# Chirp!

Project Report

Tore Engberg Timothy Lubowa Muneeb Hussain
Nikolai Li Jakob Juhl

2026-01-02

## Contents



# Design and Architecture of _Chirp!

## Domain model

The domain model diagram illustrates the data structure of the Chirp application. It focuses on the relationships between the core entities: Author, Cheep, and Like, showing how users create messages and interact with them. Additionally, the diagram demonstrates the integration with ASP.NET Identity, which handles the GitHub authentication. This connection links the login data to the Author

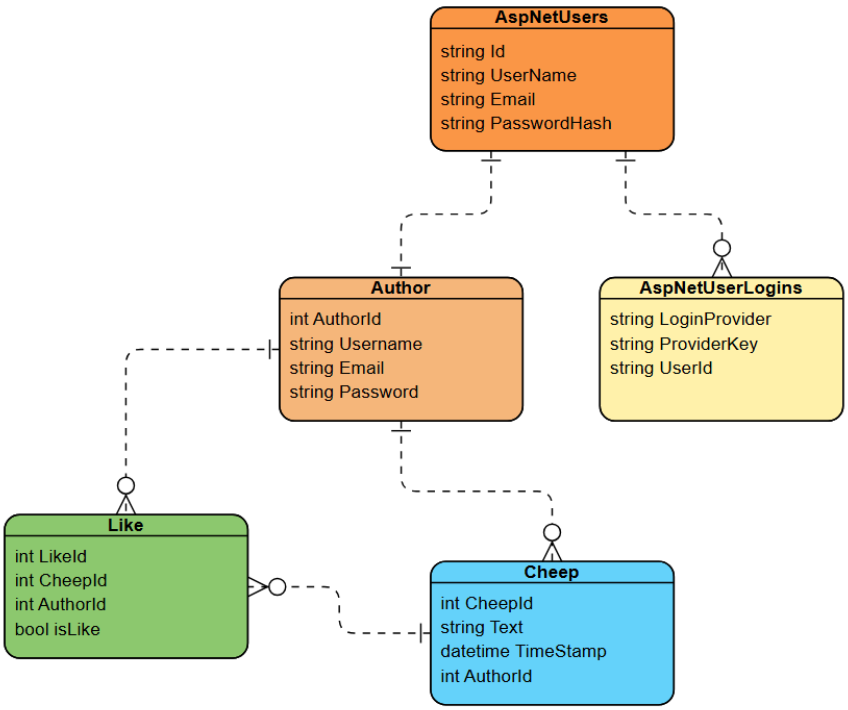



Figure 1: This figure shows how our data is structured. It links the core entities (Author, Cheep, and Like) with the ASP.NET Identity system used for authentication.

profile, ensuring that user identity is managed efficiently while keeping security separate from the application's logic.

