DevOps, Software Evolution and Software Maintenance

Final Report

Group G - Dev Janitors

Adrian Bay Dorph addo@itu.dk
Anton Marius Breinholt antbr@itu.dk
Laurits Bonde Henriksen lauh@itu.dk
Markus K. R. Johansen mkjo@itu.dk

3494 words

Course Code:
BSDSESM1KU

IT-UNIVERSITETET I KØBENHAVN

May 24 2023

Contents

Chapter 1

System's Perspective

1.1 Design and Architecture of our ITU-MiniTwit Systems

The system consists of a backend api written in C# with ASP.NET framework, a GUI frontend written in TypeScript with React.js, and a Microsoft SQL Server (MSSQL) as the database. Every component of the system(backend, frontend, database, logging, monitoring) is isolated into docker containers and is run with a docker-compose file.

Frontend

The frontend has used the `npx create-react-app` template to start the project and build on. As React applications are single-page applications we have used the `useRoute` hook to create the different pages needed. In the frontend, multiple instances of reusable components are utilized within various pages to prevent the repetition of code.

Backend

Our backend follows the onion architecture consisting of the core, infrastructure layer, and presentation layer. The core layer consists of Data Transfer Object (DTO) classes and repository interfaces. The infrastructure layer uses the repository pattern with Create, Read, Update, Delete (CRUD) operations. The presentation layer consists of the controllers with endpoints and some simple application logic. We didn't find it necessary to have an application layer because the logic in the system is quite simple. Therefore there is some application logic in the controllers instead.

Database

For the database, we use a relational MSSQL with Entity Framework (EF) Core to couple the database with the backend. The database consists of 3 tables: A 'users', a 'messages', and a 'followers' table. Messages have a many-to-one relationship with Users and Users have a two-to-many relationship with Followers, meaning that a user can have many followers but one follower entity only has two users; the user that is following and the user that is being followed.

1.2 All dependencies of our ITU-MiniTwit system

CI/CD Tools & Technologies

- GitGuardian Security Check - Used in our CI pipeline to check for secrets in the code.
- Codeclimate - Used in our CI pipeline to check code quality.
- SonarCloud - Used in our CI pipeline to check code quality.
- GitHub - Used to manage the project.
- Docker - Used to run the system.
- yaml - Used in our CI/CD pipeline for docker and other technologies
- Snyk - Used in our CI pipeline to check for security vulnerabilities in outdated components. Also does a weekly scan of the repository and creates Pull Request (PR)'s for any outdated dependencies.
- Vagrant - Used in our CI/CD pipeline to deploy the system to digital ocean droplets.

System Tools & Technologies

- C#/ASP.NET - Used to construct the backend.
- React/Typescript - Utilized to build the frontend.
- EF Core - Utilized for communicating with the database.
- GitHub - Used to manage the project.
- Prometheus - For metrics scraping/monitoring.

- ElasticSearch - For storing logging data.
- Grafana - For displaying metrics from Prometheus.
- Serilog - Used to log system data which it sends to Elasticsearch.
- Kibana - For displaying logging data from ElasticSearch.

1.3 Important interactions of subsystems

The frontend relies on interaction with the backend to get the information needed to show the user. The backend also relies on the database for data to process and pass to the frontend.

The monitoring subsystem consists of Prometheus and Grafana containers. Prometheus relies on the backend for data scraping. Grafana needs a data source in Prometheus, to display information.

The logging setup is composed of ElasticSearch and Kibana containers, as well as our logging provider in Serilog. Serilog relies on the backend for logging data which it sends to the ElasticSearch sink, which feeds that data to Kibana.

1.4 Current state of our system

1.4.1 Features implemented

All the features from the original MiniTwit have been implemented.

1.4.2 Code Quality

When looking at SonarCloud we can see that there are no Bugs or Security vulnerabilities, however, we have marked some Security issues as safe because they were from the Python simulator test code. There are issues with Maintainability and code smells in our code. Particularly unused import statements, commented-out code, unused variables, etc. There are also duplication issues, where parts of the code could be refactored as a solution.

