Chirp! Project Report

ITU BDSA 2023 Group 15

Daniel Sølvsten Millard dmil@itu.dk
Frederik Lund Rosenlund frlr@itu.dk
Jacob Vortscir Pærregaard jacp@itu.dk
Mads Christian Nørklit Jensen macj@itu.dk
Rasmus Lundahl Nielsen raln@itu.dk

1 Design and Architecture of *Chirp!*

1.1 Domain model

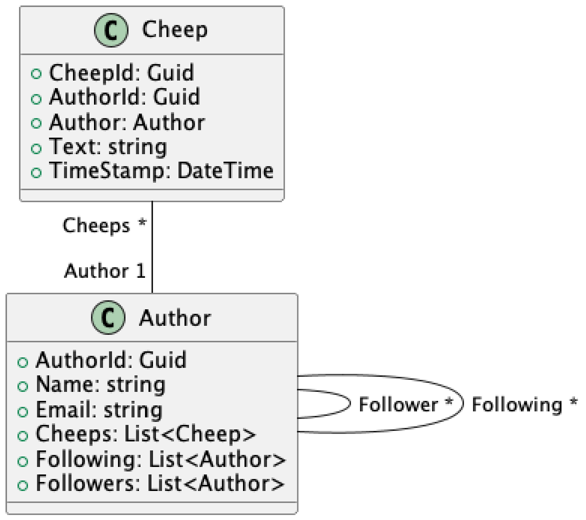


Figure 1: An overview of our domain model and its corresponding classes.

1.2 Architecture — In the small

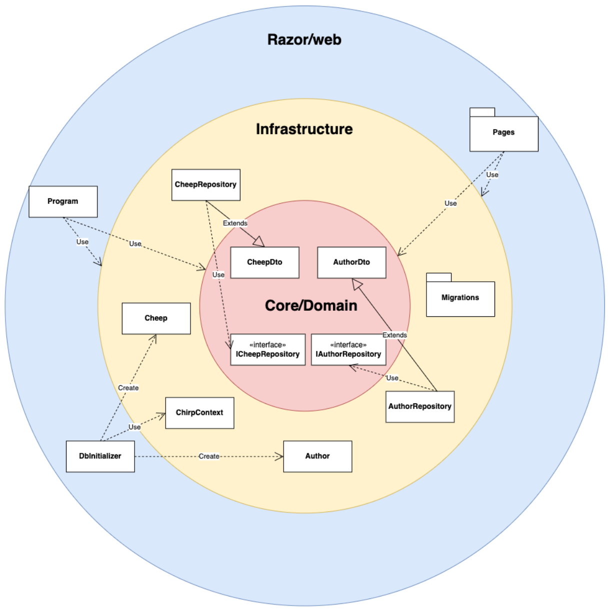


Figure 2: An illustration showing our final program architecture. The model is based on the Onion architecture model.

1.3 Architecture of deployed application

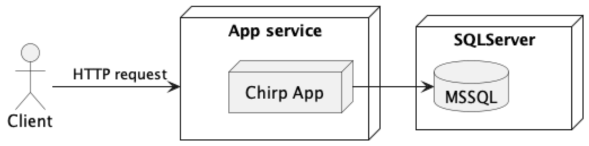


Figure 3: An illustration showing our final deployed architecture. The application is deployed on Microsoft Azure, and uses a MSSQL database.

