

Chirp! Project Report

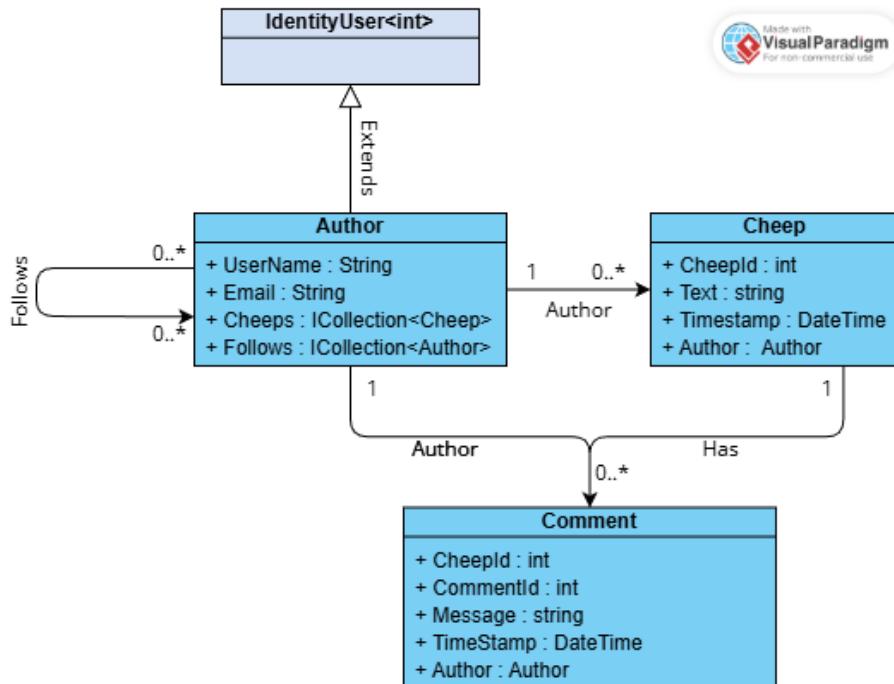
ITU BDSA 2025 Group 03

Alexandra Petersen alyp@itu.dk Elisa Esgici elie@itu.dk
Frederik Juul freju@itu.dk Frederik Schmidt frvs@itu.dk
Maryna Lvova mlvo@itu.dk Yasmin Nielsen yasn@itu.dk

1 Design and Architecture of *Chirp!*

<https://github.com/ITU-BDSA2025-GROUP3/Chirp>

1.1 Domain model



The domain model above illustrates the core entities of our Chirp! application. Author extends ASP.NET Identity for authentication and represents users of the system. Cheep represents users posted messages and Comment allows users to respond underneath Cheeps.

1.2 Architecture — In the small

The following illustration shows the onion architecture of the *Chirp!* application. It should be noted that during the course, there has been given several different and contradicting statements between lecturers, TAs, slides and weekly project descriptions about which parts of the code must reside in which parts of the onion architecture's layers. Thus the illustration is a synthesis of these different sources, mostly influenced by recommendations from TAs. Furthermore, note that tests are not included in the following diagram to minimise visual overload, however they can be viewed as their own transient layer, having dependencies to the different parts of the code base for the different tests.

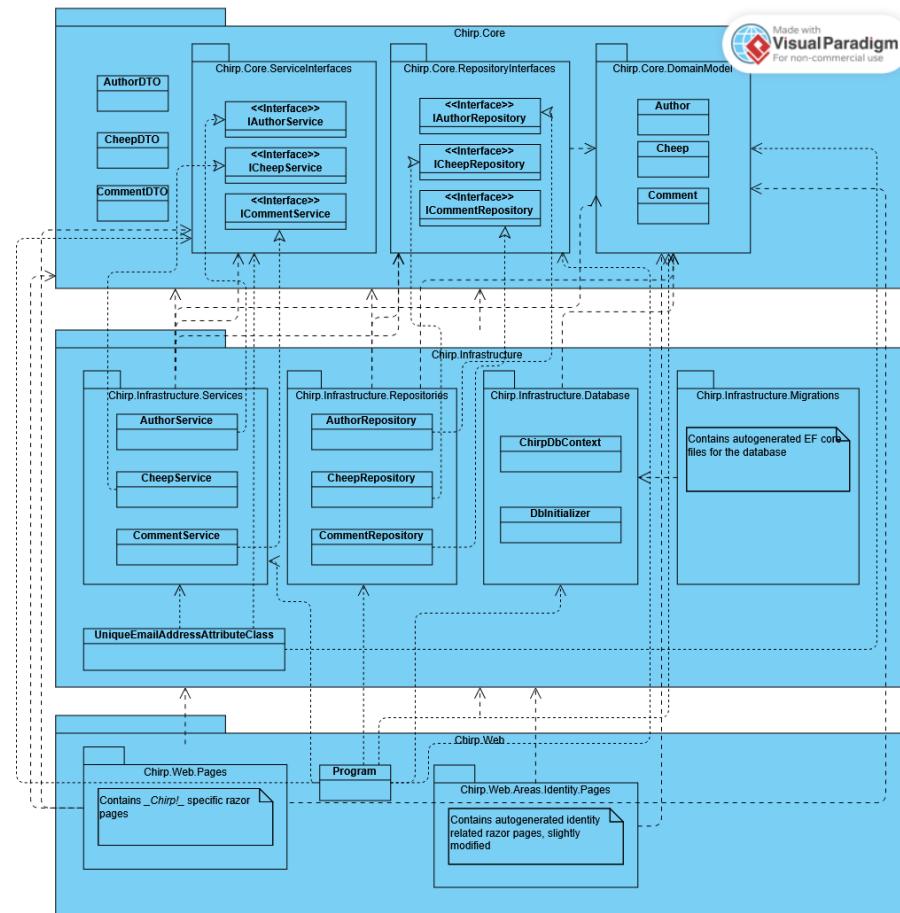
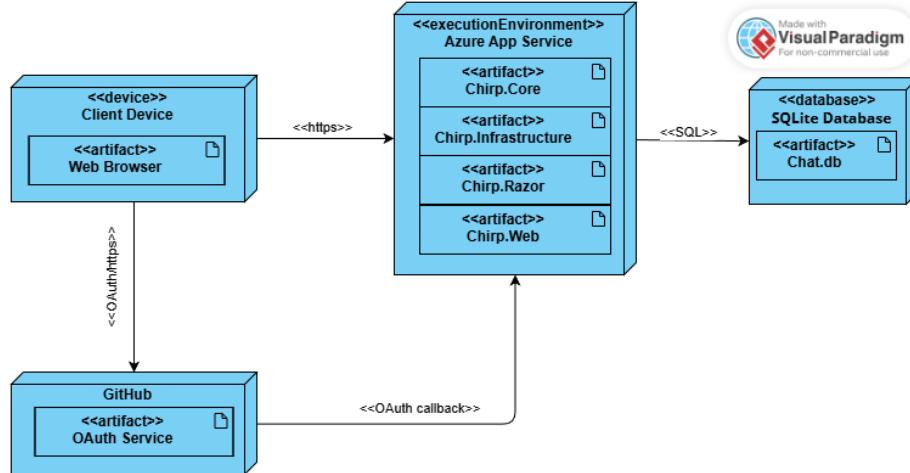


