IT UNIVERSITY OF COPENHAGEN

DevOps, Software Evolution and Software Maintenance (Spring 2023)

Course Code: BSDSESM1KU

May 23, 2023

Group k members:

Gustav Valdemar Metnik-Beck gume@itu.dk
Nikolaj Worsøe Larsen niwl@itu.dk
Simon Johann Skødt sijs@itu.dk
Victor Møller Wauvert Brorson vibr@itu.dk

Contents

1 Introduction

Throughout the course, *DevOps, Software Evolution and Software Maintenance*, the legacy *Python*-based *Flask* Twitter application *MiniTwit* was migrated to a modern *.NET-based* application. Following the initial migration, the task at hand was to handle simulated users using DevOps principles and practices in day-to-day development [2]. The system was designed to support the fundamental functionalities of a Twitter application, such as user registration, login, tweet creation, follow requests, and viewing others' tweets. The system was operated on a weekly basis, with new methods and strategies integrated into the system, such as continuous integration, delivery and deployment, logging, monitoring, and scalability, thereby enabling a comprehensive understanding of the importance of DevOps in managing software systems.

2 System Perspective

2.1 System Design

The present *MiniTwit* system represents a refactored and optimized version of the legacy *Python* codebase that was initially provided. The current system is organized into three distinct components[1], including i) a *C# ASP.NET* project that functions as the *backend component*, ii) a *React-based frontend component*, and iii) a *NoSql MongoDB database*. These particular technologies were selected based on the developers existing familiarity with these tools, which has been achieved from prior courses and practical experience.

We decided to use *Ubuntu* as common *OS* and created all services as *Docker Images*, to avoid having separate local build scripts.

2.1.1 System Architecture

As shown in figure 2.1, *MiniTwit* consists of several subsystems. However, the two main components of the system are the *MiniTwit* frontend and backend.

[1] To ensure consistency, programming languages, frameworks, and libraries will be presented in *italics*.

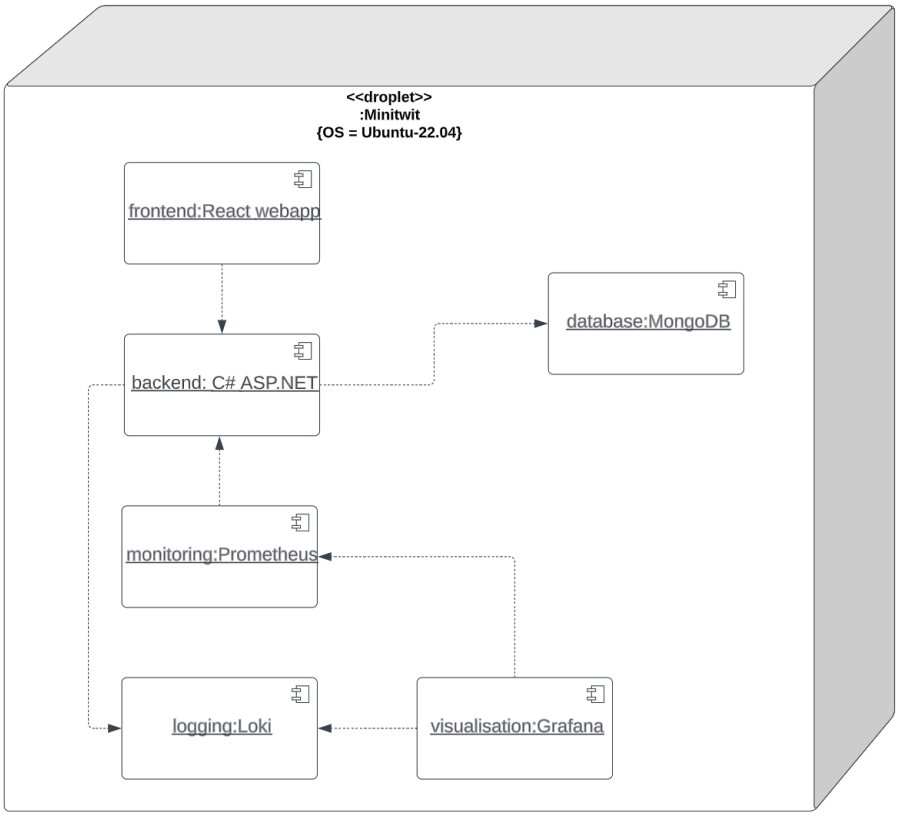

Figure 2.1: *Minitwit* deployment diagram.

MiniTwit Backend

The design of the *MiniTwit* backend is inspired by the *Onion Architecture* (see figure 2.2) which is based on the *Inversion of Control* principle. The architecture is comprised of multiple concentric layers that interface with each other toward the core. The flow of dependency goes inwards s.t. a layer only depends on the layers that come before it. Keeping the layers separate, makes them easier to test and maintain, as the coupling between them is kept low [3].

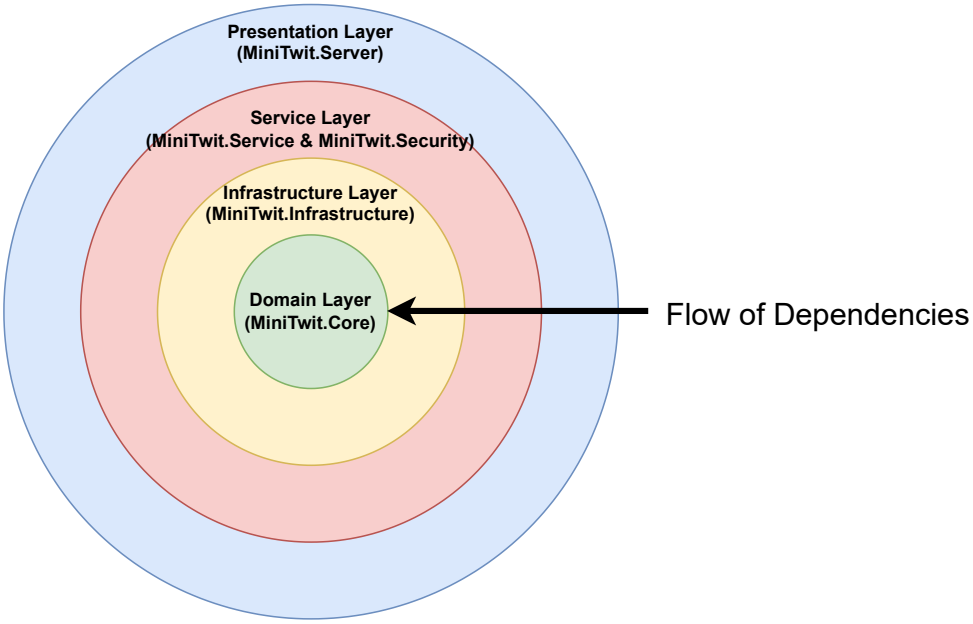

Figure 2.2: The Onion Architecture-based architecture of the *MiniTwit* backend.

