Chirp! Project Report

ITU BDSA 2025 Group 15

Frederik Haraszuk Stein frhs@itu.dk
Hjalte Krohn Frandsen hjfr@itu.dk
Kenny Izaak Egelund keeg@itu.dk
Lucas Tambour Fabricius lufa@itu.dk
Omar Osama Al Hassan omal@itu.dk
Tobias Mondrup Holm tmho@itu.dk

Contents

1 Design and Architecture of *Chirp!*

1.1 Domain model

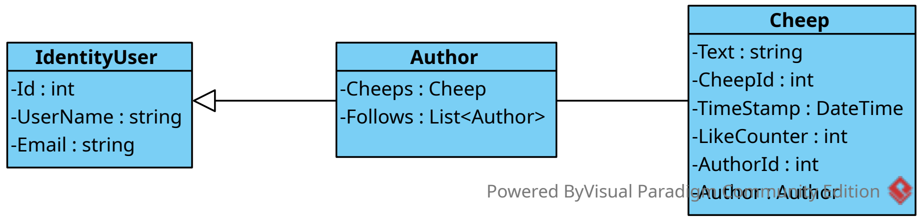


Figure 1: The domain model for our Chirp! project.

Here is the domain model for our Chirp! project. The model extends the class IdentityUser for the Author, which allows us to use ASP.NET Identity for handling user authentication. We have also added Follows to track who an author is following, and LikeCounter to track likes on a cheep.

1.2 Architecture — In the small

The figure above shows an illustration of how we implemented the Onion Architecture to our web application Chirp!. This was done with the intention of keeping easy maintainability and testability throughout the project.

Low-level modules like our Chirp.Core, doesn't depend on high-level modules like Chirp.Web. High-level modules depend on interfaces ensuring the Dependency Inversion Principle is kept. As a developer, this ensures changes within the application can be adjusted easily, so modules don't interfere with the logic and functionality across the whole program. As such our application has been split up in different layers of dependency.

The Domain layer: The center, Chirp.Core, consists of our domain model and DTO's which are fully independent.

A second and third layer, that is made of the Infrastructure.Chirp. Chirp Repository in the second layer, which access and manages the data. And Chirp Services in the third layer, that serve as a flow of data between the repositories and the outer layer.

And finally the most outer layer, Chirp.Web, that handles User interface, User interactions, page rendering, and testing, which is fully dependent across all the inner layers.

1.3 Architecture of deployed application

This is the architecture of the deployed Chirp! application. The server is deployed to Azure App Service, where the Chirp! software stack (structured after the

