Chirp! Project Report

ITU BDSA 2024 Group 6

Benjamin Ormstrup - beor@itu.dk
Marcus Frandsen - megf@itu.dk
Victor de Roepstorff - vicd@itu.dk
Christian Jörgensen - chpj@itu.dk
Valdemar Mulbjerg - vamu@itu.dk

1 Design and Architecture of Chirp!

1.1 Domain model

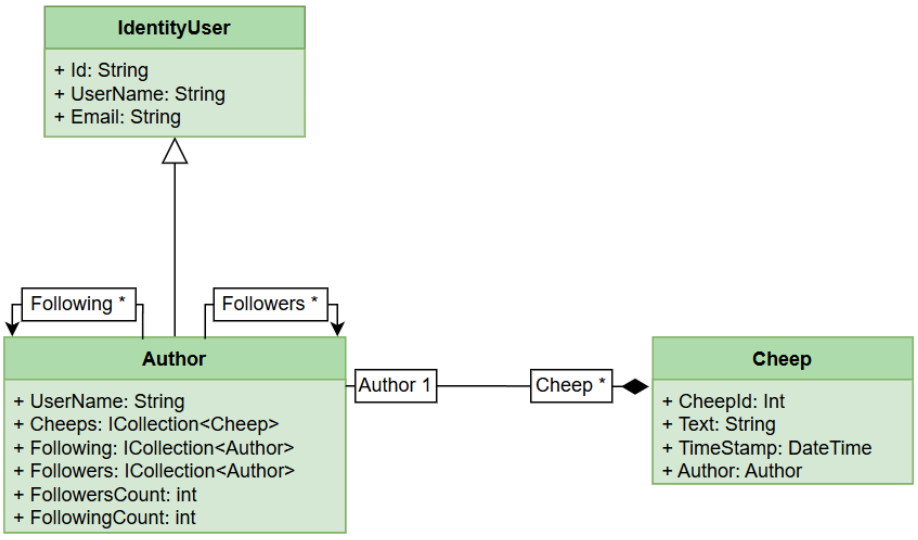

Figure 1: *Illustration of the Chirp! domain model*

The UML class diagram above provides an overview of the core domain model for our *Chirp!* application, highlighting the primary entities, their attributes, and the relationships between them.

A key aspect of the diagram is the cardinality between the entities, which defines how objects in one class relate to objects in another.

