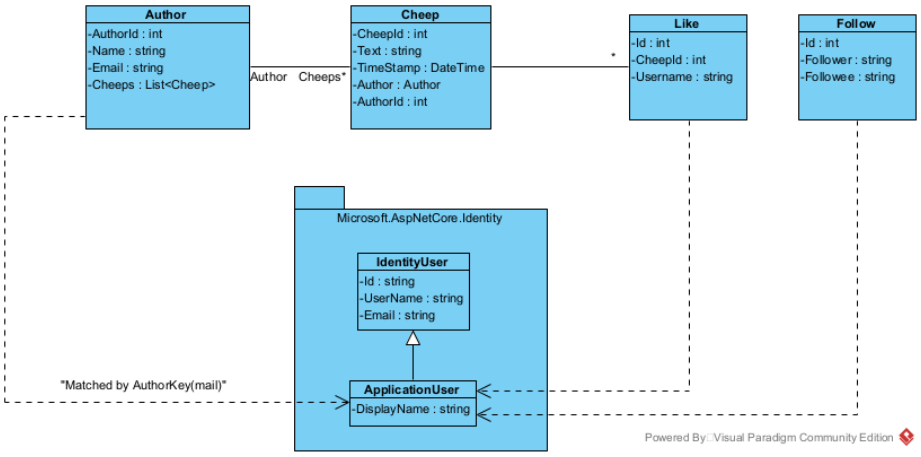
Chirp! Project Report

ITU BDSA 2025 Group <25>

Joachim Blom-Hanssen jblo@itu.dk
Heðin Mortensen hedm@itu.dk Ahmad Shahid ahsh@itu.dk
Anton Krøis antk@itu.dk William le Fèvre wile@itu.dk

1 Design and Architecture of *Chirp!*

1.1 Domain model



The domain model of Chirp! is illustrated in the diagram. Cheep represents a posted message and is the primary entity in the domain. Each cheep is owned by exactly one Author, forming a one-to-many relationship. This relationship is enforced by a foreign key (Cheep.AuthorId).

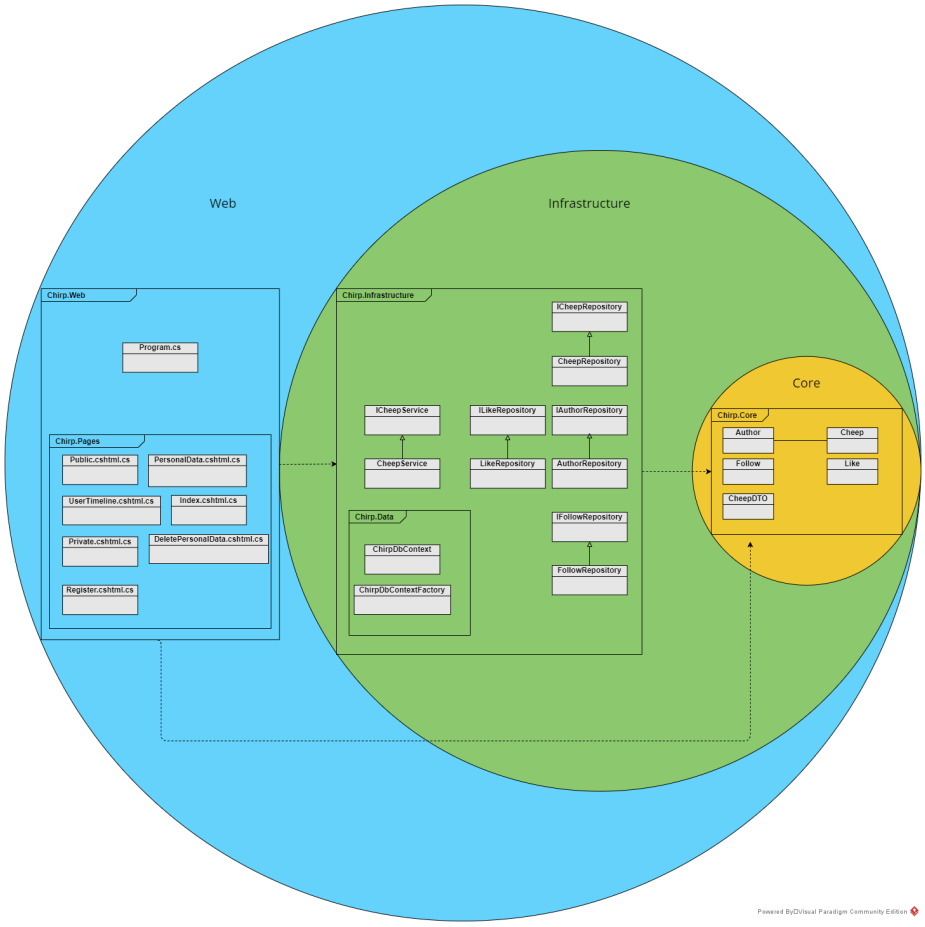
User authentication and account data are handled by ASP.NET Identity through ApplicationUser, which inherits from IdentityUser and stores account information such as Mail and DisplayName in its table. The domain model does not reference Identity users via foreign keys. Instead, it relies on string-based identifiers resolved at the application level.

