# Exam Report:
# DevOps, Software Evolution and Software Maintenance,
# MSc CS (Spring 2021)
# Course code: KSDSESM1KU

Frederik Henrik Martini, frem@itu.dk
Jesper Falkenberg Henriksen, jefh@itu.dk
Jonas Ishøj Nielsen, join@itu.dk
Nikolaj Mertz, nime@itu.dk
Sif Amalie Rybjerg Kristensen, sikr@itu.dk

19/05/2021

# Contents

# 1 System's perspective

## 1.1 Design

The Minitwit application is a simple application allowing users to post messages, see messages posted by others and follow other users in order to specifically see what they post.

The application is implemented in Java with the Spark Framework and a MySQL database. Java was chosen since it is platform independent and a popular language, meaning that extensive documentation is available and it would be easy for a future team to inherit the code. Spark was chosen as it is a lightweight framework like Flask, which was used for the inherited Minitwit system. Spark also has a reasonable learning curve and is straight forward to use for an application the size of Minitwit. The frontend consist of HTML templates connected to the backend using the template engine Jinjava, chosen to mimic the python Minitwit with little to no changes.

MySQL was chosen for persistent data storage due to its high performance, wide adoption rate, indices and familiarity to the team. Furthermore, the relational model fits the Minitwit data well and figure 1 depicts the design of the database.

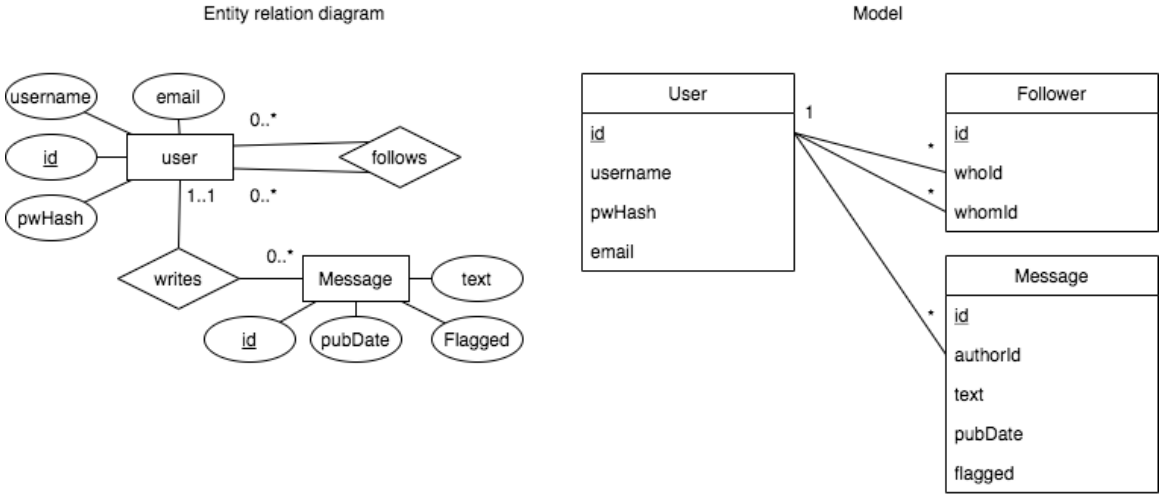


Figure 1: Entity relation diagram and diagram of model presenting the design of the database tables.

The ORM tool used is NORM, which is a lightweight ORM that does not require the complex markdown files that tools like Hibernate or JPA introduce. It is suitable for smaller projects with simple models and therefore fulfills the requirements of the Minitwit application. As an ORM tool NORM is not perfect as some queries still requires writing sql, like when joining or adding indexes.

Hibernate was implemented on branch feature/Hibernate[1] as a test to see if it would be a better fit. Unfortunately, it turned out that the tool did not deliver on its promises and actually did not support some union queries, meaning that it only introduced more unnecessary complexity without adding any benefits. Furthermore, it complicated database joins which are needed for Minitwit. We therefore decided to stick with NORM.

The application is hosted on DigitalOcean. Measures taken to ensure availability and maintenance, monitoring with Prometheus and Grafana and logging using an EFK-stack, are described later.

[1] `https://github.com/DevOps2021-gb/devops2021/tree/feature/Hibernate`

## 1.2 Architecture

Figure 2 shows the different components of the application and how they are connected.