1.2 Architecture — In the small

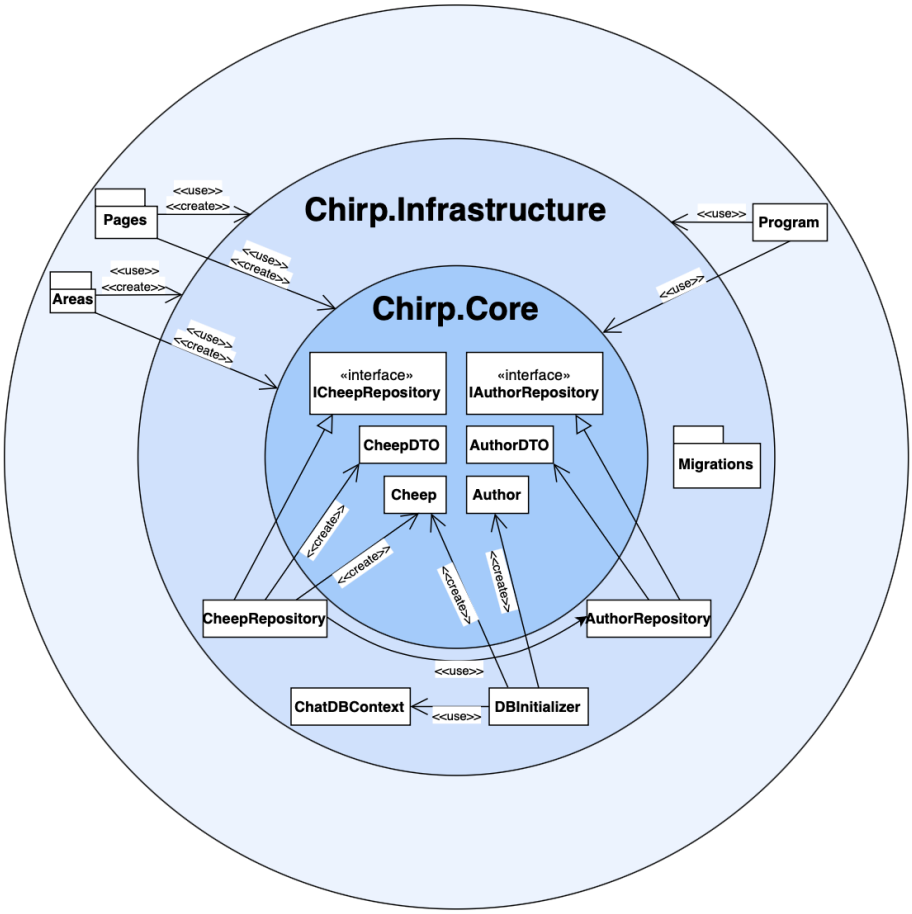

Figure 2: *Illustration of the Chirp! architecture*

The union architecture diagram above visually represents the layered structure of the Chirp! application. The diagram consists of three circles with a different shade of blue, each symbolizing one of the core architectural layers: Core, Infrastructure, and Web. The components within each circle represent the key parts or responsibilities of that layer.

The arrows throughout the diagram represent the unidirectional dependency flow of the application, where each layer depends only on the layers inside of itself:

- The Core layer is independent and does not depend on any outer layer.
- The Infrastructure layer relies on the Core while still remaining separate from the Web layer.

- The Web layer depends on both the Infrastructure and Core layers to deliver functionality to the user.

This layered structure ensures separation of concerns, making the program easily maintainable, testable, and scalable. Each layer can be adjusted, without having a direct impact on the logic and functionality of the layers above it.

1.3 Architecture of deployed application

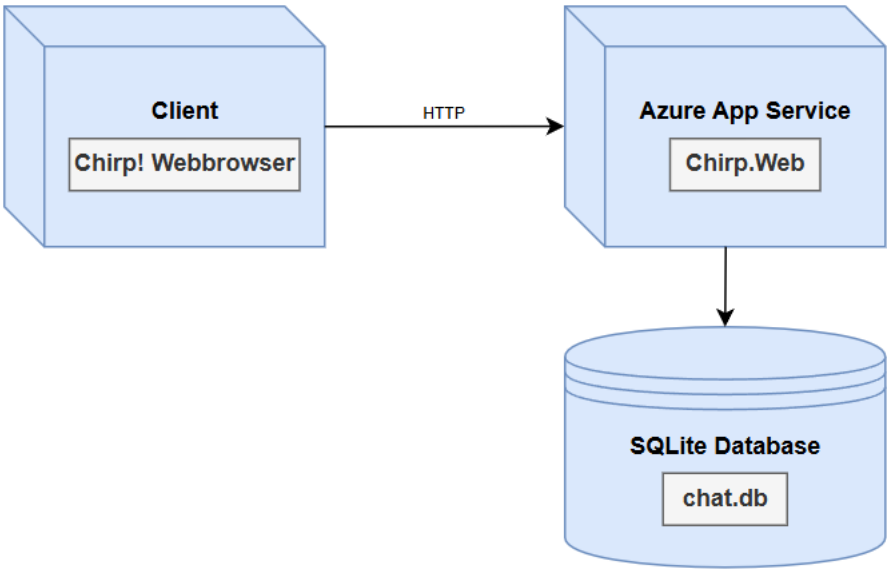

Figure 3: *Application Architecture diagram of the deployed application*

Above is a diagram that illustrates the architecture of our deployed Chirp! application which focuses on the client-server relation.

1.4 User activities

The following two figures illustrate distinct user journeys through the Chirp! application, tailored for both unauthorized users(not logged in) and authorized users (logged in). Each figure maps out two specific journeys, showing how users can interact with the system's key features.

