Chirp! Project Report

ITU BDSA 2024 Group 5

Helge Pfeiffer ropf@itu.dk Adrian Hoff adho@itu.dk

1 *Chirp!* Project Report

ITU BDSA 2024 Group 5

- Markus Sværke Staael msvs@itu.dk
- Patrick Shen pash@itu.dk
- Frederik Terp fter@itu.dk
- Nicky Chengde Ye niye@itu.dk
- Marius Cornelius Wisniewski Larsen coml@itu.dk
- Sara Ziad Al-Janabi salj@itu.dk

2 Table of Contents

3 Design and Architecture of *Chirp!*

3.1 Domain model

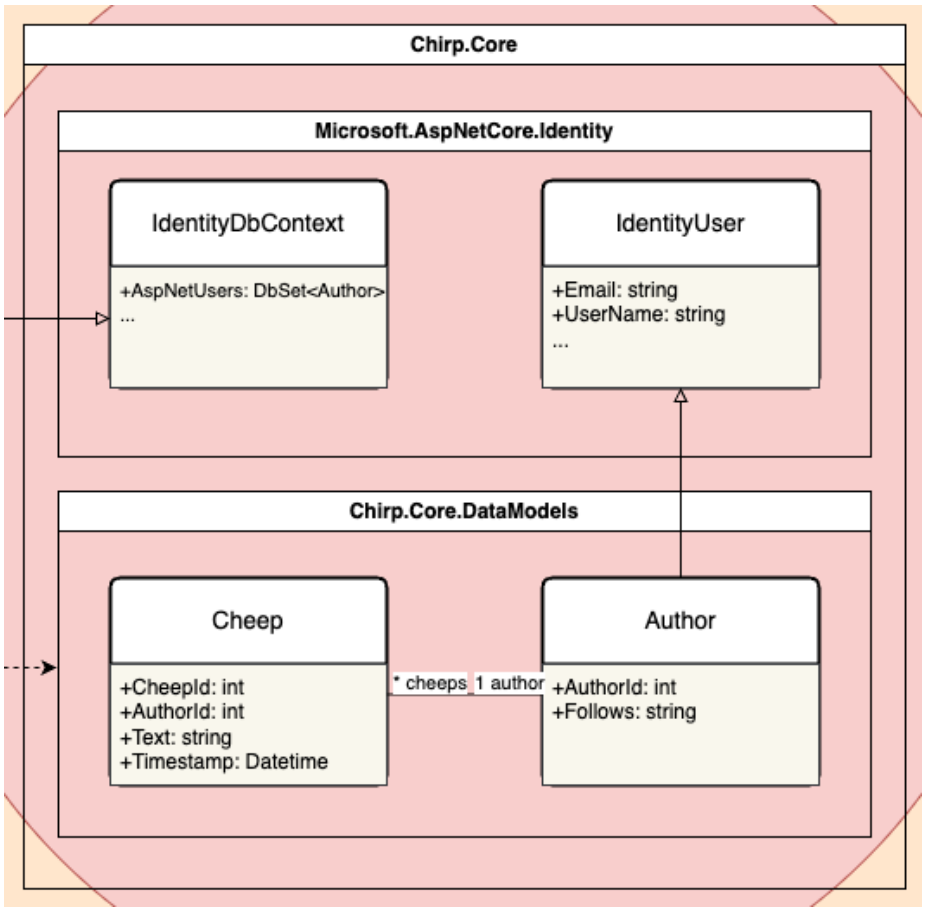


Figure 1: Illustration of the *Chirp!* data model as UML class diagram.

The Chirp application actively utilizes an onion architecture to promote a clear separation of concern. The onion has many layers but the core of it is Chirp.Core, where the domain model resides. The domain model is relatively simple and represents authors and cheeps. The author model extends an IdentityUser from Asp.Net Core Identity to make it work seamlessly with the rest of the Asp.Net Core ecosystem.

3.2 Architecture — In the small

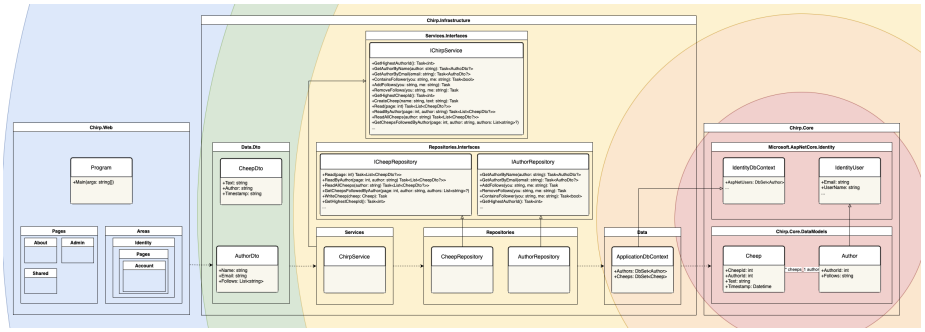


Figure 2: Illustration of onion architechture.

As previously mentioned the onion architecture has many layers, but so far we have only covered the core. The rest of the layers are categorized as Chirp.Infrastructure and Chirp.Web with the thickest layer being the infrastructure layer.

The infrastructure layer can be further broken down in three sublayers. Starting from the inside and moving out there is a ApplicationDbContext that extends an IdentityDbContext to make it work seamlessly with the rest of the Asp.Net Core ecosystem. The purpose of the ApplicationDbContext is to provide a way to interact with the entities in the database.

The next layer is the repository layer that interacts with the ApplicationDbContext by implementing methods to extract specific data from the database. To comply with the repository pattern there are two repositories, the author repository and the cheep repository that both interact with their respective entities in the database.

