Chirp! Project Report

ITU BDSA 2023 Group 25

Silas Wolff siwo@itu.dk Sebastian Blok segb@itu.dk
Karl Gustav Løhr kagl@itu.dk Adam Nørgård Aabye aaab@itu.dk
Atila Arianpour atia@itu.dk

1 Design and Architecture of *Chirp!*

1.1 Domain model

Our domain model is built around the core concept of an Author being able to Cheep, React to Cheeps and Follow other Authors. This is central to the Chirp! application's functionality. An Author represents a user of the application, encapsulating their identity and interactions within the system. The UML class diagram model the key entities that make up our application.

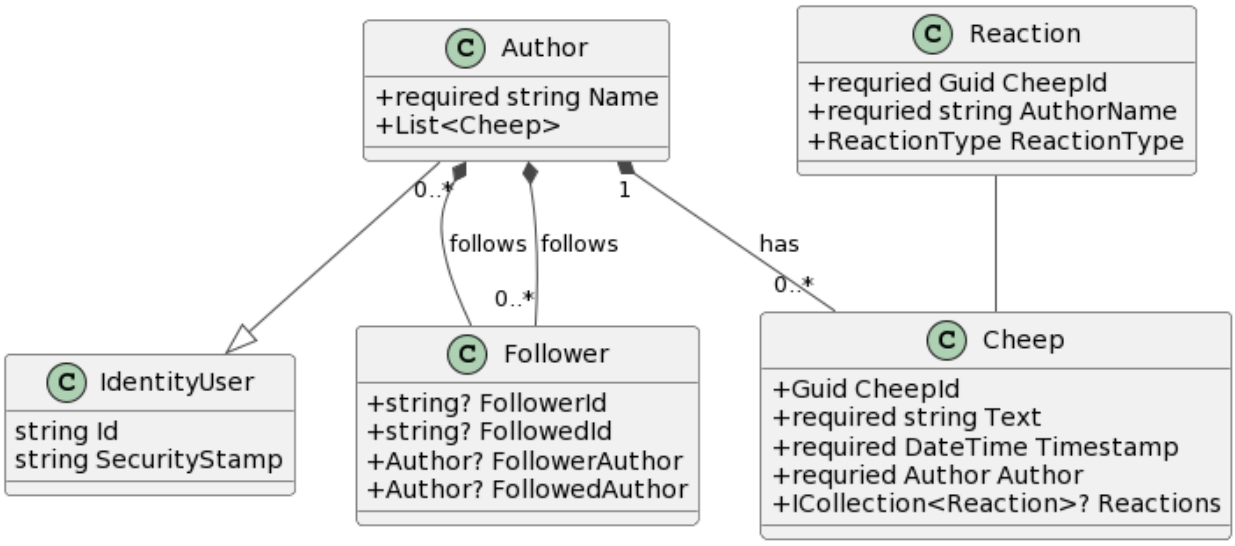

Figure 1: Domain Model

1.1.1 Author

The Author entity is an extension of the ASP.NET Identity's IdentityUser, inheriting features essential for authentication and authorization.

Each Author has a collection of Cheep entities, representing the messages, or posts, that the author creates within the application. This one-to-many relationship is depicted by a composition association, emphasizing that Cheeps are intrinsic to their Author and cannot exist independently.

In addition to creating cheeps, Authors can 'follow' other Authors. This is represented by a many-to-many referencing association, indicating that an Author can follow multiple other Authors and also be followed by

multiple others. This relationship captures the essence of the application's social interaction capabilities.

1.1.2 Cheep

A Cheep is essentially a message, or a singular piece of communication, created by an Author. Each Cheep is uniquely identified by a Guid and contains the message text along with a timestamp of its creation. Cheeps is in a one-to-many composition with Authors, meaning that, Authors can have many cheeps, but cheeps must have exactly one author. Additionally, they have a strong life-cycle dependency.

1.1.3 Following

Chirp allows its users to follow and unfollow each other. The follow entity is used to support this functionality, by storing the id of the one who followed and the id of the one getting a follower. In the database, there is a table called Followers, where a new tuple is inserted. The tuple is made up with four columns, FollowerId, FollowedId, FollowerAuthorId and FollowedAthorId. The Followed being the one who followed another user, and the Follower being the one who got a new follower. The primary key is a unique combination of the FollowerId and FollowedId. This stops a user f from following the same user multiple times, but lets two users follow each other. Optimally, FollowerAuthorId and FollowedAuthorId could be deleted, because it tracks the same info.

1.1.4 Reactions

A reaction is the entity that refers to the interactive engagement that users can express in response to a cheep. These reactions are represented by a "thumps up", "Thumps down" and "Skull" emoji that turns pink if pressed. For each cheep a number of reactions will be displayed in the application for each reactiontype. Each reaction is uniquely identified by a Guid representing the cheep, a string which represents the user who has reacted and a reactiontype representing which type of reaction it is. Reactions is in one-to-many relationships with both authors and cheeps meaning that both authors and cheeps can have many reactions but each reaction is uniquely related to one author and one cheep.

```
var cheep = _databaseContext.Cheeps
    .Include(c => c.Reactions)
    .FirstOrDefault(c => c.CheepId == cheepId);
var author = _databaseContext.Users.SingleOrDefaultAsync(a => a.Name == authorName);
```

The reactions functionality is controlled by its interface IReactionRepository with it's three mandatory methods, HasUserReacted, ReactionOnCheep and GetReactionAmount. ReactionOnCheep is an asynchronous task that takes a reactiontype, cheepid and username. The task starts by checking for the specific username and cheepId in the database and if any of those two are null the method returns an exception.

```
if (cheep != null || author != null)
{
    var reaction = new Reaction
    {
        CheepId = cheepId,
        AuthorName = authorName,
        ReactionType = reactionType
    };

    var currentReaction = _databaseContext.Reactions.FirstOrDefault(
        r => r.CheepId == reaction.CheepId
            && r.AuthorName == reaction.AuthorName
            && r.ReactionType == reaction.ReactionType
    );

    if (currentReaction is null)
    {
        _databaseContext.Reactions.Add(reaction);
    }
    else
    {
        _databaseContext.Reactions.Remove(reaction);
    }
    await _databaseContext.SaveChangesAsync();
}
else
{
    throw new NullReferenceException("Cheep not found");

}
```

If not an instance of the object reaction will be created. Along side the task instantiate a variable "currentReaction" and check if the cheep already has a reaction from the user in the database. Based on the outcome of the check the system will either add the reaction to the database or remove it from database. This is because the ReactionOnCheep task handles both cases where a user wants to react on the cheep(add a reaction to the database) or remove the reactions from the cheep(remove the reaction in the database). Additionly if a user wants to change reactiontype, then the old reaction is removed and another one created.

The asynchronous task "HasUserReacted" is responsible for letting the system know if a user has already reacted on a cheep. It takes a cheepId and username as arguments. Firstly it checks whether or not a user

is to be found in the database and if that is the case an exception is thrown. In the case where a user is found the task checks if the database contains a reaction sat on the cheep and if the user is the owner of that reaction. The result will be a boolean which depends on the user interaction with the cheep.

The last task is responsible for letting the system know how many reactions a cheep has. This is simply done be checking the database how many reactions are related to the cheepId the task takes as argument. These get put in a list where it asynchronously returns the total number of reactions.

1.2 Architecture — In the small

Our application is separated into 3 main layers, that are common for the onion architecture

