Chirp! Project Report

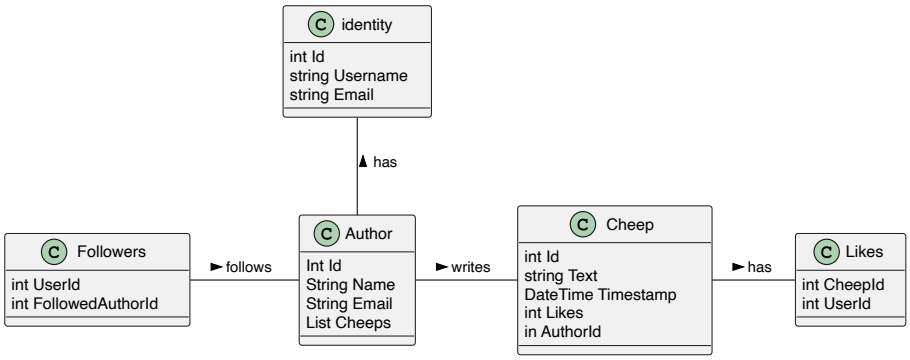
ITU BDSA 2023 Group 2

Casper Pilgaard cpil@itu.dk Jerome Rahal jrah@itu.dk
Mads Orfelt orfe@itu.dk Marius Stokkebro masto@itu.dk
Oliver Prip Hagmund ohag@itu.dk

