MiniTwit Project
DevOps, Software Evolution and Software Maintenance
IT University of Copenhagen
Course code: BSDSESM1KU

Group F

Name	**E-mail**
Jacob Møller Jensen	jacj@itu.dk
Marcus Sebastian Emil Holmgaard	seho@itu.dk
Mikkel Møller Jensen	momj@itu.dk
Rune Engelbrecht Henriksen	rhen@itu.dk

1st of June 2022

Contents

1 System's perspective

1.1 Design

When faced with the initial task of refactoring the MiniTwit system, we agreed to choose a stack consisting of technologies at least one member of the group had experience with. This enabled us to both have the advantage of working with something familiar and gave the opportunity to learn something new.

We chose React with Typescript for the frontend and when making the MiniTwit application we prioritized having the same functionality as the existing Flask application. As a result of this, the page setup and styling is identical to the original application. We have used C# for our backend. The data access technology used is Entity Framework Core.

Our Database is a Managed Azure SQL database hosted on Azure. Originally we had Azure SQL Edge running in Docker but as the database data is important to save persistently we switched to Azure SQL running on a Microsoft Azure server. Azure allowed us to enable automatic tuning of the different tables that improved performance drastically (Indexation of tables). More on these topics in future sections.

1.2 Architecture

This sections aims to describe the architecture of our system using Christensens 3+1 model [1] by showing:

- Requirements
- Module viewpoint
- Component & connectors viewpoint
- Allocation / deployment viewpoint

The system is separated into a three-tier architecture which is shown in Figure 1. The diagram includes the course simulator (which is not in our control) as a component as it has a significant impact on the system.

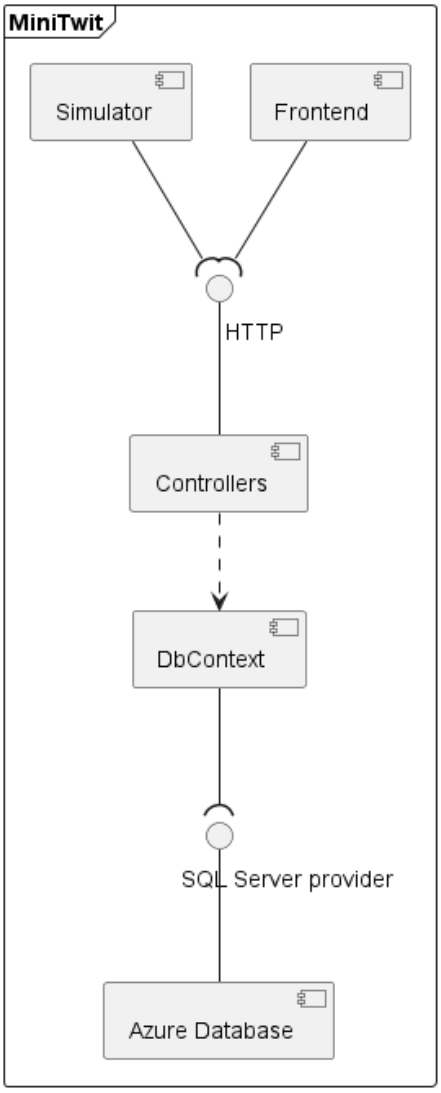

Figure 1: Overall architecture

1.2.1 Requirements

This section concerns the "+1" part of the 3+1 model i.e. the architectural requirements. Though they referenced as "scenario-based" and "quality attribute-based" in the literature, this report will use the terms functional and non-functional requirements respectively.

Functional requirements

1. The system must provide the same functionality as the original MiniTwit system given at the introduction of the course
2. The system must expose an API which adheres to the requirements of [5]

Non-functional requirements

1. The systems backend must be written in C#
2. The systems frontend must be written in React with Typescript
3. The system must be containerized with Docker ensuring isolation and portability

4. The system must be deployed on DigitalOcean
5. The system must be performant, scaleable and reliable in accordance to the SLA, Appendix A.

1.2.2 Module viewpoint

At the highest level, the system is divided into backend and frontend.

The following package diagram gives a high-level overview of the backend's file structure and dependencies.

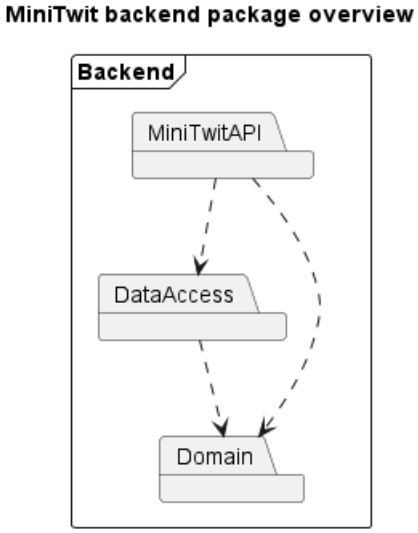

Figure 2: Backend Package overview showing how data is accessed in the system.

- **Domain:** contains the business entities of the system i.e. model and DTO classes representing users, tweets and followers.
- **DataAccess:** contains the class that handles sessions and communication with the database.
- **MiniTwitAPI:** houses the controller classes that make up both the public and internal APIs of the system, which are two separate units.

A complete package diagram for the backend, including all relevant artifacts, etc., can be seen in figure 3.

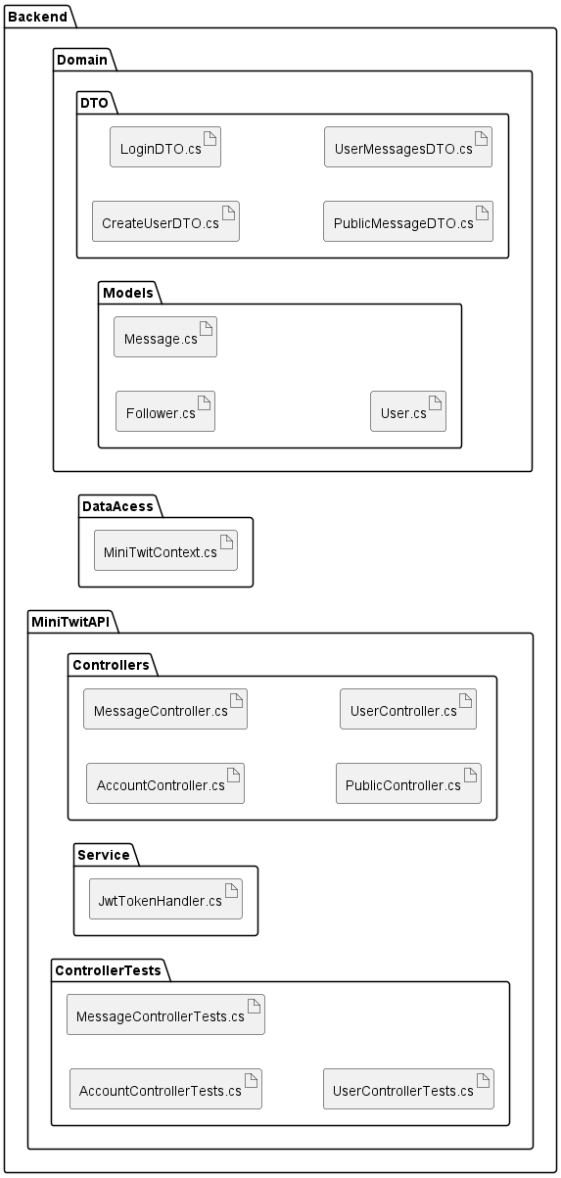

Figure 3: Package diagram showing our backend artifacts, dependencies omitted for clarity

The next part of the program is the frontend, of which a package diagram is shown in figure 4. Only the "src" package contains files of interest i.e. that we have written, as opposed to configuration or boilerplate files and as such the contents of "public", "nginx" and the root folder is not shown in the diagram. The artifacts represent the individual pages of the website as well as subcomponents, media, stylesheets and some configuration-files that are React-specific. The naming of the pages reflects the naming from the original page structure provided.

