Project **Chirp!**

BDSA Group 17

Burak Özdemir buoe@itu.dk
Hanan Choudhary Hadayat abha@itu.dk
Joshua James Medilo Calba jcal@itu.dk
Julius Dalsgaard Bertelsen jube@itu.dk
Tan Dang tanda@itu.dk

IT UNIVERSITY OF COPENHAGEN

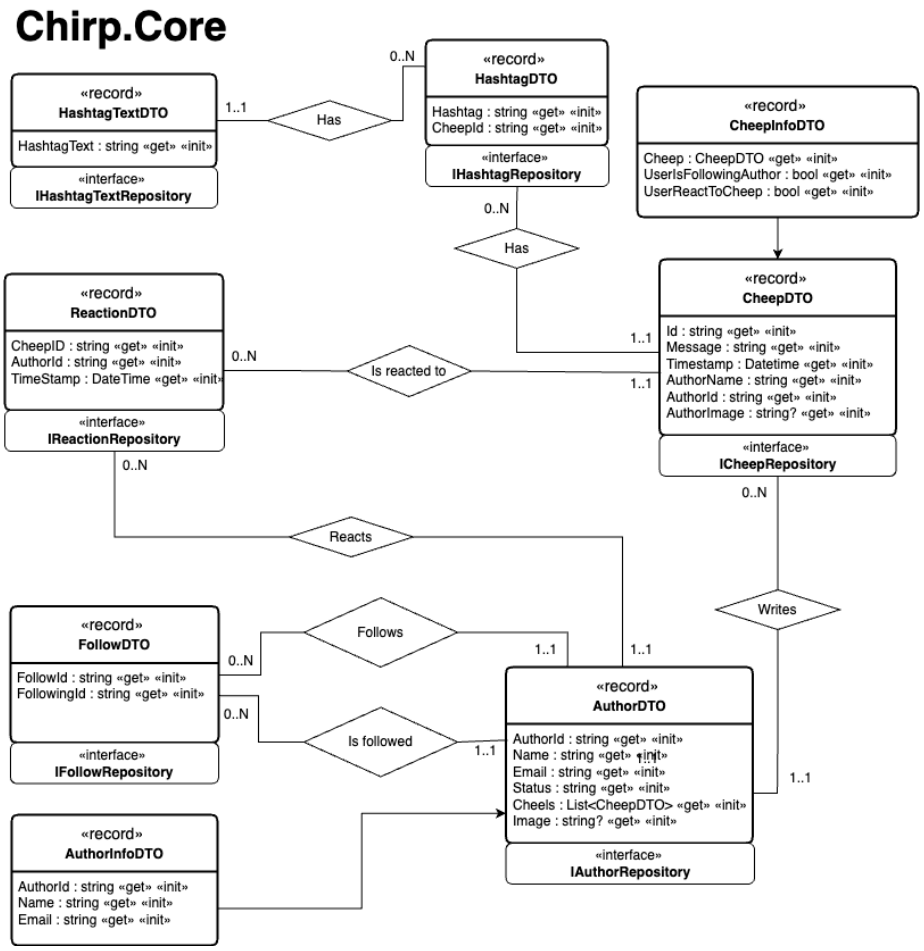
Course code	BSANDSA1KU
Name of course	Analysis, Design and Software Architecture
Course manager	Helge Pfeiffer - ropf@itu.dk
Project title	*Chirp!*
Group number	17
Date	December 21 2023
School	IT-University of Copenhagen

1 Design and architecture

1.1 Domain model

The domain model reflects the entities and their relationship of the Chirp application, as well as the interfaces of the repositories that allow for accessing and manipulating these different entities and their related data. These entities form the foundation of the business logic of the application.

The diagram below shows the domain model of the Chirp application.



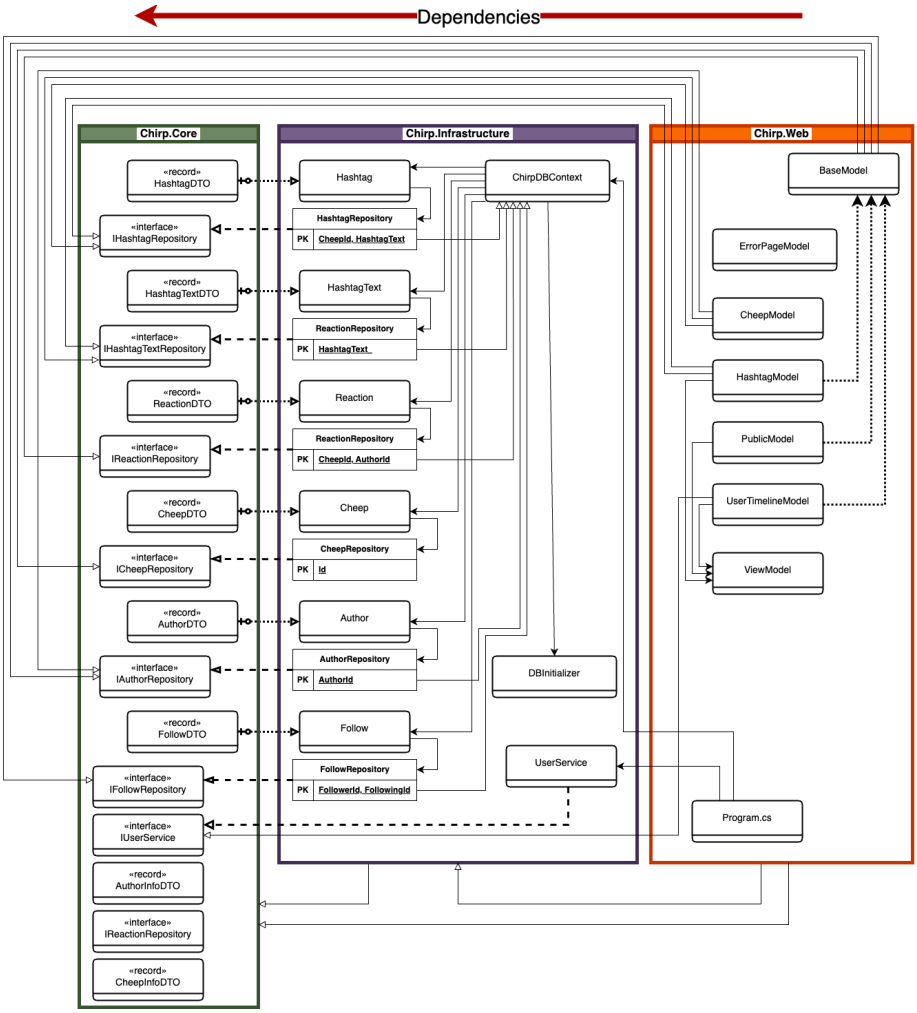
1.2 Architecture — In the small

The application Chirp has been designed with Onion architecture in mind.

