Chirp! Project Report

ITU BDSA 2025 Group 13

Dimitri Alessandro Nielsen dini@itu.dk
Karl Karl Theodor Ruby krub@itu.dk
Lukas Shaghashvili-Johannessen lush@itu.dk
Matthias Nyman Nielsen mnni@itu.dk
Matthias Schoenning Nielsen mscn@itu.dk

Contents

1 Design and Architecture of *Chirp!*

1.1 Domain model

The domain Model of the *Chirp!* application is centered around two primary entities, the `ChirpUser` and the `Cheep`. The `ChirpUser` serves as the domain's representation of a user, it extends the standard ASP.NET Core `IdentityUser` to leverage built-in security while adding custom features, such as the ability to maintain lists of followers and followed users. The `Cheep` is the main form of communication and content on our platform, encompassing the text, timestamp, and an association with its author. The model further improves user interaction through a *Like* system (implemented as a many-to-many relationship) and a nested reply structure, where `Cheeps` can reference a parent `Cheep` to form conversation trees.

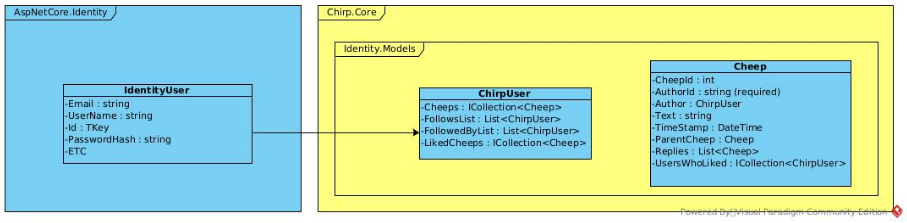


Figure 1: Domain model for ChirpUsers and Cheeps

1.2 Architecture — In the small

Below is an onion architecture diagram to illustrate the overall architecture of the *Chirp!* application. The diagram also illustrates dependencies, where the outer circles depend on the inner circles.

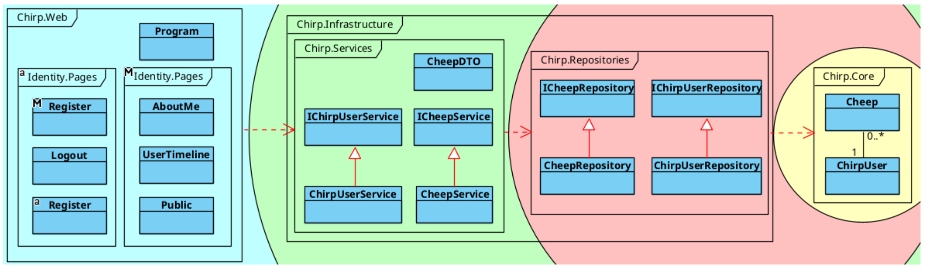


Figure 2: Dependencies are illustrated as red arrows.

1.2.1 Domain Entities

`Chirp.Core` is the center of the architecture, in the yellow part of the diagram. This layer stores the most fundamental parts of the codebase. In this project `Chirp.Core` stores the `Cheep` and `ChirpUser` domain entities.

1.2.2 Repository Layer

The repository layer of the code based is in the red part of the diagram. This layer is responsible for retrieving domain relevant information from the database.

1.2.3 Service Layer

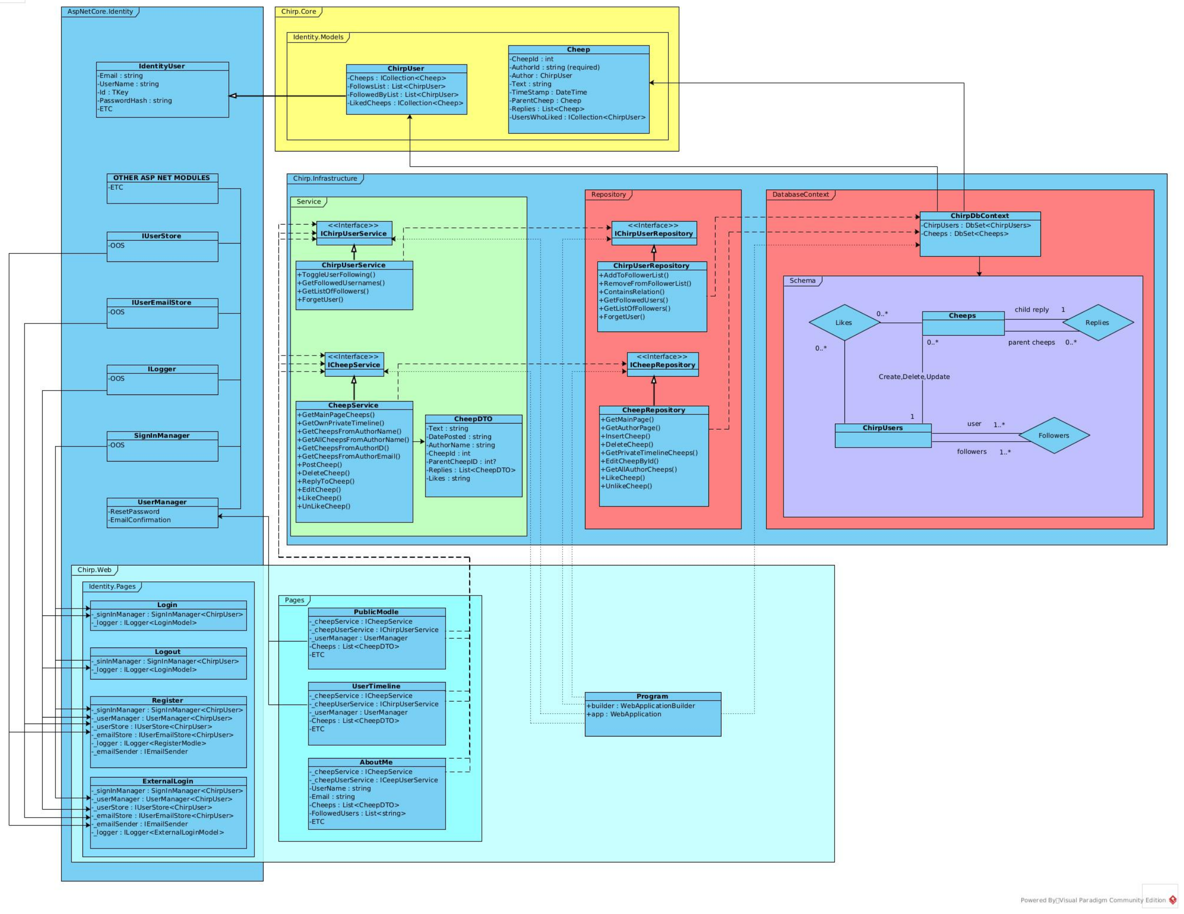
The service layer is in the green part of the diagram. This layer is responsible for translating the domain models into DTOs (Data Transfer Object) and connecting requests to the UI. This layer therefore acts as a binder between the repository and the UI layer. When a user request is received the service layer handles that requests, retrieves information from the infrastructure layer, and translates the information received into DTOs. These DTOs are then used by the UI layer to display information and data to the user.