1.4 User activities

1.4.1 Not logged in

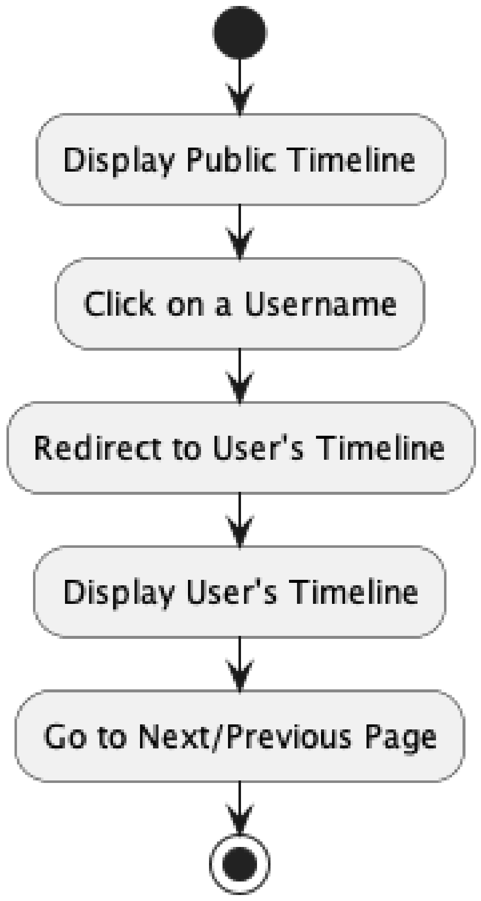


Figure 4: Activity diagram of what a user could do while not logged in.

1.4.2 Logged in

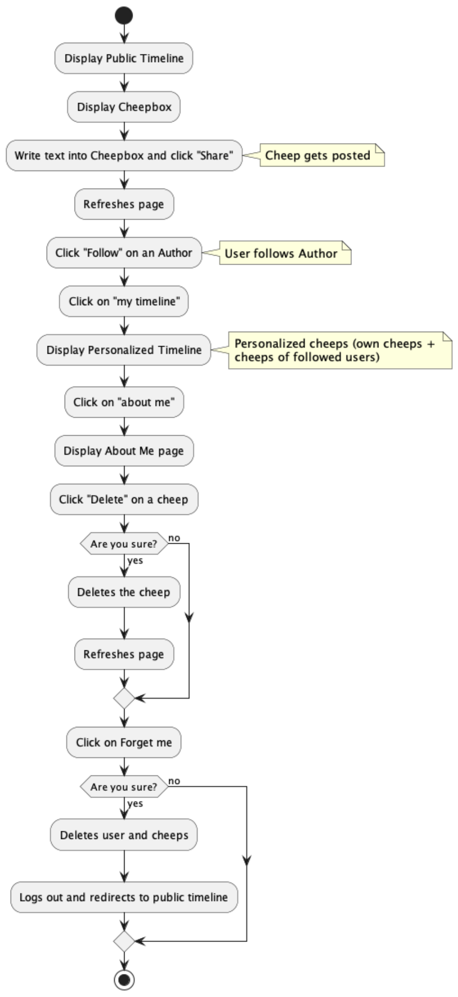


Figure 5: Activity diagram of what a user could do while logged in.

