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 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 but at the same time have the opportunity to learn something new.

Our frontend is made using React with Typescript. 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 back-end. 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

.

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 expose an API which adheres to the requirements of [3]
2. Something about minitwit website?

Non-functional requirements

1. C#
2. React with Typescript
3. digitalocean
4. docker
5. performance, scalability, reliability

1.2.2 Module viewpoint

Ting der skal være her ifølge 3+1 modellen:
- Elements: classes, packages, interfaces
- Relations: står i slides
- Mapping to UML: class diagrams (nok ikke relevant for os?)

This section attempts to describe how the functionality of our system is organized in code. At the highest level, the system is divided into backend and frontend.

The following package diagram gives a high-level overview of how the backend files are structured.

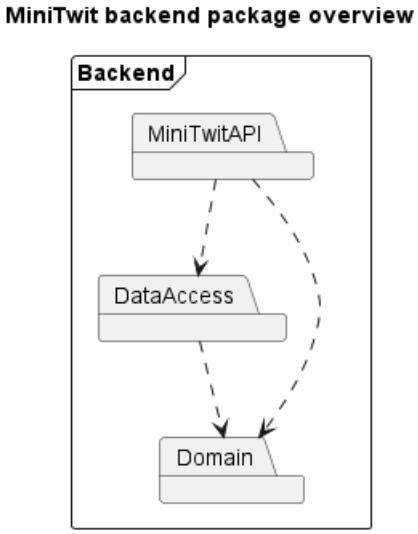

Figure 1: Backend Package overview

- **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 i.e., can be seen below.
No class diagrams will be shown, as this would not add anything interesting to the report.

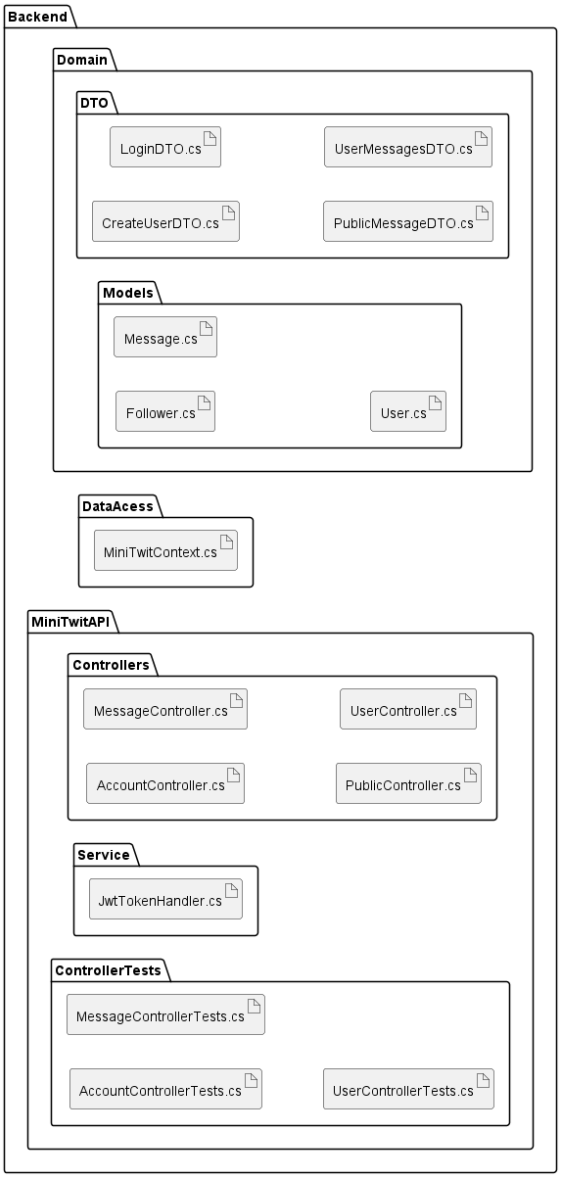

Figure 2: Package diagram showing our backend artifacts

skal dette diagram vise dependencies? skal man gå dybere i det her lort?

The other interesting part of the program is the frontend, of which a package diagram is shown below. Only the "src" package contains files of interest i.e. that we've written and not configuration or template 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 configuration-files that are React-specific. The naming of the pages reflects the naming from the original page structure provided.