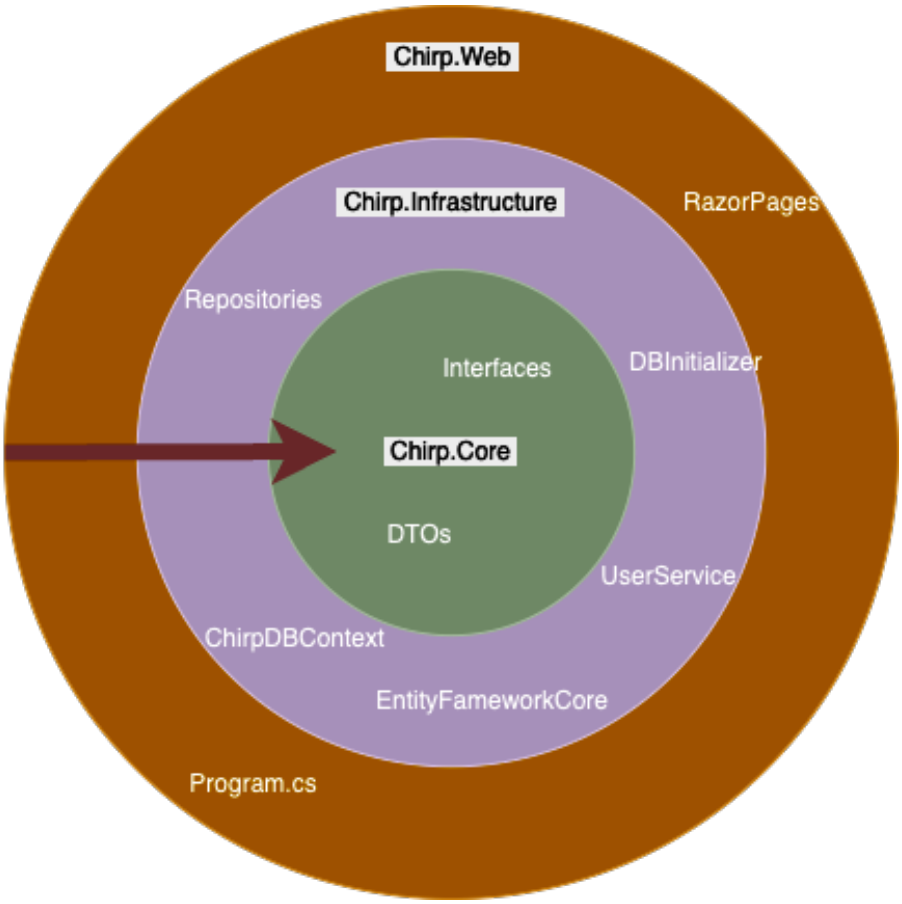
The diagram below shows the organization of the code of the Chirp application, showcasing the relationships and dependencies of different components of the program, highlighting how the code of the program is organised into different layers of the onion architecture design.

The diagram shows the inward flow of dependencies in accordance with Onion architecture, so that the inner layers remain independent of external dependencies.

Some relationships have been omitted to improve the readability of the diagram.



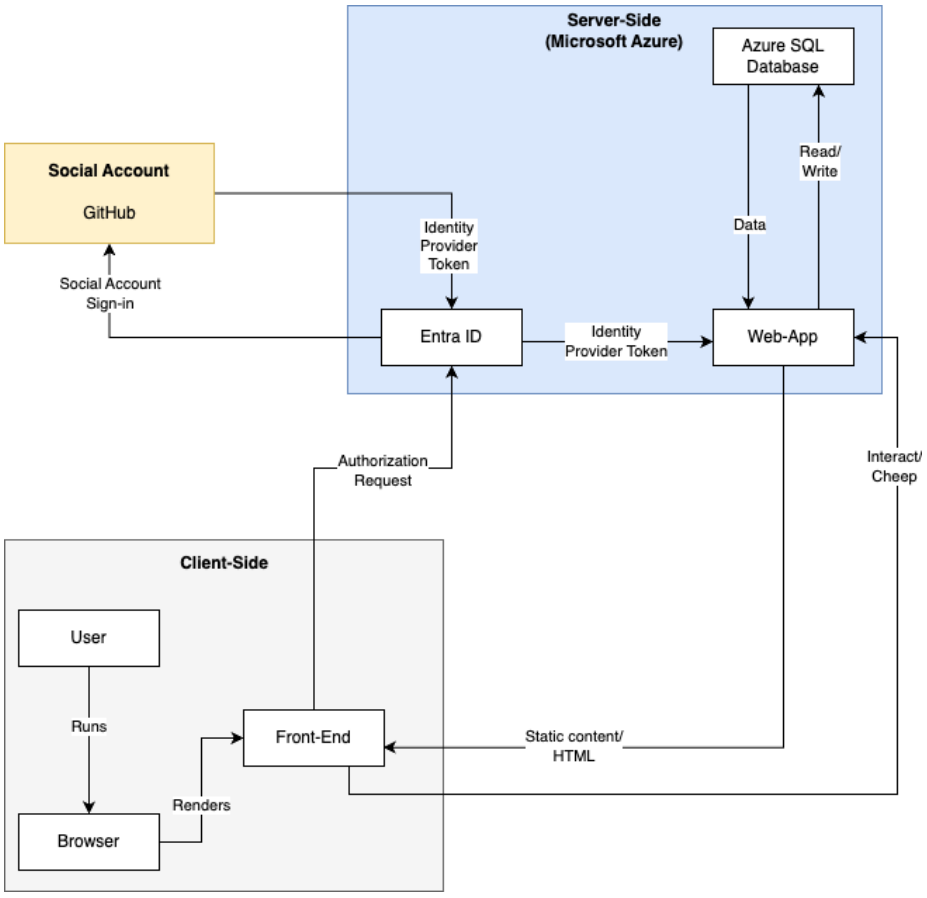
The figure below shows a more simplistic view of the onion architecture structure of the components of the application.



