Chirp! Project Report

ITU BDSA 2025 Group 27

Lukas Schultz Stryg luss@itu.dk

1 Design and Architecture of *Chirp!*

1.1 Domain model

The domain model consists of three main entities: Author, Cheep, and Hashtag.

The Author class inherits from IdentityUser for authentication and contains AuthorId, UserName, Email, and Id.

The Cheep class contains CheepId, Text (with a 160-character constraint), TimeStamp, and AuthorId.

The Hashtag class contains HashtagId and TagName (with a 50-character constraint, unique).

A Cheep can have multiple Hashtags, and a Hashtag can be associated with multiple Cheeps. Authors can follow multiple other Authors.

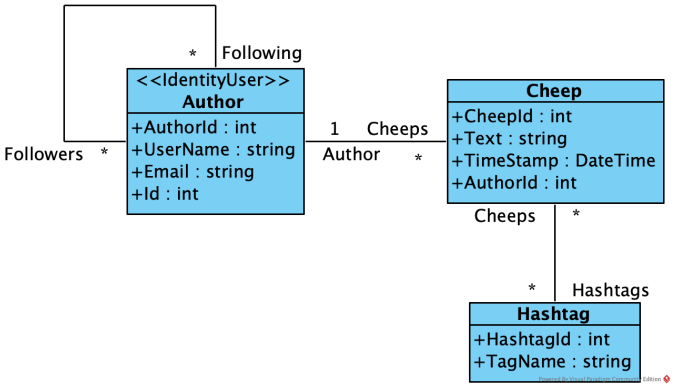

Figure 1: Illustration of the *Chirp!* data model as UML class diagram.

1.2 Architecture — In the small

The application follows the onion architecture pattern, where each layer depends only on inner layers.

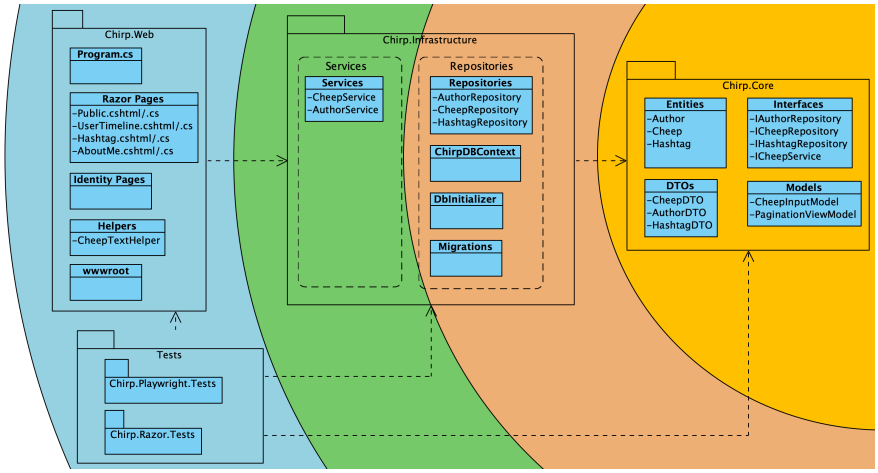

Figure 2: Illustration of the *Chirp!* architecture as onion architecture diagram.

1.2.1 Chirp.Core

Chirp.Core is the innermost layer and contains the domain entities (Author, Cheep, Hashtag) and Data Transfer Objects (DTOs). It defines the interfaces for both services (ICheepService, IAuthorService), and repositories (ICheepRepository, IAuthorRepository, IHashtagRepository). This layer has no external dependencies.

1.2.2 Chirp.Infrastructure

Chirp.Infrastructure depends on Chirp.Core and contains implementations of its interfaces. It consists of two layers:

- **Services:** Contains service implementations (AuthorService, CheepService) that orchestrate business logic between the presentation and repository layers.
- **Repositories:** Contains repository implementations (AuthorRepository, CheepRepository, HashtagRepository), the database context (ChirpDBContext), and migrations.

1.2.3 Chirp.Web

Chirp.Web is the outermost layer. It contains the Razor Pages that render the UI and their corresponding page models. It depends on Chirp.Infrastructure.

