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

Course: DevOps, Software Evolution and Software Maintenance, MSc CS (Spring 2021)
KSDSESM1KU

Contents

1 System's perspective

argue for the choice of technologies and decisions for at least all cases for which we asked you to do so in the tasks at the end of each session.

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?.

Indexes?

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 the Minitwit application.

Skal vi have design af db tables? eventuelt som et billede/screenshot?

Hibernate was implemented as a test to see if it would be a better fit than NORM 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 in the Minitwit application. It was therefore decided to stick with NORM.

The application is hosted on DigitalOcean and measures taken in terms of scaling and ensuring availability is described in section 2.8. Monitoring and logging is described in section 2.5 and 2.6, respectively.

1.2 Architecture

Figure 1 shows the different components of the application.

TODO: MISSING TEXT ON EACH CONNECTOR.

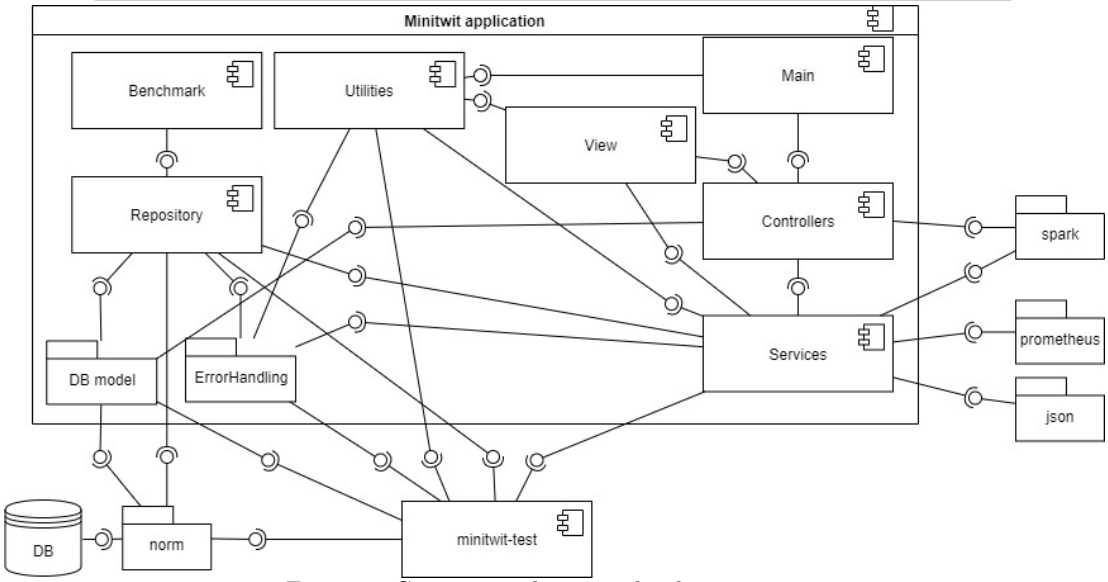

Figure 1: Component diagram, development view

1.2.1 Main

Main is the entrance point into the program and contains arguments to modify which database is used. Main also starts the maintenance and defines the max number of threads for the program.
The class diagram can be seen in figure 2.

hvad er maintenance for læseren?

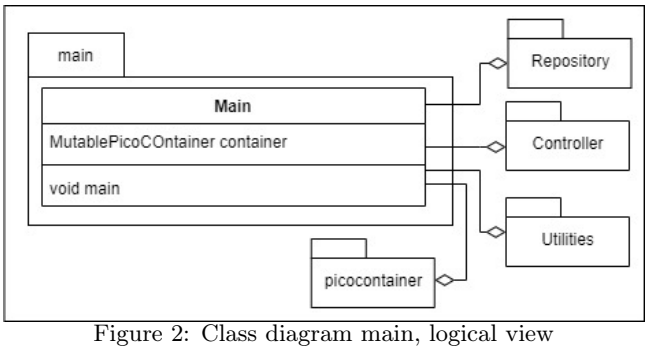

Figure 2: Class diagram main, logical view

1.2.2 Utilities

Utilities contains helper methods like hashing, formatting dates and JSON, as well as housing Request, Response and Session wrapper objects.
The class diagram can be seen in figure 3.

