DevOps - Group M

Casper Wassar Skourup - caws@itu.dk
Lukas Hjelmstrand - luhj@itu.dk
Mads Piriwe Risom Andersen - mpia@itu.dk
Niclas Søgaard Hjortkjær - nihj@itu.dk
Villum Nils Robert Sonne - vson@itu.dk

24/05-2023

`DevOps, Software Evolution and Software Maintenance, BSc`
`Course code: BSDSESM1KU`

Github link: `https://github.com/NiclasHjortkjaer/itu-minitwit`

Contents

1 System Perspective

1.1 Design and architecture

The technologies we selected to recreate the legacy Flask application ITU-MiniTwit, were C# with the ASP.NET Core framework for the server and Postgresql as the database. The decision to use ASP.NET Core was made based on the team's prior experience, which enabled us to re-implement the Flask application's features quickly. Another reason for our selection was that with the performance enhancements of .NET 7, this is a very well performing framework[1]. *README.md* in our GitHub repository contains instructions on how to run and deploy the system.

Figure 1 shows the internal design of the MiniTwit application. We used Razor pages to render the web pages on the server side. For the course simulator to send request to our system we created an API at */simulator*.

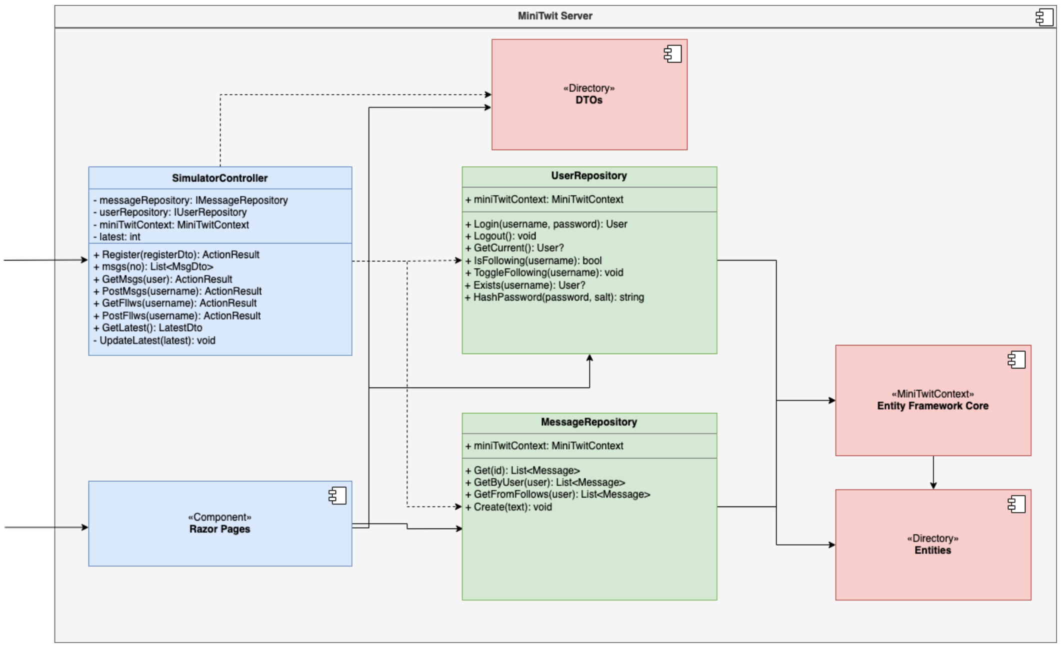

MiniTwit Server
«Directory»
DTOs
SimulatorController
- messageRepository: IMessageRepository
- userRepository: IUserRepository
- miniTwitContext: MiniTwitContext
- latest: int
+ Register(registerDto): ActionResult
+ msgs(no): List<MsgDto>
+ GetMsgs(user): ActionResult
+ PostMsgs(username): ActionResult
+ GetFllws(username): ActionResult
+ PostFllws(username): ActionResult
+ GetLatest(): LatestDto
- UpdateLatest(latest): void
UserRepository
+ miniTwitContext: MiniTwitContext
+ Login(username, password): User
+ Logout(): void
+ GetCurrent(): User?
+ IsFollowing(username): bool
+ ToggleFollowing(username): void
+ Exists(username): User?
+ HashPassword(password, salt): string
«MiniTwitContext»
Entity Framework Core
MessageRepository
+ miniTwitContext: MiniTwitContext
+ Get(id): List<Message>
+ GetByUser(user): List<Message>
+ GetFromFollows(user): List<Message>
+ Create(text): void
«Component»
Razor Pages
«Directory»
Entities

Figure 1: Design of the MiniTwit service.