1 Design and Architecture of *Chirp!*

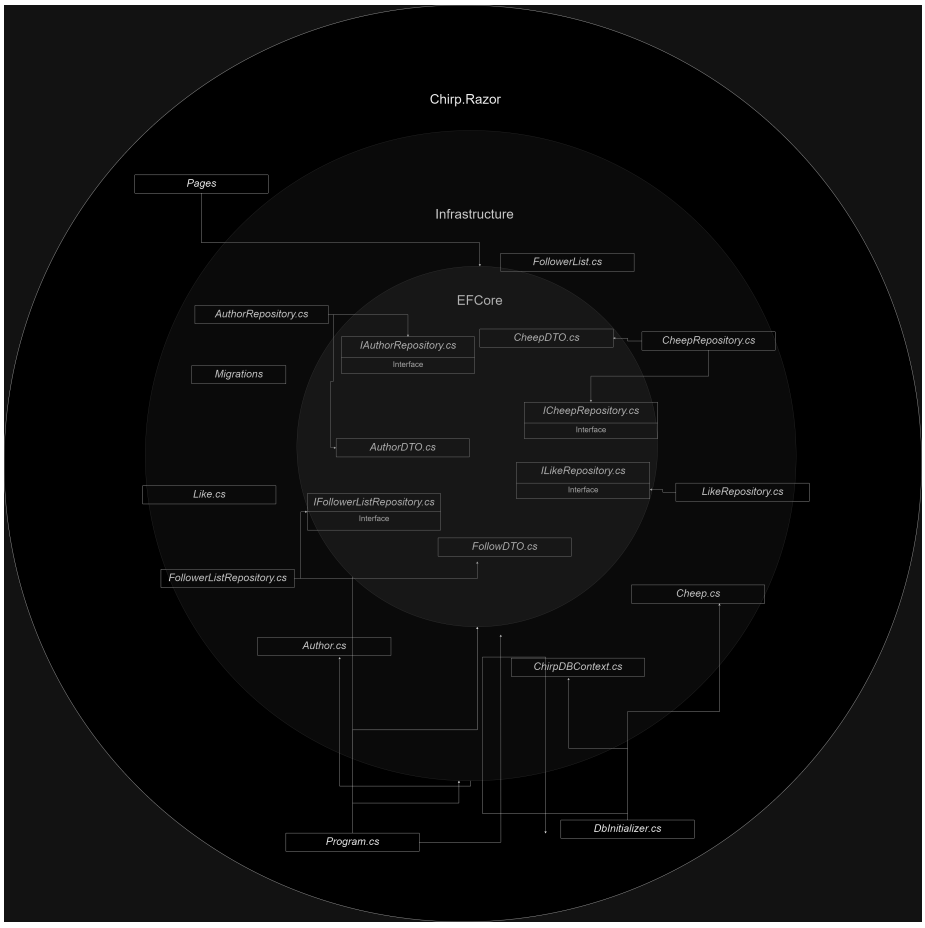
1.1 Domain model

The diagram below shows our domain model. The classes are the important entities in our program, as they relate to the real world domain of interest. The identity entity is made to show that we use asp.net core identity.



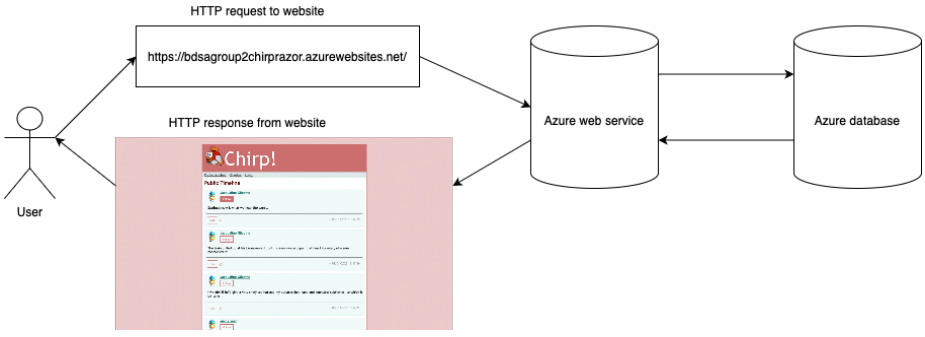
1.2 Architecture — In the small

The illustration below highlights the organization of our code base, within our onion architecture.



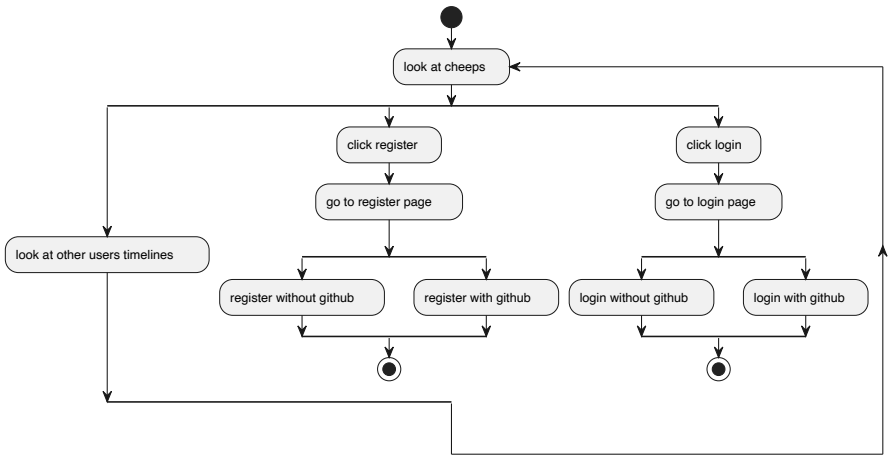
1.3 Architecture of deployed application

The illustration below exemplify the architecture of our deployed application, with focus on our client-server relation.

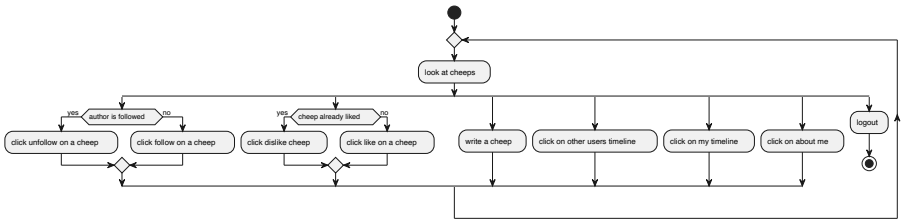


1.4 User activities

The diagram below illustrates how a non-authorized user will use the "Chirp!" application.