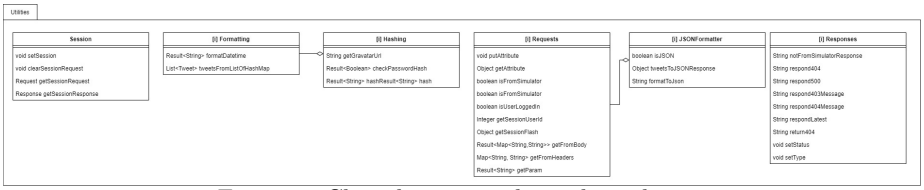

Figure 3: Class diagram utilities, logical view

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

1.2.3 Error handling

Error handling contains objects to hold results of computations which could result in exceptions, allowing for better error handling and reducing amounts of try-catch statements.
The class diagram can be seen in figure 4.

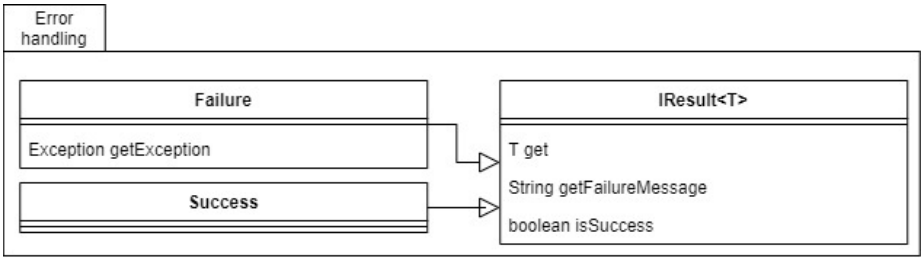

Figure 4: Class diagram errorhandling, logical view

1.2.4 Controllers

Controllers Handles the different endpoints the application provides by mapping paths for post and get request to methods handling the requests and responses.
The class diagram can be seen in figure 5.

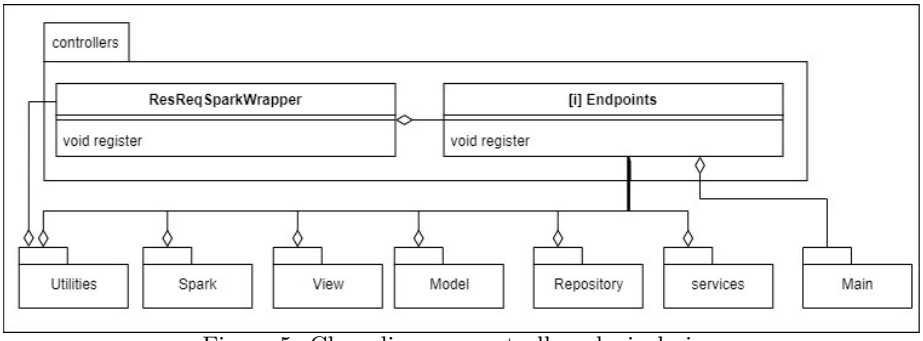

Figure 5: Class diagram controllers, logical view

1.2.5 View

View contains code for rendering the website by redirecting to new paths or specifying HTML templates to be used along with what content to show.
The class diagram can be seen in figure 6.

