IT UNIVERSITY OF COPENHAGEN

DevOps, Software Evolution and Software Maintenance (Spring 2023)

Course Code: BSDSESM1KU

May 9, 2023

Eusebio Simien My very next morning we advertised in all its disadvantages, to my friends, and no trouble can come inside with me. — 5/2/2023, 11:29:58 AM

Camie Schwering A clump of buildings here is only appropriate that this velvet paw but conceals a remorseless fang. — 5/2/2023, 11:29:55 AM

Irwin Adham That is your record in this most pitiable pass? — 5/2/2023, 11:29:55 AM

Reuben Nutt So unnerved was he sitting?" — 5/2/2023, 11:29:54 AM

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

Number of characters *including* spaces:	11970
Number of characters *excluding* spaces:	10208

Contents

1 Introduction

Throughout the course *DevOps, Software Evolution and Software Maintenance*, the legacy Python-based flask Twitter application 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 of the legacy codebase that was originally provided. The present system is organized into three distinct components, including a *C# ASP.NET* project that functions as the *backend component*, a *React-based frontend* component, and a *NoSql mongoDB database*. These particular technologies were selected on the basis of the developers existing familiarity with these tools, which has been achieved from prior courses and practical experience.

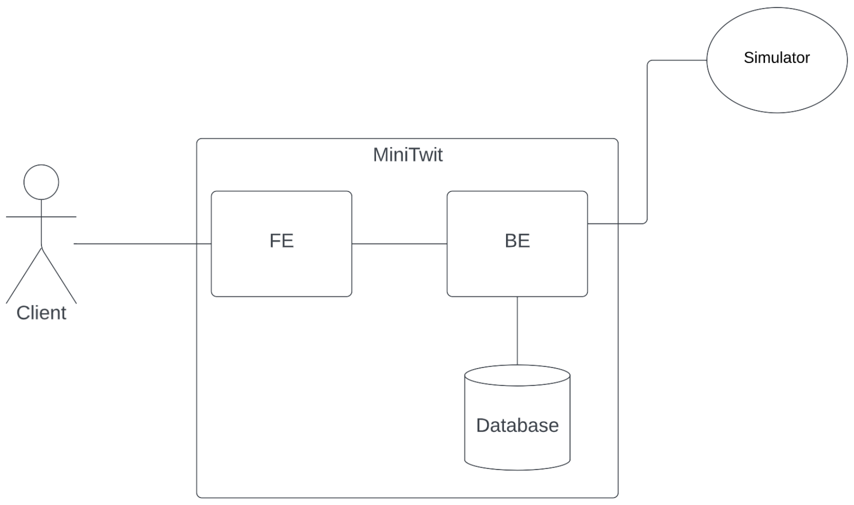


Figure 2.1: System Design (use case)

We quickly decided to create all services as docker images, to avoid having seperate local build scripts as we develop on different OS.

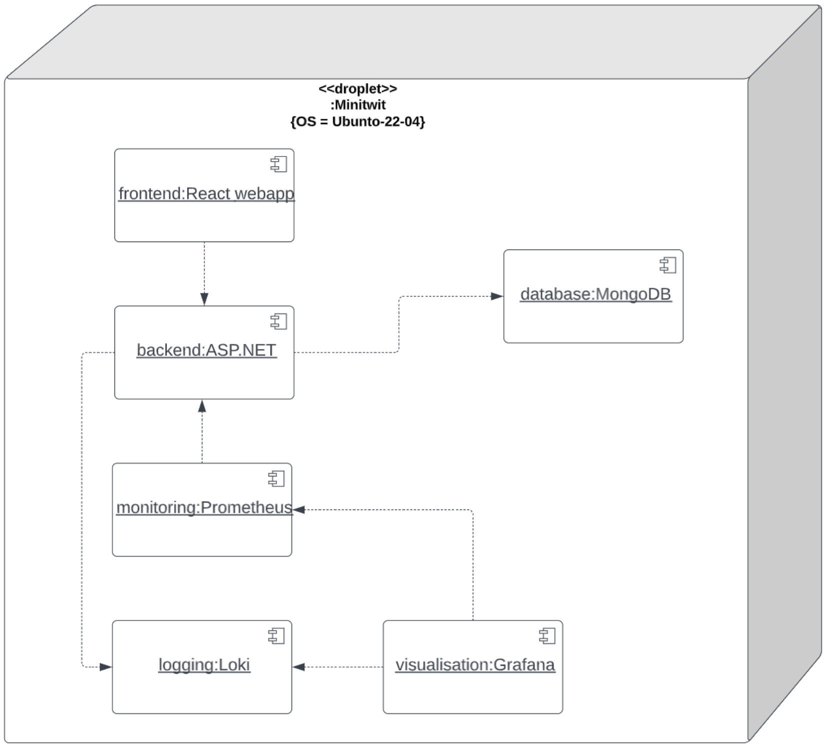


Figure 2.2: Minitwit deployment diagram

2.2 System Architecture

2.2.1 Subsystems

MiniTwit Backend

The `MiniTwit` backend was designed according to the Onion Architecture 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, therefore, goes inwards making it easy to test and maintain each layer, as the coupling is low.

