IT UNIVERSITY OF COPENHAGEN

BDSA GROUP 17 2023

Course code: BSANDSA1KU

Date: December 21, 2023

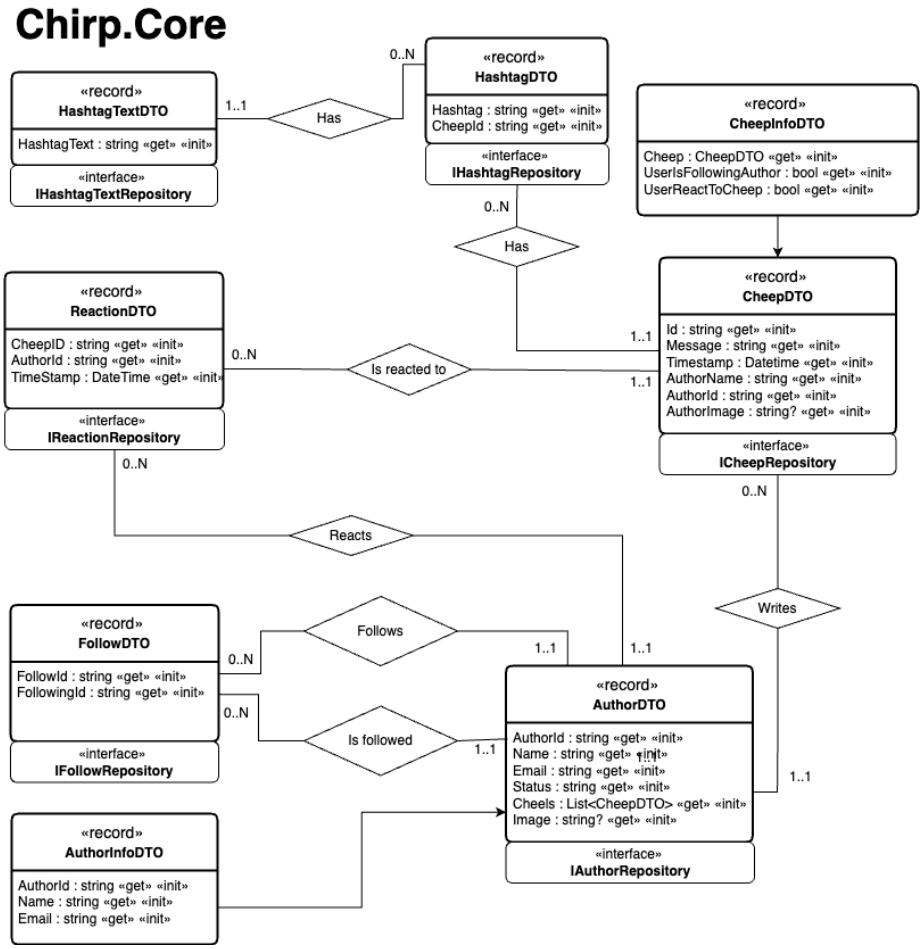
Name	Email	Date of Birth
Burak Özdemir	buoe@itu.dk	
Hanan Choudhary Hadayat	abha@itu.dk	
Joshua James Medilo Calba	jcal@itu.dk	
Julius Dalsgaard Bertelsen	jube@itu.dk	
Tan Dang	tanda@itu.dk	

Design and architecture

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.



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.

