DevOps, Software Evolution & Software Maintenance Rapport

Course code: KSDSESM1KU

Group O - Oh sorry

Repository: `https://github.com/Group-O-Minitwit/minitwit-group-o`

Christoffer Tofteng (chtof@itu.dk)
German Alexander Garcia Angus (gega@itu.dk)
Hristiyana Hristova Toteva (hrto@itu.dk)
Lasse Emil Sonn (lson@itu.dk)
Mikkel Munkholm Blak Nilsson (muni@itu.dk)
Nohely Gedeon (noho@itu.dk)

May 2023

Contents

1 Introduction

This paper aims to describe the our experience on working with development of the simplified twitter clone - MiniTwit. Our primary focus has been on designing a reliable system, which is transparent and easy to maintain.

2 System's Perspective

2.1 Design of the systems

We have designed our system in a monolithic structure[1] where all our components are build and deployed together. The system has been separated into smaller components in order to maintain a loosely coupled application. These components consist of a frontend system, a remote server, a database, a monitoring system, and a logging system.

2.2 Architecture of the systems

Our version of the MiniTwit application consists of a web application, an API, and a database. The application is inspired by Twitter. It allows users to create a profile, log in, post short messages to a public timeline, and follow and unfollow other users. The web application is supported by our API, which has endpoints to handle those functionalities. The frontend was developed using the JavaScript React.js library with Node.js as the runtime environment. The API was built with C# using the .Net framework. A more in-depth discussion as to why we chose these technologies for our system can be found in section 2.7.

[1] Awati et al.

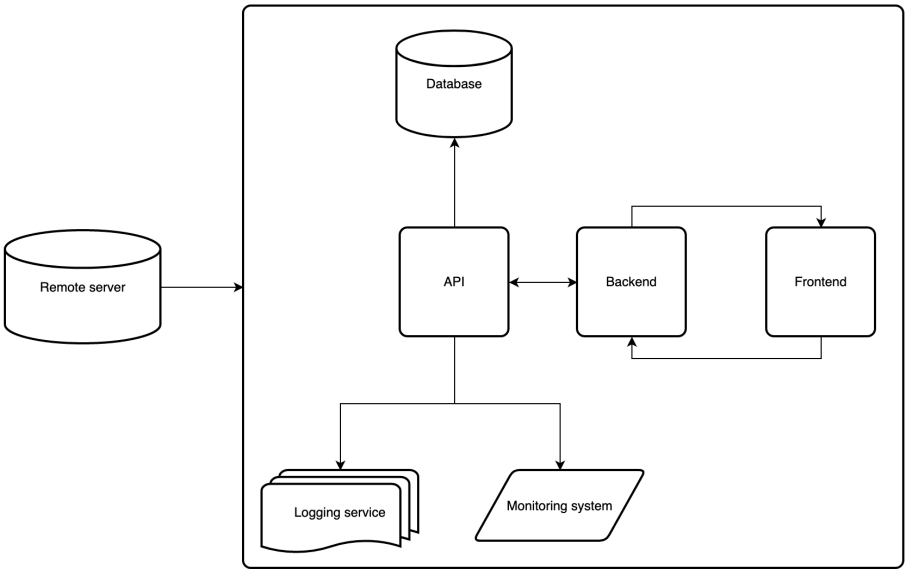

Figure 1: Architecture of our software

2.3 Dependencies

For a visualization of the dependencies please see figure 2

- **C#** [12]
  We use C# and the .NET framework to handle all the API and data manipulation in our application.
- **React** [19]
  The React library is used to visualize our data through the frontend with JavaScript
- **PostgreSQL** [7]
  We're using PostgreSQL as our database system.
  - **npgsql** [15]
    The database for our project is NPGSQL which is a .NET version of a Postgres database.
- **Prometheus** [1]
  Prometheus collects and stores the metrics of our application. The metrics can be access with Prometheus's queries that help to connect with the Grafana monitoring tool.
- **Grafana** [9]
  To visualize and analyze our monitoring data we are using Grafana, which

shows different metrics that can be access by using the Prometheus's queries.

