Chirp! Project Report

ITU BDSA 2024 Group `10`

Christoffer Grünberg gryn@itu.dk
Rasmus Rosenmejer Larsen rarl@itu.dk
Mathias Labori Olsen mlao@itu.dk
Alex Tilgreen Mogensen alext@itu.dk
Anthon Castillo Hertzum acah@itu.dk
Bryce Raj Karnikar brka@itu.dk

Contents

1 Design and Architecture of *Chirp!*

1.1 Domain model

The *Chirp!* domain model is setup around the Author class. Authors inherit traits for account management from IdentityUser. Authors are able to create Cheeps and interact with them with Likes or Comments. Each Author keeps a list of Likes and Comments enabling logging of which Authors have interacted with which Cheeps. Furthermore Authors are able to follow other Authors, storing a list Authors they follow and Authors that follow them.

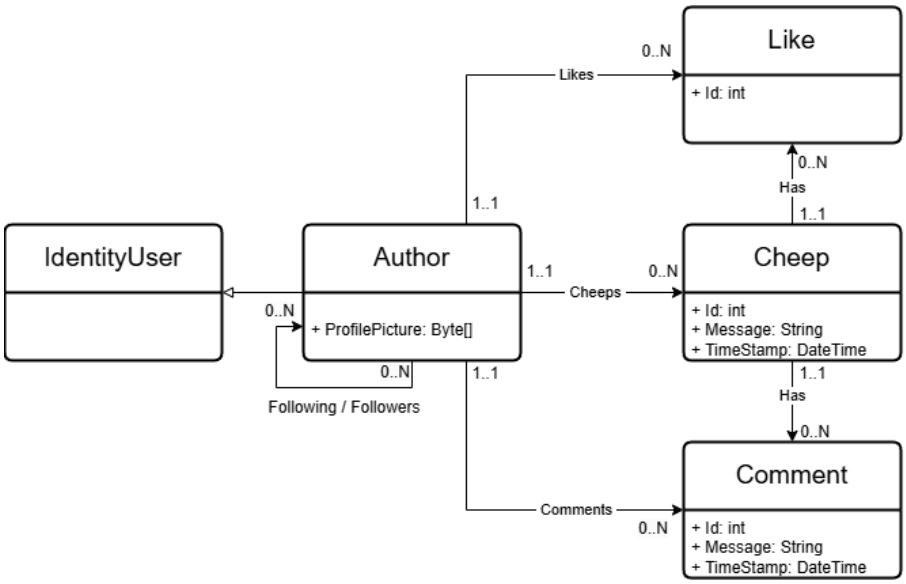

Figure 1: Illustration of the *Chirp!* data model as UML class diagram.

1.2 Architecture — In the small

Due to the application's size, each layer consists only of a single project, as highlighted in bold. **Chirp.Web** references **Chirp.Infrastructure**, which deviates from the Onion architecture, for two reasons:

1. **Program.cs** requires it to configure services.
2. ASP.NET Identity uses it for user registration and verification.