1.3 Architecture of deployed application

The application follows a client-server architecture. Users access it through a web browser that communicates with the server over HTTPS.

The server is deployed to Azure App Service. It uses a SQLite database and supports user authentication through both email+password and GitHub OAuth.

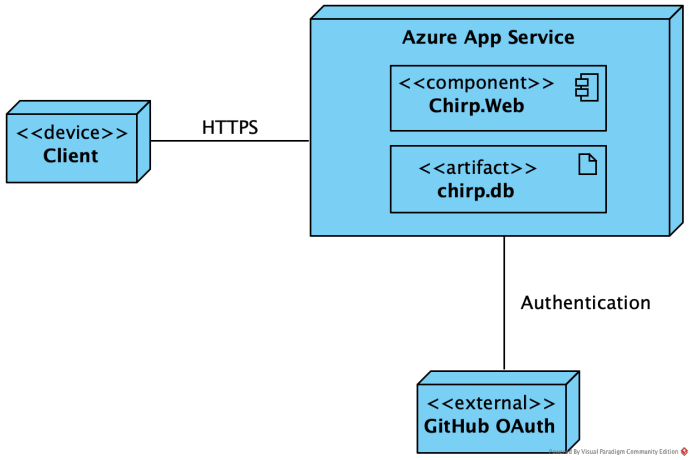

Figure 3: Illustration of the *Chirp!* deployment architecture as deployment diagram.

1.4 User activities

1.4.1 Unauthenticated users

- View the Public Timeline with all cheeps
- View author Private Timelines by clicking on author names
- Click hashtags to view all cheeps with that hashtag

To access the rest of the application, they must either log in with an existing user or register a new user.

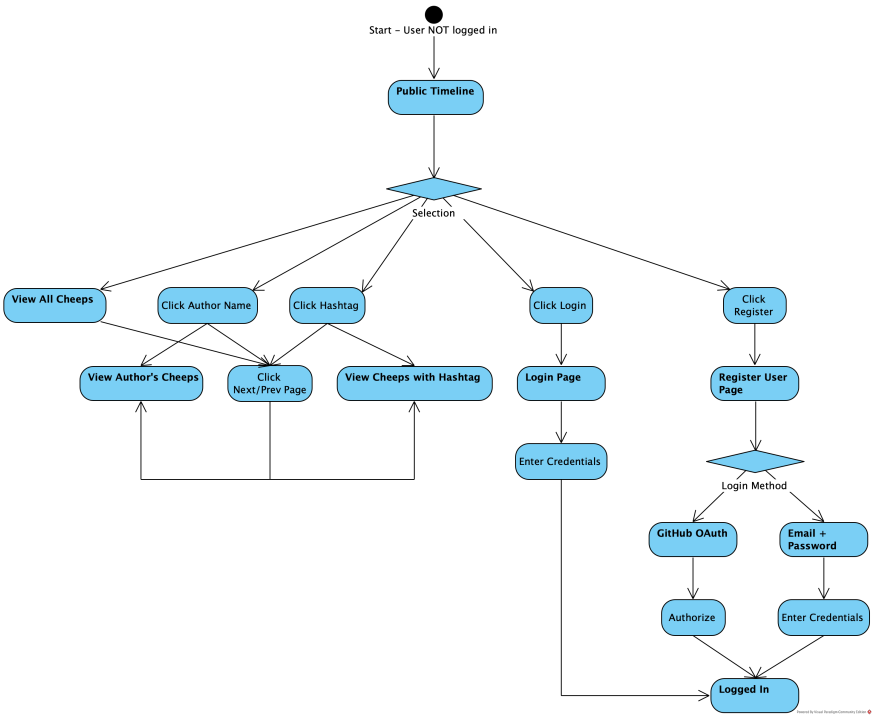

Figure 4: Illustration of unauthenticated user activities as UML activity diagram.