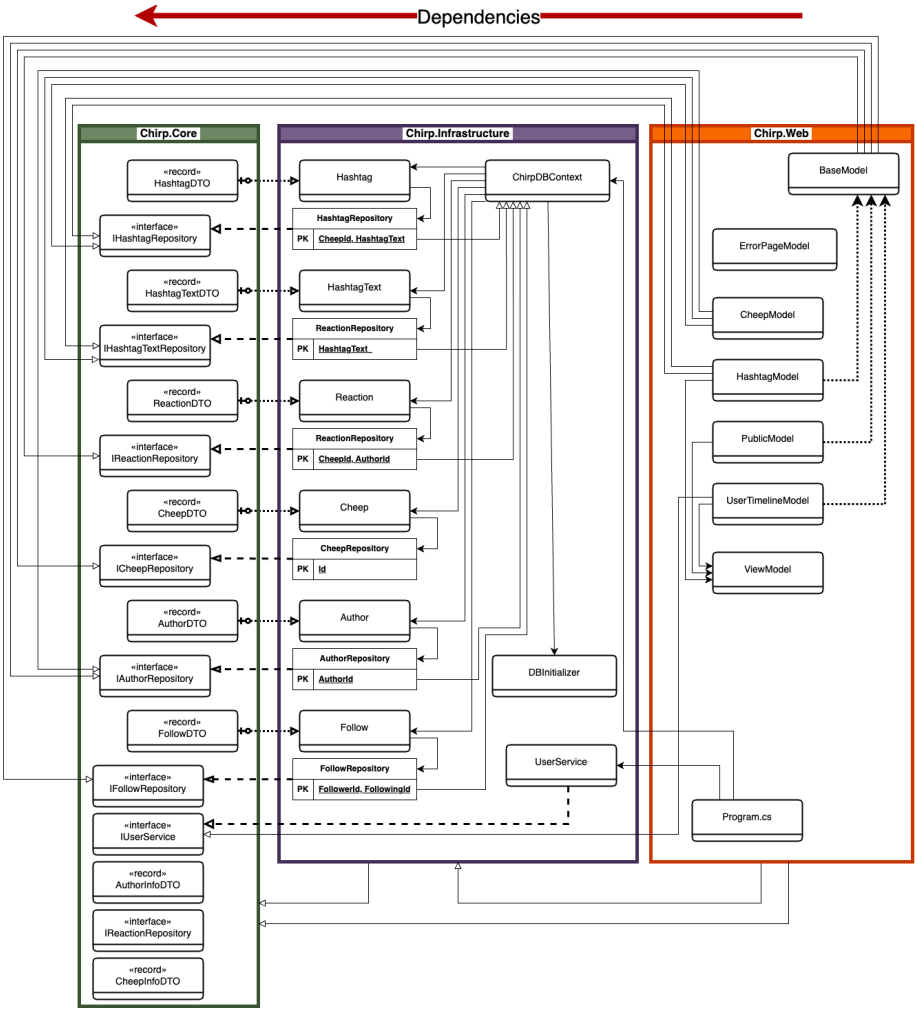
At the centre of the onion architecture are the domain entities in Chirp.Core, as seen in the domain model shown above.

Surrounding the Core is first the Chirp.Infrastructure layer. This layer contains the implementations of the repository interfaces, and is responsible for actually handing mechanisms of data storage, access and manipulation connected to the core entities.

The Web layer surrounds these layers. It is responsible for the presentation of the application, handling the user interface and user interactions. The layer interacts with the core and infrastructure layers, using the domain entities and data access mechanisms to allow the user the interact with the business logic and data of the application.

The diagram shows how the Chirp application has been designed with an 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.