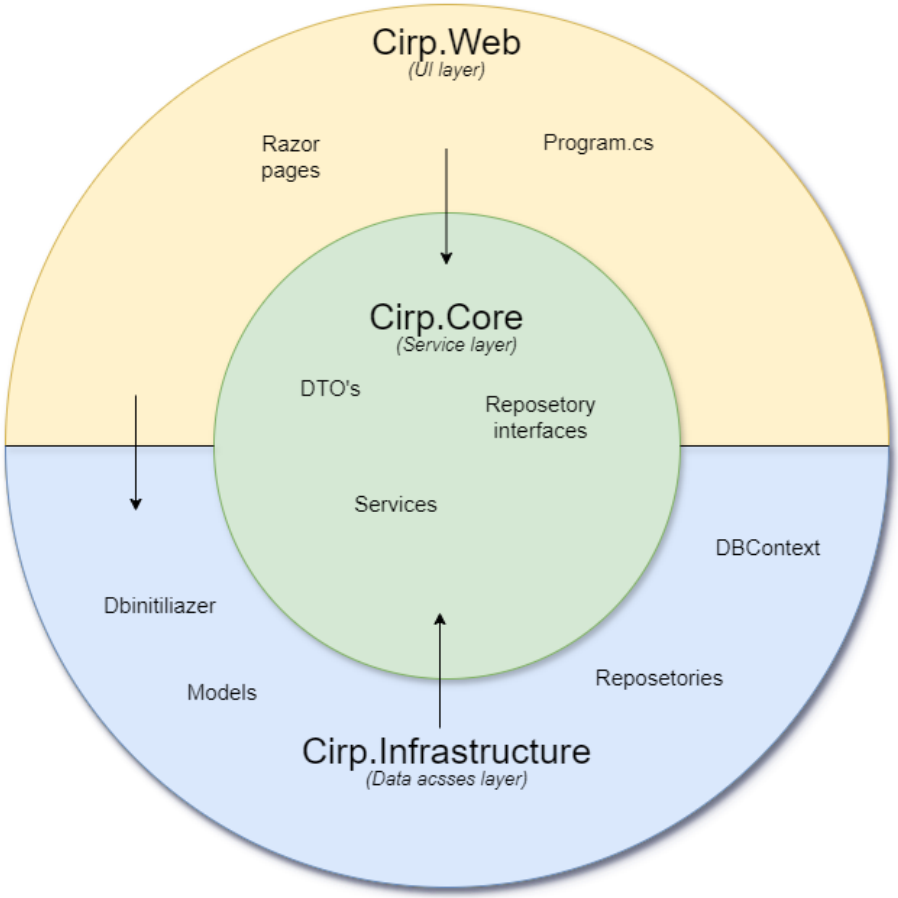

Figure 2: Illustration of the *Chirp!* program architecture.

1.3 Architecture of deployed application

The *Chirp!* application is deployed to the Microsoft Azure App Service as a complete component consisting of Chirp.Web for the GUI, Chirp.Infrastructure handling the domain model and repositories. The User connects to Chirp.Web through Azure. On read and write requests the Azure Web App will make calls to the deployed SQLite server. If users attempt to login or register with OAuth via github Chirp.Web will make calls to GitHub Authentication.

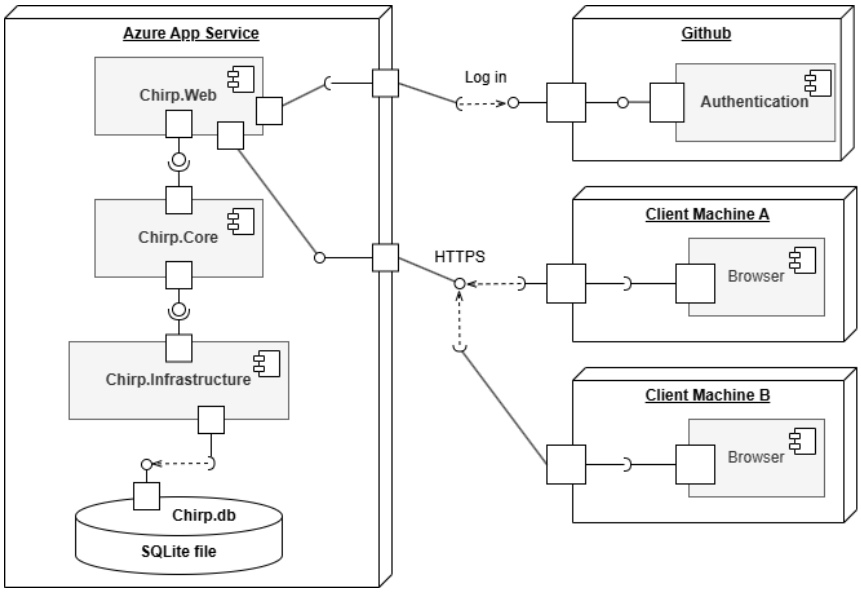

Figure 3: Illustration of the *Chirp!* deployment architecture of the application.

1.4 User activities

In order to increase the readability of the UserActivities diagram, the total diagram has been decomposed to show activities depending on whether the User is signed in or not.