1.2.4 UI Layer

The UI layer is in the blue part of the diagram. Here the UI is displayed to the user via `.cshtml` pages. Here *page models* send user requests to the service layer and decide the state which to display for the user. The state can change over the lifetime of the application, for example, when the user logs in, which changes the formatting of the page.

1.3 System Architecture Overview

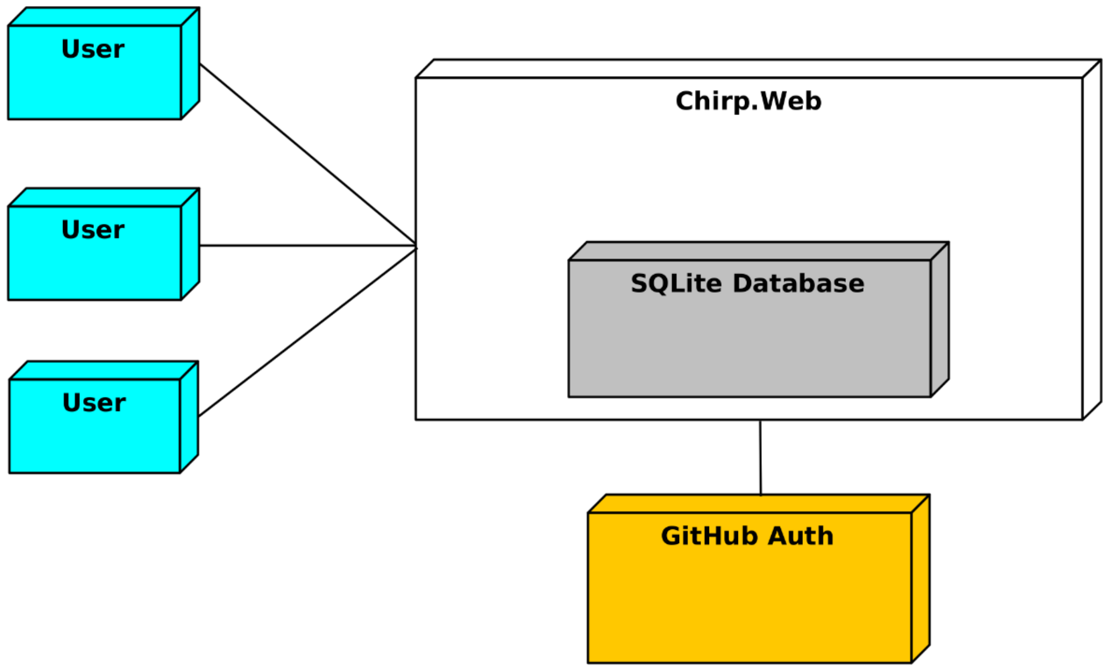
The model below shows the general flow of the System and its Architecture.



1.4 Architecture of Deployed Application

Our website is hosted on Azure via their *App Service*, using the free F1 plan which comes with some restrictions, e.g., a maximum of 1 hour of shared vCPU time per day. This allows us to run our application in a live, production environment and to host our website online with SSL certification.

1.4.1 Diagram of Deployed Application



Since clients can login via OAuth (GitHub), our service would be dependent on the availability of GitHub as an OAuth provider.

1.5 User Activities

1.5.1 Activity Diagram for Unauthorised- and Authorised Users

Below is an activity diagram illustrating what actions the user can take when they are either authenticated or unauthenticated.

