

# Minitwit Documentation

DevOps, Software Evolution and Software Maintenance - BSc

Course code: BSDSESM1KU

IT University of Copenhagen

June 1st, 2022

## Authors:

Leonora Andrea Lindqvist Nielsen ([lann@itu.dk](mailto:lann@itu.dk))

Ahmet Eren Tomurcuk ([ahto@itu.dk](mailto:ahto@itu.dk))

Smilla Maria Dion ([smdi@itu.dk](mailto:smdi@itu.dk))

Simon Amundsen ([siam@itu.dk](mailto:siam@itu.dk))

Lucas Heede Vulpius ([luvu@itu.dk](mailto:luvu@itu.dk))

## System Perspective

This section will illustrate the current state of the Minitwit system through various diagrams and descriptions.

## Architecture

The Minitwit application is built on a client-server architecture. When connecting to Minitwit through a browser, users will interact with a web server. This web server stores all information about its users in a [PostgreSQL](#) database.

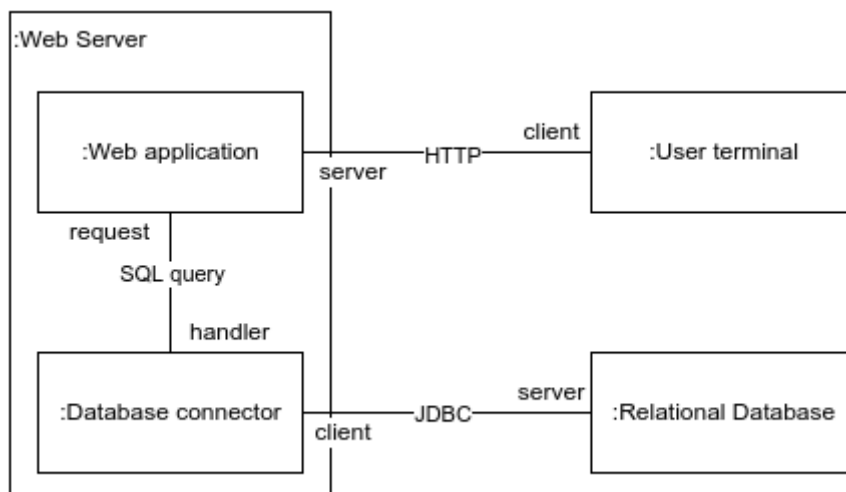


Figure 1. An overview of the runtime interaction in the system.

The web application consists of two Java classes: *WebApplication* and *SqlDatabase*. The first encapsulates the entire Minitwit application and receives user requests, while the latter handles connections to the database initiated by *WebApplication*. Essentially, we kept the simple structure

from the original Python application. At the time of writing, we have been working on splitting the *WebApplication* class into multiple classes to improve cleanliness and maintainability. As an illustration, we have moved logic related to user messages into its own class in [this pull request](#). This process is not done however, which is why we present the web application as a single class in this section.

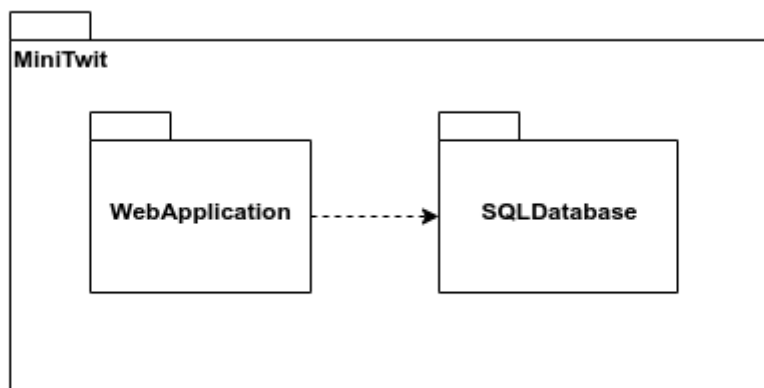


Figure 2. An overview of the two Java classes making up the Minitwit application.

The entire Minitwit system, including the web servers, database, CI and monitoring, is hosted on [DigitalOcean](#). To interact with and modify the servers hosted on DigitalOcean we use [Terraform](#).