- **SeriLog** [23]
  SeriLog is a C# logging tool that creates logs and formats them before sending them to ElasticSearch.
- **ElasticSearch** [20]
  This tool is a data ingestion toool used to index the logs send from SeriLog in order to allow for easier access.
- **Kibana** [22]
  Kibana is used to query the logs from ElasticSearch and visualize the data from the log.
- **filebeat** [21]
  Filebeat is a logshipper used to ship the logs to elasticsearch
- **nginx** [3]
  We are using the proxy service nginx for port forwarding in relation to our logging.
- **Docker** [8]
  We're using docker to containerize our application. This is done to keep the setup intuitive and makes it easy to do continuous deployment. Together with docker we're specifically using docker-compose which helps centralize the setup to a single yaml file.
- **DigitalOcean** [10]
  DigitalOcean is used to host the server which runs our application. The server is running an image of Ubuntu 22.04 with Docker preinstalled. Together with this we have a static IP with locks our application to a specific IP, so we have the possibility to change the remote server but keep the same IP.
- **GitHub Actions** [5]
  Github Actions is used for the CI/CD pipeline.
- **Static analysis tools** We have InferSharp, CodeQL and SonarCloud running in our CI/CD chain to ensure software quality in what we are committing to the project.
  - **InferSharp** [13]
    InferSharp is the C# version of the static analytic tool Infer used by Facebook. InferSharp both analysizes for null dereferences, resource leaks, and thread-safety violations and has a security check for the likes of SQL injections.

- **CodeQL** [4]
  CodeQL is a static analytics tool that run through the code and treats it like data. Therefore it is possible to also make CodeQL custom queries, however, we did not make use of that feature as our codebase is not that big and complex and the standard analysis is enough.
- **SonarCloud** [18]
  SonarCloud is a quality measuring tool to give your committed code a grade based on some predefined parameters that may or may not be relevant. In our case it checks for duplicated lines, "smelly" code, some security and reliability. In addition to the two "heavier" analyzers, we think SonarCloud seems like a solid one to handle the things that the two first does not.