1.4.3 Logging in

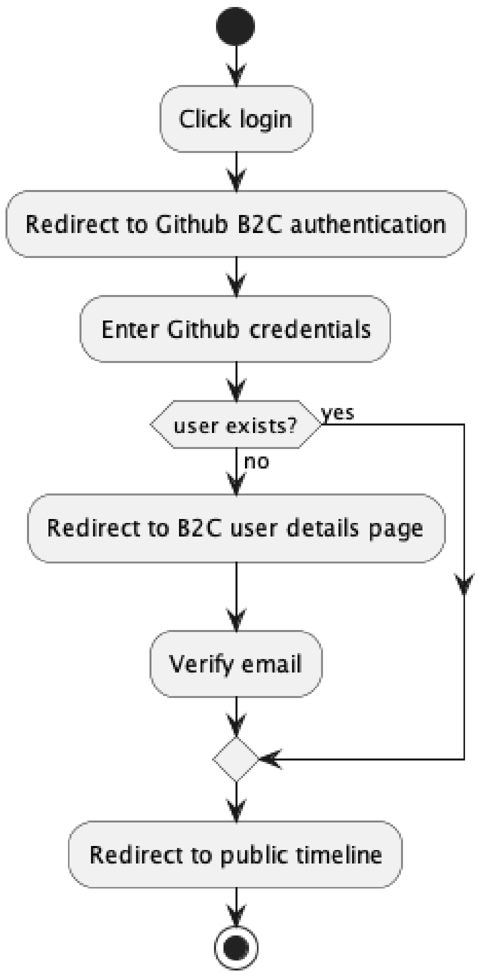

Click login
Redirect to Github B2C authentication
Enter Github credentials
user exists?
yes
no
Redirect to B2C user details page
Verify email
Redirect to public timeline

Figure 6: Activity diagram of a user logging into the application.

1.5 Sequence of functionality/calls trough *Chirp!*

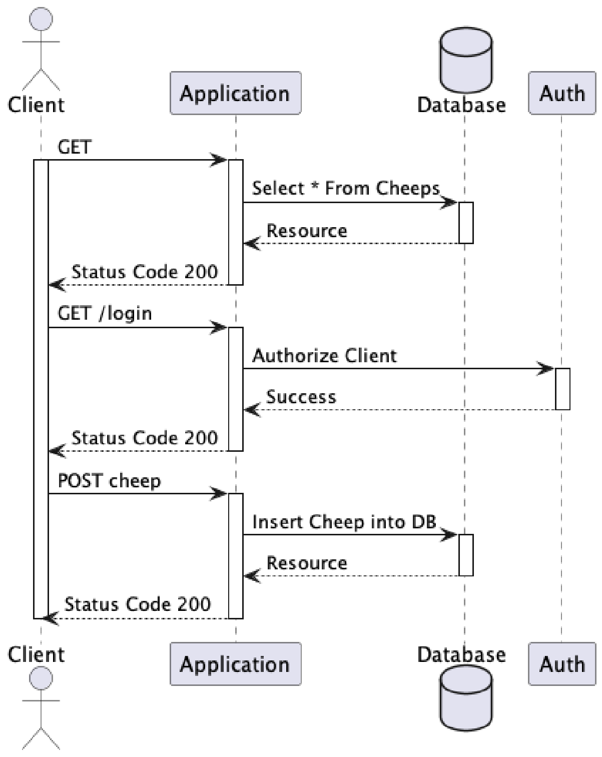


Figure 7: This is a sequence diagram showing the calls through the application. The diagram gives 3 different scenarios of how the application can be used, with corresponding functionality and calls.

2 Process

2.1 Build, test, release, and deployment

2.1.1 Building and testing the application

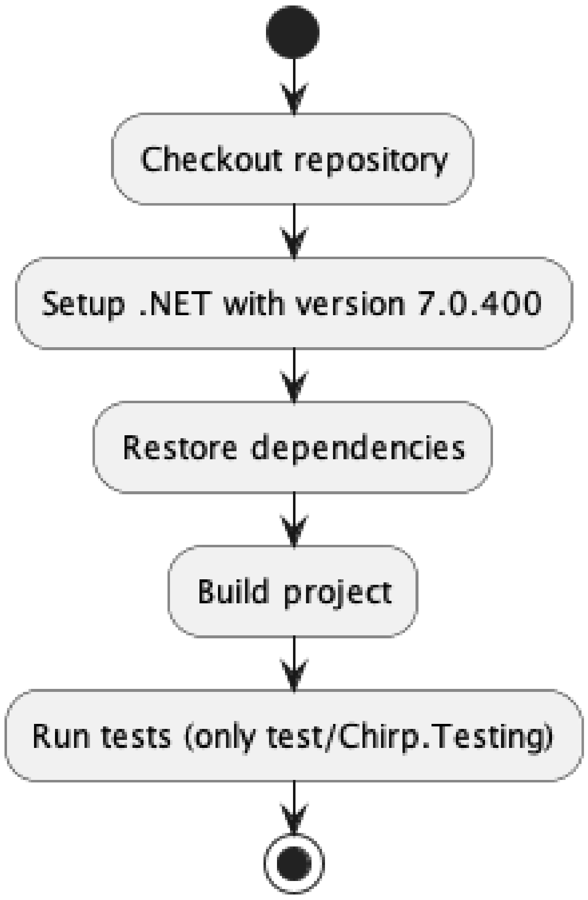

Checkout repository
Setup .NET with version 7.0.400
Restore dependencies
Build project
Run tests (only test/Chirp.Testing)

