Chirp! Project Report

ITU BDSA 2024 Group 13

Alexander Hvalsøe Holst alhh@itu.dk
Andreas Løvsgren Nielsen anln@itu.dk André Racraquin Birk arbi@itu.dk
Peter Aksel Skak Olufsen polu@itu.dk
Símun Larsson Løkke Rasmussen simra@itu.dk

Contents

1 Design and Architecture of *Chirp!*

1.1 Domain model

Figure 1.1.1: Domain model

The Domain for chirp is based off of two ***Entities***, and one superclass, `IdentityUser`, which ***Author*** extends. They derive attributes such as `id`, `username`, `email` and an encrypted password.

Authors are the equivalent of application users. Apart from the extended attributes, ***Authors*** contain all ***Cheeps*** (messages), that they have written. They keep track of who they follow, who they are followed by, ***Cheeps*** they have liked, and ***Cheeps*** they have disliked. They cannot both like and dislike a ***Cheep*** at the same time. Two ***Authors*** cannot have the same case-insensitive username or email i.e. An ***Author*** cannot be named "helge", if "Helge" already exists.

Cheeps are all the messages of the application. ***Cheeps*** are identified with a unique id. They contain attributes, like text, an optional image, the time of posting, and their associated ***Author***. They contain data for which ***Authors*** have liked or disliked. They also contain a calculated float based on their amount of likes, which is used for sorting by relevance.

Additionally, the attribute `Text` in ***Cheep*** cannot contain more than 160 characters and they can only have a single ***Author***.

1.2 Architecture — In the small

1.2.1 Design and architecture

Figure 1.2.1: Core diagram

The `Chirp.Core` package contains the domain ***Entities*** and data transfer objects, for database transactions.

The `DTO`'s are split up into two groups, one for each ***Entity***. In order to obtain the **SOLID principles** (Single responsibility, Open/closed, Liskov substitution, Interface segregation, and Dependency inversion). `DTO`'s are split up even further to strive for the single responsibility principle. Thus there is the `NewCheepDTO`, which is for sending data of new ***Cheeps*** into the database. The default `CheepDTO` is for reading ***Cheeps***, for showing on the timeline. The `UpdateCheepDTO` is for editing existing ***Cheeps***, by changing their content. Lastly, there is the `CheepDTOForRelevance` which is used for the relevance sorting algorithm.

For the ***Author***, two `DTO`'s have been made for either creating an ***Author*** or to get information on the ***Author*** from the database.