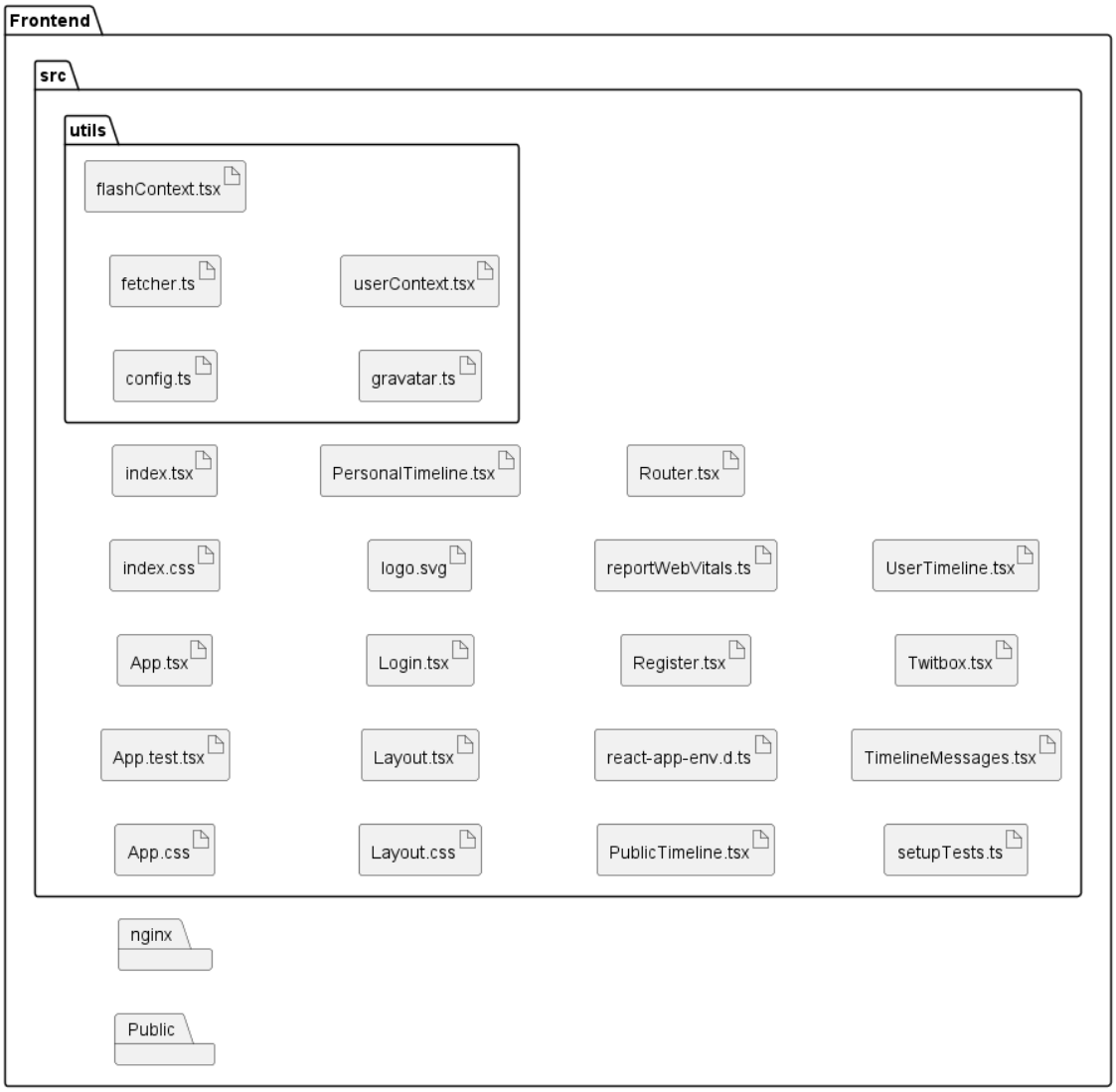

Figure 4: Package diagram of the frontend folder, some folders omitted for clarity

1.2.3 Components & connectors viewpoint

Figure 5 shows the six components e.g. Docker containers of our primary DigitalOcean droplet and how they interact with each other at runtime. In the diagram, the "MiniTwit backend" component contains two separate API subcomponents. The actual Docker container uses the same port to handle both API requests from the simulator and from the website but we have chosen to show it as two components here to indicate that they serve two different purposes.

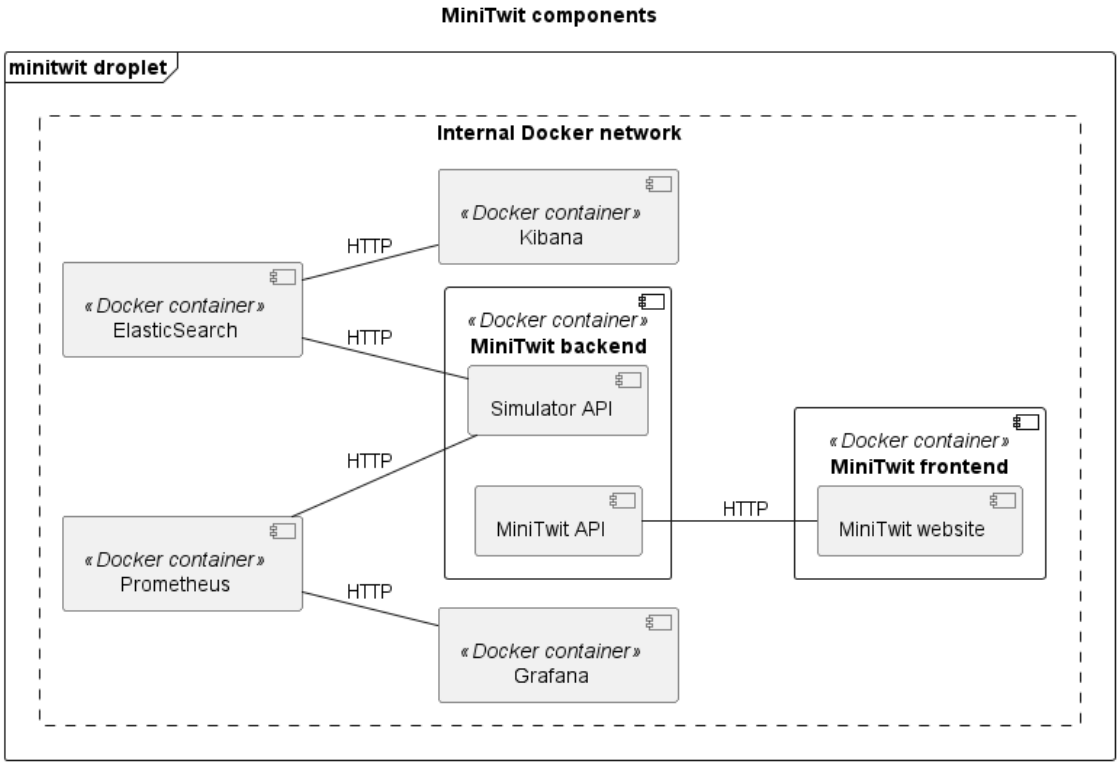

Figure 5: Component diagram of our primary DigitalOcean droplet, Docker volumes omitted

1.2.4 Allocation / deployment viewpoint

As seen in Figure 6, the system consists of a managed database on Microsoft's Azure platform as well as two separate droplets on DigitalOcean. The two droplets constitute our scaling setup which is described in depth in a future section.