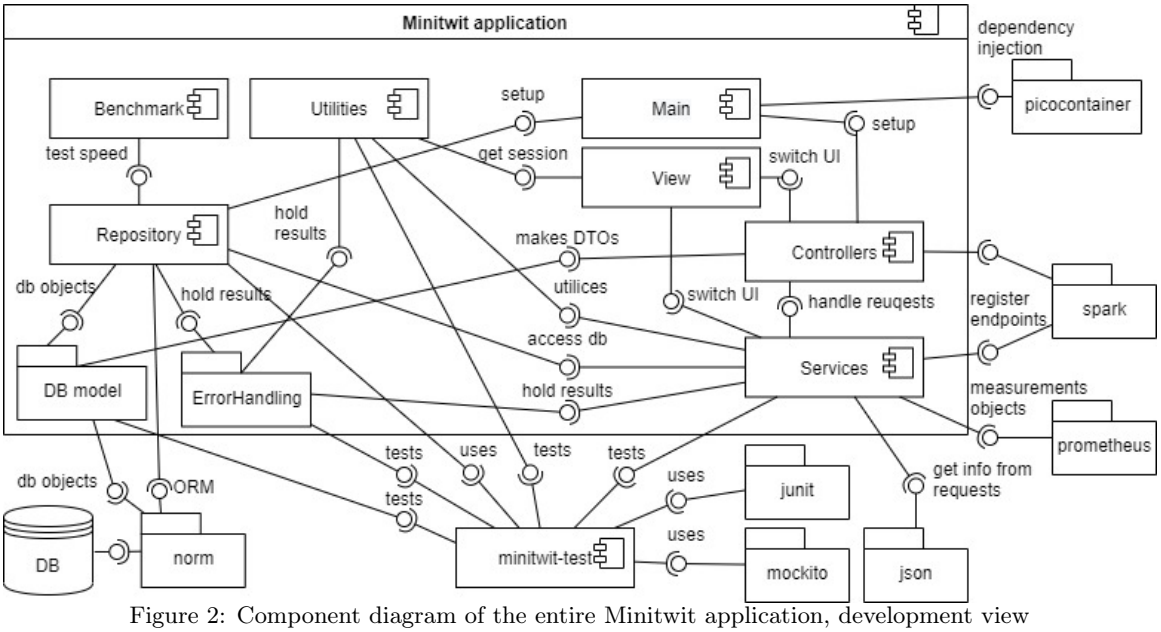


Figure 2: Component diagram of the entire Minitwit application, development view

### 1.2.1 Main

Main is the system entry point and allows specification of which database to use. Main starts monitoring and logging services and defines the number of threads to use. Figure 3 presents a class diagram.

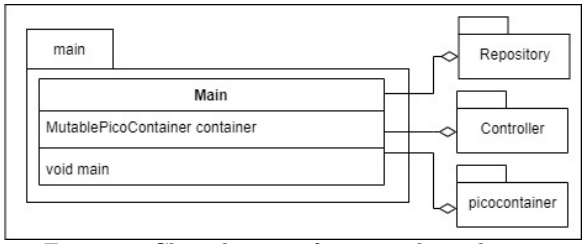


Figure 3: Class diagram for main, logical view

### 1.2.2 Utilities

Utilities, shown in figure 4, contains helper methods like hashing, formatting dates and JSON, and houses Request, Response and Session wrapper objects to use with Spark.

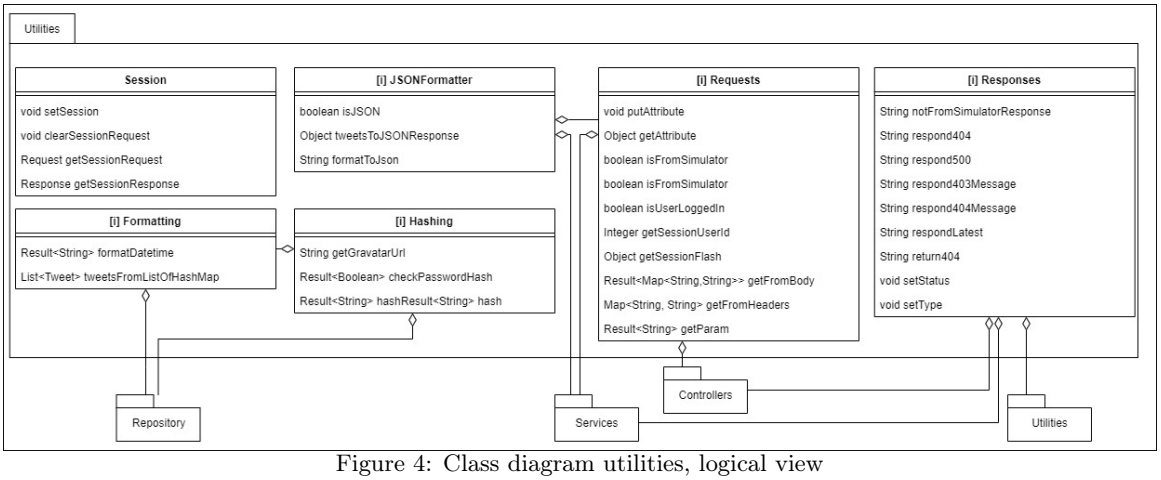


Figure 4: Class diagram utilities, logical view

### 1.2.3 Error handling

Error handling contains objects used to wrap results of computations that can result in exceptions, allowing for better error handling and reduction of try-catch-statements. A class diagram is presented

in figure 5.

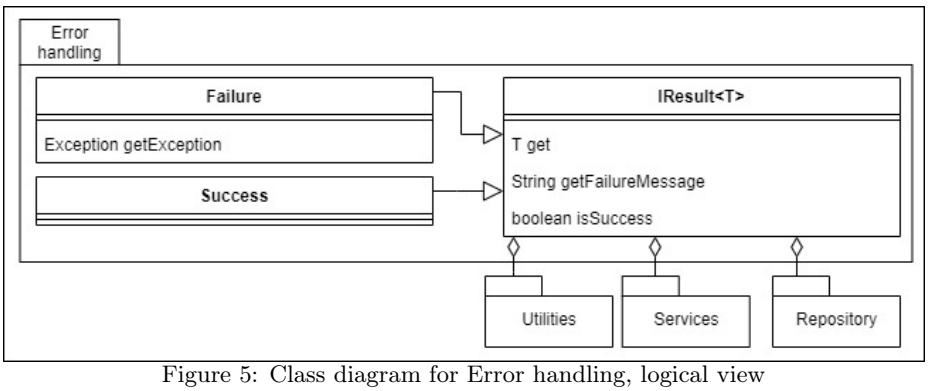


Figure 5: Class diagram for Error handling, logical view

### 1.2.4 Controllers

Controllers handles the API endpoints the application provides by mapping post and get request to appropriate methods. The class diagram can be seen in figure 6.