The total diagram can be under *docs/images/UserActivitiesDiagram.png*

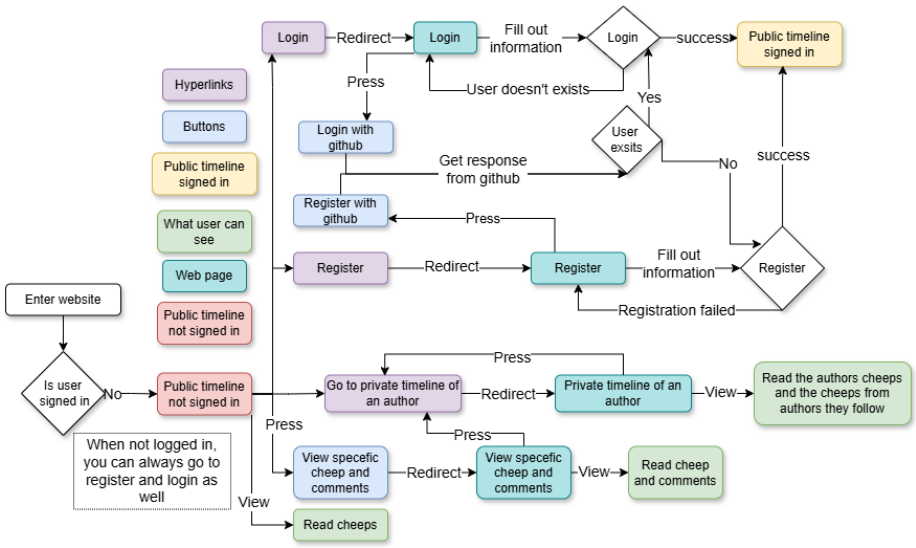

Figure 4: Illustration of the *Chirp!* functionality while signed out.

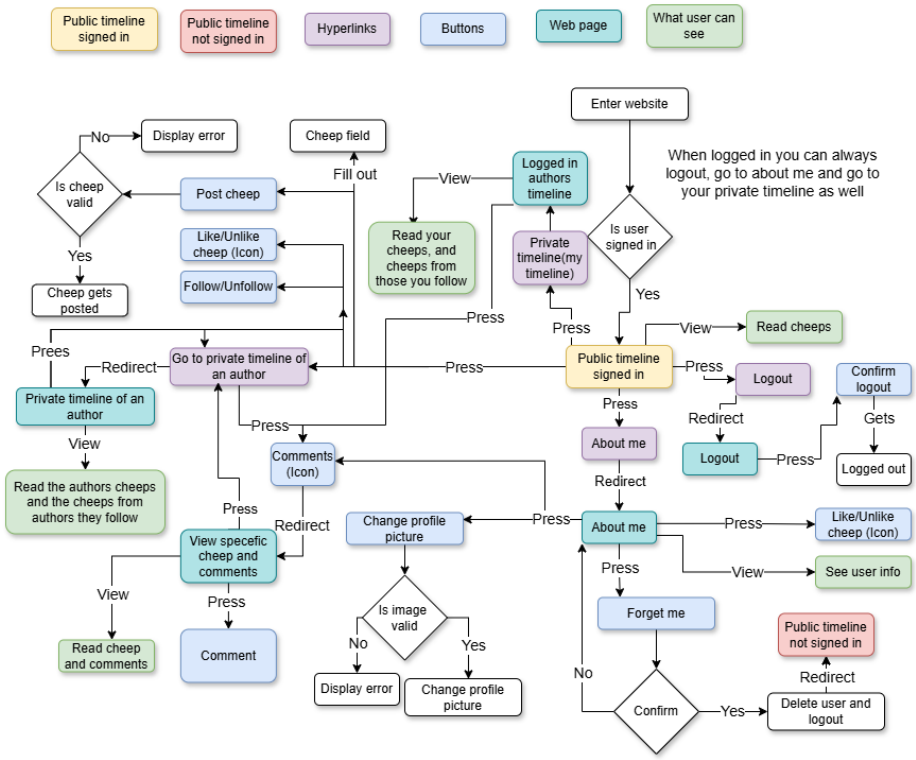

Figure 5: Illustration of the *Chirp!* functionality while signed in.

