Chirp! Project Report

Table of Contents

1 Introduction

Chirp! is a microblogging web application where users can post short messages ("cheeps") and interact with other users through following and liking. The goal of the project is to design and implement a maintainable ASP.NET Core application with authentication, persistence, Automated testing, and CI pipeline.

Unauthenticated users can view the public timeline. After logging in, users can create cheeps, follow and unfollow other authors, like cheeps, unlike cheeps, and view personalize timelines such as (Mypage, and FollowingTimeline). Chirp! is built using ASP.NET Core Razor Pages, Entity Framework Core, and ASP.NET Identity with a relational database backend

Out of scope: DMs and media uploads.

2 Design and architecture

2.1 Domain model

The following domain model captures the main entities and relationships in Chirp!.

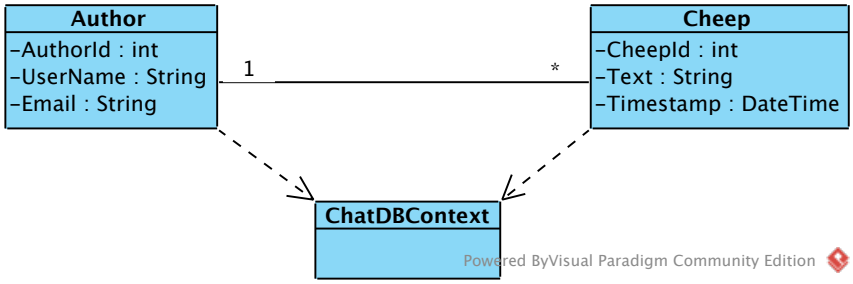

Figure 1: Domain model

The domain model consists of **Author** and **Cheep**.

Author represents a user in the system and is integrated with ASP.NET Identity. An author can create multiple cheeps, and each cheep belongs to exactly one author.

Authors can follow other authors, forming a many-to-many relationship. A cheep contains textual content and a timestamp.

All entities are persisted using Entity Framework Core via **ChatDBContext**.

2.2 Architecture — In the small

The following diagram illustrates the organization of the code base and its project dependencies.

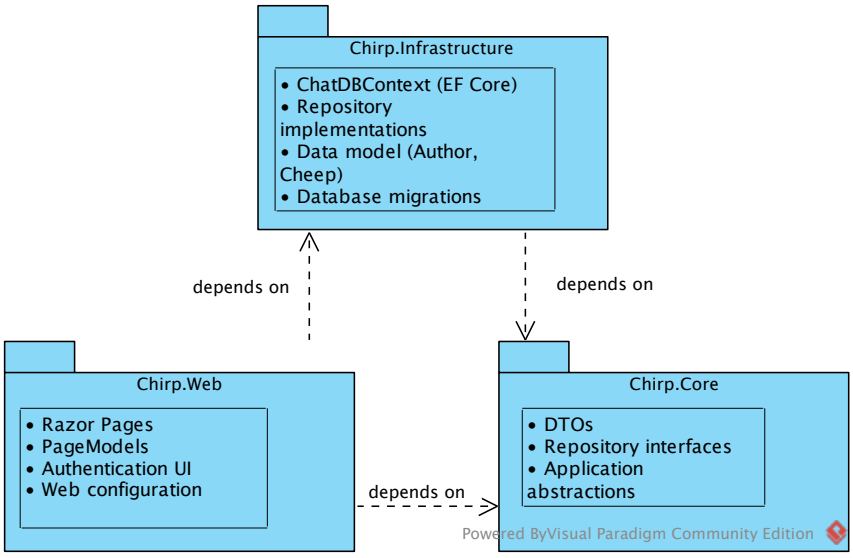

Figure 2: Architecture - in the small

The application follows an **onion architecture**.

- The **presentation layer** is implemented in **Chirp.Web** and contains Razor Pages, page models, and authentication endpoints.
- The **application layer** is represented by service and DTO abstractions defined in **Chirp.Core**.
- The **domain logic** is centered around the **Author** and **Cheep** entities.
- The **infrastructure layer** is implemented in **Chirp.Infrastructure** and contains repository implementations, database context, migrations, and ASP.NET Identity integration.