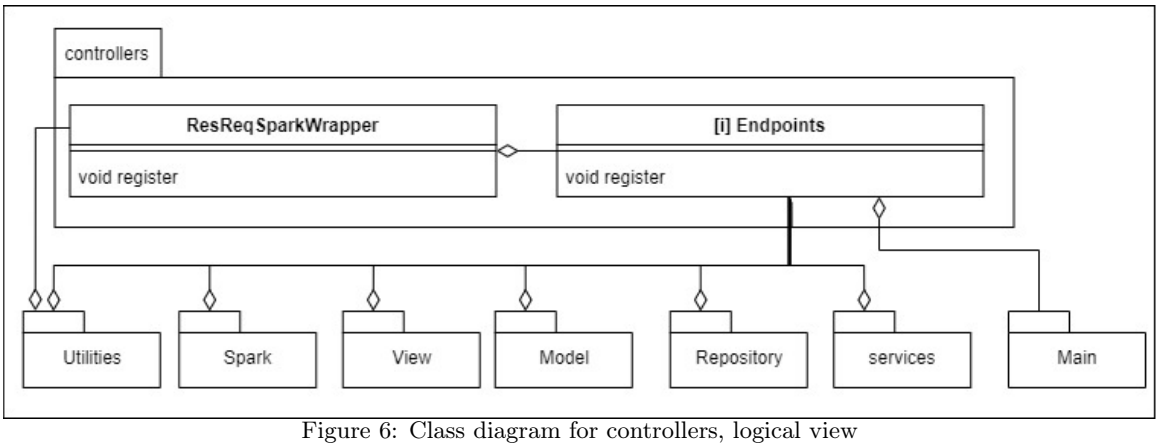


Figure 6: Class diagram for controllers, logical view

### 1.2.5 View

View contains code for rendering and redirecting between pages of Minitwit. Pages are rendered html-templates. The class diagram can be seen in figure 7.

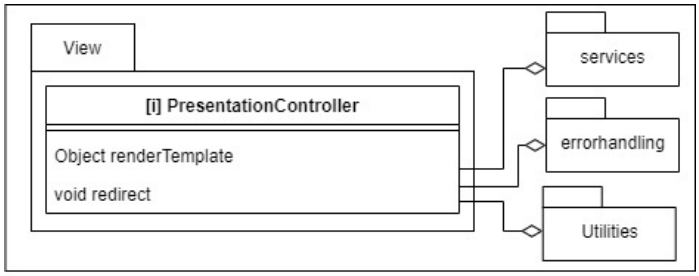


Figure 7: Class diagram for View, logical view

### 1.2.6 Services

Services includes functions executing requests and responses along with code for gathering and printing information to be used by the monitoring and logging services. Class diagram presented in figure 8.

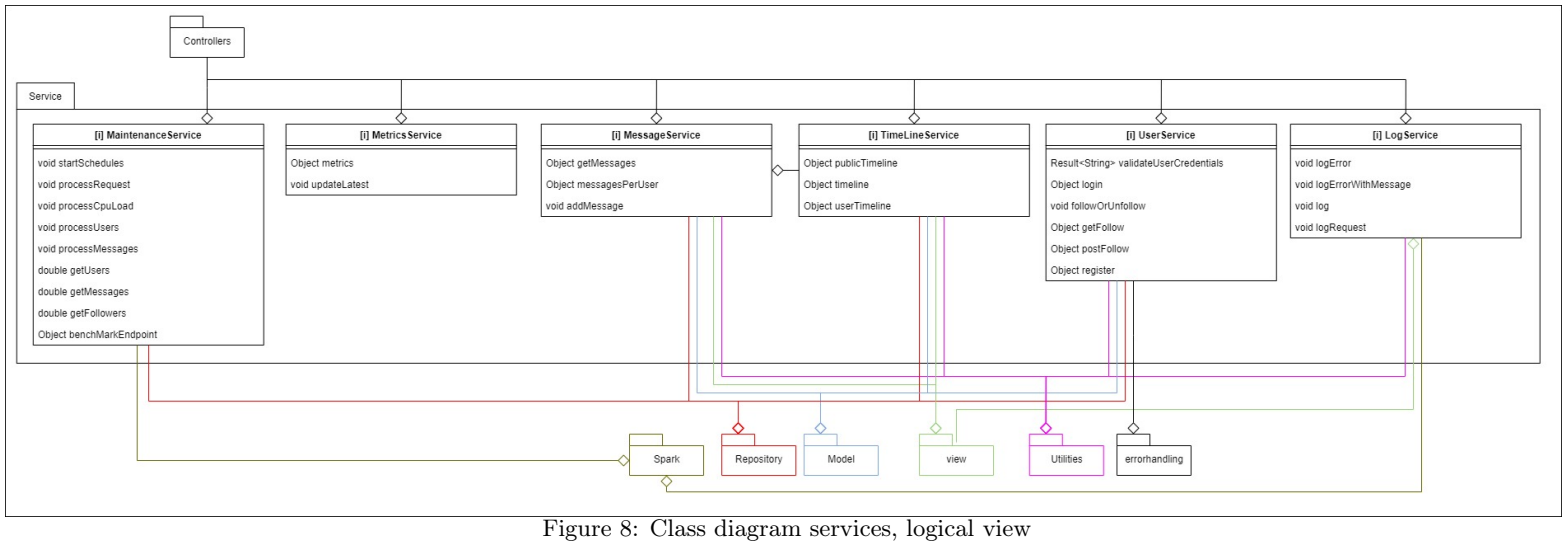


Figure 8: Class diagram services, logical view

### 1.2.7 Repository

Repository includes code for starting the database and executing queries needed by the services. Class diagram can be seen in figure 9.