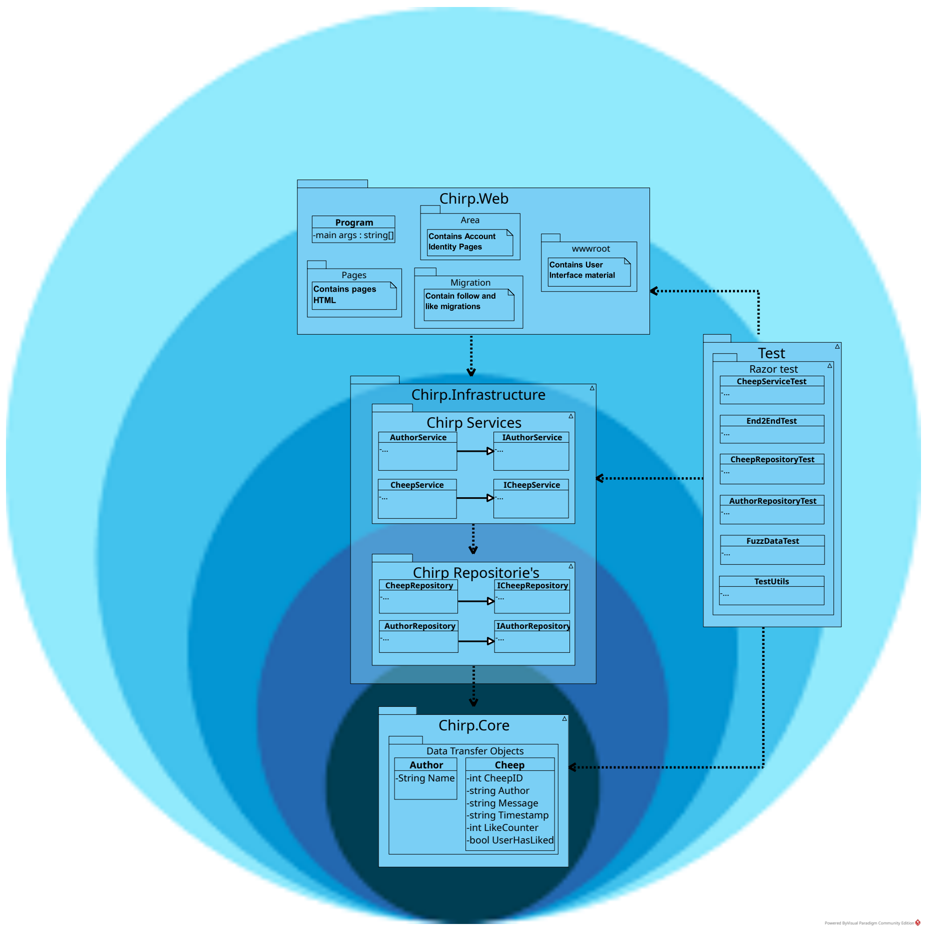


Figure 2: Illustration of the *Chirp!* onion architecture

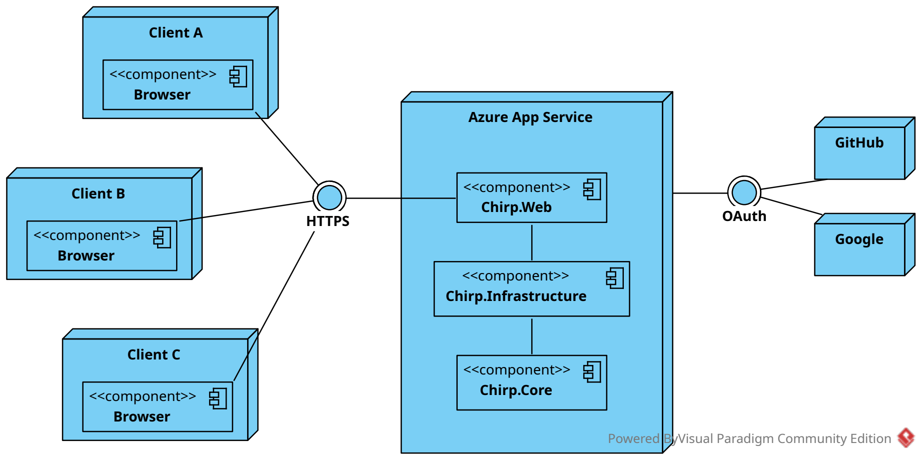


Figure 3: Illustration of the *Chirp!* deployment architecture

onion architecture) is running.

Clients connect to the Chirp.Web service via HTTPS through their browser. The Chirp.Web service then serves a rendered page via Razor Pages.

To support external logins in Chirp!, we also maintain OAuth connections to the login providers. For our project, we have added both GitHub and Google as OAuth login providers.

1.4 User activities

Here is an activity diagram. This diagram shows the difference in what an authorized and unauthorized user can in the application.

1.5 Sequence of functionality/calls through *Chirp!*

Here is the call sequence for when a non-authenticated user accesses the front page of Chirp!.

The web layer processes the request in the OnGet() method, which first checks whether the user is logged in. This is used to display the correct header.

Then we call LoadCheeps(), a helper method which queries the service layer and retrieves a list of CheepDTOs. The service calls the repository, which in turn calls the SQLite database using EFCore.

