Chirp! Project Report

ITU BDSA 2023 Group 4

Anna Høybye Johansen annaj@itu.dk
Lukas Andersson lukan@itu.dk
Marius Thomsen mariu@itu.dk
Niels Christian Skov Faber nfab@itu.dk
Oliver Asger-Sharp Johansen oash@itu.dk

1 Design and Architecture of *Chirp!*

1.1 Domain Model

Provide an illustration of your domain model. Make sure that it is correct and complete. In case you are using ASP.NET Identity, make sure to illustrate that accordingly.

1.2 Architecture — In the small

In the Onion Architecture diagram seen at fig. 1 you'll see our applications. In the centre we have our core package. This is the lowest layer of the application, contains no dependencies and is not likely to change. As we move outwards through the layers. The layers get more specific and dependent on the earlier layers and are more likely to change. At the outermost layer, we end with our SQL-Server and razor pages, which interact with our Azure application as external elements.

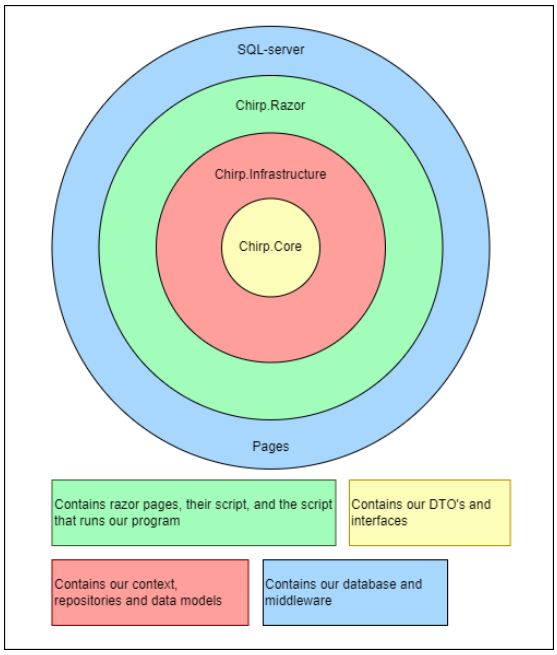

Figure 1: Onion Architecture Diagram

In order not to overwhelm the diagram. The details of the classes are kept minimal in the Onion class diagram seen on fig. 6. There is a UML class diagram for each package. All of these are shown in the Onion class diagram. This is done to keep the diagrams clear and readable. The interaction between layers and packages is shown in the Onion class diagram. The internal interaction is shown in the UML class Diagrams figs. 2, 3, 4, 5.

You will see in our repositories, that we're deleting the author at some point, this was a project demand. We had two possibilities; delete the user in the sense that they will no longer be traceable, that is make everything anonymous and delete their information, or we had the possibility of deleting everything that the user ever created. We chose to be sure that the user wouldn't come back complaining that their username/normal name still was in a cheep, so we deleted everything that they created. We chose to give the user full control and ownership over their content, so we deleted everything that they had created. This was also the easier approach since we could delete everything that contained that user's id or name, instead of altering everything. The implementation chosen also allowed us to let some of the data in the database, be deleted through cascading, instead of having to write logic for it.

The method IncreaseLikeAttribute in the CheepRepository, which can be seen in the diagram fig. 3, reveals that like is an attribute on the Cheep entity since its only parameters are a Cheep id and not an author id. This is the simplest

implementation of the feature, we could come up with. We have chosen to use this implementation due to the overall time constraint of the project. It has the impact, that it is not possible to see or retrieve data from the database, about who has liked a cheep. It is possible for each author to look multiple times. It is not possible to regret a like in the current state of the application, although a dislike method could be implemented.

