Chirp! Project Report

ITU BDSA 2025 Group 9

Kristine Nielsen krini@itu.dk
Isabella Bjerre Wahl iswa@itu.dk
Nickolaj Toft Carman nitc@itu.dk
Mikkel Dybro Hansen midh@itu.dk
Marcus Alexander Ryssel Mader almm@itu.dk

IT UNIVERSITY OF COPENHAGEN

1 Introduction

This report documents the design, implementation, and collaboration process behind *Chirp!*, an application developed as part of the *Analysis, Design and Software Architecture* course project in 2025.

The purpose of the report is to describe the system's functionality, architectural decisions, and development workflow, as well as to provide clear instructions for running and testing the application locally.

2 Design and Architecture of *Chirp!*

This section describes the design of *Chirp!* at multiple levels, including the domain model, internal and deployed architecture, and user interaction flows. The section focuses on how the core concepts of the system are modeled, how responsibilities are structured across architectural layers, and how users interact with the application through user scenarios.

2.1 Domain model

The domain model of our *Chirp!* application represent the core concepts of our system. In the project, these are located in *Chirp.Core*.

The domain consist of three entities: *Author*, *Cheep* and *Comment*. An author represents a user in the system, who can create cheeps and comments, follow other authors and be followed. A cheep represents the message posted to our platform by a author and contains text, a timestamp and the author who created it. A cheep can have multiple comments. A comment contains text and are posted by a single author.

In the following diagram, the relation between these entities are clearly modeled through the lines between the entities. An author can create multiple cheeps, while a cheep only can have one author. A cheep can then also have multiple comments and each comment are associated to exactly one cheep and author. These relations reflect the core rules and structure of our application domain.

ASP.NET identity is used for authentication, which is an external part of our program, and therefore illustrated as an external dependency on the domain model. The *Author* entity inherits from *IdentityUser* allowing the domain to stay free of any authentication logic. This is also why we have to draw the ASP.NET identity in a package for itself.

In our *Chirp.Core* folder, we also store our DTO's. These are intentionally omitted from the domain model diagram since DTO's are not a part of the core domain concepts, but are used for transferring data.

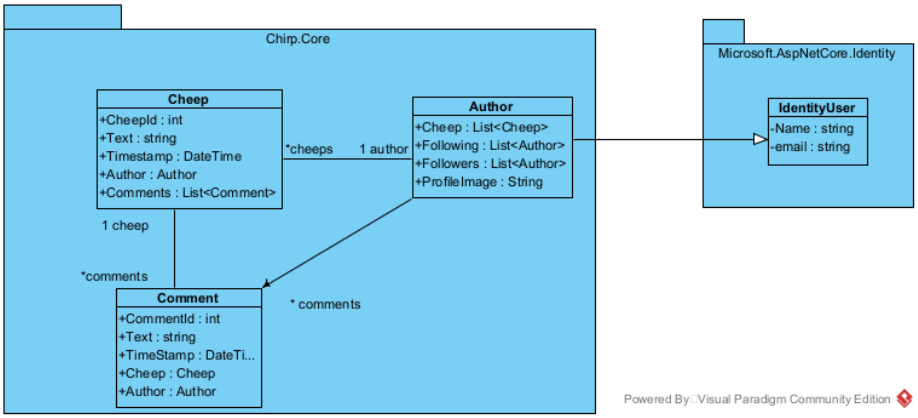

Figure 1: Illustration of the *Chirp!* domain model as UML class diagram

2.2 Architecture — In the small

Our *Chirp!* application is structured according to the *Onion architecture* pattern. In other words, our system architecture is organized into concentric layers, where external dependencies always point inwards, from the outermost layer to the center. The reasoning for using such an architecture design is that it can make the application easier to maintain and test (i.e. it allows for testing, replacing and modifying loosely coupled components separately).

The onion architecture can be divided into the following layers: *domain*, *repository*, *service* and *UI*. In our implementation, these layers are reflected in the structure of the code. That is, the application is split into separate projects and directories, namely *Chirp.Core*, *Chirp.Infrastructure*, and *Chirp.Web*, as well as a separate test directory containing *Chirp.Tests* and *Chirp.PlaywrightTests*.