Figure 2 shows the architecture of the final version of the system that was deployed, using three droplets (virtual machines) on DigitalOcean. Docker containers are within a docker swarm consitsting of one manager and two workers. There are three replicas of the MiniTwit server in the swarm, one on each droplet, to ensure availability. The two worker nodes only contain an instance of the MiniTwit server, while the manager includes load balancing, logging, and monitoring services. We decided to keep all these services on the same node to be accessible on the same IP address. Nginx is used for load balancing between the MiniTwit instances. Rolling

[1]14th in this benchmark: `https://www.techempower.com/benchmarks/#section=data-r21`

updates are implemented through the docker swarm.

The database was hosted as a DigitalOcean managed database so that it was guaranteed to be reliable. Initially, we managed the Postgres database ourselves. However, one day, it suddenly crashed, which caused approximately 12 hours of downtime. We decided to switch to a managed database in our deployment to avoid worrying about this.

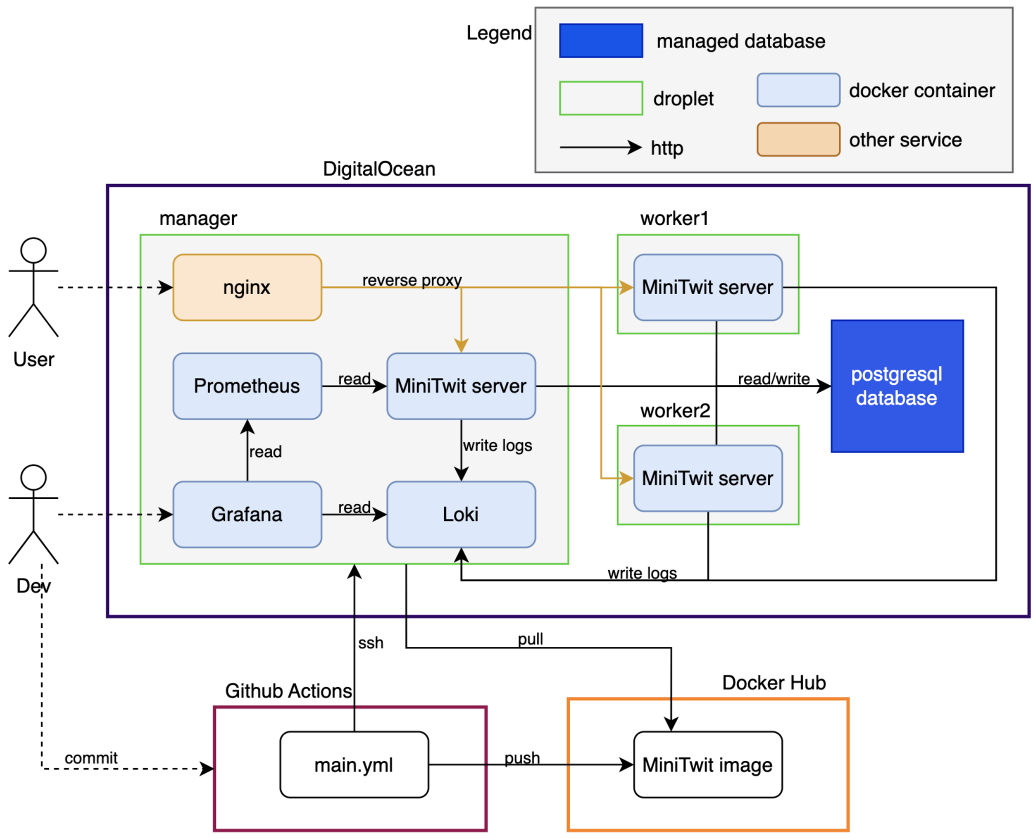


Figure 2: Architecture of all subsystems in the final system.