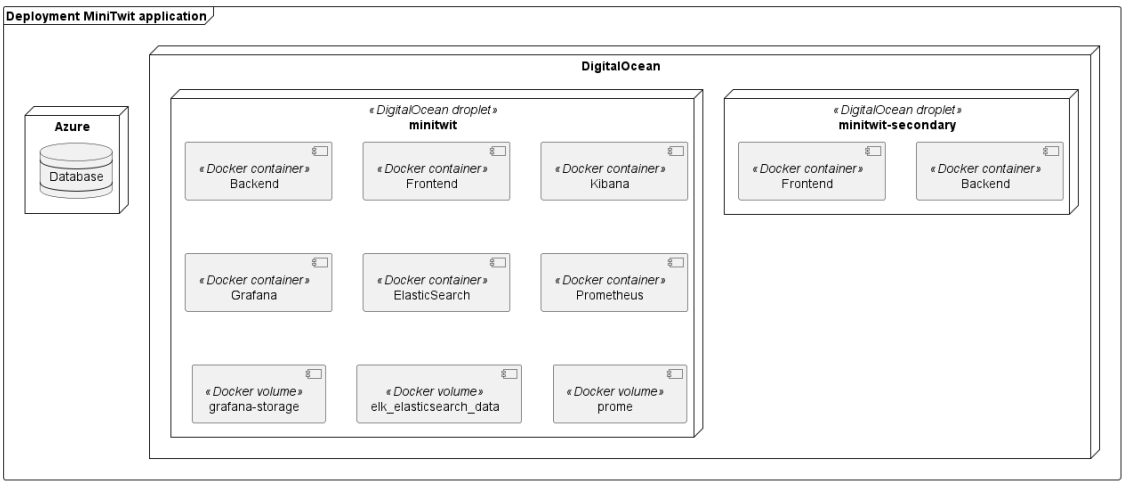

Figure 6: Deployment diagram illustrating the overall deployment view of our MiniTwit system including Docker volumes for monitoring and logging

Both droplets are hosted on servers located in Frankfurt, Germany and have the following specs

	Primary	Secondary
vCPU	1	1
Memory	2 GB	1 GB
Disc Space	50 GB	25 GB

They do not have the same specifications as they do not handle the same tasks but this will be elaborated in a later section.

1.3 Dependencies - technologies and tools

The following is a table of the dependencies and tools that the project relies on as well as a brief comment on what they're used for.

Name	Responsibility
Git	Version control
GitHub	Repository management
C# / ASP.NET CORE / .NET	Used for our backend
Typescript	Used for React frontend
Azure SQL Database	The database system used for our MiniTwit application
Serilog	A library used for logging in the backend
Prometheus	Used for system monitoring
Grafana	Used to display monitoring information
ElasticSearch	Used to store the systems logs
Kibana	Used to present the aggregated logs
Docker	Used for containerizing the system
Ubuntu	The OS that our droplets run on
React	JavaScript library for building the website
Nginx	A web server that runs React
Docker-compose	Handling multiple docker containers
Docker Hub	Storing docker images
Sonarcloud	Static analysis
DigitalOcean	Provider of infrastructure
Vagrant	Provisioning of virtual machines
CircleCI	Continous integration pipeline
Better Code Hub	Provides static analysis of the code
Keepalived	High-availability / scaling
Python	Used in a script for switching floating IP on DigitalOcean
Bash	Used as terminal for the developers and for scripts.
SSMPT	Used to configure the mail server.
MailUtils	Used to send a mail directly from a script using SSMTP server settings.
Gmail	Used as a central mailbox that forwards to our individual ITU mails

1.4 Important interactions of subsystems

The most important interactions of subsystems in our MiniTwit application are the following

- HTTP communication between frontend and backend
- HTTP communication between backend and the simulator
- HTTP communication between backend and Prometheus
- HTTP communication between Prometheus and Grafana

- HTTP communication between backend and ElasticSearch
- HTTP communication between ElasticSearch and Kibana
- Communication between the backend and the Azure database via EF Core

The sequence diagram in figure 7 shows the interaction between frontend, backend and database when a user posts a new message.
Most interactions between frontend and backend as well as between the backend and simulator follow the same flow, and as such these will not be depicted.

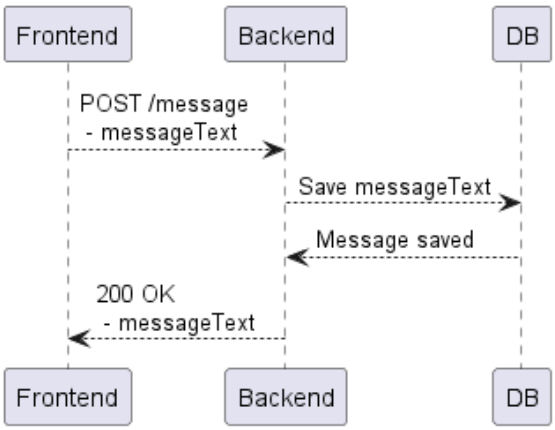

Figure 7: Sequence diagram depicting the interaction between frontend, backend and database when a user posts a new message

Figure 8 shows an example of an interaction between the simulator, backend, ElasticSearch and database. In particular it depicts the sequence of logging and database calls that occur between a request to post a message being made and the response being returned to the simulator.

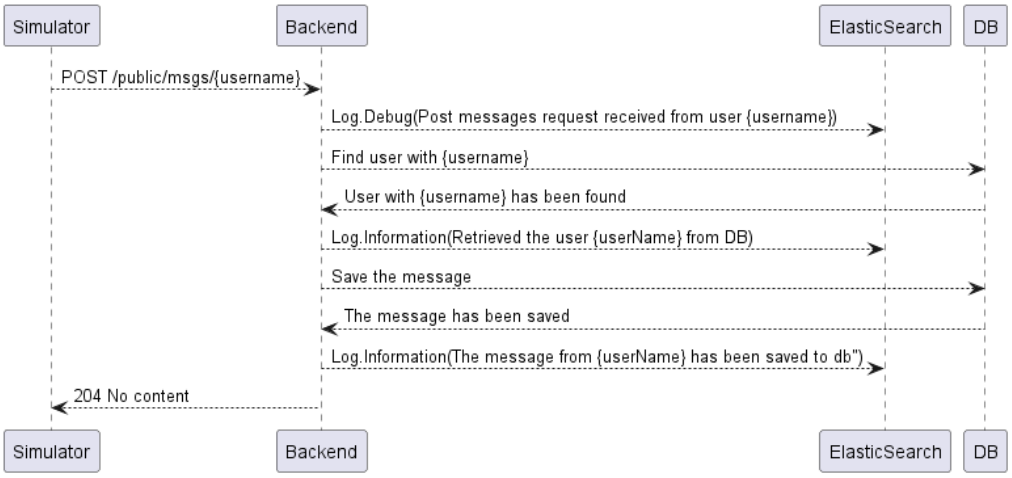

Figure 8: Sequence diagram depicting the interactions between the simulator, our simulator API, ElasticSearch and database that happen when the simulator posts a new message from a user

This final sequence diagram in figure 9 shows the interaction between simulator, backend, database, logging and monitoring when a user is created, including the two possible execution paths determined by whether the user already exists or not. Note that Kibana and Grafana will regularly retrieve data from ElasticSearch and Prometheus respectively but these interactions have been omitted along with any responses that ElasticSearch and Prometheus might give.