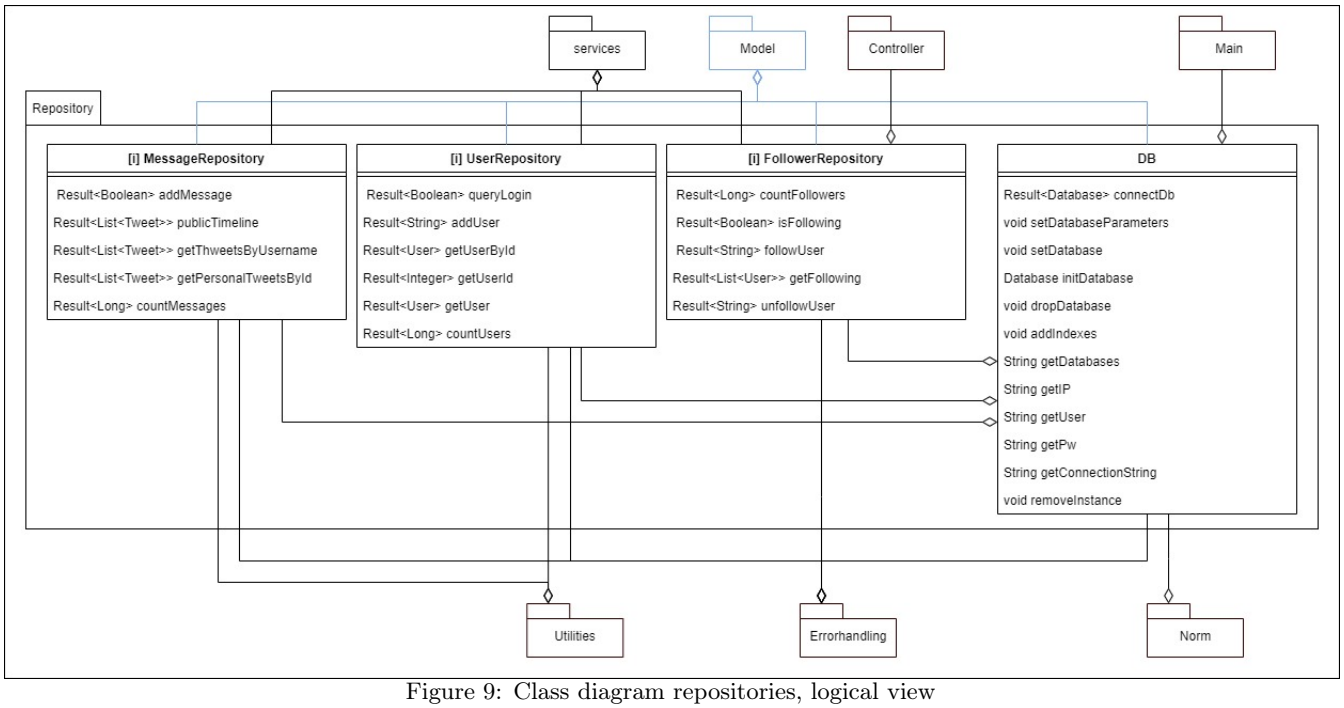


Figure 9: Class diagram repositories, logical view

### 1.2.8 Model

Model contains classes used by NORM as templates for creating database entities. Class diagram can be seen in figure 10.

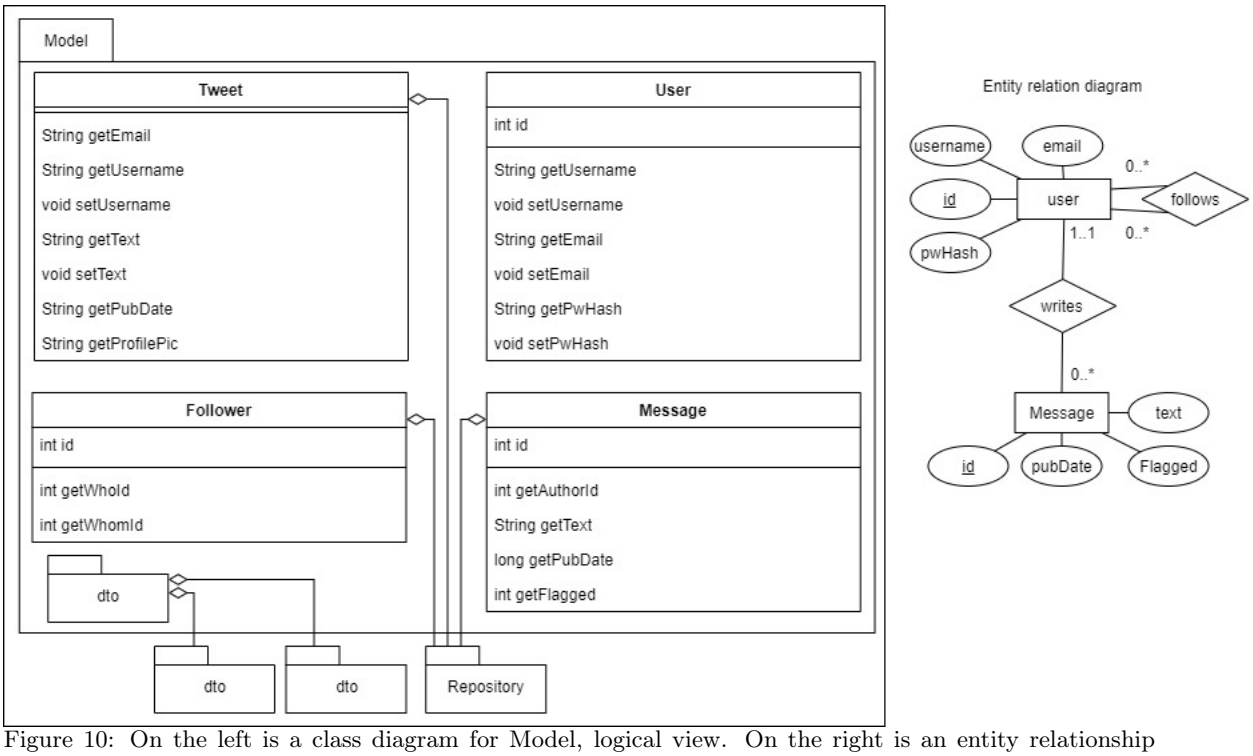


Figure 10: On the left is a class diagram for Model, logical view. On the right is an entity relationship diagram.

#### 1.2.9 Benchmark

Benchmark is not in deployment and contains code to be run locally only. It is used for measuring the speed of different repository operations and was used to argue for speedups related to extensions to the database, e.g. when indices was added.

### 1.3 Dependencies

- **Spark Java**: A micro framework for creating web applications with minimal effort.
- **jinjava**: Template engine based on django template syntax, used to render jinja templates.
- **PicoContainer**: General purpose IoC container.
- **NORM**: Lightweight ORM tool.
- **MySQL**: Relational database.
- **JUnit 5**: Most popular unit-testing framework.
- **Mockito**: Mocking framework with a simple API.
- **SLF4J**: Logging Facade for EFK stack.
- **prometheus**: Prometheus JVM Client for introducing instrumenting such as gauges, counters and other metrics.
- **org.json**: Toolkit for JSON.