Finally, the page is rendered in Razor Pages and sent to the user.

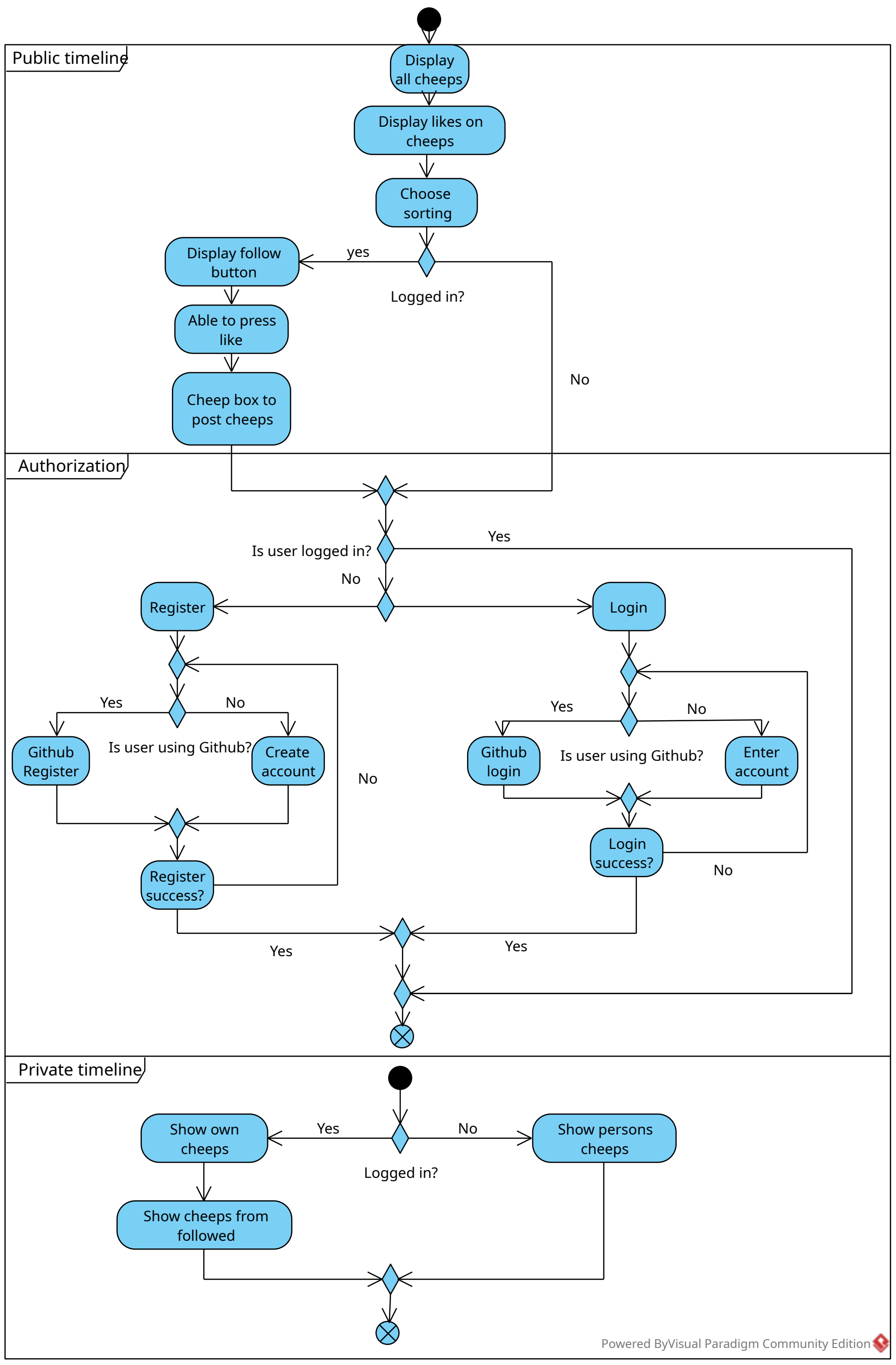


Figure 4: Illustration of User activities

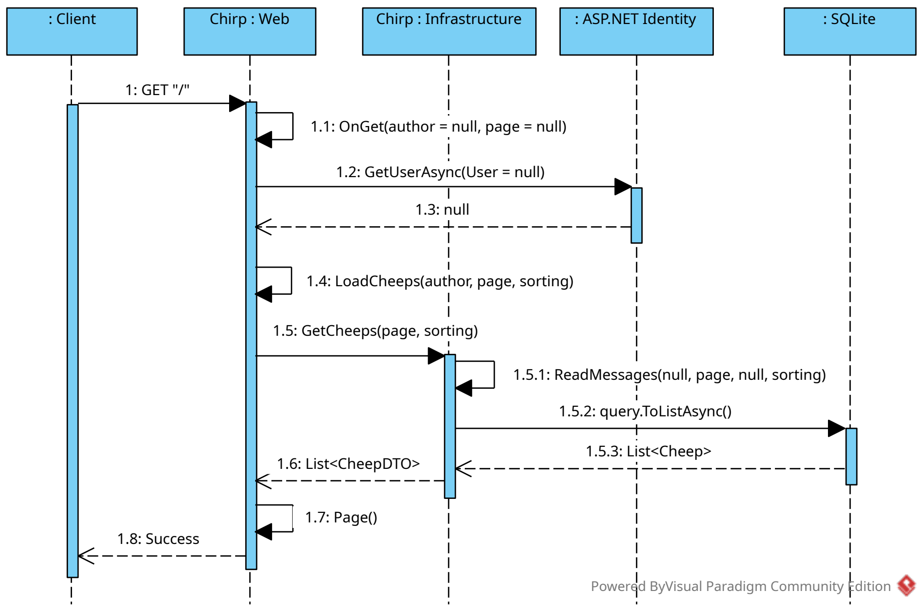

: Client
Chirp : Web
Chirp : Infrastructure
: ASP.NET Identity
: SQLite
1: GET "/"
1.1: OnGet(author = null, page = null)
1.2: GetUserAsync(User = null)
1.3: null
1.4: LoadCheeps(author, page, sorting)
1.5: GetCheeps(page, sorting)
1.5.1: ReadMessages(null, page, null, sorting)
1.5.2: query.ToListAsync()
1.5.3: List<Cheep>
1.6: List<CheepDTO>
1.7: Page()
1.8: Success
Powered ByVisual Paradigm Community Edition

Figure 5: Illustration of call sequence when new user accesses the Chirp! web page.

2 Process

2.1 Build, test, release, and deployment

Chirp! is built, tested, released, and deployed using automated GitHub Actions workflows. We have three workflows relevant for the deployment of the Chirp! application:

- Push/PR update: Builds and tests the project, ensuring all tests pass. If anything fails, the PR is blocked.
- Push/PR merge to main: Creates a Chirp! ZIP with release configuration, which is then deployed on the Azure web app.
- Push to tag v*.*.*: Creates a ZIP and adds it to release with the pushed tag. Note: this workflow runs as a matrix, creating ZIPs for Windows, macOS and Linux.

The GitHub repository also has a Discord notification workflow that pings a Discord channel every time a new pull request was made. This is because Discord was our main platform of communication and the easiest way to get a hold of us.

2.2 Team work

All required features for the Chirp! project should be in place, with most features having been covered by either a unit test, integration test or end-to-end test.