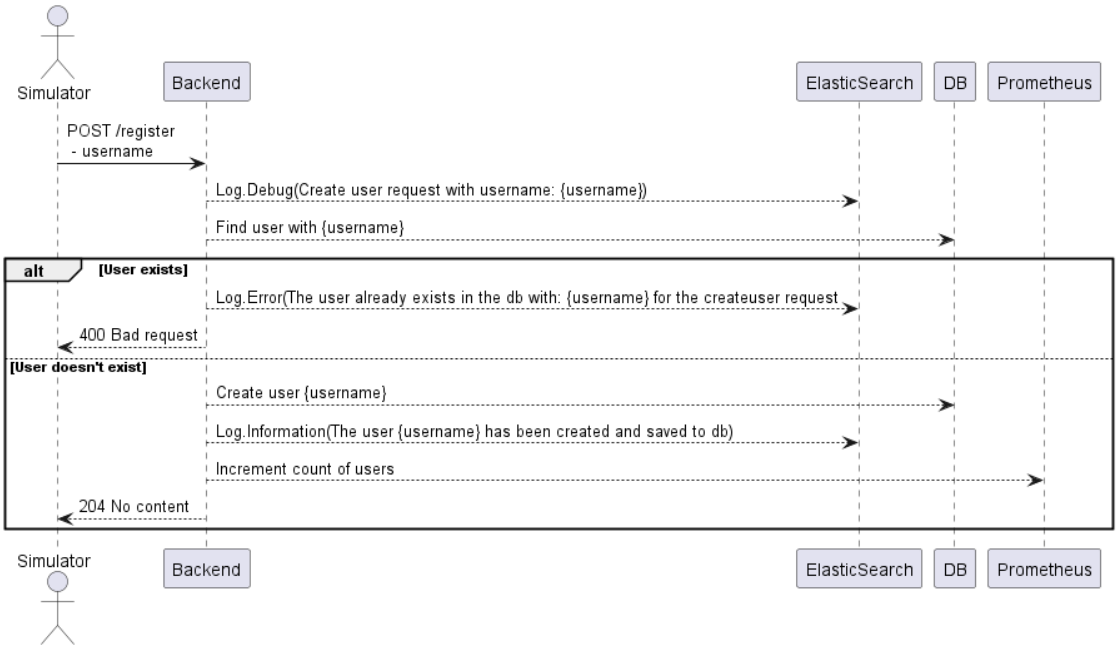

Figure 9: Sequence diagram depicting the interactions between the simulator, our simulator API, ElasticSearch and database that happen when the simulator posts a new message from a user

1.5 Current state of the system

The current system has an improved availability with the use of Keepalived. The system is easier to maintain, because the system now uses static analysis and diagrams that makes the development of the system more smooth. Results from the static analysis tool bettercodehub.com can be seen in Appendix C. Containerization of the system creates a uniform and isolated development environment. Moreover, a database abstraction layer reduces vendor-specific coupling and improves the systems security against e.g. SQL injection. The system uses continues integration pipelines to increase the reliability, as well as continuous delivery that optimises the deployment of the system for the developers. The system has monitoring and logging providing insights into the running system.
The following are features and technologies that we would have liked to implement but did not.

1. Automatic release on deployment
2. Running tests as part of CI (and terminating on test failure)
3. Rolling updates (and including secondary server in CI)
4. Provisioning of VM for secondary server
5. Infrastructure as code with Terraform

1.6 License compatibility

The license chosen for this project is: **GNU GPL-3.0**. A list of the licenses for direct dependencies in the project can be seen in appendix B. According to Wheeler[6] the projects licence is compatible with the following licenses:

- MIT
- Apache 2.0
- GNU GPL-3.0

- AGPL v2

The license of the project should therefore be compatible with the licenses of its direct dependencies.

2 Process' perspective

2.1 Developer interaction & Team organization

In week 2 of the project we agreed on how we wanted to collaborate and described it in CONTRIBUTE.md.

2.1.1 Discord

It was decided to use the communication platform *Discord*, which is great for facilitating discussions. The main communication line was in text channels. Voice channels were used to do collaboration sessions or further explain and share thoughts on the subjects from the text channels. During these online collaboration sessions we used screen sharing as a way to program in pairs.

2.1.2 Physical meet-ups

On Tuesdays after lecture, the developers would meet-up at the weekly exercise session and focus on the current weeks' workload. This was done so every team member could get insight in what was happening and could share thoughts and troubles that might occur during development of the project.

2.2 Stages and tools in CI/CD chain

2.2.1 Tools

A variety of different tools and technologies were used in the *Continuous Integration* (CI) chain. The list below, summarizes tools and technologies used in the pipeline.

- Circle CI is a CI/CD service, and was used for the build server service.
- Docker Containers and Docker hub as a public artifact registry.
- SonarCloud used for static analysis.
- BetterCodeHub evaluates the GitHub code base against 10 software engineering guidelines.
- Digital Ocean a cloud server provider.
- GitHub Actions is used for building and linting the React project as well as compiling and generating a PDF report.

The configuration file for the CircleCI part of the pipeline is .circleci/config.yml and the configuration files for GitHub actions are found in .github/workflows.

Static analysis

Included in the pipeline is also scanning of the frontend with SonarCloud and the backend with docker scan[1].
Moreover Better Code Hub[2] is used to scan the GitHub repository against 10 engineering guidelines devised by the authority in software quality, Software Improvement Group (SIG).

[1] https://docs.docker.com/engine/scan/
[2] https://bettercodehub.com/

2.2.2 Stages

The projects' CI/CD setup is implemented utilizing CircleCI and includes 3 primary stages:

1. Build docker image with frontend, scan the frontend, and push to Docker Hub.
2. Build docker image backend, scan the image, and push to Docker Hub
3. SSH into our DigitalOcean droplet to stop running containers, pull newest images from Docker Hub, curl the newest Docker Compose file and then start it all again with the Docker Compose file (docker-compose-prod.yml)

The diagram in figure 10 gives an overview of the CI pipeline and how we developers as well as the different platforms and technologies interact.