#### 1.3.1 Plugins

- **Maven**: Build automation tool used to manage Java project and its dependencies.
- **PMD**: Code analyzer, used to find common programming flaws like unused variables, unnecessary object creation etc.
- **Forbidden API Checker**: Static code analysis that parses java byte code to find invocations of dangerous and deprecated method/class/field signatures.
- **SonarCloud**: Bug, Vulnerability and Code Smell detection tool with issue contextualization and remediation guidance. Used in our CI for Continuous code inspection.[2]

[2] https://github.com/DevOps2021-gb/devops2021/wiki/Static-Analysis-Tools

## 1.4 License

`www.licensediscovery.io` was used to analyze Maven dependency licenses. As seen in figure 11 the strictest license that the majority of dependencies are ruled by is the Apache Software License Version 2. This is the license that we therefore went with.[3]

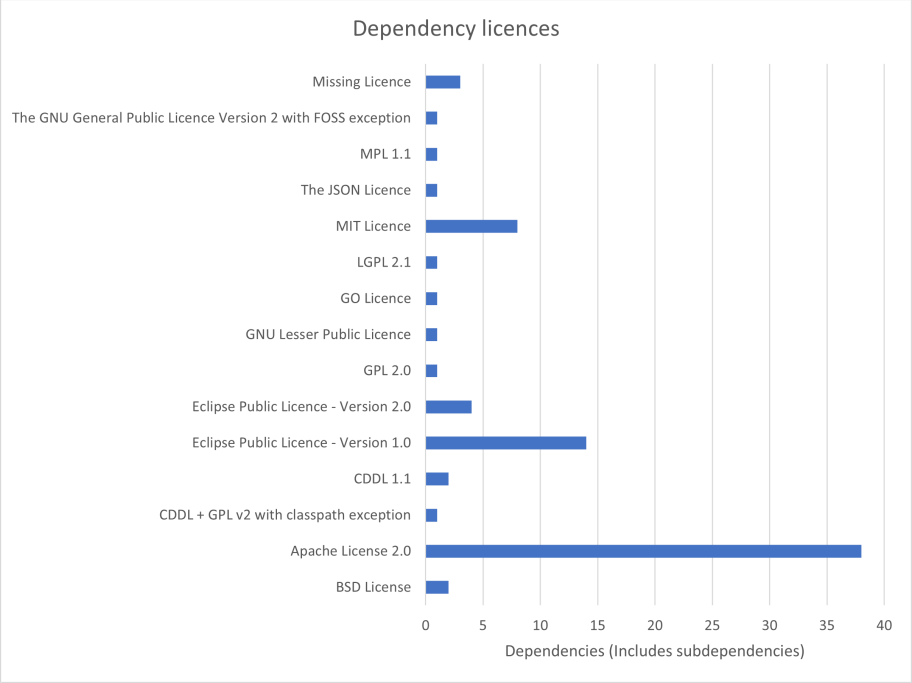


Figure 11: Overview of dependency licenses. Sub-dependencies included.

## 1.5 Important interactions of subsystem

Figure 12 shows the interaction between different parts of the system and between users and system.

Code is stored on GitHub, which developers pull and push to during development. When running the code locally, a local database should be used, called `minitwit`, which runs on port 3306 localhost with password and username "root". The repository contains a docker compose file for running the Minitwit stack locally which includes an empty MySQL database container.

GitHub actions analyze the files pushed using SonarCloud. GitHub Actions deploys to a docker image and pulls down official Prometheus, Grafana, Filebeat, ElasticSearch, Kibana and Nginx images.

The images are all deployed on two DigitalOcean droplets, where the Minitwit containers utilize a remote DigitalOcean database. One droplet is the primary server and uses a floating IP. The other droplet is regarded as a backup and regularly checks through http whether the Minitwit service is responding successfully. If not, the backup droplet obtains the floating IP until the primary is responding as expected.

The Kibana and Grafana images are for viewing monitoring and logging and the remaining four are for the EFK logging stack.