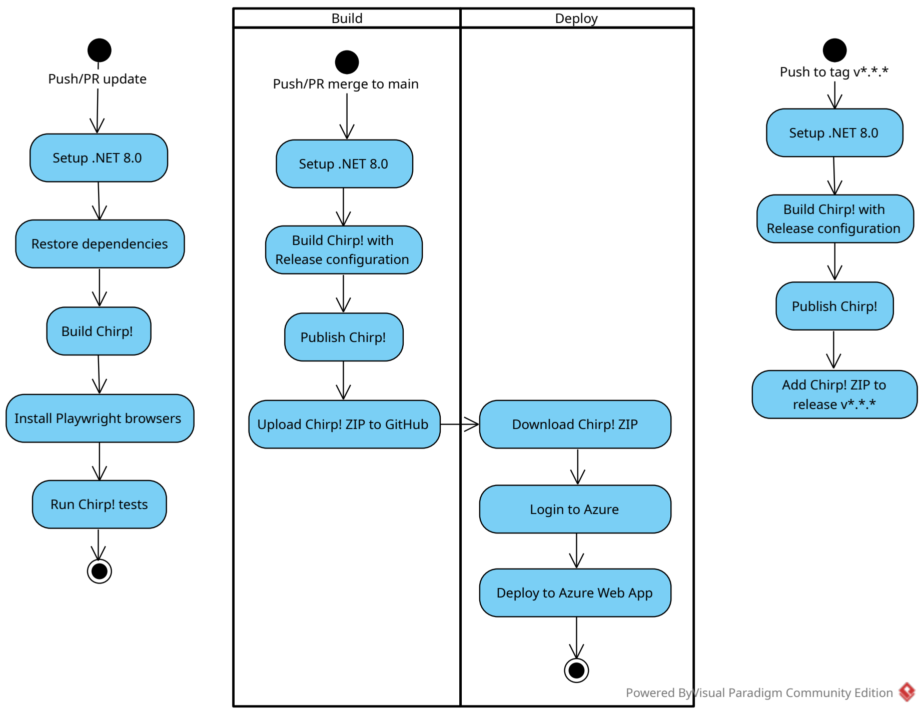
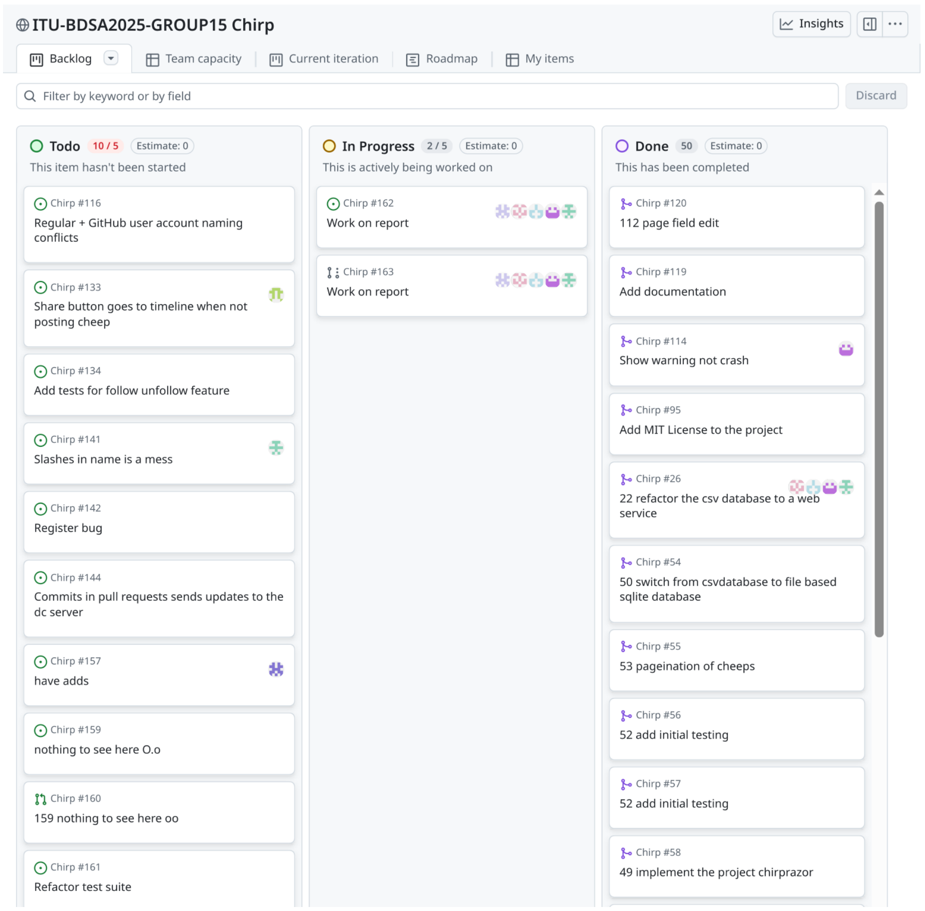