For aggregating and querying logs we use [Grafana Loki](#), where the output of these logs and the current state of the system is monitored by [Grafana](#) using metrics from [Prometheus](#).

Whenever a pull request is merged to the main branch, the build server provided by [Drone](#) will build a [Docker](#) image based on the new code and push it to the [DigitalOcean Container Registry](#). The build server then notifies the two web servers that there is a new image ready, thus allowing them to pull it and deploy the new code.

The database server as well as the Drone server and the monitoring server pull images from [DockerHub](#) with all their necessary dependencies on creation.



- apk and apt
- grep
- Asciidoctor and Asciidoctor-pdf
- Ruby

Additionally we use [Docker](#) 20.10.9 to run services and Ubuntu Focal as the operating system.

[Prometheus](#) provides a set of metrics which can be visualized in Grafana. Monitoring is managed on a separate server which pulls all relevant images from DockerHub. These are the relevant dependencies:

- Prometheus v2.33.5
- Grafana OSS 8.4.3
- Loki v2.3.0
- Promtail v2.3.0
- Loki Docker Driver v2.4.2
- Node Exporter v1.3.1

In addition, we also use the following static analysis and maintainability tools:

- Better Code Hub
- Code Climate
- SonarQube
- ErrorProne
- Semgrep v1
- Snyk

## Quality Assessment

We made use of various tools in the project to keep track of the quality of our code and avoid introducing new bugs. This section will describe each of these tools and how we used them.

### Maintainability and Technical Debt

To evaluate the maintainability and technical debt of our code we used three tools: SonarQube, Code Climate and Better Code Hub. In particular, SonarQube was added as a stage in our build chain so that we could track the current code quality for every pull request. SonarQube keeps track of issues such as bugs and security vulnerabilities as well as issues pertaining to maintainability such as bad code style (which SonarQube has labeled 'code smells').

Due to the structure of the course, we added these tools after having refactored the application from Python to Java. We had limited time to accomplish this refactoring and create a working application, which naturally led to a lot of code smells being exposed by the tools. We made sure to fix the most critical bugs that the tools found at the time, but decided to leave the majority of the

code smells as we had to prioritize the other weekly tasks. Instead, we made sure that any new Minitwit features would not introduce more code smells than we already had. This meant that a pull request was not allowed to be merged to production if it got a worse rating than the original code.