1.4.1 Unauthorized Journey

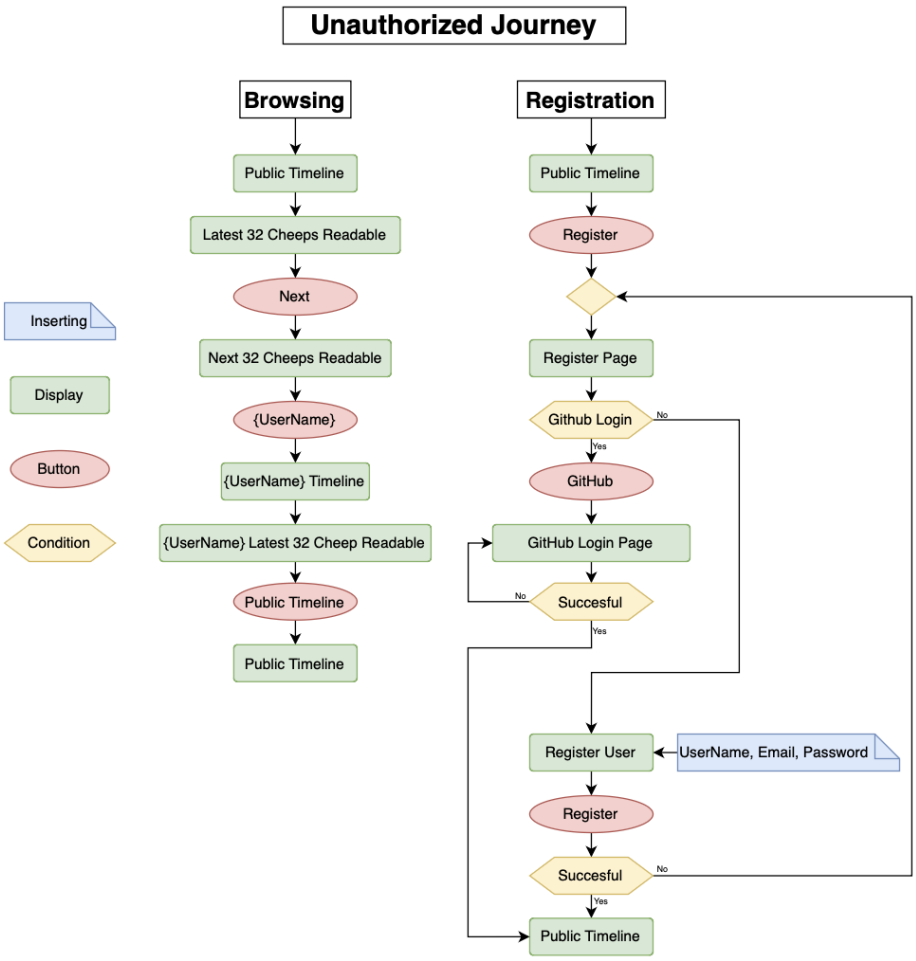

Figure 4: *Activity diagram of an unauthorized users journey*

This diagram focuses on the experience of users who are not logged into the system.

The user journey on the left, "Browsing", highlights the program's accessibility

for unauthenticated users, allowing them to explore content without having to register first. The user can navigate the public timeline as well as other authenticated users timelines.

The user journey on the right, "Register", emphasizes the application's user-friendly onboarding process, using GitHub OAuth for convenience and ensuring new users can easily register as a user in the system. After registering the user is automatically logged in and navigated to the public timeline.