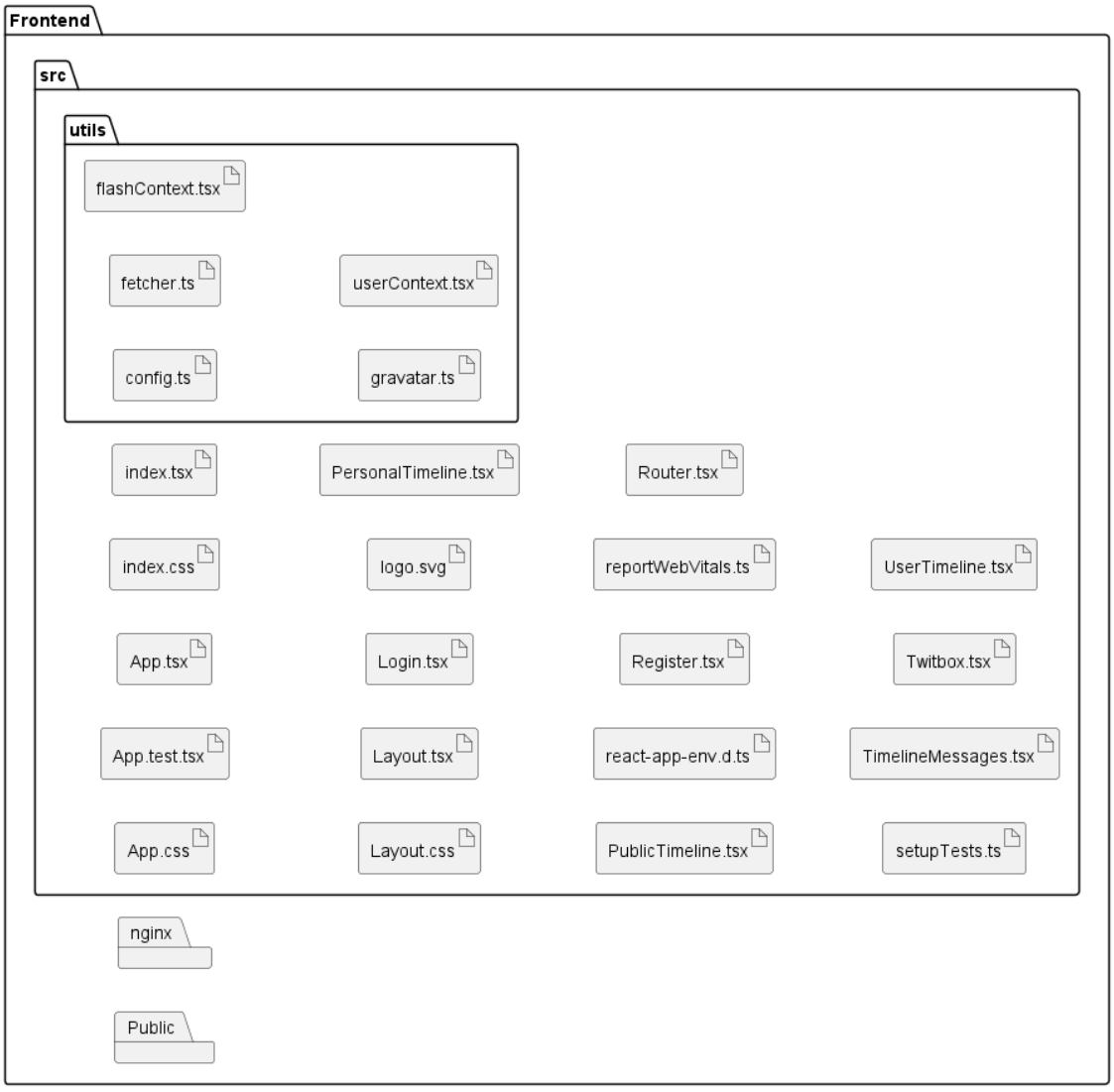

Figure 3: Package diagram of the frontend folder, only depicting the interesting parts

1.2.3 Components & connectors viewpoint

Denne sektion mangler tekst
Dette viewpoint handler om:
- Components og deres "functional behaviour" / runtime
Jeg er ikke sikker på at diagrammet passer 100% ind her eller at jeg forstår dette viewpoint helt. Det er dog meget vigtigt i følge ham der har lavet modellen.

Hvad skal vi skrive til diagrammet?

The following diagram shows the six components i.e. Docker containers of our primary DigitalOcean droplet and how they interact with each other at runtime. The "MiniTwit backend"

component contains two separate API subcomponents. The Docker container uses the same port to handle both API requests from the simulator and from the website but we've chosen to show it as two components in the diagram to enforce that they serve two different purposes.