1.3 Architecture of deployed application

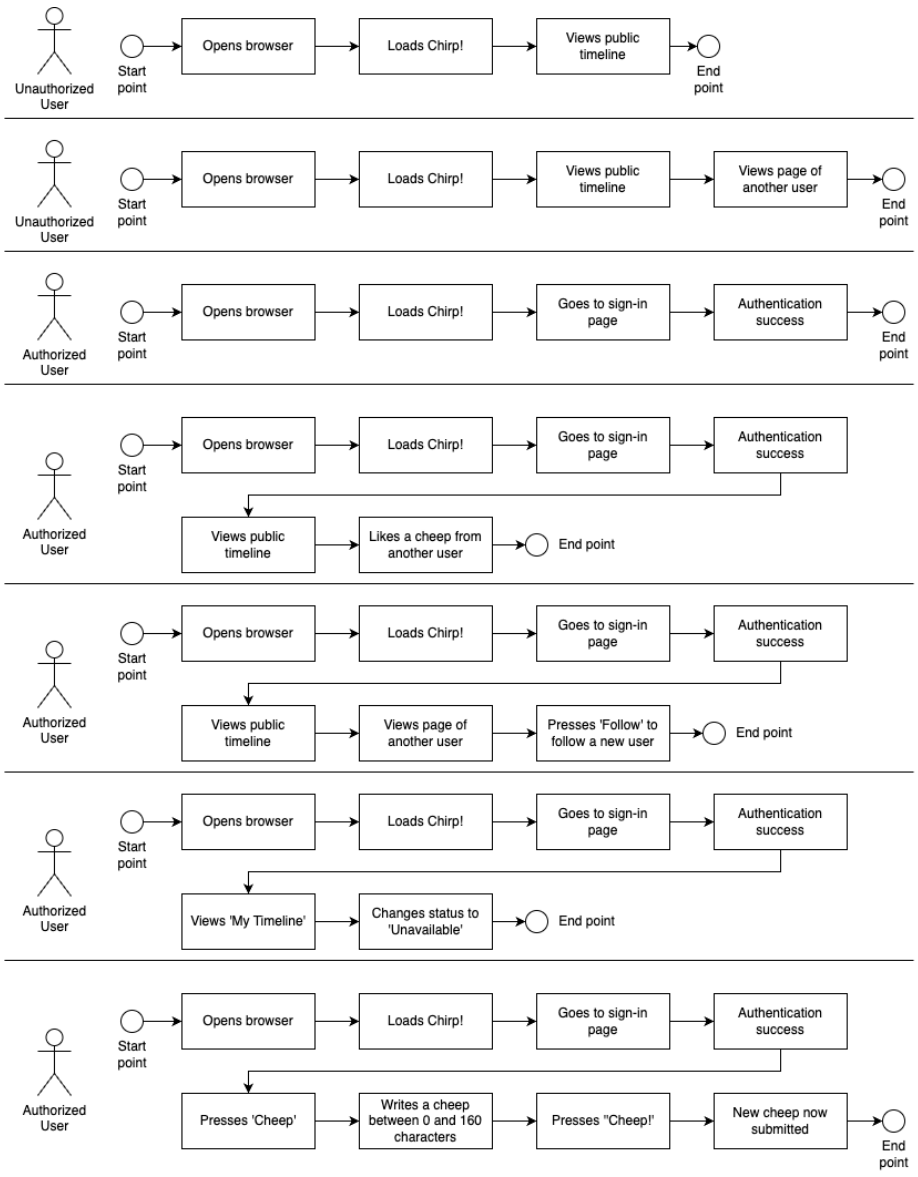
The Chirp application is deployed on Microsoft Azure, utilizing Azure services with an Azure SQL database.

The diagram below shows the interaction between the client component with a user interface allowing the user to make requests and the application deployed on Microsoft Azure as the server component. The diagram also shows the interaction of this deployed application with the Azure SQL Database, as well as signing in through a social account (GitHub).