Dependencies point inward toward the core and domain layers.

2.3 Architecture of deployed application

The following diagram shows how the deployed Chirp! system is composed and how components communicate.

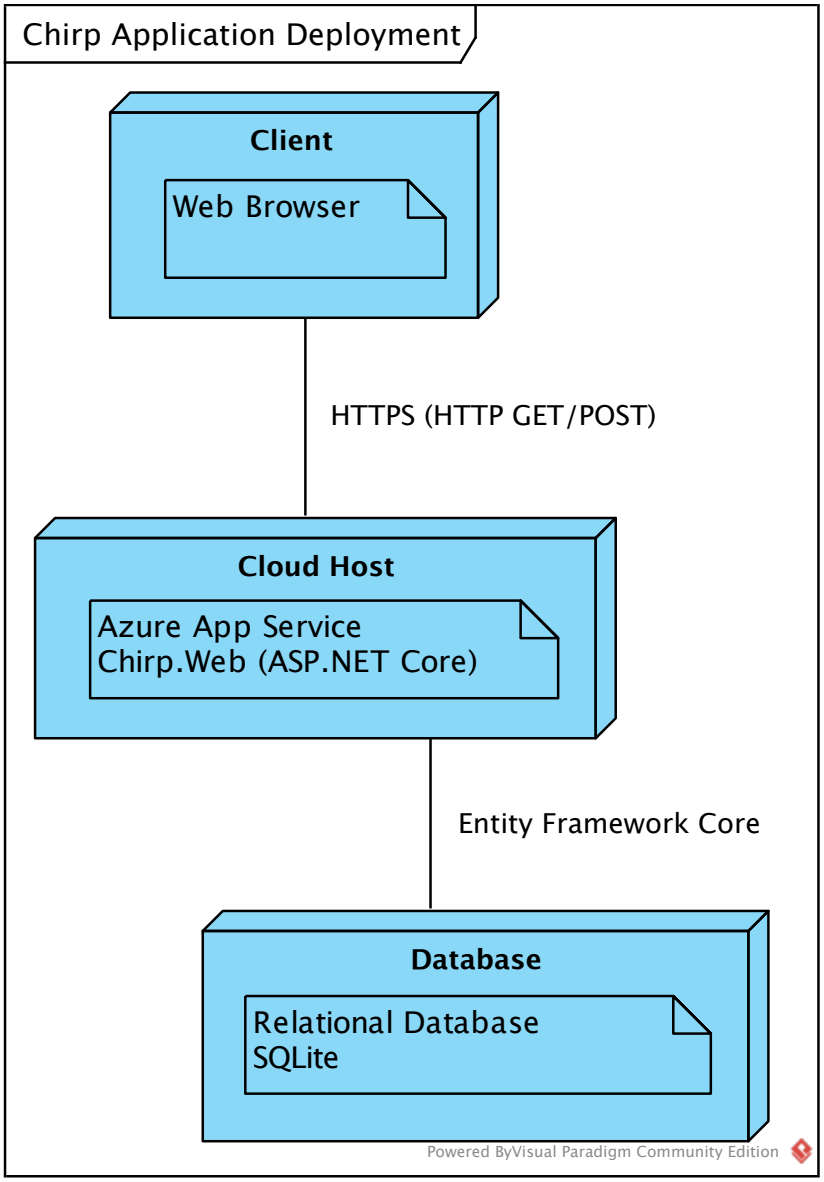

Figure 3: Deployment architecture

Chirp! is implemented as a client-server web application.

The client is a web browser that renders Razor Pages generated by the server. The server is an ASP.NET Core application built from **Chirp.Web** and deployed using GitHub Actions.

The server communicates with a relational database through **ChatDBContext** using Entity Framework Core. All communication between client and server is performed over HTTPS.