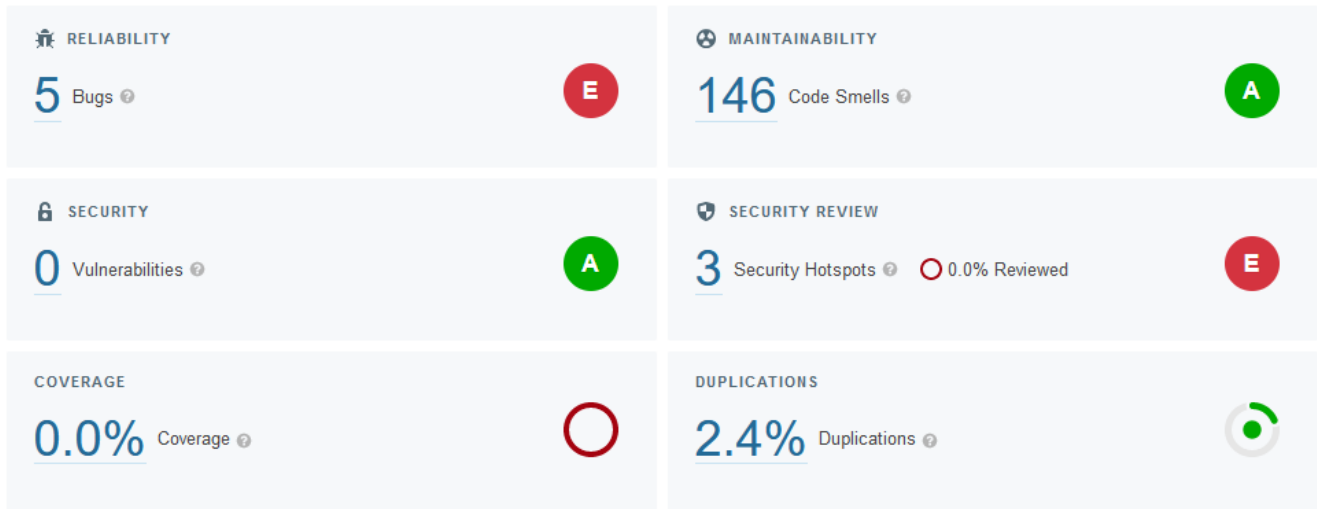


Figure 4. The current software quality rating of our project, analyzed by SonarQube. As can be derived from the image, we did not manage to include unit testing due to time constraints.

## Static Analysis

To conduct static analysis we made use of ErrorProne, Snyk and Semgrep. The first tool, ErrorProne, checked for errors and bug patterns at compile-time, blocking builds if one occurred. The latter two were added as stages in our build chain and thus also ensured that pull requests did not introduce new bugs or security vulnerabilities.

## License

The license chosen for this project is the Apache License version 2.0, which can be found in its entirety in the repository as [license.md](#). This license was chosen after carefully examining the licenses of all dependencies. Most of the dependencies were licensed either under Apache License 2.0 or The MIT License, which are both compatible. This means that we can bundle our whole project under either of them. However two of the dependencies used Lesser General Public License (LGPL) and Mozilla Public License 2.0. These are so-called weak copyleft licenses, meaning all modifications need to be open source - however other dependencies, and our own code, will not inherit the copyleft license. This means simply noting it as an extension in the end of our license will suffice.

## Process Perspective

This section will describe how we worked together as a team and the process within which development took place.

# Work Process

## Meetings

We had our meetings as a team every week in the exercise session since it was the best time for everyone to meet up. We mostly discussed what will be happening from then on, distributed the work of that week's exercises and evaluated finished development. Unfinished tasks and long-term tasks were briefly discussed to stay up to date on the ongoing development.

Work division for available tasks was overseen at the exercise sessions. While most work division was simple since there were multiple tasks, a few tasks were indivisible and forced a group member to focus on them for longer.

## Work Division and Issue Tracking

For the ongoing tasks, online communication was done on Discord since it was much faster than meeting up and quick to evaluate actions. We had a specific channel for our pull requests where new ones were posted, allowing for quick reviews by sending notifications to everyone.

If any sort of issue happened that was not dependent on the current tasks, a message including the details was sent out so that it could be investigated.

We mostly had online meetings when such were required outside the exercise sessions, and we shared information either via Discord messages or GitHub issues. These meetings were held for knowledge sharing, major changes, or explaining hard-to-understand features.

# Development Process

## Handling of GitHub Issues

The entire Minitwit application is in a single repository. [GitHub issues](#) were utilized in two different ways.

1. **Task distribution:** Weekly course tasks that had to be done were added as GitHub issues for easy tracking and assignment. These issues were also tagged with a milestone.
2. **Found bugs or problems:** In case of an occurrence of a bug or a newfound problem, a quick issue with its details was opened.

GitHub issues were also helpful for organisation since it offered great commenting abilities with text editing support and checklists for ongoing development.

## Pull Requests, Branching and Merging

New features were developed on separate branches with descriptive names. As such, all new pull requests were opened from within their own branches. This was done to avoid mixing up code whilst also organising the completion of issues. This also helped with reverting back to old code if required.

Every pull request also required at least one person to do a review before merging it to the main branch. There were no assigned reviewers for specific pull requests (except licensing, which everyone checked), so anyone available would check the pull request and allow it if everything seemed all right. If not, comments pointing out where the code had problems or potential future issues were given and it was rejected.

More information can be found in the [contribute](#) file.

## Automations

On every pull request we also ran the static code analysis tools SonarQube (using SonarCloud), CodeClimate and Semgrep. These helped us test, and find bugs, vulnerabilities and do other code checks.

