DevOps project

BlackOps - lakl, frot, adbe, cafm, anlf

May 31, 2022

Contents

1 Introduction

The following report was written as a part of the course *DevOps, Software Evolution and Software Maintenance*. This report will take an in depth look at the final product made by us, *Group G - Call of Duty BlackOps* consisting of the following members:

1. Anton Dalsgaard Bertelsen {adbe@itu.dk}
2. Anton Lukas Dørge Friis {anlf@itu.dk}
3. Caspar Marschall {cafm@itu.dk}
4. Frederik Rothe {frot@itu.dk}
5. Lasse Faurby Klausen {lakl@itu.dk}

During this course we learned to refactor a legacy code base to work on a more modern system, as well as maintaining and improving the system while in use. The product that we needed to maintain was a social media platform working similarly to the microblogging and social networking platform *Twitter*. With the help of a script and an API, we could simulate a scaling amount of users using our platform by signing up, following other users and posting messages. The intensity of requests started out very low, but would from then on exponentially increase until the end of the simulation, mimicking the growth of our platform. This was done to increase the difficulty of maintaining of the system as well as increasing the severity of mistakes, bugs and server failures.

The platform was accessible on *http://157.230.107.58:5142* until the 10th of May, 2022, where after it was taken down.

2 System Perspective

2.1 System Design

We chose to convert the legacy code base into a *C# ASP.NET* and *Blazor* project. This was mainly done due to our prior experience working with web applications in *C#*. Furthermore, we also knew that a lot of the pipeline tools we had to implement down the line had good, if not great, support for the *.NET* platform.

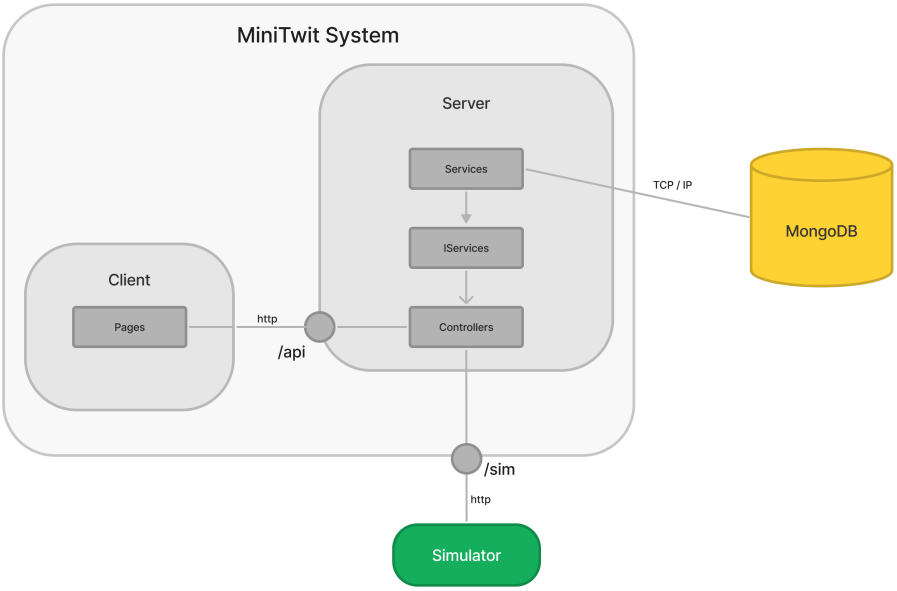

Figure 1: Component diagram of the core system design.

Our system is split into two parts: *Server* and *Client*. The *Server* is responsible for the back-end part of the system, that is the business logic, exposing endpoints and interaction with the database. The *Client* represents the frontend part of our system, and includes everything related to the user interface.

As for our database, we decided to use the *NoSql* database *MongoDB*. We found *MongoDB* to be a fitting tool for the job as it goes nicely together with the idea of a cloud and scalable structure, which this whole project is based on. This also meant that we had the option of easily scaling our database in the future, should the need should arise.

2.2 System Architecture

The system has a micro-service based architecture and is implemented by the use of docker containers. Our system is deployed on a *Digital Ocean* Droplets[1] which is set up to run an *Ubuntu* based virtual machine (VM). In total we have two VM's where several different applications runs, providing and supporting the *MiniTwit* application.

We have defined a common sub-network[2] inside the *MiniTwit* VM, which includes all the containers there, enabling them to interact with each other.

The database is located on a different *Droplet* which excludes it from the common network mentioned in the section above. The reasoning behind placing the database on a separate *Droplet* was a conscious effort in making the system more reliable and less prone to corruption of our live database. In case of a total system crash failure in our main Droplet, our data would still be intact in an other environment. However, as our deployment diagram[3] reflects we have not set up any redundant databases, which would further increase the reliability. An easier alternative to the redundancy approach, is the use of docker volumes[4]. This approach would ensure the data being persisted even though the container might crash, but would still be liable to failure if the whole VM were to shut down.