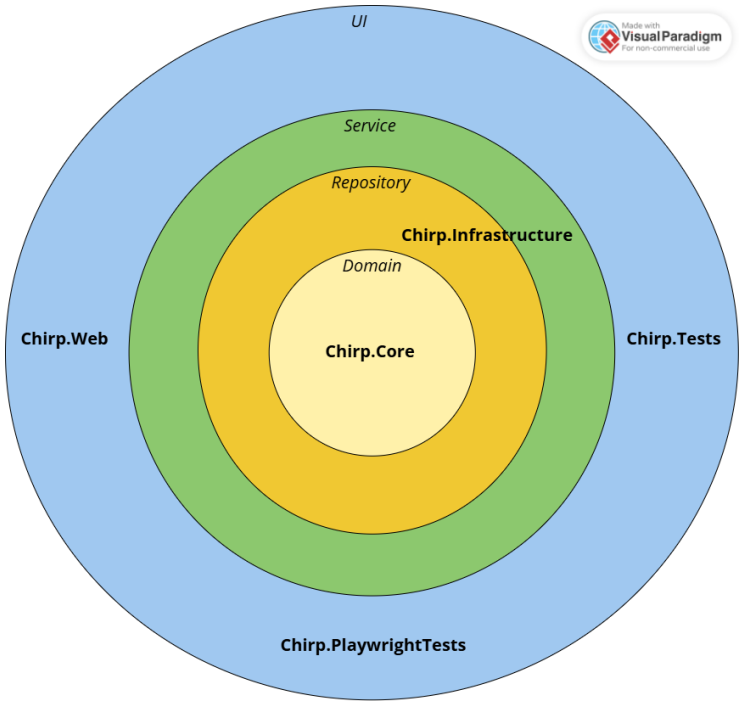

Figure 2: Illustration of the *Chirp!* system architecture as an onion architecture diagram. Made using *Visual Paradigm Online* tool for *Circle Maps*.

The *domain* layer is the innermost layer and contains our domain entities, such as *Author*, *Cheep* and *Comment* - as well as their corresponding DTOs. These do not have any external dependencies, and therefore act as the inner core, *Chirp.Core*, of our application. The *repository* layer is part of *Chirp.Infrastructure* and is responsible for data access. It handles interaction with the database, including retrieving, storing, and mapping domain data. The *service* layer, also located in

Chirp.Infrastructure, contains the application services and acts as an intermediary between the UI layer and the repository layer. The UI layer is the outermost layer and is responsible for the user interface, *Chirp.Web*, and contains the testing infrastructure through *Chirp.Tests* and *Chirp.PlaywrightTests*.

The following diagram illustrates this structure as a package diagram, where each layer is represented using a distinct color. Blue represents the *UI* layer, green the *service* layer, orange the *repository* layer, and yellow the *domain* layer. To keep the diagram readable and focused on the architectural structure, not all directories and files are shown. It shows how the code is organized across layers following the onion architecture, omitting individual inner files, such as for example *AuthorService*, *CommentService*, and *CheepService* within the *Service* directory. The same applies to the other innermost directories shown in the diagram.

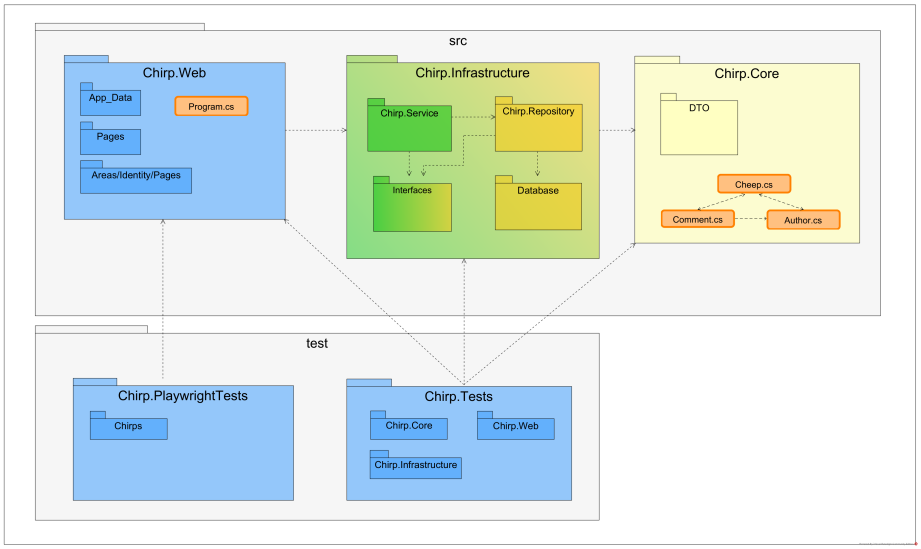

Figure 3: Package diagram illustrating the organization of *Chirp!* across the onion architecture layers.

2.3 Architecture of deployed application

The following two diagrams illustrates the deployed architecture of our first version of *Chirp! "Chirp.CLI"*, and our final version of *Chirp! "Chirp.Web"*. The diagrams illustrates the communication flow between the user, i.e. client, and the deployed application.