1.2 Technologies and Dependencies

The technologies the system uses and depends on are listed below. All versions used were the most recent stable releases as of the start of 2023:

- **ASP.NET Core**, for the web application.
- **PostgreSQL**, for data storage.
- **nginx**, for reverse proxy and load balancing.
- **Grafana**, for viewing metrics and logs.
- **Loki**, for storing logs.
- **Prometheus**, for storing metrics.
- **Docker**, for containerization.
- **Docker Hub**, for container image registry.

- **Ubuntu**, for the server OS.
- **Python**, for the tests.
- **Nuget**, for package management.
- **Github Actions**, for CI/CD.
- **DigitalOcean**, for infrastructure.
- **SonarCloud**, for static analysis.
- **Snyk**, for security assessments.

The libraries which the ASP.NET Core MiniTwit application depends on are:

- **Entity Framework Core**, library for database abstraction layer.
- **Npgsql**, library for connecting to Postgres from .NET.
- **Serilog**, library for logging.
- **Serilog.Sinks.Grafana.Loki**, library for shipping logs to Loki.
- **Prometheus-net**, library to instrument .NET code with Prometheus metrics.
- **SignalR**, library for real-time communication between server and client.
- **code-cracker**, library for analyzing code.

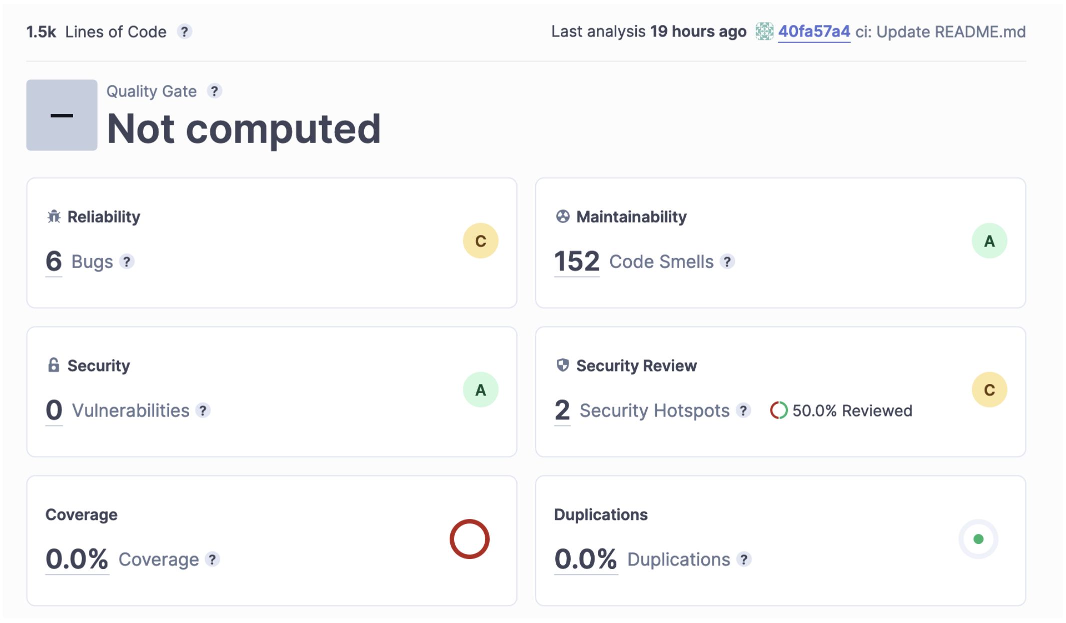
1.3 Important interactions of subsystems

Figure 2 also shows the interactions of the subsystems. Docker containers communicate via the internal network within the swarm. A user accesses the system via the address of the Nginx load balancer which reverse proxies to one of the active MiniTwit servers. The logging libraries we have used allows our MiniTwit servers ship logs itself, avoiding the need to have a log shipper instance, such as Promtail, on every node in the swarm.

A limitation of our chosen architecture is that Prometheus only reads from one of the MiniTwit server instances, the one on the same node. Since the load is balanced, only 1/3 of the traffic is monitored by metrics, and visible in Grafana. A similar problem appeared with the real-time updates we implemented with SignalR. Because of the load balancing, only every third message will, on average, appear for the user without reloading the page.