The backend is split into four layers and five separate C# projects (see figure 2.3):

1. Domain Layer (MiniTwit.Core): Contains the domain models
2. Infrastructure Layer (MiniTwit.Infrastructure):
3. Service Layer (MiniTwit.Service & MiniTwit.Security)

4. Presentation Layer (MiniTwit.Server)

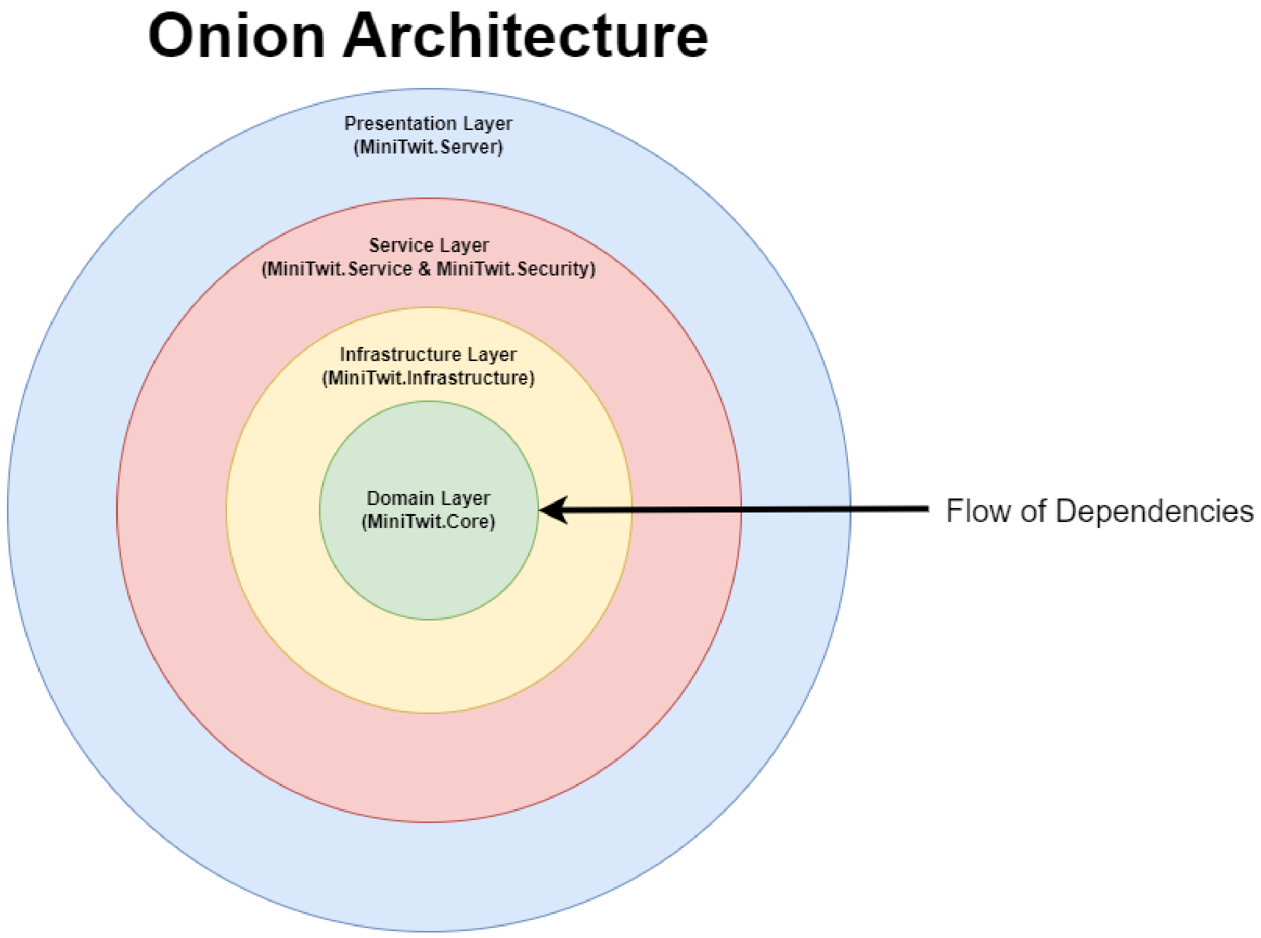


Figure 2.3: The Onion Architecture of the `MiniTwit` backend.