[3] `https://github.com/DevOps2021-gb/devops2021/blob/main/LICENSE.md`

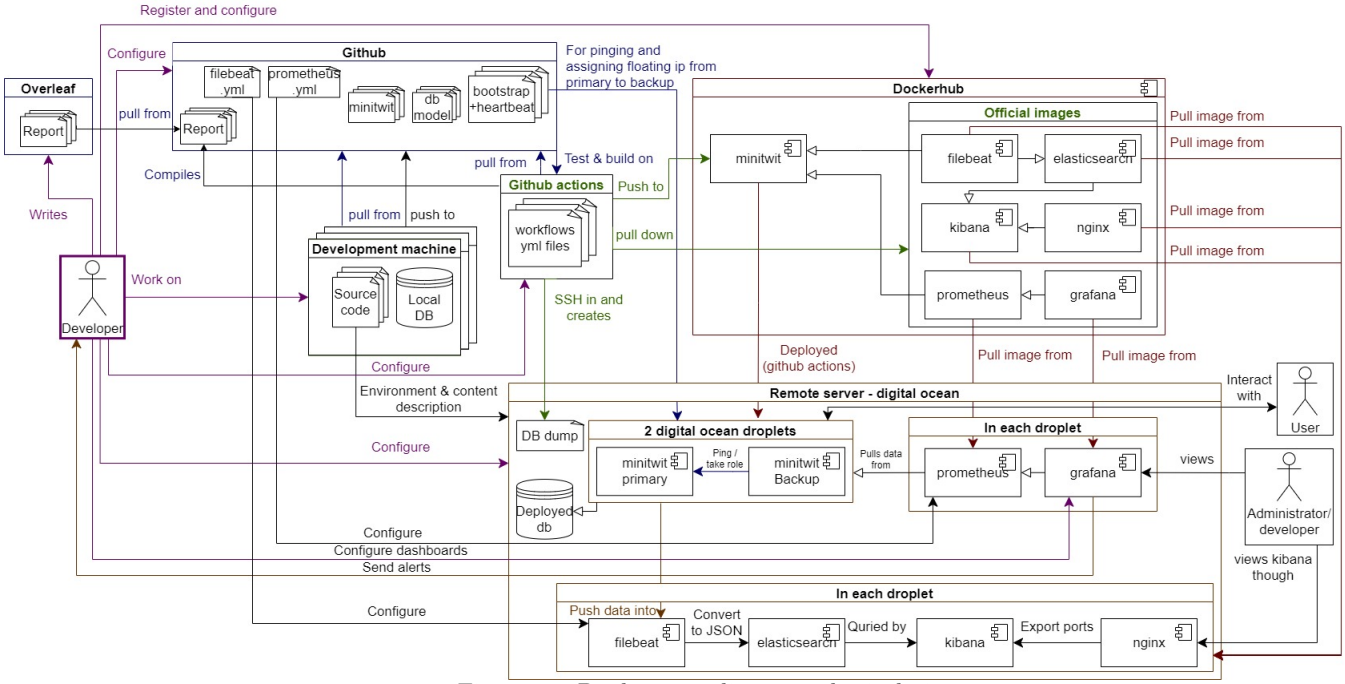


Figure 12: Deployment diagram, physical view

### 1.6 Current state of system

The system currently has a single known bug regarding the scaling of the system, see section 3.2. The main tool used for static analysis, Sonarcloud, reports the best rating, *A*, on all measures, with

- 0 bugs
- 0 code vulnerabilities and 0 security hotspots
- 6 hours technical debt due to 63 code smells
- 0.0% code duplication and 0 duplicated blocks

Out of the 63 code smells only 11 are major, critical or blockers and all 63 has been assessed as issues that can be ignored.

## 2 Process's perspective

### 2.1 Interactions and organization of developer team

The team used Microsoft Teams for communication with regular meetings every Monday and usually one or two additional "stand up" type meetings throughout the week to check in on progress. Microsoft Teams is also used for links and resources as well as communication regarding individual tasks. Major issues are typically done with everyone present, if they require important decisions, otherwise work is usually done individually or in groups of two or three people - depending on the complexity of the task at hand.

### 2.2 CI/CD chain

The initial CI/CD was set up using TravisCI, as everyone in the team had encountered it during the lectures. TravisCI was abandoned due to an overwhelming number of issues with files and folders not being added properly, in addition to secret handling being bothersome. We instead settled for using GitHub Actions, which had all the features we wanted (triggers, stages, customizable images and non-local storage of secrets).

The following workflows are set up:

- **Java CI with Maven**: Invoked on all pushes to the repository. It builds the Maven project using *mvn_package* and runs all tests, including integration tests on a remote test database. The workflow utilizes caching to ensure that Maven dependencies are not fetched on consecutive runs.

- **Staging deployment**: Invoked on all pushes to the main branch. This workflow logs into DockerHub, builds and pushes the Minitwit docker image with database secrets. Afterwards it SSHs into the primary Minitwit server and fetches the Minitwit config files, including the docker-compose file. Immediately after, all containers are torn down and rebuild using the newly pushed Minitwit image from DockerHub. Finally, a release is created with new commits linked and the backup server is redeployed similarly. In the downtime of the primary server, the backup server obtains the floating IP and vice versa.
- **Backup DB**: Once a day a GitHub action SSH into the primary server and creates a database dump with time and date in the name. Created to minimize potential data loss.
- **Build LaTeX document**: When a push to develop occurs, the LaTeX files are fetched from the report folder and a pdf document is built. The LaTeX files are manually fetched from a remote Overleaf repository, as Overleaf facilitates the teams collaborative writing.
- **SonarCloud**: Official SonarCloud GitHub action that builds and analyzes the Java project using *mvn_ verify*.

For secret handling we used GitHub's Encrypted secrets. This ensured that no potentially harmful information was exposed in the GitHub Action workflows. Among secrets were DB connection strings, DigitalOcean access tokens, DockerHub credentials and SSH keys together with host information[4].

## 2.3 Organization of repository

The code is organized using a mono repository setup, having all code and scripts necessary to run the application gathered in a single repository. The team deemed this sufficient as everything located in the repository (apart from local dockerfiles and simulator) is involved in the deployment process and the code-base is small and simple enough that having multiple repositories would only introduce unnecessary complexity

## 2.4 Applied branching strategy and development process

The main branch is used for the latest release, meaning that the code on this branch matches code found in production. The develop branch facilitated the main development of the project and should always contain a working build. From develop each team member could create feature branches (`feature/logging` for example) and then merge back into develop once tested and approved. Develop could be subject to hotfixes, which would then be merged into main. The full process can be seen in CONTRIBUTE.md.

The team used GitHub issues to track development progress, labeling them as needed with tags such as *bug*, *documentation*, *feature* and *enhancement*. The status of each issue and the issues to focus on each week was tracked using GitHub's kanban board, having columns *Todo*, *This sprint*, *In progress* and *Done*.

## 2.5 Monitoring

Monitoring is set up using Prometheus and Grafana, where Prometheus is used to extract data from the system, and the data is displayed on dashboards in Grafana.

The newest data points are collected and stored using `MaintenanceService.java`. Each data type is stored in Prometheus library's Gauge or Counter objects, which is registered with a unique name and help description. Data gathered is: the current CPU load, database information and response-time of each endpoint. The data is collected with an interval of 30 seconds. The configuration of Prometheus is found in `prometheus.yml`[5].

On *Grafana* are three dashboards:

- **Minitwit DB info** shows information about the data stored in the database, e.g. the current number of users registered for the application [6].

