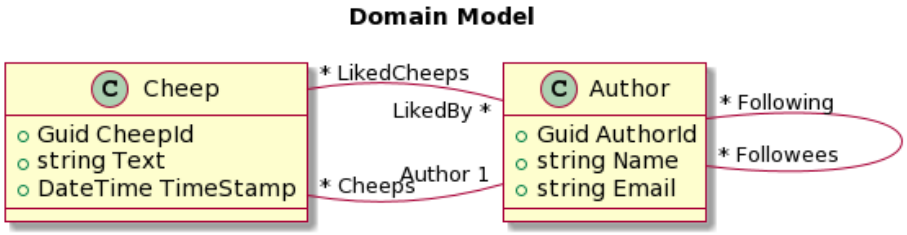
Chirp! Project Report

ITU BDSA 2023 Group 5

Jakob Arnfred Nielsen arni@itu.dk
Johan Brandi johbr@itu.dk
Niklas Zeeberg Hessner Christensen nizc@itu.dk
Olivier-Baptiste Hansen oliha@itu.dk
Philip Guozhi Han Pedersen phgp@itu.dk

1 Design and Architecture of *Chirp!*

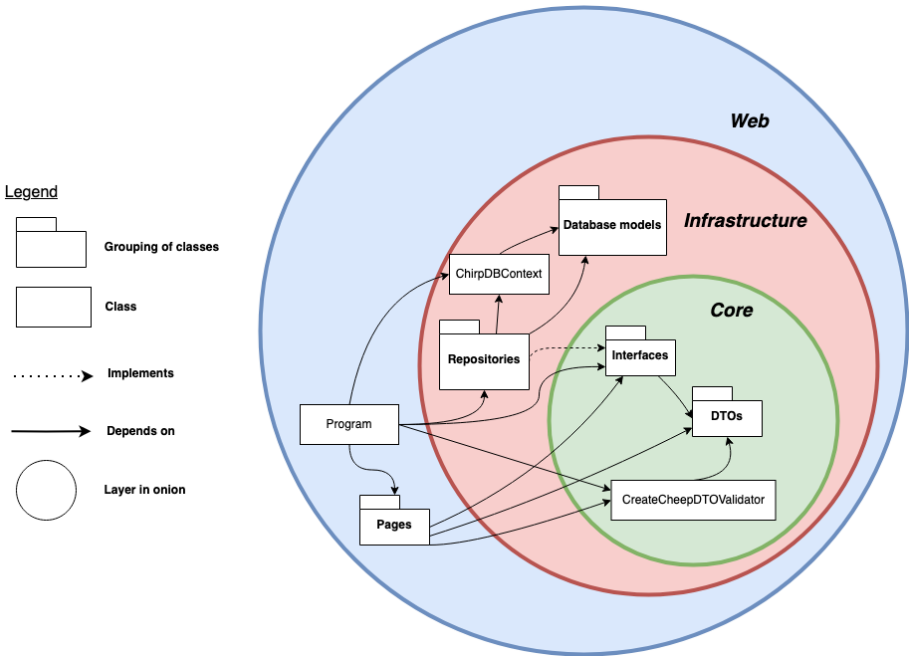
1.1 Domain model



The Author class represents a user in *Chirp!*. The Cheep class represents the messages shared on the *Chirp!* platform. A Cheep is written by an Author, and an Author can write many Cheeps. An Author can follow, as well as be followed by, many other Authors. An Author can like many Cheeps, at most one time per cheep. A Cheep can be liked by many Authors.

1.2 Architecture — In the small

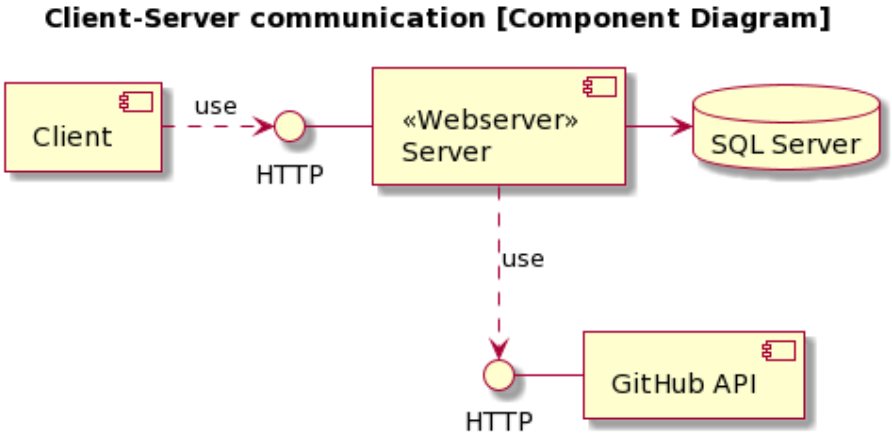
The architecture is illustrated on the figure below with three circles representing the layers of our onion architecture. The legend to the left should be self-explanatory except for the term *"Grouping of classes"*. Grouping of classes refers to a set of classes that serve similar functions e.g. the Repositories - it is an abstraction made to increase readability.