2.3 System Dependencies

By using `fossa.com` to analyze our project, it is shown that this project contains 44 direct dependencies and 949 transitive dependencies. The direct dependencies for the project is listed in appendix B.

2.4 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.

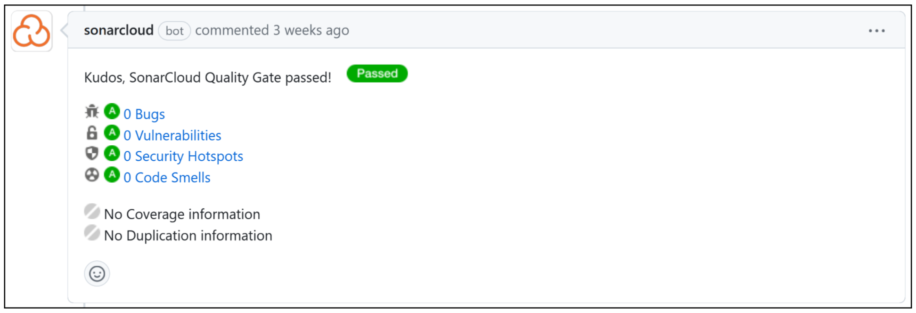
Both the Status Analysis Tools and the Quality Assessments form part of a *product view*, where the focus is on internal software metrics. In contrast, UI tests are highly relevant to the *user view* since they assess the frontend behavior, which is the only access level for the client/user [1, pp. 13-15].

2.4.1 Static Analysis Tools

ESList (TypeScript), *CodeQL* (csharp), 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 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.

2.4.2 Quality Assessments

SonarCloud and *Code Climate* (see Figures 2.4 and 2.5) automatically scan the entire code base and grade the quality, it highlights what can be improved in order to obtain higher marks.


sonarcloud bot commented 3 weeks ago

Kudos, SonarCloud Quality Gate passed! Passed

A 0 Bugs
A 0 Vulnerabilities
A 0 Security Hotspots
A 0 Code Smells

No Coverage information
No Duplication information

Figure 2.4: *SonarCloud* issue detector for pull request

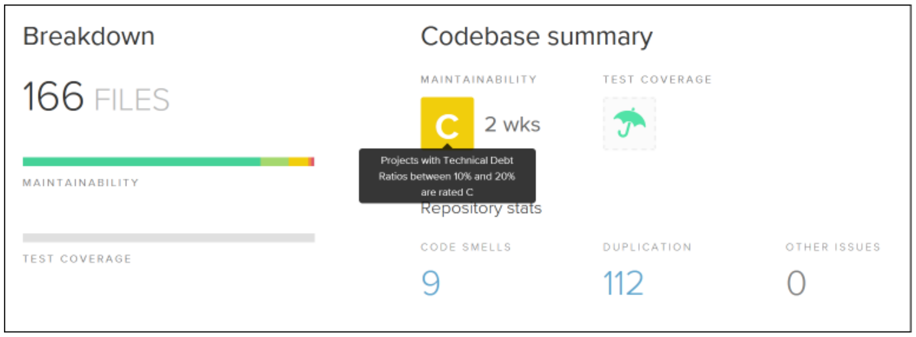


Figure 2.5: *Code Climate* code base summary

2.4.3 Tests

The back- and frontend testing are crucial quality checkpoints, as a failure in either would prevent the deployment of the system (see Figure 2.6). Unit, integration, and UI tests have been implemented, but end-to-end tests have not been finalized. Testing of the onion architecture, as described in Section 2.2.1, was done using 39 unit tests. Additionally, 17 integration tests were employed to test the behavior of combined layers.

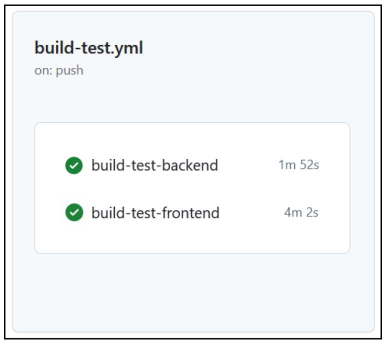


Figure 2.6: Chain of back- and frontend tests

2.5 Licence Compatability

This project uses a MIT licence. This license is one of the most permissive licenses and therefore also compatible with other open source licenses [3].

Show how the licenses are distributed using ScanCode - linux

3 Process Perspective

3.1 Developer Interaction

During the development process, the majority of interaction among the developers has occurred on-site. The group have 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 guide for local env. 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 has represented an important aspect of the project success, considering that each developer attend 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 dashboard displaying pending tasks, their respective type, and current progress. By doing so, the group has been able to maintain transparency 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 scan to mitigate the risk of the service to break. Once these checks are successfully completed, the PR is deemed safe for merging into the master branch, which initiates a deployment pipeline.