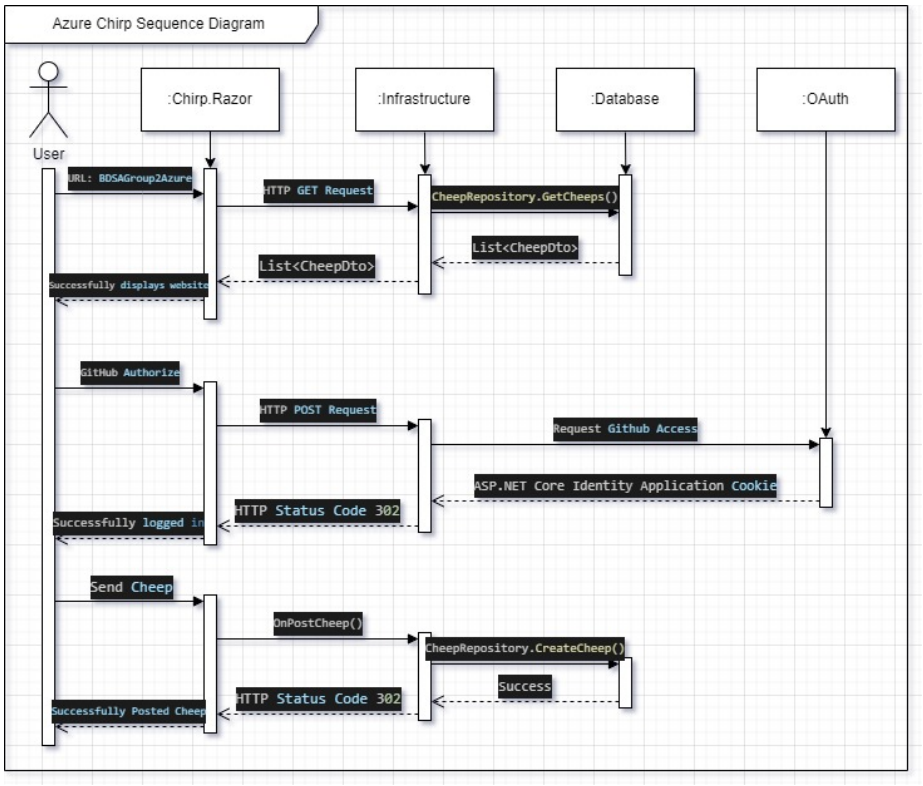


The diagram below illustrates how an authorized user can use the "Chirp!" application. The diagram is illustrated as a loop, since most of the tasks that an authorized user can do will end up with them at the "Public timeline" looking at cheeps. We omit how an authorized-user can interact with the "About me"- and "My timeline"-page. The difference between the "Public timeline" and "My timeline" are the cheeps they show. The "Public timeline" shows all cheeps and the "My timeline" only shows personal cheeps and cheeps from authors they follow.



1.5 Sequence of functionality/calls through *Chirp!*

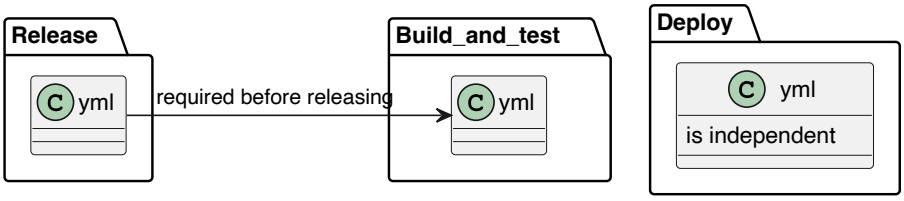
The sequence diagram below demonstrates the sequences of three different functionalities. Firstly the user attempts to access the webpage. Secondly the user attempts to gain authorization, by becoming an authorized user. Upon becoming an authorized user, they attempt to create a cheep.