When a cheep is created, the logged-in user's email (User.Identity.Name) is used as the author identifier and stored in the Author entity. When cheeps are displayed, presentation data is resolved by matching this identifier against

ApplicationUser.Email to obtain the corresponding display name. Similarly, Follow (Follower, Followee) and Like (Username) store user references as strings and are interpreted by comparing them to Identity user data at runtime. These relationships are therefore shown as dotted, application-level dependencies rather than foreign key–enforced associations.This design avoids additional database joins but shifts responsibility for consistency to the application layer.

Finally, Like is conceptually associated with Cheep through CheepId, meaning that a cheep can have multiple likes. Since this association is not enforced by a database foreign key, it is represented as a non-enforced domain relationship.

1.2 Architecture — In the small



The Chirp application follows an onion architecture pattern organized into three concentric layers.

At the center is the Core layer (Chirp.Core), which contains the domain entities (Author, Cheep, Follow, Like) and data transfer objects CheepDTO defined

in DataModel.cs. This layer has no external dependencies and represents the application's business domain.

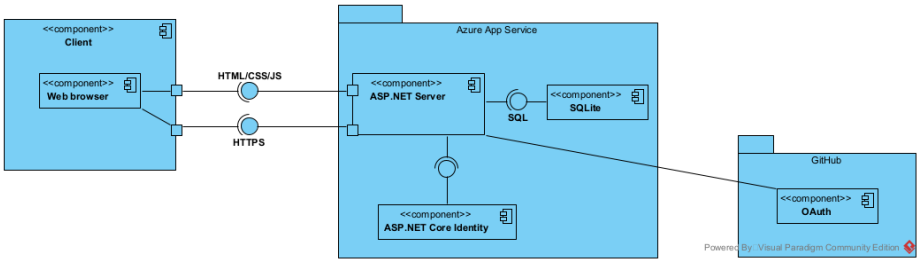
The Infrastructure layer (Chirp.Infrastructure) surrounds Core and handles data access concerns. It contains the Entity Framework ChirpDbContext (in Data/ChirpDbContext.cs), repository implementations (CheepRepository, AuthorRepository, FollowRepository, LikeRepository), and their corresponding interfaces. The CheepService class provides business logic operations. This layer depends only on Core.

The outermost Web layer (Chirp.Web) contains the presentation logic with Razor Pages (Public.cshtml.cs, UserTimeline.cshtml.cs, Private.cshtml.cs) and the application entry point (Program.cs) where dependency injection is configured. This layer depends on both Infrastructure and Core.

Dependency Flow

Dependencies flow inward: Web → Infrastructure → Core. The Core layer has zero external dependencies, Infrastructure references only Core, and Web references both outer layers. This ensures the domain logic remains independent of infrastructure and UI concerns.

1.3 Architecture of deployed application



The Chirp application follows a client-server architecture deployed on Microsoft Azure.

The client component consists of standard web browsers (Chrome, Firefox, Safari, Edge) that communicate with the server via HTTPS. Users interact with the application through rendered HTML pages with embedded CSS styling and minimal client-side JavaScript. The browser sends HTTP requests and receives HTML responses containing the complete page content for display.