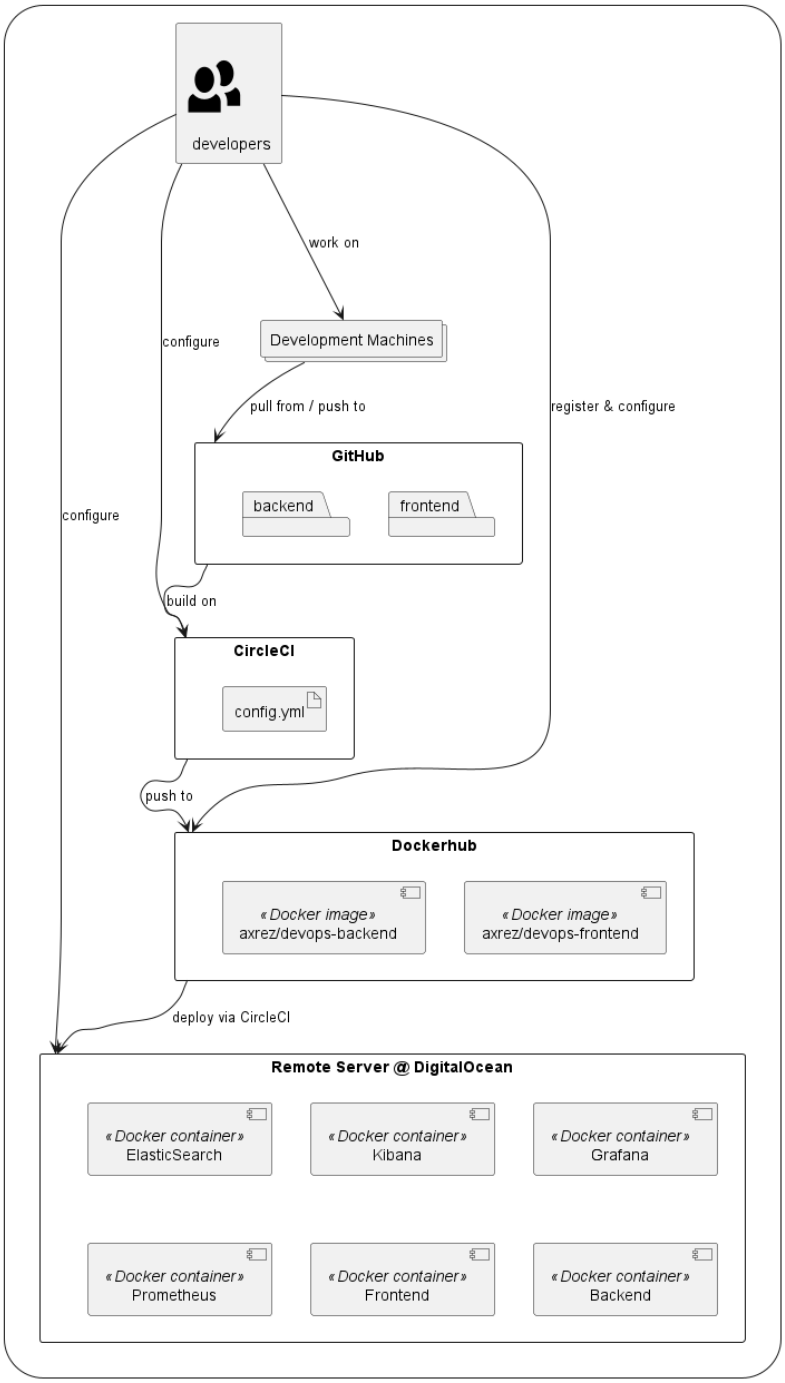

Figure 10: Overview of the CI pipeline

2.2.3 Environment variable

The pipeline uses environment variables both in CircleCI and directly on our droplet to properly manage secrets like Docker Hub credentials, the database connection string and credentials for third party tools like Grafana, Elasticsearch and Kibana.

2.2.4 Future plans

At the moment, the CI/CD chain only deploys onto one droplet (primary) as seen in stage 3. Therefore, the project does not support for Continuous Deployment onto multiple droplets, this

will have impact on an implementation with rolling updates, depending on the configuration. The thought was to deploy to a secondary droplet first and after that goes *live*, redirect traffic from the primary droplet to the secondary. Shut-down the primary droplet and then deploy onto it, and after its live again redirect traffic back to the primary.

2.3 Repository organization

We have used a mono-repository setup for this project, where the root of the repository contains files related to configuration like Dockerfiles, Vagrantfiles etc. as well as two folders designated for frontend and backend code respectively.

2.4 Applied branching strategy

The branching strategy used is feature branching [4]. The branches is organised in a main branch that contains the code base for the current running system. Main is also the target of our CI pipeline.
Feature branches are then branched out from main. When a feature is completed a pull request is opened, which needs to be reviewed and approved by a team member before it can get merged into main.

2.5 Applied development process and tools supporting it

During the project we have actively used GitHub issues to keep track of tasks both related to weekly assignments but also bugs, refactorings etc. that we have discovered or wanted to do ourselves. The issues is assigned labels to organize the priority, where the label *important* is the most urgent and should be prioritised over a label like *Nice-to-have*. To organise work on the issues the built-in Kanban board feature that GitHub provides is used during the development. This board helps the team with organizing the development, where the board shows what is being worked on by who. The board can be seen in Appendix D.

2.6 Monitoring

To monitor the system the Prometheus package for .NET, *prometheus-net*, is used. Prometheus exposes some metrics from an ASP.net application by default. These metrics includes number of HTTP request in progress, number of received HTTP requests in total and duration of HTTP requests.
A Grafana dashboard from the template: *ASP.NET core - controller summary* is used to visualize the technical monitoring information that Prometheus gathers from our system. The dashboard shows the following:

Business information

- Amount of users registered in the system

Technical information

- Request received
- Error rate
- Total requests/s
- Request duration
- Number of requests in progress

Docker volumes are used to persist data from both Prometheus and Grafana.
The Grafana dashboard can be accessed at http://157.245.27.14:3000/login with the credentials provided during the course.
The raw monitoring metrics can be accesses through Prometheus at http://157.245.27.14:9090/.

2.7 Logging

Our logging setup deviates from the popular ELK stack proposed in the course material and uses the following technologies. Instead of using Logstash the package Serilog is used in the system to aggregate logs.

- Serilog[3]
- ElasticSearch
- Kibana

Serilog is imported as a dependency in the .NET project while both ElasticSearch and Kibana runs in their own respective Docker containers with volumes associated for persistence. The logs can have different log levels such as: debug, information, error, warning - which can be used for filtering. The logging is centered around the simulator API and as such does not include e.g. the API used by the MiniTwit website.
The system logs can be accessed via Kibana at http://157.245.27.14:5601 with the credentials provided during the course.

2.8 Security assessment

Based on our security assessment we have taken precautionary steps which concretely has resulted in enabling DigitalOcean Two-Factor Authentication, enabling Dependabot and investigation of how we could add API Authentication. We have also talked about Security in relation to DDoS protection and developer-device security. The full Security assessment can be found in appendix E.

2.9 Scaling and load balancing

We chose to implement high-availability by using the configuration with Keepalived discussed in class, though with some modifications as the article[3] provided is deprecated. We have created our own updated version of Keepalived using Ubuntu 22 for documentation of the process and future use of others [4].
This means that we have two droplets on DigitalOcean - one which is our Primary droplet that we have had since the beginning of the project and one secondary which is activated if the primary crashes. A floating IP is used to support the setup.
The primary droplet contains all monitoring and logging and the secondary does not.
We agreed that, in case of an incident, the most important task is to be able to keep on serving clients, which is possible with this setup. We also agreed that the logs relevant to an incident e.g. before a crash, will come from the primary droplet and thus still be aggregated in this case.
If the secondary droplet goes active, we have an emergency situation that will need to be handled quickly and thus the lack of logging and monitoring is considered non-crucial.
Another approach to this would be to have a three droplet setup where monitoring and logging is located on the third droplet. This would allow the primary and secondary droplet to log to one central place. We chose not to do this as the third droplet would still be a single point of failure and because of what we agreed in regards to handling emergencies.
We have created alerts by email using sSMTP [5] and Mailutils [6] which executes when the secondary droplet registers that the primary droplet is down (and therefore takes the floating IP). This allows us to be informed about this situation quickly and act upon it to recover and bring the service back to the primary droplet.
The email alerts are sent to all developers and can be seen in Appendix F.