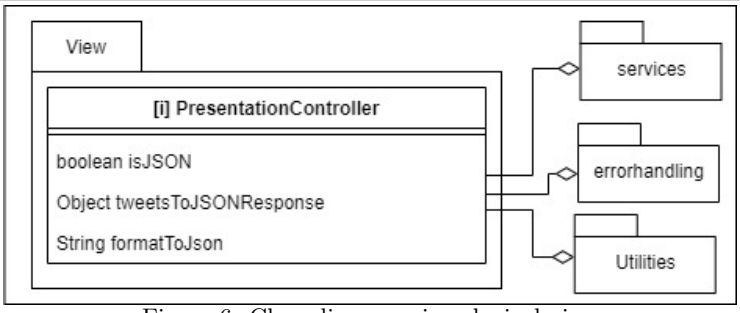

Figure 6: Class diagram view, logical view

1.2.6 Services

Services includes the code for maintenance, which uses Prometheus library to store the values Prometheus should collect, and the log service, which prints in a manner easy to filter for in Kibana. Services also includes functions executing the requests and responses.
The class diagram can be seen in figure 7.

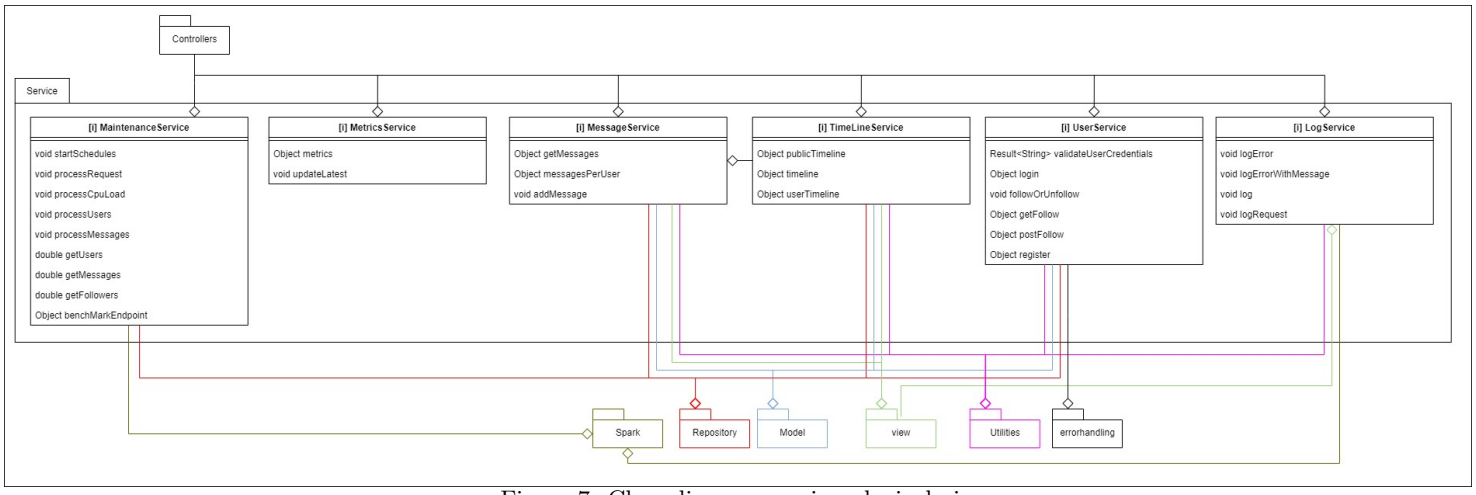

Figure 7: Class diagram services, logical view

1.2.7 Repository

Repository includes code for starting the database and for executing queries needed by the services. The class diagram can be seen in figure 8.