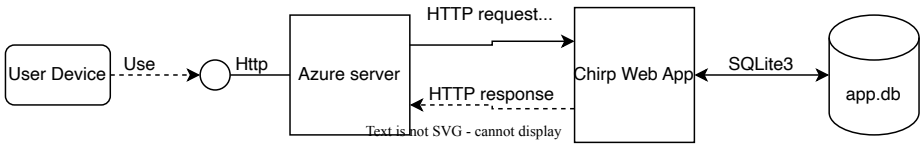
To interact with both author and cheep entities in a simple manner, a chirp service is implemented that uses the two repositories. The service combines the two repositories by implementing identical methods that call the repository methods. Another purpose of the service is also make development easier by providing only a single point of access to the database, to be injected.

Both repositories and the service implement respective interfaces to enable dependency injection, which makes it easier to test for functionality and coverage.

The last layer of the infrastructure layer is the data transfer object layer. The data transfer objects serve the purpose of only providing the necessary data to not expose the entire domain model to the user as there can be sensitive or unnecessary data.

The web layer is the outermost layer and is responsible for handling the frontend portion of the chirp application by providing a user interface.

3.3 Architecture of deployed application

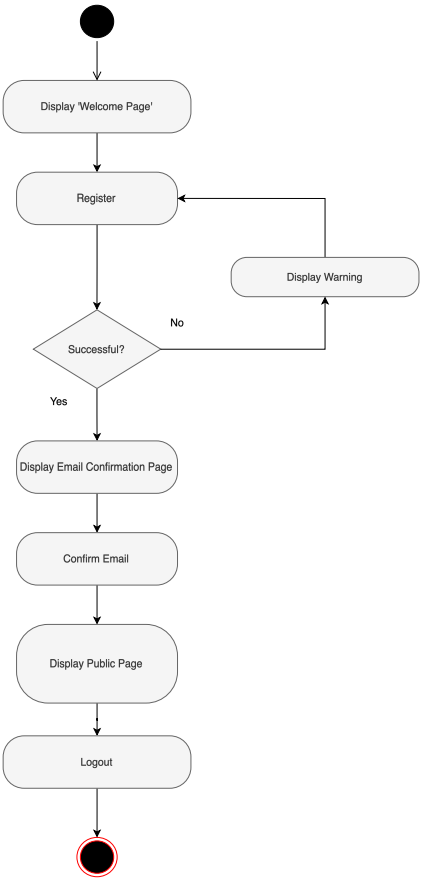


As illustrated the user send requests and gets responses to the azure server through the http-protocol. Multiple clients can connect to the azure server at a time. The azure server sends a http request to the chirp web app with a required cookie for the user session. The Chirp web app communicates with the database itself through sqlite3. The real deployment uses the https-protocol which ensures that the vulnerable user data found in the responses are encrypted with a tls-certificate.

3.4 User activities

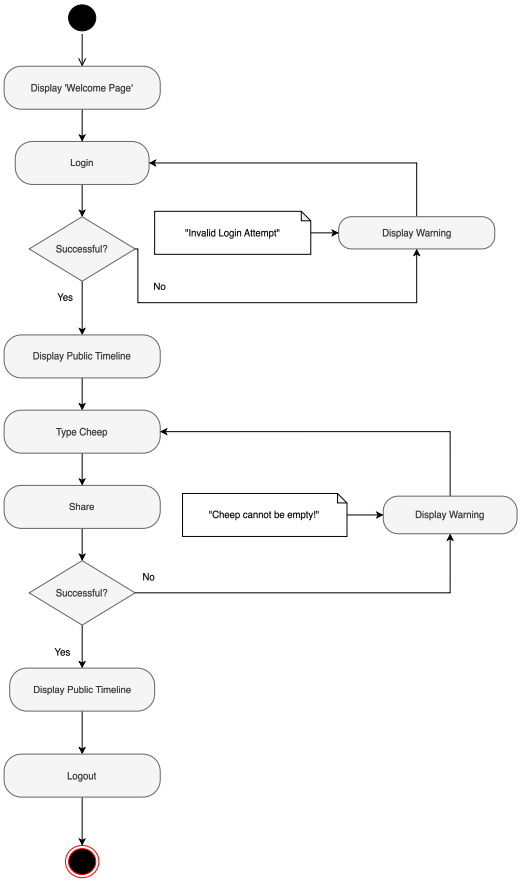
This section illustrates typical scenarios that the user may go through when using our `Chirp!` application. This goes for both unauthorised and authorised users, in which both cases have been included. The illustrations are shown as sequence of activities in the format of UML Activity Diagrams.

3.4.1 Register



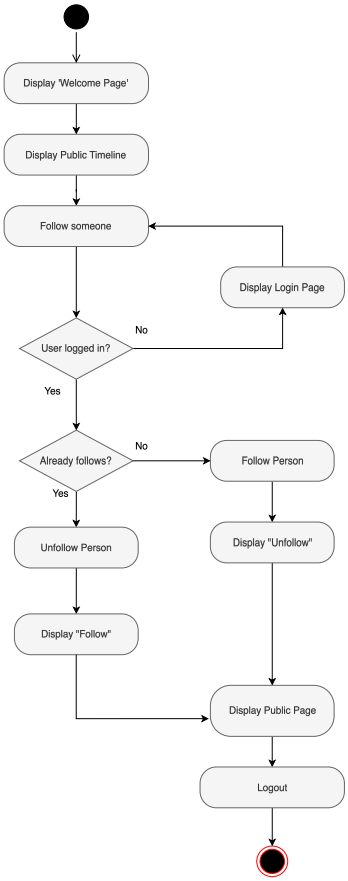
This diagram illustrates the registration of a user. When a user registers, if all criteria fulfilled, they will be led to the email confirmation page. In the case of a missing criteria, e.g. the user has typed an invalid e-mail address, the warning displayed will inform the user about said missing criteria.