3.3.1 `build-test.yml`

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

build-test-backend, which 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, which runs on an ubunto 20.04 machine and includes the following steps:

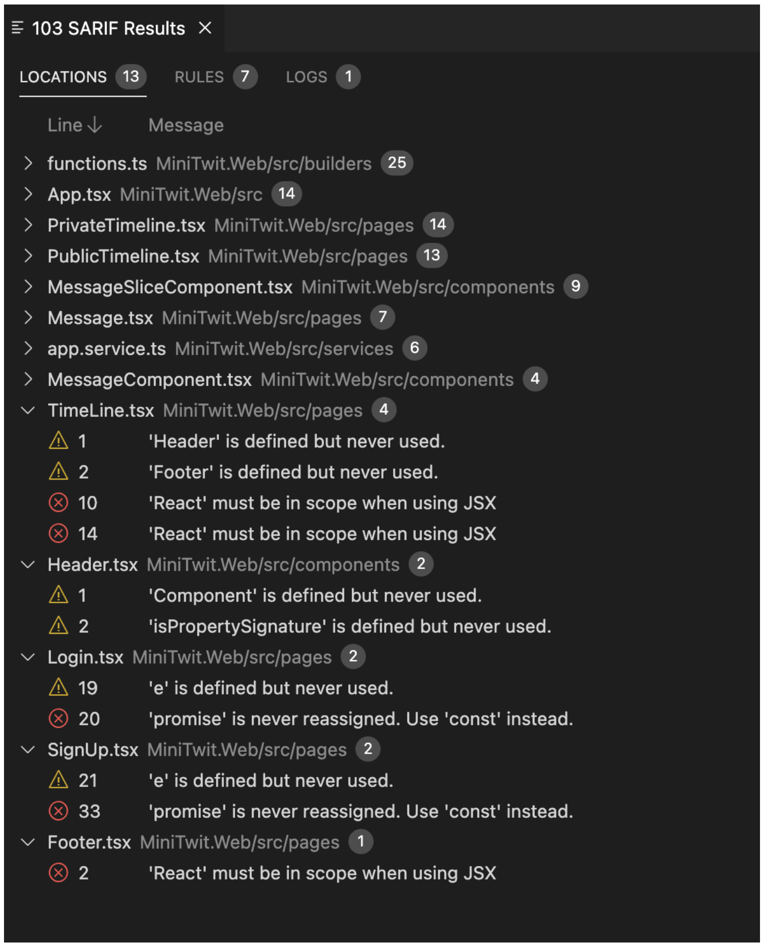
1. Checkout the repository
2. Install Node.js version 18 and dependencies for caching
3. Install dependencies for the MiniTwit frontend
4. Build the MiniTwit frontend
5. Run frontend tests using Playwright, in a docker container started by docker-compose -f docker-compose.ui.yml up -d
6. Stop the UI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

3.3.2 `codeql.yml`

Pipeline that integrates codeQL (C sharp analysis), which scans the back end code for C # specifically vulnerabilities and bugs, and provide guidance on how to fix them.

3.3.3 `eslint.yml`

Pipeline that integrates *eslint* for checking front end code, this workflow automatically detect and report issues in the code by uploading a *.sarif* file to GitHub. It provide guidance on how to fix the issues reported, such as unused variables, unused import and type-related errors.

