Chirp! Project Report

ITU BDSA 2025 Group 9

Kristine Nielsen krini@itu.dk
Isabella Bjerre Wahl iswa@itu.dk
Nickolaj Toft Carman nitc@itu.dk
Mikkel Dybro Hansen midh@itu.dk
Marcus Alexander Ryssel Mader almm@itu.dk

1 Design and Architecture of *Chirp!*

1.1 Domain model

Here comes a description of our domain model.

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

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

1.2 Architecture — In the small

Our *Chirp!* application is structured according to the *Onion architecture* pattern. In other words, our system architecture is organized into concentric layers, where external dependencies always point inwards, from the outermost layer to the center. The reasoning for using such an architecture design is that it can make the application easier to maintain and test (i.e. it allows for testing, replacing and modifying loosely coupled components separately).

The onion architecture can be divided into the following layers: *domain*, *repository*, *service* and *UI* layer. In our implementation, these layers are reflected in the structure of our code. That is, we have split the application into separate projects and directories, namely *Chirp.Core*, *Chirp.Infrastructure*, and *Chirp.Web*, as well as a separate test directory containing *Chirp.Tests* and *Chirp.PlaywrightTests*.

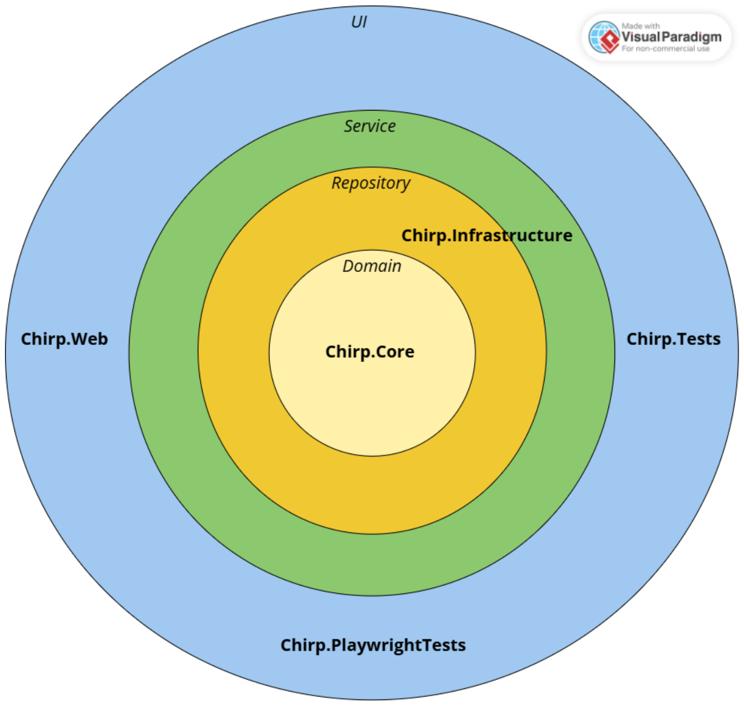


Figure 2: Illustration of the *Chirp!* system architecture as an onion architecture diagram.

The *domain* layer is the innermost layer and contains our domain entities, such as *Author*, *Cheep* and *Comment* - as well as their corresponding DTOs. These do not have any external dependencies, and therefore, act as the inner core, *Chirp.Core*, of our application. The *repository* layer is part of *Chirp.Infrastructure* and is responsible for data access. It handles interaction with the database, including retrieving, storing, and mapping domain data. The *service* layer, also located in *Chirp.Infrastructure*, contains the application services and acts as an intermediary between the UI layer and the repository layer. The UI layer is the outermost layer and is responsible for the user interface, *Chirp.Web*, and contains the testing infrastructure through *Chirp.Tests* and *Chirp.PlaywrightTests*.

The following diagram illustrates this structure as a package diagram, where each layer is represented using a distinct color. Blue represents the *UI* layer, green the *service* layer, orange the *repository* layer, and yellow the *domain* layer. To keep the diagram readable and focused on the architectural structure, not all directories and files are shown. It shows how the code is organized across layers following the onion architecture, omitting individual inner files, such as for example *AuthorService*, *CommentService*, and *CheepService* within the *Service* directory. The same applies to the other innermost directories shown in the diagram.