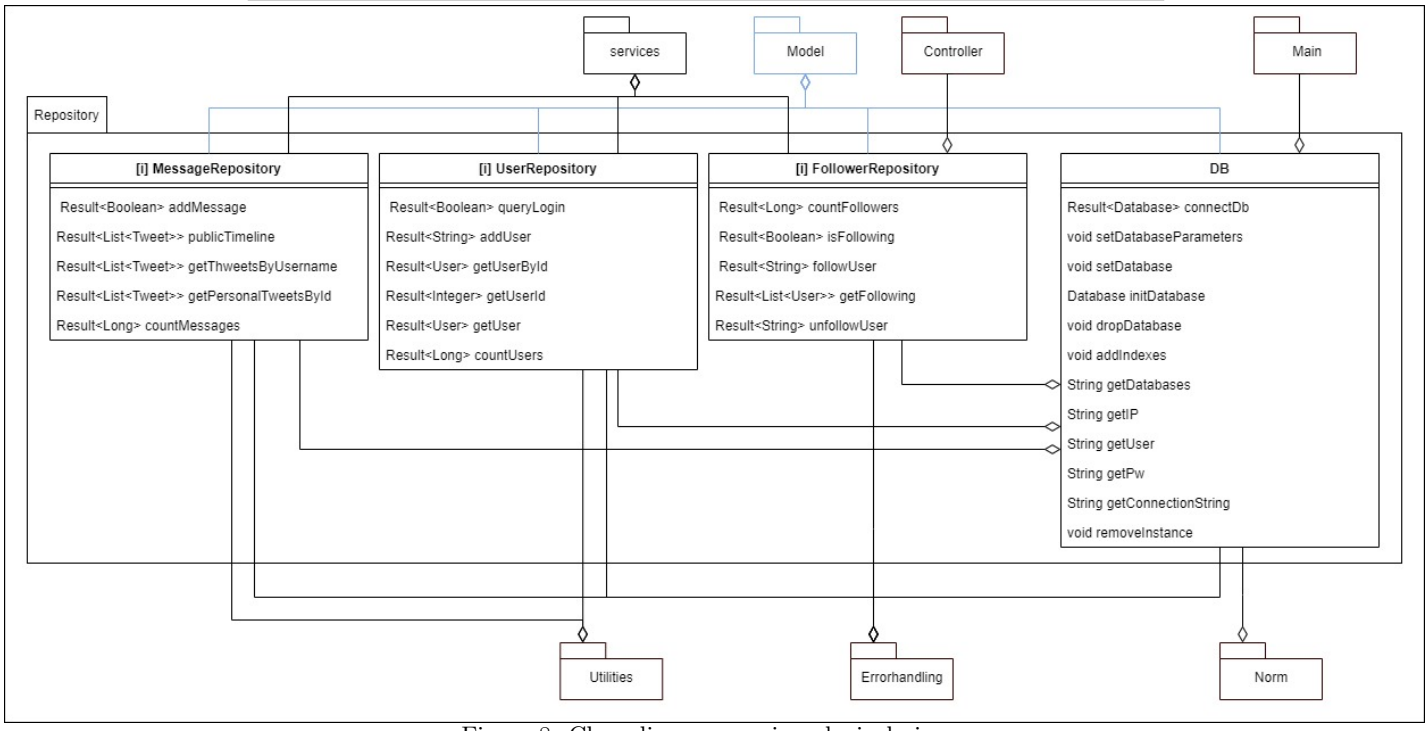

Figure 8: Class diagram services, logical view

1.2.8 Model

Model contains the database relations with one object per entity. The objects in the folder matches the entities in the database as they are the template NORM uses to create them.
The class diagram can be seen in figure 9.

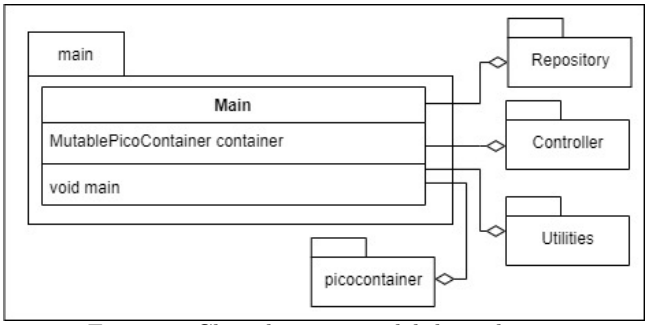

Figure 9: Class diagram model, logical view

1.2.9 Benchmark

This is not in deployment and contains code to be run locally only. The packet component relates to measuring speed of different operations in the repository

and was used to argue for speedups related to any extension to a database of same size as the one in deployment, as when indexes was added.