We chose to split the backend into four layers and five separate *C# projects* (see figure 2.2):

1. **Domain Layer (MiniTwit.Core):** Contains the domain models, data transfer objects (*DTOs*), and interfaces for the business logic implemented in the infrastructure layer.
2. **Infrastructure Layer (MiniTwit.Infrastructure):** Contains the business logic and database abstraction layer.
3. **Service Layer (MiniTwit.Service & MiniTwit.Security):** Is split in two projects; Service and Security:
   - **Service:** Contains abstractions to the infrastructure layer in the form of services.
   - **Security:** Contains security-related abstractions e.g., in the form of hashing algorithms.
4. **Presentation Layer (MiniTwit.Server):** Contains the main API of the backend in the form of controllers and authentication schemes.

As can be seen in the package diagram in figure 2.3, the various layers only depend on the layers that come before them.

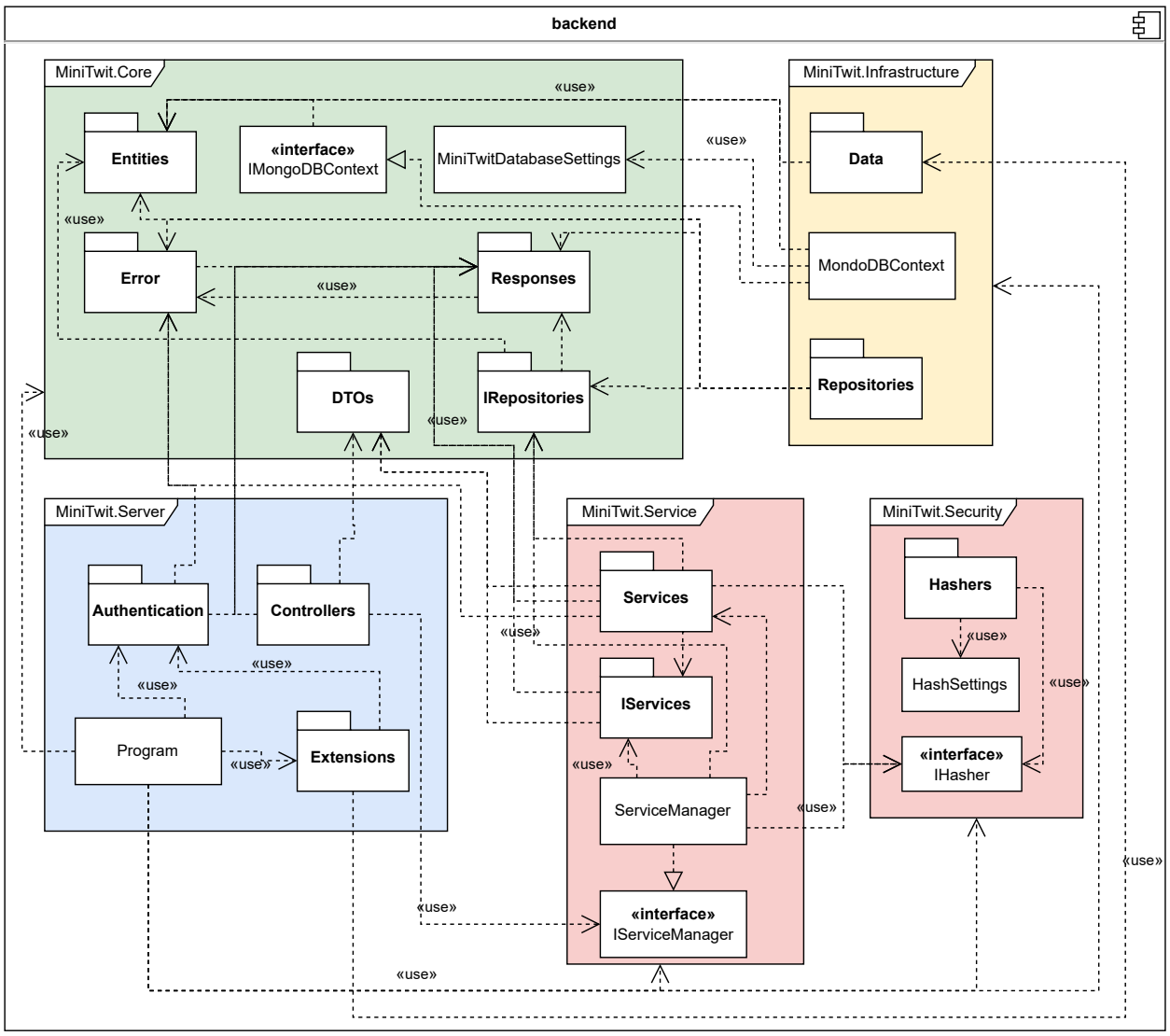

Figure 2.3: Package diagram showing the contents of the various components of the *MiniTwit* backend.

MiniTwit Frontend

The *MiniTwit* frontend, a *React* web app, was designed by splitting the functionalities of the web app into several packages to ensure maintainability. The packages include:

- **models:** Contains domain-specific interfaces according to the backend API.
- **services:** Contains abstractions per domain-specific model used to communicate with the backend (*CRUD* operations).

- **components:** Contains web components used in the web pages of *MiniTwit*.
- **pages:** Contains all pages displayed on the *MiniTwit* website.
- **state:** Contains functions to update and read entries from the session storage in the client's web browser.

The various packages of the frontend and their dependencies are shown in the expanded package diagram in figure 2.4.

