

Exam Report

Group I

Christian Skovsgaard Rieck (crie@itu.dk)
Daniel Stokholm Thomsen (dant@itu.dk)
Harpa Gudrún Hreinsdóttir (hahr@itu.dk)
Johan Fritze Neve (jone@itu.dk)

DevOps, Software Evolution and Software Maintenance
KSDSESM1KU

IT UNIVERSITY OF COPENHAGEN

Department of Computer Science
Denmark
1st of June 2022

Contents

1	Introduction	1
2	System's perspective	1
2.1	Architecture and Design	1
2.1.1	Frontend	4
2.1.2	API	5
2.2	Dependencies	5
2.3	Current state of the systems	7
2.4	Licenses	8
3	Process' perspective	9
3.1	Organization of Repository	9
3.2	Branching strategy	9
3.3	CI/CD chain	10
3.3.1	Flow of the CI/CD chain	10
3.3.2	Releases	11
3.3.3	Pull Request	11
3.3.4	Deployment	11
3.4	Development process	11
3.5	Monitoring	12

3.6	Logging	13
3.7	Security assessment	14
3.8	Critical incident	15
3.9	Scaling and load balancing	16
4	Lessons Learned Perspective	16
4.1	Evolution and refactoring	17
4.2	Operation	17
4.3	Maintenance	18
5	Future work	18
A	Dependencies of the project	b
B	Project artefact links	f

1 Introduction

This report covers various aspects of Group I's project in the course, DevOps, Software Evolution and Software Maintenance. The most important artefacts of the project are listed with their respective link in appendix B.

2 System's perspective

2.1 Architecture and Design

The system is a mini-version of Twitter, called MiniTwit. The system has two main packages, a Blazor frontend and a ASP.NET Core API. The API and Frontend decisions are described in detail in sections 2.1.1 & 2.1.2, respectively. The overall decision log can be found on our [GitHub Repository](#). See Fig. 1 for the package module view. The architectural views have been created based on the 3+1 model proposed by Christensen et al[1].

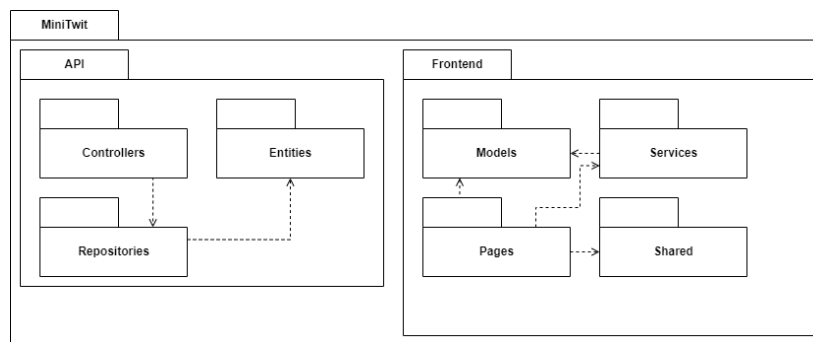


Figure 1: Package diagram

The system relies on various other services. The system uses ElasticSearch and Kibana for logging and Prometheus and Grafana for monitoring. The system is depicted in an overall context diagram in Fig. 2. The six mentioned services are part of a docker stack swarm, which is managed by one docker swarm manager machine and supported by one docker swarm worker machine. The system, specifically the API, interacts with a managed database system. The database and the two machines are located and hosted in DigitalOcean.

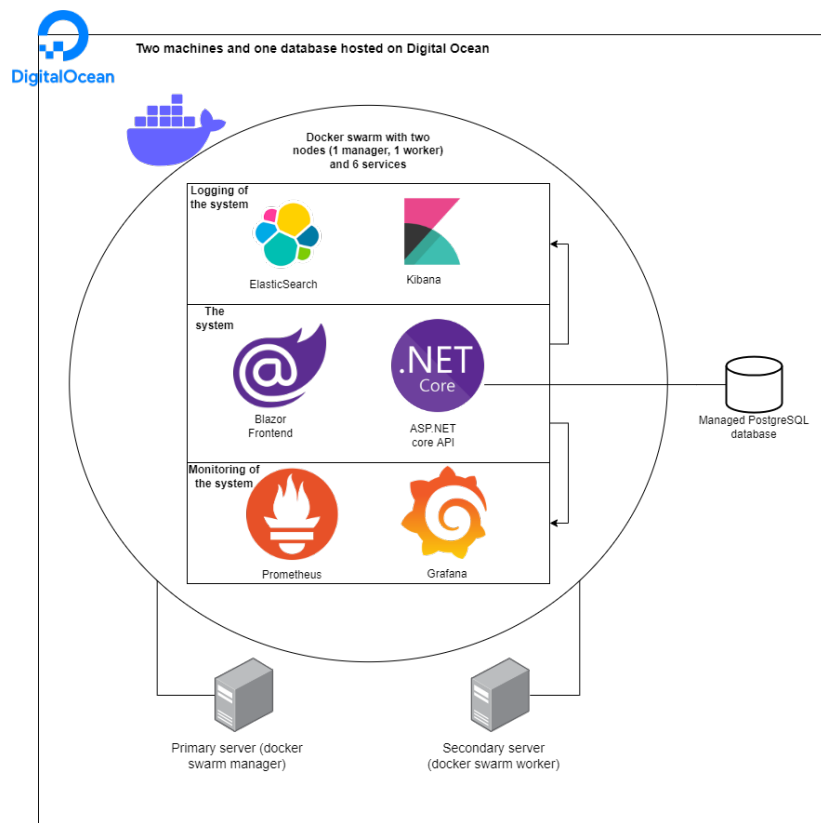


Figure 2: Context diagram

The Component & Connector view can be seen in Fig. 3, which shows how the different components of the system are connected. The greyed out components are external but are included to provide a better overview of how the system has been used in this project.