[1] https://www.digitalocean.com/products/droplets
[2] https://docs.docker.com/compose/networking/
[3] Figure 2
[4] https://docs.docker.com/storage/volumes/

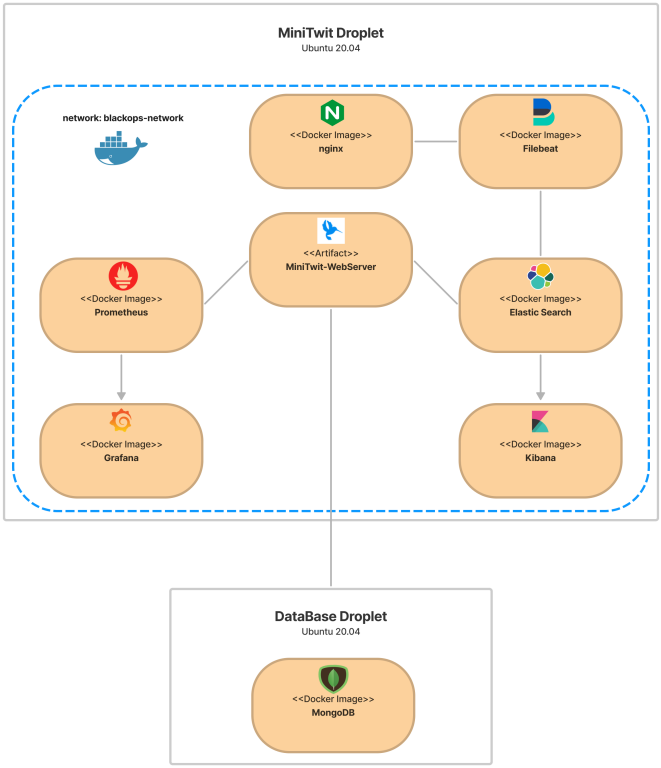

Figure 2: Deployment diagram of the micro service based architecture.

2.2.1 Subsystems

Our system is combined of many important subsystems that support and enable the *MiniTwit* web application. The different subsystems all have very different purposes and together create a fully fledged system that allows for monitoring, logging, scalability and storage.

Even though most of the containers that the subsystems run in are instantiated in the same environment, they are not visible to each other by default.

This realisation occurred to us when we instructed our *Grafana* container to pull data from the Prometheus container on port :9090. Naturally, the container wasn't able to succeed as there was no receiver on the port in its sub network.

Many of the subsystems will be further elaborated upon later in the report in context to their area of responsibility.

2.3 System Dependencies

A list of all direct technologies and tools we use in this project can be found in Appendix 7.5. For most parts we are using "static" versions, instead of latest version. Using the latest version ensures that the technology is up-to-date, however there is also the risk of newer versions not being able to work together. An example is, if a tool is updated so it uses a new API, then your system will fail.

Making sure everything is running on the same version does however allow for an increase in stability amongst the technologies.

In the group we haven't really decided which way we prefer, but currently in the project we are leaning towards static version, which we can update as needed.

2.4 Current State

The system in its current state contains some issues that preferably should be resolved. These are revealed by static code scanning tools.

2.4.1 Test Coverage Percentage

We prioritized implementing tests for the **MessagesController** and **UsersController** classes.

The test coverage of the entire system is 44%, as seen in Figure 4. The server assembly has a test coverage of 30%.[5] Preferably, test coverage should be greatly expanded to cover more aspects of the application and exhaustively test all controller endpoints.

As it stands, controllers are tested independently using Mock data. We would benefit from using integration testing in addition to this, which would allow us to discover issues when the controllers retrieve real data.

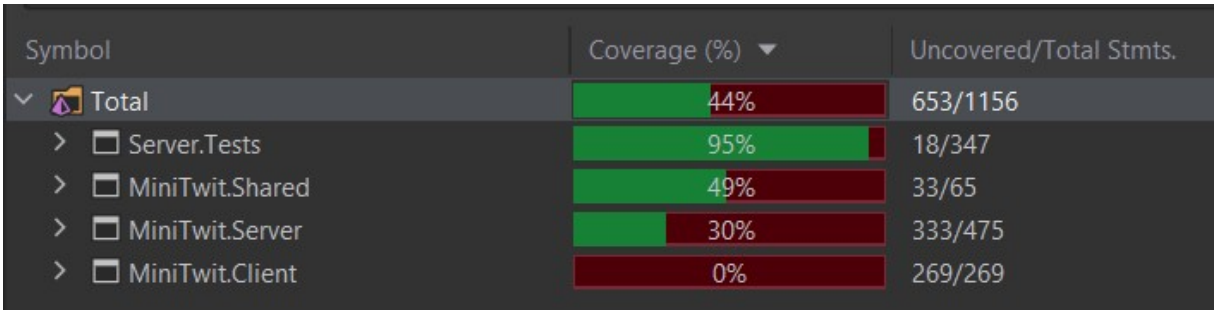

Figure 3: Test coverage of Minitwit.