1.4 Current state of systems

Due to our project utilizing infrastructure as code, it is possible to turn the system on/off quickly. We have created a script, *infrastructure.sh*, which deploys the entire system and provisions the infrastructure on DigitalOcean. When writing this report, we have turned the system off to minimize payments to Digital Ocean. When turning our system on, everything works as expected. The current code quality can be seen on our SonarQube page, which is showcased in figure 3.


1.5k Lines of Code ?

Last analysis **19 hours ago** 40fa57a4 ci: Update README.md

Quality Gate ?

Not computed

Reliability 6 Bugs ? C	**Maintainability** 152 Code Smells ? A
Security 0 Vulnerabilities ? A	**Security Review** 2 Security Hotspots ? 50.0% Reviewed C
Coverage 0.0% Coverage ?	**Duplications** 0.0% Duplications ?

Figure 3: An overview of the code quality of the project in SonarQube.

The current architecture and infrastructure comfortably handles the load of the simulator using three DigitalOcean droplets, which is likely overkill.

If we were to continue working on the system, a thing that we would work on is fixing the issues currently caused by having load balanced between multiple instances of the application server. Namely that 2/3 of metrics are not captured by Prometheus.

1.5 License of our systems

As we use the MIT license, and most of the direct dependencies use AGPLV3, we do not have any license conflicts. Our API also has an SLA.

2 Process Perspective

2.1 Teamwork

2.1.1 Team agreement

Within the period where the project was worked on, the team formed some rules and guidelines based on some of the points presented in the "Three Ways" to characterize DevOps from "The DevOps Handbook"[2]: Flow, Feedback and Continual Learning/Experimentation . This was to enhance efficiency and ensure transparency within the group.
To ensure flow, we did the following:

- Mostly through communication on discord, but also through the use of Github projects, we made work visible and transparent for each other.
- Utilized a small batch strategy, where single features are developed, pushed, and deployed at a time.
- By utilizing continuous integration and deployment we limited the amount of intermediary steps, which had to be manually executed.
- By being aware of what limitations our architecture and processes bring. Early on we realized that our CI/CD tools making a new database at each deployment was not a good idea, and moved to a managed database.
- We tried to reduce working on tasks which did not grant any value to the customer by minimizing dead code, feature switching, and manual work.

To ensure feedback, we did the following:

- We used logs, monitoring, and testing to automate quality checking and see problems as they occurred, and communicated them to the team, if they could not be solved immediately.

To ensure continual learning and experimentation, we did the following:

- We kept an open discussion about what we could improve and automatize in our work such as when we automatized releases.
- Ensuring a learning culture by sharing knowledge at weekly meetings and being open about personal strengths and weaknesses.

2.1.2 Working schedule

The team's interaction has been physical at ITU and online through Discord. The primary working day was Tuesday after the exercise session. First, we solved the exercises together and then implemented the feature in our system. If there were leftover work, we would split into subgroups, as different schedules made it difficult to meet physically more than once a week. Since most of the features and tools were new to the group, the overall focus was on implementing and correctly using them.

[2]https://www.oreilly.com/library/view/the-devops-handbook/9781457191381/

This led us to use pair programming, where one team member shared his screen, and others provided input. The other team members would write code individually if the work were trivial. This was to speed up the process, especially when working with something we did not initially understand.

We incorporated retrospectives from Scrum every Tuesday before we began working on the following week's tasks. This means that we discussed what had been implemented since last Tuesday to share our individual knowledge with the group.

2.2 CI/CD

We used GitHub Actions to implement our CI/CD. Our CI/CD consists of test, deploy, release, and code quality analyzer. When a developer pushes his commits, GitHub Actions will run the following steps:

2.2.1 Code quality analyzer

SonarCloud was used as a code quality analyzer to analyze the technical debt of the code base. Our repository contains badges about technical debt, security, vulnerabilities, and bugs based on the latest commit to SonarCloud. We also used the C# linting tool *code-cracker*, but this was only used when building on developer machines and not a part of the workflow. Furthermore, we have used Snyk to scan the project, which is not part of the workflow either.

2.2.2 Build and Test

We have different tests as a quality gate before deploying and releasing. We have refactored and expanded the Python tests from the exercises. The system is built on the workflow server via Docker-Compose. First, we have a smoke test to verify that essential elements of the application are functional. Second, we use integration tests to test the API. Last, we use Selenium for end-to-end tests. The Python tests interact with the system via HTTP.