### Architecture — In the small

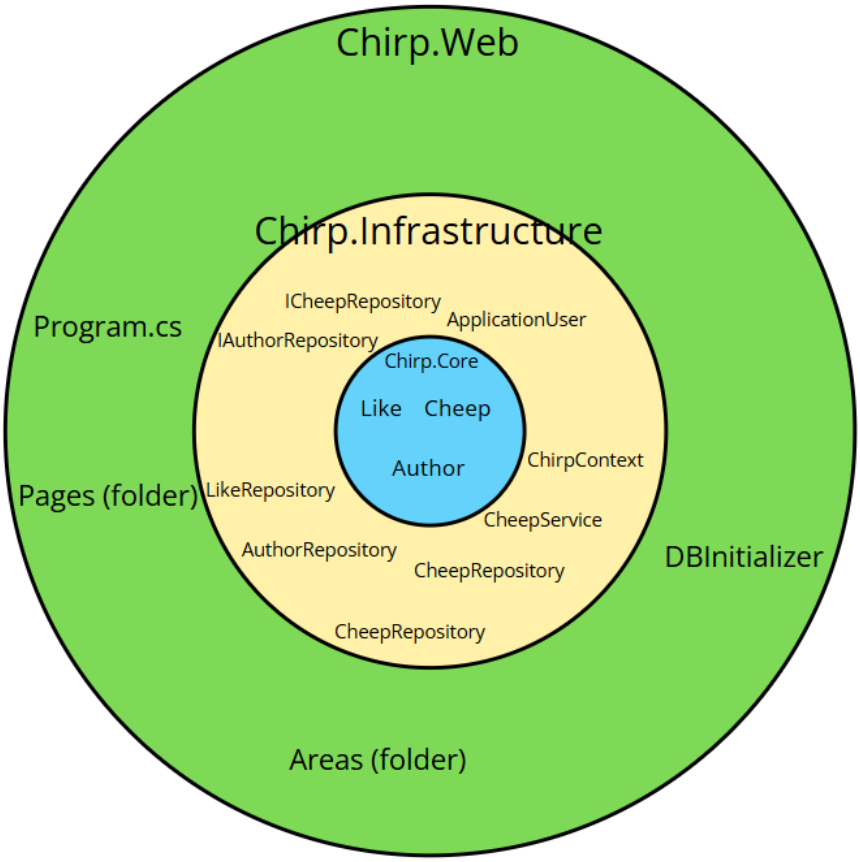



Figure 2: This figure shows how our Model is structured. Its structured after the union architecture model

This model illustrates the architecture of our chirp application which is constructed after the principals of onion architecture. The system is devided by three layers. Chirp.Core, Chirp.Web, Chirp.Infrastructure. Det outer most layer Chirp.Web consists of the visual that the applications presents. The middle layer Chirp.Infrastructure is responsible for the data flow, services and integration with the database, and the inner most layer Chirp.Core is responsible for the domain models and logik for the application. Each layer is independent of its outer layers

### Architecture of deployed application

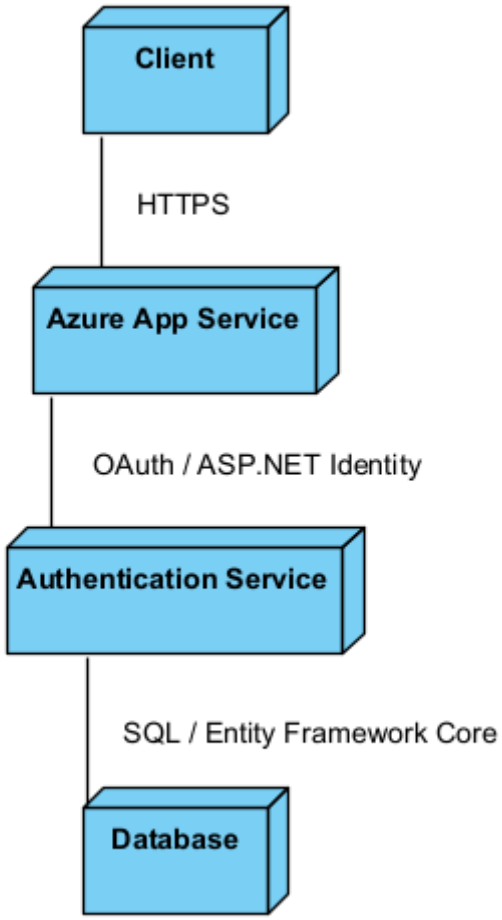



Figure 3: ASP.NET flow diagram

The application is built using ASP.NET Core and is deployed as a web application on Azure App Service. ASP.NET Core provides the framework and built-in templates, including Razor Pages for server-side page rendering and authentication services for user login and access control.

Clients interact with the system through a web browser using HTTPS, requesting Razor Pages that are processed on the server. The Azure App Service handles authentication requests via the authentication service and accesses persistent application data through a database using Entity Framework Core.