When looking at Codeclimate, we can see that the code maintainability rating is B, but there were a bunch of issues regarding parts of the code that we did not write ourselves e.g. python code that is a part of the

simulator and simulator tests, and auto-generated database migration files. These issues have therefore been marked as invalid or won't fix. There are also a few other issues that we marked with invalid or won't fix if we didn't think it was a problem, for example, if a method exceeded the 25-line limit with one line.

1.5 License Compatibility

In our project we use the MIT license and our dependencies use the following licenses:

- MIT
- ISC
- Apache-2.0
- NOASSERTION
- CC0-1.0
- CC-BY-4.0
- 0BSD
- BSD-2-Clause
- BSD-3-Clause
- Python-2.0
- MPL-2.0
- Unlicense

All of these licenses that our dependencies use are compatible with the MIT license that our project uses.

Chapter 2

Process' Perspective

2.1 Developer Interaction

Beyond the exercise sessions, we had a work session together every Friday. Most Fridays we met in person, but sometimes we met online using Discord. Discord is also the place we used to communicate daily with meeting times, when tasks would be done, file sharing, and so on. We also used certain GitHub features to communicate, for instance comments on pull requests.

2.2 Team Organization

We have an organization set up on GitHub that contains our repository. We could however have used more of the organization features, for instance creating teams. Otherwise, we planned from meeting to meeting what needed to be worked on in between. At the beginning of meetings, we did a little catch-up on the progress that had been made on the tasks.

2.3 Stages and tools included in the CI/CD chains

Most of our CI/CD chain is handled by GitHub Actions. The first thing that happens in our CI/CD chain is a workflow that runs to build and test the newly added code, before it is merged to master. This workflow exists in the file `dotnet.yml`. Currently, it only builds our .NET project for the backend part of our system. Ideally, the workflow would also test our frontend, however, at the time of implementation we had no way to test the frontend. The `dotnet.yml` workflow does the following things: Checks out the newly added code sets up .NET and installs dependencies, builds our

project, and tests against our unit tests. This workflow ensures that code integrates seamlessly with the production branch, allowing us to practice continuous integration.

Additionally we use a range of static analysis tools. GitGuardian helps us to detect secrets, such as login credentials, that have been pushed to GitHub by mistake. Snyk checks through the code and our dependencies to uncover any security vulnerabilities. It also proposes changes to fix detected vulnerabilities through a PR. CodeClimate checks our code for code quality, which includes duplication, lines of code, and code complexity. SonarCloud also checks on code quality, security vulnerabilities and bugs and provides a quality gate feature. We have not used these tools as an automated part of our CI/CD chain, but more as an added check to make us aware of issues in the code and help us fix it fast.

When new code additions are merged into the master branch of our repository, a GitHub action workflow `continuous-deployment.yml` is triggered. The workflow uses GitHub Secrets, which allows us to hide our credentials from the plaintext in the .yml file. They are used to log in to Docker Hub and ssh to our Digital Ocean droplet, and they must be set up in the repository for the workflow to function correctly. The secrets that must be set are: `DOCKER_USERNAME`, `DOCKER_PASSWORD`, `SSH_USER`, `SSH_KEY`, and `SSH_HOST`. The workflow then does the following things in order:

1. Checkout the master branch with the newly merged code.
2. Login into Docker Hub.
3. Setup docker build tools.
4. Build the Docker images for our backend and frontend.
5. Push the Docker images to Docker Hub.
6. Configure SSH information.
7. Open a shell to the droplet via SSH, and execute a script `deploy.sh` (can be seen in the remote_files directory in our repository).
8. The shell uses a `docker-compose.yml` file that first pulls our images from Docker Hub, and then starts the containers with specified ports and volumes.

9. Create a release on our GitHub repository, that uses the next available release version (currently `v1.0.x`), and has the commit messages from the merge as release notes.

We have a similar workflow called `continuous-deployment-staging.yml`. This has been used in a loose way to test out code on a staging droplet. It does the same things as the other CD workflow, except it pushes staging images to Docker Hub, and does not release. The workflow is not part of our automated CI/CD chain however this could be implemented by having a staging branch, as a step before the production branch. Here we could for example run end-to-end tests.

2.4 Repository Organization

All of our code is located in a single repository `Dev-Janitors/minitwit`. The repository is owned by our Github organization DevJanitors. The reason for having an organization is that it provides a sense of equal ownership and control over the repo. However, practically this is the same as one person owning the repo and giving admin rights to team members.