[3] https://serilog.net/
[4] https://github.com/JacobMoller/Keepalived-DigitalOcean-Ubuntu-22.04/
[5] https://wiki.debian.org/sSMTP
[6] https://mailutils.org/

3 Lessons learned perspective

The following problems taught the lesson of taking the time to do things right the first time (or at least when you recognize that a problem exists) is invaluable compared to continue working with incomplete and ineffective workarounds. e.g. Being eager to finish tasks fast and thus not prioritizing attention to important details.

3.1 Hacking directly on the server

For quite some time, until commit e455, our CI pipeline was not correctly set up resulting in manual labor being necessary when we were to deploy changes. The misconfiguration stemmed from the pipeline not updating the Docker Compose file on the droplet and as such we had to manually stop all containers, remove images and then run the docker-compose file.
Doing this manual task is not hard in itself but problems arose when things did not work in the first go. In these scenarios group members would take the path of least resistance and make changes to e.g. the Docker setup directly on the server making it hard to track which changes actually worked and then correctly adding them to version control afterwards. A problem that we faced because of this was one time where we actually lost a working change because we got confused about which changes made where had resulted in the system working correctly.
This happened when we decided to try and make our database not publicly exposed (discussed in the next section). We are not entirely sure what happened, but it had something to do with us manually changing configurations directly on the server, not correctly committing the same changes to version control and on top of that, not knowing that the CI pipeline was not correctly configured to run the newest Docker images.
This really drove home the lesson that automation, configuration and reproduceability beats manual work, because it is too easy to forget something and not be able to get repeated results.

3.2 Attacks on database and migrating to cloud

As mentioned in a previous section, we used a local Azure SQL Edge database running in a Docker container on our droplet for the first roughly seven weeks (migration to Azure 15/3, commit 64df). With this database we experienced regular attempts to brute-force the password to the superuser of the database, resulting in the Docker container with the database crashing roughly every 6-12 hours. Logs from one of the attempts is seen in appendix G. The downtime caused by these crashes led to missing requests from the simulator, and in general decreased the reliability and availability of the system. We first solved the problem of attacks by not exposing our database publicly with commit 2a02. However, It the attacks also made us fear losing all of the data, which led to creation of (manual) backups, but having a local db was still hard to maintain and not without risk.

Migrating to the cloud

For the reasons specified above we decided to migrate to a database hosted with Microsoft Azure. This increased the systems maintainability and reliabilty, as the managed db provided features such as automatic backups and security.

3.3 Handling secrets

For the first weeks of the project, secrets was not handled correctly (database connection strings) as they were committed to our GitHub repository.
As described above, malicious people systematically tried to access our database and thus assuming that there is a real risk of someone scanning public repositories for secrets that can be used to exploit or damage systems, made sense.
Seen from another group which had gotten their data stolen because they did not password protect their database. With the somewhat careless way we handled secrets, the same could just as well have happened to us - which again relates to the point about doing things right the first time.

3.4 msgs endpoint becoming very slow

Towards the end of the simulator period it was discovered that our /msgs endpoint was becoming very slow. We identified that the way we had implemented it was ineffective, as we:

1. Sorted them according to the date they were entered
2. Picked the top 100 of them to show on the site

With the amount of posts reaching almost 3 million (2.867.986) near the end, it was becoming increasingly slow.
If two of us tried to access the endpoint at the same time, the databases compute utilization would increase to around 100% and make the system unresponsive.
To resolve this issue we opted to sort by the id instead, and turn on automatic indexing on the database, which drastically improved the speed of retrieval.

4 Conclusion

There was a great focus on incremental updates and availability as the public API was constantly used. When deploying a new update, throughout the project, the group became very aware on the logging and testing the live system status. This also resulted in doing rollbacks for availability. Given the organization and size of our team and the way we have worked together, we strived for a culture of continual experimentation, sharing of knowledge and learning [2]. It has given the freedom and trust to try out different solutions in an attempt to both gain knowledge and find the best possible solution.
Nonetheless, having the responsibility for maintenance, refactoring and evolution has given a new view of the software development process. Prior to this project nobody in the team had tried anything else than rushing to get a project done, handing it in and then not having to deal with it anymore. Here, this project showed the value of minimizing work in progress and incremental upgrades with logging, monitoring etc. which increased the observability of the system, therefore allowing for easier troubleshooting.

References

[1] Christensen et al. "An Approach to Software Architecture Description Using UML Revision 2.0". In: (June 2007). URL: https://pure.au.dk/portal/files/15565758/christensen-corry-marius-2007.pdf.

[2] Gene Kim et al. Distributed by lecturer so author and publishing year is taken from Google. Oct. 2016. URL: https://ituniversity-my.sharepoint.com/personal/ropf_itu_dk/_layouts/15/onedrive.aspx?id=%5C%2Fpersonal%5C%2Fropf%5C%5Fitu%5C%5Fdk%5C%2FDocuments%5C%2FDevOps%5C%2C%5C%20SW%5C%20Evolution%5C%20and%5C%20SW%5C%20Maintenance%5C%2FDevOps%5C%20Handbook%5C%20Part%5C%201%5C%2Epdf&parent=%5C%2Fpersonal%5C%2Fropf%5C%5Fitu%5C%5Fdk%5C%2FDocuments%5C%2FDevOps%5C%2C%5C%20SW%5C%20Evolution%5C%20and%5C%20SW%5C%20Maintenance&ga=1.

[3] Justin Ellingwood. *How To Set Up Highly Available Web Servers with Keepalived and Floating IPs on Ubuntu 14.04*. Oct. 2015. URL: https://www.digitalocean.com/community/tutorials/how-to-set-up-highly-available-web-servers-with-keepalived-and-floating-ips-on-ubuntu-14-04.

[4] Martin Fowler. May 2020. URL: https://martinfowler.com/bliki/FeatureBranch.html.

[5] Helge Pfeiffer & Mircea Lungu. *minitwit_sim_api.py*. URL: https://github.com/itu-devops/lecture_notes/blob/master/sessions/session_03/API%5C%20Spec/minitwit_sim_api.py.

[6] David A. Wheeler. *The Free-Libre / Open Source Software (FLOSS) License Slide*. 2017. URL: https://dwheeler.com/essays/floss-license-slide.html.

A Service-Level Agreement

Service Level Agreement

Our Monthly Uptime Percentage (hereafter abbreviated as MUP) describes the percentage of time where the server responds to requests.

Covered Service	MUP
Group F Minitwit	98%

If Group F does not adhere to the SLA's target for MUP or any other qualifying metric, the client can send an inquiry formulated as a GitHub issue with relevant title, description and labels.

Definitions

Back-off requirement: For each failed request, wait 30 seconds before sending the next one.