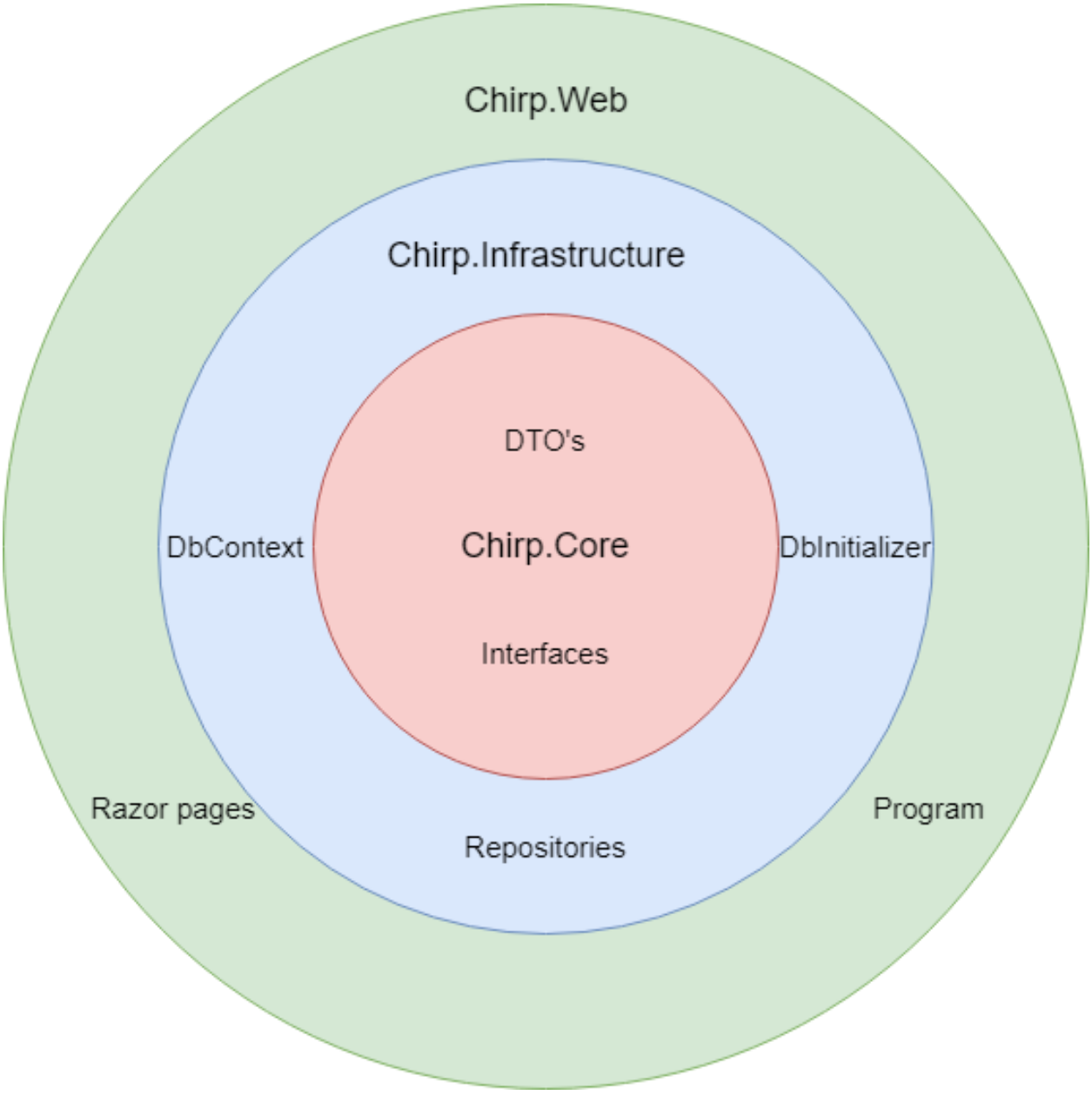

Figure 2: Onion model

Layers: * core * infrastructure * web

Dependencies: * identity -> infrastructure * ef core -> infrastructure * core -> infrastructure * OAuth -> web * core -> web * infrastructure -> web

1.3 Architecture of deployed application

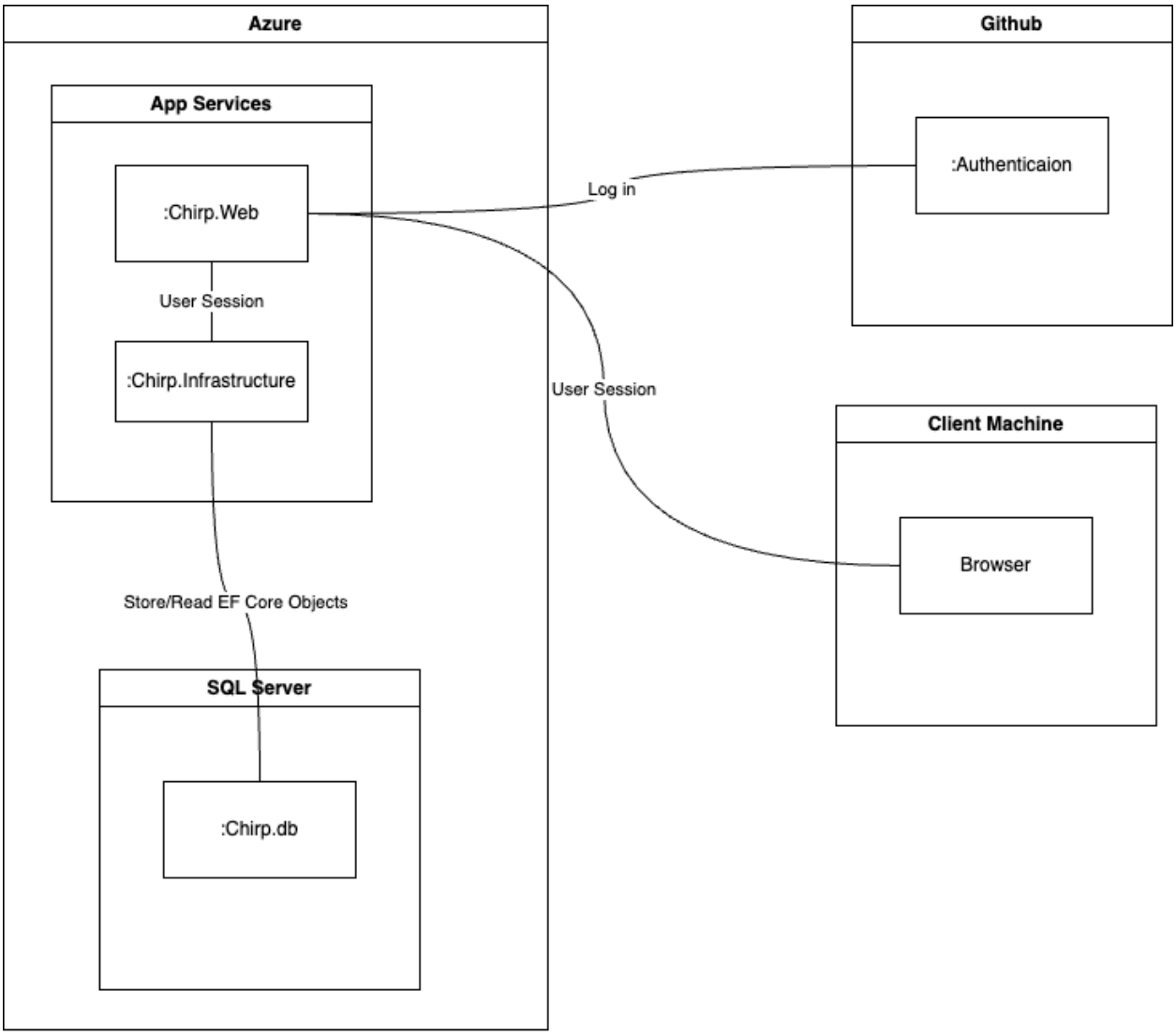

Figure 3: Deployment diagram

Chirp! uses Azure App Service to host its website. Here Chirp.web handles the frontend of Chirp! and Infrastructure handles the database context and the repositories. As shown on the image above, a client can access Chirp.Web through azurewebsites and thereby be authorized and authenticated by github, before unlocking all features.

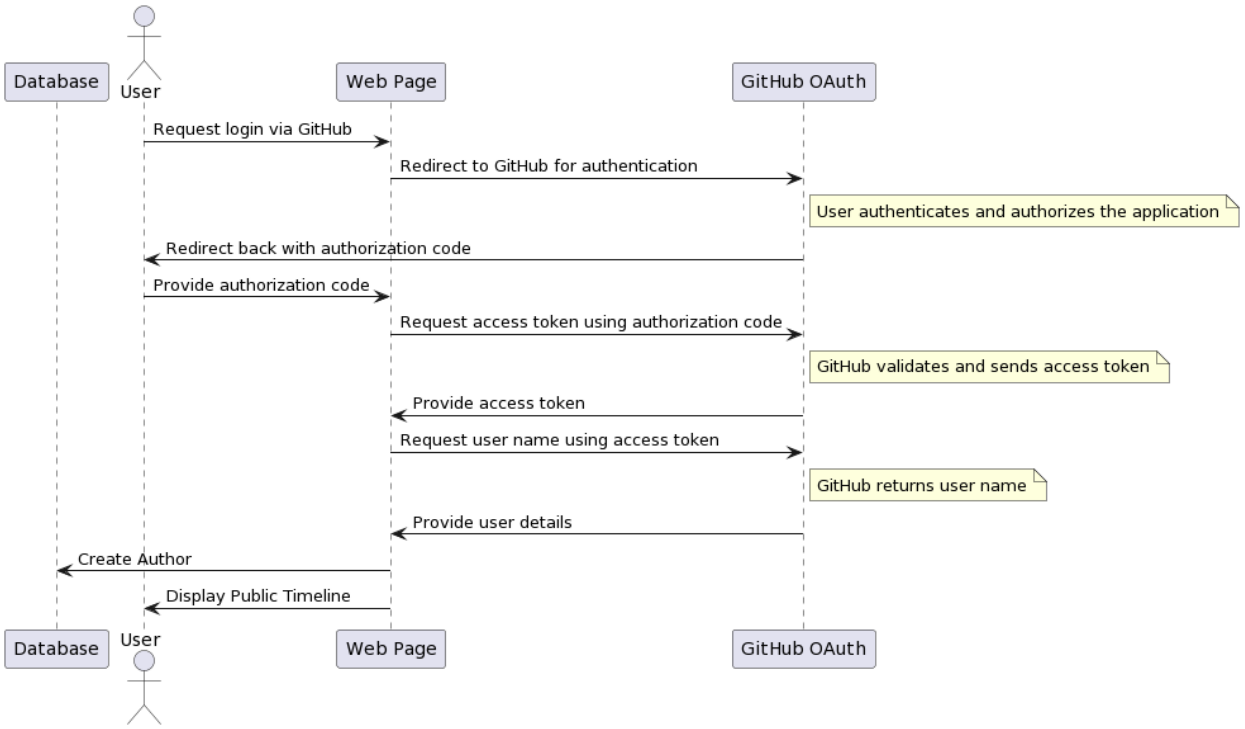

Figure 4: Sequence diagram

As shown on the image above, Chirp! uses a third party external login provider (GitHub), to authenticate and authorize the user. Because Chirp! needs a name claim from github, to create an Author, a sequence of request are made.