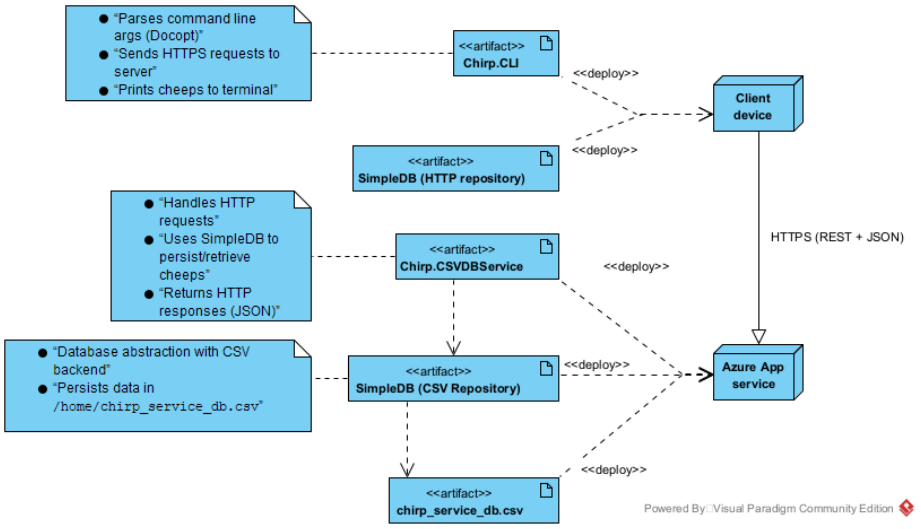

Figure 4: Illustration of the deployed architecture of the *Chirp.CLI* application.

The above diagram show, how *Chirp.CLI* allows the user to make use of commands to read existing cheeps and post new cheeps, which are sent as HTTPS requests to the backend service. The diagram also illustrates the server-side persistence architecture, where the *CSVDBService* depends on the *SimpleDB* repository, which in turn persists data in the *chirp_service_db.csv* file. This shows how data flows from the client through the server and into storage.

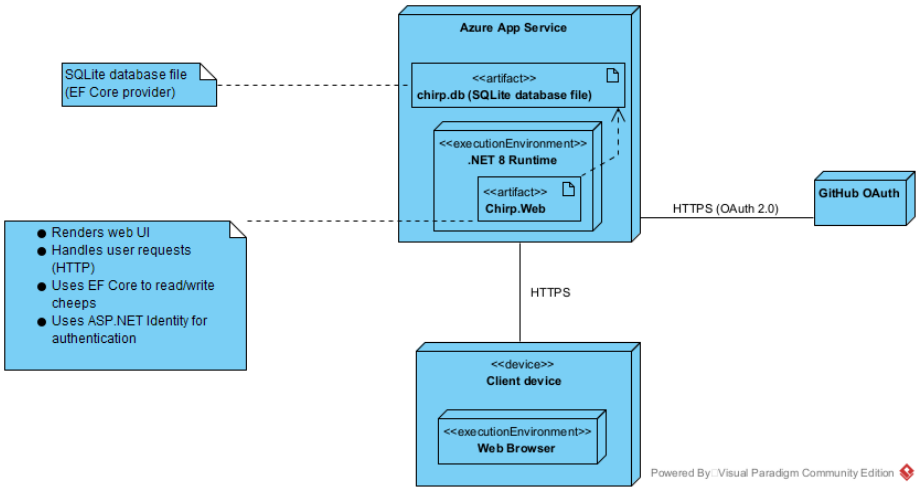

Figure 5: Illustration of the deployed architecture of the *Chirp.Web* application.

The diagram illustrates the deployed architecture of *Chirp.Web*. Users access the system through a web browser, which communicates with the *Chirp.Web* application hosted on an Azure App Service via HTTPS. The web application is responsible for rendering the user interface and handling incoming HTTP requests. The application uses EF Core with a SQLite database file (*chirp.db*) for data storage, and supports authentication via GitHub OAuth. This diagram shows how the client, server, storage, and authentication components interact in the deployed system.

2.4 User activities

In the following section, three UML activity diagrams illustrate typical user scenarios and the user journey through our *Chirp!* application, starting from a non-authorized user and ending with an authorized user.

The first diagram shows what a user can do when they are *not logged in*. In this state, the user can view the public timeline, view the timelines of other users by clicking on their usernames or decide to authenticate by viewing the login or register page.