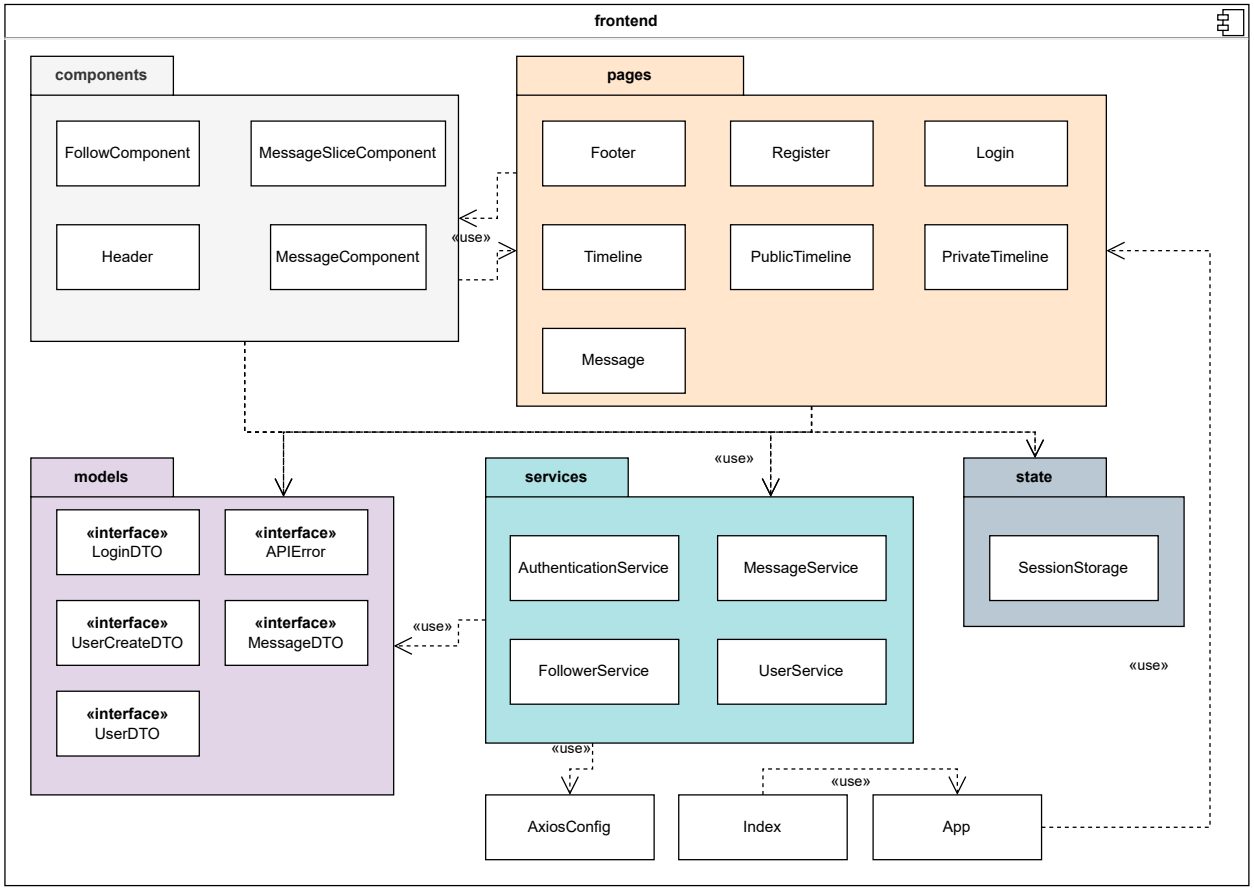

Figure 2.4: The architecture of the *MiniTwit* frontend.

2.2 System Dependencies

By using fossa.com to analyze our project, we can conclude that the project contains 44 direct dependencies and 949 transitive dependencies. The dependencies are divided into *NPM* dependencies for the frontend, *C#* application dependencies for the backend, and CI/CD dependencies in the form of GitHub Actions. The *NPM* dependencies are primarily *React* libraries and the backend

dependencies are primarily *Mongo* and *Microsoft* dependencies. The CI/CD dependencies in the form of *GitHub Actions* are a mix of many dependencies. A concern is the direct dependencies, as they are the ones directly used in our source code, and might be outdated or vulnerable.

The above-mentioned dependencies can be seen listed per subcomponent of *MiniTwit* in Appendix A.1, A.2 and A.3.

2.3 Current State

To obtain a comprehensive understanding of the current state of our system, this section outlines the status of three static analysis tools: *ESLint*, *CodeQL*, and *Snyk*. Additionally, we provide our quality assessments from *SonarCloud* and *Code Climate*. Finally, the system's quality is evaluated through unit-, integration-, and UI tests. These three quality gates test the software quality in different ways and serve as an important benchmark for evaluating the system's overall quality.

Both the Status Analysis Tools and the Quality Assessments form part of a *product view*, where the focus is on internal software metrics. In contrast, UI tests are highly relevant to the *user view* since they assess the frontend behavior, which is the only access level for the client/user [1, pp. 13-15].

2.3.1 Static Analysis Tools

ESList (TypeScript), *CodeQL* (C#), and *Snyk* (security) serve as benchmarks of quality for the various languages and tools in our repository. Changes to the code that fail the quality gate metrics on certain parameters must be addressed before they can proceed in the *CI/CD* chain. However, for code that passes the quality gates and only exhibits minor warnings, it may move forward. For a more detailed explanation of the various *YAML* scripts, refer to section 3.3.

2.3.2 Quality Assessments

SonarCloud and *Code Climate* (see figure 2.5 and 2.6) automatically scan the entire code base and grade the quality. Furthermore, it highlights what can be improved to obtain higher marks.

Figure 2.5: *SonarCloud* issue detector for pull request.

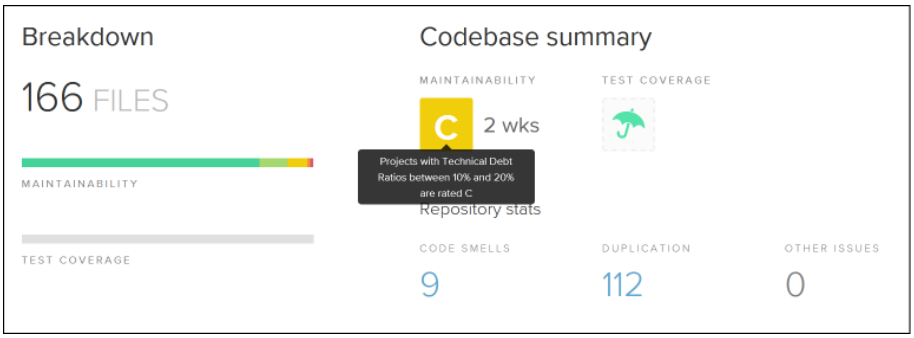

Figure 2.6: *Code Climate* code base summary.