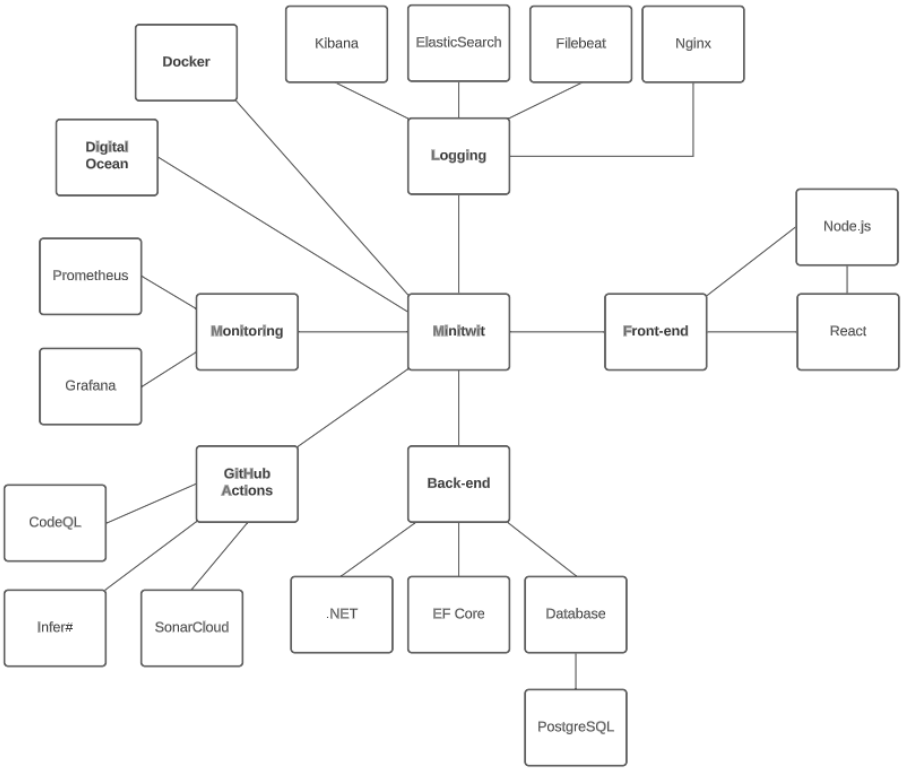

Figure 2: Structure of our dependencies

2.4 Important interactions of subsystems

Given that the frontend is built independently from our API with the frontend being React and the API being C# the frontend accesses endpoints of the API in order to visualize the data and send new data if a new tweet or user is created.

2.5 Current state of the system

The only tool we use that can really give us a current picture of our system is SonarCloud. As you can see on figure 3 below we get the grade A based on the parameters we have chosen.

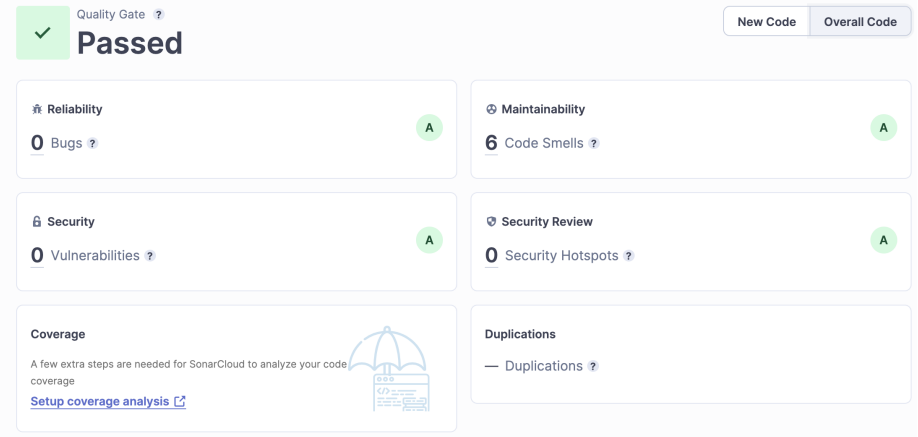

Figure 3: Current state of the system

2.6 Licensing

As there was no explicit license from the original Minitwit application we copied we see this program as our own minitwittware and therefore will have to license it ourselves. If we take a look at the denpendencies we use in the project and their licenses we see the following.

- prometheus
  Apache 2.0 license
- grafana
  GNU AGPLv3 license
- SeriLog
  Apache 2.0 license
- ElasticSearch, Kibana and filebeat
  Elastic License and Server Side Public License (SSPL) (Elastic License v2, or ELv2)

- nginx
  2-clause license ("Simplified BSD License" or "FreeBSD License"
- .NET (C#)
  MIT license
- npgsql
  PostgreSQL License
- React
  MIT license

Looking at the licences above the GNU AGPLv3 license seems like the strongest and since it is free it makes sense to go with the same[2]. The GNU AGPLv3 is a strong copyleft license which we feel like makes sense for the project in case other people wants to help and make changes or use it as an inspiration for their own work then they have a starting platform.

2.7 Technologies

2.7.1 Programming language and Framework

For the backend we had to choose another language than python. This made our choice to quickly be between Go or C# as we knew this were a course with a steep learning curve and we wanted the refactoring of the API for the application not take much time from other contents of the course. The reason for these two despite other recommendations was primarily the amount of documentation and resources available online when stuck on a problem[3].
The reason why we chose C# over Go comes down to that most of us are more comfortable with the object oriented nature of C# and it is an industry standard. By using this project as a portfolio project, it could make us more attractive for future job opportunities.