Figure 1.2.2: Repository diagram

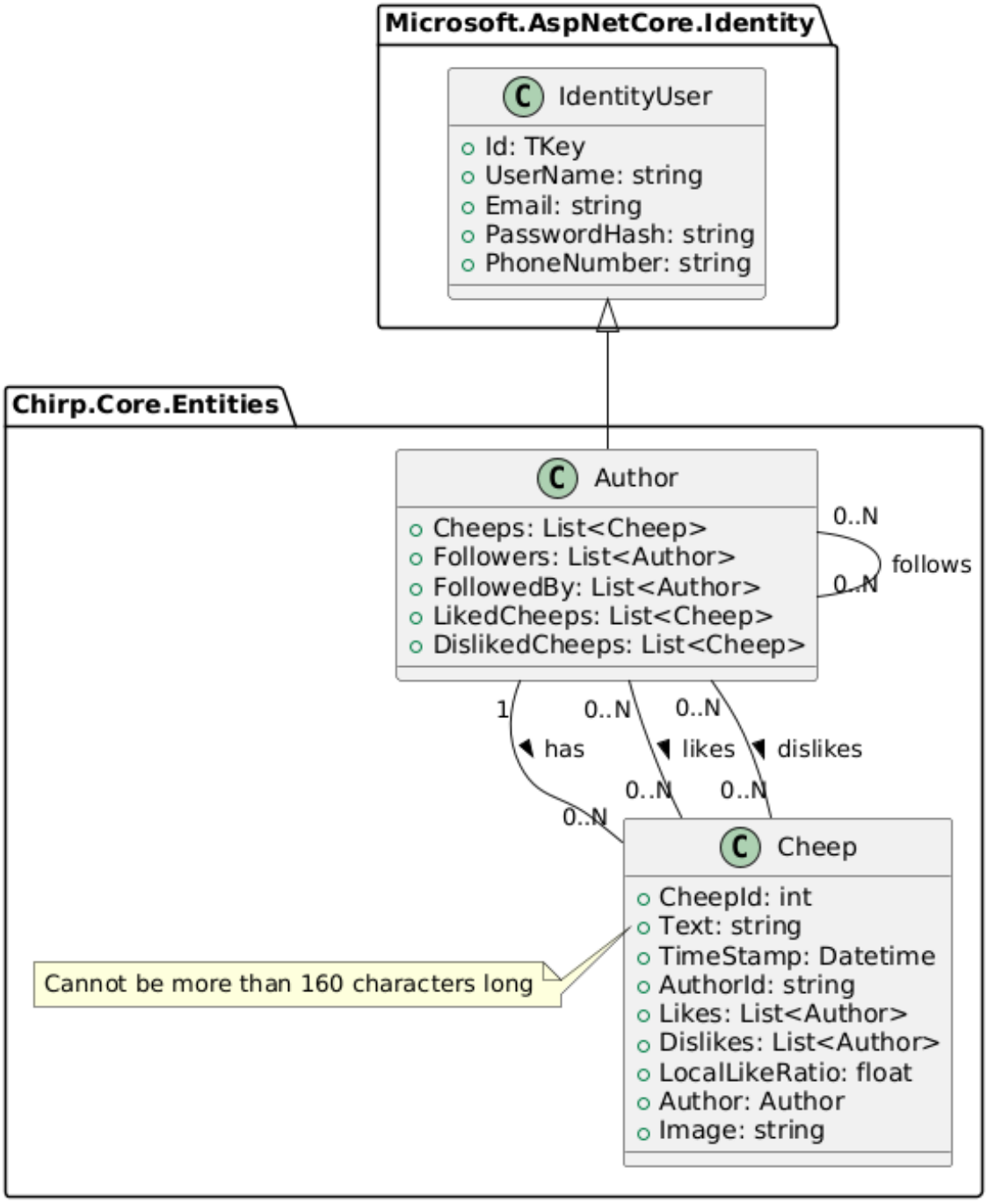

Figure 1: Diagram of Domain

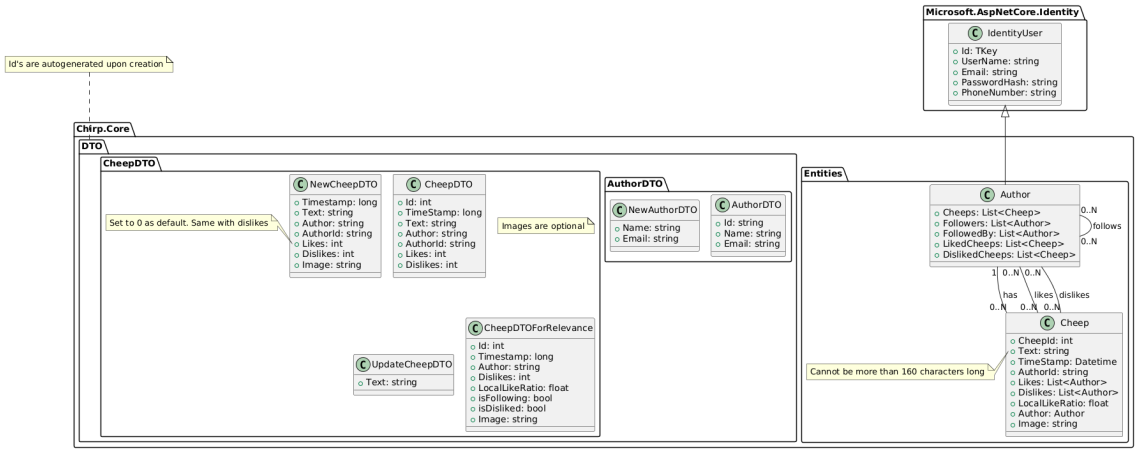

Figure 2: Core Diagram

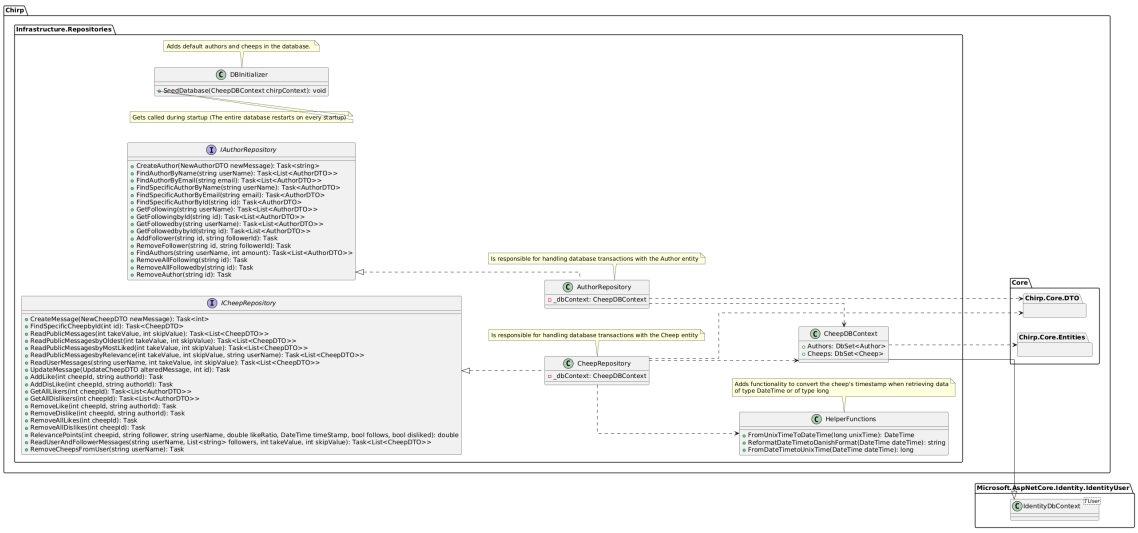

Figure 3: Repo diagram

The ***Chirp.Infrastructure.Repositories*** package, contains *classes* and *interfaces* regarding the database and *classes* which seed or query the database.

`AuthorRepository` and `CheepRepository` queries the database depending on whether ***Author*** or ***Cheep*** is the main ***Entity***.

The `CheepDBContext` defines the database ***Entities*** and the relations between them.

Both `AuthorRepository`, `CheepRepository` and `CheepDBContext` are dependency injected into the application. This ensures one and only one instance of each.

The `DbInitializer` seeds the database with default ***Cheeps*** and ***Authors***. This makes it easier to make in-memory testing.

The static class `HelperFunctions` provides functionality to the `CheepRepository`. ***Cheeps*** contain `DateTime` and `DTO`'s should only store predefined types, `DateTime` needs to be converted to unixTime of type `long` and vice versa.

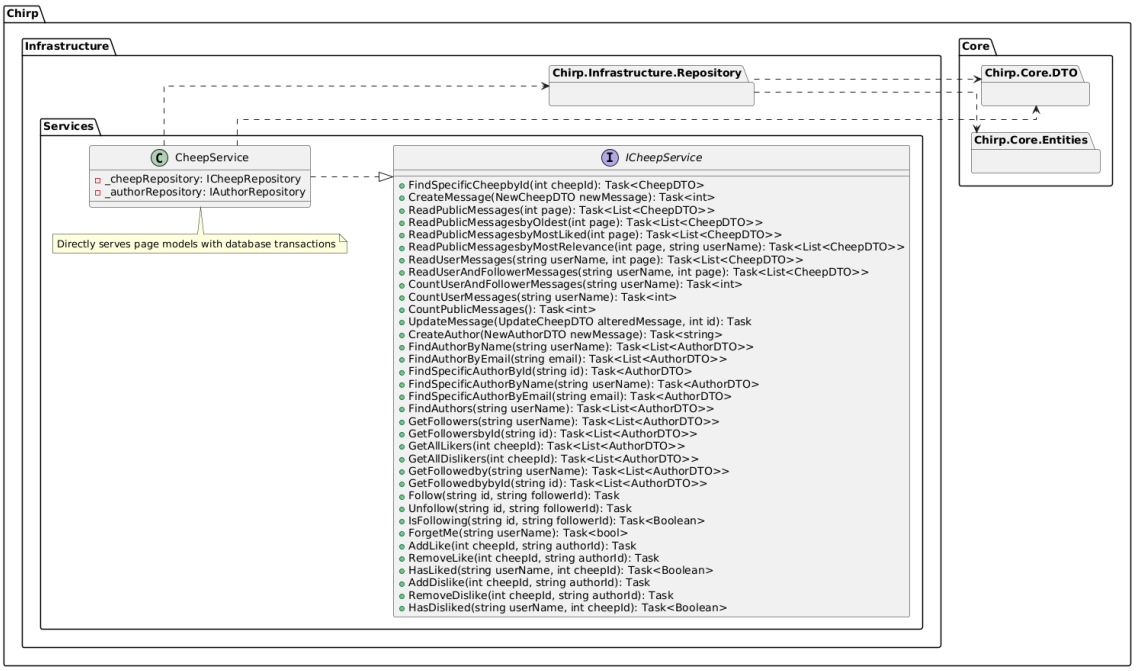

Figure 4: Service diagram

Figure 1.2.3: Service diagram

The ***Chirp.Services*** package contains the `CheepService` class, which directly communicates with the page models.

The service transacts data between the *page models* and indirectly the database using the *repositories*.

`CheepService` contains the dependency injected `IAuthorRepository` and `ICheepRepository`.

The `CheepService` itself is also dependency, injected into the application. Page models refer to the same service, which refers to the same repositories, which refer to the same database.