1.3 Dependencies

- **Spark Java**: A micro framework for creating web applications with minimal effort. Suitable for the project size and gave us the freedom to structure the application how we wanted it.
- **jinjava**: Template engine based on django template syntax, used to render jinja templates.
- **PicoContainer**: General purpose IoC container.
- **NORM**: Lightweight ORM that doesn't require the complex markdown files that Hibernate, JPA etc. introduce. Suitable for smaller projects with simple models.
- **MySQL**: Relational database. Chosen based on familiarity to team members and it's renowned reliability for how long it's been around.
- **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.

Argumentere for ORM her eller i design? - Gør det i design, her skal vi bare vise vores deps

1.3.1 Plugins

- **Maven**: Build automation tool used to manage our 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 with issue contextualization and remediation guidance. Used in our CI for Continuous code inspection.

1

We used `www.licensediscovery.io` to analyze our 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]

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

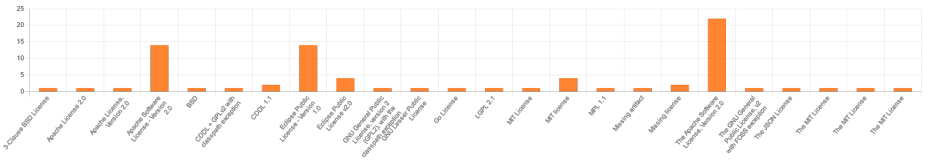

Figure 10: Overview of dependency licenses. Sub-dependencies included.

1.4 Important interactions of subsystem

Figure 11 shows the communication between different parts of the system and between users and system.
Code is stored in Github, which developers pull and push to during development. As mentioned main contains code for which database to use, and when running just the code locally it should use a local database called `minitwit` running on port 3306 localhost with password and username "root". When running locally with docker compose the MySQL services should be stopped so a container for the database can be created.
Github actions test the files pushed using SonarCloud. Github actions also deploy to a docker image and pull down official Prometheus, Grafana, Filebeat, ElasticSearch, Kibana and Nginx images.
The images are deployed on a Digital Ocean server containing one database and two digital ocean droplets. Each droplet contain a minitwit application and an official image. One image's floating point IP is in production and the other use heartbeat to test whether primary is still up or whether it should become the new primary.

Heartbeat

The Prometheus and Grafana images are for viewing maintenance and the remaining four are for logging purposes.

[1] `https://github.com/DevOps2021-gb/devops2021/wiki/Static-Analysis-Tools`
[2] `https://github.com/DevOps2021-gb/devops2021/blob/main/LICENSE.md`