A piece of un-centered figure text

1.3 Architecture of deployed application

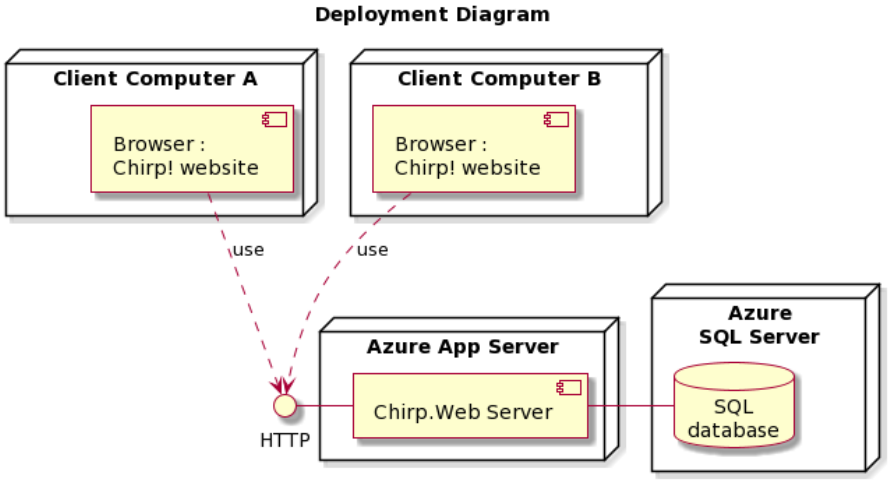
The following component diagram shows how the different high level components communicate with each other.



The client communicates with the Azure web server using HTTP requests. The Azure web server itself uses HTTP requests to get information about the user from the GitHub API. Furthermore, the Azure web server has a connection to

the Azure SQL Server which contains the database.

The following deployment diagram shows the artifacts of the system and where they are deployed.



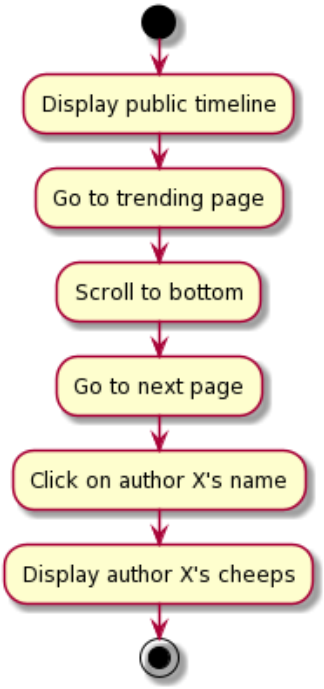
Each client computer uses a browser, which uses HTTP requests, to access the website. The Chirp.Web project executable is deployed on an Azure App Server which hosts it, and has a connection to an Azure SQL Server. The SQL Server contains a database with all the persistent data of the *Chirp!* application.

1.4 User activities

The following user activity diagrams are for typical user journeys in our *Chirp!* application, where a user completes a couple of tasks.