Figure 8: Illustration of build and test

Whenever a push is made to GitHub, a workflow will start testing the application. This is the case for every branch and makes sure that whenever we make something - even if it is a new feature - the tests will run and the program won't fail when merged to main. This only includes the Chirp.Testing-folder as we had issues with GitHub-login for the UI-testing-workflow.

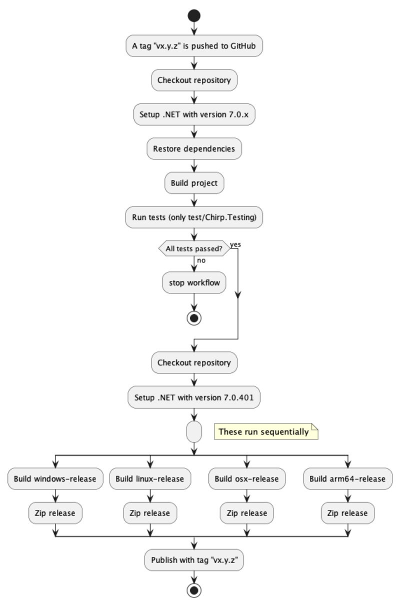


Figure 9: Illustration of auto release

2.1.2 Automating GitHub releases

When a tag is pushed to GitHub using the syntax “vx.y.z”, a workflow is started that first tests the application (same as building and testing the application). If the tests passes a new release is made where the version number (tag) is the title. The workflow then builds the application for windows, linux, macOS and macArm separately, zips them and uploads them to the release-page.

We use *semantic versioning* as our “guide” on how to determine the version number.

2.1.3 Automating Azure deployment

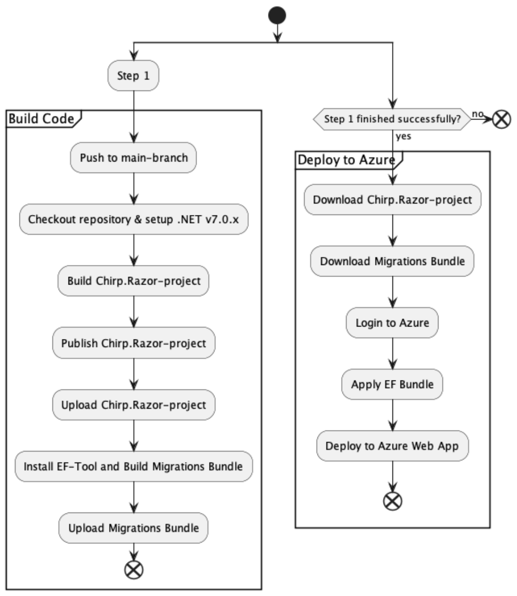

Step 1
Build Code
Push to main-branch
Checkout repository & setup .NET v7.0.x
Build Chirp.Razor-project
Publish Chirp.Razor-project
Upload Chirp.Razor-project
Install EF-Tool and Build Migrations Bundle
Upload Migrations Bundle
Step 1 finished successfully?
no
yes
Deploy to Azure
Download Chirp.Razor-project
Download Migrations Bundle
Login to Azure
Apply EF Bundle
Deploy to Azure Web App

Figure 10: Illustration of auto deploy

When a push is made to the main-branch a workflow deploying the application to our Azure server is started. The first tasks in the workflow builds the application, installs ef-tool and creates a migration bundle with the ef-tool. This is then uploaded.