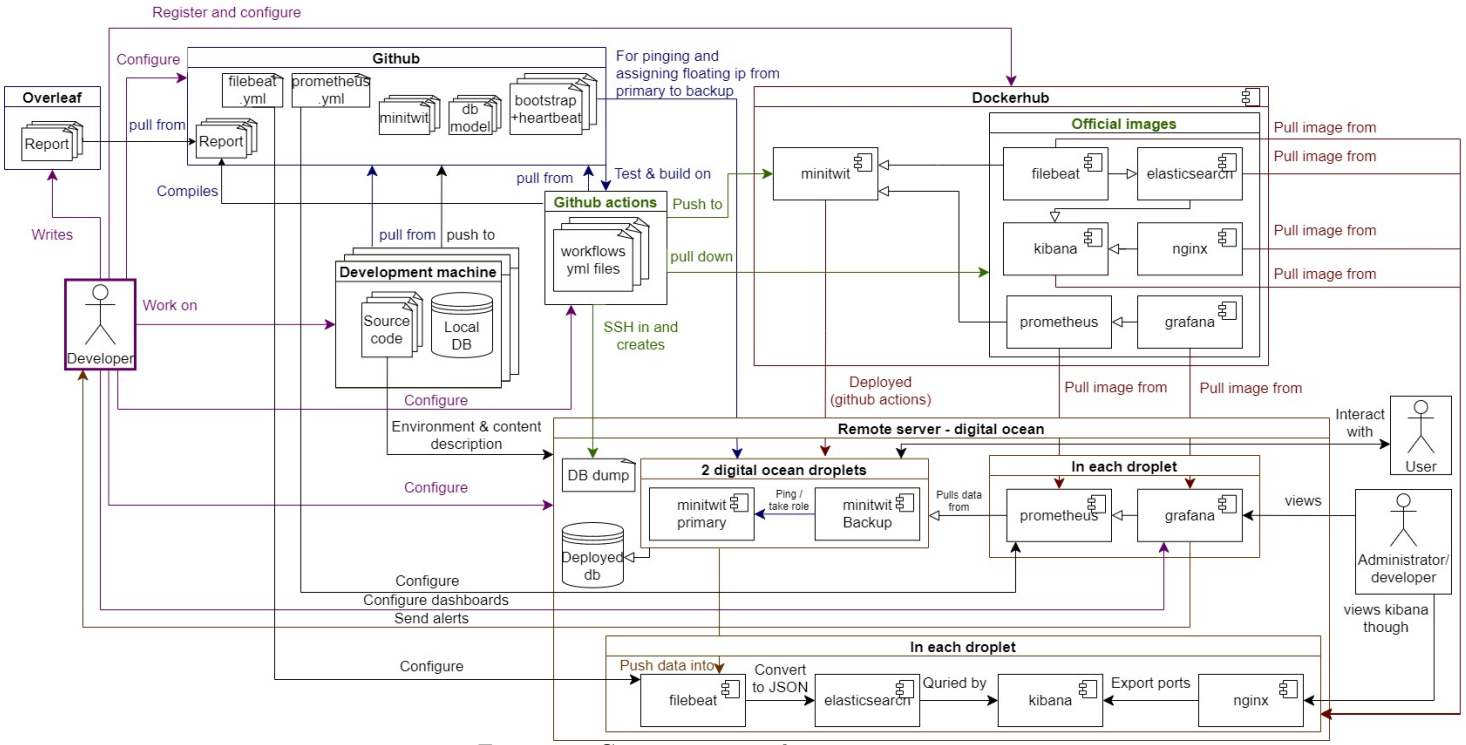

Figure 11: Communication diagram, process view

1.5 Current state of system

The system is currently has a single known bug regarding the scaling of the system, see section 3.2. The 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 code 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

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

2.1 Interactions and organization of developer team

The developer team stay in contact through a Microsoft Teams team, where we have regular meetings every Monday and usually one or two additional "stand up" type meetings to check in on progress throughout the week. The Teams chat is also used for some links and logins, 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 or 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. We ended up abandoning TravisCI due to an overwhelming number of issues with files and folders not being added properly, in addition to not being able to make proper use of necessary secrets.

We instead settled for using GitHub Actions, which had all the features we wanted (triggers, stages, customizable images and non-local storage of secrets).

- **Java CI with Maven**: This is invoked on all pushes to the github repository. It builds the maven project using *mvnpackage* and runs all tests, including integration tests on a remote test database. This workflow utilizes caching to ensure that maven dependencies aren't fetched on consecutive runs.
- **Staging deployment**: This is 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 SSH's 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 and a pdf document will be built when changes are pushed to the report directory in the repository. The LaTeX files are manually fetched from a remote Overleaf repository, as Overleaf facilitates the groups collaborative writing.

isn't this contradicting?

- **SonarCloud**: Official SonarCloud github action that builds and analyzes the java project using *mvn_ verify*.

describe secret handling

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 codebase is small and simple enough that having multiple repositories would only increase complexity.

2.4 Applied branching strategy and development process

The group decided on using the master branch for the latest release meaning that the code found in production, could also be found on the master branch. The develop branch facilitated the main development of the project which 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 hotfixes, which would then be merged into master when issues arose. 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, enhancement and so on. 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 displayed collected by Prometheus 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, data base information and response-time of each endpoint. The data is collected with an interval of 30s by it calling the "/metrics" endpoint. The configuration of Prometheus can be seen in file `prometheus.yml` [3].