2.3.3 Tests

The back- and front-end testing are crucial quality checkpoints, as a failure in either would prevent the deployment of the system (see figure 2.7). Unit-, integration-, and UI tests have been implemented, but end-to-end tests have not been finalized. Testing of the backend's Onion Architecture, as described in section 2.1.1, was done using 39 unit tests. Additionally, 17 integration tests were employed to test the behavior of the combined layers.

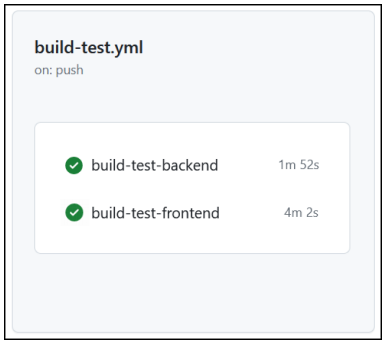

Figure 2.7: Chain of back- and frontend tests

2.4 Licence Compatability

MiniTwit uses an *MIT* license. *MIT* is one of the most permissive licenses and therefore also compatible with other open source licenses [4]. All licenses found in the direct dependencies can be seen in appendix A.4.

3 Process Perspective

3.1 Developer Interaction

During the development process, the majority of interaction among the developers has occurred on-site. The group has met one to two times per week to collaboratively develop new features, while smaller ad hoc tasks have been solved individually. Communication in remote contexts has been facilitated by the use of *Discord* and *Messenger*. All documentation such as a guide for local environment setup, release notes, money spent, and code guidelines have been manually written and shared on *Notion*.

3.2 Team Organisation

The organization of the team and distribution of tasks have represented an important aspect of the project's success, considering that each developer attends different courses and jobs which can disrupt communication. To address this challenge, the group has used *GitHub Issues*, which allows for the creation of a *Kanban board* displaying pending tasks, their respective type, and current progress. By doing so, the group has been able to maintain transparency of work progress without relying on constant communication.

3.3 CI/CD

To facilitate the Continues Integration of new code by all developers, several workflows/pipelines have been implemented to ensure that each pull request (*PR*) is subjected to checks and analyses, including building, testing, and code scanning to mitigate the risk of the service breaking. Once these checks are successfully completed, the *PR* is deemed safe for merging into the main branch, which initiates a deployment pipeline. The following subsections give a brief explanation of each workflow.

3.3.1 *build-test.yml*

This workflow performs the build and testing of the stack in the service, including the backend and frontend. The pipeline is triggered on push events to any branch and includes two jobs:

build-test-backend: Runs on an *Ubuntu 20.04* machine and includes the following steps:

1. Checkout the repository
2. Set up *.NET* version 7.0.x
3. Restore dependencies
4. Build the *MiniTwit* backend
5. Run backend tests

build-test-frontend: Runs on an *Ubuntu 22.04* machine and includes the following steps:

1. Checkout the repository
2. Install *Node.js* version 18 and dependencies for caching
3. Install dependencies for the *MiniTwit* frontend
4. Build the *MiniTwit* frontend
5. Run frontend tests using *Playwright*, in a docker container started by *docker-compose -f docker-compose.ui.yml up -d*
6. Stop the UI service by shutting down the Docker containers created with docker-compose

3.3.2 *eslint.yml*

A pipeline that integrates *eslint* for checking the frontend code. The workflow automatically detects and reports issues in the code by uploading a *.sarif* file to GitHub. It provides guidance on how to fix the issues reported, such as unused variables, unused imports, and type-related errors 3.1.

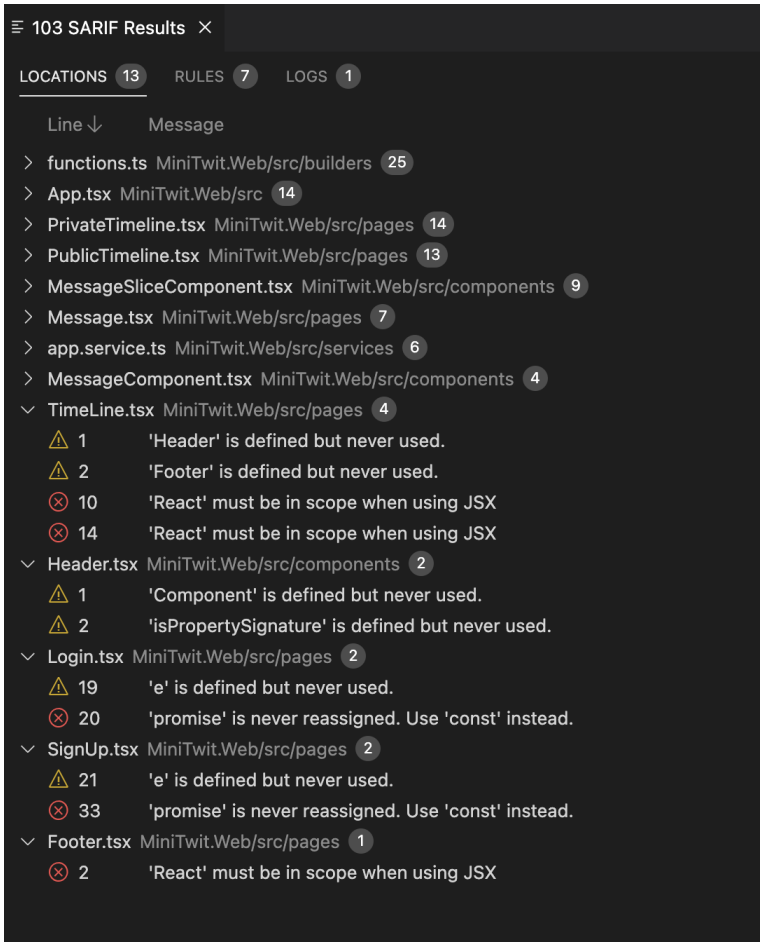

Figure 3.1: Eslint scan - Report.