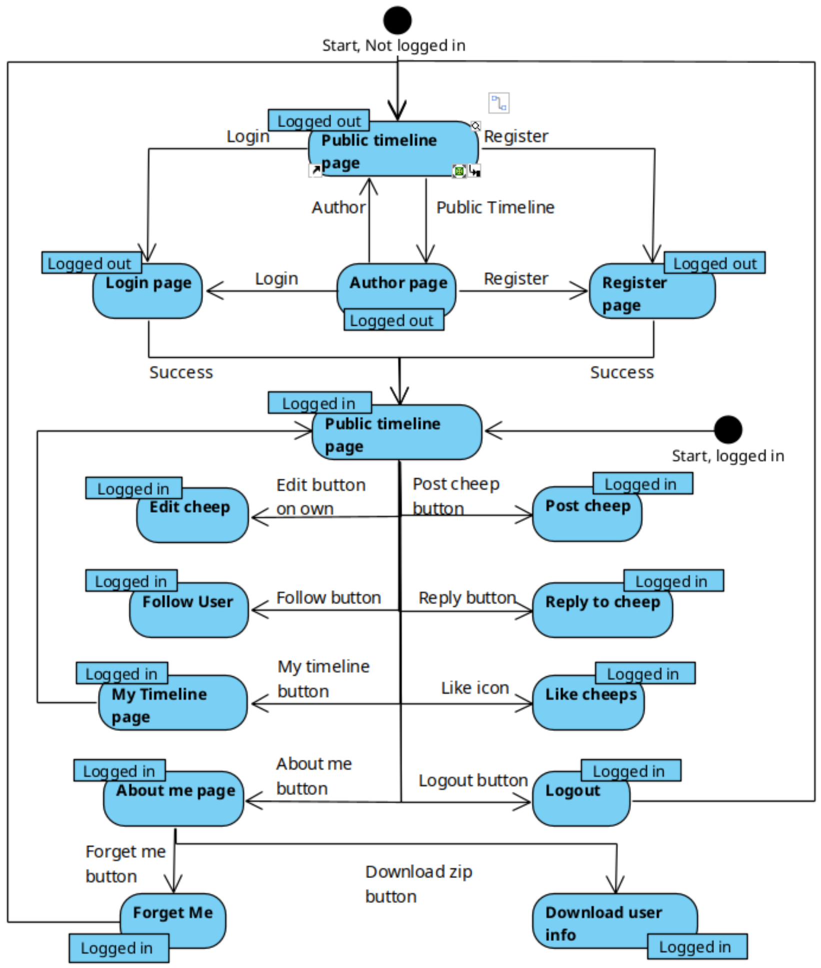


Figure 3: Activity diagram for unauthenticated- and authenticated users

1.5.2 Follow User

Below is an activity diagram illustrating what happens when a user tries to follow another user. Following has the effect of adding the followed user's `Cheeps` to one's own *My Timeline*. Following is therefore essential for when a user wants to see what new `Cheeps` the other user has posted.

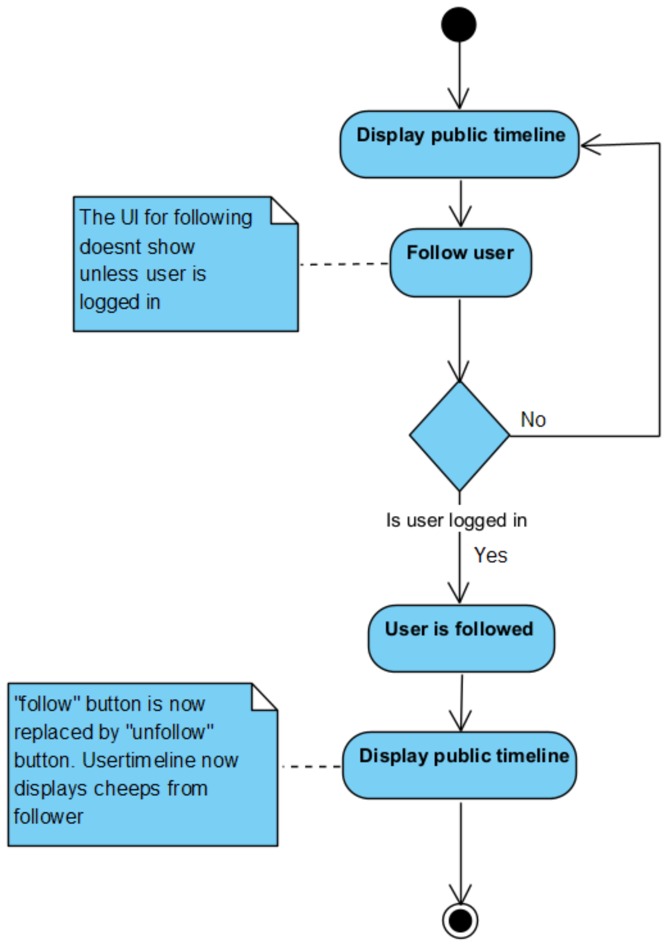

Display public timeline
The UI for following doesnt show unless user is logged in
Follow user
No
Is user logged in
Yes
User is followed
"follow" button is now replaced by "unfollow" button. Usertimeline now displays cheeps from follower
Display public timeline

Figure 4: Activity diagram of a user following another user

1.5.3 Delete User

The diagram below shows the actions performed when a user tries to delete their data. This feature is called *Forget Me!* in the *Chirp!* application, and can be performed under the `/user/<username>/about` endpoint. It is worth noting that the *About Me* site exists for every user, but the information on the site is only loaded for the user who is authenticated on the platform, meaning, *user1* cannot access the *About Me* for *user2*.