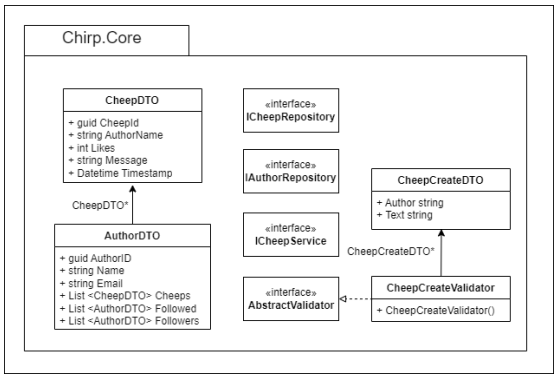

Figure 2: UML Class Core

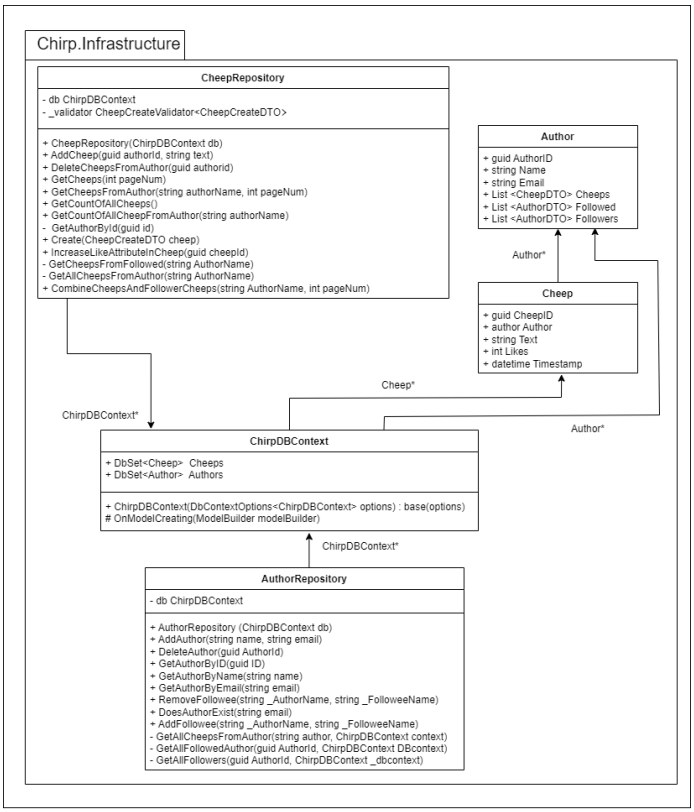

Figure 3: UML Class Infrastructure

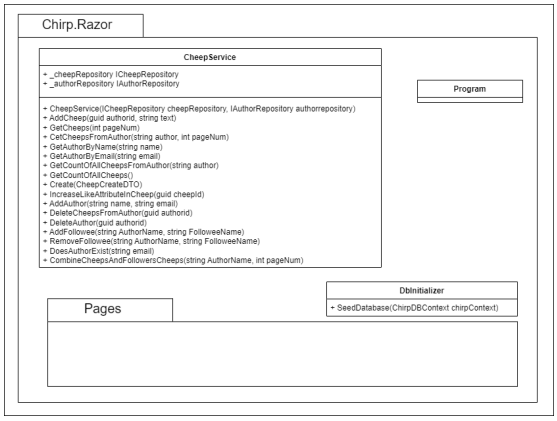

Figure 4: UML Class Razor

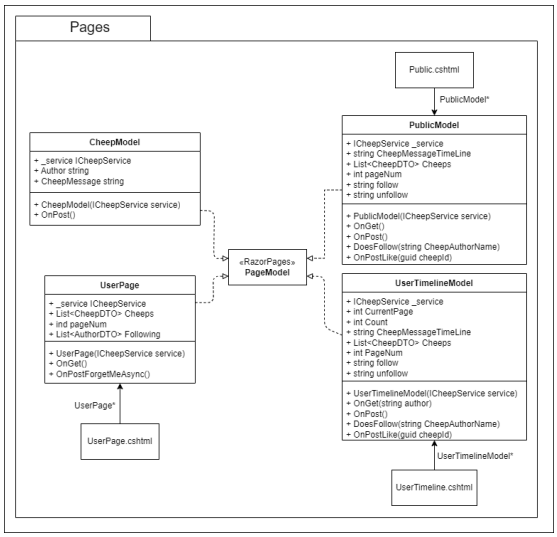

Figure 5: UML Class Pages

The Onion Architecture (otherwise known as Clean Architecture), is great for having low coupling and high cohesion. When looking at the UML in the more specified onion diagram bellow, there is no unnecessary communication between scripts. Having low coupling increase the readability and maintainability of the program. Since there are less dependencies to take into account, even though some of the repositories contain a fair amount of methods. When moving outward you'll see the packages only use entities further in or in the same layer.

It is worth mentioning that the only way of interacting with the repositories is through their interfaces, which is an important factor in making sure the application has low coupling. The same goes for the CheepService, since every class that needs to access it uses information from the interface, and that interface uses from the other interfaces.

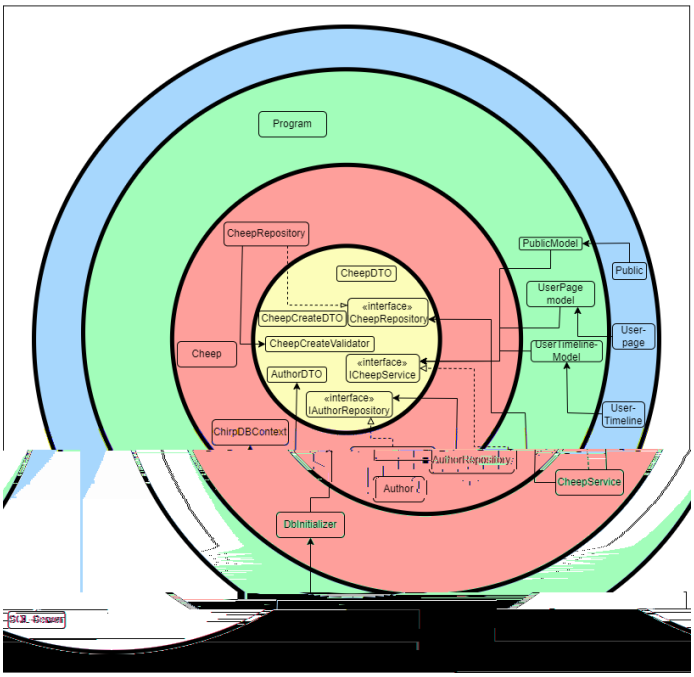

Figure 6: OnionClassDiagram

1.3 Architecture of deployed application

In fig. 7 a deployment diagram can be seen of our Chirp application.

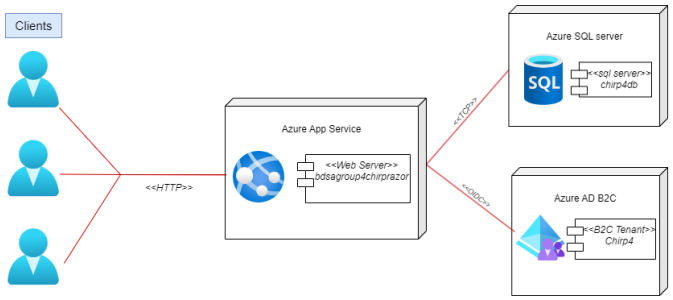

Figure 7: Deployment diagram

Chirp is a client-server application hosted on the Azure app service as a Web App. The web app is connected to an Azure SQL server where the database can be found. Furthermore the application makes use of an Azure AD B2C tenant for user-authentication. Each node and its means of communication are

represented in the diagram.

1.4 User activities

For each user activity bellow, there's a headline describing their scenario. There's one activity diagram which shows the application for a non-authorized user. Since most of our features requires the user to be logged in we had no need to show more. This diagram shows what the user can see as their only available page, the public page, with no additional features like a cheepbox or the possibility of following/unfollowing. They can show a specific page of the author for a cheep if they press their name.