A new task is then created that downloads the application and migrations bundle. The workflow then logs in to Azure and applies the bundle to our application through a connection string. Lastly the application is deployed to Azure using an Azure-webapp-deploy-action.

2.2 Team work

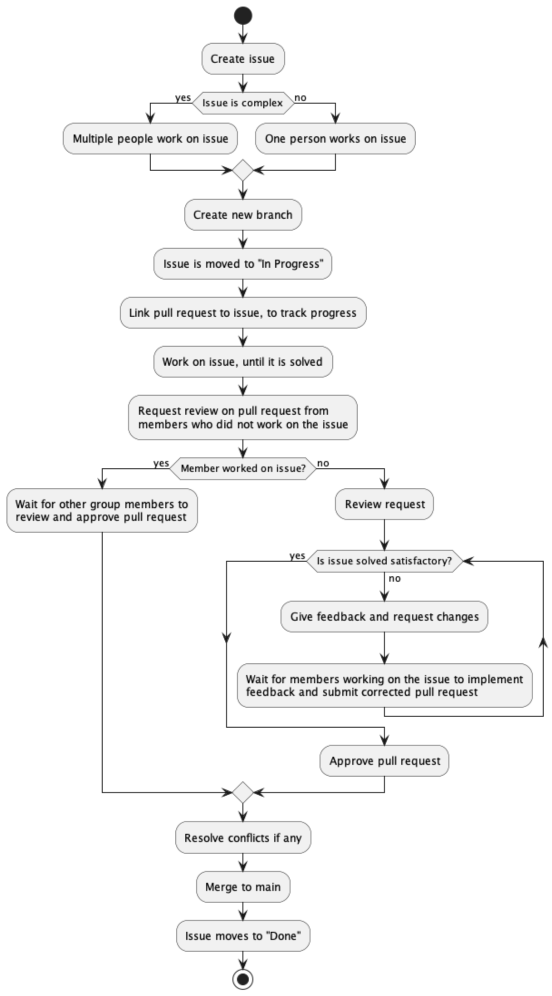
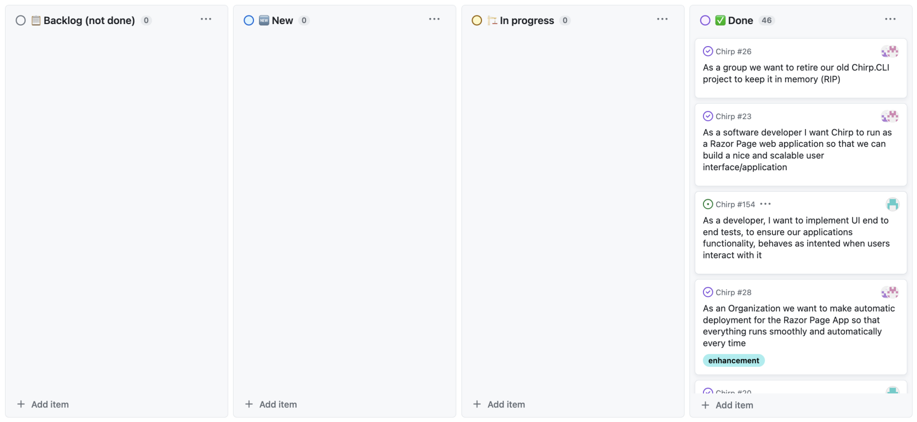

Create issue
Issue is complex
yes
no
Multiple people work on issue
One person works on issue
Create new branch
Issue is moved to "In Progress"
Link pull request to issue, to track progress
Work on issue, until it is solved
Request review on pull request from members who did not work on the issue
Member worked on issue?
yes
no
Wait for other group members to review and approve pull request
Review request
Is issue solved satisfactory?
yes
no
Give feedback and request changes
Wait for members working on the issue to implement feedback and submit corrected pull request
Approve pull request
Resolve conflicts if any
Merge to main
Issue moves to "Done"