2.2.3 Deployment

If the tests are successful, the push will be deployed. First the new image for the application server is pushed to Docker Hub. Then, via ssh, a script is executed on the deployed manager node which pulls the image and redeploys the system.

2.2.4 Release

After deploying the application, we use Versionize and Husky for automatic releases on GitHub. Husky is a linter for commit messages, which checks that a commit message contains the right format. Versionize increments the release version when desired. If the deploy was successful, and the commit message increments the release tag, then our pipeline creates a new release.

2.3 Repository organization

We use a mono repository for our source code. We have a trunk-based development, meaning developers push directly to the main branch. If the commits pass the GitHub Workflow, then they will be released. We omitted code reviews because we thought it took too much time to do them. Initially, we considered using it for knowledge sharing. However, since we used pair programming, we already had a way of knowledge sharing. This way we could release new code fast.

2.4 Development Process

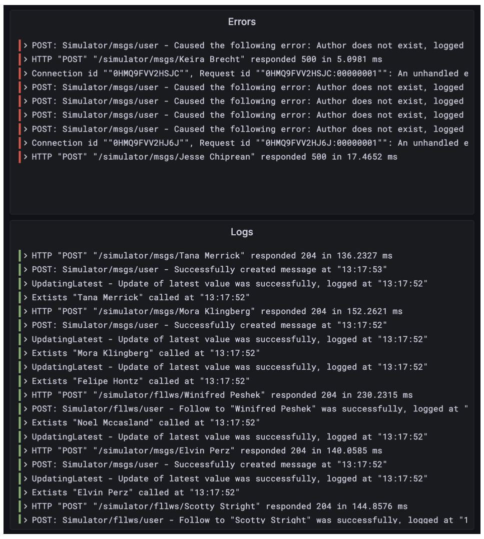
To keep track of issues and tasks which need to be done, we have experimented with using GitHub Projects as a Kanban board. We have not used this feature rigorously since the course structure already provided some structure of what needed to be done and when. The primary usage has been when we split our group into many parts (individual work) and needed to keep track of it. Mostly development has happened in group work, either all together or split into two groups. Since our project and group size are relatively small, it is still possible to hold developers accountable by reviewing the commit history. However, a more rigorous use of the Kanban board would have made it possible to measure implementation progress retroactively.

2.5 Monitoring & Logging

During the lifetime of a system, it is rare for everything to work smoothly and without any problems. System components will break down. The system will crash in production, server response times will grind to a halt, and so on. To (1) be able to estimate when the system is not operating within the expected limits and (2) be able to track down the source of this disturbance, it is necessary to have monitoring and logging setup in the project.

Our monitoring is done via Grafana dashboards, where we have created two dashboards: One for system metrics (RAM, CPU usage, response times) and one for business logic (active users, timeline requests, number of tweets made). When the first dashboard showed irregular behavior, it was a sign that our virtual machine in the cloud required scaling - or that poorly performing code had been pushed recently. When the second dashboard showed irregular behavior, it was a sign that our website was not deployed properly and was inaccessible to users or that some recently pushed code had broken core functionality.

We have implemented logging with Loki and Grafana to track all request the system handles. This allows us to determine whether or not something is wrong in case these API calls fail. We have a dashboard on Grafana that displays errors and all logs. Figure 4b shows the logs immediately after destroying the database. Here we can see that errors surface.



