

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	3
1.1	Design	3
1.2	Architecture	3
1.2.1	Main	4
1.2.2	Utilities	4
1.2.3	Error handling	5
1.2.4	Controllers	5
1.2.5	View	5
1.2.6	Services	6
1.2.7	Repository	6
1.2.8	Model	7
1.2.9	Benchmark	7
1.3	Dependencies	8
1.3.1	Plugins	8
1.4	Important interactions of subsystem	9
1.5	Current state of system	10
2	Process's perspective	11
2.1	Interactions and organization of developer team	11
2.2	CI/CD chain	11
2.3	Organization of repository	12
2.4	Applied branching strategy and development process	12
2.5	Monitoring	12
2.6	Logging	14
2.7	Security assessment	14
2.8	Scaling and load balancing	15
3	Lessons learned perspective	16
3.1	Issues: evolution and refactoring	16
3.2	Issues: operation	16
3.3	Issues: maintenance	17

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 using Java with Spark 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 similar to Flask which was used for the inherited Minitwit system and since it has a reasonable learning curve and is straight forward to use for an application of the size of Minitwit. The frontend consist of html templates connected with the backend using the template engine Jinjava, chosen to mimic the python Minitwit with minimal effort.

MySQL was chosen for database since the relational model fits the system data well. **Why is it better than eg SQLite?**

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 for Minitwit.

Hibernate was implemented as a test to see if it would be a better fit since it is more complex but it turned out that the tool did not deliver on its promises, meaning that it only introduced more unnecessary complexity without adding any benefits. It furthermore complicated database joins that are needed for Minitwit. We decided to stick with NORM.

The application is hosted on DigitalOcean. Measures taken to ensure availability and for maintenance purposes, monitoring with Prometheus and Grafana and logging using an ELK-stack, are described in further sections.

1.2 Architecture

Figure 1 shows the different components of the application.

gå igennem alle ugers "afleveringer" og tjek vi har argumenteret for alt

Indexes?

Skal vi have design af db tables? eventuelt som et billede/screenshot? Ja det ville være godt med en kort beskrivelse af tables og model

TODO: MISSING TEXT ON EACH CONNECTOR.

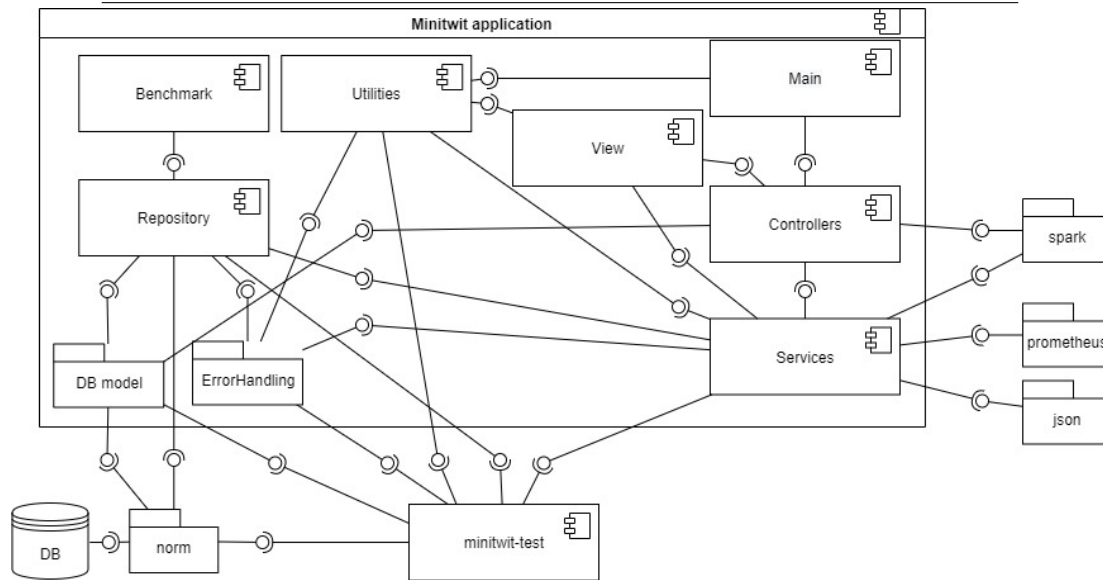


Figure 1: Component diagram, development view

1.2.1 Main

Main is the system's entrance point and allows specification of which database to use. Main starts monitoring and logging services and defines the number of threads to use. Figure 2 presents a class diagram.