Figure 6: Activity diagram showing deployments via GitHub Actions workflows.



ITU-BDSA2025-GROUP15 Chirp

Insights

Backlog | Team capacity | Current iteration | Roadmap | My items

Filter by keyword or by field

Discard

Todo 10 / 5 Estimate: 0

This item hasn't been started

- Chirp #116 — Regular + GitHub user account naming conflicts
- Chirp #133 — Share button goes to timeline when not posting cheep
- Chirp #134 — Add tests for follow unfollow feature
- Chirp #141 — Slashes in name is a mess
- Chirp #142 — Register bug
- Chirp #144 — Commits in pull requests sends updates to the dc server
- Chirp #157 — have adds
- Chirp #159 — nothing to see here O.o
- Chirp #160 — 159 nothing to see here oo
- Chirp #161 — Refactor test suite

In Progress 2 / 5 Estimate: 0

This is actively being worked on

- Chirp #162 — Work on report
- Chirp #163 — Work on report

Done 50 Estimate: 0

This has been completed

- Chirp #120 — 112 page field edit
- Chirp #119 — Add documentation
- Chirp #114 — Show warning not crash
- Chirp #95 — Add MIT License to the project
- Chirp #26 — 22 refactor the csv database to a web service
- Chirp #54 — 50 switch from csvdatabase to file based sqlite database
- Chirp #55 — 53 pageination of cheeps
- Chirp #56 — 52 add initial testing
- Chirp #57 — 52 add initial testing
- Chirp #58 — 49 implement the project chirprazor

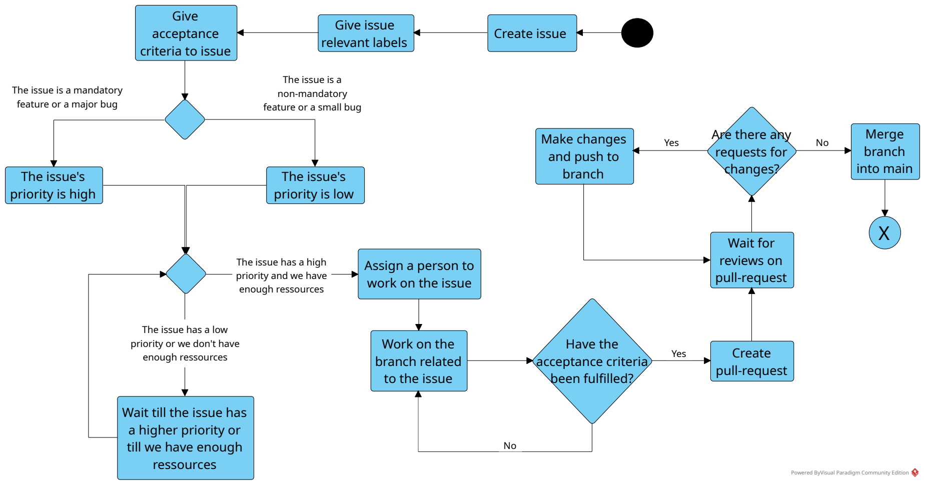
Figure 7: The final GitHub project board for our Chirp! project.

However, there are still some bug fixes, edge cases and other improvements to be made.