[5]See appendix: Figure_9

2.4.2 Snyk Report

Snyk is able to identify and highlight security vulnerabilities in dependencies, container images and infrastructure configurations. Here *Snyk* has identified a series of vulnerabilities (see Figure 5), some of which are easily addressed. It is worth noting, that two issues of critical severity have been discovered. These are both instances of allowing remote code execution, by being dependent on a vulnerable version of the the library '*System.Text.Encodings.Web*'. Preferably, this critical security issue should be addressed by updating to a newer version.

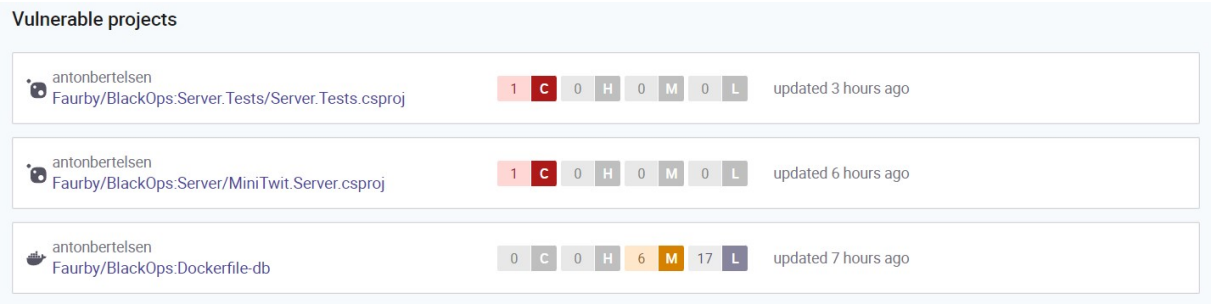

Figure 4: Snyk dashboard showing identified security issues.

2.4.3 SonarCloud Report

SonarCloud automatically scans code and highlights opportunities to improve quality and security. In Figure 6, it is worth noting that 15 of the 22 identified issues relate to using razor specific CSS syntax that *SonarCloud* is unfamiliar with and presumes to be a mistake.

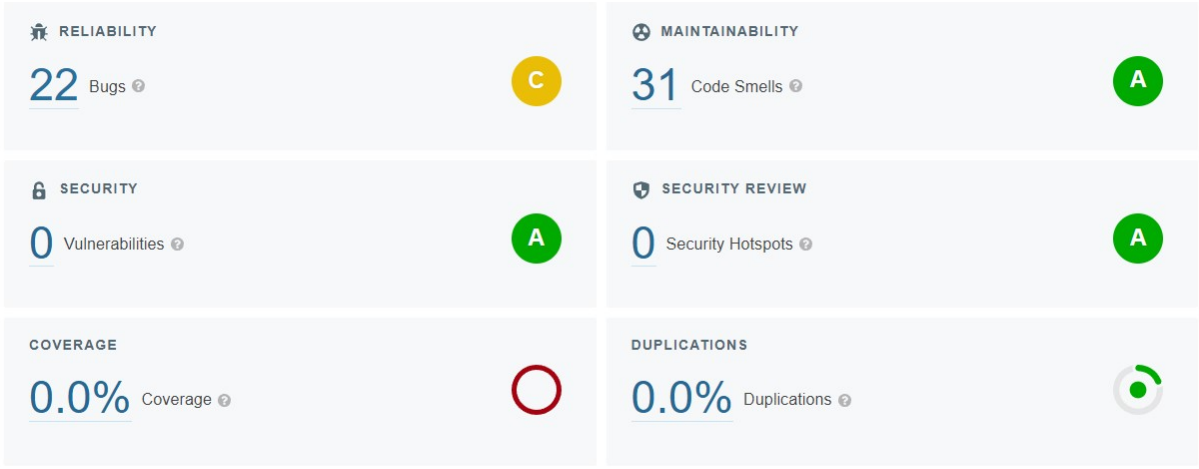

Figure 5: SonarCloud dashboard showing identified bugs and code smells.

2.4.4 Lighthouse Report

Lighthouse rates the page on a series of performance factors to give it an overall performance score. In Figure 7, *Minitwit* receives a score of 43, which shows there is much room for improvement. One reason that the page is slow to start is that we use *Blazor* Web assembly, which requires a large amount of data to be sent to the client when they first connect to the page.