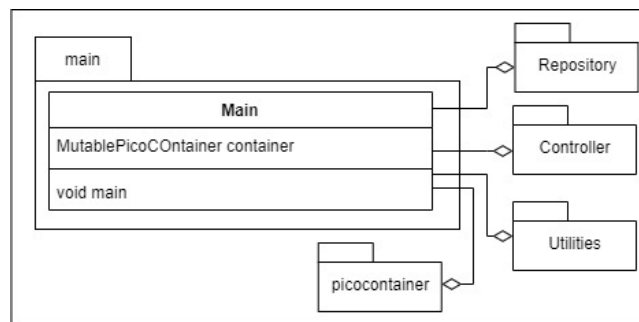


Figure 2: Class diagram main, logical view

1.2.2 Utilities

Utilities, shown in figure 3, contains helper methods like hashing, formatting dates and JSON, and houses Request, Response and Session wrapper objects to use with Spark.

stemmer det
her diagram
overens med
connectorne
i figur 1
egentlig?

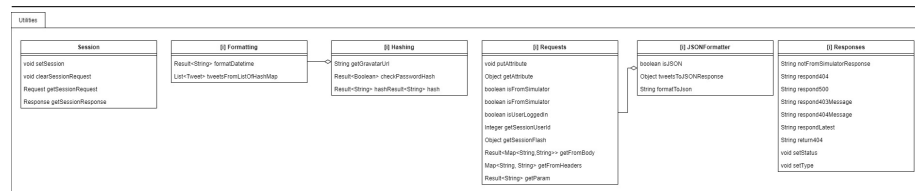


Figure 3: Class diagram utilities, logical view

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 4.

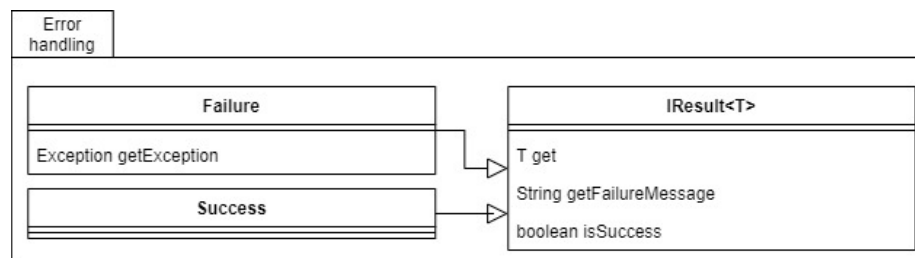


Figure 4: Class diagram errorhandling, logical view

1.2.4 Controllers

Controllers handles the endpoints the application provides by mapping post and get request to appropriate methods. The class diagram can be seen in figure 5.

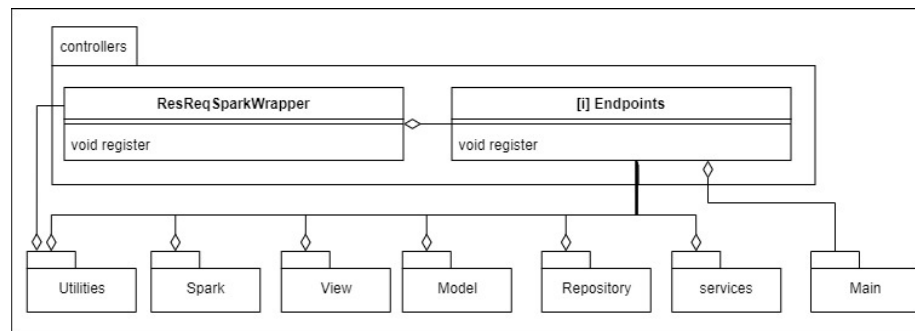


Figure 5: Class diagram controllers, logical view

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 6.

Eventuelt flyt de tre sidste klasser ned på en ny linje, så billedet kan gøres bredere = teksten nemmere at læse

Er det her ikke de forkerte metoder i diagrammet? Skal det ikke være renderTemplate og redirect?