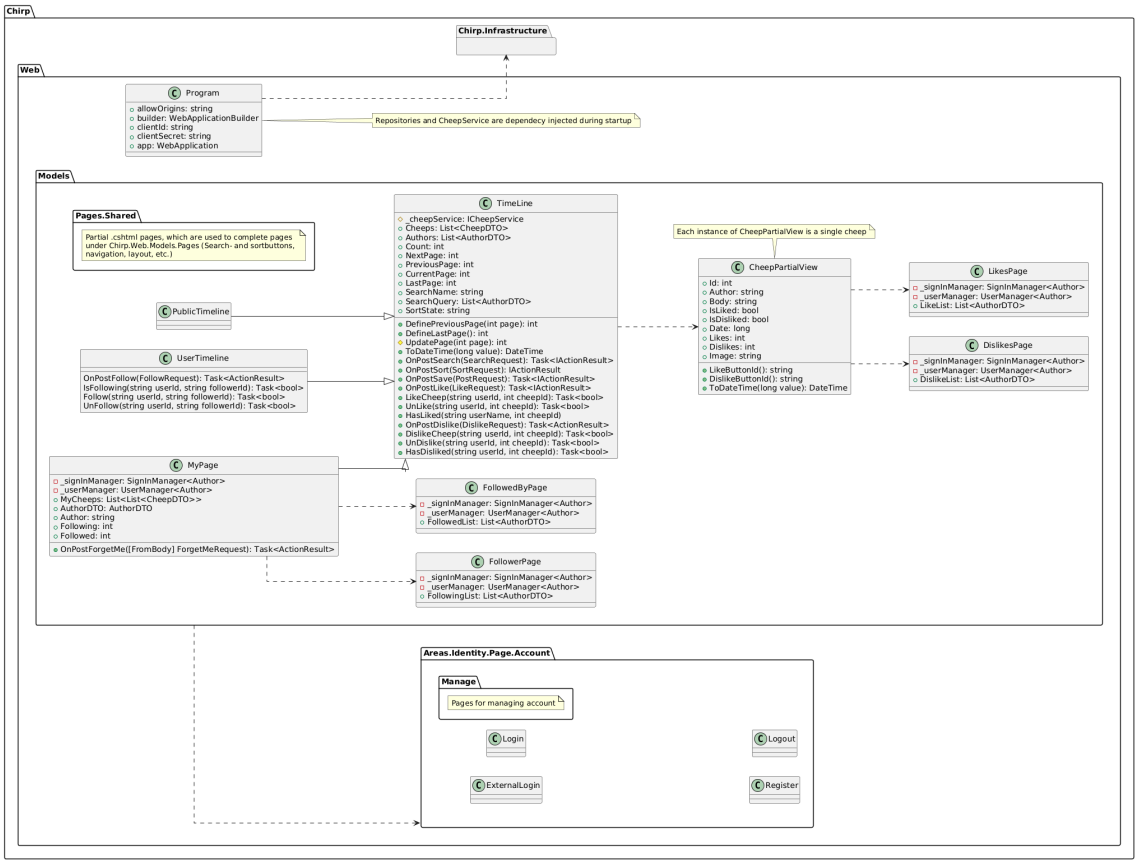

Figure 5: Web Diagram

Figure 1.2.4: Web diagram

The ***Chirp.Web*** package contains all the pages, as well as the startup program.

The pages are made up of page models written in `C#` and the pages in `cshtml`.

The `cshtml` pages send requests to the model which are handled by reading or writing to the database using the application's associated *service interface*.

The scaffolded package ***Area.Identity.Page.Account*** is used to handle getting an identity token when logging in and managing the account using ***Microsoft.AspNet.Identity***'s IdentityUser.

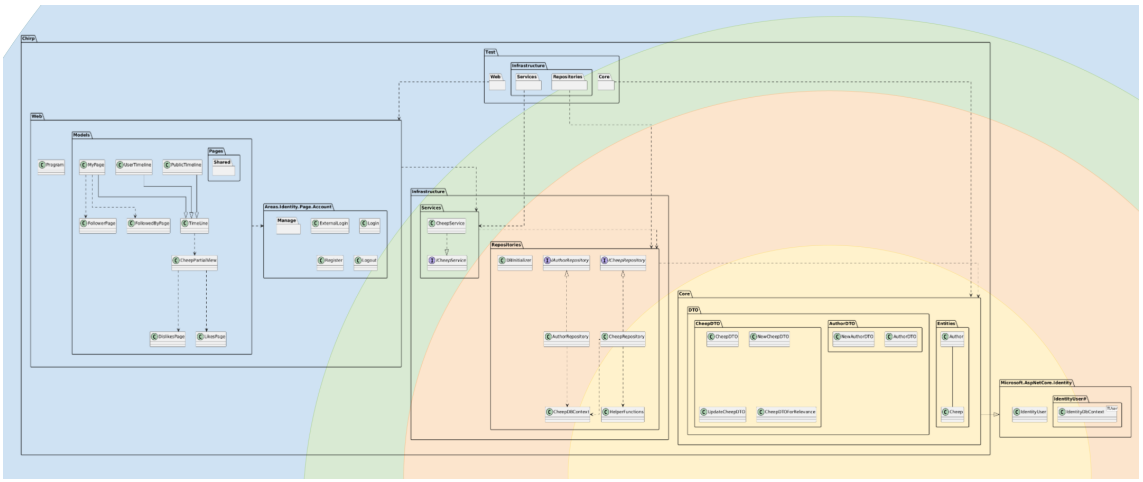

Figure 6: Onion Coloured

Figure 1.2.5: Onion architecture

The entire `Chirp` package fulfills the *onion architecture*. Since ***Chirp.Core*** does not need to refer to any of the outer layers. The same goes for the *repository layer* and the *service layer*.

Figure 1.3.1: New User

This diagram illustrates the *architecture of the Chirp application* as well as the interaction between its key components. The system is divided into three main layers: *the Client, the Server, and the Database*, all hosted within the *Azure* environment.

- **Client Layer**: The user interacts with the application using a web browser. When the user enters a `URL` or performs an action, the browser sends `HTTP` requests to the server. The browser receives responses in the form of `HTML`, `CSS`, and `JavaScript`, which it uses to render and display the user interface.
- **Server Layer**: Hosted on the *Azure App Service*, the server is made up of two primary components: *the Web Server* and *the Application Logic*. The web server handles incoming `HTTP` requests from the *client* and *routes* them to the appropriate *application logic*. This logic processes the request and then queries the database for data or updates if necessary.
- **Database Layer**: The database is integrated into the *Azure* ecosystem and stores all the application data. This includes user information, *Cheeps*, and *relationships*. It responds to

queries from the application logic with the requested data or confirms successful updates.

The flow begins when a user interacts with the browser (e.g., entering a `URL`). The browser sends a request to the *web server*, which processes the request and interacts with the *application logic*. The database returns the required information, which is processed and formatted by the *application logic* and *web server* before being sent back to the client as an `HTTP` response. The browser then renders the returned content and updates the *user interface* accordingly.

2 Useractivity

When it comes to *webdevelopment*, the overall *userexperience* and functionality of the website is crucial. Giving the user access to the functions of the website while also maintaining the safety of the website, can end up being one of the more important aspects of *webdevelopment*.

The sitemap in figure 1.4.1, contains most of the traversal possibilities for a user, when logged in and logged out, to illustrate the user's accessibility in different parts of the website and the general structure of the website. - *Notice: To understand the elements in the sitemap, it is recomended to first have a general understanding of the definitions of colors and arrows in the bottom part of figure 1.4.1.*

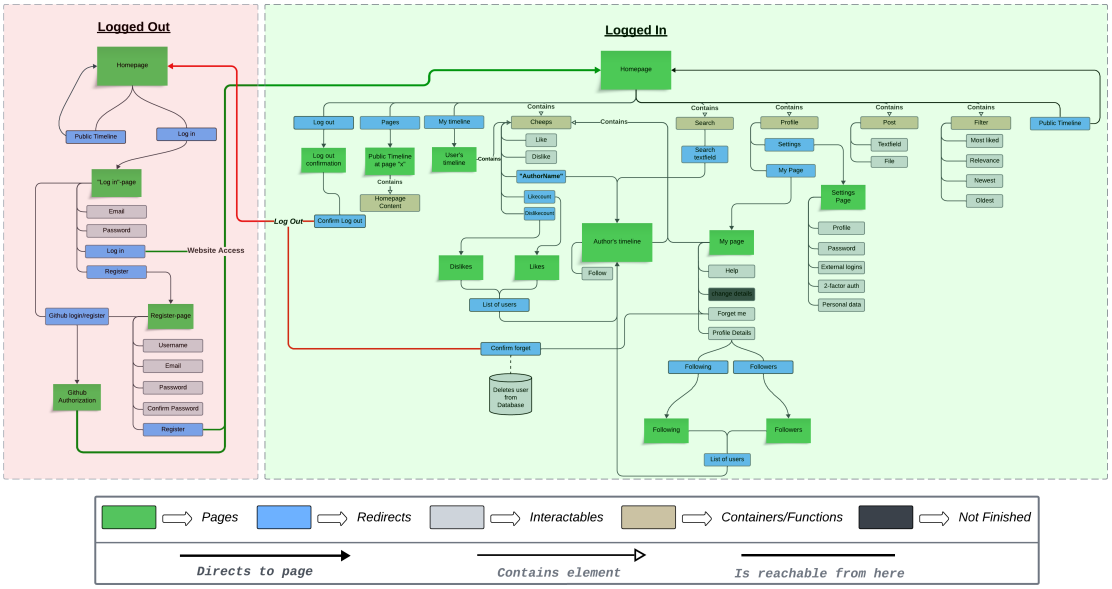

Figure 7: Current Project Board

Figure 1.4.1: sitemap

2.0.1 Logged out

When a user is logged out, they do not have the same accessibility as a user who is logged in. Their access is very limited, and it only allows the user to log in or register. Any references to the websites structure in this chapter, will be directed towards the illustration in figure *1.4.2*

Figure 1.4.2: Logged out

2.0.1.1 Github login/register

When a user enters the website, they will see the front page without content. To view any content, the user must press the "Log in" button, which will lead them to the "Log in"-page. From here the user can choose to use an external login to access the website. The external login uses the *GitHub Authorization* process giving the website a token, which allows the website to access the information about the *github user*.