1.4.2 Authorized Journey

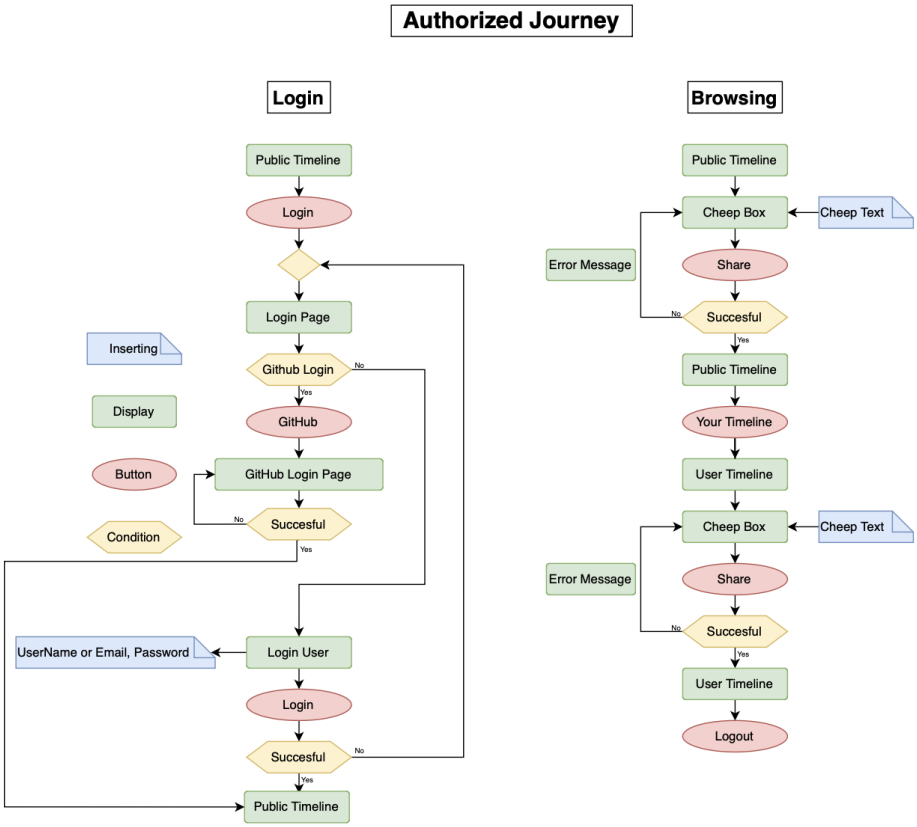

Figure 5: *Activity diagram of an authorized users journey*

In contrast to the previous figure, this diagram depicts the program's accessibility for a user who is already registered with the Chirp! application.

The user journey on the left, "Login", showcases the login process of an authenticated user, ensuring easy access to the full functionality of the program by allowing users to login with both username and email as well as a separate GitHub option.

The user Journey on the right, “Browsing”, highlights the interactive features available to logged-in users, such as posting content and engaging with their own timeline, which form the core functionality of the Chirp! application.

1.5 Sequence of functionality/calls trough Chirp!

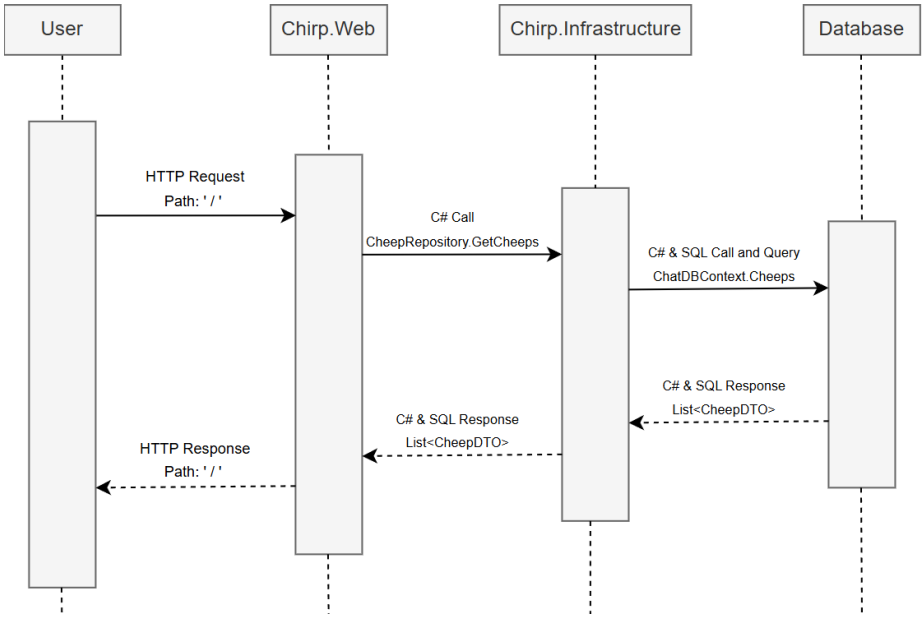

Figure 6: *Sequence of functionality diagram of the calls through Chirp!*

The diagram above shows the sequence of functionality from when an unauthenticated user sends a HTTP GET request to the root endpoint of our application (/) until the server responds with a fully rendered HTML page.