Architecture of deployed application

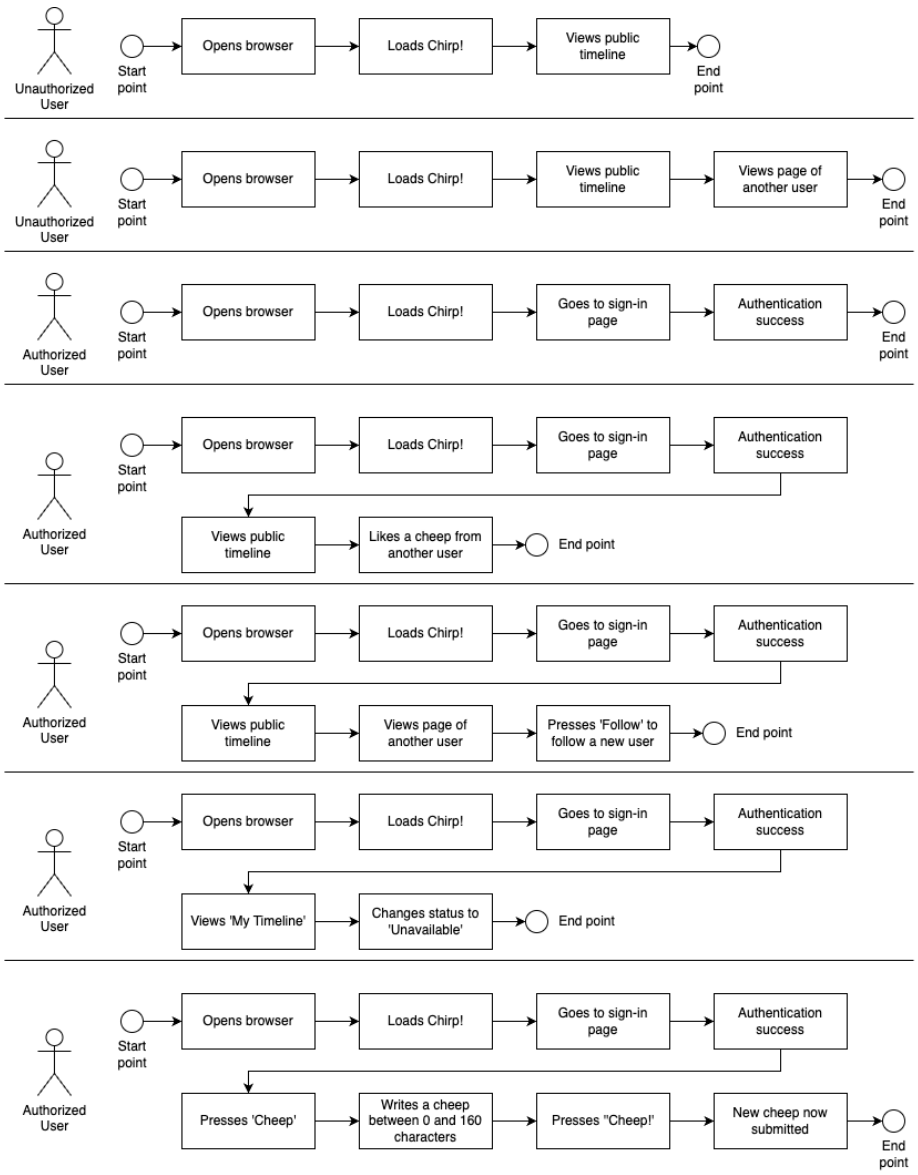
The Chirp application is deployed on Microsoft Azure, utilizing Asure services with an Azure SQL database.

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).

