Chirp! Project Report

ITU BDSA 2024 Group `19`

Lluc Paret Aguer llpa@itu.dk Gustav Hejgaard ghej@itu.dk
Ronas Jacob Coban Olsen rono@itu.dk Jacob Sponholtz spon@itu.dk
Jakob Sønder jakso@itu.dk

Contents

1 Design and Architecture of *Chirp!*

1.1 Domain model

The two main entities of *Chirp!* are `Author` and `Cheep`.

- An `Author` represents one of our users. This type is implemented as an extension of the default `IdentityUser` type.[1] Authors can follow each other on *Chirp!*
- A `Cheep` is a message posted to *Chirp!* by an author. Additionally for this project, users are able to "like" Cheeps.

These entity types are saved to a database using *Enitiy Framework Core*.[2] EF Core supports saving these entities in a relational database for the application. As the entities change and features are added, EF Core is able to apply *migrations* to any existing database.

In addition to the main entities, we have derived types for data transfer and page model types.

Data transfer objects such as `CheepDto` are used where data leaves `Chirp.Infrastructure` for other services. In this way we have greater control over data leaving our infrastructure layer. Example: Each `Cheep` has an `Author` object reference to the writer. This entity has fields with email data, password hashes, etc. A `CheepDto` passed to another service only contains the author name and profile picture link.

Content preparation in `Chirp.Web` formats `CheepDto` data for web display as `CheepViewModel`. This allows `CheepDto` to be "reused" as a type for serving in other formats such as JSON.

[1] *IdentityUser Class Reference* https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.identity.identityuser

[2] Andrew Lock, "ASP.NET Core in Action", Chapter 12: *Saving data with Entity Framework Core*, 3rd edition, 2023

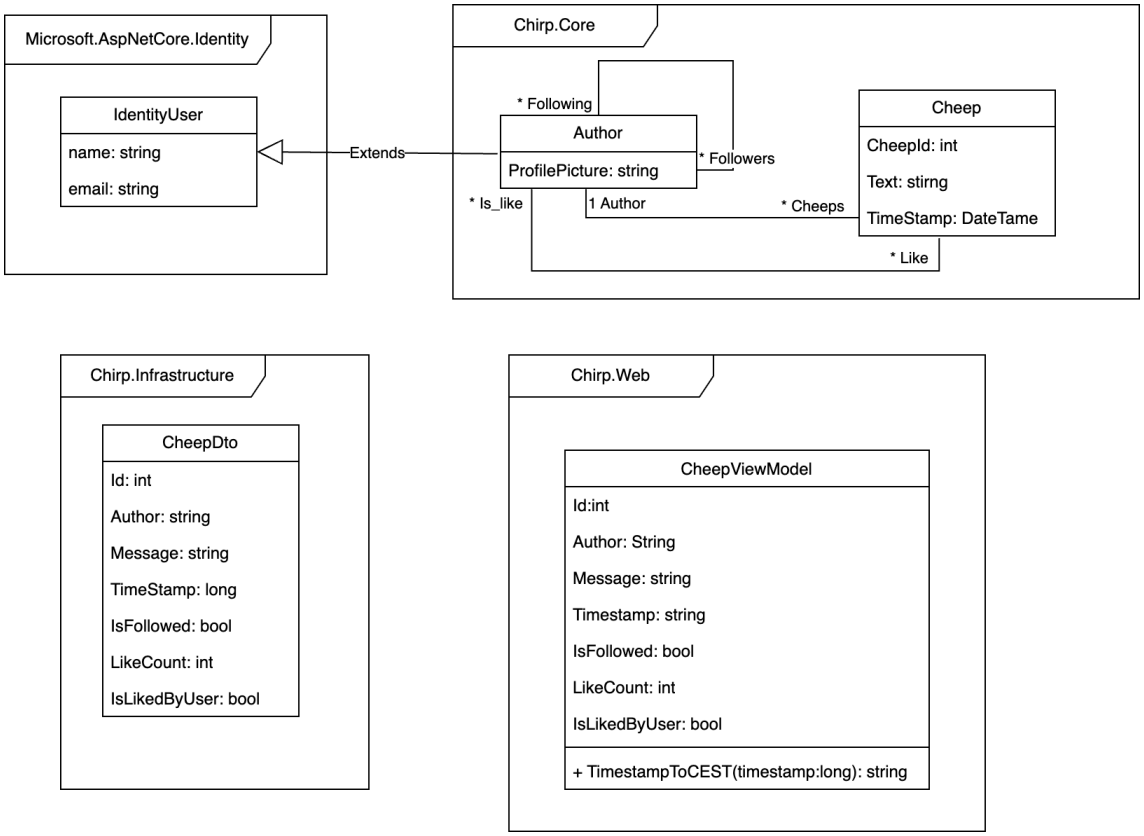

Figure 1: Illustration of the *Chirp!* data model as a UML class diagram.