Figure 11: Illustration showing how an issue is done

When a new issue is created, it is automatically assigned to the “new” column in the project board. Members of the team can then assign themselves to the issue, with the amount of people working on it, being mainly dependent on the complexity of the issue. When a member assigns themselves to an issue, they move the issue to the “in progress” column. A new branch is created to work on the issue, and a pull request is linked to the issue, to track the progress on it. When the issue is considered completed, by the working members, the pull request is reviewed by the other members of the team. Members then review whether they find the solution satisfactory or not. If the solution is not found satisfactory, they provide feedback, throughout their review and await the working members to consider feedback and submit a corrected pull request for review. If the solution is found satisfactory, the pull request is merged into the main branch. The issue is then moved to the “done” column.



Chirp #26
As a group we want to retire our old Chirp.CLI project to keep it in memory (RIP)

Chirp #23
As a software developer I want Chirp to run as a Razor Page web application so that we can build a nice and scalable user interface/application

Chirp #154
As a developer, I want to implement UI end to end tests, to ensure our applications functionality, behaves as intented when users interact with it

Chirp #28
As an Organization we want to make automatic deployment for the Razor Page App so that everything runs smoothly and automatically every time
enhancement

Figure 12: Project Board right before handin

Seen above is our project board, after we stopped working on our project. We have no unresolved issues, as we managed to implement the features we set out to implement. As far as we know and have tested, our functionality of the program should be complete to the extend we intended it to be.

2.3 How to make *Chirp!* work locally

2.3.1 Initial setup for computers that do not have an ARM CPU

To run our Chirp application locally it is required that you clone the project, which can be done by running the following command in the terminal:

```
git clone https://github.com/ITU-BDSA23-GROUP15/Chirp.git
```

As we are using a MSSQL database, it is required that you have docker installed on your machine. If you do not have docker installed, you can find a guide on how to install it here.

To download a Docker image, create a container for it and run it, run the following command in the terminal:

```
sudo docker run -e "ACCEPT_EULA=Y" -e "MSSQL_SA_PASSWORD=<YourStrong@Passw0rd>" \
   -p 1433:1433 --name sql1 --hostname sql1 \
   -d \
   mcr.microsoft.com/mssql/server:2022-latest
```

Replace `<YourStrong@Passw0rd>` with a password of your choice. This password will be used to access the database, so make sure to remember it.

2.3.2 Initial setup for computers with an ARM CPU

If you have a Mac with an M1 chip, you need to run the following command instead:

```
docker run -d --name sql1 --cap-add SYS_PTRACE \
  -e 'ACCEPT_EULA=1' \
  -e 'MSSQL_SA_PASSWORD=<YourStrong@Passw0rd>' \
  -p 1433:1433 \
   mcr.microsoft.com/azure-sql-edge
```

Furthermore in Docker Desktop, navigate to settings -> General and check the box “Use Rosetta for x86/amd64 emulation on Apple Silicon”. This is required to run the MSSQL database on an ARM CPU.

2.3.3 Connecting to the database

We now need to set a Connection string with the password to the docker, that you have just set in the previous command. This is saved in the user secrets, which can be done by running the following command from a terminal at the root of the project:

```
dotnet user-secrets set "ConnectionStrings:ChirpDb"
"Server=localhost;Database=ChirpDb;User Id=SA;
Password=<YourStrong@Passw0rd>;MultipleActiveResultSets=true;TrustServerCertificate=True"
--project src/Chirp.Razor/
```

To verify that the connection string has been set correctly, run the following command:

```
dotnet user-secrets list --project src/Chirp.Razor/
```

From the root of the project run the following commands to run the project locally:

```
dotnet build
dotnet run --project src/Chirp.Razor/
```

This will run a local instance of the program on your machine, at the address https://localhost:5273/. You can now access the program from your browser, by entering the address in the address bar.