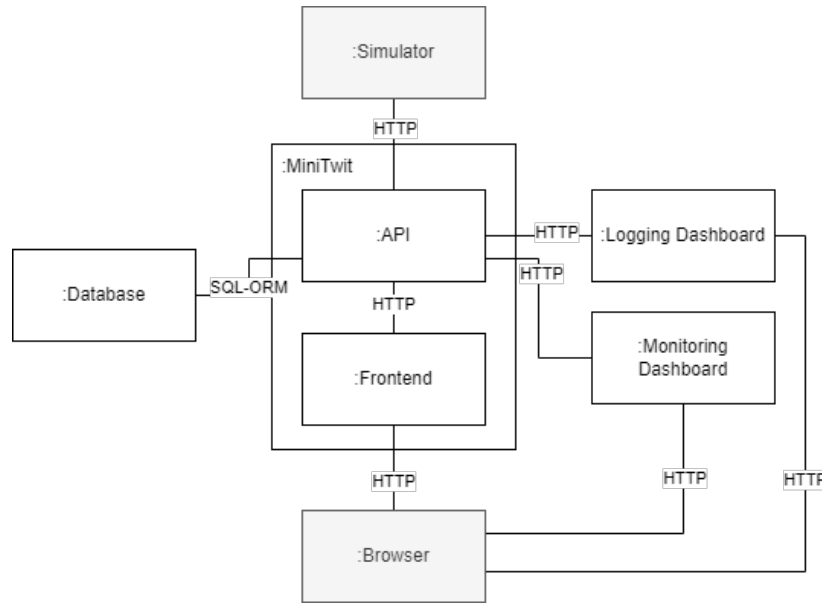


Figure 3: Component & Connector view

An example of the interactions of subsystems can be seen in the sequence diagram in Fig. 4. The example revolves around a user registration, and shows two scenarios, one where a user already exists and one where the user does not exist. The two scenarios return different status codes and communicate differently with the logging and monitoring components.

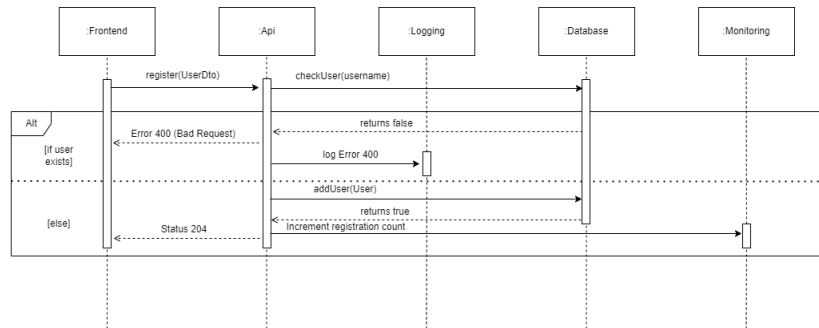


Figure 4: Sequence diagram

The deployment view of the system can be seen in Fig. 5. The different entities are as follows:

- **Digital Ocean Managed DB:** PostgreSQL database responsible for storing user data, following data and tweets hosted on Digital Ocean. Kept separately from the servers to ensure backups and a less error-prone environment for data storage.

- **Primary Server:** Ubuntu server responsible for managing the docker swarm.
 - **API:** ASP.NET Core API is responsible for communicating between the Frontend and database, acting as our backend.
 - **Frontend:** Blazor Frontend provides a user interface to view and create twits.
 - **Prometheus:** Providing metrics for monitoring with default Prometheus metrics and custom metrics for the API. The choice of Prometheus is based on industry standards.
 - **Grafana:** Monitoring dashboard based on metrics from Prometheus to provide business and technical insight into the running system. The choice of Grafana is based on industry standards.
 - **ElasticSearch:** Search for the aggregation of our logs sent from the application.
 - **Kibana:** A dashboard showing the logs aggregated in ElasticSearch.
- **Secondary Server:** Ubuntu server responsible for being a docker swarm worker and hosting the docker instances requested by the manager.