For the frontend we chose React.js. The choice for us here was either React or .NET's own frontend framework Blazor. Choosing Blazor would allow us to keep programming in C# for the frontend.

The reason why we chose React and JavaScript over Blazor was because React already is used by Twitter, so for making a Twitter clone, it seems natural to use the same framework. React is also faster given it is browser native where as C# uses web assembly. Again we also use the argument of attractiveness on the job market where very few applications are written in Blazor. Like C#, React has gained a lot of following since its release, which makes it easy for us to find resources and documentation for the language.

[2] https://github.com/Group-O-Minitwit/minitwit-group-o/blob/main/LICENSE.md
[3] Stack Overflow 2022

2.7.2 Virtualization techniques and deployment targets

We have chosen to deploy our application on DigitalOcean as we as students get free credits and therefore didn't have to use any money ourselves.
As virtualization technique we have chosen to go with a droplet which has multiple containers which is all constructed through our `docker-compose.yml` file[4].

We have chosen not to go with Vagrant since it added a layer of unnecessary complexity and because our application is so small that there is not real reason for managing our VM.

Our deployment target is docker 20.10.21 with ubuntu 20.04 as this is the newest LTS of Ubuntu making it a safe deployment target. Our implimentation with

2.7.3 ORM framework and DBMS

For the Object Relation Mapping, we're using ASP.NET Core. This is a .NET library which allows us to easily create models which directly relate to the database and set up API endpoints. This framework is the obvious choice when you're creating an API closely related to a relational database. This was for some people a new technology, but since it is widely used[24], it's always a good tool to know.

For the Database Management System (DBMS) we're using PostgreSQL[7]. This is partly because of how easy it is to set up, both with integrating with ASP.NET Core and with its standalone docker image. Another reason why we chose this management system is for it's data types. When working with C#, it's very easy to directly transfer the standard types to the PostgreSQL types.

3 Process' perspective

3.1 Developer interactions

For internal communication in the group we have created a Slack workspace to ask questions, arrange meetings and share documents. We have used Microsoft Teams to facilitate our meetings as teams is also used in the class and offers a more reliable platform for online meetings compared to Slack.

We had biweekly scrum-like meetings Tuesday and Friday to give an update on what tasks you had done, what you had started on, and if you needed any help with an ongoing task.

We kept track of work in progress and future work by using GitHubs issues function. In the beginning of the project we also used the projects option as a sort of kanban board in order to track when the application was refactored and which steps where still needed.

[4]https://github.com/Group-O-Minitwit/minitwit-group-o/blob/main/remote_files/docker-compose.yml

We have used pull requests in order to have some quality assurance and outside perspective of work when a task was done. A pull request to the main branch had to be approved by at least one other member before it could be merged.

3.2 Team organization

We are a team of 6 student developers each with different time schedules and therefore we have split us into smaller teams responsible for backend and frontend according to our schedules. We primarily made use of pair programming, to ensure some sparring, when developing new features.

3.3 Stages and tools included in the CI/CD chain

In our CI/CD pipeline everytime there is a push to the main branch, the `continous-deployment.yml` script[5] builds, delivers and deploys the application to DigitalOcean. This script contains multiple steps starting with running tests as we do not want to deploy faulty software. After the tests passes we build and push the backend and frontend to DockerHub. When this is done we configure the SSH and copy our files from the `remote\_files` directory onto the server as it contains the `docker-compose.yml`[6] and `deploy.sh`[7] so that the server always has the latest files

3.4 Repository organization

We have decided to use github for our repo management. This is something we were encouraged to use and a free service that we also had experience with in advance. We have chosen a mono-repo approach as the project is not going to be so big that more repositories would be necessary. There could be arguments for a dual repo setup with a repo for the backend and another for the frontend but sticking to a single made more sense as the arguments for a poly-repo setup are not strong enough in the case of 2 repos over a single. This also helped keeping complexity to a minimal when we had to deploy to the server.