If a user is logged in, there's a few possible user activities as shown bellow. These diagrams show what the user can experience when following or unfollowing, how the user can see who they're following and vice versa, how they write and share a new cheep, how to delete their information and what happens upon login and logout. If a new user wants register they will need to login with github. Since this will redirect from our own razor pages, this hasn't been included in the activity diagrams.

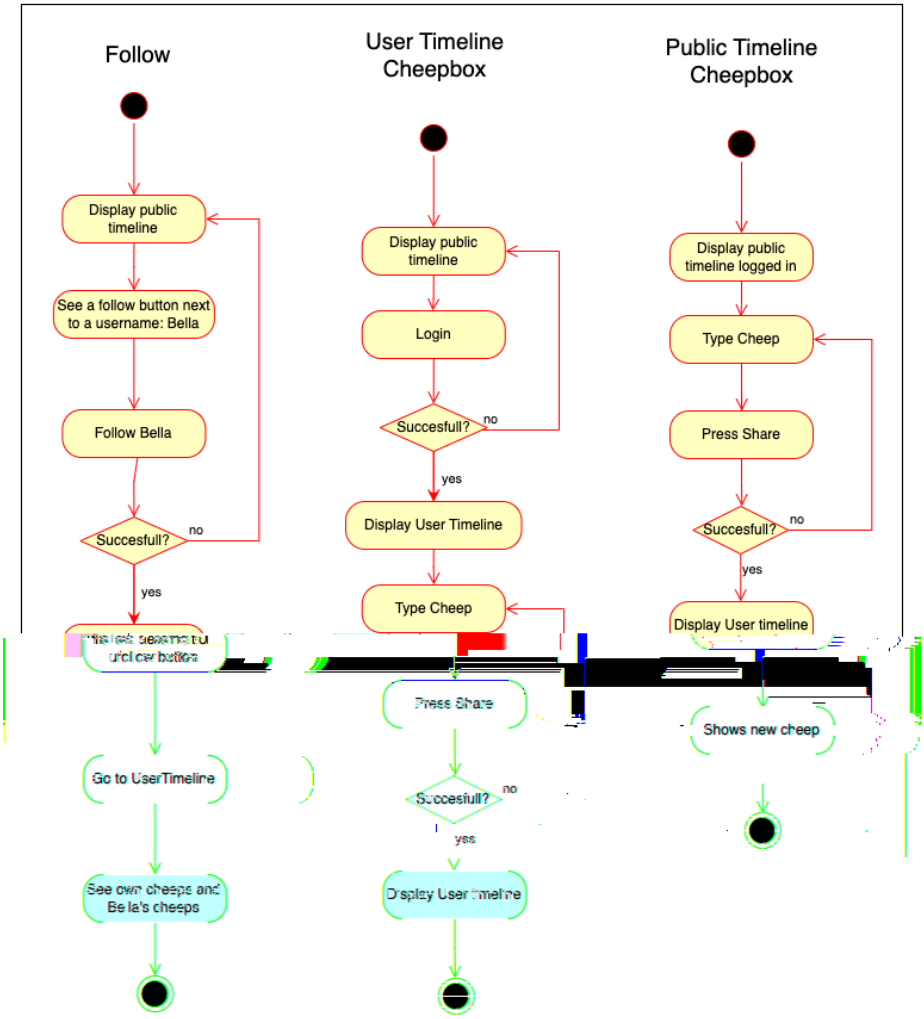

Figure 8: ActivityDiagramArchitecture

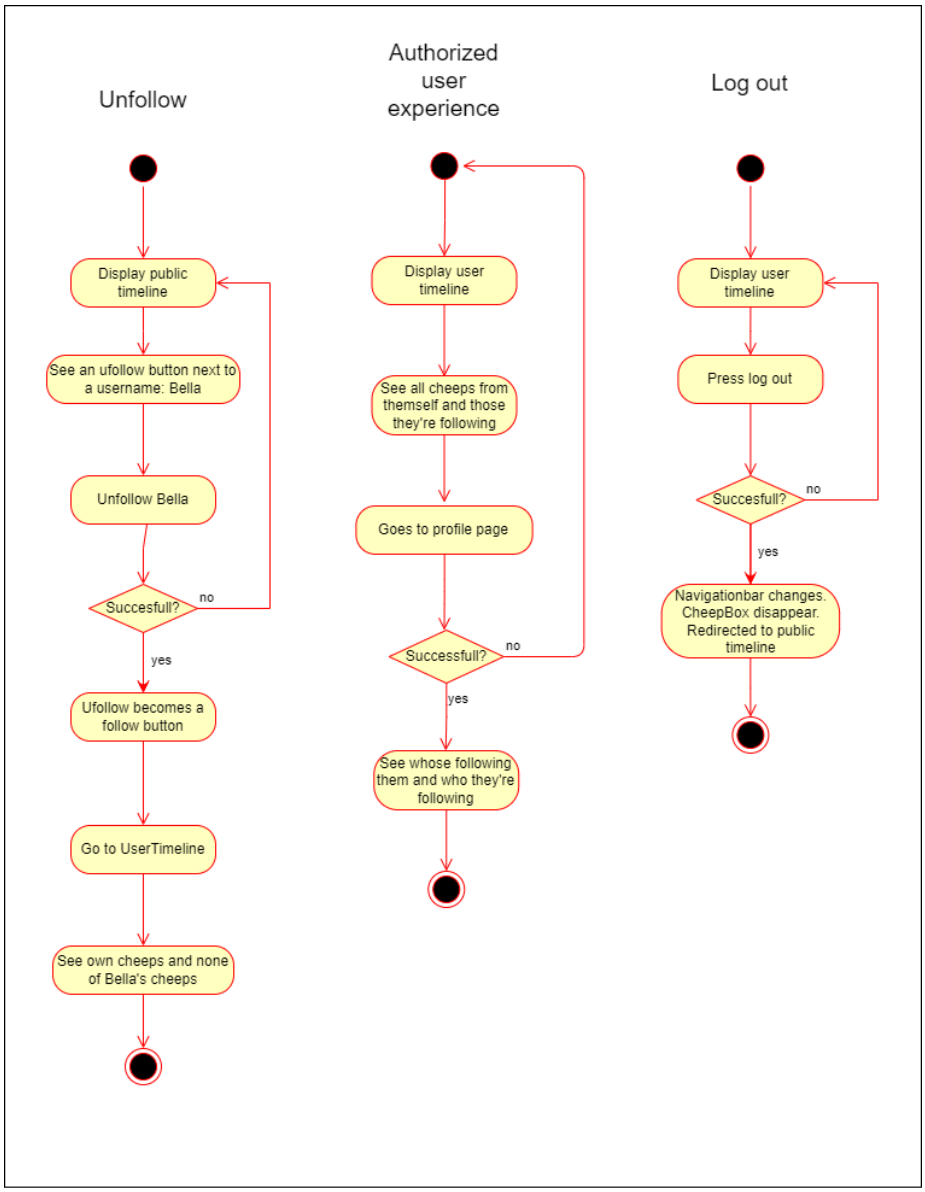

Figure 9: ActivityDiagramArchitecture

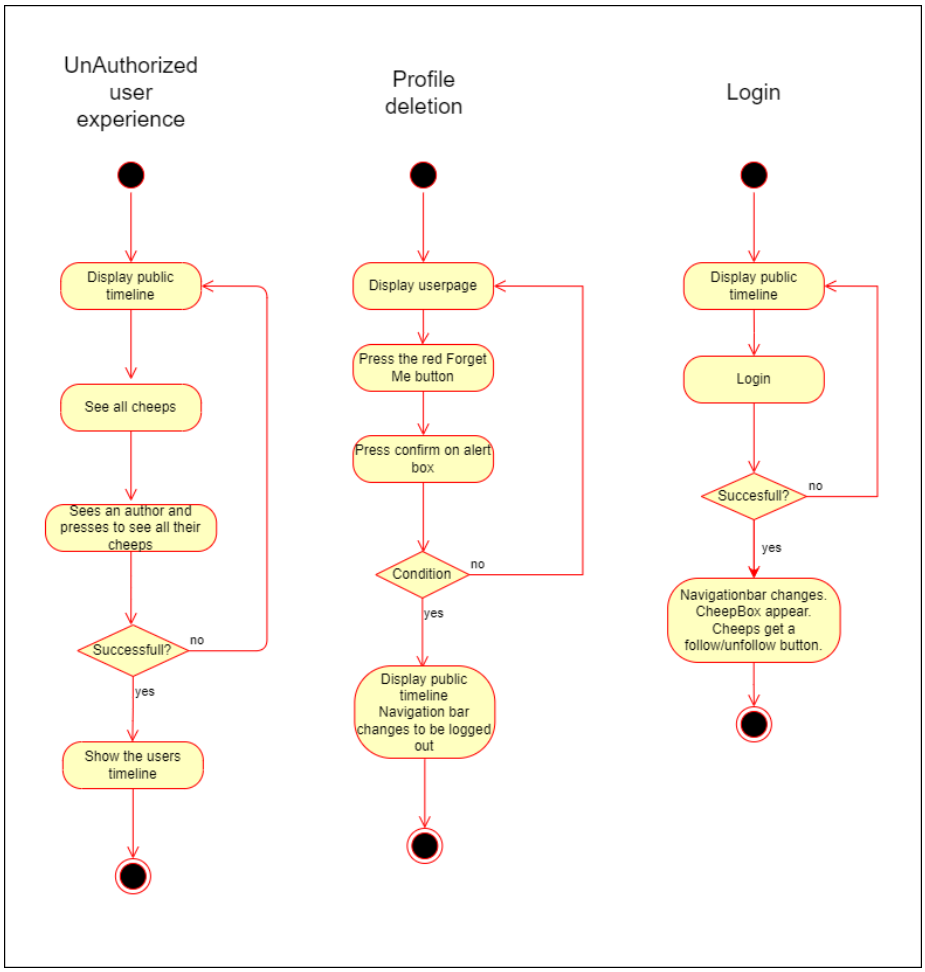

Figure 10: ActivityDiagramArchitecture

The userpage will show more detailed information than who's following you, the user can find their claims (such as their email and username), they'll be able to read about how we're using their information, it's also the location of the Forget Me feature (which deletes their profile and information). Most of these diagrams are not very detailed, to see more detail of the application you can run it with the help of our guide (How to make *Chirp!* work locally). There you will see the interface as well as our applications behavior.

1.5 Sequence diagram

In fig. 11. A sequence diagram of an unauthorized actor. Hereafter, referred to as UA, accessing our project. It shows the UA sending the HTTP Get request

to receive the website. After the initial request, the Chirp.Razor starts to build the HTML. Here, an asynchronous object creation message is sent through the interface in the core and onto the repository.The repository returns the same static content for all actors sending this request. Using Linq, the repository queries the SQL database for the 32 most recent cheeps.

The database sends the 32 cheeps to the repository. Which inserts each cheep into a CheepDTO before returning a list of 32 CheepDTOs. This list is sent back through the system, shown in fig. 11. Arriving in Chirp.Razor. It is weaved into the HTML, checking the if the user is Authorized. Before the page is returned to the UA.