1.4 User activities

User Journey for Unauthenticated Users

The unauthenticated user is able to browse all cheeps on the public timeline. That is, they can only view them. Since they aren't authenticated they have no personal timeline and no 'About me' page to visit. Besides viewing the public timeline they have the ability to log in. In the log in page they can be authenticated through github.

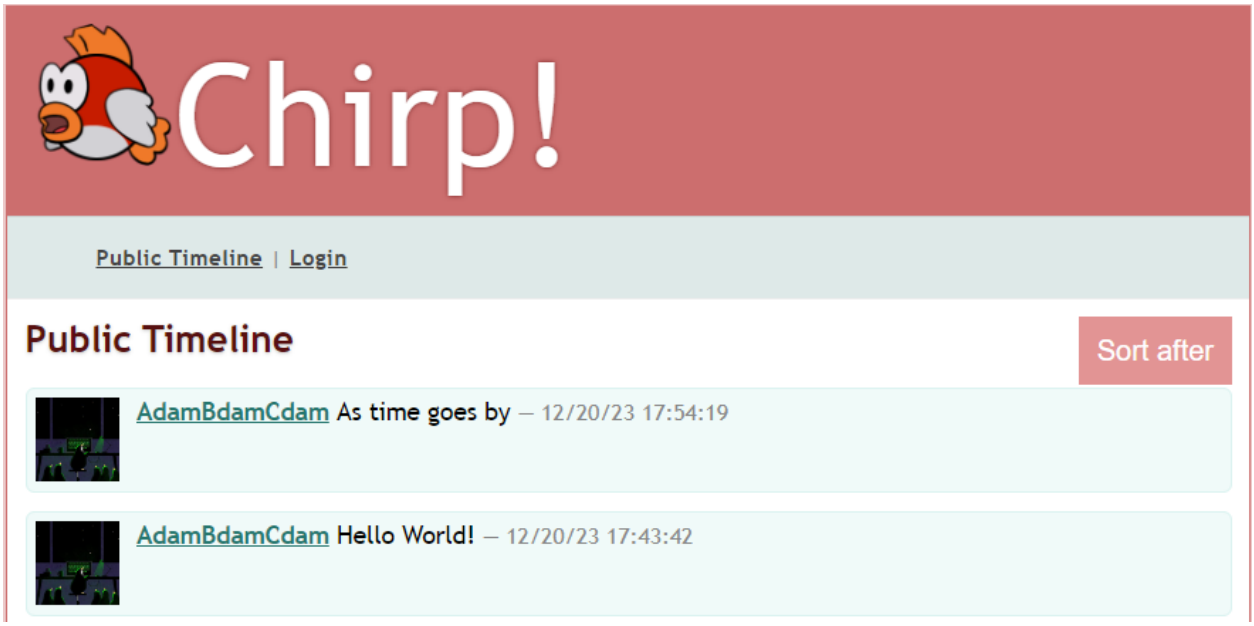

Figure 5: The public timeline for an unauthenticated user

Image 7 shows the top of the webpage, that the unauthenticated user sees. Under the logo, the user is able to navigate the website by clicking on the links 'Public Timeline' and 'Login'. The public timeline serves as the frontpage, so the 'Public Timeline' link is meant to return to the front page, when the user is somewhere else on the website.

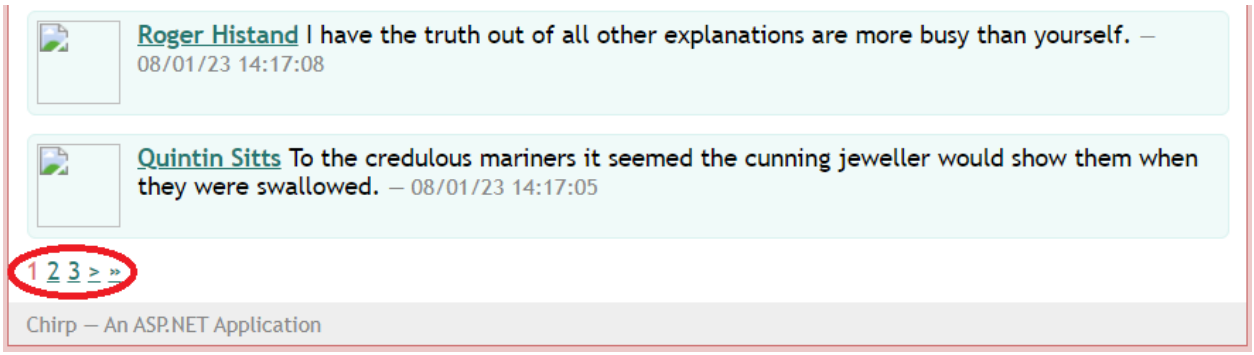

Figure 6: The bottom of the public timeline for the unauthenticated user

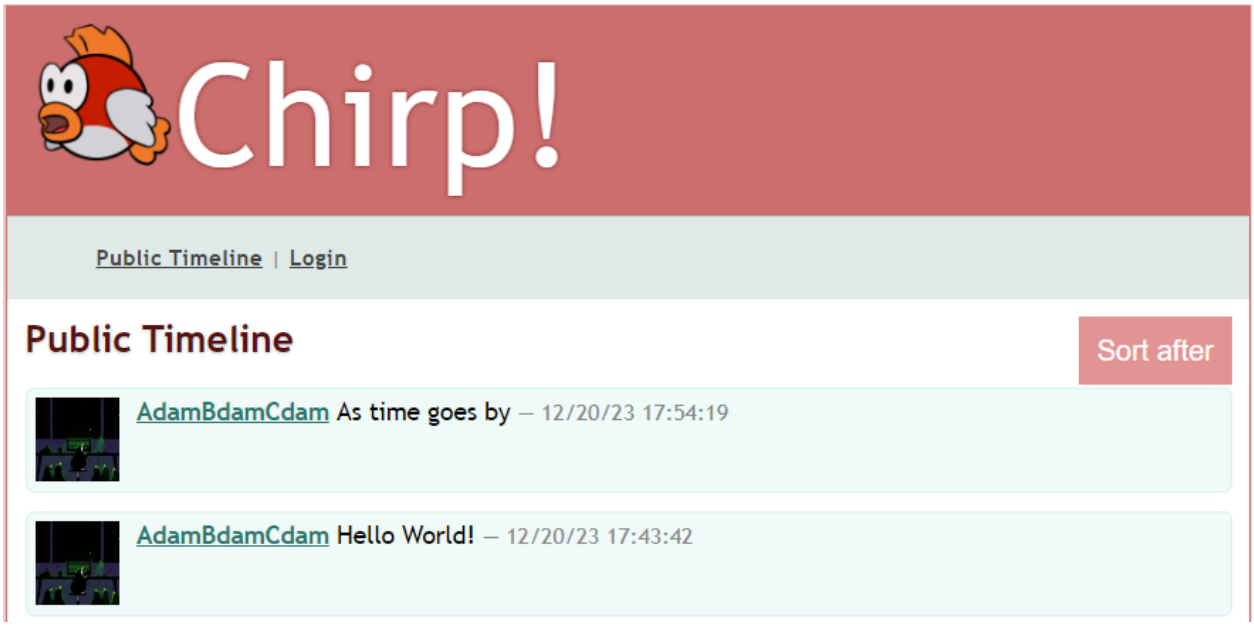

Figure 7: User activities3

Image 8 shows the bottom of the frontpage for the unauthenticated user. Here the user can use the numbers and arrows highlighted in the image, to scroll through the different pages of cheeps. There are a maximum of 32 cheeps per page.