2 Process

2.1 Build, test, release, and deployment

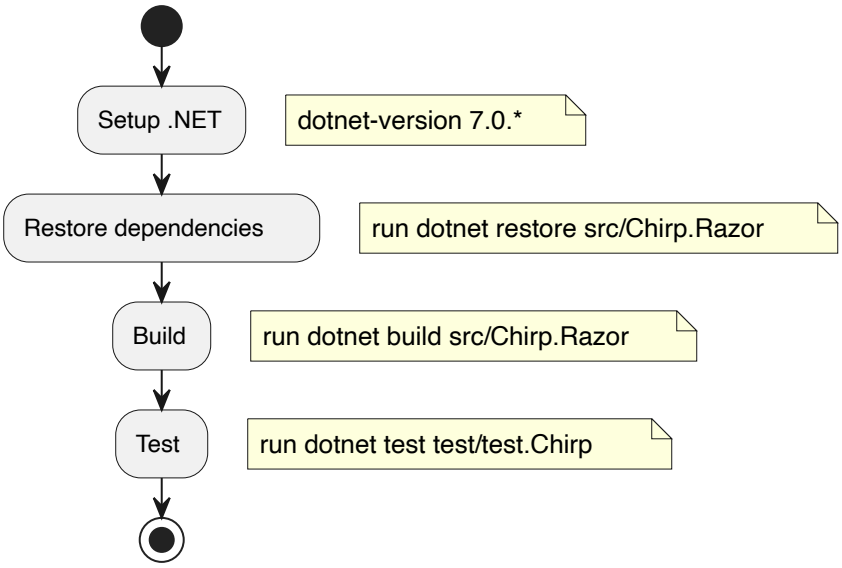
We have used Github Actions to continuously deploy our Chirp application. We will provide an explanation of how our continuous deployment tackles building-, testing- and deploying the program to the web platform, and ensuring working releases for all relevant platforms.



Building

To build the program we use the "build_and_test.yml" workflow. This workflow runs on pushes and pulls to main branch, this is to view the status of our tests whenever something gets added or merged. In the workflow, the three first steps

are responsible for building the program before testing.



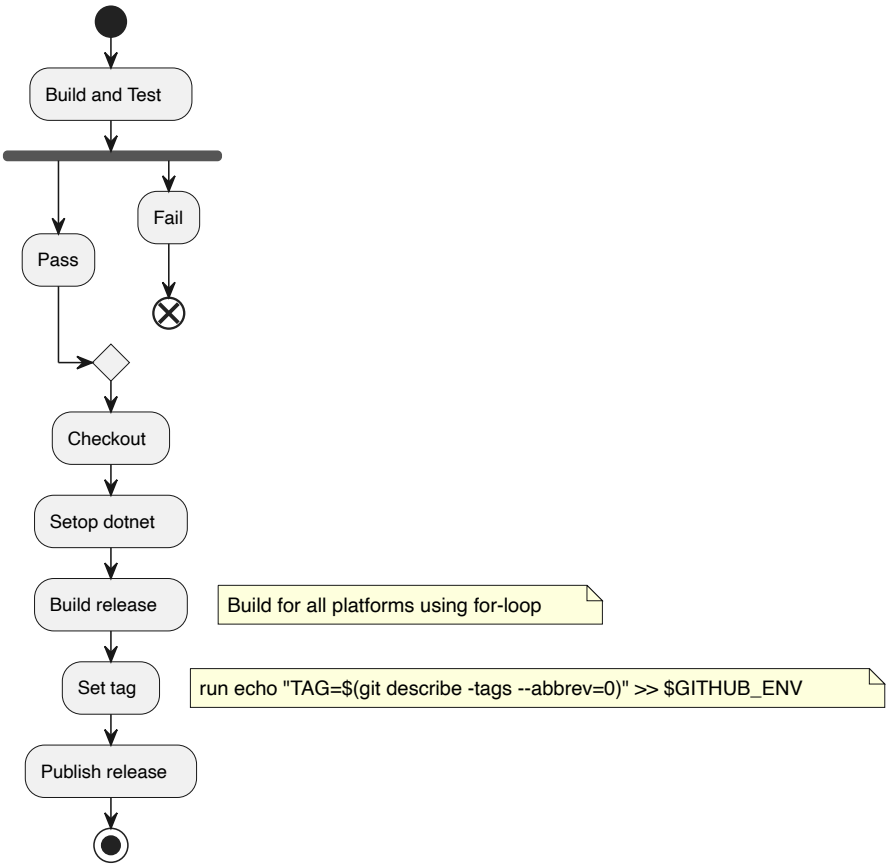
The first step is "Setup .NET" which uses "actions/setup-dotnet@v3", with dotnet version 7.0. This is responsible for setting up the program before building it, and is a part of the "action/checkout@v3" collection. The next step is to restore dependencies, this is a necessary step to ensure all our dependencies are restored. Lastly we run the "Build" step, which is responsible for building the program so that it can be tested upon.