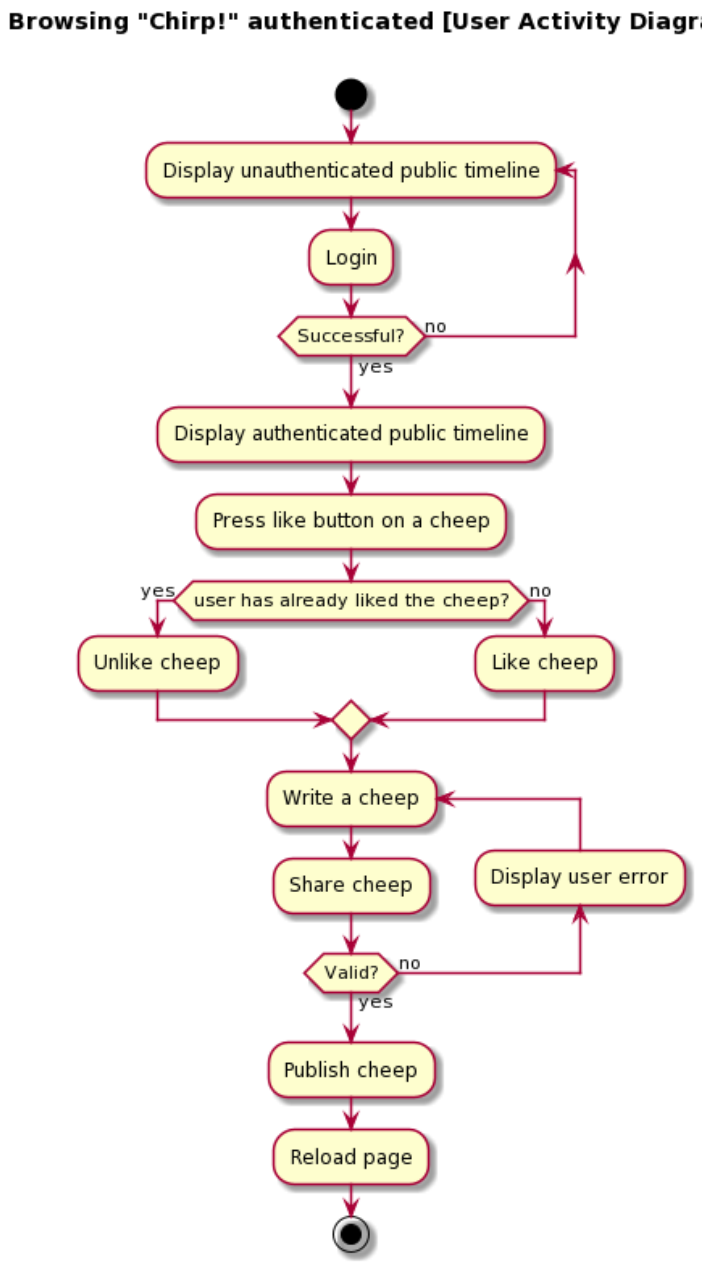
The diagram below illustrates an example of a user journey for an **unauthenticated** user of the application.

Browsing "Chirp!" unauthenticated [User Activity Diagram]



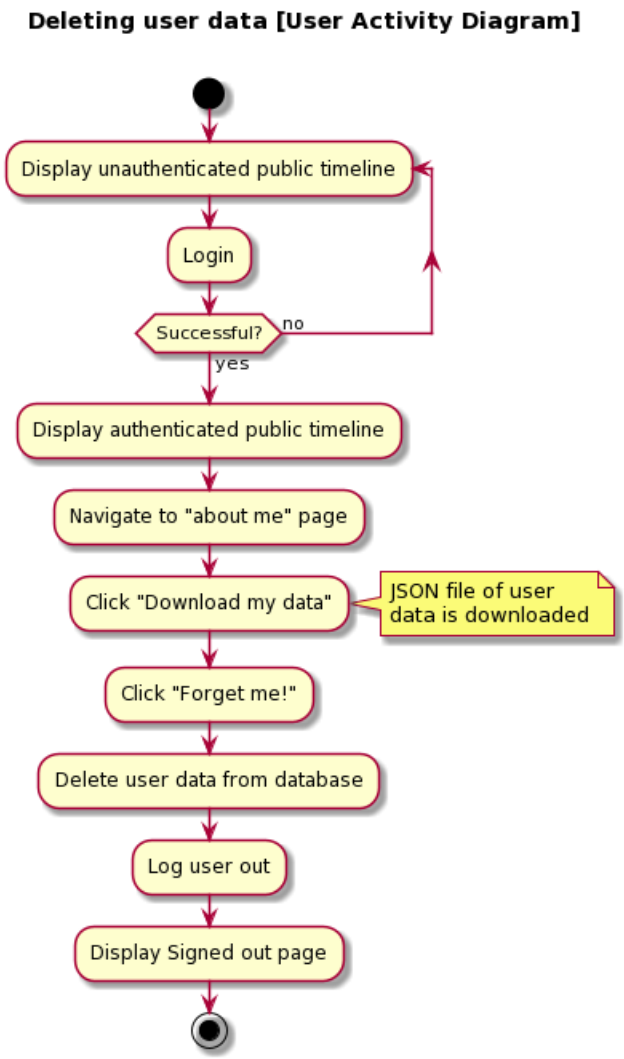
"Author X" represents any author present on the second page of the public timeline. The user browses the trending page, finds an interesting cheep, and decides to check out what else author X has to offer.