1.2 Architecture — In the small

Below is an illustration of the organization of our code base. We have implemented the 'Onion Architecture'. The four colours in the diagram refer to the four layers of the architectural pattern[3].

Our code base is divided into three folders:

- `Chirp.Web` handles the outer UI layer. In this folder all code referring to displaying our application resides. This includes all Razor Pages and their respective models. As this is the outermost layer, `Chirp.Web` has knowledge of all architectural layers.
- `Chirp.Architecture` contains both the Service and Repository layer. This has been done to make the `Author` and `Cheep` service and repository pairs easily accessible. The services depend on the repositories, and the repositories depend on the entity classes in `Chirp.Core`.
- `Chirp.Core` only contains the two Domain Entities `Author` and `Cheep`. Hence `Chirp.Core` has no knowledge of any other layers.

The diagram shows that our code base only has inward dependencies, in compliance with the 'Onion Architecture'. Thereby no inner layer has any knowledge of outer layers.

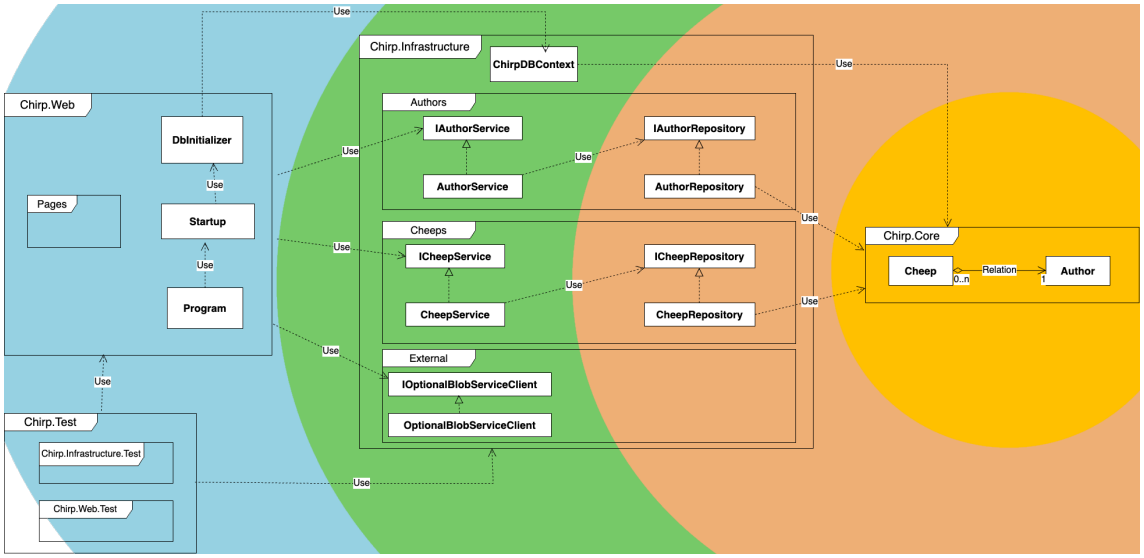

Figure 2: Illustration of the *Chirp!* code architecture as a UML class diagram.

[3] *Colouring of the 'Onion Architecture' inspiration* https://raw.githubusercontent.com/itu-bdsa/lecture_notes/refs/heads/main/sessions/session_07/images/onion_architecture.webp

1.3 Architecture of deployed application

The class diagram below shows how the architecture of *Chirp!* looks when deployed. It is a client-server application where clients connect to a webservice through HTTP. All of the HTTP GET/POST requests are then handled by Azure App Service using an SQLite database. The webserver depends on two services that handles different core components in our program. The first one is the Blob Service which is based in the Azure environment. Blobs are defined as unstructured data or files, which in our case is the users profile pictures. The second one is an external API provided by Github. This API makes third party authentication possible through webservice calls, that provide authentication tokens for our login process.