2.4 User activities

The following activity diagram illustrates a typical user journey through Chirp!.

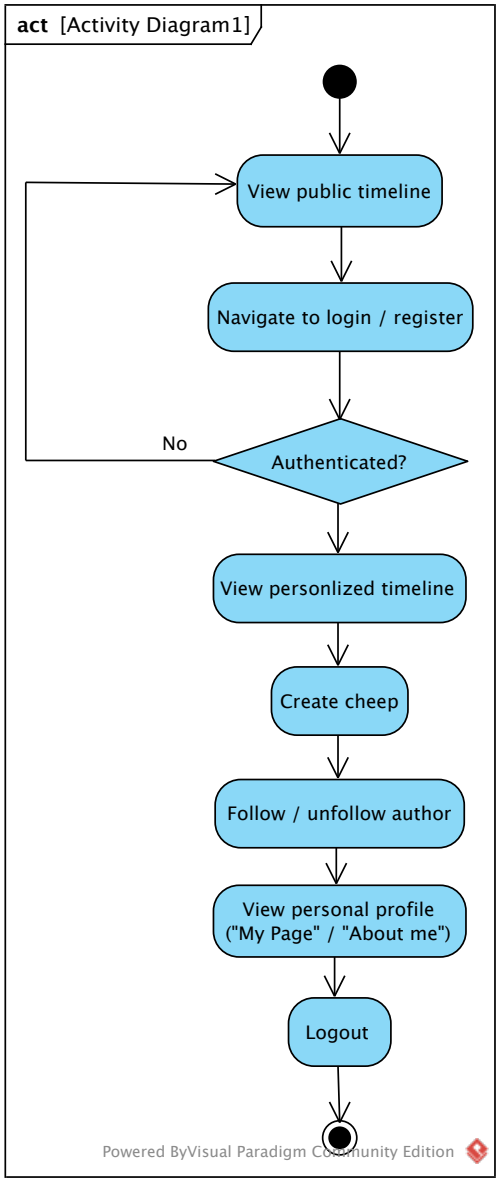

Figure 4: User activities

A non-authenticated user is presented with the public timeline rendered by

Public.cshtml. Non-authenticated users can view cheeps but cannot post cheeps or follow other users.

The user can authenticate using ASP.NET Identity login functionality. After authentication, the user can post cheeps, follow other authors, and view personalized timelines such as **MyPage** and **FollowingTimeline**.

Authenticated users can log out at any time.

2.5 Sequence of functionality/calls through Chirp!

The following sequence diagram shows the flow of an unauthenticated request to the root endpoint and the data retrieval required to render the page.

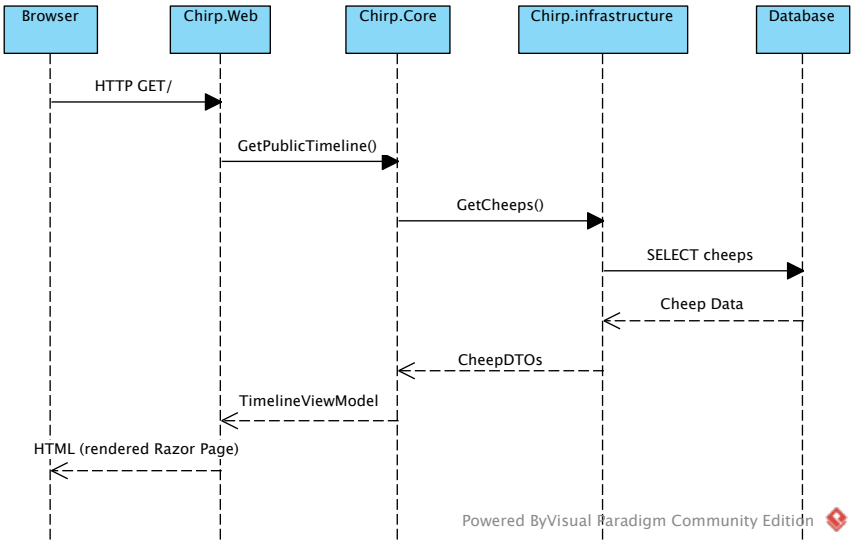

Figure 5: Sequence diagram: GET /

The sequence starts with an HTTP GET request to the root endpoint by a

non-authenticated user.