3.5 Applied branching strategy

We used a version of the GitHub Flow[8] strategy to manage our branches. This involves having our main branch, which is the code we release to the world, a Development branch where we would merge features in order to ensure nothing breaks in the grander scheme and finally we would create feature branches based on GitHub issues to implement features. Using this branching strategy easily

[5] https://github.com/Group-O-Minitwit/minitwit-group-o/blob/main/.github/workflows/continous-deployment.yml

[6] https://github.com/Group-O-Minitwit/minitwit-group-o/blob/main/remote_files/docker-compose.yml

[7] https://github.com/Group-O-Minitwit/minitwit-group-o/blob/main/remote_files/deploy.sh

[8] GitHub

gives an overview of which issues are being worked on as the branches names are usually `feature/issue-number/name-of-feature`

3.6 Applied development process and tools supporting it

We have decided to split up our work throughout the group. The group members would have different responsibilities for parts of the website such as frontend, backend etc.
The reason for this partitioning was because we all have different schedules, it would not be feasible for a team of 6 to be a part of everything.

To support our process we used the GitHub tools of creating issues as also written above in section 3.5. Another tool we used at the start was the projects tool which is a kanban board as described in section 3.1.

3.6.1 How do we expect contributions to look like?

A contribution should only be allowed into the development branch if the code is written in a readable manner, is tested and solves an issue contained in the backlog. The contributions has to pass the github actions checks unless there could be made arguments for a failing check is redundant. This is to ensure that no feature of functionality, which could break the app, is deployed to the server.

3.6.2 Who is responsible for integrating/reviewing contributions?

Every member in the group is responsible for reviewing contributions but a contributor will request reviews from team members more knowledgeable about the code and issues.

3.7 Monitoring

Our MiniTwit application supports monitoring with Prometheus and Grafana as a dashboard. We configured the setting for Prometheous and Grafana in the the `prometheus.yml`[9] and the docker-compose file. This allowed us to access Grafana, Prometheus, and their metrics, enabling us to build our dashboard.
Monitoring is an essential aspect of managing IT systems and applications, providing feedback on their performance and quality of service. In our MiniTwit application. Prometheus and Grafana allow us to perform white-box monitoring (Pfeiffer, 2023), focusing on what's inside the application. With Prometheus, we are able to check the performance of our application's responsiveness regarding HTTP requests to identify potential bottlenecks, performance issues, and trends.
In addition to monitoring responsiveness, we also tracked other metrics, including CPU time usage, garbage collector run time, maximum file descriptors, and

[9] https://github.com/Group-O-Minitwit/minitwit-group-o/blob/main/remote_files/prometheus.yml

remote read queries. Prometheus collected and stored these metrics from our application, which can be accessed using Prometheus's queries in Grafana. This integration with Grafana allowed us to visualize and analyze the metrics, providing valuable insights into the performance of our application.

3.8 Logging

To implement logging in our system, we decided to go with part of the EFLK stack. Namely, ElasticSearch, filebeat and Kibana and replacing Logstash with .NET's own logging library called Serilog. With this setup we are logging all API activity so that we through Kibana can visualize all the data we get.

We decided to drop Logstash as it is has high resource consumption which is not desirable for our system where we can risk getting a lot of API calls due to a large amount of users.

3.9 Security assessment

The output from the Metasploit Framework's Web Application Scanning module (`wmap_vulns`) after scanning a target IP address (157.230.79.99) for vulnerabilities in directories and files seems like there were no vulnerabilities found. The scanner tested various modules to check for SSL, web server, and file/directory vulnerabilities, but there were no positive results indicating that the target site was vulnerable to any of them.
Some of the tests did not receive a response from the target site, which could indicate that the site is blocking requests or the scanner is being detected and blocked. However, without further information or analysis, it's not possible to determine the exact reason why the site did not respond.
The output shows that the scanner found three directories (/Login/, /login/, and /register/) and two files (/register and /login) on the target. For each directory or file found, the scanner performed an HTTP GET request and received a response code.
The response codes are 405 (Method Not Allowed) for the directories, indicating that the HTTP method used (GET) is not supported for those directories. For the files, the response codes are 404 (Not Found), indicating that the files were found but could not be accessed.
Overall, the output suggests that the target has directories and files that may be of interest to an attacker, but the access to them is restricted.