1.5 Sequence of functionality/calls trough *Chirp!*

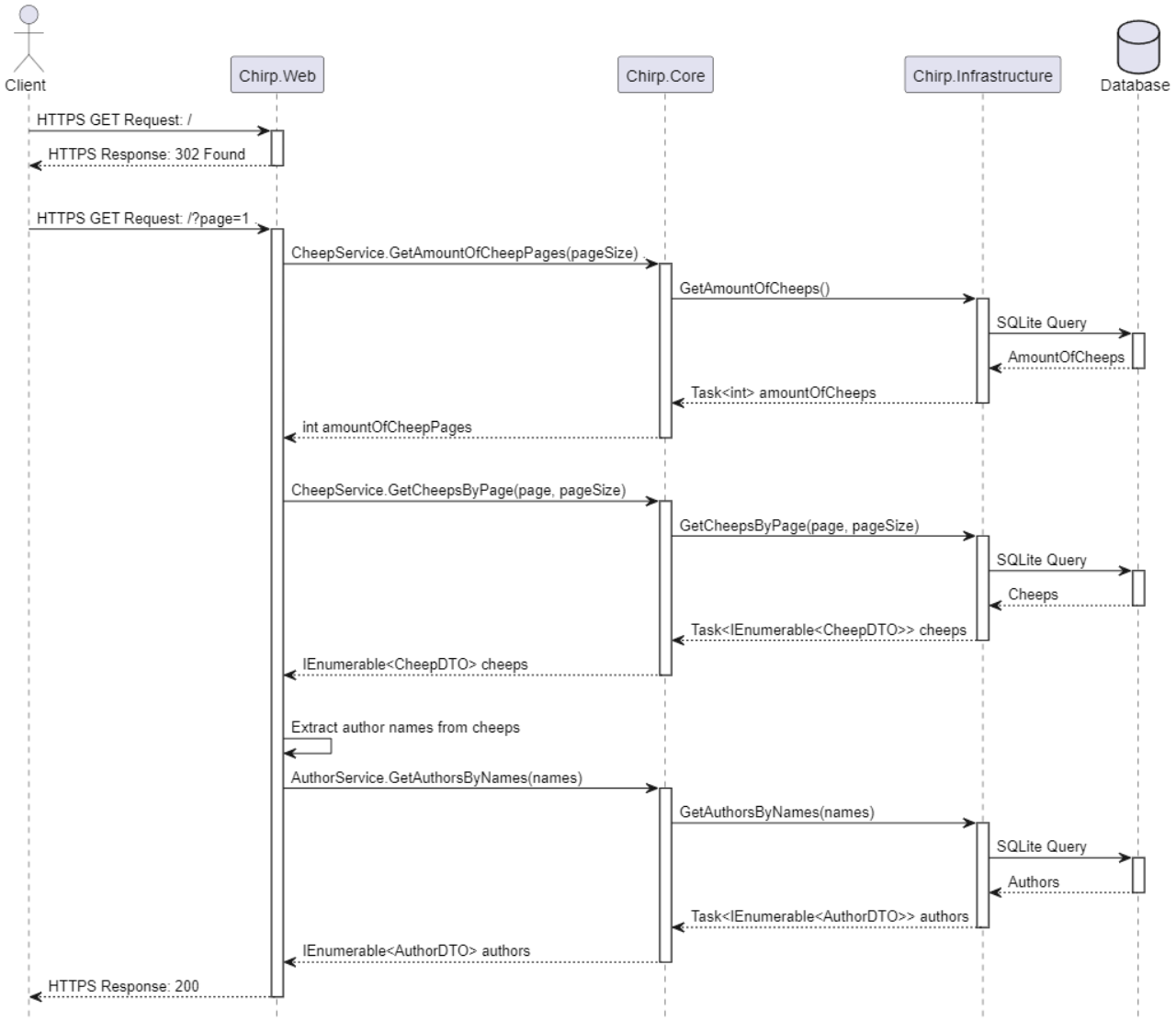

Figure 6: Sequence diagram of the flow of messages through the *Chirp!* application.