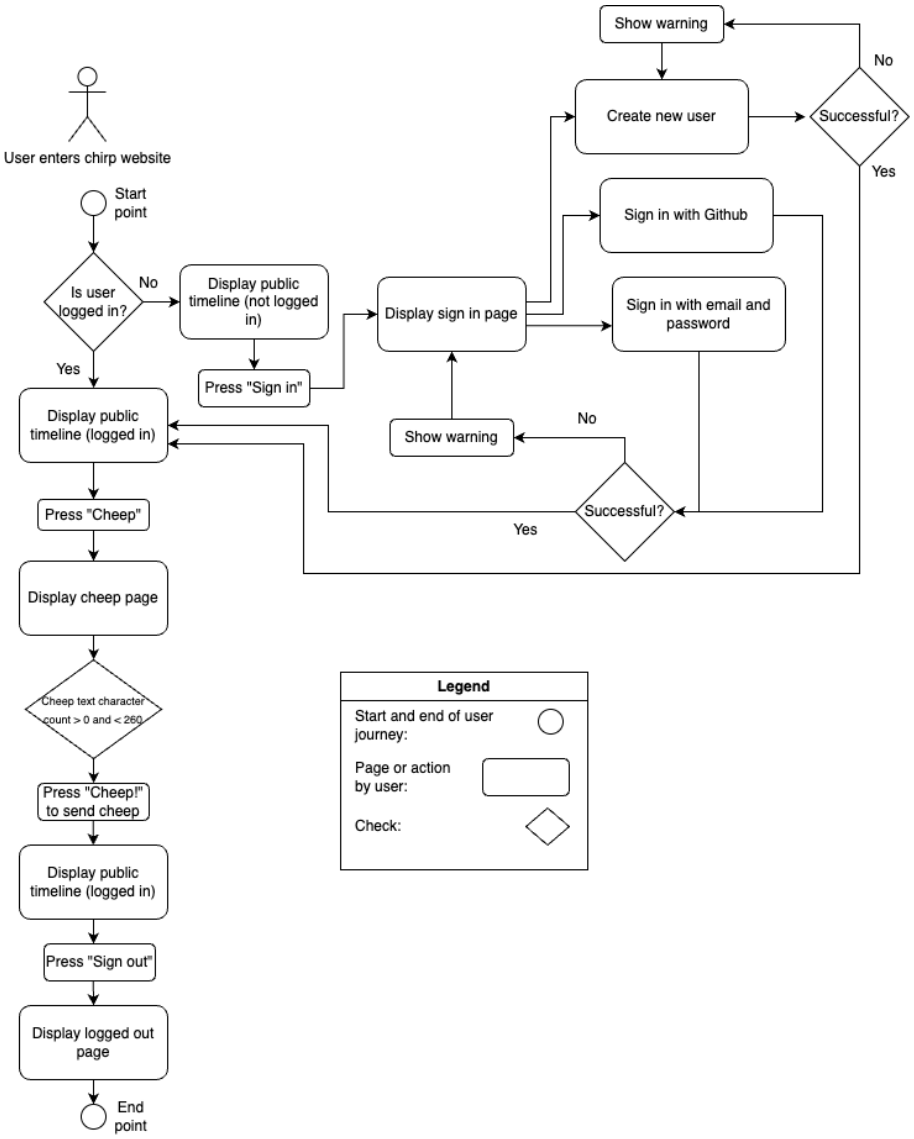


1.4 User activities

The diagram below shows a series of typical user activities through the Chirp application while remaining unauthorized and after logging in and becoming authorized.

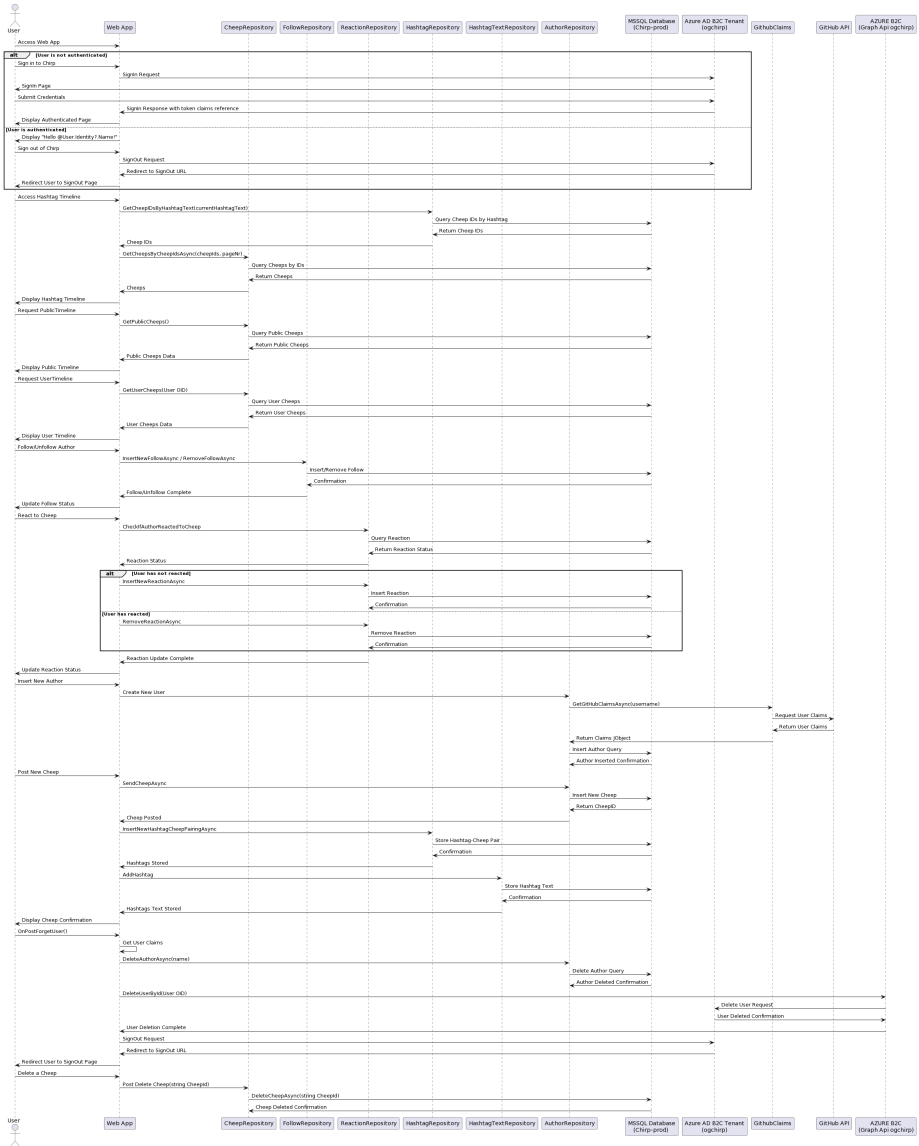


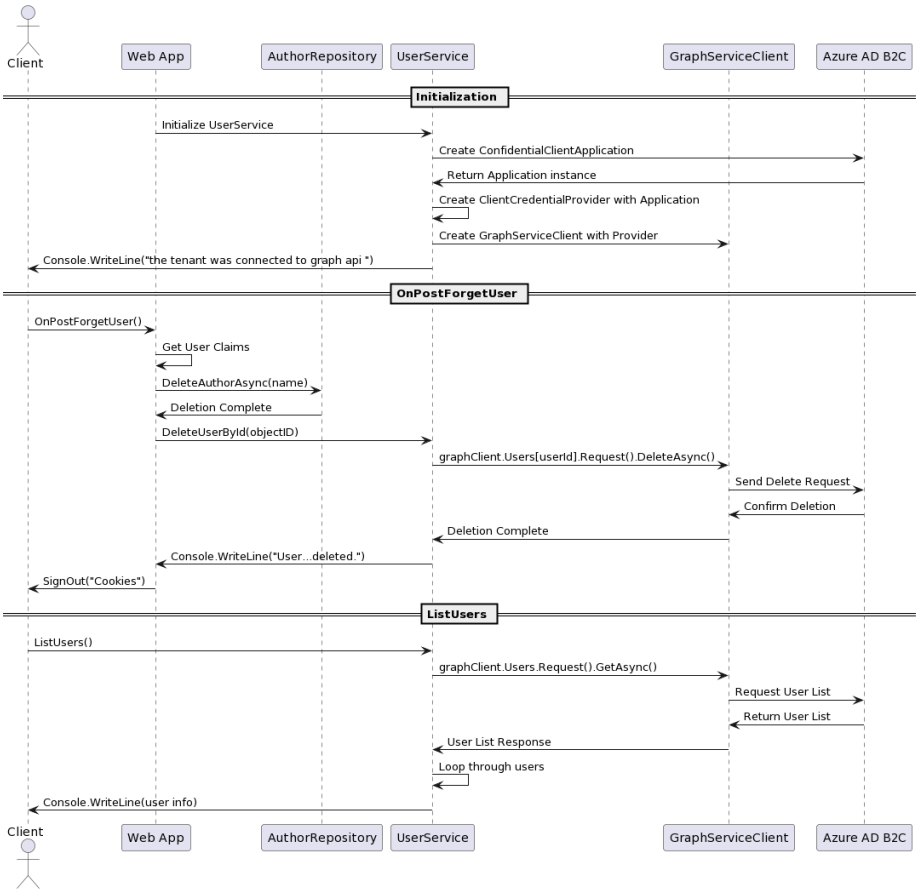
The diagram below shows a slightly more detailed view of possible scenarios of a user journey, in which a user enters the chirp website, logs in or creates a profile if necessary, sends a cheep, and then logs out.