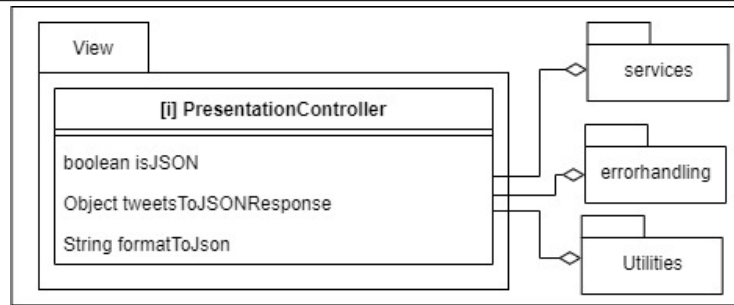


Figure 6: Class diagram view, logical view

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 7.

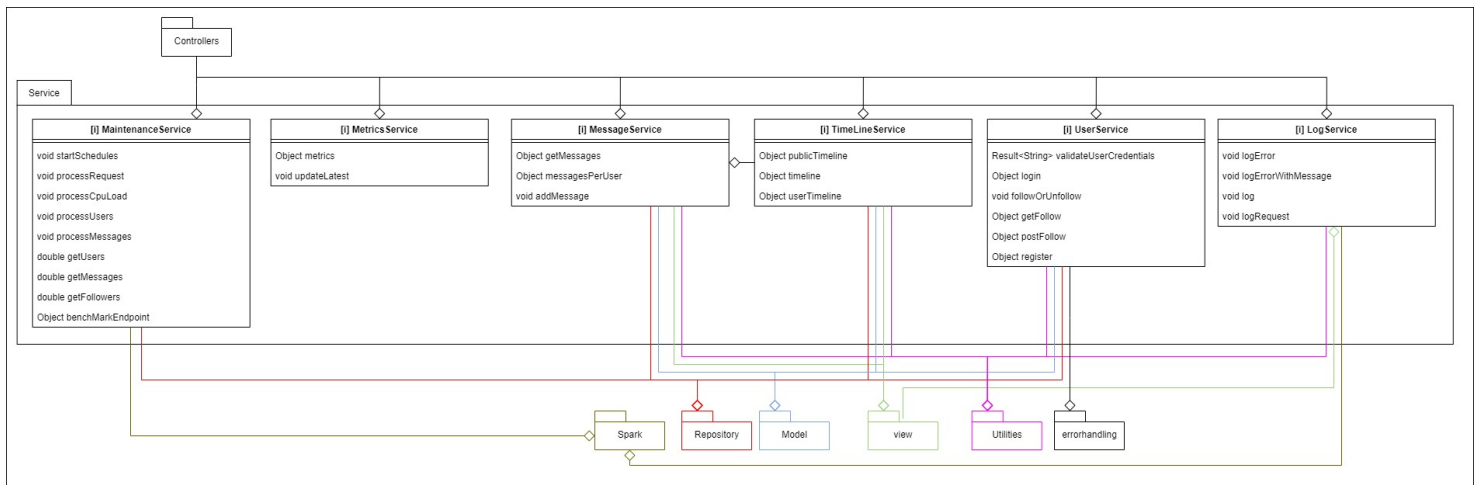


Figure 7: Class diagram services, logical view

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 8.