On *Grafana* three dashboards were created:

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

[3] `https://github.com/DevOps2021-gb/devops2021/blob/main/prometheus.yml`
[4] `http://144.126.244.138:3000/dashboard/db/minitwit-db-info?orgId=1`

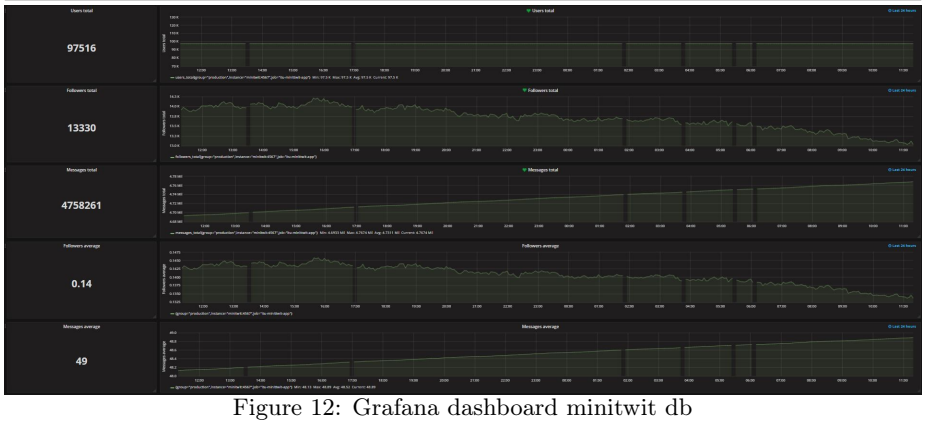

Figure 12: Grafana dashboard minitwit db

- **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.

Figure 14: Grafana dashboard Prometheus Stats

[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

The graphs in the dashboards show min, max and average of the platted data and those with a heart in the title contains alerts that every 60 seconds check if latest value is below certain values which indicates that a critical error has occurred. When/if such an alert is triggered Grafana should be setup to email the members of the team.

2.6 Logging

Logging is set up as an EFK stack. Initially only exceptions were logged but during the logging exercise we found only one of the three introduced bugs could by detected by our logging system, telling us our logging was insufficient. Afterwards 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. Logging is done using Java's Logger object and though our helper-functions each print is extended with a classification/level. The prints also gets an code to easier filter only actual warnings instead of also messages containing the word warning. The class printing is also added to the message to easily track down which class caused that message.
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, exceptions

2.7 Security assessment

In order to assess the security of the system two tasks were undertaken. A risk assessment was conducted and a penetration testing was performed.
For the rick 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 7 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 or tested another group by mistake.

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 script to gain the sought after functionality. The first script[8] continuously running 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 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 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 endpoints the response code was changed which led to a massive increase in errors the simulator reported, this could have been avoided if a fraction of the provided `minitwit_simulator.py` file had been a part of the test pipeline.

[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

[10] https://github.com/DevOps2021-gb/devops2021/tree/ae6cd48a06fde2a5c403056b33895b7589cb039d

[11] https://github.com/DevOps2021-gb/devops2021/commit/07914442a09e204f0be2687fe8e5b9bd07799ee4

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 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 our MySQL database was deployed together with our 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 then 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 ensure minor 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 our 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].

no automatic versioning

3.3 Issues: maintenance

The group never figured out how make SonarCloud read the test coverage[14], 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 though Intellij.

Furthermore, the group tried to implement alert notifications on Grafana[15] in order to be notified of unexpected data from the monitoring but this was never successfully implemented.

[12] https://github.com/DevOps2021-gb/devops2021/issues/171
[13] https://github.com/DevOps2021-gb/devops2021/issues/94
[14] https://github.com/DevOps2021-gb/devops2021/issues/140
[15] https://github.com/DevOps2021-gb/devops2021/issues/87