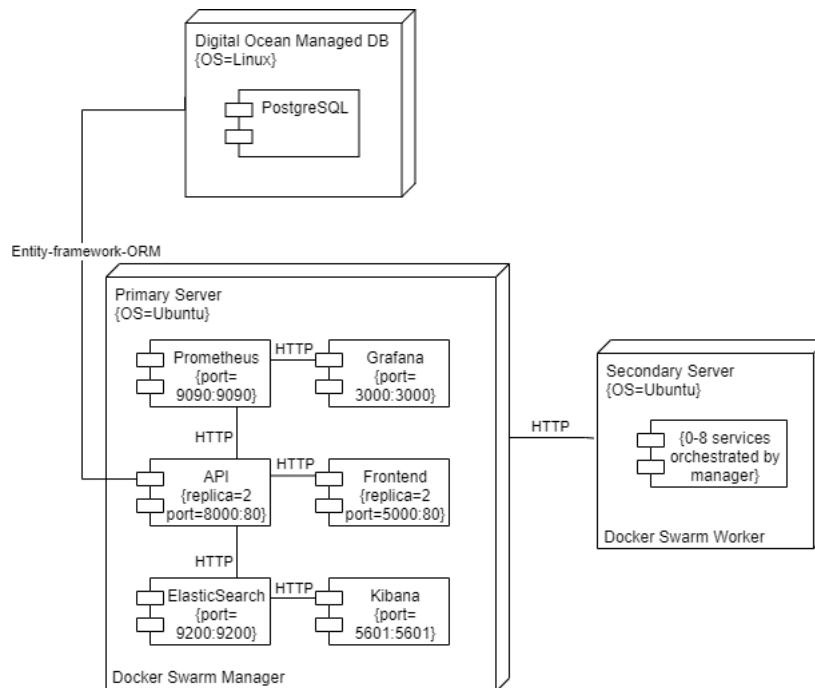


Figure 5: Deployment view

2.1.1 Frontend

The reasons why we decided to go with Blazor are based on the following factors:

1. It is convenient to use the same language pack for both the frontend and API
2. Simple routing, HttpClient and Dependency Injection

To store the state of the logged-in user we decided to go with a State class, which contains private functions to update the authentication state. When the state is changed it notifies the dependant classes (which have injected the State class).

2.1.2 API

We have chosen to do the API using ASP.NET Core using the shared data pattern. There are several reasons why we chose this stack:

1. Endpoints are automatically serialized to properly formatted JSON, which is being used in our frontend, as well as the simulator.
2. URL-Routing is done inline easily, as well as query parameters and request bodies, are automatically bound to method parameters
3. Logging of database interactions is automatically added. When we set up the ESK-stack¹, we have to only do minor additions to have meaningful logging in our entire application.
4. Entity Framework Core makes it easy to communicate with our database and is the base for the shared data architectural pattern choice.

2.2 Dependencies

The direct dependencies of the API and Frontend are visualized as a graph in Fig. 6. The graph shows us that the API has 16 dependencies and the Frontend has 7.

¹ElasticSearch-Serilog-Kibana

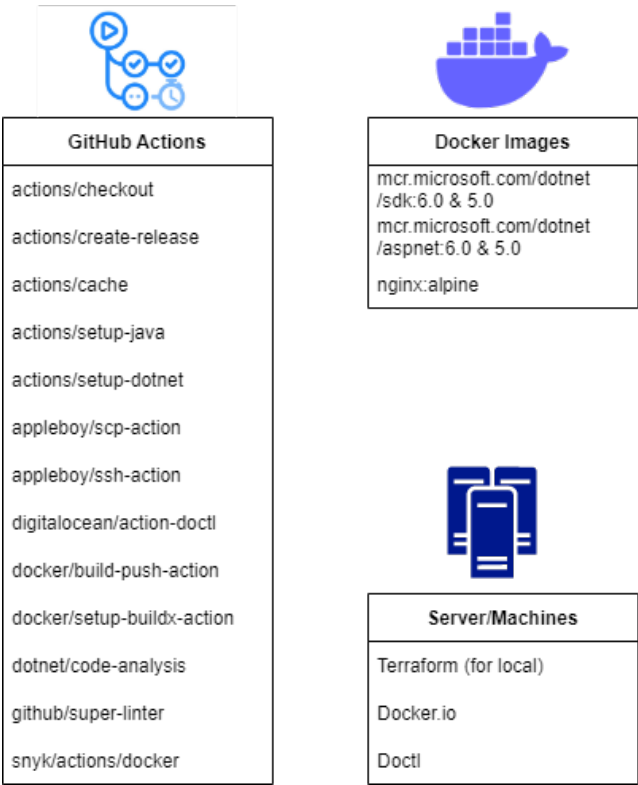


Figure 7: Other dependencies overview

An extensive list of all direct dependencies of this project, their name, description, and license can be found in appendix A.

2.3 Current state of the systems

The current state of the artefacts in the repository is summarized below, the state is found with different tools. The state as analyzed by Better Code Hub can be seen in Fig. 8, it shows 8/10 checks look good, however, automatic testing and short units of code are not passed. Another static analysis tool places our system in category C, which means there is 10-20% technical debt and they estimate it will take one week to clear out the debt. Moreover, the technical debt relates to 11 cases of code smells and 12 cases of code duplication. However, Sonar Cloud shows a better technical debt of less than 5%, but a lower reliability score due to bugs. The Sonar Cloud state can be seen in Fig. 9. The overall state of the system is acceptable, but it opens up for various maintenance needs.











	Write Short Units of Code	✗
	Write Simple Units of Code	✓
	Write Code Once	✓
	Keep Unit Interfaces Small	✓
	Separate Concerns in Modules	✓
	Couple Architecture Components Loosely	✓
	Keep Architecture Components Balanced	✓
	Keep Your Codebase Small	✓
	Automate Tests	✗
	Write Clean Code	✓

Figure 8: Better Code Hub state results



Figure 9: Sonar Cloud state results

2.4 Licenses

We are using a MIT license which is a permissive, free, and open source license. The choice of the license is due to wanting to support the open-source community and thereby not restricting the use. All of our direct dependencies

are using either MIT or Apache 2.0, which are both permissive licenses. That makes our MIT license compatible with all of our direct dependencies.