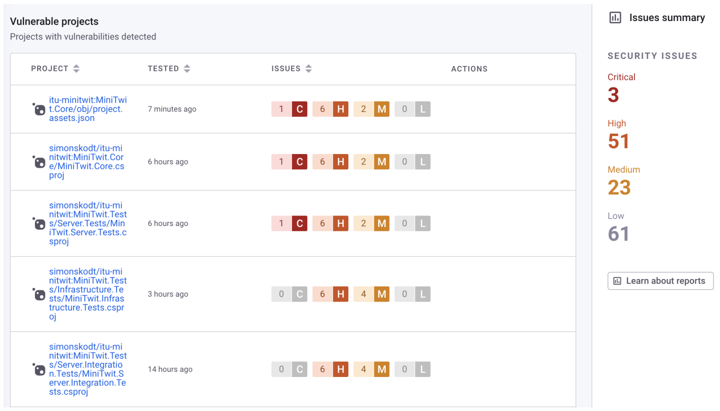

```
103 SARIF Results ×
LOCATIONS 13   RULES 7   LOGS 1

    Line ↓      Message
>  functions.ts MiniTwit.Web/src/builders 25
>  App.tsx MiniTwit.Web/src 14
>  PrivateTimeline.tsx MiniTwit.Web/src/pages 14
>  PublicTimeline.tsx MiniTwit.Web/src/pages 13
>  MessageSliceComponent.tsx MiniTwit.Web/src/components 9
>  Message.tsx MiniTwit.Web/src/pages 7
>  app.service.ts MiniTwit.Web/src/services 6
>  MessageComponent.tsx MiniTwit.Web/src/components 4
∨  TimeLine.tsx MiniTwit.Web/src/pages 4
    ⚠ 1         'Header' is defined but never used.
    ⚠ 2         'Footer' is defined but never used.
    ⊗ 10        'React' must be in scope when using JSX
    ⊗ 14        'React' must be in scope when using JSX
∨  Header.tsx MiniTwit.Web/src/components 2
    ⚠ 1         'Component' is defined but never used.
    ⚠ 2         'isPropertySignature' is defined but never used.
∨  Login.tsx MiniTwit.Web/src/pages 2
    ⚠ 19        'e' is defined but never used.
    ⊗ 20        'promise' is never reassigned. Use 'const' instead.
∨  SignUp.tsx MiniTwit.Web/src/pages 2
    ⚠ 21        'e' is defined but never used.
    ⊗ 33        'promise' is never reassigned. Use 'const' instead.
∨  Footer.tsx MiniTwit.Web/src/pages 1
    ⊗ 2         'React' must be in scope when using JSX
```

Figure 3.1: Eslint scan - Report

3.3.4 `snyk-security.yml`

Pipeline that integrates with Snyk, which provide continues security monitoring. The pipeline scans the project's dependencies to identify any known vulnerabilities and automatically sets up a pull request that updates packages which are outdated or contains vulnerabilities.



Vulnerable projects

Projects with vulnerabilities detected

PROJECT	TESTED	ISSUES	ACTIONS
itu-minitwit:MiniTwit.Core/obj/project.assets.json	7 minutes ago	1 C 6 H 2 M 0 L	
simonskodt/itu-minitwit:MiniTwit.Core/MiniTwit.Core.csproj	6 hours ago	1 C 6 H 2 M 0 L	
simonskodt/itu-minitwit:MiniTwit.Tests/Server.Tests/MiniTwit.Server.Tests.csproj	6 hours ago	1 C 6 H 2 M 0 L	
simonskodt/itu-minitwit:MiniTwit.Tests/Infrastructure.Tests/MiniTwit.Infrastructure.Tests.csproj	3 hours ago	0 C 6 H 4 M 0 L	
simonskodt/itu-minitwit:MiniTwit.Tests/Server.Integration.Tests/MiniTwit.Server.Integration.Tests.csproj	14 hours ago	0 C 6 H 4 M 0 L	

Issues summary

SECURITY ISSUES

Critical	3
High	51
Medium	23
Low	61

Figure 3.2: Snyk Security - Report

3.3.5 `continuous-deployment.yml`

Every pipleline highlighed so far, is triggered either by pushing to a development 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 [4]. If the PR passes the gate, it can be deployed. This is done by mergin the PR, which will trigger the continuous-deployment pipeline.

This pipeline is separated into two jobs, namely build an 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 rsync to copy the Docker Compose files and monitoring/logging configuration files to the live server. Then it 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 emplyed 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 branch 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 where where we work with two types of branches.

- The main branch / trunk (long-lived)
- Severel 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 development branch by means of pull requests. Once the merge is accomplished, the feature branch is deleted. Due to branch protection rules merge request can only be accepted onto the main branch after it has undergone review, approval, and has passed all pipeline checks. In order 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 has the following format.