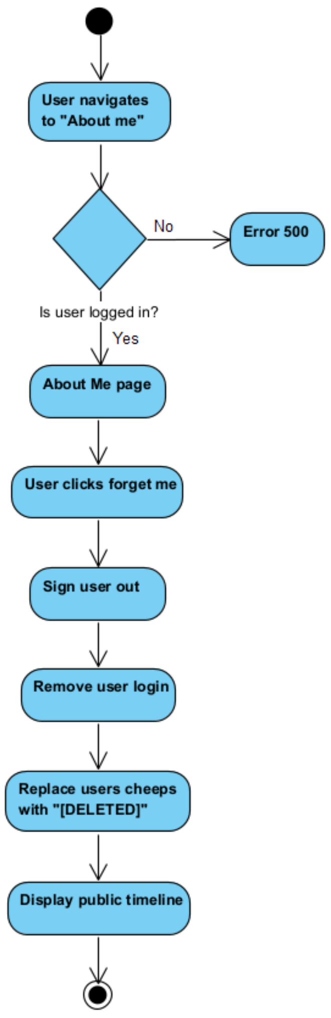
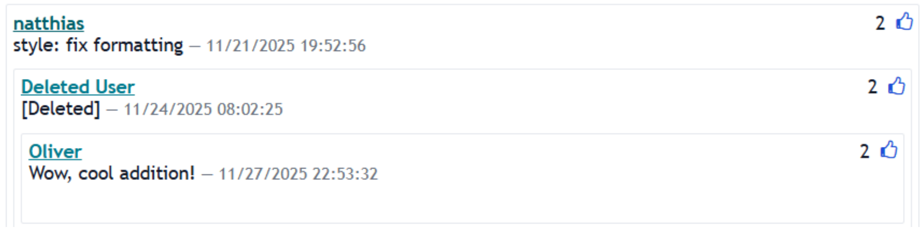


Figure 5: Activity diagram of a user trying to delete their information

When deleting user data, shown in the illustration after “User clicks forget me”, an important design decision had to be made. When the user deletes their data, they expect it to be deleted. Normally in systems this effect can be obtained by either soft deleting or hard deleting user data and information. Before GDPR, a lot of software used to just mark data as “deleted” in databases and never query them again. Now, because of GDPR, it is mandatory by law to always delete or anonymise user data when requested to do so, or when it is no longer necessary to keep that data stored[1].

Hard deletes often create a lot of problems behind the scenes, problems like syncing, irreversible data loss and compromising database schema integrity. For the *Chirp!* application there was the issue of what to do with replies. Since replies are linked with a child-parent relation, deleting a parent `Cheep` would result in all subsequent child `Cheeps` being deleted. This is why we opted for a deletion style similar to Reddit. On Reddit, posts and replies are not removed and deleted, but instead noted as *[Deleted by user]*. With this method users will not lose their replies, simply because the author of the main `Cheep` decided to delete their post. An example of the visual effect of anonymisation of user data can be seen below.


natthias
style: fix formatting — 11/21/2025 19:52:56
2

Deleted User
[Deleted] — 11/24/2025 08:02:25
2

Oliver
Wow, cool addition! — 11/27/2025 22:53:32
2

Figure 6: Here a user who has replied decided to delete their post. With a hard removal of posts, the user *Oliver* would have lost his reply in the thread.

[1] Ante

1.5.4 Login

When a user tries to log in they have the option of either creating an account directly on the website, or using GitHub as an external login service. When a user logs in with GitHub, the application automatically fetches the user data necessary for account creation. The user is then auto-redirected to the public timeline, after GitHub returns a valid authentication. Below is a diagram of a typical scenario of a user logging into the *Chirp!* application.