2.1.1 Testing

Running the tests is done with the last step in the "build_and_test.yml" workflow. This step is called "Test" and runs our test suit. We have put a lot of thought and effort into having the correct and necessary tests such as; Unit- and Integration-tests.

2.1.2 Release

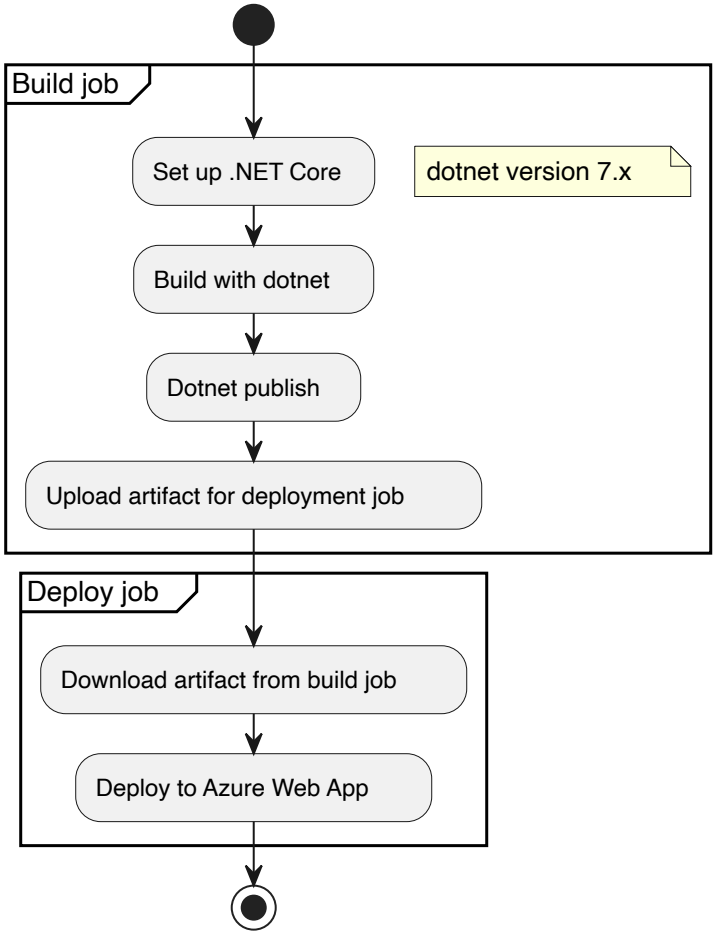
The intended use of the release.yml workflow is to create and release zip files, for windows, mac and linux where the program can be found and ran. Here is an illustration of how the release workflow works:



There are certain requirements for our release workflow to run which we view as crucial for the design of the worklow. The first major requirement our workflow has, is that it only runs once a tag has been pushed. The tag format is "v..*" which is in accordance with Semantic versioning. The next major requirement in our release workflow is that we have added the build_and_test.yml as a job which needs to run and complete appropriately, meaning the program must build correctly and all the tests must pass before the release job runs. This is a suitable requirement since we would never want to release a version of the program where the tests do not pass.