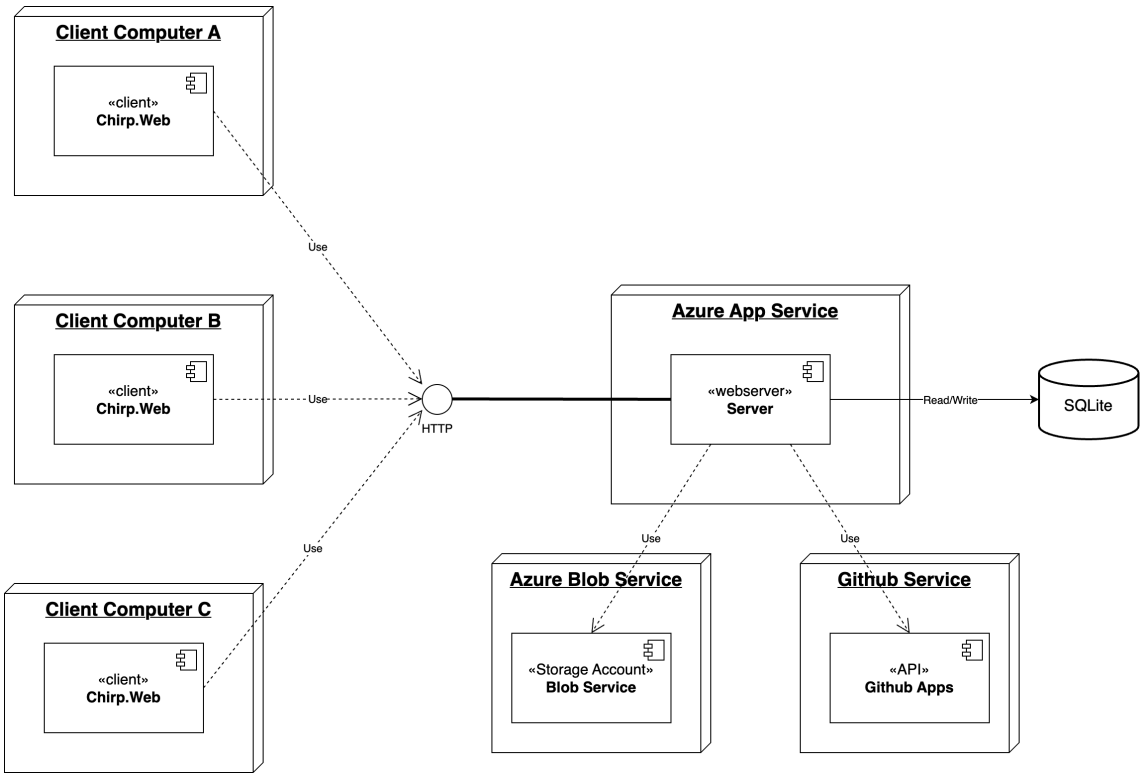

Figure 3: Illustration of the *Chirp!* deployment as a UML package diagram.

1.4 User activities

The illustrations below show two user journeys. The first one shows the journey of a non-authorized user. The second one shows what features become available when the user has been authorized.

We can see that the actions that change the website are only available to authorized users. This includes the ability to post a Cheep, follow other users, and like Cheeps.

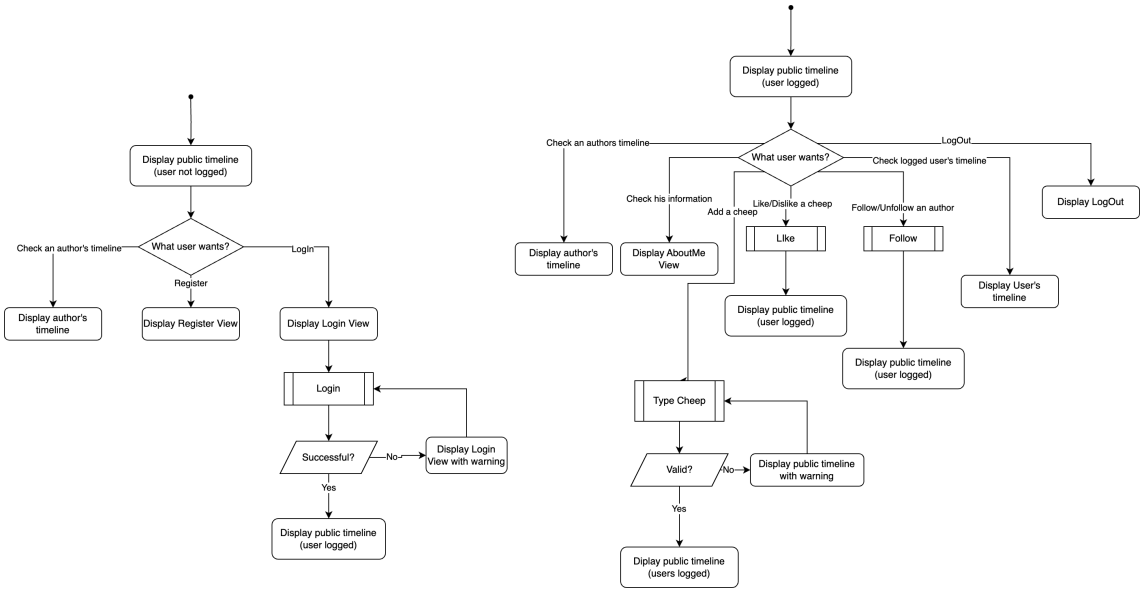

Figure 4: Illustration of *Chirp!* user activities as a UML activity diagram.