The request is handled by a Razor Page in **Chirp.Web**. The page model invokes repository methods through application-level abstractions. **CheepRepository** retrieves cheep data from the database via **ChatDBContext**.

The retrieved data is mapped to DTOs and returned to the Razor Page. The Razor Page renders the HTML response, which is sent back to the client browser.

3 Process

3.1 Build, test, release, and deployment

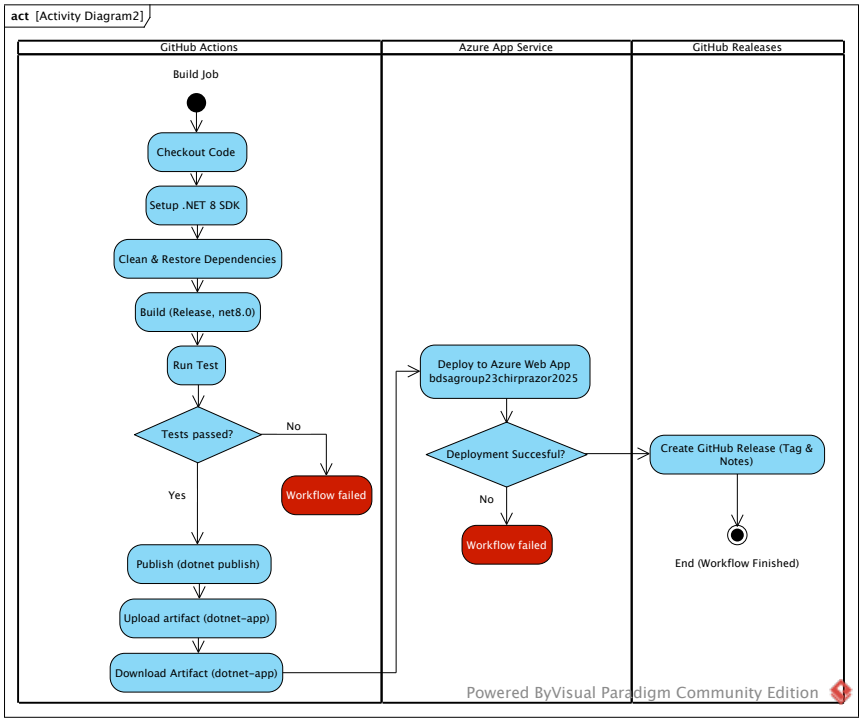

Figure 6: Build, test, release and deployment

Our GitHub Actions workflow "Build & Deploy — bdsagroup23chirprazor2025" is triggered on pushes to `main`, and can also be started manually via `workflow_dispatch`.
The `build` job checks out the repository, installs .NET SDK 8.0.415, restores and builds `src/Chirp.Web/Chirp.Web.csproj` in Release for `net8.0`, runs the automated test suite, publishes the application, and uploads the publish output as the `dotnet-app` artifact.
The `deploy` job downloads the artifact and deploys it to Azure App Service using `azure/webapps-deploy@v3` with the app name `bdsagroup23chirprazor2025` in the `Production` environment.

After a successful deployment, the `release` job creates a GitHub Release using `softprops/action-gh-release@v1` with the tag `deploy-${{ github.run_number }}` and metadata referencing the commit SHA and branch.