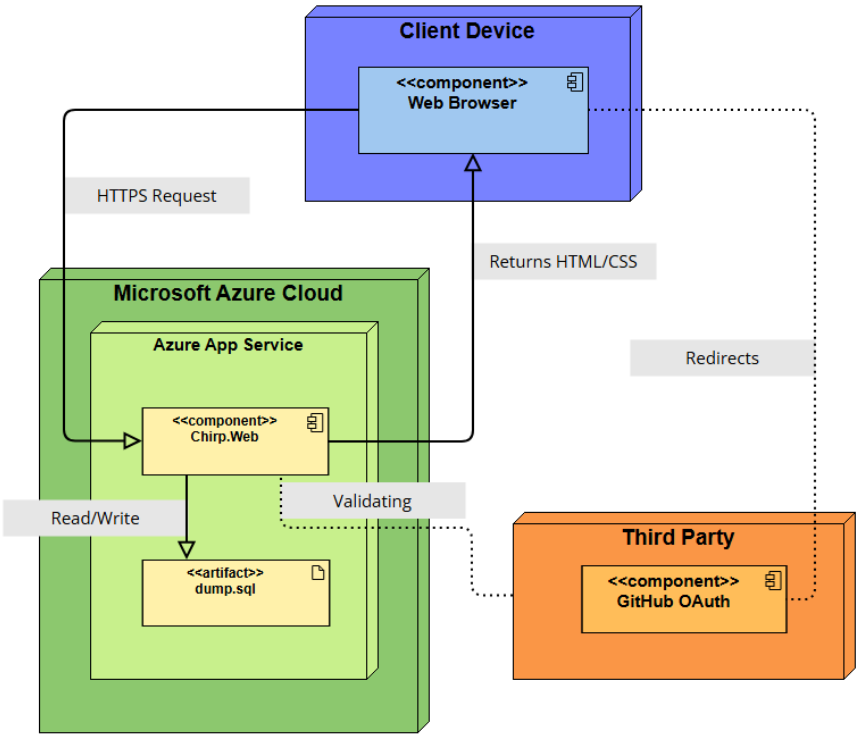
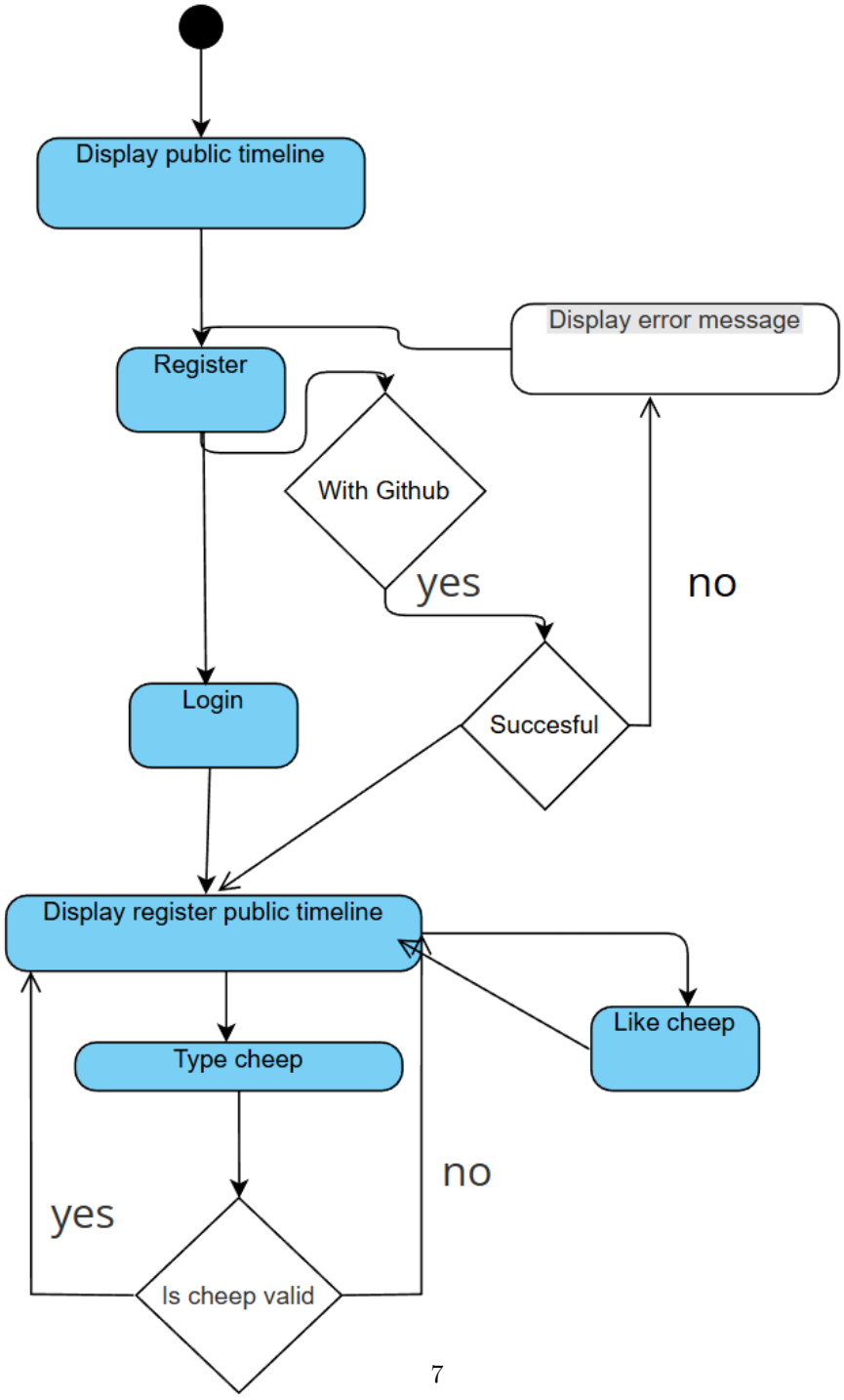