3.4.2 Submit cheep



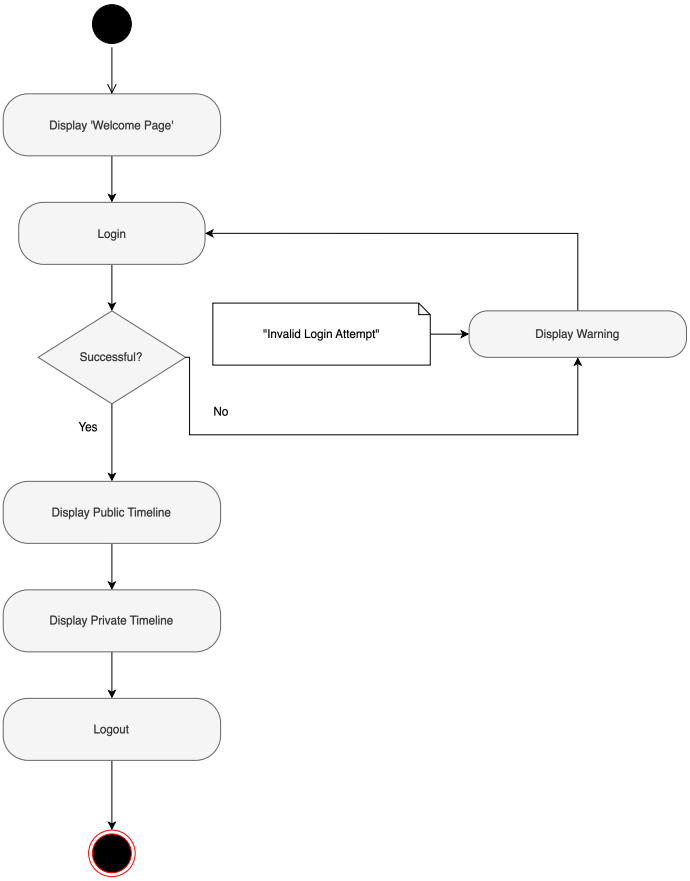
This diagram displays the sequence of user activity, if the user wishes to type a cheep.If the message box is empty, a warning will be displayed.

3.4.3 Follow



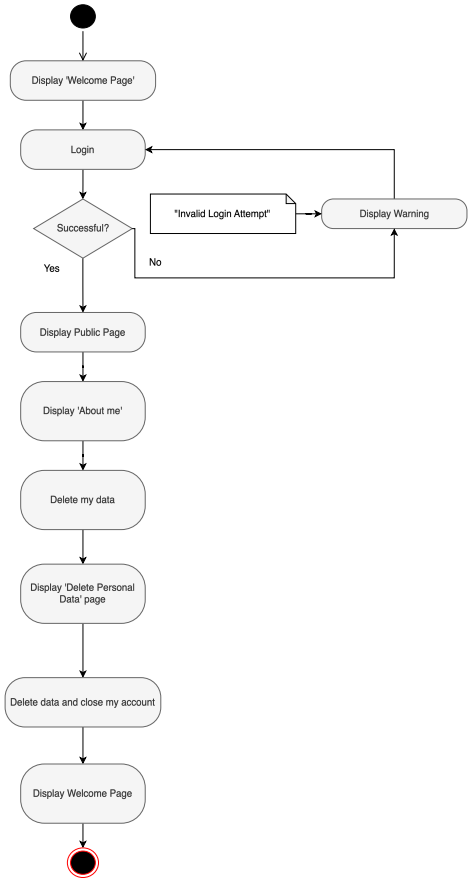
This diagram shows what occurs once a user tries to follow another user. If user isn't logged in, they will be redirected to the login page. Otherwise, whether the user already follows someone else or not, either 'Follow' or 'Unfollow' will be displayed.

3.4.4 User login



This diagram simply views the sequence if a user wishes to view their own page. User must be logged in before being able to do so.

3.4.5 Delete account



If a user wishes to delete their data, this user activity sequence would be a typical scenario.

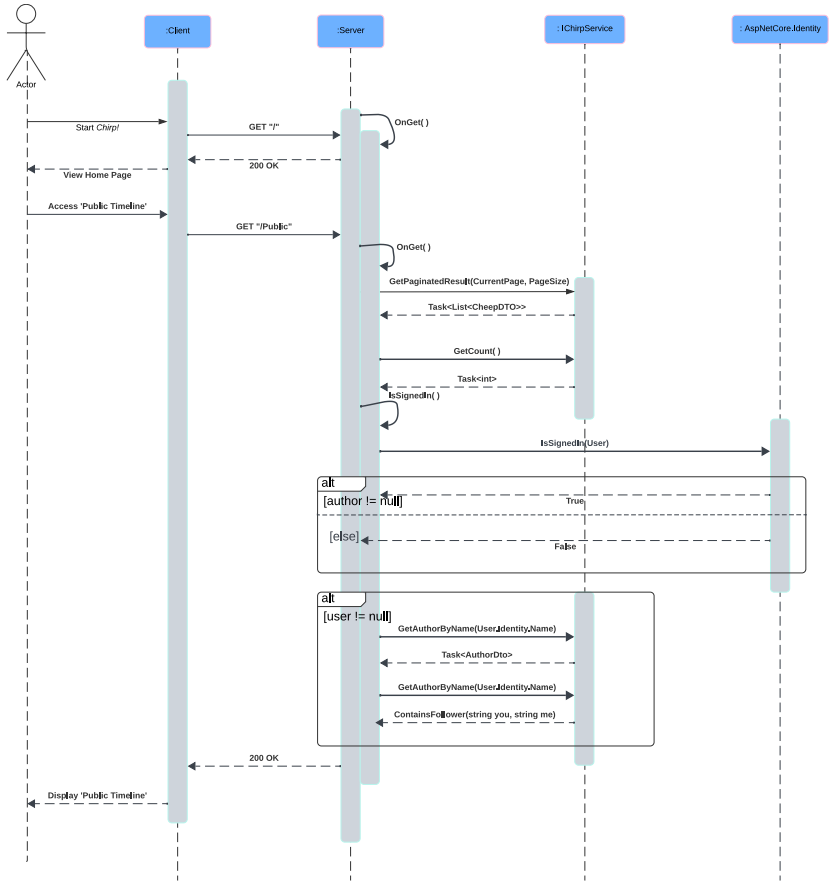
3.5 Sequence of functionality/calls through *Chirp!*

