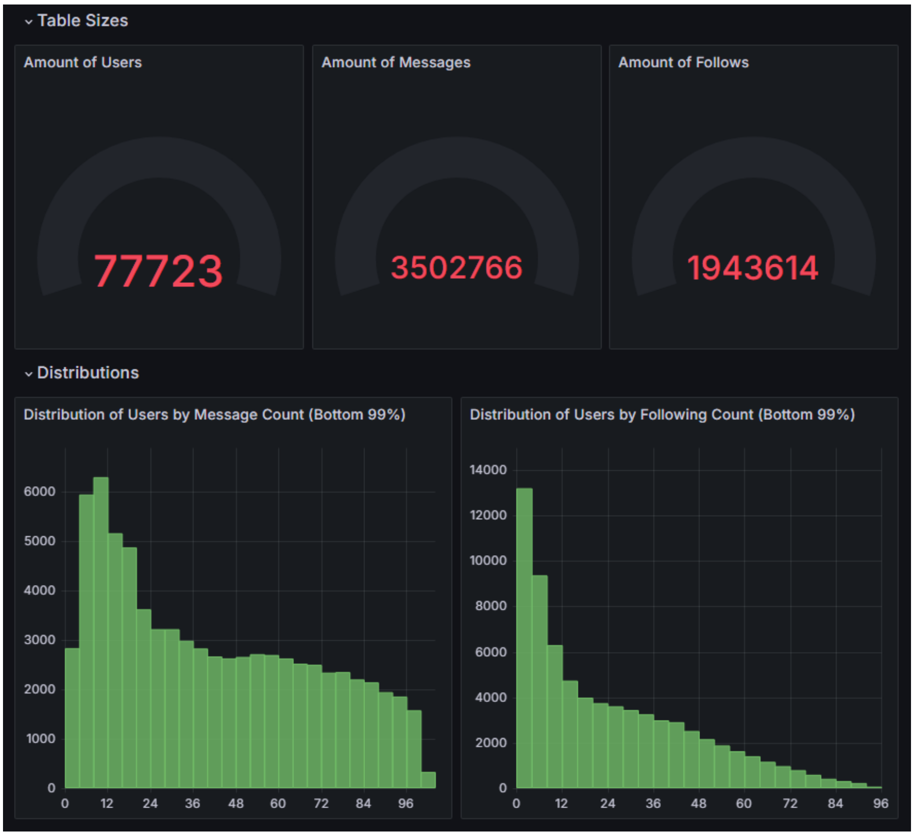

Figure 2.4: Database Dashboard in Grafana

2.3 Logging

The main events of the application and API are logged using 4 levels: info, debug, error, and fatal. We used the "info" level to communicate the events in a general level of how things are happening; "debug" for debugging purposes, how parameters are being changed and passed; "error" when the program encounters some non-fatal errors for example, in our Go app when the "err" parameter is not nil we notify it with the error level; finally "fatal" is used for errors that halt or crash the application.

All logs are collected in a single index pattern from all sources from the "app" and "application. We can distinguish the sources in Kibana because you can see from which folder and which file the log is generated, depicted in Figure 2.5.

>	May 10, 2024 @ 13:06:59.269	2024-05-10T11:06:59.269Z	DEBUG	controller/login.go:20 Getting	{"Username": ""}
>	May 10, 2024 @ 12:46:52.927	2024-05-10T10:46:52.927Z	DEBUG	controller/messages.go:18 Getting messages from:	{"User": "", "Page": "0"}
>	May 10, 2024 @ 12:46:52.927	2024-05-10T10:46:52.927Z	DEBUG	controller/login.go:20 Getting	{"Username": ""}
>	May 10, 2024 @ 12:45:42.327	2024-05-10T10:45:42.327Z	DEBUG	controller/messages.go:18 Getting messages from:	{"User": "", "Page": "0"}
>	May 10, 2024 @ 12:45:42.327	2024-05-10T10:45:42.327Z	DEBUG	controller/login.go:20 Getting	{"Username": ""}
>	May 10, 2024 @ 12:40:34.264	2024-05-10T10:40:34.250Z	INFO	src/main.go:108 Setting up db...	
>	May 10, 2024 @ 12:40:34.264	2024-05-10T10:40:34.250Z	INFO	database/database.go:16 Setup DB, getting env variables	
>	May 10, 2024 @ 12:40:34.263	2024-05-10T10:40:34.250Z	INFO	src/main.go:102 Starting endpoint...	
>	May 10, 2024 @ 12:40:34.116	2024-05-10T10:40:34.116Z	INFO	database/database.go:16 Setup DB, getting env variables	

Figure 2.5: Logs from Kibana

2.4 Security assessment

The application components are as described in chapter 1 section 1.1.

The user data that we store in our database are username, password, and email. The username and email are public information and are therefore not sensitive data. Depending on the user, and if they are reusing their password on multiple applications, the password is not sensitive data.

Impact / Likelihood	Very unlikely	Unlikely	Possible	Likely	Very likely
Catastrophic				Cross-site script	
Major				SQL Injection	Security logging failure
Moderate				Outdated components	
Minor		Cryptographic Failure		MITM databreach	
Insignificant			Weak Passoword	DDOS Attack	

Figure 2.6: Security risk assessment matrix[5][6]

2.5 Scaling strategy

The system induced both horizontal and vertical scaling. Horizontal scaling is implemented with Docker Swarm, with two additional worker replicas to assist and backup the single web server in its tasks from the app and API, which can easily be scaled by increasing workers. The web server is the manager who delegates tasks to the workers and can handle tasks itself if delegation is unnecessary.