1.5 Sequence of functionality/calls trough *Chirp!*

The sequence diagram below illustrates the sequence of messages and data needed to render the entire public timeline for a non-authorized user. The sequence starts with the user loading the application thereby sending a HTTP GET request. The diagram ends with a fully rendered public timeline returned to the user.

The LINQ query returns a `Task<List<CheepDTO>>` that is converted to a `<List<CheepDTO>>` and then finally the result is transformed to a `List<CheepViewModel>` with the loop.

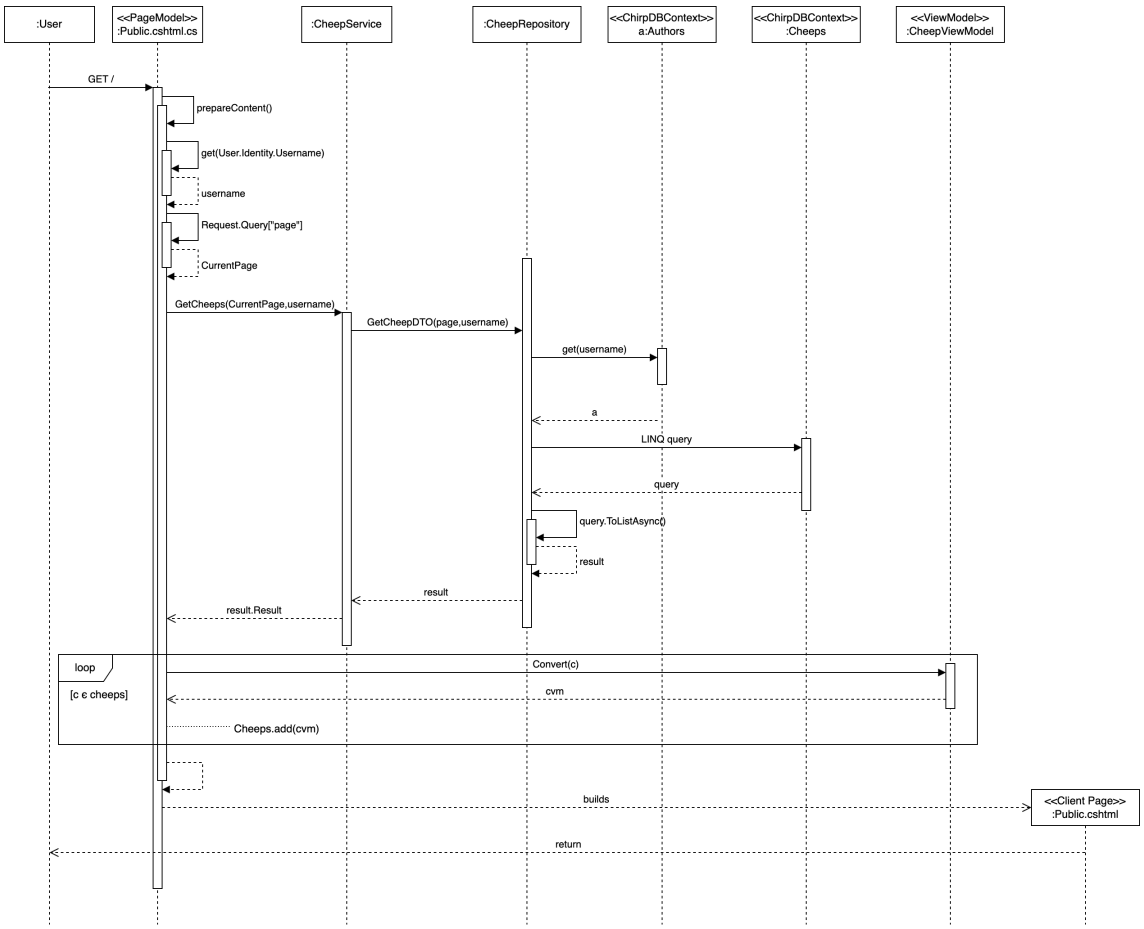

Figure 5: Illustration of a sequence of calls through *Chirp!* as a UML sequence diagram.