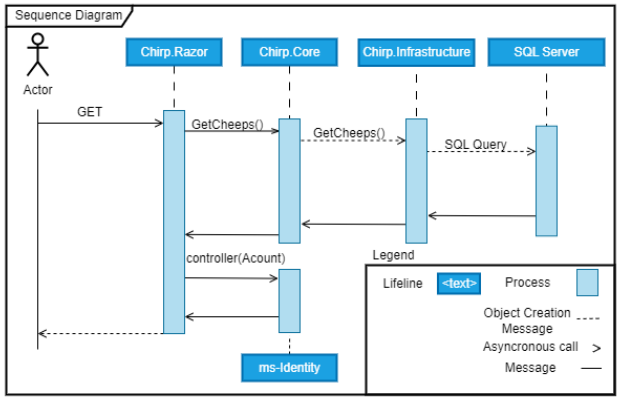

Figure 11: Sequence Diagram Unauthorized

fig. 12 show a known actor accessing our site, logging in and sending a Cheep. The first Get request is the same as seen in fig. 11. It deviates during the authentication step as the actor presses the login link. As the actor logs log in, Microsoft Identity redirects them to Azure OIDC. Which then redirect to GitHub.

After the actor has logged in, GitHub sends a token back to be checked by Azure. The token is in the URL. With it confirmed, the Razor page HTML Will change.

Then the authorized user fills out the desired cheep and Chirps it. When that happens, Chirp.Razor constructs a CheepDTO and sends it through the core, where it is validated and sent to the repository. Afterwards it is committed to the database granted tha validation confirms.

Then, confirmation of success is sent back, at which point the razorpage redirects to itself to reload the content.

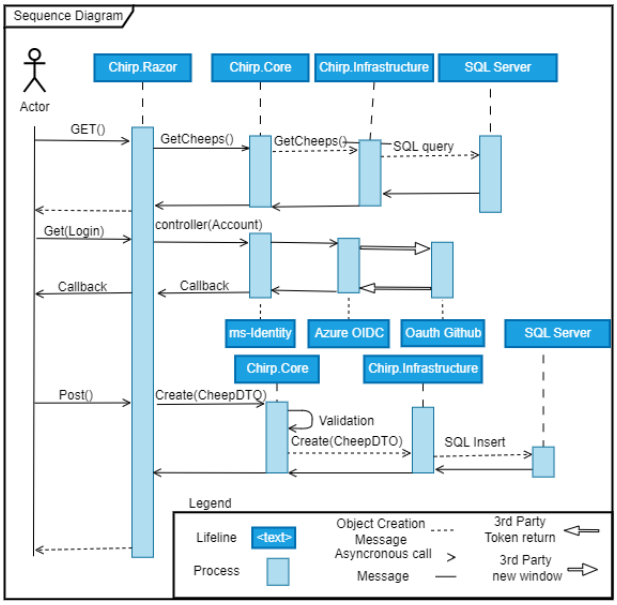

Figure 12: Sequence Diagram Authorized

2 Process

2.1 Build, test, release, and deployment

2.1.1 GitHub workflows

To ensure the flow of the project, we use a tool developed by GitHub known as. GitHub Action, otherwise known as a workflow. This will also include when the workflows are activated and used.

2.1.1.1 Build and Test The build and test workflow can be found on fig. 13. The activity diagram shows how GitHub ensures what is merged into main does not damage it.

This workflow is run on a pull request every time a commit is made to the branch belonging to the pull request. This ensures that main will stay functional by building and testing the project with dotnet and our test suite. If anything fails, it will stop and prevent the branch from merging into main.

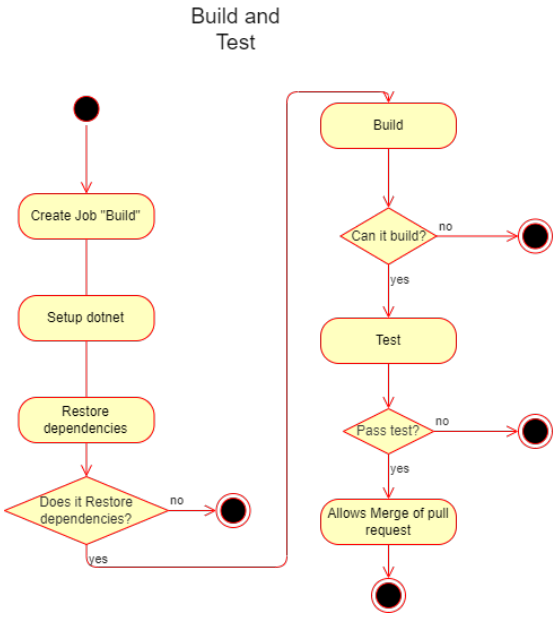

Figure 13: Build and test activity diagram

2.1.1.2 Publish and release This workflow is made to automate the creation of a GitHub release when a tag is added, see fig. 14. It will create a release of the tag. But first, the workflow in succession builds a version for Windows, MacOS and Linux. After that, it will zip the files and add them to the release if a release was made.

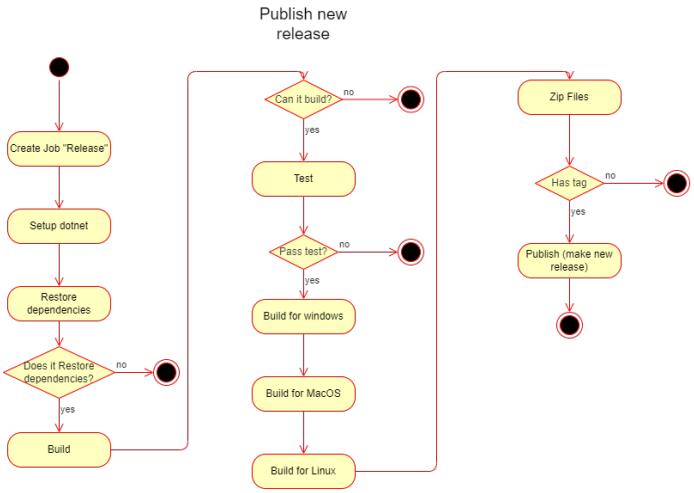

Figure 14: Publish New Release Activity diagram

2.1.1.3 Build and deploy This workflow can be seen on fig. 15. The workflow is made so it will build the program and run the "publish" command to build a version for Linux to be run on the Azure web app. After the publish command, it uploads the artefacts so the next job can use the files of the artefacts. The deploy job will download the artefacts and use their files to deploy to our Azure web app.