3.10 Applied strategy for scaling and load balancing.

For scaling the application we decided to go with Docker swarm. Docker swarm is a framework that provides both scalability and reliability via replication and load balancing.

3.10.1 Implementing scaling

When using the application towards the end of the simulator we realized the performance was getting slower. This made sense since we hadn't really upgraded our remote server or performance within our application so far. This is of course not ideal. We realized too late, that we actually weren't that far from utilizing horizontal scaling. We were already using docker-compose, and with some small changes to the compose file, it would be possible to use the "`docker stack deploy ...`" to deploy our system in a docker swarm environment. Though because we realized this too late, we, unfortunately, did not get the pipeline to reflect these changes. Our plan was to increase the amount of replicas running the API, to hopefully allow more requests to process simultaneously.

3.10.2 Implementing availability

We, unfortunately, also did not get to implement availability scaling during the period of this project. To increase availability in our system we would need to ensure that our services are constantly running, which would be possible by replication. This is not only replicating the processes but also increasing the number of servers used. This could be done by having a *hot and standby*[10] setup. The reason it would be necessary to have multiple servers would be to ensure that even if the power got shut off to the main server, it could be detected by a heartbeat process[11], and prompt the system to switch to the standby server.

3.11 AI assistance

AI assistance was not used in the context of refactoring the project. However AI assistance was used to a minimal level, in the form of a lookup tool. It was used to resolve mistakes made in the codebase, when actually developing, as a replacement to conventional searching on the web. No code has been directly copied from the AI tool into the codebase. It was solely used as a guideline to give an idea of how to resolve common errors prompted by the compiler.

Tools used includes Github Copilot[12], and GPT-4[13].

4 Lessons Learned Perspective

Throughout the development of our project, we encountered various challenges and gained valuable insights.

[10] Lungu 2023

[11] This is a process running on separate setup, pining the system in intervals to check it's availability.

[12] https://github.com/features/copilot

[13] https://openai.com/product/gpt-4

4.1 Evolution and refactoring

When we started out the project, we chose to use ASP.NET and PostgreSQL, which is what we stuck with using. But we connected it in such a way that we wouldn't be able to migrate the database (i.e. change datatypes, alter tables, and such). We could fix this by recreating the database, but since the system was live, this was not an option for us. We ended up fixing it by carefully creating a backup, and with multiple steps, tricking the system into using the old database, but with the database being migratable. Due to this, the system was only down for about half an hour.

When developing our frontend, we ran into some issues, where our frontend would not send back the data to the API. This happened even though the information it got, which was formatted to json, seemed like it should work. We tried posting the same json to the API through postman wihtout any errors. We found out it was caused by the browser, which uses CORS (Cross-origin Resource Sharing[14]) a a security measure, to protect the user from potential attacks. We eventually used a proxy to bypass the CORS error. This is not an optimal solution, but we couldn't find any other solution, at the time.

4.2 Operation

While we were always able to operate the simulator through the API, it took us quite a while to get a frontend running due the problems mentioned in section 4.1 above.

4.3 Maintenance

The intricacies of continuous integration and deployment demand careful consideration from start to finish. This was made evident when we first embarked upon configuring our own CI/CD pipeline – the initial setup presented several difficulties. We soon realized that grasping how every component interconnects with others within the pipeline was key, alongside ensuring precise configuration throughout. We encountered some setbacks along the journey but found success by persevering through experimentation until resolving any corresponding troubles related to remote server deployment.