2 Process

2.1 Build, test, release, and deployment

Below is an illustration of how our two workflows interact with different GitHub action triggers.

We have three workflows:

- *build_and_test* makes sure the program builds and tests locally. This workflow is activated whenever a pull request is first made, as a check prior to review.
- *github-auto-release-WebApp* creates new releases of *Chirp!*. This workflow is activated whenever we tag a specific commit on the main branch with a version tag.
- *bdsa2024group19chirprazordeploy* handles deployment to Azure. When a pull request to the main branch closes with a merge, this workflow deploys the new version to our website.

All of our workflows run on a local Ubuntu instance, that is created at the start of the workflow. All the workflows check out the code base, set up *.NET* and use it to restore, build and test the project. The deployment and release workflow then deploys the newly built application to Azure. Finally the release workflow creates a new GitHub release with the necessary files.

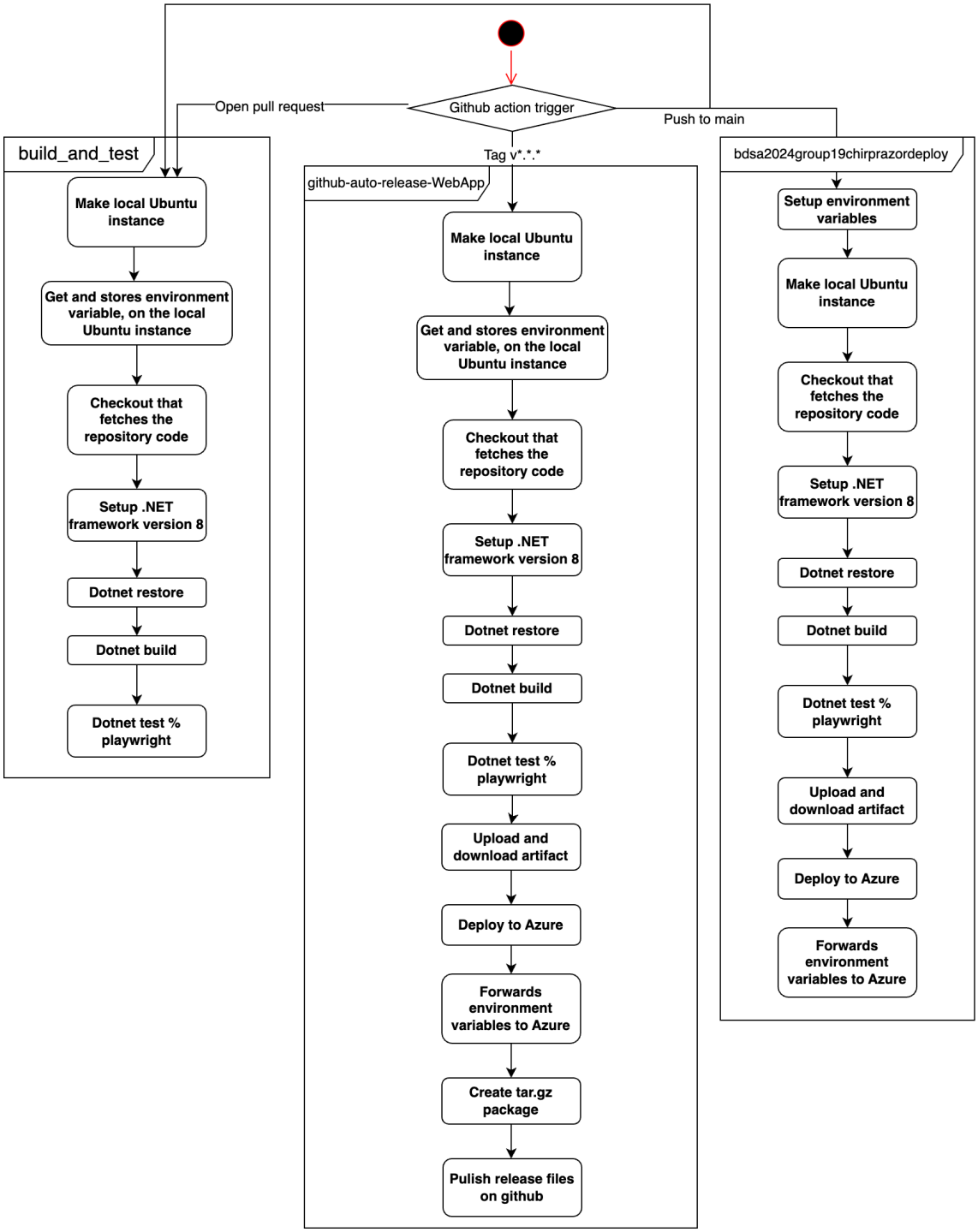

Figure 6: Illustration of _Chirp! build, test and deployment workflows as a UML activity diagram.