When running the application, there are required flows of messages and data sent back and forth all the way from the requests from the user to the communication between the server and ASP.NET Core. We have chosen to create UML Sequence Diagrams, that show how the system works, and how each entity interacts with each other. The intention of the diagrams is to visualise the 'behind-the-scenes' of a user-request to the final rendered webpage shown to the user.

We have chosen to illustrate the following sequences: 1. when a user registers a new account, and login 2. when a user accesses the Public Page 3. when a user accesses their own private timeline 4. when a user types a cheep 5. when a user deletes their account 6. when a user follows someone else

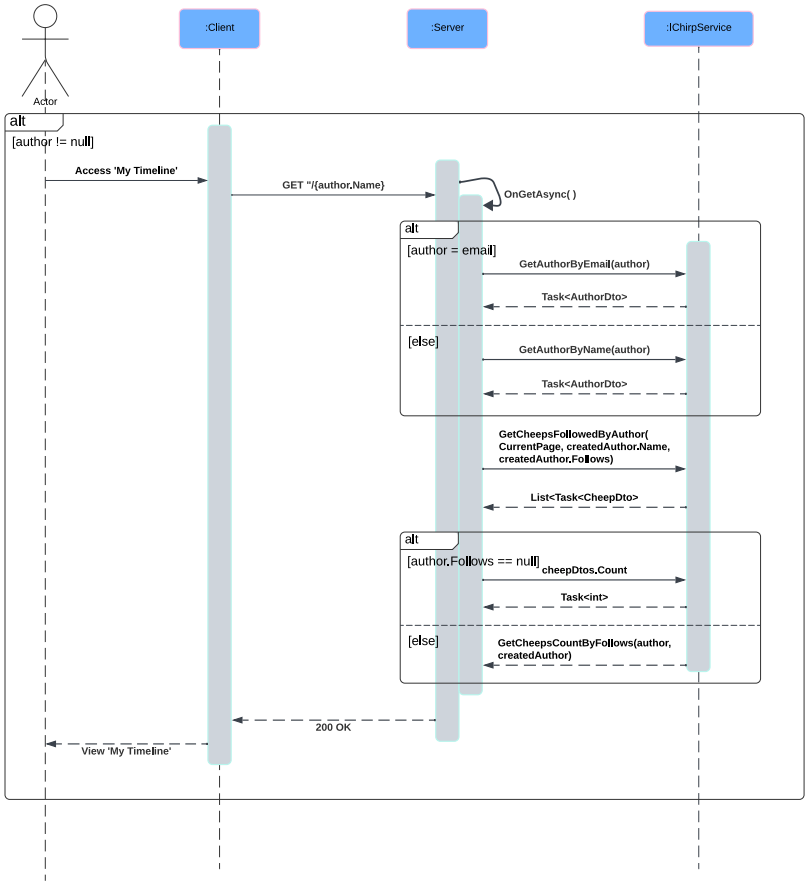
The diagrams are shown below:

3.5.1 Public Page



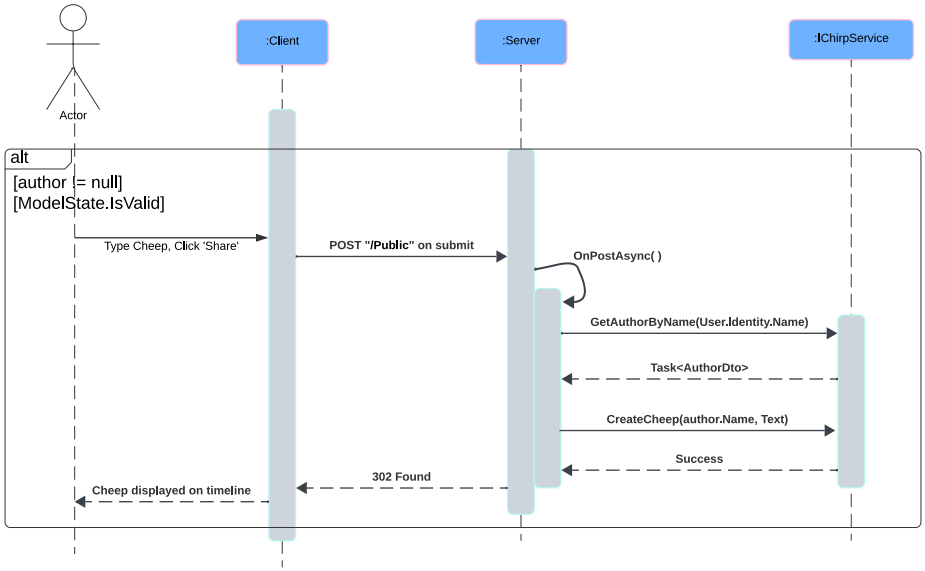
This diagram shows the flow from when a user starts the application, and then tries to access the Public Timeline-site.