The user journey of the unauthenticated user is shown in a sequence diagram in figure 10 below.

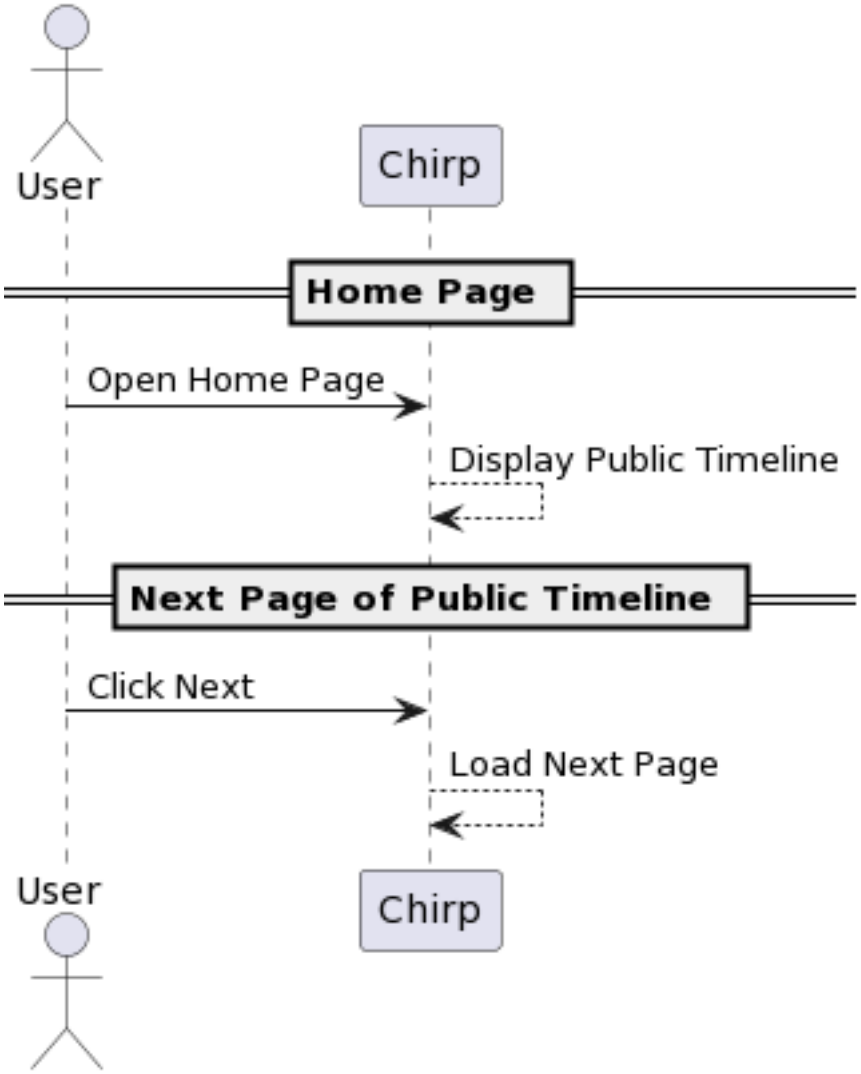

Figure 8: User activities4

Diagram 8. Sequence diagram of the user journey of the unauthenticated user

As illustrated by the above diagram, the options of the unauthenticated user are very limited, as most of the functionalities of the applications are made specifically for authenticated users.

Once the user is authenticated, a lot more functionalities become available compared to the unauthenticated user. These functionalities include the ability to post cheeps, delete cheeps, adding a reaction on other user's posts, following other users, viewing 'My timeline', viewing other users' timelines and deleting all personal information related to the user as well as deleting all cheeps posted by the user. All this functionality is visualized it the sitemap.

Posting, reacting, following and deleting - A user story

The authenticated user is met with a front page that can be seen in image XX.

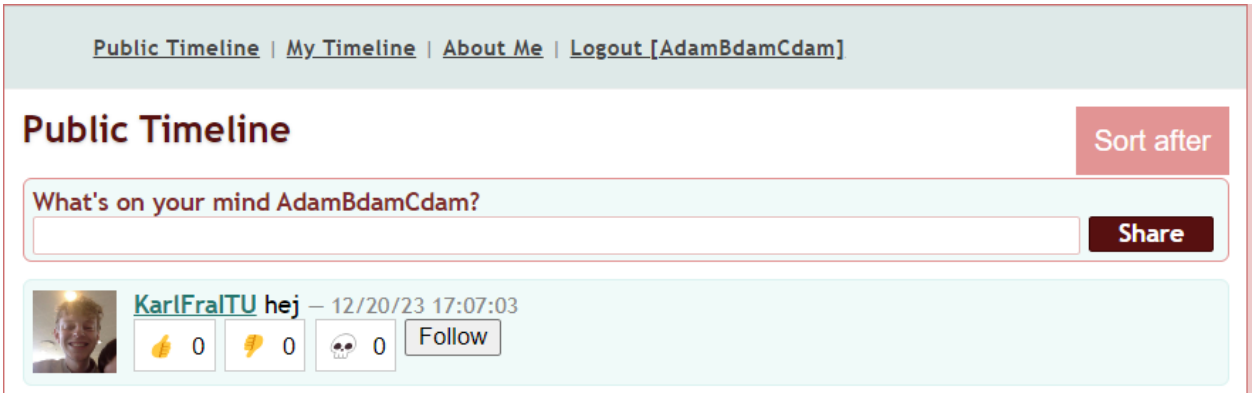

Figure 9: User activities5

A typical use application like this, entails posting a cheep. This is done from the front page, just under the headline. The user will write a cheep in the input field, and press 'enter' or press the 'Share' button, to post it.

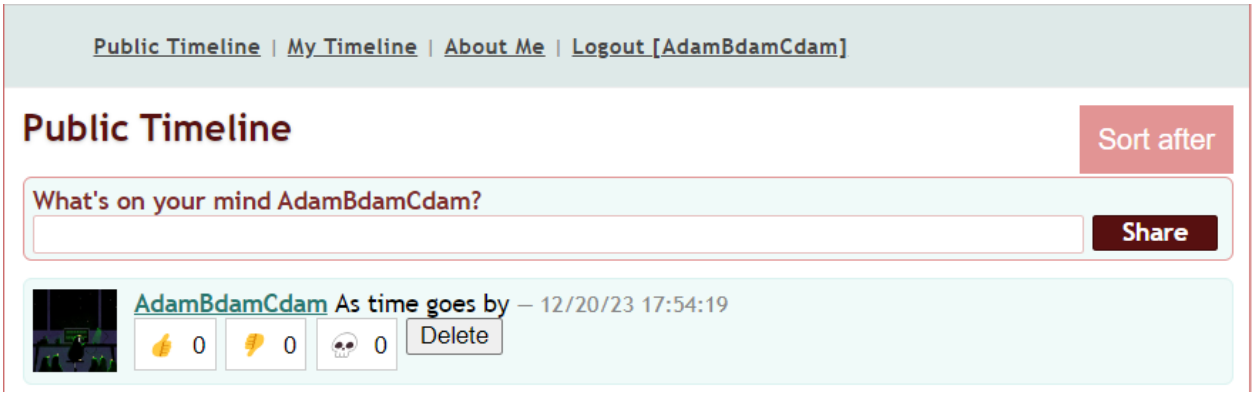

Figure 10: User activities6

After posting the cheep the user wants to react to another user's cheep. This is done by pressing one of the react buttons, that can be seen on Image XX, on the cheep that is there. The user can either give a 'thumbs up', a 'thumbs down' or react with a skull. The user can only react in one way, meaning they can not leave both a 'thumbs up' and a 'thumbs down' on the same post. After reacting to a post the chosen reaction will be highlighted with the color pink, as shown in image XXX below.

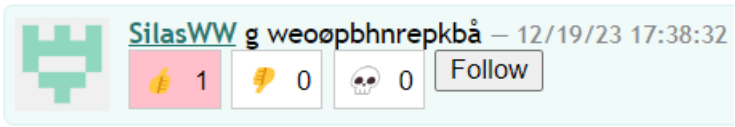

Figure 11: User activities7

After reacting to the cheep, the user now wants to follow the user that posted it. This is easily done by pressing the 'Follow' button, which is seen to the right of the reactions. After pressing the 'Follow' button it changes to 'Unfollow' as seen image XXXX below.

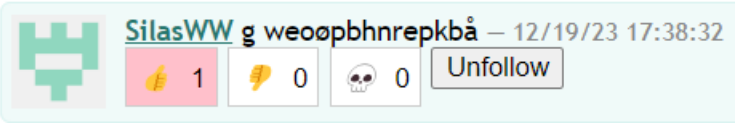

Figure 12: User activities8

The final act of the user will be deleting the cheep they posting in the beginning. On cheeps posted by the user, a 'delete' button can be found where the 'Follow' button would be, if it was a cheep of another user. Clicking the 'delete' button result in the cheep being deleted.