2 Process

2.1 Build, test, release, and deployment

The figure 5 below illustrates the workflows used for building and deploying the *Chirp!* application. The process stars, when pull request is merged into the main branch. The blue boxes represents workflows

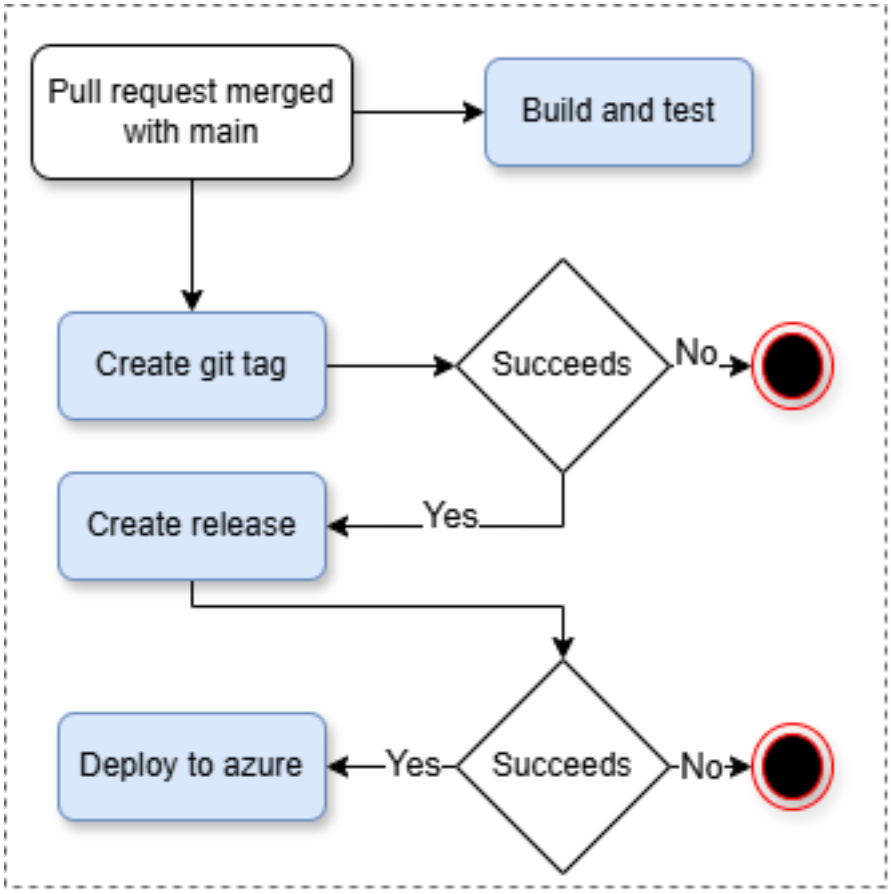

Figure 7: Illustration of github workflows for building and deploying the *Chirp!* application.