3.5.2 Public Page



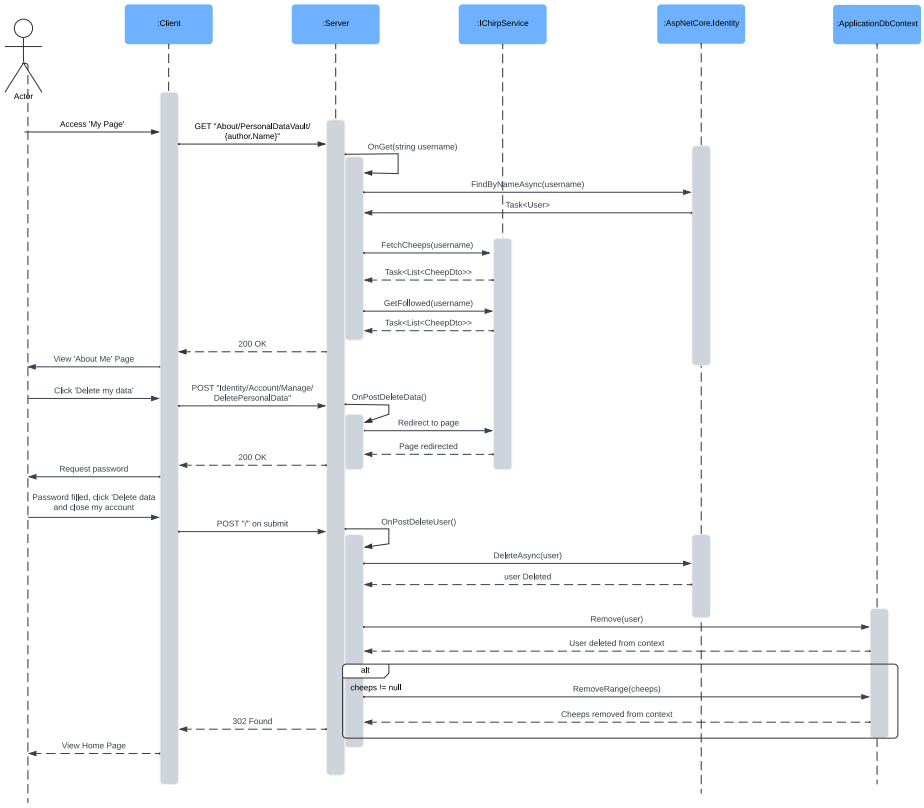
This diagram shows the flow of a user accessing their own timeline, 'My Timeline'. This sequence is only available when a user is logged in (as shown in the diagram).

3.5.3 Type cheep



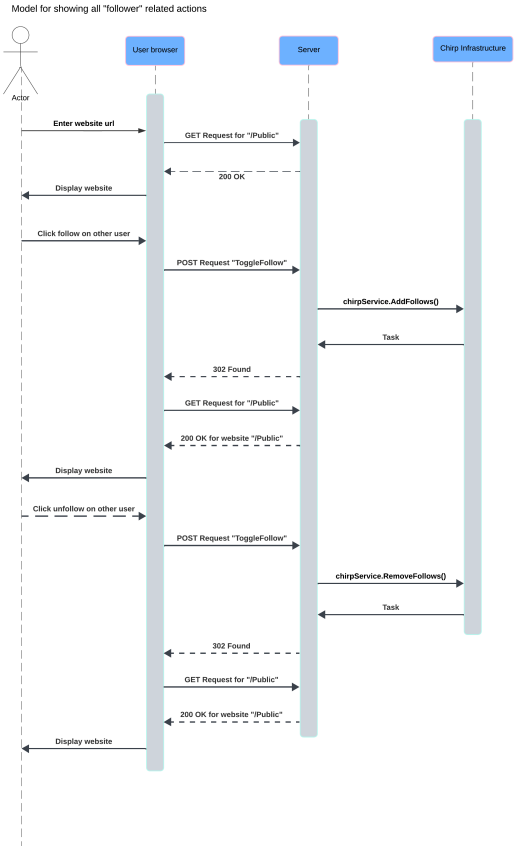
This diagram shows the interaction between the entities when the user wants to type a cheep in the application. This function is only available when a user is logged in (as illustrated in the diagram).