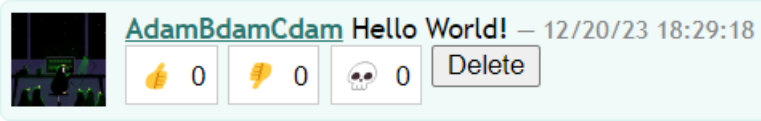

Figure 13: User activities9

'My Timeline', other user's timeline and deleting profile - a user story

The user starts their journey by viewing 'My Timeline'. The link to this page is found in the navigation bar. Here the user can see the cheeps they have posted, and the cheeps of the users they are following.

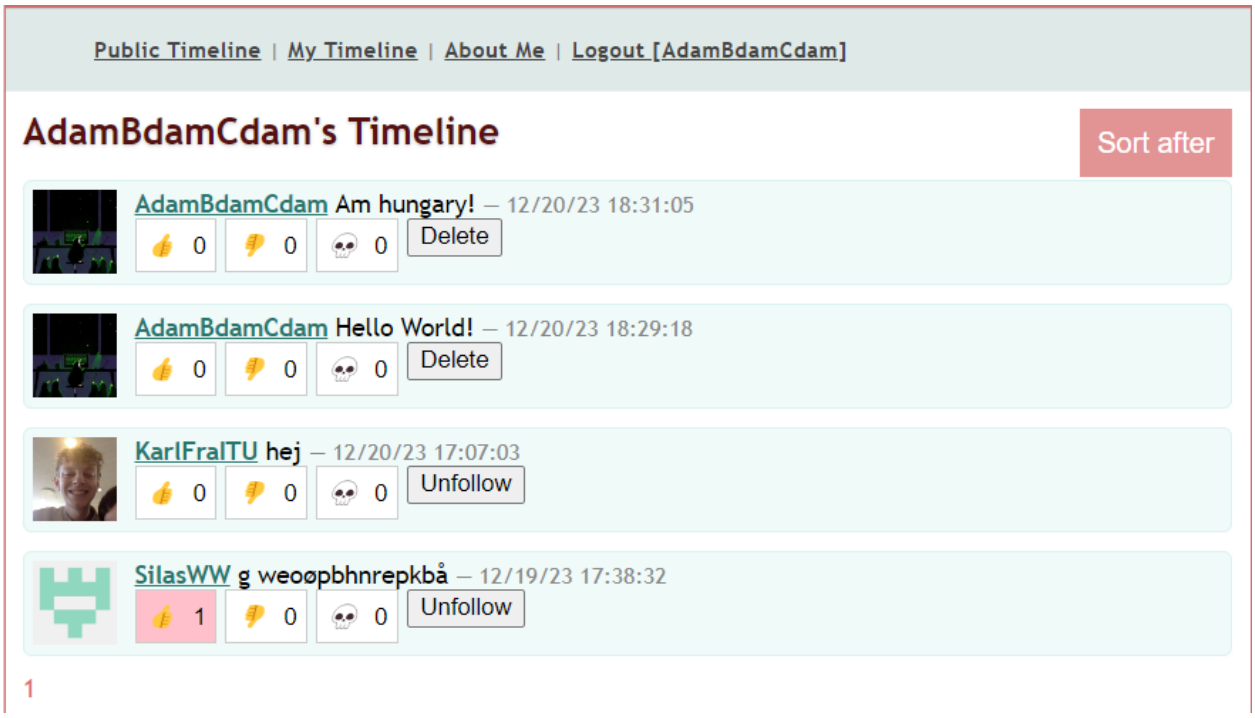

Figure 14: User activities10

After viewing the 'My Timeline' page, the user wants to see the timeline of another user, so they click on the username of the user who's timeline they want to view.

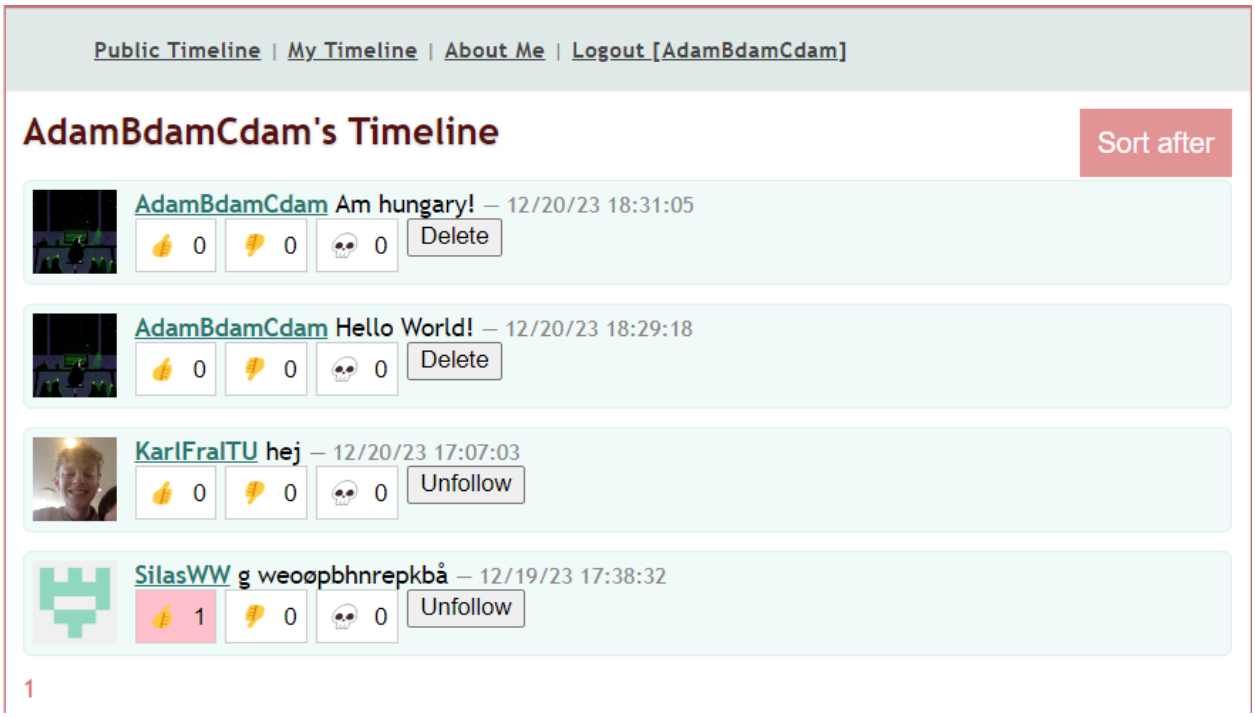

Figure 15: User activities11

The final act of the user is to delete their profile. They do this by going to the 'About Me' page. This page is seen in image X.