Inquiry response time: The Inquiry response time is the amount of time in hours that a client can expect to wait. The time period is determined by the weekday of the request and follow this scheme:

Day	Time period
Mon-Thu*	24 hours
Fri-Sun*	72 hours

* Excluding Danish holidays

SLA Exclusions

The SLA does not cover: Client hardware or software (or both) errors in relation to our product; errors caused by factors outside of Group F's reasonable control; Errors caused by the client not adhering to the documentation, e.g. invalid request, unauthorized or unrecognized users, or inaccessible data

Figure 11: Screenshot of Service-Level Agreement. For the full markdown file, see https://github.com/Chillhound/DevOps2022F/blob/main/SLA.md

B Licenses

.NET: MIT https://github.com/microsoft/dotnet/blob/master/LICENSE
ASP.NET Core: MIT https://github.com/dotnet/aspnetcore/blob/main/LICENSE.txt
React: MIT https://github.com/facebook/react/blob/main/LICENSE
TypeScript: Apache 2.0
https://github.com/microsoft/TypeScript/blob/main/LICENSE.txt
Serilog: Apache 2.0 https://github.com/serilog/serilog/blob/dev/LICENSE
Prometheus dotnet: MIT
https://github.com/prometheus-net/prometheus-net/blob/master/LICENSE
Docker-Compose: Apache-2.0 license https://github.com/docker/compose/blob/v2/LICENSE
Keepalived: GNU GPL (2 or above)
https://keepalived.readthedocs.io/en/latest/license.html
Vagrant: MIT License https://github.com/hashicorp/vagrant/blob/main/LICENSE
Grafana: AGPLv3 https://grafana.com/licensing/
ElasticSearch: Apache 2.0 https://www.elastic.co/pricing/faq/licensing
Kibana: Apache 2.0 https://www.elastic.co/pricing/faq/licensing

C Results from bettercodehub.com

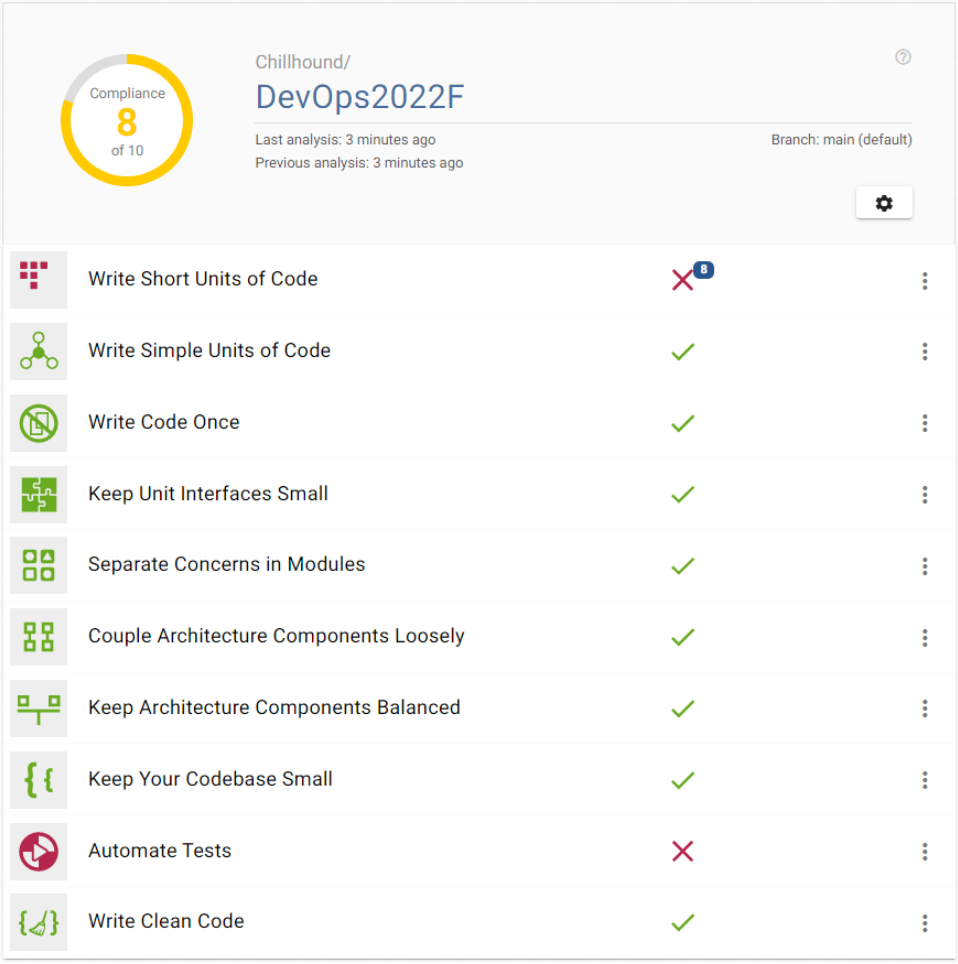

Figure 12: Results from bettercodehub

D GitHub Board

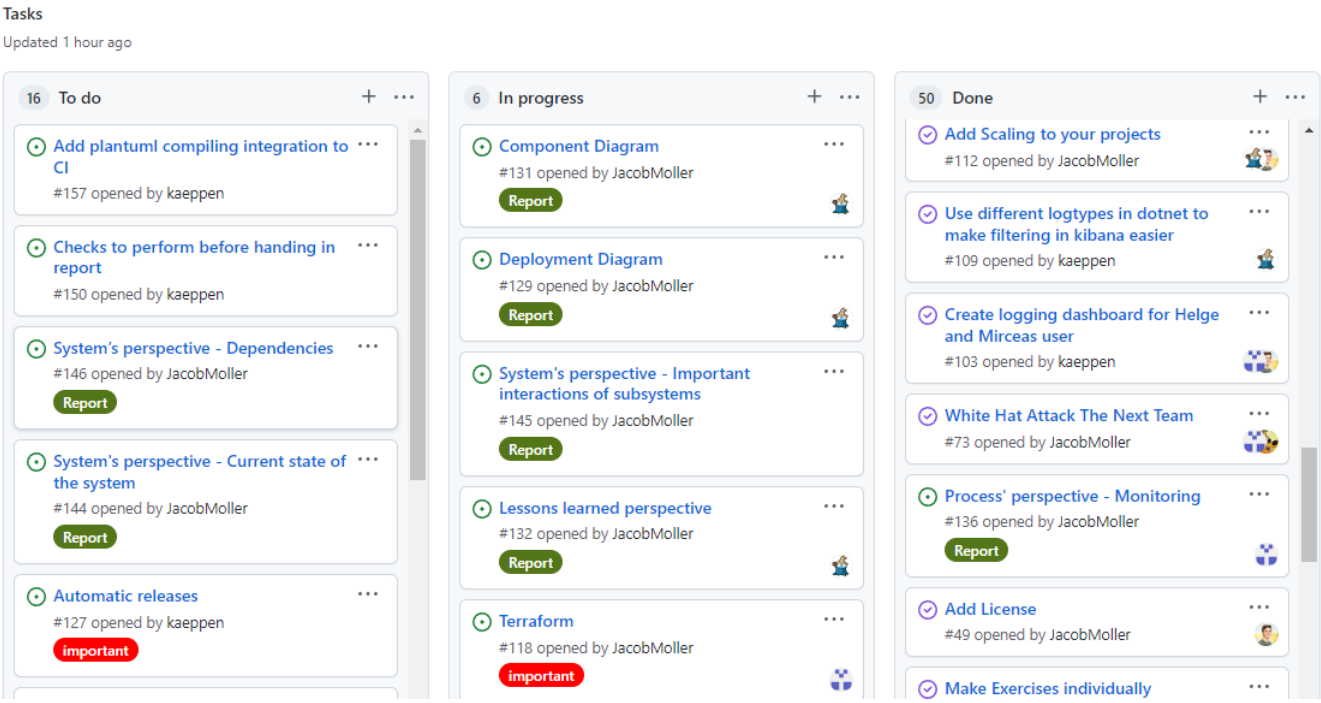

Figure 13: GitHub Project board

E Security Assessment

Risk Identification

1. Identify assets (e.g. web application)

- Web application
- Azure Database
- CircleCI Integration
- DigitalOcean Droplets
- GitHub Repository
- DockerHub Images
- Prometheus/Grafana
- ElasticSearch/Kibana

2. Identify threat sources

Identification and authentication failures: Public API does not check for any authentication and can change entities in the database. This means that you can access and create user tweets without being authenticated.