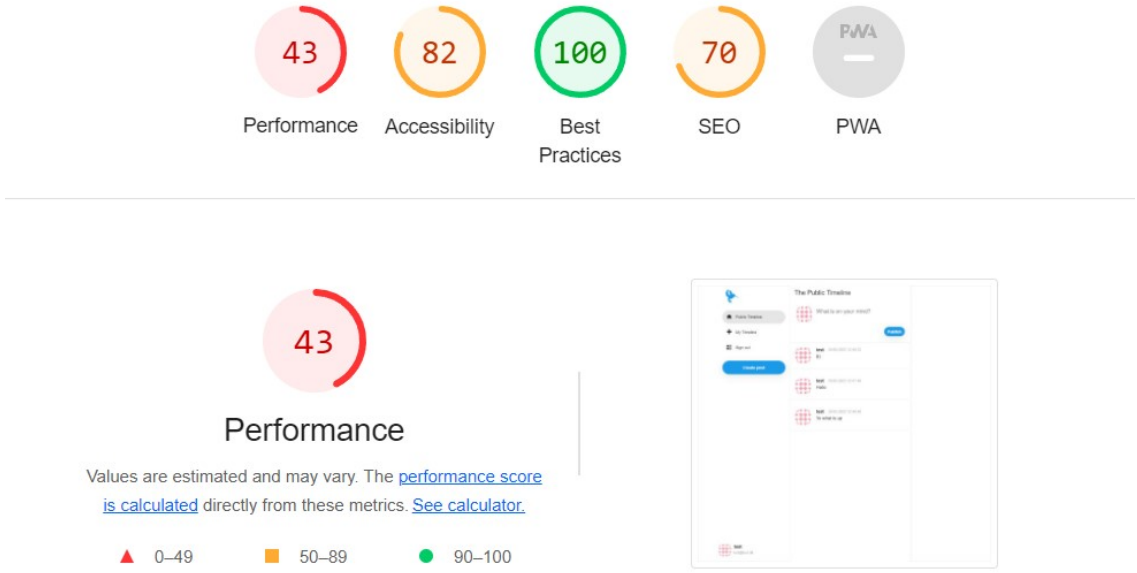

Figure 6: Lighthouse report showing page performance.

2.5 License compatibility

We are using the *MIT* license, which from this source[6] says it is compatible with the rest of the open source dependencies. However some dependencies like *ASP NET Core* have their own *Microsoft* license, so they are not open source.

We know that the *MIT* license is the most popular license on *GitHub*.[7]

Figure 7 shows how the dependencies in our project are primarily MIT licensed, and open source.

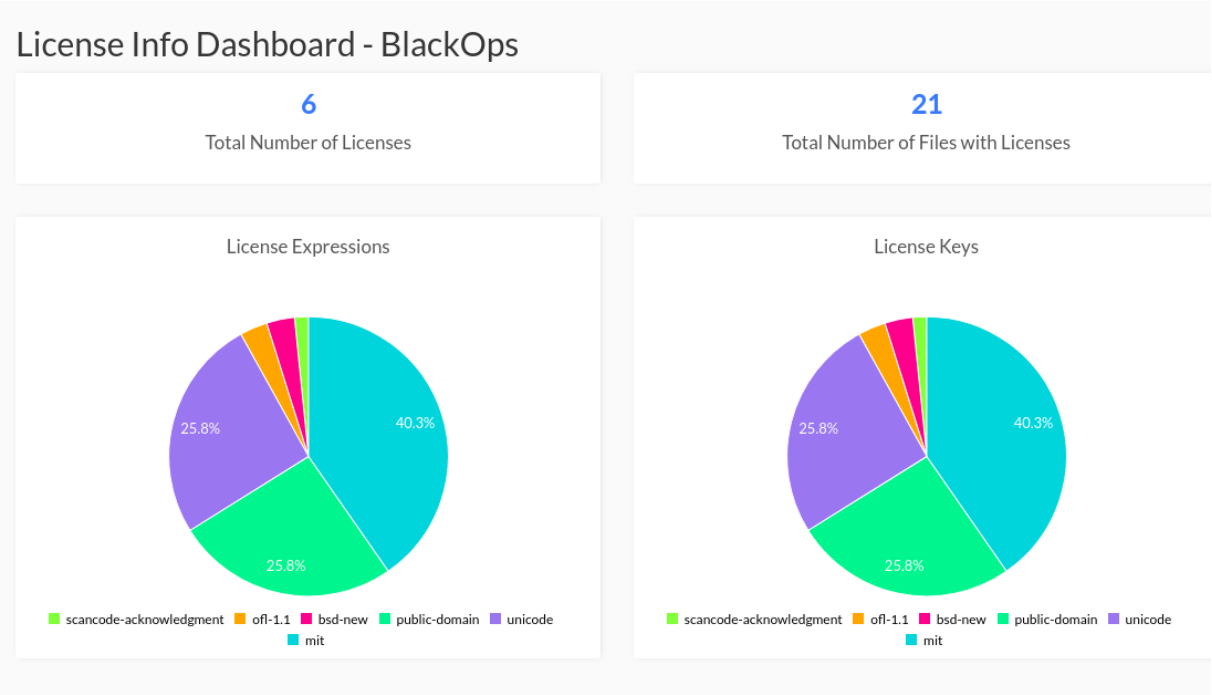

Figure 7: ScanCode licenses visualized

3 The Developing Process

3.1 Developer interaction

Our main way of interacting with each other on a weekly basis has primarily been on the communication platform *Discord*[8]. We have used this platform to not only communicate, but also share relevant articles, repositories and notes. This has worked pretty well, and we did not experience any issues with lack of communication within the team.

[6] https://dwheeler.com/essays/floss-license-slide.html
[7] https://github.blog/2015-03-09-open-source-license-usage-on-github-com/
[8] https://www.discord.gg

This project included a large amount of tasks that we needed to keep track of and prioritise. We therefore decided to use *GitHub Issues* along with the *labels* that this service presented us with. This will be explained in greater detail in the next section.