3 Process' perspective

We have aimed to adhere to the *"Three Ways"* as described in *"The DevOps Handbook"*[2]. A summary of our use of the three ways is described below, the initial internal notes concerning it can be found on our GitHub repository. A key task has been to simplify release to production in order to accommodate hotfixes and avoid big-bang releases. We use the GitFlow branching strategy together with, a Kanban board, and a streamlined CI/CD pipeline. The CI/CD pipeline is visualized in section 3.3.1. The Kanban board is described in section 3.4. In the kickoff phase of the project, we spent some time adjusting expectations to help establish psychological safety. Furthermore, we try to have shared responsibility for the work by doing code reviews.

3.1 Organization of Repository

We chose to version control all our code and files as a mono-repository on GitHub. The repository can be viewed here: [DevopsITUproject](#). Every project has its own folder. However, the two main projects, the Frontend and the API, share a .NET solution file, which is located in the root of the repository. We chose this option because we want to keep the number of repositories as low as possible, and also this makes our lives easier when working with the application since a change in the API often triggers some code changes to the Frontend and vice versa and therefore simpler to keep them in the same repository.

3.2 Branching strategy

We have been using the branching model known as GitFlow. This effectively means that we have a main branch for releases and a develop branch for development/testing. We avoid the extra step of making a release branch and merge directly to main from develop for simplicity. An alternative could have been Trunk-based, but since we were not experienced in DevOps, we chose to make it more strict, another important reason is that the repository is public and open for contributions. The contributions guidelines can be found on our GitHub repository. The workflow consists of the developer branching out from develop and after finishing their feature, they will do a pull request from their respective branch to the develop branch, where they have to assign at least one reviewer. When we are ready for a release, we create a PR from develop to main and once that is approved, our GitHub Action will automatically deploy

the artefacts to DigitalOcean, and our newly improved system will be in production. A visualization of the branching strategy in action can be seen in Fig. 10.

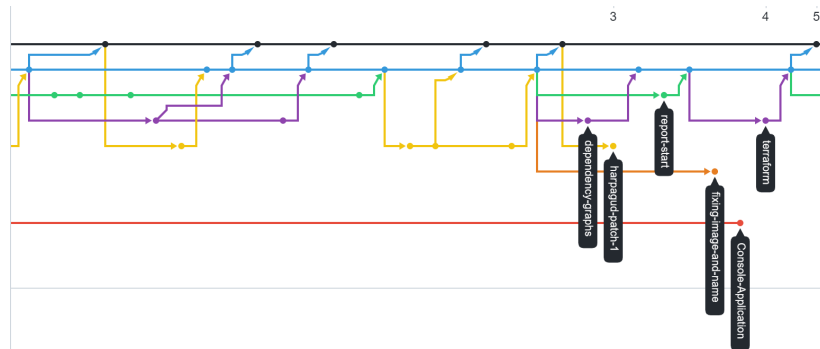


Figure 10: Branching visualization

3.3 CI/CD chain

3.3.1 Flow of the CI/CD chain

Fig. 11 is a visualization of the entire process from a developer's idea, to it being pushed to production.

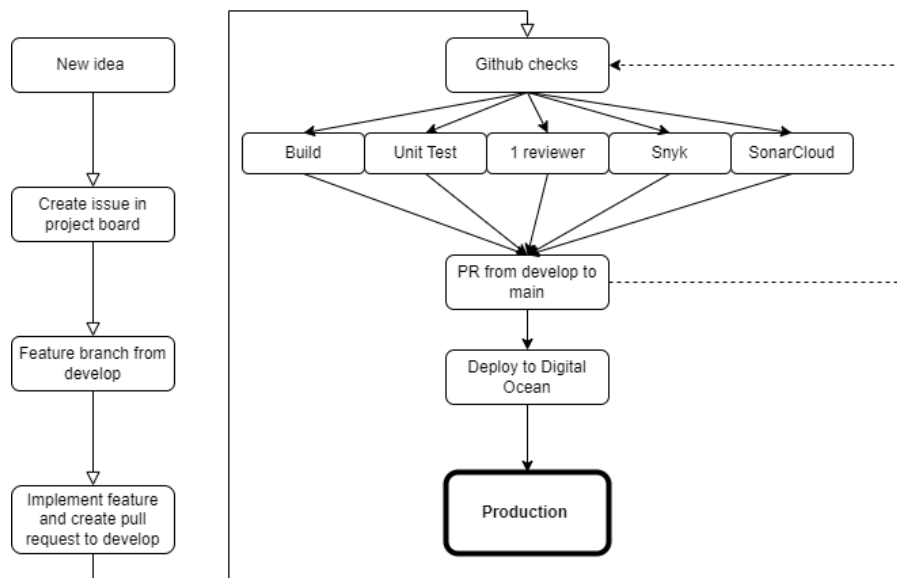


Figure 11: CI/CD flow

3.3.2 Releases

To ensure we have regular releases, we have set up GitHub actions that auto release. One of them does a biweekly release and the other one does a release on pushes to the main branch. The GitHub actions can be found at the following links:

1. Release on pushes to main
2. Biweekly release

3.3.3 Pull Request

To ensure some consistency and quality in our code we have added various checks that validate our code when creating a pull request, these are:

1. The API and Frontend are buildable and unit tests are run
2. SonarCloud Code Analysis
3. Better Code Hub
4. Code Climate
5. Snyk vulnerability scanning

3.3.4 Deployment

We have automated our deployment to production using a GitHub action: *deploy action*. This action will trigger on every push to the main branch and will publish our system on our servers. The action includes some checks, namely .NET code analysis, Snyk Docker image scan, unit tests, and linting of the codebase. The state of the checks is described in the decision log.