The diagram below illustrates an example of a user journey for an **authenticated** user of the application.



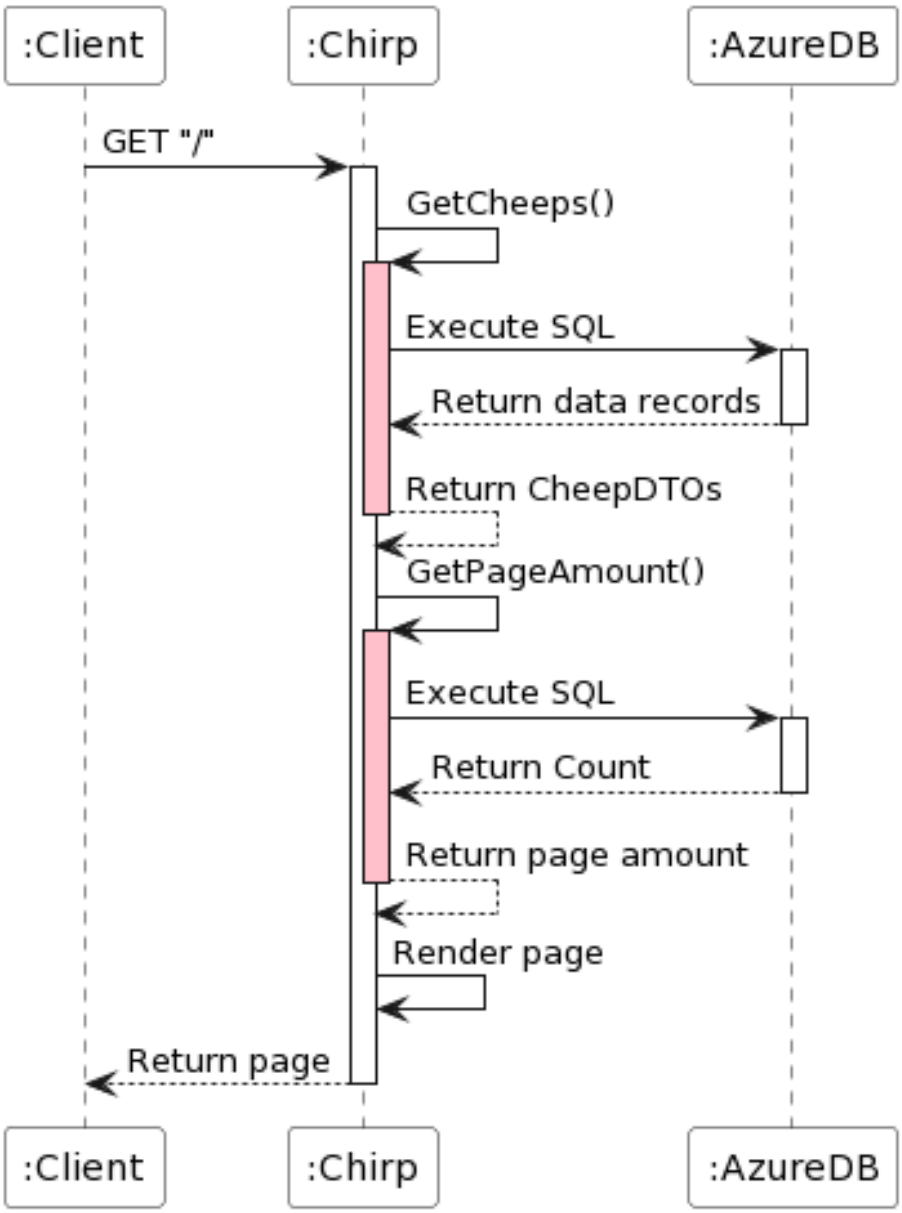
The user logs in, likes a cheep from the public timeline, and decides to write their own.