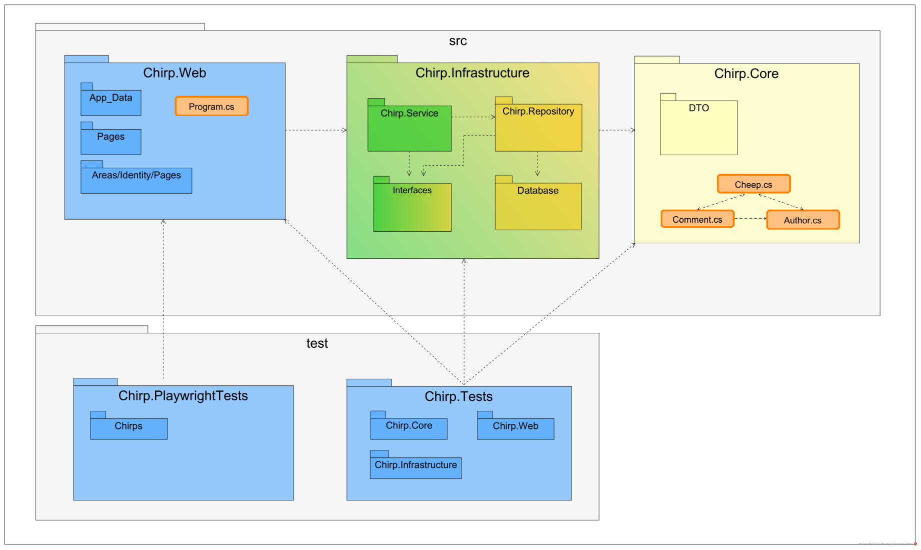
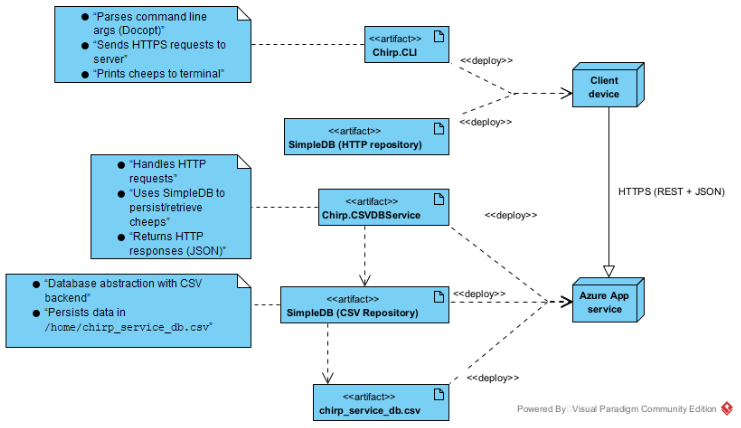


Figure 3: Package diagram illustrating the organization of *Chirp!* across the onion architecture layers.

1.3 Architecture of deployed application



● "Parses command line args (Docopt)"
● "Sends HTTPS requests to server"
● "Prints cheeps to terminal"
<<artifact>> Chirp.CLI
<<deploy>>
Client device
<<deploy>>
<<artifact>> SimpleDB (HTTP repository)
● "Handles HTTP requests"
● "Uses SimpleDB to persist/retrieve cheeps"
● "Returns HTTP responses (JSON)"
<<artifact>> Chirp.CSVDBService
<<deploy>>
HTTPS (REST + JSON)
● "Database abstraction with CSV backend"
● "Persists data in /home/chirp_service_db.csv"
<<artifact>> SimpleDB (CSV Repository)
<<deploy>>
Azure App service
<<deploy>>
<<artifact>> chirp_service_db.csv
Powered By Visual Paradigm Community Edition

Figure 4: Illustration of the *Chirp!* system architecture as an onion architecture diagram.

1.4 User activities

In the following section, three UML activity diagrams illustrate typical user scenarios and the user journey through our *Chirp!* application, starting from a non-authorized user and ending with an authorized user.