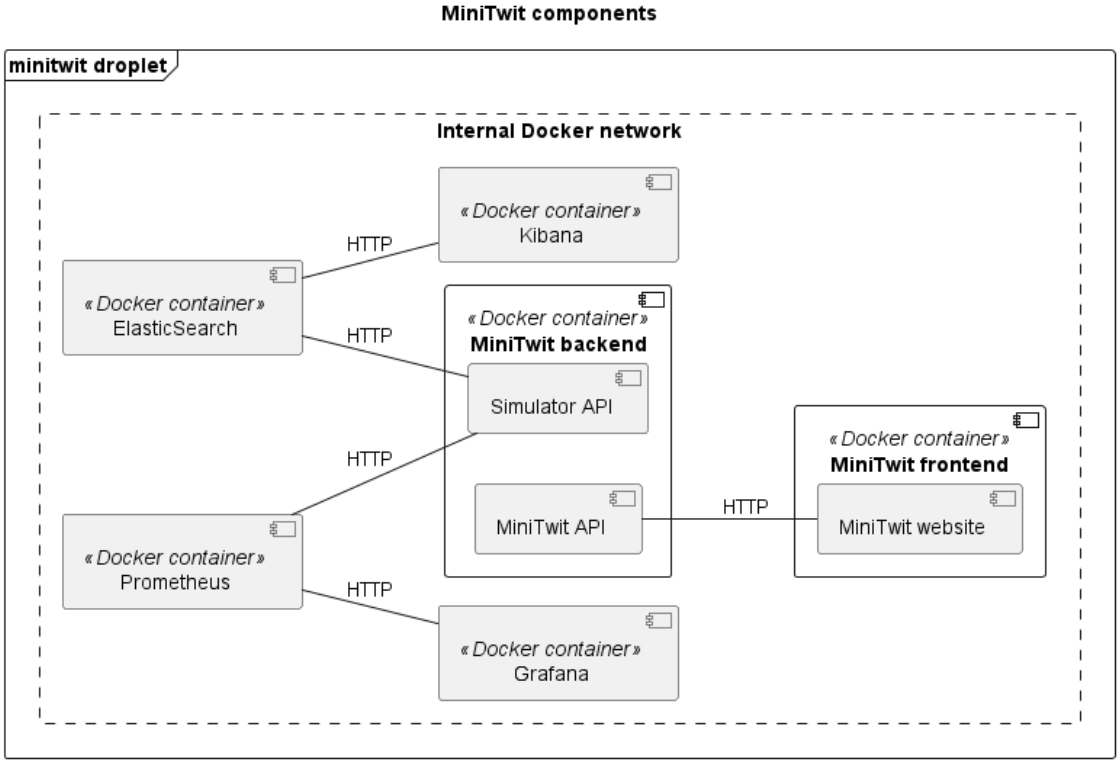

Figure 4: Component diagram of our primary DigitalOcean droplet

1.2.4 Allocation / deployment viewpoint

This section illustrates how elements of our system is mapped to infrastructure.

Denne sektion mangler tekst

Physical deployment:
- computers / execution environments
- Deployment units - Communication links and dependencies

The following diagram ..

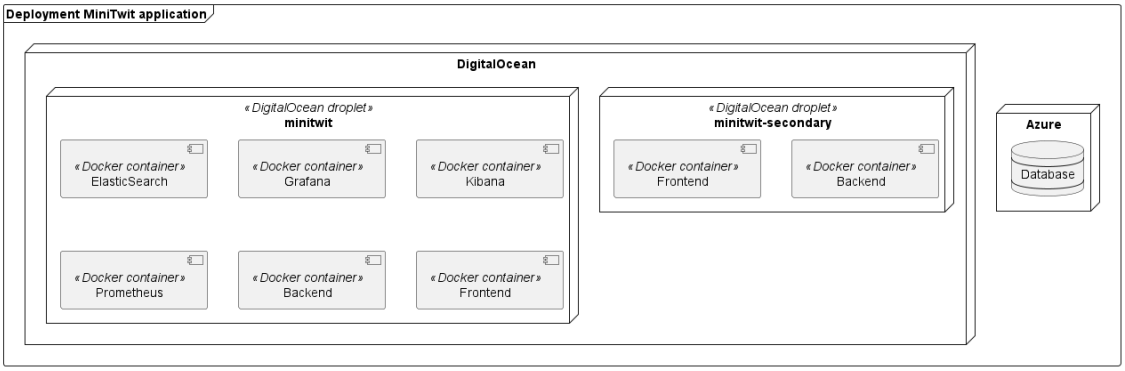

Figure 5: Deployment diagram illustrating the overall deployment view of our MiniTwit system

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

specs:
Primary
- 1 vCPU - 2 GB memory
- 50 GB disc space

Secondary
- 1 vCPU
- 1 GB memory
- 25 GB disc space

They don't have the same specifications as they don't handle the same tasks but this will be elaborated in a later section.