3.2 Team work

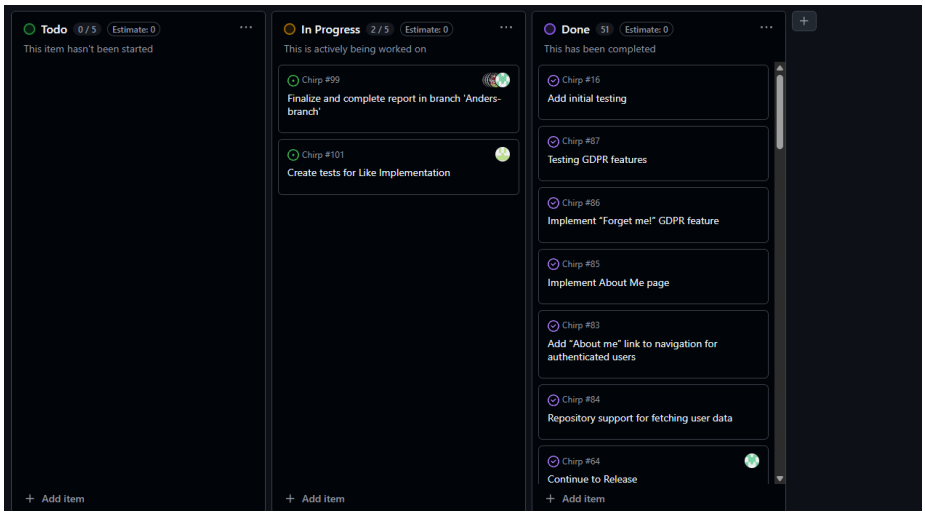

Figure 7: Project Board

The project board reflects the state of development immediately before hand-in. A small number of tasks remain unresolved, primarily related to optional features and UI polish. All required functionality for the Chirp! application is implemented.

Development starts with the creation of an issue describing a task. A feature branch is created from `main`. The feature is implemented and tested locally. A pull request is opened and reviewed. After approval, the changes are merged into the `main` branch.

We used pair working for complex tasks and areas with higher risk for example in

our integration of authentication. We typically worked in a “Driver” / “Navigator setup and rotated roles to spread knowledge across the team. These pair sessions helped align coding style and architecture design and reduced rework during code review, because design discussions happened before we implemented it.

We held weekly sync meeting to align on progress and priorities. in each meeting we: - Reviewed what was completed since the last sync (features, bugs, PRs merged) - Identified blockers and assigned owners to resolve them - Agreed on the next set of tasks and updated the project board accordingly

3.3 How to make Chirp! work locally

The repository is cloned from GitHub. Dependencies are restored using `dotnet restore`. The database is initialized using Entity Framework Core migrations. The application is started using `dotnet run` from the **Chirp.Web** project.

OBS To run GitHub OAuth, setup GitHub client ID and client secret. To set up use credentials dotnet user.secrets:

dotnet user-secrets init

dotnet user-secrets set “authentication_github_clientId” “Ov23liNbMZLbyI73hKwK”

dotnet user-secrets set “authentication_github_clientSecret” “7e0b07f9b29107dcb10433c3af8804345f31ee72”

The application is accessible in a web browser on the configured local port.

3.4 How to run test suite locally

To run the test suite locally, execute the following command from the root of the repository:

```
dotnet test
```

The test suite includes: - **Unit tests**: Testing individual components in isolation (e.g., repository methods). - **Integration tests**: Testing the interaction between multiple components and the database. - **UI/End-to-End tests**: Testing the application from a user’s perspective using Playwright.

4 Ethics

4.1 License

The application is released under the MIT License.

The MIT License allows unrestricted use, which encourages adoption and reuse. Its simplicity lowers barriers for others to build upon the software while still requiring attribution on the original authors. However, it has notable drawbacks. Modified or improved versions do not have to be shared publicly, which can limit contributions back to the open-source community. In addition, the license

provides no warranty or liability, meaning the software is offered “as is.” This protects the authors but places all rick on the users. Despite these downsides, the benefits outweigh the limitations, making the MIT License a suitable choice for out application.

4.2 LLMs, ChatGPT, CoPilot, and others

ChatGPT was used during development as a support tool. It was primarily used for explanations, code suggestions, and assistance with documentation.

All generated content was reviewed and adapted manually. Overall, the use of LLMs reduced development time without replacing developer judgment.