The next diagram illustrates an example of a user journey where a user wants to exercise their GDPR rights.



The user logs in, goes to the about me page to download, and delete their data.

1.5 Sequence of functionality/calls through *Chirp!*

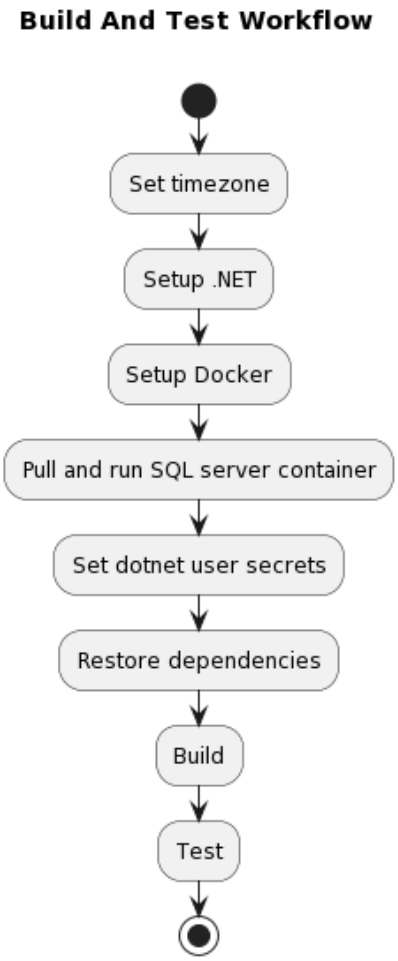


The figure above illustrates the sequence of events, from a user requests the root of *Chirp!* to a page is rendered and returned.