1.3 Dependencies - technologies and tools

Docker is used for containerizing the system
Python is used for the old MiniTwit application.
Bash is used as terminal for the developers.
C# / ASP.NET CORE / dotnet is used for our backend
Ubuntu, the OS that our droplets run on.
React our JavaScript library for building the website.
Nginx is a web server that runs React
Keepalived high-availability / scaling
Docker-compose handling multiple docker containers
Dockerhub storing docker images
Typescript is used for React frontend
Git version control
Github repository management
Sonarcloud static analysis
DigitalOcean provider of infrastructure
Vagrant
CircleCI continous integration pipeline
Serilog a library used for logging in the backend
Prometheus is used for system monitoring
Grafana is used to display monitoring information
ElasticSearch is used to aggregate the systems logs
Kibana is used to present the aggregated logs
Azure SQL Edge was used in a container in the first weeks of our project. As we wanted a separate permanent place to store the database we moved away from this.
Azure SQL Database is the Database system used for our MiniTwit application.
Better Code Hub provides static analysis of the code
Microsoft Data Migration Assistant is a tool that helped us migrate from Azure SQL Edge Database in a Droplet to the Azure SQL Database.
SSMPT is used to configure the mail server.
MailUtils is used to send a mail directly from a script using SSMTP server settings. Concretely this is an alert to notify us when Keepalived switches to another server than the Primary (when the primary dies)
Gmail is 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 applciation are the following

- HTTP communication between frontend and backend
- HTTP communication between backend and the simulator
- HTTP communication between backend and Prometheus
- ?? communication between Prometheus and Grafana
- HTTP communication between backend and ElasticSearch
- ?? communication between ElasticSearch and Kibana

The following sequence diagram shows the interaction between frontend and backend when a user posts a new message.
The flow of most? all? interactions between frontend and backend as well as between the backend and simulator follow the same flow, and as such we won't waste space on showing these.

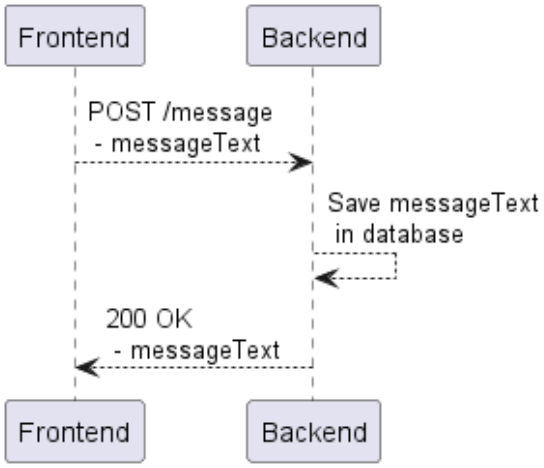

Figure 6: Sequence diagram depicting the interaction between frontend and backend

The next sequence diagram 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.

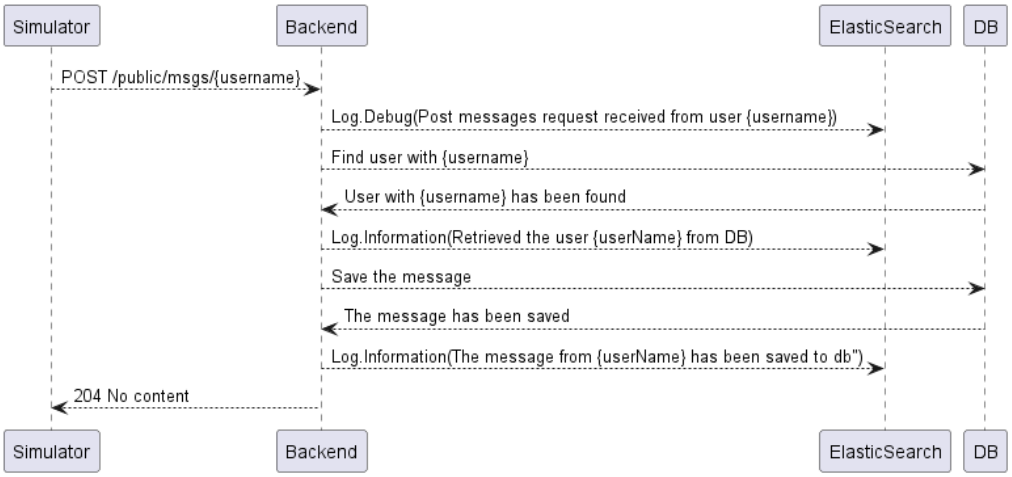

Figure 7: Sequence diagram depicting the interactions between the simulator, our simulator API, ElasticSearch and database that happen when the simulator posts a new message from a user