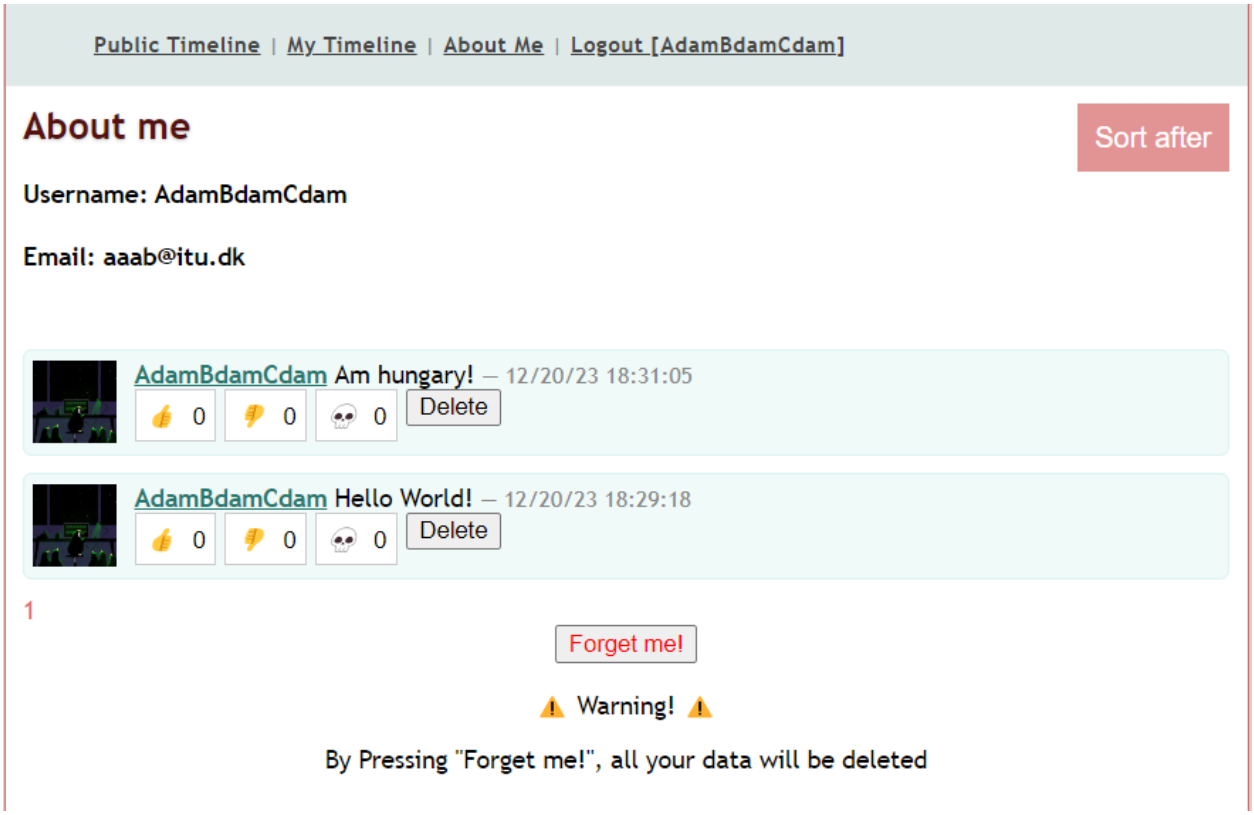

Figure 16: User activities12

On the 'About Me' page the user, can see information about themselves, as well as see the cheeps they have posted. A button with red text saying 'Forget Me!' can be seen below the cheeps. By pressing this button, all the user's information will be deleted, and their cheeps as well. All reaction they have left on other users' posts will be delete and their following of other users will also be deleted. Lastly the user will be unauthenticated and redirected to the 'Public Timeline'.

1.5 Sequence of functionality/calls trough *Chirp!*

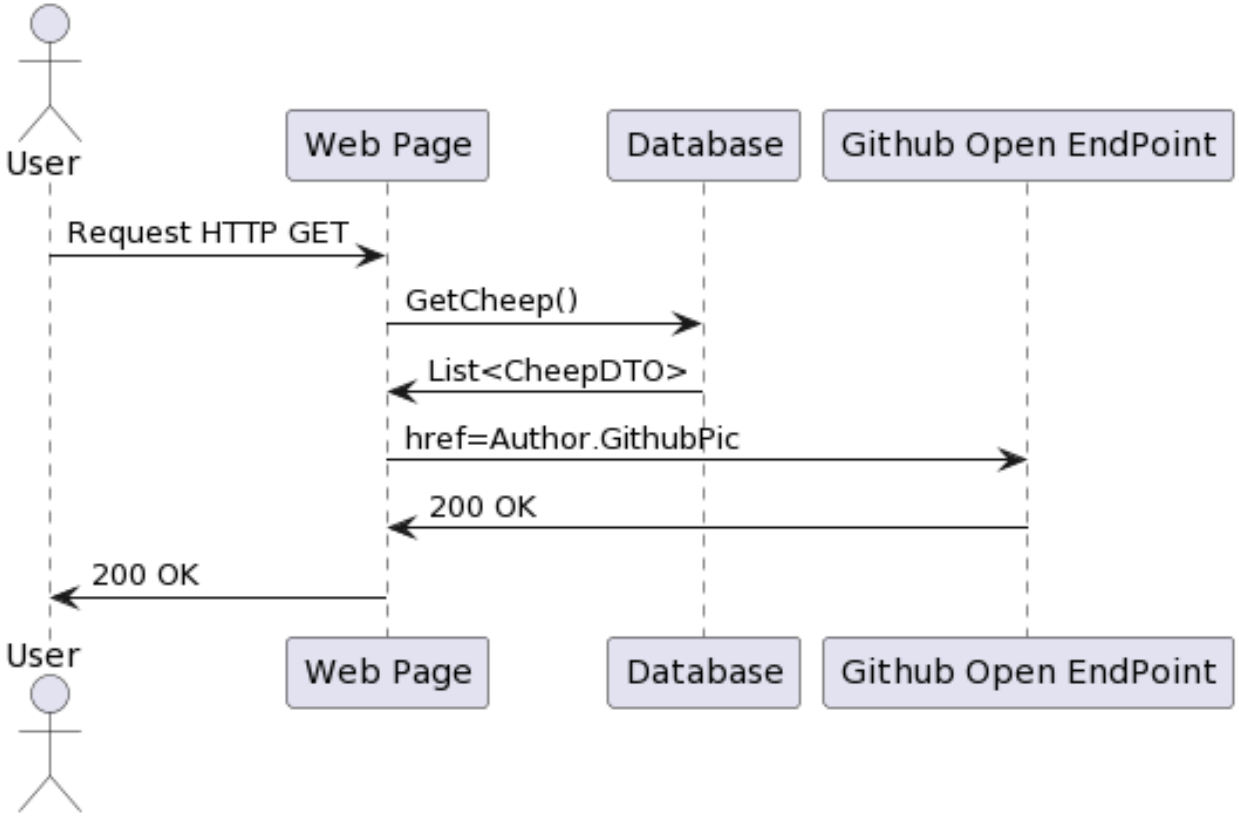

Figure 17: sequence_of_functionality

The UML-sequence-diagram above showcases the sequence of actions between User, Web page, Database and a Github Open Endpoint. When a user (client) tries to access the webpage, their computer sends a HTTP GET-request, and when the Azure Web page receives the request, it calls the c# method GetCheep, which returns a list of CheepDTO's (Cheep Data Transfer Objects). Then the Web Page will access an open-end github endpoint made up by www.GitHub.com/.png, where the NameOfAuthor is gotten from the CheepDTO. This returns the image and 200 OK. At last, the Web Page formats and displays the cheeps.

2 Process

2.1 Build, test, release, and deployment

We use Github Actions to automate the build, test, release and deployment process of our executables and website. This pipeline is centered around two workflows, one for the executables, and one for the website.

2.1.1 Publishing workflow

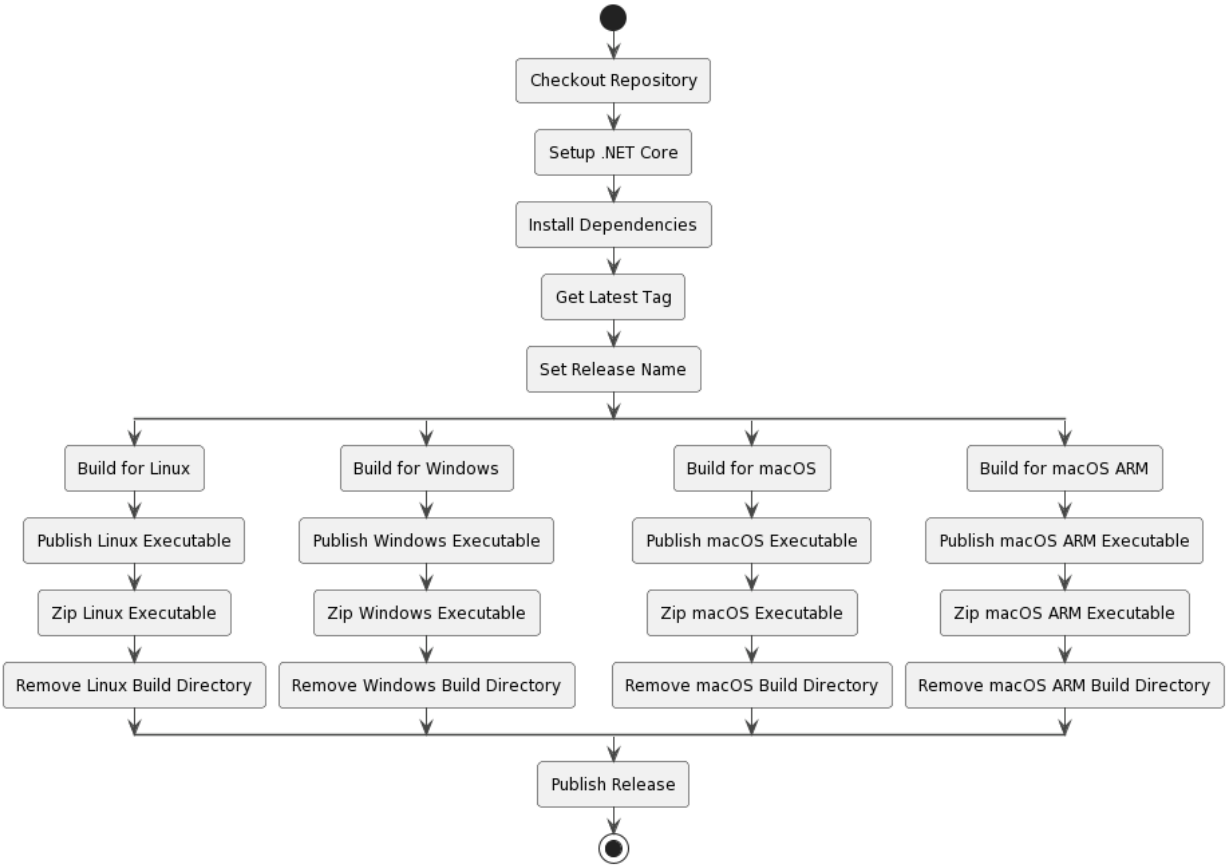

Figure 18: Activity Diagram for Publishing

The publishing workflow focuses on creating executable artifacts for various platforms, including Linux, Windows, macOS, and macOS ARM. This workflow is triggered by pushing specific version tags, reflecting our versioning strategy. We apply the semver strategy.