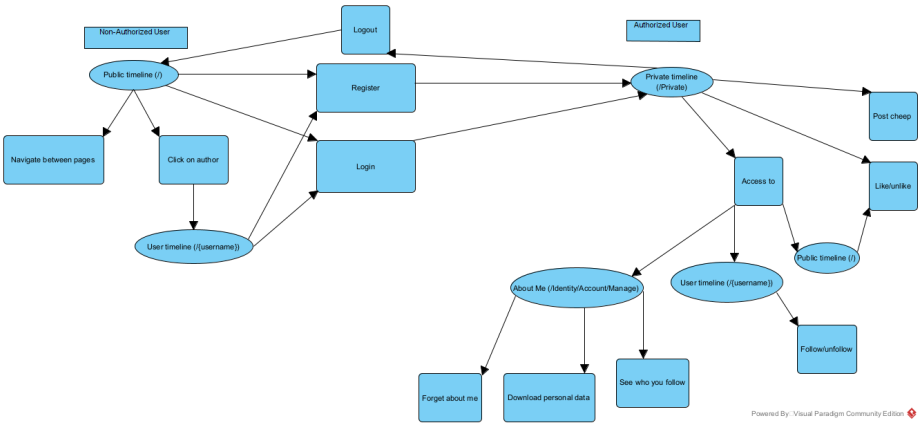
The server component is an ASP.NET Core 9.0 web application hosted on Azure App Service (bdsagroup25chirprazor3) in the Production slot. The server runs the Chirp.Web project compiled as a self-contained deployment. It handles all business logic, data access, authentication, and page rendering using Razor Pages. The application uses SQLite as its database (Cheep.db), which runs on the same server instance as the web application. User authentication is provided through two mechanisms: GitHub OAuth for social login and ASP.NET Core

Identity for local account registration and management.

Communication between client and server occurs over HTTPS (HTTP Secure) using the standard request-response pattern. When a user navigates to a page, the browser sends an HTTP GET request to the server. The server processes this request by executing the appropriate Razor Page handler, querying the database through Entity Framework Core, and rendering the complete HTML page. This server-rendered HTML is then sent back to the client as an HTTP response. For user actions such as posting cheeps or following users, the browser submits HTTP POST requests containing form data, which the server processes and responds to with either a redirect or a new page rendering.

Deployment occurs automatically through GitHub Actions. When code is pushed to the main branch, a workflow builds the application using .NET 9.0, runs the test suite, publishes the Web project, and deploys the compiled artifacts to Azure App Service using Azure login credentials stored as GitHub secrets.

1.4 User activities



The Chirp application provides different functionality based on user authentication status.

Non-Authorized User Journey

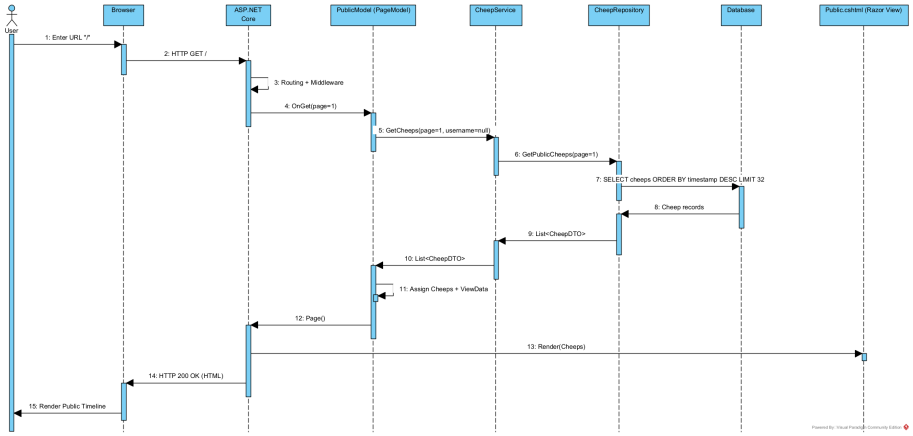
An unauthorized user arriving at the application is presented with the public timeline at the root endpoint (/). This page displays cheeps from all users in reverse chronological order, paginated with 32 cheeps per page. Users can navigate between pages and click on author names to view individual user timelines (/{username}). Each cheep displays the author name, message text, timestamp, and like count. Unauthorized users cannot interact with the content; they are unable to post cheeps, follow users, or like cheeps. The application header provides authentication options for registration and login. Registration can be performed using a local account with a display name, email, and password, or via GitHub OAuth. In the GitHub-based registration flow, only the user's email

address is provided by the OAuth provider, and the application therefore assigns the display name to be equal to the email address. This behavior is a known limitation, and attempts were made during development to improve this mapping. Login supports both local credentials and GitHub OAuth authentication.