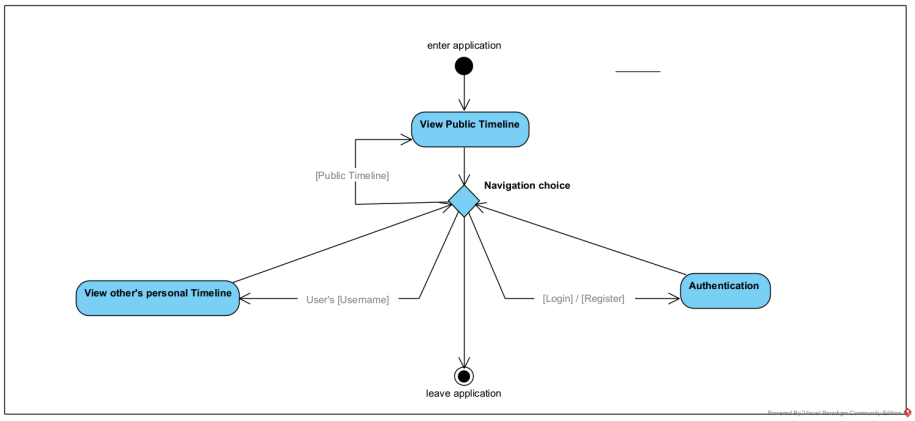

Figure 6: UML activity diagram illustrating the user journey of a non-authorized user through *Chirp!*

If the user decides to log in, they must choose between logging in with an existing account or registering a new one. Both options can be done using either a local account login or an external GitHub login. This is illustrated in the second diagram. After successful authentication, the user is logged in and returned to the public timeline as an authorized user.

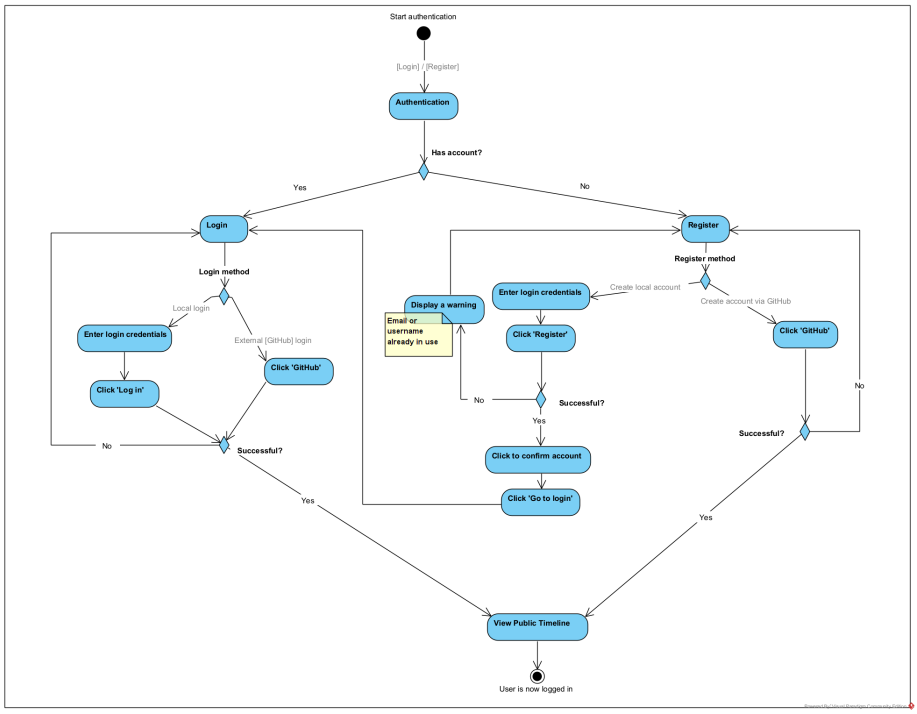

Figure 7: UML activity diagram illustrating the authentication flow in *Chirp!*

The final diagram shows what an authorized user can do, i.e. when the user is *logged in*. An authorized user can view the public timeline, view their own timeline, and view the timelines of other users. From each of these, the user can follow or unfollow other users, post or delete cheeps, unfold collapsed comments, comment on cheeps, and delete their own comments. An authorized user can also choose to view their own personal about-me page or log out of the application. From the about-me page, the user can change their profile picture, download their personal information, or delete their account.