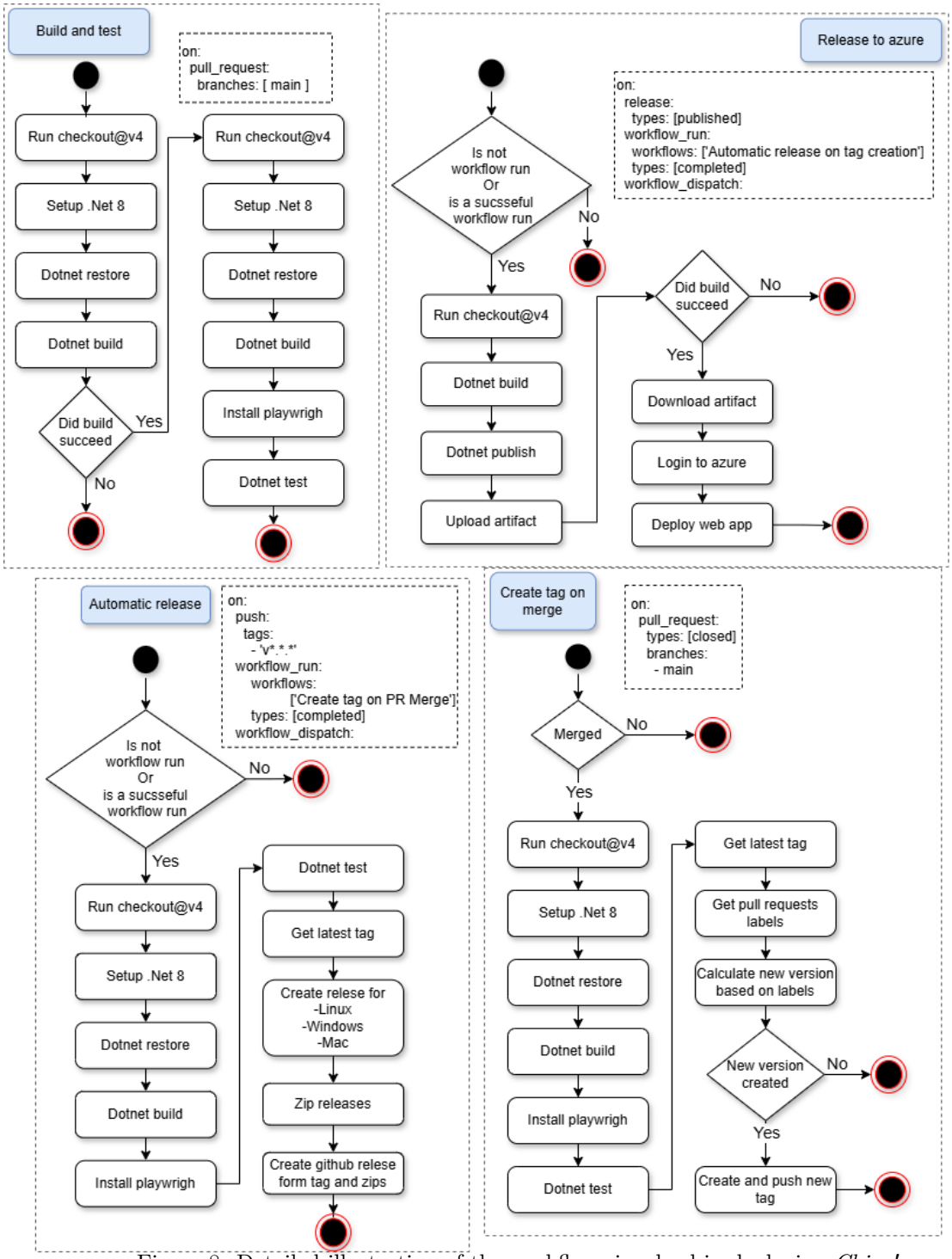

Figure 8: Detailed illustration of the workflows involved in deploying *Chirp!*.

2.2 Pull Requests

To help validate pull requests and help make sure only code, that lived up to the following,

- Contain no warnings
- Be able to build
- Have no failing test was pulled into main, the following workflow structure was setup.

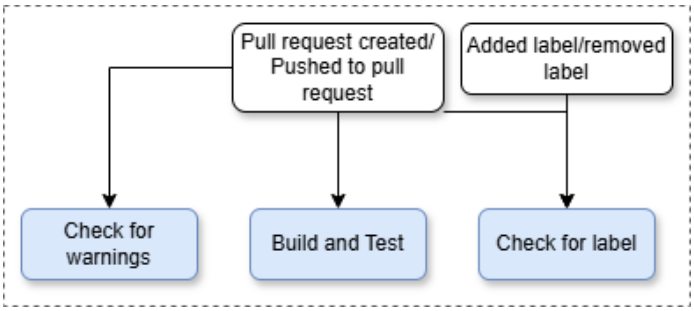

Figure 9: Illustration of github workflows for pull requests into the main branch of *Chirp!*