2.2 Team work

2.2.1 What tasks remain unresolved

All features for the *Chirp!* application have been completed. The last unresolved tasks are refinements that would have been nice to have for a smoother user experience.

These include:

- The ability for users logged in through GitHub, to delete their account without creating a password.
- Small refactorings, enabling easier readability of our code base.

As for our test suite, testing for the `AuthorRepository` and the forget me feature have not been completed. Our playwright tests only works locally, so we filtered them out of our GitHub workflows. In addition a larger playwright test suite would have been ideal as more edge case UI errors would have been detected sooner.

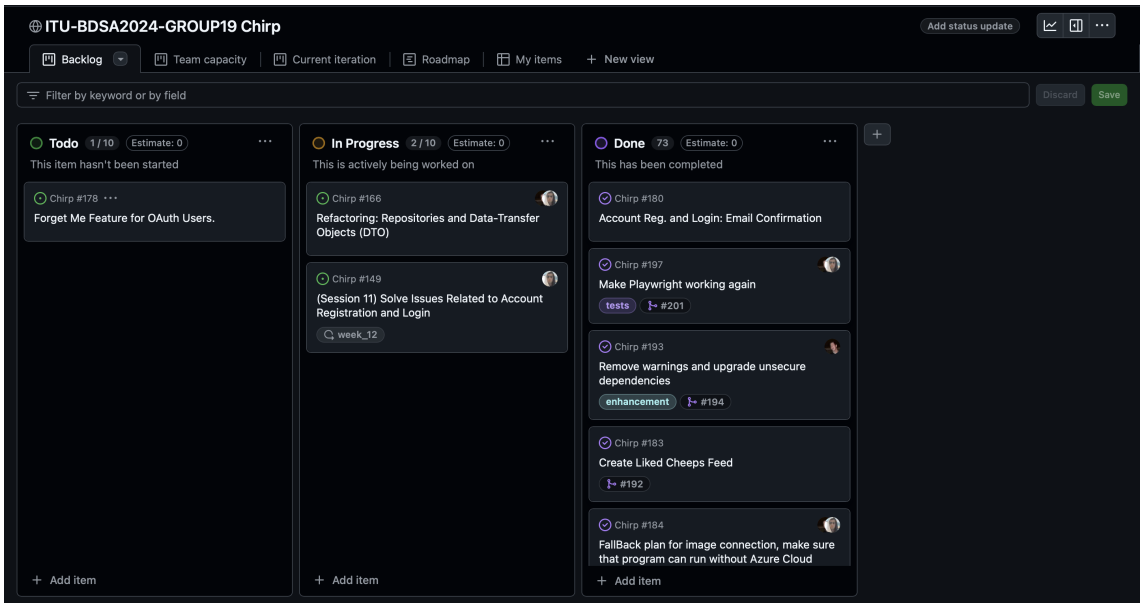

Figure 7: Screenshot of our GitHub project board.

2.2.2 Our development workflow

When a new task arises, an issue is created on GitHub using it's build-in ticket system.

The issue is then created based on a set of rules shown in our `README.md` file, including:

- Issue Title Format: "('Session week number', 'issue number') 'Title of the issue at hand' "
- User Story Format: "As a (user type), I want to (task) so that (goal)." (Please write in issue description)
- Acceptance Criteria: Follow a point format of the intended outcomes of the issue.

After the issue has been created, a development branch is added. If we identify the task as a feature, the branch name has the prefix *feature/*, thereby creating a feature directory. Otherwise the prefix

should be *issue/*. Afterwards, development begins and runs iteratively until all acceptance criteria are satisfied. If the developer thinks the task is done, they create a pull request. If another group member believes changes are necessary, a new development cycle begins. The contents are only pushed to main when the pull request gets a minimum of one approval.