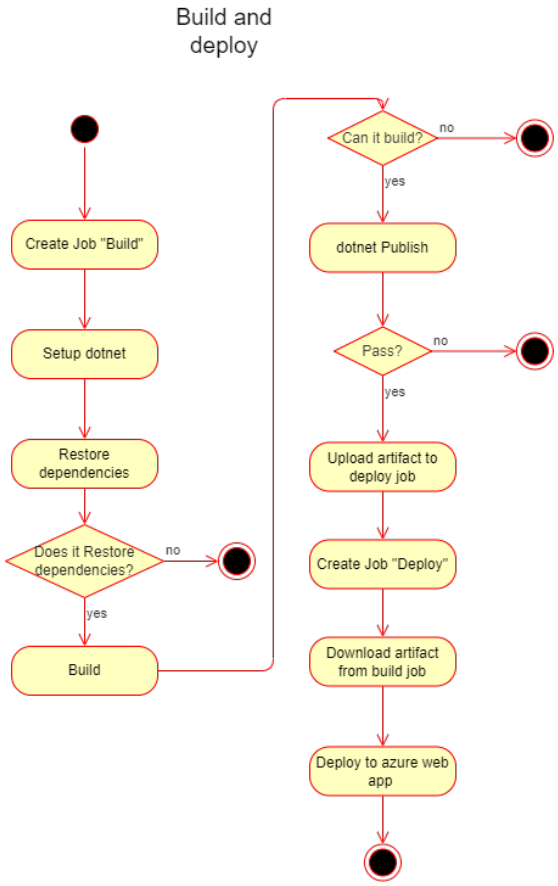

Figure 15: Build And Deploy Activity diagram

Before committing anything and thus starting the suitable workflow, we test locally with the `dotnet test` command. There is an activity diagram showing this. Tests are implemented with the logic of expected functionality in mind. Testing that the method in question works as expected. It should react as expected, both with harmless and malicious inputs.

For example, the tests for the method Create(CreateCheepDTO) can be examined. They can be found in the infrastructure tests for the tests for Cheap Repository.

First, we test that it works as intended with the intended input. After this, we challenge it in the test by giving it the wrong input and testing if validation exceptions are thrown.

3 Teamwork

This section will describe what features and implementation weren't completed and how the group worked with creation of issues and development.

3.1 Project Board

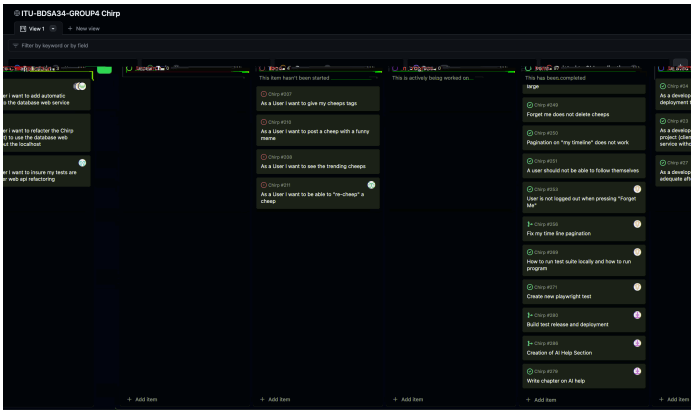

Figure 16: Project Board

This figure shows the Project board of Chirp on the day of the hand-in. We have four issues that haven't been implemented before the deadline. All four issues are under the Todo section. None of them are in the project requirements. That is to say, they were formulated under the *Wildstyle* development section. They were, adding tags to cheeps, being able to cheep a meme, trending cheeps and a re-cheep feature. As can be seen from the project board the re-cheep feature had an assigned developer but wasn't completed in time as other requirements had to be met. One *Wildstyle* feature was implemented a like button on the Cheeps. Although the like implementation is missing some functionality. A user can't see which cheep they've liked and they can like, a Cheep infinitely many times.

Three issues regarding the old retired Chirp CLI application is closed, but not implemented. The issues can be seen in the far right column, and is: - Adding automatic deployment from GitHub to the host service containing the web api. - Changing the application to use the database on the web service instead of the local hosted database - Ensuring that the test coverage are adequate after refactoring our wep api

3.2 Issue creation

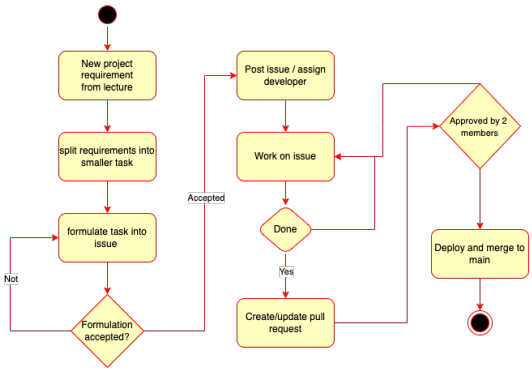

Figure 17: Flow of issues

This activity diagram shows the flow of our work process. At first, the new requirements are read and understood, and then the group gathers and tries to formulate the tasks into small issues which ideally can be completed within a day's worth of work. If a formulation gets accepted by the group it gets posted on the issue board on Github. A developer assigns themselves to an issue to let others Aknow what they are working on. When the developer feels like they've implemented the feature adequately, that is the acceptance criteria are met, they commit and create a pull request. When a pull request is posted a minimum of two reviewers from the group are needed to further merge it to main and deploy. When reviewing the code a reviewer can request changes and then further work on the issue is required. This process repeats until two reviewers accept the changes and then the code can be merged into main.