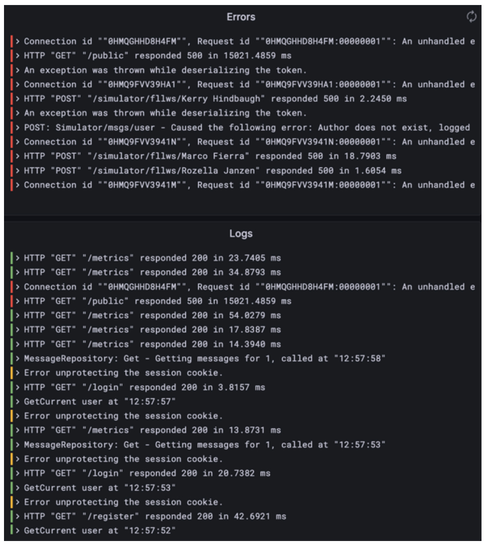
```
Errors

> POST: Simulator/msgs/user - Caused the following error: Author does not exist, logged
> HTTP "POST" "/simulator/msgs/Keira Brecht" responded 500 in 5.0981 ms
> Connection id ""0HMQ9FVV2HSJC"", Request id ""0HMQ9FVV2HSJC:00000001"": An unhandled e
> POST: Simulator/msgs/user - Caused the following error: Author does not exist, logged
> POST: Simulator/msgs/user - Caused the following error: Author does not exist, logged
> POST: Simulator/msgs/user - Caused the following error: Author does not exist, logged
> POST: Simulator/msgs/user - Caused the following error: Author does not exist, logged
> Connection id ""0HMQ9FVV2HJ6J"", Request id ""0HMQ9FVV2HJ6J:00000001"": An unhandled e
> HTTP "POST" "/simulator/msgs/Jesse Chiprean" responded 500 in 17.4652 ms

Logs

> HTTP "POST" "/simulator/msgs/Tana Merrick" responded 204 in 136.2327 ms
> POST: Simulator/msgs/user - Successfully created message at "13:17:53"
> UpdatingLatest - Update of latest value was successfully, logged at "13:17:52"
> Extists "Tana Merrick" called at "13:17:52"
> HTTP "POST" "/simulator/msgs/Mora Klingberg" responded 204 in 152.2621 ms
> POST: Simulator/msgs/user - Successfully created message at "13:17:52"
> UpdatingLatest - Update of latest value was successfully, logged at "13:17:52"
> Extists "Mora Klingberg" called at "13:17:52"
> UpdatingLatest - Update of latest value was successfully, logged at "13:17:52"
> Extists "Felipe Hontz" called at "13:17:52"
> HTTP "POST" "/simulator/fllws/Winifred Peshek" responded 204 in 230.2315 ms
> POST: Simulator/fllws/user - Follow to "Winifred Peshek" was successfully, logged at "
> Extists "Noel Mccasland" called at "13:17:52"
> UpdatingLatest - Update of latest value was successfully, logged at "13:17:52"
> HTTP "POST" "/simulator/msgs/Elvin Perz" responded 204 in 140.0585 ms
> POST: Simulator/msgs/user - Successfully created message at "13:17:52"
> UpdatingLatest - Update of latest value was successfully, logged at "13:17:52"
> Extists "Elvin Perz" called at "13:17:52"
> HTTP "POST" "/simulator/fllws/Scotty Stright" responded 204 in 144.8576 ms
> POST: Simulator/fllws/user - Follow to "Scotty Stright" was successfully, logged at "1
```

(a) The logs showing the API calls registered



```
Errors

> Connection id ""0HMQGHHD8H4FM"", Request id ""0HMQGHHD8H4FM:00000001"": An unhandled e
> HTTP "GET" "/public" responded 500 in 15021.4859 ms
> An exception was thrown while deserializing the token.
> Connection id ""0HMQ9FVV39HA1"", Request id ""0HMQ9FVV39HA1:00000001"": An unhandled e
> HTTP "POST" "/simulator/fllws/Kerry Hindbaugh" responded 500 in 2.2450 ms
> An exception was thrown while deserializing the token.
> POST: Simulator/msgs/user - Caused the following error: Author does not exist, logged
> Connection id ""0HMQ9FVV3941N"", Request id ""0HMQ9FVV3941N:00000001"": An unhandled e
> HTTP "POST" "/simulator/fllws/Marco Fierra" responded 500 in 18.7903 ms
> HTTP "POST" "/simulator/fllws/Rozella Janzen" responded 500 in 1.6054 ms
> Connection id ""0HMQ9FVV3941M"", Request id ""0HMQ9FVV3941M:00000001"": An unhandled e

Logs

> HTTP "GET" "/metrics" responded 200 in 23.7405 ms
> HTTP "GET" "/metrics" responded 200 in 34.8793 ms
> Connection id ""0HMQGHHD8H4FM"", Request id ""0HMQGHHD8H4FM:00000001"": An unhandled e
> HTTP "GET" "/public" responded 500 in 15021.4859 ms
> HTTP "GET" "/metrics" responded 200 in 54.0279 ms
> HTTP "GET" "/metrics" responded 200 in 17.8387 ms
> HTTP "GET" "/metrics" responded 200 in 14.3940 ms
> MessageRepository: Get - Getting messages for 1, called at "12:57:58"
> Error unprotecting the session cookie.
> HTTP "GET" "/login" responded 200 in 3.8157 ms
> GetCurrent user at "12:57:57"
> Error unprotecting the session cookie.
> HTTP "GET" "/metrics" responded 200 in 13.8731 ms
> MessageRepository: Get - Getting messages for 1, called at "12:57:53"
> Error unprotecting the session cookie.
> HTTP "GET" "/login" responded 200 in 20.7382 ms
> GetCurrent user at "12:57:53"
> Error unprotecting the session cookie.
> HTTP "GET" "/register" responded 200 in 42.6921 ms
> GetCurrent user at "12:57:52"
```