1.5 Sequence of functionality/calls trough *Chirp!*

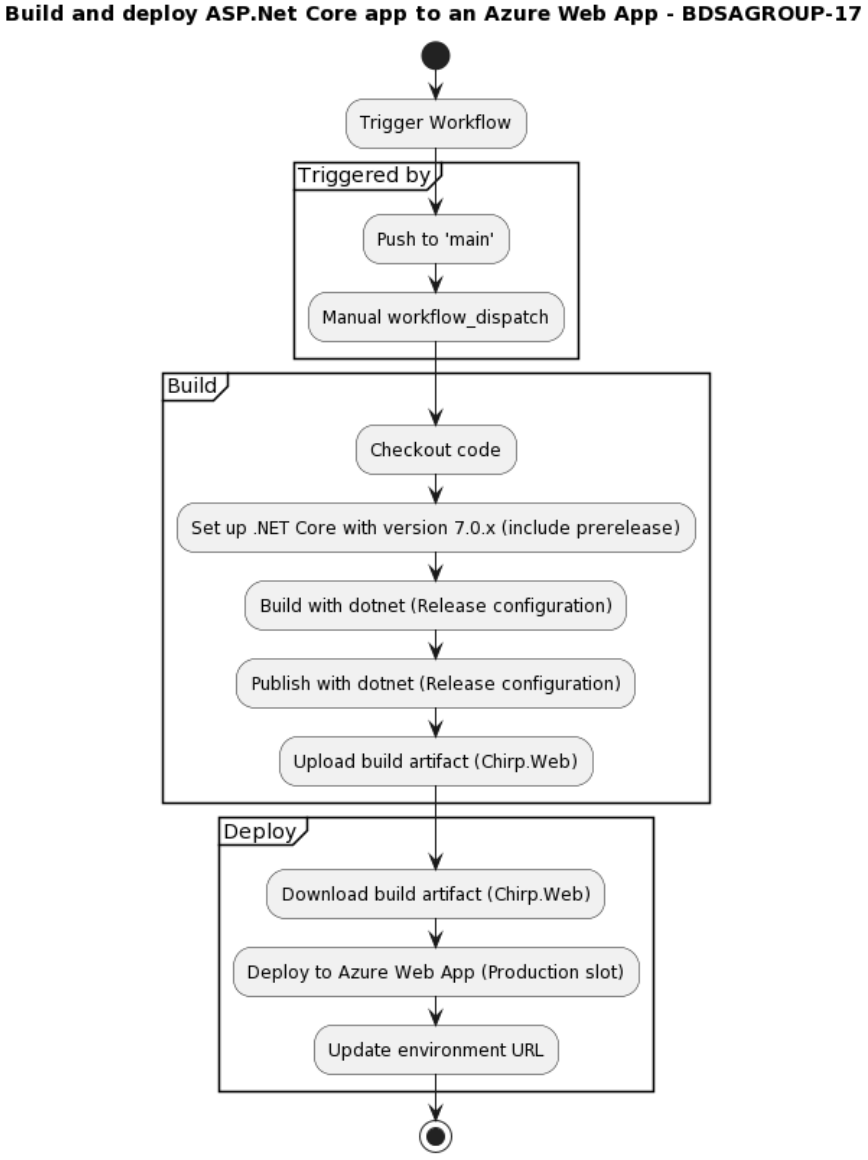
The diagrams below illustrates the flow of messages and data through the Chirp application.


User
Web App
CheepRepository
FollowRepository
ReactionRepository
HashtagRepository
HashtagTextRepository
AuthorRepository
MSSQL Database (Chirp-prod)
Azure AD B2C Tenant (ogchirp)
GitHubClaims
GitHub API
AZURE B2C (Graph Api ogchirp)
Access Web App
alt
[User is not authenticated]
Sign in to Chirp
SignIn Request
SignIn Page
Submit Credentials
SignIn Response with token claims reference
Display Authenticated Page
[User is authenticated]
Display "Hello @User.Identity?.Name!"
Sign out of Chirp
SignOut Request
Redirect to SignOut URL
Redirect User to SignOut Page
Access Hashtag Timeline
GetCheepIDsByHashtagText(currentHashtagText)
Query Cheep IDs by Hashtag
Return Cheep IDs
Cheep IDs
GetCheepsByCheepIdsAsync(cheepIds, pageNr)
Query Cheeps by IDs
Return Cheeps
Cheeps
Display Hashtag Timeline
Request PublicTimeline
GetPublicCheeps()
Query Public Cheeps
Return Public Cheeps
Public Cheeps Data
Display Public Timeline
Request UserTimeline
GetUserCheeps(User OID)
Query User Cheeps
Return User Cheeps
User Cheeps Data
Display User Timeline
Follow/Unfollow Author
InsertNewFollowAsync / RemoveFollowAsync
Insert/Remove Follow
Confirmation
Follow/Unfollow Complete
Update Follow Status
React to Cheep
CheckIfAuthorReactedToCheep
Query Reaction
Return Reaction Status
Reaction Status
alt
[User has not reacted]
InsertNewReactionAsync
Insert Reaction
Confirmation
[User has reacted]
RemoveReactionAsync
Remove Reaction
Confirmation
Reaction Update Complete
Update Reaction Status
Insert New Author
Create New User
GetGitHubClaimsAsync(username)
Request User Claims
Return User Claims
Return Claims JObject
Insert Author Query
Author Inserted Confirmation
Post New Cheep
SendCheepAsync
Insert New Cheep
Return CheepID
Cheep Posted
InsertNewHashtagCheepPairingAsync
Store Hashtag-Cheep Pair
Confirmation
Hashtags Stored
AddHashtag
Store Hashtag Text
Confirmation
Hashtags Text Stored
Display Cheep Confirmation
OnPostForgetUser()
Get User Claims
DeleteAuthorAsync(name)
Delete Author Query
Author Deleted Confirmation
DeleteUserByUserOID()
Delete User Request
User Deleted Confirmation
User Deletion Complete
SignOut Request
Redirect to SignOut URL
Redirect User to SignOut Page
Delete a Cheep
Post Delete Cheep(string CheepId)
DeleteCheepAsync(string CheepId)
Cheep Deleted Confirmation