3.3.3 *codeql.yml*

A pipeline that integrates *CodeQL* (C# analysis), which scans the backend code for *C#* specific vulnerabilities and bugs. It also provides guidance on how to fix them.

3.3.4 *snyk-security.yml*

A pipeline that integrates with *Snyk*, which provides continuous security monitoring 3.2. The pipeline scans the project's dependencies to identify any known vulnerabilities and automatically sets up a *PR* that updates outdated packages or packages containing known vulnerabilities.

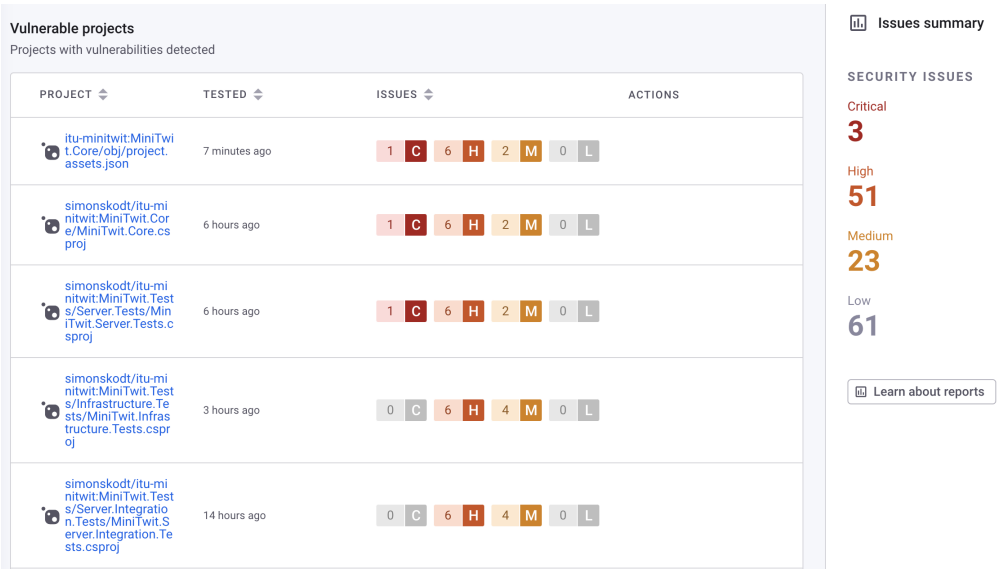

Figure 3.2: *Snyk* Security — report.

3.3.5 *continuous-deployment.yml*

All pipelines highlighted so far, are triggered either by pushing to a feature branch or creating a *PR* to the main branch. The combination of these pipelines constitutes the project's *deployment gate* which is a collection of predefined checks and signals that must pass before a deployment may be triggered [5]. If the *PR* passes the gate, it can be deployed. This is done by merging the *PR*, which will trigger the continuous-deployment pipeline.

The continuous-deployment pipeline is separated into two jobs, namely build and deploy. The *build* job is responsible for building and pushing the Docker images for the *MiniTwit* backend and frontend, respectively. The job uses the Docker Build tool to build the images and then pushes them to Docker Hub.

The *deploy* job is responsible for deploying the *MiniTwit* application and monitoring/logging components to the live server. This job is dependent on the *build* job to ensure that the images are built and available for deployment. The job first uses the *rsync* command to copy the Docker Compose files and monitoring/logging configuration files to the live server. It then uses the *SSH Action* to log in to the live server and deploy the *MiniTwit* application and monitoring/logging components by executing the docker-compose files: *docker-compose.prod.yml* and *docker-compose.monitoring.yml*.

3.4 Repository Organisation

The group has employed a mono-repository approach, in which all components are stored within a single GitHub repository. While the option to maintain separate repositories for the frontend and backend existed, it was decided that, given the size of this project, a mono-repository was better with regards to the standardization of branching strategies, review of pull requests, and deployment strategy, i.e. we can deploy the whole service from a single *PR*.

3.5 Branching Strategy

We have used a trunk-based branching strategy using two types of branches.

- The main branch/trunk branch (long-lived)
- Several feature branches (short-lived)

The main branch always contains the newest working instance of *MiniTwit*. The feature branches are short-lived branches containing new features for *MiniTwit*. As soon as a feature is completed, the feature branch is merged with the main branch using pull requests. Once the merge is accomplished, the feature branch is deleted. Due to branch protection rules pull request can only be accepted onto the main branch after it has undergone review, approval, and has passed all pipeline checks. To maintain a clean history of the version control system, it is recommended that each branch correspond to an issue on GitHub, and that commit messages have the following format:

```
<commit type>([scope]): <message>
```

3.6 Monotoring

When it comes to monitoring *MiniTwit*, we only monitor the backend. For this, we use *Prometheus* by importing its NuGet package in the *Presentation Layer*. Out of the box *Prometheus* provides a */metrics* endpoint to scrape for the following metrics:

- Whether *Prometheus* and the backend is up
- Requests per controller per endpoint
- Total number of requests
- Average request response time
- CPU usage

- Memory usage
- Number of requests with successful and unsuccessful response code

Prometheus is scraped by *Grafana* and the metrics are visualized in our monitoring dashboard, which can be seen in figure 3.3. The dashboard visualizes the information about the average response time, number of requests, etc. which has been useful to detect any abnormalities in the project. Especially the average response time, which in the beginning was way too high for the system to work properly when multiple incoming requests were handled.