Our repository is split up into several sub-directories.

- `.github/workflows`: Contains the .yaml files that define the Github actions we use.
- `Backend`: Contains everything related to the backend of our system (except the database itself). This includes a complete ASP.NET API and a dockerfile specifying how to build our backend Docker image from the code.
- `Database`: Contains only a single dockerfile, that specifies how to build the Docker image for the database in our system.
- `Frontend`: Contains all code related to the frontend of our system. This consists of a React.js project with our components located in the src sub-directory. Again, there is also a dockerfile for building.
- `remote_files`: Contains all the files that will be uploaded to our main Digital Ocean droplet created by calling `$ vagrant up`.

- `remote_files_staging`: Contains the same files as `remote_files`, but specifically for our staging droplet. This way we can test if updated code works on the staging server before we add it to the main server.

Additionally, the master directory of our repository contains some important files. `Vagrantfile` specifies how we want to spin up our droplet for hosting the backend, frontend, and database. The file `docker-compose.yml` can be used to start the complete system locally. It builds all our images and starts containers. `prometheus.yml` contains the settings for how Prometheus should scan our system metrics for monitoring. The files `minitwit_simulator.py`, and `minitwit_scenario.csv` are for testing our API on sample requests from the minitwit simulator.

Having a single-repository setup saves time when we as developers want to quickly shift from working on the frontend to the backend to testing to infrastructure. It does however make the codebase more complex and confusing for new developers looking at the system for the first time. Another approach would be to separate the different parts of the system into more repositories or branches.

2.5 Applied development process and supporting tools

2.5.1 Git & GitHub

We utilized Git as our version control system to manage our source code efficiently. Git facilitated team collaboration, branching, merging, and ensured a coherent codebase throughout the project. We leveraged GitHub as a central repository to host our Git repository.

Applied Branching Strategy

In our branching strategy, we have set restrictions for merging and committing to the master branch, so that it is impossible to push commits directly into the master branch, and when merging branches into master, it must be approved by at least one other team member. Therefore, when we have to create a feature or fix a bug, we create a supporting branch for it, and when that feature is done, we create a PR to the master branch, merge it in, and delete the supporting branch.

2.5.2 Pair Programming

Pair Programming was used in the development process. It involved multiple team members working together via LiveShare, collaborating in real-time, sharing knowledge, and producing code together.

2.6 System Monitoring

2.6.1 Prometheus and Grafana

Prometheus is a monitoring system that collects and stores metrics from configurable sources.
Grafana is a platform for data visualization and analysis that integrates with Prometheus. It provides an interface for creating dashboards where queries can be executed and illustrated to provide an overview of the health and performance of the system.

2.6.2 Our usage of Prometheus and Grafana

We have configured Prometheus to scrape metrics from our backend at http://104.248.101.163:2222/metrics.

We have used Grafana to build queries utilizing the metrics gathered by Prometheus. The queries are visualized in Diagrams, and the Diagrams are all located in our Minitwit dashboard.

We will now go through each of the queries/diagrams on the Dashboard and argue why we have chosen them.

Our first metric shows endpoint hits per 5 minutes for all endpoints combined in a single graph. We have chosen to include this graph because it allows us to track the usage and popularity of each endpoint, but more importantly, it enables us to check for anomalies that could indicate that something is wrong with the system.

The second metric shows latency/response time for every endpoint. We use this graph to identify anomalies and the performance of the system. This graph combined with the endpoint hit graph, enables us to showcase the performance compared to the usage. We can see how well the system scales and if our endpoints are slow due to high usage or if there could be a differ-

ent problem, that results in high latency.

The third metric shows endpoint hits for the msgs/username endpoint. In this graph, we both show the number of responses with code 200 and the responses with code 404. With this comparison of how many successful endpoint hits there are compared to errors, we can detect if there is a problem with this specific endpoint. The fourth metric is the same as the third but with the fllws/username instead. We have set up this metric for the same reasons.