3.4 Development process

To have an overview of all issues and a platform to distribute and assign these, we have decided to use GitHub Project Board. The board is divided into categories, and each issue is assigned one of the following categories:

1. **Brainstorm/Ideas:** Where we keep track of new ideas.
2. **Todo:** Where we keep our issues that need to be done in the future.
3. **In Progress:** Here we keep our issues we are currently working on - We try to limit our work in progress to work in small batches to increase productivity. This also helps us in avoiding context switching, since we try to focus on one issue at a time.
4. **Code review:** Here we keep our issues which are to be code reviewed before merging to the develop branch.
5. **Ready for release:** Here we keep our issues which are approved and waiting in develop to be merged for a release to main.
6. **Done:** Here we keep our issues which are done and deployed to main.

Each issue also has an assigned person. Combined with the status of the issues, we have a neat overview of open tasks.

The overview can be seen in Fig. 12

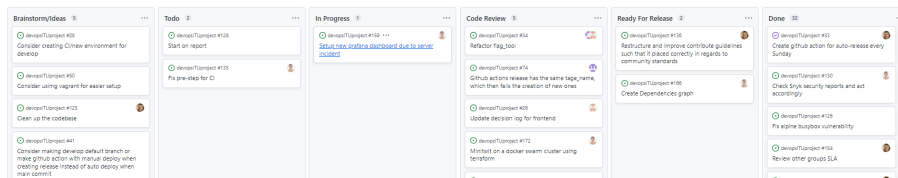


Figure 12: GitHub Project Board

3.5 Monitoring

We are using Prometheus and Grafana as the monitoring platform for our application. There is a ASP.NET Core package² for Prometheus so it is easy for us to incorporate this into our system.

We monitor several metrics in our system. The business metrics are:

1. The number of registered users since the last release
2. The number of times users that have logged in to our system since the last release

These are metrics which can be used by the business responsible to measure the growth and size of our application or if a new update is well received. The business metrics can be seen in Fig. 13.

²<https://www.nuget.org/packages/prometheus-net.AspNetCore/>

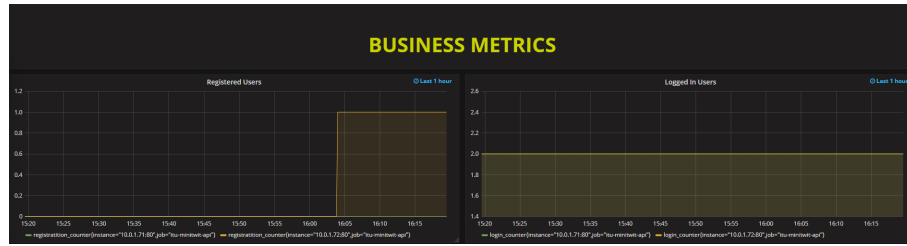


Figure 13: Business metrics visualized from Grafana

The monitoring also includes technical metrics that can help with server/system maintenance, these are:

1. The total user and system CPU time spent
2. The amount of http requests to our application
3. Occupied memory

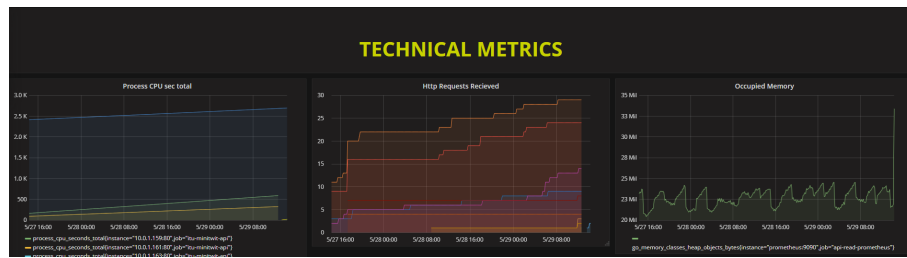


Figure 14: Technical metrics visualized from Grafana

3.6 Logging

We have decided to use the logging as is in Entity Framework. This means that every time our applications send a command to the database, whether it is inserts, updates, or deletes, it is being logged. This means that all interactions with the API are being logged, and since the Frontend mostly consists of API requests, we have a transitive logging relation to the Frontend.

We have decided to use a variation of the popular EFK-stack which is a logging aggregation ecosystem. First of all, we use Serilog, which is a .NET logging library, to get all of our logging information sent to ElasticSearch. We then use Kibana to better query and visualize the data stored in the ElasticSearch database. We then have the possibility to filter the data on a lot of criteria which can give us good troubleshooting capabilities.

An example of the visualization of the logging can be seen in Fig. 15:

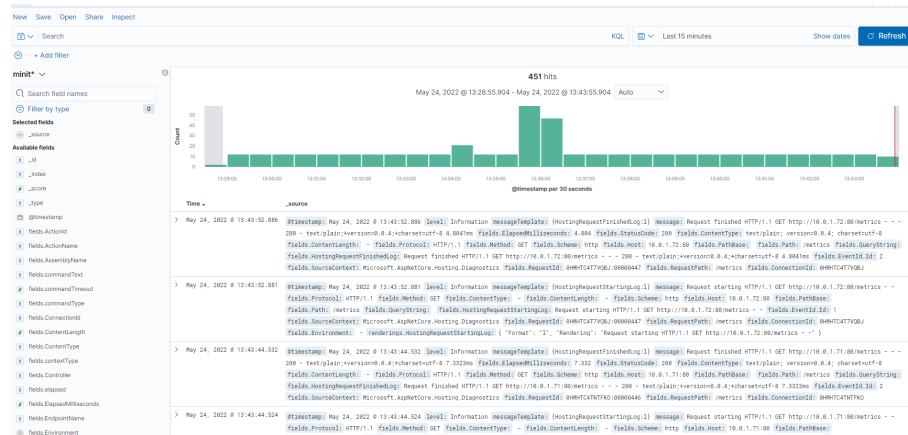


Figure 15: Logging visualization

3.7 Security assessment

The system has been assessed with regards to security with two automatic penetration testing tools and a manual penetration testing. The whole security assessment report can be found in Our GitHub Repository, this summary contains references found only there. The results are as follows:

- Metasploit WMAP on Kali has been run and it shows no vulnerabilities
- Skipfish on Kali has been run and it shows nothing important, but indicates missing charset
- The user info API endpoint sends the password in plain text (everyone can see this). This is broken access control and relates to risk no 6.
 - The hashing of the passwords seems broken
 - It is possible to use this data to log in as other users

The actions points based on the results are:

- Implement countermeasure for risk no 6
 - Ensure proper salting, hashing and authentication
 - Ensure proper endpoint security
 - Ensure no unnecessary data is sent as responses

- Switch from HTTP to HTTPS for enhanced security

Furthermore, OWASP³ includes insufficient logging and monitoring as a security risk. The current state of our logging is acceptable since it is possible to manually see some results from the penetration test, such as tools trying to access config sub-sites. However, nothing is automatic. Furthermore, the monitoring does not give much information, other than a few technical and business-related metrics, such as the numbers of logins. The security issue concerning risk no 6 that was found, can't be seen with the current logging/monitoring.

3.8 Critical incident

The group lost access to the first server of the project. It was no longer possible to log in via SSH-keys or in the DigitalOcean UI. The server was put in recovery mode, but access was still not possible to obtain even when resetting the root password. The solution was to quickly deploy new servers, but now with two machines using Docker Swarm. The downtime was about 4 hours mostly due to creating a better setup with Docker Swarm and little manual configuration of the servers. It was not possible to find any suspicious activity, moreover, there was no extra load on the CPU or other metrics of the server, which indicates that no one has installed a BitCoin miner or other types of software that exploits our server. The root cause was never found, so the best corrective measure is to create the infrastructure as code. We now have support for Terraform, but this was not implemented at the time of the incident. The full incident report can be found on our GitHub repository.

Moreover, the incident can clearly be seen on the graph in Fig. 16, which shows how many requests our API accepts at specific times. There are notable drops, the two first being lack of a green-blue deployment strategy. The third and biggest is our server incident, due to lack of updating the API URL for the simulator even though the actual downtime was only 4 hours. The last drop is simply the end of the simulator.

³<https://owasp.org/>

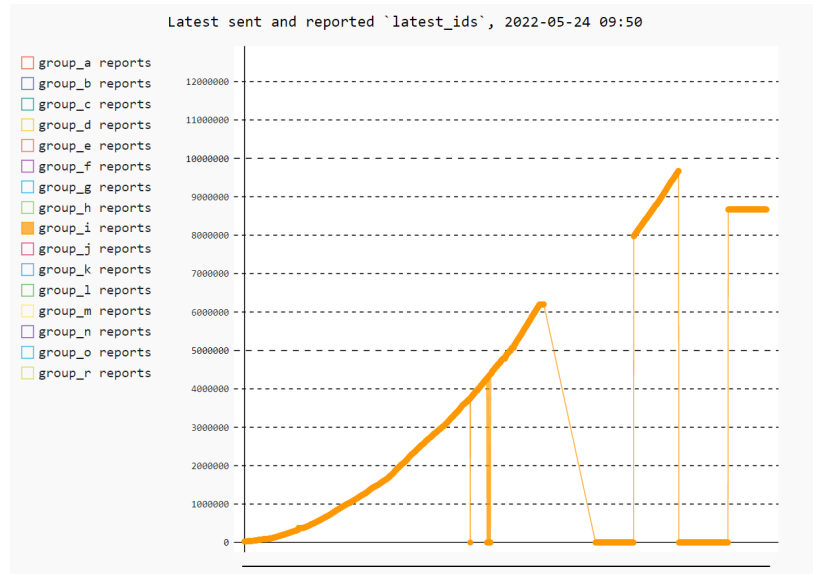


Figure 16: Number of accepted API request over time

3.9 Scaling and load balancing

The initial scaling was vertical since the logging service used up all the CPU, and more resources were needed. However, to avoid having a single point of failure, horizontal scaling was added via two machines and replicas for Docker containers. The scaling and load balancing is set up with Docker Swarm. We have six Docker Services as a Docker Stack, two of them have two replicas, the API and Frontend. The scaling is static with Docker Swarm and more replicas are not created during higher loads. The Docker Swarm has one manager and one worker node. The manager is responsible for load balancing with the default Ingress routing load balancer. However, having only one manager node is a single point of failure, and in a real setup, there should be more managers and workers to ensure smooth operation. The deployment uses the Blue-Green strategy to ensure that a new upgrade starts before the current one stops, and then it shifts the load to the new one when it is ready. However, it is important to note that the database is not included in this strategy, which could yield problems if there are migrations to the database in the update.