Whenever a sizable pull request was made. We had an internal understanding that even after the minimum requirements of two approved reviews. We would wait a day. This was done to include everyone and provide them with an opportunity to request changes to ensure readability and collaboration with the rest of the application.

Another more simple *"issue-creation-flow"* was also used. If a developer found a bug within the application or other small adjustments were made. An issue was created. This created good documentation for the other developers so all group members could understand why a pull request was made.

4 How to make Chirp! work locally

Prerequisites:

1. download .NET

2. IDE of your choice

4.1 Clone the repository - Step 1

Follow this link: github.com/ITU-BDSA23-GROUP4

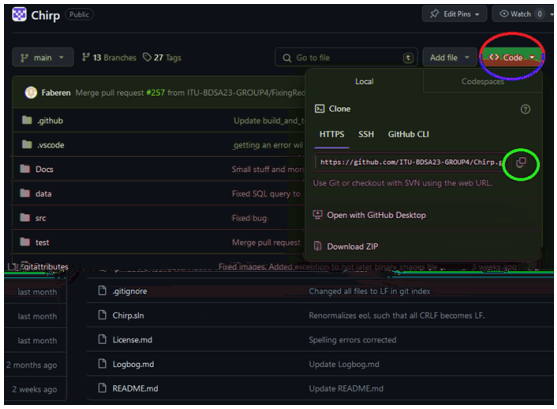

Figure 18: Cloning from git

Copy the url and run the following command in your terminal where you want to clone the repository to.

```
git clone https://github.com/ITU-BDSA23-GROUP4/Chirp.git
```

4.2 Running and installing migrations - Step 2

Navigate to the root folder of the program, run the following command in your terminal.

```
--global dotnet-ef
```

navigate to *Chirp/src/Chirp.Infrastructure* Delete all migrations file if they exists.

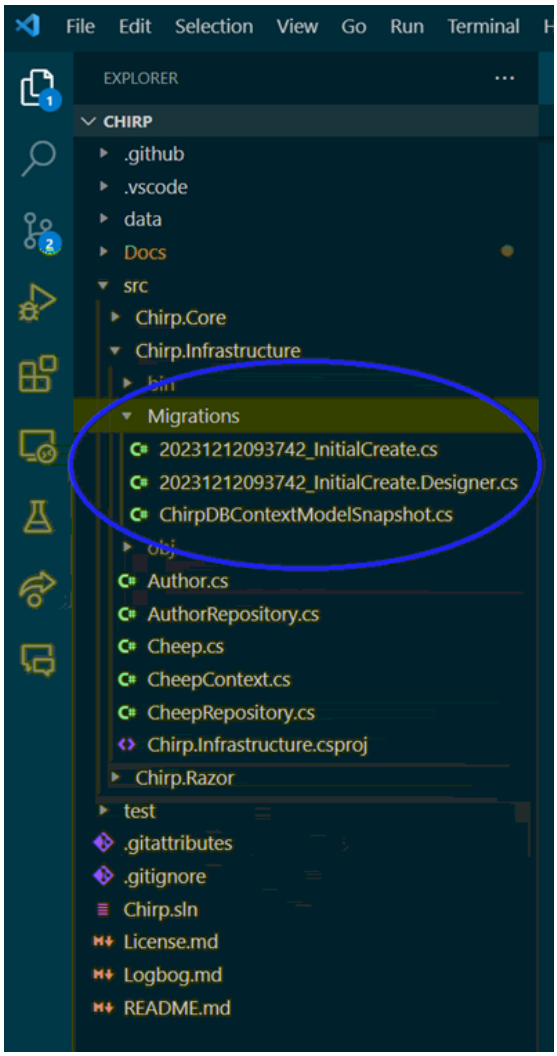

Figure 19: Deletion of migrations

Then run the following command

```
dotnet ef migrations add InitialCreate
dotnet ef database update
```

4.3 Setting up docker - Step 3

To setup the Docker container for development on your own pc you need to run the following command: `docker run -e "ACCEPT_EULA=Y" -e "MSSQL_SA_PASSWORD=Admin123" -p 1433:1433 --name chirpdb`

`--hostname chirpdb -d mcr.microsoft.com/mssql/server:2022-latest`
After this the Container should have been created and a new Image can be seen in your Docker Desktop app. With the new lines of code in Program.cs it should create the database on the container. We can all just use the same command since the connectionstring is already made for this password. hostname and port.

4.3.1 Setup Database on docker

The last step is to create the database on the docker server. To do this you are to navigate to the `Exec` on your new server. To get there go to "Containers" and click on your container.