1.4.2 Authenticated users

- View the Public Timeline with all cheeps
- View author Private Timelines by clicking on author names
- Click hashtags to view all cheeps with that hashtag
- Post cheeps
- Follow and unfollow authors
- Log out
- View the About Me page:
  - User information (username and email)
  - List of followed users
  - All user's cheeps
  - Download personal data
  - Delete account

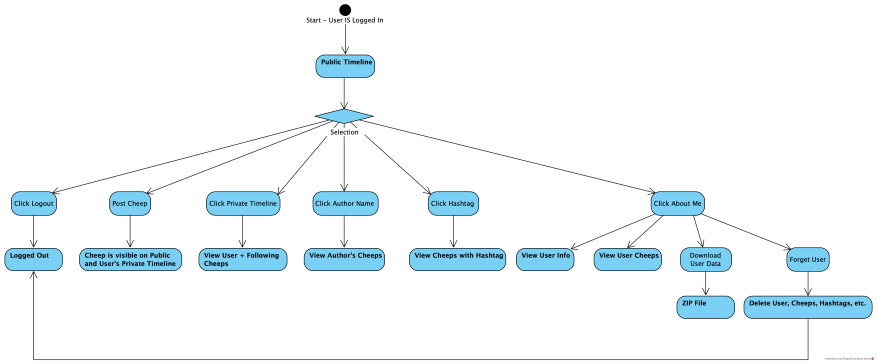

Figure 5: Illustration of authenticated user activities as UML activity diagram.

1.5 Sequence of functionality/calls through *Chirp!*

The sequence diagram shows the flow when a client requests the Public Timeline. It illustrates how the HTTP request is processed through Azure App Service, Routing, PublicModel, CheepService, CheepRepository, EF Core, and the SQLite database, then returned as rendered HTML.

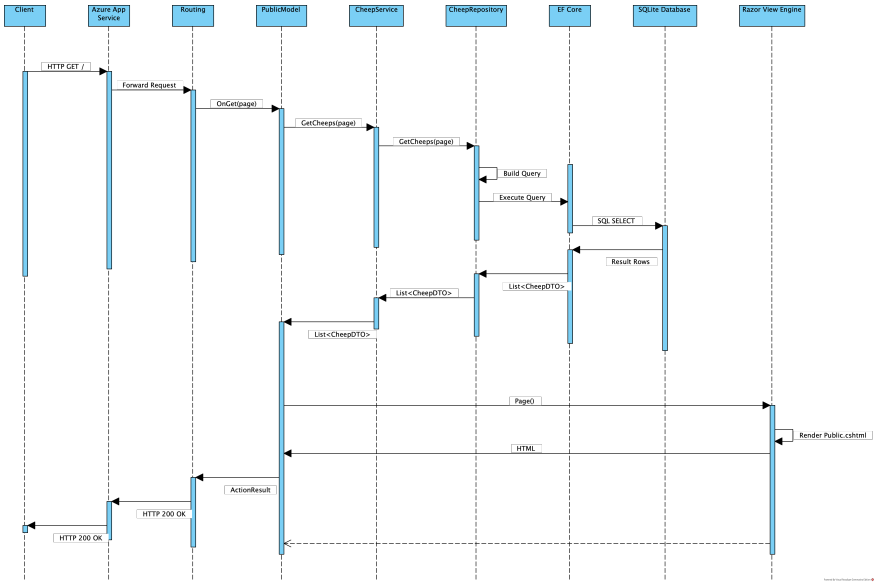

Figure 6: Illustration of the request flow through *Chirp!* as UML sequence diagram.

2 Process

2.1 Build, test, release, and deployment

The application uses GitHub Actions workflows for automated build, test, release, and deployment.

2.1.1 Build and Test

Triggered on push to main or on Pull Requests. The workflow checks out the code, sets up .NET, restores dependencies, builds the project, installs Playwright browsers, and runs tests.

2.1.2 Release

Triggered when a tag matching v* is pushed. The workflow builds and tests the application, then publishes for Windows, macOS, and Linux in parallel. ZIP files are created and attached to a release with the version tag.

2.1.3 Deploy to Azure

Triggered on push to main. The workflow has two jobs: Build (checkout, setup, build, publish, upload artifact) and Deploy (download artifact, login to Azure, deploy to Azure App Service).