4 Lessons Learned Perspective

There are numerous reflections on incorporating a DevOps style compared to previous development projects, these include:

- **Emphasize automation.** Building automation into the development life-cycle is critical to ensure that the software releases are consistent. Also, it allows the programmers to be liberated from less significant and time-consuming tasks to focus on processes that require more thinking and creativity.
- **Using Docker.** Docker makes it easier to build and run the code in any setup to avoid on-boarding hassles for developers. This was even more empathized in the beginning before docker was added to the project, since developers were using different versions of .NET and different operating systems.
- **Static analysis of codebase.** Using different static analysis tools helps determine areas to improve in the codebase, which could have been left unnoticed otherwise.

4.1 Evolution and refactoring

Refactoring the database using Object–Relational Mapping

A lesson that we can take along with us is the fact that we can utilize a lot of support given by the programming languages and its associated framework and libraries. When we were first given the task to refactor the python code, we chose to do it using .NET (reasons described in 2.1.2), here we were given the complete SQL schema, we could then have used the effective scaffolding of the Entity Framework, to efficiently create all the model classes and the database context using the existing database. The issue regarding this can be seen at: Issue 11

4.2 Operation

Importance of following separation of concerns when hosting on multiple servers

One of the lessons we learned during the project, is the importance of separating concerns when writing code that is to be hosted on different servers. This came to importance when we were implementing logging. We are using a logging library, Serilog, and when we configured it we had to connect to the right ElasticSearch database. We had to make sure it worked in test before deploying it to production. However, since our test environment is different from our production, the ideal solution would be to do the configuration in a higher level of abstraction, meaning that we never have to hard code the IPs of the servers we are connecting to. Not having this abstraction leads to a wide range of commits. Staring with commit 30149bd and ending with commit 686ee38 The ticket relating to these commits can be seen here: *Add Logging to the System* Luckily this issue was resolved but took a significant amount of time, which could have been spent more wisely.

Difficulties testing operation and configuration leads to many fails and commits

An important lesson concerned the configuration of automation of workflows, like the GitHub Action used to deploy the latest main branch to Digital Ocean. Operations and configuration are hard to test, the deploy action can only be tested by actually deploying to production. This led to many small commits in order to adjust when the action failed. It would have been nice to have a similar setup for the develop branch, such that the configuration could be tested without affecting production. A develop environment was not prioritised due to the costs associated. An example is on the 14th of March, when there were around 20 consecutive commits to try and fix the deployment of the system. Everything from missing nginx in the docker image to serve the static contents of the Blazor frontend and including environment variables in the deploy action. It starts with commit 3cd6a11 and ends with commit 109eb16.

4.3 Maintenance

Cleaning old code to ensure a clean codebase

An important part of a software life cycle is maintenance, correcting faults and improving the existing design. This also goes for our application. One of the things that is in need of cleaning is the file and folder structure of our code. As the application grows there is quite a big chance that the folder structure should evolve with it. However, taking the time and effort to do so, should be of a higher prioritization than what we have done in the project. However, it is very difficult to find the time to go back and optimize when we have to stay ahead of the curve with the deadlines of the following week. These were addressed in the issue 125 this has sadly been lying dormant for a long time. But for coming projects, we clearly see the value of having a clean base to develop upon.

5 Future work

There are various artefacts of the system that could have an improved setup, these are, but are not limited to:

- Elasticsearch hosted on its own machines to avoid the logs being dependent on the same server as the API. Moreover, the heavy load of Elasticsearch won't affect the other services.
- Better monitoring and logging to ensure better security and traceability. Especially alerts when something seems off.
- Add file change policy to CI/CD pipeline to avoid building unnecessary parts and speed up the process

- Enforce quality gates and check requirements both for pull requests and the deployment GitHub action to ensure overall quality
- Add an extra environment as Production but for the develop branch. A development/testing environment could help to avoid problems and coding/configuring in the dark. This, however, could slow the deployment, but ensure better quality assurance.

References

- [1] Henrik Bærbak Christensen, Aino Corry, and Klaus Marius Hansen. “The 3+ 1 Approach to Software Architecture Description Using UML Revision 2.4”. In: *Workingpaper, Århus Universitetsforlag* (2016).
- [2] Gene Kim et al. *The DevOps handbook: How to create world-class agility, reliability, & security in technology organizations*. IT Revolution, 2021.

A Dependencies of the project

Below is a list of all the dependencies we are using in our project. Some of them have a strike-through which means we did not end up using them but they are currently still in our solution. One of the part of the maintenance phase is to get rid of dependencies that are not being used and we intend to do so in our solution as well.

- Frontend dependencies:

- Microsoft.AspNetCore.Components.WebAssembly:

Description: Blazor Webassembly allows developers to create a SPA (single-page applications).

License: MIT

- Microsoft.AspNetCore.Components.WebAssembly.Authentication:

Description: Helps the app to authenticate users and obtain tokens

License: MIT

- Microsoft.AspNetCore.Components.WebAssembly.DevServer:

Description: Server to use when building Blazor Applications

License: MIT

- **Microsoft.Extensions.Http:**

Description: Used to configure HttpClient

License: MIT

- ~~Microsoft.VisualStudio.Azure.Containers.Tools.Targets:~~

~~Description: Enables Visual Studio Tooling for Docker file~~

~~License: MICROSOFT SOFTWARE LICENSE TERMS~~

- Newtonsoft.Json:

Description: Used to serialize/deserialize objects from C# to Json

License: MIT

- System.Net.Http.Json:

Description: Provides methods for the HttpClient to serialize/deserialize without using System.Text.Json

License: MIT

- Backend dependencies:

- **DotNetEnv:**
Description: .NET library to load environment files.
License: MIT
- ~~Merio.Configuration.Provider.Docker.Secrets~~