Figure 1: Onion

1.3 Architecture of deployed application

This deployment diagram illustrates the architecture of our deployed *Chirp!* application. Below we can see that the client device runs on a web browser that communicates with the server via HTTPS. The server component is deployed to Azure app service that hosts the server side application. The server communicates with a SQLite Database.

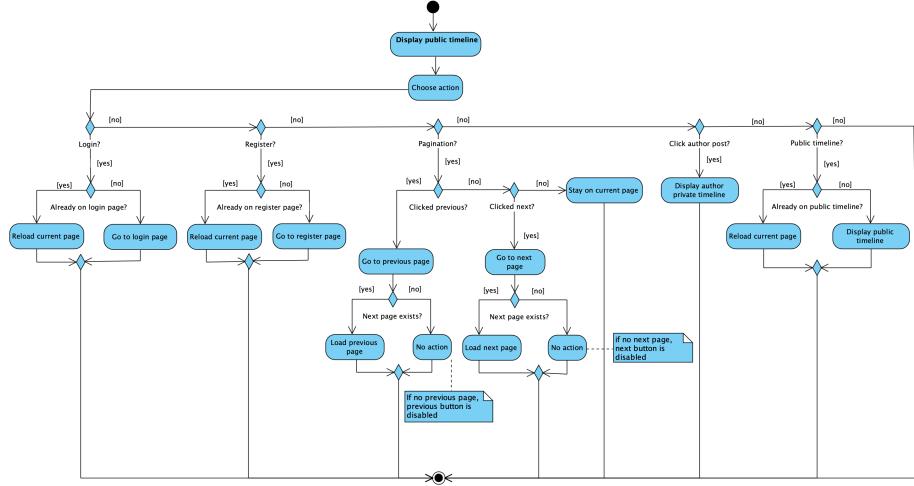


User activities The activity diagrams model the flow from one action to another, in accordance to the application's actual behaviour. The UML diagrams visualize how operations correlate, occasionally overlap and requires coordination.

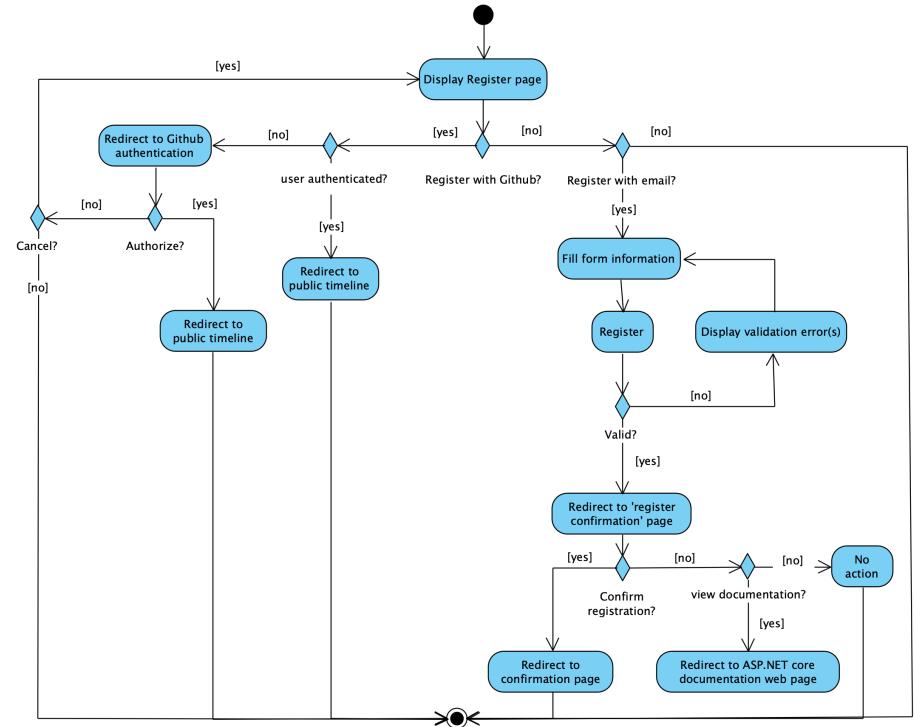
1.3.1 Unauthorized user

The following three scenarios showcase typical user journeys through our application for unauthorized users. Their actions are limited in comparison to authorized users. Unauthorized users can register or login and can view their own private timeline or that of other authors.