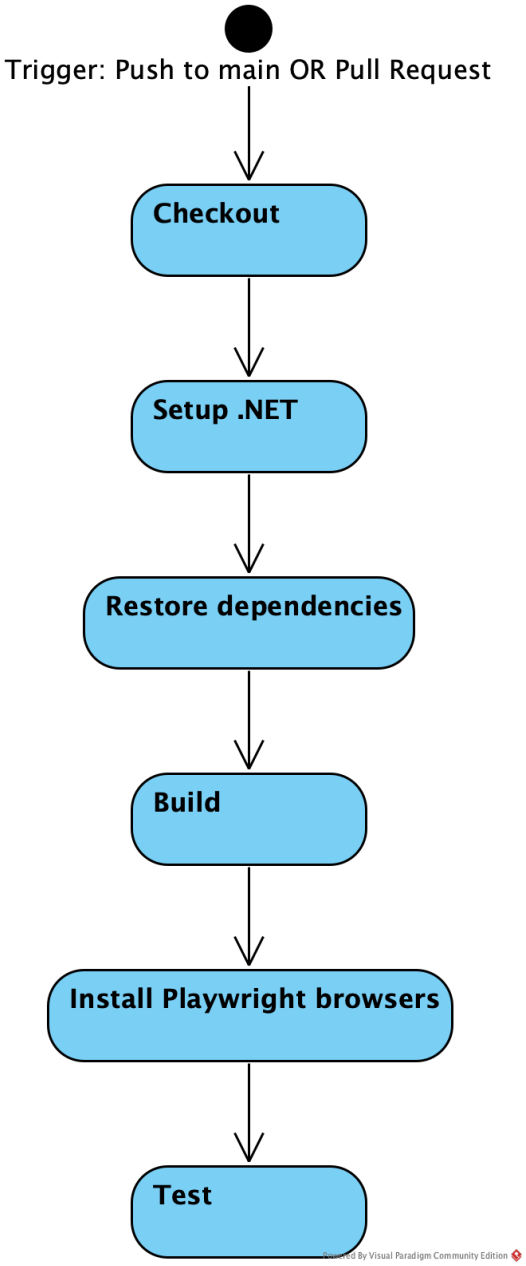

Figure 7: Illustration of the build and test workflow as UML activity diagram.