1.5 Current state of the system

In it's current form the system has a public API

Results from the static analysis tools X and Y are ...

Hvis vi af en eller anden grund når noget af det her, så slet punkter
The following are features and technologies that we would have liked to implement but did not.

1. Infrastructure as code with Terraform
2. Rolling updates
3. Automatic releases
4. Running tests as part of CI (and terminating on test failure)

1.6 License compatibility

The license used for this project is: **GNU GPL-3.0**. The list of licenses in the project can be seen in appendix A. According to license slide by Wheeler[4] the projects licence is compatible with the licenses:

- MIT
- Apache 2.0
- GNU GPL v3
- 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

This section will introduce the way the developers chose to interact and communicate over the course.

2.1.1 Discord

It was decided to use the communication platform *Discord*, since it is a that is great for facilitating discussions. The main communication line was in text channels, where various subjects was discussed such as; how to tackle different tasks and problems, plan collaboration sessions either physically or digitally, etc. Lastly, voice channels were used to execute collaboration session or further explain and share thoughts on the subjects in the text channels. Additionally, it was common to work in the weekends if the main task(s) had not been completed during the weekly exercise session.
During these online collaboration sessions we used screen sharing as a way to program in pairs and sanity checking the work being done.

2.1.2 Physical meet-ups

On Tuesdays after lecture, the developers would meet-up at the weekly exercise session and focus on the concurrent weeks' workload. This was done so every team member has some insight in what was happening and could share thoughts and troubles that might occur during development of the project. Additionally, it is a good incentive getting most of the workload for the week done as fast as possible. Towards the end of the course a weekly status meeting was implemented to exchange experiences with other groups using similar techstacks & technologies and have a forum to facilitate information sharing.

2.1.3 Github Projects

Github Projects was used for posting various tasks that were discussed on discord, functioning as a Kanban board. This gave an overview of who were working on what, since each developer could assign themselves to the tasks. Allowing the developers to organize accordingly.

2.2 Stages and tools in CI/CD chain

This section aims to show the different stages and tools used in the projects Continuous Integration and Continuous Deployment. Additionally, discussing any future plans of changes on the 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 CD/CI 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.

The configuration file for the pipeline is located here: https://raw.githubusercontent.com/Chillhound/DevOps2022I

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).

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 digital ocean droplet to stop running containers, pull newest images from Docker Hub and then start it all again with a Docker Compose file
4. der kommer måske et step omkring rolling updates her

The reason behind SonarCloud only scans frontend is that static analysis works of C# works differently. In order to scan C# projects the system containing the MSBUILD needs to have a dedicated sonar-scanner installed. Since the projects' MSBUILD is being containerized it was decided that it would be easier to use the build-in feature *Docker scan* from Docker, to do the static analysis instead of installing the sonar-scanner on the image each build.

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 when it goes back live and redirect traffic back to the primary.

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

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 uses the feature branching strategy. 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 it gets merged into the main branch after it has been reviewed by a team member.

2.5 Applied development process and tools supporting it

During the project we've actively used GitHub issues to keep track of tasks both related to weekly assignments but also bugs, refactorings etc. that we've 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.

2.6 Monitoring

To monitor the system the Prometheus package for dotnet, prometheues-net, is used. Prometheus exposes metrics from a ASP.net application. 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 monitoring information that Prometheus gatherers from our system. The dashboard shows the following informations:

- Request received
- Error rate
- total request/s
- Request duration
- request in progress

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. Serilog has the same role as Logstash and handles the aggregation. this means, that Serilog parses and sends the data to ElasticSearch as Logstash would in a normal ELK stack.

- Serilog[3]
- ElasticSearch
- Kibana

[3] https://serilog.net/

Serilog is imported as a dependency in the dotnet 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. These log levels can then be used for searching for logs with the specific level. The logging is centered around the simulator API and as such does not include e.g. the API used by the MiniTwit website.