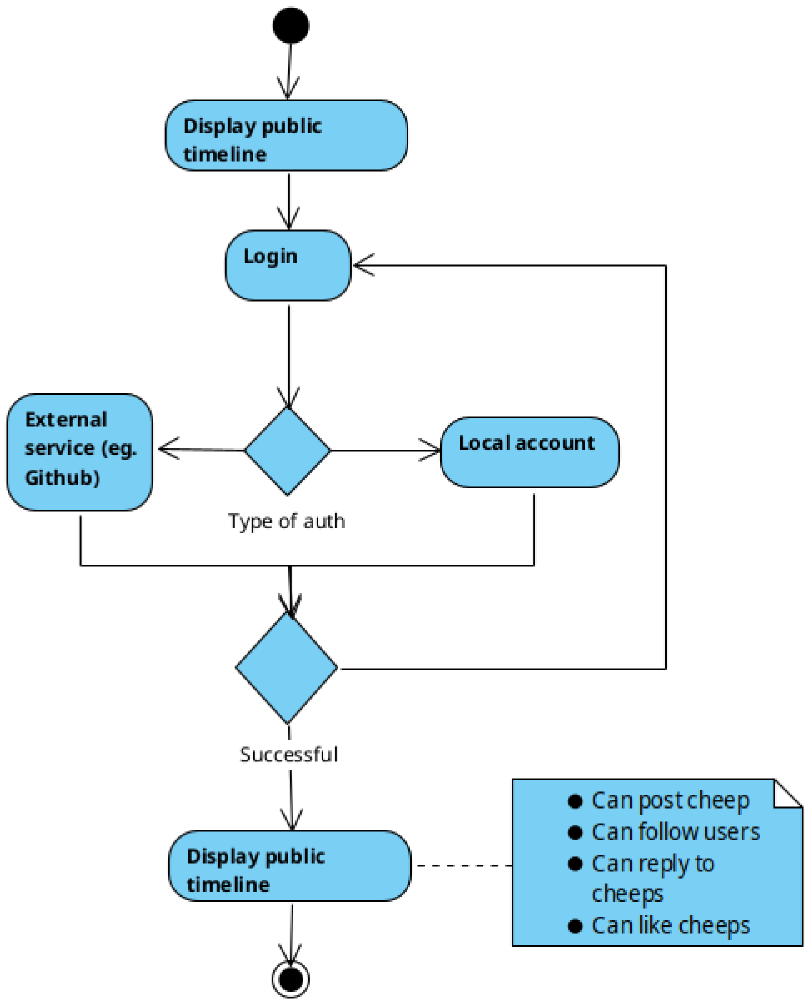


Figure 7: Activity diagram of a user trying to login to the *Chirp!* application

1.5.5 Reply

When designing replies it was chosen to use the same `Cheep` model as both a "root post" and the following replies to said post. This method was chosen because we wished to design a *thread* style of replies, similar to Reddit. Instead of only having one layer of replies, users could now reply to other peoples' replies, and continue a *thread* of replies. Below is a diagram of a typical scenario of a user replying to another user in the *Chirp!* application.

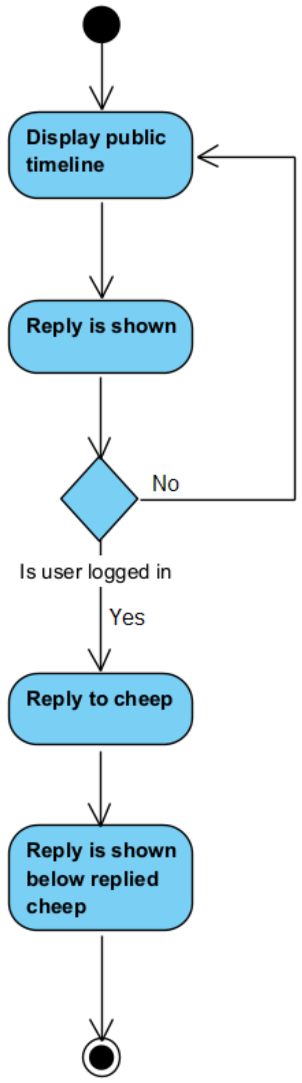


Figure 8: Activity diagram of a user replying to another users `Cheep`

1.6 Sequence of Functionality/Calls through *Chirp!*

Below is a UML sequence diagram illustrating the sequence of calls in the *Chirp!* application when it receives an HTTP `GET /` request from a client. Such a request corresponds to a request for the *Public Timeline*. More specifically, the diagram focuses on the calls relevant to retrieving and displaying the *Public Timeline* `Cheeps`. The diagram is intentionally kept at an architectural level of abstraction. The intention is to depict how a user's incoming request is handled end-to-end, from the incoming HTTP request to data retrieval and the final rendering of the *Public Timeline* with the relevant `Cheeps`. It therefore emphasises the collaboration between the main system components (`Chirp.Web`, `Chirp.Infrastructure`, and the database), rather than the internal implementation details. As a result, certain methods (e.g. `Public.OnGet()`) have been deliberately omitted. This abstraction is chosen to keep the diagram readable while still conveying the essential technical flow through the application's layered architecture.

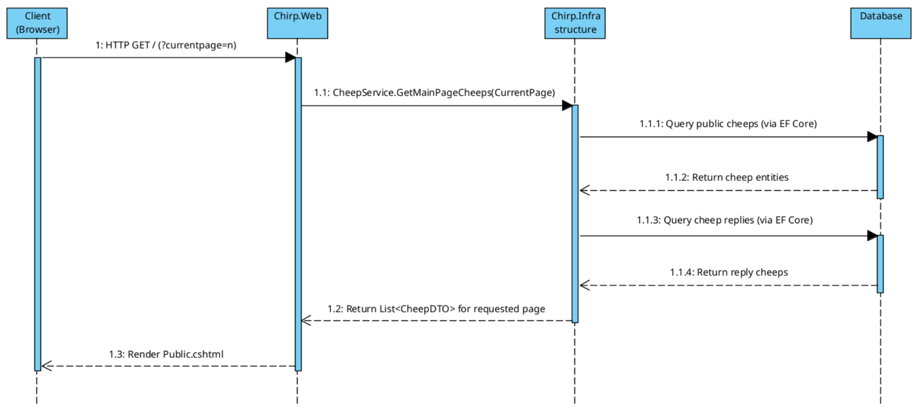

Client (Browser)
Chirp.Web
Chirp.Infra structure
Database
1: HTTP GET / (?currentpage=n)
1.1: CheepService.GetMainPageCheeps(CurrentPage)
1.1.1: Query public cheeps (via EF Core)
1.1.2: Return cheep entities
1.1.3: Query cheep replies (via EF Core)
1.1.4: Return reply cheeps
1.2: Return List<CheepDTO> for requested page
1.3: Render Public.cshtml