3.5.4 Delete account



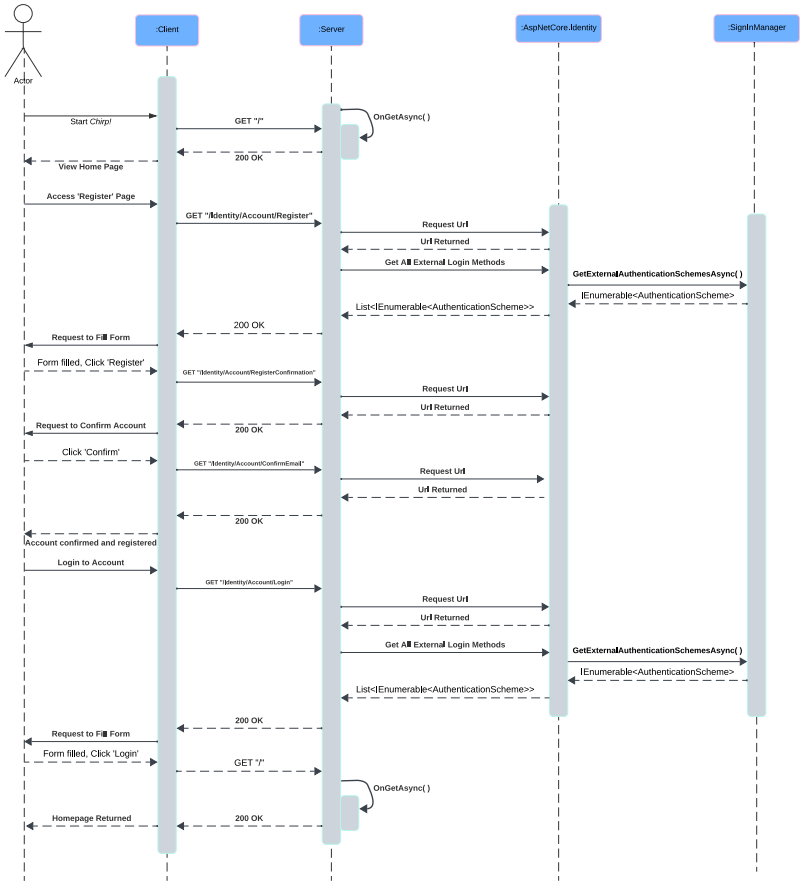
This diagram shows the interaction between the entities when a user decides to delete their account.

3.5.5 Follow diagram



This diagram views how the user accesses the public page, and chooses to follow and unfollow another user from said page.

3.5.6 Register

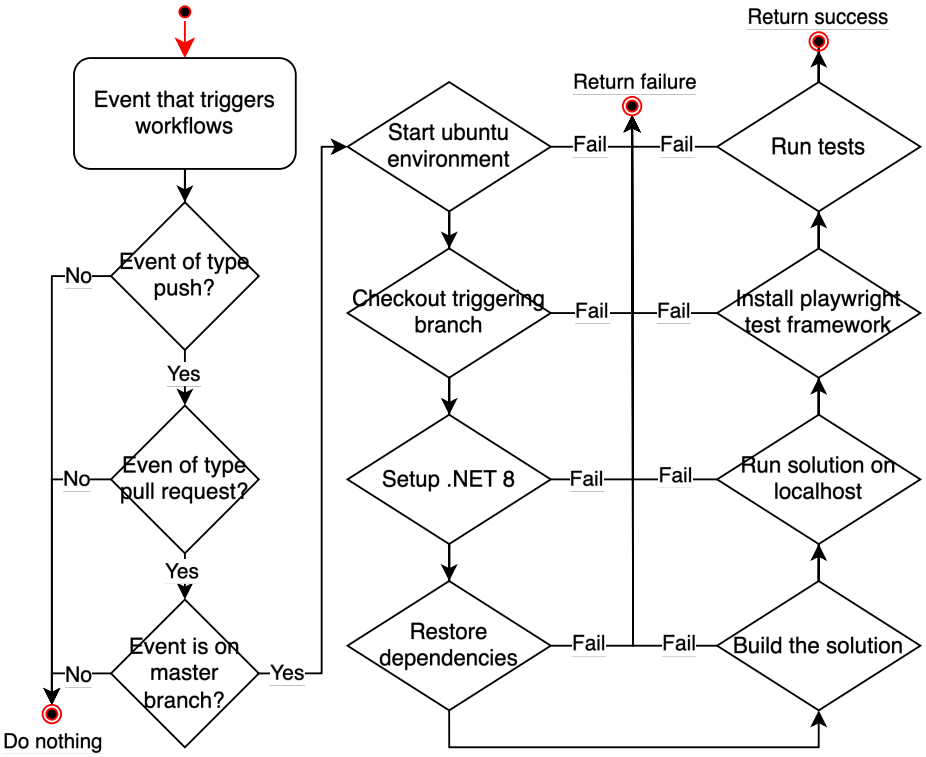


This diagram shows the flow from when a user starts the application and wants to register a new account. After registering, the user logs in to their newly registered account.

4 Process

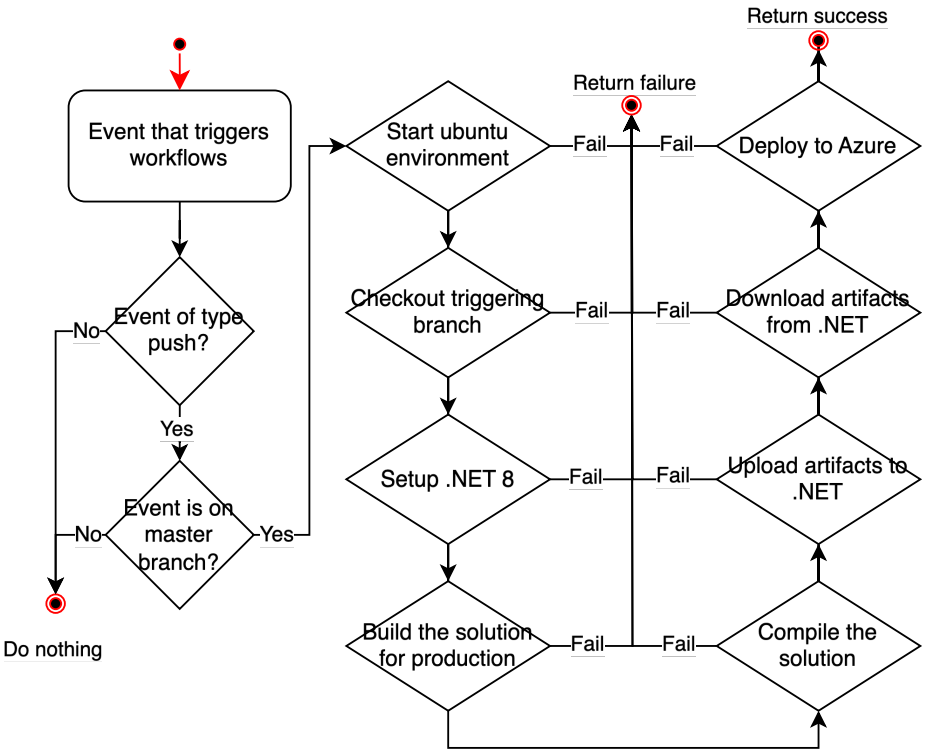
4.1 Build, test, release, and deployment