2.8 Security assessment

Based on our security assessment we have taken precautionary steps which concretely has resulted in 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 B?

2.9 Scaling and load balancing

We choose to implement high-availability by using the configuration with Keepalived discussed in class, though with some modifications as the article[2] 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've had since the beginning of the project and one secondary which is activated if the primary crashes.
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 i.e. 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 the 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 it also creates a single point of failure at the logging-droplet.
her kan vi evt. kort nævne det med at det også er en emergency og derfor ikke vigtigt at det hele virker

We have created alerts by email using sSMTP [5] and Mailutils [6] which executes when the Secondary droplet is registering 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.

3 Lessons learned perspective

3.1 Biggest issues, how were they solved and what we learned

The following subsections present some of the biggest issues we've faced during this course, how we solved them and what we learned from it. Common to almost all of them is that they've taught us that taking the time to do things right the first time (or at least when you recognize that something is a problem) is invaluable compared to continue working with incomplete and ineffective workarounds.
Some of us have been eager to finish tasks fast and thus not prioritized attention to important details like e.g. handling user secrets or reproducibility.
But having the responsibility for maintenance, refactoring and evolution has given us a new view of the software development process, as you're only hurting your future self when doing things

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

fast and easy.
Prior to this course none of us have tried anything else than rushing to get a project done, handing it in and then not having to deal with it anymore - so this has really been an eye-opener.

3.1.1 Hacking directly on the server

For quite some time (indtil commit https://github.com/Chillhound/DevOps2022F/commit/e45526bdfac81a342d1c74413dfd561e0bf05a89 d. 3/4?? måske lidt før) 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. (find lige et konkret eksempel med commit - var det ikke noget med noget db på et tidspunkt? altså da vi skiftede til azure måske? JO DET VAR! den korrekte ændring blev lavet direkte på serveren, men vi committede noget andet til repoet som fuckede det op, mener jeg)

- det er "nemt" nok at gøre men man glemmer hele tiden noget - should have fixed it earlier reflekter over hvorfor det er dårlig devops praksis

3.1.2 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 (migrated to Azure 15/3, commit https://github.com/Chillhound/DevOps2022F/commit/64df2d7b400a116b25d448e2005b062c0fb2bf72).
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 3 "Logs from attempts to access database".
The downtime caused by these crashes led to missing requests from the simulator. It also made us fear losing all of the data, which led to creation of (manual) backups. We did not automate the backup process, as we agreed to move to a hosted database as fast as possible to reduce risk of losing data, especially because we did not think about using Docker volumes from the beginning.

Migrating to the cloud

For the reasons specified above we decided to migrate to a database hosted with Microsoft Azure. By this time we had become comfortable with creating backups and restoring from them but our research showed that these backups were not directly compatible with the new Azure SQL database making it necessary to use a migration tool (Microsoft Data Migration Assistant [7]).
After migrating the data we had some issues with getting connecting the .NET project to new the database and ultimately left the connection string in the repo for ??? days.

3.1.3 Handling secrets

For a good part of the project we did not handle secrets correctly (database connection strings), as they were committed to our Github repository.
As described above, we saw that malicious people systematically tried to access our database and thus we assume that there's a real risk of someone scanning public repositories for secrets that can be used to exploit or damage systems.

[7] https://docs.microsoft.com/en-us/sql/dma/dma-overview?view=sql-server-ver16

We learned from another group that they 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.

3.1.4 /msgs endpoint becoming very slow

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

1. Retrieved all of the posted messages from the database
2. Sorted them according to the date they were entered
3. 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 understandable that this became 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 create an clustered index on the primary key of the Messages table, which drastically improved the speed of retrieval.

3.2 DevOps style of work

Der skal nok læses lidt på hvad søren dette præcist betyder :D Der er noget om "three ways" fra devops handbook, link i discord First way = systems thinking
Second way = amplify feedback loops
third way = culture of continual experimentation and learning
Men måske det egentlig ikke kun handler om det? jeg antog det bare lidt
Det giver nok også mening at se lidt i slides fra lektion 5

The previous section briefly touched on how this project has been different to our previous experiences. well, så skal det måske kun stå et sted?

Given the organization and size of our team and the way we have worked together, we definitely had a culture of continual experimentation and learning. Kan vi godt claime det?

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] 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.

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

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

A 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

B 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[8].
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
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[9] 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://blog.cloudflare.com/ddos-attack-trends-for-2021-q4/
[9] https://github.com/dependabot

C 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