Lastly, we have an endpoint hit counter for all endpoints. With this graph, we can see how much each endpoint is used over the entire up-time of the system. The first four graphs can be seen in Figure 2.1

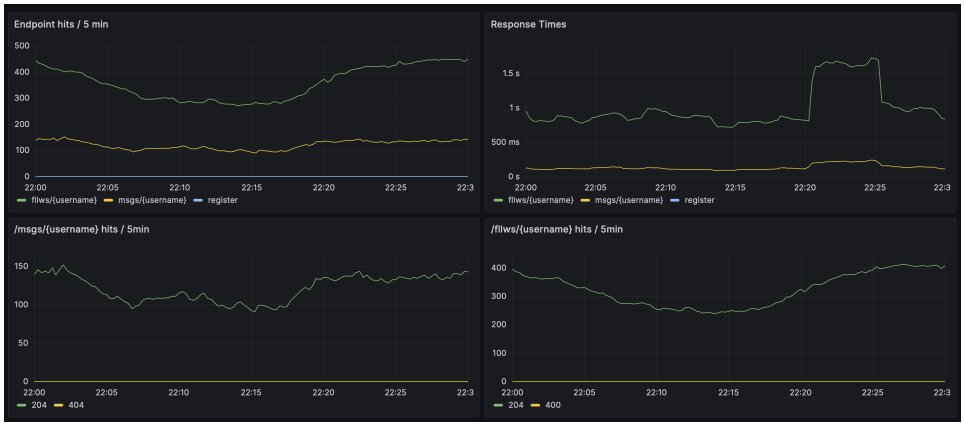

Figure 2.1: Monitoring dashboard in Grafana.

2.7 System Logging

2.7.1 ElasticSearch, Serilog, and Kibana

For our system logging, we use ElasticSearch, Kibana, and Serilog. We send Serilog data to ElasticSearch with an ElasticSearch sink in our Program.cs file. We use Kibana to visualize the ElasticSearch data.

2.7.2 What do we log?

We log all communication going to and from the server such as endpoint requests and backend metrics scraping with Prometheus.

2.7.3 How do we aggregate the logs?

To aggregate the logs we simply use the Discover tab in our ElasticSearch configuration. This provides a nice overview of the logging activity for our system by, for instance, enabling us to examine logs in specific timeframes.

2.8 Results of Security Assessment

2.8.1 Risk Identification

Based on identified threat sources (omitted due to brevity), we developed risk scenarios.

- Risk Scenario 1 (RS1): An attacker gains unauthorized access to Bob's account using his username and email address, allowing them to impersonate Bob and post tweets.
- Risk Scenario 2 (RS2): An attacker discovers the appsettings.json file in our public GitHub repository, retrieves the database connection string, and proceeds to delete our entire database.
- Risk Scenario 3 (RS3): An attacker exploits the publicly available ElasticSearch configuration, downloads and deletes critical system logs, and demands a ransom for their return.
- Risk Scenario 4 (RS4): An attacker uses tools like Wireshark to intercept and eavesdrop on login data transmitted over the network, potentially gaining unauthorized access.
- Risk Scenario 5 (RS5): An attacker orchestrates a distributed denial-of-service (DDoS) attack by flooding the /msgs endpoint with a large volume of requests, causing server overload and disrupting service availability.
- Risk Scenario 6 (RS6): An attacker identifies an outdated dependency within our assets and exploits known vulnerabilities associated with it.

2.8.2 Risk Matrix & Solutions

Next, we created a Risk Matrix to visualize the severity of the risks. (Figure 2.2)
Based on the risk matrix analysis, we determined that addressing RS2 should be our highest priority. Following that, RS3 and RS5 are relatively urgent and would be addressed as the next steps. RS1, RS6, and RS4 will be tackled subsequently.
To mitigate the identified risks, the following solutions have been formulated:

	Low impact	Medium impact	High impact
High likelihood	RS1	RS3	RS2
Medium likelihood		RS6	RS5
Low likelihood	RS4		

Figure 2.2: Risk matrix for our risk scenarios. RS refers to Risk Scenario.