Scenario A: Unauthorized user - public/private/author timeline



Scenario B: Unauthorized user - register user



Scenario C: Unauthorized user - login user

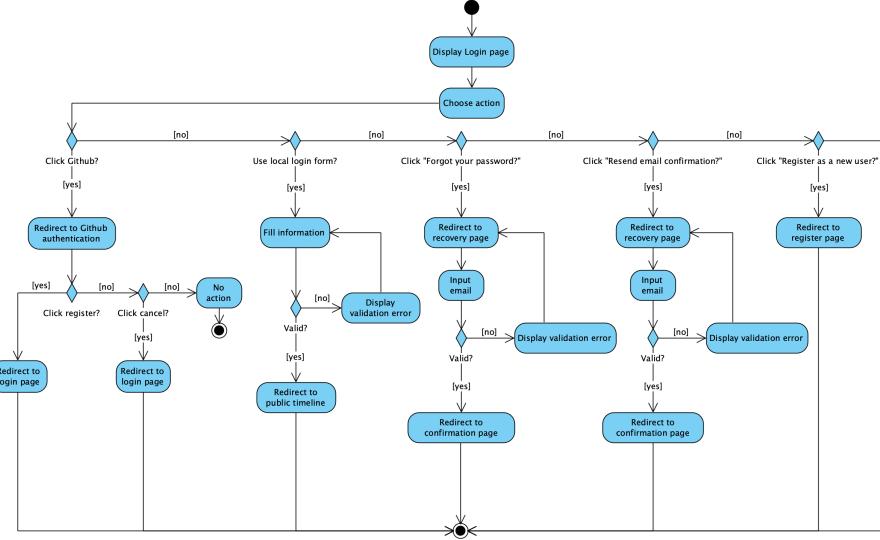
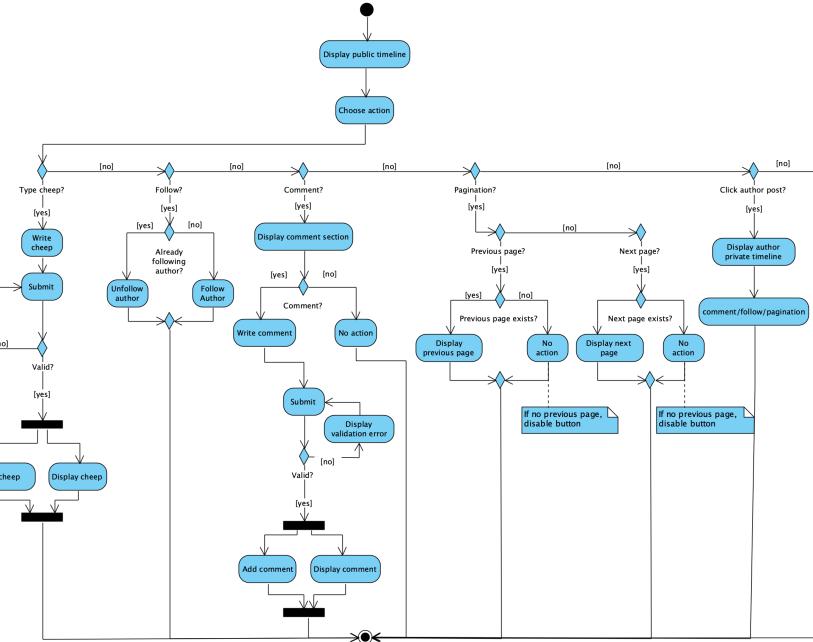


Figure 2: Scenario C unauthenticated - login

1.3.2 Authorized user

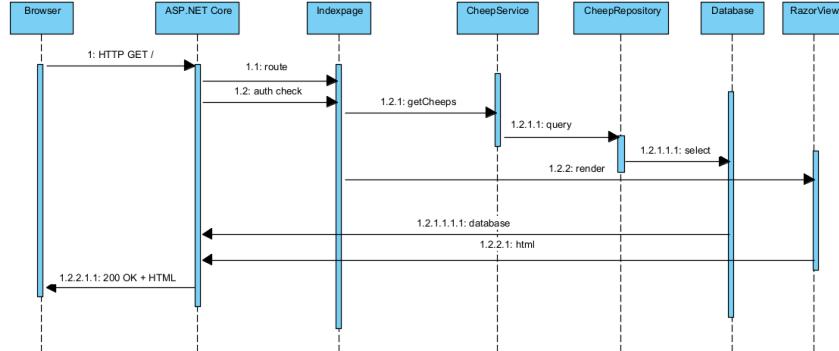
The following two scenarios showcase the possible user interactions of authorized users. In this state a user can, in addition to unauthenticated user actions, cheep new messages, comment on other author's posts, follow each other and access a ‘my information page’ that displays all user information. The same functionality is possible on private/author timeline, but is abstracted away for simplicity.