2 Process

2.1 Build, test, release, and deployment

2.1.1 Build and Test

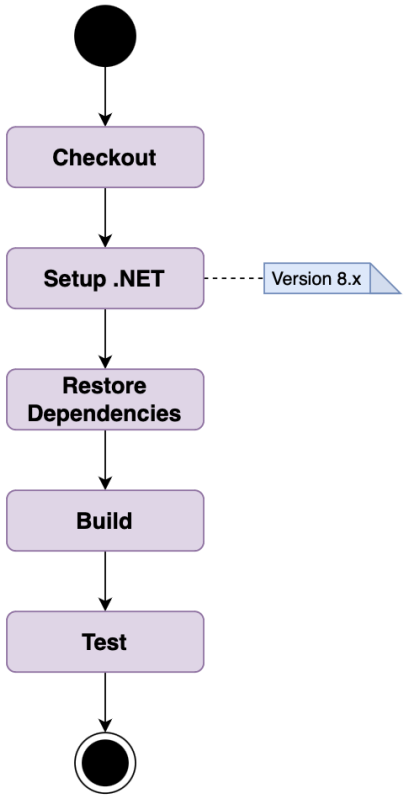

Figure 7: *Build and Test diagram of the build_and_test.yml workflow*

The diagram above shows our GitHub Actions workflow for building and testing the *Chirp!* application. The workflow runs when code is pushed to the main branch or a pull request targets it.

It starts by checking out the repository and setting up .NET version 8.x. The dependencies are restored, and the application is built. Tests are then executed, ensuring both functional and browser-based components work as expected.

This workflow ensures new changes do not break the build or tests.

2.1.2 Release

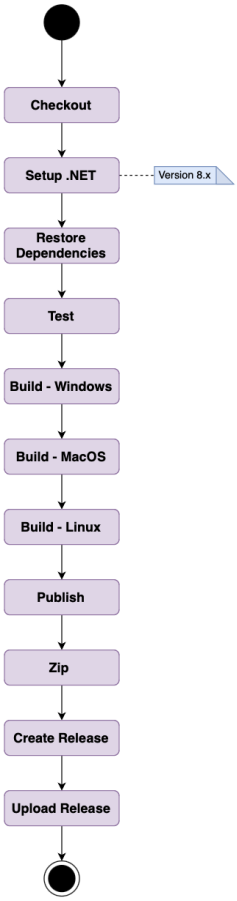

Figure 8: *Release diagram of the release.yml workflow*

The diagram above illustrates our *Chirp!* release workflow, which runs when a tag matching the format v* is pushed to GitHub. This triggers the process to build, test, and release the application.

The workflow starts with Checkout and setting up .NET version 8.x. It restores dependencies, runs unit tests, and builds the project for deployment. The application is then published, zipped, and a GitHub release is created. Finally, the zipped build is uploaded as a release asset, allowing it to be downloaded and deployed.

This ensures that every tagged release is thoroughly tested and packaged for distribution.

2.1.3 Deploy

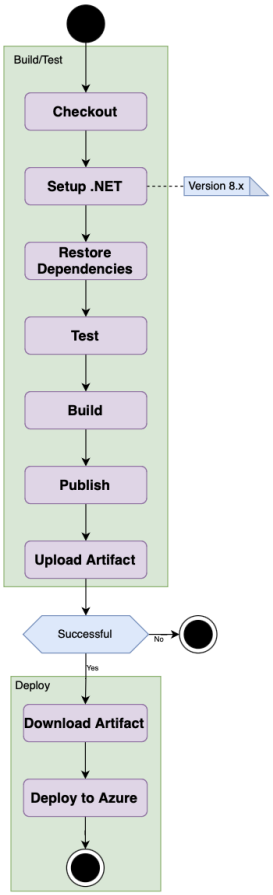

Figure 9: *Deploy diagram of the deploy.yml workflow*

The diagram above shows the Build and Deploy workflow for our *Chirp!* application. It runs on pushes to the main branch or manual triggers. The workflow builds, tests, and deploys the application in two steps:

- Build and Test: The code is checked out, dependencies are restored, unit tests are run, and the application is built and published as an artifact.
- Deploy: The artifact is downloaded and deployed to the Production environment on Azure.

This keeps our application up to date and ensures it works before deployment.

2.2 Team work

2.2.1 Tasks and Issues

The diagram above illustrates how we managed tasks and issues throughout the development process. At the start of each week, all requirements were broken into separate issues and moved to the "*Todo*" column.