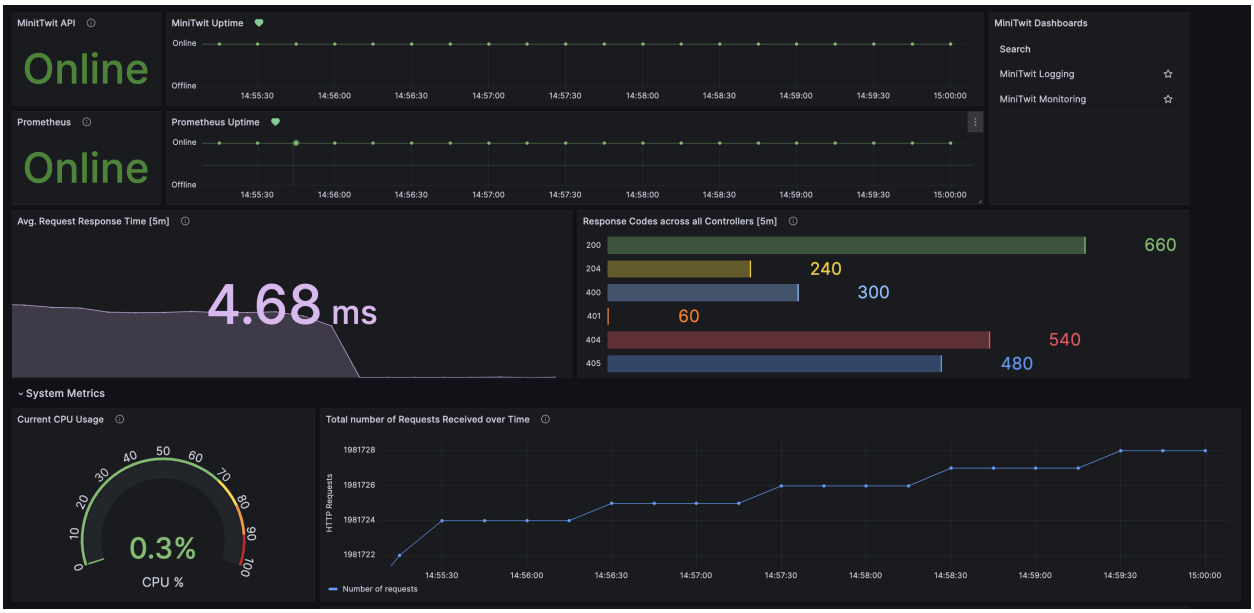

Figure 3.3: *Grafana* monitoring dashboard for monitoring the *MiniTwit* backend.

3.7 Logging

For logging the *MiniTwit* backend, we used *Loki*. When logging, *Loki* automatically sends the logs to a running *Loki* server instance which is directly queried by *Grafana* in a logging dashboard (see figure 3.4). We only log within the *Presentation Layer* in our controllers. The log level used when logging depends on the response state from the service layer, specifically, we use:

- **Error:** When the *Service Layer* returns errors e.g. when usernames or ids are not found.
- **Warning:** When an unexpected status is returned from the *Service Layer*.
- **Information:** When a request is successfully executed, it describes what happened.

- **Debug:** When we want to debug information such as the number of Tweets fetched from the backend etc.

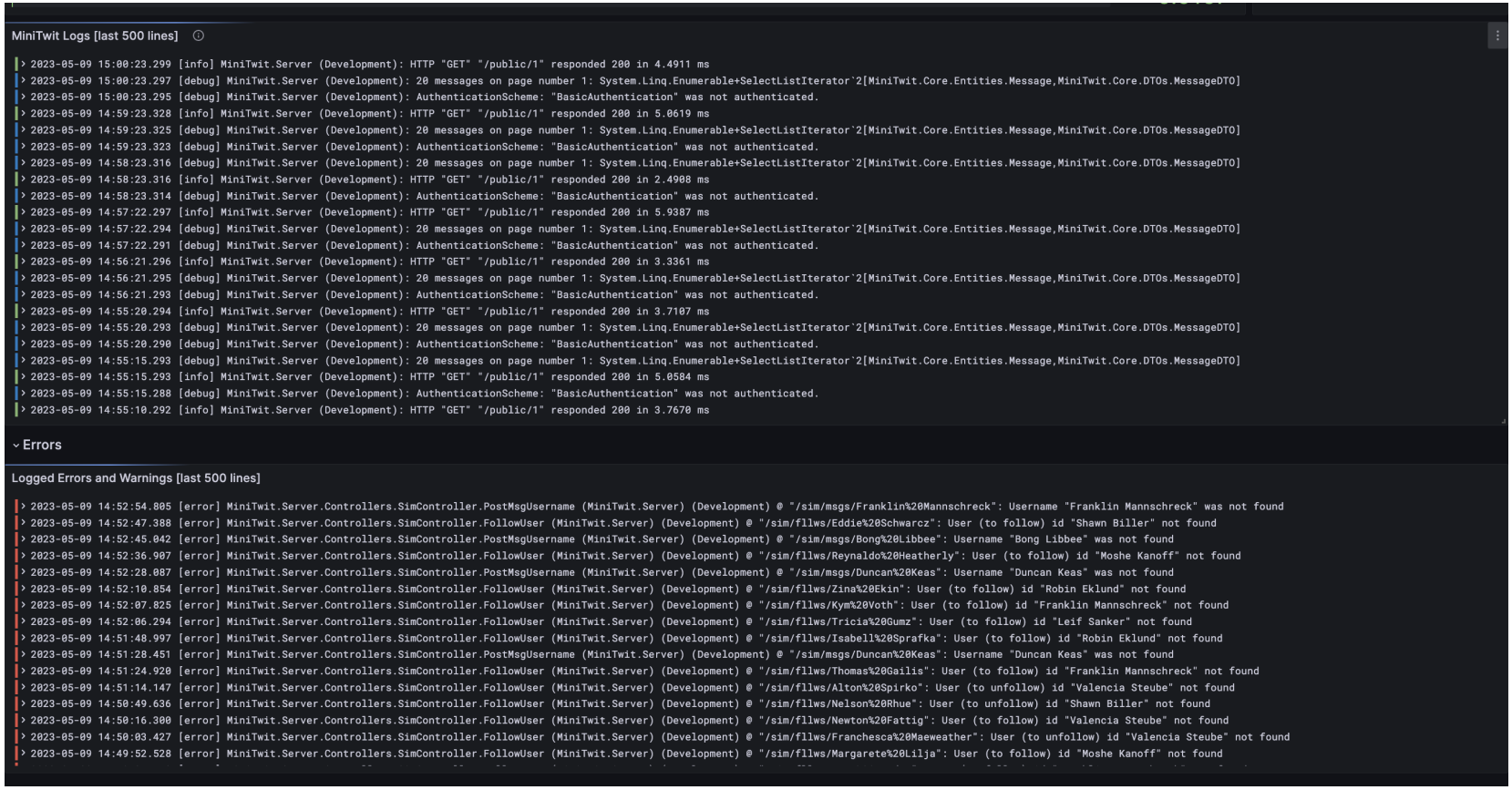

Figure 3.4: Logging from our *Grafana* logging dashboard.