Scenario A: Authorized user - public/private/author timeline actions



Scenario B: Authorized user - navigation bar actions

1.4 Sequence of functionality/calls through *Chirp!*



The sequence diagram above shows how a request moves though the *Chirp!* application. It start when an unauthorized user opens the root page in their browser. The request is handled by the ASP.NET core, which routes it to the IndexPage and checks whether the user is allowed to access the page where requests continues. The indexPage then retrieves the data though the service and repository through the database. Once the data has been collected it is then send to the RazorView, where is it rendered into a complete HTML page. This HTML page is then, at last, returned to the browser as an HTTP response

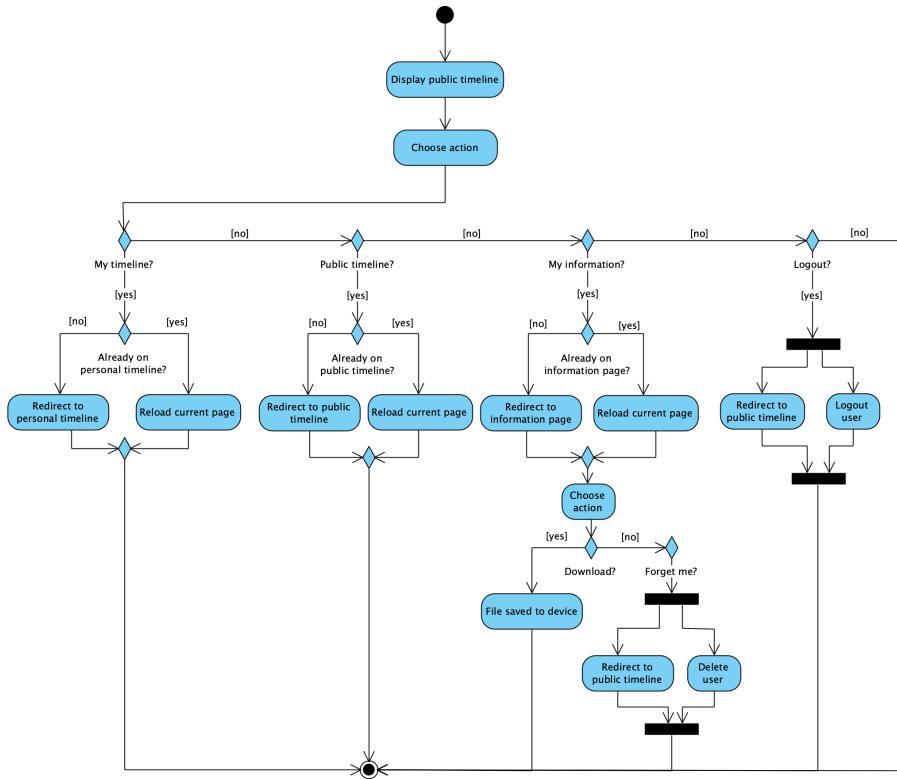


Figure 3: Scenario B authorized - navigation

and displayed to the user.

2 Process

2.1 Build, test, release, and deployment

Build and testing pipeline

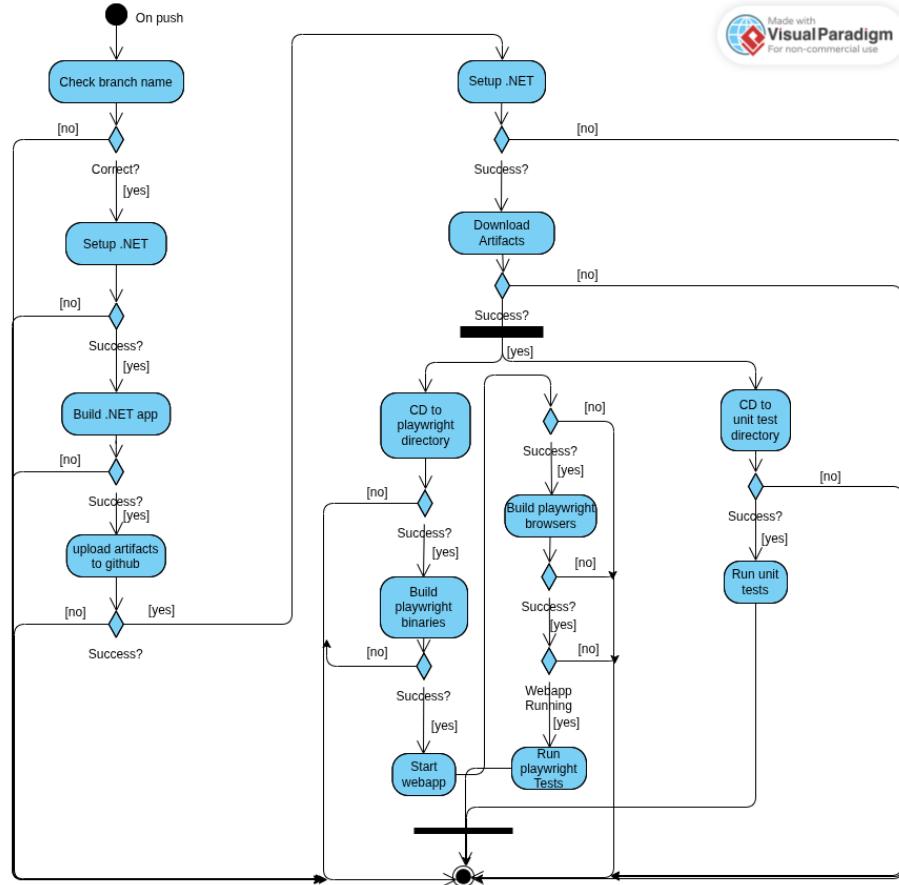


Figure 4: Build & Test Pipeline

Release Pipeline

To reduce redundancy for the release pipeline and avoid confusion, we have decided all OS's in the matrix will assume the same logic as linux-x64 on the far left.