If the user exists within the database, the user is logged in and is now able to access the application with all of its functionalities. If a user does not exist, the user's *Github User ID* becomes the username and they are registered as a new user.

2.0.1.2 Normal login/register

As shown in figure 1.4.2 to log in, without an external login, to the website, a user must write their email and password of their account. If they do not exist in the database, an error message will be sent back to inform the user that the log in process has failed. If it succeeds, the user will be given access.

However if a user does not have an account, the user can access the register-page, where they are prompted to enter their email, username and password. If the information does not already exist within the database, the user will be registered as a new user and be given authentication, and be authorized to see *Cheeps* and post *Cheeps*. But should the information already exists, the person will be given an error message, informing the user that the information is already in use.

2.0.2 Logged in

When a user is logged in,y have full authorization to the website, which includes both functions and content. Most of the interactability is present on the front page. References to the websites structure in this section, will be directed towards the illustration in figure *1.4.3*

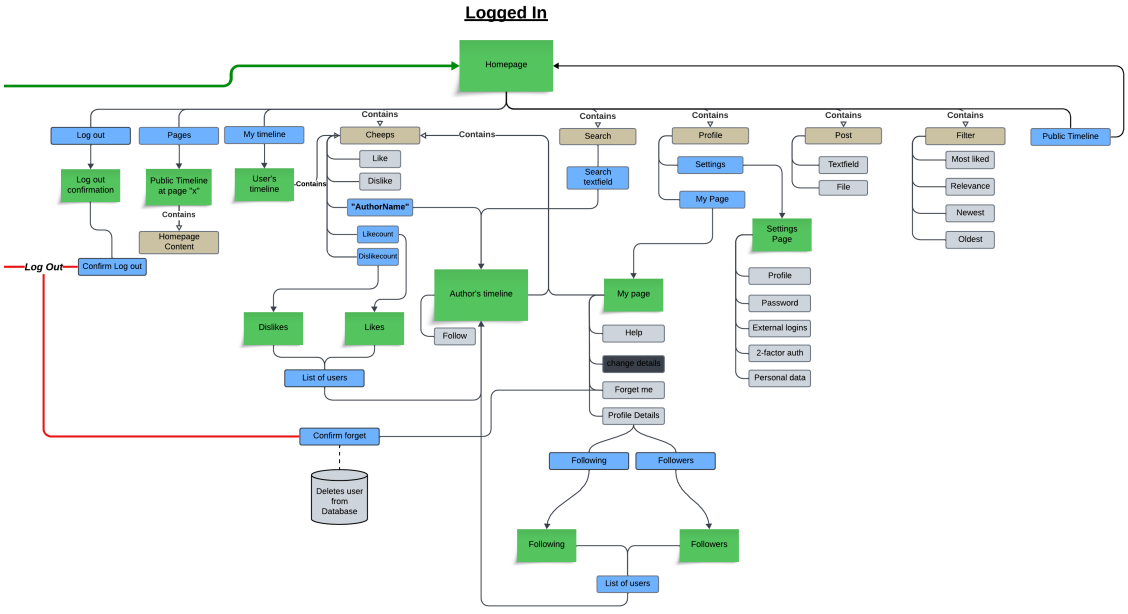

Figure 1.4.3: Logged in

2.0.2.1 Main page

2.0.2.1.1 Cheeps As seen in *figure 1.4.3* under *Cheeps* a user can read and then like or dislike a *Cheep*. From the *Cheep*, a user can access the *Author*'s page and timeline as well as see who liked or disliked the *Cheep*.

2.0.2.1.2 My Timeline The user's timeline displays the user's own *Cheeps*, and the *Cheeps* posted by who they follow.

2.0.2.1.3 Post and Search The search function searches the database for *Authors* When searching, the user is presented with a textfield, which will find possible search results for any given *Author* which matches the content of the search-bar. The results of *Authors* that are returned can be clicked, which redirects to the *Author*'s timeline.

The `Post` function as seen in *figure 1.4.3* has two elements; a *textfield* and a *file button*. The *textfield* can be filled out by the user. The *file button*, allows the user to choose a picture from their own computer, which will then be included in their *Cheep*, along with the text.

2.0.2.1.4 Filter As seen in *figure 1.4.3* a user can choose to check out popular *Cheeps* by pressing the option *MostLiked*. This filters the *Cheeps* in a descending order with the most liked *Cheep* being at the top. Other options such as *Newest* or *Oldest* will order the *Cheeps* based on time. The *Relevance* option in the filter will give you a order based on time, however the *Cheeps* shown, will be *relevant* - which means *Cheeps* from the user's following and liked *Cheeps*.

2.0.2.2 Profile To explore the user's account and their own information, the user can click on their profile picture. This displays a new page with the user's *My Page* and *Settings*. References to the websites structure in this section, will be directed towards the illustration in figure *1.4.4*

Figure 1.4.4: Profile

2.0.2.2.1 My Page The *My Page* element, when accessing the profile, redirects the user to the user's *My Page*. Within this page is most of the relevant information of the user, such as email, username and the amount of followers and amount of people that the user follows. The user can access a page with a list for all the users that they follow, and a list of all the users that follow them. Moreover, the *My Page* also has other functionalities and information, such as the *Forget Me* option, which the user can choose, when looking at the overall sitemap in figure *1.4.1*, to delete all their information. The *Help* includes a basic guide on how to use the **Chirp** website.

2.0.2.2.2 Settings The *Settings* button allows the user to see a more detailed view of their *account information*. By looking at figure *1.4.4* the *Settings* page contains more options than the user's *My Page*. On the *profile page* the user can view their account info such as the username or register a telephone number. Other elements such as *Password*, gives the user the ability to change their password. If the user wanted to link an external login to their user, they can navigate to *External Logins* where they are able to link their github account to their Chirp account. If the user wishes additional safety measures for their account, they can navigate to the 2-factor authentication, which will allow the user to link an authentication app. The *Personal Data* allows the user to download their data from the Chirp website.

2.0.3 Flow of new user

Figure 1.5.1: New User

The diagram above illustrates the flow of a user signing up or in to the Cheep service. The user, upon navigating to the root endpoint, initiates an `HTTP GET` request which is processed by the *web server*. The server checks if the user is authorized. Since the user is not yet authenticated, the server responds with an `HTML` page that displays an empty public timeline and a login button.

If the user logs in, they interact with the login page by clicking the login button, which sends another `HTTP GET` request to the `/login` endpoint. The server responds by rendering the *Login.cshtml* Razor page, presenting the user with a form to either log in or register.

Once the user submits their credentials through the form, an `HTTP POST` request is sent to the `/login` endpoint. The web server validates the credentials. If authentication succeeds, the server redirects the user back to the public timeline view.

As an authenticated user, the browser sends another `HTTP GET` request to the root endpoint. This triggers the server to render the *PublicTimeLine.cshtml* Razor page by invoking the `OnGetAsync()` method. During this process, the PublicTimeLine component calls the `CheepService` to retrieve public messages and their count. The `CheepService`, in turn, queries the `CheepRepository`, which executes database queries to fetch the required data.