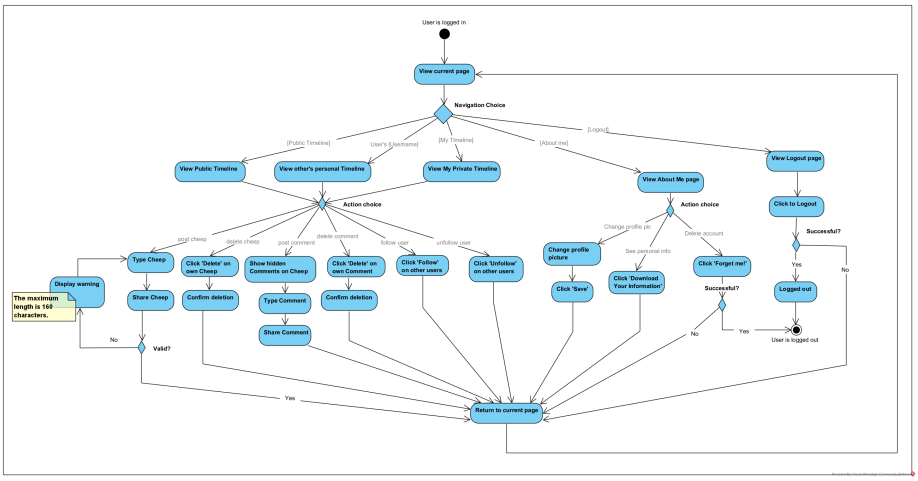

Figure 8: UML activity diagram illustrating the user journey of an authorized user in *Chirp!*

To keep this diagram simple and readable, the action *"Return to current page"* represents that, after completing an activity, the user is returned to the page they were previously viewing. For example, if a user posts a cheep while viewing their own timeline, they will either stay or return to their own timeline once the action is done.

2.5 Sequence of functionality/calls through *Chirp!*

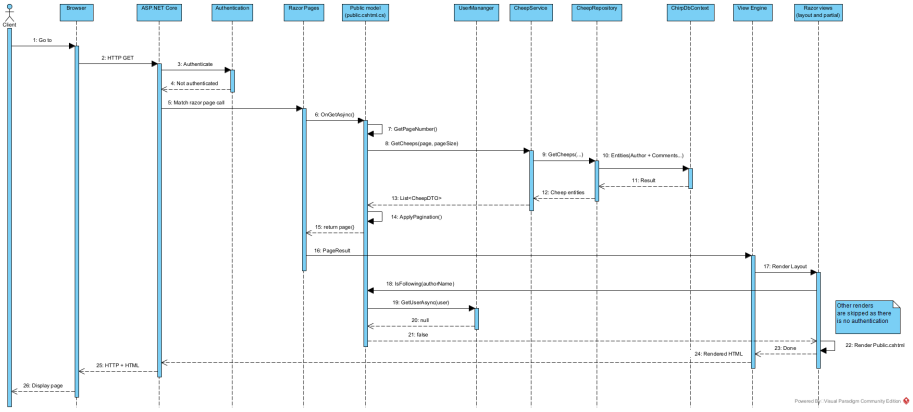

Figure 9: UML Sequence diagram over unauthenticated user through *Chirp!*

This diagram shows the various calls and replies the application makes throughout the process of rendering the page upon a HTTP request. As the user is unauthorized, some calls have been omitted for simplicity (as stated in the note). These would be the cheep box or the personal timeline. The calls to static elements like images or css have also been omitted.

3 Process

This section describes the processes used during the development of *Chirp!*, covering build, test, release, and deployment workflows, as well as team collaboration and development practices. The section outlines how automation was used to ensure consistent builds and deployments, how work was organized and tracked within the team, and how the application can be built, tested, and run locally.

3.1 Build, test, release, and deployment

The following UML activity diagram shows the process of building, testing, releasing, and deploying *Chirp!* to our GitHub and Azure Web App. In our project, we have three distinct workflows: a *release* workflow for publishing the application, a *test* workflow for building and testing the application, and a *deploy* workflow for deploying the application to Azure.

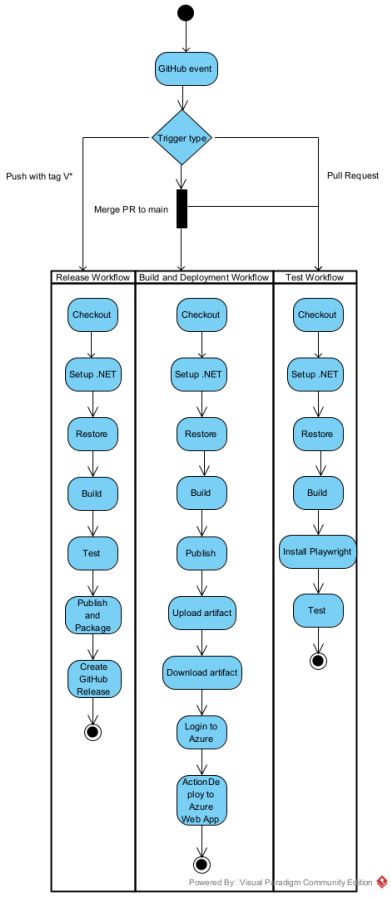

Figure 10: Illustration of the *Chirp!* event process as UML class diagram.