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

3.6 Monotoring

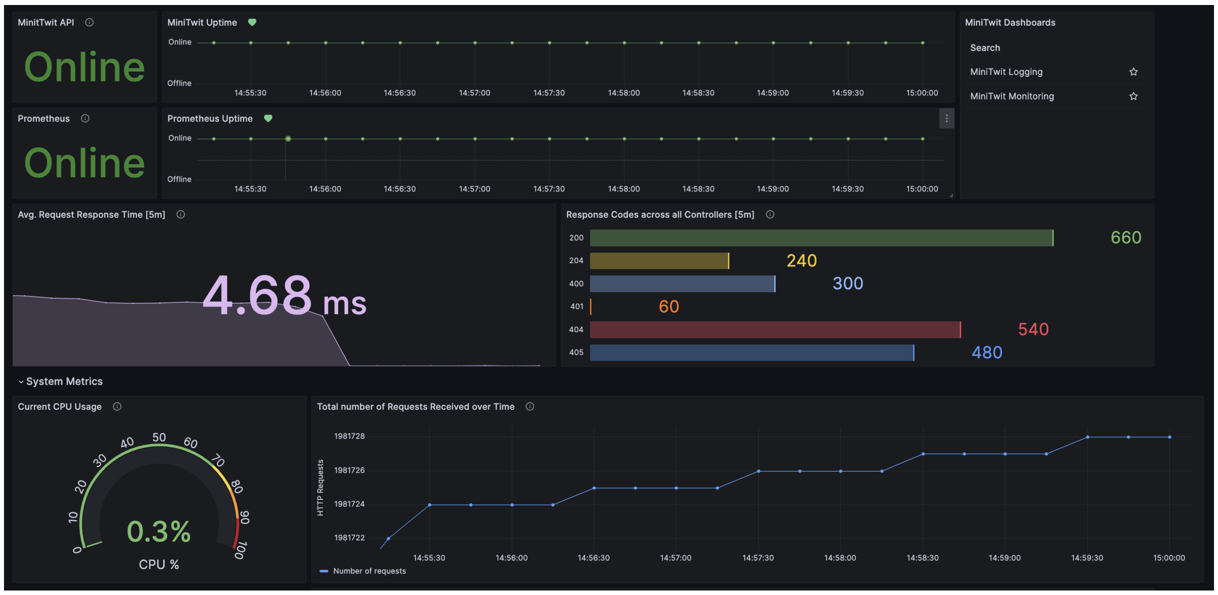
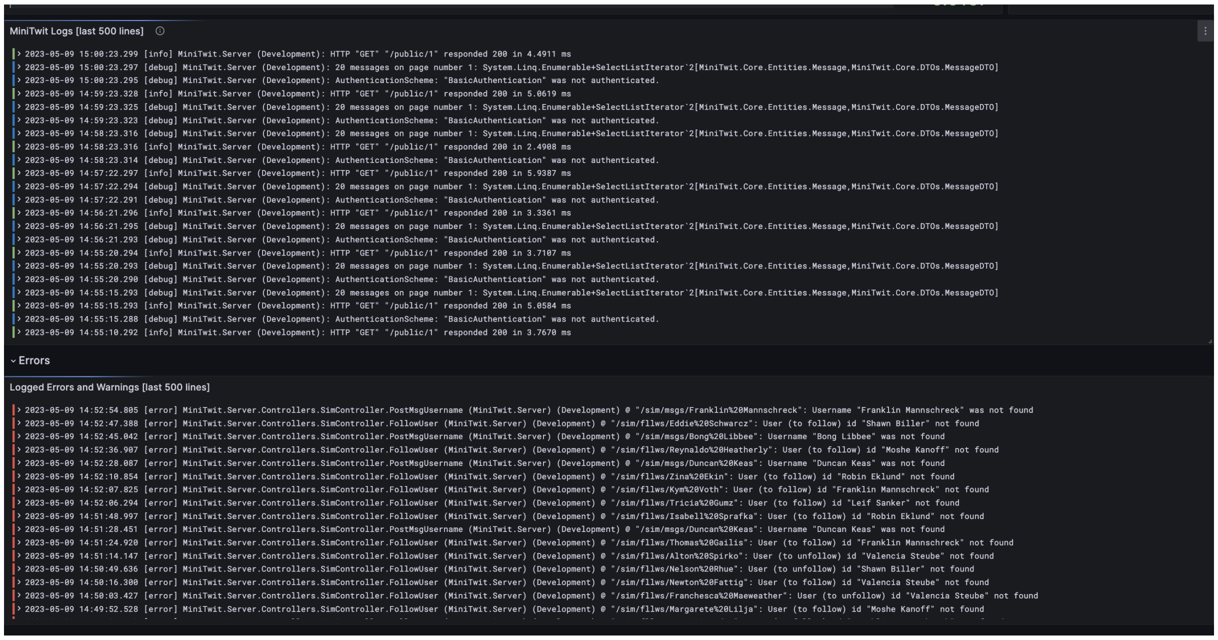