Figure 9: Sequence diagram depicting the sequence of calls through the *Chirp!* application when it receives an HTTP GET request

2 Process

2.1 Build, Test, Release, and Deployment

2.1.1 Building & Releasing

The script `release.yml` was used for building and releasing the project. The script uses a matrix of `[linux-x64, linux-arm64, win-x64, win-arm64, osx-x64, osx-arm64]` to create release artifacts on each release. When releasing a new version release please was used to generate commit changelogs.

2.1.2 Testing

The script `coverage.yml` was used for checking if test suites existed for each package in the solution, after testing the script uses reportgenerator to generate a coverage report which could be used to analyze the test coverage and quality of tests.

2.1.3 Versioning

Before the lecturers introduced us to semantic versioning and told us it was a requirement, we used CalVer[2]. CalVer was initially chosen, as it uses the calendar date for versioning, and seemed to be a good way to coordinate our weekly releases.[3]

Once we switched to semantic versioning we decided that it would make sense to automate this process. The tool we used to automate this was Release Please from Google. *Release Please* continuously monitors the git history of a project through a GitHub action. The action identifies commits which use the conventional commit standard and generates a changelog based on changes in its own branch. The action also opens a pull request, which when merged, merges the changelog into main and creates a new release with the changelog added as a description. This helped give us, and potential users, an overview over what has changed between releases.

In addition to this, *Release Please* also automatically computes the next version number based on the `feat`, `fix`, and `feat!` tags from conventional commit. This was a good way to automate versioning, since we did not have to consider what our next release number should be.

One issue we faced with this was, that we ended up with a rather high major version (*v5.5.1*). The reason for this was our failure to consider what was actually a breaking change. We followed the convention of tagging any breaking API change as a breaking change, which would make Release Please update the major version[4]. However, these breaking API changes were often only breaking for internal APIs, for many major releases, no user-facing APIs changed. We should not have considered these internal API changes as breaking, since for the end user, these changes were not breaking. What we should have considered a breaking change should be the switch from a CLI to a web page, and potentially the addition of identity. This would mean that *Chirp!* would be on *v3.x.y* or *v2.x.y*, depending on whether the addition of identity was considered breaking, not *v5.5.1*.

[2]CalVer
[3]The initial release tag was deleted and tagged again using semver
[4]Lecture slides on Semantic Versioning

2.1.4 Deployment

Whenever we deploy code to GitHub, a number of GitHub Actions scripts were run. These can be found in the `.github/workflow directory`.

- `coverage.yml`: Runs a code coverage test and fails upon not reaching the coverage threshold
- `format.yml`: Runs `.NET` format, this maintains `.NET`'s official styleguide. These formatting commits have been attributed to our group member mnni@itu.dk
- `main_bdsagroup13chirprazor.yml`: Builds and deploys our code to our Azure Webapp instance, using our GitHub Secrets to access login information. This file was auto-generated by Azure and afterwards customised for our needs.
- `release.yml`: Builds and publishes our project to the GitHub repository, with builds for different operating systems.

2.1.5 Linear Git History

Initially, we did not enforce a linear git history as a requirement for our project, but it was a self-imposed requirement to attempt to keep the history linear. However, after #87381, we decided to enforce a linear history on the project through the git repo settings.

There are many reasons for enforcing a linear history. For one, having a linear history makes managing and maintaining a project easier. It is much easier to understand when a certain feature was added, as each commit is a patch, which cleanly applies atop the previous commit.

It also allows for external tools, such as `git bisect` to traverse the commit graph easier. Since a linear history is just a line, it is much easier for `git bisect` to perform a binary search and find when a regression was introduced. On the other hand, when the commit graph is a large interleaving of commits and merge commits, it can be difficult to perform a binary search. This does not mean it is impossible to perform a binary search on a non-linear history, but the cognitive load on the developer is increased.

Finally, one of the primary reason we used a linear git history was for external tooling. As discussed earlier, we used an action called release-please for automated changelog generation. This tool recommended a linear history to ease parsing of the commit graph[5].