Figure 4: The diagram illustrates the deployment architecture on Microsoft Azure.

The diagram illustrates the deployment architecture on Microsoft Azure. Users access the application via a web browser, which connects securely to the Azure App Service using HTTPS. Inside the server, the application logic runs alongside a local dump.sql file, which is used for data storage instead of an external database server. Finally, GitHub is integrated as a third-party service to handle user authentication.

## User activities

### Register, Cheep and Like






A non authorized user first sees the public timeline. From this page, the user can

choose to register and afterwards log in. During registration, the user can choose to register using GitHub. If the registration is not successful, an error message is displayed. If the registration is successful, the user can log in. After logging in, the user is shown the registered public timeline. At this stage, the user can type a cheep and like cheeps from other users. When a cheep is typed, the system checks if it is valid. If the cheep is valid, it is posted and displayed on the timeline. If the cheep is not valid, it is not posted and the user stays on the timeline. While on the timeline A users can choose to like a cheep.

**Follow and Unfollow**

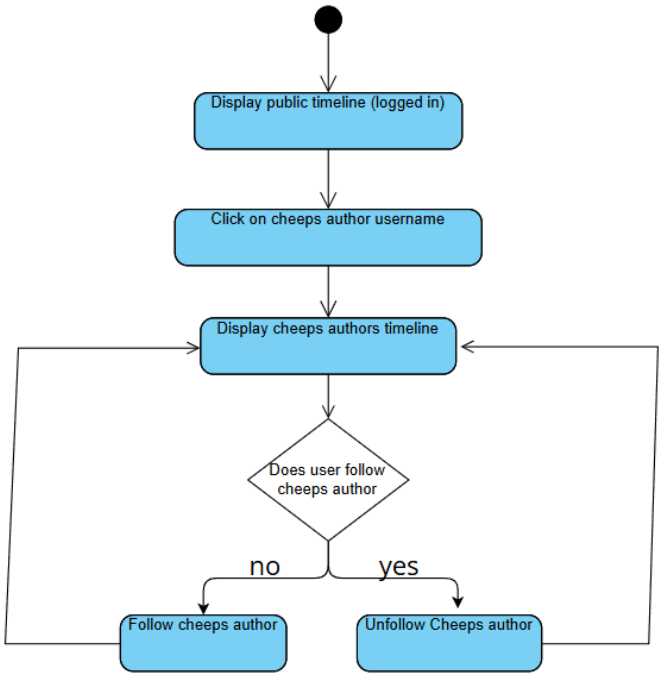


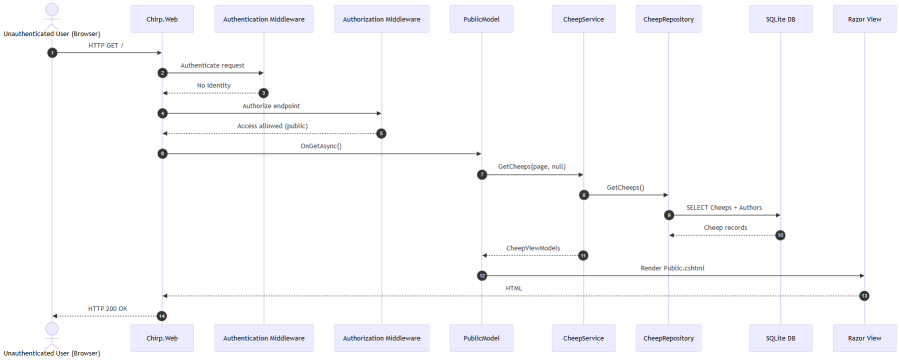

Figure 5: The diagram illustrates the Road in which a user can follow and unfollow other users.