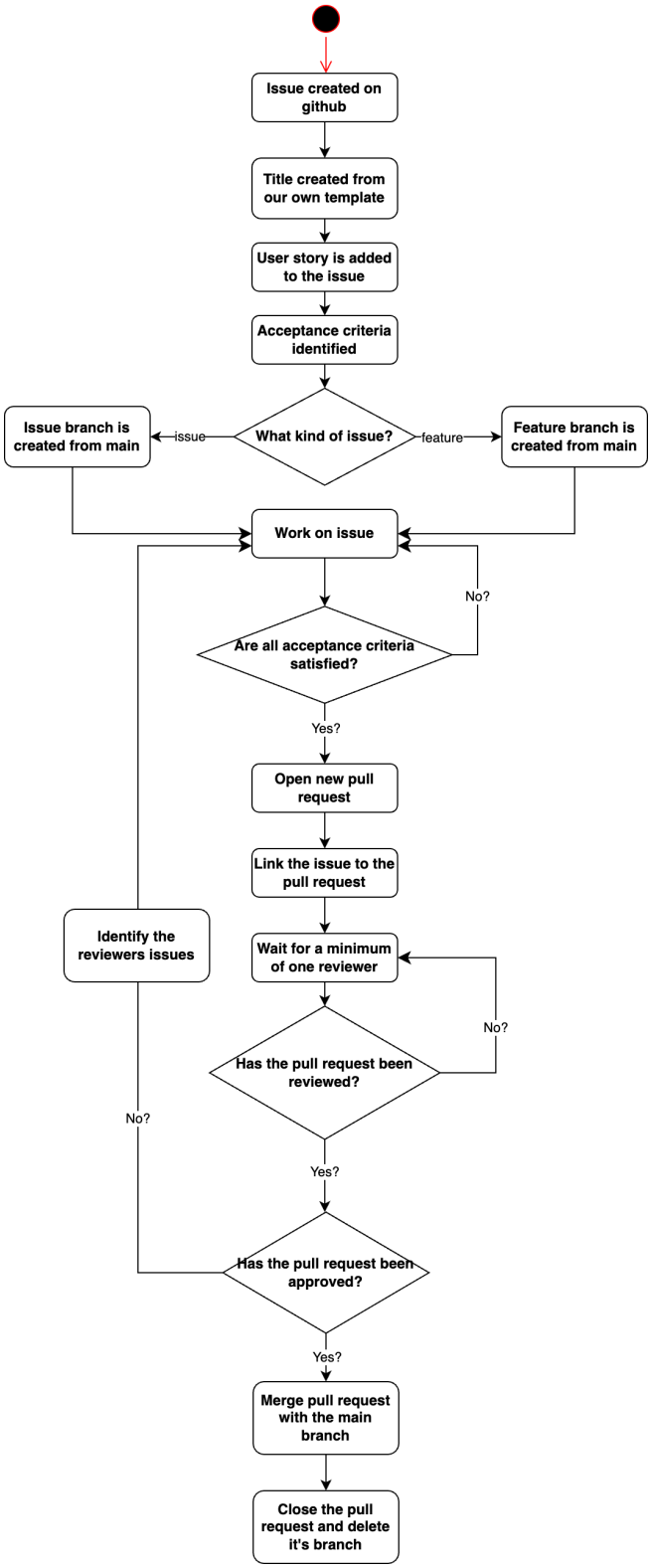

Figure 8: Illustration of the workflow when working on *Chirp!* as a UML activity diagram.

2.3 How to make *Chirp!* work locally

2.3.1 How to run the project from source code

- The project can simply be run locally using the `dotnet run` command from the Chirp.Web directory in the terminal.
- Environment variables for the database connection string and the Azure storage connection string depend on if you run the source code directly or not. The app is able to detect that it is running in a developer environment. When the app is deployed to a live server or otherwise published, the environment variables must be set.
- If in the development state, the app normally uses an in-memory SQLite database with sample data. It is also possible to use a persistent database in testing by setting `CHIRPDBPATH` to the path of a database file. A default picture replaces custom profile pictures, if a connection to Azure Cloud Storage is unavailable. Environment variables are not needed to be set, but can still be set to make Azure Storage and GitHub OAuth work in the development state.

Examples of these environment variables are:

```
export CHIRPDBPATH=":memory:"
export CHIRPDBPATH="/Temp/db.db"
export AZURE__STORAGE__CONNECTION__STRING="DefaultEndpointsProtocol=https;
AccountName=chirpstorage;AccountKey=yourkey;EndpointSuffix=core.windows.net"
```

- Warnings of deprecated or old dependencies can be fixed by deleting old packages from the running machines NuGet cache. This can be done by running `dotnet nuget locals all --clear` in the terminal. Or manually deleting the packages from `C:\Users\<INSERT_YOUR_OWN_USERNAME>\.nuget\packages` and delete the packages that are no longer needed.
- Example of this issue could be the `.nuget\packages\system.text.json` that is being relied on in the project by the package Azure.Storage.Blobs. This states that System.Text.Json version 6.0.10 or greater is needed, but this allows for packages that are older and have warnings and secuirity issues.