2.1.3 Deploy

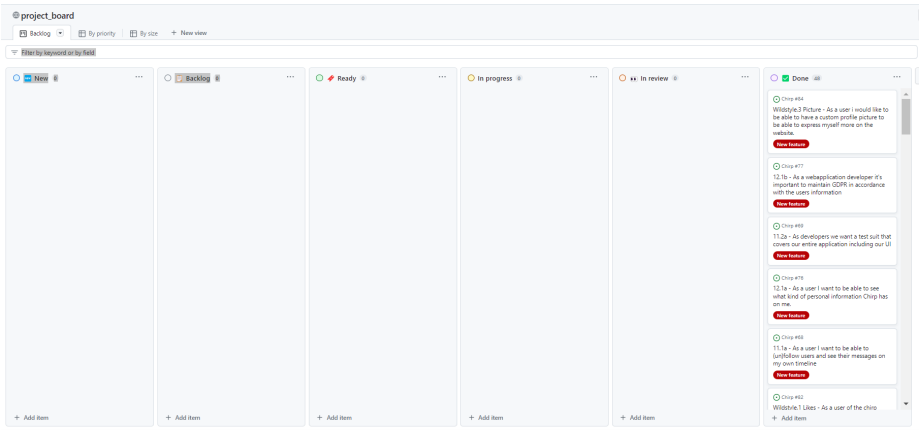
We deploy using the main_bdsagroup2chirprazor.yml. This workflow only runs on pushes to main branch, to ensure all new features are deployed to the website. The deploy-workflow consists of two jobs; build and deploy. Here is an illustration of the workflow:



The build job has four steps; The first step is to set up .NET core. Second step is to build. Thirdly it publishes and lastly it uploads the artifact. The deploy-job requires the build-job to finish successfully. The first step of the deploy-job is to download the artifact uploaded by the build job. The second step is to deploy that artifact to the azure web app.

It is important to note that this workflow makes and runs a separate build-job which does not require tests to pass. We recently found out about this and would have liked to change it. This would ensure that we only deploy once all tests pass, since we do not want to deploy a faulty application.

2.2 Team work



After each lecture we would quickly take each new software development and process step and convert them into issues. We had a GitHub workflow that automatically added the new feature tag to issues. Usually two people would sit down and write the coming weeks issues. The other people would work on previous tasks we had not completed yet. We would then delegate the new issues between us. Depending on the problem size we would sometimes work two or three people on one issue using pair programming otherwise it would be a single person working on one issue each. We would also make a new branch, specifically for that issue. We were not the best at keeping track of our project board. We would sometimes place an issue in the "in-progress"-column and then forget about it, even after it had been resolved. When we had new issues to write, we would make a pull request for the old issue and move it to the "Done"-column. Most of the time another member of the team who had not worked on the issue would review and approve the merge, but if no one answered the messages for a pull request. One of the team members who worked on the issue would approve and merge themselves, so the branches did not stay alive for too long.

We do not have any unfinished issues on our project board, and we have made all the features that we set out to make. One part of our program that we could have worked more on would have been to add more UI test, specifically for our wild-style features (see 'How to run test suite locally').

2.3 How to make *Chirp!* work locally

There are a few things you need to set up, before you are able to use Chirp locally. You will need to install Docker, which you can install from this link: https://www.docker.com/products/docker-desktop/ And git which you can install from this link: https://git-scm.com/downloads

You can now clone the repository by running:

```
git clone https://github.com/BDSA23-GROUP2/Chirp
```

You now need to get the database up and running. You need to run one of these commands depending on your system.