Deployment pipeline

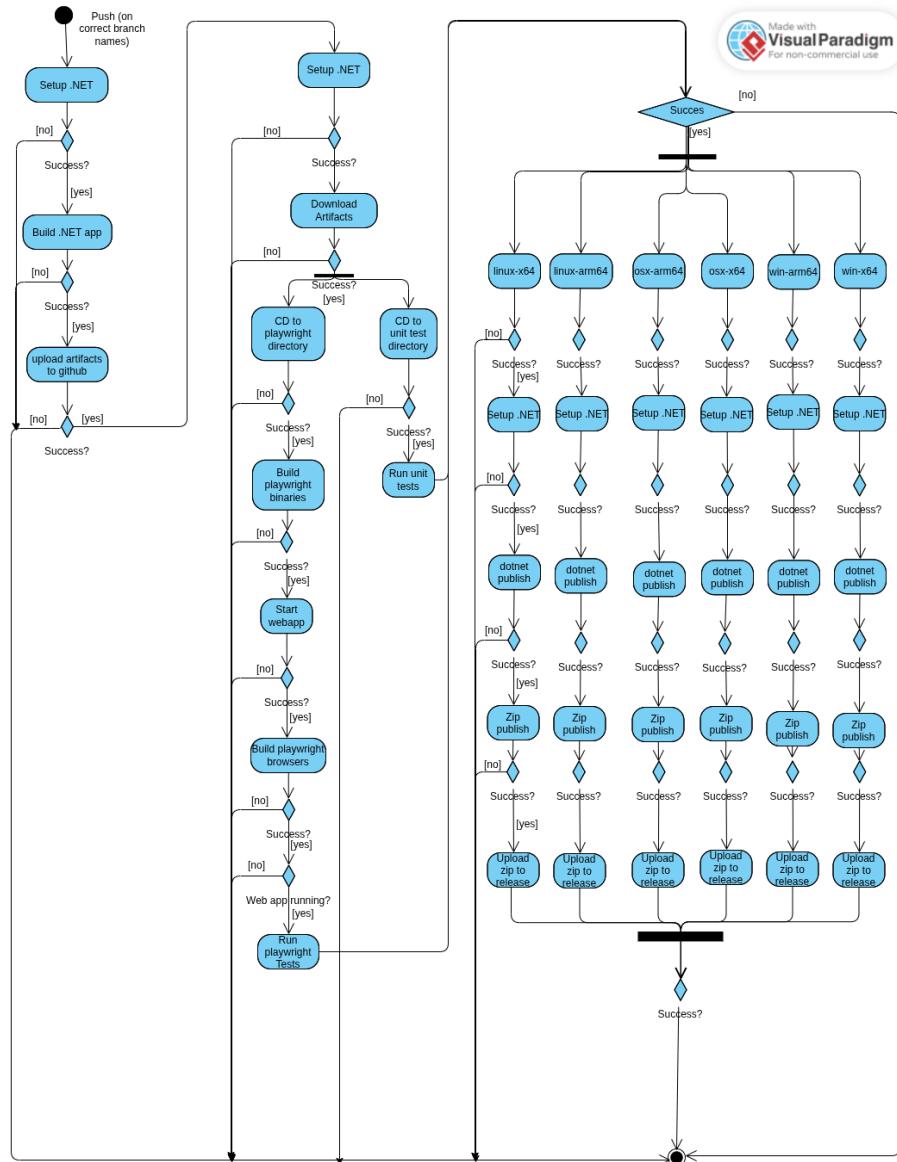
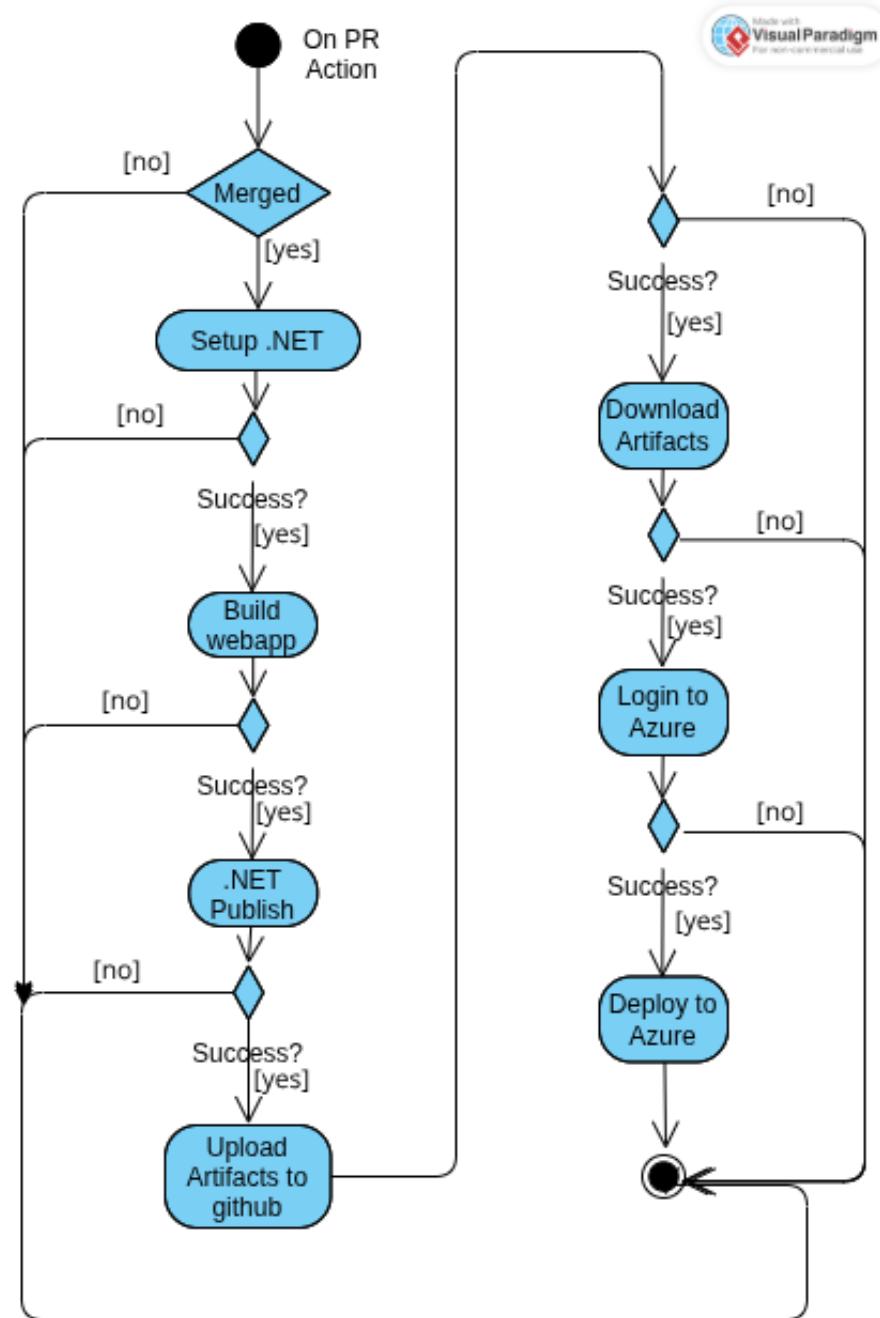