1. Go to the "Containers"
2. Open your Container ours is "chirpdb"

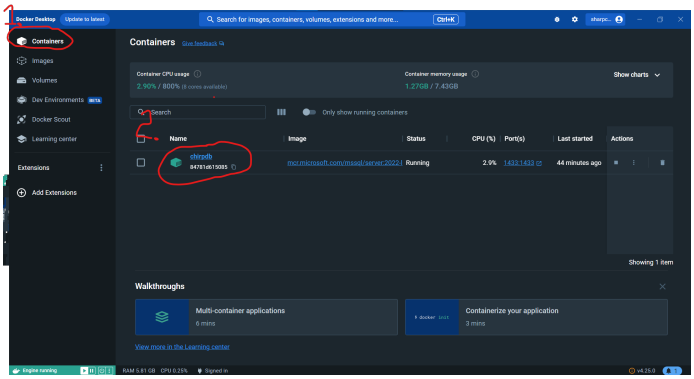

Figure 20: Docker Container

3. Open `Exec`

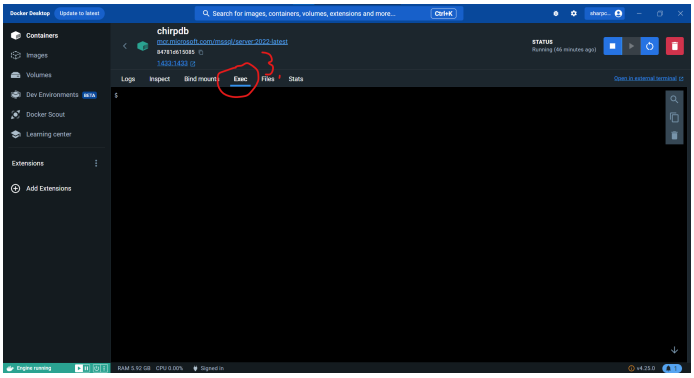

Figure 21: Docker Exec

Here you can run bash commands on your container and look around the container. We are here to use the MsSQL tool to make a database on this container. To do this we run this `/opt/mssql-tools/bin/sqlcmd -S localhost -U SA -P Admin123` (the `-U` is the user in our case we will just use SA which is System Admin and `-P` is the password for SA) this will gain access to the MsSQL tool. Here we can run SQL commands. Bare in mind that this is a different tool the usual and have different commands. The last part is to add the docker connectionstring to the user secrets. Navigate to src/Chirp.Razor and run this command. **Attention:** the line below is one continuous line and should be written like that, although the space between the ChirpDB" and "Server=localhost has to be there.

```
dotnet user-secrets set "ConnectionStrings:ChirpDB"
"Server=localhost,1433;Database=ChirpDB;User=SA;
Password=Admin123;TrustServerCertificate=True;
MultipleActiveResultSets=True;"
```

You can also give your docker container another name if you want to.

4.4 Running the program - Step 4

Navigate to *src\Chirp.Razor* and run the following command

```
dotnet run
```

The program should now be able to run correctly.

5 How to run test locally

The test suite of Chirp consists of 3 test folders each targeting their own part of the application, Infrastructure, Razor and playwright tests. All the tests are found in *Chirp/test/*

5.1 Infrastructure.Tests

No prerequisites are needed to accomplish the infrastructure test, simply cd into the *Chirp/test/Chirp.Infrastructure.Tests* folder in your terminal and run `bash dotnet test` Our Infrastructure tests targets our database and repositories, it creates an in memory database which all the test are run against.

```
var builder = new DbContextOptionsBuilder<ChirpDBContext>();
builder.UseSqlite("Filename=:memory:");
ChirpDBContext context = new(builder.Options);
_connection = context.Database.GetDbConnection() as SqliteConnection;
if (_connection != null)  //Takes care of the null exception
{
    _connection.Open();
```

```
}
context.Database.EnsureCreated();
```

5.1.1 What is tested

- AuthorRepositoryUnitTests This class targets our AuthorRepository. It performs unit tests for almost every method created in the repository with both correct and incorrect input. e.g. finding author by email or adding a follower.
- CheepRepositoryCreateUnitTests This class targets our CheepRepository. It specifically targets the methods around the creation of Cheeps. e.g. Adding a Cheep and checking if a Cheep is not empty
- CheepRepositoryUnitTests This class targets our CheepRepository. It performs unit tests on liking a cheep. e.g. liking increases a Cheeps total likes.
- InMemoryDatabaseTests This class tests if the in memory database is created correctly which is crucial for the other classes since they all rely on it.
- RestrictedCheepsUnitTests This class targets the Cheepvalidator. It performs unit tests to check if a Cheep has the correct information, such as not being empty or over 160 characters and having a valid author.

5.2 Razor.Tests

To run the tests you need to setup and download docker. A complete guide for downloading and setting up docker correctly with our application can be found here: sec. 4.3

After following the guide cd into the Chirp.Razor.tests folder and run the following command

```
dotnet test
```

5.2.1 what is tested

The razor tests consist of one class, **IntegrationTest.cs**. The class creates a local instance of our web application using the WebApplicationFactory class. With this we can test that our applications ui functions as we expect before we deploy it to azure. The test include, testing 32 Cheeps per page, ordering of Cheeps by date and the functionality of dynamic buttons.

5.3 Playwright.tests

To run the test first download playwright with the following command

```
pwsh bin/Debug/net7.0/playwright.ps1 install
```

This install various browsers and tools to run UI tests. The browser we use is chromium based. if you run in to issues with the version of .net replace net7.0 in the command with the correct version if you don't have powerShell installed follow these instructions Install PowerShell

After completing these steps you can run the test with:

```
dotnet test
```

When you run the test a chromium based browser will open and the first step tries to log in. Here the automation stops and the user has to log in through Github themselves. **No passwords are saved!** After this step is completed playwright will do the rest itself.

5.3.1 what is tested

The playwright test differs from the razor test in that it, mimics user behavior on our live website compared to the razor test which test locally. The test navigates through different pages and interacts with the website's functionality confirming that what it interacts with is as expected in the test.

6 Ethics

6.1 License

We chose the *MIT license* for our application, with the major reason being it's open-source nature towards programming-collaboration. Furthermore all the dependencies which we use in our application are also under the MIT license except one, which encourages the collaborative nature of the programming community. A list of all our dependencies and their licenses can be found in the license.md file in the Chirp application. One of our packages is under the *Apache-license*, which is fine since both are permissive licenses meaning they are able to be used together. This is also stated in our license file.

6.2 LLMs, ChatGPT, CoPilot, and others

In this section we will go over the AI help that has been through out the process of creating the project.

6.2.1 ChatGPT

ChatGPT is one of the AI's that we have used through out the project for smaller questions. Theses questions range from C# related code, refactoring of code, setup of docker or .NET console commands. With the help from the AI we can ask questions and get quick response that can help us in the right direction. There are somethings to consider when using ChatGPT the main thing will be the validity of the response since we can't guarantee that it's right.

6.2.2 Github CoPilot

Github CoPilot has been used through out the project to help with speeding up the process of writing code. This range from repetitive code to unit tests. This is where CoPilot shines with helping auto completion. However, the recommendation that you get from CoPilot may not work since the AI can't know the full extend of the program.