The user starts by viewing the public timeline while being authenticated. From the timeline, the user can click on a cheep author's username, which opens the author's personal timeline. The system then checks whether the user already follows this author. If the user does not follow the author, the user can choose to follow them. If the user already follows the author, the user can choose to unfollow them. After following or unfollowing, the user remains on the author's

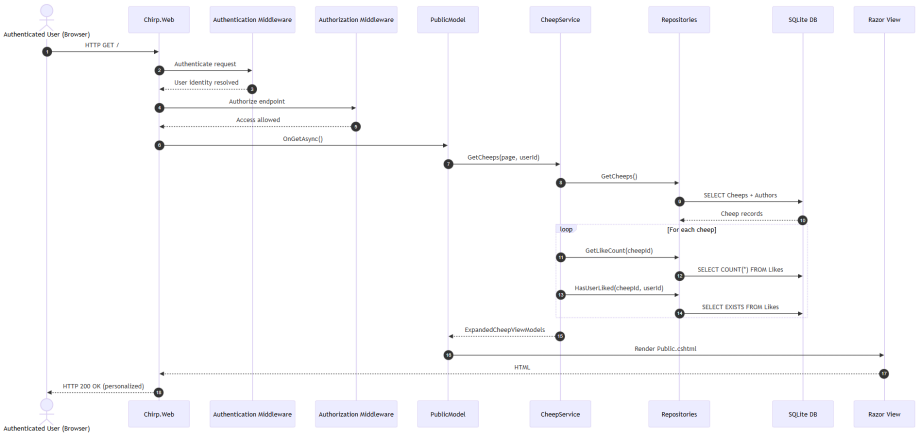
timeline and can continue browsing cheeps. ## Sequence of functionality/calls through _Chirp! For our web application the sequence of data flow shares an overall arhictectural structure. The main difference occurs when two types of users try to make requests to the application. Unauthenticated users, who are not logged will get, and authenticated users, who are logged in. Both requests pass through the ASP.NET core pipeline, to return a rendered Razor view to the user client. When no authentication identity can be resolved, the client is served a lightweight read-only site. In contrast, when an authentication is resolved, the user is served a more enriched experience.

### HTTP Request made by an Unauthorized user

This sequence diagram shows an unauthenticated user requesting to access the Chirp web application. When a HTTPS request is made to the Chirp timeline, the request is directed through ASP.NET Core middleware pipeline, into the web layer, through the services and infrastructure layers, in the end returning the HTML page. The middleware will check if the user is logged in, sicne they are not, they will only gain access to the public timeline - which does not require any authentication.




### HTTP Request made by an authorized user This sequence diagram shows an authenthicated user requesting to access the Chirp web application. The core pipeline for sequences of data during requests, is mostly identical when the user is authenticated. Though when the user is authenticated, the users auth token are checked, and is confirmed to access the endpoints. This in turns changes how the website looks for the user, e.g. the authenticated user will see a text box to post cheeps. In the devlogs we see that a query is made against all authors (Authenticated users), then user specific checks are made, such as their likes and follows, for their own custom timeline - Data which is unaccessible if the user is not authenticated.




# Process

## Build, test, release, and deployment

The activity diagram illustrates the three automated workflows in GitHub Actions. The first two columns show the Continuous Integration and Deployment processes, where code is automatically tested with Playwright and deployed to Azure when files are changed. The third column shows the release workflow, which builds and publishes the tool whenever a new version tag is pushed.

## Team work

Effective teamwork is crucial for managing our project efficiently. This section outlines our collaborative workflow, focusing on how we organize tasks using Project Boards and how we handle development through our Activity Flow. These processes help us maintain structure, ensuring that every new feature is properly implemented, reviewed, and integrated into the application.

### Project board

A project board was created for each project session. This screenshot illustrates the different phases that issues progressed through before being marked as complete for Sessions 1–5. From Session 6 onward, separate project boards were created for each session. Perhaps this made things unclear and incoherent and we should have just used a single project board for all issues.