[4] `https://github.com/DevOps2021-gb/devops2021/wiki/CI-CD-Description:--Github-Actions`
[5] `https://github.com/DevOps2021-gb/devops2021/blob/main/prometheus.yml`
[6] `http://144.126.244.138:3000/dashboard/db/minitwit-db-info?orgId=1`

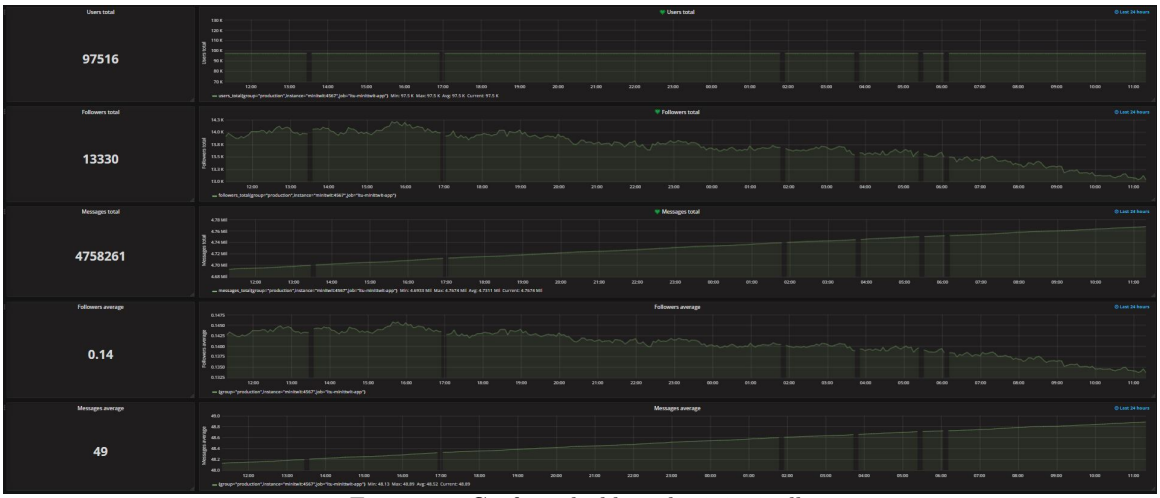


Figure 13: Grafana dashboard minitwit db

- **Minitwit request** shows response times for requests to the system.

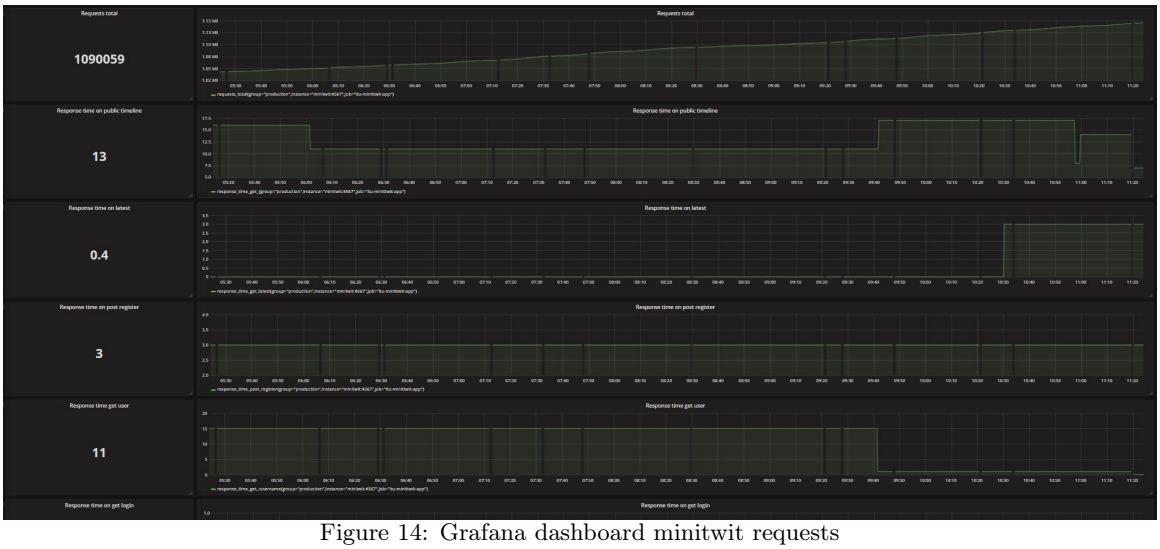


Figure 14: Grafana dashboard minitwit requests

- **Prometheus Stats** shows the system uptime.


Figure 15: Grafana dashboard Prometheus Stats

The graphs in the dashboards show minimum, maximum and average of the plotted data. The graphs with a heart in the title contains alerts that every 60 seconds check if latest value is below a threshold, indicating that a critical error has occurred. When/if such an alert is triggered Grafana should be setup to email the members of the team, see section 3.3.

### 2.6 Logging

Logging is set up as an EFK stack. Initially only exceptions were logged but during a test of the logging only one of three introduced bugs was detected, implying that the logging was insufficient. More detailed logging was introduced, now also logging requests and their payloads - except plaintext passwords, which were removed. This change made all three kinds of bugs show up in the log. The description of the experiment can be found in the wiki entry Catch a Bug By Looking at the Logs.

Logging is done using Java's Logger object and through our helper-functions each print is extended with a classification/level. The print statements also get a code to easier filter actual warnings rather than messages containing the word 'warning'. The class printing is also added to the message for traceability.

These measures allowed us to create filters for Kibana dashboards such that we could see all messages, all warnings and all of the different kinds of exceptions that could be thrown. A filebeat-elasticsearch index is created once a day for logs and are persistent through Docker volumes.

### 2.7 Security assessment

In order to assess the security of the system two tasks were undertaken. A risk assessment was conducted and penetration testing was performed. For risk assessment three assets to protect was identified: the web application, the database and the logging. The following potential threats were assessed:

- Web application: SQL injection, cross site scripting, DOS, brute forcing login and insufficient logging and monitoring.
- Database: SQL injection through web application/API and lost authentication secrets.
- Logging: DOS attacks, brute forcing login page.