# Deployment

## Minitwit application

To deploy a change to the Minitwit application, a pull request must be merged to main. This action will set off a pipeline in our CI/CD system that goes through the following steps:

0. Pull copy of newest code to docker container
1. Create and push docker image
  - a. Using Drone's "plugins/docker" image, we build a docker image of Minitwit and push it to our private docker registry
2. Snyk scan
  - a. Scans the new docker image for security issues
3. Update live docker images

Using ssh, the Drone server will upload a script to each host. The script will:

- a. Log in to our private docker registry
- b. Pull the new image
- c. Stop the old container
- d. Start the new container
- e. Delete old images
- f. Log out of the docker registry

If this script fails on a host or the web application doesn't become accessible with HTTP code 2XX on /status, the deployment will stop and be marked as failed.

4. Create GitHub release using the version number in Minitwit's `pom.xml` file.

This rolling release strategy will allow for zero downtime during upgrades, but might be problematic if two servers have different behavior on the same endpoint, which could happen in the middle of the upgrade when one server has been updated and the other hasn't.

## System changes

Everything else than our web application must be released semi-manually using different scripts. We use Terraform to manage infrastructure and a number of bash scripts to set up the servers. For example to prepare a new Drone server one must open a terminal in the Terraform folder and run `./setup_server_drone.sh $IP`, where the IP can be found using Terraform's CLI or the hosting provider's web panel. Some services need some manual configuration - for a deeper explanation of how to set up each application, refer to the [README](#).

## Repository Organization

For this project we have chosen a mono-repository setup. These work great, and we would consider that the standard unless you end up on a scale where multiple teams will be working on different systems, perhaps in a project involving microservices. For our project the scope of the system is rather small, and generally you could make a case for having the backend and frontend in two different repositories. However in our case the backend and frontend has a pretty tight coupling, making it not feasible without some intrusive refactoring.

## Monitoring

In this section we will go through some of the ways that the system was monitored.

As described earlier we used Grafana to visualize different aspects of the running application. We monitored:

- Actions in Minitwit:
  - Follows
  - New messages
  - Sign-ups
  - Followers count
  - Following count
- Status of the server:
  - HTTP errors in the past 24 hours
  - API request processing time
  - Web request processing time
  - HTTP requests
  - System uptime

These were used for debugging or checking the health of the server in case of a new merge messing something up or server faults occurring.

We also have hardware monitors, e.g. CPU utilization, used RAM etc. for checking server status if necessary.



The idea is to have views regarding business, development and system metrics, so people with different perspectives and roles can see data relevant for them.

## Logging

A very common logging stack is the ELK stack, which consists of elasticsearch, Logstash and Kibana. However when setting up our monitoring earlier in the project, we opted to use Grafana for our visualizations, which is a technology similar to Kibana. This meant that for implementing logging we had two choices: use the traditional ELK stack, or make our Grafana interface multipurpose.

We ended up using Grafana in combination with Loki to have our monitoring and logs in the same central place, hopefully simplifying things a bit. In addition, the ELK stack would have required high resource requirements as well. For log aggregation we used another solution by Grafana Labs, namely Loki. Its highly performant, being able to get logs from multiple sources, while being able to index various parts of the log string. This makes us able to sort the logs by time, severity, or any other tags we should find necessary in our debugging.

### Log Collection

For the actual local log collection on the machines we use Promtail. It's an agent designed to be used with Loki, and makes log collection easy. It can be run in a docker container, and simply reads the output of another docker container or service before sending it off to Loki.

[This is the pull request](#) that added most of the logging, setting up Promtail and Loki services.

In the code of our API, we added slf4j logger library [in this pull request](#), and in turn created a log message for every time our web server was pinged. As can be seen in the pull requests, we logged the timestamp, request status code, request method (GET, POST, etc.), IP address, and the URI called.

### Use

Using the logs we identified errors, such as when the `/metrics/stats` endpoint stopped working after switching database. This meant we could identify the problematic statements and fix the SQL query.

## Security Assessment

We have multiple tools to scan for security. During CI/CD steps we scan with SonarQube which detects problematic Java dependencies, and Snyk which also detects problematic docker images and libraries in docker images.