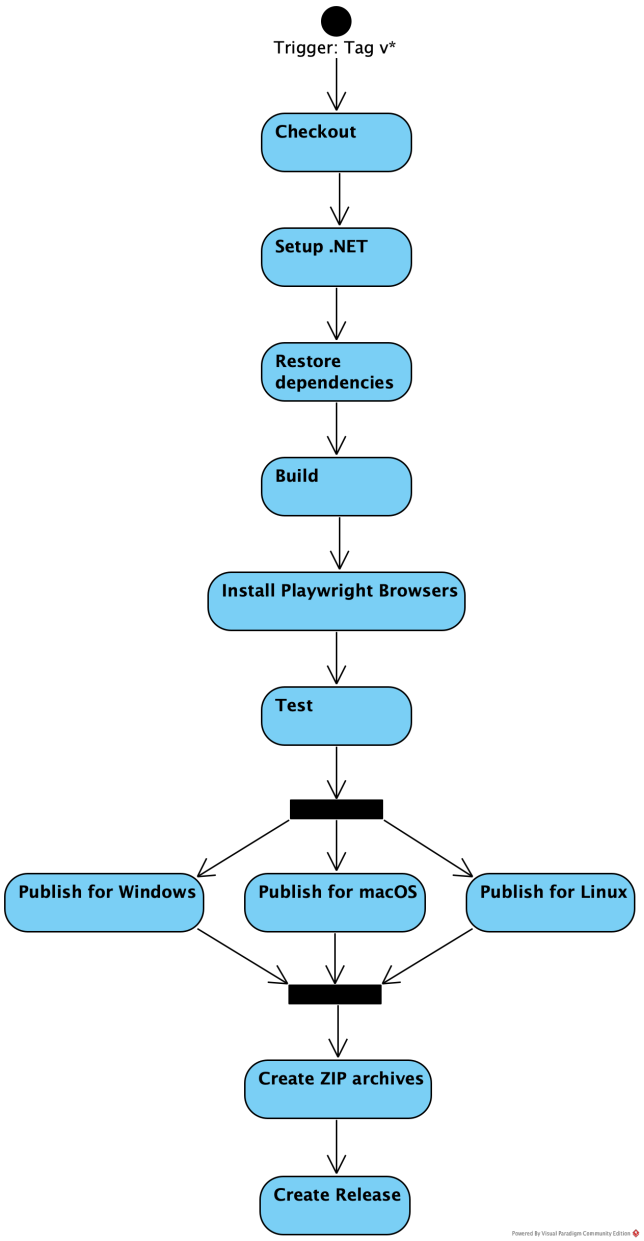

Figure 8: Illustration of the release workflow as UML activity diagram.

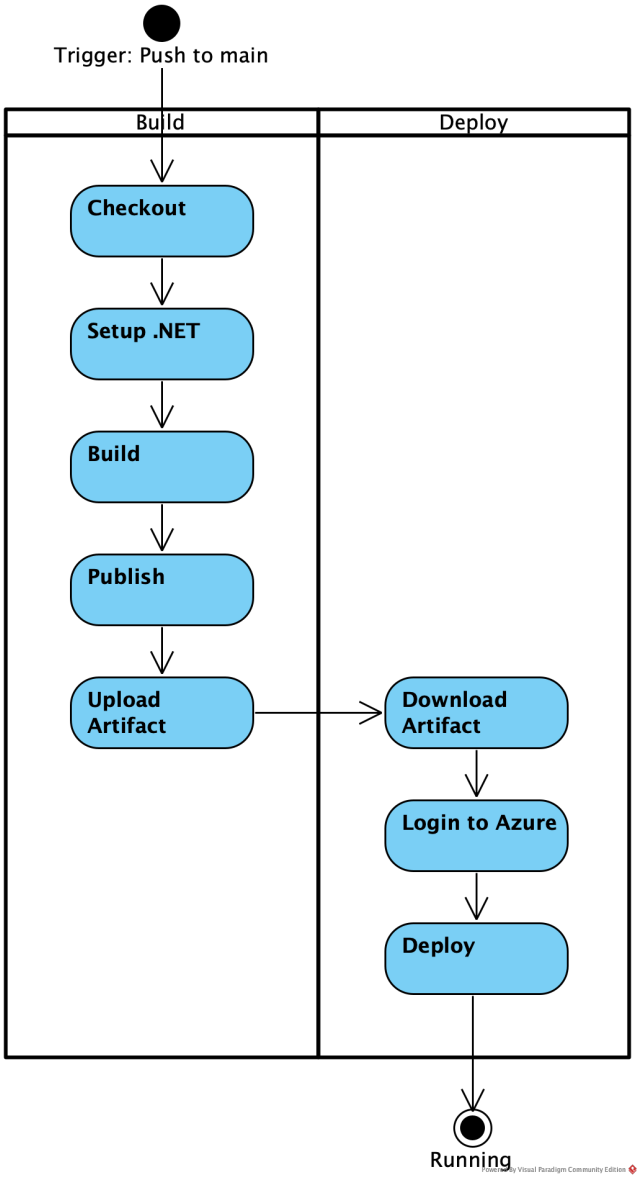

Figure 9: Illustration of the Azure deployment workflow as UML activity diagram.