One thing that we didn't implement is checking for duplicate names when someone signs up through GitHub integration. Imagine that a user has the username "Bob". If another user signs up through the GitHub integration with a GitHub username that is also "Bob", the backend does not check if this username is already taken. This could result in a crash.

Here are some other bugs/features we didn't end up fixing/implementing:

- Not clicking email confirmation breaks the user.
- More unit test to make sure our distribution of test are like a pyramid.
- Mock services for tests.
- More expansive fuzz testing.
- Testing follow functionality



Create issue
Give issue relevant labels
Give acceptance criteria to issue
The issue is a mandatory feature or a major bug
The issue is a non-mandatory feature or a small bug
The issue's priority is high
The issue's priority is low
The issue has a high priority and we have enough ressources
Assign a person to work on the issue
The issue has a low priority or we don't have enough ressources
Wait till the issue has a higher priority or till we have enough ressources
Work on the branch related to the issue
Have the acceptance criteria been fulfilled?
No
Yes
Create pull-request
Wait for reviews on pull-request
Are there any requests for changes?
Yes
Make changes and push to branch
No
Merge branch into main
X
Powered By Visual Paradigm Community Edition

Figure 8: Flowchart of issues from creation to merge

2.2.0.1 From issue to merge This flowchart shows the process from the creation of an issue to when the related branch is merged into the main branch. When a new issue is created, our GitHub workflow automatically sets its status to "To-do" in our project board. Then when that branch is merged into main, the workflow then sets the status of the related issue to "Done".

2.3 How to make *Chirp!* work locally

2.3.1 Before you run Chirp!

The Chirp! application supports environment variables to customize your setup. Be sure to register any needed environment variables before you run the server.

2.3.1.1 Setting up GitHub authentication

Chirp! supports GitHub integration via OAuth. Without this setup, the feature to log in with GitHub does not work when running Chirp! locally. To enable this feature, you must first register an OAuth application on GitHub.

For registering the OAuth app (assuming you are hosting on `http://localhost:5000`), use the following values:

- Homepage URL: `http://localhost:5000`
- Authorization callback URL: `http://localhost:5000/signin-github`

Then you can get a client ID + client secret, which you must register in the following environment variables:

```
authentication:github:clientId=<Your Client ID>
authentication:github:clientSecret=<Your Client Secret>
```

Alternatively, if you have downloaded the Chirp! source code, you can use `dotnet user-secrets` to register them instead:

```
dotnet user-secrets set "authentication:github:clientId" <Your Client ID>
dotnet user-secrets set "authentication:github:clientSecret" <Your Client Secret>
```

Your Chirp! instance will now use your GitHub OAuth application to allow GitHub logins on the site.

2.3.1.2 Setting up Google authentication

Chirp! also supports Google authentication via OAuth. To enable this feature, register an OAuth application on Google Cloud Console.

For registering the OAuth app (assuming you are hosting on `http://localhost:5000`), add the following authorized redirect URI:

- `http://localhost:5000/signin-google`

Then you can get a client ID + client secret, which you must register in the following environment variables:

```
authentication:google:clientId=<Your Client ID>
authentication:google:clientSecret=<Your Client Secret>
```

Alternatively, if you have downloaded the Chirp! source code, you can use `dotnet user-secrets` to register them instead:

```
dotnet user-secrets set "authentication:google:clientId" <Your Client ID>
dotnet user-secrets set "authentication:google:clientSecret" <Your Client Secret>
```

Your Chirp! instance will now use your Google OAuth application to allow Google logins on the site.

2.3.1.3 Setting a custom database location

By default, Chirp! will store the database in your temporary directory as `chirp.db`.

You can specify a custom location for the database using the environment variable `CHIRPDBPATH`. For example:

```
CHIRPDBPATH=/home/user/my-database-folder/chirp.db
```

2.3.2 Running Chirp! from release builds

Chirp! is released for 64-bit versions of Windows, Mac and Linux. Releases can be downloaded here.