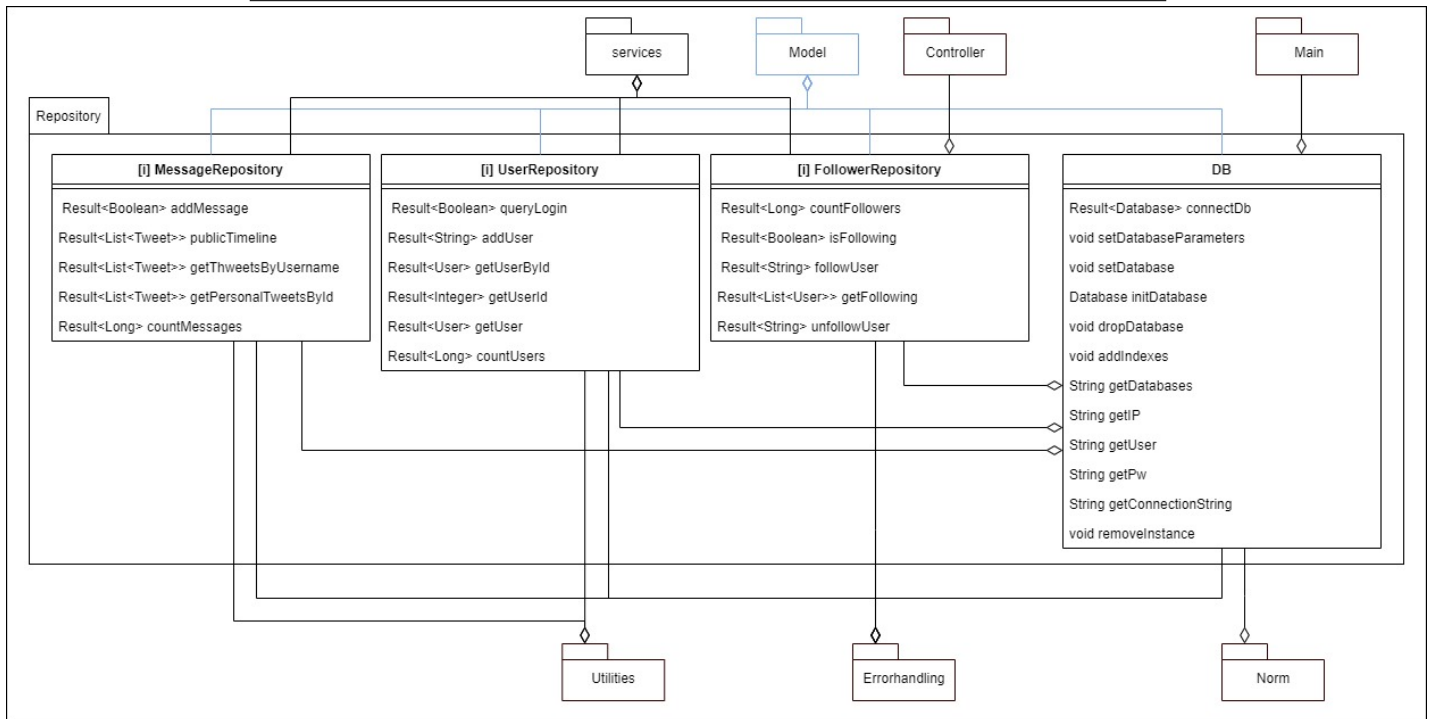


Figure 8: Class diagram repositories, logical view

1.2.8 Model

Model contains classes used by NORM as templates for creating database entities. Class diagram can be seen in figure 9.

forkert diagram

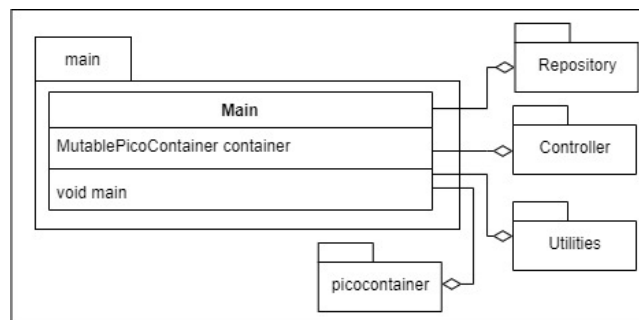


Figure 9: Class diagram model, logical view

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 indexes 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 ELK 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/fields signatures.
- **SonarCloud**: Bug, Vulnerability and Code Smell detection tool with issue contextualization and remediation guidance. Used in our CI for Continuous code inspection.¹

www.licensediscovery.io was used to analyze Maven dependency licenses. As seen in figure 10 the strictest license that the majority of dependencies is ruled by is the Apache Software License Version 2. This is the license that we therefore went with.²

¹<https://github.com/DevOps2021-gb/devops2021/wiki/Static-Analysis-Tools>

²<https://github.com/DevOps2021-gb/devops2021/blob/main/LICENSE.md>

Er det begrundelse nok? kan ikke komme på mere at sige om det i hvert fald

ryk billede i bilag eller til repository, det er umuligt at se hvad der står

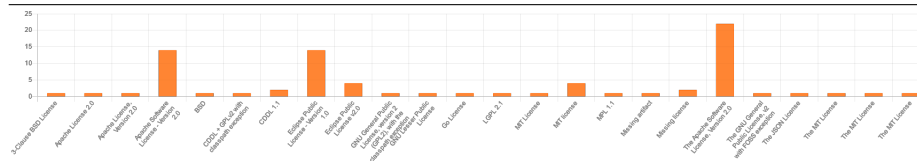


Figure 10: Overview of dependency licenses. Sub-dependencies included.