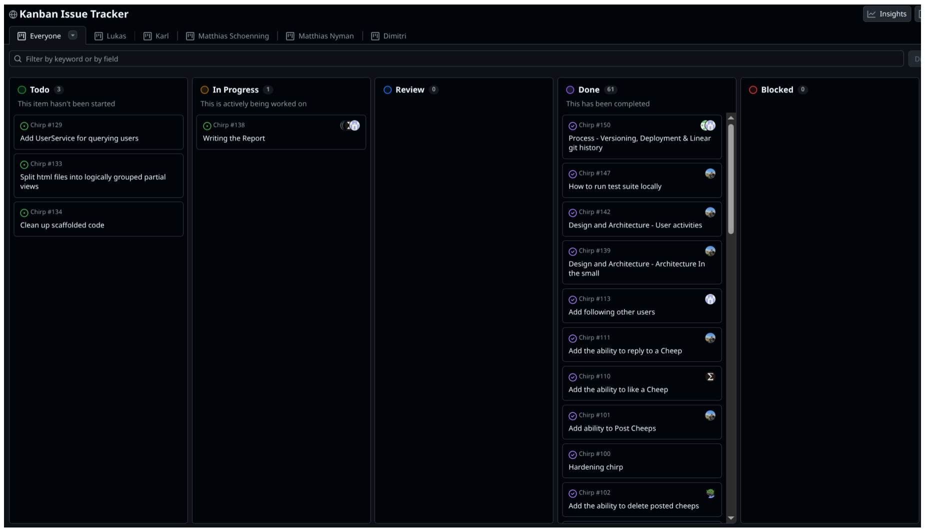
The workflow for a linear git history is rather simple. When a commit is ready to be merged with the main branch, you run `git rebase origin/main` which changes the base of the branch to be the current `HEAD` of the main branch. This might introduce merge conflicts, which you resolve as normal. Once the conflicts

[5]Release Please documentation about linear history

have been resolved, and the PR has been approved, it can be merged with `git merge --ff-only`.

Having a linear git history is one way to manage a project which comes with its set of benefits. However, having a non-linear history has its own set of benefits. Chiefly, some metadata is lost when a branch is rebased, since the commit that the branch was initially based on has changed. In addition to this, a non-linear history can make the history of long-lived feature branches more clear. However, since we used trunk-based development this was not a concern for us.

2.2 Teamwork



Todo 3
This item hasn't been started

Chirp #129
Add UserService for querying users

Chirp #133
Split html files into logically grouped partial views

Chirp #134
Clean up scaffolded code

In Progress 1
This is actively being worked on

Chirp #138
Writing the Report

Review 0

Done 61
This has been completed

Chirp #150
Process - Versioning, Deployment & Linear git history

Chirp #147
How to run test suite locally

Chirp #142
Design and Architecture - User activities

Chirp #139
Design and Architecture - Architecture In the small

Chirp #113
Add following other users

Chirp #111
Add the ability to reply to a Cheep

Chirp #110
Add the ability to like a Cheep

Chirp #101
Add ability to Post Cheeps

Chirp #100
Hardening chirp

Chirp #102
Add the ability to delete posted cheeps

Blocked 0

Figure 10: Final state of kanban board

The Kanban board of our project immediately before finishing the project had 4 issues open. The one issue, which was in the *In Progress* tab, was a meta issue for writing the report.

The other 3 issues which were still in the *Todo* section were extra features and maintenance tasks which we discussed adding in order to improve the maintainability of the project. We didn't handle these additional todos, as the scope of the project did not require that these changes were present. However, if we were to continue working on the project, these would be high on our list of priorities, as they would greatly improve the maintainability of the project.

The groups workflow, when we received the tasks for the week was as follows:

1. We received the weeks tasks, and discuss them
2. We created issues and user stories for them
    - These issues also included acceptance criteria and test criteria
3. The issues were assigned to the person willing to work on them
    - If we hadn't finished a task from last week, these were prioritized above new tasks
4. A pull request for an implementation of the task is created, and the issue is automatically moved to the "Review" tab of the Kanban board
5. The PR is reviewed
    - If it is approved, it was merged to main, and the associated issue was closed
    - If further iterations were required, the associated issue was moved back to the *In Progress* tab of the Kanban board

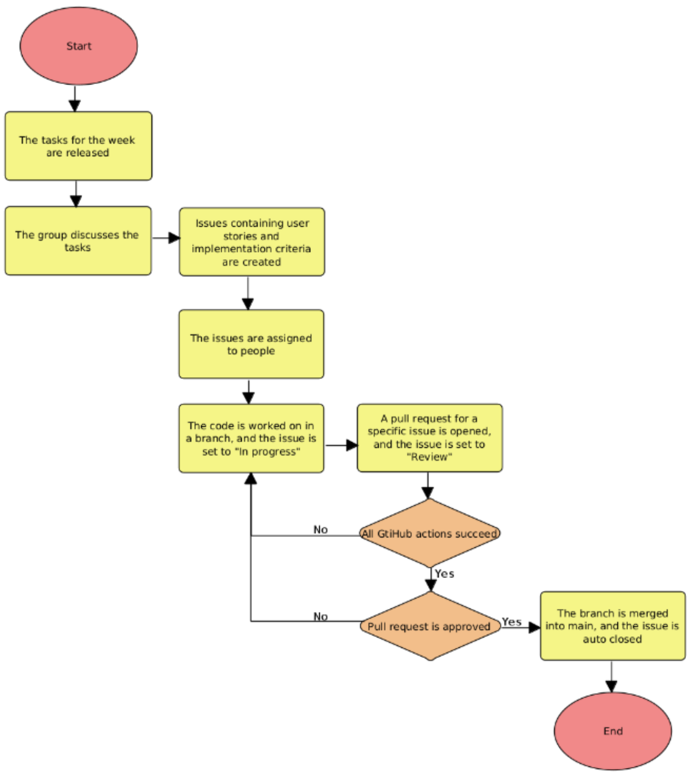

Start
The tasks for the week are released
The group discusses the tasks
Issues containing user stories and implementation criteria are created
The issues are assigned to people
The code is worked on in a branch, and the issue is set to "In progress"
A pull request for a specific issue is opened, and the issue is set to "Review"
All GtiHub actions succeed
No
Yes
Pull request is approved
No
Yes
The branch is merged into main, and the issue is auto closed
End