The workflow that is triggered depends on the type of action performed in the repository. A push with a tag named `v*`, where `*` represents the release version (e.g., `v1.0.1`), triggers the *release* workflow. Creating a pull request triggers the *test* workflow, while accepting and merging a pull request into the main branch triggers both the *test* and *deploy* workflows.

Once triggered, each workflow executes its own set of straightforward steps and does not depend on the others. All workflows follow a similar structure, where they set up, restore and build the project, before handling their respective independent tasks.

3.2 Team work

3.2.1 Project Board Status

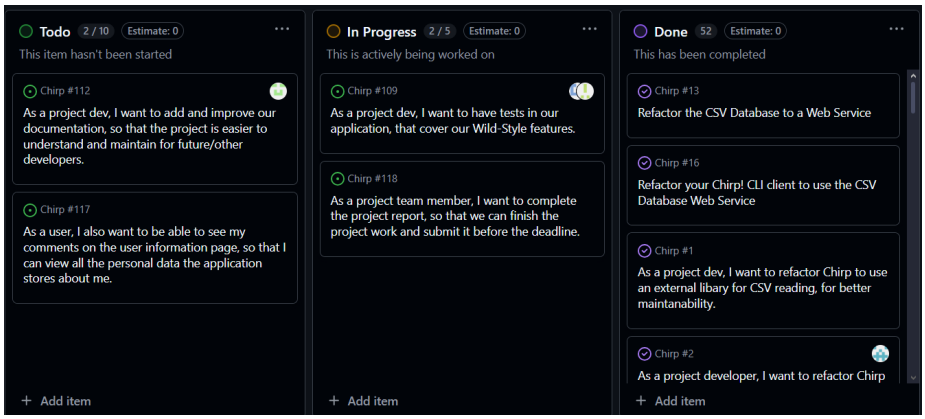

Figure 11: Screenshot of the project board

The screenshot above shows the project board status shortly before hand-in. The board is organized into three columns: *Todo*, *In Progress*, and *Done*, representing the current state of each task.

At the time of hand-in, the majority of tasks have been completed and placed in the *Done* column. These include major refactorings, maintainability improvements, and completed user stories related to the core functionality of *Chirp!*.

However, a small number of tasks are still, as of now, unresolved:

Chirp #112 - **Improve and extend documentation** - This task concerns further improving and polishing the project documentation. While the most critical documentation is present, additional refinements for long-term maintainability were not completed, before this screenshot was taken.

Chirp #117 - **Show user comments on the user information page** - This feature would allow users to see all their own comments on their 'about me' page. The functionality was planned but not fully implemented.

Additionally, two tasks were still *In Progress*:

Chirp #109 - **Add tests for Wild-Style features** - Some tests were implemented during development, but comprehensive coverage of all Wild-Style features have not yet been completed, as these features still need to be covered by Playwright tests.

Chirp #118 - **Complete the project report** - This issue represents the final writing and polishing of the project report, which was still ongoing at the time, the screenshot was taken.

All core application functionalities have been implemented, while the remaining unresolved tasks mainly concern documentation, additional tests, and minor feature extensions.

3.2.2 Development workflow

Figure 12: Illustration of development process from issue creation to main branch merge

1. **Issue creation** - New work criteria were created as GitHub issues and formulated as short user stories. Each issue described a concrete task, feature, or refactoring goal.
2. **Task planning and prioritization** - Issues were added to the project board and initially placed in the *Todo* column. During planning, tasks were discussed and prioritized based on importance and dependencies.
3. **Development phase** - When work on a task started, the corresponding issue was moved to *In Progress*. Development was performed on a separate *feature branch*, ensuring that unfinished or experimental code did not affect the main branch.
4. **Completion and review** - Once a task was implemented and tested locally, a pull request was made and the corresponding branch was merged into the main branch after a review, ensuring that the main branch always contained a working version of the system. The issue was lastly marked as *Done* on the project board.

Larger tasks (such as refactoring or architectural changes) were sometimes split into smaller follow-up issues. This allowed incremental improvements without blocking overall progress.

This workflow represents how most of the project work was structured. However, it was not followed strictly at all times - in particular, during the early stages of the project. Because group members often worked closely together — either physically or online — pull requests were not used as a forum for code discussion. Team members were typically already aware of the changes being made, which meant that reviews often consisted of brief comments (e.g., "Looks good") rather than actual discussion. Nevertheless, the use of pull requests was still useful, as it allowed us to prevent code with failing tests from being merged into the main branch. Since much of the work was done collaboratively, we also sometimes forgot to add each other as co-authors on commits, even when the work was done jointly.