Once the database returns the list of *Cheeps* and the count, the information is passed back to the *service* and *controller layers* to the `PublicTimeLine` component. The Razor page is rendered with the retrieved *Cheeps*, and an `HTML` response is returned to the browser. The user's browser then displays the fully rendered public timeline with the fetched *Cheeps*.
Sequence of functionality/calls through *Chirp!*

Figure 1.5.2: Post Cheep

When posting a cheep, a user would initiate the following flow. Typing a post into the input field and pressing "Enter," the browser triggers an event to process the input. This sends an `HTTP POST` request with the form data to the web server. The server invokes the `OnPostSave()` method in the `PublicTimeLine` component to handle the post submission.

Firstly, the post content is validated, and if an image is included, it is saved with a `URL`, designated to the filepath of the stored image. The `PublicTimeLine` component then interacts with the `CheepService` to locate the author of the post. The `CheepService` queries the `CheepRepository`, which fetches the author data from the database.

With the author information, a new `Cheep message` is created and sent to the `CheepRepository` and is stored in the database. Once the database confirms the save, the success response propagates back through the `CheepService` and `PublicTimeLine` component.

Finally, the server renders the updated `PublicTimeLine.cshtml` Razor page, including the new *Cheep*, and sends the `HTML` response to the browser. The browser then displays the updated timeline to the user with their new message.

Figure 1.5.3: Search

When using the search functionality, the browser triggers an input event for each keystroke, as the user types into the search input field. This sends an `HTTP POST` request to the web server, containing the current search string. The server calls the `OnPostSearch()` method in the `TimeLine` component to handle the search.

The `TimeLine` component extracts the *search string* and interacts with the `CheepService` to find matching authors. The `CheepService` queries the `CheepRepository`, which searches the database for users matching the input. The results are returned as a list of *Author* data objects, which are sent back to the browser in a `JSON` response.

Upon receiving the response, the browser dynamically calls the `showResults()` function, which creates and updates `DOM` (Document Object Model) elements to display the search results. This allows the user to see search results updating in real-time as they type, rather than having to complete a search and press Enter.

Figure 1.5.4: Follow Action

When a user presses the *follow button* on an *authors page*, the browser triggers an input event and sends an `HTTP POST` request to the web server, containing the usernames of the follower and the followee. The server invokes the `OnPostFollow()` method in the `UserTimeLine` component to handle the follow and unfollow actions.

The `UserTimeLine` component calls the `CheepService` to retrieve both user profiles. The `CheepService` queries the `CheepRepository`, which fetches the users' data from the database. Once the user data is returned, the service checks whether the initiating user *(Author A)* is already following the target user *(Author B)*. - If *Author A* is not following *Author B*, the `CheepService` creates a follow relationship between the two users and stores it in the database. - If *Author A* is already following *Author B*, the `CheepService` removes the follow relationship from the database.

Upon a successful follow or unfollow, a response is propagated back through the `CheepService` and `UserTimeLine` component. The updated `UserTimeLine.cshtml` Razor page is rendered by the server and sent to the browser. The browser then displays the updated timeline, reflecting the user's changed follow or unfollow status dynamically.

3 UML Activity Diagram for Chirp! Application

3.1 Overview

The following describes the **UML activity diagrams** representing how the Chirp! application is **built**, **tested**, **released**, and **deployed** using **GitHub Actions** workflows.

The key activity of this project has been **automating mundane tasks**, which significantly decreases the accumulated workload and speeds up processes. Using **GitHub Actions**, the need for manually creating releases, generating DLLs, and deploying the service to Azure has been **eliminated**—excluding the time invested in creating these workflows.

While testing with **Playwright** caused some issues on GitHub, leading to skipped testing steps in workflows, the focus has been on maintaining and ensuring workflows function correctly. Code

quality was considered less critical because **Git rollbacks** can revert any problematic changes.

3.1.1 Key Workflow Triggers:

1. **Primary Trigger**: Push to the **main** branch (e.g., after an accepted pull request).
2. **Secondary Trigger**: A scheduled workflow run every **Sunday at 08:00 UTC**.

Once the **Create Release** workflow completes, it triggers two subsequent workflows: - **Make DLL** - **Build and Deploy**

3.2 Creating a Release Workflow

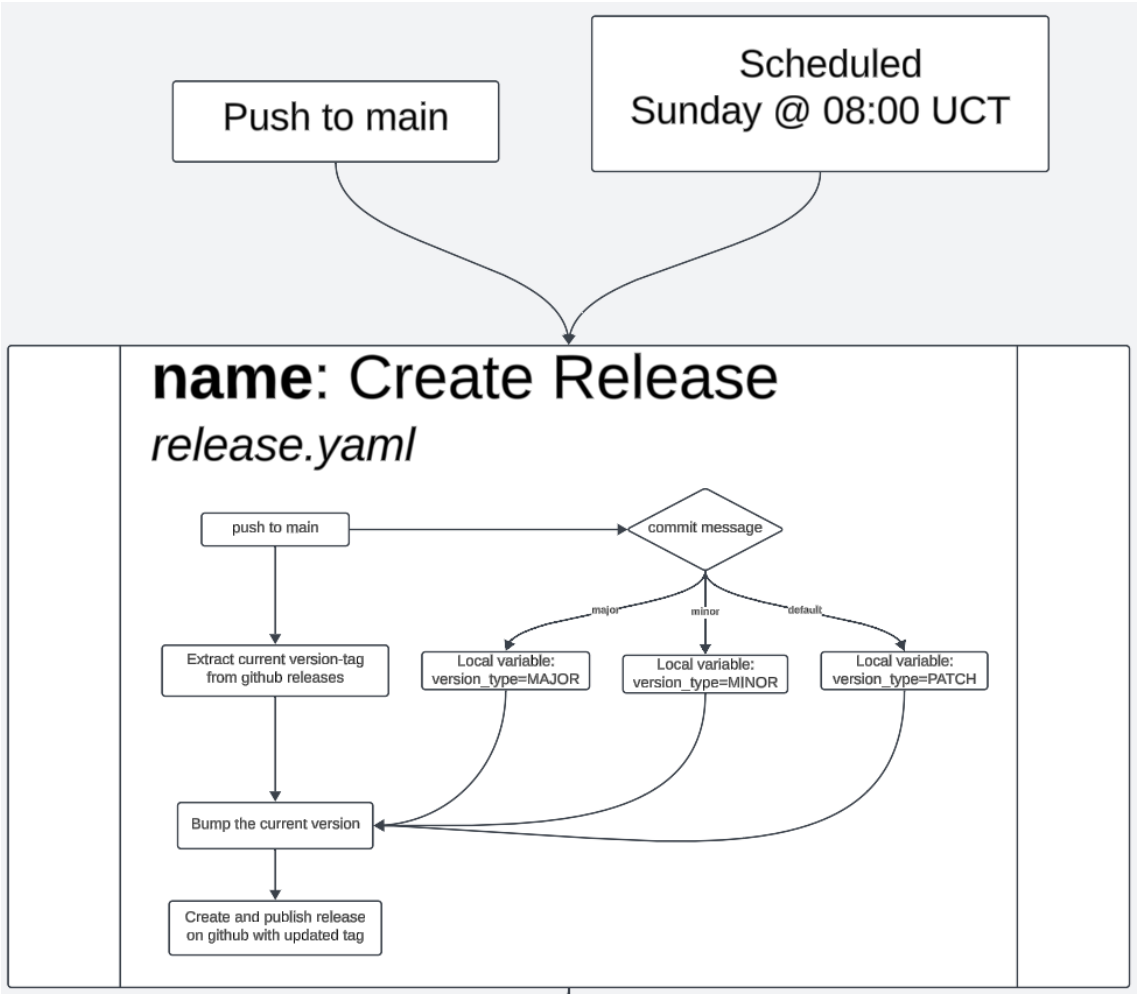

Figure 8: Create Release Workflow

Figure 2.1.1: Create release workflow

3.2.1 Description

The **Create Release** workflow triggers under two conditions: 1. **Push to `main`** branch. 2. **Scheduled run** on **Sunday at 08:00 UTC**.