A risk analysis table was constructed and 11 scenarios and seven issues were identified and graded. For most of the scenarios actions had already been taken to prevent or mitigate the risks. For a few other scenarios possible actions were identified but not taken as they were deemed unnecessary. The full details of the risks assessment can be found under Risk Assessment.

For the penetration testing three tools were used: *nmap*, *Metasploit* and *SQL map*. In each case no vulnerabilities were found. In addition to using *SQL map*, cross-site scripting was also tested manually in multiple browsers, by trying to use script-tags in the message input field on the website. In all tested browsers the script tag was not rendered. Lastly, we checked that traces of the tools used showed up in the logging. The full details of the penetration test can be seen in Penetration testing.

Another group was supposed to perform a white hat attack on the system. Nothing was heard from the group and it must be assumed that they found nothing to report.

### 2.8 Scaling and load balancing

The team chose to implement a high-availability setup in terms of a primary and a secondary droplet. First the traditional Heartbeat with Floating IPs on DigitalOcean was implemented[7]. Unfortunately this would only ensure that floating IP were switched to the backup and back again, when the entire machine would go down. We did not find a configuration of Heartbeat that would allow us to listen on a specific port, and switch floating IP's when the Minitwit service was down. We therefore decided to write two shell scripts to gain the sought after functionality. The first script[8] continuously runs on the secondary droplet checking the availability of the primary droplet, reassigning the floating-ip to itself should the application on the primary droplet go down. The second script would run on the primary droplet to reassign the floating IP to itself when it has restarted.

## 3 Lessons learned perspective

### 3.1 Issues: evolution and refactoring

Initially, the inherited code was translated to a basic Java application using the Java Spark framework[9]. At the time, this code was very bare-bones with all logic and endpoints located in a single

[7] https://www.digitalocean.com/community/tutorials/how-to-create-a-high-availability-setup-with-heartbeat-and-floating-ips-on-ubuntu-16-04

[8] https://github.com/DevOps2021-gb/devops2021/tree/main/heartbeat

[9] https://github.com/DevOps2021-gb/devops2021/issues/152

file. As time went on, the team refactored the application to make use of a more well structured system architecture (controller-, service-, persistance-layer)[10] eventually adding dependency injection and mock testing[11] to the application. During development one of the endpoint's response code was changed which led to a massive increase in errors the simulator reported, which could have been avoided if a fraction of the provided `minitwit_simulator.py` file had been a part of the test pipeline.

## 3.2 Issues: operation

The self-made heartbeat protocol had some issues related to Spark. An issue[12] was created trying to remedy this but was never finished, rendering the availability part of the system unstable as the backup would perceive the primary to be down and then reassign the floating-ip to itself exposing a single point of failure. The issue is likely related to Spark's session handling. These scripts can be found in the repository[13]

Early on in the project, the database was deployed together with the Minitwit application using Vagrant. Due to SSH key mismanagement, we lost access to our vagrant deployed DigitalOcean Minitwit server containing simulator generated data. Luckily, we were able to reset the root user through Digital Ocean's web terminal functionality. Through this terminal we could add SSH public keys to the *authorized_keys* file. Finally, we could then SSH into the server and get a database dump. This database dump was based on an outdated model with no ORM, so we had to alter the tables before transferring it to the new remote database server. Only a minor amount of data was lost during the data transfer. To minimize data loss in the future, we created a backup GitHub Action that made a database backup once a day. In addition, SSH keys for the server were backed up in a keystore.

On the 18th of March the webserver crashed due to SSL exceptions. After looking through Docker logs, we narrowed it down to an incompatibility between java-mysql-connector 8.0.15 and Java 11. The solution to this was to update java-mysql-connector 8.0.15 to 8.0.23 [14].

## 3.3 Issues: maintenance

The team never figured out how to make SonarCloud read the test coverage[15], eventually scrapping the idea after spending much time trying to make it work. It should, however, be said that it is able to identify which test functions are proper test functions and that code coverage is obtainable through an IDE like IntelliJ.

Furthermore, the team tried to implement alert notifications on Grafana[16] in order to be notified of unexpected data from the monitoring, but this was never successfully implemented.

As the only manual part of the release process, the team had to rename the previous release from 'development release' to the correct tag/version. An issue[17] was created in an attempt to automate the entire release process and have an internal counter of the current version, but was never solved.

## 3.4 Reflection on DevOps style of work

The automatic and continuous deployment was very nice to work with since it made it possible for all team members to deploy code without understanding some complicated set of steps to go through. In earlier projects team members have experienced cases where only few group members knew how to deploy.

The idea of having containers, which should enable all members to run the code regardless of tools and dependencies was nice, but the team experienced issues with this resulting in all members still not being able to run the application locally. The issues occurred after adding the EFK stack.

Regarding the way of working together, structuring the repository and branching, this way was not new to any group members but the technique worked well. Some of the techniques group members had used were: making flow visible, limiting work in process, reducing batch/issue sizes and the different ways of eliminating waste in the value stream. In regards to the second way, mobilizing mix of experts and people needing the knowledge has been common practice, and requiring approval

[10] https://github.com/DevOps2021-gb/devops2021/tree/ae6cd48a06fde2a5c403056b33895b7589cb039d
[11] https://github.com/DevOps2021-gb/devops2021/commit/07914442a09e204f0be2687fe8e5b9bd07799ee4
[12] https://github.com/DevOps2021-gb/devops2021/issues/171
[13] https://github.com/DevOps2021-gb/devops2021/tree/main/heartbeat
[14] https://github.com/DevOps2021-gb/devops2021/issues/94
[15] https://github.com/DevOps2021-gb/devops2021/issues/140
[16] https://github.com/DevOps2021-gb/devops2021/issues/87
[17] https://github.com/DevOps2021-gb/devops2021/issues/170

near source and experts was common practice. As this is a student project the third ways of creating high-trust culture came naturally.

Adding tests to CI was effective as it removed assumptions and increased resilience of anything committed - and multiple times stopped broken code, which otherwise would have been approved. The logging and maintenance were helpful to provide feedback, allowing us to see the problems as they occurred.