After the standard initial steps of checking out the repository and setting up the .NET Core environment, the workflow splits into parallel tasks for each platform. Each branch involves building the application, publishing the executable, zipping the file, and removing the build directory. This parallel structure allows for efficient and simultaneous preparation of executables for different platforms.

The final step is publishing the release on GitHub, attaching all the zipped executables. This process not only automates the release creation but also ensures that our application is readily available for a wide range of platforms, enhancing its accessibility to users.

2.1.2 Deployment workflow

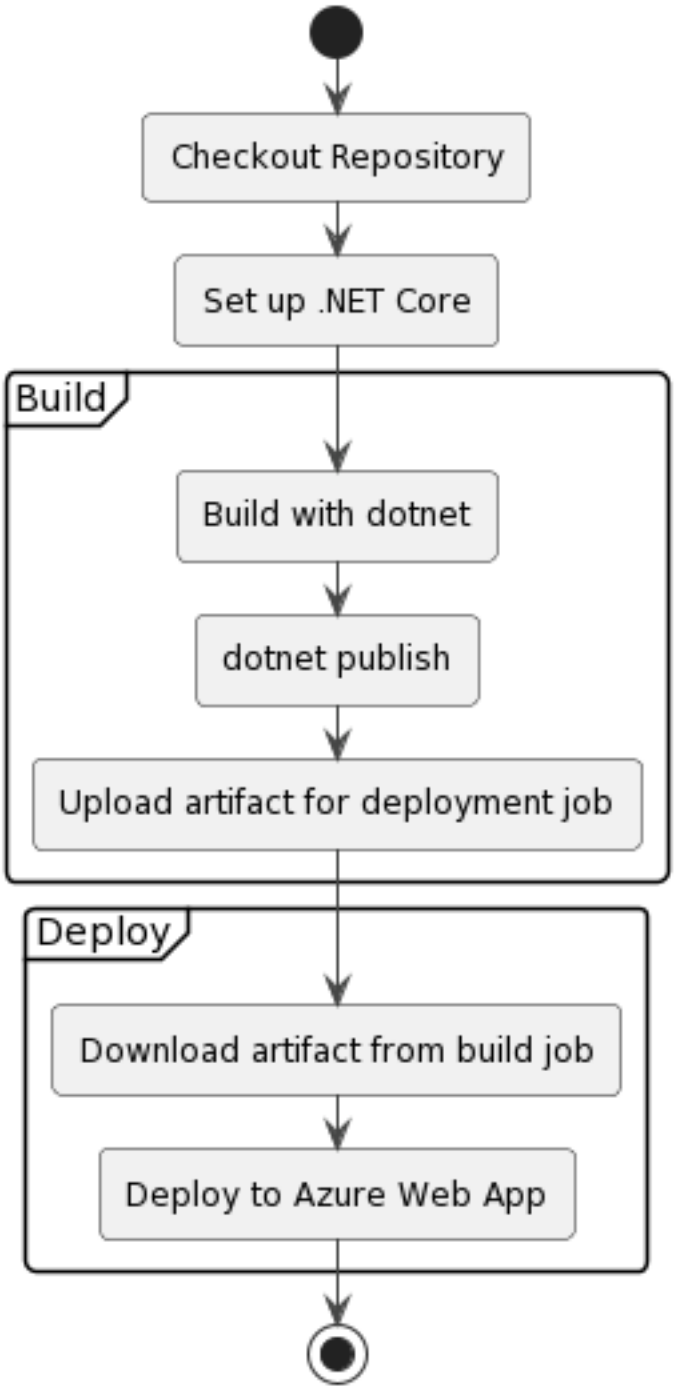

Figure 19: Activity Diagram for Deployment

Note: Figure x, 'Activity Diagram Deployment' should have been here, but latex has its own life, and its not easy to fix when it needs to go through pandoc.

Our deployment workflow, as visualized in the activity diagram, efficiently manages the deployment of the

Chirp! application to Azure Web App. The process is triggered upon pushes to the main branch or via manual dispatch, ensuring that our latest stable build is always deployed.

The workflow begins with checking out the repository and setting up the .NET Core environment. The build phase involves compiling the code and publishing it to a specified directory. The published application is then uploaded as an artifact, ready for deployment.

In the deployment phase, the build artifact is downloaded and deployed to the Azure Web App. This automated process ensures a consistent and reliable deployment strategy, minimizing human error and streamlining our release process.

2.2 Build and Test

This workflow builds and test the aplication before each push to branch, and pull request to main. The workflow runs every test but the end to end test.