Figure 5: Release Pipeline



Throughout the project, github action pipelines were used to ensure automation and higher quality of code. These workflows are shown above as UML diagrams, displaying the journey from trigger to either the build stage, the test stage, the

release stage or the deployment stage. All of these stages are crucial for rapidly evolving programs and are, as in the name, continuously used and improved upon.

Our strategy for integration and deployment pipelines is simple. When code is pushed to a specific branch with a specific name, ensure the code can be built into an application and that all tests are still working. When you want to merge code into production (and thus trigger the deployment pipeline), it goes through all tests again to ensure no code has been merged between the initial creation of the PR and when the PR gets merged (if it gets merged of course).

Releases were something we wanted to optimise and automate more, but was ultimately made into a more manual approach which involved creation of tags triggering the pipeline. The actual pipeline is very similar to the local building and testing pipelines before PRs and only has an extra step that builds the binary for defined OS's in a matrix, zips the application and uploads said zip to the release page.

2.2 Team work

The activity diagram below shows the journey from an idea to said idea being developed, merged into the main branch and ultimately deployed.

Whether through new weekly requirements, bug reports or otherwise, tasks are written as issues with the goal of them being written as user stories. Clearly stating for whom the issue is of interest, the why, the what and how to determine if the issue is resolved. For the case of weekly requirements we divide up the work between us, assigning everyone to at least one issue. For the case of non-weekly requirements these are handed out based on how they interact with our other assigned issues and/or interest/whomever is quickest. Thereafter we set out to pair-/mob-program either in person or online to work on the issue. We create new branches for that specific issue, incrementally commit and push W.I.P. code until completion. Upon completion a pull request is made into the main branch and Github Workflows compiles and tests the code, making sure no obvious bugs or crashes are present, awaiting approval of at least 2 reviewers. When the request is approved or changes are made according to reviews, the branch is merged into main and the task and issue are considered completed.

As part of this workflow, end-to-end tests are used to check if the application works correctly as a whole. By running the tests through the GitHub Workflows, we ensure that newly merged features integrate correctly with existing functionality and that the application remains usable from a user's perspective.