2.2 Team work

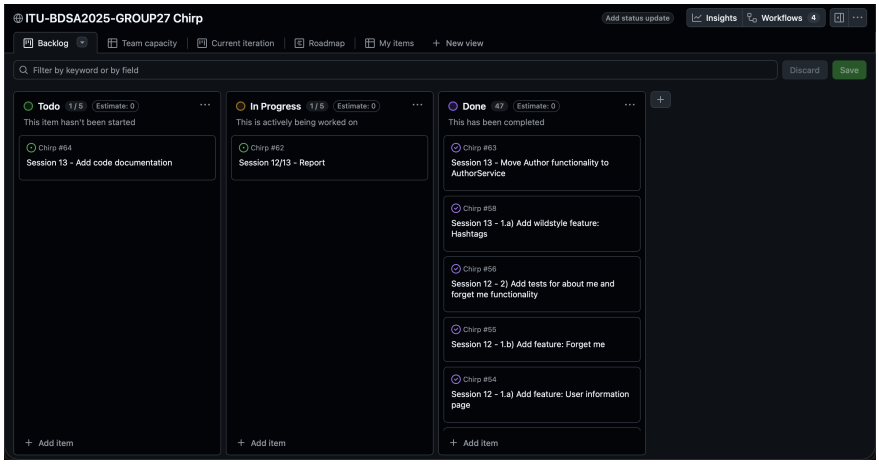

Figure 10: Illustration of the project board as screenshot.

At the time of writing this report, 47 issues have been completed. One issue remains in **TODO** (Session 13 - Add code documentation) and one is **In Progress** (Session 12/13 - Report). All required functionality has been implemented.

2.2.1 Issue workflow

Issues are created with the format: `Session <number> - <description>`. The issue body contains a user story at the top following the format `"In order to <receive benefit> as a <role>, I can <goal/desire>"`, followed by acceptance criteria below.

A branch is created using the naming convention `<type>/<description>`, where type is `feature/`, `refactor/`, or `docs/` depending on the issue type. This follows trunk-based development with short-lived feature branches. When a feature is complete, a pull request is created that needs to pass the Build and Test workflow to be merged to main. Usually reviewers would be assigned to the pull request, but as I work alone, I was allowed by TAs to merge without having to review my own pull requests. Once merged, the branch is deleted.

2.3 How to make *Chirp!* work locally

2.3.1 Prerequisites

The application requires **.NET 8.0 SDK** to be installed.

2.3.2 Clone the Repository

```
git clone https://github.com/ITU-BDSA2025-GROUP27/Chirp.git
cd Chirp
```

2.3.3 Configure GitHub OAuth

To enable GitHub login, register the application on GitHub.

- Homepage URL: `http://localhost:5273/`
- Authorization callback URL: `http://localhost:5273/signin-github`

Set up user secrets with the following commands:

```
dotnet user-secrets init
dotnet user-secrets set "authentication:github:clientId" "<YOUR_CLIENTID>"
dotnet user-secrets set "authentication:github:clientSecret" "<YOUR_CLIENTSECRET>"
```

2.3.4 Run the Application

From root, run:

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

The application will:

- Restore dependencies
- Apply migrations
- Seed the database with initial data
- Start the web server

2.3.5 Access the Application

Open a browser and navigate to `http://localhost:5273`. The application should be running with the public timeline visible.

2.4 How to run test suite locally

The test suite requires Playwright for UI and End-to-End tests. Install Playwright browsers:

```
pwsh test/Chirp.Playwright.Tests/bin/Debug/net8.0/playwright.ps1 install
```

Run all tests:

```
dotnet test
```

2.4.1 Tests

Unit Tests (Chirp.Razor.Tests):

- Repository tests (AuthorRepository, CheepRepository, HashtagRepository)
- Service tests (CheepService)
- Helper tests (CheepTextHelper)

Integration Tests (Chirp.Razor.Tests):

- Follow functionality
- Hashtag functionality
- Razor Page rendering

End-to-End/UI Tests (Chirp.Playwright.Tests):

- About Me page functionality
- Forget Me functionality
- Hashtag page interactions
- General UI interactions

3 Ethics

3.1 License

The project uses the MIT License. This license was chosen for its simplicity and compatibility with dependencies.

3.2 LLMs, ChatGPT, CoPilot, and others

Claude was used during debugging when errors were unclear or blocking progress. It helped interpret error messages and point out mistakes in the code.

Commits where Claude assisted were co-authored with Claude noreply@anthropic.com. Files containing AI-assisted code include comments noting this. Overall, Claude was helpful during debugging and reduced the time spent resolving errors.