3.2.2 Purpose

The workflow **automates the creation of a new release** by: - Scanning the **commit message** for keywords to determine the version bump. - Following the **Major.Minor.Patch** versioning convention: - **Major**: Total rework of the system (e.g., switching from CLI to a web-based service). - **Minor**: New features added to the existing system. - **Patch**: Bug fixes, formatting changes, or refactors.

If the commit message includes: - `Major`: The version bump will increment the **Major** version. - `Minor`: The version bump will increment the **Minor** version. - **Default**: If no keywords are detected, the version will default to a **Patch**.

3.2.3 Notes:

- This workflow **previously contained a testing step**, but it was **removed** due to compatibility issues.

3.3 Making DLLs Workflow

Figure 2.1.2: Make DLL workflow

3.3.1 Description

The **Make DLL** workflow builds the program and generates a **zip file** containing the `.dll` files for distribution.

3.3.2 Matrix Strategy

- A **matrix** is used to optimize the step, specifically the **"Process for creating a zip file with .dll"**.
- The matrix reduces **code redundancy** and simplifies supporting multiple operating systems.
- If additional OS platforms need to be supported in the future, the matrix makes it easy to extend the workflow.

3.3.3 Workflow Steps:

1. **Build the Program**:
   - The program is compiled to generate `.dll` files.
2. **Create a ZIP File**:
   - The DLLs are packaged into a zip file for easy distribution.
3. **Attach Files to the Latest Release**:
   - The zip file containing DLLs is appended to the **latest GitHub release** created by the **Create Release** workflow.

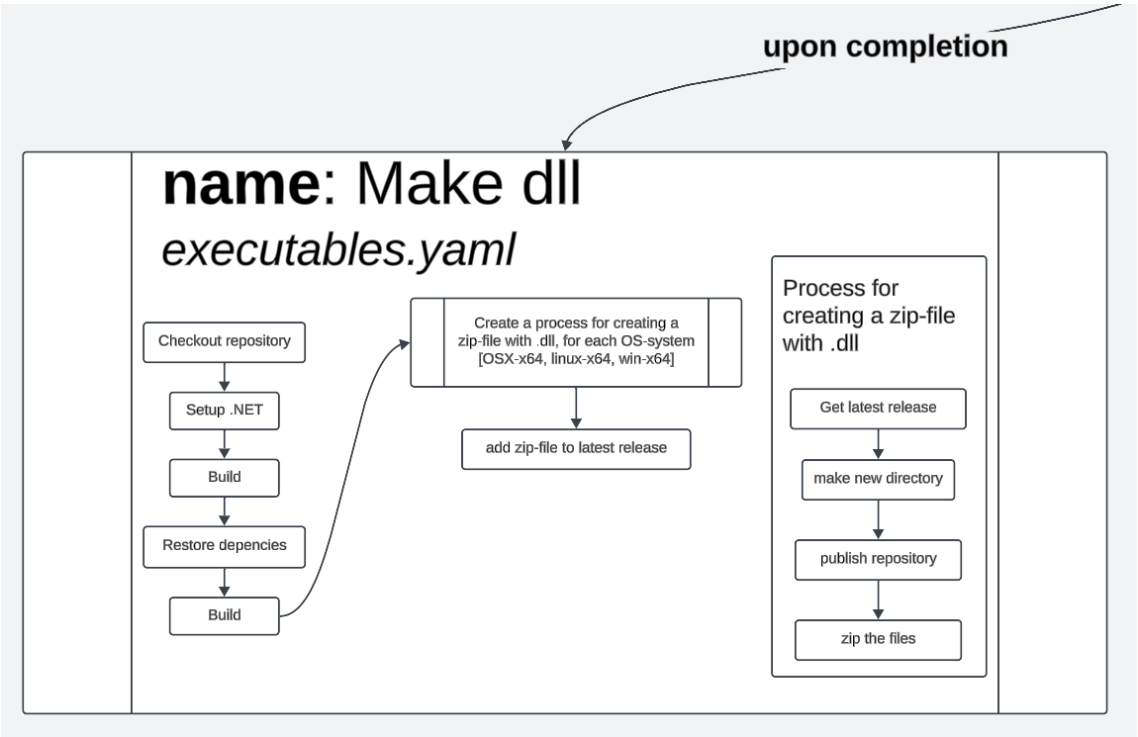

Figure 9: Make DLL Workflow

3.3.4 Dependency:

- It is **crucial** that the **Create Release** workflow runs successfully before `Make DLL` starts.
- If no new release is created, this workflow may **overwrite the files** in the most recent release.

3.4 Deploying to Production Workflow

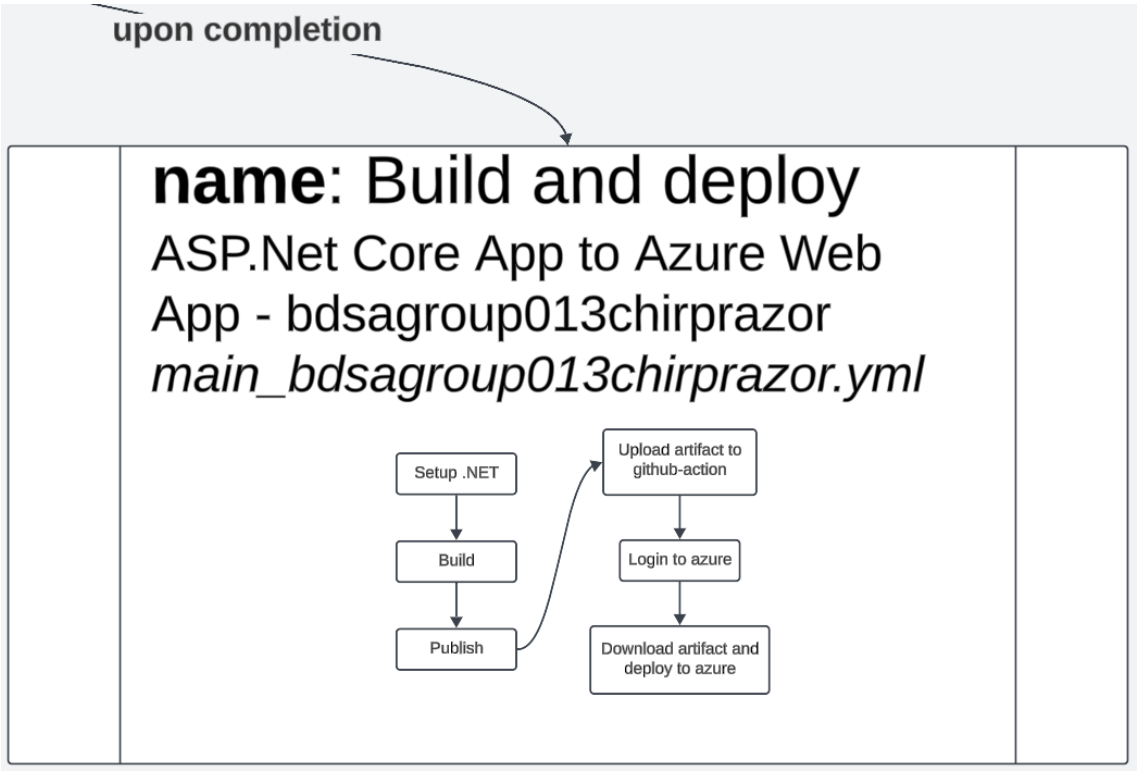

Figure 10: Build and Deploy Workflow

Figure 2.1.3: Build and deploy workflow

3.4.1 Description

The **Build and Deploy** workflow is based on a **template provided by Azure** and has been modified to integrate with the **Create Release** workflow.