4.1.1 build_and_test



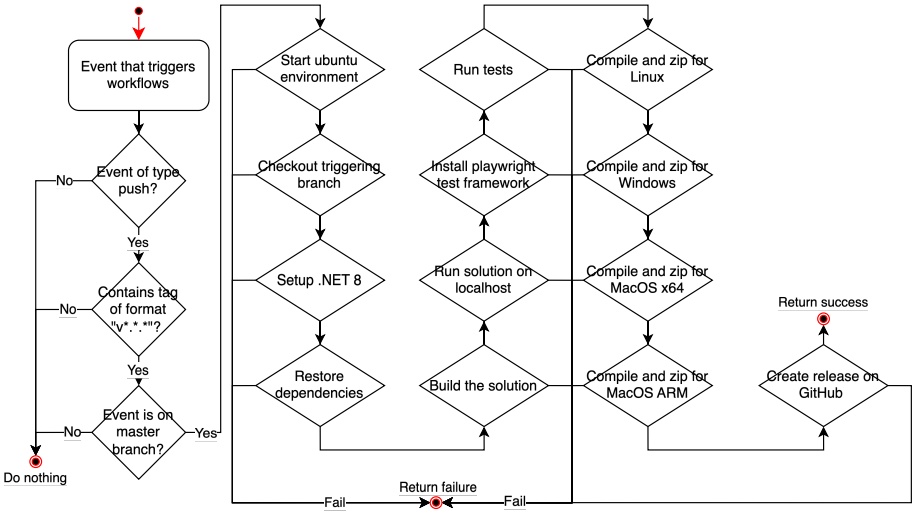
This workflow builds and tests the code on push and pull-requests on the master branch. When this condition is achieved it restores dependencies, builds with no restore because of the last step and attempts to run it locally. Then it runs all tests made, but before running the tests it installs the test-framework "playwright" that the tests found in test/PlaywrightTests depend on. The ones found in test/Chirp.Razor.Test are run by xUnit.
If any of these steps fails the workflow fails and the push or pull-request on master branch is cancelled. If not it proceeds with the action.

4.1.2 master_bdsagroup5chirprazor2024



This workflow is triggered on push at master branch and is responsible for deploying the code/build to azure for running the web application. When triggered it creates a build with the release configuration. Next it publishes the project to the output folder defined after -o and uploads the published folder as an artifact for the azure web app to deploy The deploy job deploys the application to the Production env with the webapp url.

4.1.3 release.yml



Triggered when adding the following tag on push: `- v*.*.*` The steps including restore, build and tests are the same and in the previously mentioned build_and_test workflow. If that succeeds it proceeds with the workflow by publishing the following project files:

1. `src/Chirp.Core/Chirp.Core.csproj`
2. `src/Chirp.Infrastructure/Chirp.Infrastructure.csproj`
3. `src/Chirp.Web/Chirp.Web.csproj`

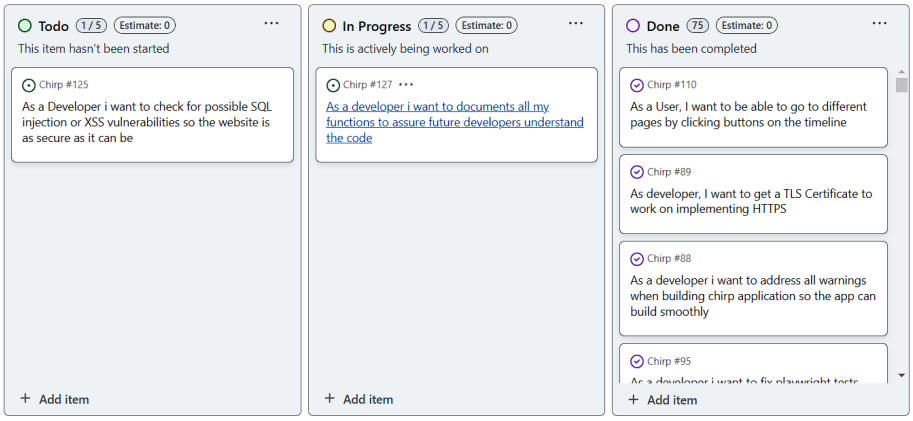
With the following release configurations: linux-x64, win-x64, osx-x64 and osx-arm64 with an corresponding output folder for it and zipping it. The release then include those zip-files and the source code