The first diagram shows what a user can do when they are *not logged in*. In this state, the user can view the public timeline, view the private timelines of other users by clicking on their usernames or decide to authenticate by viewing the login or register page.

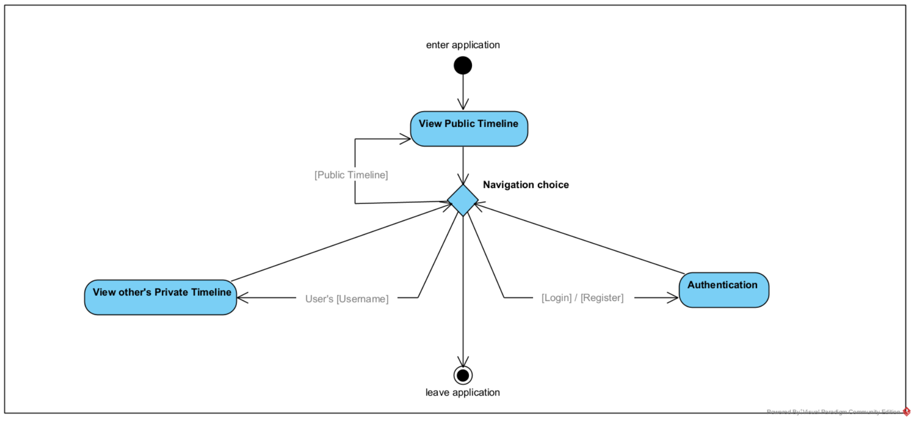


Figure 5: UML activity diagram illustrating the user journey of a non-authorized user through *Chirp!*

If the user decides to log in, they must choose between logging in with an existing account or registering a new one. Both options can be done using either a local account login or an external GitHub login. This is illustrated in the second diagram. After successful authentication, the user is logged in and returned to the public timeline as an authorized user.

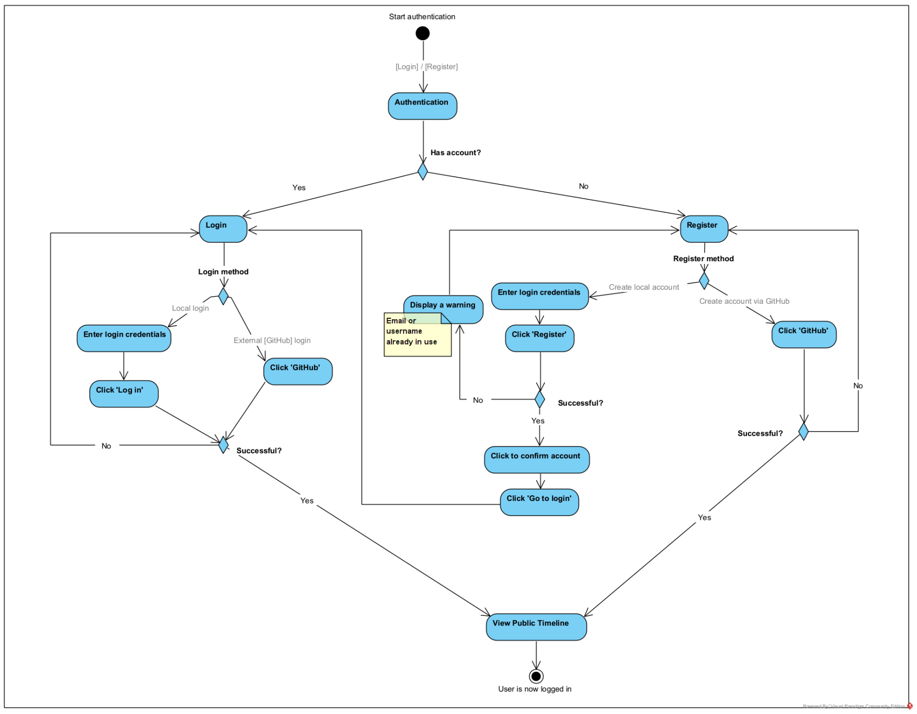


Figure 6: UML activity diagram illustrating the authentication flow in *Chirp!*

The final diagram shows what an authorized user can do, i.e. when the user is *logged in*. An authorized user can view the public timeline, view their own private timeline, and view the private timelines of other users. From each of these, the user can follow or unfollow other users, post or delete cheeps, unfold collapsed comments, comment on cheeps, and delete their own comments. An authorized user can also choose to view their own personal about-me page or log out of the application. From the about-me page, the user can change their profile picture, download their personal information, or delete their account.