### Activity flow

The workflow activities start with the weekly lectures, where new material is introduced. Next, the README_PROJECT.md file provides descriptions of how the material is to be implemented. From there, issues are created for each

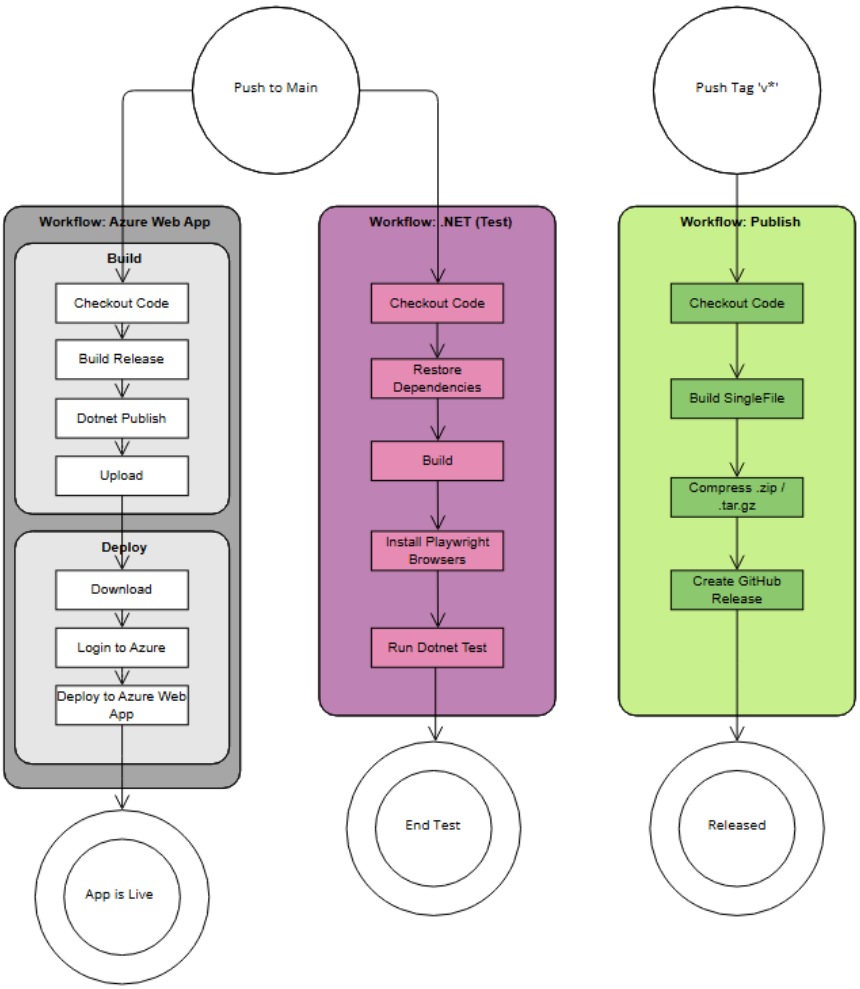



Figure 6: The activity diagram illustrates the three automated workflows in GitHub Actions.