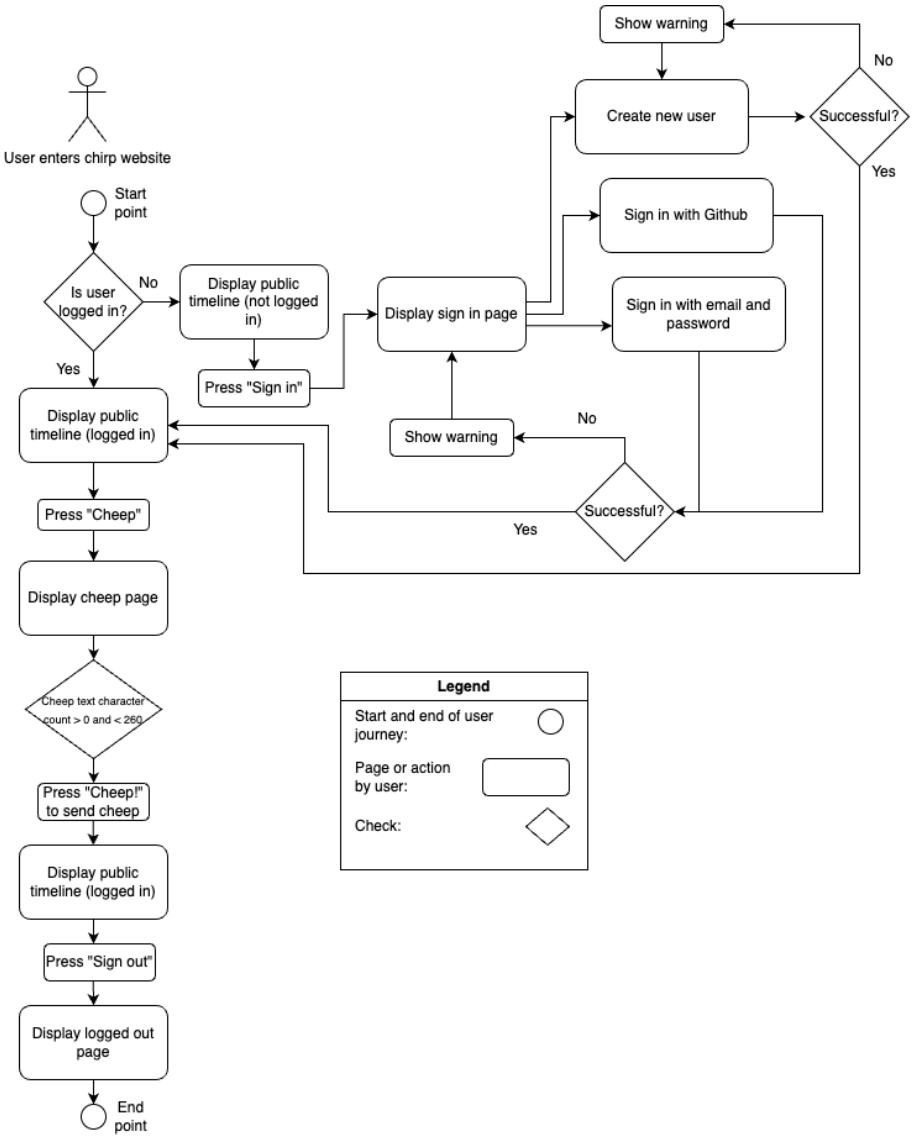
User activities

A user may follow various paths when using the Chirp application.

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



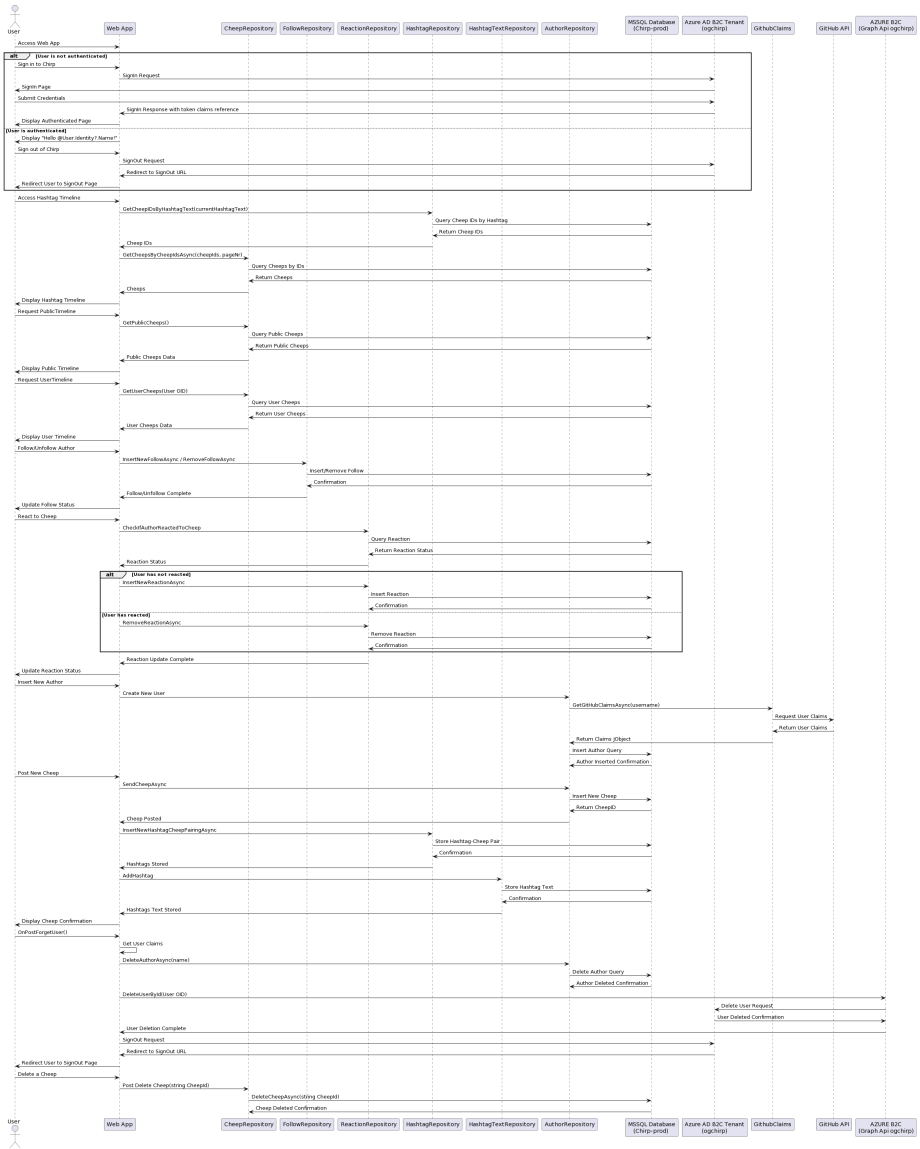
Additionally, the diagram below shows a slightly more detailed view of possible scenarios of a user journey through Chirp, in which a user enters the chirp website, logs in, or creates a profile if necessary, sends a cheep, and then logs out.