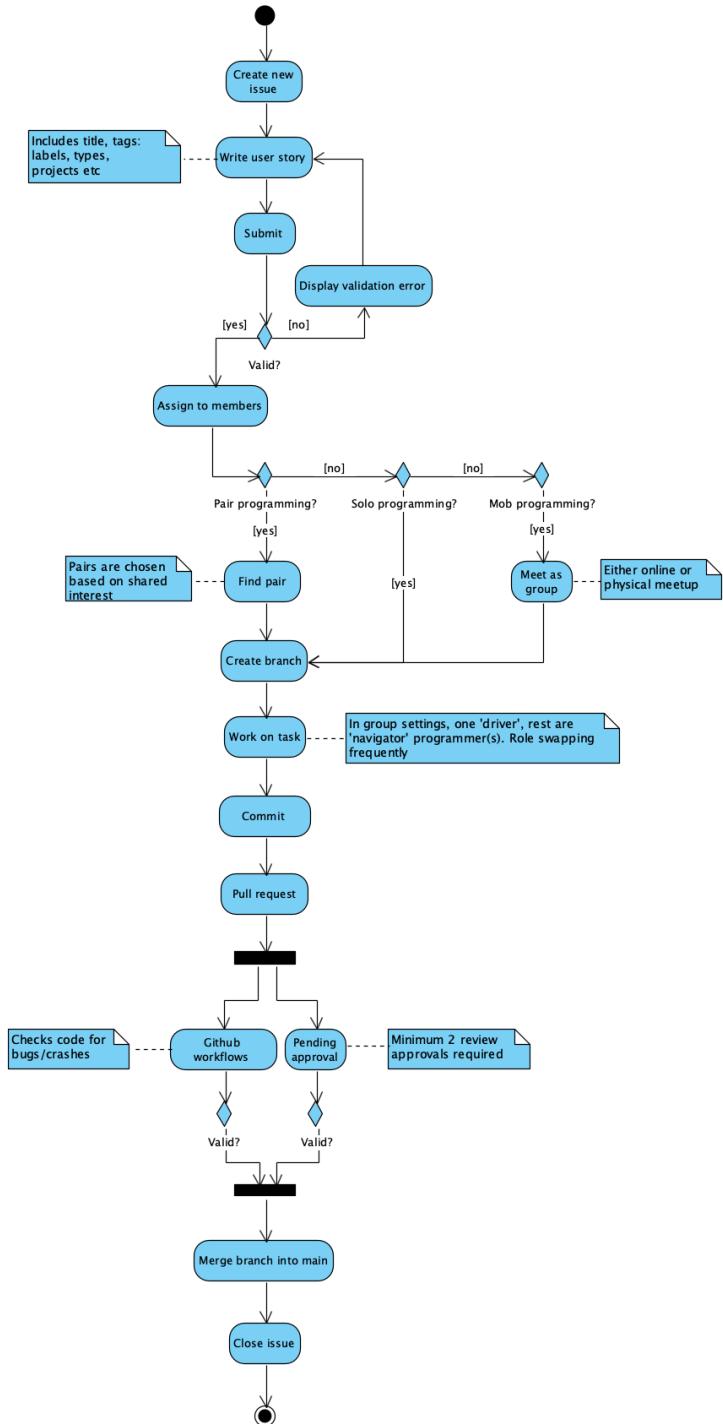
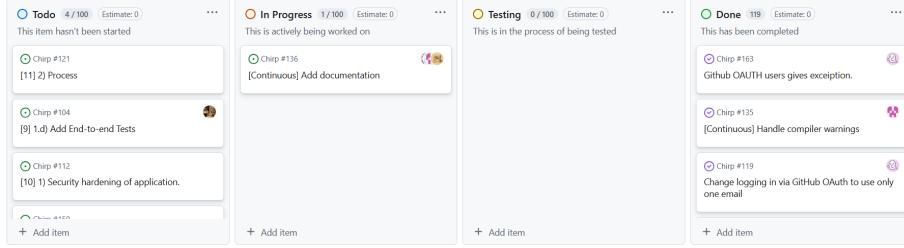


Figure 6: Team work Journey
12

2.2.1 Project board at handin



Open Issues

Although our application is feature complete according to the minimum requirements, several improvements could still be made and we are missing some testing and improvements to our security. On our Github these missing elements are visible in the issues that are still open. Below you'll find the reasoning behind why.

Issue #150 (Playwright with mock database): Currently our playwright tests don't use a mocked database which of course is problematic as it doesn't cleanly separate the testing environment from that of the production environment. We had trouble getting a mocked database to work with the playwright framework, because it depends on the actual program running to run the tests.

Issue #121 (Follow end-to-end test): We tried to solve this using UI end-to-test with playwright. The result became unreliable due to the dependencies on the authentication state. The timeline sometimes contained two cheeps, which made the results inconsistent. Solving this would have probably required restructuring the test setup and repository access.

Issue #112 (Security Hardening): Most of issue #112 has been solved by default on ASP.NET core and Identity. Things such as SQL Injection and XML injection vulnerabilities in our application has yet to be discovered by us. One thing that could've been implemented to harden our security would have been the HTTPS protocol to ensure our data was encrypted. Implementing tests against attacks could have been beneficial as well. This issue would theoretically not take much time but could potentially mess up some backend protocols, so we decided to not rush the implementation and let it be an unsolved issue as of this moment.

Issue #104 (end-to-end tests): We wanted to verify that a cheep entered by a user is saved in the database for the correct author. However, these tests depend on authentication and a clean database state. Since our Playwright setup depends on the running application and real infrastructure, the results could not be correct. To fix this would require us to use a separate test database, which we could not prioritize due to the project timeframe.

2.3 How to make *Chirp!* work locally

The guide below will get you started using our *Chirp!* application locally and how to setup Github OAuth locally.

2.3.1 Starting the webapp

Prerequisites:

- .NET 8.0
- (if on Linux) aspcore .net 8.0

1. Clone the github repository down locally

```
git clone https://github.com/ITU-BDSA2025-GROUP3/Chirp.git
```

2. Navigate to the following directory:

```
cd src/Chirp.Web
```

3. Run the following command:

```
dotnet run
```

That's it. The website will now run by default on port 5273 and can be visited on <http://localhost:5273>

2.3.2 Setup Github Oauth with webapp

When using our *Chirp!* web application locally, we have hidden OAuth by default when no tokens are provided. If you wish to test our way of handling github OAuth you will need to get your own github OAuth secrets.

The expected homepage URL should be:

```
http://localhost:5273/
```

The expected callback for your OAuth app should be:

```
http://localhost:5273/signin-github
```

Prerequisites:

- Github OAuth Secrets for the web app (with the callback defined above)

1. Navigate to the following directory:

```
cd src/Chirp.Web
```

2. Set the client ID of your OAuth app (replace it with your actual app client ID)

```
dotnet user-secrets set "authentication:github:clientId" "your-client-id-here"
```

- Set the secret to be your OAuth app (replace it with your actual app secret)

```
dotnet user-secrets set "authentication:github:clientSecret"  
"your-secret-here"
```

- Run the webapp again (if you haven't done that before, go to the documentation above!).