- RS1: Introduce a password-based authentication mechanism, requiring users to provide both their email and password for login.
- RS2: Remove any sensitive information from the publicly accessible GitHub repository, and consider generating new secrets to enhance the system's security.
- RS3: Implement a secure login mechanism for ElasticSearch, ensuring that only authorized users can access and modify the system's logs.
- RS4: Configure the application to utilize HTTPS instead of HTTP to encrypt data transmission and prevent eavesdropping.
- RS5: Modify the system's default behavior to limit the number of returned messages to a reasonable amount, preventing excessive data exposure.
- RS6: Utilize the Snyk tool to identify outdated dependencies within our assets and promptly update them to mitigate potential vulnerabilities.

Due to time constraints, we were only able to address RS5 by setting the default number of returned messages to 100, ensuring a more controlled data exposure.

2.8.3 Pentesting with Nmap

In addition to our security assessment, we conducted a penetration test using Nmap to identify potential vulnerabilities in our system. From the test, we discovered that running SSH on port 22 exposes our system to known exploits, which could compromise its security. Furthermore, we identified that our ElasticSearch instance lacks built-in security features, leaving it vulnerable to unauthorized access.

2.9 Applied strategy for load balancing

Currently on our deployed system we do not have a load balancing strategy. We struggled a lot with Docker Swarm and connecting it to the the database. However we managed to get Docker Swarm working, except for CORS issues between backend and frontend. So in a development branch we use Docker swarm with a swarm Manager node and we run replicas of our frontend and backend and only single instances of the other services. Since we only have a single swarm manger there is still potential for single point of failure. In the future it could be worth discussing implementing dedicated load balancers.

2.10 Our usage of AI

In our project, we have used two different AI's, GitHub Copilot and ChatGPT. Copilot has been used as an advanced linter just to support development and sometimes give inspiration, but no full function has been written using Copilot. ChatGPT has been used to gather information and setup examples of the different technologies we use, especially for monitoring and logging. Text generated with ChatGPT has also been used as a starting point for some of the chapters in this report.

Both AIs have been useful to us when used in moderation. A big codeblock from Copilot can have bugs that you would have to go through the code and find, which could be slower than just writing the function yourself. The same goes for ChatGPT, but when you use it as a part of your information gathering, and not the only source, it can be very helpful.

Chapter 3

Lessons Learned Perspective

3.1 Evolution and Refactoring

Early in the project we decided to develop our frontend from scratch instead of using the static HTML templates provided with the base minitwit project. This meant that we had to spend much time writing frontend code, while also integrating new technologies each week. Because of this, we were behind in the beginning and when the simulator started there were issues with our API. Throughout the course, there was an emphasis on taking micro steps. Regarding the frontend, we failed to do this and we bit over more than we could chew. Ideally we had started by ensuring that our system worked with the frontend template, and then slowly worked towards implementing a different frontend.

3.2 Operation and Maintenance

The biggest issue is that we were faced with the loss of our production database at the start of the project. This happened because we didn't use volumes in Docker at the time and upon a `docker-compose down` and `docker-compose up` command in our CI/CD pipeline, the database was thrown away and a new one would be created. We realized that we needed to have a way of rebooting the entire system in our pipeline that would use the same database as before. Therefore we started to use Docker volumes. In future projects, we would also use a backup system for ensuring data recoverability if something unintended were to happen which results in the loss of a production database.

3.3 DevOps Approach

The requirement to release each week made us work on the project much more consistently. In previous projects, we have been more inclined to work slowly throughout the semester and then do a large chunk of the development before the hand-in deadline. This way of working is not very sustainable and doesn't reflect how a "real world" development project would function. In this course, we have taken a much more agile and sustainable approach, by developing new features each week at a steady pace.

The same goes for the maintenance part. Because the project is live and running with simulated users accessing our service, we needed to be able to fix errors as soon as they were detected. This means we had to communicate closely within the team and coordinate which team members are available to fix issues.

We have worked in a way in which we are constantly learning about new technologies and then actually implementing them in our system. By working this way we can make sure that we are always using the technologies that fit our scenario the best, and that we are making our lives as developers as easy as possible.

Abbreviations

CD Continuous Deployment. 1, 4, 7, 9, 17

CI Continuous Integration. 1, 4, 7, 9, 17

CRUD Create, Read, Update, Delete. 3

DTO Data Transfer Object. 3

EF Entity Framework. 4

MSSQL Microsoft SQL Server. 3, 4

PR Pull Request. 4, 8, 10