Overall, we have experienced a good workflow, with only a few complications. We did however, experience difficulty with implementing logging, which led to the server crashing and caused panic within the team. This will be covered in section 3.7.

3.2 Team Organisation

The team organisation has been playing a key part in our *MiniTwit* project. Especially creating a structured approach to the organisation of tasks has proven to be an important functionality. This is due to the team having a lot of other projects and work running in parallel with this course, causing a quick loss of overview if not helped.

As mentioned in the previous section, one of the tools that has helped organising our effort has been *GitHub Issues*. The Kanban board feature that *GitHub Issues* makes use of has been the dashboard of productivity in our project, and has enabled us to quickly gain an overview of pending tasks and current progressing. As with most project organising tools, the ability to assign and prioritise tasks is powerful features that allows for distributing work with less effort and increase team-wide oversight.

3.3 CI/CD Description

All CI/CD related workflows are running through the use of *GitHub Actions*. We made a total of six workflows:[9]

3.3.1 deployy.yml

The deployment workflow is activated once there is a push to the main branch, which happens after a successful merge. First it logs into the *DockerHub* account, which has our custom images, and then it pushes newest local images. After this it SSH into the root of our *MiniTwit* droplet, which is running the application. Then it pulls git, pulls docker, and deploy our two updated stacks to the docker swarm.

We later realized, that there was no need to pull our entire code base to the droplet, as it only needed the two docker compose files to run our services. This could have been optimized, since unnecessary places for the code base to be, could lead to a higher security risk.

[9] https://github.com/Faurby/BlackOps/tree/main/.github/workflows

3.3.2 dotnet.yml

This is a simple template from *GitHub Marketplace*, that tests your *.NET* environment. It is activated on all pull requests to main.

3.3.3 release.yml

The release workflow was made to help ensure that we made at least one weekly release. The release workflow was made following a guide made by *Cees-Jan Kiewiet* [10]. There were two ways of triggering this workflow: a minor release is made when pushing to main, and a major release is made on a cron job running every Sunday at 19. Unfortunately, we had some issues regarding the use of tags, and *GitHub's* naming convention, leading to multiple issues regarding major releases. To fix this, we would need to find another way of incrementing the tags, while allowing for the possibility that a tag could already exist.

3.3.4 snykimagescan.yml

This workflow activates on pull requests to main. It logs into our *DockerHub*, builds the custom images we have, and scans them for vulnerabilities. This is then displayed in the *GitHub* pull requests.

3.3.5 sonarcube.yml

This workflow activates on pull requsts to main. It pushes our code to the *SonarCloud* application, where it is scanned for vulnerabilities, code smells, warnings, etc. This is then displayed in the *GitHub* pull request.

3.3.6 yamllint.yml

This is a simple workflow which is activated on pull requests to main. It scans our .yml files and finds wrong syntax. This is displayed in the *GitHub* pull request.

We could improve a lot on our three workflows that scan our code base or docker images. At the current state, we have a very relaxed policy regarding errors and warnings. As long as it does not break the production, we don't prioritize fixing it. This could have been improved upon by having stricter policies. E.g., have a x score, x amount of warnings, before you could merge into main. This would optimize the quality of our code.

[10] https://github.com/WyriHaximus/github-action-next-semvers

3.4 Repository Organisation

Our entire code base was stored in a single repository. When making a new *Blazor* project, the template starts off in a single repository, so we decided to stick with this. We felt it unnecessary to split up the frontend and backend part of our system, as the project was quite small, and we had limited time to convert the legacy codebase.

3.5 Branching Strategy

We chose to follow a trunk based management practice for this project. We tried to make small and frequent changes, and did our best to merge these into the main branch as quickly as possible. We did this to try and follow the principle of continuous delivery. This streamlined the DevOps process, ensuring no branches diverged too much, which allowed us to avoid large merge conflicts. Frequent merges to main also let us stress test our CI/CD pipeline and discover issues. Branch protection rules were set up where pull requests to the main branch would require a review from someone else in the group. As such, the merging branches into main frequently also enforced frequent code reviews. Here automated tests implemented in the CI/CD pipeline would preemptively flag issues before a manual code review would be performed by another developer on the team. Code reviews were mostly performed immediately after the creation of a pull request. We attempted to avoid long lived branches, but exceptions to this did occur when working on larger features that would break functionality until they were fully implemented.

3.6 Monitoring

For the monitoring of our system we decided to use *Prometheus* and *Grafana*. This setup allowed us to store and visualize custom metrics from our system.

Prometheus is used for pulling data from the system and storing it as metrics. This could as an example be a count on the current amount of users registered or the amount of errors per second/minute etc.

Grafana is used for visualizing the data from *Prometheus*. *Grafana* has many different customizable options for creating different dashboards, which allows for a quick overview of the current status of the system.