5 Conclusion

Ensuring the security and integrity of our application was a top priority throughout the development process. We recognized the importance of implementing monitoring systems and conducting thorough penetration testing to identify and mitigate potential vulnerabilities.

[14] Mozilla

By using GitHubs tools to plan and organize our work we were able to closely monitor issue tracking and use an appropriate branching strategy to manage tasks holistically.

Furthermore, the intricacies of continuous integration and deployment required meticulous attention and precise configuration. We recognized the importance of understanding how each component is interconnected within the pipeline. By persevering and conducting thorough experimentation, we overcame challenges related to remote server deployment and established a reliable and efficient CI/CD pipeline.

In summary, our project has provided us with invaluable lessons in monitoring, security, scalability, and continuous integration and deployment. These lessons will undoubtedly shape our future projects and contribute to our professional growth as software developers. The project stands as a testament to our dedication and commitment to delivering high-quality solutions to real-world problems while prioritizing effective communication, meticulous planning, scalability, and security measures.

References

[1] Prometheus Author. Prometheus. https://prometheus.io/, visited: 24/05-2023.

[2] Rahul Awati and Ivy Wigmore. monolithic architecture. 2022. https://www.techtarget.com/whatis/definition/monolithic-architecture, visited: 23/5-2023.

[3] F5. Nginx. https://www.nginx.com/, visited: 24/05-2023.

[4] GitHub. Codeql. https://codeql.github.com/, visited: 24/05-2023.

[5] GitHub. Github actions. https://github.com/features/actions, visited: 24/05-2023.

[6] GitHub. Github flow. 2023. https://docs.github.com/en/get-started/quickstart/github-flow visited: 23/05-2023.

[7] The PostgreSQL Global Development Group. Postgresql. , visited: 24/05-2023.

[8] Docker Inc. Docker. https://www.docker.com/, visited: 24/05-2023.

[9] Grafana Labs. Grafana. https://grafana.com/, visited: 24/05-2023.

[10] DigitalOcean LLC. Digitalocean. https://www.digitalocean.com/, visited: 24/05-2023.

[11] Mircea Lungu. Session 10 slides. 2023. https://github.com/itu-devops/lecture_notes/blob/master/sessions/session_10/Slides.md, visited: 22/05-2023.

[12] Microsoft. C#. https://learn.microsoft.com/en-us/dotnet/csharp/, visited: 24/05-2023.

[13] Microsoft. Infer#. https://github.com/microsoft/infersharp, visited: 24/05-2023.

[14] Mozilla. Cors errors. 2023. https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS/Errors, visited: 24/05-2023.

[15] The npgsql team. Npgsql. https://github.com/npgsql/npgsql, visited: 24/05-2023.

[16] Stack Overflow. Developer survey. 2022. https://survey.stackoverflow.co/2022/#most-popular-technologies-language-prof, visited: 23/05-2023.

[17] Helge Pfeiffer. Session 06 slides. 2023. https://github.com/itu-devops/lecture_notes/blob/master/sessions/session_06/Slides.md, visited: 22/05-2023.

[18] SonarSource. Sonarcloud. https://www.sonarsource.com/products/sonarcloud/, visited: 24/05-2023.

[19] Meta Open Source. React. https://react.dev/, visited: 24/05-2023.

[20] The ElasticSearch Team. Elasticsearch. https://www.elastic.co/, visited: 24/05-2023.

[21] The ElasticSearch Team. Filebeat. https://www.elastic.co/beats/filebeat, visited: 24/05-2023.

[22] The ElasticSearch Team. Kibana. https://www.elastic.co/kibana/, visited: 24/05-2023.

[23] The SeriLog team. Serilog. https://serilog.net/, visited: 24/05-2023.

[24] DJ Wardynski. Is asp.net still used today? asp then and now. 2022. https://www.brainspire.com/blog/is-asp-dot-net-still-used-today-asp-then-and-now-brainspire, visited: 23/05-2023.