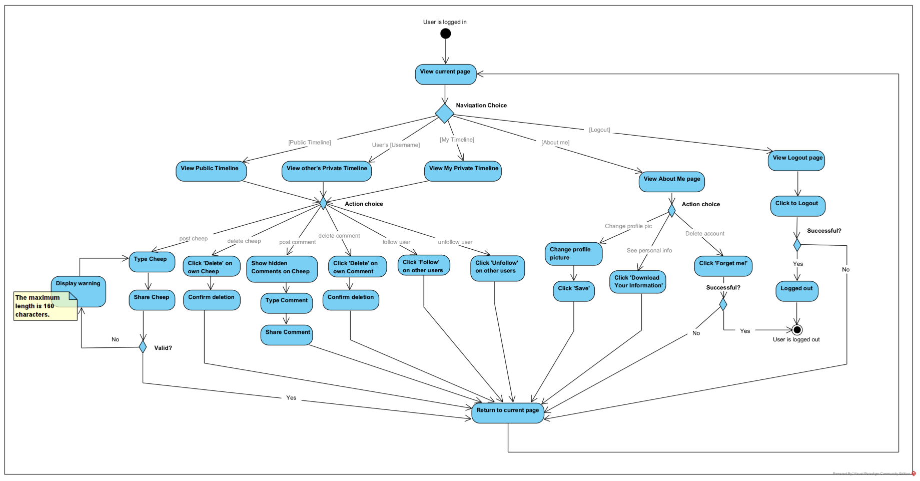


Figure 7: UML activity diagram illustrating the user journey of an authorized user in *Chirp!*

To keep this diagram simple and readable, the action "*Return to current page*" represents that, after completing an activity, the user is returned to the page they were previously viewing. For example, if a user posts a cheep while viewing their private timeline, they will stay / return to their private timeline once the action is done.

1.5 Sequence of functionality/calls through *Chirp!*

2 Process

2.1 Build, test, release, and deployment

This UML activity diagram shows the process of building, testing, releasing, and deploying *Chirp!* to our GitHub and Azure Web App. In our project, we have three distinct workflows: a *release* workflow for publishing the application, a *test* workflow for building and testing the application, and a *deploy* workflow for deploying the application to Azure.

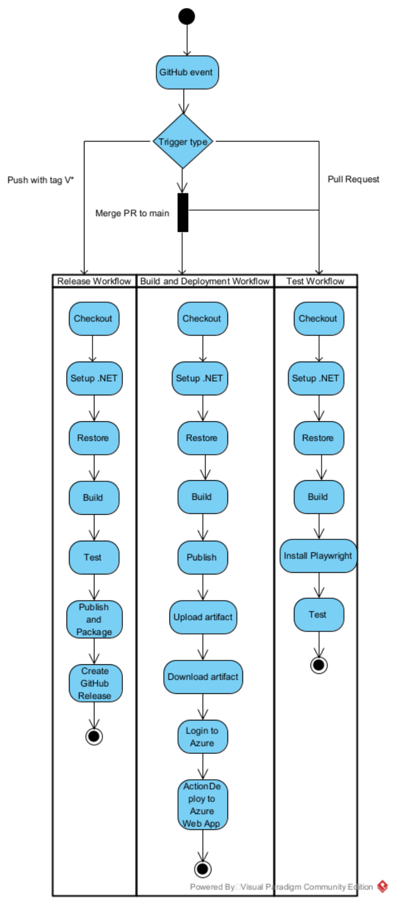


Figure 8: Illustration of the *Chirp!* event process as UML class diagram.