Figure 8 depicts our final dashboard, which helped us keep track of different metrics, including the two we found to be the most important: The average response time of our controller and the error rate. We chose these as we came to realize that the server could not keep up the amount of simulator requests. Therefore, we decided to monitor the response time against the error rate to ensure the health of the server. We had to upgrade the server on multiple occasions, as it was starting to get overwhelmed with the increased rate of requests.

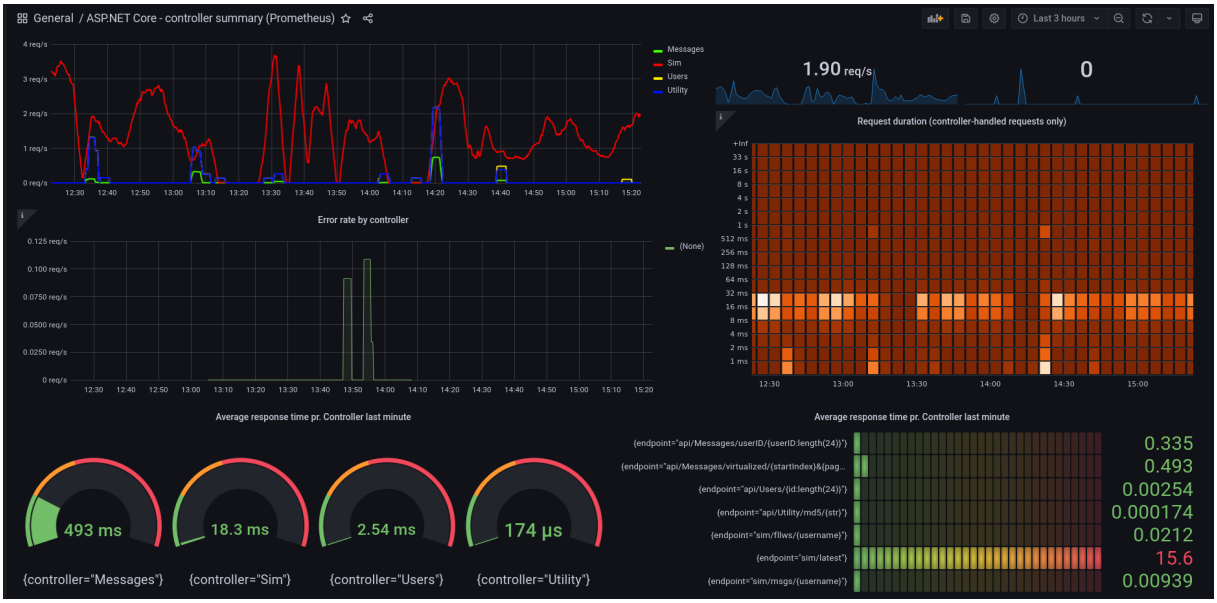

Figure 8: Final version of our Grafana dashboard

3.7 Logging

For implementing the logging of our system we added the ELK stack. This stack consist of the three open source products: *Elasticsearch*, *Logstash* and *Kibana*. For added security to the ELK stack we decided on using *Nginx* as a reverse proxy to redirect clients to the appropriate backend server. The ELK stack helps us by providing an easy to manage logging platform, which collects and processes all necessary logs.

Logstash is used as a log aggregator, collecting logs from multiple parts of the code.

Elasticsearch is used as a full-text search and analysis engine.

Kibana is a product used on top of Elasticsearch for visualizing data.

For getting the most optimized experience of the ELK stack, we could have added custom logs from our API endpoints. This would make it much easier to figure out when different errors, like the above mentioned, would occur and where to fix them.

3.8 Security

Using the security assessment we made[11], we found that our system was not completely secure. Exploring a couple of risk scenarios featuring some of the more common threats, we could establish a couple of flaws in our system. Some of the larger flaws included unwanted access to our database, as our connection string is openly available to anyone who knows the url of our

[11] https://github.com/Faurby/BlackOps/blob/main/Security_Assessment.md

GitHub repository. Another issue is that passwords are openly available when using our API. At this given moment, we are trying to fix this by changing our **User** class to a **UserDTO**. This DTO will not have a password, thus eliminating the problem. The security evaluation made it very easy to find potential problems, and was something we were planning to do again, if the course ran for a longer period of time.

3.9 Scaling and load balancing

To implement the possibility for scaling and load balancing, we use *Docker Swarm*[12]. *Docker Swarm* gives us the possibility to spawn multiple containers of the same image, and internally load balance incoming requests between these.

In its current state, our setup uses a single replica of each underlying system in our architecture, while our webserver has two replicas. This allows for an easier time updating the service, along with the absence of down time during an update. We have included a 10 second delay between updating the two webservers, to ensure that potentially harmful situations do not arise.

4 Lessons

4.1 Evolution and refactoring

The difficulty of handling systems tends to increase exponentially as the size of the system grows. This is why we learned about the automation of different tasks, to lower the workload of tedious but simple tasks.

This was done through different workflows, with one of the first being the *deployy.ylm* [13] that automatised the deployment of new code integration, when merging a pull request into the main branch. The workflow uses *docker compose* to start new containers only when a change has been made to the container images.