Figure 11: Flowchart of a task from when it is released till it is merged to main

2.2.1 Trunk-based Development

During the development of this project, we were encouraged to use trunk-based development. Trunk-based development is a source-control strategy centered on frequent integration of small batches of work into the main branch (trunk), rather than long-lived feature branches, as is common in many other popular workflows, e.g., GitFlow. Throughout the development process, we didn't completely avoid long-lived branches. Many times, it proved impractical to adhere to the "merge every day" mantra, which was (admittedly) partly due to our inexperience with the workflow. Being able to merge every day depends on the assumption that the given task can be completed and reviewed within that timeframe. However, estimating the time needed to complete tasks was not always straightforward, especially for larger tasks that didn't naturally split into logically independent subtasks. An example of such a task is the large rewrite required to migrate from raw SQLite to EF Core. Merging at any point between starting and finishing this task would result in a broken main branch. Of course, this is an extreme example, and often it would also be due to interpersonal dynamics within the team. Such as, waiting for code review, from team members with their own daily lives, schedules, and responsibilities. In short, we opted to let the branches live a bit longer rather than merge unfinished or unreviewed code, as we valued high-quality code over minimizing branch staleness. However, we did avoid intentionally long-lived branches like the `develop` and `staging` branches found in other source-control strategies, and branches were generally merged quite quickly.

2.3 How to Make *Chirp!* Work Locally

The get the application running locally either clone this repository or download the latest release here. While *Chirp!* will run without a GitHub OAuth client, the application will have degraded functionality if you do not have one. To create a GitHub OAuth client follow these instructions[6].

2.3.1 Running from Latest Release

On Windows:

1. Unzip folder
2. Navigate to `chirp-main-<OS>-<architecture>`
3. Run the `Chirp.Web.exe` file.

On Linux & macOS

1. Unzip folder
2. Navigate to `chirp-main-<OS>-<architecture>`
3. Run `./Chirp.Web`

[6]Release artifacts run on port :5000, not :5273

2.3.2 Running from Repository

1. From the root folder of the project:

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

2.3.3 Github OAuth (Optional)

Neither release artifacts nor the github repository contain GitHub OAuth ClientID or ClientSecret. These however these can be read from the environment variables `$authentication__github__clientSecret` and `$authentication__github__clientId` Do note, that releases between 5.3.0 and 5.5.0 inclusive do not work without secrets in the environment, as GitHub OAuth was not optional. This was fixed in 5.5.1.

2.4 How to Run Test Suite Locally

All tests, including Playwright, E2E, Integration and Unit tests are stored in the `test` directory. Playwright needs to be downloaded and installed first. Following is the steps to build and run the test suite (all done from the root folder of the project):

1. Build the project (needed for downloading Playwright)

   ```
   dotnet build
   ```

2. Install Playwright for tests

   ```
   pwsh test/Chirp.E2E.Tests/bin/Debug/net8.0/playwright.ps1
   install --with-deps
   ```

3. Run the project tests

   ```
   dotnet test
   ```

3 Ethics

3.1 License

We chose the 3-Clause BSD License, which is a permissive, OSI approved[7], open source copyright license. This license is slightly more restrictive than the MIT License or the 2-Clause BSD License. The license includes a non-endorsement clause stating that any derivative work may not use the name of the original work, or its authors as an endorsement of the derivative.

We felt that this license was a good choice for an educational project, as it preserves the permissive nature of the MIT License or the 2-Clause BSD License. This allows for further contributions to the project through a fork, while protecting the original authors and project from both any implications of warranty or liability.

3.2 LLMs, ChatGPT, CoPilot, and Others

During the development of this project, LLMs were primarily used for research prior to (or while) implementing a feature. Prior to adding a feature, we would conduct research and find all of the necessary libraries required. Our group found the documentation, especially from Microsoft themselves, to be lacking. This proved a great use for LLMs, primarily ChatGPT, Claude, or Gemini, to be used in the context of clarifying a subject from various internet sources. Once a feature had been thoroughly researched, an attempt would be made at implementing the feature in question. An LLM might assist in the theoretical structure of how this feature would be implemented. This has often proven to work well, as LLMs are good at reading large amounts of documentation, extracting the important parts, and relaying them to the reader. We did not consider this as co-authorship, since the actual code from an LLM was not used.

Whenever we did co-author an LLM, authorship was was attributed to the LLM eg. with a `Co-authored-by: ChatGPT <chatgpt@openai.com>` footer in the git commit. This was done when code generated by an LLM was used verbatim. This had some success, however, often the code was not quite right, or simply did not work, which can be seen in #d2a073e, #899ca30, and #270892b where 3 LLM generated workflows were reverted as they did not work correctly.

Another issue we faced with LLMs was that when researching topics, they would often get confused and produce increasingly inconsistent or incorrect responses. This was likely because the context window of the conversation became too large, which caused the LLM to hallucinate.

[7] OSI approved licenses