Figure 3.3: Monitoring from grafana dashboard

3.7 Logging



MiniTwit Logs [last 500 lines]

```
> 2023-05-09 15:00:23.299 [info] MiniTwit.Server (Development): HTTP "GET" "/public/1" responded 200 in 4.4911 ms
> 2023-05-09 15:00:23.297 [debug] MiniTwit.Server (Development): 20 messages on page number 1: System.Linq.Enumerable+SelectListIterator`2[MiniTwit.Core.Entities.Message,MiniTwit.Core.DTOs.MessageDTO]
> 2023-05-09 15:00:23.295 [debug] MiniTwit.Server (Development): AuthenticationScheme: "BasicAuthentication" was not authenticated.
> 2023-05-09 14:59:23.328 [info] MiniTwit.Server (Development): HTTP "GET" "/public/1" responded 200 in 5.0619 ms
> 2023-05-09 14:59:23.325 [debug] MiniTwit.Server (Development): 20 messages on page number 1: System.Linq.Enumerable+SelectListIterator`2[MiniTwit.Core.Entities.Message,MiniTwit.Core.DTOs.MessageDTO]
> 2023-05-09 14:59:23.323 [debug] MiniTwit.Server (Development): AuthenticationScheme: "BasicAuthentication" was not authenticated.
> 2023-05-09 14:58:23.316 [debug] MiniTwit.Server (Development): 20 messages on page number 1: System.Linq.Enumerable+SelectListIterator`2[MiniTwit.Core.Entities.Message,MiniTwit.Core.DTOs.MessageDTO]
> 2023-05-09 14:58:23.316 [info] MiniTwit.Server (Development): HTTP "GET" "/public/1" responded 200 in 2.4908 ms
> 2023-05-09 14:58:23.314 [debug] MiniTwit.Server (Development): AuthenticationScheme: "BasicAuthentication" was not authenticated.
> 2023-05-09 14:57:22.297 [info] MiniTwit.Server (Development): HTTP "GET" "/public/1" responded 200 in 5.9387 ms
> 2023-05-09 14:57:22.294 [debug] MiniTwit.Server (Development): 20 messages on page number 1: System.Linq.Enumerable+SelectListIterator`2[MiniTwit.Core.Entities.Message,MiniTwit.Core.DTOs.MessageDTO]
> 2023-05-09 14:57:22.291 [debug] MiniTwit.Server (Development): AuthenticationScheme: "BasicAuthentication" was not authenticated.
> 2023-05-09 14:56:21.296 [info] MiniTwit.Server (Development): HTTP "GET" "/public/1" responded 200 in 3.3361 ms
> 2023-05-09 14:56:21.295 [debug] MiniTwit.Server (Development): 20 messages on page number 1: System.Linq.Enumerable+SelectListIterator`2[MiniTwit.Core.Entities.Message,MiniTwit.Core.DTOs.MessageDTO]
> 2023-05-09 14:56:21.293 [debug] MiniTwit.Server (Development): AuthenticationScheme: "BasicAuthentication" was not authenticated.
> 2023-05-09 14:55:20.294 [info] MiniTwit.Server (Development): HTTP "GET" "/public/1" responded 200 in 3.7107 ms
> 2023-05-09 14:55:20.293 [debug] MiniTwit.Server (Development): 20 messages on page number 1: System.Linq.Enumerable+SelectListIterator`2[MiniTwit.Core.Entities.Message,MiniTwit.Core.DTOs.MessageDTO]
> 2023-05-09 14:55:20.290 [debug] MiniTwit.Server (Development): AuthenticationScheme: "BasicAuthentication" was not authenticated.
> 2023-05-09 14:55:15.293 [debug] MiniTwit.Server (Development): 20 messages on page number 1: System.Linq.Enumerable+SelectListIterator`2[MiniTwit.Core.Entities.Message,MiniTwit.Core.DTOs.MessageDTO]
> 2023-05-09 14:55:15.293 [info] MiniTwit.Server (Development): HTTP "GET" "/public/1" responded 200 in 5.0584 ms
> 2023-05-09 14:55:15.288 [debug] MiniTwit.Server (Development): AuthenticationScheme: "BasicAuthentication" was not authenticated.
> 2023-05-09 14:55:10.292 [info] MiniTwit.Server (Development): HTTP "GET" "/public/1" responded 200 in 3.7670 ms
```