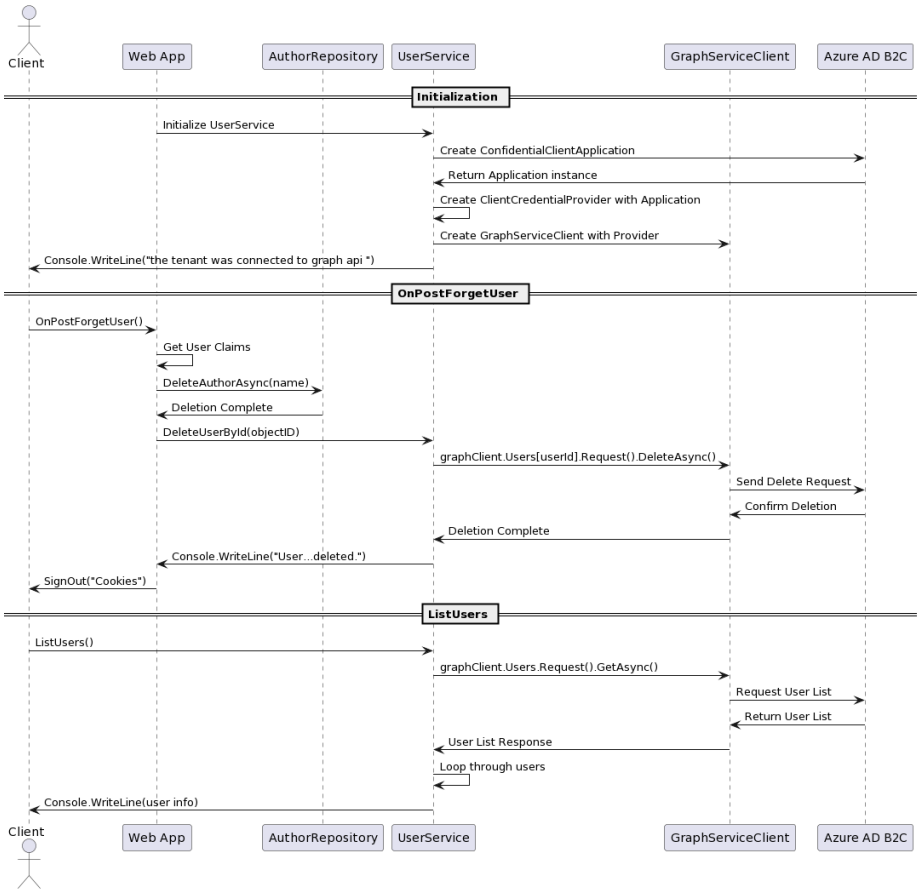
Sequence of functionality/calls trough *Chirp!*

There is a flow of messages and data through the chirp application, which allow the user to see and interact with a completely rendered web page.