2.3.2 How to run the published program

- Download the .tar.gz file from the release page on GitHub. This is a release for Linux x64.
- Extract the published program to a directory.
- The project can be run by executing `Chirp.Web` in the bash terminal set to the release directory.
- It is mandatory to set the environment variable `CHIRPDBPATH` to the SQLite database location. Optionally set the Azure Blob Storage and GitHub OAuth variables.
- Setting `CHIRPDBPATH` to `":memory:"` will run *Chirp!* with an in-memory database and sample user data.
- For persistent user data, set `CHIRPDBPATH` to the path for a database file. E.g. `/chirpdata/chirp.db`. The database will automatically be created if it does not exist. The directory must exist.

Examples of these environment variables are:

```
export CHIRPDBPATH=":memory:"
export CHIRPDBPATH="/Temp/db.db"
```

```
export AZURE__STORAGE__CONNECTION__STRING="DefaultEndpointsProtocol=https;
AccountName=chirpstorage;AccountKey=yourkey;EndpointSuffix=core.windows.net"
```

2.4 Test suite

2.4.1 How to run test suite locally

With a terminal open in the project root, the test suite can be run using the command "`dotnet test`". Integration tests and end-to-end tests will run the app locally using an in-memory database.

- Regular unit tests and integration tests are known to work on all platforms. These tests are integrated into our workflow on GitHub.
- Playwright tests can run locally. A test fixture has been incorporated to help setup Playwright on new machines.[4] The test fixture has been used successfully on Windows machines. There are still problems running these tests on Linux and OSX platforms.

2.4.2 Our tests

In our test suite, we have created Unit tests for the `CheepRepository` and `CheepService`. These tests are run on a repository stub imitating the real `CheepRepository`. These tests cover most of the methods in both `CheepService` and `CheepRepository`.

Furthermore we have created different End-to-End UI tests using the Playwright framework. These tests focus on checking that the *Chirp!* website runs as intended.

This includes:

- Testing against XSS attacks.
- Testing that the authentication works as intended.
- Testing that the website displays cheeps as intended.

Lastly, we have an integration test, making sure that SQL injection attacks cannot happen. Although not all aspects of the application are tested, we have covered what we deemed most important. Over the duration of the project, we made sure to learn and use all the testing frameworks that were introduced.

3 Ethics

3.1 License

We have chosen to release our project under the MIT License. Anyone can use the project for any purpose, provided that the license and our mark is retained with the work. We accept no liability and provide no warranty for such use.

To adopt this license, we first looked at any dependencies referenced by our project. These were the Microsoft `.nuget` packages released under the MIT license and other third party tools such as `Moq` released under the BSD-3 license. The dependencies we identified do not place restrictions on our project source release through their licensing.

An important reference for an overview of software licensing options in the open source community has been `https://choosealicense.com/`. We found insight into the importance of licensing for

[4] Xavier Solau, https://github.com/xaviersolau/DevArticles/tree/e2e_test_blazor_with_playwright, 2022

promoting competition and continued development from a guest lecture on the subject by Martin von Haller Grønbæk at ITU.[5]

3.2 LLMs, ChatGPT, Copilot, and others

3.2.1 ChatGPT

We have tried not to rely on services such as ChatGPT. We found that it often lacks the perspective required to write code that adheres to most common programming principles.

However, there are cases where ChatGPT has been used. Here are some key examples:

- Throughout the project, ChatGPT has in some cases been used for added inspiration. Provided "solutions" to issues were not satisfactory for direct application, but did provide insight into how we could solve these issues ourselves. However in the time it took to get to a helpful answer, we could have just as easily found help somewhere else.
- In the first stages of the project, it was a useful tool for installing software correctly.
- It proved quite useful when setting up the connection between Azure and GitHub OAuth. The instructions provided in the course material did not reflect the current state of the Azure website.
- When receiving error messages of considerable length, it was good at translating these to a more readable form.

3.2.2 Copilots

Several IDE's have been in use during the project, namely Visual Studio 22, Visual Studio Code, and Jetbrains Rider. By default, Rider has a built-in feature called IntelliSense, that adds suggestions to complete statements.

IntelliSense has been used often, although many of its autocomplete suggestions have been ignored. Much like ChatGPT, it does not always understand the context of the environment and has a tendency to provide inapplicable suggestions. To the extent that this tool has found use, it has still largely required a human in the loop with project insight.

The use of IntelliSense has also been extended with GitHub Copilot, an extension for Rider. This service provides optimized autocomplete suggestions and improves on IntelliSense.

No other LLM's have been used during the project.

[5]Martin von Haller Grønbæk, "Guest lecture on Software Licenses" (lecture, ITU, Copenhagen, October 9, 2024).