3.8 Security

As seen in the following security assessment: *secuirty assesment*, we found that our service lacked some security measurements, particularly the absence of a TLS configuration. To address this issue, the team attempted to implement TLS using *NGINX* and *Certbot*, aiming to secure the obtained domain "minitwit.live". However, we didn't manage to obtain a certificate, as we experienced difficulties, likely caused by the server's firewall blocking the request to *Certbot*.

3.9 Scaling and Load Balancing

In terms of scaling, we created a Docker Swarm for horizontal scaling, following the instructions provided during the lecture. The swarm was successfully set up, and it functioned as intended in theory. Although we had three instances of our service running within the swarm, we were hesitant to replace our production droplet with the swarm due to our lack of confidence in operating and maintaining it effectively, as the communication and mapping of frontends to corresponding backends were unclear.

The Swarm was created and setup using the following shell script swarm.sh, which served as our implementation of Infrastructure as Code. Scripts were not employed for creating virtual machines (VMs) beyond this point, as we found the *DigitalOcean* user interface to be very intuitive. Nevertheless, we acknowledge the importance of employing tools such as *Vagrant* or *Terraform* for VM creation, as they would simplify a handover of the project.

3.10 Use of Artificial Intelligence

This project utilized *ChatGPT* to solve specific code issues. Additionally, AI has been used to grammatically correct sentences. However, the AI's suggestions were not blindly copied, as the essence of the sentences could be lost in the process.

4 Lessons Learned Perspective

4.1 Evolution and Refactoring

In the context of a legacy Python *Flask* application, the aim was to create abstractions that would minimize technical debt for potential DevOps engineers who would take over our application. After an initial major rewrite to a C# *ASP.NET* project based on the Onion Architecture, which allows for easier scaling of the application, we realized that this was a time-consuming process and in real development, would have been an expensive migration. For future projects, we propose the use of a small *C#* minimum viable product (MVP) using a micro-framework like *Nancy* [7], which is equivalent to *Flask* for Python. This approach would help in quickly testing the application and identifying alterations in the application's functionality early on.

4.2 Operation

Operating a live cloud-based *MiniTwit* service, with several other focuses, such as other courses and jobs, has not been an easy task. However, the implementation of *Loki* and *Prometheus* with *Grafana* as a visualization tool, made the process much easier. Throughout the course, we have been seeing many errors originating from the simulator, these errors, indicated by the logging board in *Grafana*, are particularly in the form of 404 errors associated with the simulator endpoints: */Sim/msgs/*{*username*} and */Sim/fllws/*{*username*}. This situation arose due to our initial approach to the simulator part of the project. Unfortunately, we never successfully implemented the simulator tests, resulting in an incomplete configuration of our endpoints when the simulator began making API calls.

At this point, we had not yet implemented logging, which prevented us from quickly identifying these errors before the first graph of the simulator status was presented to us in class (approximately one week into the simulation). As soon as we became aware of the issue, we immediately fixed the misconfigured API endpoints. However, due to the faulty implementation at the project's start, many users had already failed to register. Consequently, these users continued to submit follow and tweet requests, thereby contributing to the errors observed on our *Grafana* logging board.

Despite this, our project's performance in the simulator status and API error graphs remained within average. However, as time progressed, we eventually emerged at the top of the error board.

Concurrently, we encountered a significant number of registration errors and service breakdowns, prompting us to investigate the problem. Our findings revealed that the service would crash when too many users register simultaneously. This came to our attention as we had implemented email notifications in *Grafana* which fired when the server crashed.

Upon closer examination, we discovered that our droplet ran out of memory during the simultaneous hashing of many usernames and passwords. Consequently, when the simulator rapidly registered new users, the service experienced repeated breakdowns. This led to an increased number of users not being registered while still posting tweets and following other users.

4.3 Maintenance

To maintain the problem mentioned above, we lowered the memory consumption of our hashing allocating from *131MB* to *51MB* per hash. During the operation of our live system, we found another problem while looking at the monitoring in the *Grafana* dashboard. We were experiencing significant delays in response times for requests made to the *HttpGET* API endpoint */Timeline*. Consequently, we initiated an investigation into the issue and discovered that the database contained over three million tweets (3,158,280). This endpoint triggered a query to the database that required sorting all the messages to display the most recent ones, i.e. for every request the database had to sort over three million tweets, which is very time-consuming.

To maintain a good user experience within the system, we immediately decided to create an index on the message collection based on their creation dates[1]. This optimization resulted in an improvement in response time, reducing it from an average of 2.5 seconds to only 200 milliseconds, observed through our monitoring system.

4.4 DevOps Style

Our main success for this project, as opposed to prior projects, lies in our adherence to two of the principles of The Agile Manifesto:

> *"1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software. [...]*
> *3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale."* [6]

[1] Link to *PR* for index creation

By releasing new features and fixes every week, we have continuously delivered valuable software to our live server, something we did not do in prior projects. This approach complements our strategy of using short-lived branches, resulting in a faster flow. In best cases, the time taken to complete and deploy a certain task was less than two hours.

4.4.1 The Three Ways

Reflecting on our project process in relation to the Three Ways [5], we can identify further improvements to enhance our DevOps implementation in future projects.

The First Way: Flow/Systems Thinking. By establishing a robust CI/CD chain, we were able to quickly move left-to-right work (Dev-Ops) with a low number of defects.

The Second Way: Amplify Feedback Loops. We incorporated a right-to-left feedback loop by monitoring and logging live system behavior on relevant metrics and critical areas of our application. By setting up a real-time email alert when our service went down, team members were quickly informed about this issue.

The Third Way: Culture of Continual Experimentation and Learning. We did not actively spend much time fostering a culture of learning or experimentation. However, since we have worked together on previous projects, we have developed trust in each other's work, allowing us to take risks. Our face-to-face communication and collaboration have enabled us to share knowledge, avoiding silos [6]. We have continuously encouraged each other to make improvements to the codebase. To achieve the Third Way, we could potentially foster even more innovation and freedom to explore new ideas.

References

[1] Barbara Kitchenham. "Software Quality: The Elusive Target". In: *National Computing Centre, Systems/Software, Inc.* (1996), pp. 12–21.

[2] Gene Kim et al. *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. Portland, OR: IT Revolution Press, 2016. Chap. 1.