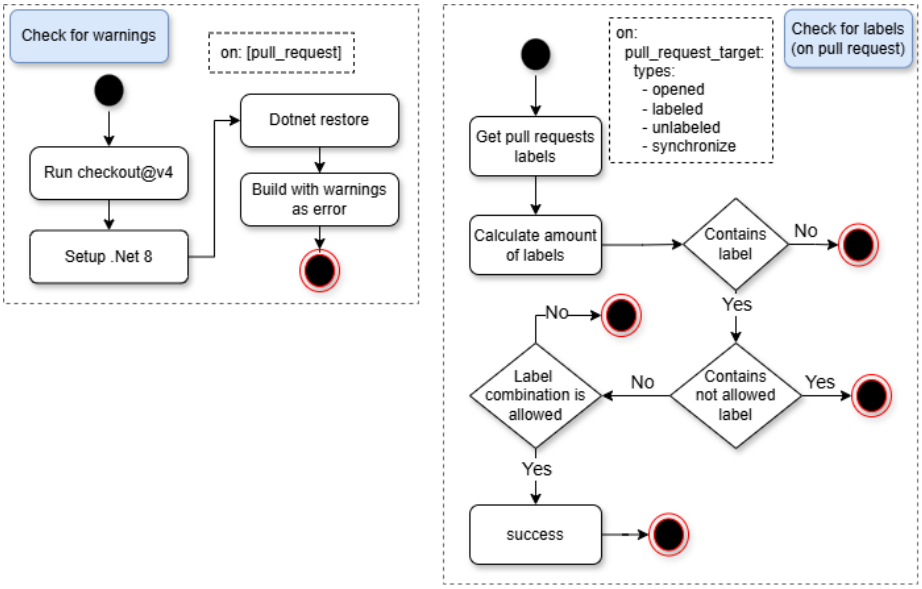

Figure 10: Workflow structure for pull requests into the main branch of *Chirp!*

2.3 Team work

2.3.1 Project Board

2.3.2 Unclosed Issues

Some issues still remain open in the Todo column, these are extra features that the group found interesting but did not get to implement within the time frame of the project work.
In order to better mimic the functionality of *X* (f.k.a *Twitter*), users should be able to leave comments directly on the timeline pages. This would be implemented by having a popup window appear, where users could leave comments, when clicking a Cheep. However getting this to work while handling and displaying message-format-errors proved to be an issue, and the feature was given an *Extra* tag and left open.
We also wanted to make a big refactor, which involved moving what database access we could to an API project. Since we use ASP.NET Identity for user registration and verification, a local database would still be required for the web project to store user information. The main reason for the API project is to decouple data access from the web application, making it easier to build additional features, such as a mobile app, by enabling shared data across projects. While a centralized database could achieve similar results, an API is more future-proof, as it abstracts the database layer, making the switching of the database, have no impact on the projects using the API.

For the Command-Line-Interface version of *Chirp!* an error with the end-to-end tests still exists. The tests pass when the database file contains the expected Cheep and the test is run on Windows. However the group was unable to make the test work in isolation from the actual database and cross-platform and thus the end-to-end-test branch remains open.

2.3.3 Issue Progression

The illustration below shows how the group worked with issues during the project. Steps highlighted in blue show issue creation, red boxes show the development process and green how issues are merged from the feature branch into main.

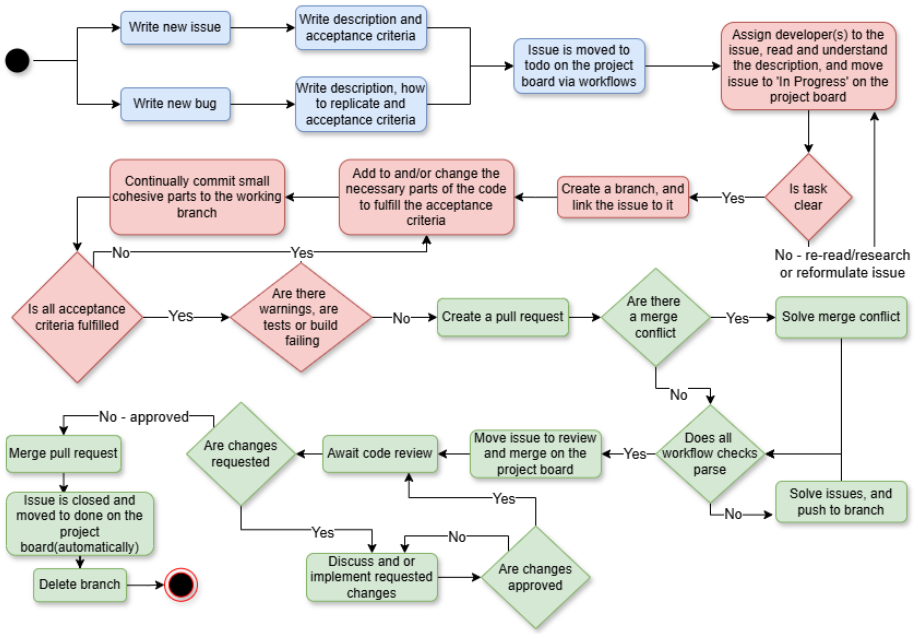

Figure 11: Illustration of the *Chirp!* issue progression from creation to merge.