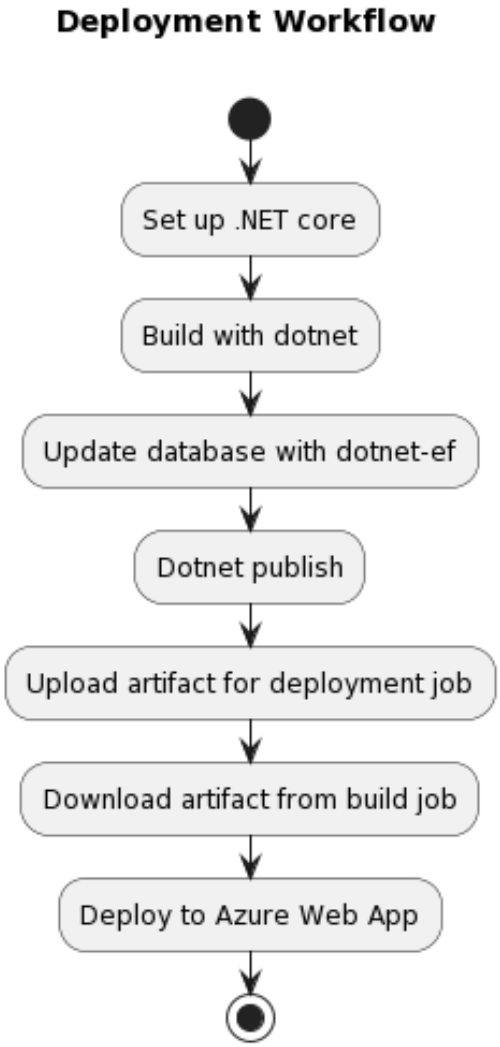
2 Process

2.1 Build, test, release, and deployment

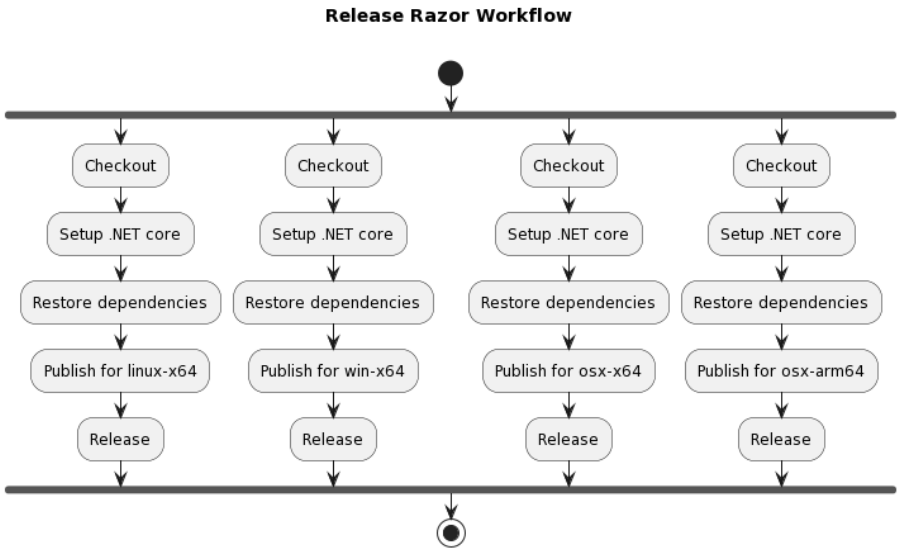
The following UML activity diagrams illustrate the GitHub actions workflows that are run when different criteria are met. This will be briefly described under the respective diagrams. Note that each step can fail and will result in the Github Action aborting.



The workflow on the figure above is run upon every push and pull request to main. It builds and tests the application in order to keep main void of faulty code (as a **safety net**).

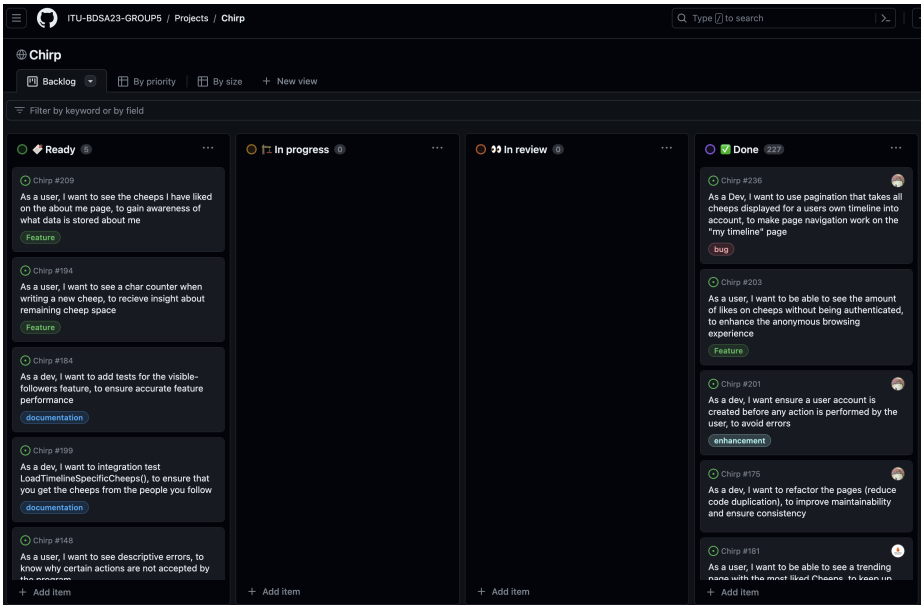


The deployment workflow illustrated on the figure above is run upon every push to main. Note the redundant "build" step. We do not need this since the "publish" step already builds the application. This redundancy was not noticed during development and has not been removed due to time constraints. The illustration describes the two jobs: "build" and "deploy". Jobs are normally run in parallel, however, these are run **sequentially** as we do not want to deploy before the application is successfully built.



The release workflow illustrated on figure above only runs if a push to main contains a version tag. We've made use of a matrix strategy in order to automatically create multiple **parallel** job runs that build, publish, zip and release the application for their respective platform.

2.2 Team work



No additional functionality is in progress. Planned features we did not get to implement include:

- A character counter when writing cheeps
- Adding liked cheeps to about me page (as well as the "Download my data" json data)