Authorized User Journey

After successful authentication, users are redirected to their private timeline (/Private). This personalized feed displays cheeps authored by the user themselves as well as by users they follow. Cheeps are shown in reverse chronological order, paginated with 32 cheeps per page, and can optionally be sorted by most likes. At the top of the page, users are provided with a form for posting new cheeps, subject to a 160-character limit. From any user timeline, authenticated users can follow or unfollow other users using dedicated buttons. Each cheep includes a heart icon that allows users to like or unlike the cheep, with the like count updating accordingly. Authenticated users retain full read access to the public timeline and all user timelines, while gaining write and interaction capabilities in their authenticated context. The header navigation displays the logged-in user's identifier and provides access to additional user-specific pages. The "About Me" page presents an overview of the users that the authenticated user is currently following. A separate personal information section allows users to download their personal data. Users may also choose to permanently remove their account using the "Forget about me" functionality. To confirm this action, users authenticated via local credentials must re-enter their password, while users authenticated through GitHub OAuth must explicitly confirm the action via a checkbox. Upon confirmation, all user-specific information is deleted, and any cheeps previously posted by the user are anonymized rather than removed. This ensures compliance with GDPR

1.5 Sequence of functionality/calls trough *Chirp!*

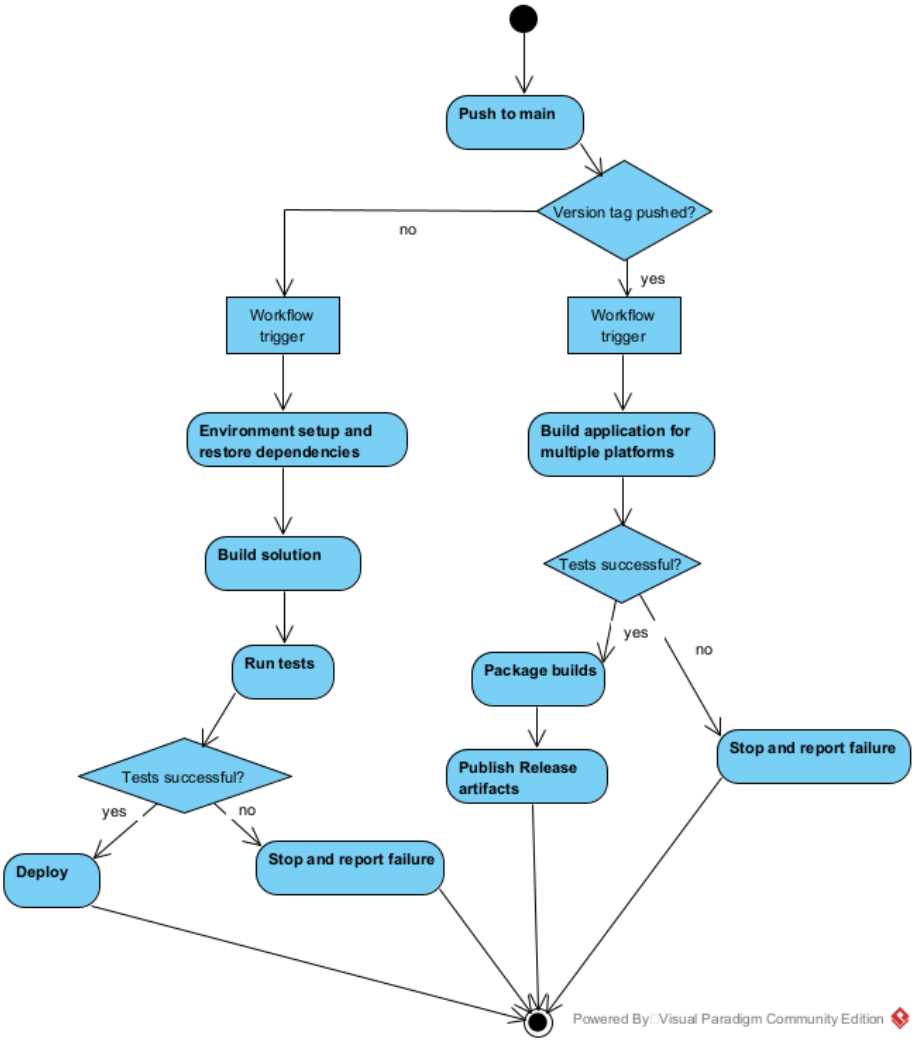


This Chirp sequence diagram illustrates the flow of data for an unauthorized user.