Logged Errors and Warnings [last 500 lines]

```
> 2023-05-09 14:52:54.805 [error] MiniTwit.Server.Controllers.SimController.PostMsgUsername (MiniTwit.Server) (Development) @ "/sim/msgs/Franklin%20Mannschreck": Username "Franklin Mannschreck" was not found
> 2023-05-09 14:52:47.388 [error] MiniTwit.Server.Controllers.SimController.FollowUser (MiniTwit.Server) (Development) @ "/sim/fllws/Eddie%20Schwarcz": User (to follow) id "Shawn Biller" not found
> 2023-05-09 14:52:45.042 [error] MiniTwit.Server.Controllers.SimController.PostMsgUsername (MiniTwit.Server) (Development) @ "/sim/msgs/Bong%20Libbee": Username "Bong Libbee" was not found
> 2023-05-09 14:52:36.907 [error] MiniTwit.Server.Controllers.SimController.FollowUser (MiniTwit.Server) (Development) @ "/sim/fllws/Reynaldo%20Heatherly": User (to follow) id "Moshe Kanoff" not found
> 2023-05-09 14:52:28.087 [error] MiniTwit.Server.Controllers.SimController.PostMsgUsername (MiniTwit.Server) (Development) @ "/sim/msgs/Duncan%20Keas": Username "Duncan Keas" was not found
> 2023-05-09 14:52:10.854 [error] MiniTwit.Server.Controllers.SimController.FollowUser (MiniTwit.Server) (Development) @ "/sim/fllws/Zina%20Ekin": User (to follow) id "Robin Eklund" not found
> 2023-05-09 14:52:07.825 [error] MiniTwit.Server.Controllers.SimController.FollowUser (MiniTwit.Server) (Development) @ "/sim/fllws/Kym%20Voth": User (to follow) id "Franklin Mannschreck" not found
> 2023-05-09 14:52:06.294 [error] MiniTwit.Server.Controllers.SimController.FollowUser (MiniTwit.Server) (Development) @ "/sim/fllws/Tricia%20Gumz": User (to follow) id "Leif Sanker" not found
> 2023-05-09 14:51:48.997 [error] MiniTwit.Server.Controllers.SimController.FollowUser (MiniTwit.Server) (Development) @ "/sim/fllws/Isabell%20Sprafka": User (to follow) id "Robin Eklund" not found
> 2023-05-09 14:51:28.451 [error] MiniTwit.Server.Controllers.SimController.PostMsgUsername (MiniTwit.Server) (Development) @ "/sim/msgs/Duncan%20Keas": Username "Duncan Keas" was not found
> 2023-05-09 14:51:24.920 [error] MiniTwit.Server.Controllers.SimController.FollowUser (MiniTwit.Server) (Development) @ "/sim/fllws/Thomas%20Gailis": User (to follow) id "Franklin Mannschreck" not found
> 2023-05-09 14:51:14.147 [error] MiniTwit.Server.Controllers.SimController.FollowUser (MiniTwit.Server) (Development) @ "/sim/fllws/Alton%20Spirko": User (to unfollow) id "Valencia Steube" not found
> 2023-05-09 14:50:49.636 [error] MiniTwit.Server.Controllers.SimController.FollowUser (MiniTwit.Server) (Development) @ "/sim/fllws/Nelson%20Rhue": User (to unfollow) id "Shawn Biller" not found
> 2023-05-09 14:50:16.300 [error] MiniTwit.Server.Controllers.SimController.FollowUser (MiniTwit.Server) (Development) @ "/sim/fllws/Newton%20Fattig": User (to follow) id "Valencia Steube" not found
> 2023-05-09 14:50:03.427 [error] MiniTwit.Server.Controllers.SimController.FollowUser (MiniTwit.Server) (Development) @ "/sim/fllws/Franchesca%20Maeweather": User (to unfollow) id "Valencia Steube" not found
> 2023-05-09 14:49:52.528 [error] MiniTwit.Server.Controllers.SimController.FollowUser (MiniTwit.Server) (Development) @ "/sim/fllws/Margarete%20Lilja": User (to follow) id "Moshe Kanoff" not found
```

Figure 3.4: Logging from grafana dashboard

3.8 Security

3.9 Scaling and Load Balancing

4 Lessons Learned Perspective

4.1 Evolution and Refactoring

4.2 Operation

4.3 Maintenance

4.4 DevOps Style

5 Conclusion

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

[4] Microsoft. “Deployment gates”. In: (2022).

Appendices

A Test

B Dependencies

B.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/

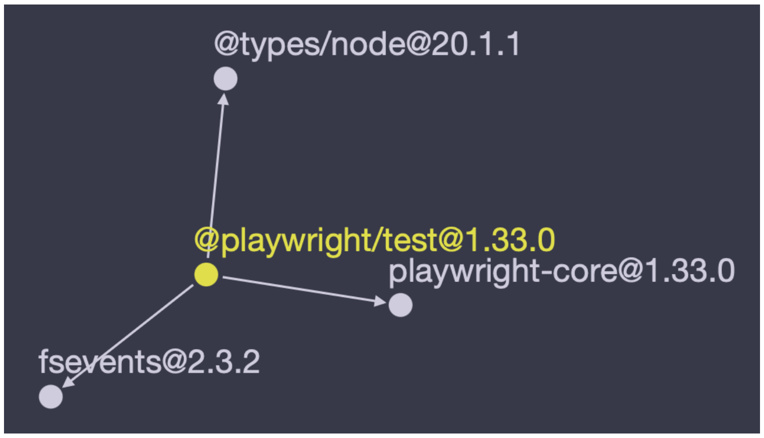


Figure B.1: Playwright dependency graph

B.1.1 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