4.2 Maintenance

Maintaining cloud services is not a simple task. Unlike running a service locally, maintaining a live server requires not only the error handling and correctness of features, but also the preservation of the service's health.
When implementing *Kibana* to log important details regarding the health of our system, we unfortunately had some issues. It took us two weeks, two database crashes and a lot of needless panic to implement our logging system as seen from pull request *https://github.com/Faurby/BlackOps/pull/140* to pull request *https://github.com/Faurby/BlackOps/pull/150*. When

[12] https://docs.docker.com/engine/swarm/
[13] https://github.com/Faurby/BlackOps/pull/63

implementing *Kibana*, it worked just fine locally, but as soon as we deployed it to the webserver, *Kibana* used up almost all of our available resources. We then decided to revert to the prior deployment, however due to our mismanagement of Docker volumes, our database crashed when reverting.
We ended up improving the way we used docker volumes, and considered making a workflow that would backup our database once a day using a cron job. This would mean that if the database were to crash again, we would have a newer backup, leading to a smaller data loss.We did not get around to implementing the workflow, but discussed the practice of it in great detail.

4.3 Operation

When the database crashed, our latest backup was unfortunately around a week old. The old backup was missing some of the users that were created over the week prior to the crash, which led to our group getting a lot of **tweet**, **follow** and **unfollow** errors, as these were all user specific. However, since this our database crashed close to the end of a simulator, we did not lose that many requests.
This was unfortunately not the case for the second crash, as this happened just after the simulator started its next iteration, leading to us losing almost all of the users resulting in a lot of panic and more simulator errors. This taught us the importance of keeping a new backup at hand, as well as the difficulty of maintaining live services once again.

4.4 Our DevOps Style

We started this project with no prior experience working with cloud services. This meant that we had to change our way of thinking, planning and developing, as we no longer simply had to introduce something to a local environment.

When encountering the concept of the three ways mentioned in *The Phoenix Project*, we started acknowledging the importance of a continuous cycle. Through continuous build, integration, test and deployment processes, we achieved the foundation of the first way. This allowed us to reduce complications such as the lead time to integrate the features we were asked to implement over the course. Having a good understanding of the service we had created along with a well grounded sense of organisation, we also achieved the second way, by reflecting on problems that arose over time. This was where we recognised faults in our prior work together with discussing possible solutions for the future. We did however lack a sense of external feedback, as all of us in the group would have some bias towards the work that we had created. Working with a weekly integration scheme, we also achieved the beginnings of the third way, as we were confident on taking risks, knowing that nothing major would change from one day to another. As this course has been a learning experience, with no real drawbacks of failure, everyone in the group also felt confident to explore territory that seemed daunting and difficult at first glance. Integrating these practices allowed us to start noticing ways of increasing the efficiency of maintaining the

health of our service.

This past semester has shown us the difference and difficulty of developing and integrating software meant for live users. There is a noticeable difference in the amount of variables one has to take into account when developing software for a live service, as well as acute consequences should something go wrong. DevOps has given us a perspective into the software world that most new programmers are only familiar with on a concept basis, along with an initial understanding and rewarding nature of automating certain practices.

5 Conclusion

During the span of this project, we have not only learned how to manage and maintain a cloud service, but also how to develop a product using a DevOps mindset. In addition, we have learned about the operation of different tools and techniques, that have helped maintain the health of our service.

Furthermore, we have expanded our previous knowledge of terms such as **Continuous Integration** and **Continuous Delivery**, which have given us an understanding of the importance of the automation of inconvenient tasks. Moreover, we have achieved a fundamental understanding of how to develop a product using a DevOps mindset, and how it may differ from the standard developing process. The DevOps philosophy presents developers with a way to improve a continued customer satisfaction, along with a way of reducing the time spent on repetitive tasks.

In conclusion, the course *DevOps, Software Evolution and Software Maintenance* has given us a new and different perspective into the development world. It has provided us with a new approach to handling the development of cloud services, as well as raised our current level of understanding regarding web development. DevOps is a concept and a philosophy that we hope to bring with us in the future, offering a new outlook on the challenges we are going to encounter not only in school, but also working professionally.

6 References

- DevOps Handbook (2016) - Gene Kim, Jez Humble, Patrick Debois, John Willis
- https://www.digitalocean.com/products/droplets
- https://dwheeler.com/essays/floss-license-slide.html
- https://docs.docker.com/compose/networking/
- https://docs.docker.com/storage/volumes/

- https://github.blog/2015-03-09-open-source-license-usage-on-github-com/
- https://www.discord.gg/
- https://github.com/WyriHaximus/github-action-next-semvers
- https://docs.docker.com/engine/swarm/
- NoSql vs Sql:
  - https://www.bmc.com/blogs/sql-vs-nosql/,
  - https://tinyurl.com/pk6badps,
  - https://tinyurl.com/ah4x4thn,
  - https://www.alexdebrie.com/posts/dynamodb-no-bad-queries/