1.4 Important interactions of subsystem

Figure 11 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 analyzes 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 maintenance and logging and the remaining four are for the ELK logging stack.

skal vi have deployment diagram???

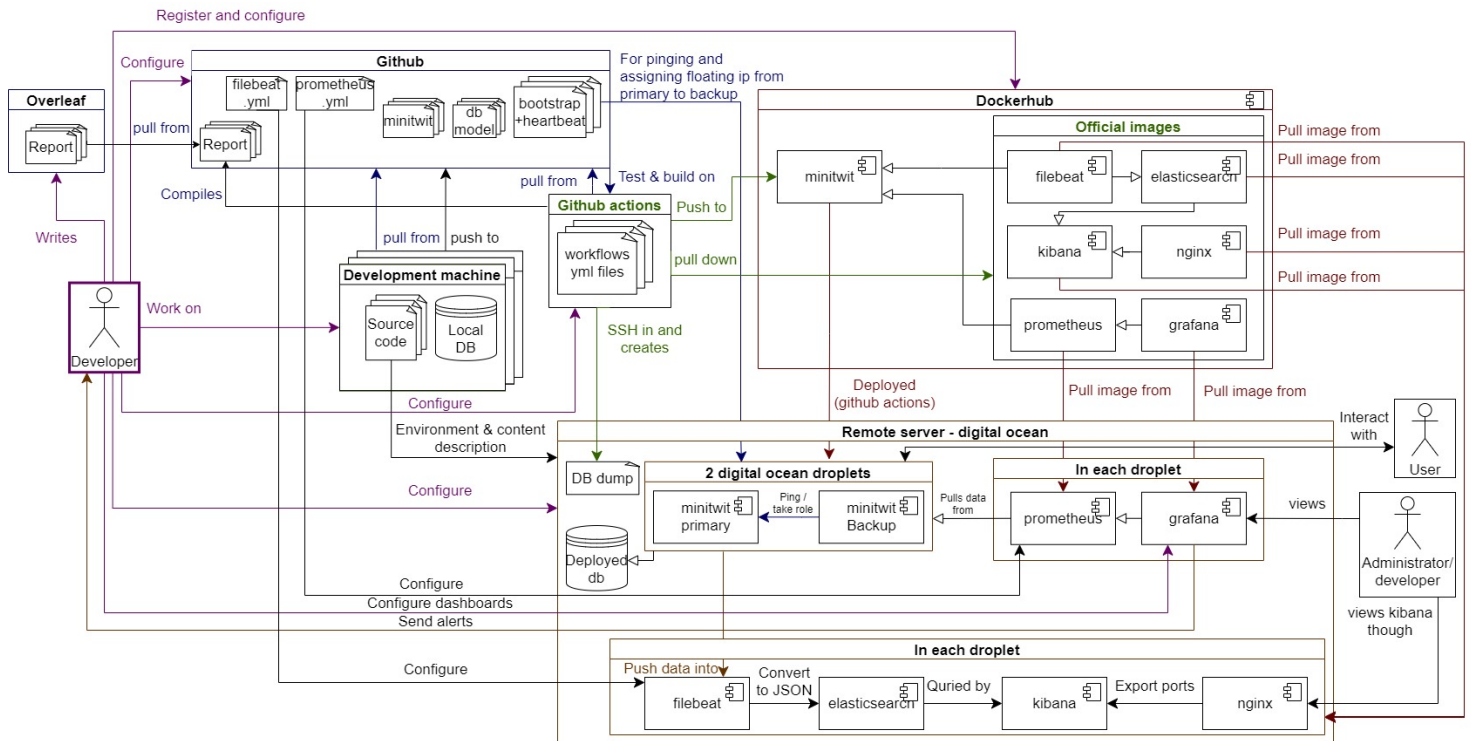


Figure 11: Communication diagram, process view

1.5 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 ignored.

2 Process's perspective

2.1 Interactions and organization of developer team

The developer team stays in contact through a Microsoft Teams 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 ahead.