4.2 Team work

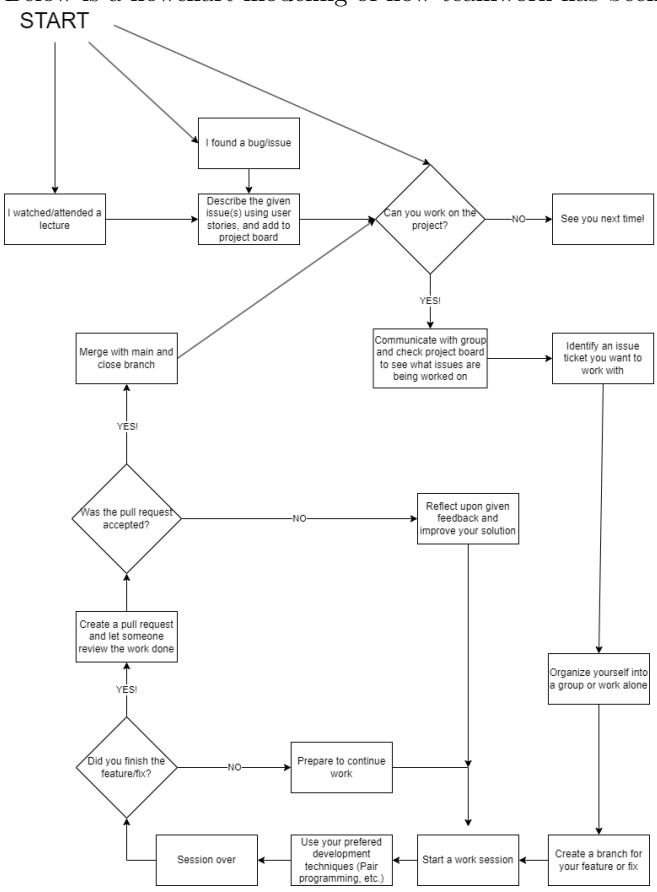
Below is the project board for group 5. The uncompleted tasks are:

1. `As a Developer i want to check for possible SQL injection or XSS vulnerabilities so the w`
2. `As a developer i want to documents all my functions to assure future developers understar`

All other features have been completed, this solution for Chirp should not be missing any features or functionality.



Below is a flowchart modeling of how teamwork has been conducted in group 5.



The process “from issue to solution” starts after all members having attended any weeks lecture. Shortly after that lecture the group will find a room to sit and organize themselves, here a few group members start identifying and then

quantifying this weeks problems by creating issue tickets. When all problems have their respective issue tickets, the group will subdivide itself and create smaller groups where individuals work together to solve the specific issue(s). A new branch will be created where all work for the feature/fix will be deposited. Whenever a specific issue is solved, its respective branch may be merged into main and their issue will be closed. If an issue is not solved during that day, individuals will work from home to solve/close the issue, or if needed, the group will meet again before the next weeks lecture (when new issues will be added).

4.3 How to make *Chirp!* work locally

4.3.1 Running from Compiled

1. Access our release page.
2. Download the zip containing the compiled version of the program corresponding to the system you want it to run on.
3. Unzip the zip file into a given directory / %unzippedcontentdir% and run

   UNIX-based systems

   3.1. Open up terminal and run the following

   3.2.

   ```
   cd %unzippedcontentdir%/publish/%systemarchitecture%
   ./Chirp.Web
   ```

   Windows

   3.1. Open up CMD and run the following

   3.2.

   ```
   cd %unzippedcontentdir%/publish/win-x64
   Chirp.Web.exe
   ```

4. The terminal/cmd should now show the following: Now listening on: http://localhost:5000
5. Accessing your localhost on the given port should now give you access to the local running instance of the web-app

4.3.2 Running from Source code

1. Pull the source code from github, can be done by opening terminal/cmd and typing the following

```
git pull https://github.com/ITU-BDSA2024-GROUP5/Chirp.git
```

2. Navigate to the project directory and run

```
cd Chirp/src/Chirp.Web
dotnet run
```

3. By default the terminal should now show - Now listening on: http://localhost:5177

If not follow the following steps. 1. Run `dotnet dev-certs https --trust`

2. Set user secrets by

```
dotnet user-secrets init
dotnet user-secrets set "authentication:github:clientId" "YOURCLIENTID"
dotnet user-secrets set "authentication:github:clientSecret" "YOURCLIENTSECRET"
```

4.4 How to run test suite locally

The test suite is seperated into two folders: Chirp.Razor.Test contains unit and integration tests, and PlayWrightTests contains end-to-end tests. Unit tests are made for all the methods in AuthorRepository and CheepRepository. A couple of integration tests are created in the Razor Page framework. The end-to-end tests are run with PlayWright and tests the UI elements Chirp by simulating user input.

5 Ethics

5.1 License

The license chosen for the program is the MIT license due to its simplicity and flexibility. The license is short and transparent, making it easy to understand. It has minimal restrictions and allows for both commercial and non-commercial use. Anyone wanting to use the source code are allowed to use it for their purposes but as it is, meaning that the source code is delivered as is without any warranty and that we the developers do not hold any liability .

5.2 LLMs, ChatGPT, CoPilot, and others

LLMs have been used in a limited capacity in the development of Chirp. CoPilot, as an integrated feature in Rider and GitHub, have mainly been used in code generation/assistance by cutting down the time spent on writing generic code. Occasionally, LLMs has been used for bug fixes. This is usually done by giving ChatGPT a block of code along with the prompt “Please fix” to identify simple syntax errors, which may have been hard to spot. Aside from directly in-code, LLM has been used in research as a substitute for search engines and documentation. An example from development would be using ChatGPT to research how to implement “Identity Core”.

In terms of helpfulness, LLMs have been used in cases where advice and guidance was needed and not when looking for a direct solution. The benefits of using

LLMs are comparable to asking a TA for help in understanding a certain topic or troubleshooting. When presented with an incorrect or false response, it is simply dismissed and heeded as bad advice.

Generally, the use of LLMs have sped up the development by enabling individual work by giving benefits similar to peer-programming.