(b) The logs when we took down the database

Figure 4: Logs.

2.6 Scaling And Load Balancing

Our load balancing uses different strategies for a 'normal' user and the course simulator. When a request is made to the */simulator* endpoint, it uses the round-robin approach, which means the request is sent to any of the active servers, distributing the simulator requests across all the active servers. However, when a connection is established through the website, we are using an IP hash approach. This means that the same user is always directed to the same server, which ensures that the user session is preserved.

Scaling can be done horizontally by adding more nodes to the docker swarm. In our deployment we were using three droplets of the second lowest tier on DigitalOcean. So if scaling becomes necessary it will make more sense to start with vertical scaling by provisioning some virtual machines with more dedicated memory and processing power.

2.7 Security Assessment

We decided to do a security assessment to discover potential security flaws in our system implementation. This was done through the following process:

1. Identifying the different components of the system.

2. Identify which component posed potential security breaches and what these were for each component.
3. Constructing risk scenarios on each potential security breach.
4. Performing a risk analysis on the different scenarios.
5. Running a preemptive pen test on our system.

	Insignificant	Negligible	Moderate	Extensive	Significant
Almost certain	🟨 Scenario E	🟧 Scenario I	🟥	🟥	🟥
Likely	🟩	🟨	🟧	🟥 Scenario B	🟥
Possible	🟩	🟩 Scenario C	🟨	🟧	🟧
Unlikely	🟦	🟦	🟩	🟨 Scenario A	🟧
Rare	🟦	🟦	🟦 Scenario G	🟩 Scenario F	🟨 Scenarios D, H

Figure 5: Security assessment of different risk scenarios (see appendix 4.1 for details on each scenario).

A report of our entire process can be found in appendix 4.1. The security assessment matrix can be seen in figure 5. The table shows us that the most critical scenario is (B) where an attacker performs a DDOS attack to make our server unresponsive. This would make our service useless and could be very bad for business. However, there is no imminent way to mitigate this other than scaling the application, having a load balancer, or paying for DDOS protection. Our system has a load balancer, but the other options were deemed to be outside this project's scope.

While the security assessment has helped us realize which parts of the system are most vulnerable and how they might be attacked, it does not guarantee that our system is not exploitable or that every possible scenario is considered. The most typical way to discover security flaws is when an attacker takes advantage of them. The security assessment did, however, give us confidence that the most ordinary tools for discovering weaknesses didn't find anything on our system.

We have also used Snyk for checking vulnerabilities in the system. Based on our findings, changing any parts of the system was unnecessary.

2.8 AI assistants

In this project, we used ChatGPT. It did not write any of our source code, but it was helpful for error handling. When googling an error, finding the correct stack overflow response can sometimes be difficult, whereas, with ChatGPT, we could paste the error message, and it could often point us in the right direction. It has also been useful for setting up Nginx and docker swarm. For instance, ChatGPT provided help on how to use IP hash for keeping sessions while using load balancers.

3 Lessons Learned Perspective

3.1 Evolution and refactoring

Initially, we created a single-page application with Blazor for the frontend of our app, which we spent a lot of hours working on the first weeks. However we realized that creating and maintaining a SPA would be much more work, we estimated it would take at least twice the amount of code. Therefore we switched to server side rendering with Razor (commit 6f1f4c0) which had the upside that it is a much more similar approach to the legacy Flask application, which made it easier to implement the features. What we took away from this is that it is important not to over-engineer a system and choose unnecessary complexity just because it is more modern.