Snyk only runs on deployment due to scan limits on our account, but an improvement could be to do regular nightly scans to detect newly discovered security issues in already running services. These scans could also include third party services packaged in Docker such as Drone, PostgreSQL etc.

We also run tools such as Skipfish and WMAP, but found the results confusing as they contained multiple false positives because they interpreted unrelated problems as positive results on their






tests.

An XSS vulnerability was found by inserting HTML and JavaScript code into different fields.

We used the following matrix to score various scenarios:

	Low impact	Medium impact	High impact
Likely			
Possible			
Unlikely			

We found the following possible scenarios:

1.  A high severity vulnerability is found in a third party Java dependency, widespread exploitation leads to information leaks or remote code execution.
2.  A faulty authentication mechanism allows unauthorized users to access other users, leading to information leaks and impersonation.
3.  An XSS vulnerability leads to user impersonation, information leaks.
4.  A vulnerability is found in Drone's access management, leading to remote code execution, source code leaks, secrets leaks.
5.  A malicious actor gets push access to a code or docker dependency, leading to remote code execution and possible information and secrets leaks.

For more information, refer to the [full report](#).

## Scaling Strategy

Out of the two options we were given to implement scaling, we chose the high-availability setup that introduces a second web server to equally share the load with the original. To do this, we provisioned a new server on DigitalOcean through Terraform. This new server was simply created as an exact replica of the original, using Terraform's [count meta-argument](#). In addition, we created a DigitalOcean load balancer through Terraform and connected the two web servers to it, thus allowing it to equally spread requests between the servers. Of course, all web requests would now have to go through the load balancer, rather than directly to the original web server.

To properly deploy new versions of the application to both web servers through our build chain, we created a [script](#) to execute rolling updates in one of the build steps. This script updates one server, then checks whether it has come back alive. If not, it aborts the update procedure such that the second server will still be online and serving users. At the moment it does not roll back the failed server, though this is something we would like to implement in the future.

This setup definitely improves availability, as it allows for the failure of one web server without the system going down. As long as one server is still running, the application will be able to serve requests. However, the load balancer is still a single point of failure - if that goes down, then the application will not work. As such, a better setup would be to include a second load balancer to act as a standby in case the first goes offline.

# Lessons Learned Perspective

This section includes the description of the primary issues and their solutions, and the valuable lessons learned regarding maintenance, operation, evolution and refactoring.

In this course we learned less about writing functionality into an application, and more about how to maintain the application and how it evolves. We learned about how to use logs and monitoring to detect issues and react to them. This caused a nice loop of incremental changes and early feedback from our tools and group members.

We worked a lot with automating tasks, such as code feedback and deployments. We had some problems getting used to using our static analysis tools as, according to their checks, we were at a very bad state code-wise when we started. We had to prioritize our time with new tasks every week, and so we decided to mostly use the tools to ensure that our code did not get worse instead of improving existing code.

Something else we could have worked more on is "truly" automatic server deployments as we had to write IP addresses manually and hard coded them into config files. Using a tool such as Ansible which provides a mechanism for using templates to generate config files and has a central place to list hosts could have been an improvement. Still, using automated deployments gave more safety when deploying than manually executing scripts and allowed for faster deployments. We also learned that fewer steps between writing code and deploying gave more confidence to the developers. For example, we used to merge to a 'dev' branch and only deploy after merging dev to main. This delayed releases and made developers more anxious during deployments. Merging directly into main made this a less cumbersome experience.

We started using Terraform as our infrastructure-as-code tool very early on, which made it easy to add more servers and provider-specific services such as a managed load balancer and floating IPs. It was a bit problematic in the beginning as the state included confidential information and could not be committed to Git, but the shared state in the cloud mechanism allowed us to collaborate better.

Using monitoring we detected a memory leak, and SonarQube helped us find the specific code that was causing it. Logs provided a way for us to find code errors and allowed us to gain insights into usage patterns, such as how often our simulated users posted and how many other users they had followed.

In general, we learned a lot, but also found the experience to be slightly stressful as DevOps includes learning a huge number of tools. Still, being able to write awesome code doesn't matter much if you can't deploy it confidently.