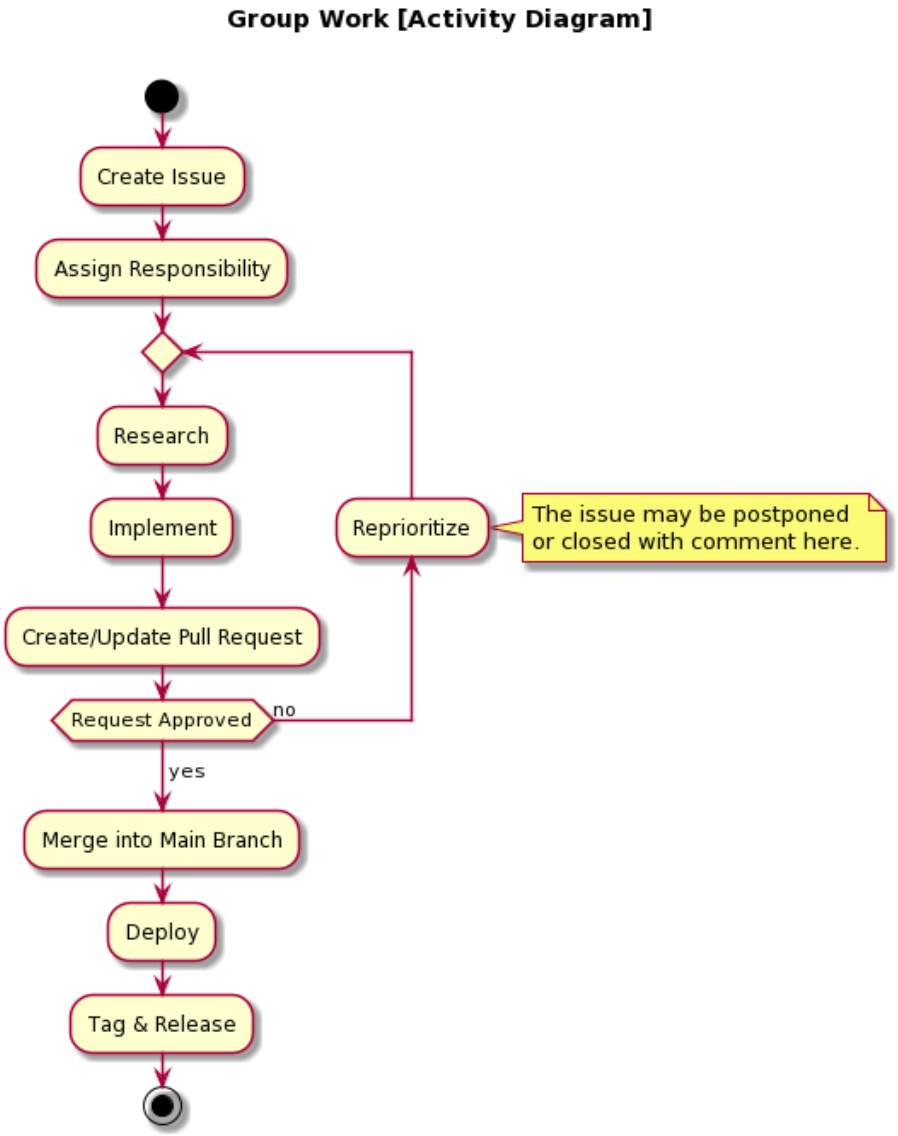
To get an overview of our group work consult the figure below.

After an issue has been created we assign responsibility for it to one or more persons.

We work alone or with pair programming iteratively in research and implement steps until the issue is handled.

Then we create a pull request and notify group members who are not involved. They strive to review within 24 hours. If the request is approved we merge it into the Main branch. Otherwise we evaluate whether or not to continue working on the issue or close it with a comment.

After merging deployment happens automatically, and ideally a tag is manually applied afterwards and pushed which triggers the release workflow.



2.3 How to make *Chirp!* work locally

2.3.1 Prerequisites

- .NET (>=7.0)
- dotnet ef-tools
- Docker
- Azure AD B2C Tenant

- Github OAuth App

2.3.2 Azure AD B2C

Chirp! uses an Azure AD B2C user flow for authentication. The `AzureADB2C` object in `appsettings.Development.json` must be configured according to Step 2. of the documentation.

1. Add a user-secret from within the `src/Chirp.Web` folder, using the following command in your terminal:

```
dotnet user-secrets set "AzureADB2C:ClientSecret" "[CLIENT-SECRET]"
```

Replacing [CLIENT-SECRET] with your client secret from Azure.

2. You must configure your Github OAuth App with your user flow on your Azure ADB2C tenant.

2.3.3 Local database

Chirp! uses an MSSQL database, which can be run locally using Docker.

1. Set up the database using the following command from your terminal:

```
docker run -e "ACCEPT_EULA=Y" -e "MSSQL_SA_PASSWORD=my}}Pass;word" \
   -p 1433:1433 --name azuresql --hostname azuresql \
   -d \
   mcr.microsoft.com/mssql/server:2022-latest
```

Password specifications can be found here. The given password is not of importance and is only used for the individual Docker instance.