The workflow that is triggered depends on the type of action performed in the repository. A push with a tag named `v*`, where `*` represents the release version (e.g., `v1.0.1`), triggers the *release* workflow. Creating a pull request triggers the *test* workflow, while accepting and merging a pull request into the main branch triggers both the *test* and *deploy* workflows.

Once triggered, each workflow executes its own set of straightforward steps and does not depend on the others. All workflows follow a similar structure, where they set up, restore and build the project, before handling their respective independent tasks.

2.2 Team work

2.3 How to make *Chirp!* work locally

2.4 How to run test suite locally

3 Ethics

3.1 License

We chose an MIT License for our project. An MIT License leaves many opportunities for others to use our project, while still giving us credit for our work without creating any obligations towards us. We felt this fit the project best, as it matches the context in which the project was developed. That is, it was created in an educational context, and we therefore, as a group, believe that others also should be allowed to use, read and learn from our code. The license is simple, permissive, and compatible with the dependencies in our *.csproj* files. For these reasons, we consider the MIT License to be the best fit for this project, whereas a more restrictive license would likely be more appropriate for an actual commercial product or application.

3.2 LLMs, ChatGPT, CoPilot, and others

During the development of our project, Large Language Models (LLMs) were used for various parts of the project, primarily ChatGPT. Although LLMs were used throughout the project, we were not always fully transparent about their use and did not consistently add, for instance, ChatGPT as a co-author to all commits, where it was actually used. The primary reason for this was that at the beginning of the project, we were not aware that this was required. Once we were informed and reminded of this, we began to apply it to our commits. However, there were still instances where we simply forgot to add it as a co-author, particularly when the use of LLMs was minimal and not a significant part of the changes in a commit.

The primary use of LLMs during our project was for error handling and debugging. There were many situations in which we encountered that changes to the code caused significant parts of the system to break. As a result, we had to identify where the issues had occurred, and how to resolve them, sometimes while dealing with a large number of exceptions simultaneously. This was not always straightforward, and LLMs were, therefore, particularly helpful in such situations. In these cases, we would, for instance, provide ChatGPT with the broken code and the associated exceptions and prompt it to suggest how to handle these.

This was for the most part useful, when frustration had begun to build, and we could not resolve the issue on our own, as it often provided us with insight into the mistakes we had made. In many instances, the underlying problems were caused by simple or easily overlooked coding mistakes. Therefore, the LLMs offered us an additional perspective that helped us resolve these issues efficiently.

A second use of LLMs was for direct code generation. This was, however, used in moderation and primarily for inspiration and guidance on more confusing or challenging tasks, allowing us to incorporate parts of the generated code while still implementing the final solution properly on our own. Lastly, we also decided to use ChatGPT for image generation, creating the profile pictures in our project that depict different colored birds. We chose this, because we wanted images that fit the bird-themed concept of *Chirp!* while maintaining a clean user interface design, and since none of us had the necessary skills or time to create these images ourselves, using ChatGPT provided us with a fast and easy solution.

The advantage of LLMs is that they provide a useful tool when one is uncertain or stuck, offering guidance on how to handle it, and since they are always available, they can help more quickly than, for instance, a teaching assistant, who is limited by time and day. Therefore, LLMs can in most cases speed up the development.

However, LLMs can sometimes also be misleading. We found that, since the LLM lacks full knowledge of the project that only we as the developers possess, it occasionally provided incorrect guidance, which slowed our development down rather than speed it up. A significant example of this occurred during the deployment of our application to Azure. For a long time, we believed that our GitHub Action workflow was the reason the application would not deploy automatically and display the new content, after integrating the EF Core database. Following guidance from ChatGPT, we spent considerable time adjusting the workflow, only to later discover that the actual issue was a missing startup command and an outdated database in Azure (See GitHub Issue #24).

In conclusion, we found that LLMs, when used appropriately, can significantly speed up the development process. However, they should be used in moderation, and while convenient, they can also introduce inefficiencies if completely and uncritically relied upon.