3.3 How to make *Chirp!* work locally

This section describes the exact steps needed to get *Chirp!* running on a fresh machine, including required tools, configuration, and what you should expect to see.

3.3.1 Prerequisites

Before cloning the project, ensure the following tools are installed:

- **Git**
- **.NET SDK 8.0**
- A modern web browser
- *(Optional but recommended)* Visual Studio or Rider

  The application uses **SQLite**, so no external database server is required.

Check that .NET is installed with:

```
dotnet --version
```

3.3.2 Clone the Repository

Open a terminal and execute:

```
git clone https://github.com/ITU-BDSA2025-GROUP9/Chirp.git
cd Chirp
```

3.3.3 Restore Dependencies and Build

From the repository root, run:

```
dotnet restore
dotnet build
```

Expected outcome - All NuGet packages are restored - The solution builds successfully with no errors

3.3.4 Trust the HTTPS development certificate (first-time only)

Our *Chirp!* application runs on HTTPS locally. On a fresh machine, trusting the dev cert avoids browser warnings and OAuth callback issues:

```
dotnet dev-certs https --trust
```

3.3.5 Configure GitHub OAuth

Chirp! uses GitHub authentication and reads two configuration values:

- `authentication_github_clientId`
- `authentication_github_clientSecret`

These values are required to enable GitHub login. Without them, authentication via GitHub will not be possible. Including our own development secrets in a public repository is considered a bad practice, therefore, these values are not provided. Thus, to enable GitHub authentication in a local environment, a GitHub OAuth App must be created and its secrets configured accordingly. Otherwise, GitHub authentication is available through our deployed application on Azure.

3.3.5.1 Create a GitHub OAuth App

Go to GitHub:

Settings → Developer settings → OAuth Apps → New OAuth App

Create an app with:

- *Homepage URL:*

`https://localhost:xxxx` (use the HTTPS URL printed in the terminal output)

- *Authorization callback URL:*

`https://localhost:xxxx/signin-github`

Copy the generated Client ID and Client Secret.

3.3.5.2 Set credentials locally using .NET User Secrets

From the repository root:

```
dotnet user-secrets set "authentication_github_clientId"
"<CLIENT_ID>" --project src/Chirp.Web

dotnet user-secrets set "authentication_github_clientSecret"
"<CLIENT_SECRET>" --project src/Chirp.Web
```

(Optional) verify secrets:

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

Expected output includes:

- `authentication_github_clientId = ...`
- `authentication_github_clientSecret = ...`

3.3.6 Run the web application

From the repository root:

```
dotnet run --project src/Chirp.Web
```

Expected terminal output includes that the app is listening on URLs similar to:

- `https://localhost:7140`
- `http://localhost:5198`

Open the HTTPS URL printed in the terminal output.

3.3.7 What you should see

When the app is running, and you open the site:

- The site loads without a crash.
- You can view pages such as the public timeline.
- You can click Login and authenticate via GitHub.
- After login, you should be redirected back to *Chirp!*.

3.3.8 Database behavior

Our *Chirp!* application uses SQLite and applies migrations automatically on startup. A database file is stored under the web project directory:

`src/Chirp.Web/App_Data/chirp.db`

If the database does not exist, it will be created. If migrations exist, they will be applied automatically.

3.4 How to run test suite locally

3.4.1 Prerequisites

- **.NET 8.0**
- **Playwright browsers**

To run all tests, run this from the repository root *Chirp/*:

```
dotnet build
dotnet test
```

To run individual tests, you may choose the specific test directory after *dotnet test*:

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

3.4.2 Installing playwright browsers locally

If needed, you may install the playwright browsers with the following command:

```
pwsh bin/Debug/net8.0/playwright.ps1 install
```

If you do not have powershell on your local machine, you must acquire it.

3.4.3 Description of test suites

The test suite is split into two main directories. The Unit tests and Integration tests are located in the directory *Chirp.Tests*, while the end-to-end tests are located in the *Chirp.PlaywrightTests* directory. The unit tests are directly focused on workings within the application and are organised according to which class their tests refer to. A few examples of architecture tested by these are: the DTO's, active deletion, or correct interaction between the different entities.
Integration tests lie in the subdirectory *Chirp_Web* and test for the correct rendering of the web application without using a browser, through the ASP.NET Core (WebApplicationFactory). These tests both check for the existence of implemented features like comments, but is also used to ensure that the pagination works as expected.