3.2 Operations

We discovered the importance of knowledge sharing. We quickly found that when we did not share what we had done to the project since last week, it took time for all members to have an overview of the entire system. To solve this issue, we started using retrospectives as a way of sharing knowledge every Tuesday before the next week's task.

3.3 Value of Logging

This project was the first time any of us had implemented a system with proper logging, which we found extremely valuable.

An example of when our logs were helpful was after we had the previously mentioned 12 hours of downtime due to our self-managed database crashing. We then had error logs for when the simulator attempted to post a message or follow with a user that had never been created due to the downtime. We then extracted all the usernames from the error logs and created a script to create the missing users in the system.

Another instance when our logs were helpful was when the public timeline page of our application began to feel very slow. We could see in our logs that the query which found the newest messages took approximately 1,000 ms to return. The fix was easy, we created an index on the date which the query sorts by (commit d0996e8), suddenly the page loading felt much faster. We could assert this in the logs with the query now taking around 5 ms to return.

3.4 Difficulties of updating a live system

Initially, we had no docker swarm and no reverse proxy. At a moment, we wanted to implement HTTPS connection for our system. The MiniTwit application was listening to port 80 on our live DigitalOcean droplet which now had to be freed for the Nginx reverse proxy which applied the SSL certificate. What we decided to do was quickly take the system down and swap the ports. However, it took many attempts to get the HTTPS connection working, causing us to take the system down multiple times. Summed up, this probably caused around 3 hours of outage. This would not be acceptable for a real world system and it could have been a violation of our SLA.

In retrospective what we could have done was redeployed the entire system on different infrastructure, debugged there, and then pointed our DigitalOcean reserved IP to the new system without causing outage. Overall the takeaway for our group is how difficult it is to update a live system. Luckily, the tests in our CI/CD pipeline assured that code that would break the system was never deployed during the course.

4 Appendix

4.1 Security Assessment

A. Risk Identification Identify assets (e.g. web application)

1. Webserver (deployed via DigitalOcean)
2. Prometheus
3. Managed Database
4. Grafana service
5. Loki

Identify threat sources (e.g. SQL injection)

1. Webserver (deployed via DigitalOcean)
   (a) Unsanitized input (sql injections)
   (b) DDOS
   (c) Brute force attacks on login
   (d) SSH brute force attack on login
2. Prometheus
   (a) Publicly exposed metrics
3. Managed Database
   (a) No/bad authorization/authentication
   (b) Entire database could be exposed/deleted
4. Grafana service
   (a) No/bad authorization/authentication
   (b) Public dashboards
   (c) Editable dashboards (not read-only)
5. Loki
   (a) Can get information about username of users

Construct risk scenarios (e.g. Attacker performs SQL injection on web application to download sensitive user data)

A Attacker performs SQL injection on web application to download sensitive user data

B Attacker performs a DDOS attack to make our server unresponsive

C Attacker brute forces a user's password and gains control over their profile

D Attacker brute forces SSH credentials and gets full access over the web server

E Attacker reads the public Prometheus metrics, and gains business insights on our service

F Attacker gets control over Prometheus and is able to misrepresent metrics. This could mask spikes or irregular patterns in our monitoring.

G Attacker is able to get access to our Grafana service, and can see/delete all our monitoring

H Attacker is able to get access to our managed database, and downloads/deletes all our data

I Attacker gets access to metrics

B. Risk Analysis Translated from english to danish: Vurderinger er lavet baseret på sandsynligheden for, at et angreb ville lykkedes Certain, likely, possible, unlikely, rare Insignificant, Negligible, Marginal, Critical, Catastrophic Determine likelihood and impact

A Likelihood: Unlikely, Severity: Extensive

B Likelihood: Likely, Severity: Extensive

C Likelihood: Possible, Severity: Negligible

D Likelihood: Rare, Severity: Significant

E Likelihood: Possible, Severity: Insignificant (ITU IP is whitelisted)

F Likelihood: Rare, Severity: Extensive

G Likelihood: Rare, Severity: Moderate

H Likelihood: Rare, Severity: Significant

I Likelihood: Almost Certain, Severity: Negligible

Use a Risk Matrix to prioritize risk of scenarios in current state of program Discuss what are you going to do about each of the scenarios

Ud fra vores matrix kan vi se at de vigtigste scenarier at gøre noget ved er B

For at fikse B kan vi: Load balancing