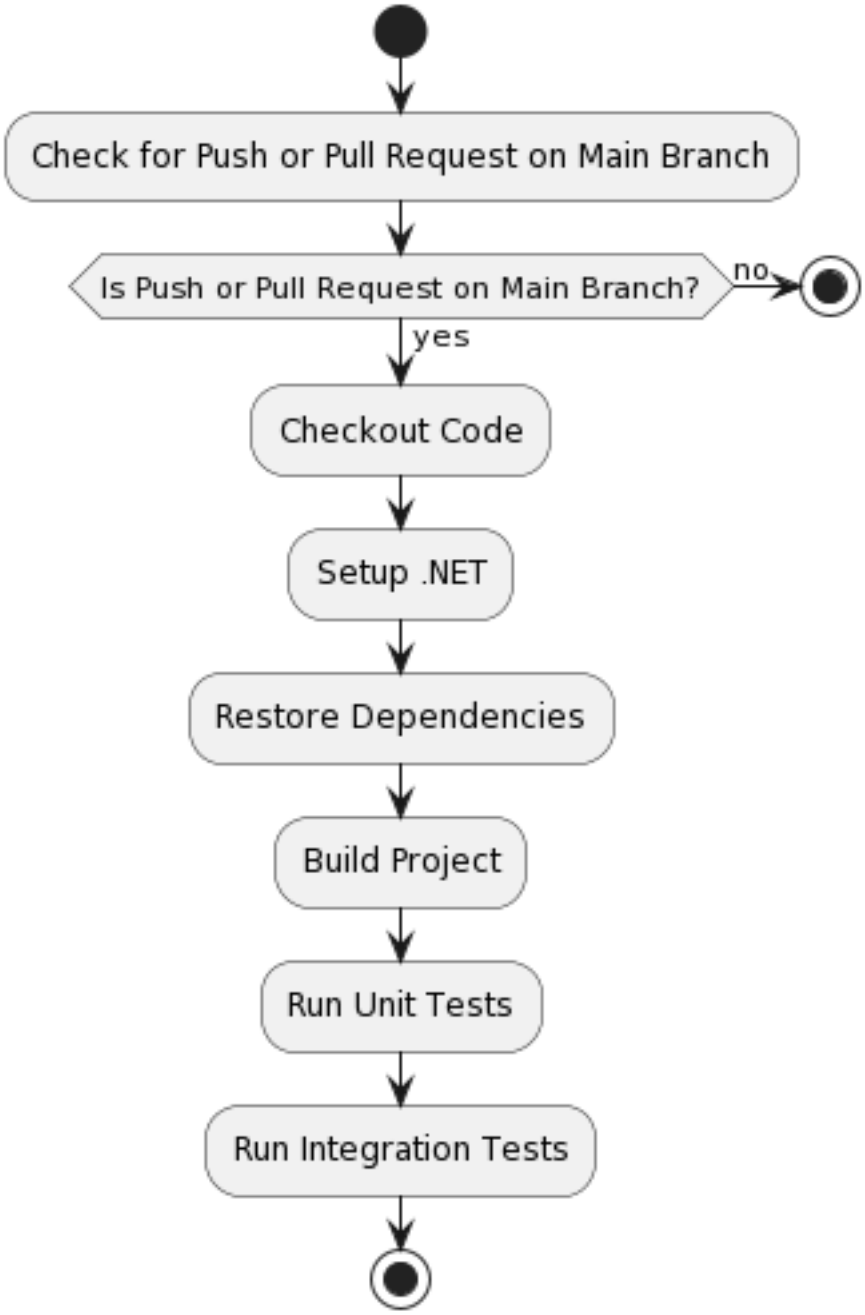

Figure 20: Build and test workflow

2.3 Team work

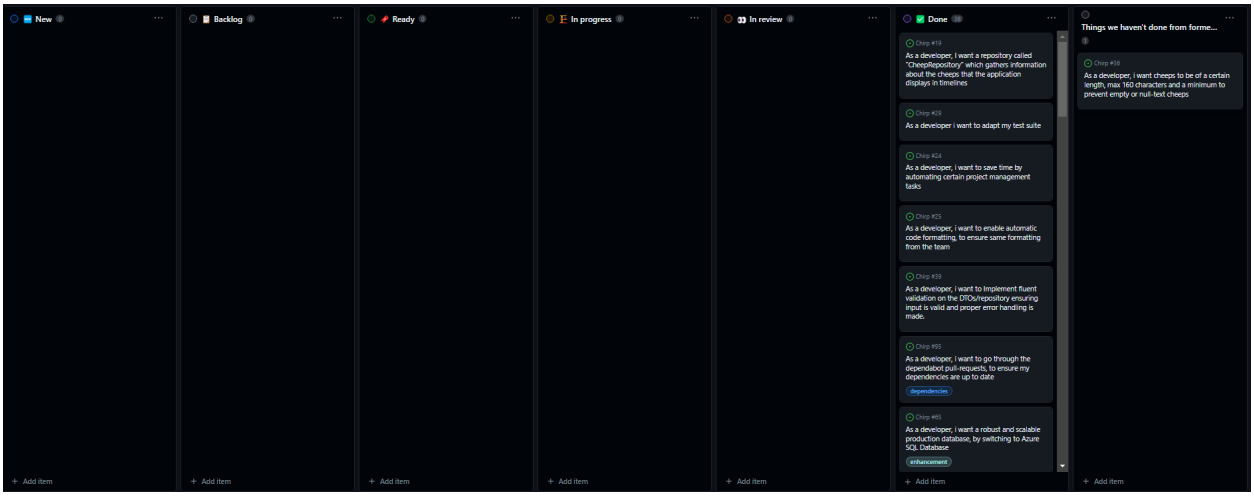

Figure 21: Backlog

The only unresolved task, was to create a constraint within the program that prohibits cheeps which have a length of more than 160 charectars.

The below image shows the flow of events from creation of issue to resolved issue:

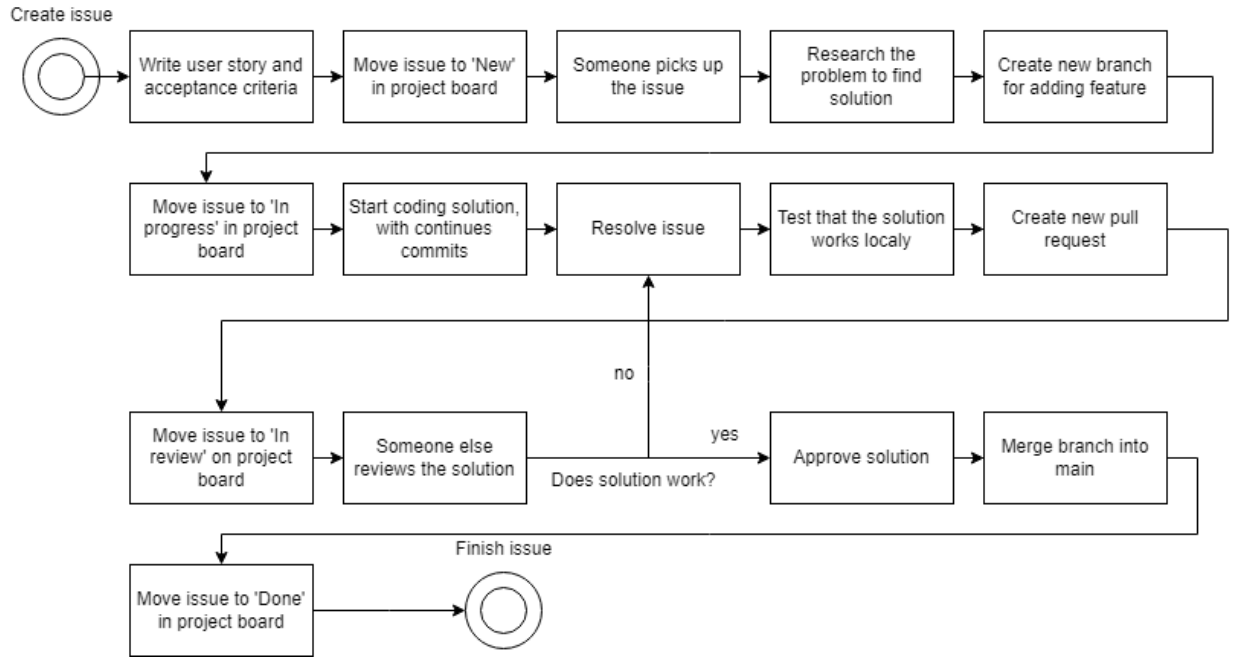

Figure 22: IssueFlowChart

When a new feature is to be implemented, or a bug is to be fixed, we make an issue, and attach it to our project board. When we write an issue, we make sure to write it as a user journey, where we highlight who would be interested in having what implemented, with what purpose.

As soon as an issue is written, it is dropped to the 'New' column in our project board. When it is agreed

upon, that it is an appropriate time to implement it, we move it to the 'Ready' column. In due time, a group member will move it to the 'In Progress' and start working on it. When done, he drops it in the 'In review' column, where we wait to get a second pair of eyes on it. If everything is as it's supposed to be, it will be approved, merged and the issue moved to 'done'.

Sometimes there are features that we would have wanted to implement, but for various reasons didn't get around to, and at some point had to abandon all together. These issues are placed in the 'Things we havent done from earlier weeks' co

2.4 How to make *Chirp!* work locally

First clone the repository into a local file structure on the computer with the following command:

Git clone https://github.com/ITU-BDSA23-GROUP25/Chirp.git

Then navigate to the directory Chirp/src/Chirp.Web. Then run 1 of the following commands to start the program:

dotnet run or dotnet watch

2.5 Our test suit

2.5.1 Unittests & integrationtests

We are testing our systems functionalities using: End-to-end tests, Integration tests, and Unit tests for the different projects in our program

Each Repository and its related methods is test thruoh either unittest or integratins test. In most cases the repository is test using a combination of both.

2.5.2 Ui-testing

For ui testing we use the open source libary playwright. The ui testing is a form of end to end test, where the test interact with the web applications features simulating user behaviour, thereby verifying that features work as intended.

2.6 How to run test suite locally

In order for the ui test to run locally, you have to make sure the libary playwright is installed.

To install run the following command in powershell:

cd test/Chirp.Web.Ui.Tests pwsh bin/Debug/netX/playwright.ps1 install

Note that if you are on a Mac or linux computer remeber that the install command needs to be run in powershell, so you might need to install powershell.

2.7 To running test suit

In order to run the test suit locally navigate to /chirp and run the following command:

dotnet test

This should run all the test.

Note if the playwright labary isnt propaly installed the UI test will automaticlly fail.

3 Ethics

We are committed to being inclusive and respectful to anyone, related or unrelated to the work done in this project. We have included a slightly modified version of the Citizen Code of Conduct, that has been included in our repository on Github under CODE_OF_CONDUCT.

3.1 GDPR

With GDPR in mind, Chirp! wants to store as little possible information as possible about the user. Here the only data needed for making an Author, are their unique username on github. Their email is gotten through claims, but isn't stored, as chirp don't need it. The users github profile picture isn't stored either, it is gotten from a public github endpoint, and is picked from the endpoint each time a cheep is displayed.

3.2 License

We have picked the MIT License for our project. The MIT License is a simple and highly permissive open-source software license. It is one of the least restrictive, of the established licenses available. Under the MIT License, users are granted almost unrestricted freedom to use, modify, distribute, and sublicense the software. Our only requirement is that the copyright notice and license is included with the software when redistributed. As our group name is included in the MIT-license, this ensures that we are attributed. Additionally, the license absolves us of any responsibility or liability of how the software is used by others.

We have included the full text of the license below:

```
MIT License

Copyright (c) 2023 ITU-BDSA23-GROUP25

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

The license can be found on our Github repository, under LICENSE.

3.3 LLMs, ChatGPT, CoPilot, and others

As of the use of LLM's, the group haven't used CoPilot, but have used ChatGPT throughout the course of developing Chirp!. We have used ChatGPT to help with debugging and css styling. The overall gain from using such LLM is varying, but for the most part, ChatGPT took of in wrong directions. Despite the limited use, we still regretfully acknowlegde that we haven't appropriately attributed ChatGPT as a co-author in our commits.