2.4 How to make *Chirp!* work locally

First clone the repository to your machine with

```
git clone https://github.com/ITU-BDSA2024-GROUP10/Chirp.git
```

In order for the program to work you will need to configure the user-secrets. To do this navigate to */Chirp/Chirp*, in the terminal, and run:

Windows:

```
dotnet user-secrets init --project .\src\Chirp.Web\
dotnet user-secrets set "authentication:github:clientId" "Ov23lisGJEMdXORhzpDr"
--project .\src\Chirp.Web\

dotnet user-secrets set
"authentication:github:clientSecret" "a9229ceee8bb014070dc9abe892cf07d7aba4d0d"
--project .\src\Chirp.Web\
```

MacOs & Linux:

```
dotnet user-secrets init --project ./src/Chirp.Web/
dotnet user-secrets set "authentication:github:clientId" "Ov23lisGJEMdXORhzpDr"
--project ./src/Chirp.Web/
```

```
dotnet user-secrets set
"authentication:github:clientSecret" "a9229ceee8bb014070dc9abe892cf07d7aba4d0d"
--project ./src/Chirp.Web/
```

Next navigate to */Chirp/Chirp/src/Chirp.Web* and in your terminal do either:

```
dotnet watch
dotnet run
```

2.5 How to run test suite locally

In order to make run the UI-Tests make sure your system has playwright installed. Next, navigate to */Chirp/Chirp* and in your terminal do

```
dotnet test
```

2.6 Our test structure

We have three kinds of test

- Unit
- UI
- End to end

Since our services are essentially return statements calling our repository, we found integration tests of these to be of lesser valued, compared to the rest of the project. However, if the project continues, testing them would be beneficial to ensure functionality remains unchanged as the services evolve.

Most of the unit tests are on the repository methods since these are where almost all of the compute is being done.

Our UI tests are set up quite general, we don't mock anything, just use an in memory database, and only validate if the UI behave as expected. You could, in addition to these, have done some tests where you isolate the UI more, e.g., by mocking the behavior of the used service methods. But we prioritized other tasks given the application's size. We have some end to end test, but could probably use some more, since our UI test are so general, they act some what as end to end tests.

3 Ethics

3.1 License

This program is licensed with the GNU GENERAL PUBLIC LICENSE Version 3

3.2 LLMs, ChatGPT, CoPilot, and others

3.2.1 CoPilot

CoPilot have been used doing the development of this project. It's a great tool for speeding up development, as it's quicker to read through the code it recommends than to write it. Not everything it recommends is usable or as we want it, but it can also help when learning a new language or framework, to introduce new methods and structure.

3.2.2 ChatGPT

ChatGPT was used primarily for the three following things.

- Understanding and debugging error messages
- Writing some HTML and CSS code
- Understand and discussing

Understanding and debugging error messages. This can be very helpful since error messages can be very long and contain a lot of information, which is ideal for LLM's. Sometimes they can also be harder to understand if you don't have a lot of knowledge of the framework you are using.

Writing some HTML and CSS code. Since HTML and CSS is time consuming we used ChatGPT to help write some of the UI code, that could be fine tuned by hand.

Understand and discussing code. If you don't have other people around, this can be very helpful, especially if you find some code online, whether it's from a stack overflow post, or if it's documentation. When learning a new language or framework, this can especially be helpful, since you don't know a lot of the tricks yet.

Overall, the use of LLM's speed-up our development process, and helped us get a better understanding of c# and dot net. While using LLM's can help speed up debugging and development processes, it is still important to learn how to work independent of AI-assistance. The data centers running the models also consume large amount of energy, and as a developer you need to be conscious of the impact of this technology.