7 Appendix

1. Editable figma files:
   https://www.figma.com/file/QHZrqRKHQ3I0fOFJN3Fcwu/DevOps2022?node-id=0%3A1
2. Security Assessment:
   https://github.com/Faurby/BlackOps/blob/main/Security_Assessment.md
3. Github Repository Workflows:
   https://github.com/Faurby/BlackOps/tree/main/.github/workflows
4. GitHub Repository Pull-Request:
   https://github.com/Faurby/BlackOps/pull/63

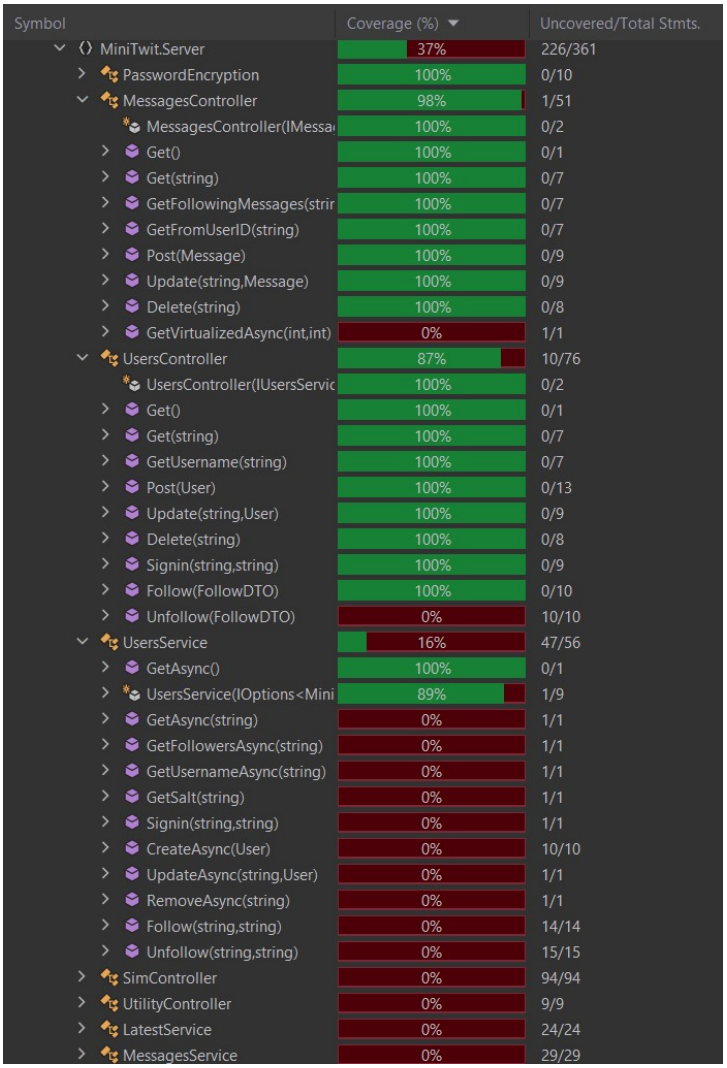

Figure 9: Test coverage of the server assembly.

5. *System dependencies:*

   Docker images:

   - Docker@20.10.16
   - Docker-Compose@1.25.0
   - Grafana@8.4.0
   - Prometheus@latest
   - FileBeat@7.17.1
   - ElasticSearch@7.17.1

- Kibana@7.17.1
- nginx@latest
- dotnet/sdk@6.0
- MongoDB@5.0
- Ubuntu@20.04

MiniTwit C# Application:

Server:

- BCrypt.Net-Next@4.0.2
- Blazored.LocalStorage@4.2.0
- Blazored.Modal@6.0.1
- Microsoft.AspNetCore.Components.WebAssembly.Server@6.0.1
- Mongo2Go@3.1.3
- MongoDB.Driver@2.14.1
- prometheus-net.AspNetCore@6.0.0
- Swashbuckle.AspNetCore@6.2.3

Server.Tests

- coverlet.collector@3.1.0
- Microsoft.Extensions.Options@6.0.0
- Microsoft.Net.Test.Sdk@16.11.0
- Mongo2Go@3.1.3
- MongoDB.Driver@2.24.1
- Moq@4.16.1
- xunit@2.4.1
- xunit.runner.visualstudio@2.4.3

Client

- Blazored.LocalStorage@4.2.0
- Blazored.Modal@6.0.1
- Microsoft.AspNetCore.Components.WebAssembly@6.0.1

Shared

- MongoDB.Driver@2.14.1

CI / CD Tools

Deploy

- actions/checkout@v2
- docker/login-action@v1
- fifsky/ssh-action@master
- Vagrant@2.2.19

Build and test

- actions/setup-dotnet@v1

Release maker

- WyriHaximus/github-action-next-semvers@v1
- actions/create-release@v1
- metcalfc/changelog-generator@v3.0.0

Snyk scanning

- snyk/actions/docker@master
- github/codeql-action/upload-sarif@v1

SonarCloud

- actions/setup-java@v1
- actions/cache@v1
- sonarscanner@latest

Yaml lint

- yamllint@latest