2. Add a user-secret from within the `src/Chirp.Web` folder, using the following command in your terminal:

```
dotnet user-secrets set "ConnectionStrings:AZURE_SQL_CONNECTIONSTRING" "Server=localhost,143
```

3. The database is configured using the dotnet-ef tool by running the following command from within the `src/Chirp.Infrastructure` folder:

```
dotnet ef database update --startup-project "../Chirp.Web"
```

2.3.4 Running *Chirp!*

To run *Chirp!*, use the following command from within the `src/Chirp.Web` folder, in your terminal:

```
dotnet run
```

2.4 How to run test suite locally

2.4.1 Running all tests

You need to have made *Chirp!* work locally, before proceeding to running the tests. Before the end-2-end (E2E) tests can be run, Playwright needs to be installed. Here is a guide to install Playwright:

1. In the terminal, run the command: `dotnet tool install --global Microsoft.Playwright.CLI`
2. Navigate to the directory: `/test/Chirp.Web.End2EndTests`
3. Run the command: `playwright install`

Once you have installed Playwright, the tests are now ready to run. Here follows a guide to running our test suite locally (Make sure you have the local *Chirp!* up and running):

1. In the terminal, navigate to the root directory (/Chirp) of the project
2. Run the command: `dotnet test`
3. A browser window will open, and at some point you will be requested to login to GitHub.

2.4.2 Running only unit & integration tests

If you do not want to run the E2E tests, they can be filtered out, allowing you to only run the unit and integration tests, with the following command:

```
dotnet test --filter "TestCategory!=End2End"
```

This command will produce the following warning: *No test matches the given testcase filter 'TestCategory!=End2End'*

This is because we use both NUnit and xUnit for our tests. We use NUnit for the E2E tests and xUnit for the rest. NUnit correctly ignores the E2E test, but xUnit will not recognize these tests as they are not part of the xUnit test suite, resulting in the warning being produced.

2.4.3 About our test suite

The test suite contains three different test projects:

1. Chirp.Infrastructure.Tests
2. Chirp.Web.End2EndTests
3. Chirp.Web.Tests

The Chirp.Infrastructure.Tests and Chirp.Web.Tests project contains unit-tests and integration tests for the Chirp.Infrastructure and Chirp.Web projects respectively.

The Chirp.Web.End2EndTests project tests the whole program, including UI and key functionality throughout *Chirp!*. This test project uses the NUnit test framework. The other projects use the xUnit testing tool.

3 Ethics

3.1 License

Our *Chirp!* application is licensed under Creative Commons Attribution Share Alike 4.0 International.

Permissions	Limitations	Conditions
Commercial use	Liability	License and copyright notice
Modification	Trademark use	Changes are stated
Distribution	Patent use	The license is kept
Private use	Warranty	

All dependencies in src use either the MIT License (e.g. Microsoft.EntityFrameworkCore.design, Microsoft.EntityFrameworkCore.SqlServer) or the Apache License 2.0 (e.g. FluentValidation), both of which are permissive and **allow for sublicensing**.

3.2 LLMs, ChatGPT, CoPilot, and others

Two of our five members used LLMs throughout the development. One of whom stopped their use, as they felt their perspective on the code was impaired in terms of possible solutions. The other primarily used it for Azure ADB2C and ASP.NET configuration.

We have used the following LLMs in some manner during the project.

We have used GitHub Copilot in some of the development process of the project. Copilot has mainly been utilized for generation of small code snippets. For example, this was done for manually accessing login and logout URLs, before we knew we had to import "taghelpers" in cshtml files to use ASP.NET specific features for links. We have forgotten to credit GitHub Copilot when writing the GetFollowing() and GetFollowers() methods. Although the final code remaining hardly resembles what was generated, co-authoring would still have been the appropriate thing to do.

GitHub Copilot's chat functionality within the workspace and ChatGPT were rarely used during the project. This was mainly used for researching purposes to get details about a concept or receive an explanation of some code examples from the internet. ChatGPT and Copilot chat primarily helped to gain an understanding of a subject, before implementing a feature.

The usage of LLMs did not have a substantial effect on the development speed. In some situations, the LLMs improved our understanding of the given issue, and sped up the development. In other situations, the response was misleading or imprecise, leading to inefficiency.