3.4.2 Key Modifications:

- The workflow waits for the **confirmation** of the **"test step"** (now deleted) from the **Create Release** workflow before proceeding.

3.4.3 Workflow Steps:

1. **Setup Environment**:

- Sets up the .NET environment to build the application.

2. **Build the Application**:
   - Compiles the application for deployment.
3. **Deploy to Azure**:
   - The compiled application artifacts are deployed to the Azure Web App.

3.5 Summary of Automation Benefits

1. **Time Savings**:
   - Manual tasks such as creating releases, generating DLLs, and deploying services are now fully automated.
2. **Scalability**:
   - The matrix strategy in the `Make DLL` workflow supports multiple operating systems efficiently.
3. **Simplified Workflow Management**:
   - By focusing on maintaining workflows, developers can roll back code if issues arise, ensuring stability.

3.6 Team work

3.6.1 Project Board

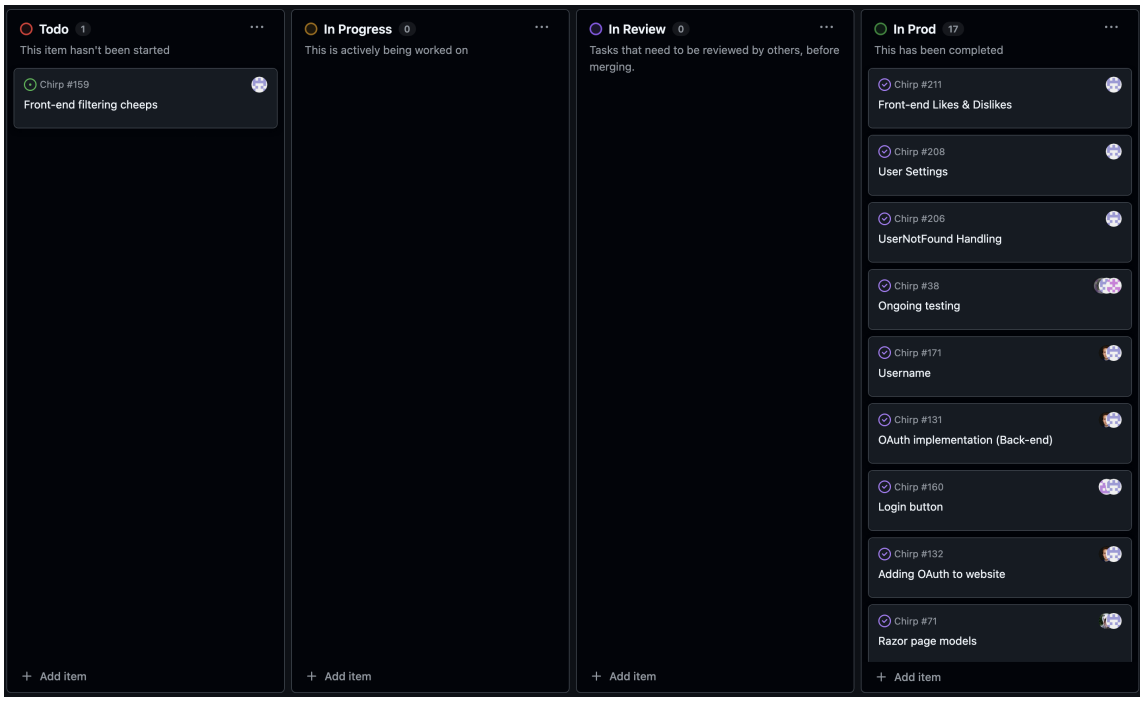

Figure 11: Current Project Board

Figure 2.2.1: Project board

This is an image of the *project board* before submission. The only issue which is incomplete prior to submission, is the "Front-end filtering cheeps". This issue refers to filtering what the user want to search by e.g. filter the search by email, author or cheep content. This issue was not a requirement, but instead an idea for extending the search function.

3.6.2 Process of Task to Implementation

Figure 12: Task to in main branch

Figure 2.2.2: Task to in main branch

Once given a task description, it is formulated into an *issue*. Once all tasks have been formulated into *issues*, they are then distributed to one or multiple contributors, depending on the assumed size of the issue.

Once the work on an issue has begun, the issue is moved from the `Todo` column to the `In Progress` column.

When the assigned contributors have decided that the requirements for the issue has been met, they create a pull request and move it into the `In Review` column. This allows other developers to read and review the pull request. Once there are no conflicts, the pull request is then accepted and the issue is moved into the `In Prod` column.

The given code which satisfies the original task has now been merged into the **main** branch.

3.7 How to make *Chirp!* work locally

3.8 Comprehensive guide to run the program locally

Please make sure you have all the right ***.Net 8*** dependencies installed here.

3.8.1 How to start the project on localhost via releases

1. Download the newest release for your operating system here.
2. Unzip the file, and navigate to the folder.
3. Run `$ dotnet Chirp.Web.dll`
4. Look in your terminal for which port the project is listening on. e.g.

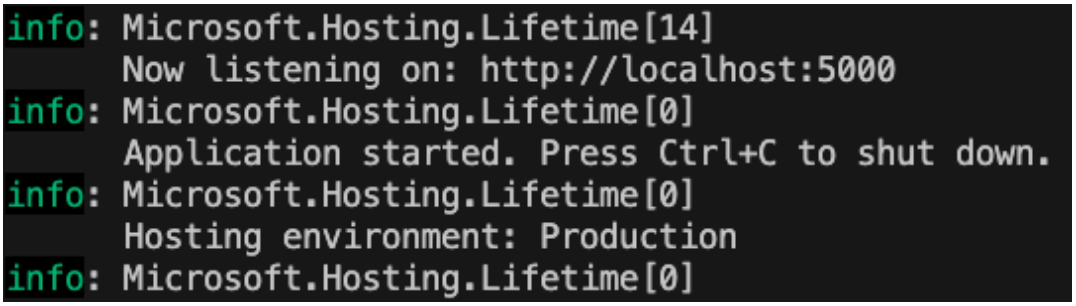

Figure 2.3.1: Local host

5. Open your browser and type `http://localhost:<port>`

3.8.2 How to start the project via cloning the repository

1. Open your terminal and type `$ cd`
2. Clone the repository and type `$ git clone https://github.com/ITU-BDSA2024-GROUP13/Chirp.git`
3. Type `$ cd ./Chirp`
4. Run the program `$ dotnet watch --project ./src/Chirp.Web`
5. Look in your terminal for which port the project is listening on. e.g.

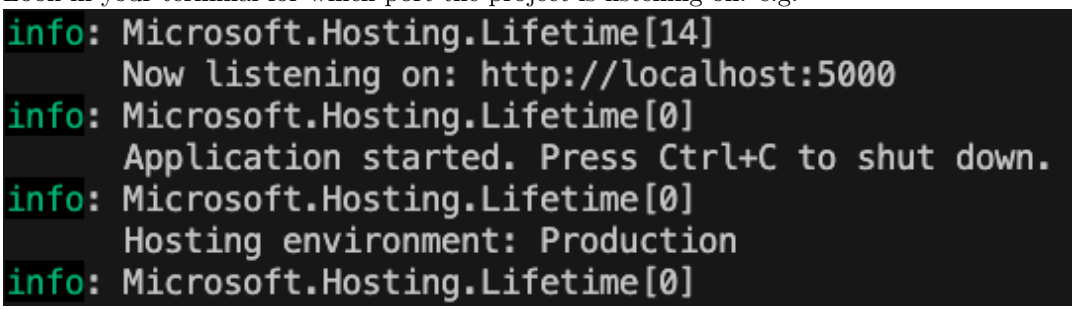

Figure 2.3.2: Local host

6. Open your browser and type `http://localhost:<port>`

3.9 How to run test suite locally

Figure 2.4.1: Code coverage report, using coverlet

The ***test*** package tests all the ***infrastructure*** and ***core*** using unit tests and integration tests.