The sequence diagram illustrates the complete request-response cycle of the Chirp application, starting from an unauthorized user requesting the root endpoint and ending with a fully rendered web page displayed in the browser. The request flows through all architectural layers of the system, including the browser, ASP.NET Core routing and middleware, Razor PageModels, application services, repositories, and the database. During this process, data is transformed multiple times: database rows are retrieved via SQL queries, mapped to entity objects, converted into Data Transfer Objects (DTOs), and finally rendered as HTML by the Razor view. The diagram highlights different types of interactions, such as HTTP requests and responses, internal C# method calls, and database queries. This layered architecture promotes separation of concerns, improves maintainability, and makes the system easier to extend and reason about.

2 Process

2.1 Build, test, release, and deployment



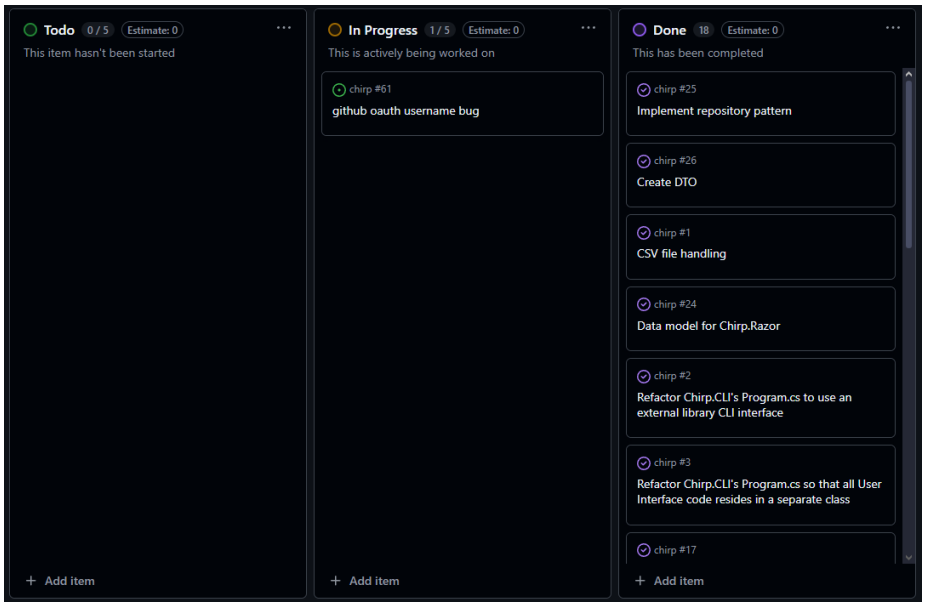
Our CI/CD pipeline is implemented through GitHub Actions with three distinct workflows that automate different aspects of the development process.

The primary deployment workflow (main_bdsagroup25chirprazor3.yml) triggers on every push to the main branch. It sets up the .NET 9.0 environment, restores NuGet dependencies, builds the solution, and executes the test suite. If all tests pass, the workflow publishes the application and deploys it to our Azure App Service instance (bdsagroup25chirprazor3) in the Norway East region. This ensures that every change merged to main is automatically tested and deployed to production, creating a continuous deployment pipeline.

Our release workflow (release.yml) activates when a version tag (following the pattern v*) is pushed to the repository. This workflow builds the application in Release configuration for multiple target platforms: Linux x64, Windows x64, macOS x64, and ARM64 variants for both Linux and macOS. After successful compilation and testing, it packages each platform-specific build as a compressed archive and publishes them as GitHub Release artifacts. This automated release process eliminates manual packaging work and ensures consistent, reproducible releases across all supported platforms.

Additionally, we maintain a lightweight automation workflow (Issue-projectBoard-automation.yml) that automatically labels newly opened or reopened issues with "triage", helping us maintain consistent issue management practices.