2.2 CI/CD chain

For CI, we initially attempted to use TravisCI, as everyone in the group had encountered it. 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 *mvnpackage* 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 to 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 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 groups collaborative writing.

Also reflect and describe what was the "DevOps" style of your work. For example, what did you do differently to previous development projects and how did it work?

- **SonarCloud:** Official SonarCloud GitHub action that builds and analyzes the Java project using *mvn_verify*.

For secret handling we used Githubs 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.

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 increase 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 group member could create feature branches (*feature/logging* for example) and then merge back into develop once tested and approved. Develop could be subject to hot-fixes, which would then be merged into main. The full process can be seen in [CONTRIBUTE.md](#).

The group 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 subtract data from the system, at 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`³.

On *Grafana* are three dashboards:

³<https://github.com/DevOps2021-gb/devops2021/blob/main/prometheus.yml>

- **Minitwit DB info** shows information about the data stored in the database, e.g. the current number of users registered for the application ⁴.



Figure 12: Grafana dashboard minitwit db

- **Minitwit request**⁵ shows response times for requests to the system.

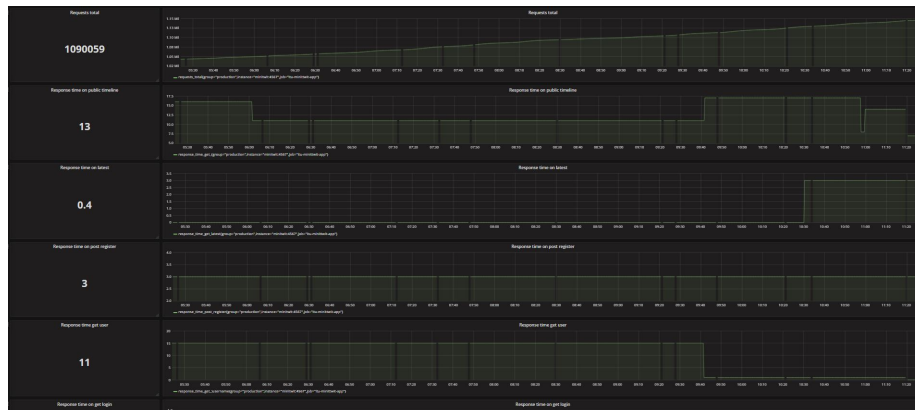


Figure 13: Grafana dashboard minitwit requests

- **Prometheus Stats**⁶ shows the uptime of the system. The time is reset when the system is redeployed or if it goes down.

⁴<http://144.126.244.138:3000/dashboard/db/minitwit-db-info?orgId=1>

⁵<http://144.126.244.138:3000/dashboard/db/minitwit-requests?orgId=1>

⁶<http://144.126.244.138:3000/dashboard/db/prometheus-stats?orgId=1>



Figure 14: 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 with the exception of passwords which was removed. This change made all three kinds of bug 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](#).

hashes eller
rigtige?

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 Grafana dashboards such that we could see all messages, all warnings and all of the different kinds of exceptions that could be thrown.

log volumes,
traces, ex-
ceptions

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 not deemed necessary. 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, it was checked that traces of the tools run 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 group first implemented the traditional Heartbeat with Floating IPs on DigitalOcean ⁷. Unfortunately this would only ensure that floating IP were switched to the backup and back 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 script⁸ 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.

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>

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 framework⁹. At the time, this code was very bare-bones with all logic and endpoints located in a single file. As time went on, the team refactored the application to make use of a more well structured system architecture (controller-, service-, persistence-layer)¹⁰ eventually adding dependency injection and mock testing¹¹ to the application. During development one of the endpoint's response codes 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 issue¹² 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.

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`¹³.

⁹<https://github.com/DevOps2021-gb/devops2021/issues/152>

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/issues/94>

no auto-
matic ver-
sioning -
ved ikke om
det er nemt
specifikt for
java 11

3.3 Issues: maintenance

The group never figured out how to make SonarCloud read the test coverage¹⁴, 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 group tried to implement alert notifications on Grafana¹⁵ in order to be notified of unexpected data from the monitoring but this was never successfully implemented.

¹⁴<https://github.com/DevOps2021-gb/devops2021/issues/140>

¹⁵<https://github.com/DevOps2021-gb/devops2021/issues/87>