Software and Data Integrity Failures: Our Backend written in C# is never analysed during our pipeline deployment and therefore the not verifying integrity.

SQL Injection: Through EF Core we automatically escape (sanitize) the user-inputs (users tweet message). This means that no user-input will be misinterpreted by our program after this point. As the user-input is not used in any logic or inserted anywhere before this point we find this approach adequate.

3. Construct risk scenarios

User uses the public API and sends requests to manipulate database without authentication.
User sends too many request to our server (DDoS attack). The primary server will stop responding and the secondary server will take over and eventually stop responding as well.
User gains access to a developer computer with User Secrets. This allows the User to access all our infrastructure.
User locates a deprecated dependency with a security vulnerability and manipulate data or takes down infrastructure.

Risk Analysis

Determine likelihood

Missing API Auth: Very High Frequency as this is a pretty apparent issue when using the API.
DDoS: High Frequency. In recent years the amount of DDoS attacks has only increased according to Cloudflare[7].
Dependency Vulnerability: Medium Frequence. Dependencies are often updated to fix vulnerabilities and we must ensure that we are using the newest versions and do not use deprecated dependencies.
Access to Developer Device: Very Low Frequency. Though this would give full access it would be password-protected and each individual infrastructure provider, like DigitalOcean, has its own account password (and for DigitalOcean also Two-Factor Authentication).

Determine impact

[7] https://blog.cloudflare.com/ddos-attack-trends-for-2021-q4/

Missing API Auth: This allows the user to tweet on others behalf which compromises all user authentication promises.
DDoS: DDoS compromises availability but not data (compromising/leaking data) so it is not mission critical.
Dependency Vulnerability: A potential vulnerability can result in partial or full data leak which in the real world can result in legal charges and fines.
Access to Developer Device: Access to one of our devices can result in total shutdown of infrastructure and code deletion.

Use a Risk Matrix to prioritize risk of scenarios

IMPACT		Very Low Severity	Low Severity	Medium Severity	High Severity	Very High Severity
	Very High Frequency					Missing API Auth
	High Frequency				DDoS	
	Medium Frequency					Dependency Vulnerability
	Low Frequency					
	Very Low Frequency					Access to Developer Device

PROBABILITY

Discuss what are you going to do about each of the scenarios

Missing API Auth: With more course-time it would be very high priority to resolve this issue as it impacts user-data. This although is not a problem in our other API as these requests are authenticated through the website.
DDoS: In situation with increased server-load it would make sense to enable DDoS-protection tools to reduce the risk of DDoS.
Dependency Vulnerability: Dependabot[8] can automatically notify us if any dependency needs to be updated.
Access to Developer Device: We have ensured that all developer devices are password-protect and that all DigitalOcean users with access to our infrastructure is protected by Two-Factor Authentication.

[8] https://github.com/dependabot

F Email alert

Figure 14: Email alert setup with Keepalived and sSMTP

G Logs from attempts to access database

```
2022-03-03 21:10:02.23 Logon        Error: 18456, Severity: 14, State: 8.
2022-03-03 21:10:02.23 Logon        Login failed for user 'sa'. Reason: Password
did not match that for the login provided. [CLIENT: 157.245.152.95]
2022-03-03 21:10:02.68 Logon        Error: 18456, Severity: 14, State: 8.
2022-03-03 21:10:02.68 Logon        Login failed for user 'sa'. Reason: Password
did not match that for the login provided. [CLIENT: 157.245.152.95]
2022-03-03 21:10:03.14 Logon        Error: 18456, Severity: 14, State: 8.
2022-03-03 21:10:03.14 Logon        Login failed for user 'sa'. Reason: Password
did not match that for the login provided. [CLIENT: 157.245.152.95]
2022-03-03 21:10:03.60 Logon        Error: 18456, Severity: 14, State: 8.
2022-03-03 21:10:03.60 Logon        Login failed for user 'sa'. Reason: Password
did not match that for the login provided. [CLIENT: 157.245.152.95]
2022-03-03 21:10:04.21 Logon        Error: 18456, Severity: 14, State: 8.
2022-03-03 21:10:04.21 Logon        Login failed for user 'sa'. Reason: Password
did not match that for the login provided. [CLIENT: 157.245.152.95]
2022-03-03 21:10:04.84 Logon        Error: 18456, Severity: 14, State: 8.
2022-03-03 21:10:04.84 Logon        Login failed for user 'sa'. Reason: Password
did not match that for the login provided. [CLIENT: 157.245.152.95]
2022-03-03 21:10:05.31 Logon        Error: 18456, Severity: 14, State: 8.
2022-03-03 21:10:05.31 Logon        Login failed for user 'sa'. Reason: Password
did not match that for the login provided. [CLIENT: 157.245.152.95]
2022-03-03 21:10:05.73 Logon        Error: 18456, Severity: 14, State: 8.
2022-03-03 21:10:05.73 Logon        Login failed for user 'sa'. Reason: Password
did not match that for the login provided. [CLIENT: 157.245.152.95]
2022-03-03 21:10:06.36 Logon        Error: 18456, Severity: 14, State: 8.
2022-03-03 21:10:06.36 Logon        Login failed for user 'sa'. Reason: Password
did not match that for the login provided. [CLIENT: 157.245.152.95]
2022-03-03 21:10:06.96 Logon        Error: 18456, Severity: 14, State: 8.
2022-03-03 21:10:06.96 Logon        Login failed for user 'sa'. Reason: Password
did not match that for the login provided. [CLIENT: 157.245.152.95]
2022-03-03 21:10:07.39 Logon        Error: 18456, Severity: 14, State: 8.
2022-03-03 21:10:07.39 Logon        Login failed for user 'sa'. Reason: Password
did not match that for the login provided. [CLIENT: 157.245.152.95]
2022-03-03 21:10:07.82 Logon        Error: 18456, Severity: 14, State: 8.
2022-03-03 21:10:07.82 Logon        Login failed for user 'sa'. Reason: Password
did not match that for the login provided. [CLIENT: 157.245.152.95]
2022-03-03 21:10:08.38 Logon        Error: 18456, Severity: 14, State: 8.
2022-03-03 21:10:08.38 Logon        Login failed for user 'sa'. Reason: Password
did not match that for the login provided. [CLIENT: 157.245.152.95]
2022-03-03 21:10:09.04 Logon        Error: 18456, Severity: 14, State: 8.
2022-03-03 21:10:09.04 Logon        Login failed for user 'sa'. Reason: Password
did not match that for the login provided. [CLIENT: 157.245.152.95]
2022-03-03 21:10:09.50 Logon        Error: 18456, Severity: 14, State: 8.
2022-03-03 21:10:09.50 Logon        Login failed for user 'sa'. Reason: Password
did not match that for the login provided. [CLIENT: 157.245.152.95]
2022-03-03 21:10:09.92 Logon        Error: 18456, Severity: 14, State: 8.
2022-03-03 21:10:09.92 Logon        Login failed for user 'sa'. Reason: Password
did not match that for the login provided. [CLIENT: 157.245.152.95]
2022-03-03 21:10:10.52 Logon        Error: 18456, Severity: 14, State: 8.
2022-03-03 21:10:10.52 Logon        Login failed for user 'sa'. Reason: Password
```

Figure 15: Logs from Docker Container