You should now expect to see the public timeline, stating at the top of the site, that you need to login to cheep, a login button should be available at the top right. You can browse different pages of the public timeline, as well as click on users to access their timelines to see only their cheeps. The rest of the features on the site, will only become available after logging in, which is done using your github account.

2.4 How to run test suite locally

2.4.1 Unit and integration tests

This project contains two test suites, as we have separated our UI test, into a separate test suite. To run our unit/integration tests, make sure docker is running, open a terminal at the root of the project and run the command:

```
dotnet test test/Chirp.Testing/
```

As our tests are run in a docker test containers, coupled with the fact that our program relies on a database, nearly every method we have requires a read from the database when testing it. Therefore our unit tests are also integration tests, as they depend on the integration between our code and the database. To have proper separation between unit tests and integration tests, we would need to make our unit tests use an in memory database. This would lead to a lot of duplicate test cases as they would be virtually copied to the integration tests, where we would use a docker test container for the database.

We have attempted to achieve a near full code coverage with our tests. Most of our repository methods are very similar, nearly identical, so a lot of our tests are very similar as well.

2.4.2 UI tests (E2E)

Our UI test acts as an end to end test, as it tests the UI and functionality of our program as a whole by simulating user input. A few things is required to run the UI test suite. As our UI test is made with Playwright, the a supported browser for this needs to be installed. This can be ensured by running the following command:

```
npx playwright install
```

Furthermore, since an authenticated user is required to access most of the functionality of the program, and since our only means of authenticating is through Github, user secrets needs to be set up with a valid Github account. This can be done by running the following command:

```
dotnet user-secrets set "UserName" "<Insert your Github username>"
--project test/Chirp.UI.Testing/Chirp.UI.Testing.csproj
```

Followed by:

```
dotnet user-secrets set "Password" "<Insert your Github password>"
--project test/Chirp.UI.Testing/Chirp.UI.Testing.csproj
```

To run the UI test suite, you need to have a local instance of the program running. This can be done from a terminal at the root of the project by entering the following commands:

```
dotnet build
dotnet run --project src/Chirp.Razor/
```

In another terminal at the root of the project, enter the following command to run the UI test suite:

```
dotnet test test/Chirp.UI.Testing/
```

If you have two factor authentication enabled on your Github account, if you do not want to set your Github account login in your user secrets or if you want to visually follow the test run the following command instead of the one above:

```
dotnet test test/Chirp.UI.Testing/ --
Playwright.BrowserName=chromium
Playwright.LaunchOptions.Headless=false
Playwright.LaunchOptions.SlowMo=1000
```

The SlowMo value can be adjusted to your liking, to slow down or speed up the test suite, to make it easier to follow what is happening.

The UI/E2E-test that this test suite contains, tests the overall functionality of our program along with the UI elements and the navigation between the URLs.

Note: This will not work if you have not specified Azure Tenant and Client ID coupled to the app registration. These are secrets and are therefore not shared.

3 Ethics

3.1 License

We've chosen the MIT License, which is a permissive free software license, because of its limited restriction on reuse. In this project we wanted to encourage reuse of our code, and therefore we've chosen a license that allows for this. The MIT License is also a very common license, which makes it easy for others to understand the terms of the license.

3.2 LLMs, ChatGPT, CoPilot, and others

During the development of our project, we have utilized ChatGPT and Copilot. ChatGPT has been used for general questions and research on project topics, to help narrow down the scopes of some of the more daunting tasks. We've also used both ChatGPT and Copilot as tools to understand error codes, when debugging our implementations, but with varying success. Sometimes suggested solutions were helpful, other times the LLM had not understood the error and provided wrong suggestions. Copilots main function in our development project, has been to auto complete code that had already been made previously, especially the repository database functions. It has also been used to create new code, but if the logic was not already specified, the code was often faulty, which resulted in us spending a lot of time on debugging the autocompleted code.