[3] Tapas Pal. *Understanding Onion Architecture*. https://www.codeguru.com/csharp/understanding-onion-architecture/. Accessed on May 16, 2023. Feb. 2018.

[4] David A. Wheeler. *FLOSS License Slide*. https://dwheeler.com/essays/floss-license-slide.html. Accessed on May 9, 2023. 2021.

[5] Microsoft. "Deployment gates". In: (2022).

[6] Kent Beck et al. *Agile Manifesto*. https://agilemanifesto.org/. Accessed on May 16, 2023.

[7] *NancyFX*. https://nancyfx.org/. Accessed on May 16, 2023.

Appendices

A Dependencies

A.1 *NPM* Dependencies

- @emotion/react@11.10.6
- @emotion/styled@11.10.6
- @microsoft/eslint-formatter-sarif@2.1.7
- @mui/material@5.11.13
- @playwright/test@1.31.2
- @testing-library/jest-dom@5.16.5
- @testing-library/react@13.4.0
- @testing-library/user-event@13.5.0
- @types/jest@27.5.2
- @types/node@16.18.16
- @types/react-dom@18.0.11
- @types/react@18.0.28
- @typescript-eslint/eslint-plugin@5.56.0
- @typescript-eslint/parser@5.56.0
- axios@1.3.4
- eslint-plugin-react@7.32.2
- eslint@8.10.0
- mongodb@5.1.0
- playwright@1.31.2
- react-dom@18.2.0
- react-router-dom@6.9.0

- react-scripts@5.0.1
- react@18.2.0
- ts-md5@1.3.1
- typescript@4.9.5
- web-vitals@2.1.4

Dependency graph of npm package

Example of a dependency graph for the npm package @playwright/test made from: http://npm.anvaka.com/#/view/2d/

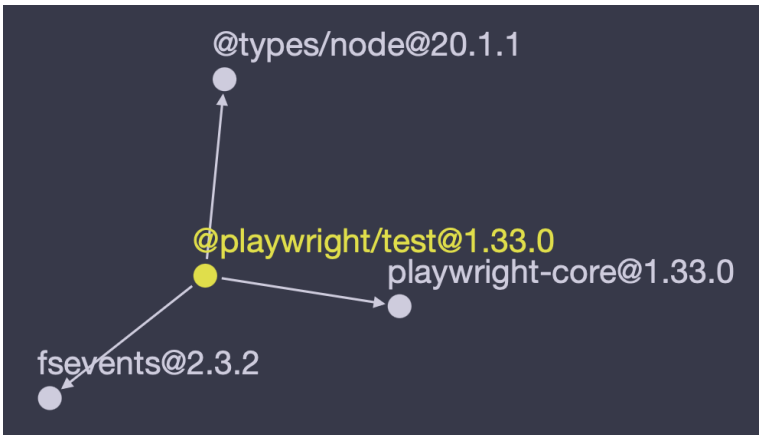

Figure A.1: Playwright dependency graph

A.2 *C#* Application Dependencies

MiniTwit.Core

- mapster
- Microsoft.AspNetCore.Mvc
- MongoDB.Bson
- MongoDB.Driver

MiniTwit.Infrastructure

- MongoDB.Driver

MiniTwit.Security

- Konscious.Security.Cryptography.Argon2
- Microsoft.Extensions.Options

MiniTwit.Server

- prometheus-net.AspNetCore
- Serilog.AspNetCore
- Serilog.Sinks.Grafana.Loki
- Swashbuckle.AspNetCore

MiniTwit.Tests/Infrastructure.Tests

- coverlet.collector
- FluentAssertions
- Microsoft.Extensions.Options
- Microsoft.NET.Test.Sdk
- Mongo2Go
- Moq
- xunit
- xunit.runner.visualstudio

MiniTwit.Tests/Server.Integration.Tests

- coverlet.collector
- FluentAssertions
- Microsoft.AspNetCore.Mvc.Testing
- Microsoft.NET.Test.Sdk
- Mongo2Go
- xunit
- xunit.runner.visualstudio

MiniTwit.Tests/Server.Tests

- coverlet.collector
- Microsoft.AspNetCore.Mvc
- Microsoft.NET.Test.Sdk
- Moq
- xunit
- xunit.runner.visualstudio

A.3 CI/CD

build-latex.yml

- ubuntu-22.04
- actions/checkout@v3
- xu-cheng/latex-action@v2
- actions-js/push@master

build-test.yml

- ubuntu-20.04
- actions/checkout@v3
- actions/setup-dotnet@v3
- ubuntu-22.04
- actions/setup-node@v3

codeql.yml

- ubuntu-latest
- actions/checkout@v3
- github/codeql-action/init@v2
- github/codeql-action/autobuild@v2
- github/codeql-action/analyze@v2

- actions/upload-artifact@v2

continuos-deployment.yml

- ubuntu-latest
- actions/checkout@v3
- docker/setup-buildx-action@v2
- docker/login-action@v2
- docker/build-push-action@v4
- burnett01/rsync-deployments@5.2.1
- fifsky/ssh-action@master

eslint.yml

- ubuntu-latest
- actions/checkout@v3
- github/codeql-action/upload-sarif@v2

snyk-security.yml

- ubuntu-latest
- actions/checkout@master
- actions/setup-dotnet@master
- snyk/actions/setup@master
- actions/setup-node@master
- github/codeql-action/upload-sarif@v2

sonarcloud.yml

- ubuntu-latest
- SonarSource/sonarcloud-github-action@de2e56b42aa84d0b1c5b622644ac17e505c9a049

A.4 Licenses of Direct Dependencies

- MIT License

- BSD 2-Clause "Simplified" License
- BSD 3-Clause "New" or "Revised" License
- Apache License 2.0
- ISC License
- Creative Commons Attribution 4.0
- Creative Commons Zero v1.0 Universal
- Do What The F*ck You Want To Public License
- Unicode License Agreement - Data Files and Software (2016)
- W3C Software Notice and Document License (2015-05-13)
- BSD-4-Clause (University of California-Specific)
- IETF Contribution Agreement
- University of Illinois/NCSA Open Source License
- UNICODE, INC. LICENSE AGREEMENT - DATA FILES AND SOFTWARE
- zlib License
- Microsoft .Net Library
- Proprietary License