2.2 Team work



Flow of activities

The Issues are created with a user story, Acceptance criteria, Label, and are automatically labeled with triage using Actions workflow. Then manually added to the project board and initially placed in Todo, and moved between In Progress, Done, accordingly by the team as work progresses.

When someone begins on a task they move the issue to In Progress and a branch is made where work is done using trunk based development. Once the implementation is done a pull request is opened where other team members review the code and comments are made either verbally when done together on premises or through the pull request itself. If approved the merge goes through the automated CI checks (build and tests) run on the pr. If accepted then its

merged into main and automatically deployed and the issue is closed and moved to the Done column on the board.

At the time of hand-in, the project board shows one unresolved task in the In Progress column: chirp #61 – GitHub OAuth username bug which describes the issue of currently, when a user registers using GitHub OAuth, the application automatically assigns the user's GitHub email address as their username. This behavior is incorrect and does not align with the expected registration flow of the application. The issue never reached the valid pull request steps.

2.3 How to make *Chirp!* work locally

Prerequisites: • .NET SDK 9.x (the web project targets net9.0) • Only needed for troubleshooting migrations) EF Core CLI: If dotnet ef is not recognized, install it: dotnet tool install –global dotnet-ef • create a GitHub OAuth App and obtain a Client ID and Client Secret dotnet user-secrets set "Authentication:GitHub:ClientId" "" dotnet user-secrets set "Authentication:GitHub:ClientSecret" ""

Clone the repository git clone https://github.com/ITU-BDSA2025-GROUP25/chirp cd chirp

Run the web app Navigate into the web project and start it: cd src cd Chirp.Web dotnet run click on localhost: link in terminal

Troubleshooting: migration / database issues If you get database/migration errors (e.g., the DB schema is out of sync), reset the local SQLite database files and re-apply migrations. Delete Cheep.db in (Chirp.Web) (possible in Chirp.Infrastructure if typed wrong commands) Recreate database from migrations (from Chirp.Infrastructure) cd src cd Chirp.Infrastructure dotnet ef database update –startup-project "../Chirp.Web/Chirp.Web.csproj" then run dotnet in Chirp.Web again after

2.4 How to run test suite locally

There are two separate test folders in the "test" folder, Chirp.Tests and PlaywrightTests. To run them you have the following options:

Option 1 1) Open a terminal (cmd, PowerShell or Terminal in your IDE) 2) Navigate to the test folder. Ensuring the path resembles this. [...]/chirp/testChirp.Tests OR [...]/chirp/PlaywrightTests 3) Run all tests with the command: dotnet test

To run playwright tests, make sure that playwright is installed on your machine.

Option 2 Using JetBrains Rider, there is a built-in tool to use tests. You can use the keyboard shortcut : Alt + Shift + 8 This will give a better and clearer overview of the different tests

The project is organised into two distinct testing folders, each dedicated to a specific area of the application.

Chirp.Tests focus on backend functionality, ensuring that the data layer operates correctly. This includes verifying critical operations such as user creation and deletion, following features, and API reliability. To protect sensitive production data and maintain test isolation, each test suite utilises an in-memory SQLite database rather than interacting directly with the live database. This is supported by two dedicated database fixtures.

PlaywrightTests handles frontend and user interface validation. Using the Playwright tool, this suite simulates real user interactions and pathways throughout the website. It ensures all UI components, such as buttons, forms, and navigational elements, function as expected in a browser environment.

3 Ethics

3.1 License

Chirp! is released under the MIT License. The license file (LICENSE.md) is located in the root of the GitHub repository. We chose the MIT License because it is a simple and permissive open-source license that allows the software to be freely used, modified, and distributed. This makes it appropriate for an educational project, while the included warranty disclaimer protects us from liability.

3.2 LLMs, ChatGPT, CoPilot, and others

ChatGPT and Claude were used as a supportive tool during development. They were primarily applied to clarify the new framework concepts introduced in this course, such as ASP.NET Core, Entity Framework Core, Razor Pages, by helping with understanding documentation and interpreting error messages. Sometimes it was used to discuss whether a given solution could be improved from a software architecture perspective, and why tell it to also reason why that might be the case However in the end all suggested solution required manual checking, adaptions and reasoning to ensure they fit the projects architecture.