Client
Web App
AuthorRepository
UserService
GraphServiceClient
Azure AD B2C
Initialization
Initialize UserService
Create ConfidentialClientApplication
Return Application instance
Create ClientCredentialProvider with Application
Create GraphServiceClient with Provider
Console.WriteLine("the tenant was connected to graph api.")
OnPostForgetUser
OnPostForgetUser()
Get User Claims
DeleteAuthorAsync(name)
Deletion Complete
DeleteUserById(objectID)
graphClient.Users[userId].Request().DeleteAsync()
Send Delete Request
Confirm Deletion
Deletion Complete
Console.WriteLine("User...deleted.")
SignOut("Cookies")
ListUsers
ListUsers()
graphClient.Users.Request().GetAsync()
Request User List
Return User List
User List Response
Loop through users
Console.WriteLine(user info)
Client
Web App
AuthorRepository
UserService
GraphServiceClient
Azure AD B2C

2 Process

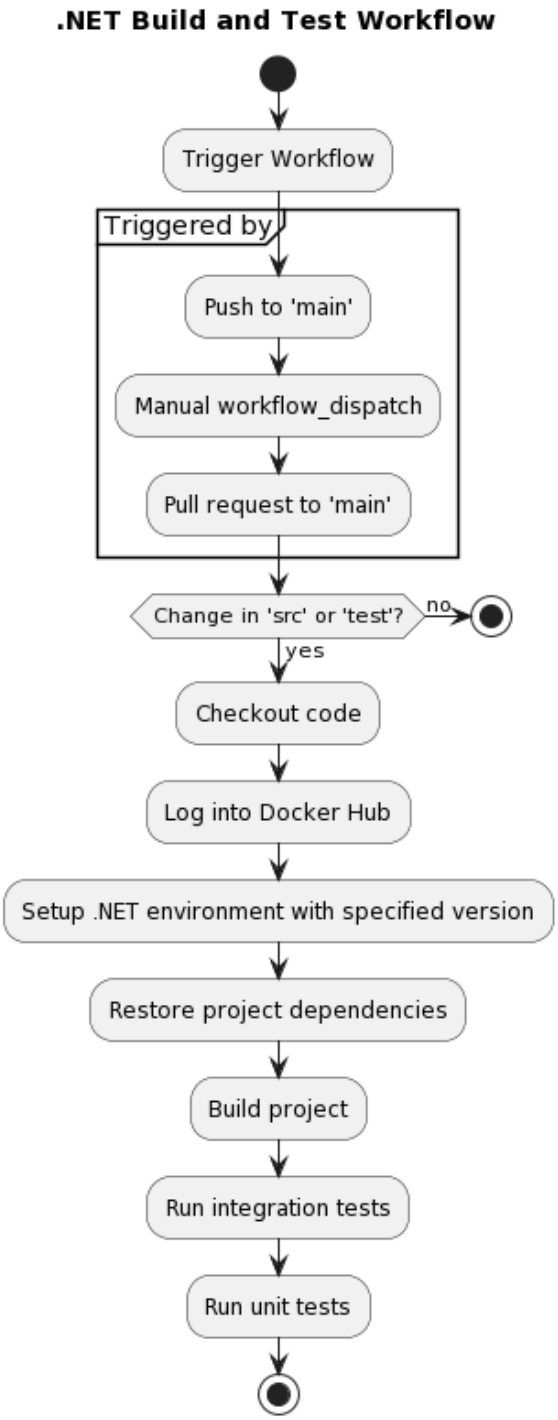
2.1 Build, test, release, and deployment



This workflow is the deploy workflow for azure

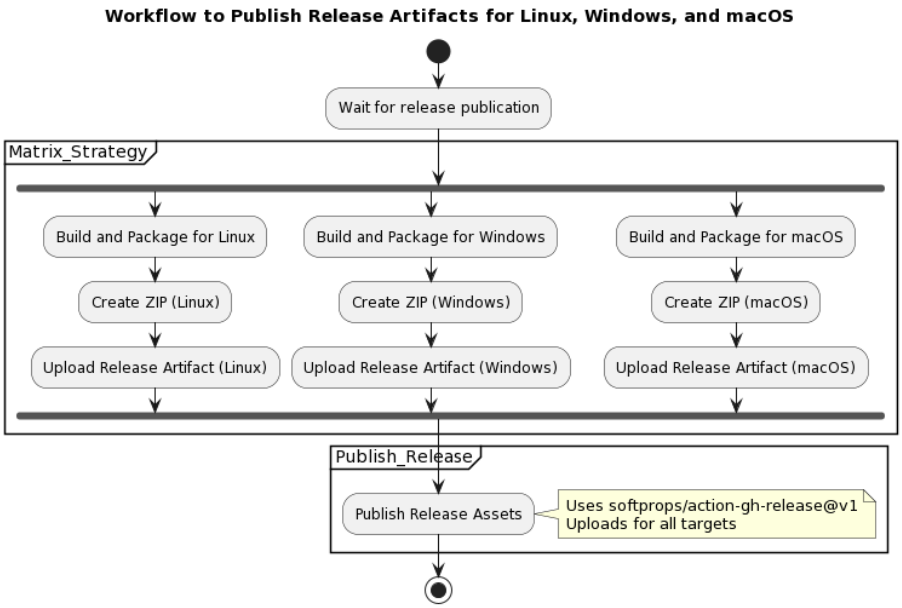
1. The User triggers the workflow on `main` branch or manually dispatching the workflow .