End-2-end tests are used to ensure that the behaviour of a generic user works as intended on the site. Thereby, simulating small targeted user flows, such as logging in, posting a cheep, viewing the personal timeline, and so on. Unlike the unit and integration tests, the end-to-end tests run against a real running instance of the application and its database. This database is isolated from the actual applications database, so there is no pollution from the test to the project's database.

4 Ethics

This section addresses ethical considerations related to the development of *Chirp!*, focusing primarily on licensing choices and the use of Large Language Models (LLMs) during the project. We describe the reasons behind selecting an open-source license and reflect on how LLMs were used as development tools, including both their benefits and limitations, as well as considerations regarding transparency and responsibility.

4.1 License

We chose an MIT License for our project. An MIT License leaves many opportunities for others to use our project, while still giving us credit for our work without creating any obligations towards us. We felt this fit the project best, as it matches the context in which the project was developed. That is, it was created in an educational context, and we therefore, as a group, believe that others also should be allowed to use, read and learn from our code. The license is simple, permissive, and compatible with the dependencies in our *.csproj* files. For these reasons, we consider the MIT License to be the best fit for this project, whereas a more restrictive license would likely be more appropriate for an actual commercial product or application.

4.2 LLMs, ChatGPT, CoPilot, and others

During the development of our project, Large Language Models (LLMs) were used for various parts of the project, primarily ChatGPT. Although LLMs were used throughout the project, we were not always fully transparent about their use and did not consistently add, for instance, ChatGPT as a co-author to all commits, where it was actually used. The primary reason for this was that at the beginning of the project, we were not aware of this requirement. Once we were informed, we began to apply it to our commits. However, there were still instances where we forgot to add it as a co-author, particularly when the use of LLMs was minimal and not a significant part of the changes in a commit.

The primary use of LLMs during our project was for error handling and debugging. There were many situations in which we encountered that changes to the code caused significant parts of the system to break. As a result, we had to identify where the issues had occurred, and how to resolve them, sometimes while dealing with a large number of exceptions simultaneously. This was not always straightforward, and LLMs were particularly helpful in such situations. In these cases, we would, for instance, provide ChatGPT with the broken code and the associated exceptions and prompt it to suggest how to handle these. This was for the most part useful, when frustration had begun to build, and we could not resolve the issue on our own, as it often provided us with insight into our mistakes. In many instances, the underlying problems were caused by simple or easily overlooked coding mistakes. Therefore, the LLMs offered us an additional perspective that helped us resolve these issues efficiently.

A second use of LLMs was for direct code generation. This was, however, used in moderation and primarily for inspiration and guidance on more confusing or challenging tasks, allowing us to incorporate parts of the generated code while still implementing the final solution properly on our own. Lastly, we also decided to use ChatGPT for image generation, creating the profile pictures in our project that depict different colored birds. We chose this, because we wanted images that fit the bird-themed concept of *Chirp!* while maintaining a clean user interface design, and since none of us had the necessary skills nor time to create these images, using ChatGPT provided us with a fast and easy solution.

The advantage of LLMs is that they provide a useful tool when one is uncertain or stuck, offering guidance on how to handle it, and since they are always available, they can help more quickly than e.g. a teaching assistant, who is limited by time and day. Therefore, LLMs can in some cases speed up development.

However, LLMs can sometimes also be misleading. We found that, since the LLM lacks full knowledge of the project that only we as the developers possess, it occasionally provided incorrect guidance, which slowed our development down rather than sped it up. A significant example of this occurred during the deployment of our application to Azure. For a long time, we believed that our GitHub Action workflow was the reason the application would not deploy automatically and display the new content, after integrating the EF Core database. Following guidance from ChatGPT, we spent considerable time adjusting the workflow, only to later discover that the actual issue was a missing startup command and an outdated database in Azure (See GitHub Issue #24).

In conclusion, we found that LLMs, when used appropriately, can significantly speed up the development process. However, they should be used in moderation, and while convenient, they can also introduce inefficiencies if completely and uncritically relied upon.

5 Conclusion

In this project, we developed *Chirp!* as a functioning web application with core social features such as user timelines, posting, and following.

The development process was organized using GitHub issues and a project board, allowing tasks to be tracked from initial description through implementation and integration.

Overall, the project demonstrates the use of collaborative development practices, version control, and structured workflows in the implementation of a web application.

6 Appendix

6.1 Project links

- *GitHub Repository:*
  https://github.com/ITU-BDSA2025-GROUP9/Chirp
- *Azure Web App:*
  https://bdsagroup9chirprazor-ekabaabpd3g4cvg6.swedencentral-01.azurewebsites.net/