Windows

```
docker run -e "ACCEPT_EULA=Y" -e "MSSQL_SA_PASSWORD=6d3a3bdb-7993-42ab-8eb4-5fb4e27ef44a" -p
```

Mac

```
docker run -e "ACCEPT_EULA=Y" -e "MSSQL_SA_PASSWORD=6d3a3bdb-7993-42ab-8eb4-5fb4e27ef44a" -p
```

Now the database should be properly set up.

Make sure you are in the Chirp.Razor directory when running the last set of commands. If you should be in the root directory, type the following command in your terminal to go the correct folder.

```
cd src/Chirp.Razor
```

This command is for connecting to the database.

```
dotnet user-secrets set "ConnectionStrings:ChirpDb" "Server=127.0.0.1,1433; Database=Master;
```

These two commands are for the GitHub authentication. (The GitHub authentication locally is not the same as the website for safety reasons)

```
dotnet user-secrets set "GITHUB_PROVIDER_AUTHENTICATION_SECRET" "1d0ace927b02173f9a878119fdb
```

```
dotnet user-secrets set "AUTHENTICATION_GITHUB_CLIENTID" "f2b9cc87834340f6215a"
```

To start the program, type:

```
dotnet run
```

2.4 How to run test suite locally

To run our unit tests suite (including in-memory tests), from the root directory of the Chirp application. Run the following commands:

```
cd test/test.Chirp
```

```
dotnet test
```

The InMemoryTests.cs file tests various repository methods. This includes, testing of the CreateCheep-, UpdateAuthor- and FollowAuthor-method. For each test it starts by creating a database and seeds it with cheeps and authors. Once the tests are complete it disposes the database.

The UnitTestChirp.cs file tests if the public- and private-timeline exists, and if it can find specific users cheeps.

Make sure you are in the root directory when running the following commands. If you should be in the root directory, type the following commands in your terminal to run our UI test suite.

```
cd test/PlaywrightTests
```

If this is the first time running our UI tests you need to run an extra command before they work, this however requires you to have power-shell installed:

```
pwsh bin/Debug/net7.0/playwright.ps1 install
```

Now you are ready to run our UI tests, just type in the following command:

```
dotnet test
```

All of our UI tests are tested directly on our Azure webpage. Our UI test suit consists of both small unit tests like checking if the title of our page is "Chirp!" and integration test that test if a user can log in and follow another author. Our UI test suit is not covering all our wild style functionalities. After we had made our first UI tests (located in the UITest.cs script), we quickly ran out of credits. Fearing we would end in a situation like other groups we stopped creating and running our UI tests for the fear of not being able to host the webpage for the entirety of the course.

3 Ethics

3.1 License

To choose the license we started by looking at the website https://choosealicense.com/licenses/ to learn which license there were and which one would fit our project best. We immediately noticed the different types of permissions, conditions and limitations. Our group confirmed we wanted as few limitations and conditions as possible because it would be easier for the public to use our code. We came to the conclusion that the MIT license was the way to go. Just to be sure we asked our TA at one of our meetings if the license was suitable for our project and dependencies. He agreed and we opted for the MIT license.

3.2 LLMs, ChatGPT, CoPilot, and others

When we encountered difficult issues throughout the program and were unable to find an answer right away on the internet, we decided to get some help from ChatGPT. Sometimes it helped us, sometimes we did not get any further. Whenever we would commit anything that the AI either wrote for us or gave us inspiration for, we would write a commit message, that we had gotten help from ChatGPT.

When we first chose the images for the website, we took the first ones we could find, so we could be ready for the project demo day. However, we were not sure about the copyright of these images. Therefore, we instead decided to use OpenAI's image generation to create the images for our website. We researched and found out that there would be no copyright issues this way. Source: Section 3.a https://openai.com/policies/mar-2023-terms.