The `web` package is tested via end-to-end tests using Playwright. Playwright does not provide code coverage.

In order to run the ***infrastructure*** and ***core*** tests:

go to the `Chirp\test` folder in your terminal.

Write `dotnet test` in your terminal to run all tests except Playwright tests.

Name	Covered	Uncovered	Coverable	Total	Percentage	Covered	Total	Percentage
Default	**1092**	**0**	**1092**	**1476**	**100%**	**18**	**18**	**100%**
C:\Users\Andre\Chirp\src\Chirp.Core\DTO\AuthorDTO.cs	4	0	4	14	100%	0	0	
C:\Users\Andre\Chirp\src\Chirp.Core\DTO\CheepDTO.cs	4	0	4	14	100%	0	0	
C:\Users\Andre\Chirp\src\Chirp.Core\Entities\Author.cs	3	0	3	33	100%	0	0	
C:\Users\Andre\Chirp\src\Chirp.Core\Entities\Cheep.cs	4	0	4	21	100%	0	0	
C:\Users\Andre\Chirp\src\Chirp.Infrastructure\Chirp.Repositories\AuthorRepository.cs	162	0	162	280	100%	6	6	100%
C:\Users\Andre\Chirp\src\Chirp.Infrastructure\Chirp.Repositories\CheepDBContext.cs	30	0	30	60	100%	2	2	100%
C:\Users\Andre\Chirp\src\Chirp.Infrastructure\Chirp.Repositories\CheepRepository.cs	65	0	65	103	100%	0	0	
C:\Users\Andre\Chirp\src\Chirp.Infrastructure\Chirp.Repositories\DbInitializer.cs	704	0	704	717	100%	4	4	100%
C:\Users\Andre\Chirp\src\Chirp.Infrastructure\Chirp.Repositories\HelperFunctions.cs	14	0	14	41	100%	0	0	
C:\Users\Andre\Chirp\src\Chirp.Infrastructure\Chirp.Services\CheepService.cs	102	0	102	193	100%	6	6	100%

Figure 13: Test coverage

If you want to see code coverage. Run `dotnet test /p:CollectCoverage=true /p:CoverletOutputFormat=lcov /p:CoverletOutput=lcov.info`

This should cover three packages: ***Chirp.Core***, ***Chirp.Repositories*** and ***Chirp.Services***.

In order to run the Playwright test, you have to:

3.9.0.1 Install the right dependencies

Make sure you have Node.js and npm (Node Package Manager) installed and/or updated. You can install Node.js from their website Node.js.

1. **update npm in a powershell terminal at the root of your pc**

```
$ npm install -g npm
```

2. **Verify you have them installed**

```
$ node -v
$ npm -v
```

3. **Install playwright package**

```
$ npm install -g playwright
```

4. **Move to Playwright folder**

```
$ cd ./testPlaywright/PlaywrightTests
```

5. **Install the Playwright Script**

```
$ pwsh .\bin\Debug\net8.0\playwright.ps1 install
```

6. ***In a new terminal*, start a server on the root folder in `Chirp`**

```
$ dotnet watch --project ./src/Chirp.Web
```

7. ***In the first terminal***

```
$ dotnet test
```

Once playwright is correctly installed you can go to the root folder of Chirp and write `dotnet test`. This will run all tests in the project.

3.10 Test suites

There are 8 test suites each focusing on different aspects of the solution. Following the **onion-architecture** allows the tests to focus on each layer individually using testing types such as *unit tests*, isolate a chain of method calls for *Integration testing* and *End-to-End (E2E) Testing*.

Test File	Unit Tests	Integration Tests	E2E Tests
AuthorTest.cs	yes	no	no
AuthorRepositoryTests.cs	yes	yes	no
CheepDBContextTest.cs	no	yes	no
CheepRepositoryTests.cs	yes	yes	no
HelperFunctionsTests.cs	yes	no	no
CheepServiceTest.cs	yes	yes	no
AzureTests.cs	no	no	yes
LocalTests.cs	no	no	yes

Tabel 1: List of the test suites and their types of testing

3.11 What is tested?

3.11.0.1 AuthorTests

Focus: Validation of the Author datatype and its behavior.

3.11.0.1.1 Types of Testing Unit tests: - Property validations on the behavior of the `Author` datatype such as its required fields.

3.11.0.2 AuthorRepositoryTests

Focus: Verifying the behavior of the repository pattern for Author.

3.11.0.2.1 Types of Testing Unit Tests: - Tests focused on individual repository methods like adding, retrieving updating and deleting authors.

Integration Tests: - Tests that validate repository methods against an in-memory database

3.11.0.3 CheepDBContextTest

Focus: Validating the setup and functionality of the database context.

3.11.0.3.1 Types of Testing Integration Tests: - Tests that involve actual database interactions, such as adding, retrieving updating and deleting authors, seeding and relationship checks.

Examples: - Testing `CheepDBContext` initialization. - Seeding the database correctly. - Validating migrations and Schema enforcements.

3.11.0.4 CheepRepositoryTests

Focus: Validating repository methods for managing Cheep ***Entities***.

3.11.0.4.1 Types of Testing Unit Tests: - Focused on verifying the behavior of repository methods like querying, filtering, or updating cheeps.

Integration Test: - Tests repository functionality against an a seeded mock database to ensure correctness with real data structures.

3.11.0.5 HelperFunctionsTests

Focus: Testing utility methods and reusable logic across the application.

3.11.0.5.1 Types of Testing Unit Tests: - Validates individual utility functions for correctness.

Examples: - Converting a unix-timestamp to a date in string. - Ensuring correct date formatting.

3.11.0.6 CheepServiceTest

Focus: Testing the business logic for cheeps at the service level.

3.11.0.6.1 Types of Testing Unit Tests: - Focus on the correctness of service methods with mocked dependencies.

Examples: - Input validation. - Business rules (e.g., maximum length)

Integration Tests: - Test the interaction of the `CheepService` with the repository and database.

3.11.0.7 AzureTests

Focus: End-to-end testing (E2E) on a live Azure-hosted application.

3.11.0.7.1 Types of Testing E2E Tests: - These validate the full application behavior in a live Azure environment, including user login, navigation, and interaction with web elements.

Examples: - `LoginTest` - `LoginChanges` - `LogOut`

3.11.0.8 LocalTests

Focus: Testing the application on a local development server.

3.11.0.8.1 Types of Testing E2E Tests: - Simulates user interactions with the application through Playwright.

Examples: - `LocalLogin` - `LocalLoginChanges` - `LocalLogOut` - `LocalShowingCheeps` - `LocalNavItems`

4 Ethics

4.1 License

This project is licensed under the **MIT license**. The **MIT license** was chosen on the basis that all of the other libraries used in this project are also under the **MIT license**, or are other open-sourced projects. Moreover, this project was solely made for academic purposes. Therefore, if any of this code would aid any others, although unlikely, there would be no reason to prohibit it.

4.2 LLMs, ChatGPT, CoPilot, and others

The *Large Language Models* (LLMs) which were used throughout the development process were: **ChatGPT**, **GitHub CoPilot** and **Codium**.

All three LLMs were used primarily for debugging. For the generation of most of the documentation, it was only **ChatGPT** which was used.

As a rule, whenever any of the LLMs generated any code which was used, it was co-authored in the commit where that piece of code was included. If an LLM was used simply for sparring to find the root cause of a bug, it was not included in the co-author message, unless it provided code to solve the bug.

In terms of the value of their responses, it varied. Sometimes, it was a small human error which was overseen, and the LLM helped discover it. In other more complex cases, it required a greater understanding of the code base which the LLMs, especially **ChatGPT** lacked. In these situations, the LLMs which are built in to the text editors, **GitHub CoPilot** and **Codium**, were able to gather more information, but were still not always able to solve errors. This may have lead to some spirals throughout the development process, of over-relying on an LLM to find a solution, taking a longer time to solve the problem.