~~Description: Maps docker secrets files to .NET core configuration ⁴.~~

~~License: MIT~~
- **Microsoft.EntityFrameworkCore:**
Description: A mapper from database to objects for .NET
License: MIT
- **Microsoft.EntityFrameworkCore.Design:**
Description: Package containing all logic for EF-Core.
License: MIT
- **Microsoft.EntityFrameworkCore.SqlServer:**
Description: Relational database management system.
License: MIT
- **Microsoft.EntityFrameworkCore.Tools:**
Description: Manages database migrations through dbcontext
License: MIT
- **Microsoft.VisualStudio.Web.CodeGeneration.Design:**
Description: Generates boilerplate code for web apis
License: Apache-2.0 license
- **Npgsql.EntityFrameworkCore.PostgreSQL 6.0.3:**
Description: Allows .NET to interact with PostgreSQL
License: PostgreSQL license
- **prometheus-net.AspNetCore:**
Description: Used for exporting metrics to Prometheus
License: MIT License
- **Serilog.AspNetCore:**
Description: Tool for logging
License: Apache-2.0 license
- **Serilog.Enrichers.Environment:**
Description: Tool for logging
License: Apache-2.0 license
- **Serilog.Exceptions:**
Description: Add on to Serilog for logging exceptions
License: MIT License
- **Serilog.Sinks.Debug:**
Description: For logging information to Visual Studios debug output window.
License: Apache-2.0 license
- **Serilog.Sinks.Elasticsearch:**
Description: A writer (sink) for the Serilog logging framework.
License: Apache-2.0 license
- **Swashbuckle.AspNetCore:**
Description: Swagger tool for APIs build with ASP.NET Core ⁵
License: MIT license

⁵<https://github.com/domaindrivendev/Swashbuckle.AspNetCore>

- Build/release dependencies:

- actions/checkout

Description: Checks out your repository under \$GITHUB_WORKSPACE, so your workflow can access it.

License : MIT License

Description : creates a GitHub release.

License : MIT License

- actions/cache

Description: Attempts to restore a cache depending on the key you provide

License: MIT License

- actions/setup-java

Description: Downloads and sets up a requested version of Java

License: MIT License

- actions/setup-dotnet

Description: Downloads and sets up a requested version of Dotnet

License: MIT License

- actions/scp-action

Description: Copies files and artifacts via SSH

License: MIT License

- actions/ssh-action

Description: Executes remote ssh commands

License: MIT License

- digitalocean/action-doctl

Description: Allows you to interact with all of your DigitalOcean resources.

License: MIT License

- docker/build-push-action

Description: Builds and pushes Docker images with Buildx

License: Apache-2.0 license

- docker/setup-buildx-action

Description: Creates and boots a builder that can be used in some steps of a workflow when using buildx

License: Apache-2.0 license

- dotnet/code-analysis

Description: runs the .NET code quality ("CAxxxx") and code style analyzers("IDExxxx")

License: MIT License

- github/super-linter

Description: Runs the Super-Linter, which is a simple combination of various linters.

License: MIT License

- snyk/actions/docker

Description: Uses Snyk to check for vulnerabilities in GitHub projects

License: Apache-2.0 license

- API Testing:

- coverlet.collector

Description: Code Coverage library with support for line, branch

and method coverage
License: MIT License

- **Microsoft.EntityFrameworkCore.InMemory**

Description: In-memory database provider for Entity Framework Core
License: MIT License

- **Microsoft.NET.Test.Sdk**

Description: To build .NET test projects
License: MIT License

- **xUnit**

Description: Unit testing tool for the .NET Framework
License: Apache-2.0 license

- **xunit.runner.visualstudio**

Description: Required to run the test project inside Visual Studio as well as with dotnet test
License: Apache-2.0 license

- **Ubuntu servers/machines:**

- **Docker.io**

Description: Docker is a containerization platform which enables applications to be packed as containers

- **Doctl**

Description: Command line interface for the Digital Ocean API in order to allow communication with our container registry

- **Terraform**

Description: Infrastructure as Code (used on local machines)

- **Docker images:**

- **mcr.microsoft.com/dotnet/sdk:6.0 & 5.0**

- **mcr.microsoft.com/dotnet/aspnet:6.0 & 5.0**

- **nginx:alpine**

B Project artefact links

The following table provides link to every significant artifact of the project.

Artifact	Link
GitHub Repository	https://github.com/Arklaide/devopsITUproject
Security Assessment	https://github.com/Arklaide/devopsITUproject/blob/main/report/sub-reports/SecurityAssessment.md
Three Ways	https://github.com/Arklaide/devopsITUproject/blob/main/report/sub-reports/ThreeWays.md
Contributions	https://github.com/Arklaide/devopsITUproject/blob/main/report/sub-reports/contributions.md
Decision Log	https://github.com/Arklaide/devopsITUproject/blob/main/report/sub-reports/decision_log.md
Incident Log	https://github.com/Arklaide/devopsITUproject/blob/main/report/sub-reports/incident_log.md
Kanban Board	https://github.com/users/Arklaide/projects/2/views/4
Frontend Application	http://164.92.132.67:5000
API Public Endpoint	http://164.92.132.67:8000/public
Kibana Logging Dashboard	http://164.92.132.67:5601
ElasticSearch	http://164.92.132.67:9200
Grafana Monitoring Dashboard	http://164.92.132.67:3000
Prometheus	http://164.92.132.67:9090