You should now see a button to register with github on: <http://localhost:5273/Identity/Account/Register>

or to login with github on: <http://localhost:5273/Identity/Account/Login>

2.4 How to run test suite locally

2.4.1 UI test suite

The UI (end-to-end) tests are implemented using Playwright for .NET and test core user interactions in the web interface. There are a few prerequisites that have to be met in order to be able to run these tests:

- .NET SDK installed
 - Playwright installed on the system
 - The Chirp web application must be runnable locally
-

2.4.1.1 Installing Playwright browsers In order to install Playwright browsers, paste this command in the terminal depending on your OS system:

MacOS/Linux

```
dotnet build test/Chirp.PlaywrightTests  
pwsh test/Chirp.PlaywrightTests/bin/Debug/net8.0/playwright.ps1  
install
```

Windows (PowerShell)

```
dotnet build test/Chirp.PlaywrightTests  
test\Chirp.PlaywrightTests\bin\Debug\net8.0\playwright.ps1 install
```

Note: This step only needs to be done once per machine. Skip it if Playwright browsers are already installed. For more details (or if these commands did not work), see the official Playwright for .NET documentation: <https://playwright.dev/dotnet/docs/intro>.

2.4.1.2 Running the UI tests After ensuring that your system has .NET SDK and Playwright installed, run the Chirp application in your terminal (how to do this is described earlier in the report).

Open a second terminal and navigate to the Playwright test folder from project root:

```
cd test/Chirp.PlaywrightTests
```

By default, the UI tests expect the *Chirp!* application to be running at:

```
http://localhost:5273
```

If your application runs on a different port, set the environment variable:

```
export CHIRP_BASE_URL="http://localhost:<your-port>" //MacOS/Linux  
setx CHIRP_BASE_URL "http://localhost:<your-port>" //Windows
```

This allows the tests to run without modifying the test source code.

Finally, write:

```
dotnet test
```

from the folder Chirp.PlaywrightTests that you navigated to earlier.

2.4.1.3 What these tests cover The UI (end-to-end) tests verify that:

- Unauthenticated users cannot see the cheep input field (therefore also cannot interact with it).
 - Authenticated users can see and use the cheep input field.
 - Cheeps longer than 160 characters cannot be created and therefore cannot be stored.
 - Users can log in and be redirected to the timeline after successful authentication.
-

2.4.2 Business logic test suite (Unit and integration tests)

The business logic test suite consists of unit tests and integration tests that verify the core functionality of the Chirp application.

To run the test suite containing unit and integration tests relating to the function of the application, navigate to the Tests folder from project root:

```
cd test/Chirp.Tests
```

Then run the following command to run the test suite:

```
dotnet test
```

Note: To run all tests together (unit, integration, and end-to-end UI tests), run the `dotnet test` command from the project root.

2.4.2.1 What these tests cover The backend test project consists of a mix of unit and integration tests:

- Repository tests (integration with in-memory SQLite)
 - RepositoryTests.cs
 - Uses an in-memory SQLite database which is created via `Utility.CreateFakeChirpDbContext()`.
 - Tests repository behaviour such as: creating cheeps/comments and verifying that they are saved in the database; reading cheeps with paging; deleting authors and verifying that related data is updated.
- Service tests (service logic that uses the repositories)
 - ServiceTests.cs
 - Tests service rules such as: validating cheep length; computing paging counts; comment validation and persistence through the service layer; author existence checks; author deletion behaviour through service layer.
 - Some of these tests use pure unit tests, while others use in-memory DB and real repositories (integration).
- Database fixture tests
 - SQLite.cs, TestDatabaseFixture.cs
 - Verifies seeding behaviour and that data can be stored in an in-memory database.
- HTTP endpoint tests (integration test of the web app pipeline)
 - TestAPI.cs, RazorPageWebAppFactory.cs
 - Run the web application in-memory for testing using `WebApplicationFactory`.
 - Verify that core endpoints respond and contain expected HTML content.

Note: No manual DB setup is needed for the tests to run. They create and seed an in-memory SQLite database so that they do not have to depend on the local Chat.db.

3 Ethics

3.1 License

We chose the MIT license as it is a common standard for open source software which this project inherently needs to be per course requirements of visibility.

It is furthermore compliant and compatible with the licenses of the third party software, libraries and frameworks which our project depends upon.

3.2 LLMs, ChatGPT, CoPilot, and others

We have used LLM's very sparingly for writing code and their contributions have been noted in the commits as co-authors where used. We have also used LLM's as sparring partners, acting like TAs, to better learn and further understand the tools we have been tasked to learn and use. Usage of the latter kind was decided, in discussion with the TA, to be of the same character as using other learning tools, like youtube, tutorials, stackoverflow etc. and therefore subject to the same criteria in relation to documentation and crediting as usage of such resources.

The specific LLM's used: ChatGPT, Grok, Gemini, Google AI Overview, Duckduckgo AI, Claude.

3.2.1 Discussion and reflection of LLMs

The use of LLM's were often met with frustrations as several prompts were required before the model understood the context of the issue. The use of LLM's are often easily accesible but ultimathly hinder production. They are most useful when exploring potential fixes to persistent bugs and suggesting alternative approaches when stuck. Overall, LLM's were good as a starting point but not as a final solution.