Vertical scaling was implemented since Elasticsearch was very performance-heavy; as a quick solution, the web server was scaled vertically with more memory from 1 to 8 GB and more processing power from 1 to 4 virtual CPUs. This solution is not easily expanded and not preferred.

3 Lessons Learned Perspective

3.1 Evolution and refactoring

We learned the whole process of making software evolve to adapt to the best technologies. We refactored a legacy Python application to Go.

The major issue we found is how to map the functionalities written in Python to Go. What we learned is that when you have to migrate a codebase from a language to another it's best to take small steps, first, map the core functionalities, test it and develop the rest of functionalities incrementally to ensure correct integrations.

3.2 Operation

Creating the workflow for our pipelines was easy, it was just a 4-step process: build, test, deploy and release operations. The main issues we found while trying to implement it were in test and releasing.

For the testing part, it was difficult since we wanted to keep the test provided by the course which were in Python, we could have written some test in Go and run some Go command which would run them automatically, but since we decided to keep the Python test, we had to have running instances of images for the APP and API tests. But the main problem we faced was that, once we had a running instances, these would be connecting to an already populated database image because the docker-compose file was connected to it, to fix this issue, we made 2 Docker Compose files, one for development and another for production, the development would have a database image, and when we bring up the instances for testing, we build the development file, and they will be connected to an empty database.

For releasing, we didn't know how to update the release number after each deployment, we thought about using the tag of the commit for releases. Since we didn't find a way to get the tag number, we wrote a shell script where it would get the latest release of our application and update it for our newest release.

3.3 Maintenance

Keeping track of the logs can help us detect any bugs in the code and trace them so they can be fixed, assisting in maintenance, which was something we never worked with before. Additionally, monitoring could check the performance of our endpoints and determine if any errors or unusual activity have occurred so we can take action if necessary, e.g., in Figure 3.1 you can see we raised an issue because we found read time out errors from our monitoring stack, this is an example of how we would find an issue and manage it. This was also something new that we never tried before.

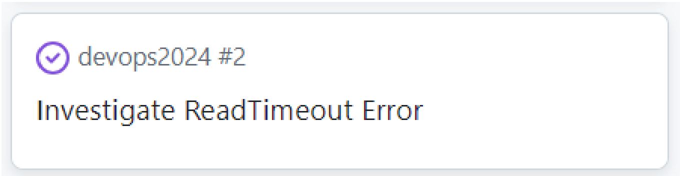

Figure 3.1: Issues for fixing Read Time out error

When scaling the application with Docker Swarm, we faced an issue where the Grafana dashboards and login credentials were removed due to creating a new container, which we fixed by adding a volume to the Grafana container, see Figure 3.2. Furthermore, Elasticsearch was very performance-heavy, so we upgraded the webserver host with more memory and CPU power.

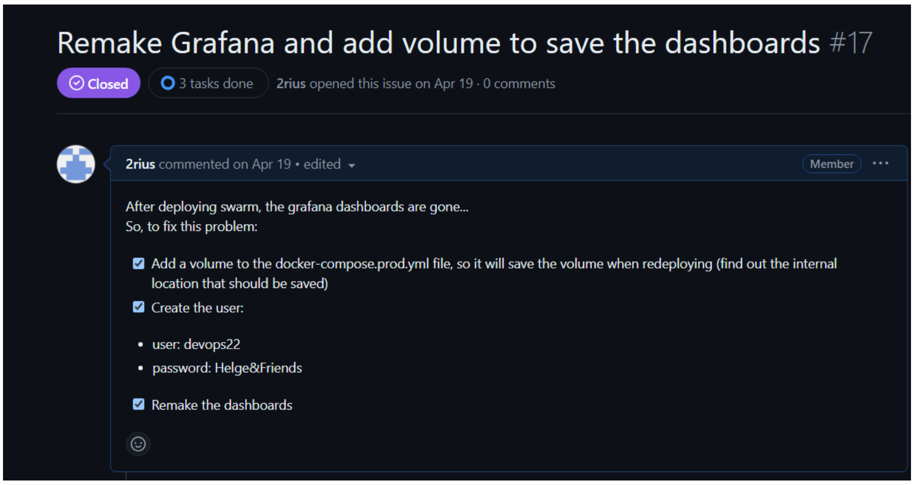

Figure 3.2: Issue about Grafana dashboards disappearing

Bibliography

[1] David Furnari. Active, or passive monitoring, that is the question, 2014. URL: https://web.archive.org/web/20170601101557/https://www.moviri.com/2014/01/active-passive-monitoring/.

[2] James Turnbull. A monitoring maturity model, 2015. URL: https://www.kartar.net/posts/a-monitoring-maturity-model.markdown/.

[3] James Turnbull. *The Art of Monitoring*. Turnbull Press, 2016.

[4] Mike Julian. *Practical Monitoring*. O'Reilly Media Inc., 2018.

[5] OWASP Foundation. Owasp top ten, 2024. URL: https://owasp.org/www-project-top-ten/.

[6] Paul Bischoff. Ddos statistics, facts, and history, 2024. URL: https://www.comparitech.com/blog/information-security/ddos-statistics-facts/.

A Appendix

A.1 Artifacts constitutional to the project

- The GitHub repository for the codebase can be found at https://github.com/DevOps2024-Organization/devops2024
- The issue tracking can be found at https://github.com/orgs/DevOps2024-Organization/projects/1.
- The application can be found at http://104.248.43.157:8080/
- The API can be found at http://104.248.43.157:5000/
- The Kibana dashboard can be found at http://104.248.43.157:5601/
- Grafana Dashboards can be found at http://104.248.43.157:3000/.