2. The GitHub repository then triggers the `build` job on the GitHub Actions Runner dedicated to building the app.
3. The build runner performs the following steps:
    - Checks out the repository.
    - Sets up .NET Core SDK version 7.0.x with prerelease versions included.
    - Builds the ASP.NET Core app from the specified source directory with the Release configuration.
    - Publishes the app to the output directory.
    - Uploads the build artifact (Chirp.Web) to GitHub's artifact storage.
4. Once the build job is complete, the repository triggers the `deploy` job on another GitHub Actions Runner.
5. The deployment runner downloads the artifact from the storage.
6. Finally, deployment runner deploys the downloaded artifact to the specified Azure Web App using the given publish profile.


.NET Build and Test Workflow
Trigger Workflow
Triggered by
Push to 'main'
Manual workflow_dispatch
Pull request to 'main'
Change in 'src' or 'test'?
no
yes
Checkout code
Log into Docker Hub
Setup .NET environment with specified version
Restore project dependencies
Build project
Run integration tests
Run unit tests

This workflow is build

1. This workflow is triggered in the same way as the deploy flow, but the build and test flow is also triggered when there is a pull request to `main`
2. The workflow then checks if there was a change to src or test if no then it stops
3. The build runner performs the following steps:
   - Checks out the repository.
   - Then workflow logs into Docker Hub
   - Sets up .NET Core SDK with the given version
   - restores project dependences
   - Builds the project without restoring dependencies again.
   - Runs the Integration tests.
   - Runs the unit tests



This is release workflow

- The workflow is triggered when a release is published.
- The `Matrix_Strategy`
  - For Linux, the workflow builds and packages the application, creates a ZIP file, and uploads the artifact.
  - For Windows, the workflow repeats the same steps but tailored for the Windows target.
  - For macOS, the workflow performs the steps for the macOS target.
- After the artifacts for all three targets are prepared and uploaded, the `Publish_Release` partition publishes the release assets using the `softprops/action-gh-release@v1` action. This step uses the uploaded

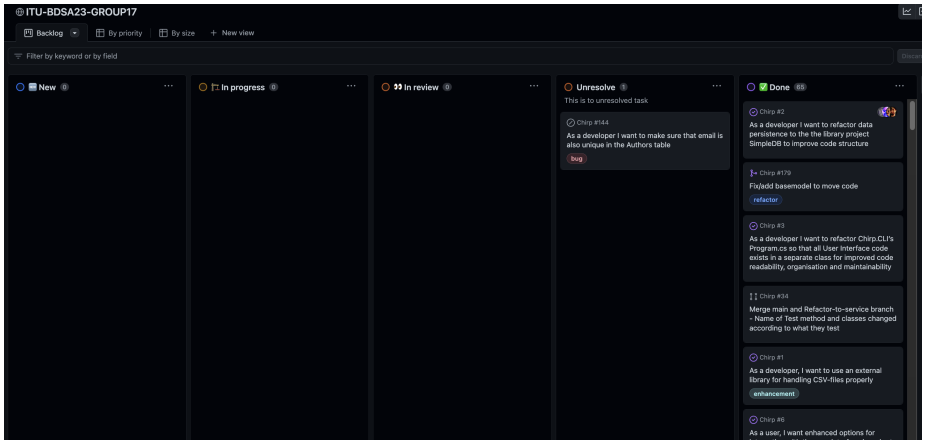
artifacts for each target as part of the release.
- The process ends after the release assets are published.

2.2 Team work

Overall, we managed to complete all the features we wanted for the application. These include all the features specified by the requirements of the project and some extra features, such as some UI changes and hashtags.

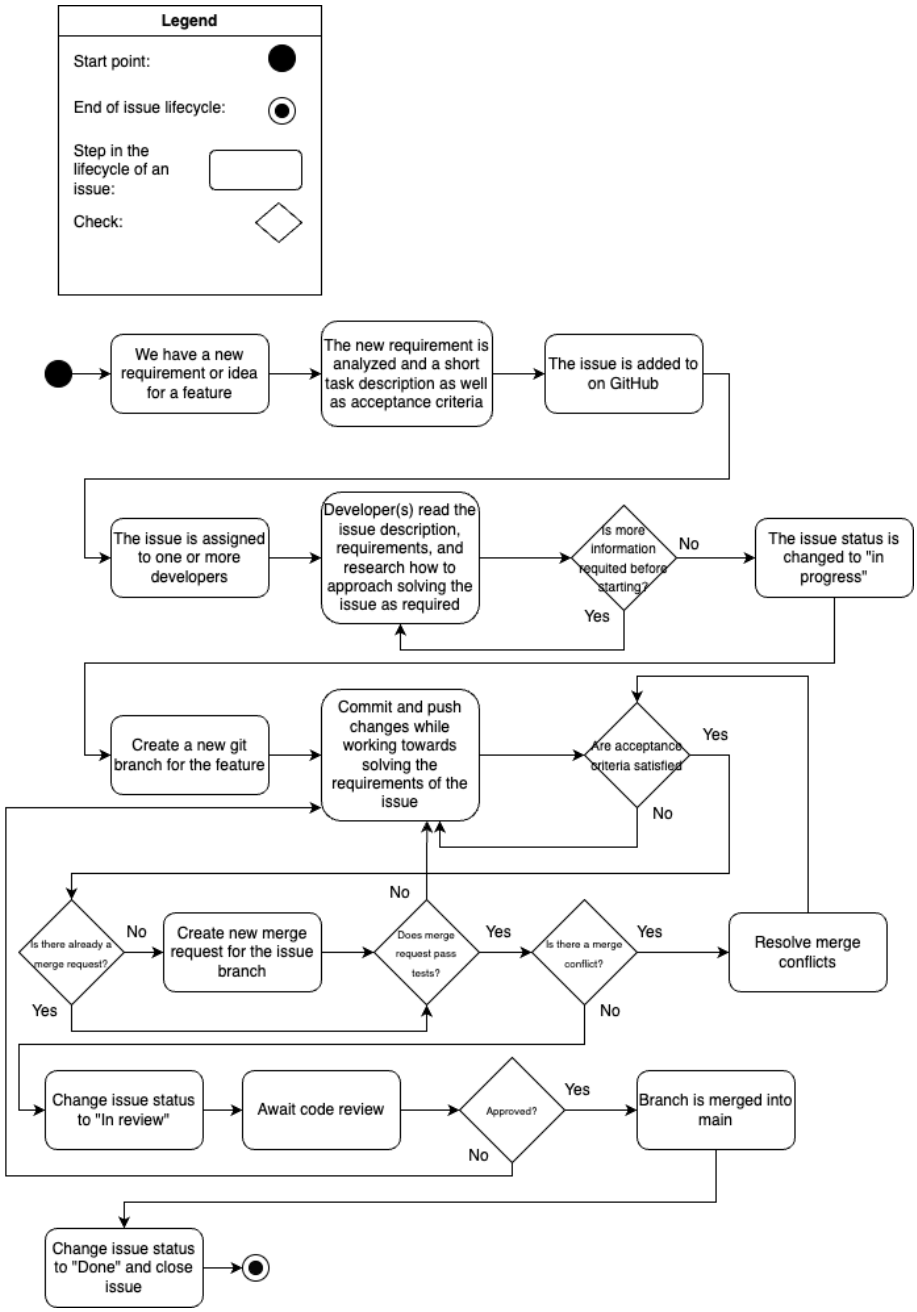
We have one unresolved task in our project board: To make the email unique in the the Authors table, since there was a possibility that an Author could appear twice or more in the table with same name and email but with different id. The reason we did not resolve it is the low priority.