For most issues, the team worked together since tasks were often sequentially dependent. This meant that starting a later issue simultaneously was not feasible. However, when possible, we split the group into two teams to work on independent issues. In such cases, two branches were created, and both teams worked separately to resolve their tasks. Once finished, the branches were merged into one and thoroughly tested to ensure that no conflicts or functionality issues occurred. If all tests passed, the merged branch was integrated into the main branch, and the related issues were moved to "*Done*".

This approach allowed us to maintain stability in the main branch while effectively resolving multiple issues in parallel when the tasks permitted it.

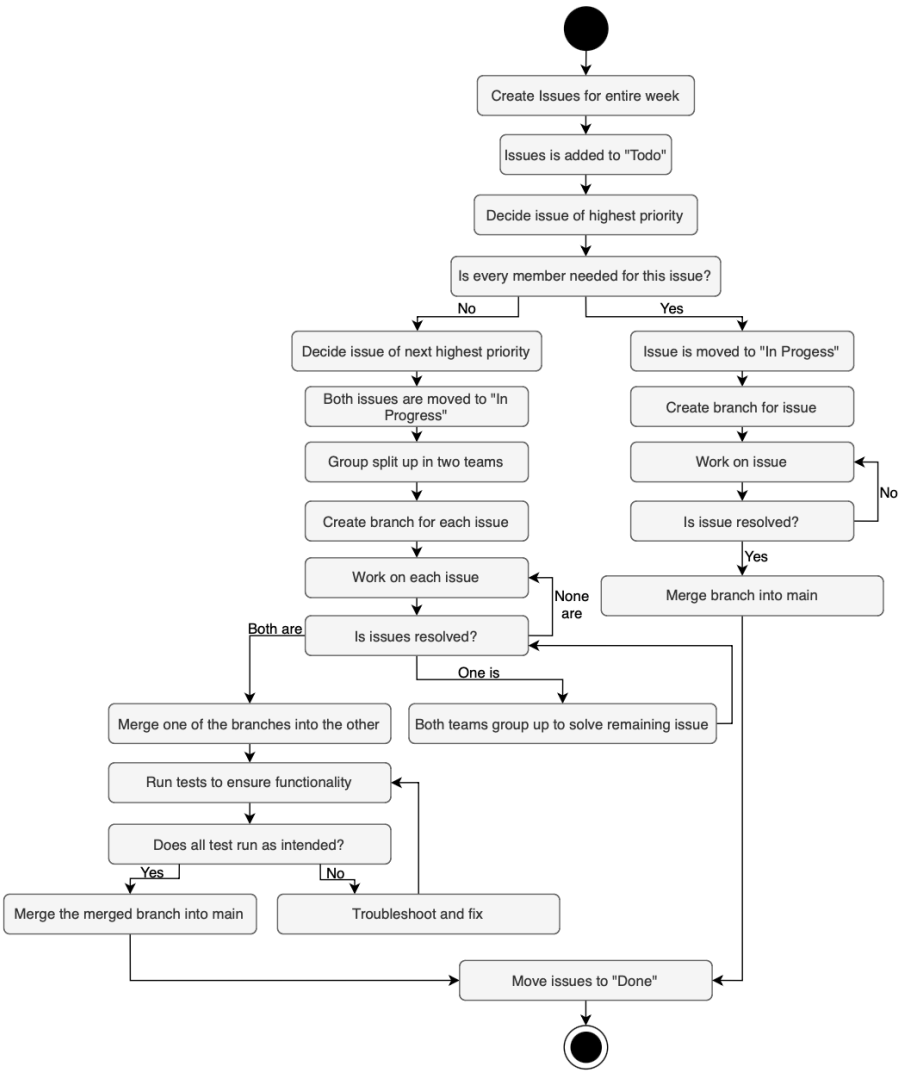

Figure 10: *Team work diagram of our work progress*

2.2.2 Project Board

Project board screenshot from GitHub

Figure 11: *Project board screenshot from GitHub*

The project board screenshot shows its state right before hand-in. The board is divided into four columns: Todo, In Queue/Temporarily Paused, In Progress, and Done.

The empty Todo and In Progress columns reflect that all planned features are complete, and no unresolved tasks remain. This organization ensured a clear workflow and efficient tracking of progress throughout development.