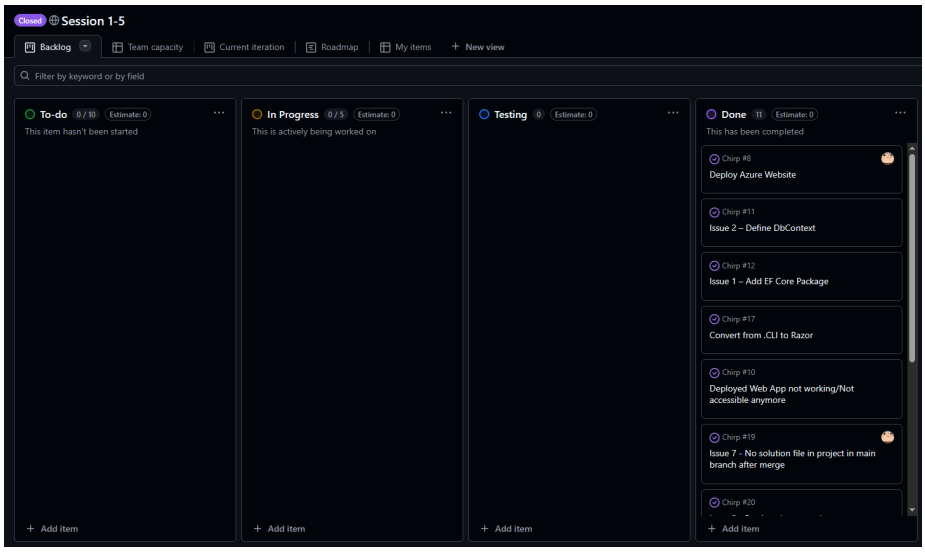



Figure 7: project board screenshot

new feature described in the project. These issues are assigned to members, who branch out from the main branch to implement their respective tasks. Once an issue is implemented, a pull request is created for other members to review the new feature's code. If the code is valid and all tests pass, the pull request can be closed and merged into the main branch, and the old feature branch can be deleted. ## How to make *Chirp!* work locally Requirements to run *Chirp!* locally: * .NET 8.0.x SDK * .NETCore 8.0.x Runtime * AspNetCore 8.0.x Runtime

### Download and run

1) .NET 8 can be downloaded from the release page releases page.
2) Extract the zip file.
3) Open the extracted folder in the terminal.
4) Run executable.

### Building from source

1) Clone the repository

```
git clone (https://github.com/ITU-BDSA2025-GROUP22/Chirp.git)
```

2) Change directory to the repository

```
cd Chirp
```

3) Run the project

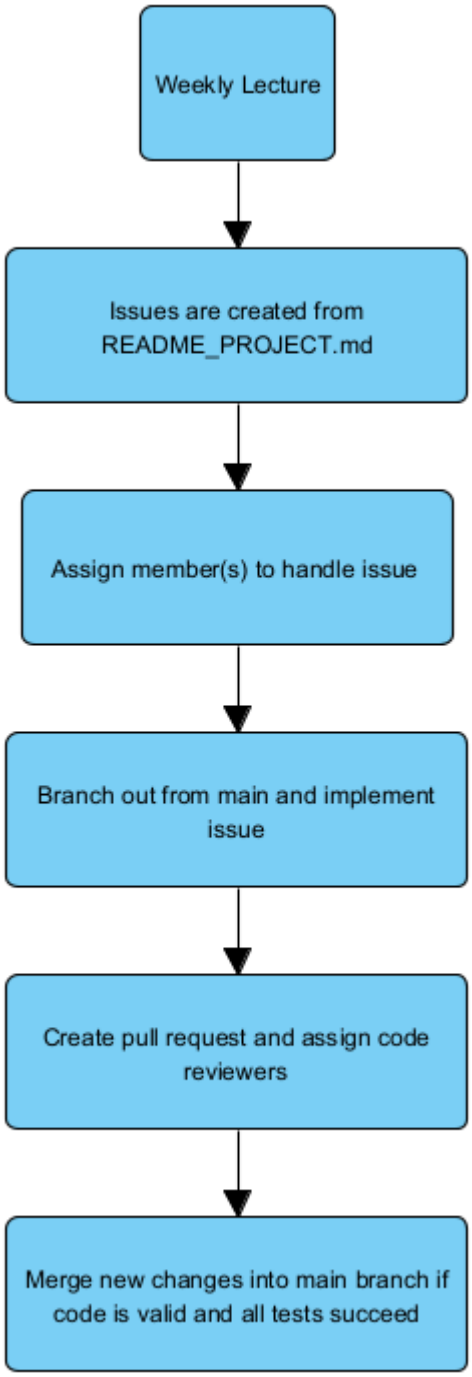



Figure 8: activity flow diagram

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

## How to run test suite locally

After completing the previous steps you can preceed by with the following

```
dotnet test
```

This will run every test using in-memory Sqlite databases. # Ethics

## License

We use the MIT license ## LLMs, ChatGPT, Gemini, and others ChatGPT and Gemini was used minimally, and no production code was copied directly from them. It was mainly used to clarify concepts and interpret error messages. Github copilot was also used for the same purpose of explaining error messages related to workflows. Stackoverflow and the official documentation often proved more efficient and reliable than AI suggestions since they have a very narrow context of the project.