2.3.2.1 How to run

Depending on your operating system, the executable will have a different name (e.g. `Chirp.Web.exe` on Windows, `Chirp.Web` on Mac/Linux). In your terminal of choice, run the following:

```
./Chirp.Web
```

The Chirp! server will start. The server is ready when you see output similar to this:

```
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Production
info: Microsoft.Hosting.Lifetime[0]
      Content root path: /Chirp
```

You can now access the Chirp! service in your browser on the following link: http://localhost:5000

2.3.3 Running Chirp! from source code

2.3.3.1 Clone source code

1. Make sure git is installed. Installation guide.
2. To clone the source code go to your terminal and type: `git clone https://github.com/ITU-BDSA2025-GROUP15/Chirp.git`
3. You should see an output similar to this:

```
Cloning into 'Chirp'...
remote: Enumerating objects: 2750, done.
remote: Counting objects: 100% (490/490), done.
remote: Compressing objects: 100% (313/313), done.
remote: Total 2750 (delta 293), reused 196 (delta 175), pack-reused 2260 (from 3)
Receiving objects: 100% (2750/2750), 110.96 MiB | 21.22 MiB/s, done.
Resolving deltas: 100% (1651/1651), done.
```

4. The folder should now be cloned into a new folder called `Chirp`.

2.3.3.2 How to build Chirp!

1. Make sure you have installed .NET version 8.0. You can get it here.
2. Open your terminal and navigate to the root folder of the project.
3. Run the following command: `dotnet build ./src/Chirp.Web/Chirp.Web.csproj`
4. You should now successfully have built the project.

2.3.3.3 How to run Chirp!

1. Begin by navigating to the Chirp.Web folder found under src in the terminal.
2. Now run the command `dotnet run`. This will start the server.
3. In your Web browser open `localhost:5273`. This will also appear in the terminal.
4. You have now opened Chirp!

2.4 How to run test suite locally

1. First make sure the project is build by navigating to the root folder of Chirp and run `dotnet build`
2. Make sure PowerShell is installed. PowerShell installation guide.
3. Install playwright into the Razor.Tests project folder.

- Navigate to the test project in your terminal and run the following: `pwsh bin/Debug/net8.0/playwright.ps1 install --with-deps`
- For further issues with Playwright, consult Playwright.

4. Now the tests can be run with the command `dotnet test` (This may take a moment).

There are three different kinds of test: unit test, integration tests and end-to-end tests. The unit tests are testing the individual methods to see if they work. The integration tests makes sure methods work together correctly. The many end-to-end tests ensure that the entire program is running correctly and therefore also handles security tests like SQL injection, XSS, and CSRF to ensure there are no security vulnerabilities.

3 Ethics

3.1 License

For the applications license we chose the MIT license because of its simplicity and non-restrictive nature leading to more collaboration. The MIT license is also compatible with most other licenses. To see the full license, see our GitHub repository.

3.2 LLMs, ChatGPT, Copilot, and others

During the group's work on the application LLMs were used to a limited degree. This can be seen in the contributors tab under insights where both Copilot and CodeFactor have been contributing to a total of 25 commits. ChatGPT also has some attributions to commits, but they do not show up in GitHub insights because ChatGPT does not have a GitHub account.

CodeFactor, unlike ChatGPT and Copilot, is not an LLM, but a bot that does static analysis based on rules and therefore consumes much less power than an LLM. The energy and water consumption of an LLM is hard to figure out, due to the companies not releasing official numbers. The amount is likely high and is the reason the group only used the LLMs to a limited degree. CodeFactor caught many common mistakes like too many newlines or other redundancies and did improve overall code quality.

Whether the LLMs sped up or slowed down our process depended on the response gotten from the LLMs. Issue #123 where Copilot reduced the size of our release files without us having to research how one might do so, is an example of LLMs speeding up our process. There were also instances where the LLMs slowed down our process by leading us astray or coming up with incorrect solutions.