The image below shows the project board just before hand-in, with the remaining unresolved issue.



There are of course many more features we could have implemented given time. Some ideas, which were discussed during development but not prioritized include the ability to comment on a cheep and sharing a cheep to name a few.

The diagram below shows the lifecycle of a GitHub issue from it's creating until it is closed and resolved.


Legend
Start point:
End of issue lifecycle:
Step in the lifecycle of an issue:
Check:
We have a new requirement or idea for a feature
The new requirement is analyzed and a short task description as well as acceptance criteria
The issue is added to on GitHub
The issue is assigned to one or more developers
Developer(s) read the issue description, requirements, and research how to approach solving the issue as required
Is more information requited before starting?
No
Yes
The issue status is changed to "in progress"
Create a new git branch for the feature
Commit and push changes while working towards solving the requirements of the issue
Are acceptance criteria satisfied
Yes
No
Is there already a merge request?
No
Yes
Create new merge request for the issue branch
Does merge request pass tests?
No
Yes
Is there a merge conflict?
Yes
No
Resolve merge conflicts
Change issue status to "In review"
Await code review
Approved?
Yes
No
Branch is merged into main
Change issue status to "Done" and close issue

2.3 How to make *Chirp!* work locally

2.3.1 Run locally

In order to run the application locally, you can either

1. Clone this repository
2. Run the release version

2.3.2 Clone the repository

In order to run the application locally by cloning the repository, please do as follows:

Clone the repository using this git command:

```
git clone https://github.com/ITU-BDSA23-GROUP17/Chirp.git
```

Change directory into

```
cd "src/Chirp.Web"
```

Inside the directory, run one of the following commands:

```
dotnet watch --clientsecret [your-secret]
```

```
dotnet run --clientsecret [your-secret]
```

You should now have access to a localhost with a specific port, in which this web-app can be accessed

2.3.3 Run the release

In order to run the release versions, please do as follows:

On the main page of this repository, click on the ***Releases*** section There will be a few assets available (including source code), but only one of the following three will be relevant for us:

- Chirp-win-x64.zip, for Windows users
- Chirp-osx-x64.zip, for Mac users
- Chirp-linux-x64.zip, for Linux users

Please install and unzip one of the three folders, depending on your operating system Now, there should be the following application available in the extracted folder:

- Chirp.Web.exe for Windows users
- Chirp.Web for Mac and Linux users

Now, you have an runnable (as described in step 4). Depending on your operating system, you can run the web-app as follows:

Run the following commands:

```
dotnet dev-certs https -t
```

```
 ./Chirp.Web --urls="https://localhost:7102;http://localhost:5273" --clientsecret [your-secr
```

Upon running the application, a terminal will pop up, indicating in which port (in the localhost) the web-app is up and running

2.4 How to run test suite locally

In the root folder run this command to test all the test

```
dotnet test
```

Make sure you have docker running in your machine

The following test have been implemented

2.4.1 Unit test

The unit tests are designed to test each individual component of our application by itself.

We have designed a series of unit tests to verify that our DTOs correctly encapsulate data. These tests confirm that each DTO retains and accurately represents the data passed to its constructor.

To run only the unit tests, use the following command in the root folder of the project:

```
dotnet test --filter Category=Unit
```

2.4.2 Integration test

The integration tests are designed to test how different parts of the application interacts with eachother. These tests involves instances of the database containers and checks if the application does the CRUD operations as expected.

The integration tests test that the repositories classes are able to correctly recieve and modify the relevant data in the database.

Before running integration tests, make sure Docker is running as the tests rely on `Testcontainers.MsSql` to create a containerized MS SQL Server instance.

To run only the integration tests, use the following command in the root folder of the project:

```
dotnet test --filter Category=Integration
```

2.4.3 End to end test

The playwright can be going into the folder in which the test is saved:

```
cd test\Chirp.Web.Test\Playwright.Test\PlaywrightTests
```

And then run the build command

```
dotnet build
```

After this you need to install the browser:

```
pwsh bin/Debug/net7.0/playwright.ps1 install
```

If you are on Linux or do not have Powershell you can use https://nodejs.org/en

Refer to the given link for installation guide https://docs.npmjs.com/downloading-and-installing-node-js-and-npm

Then run

```
npx playwright install --with-deps
```

Then in the project you can run:

```
dotnet test
```

Which should start the test

3 Ethics

3.1 License

We chose to use the MIT license for our Chirp application, since it allows other developers to distribute, use and copy our software without imposing significant restrictions.

3.2 LLMs, ChatGPT, CoPilot, and others

Utilizing Large Language Models (LLMs), particularly ChatGPT, has proven advantageous for gaining a basic understanding of frameworks like Entity Framework Core, Docker, and Onion architecture through direct questioning. However, reliance on LLMs for information poses a challenge, as the responses must be approached with skepticism, often necessitating validation from more reliable sources. Employing LLMs in code generation has been helpful for explaining errors and facilitating the debugging process, but the generated code is frequently flawed, requiring additional effort in rectification. While GitHub Co-pilot aids in autocomplete and simple repetitive tasks, it may not be as effective for comprehending core concepts, making LLMs a valuable yet limited addition to a developer's toolbox.