The diagrams below illustrates this flow of messages and data, starting with the sending of an HTTP request by an authorized user to the root endpoint of the application and ending with the completely rendered web-page that is returned to the user. The diagram shows the different kinds of calls and the responses.

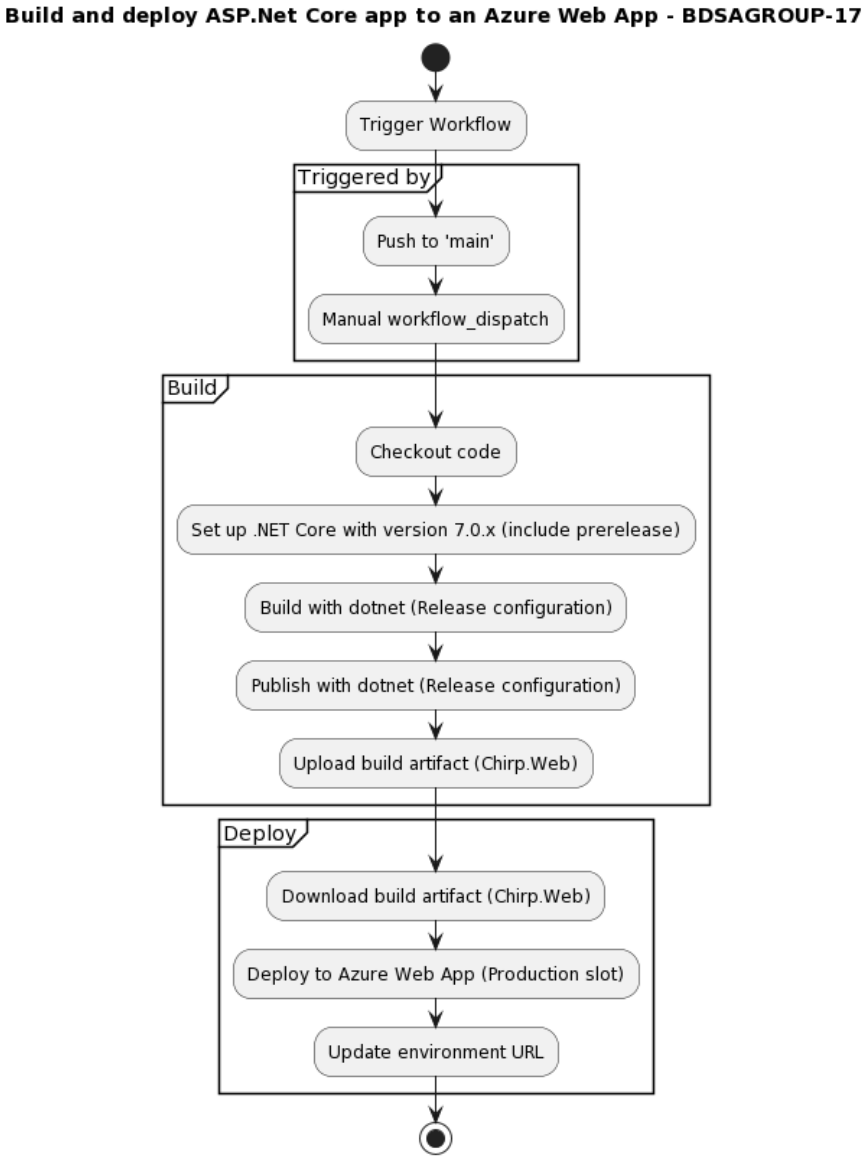


Another diagram, this one shows . . . ..


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

Process

Build, test, release, and deployment

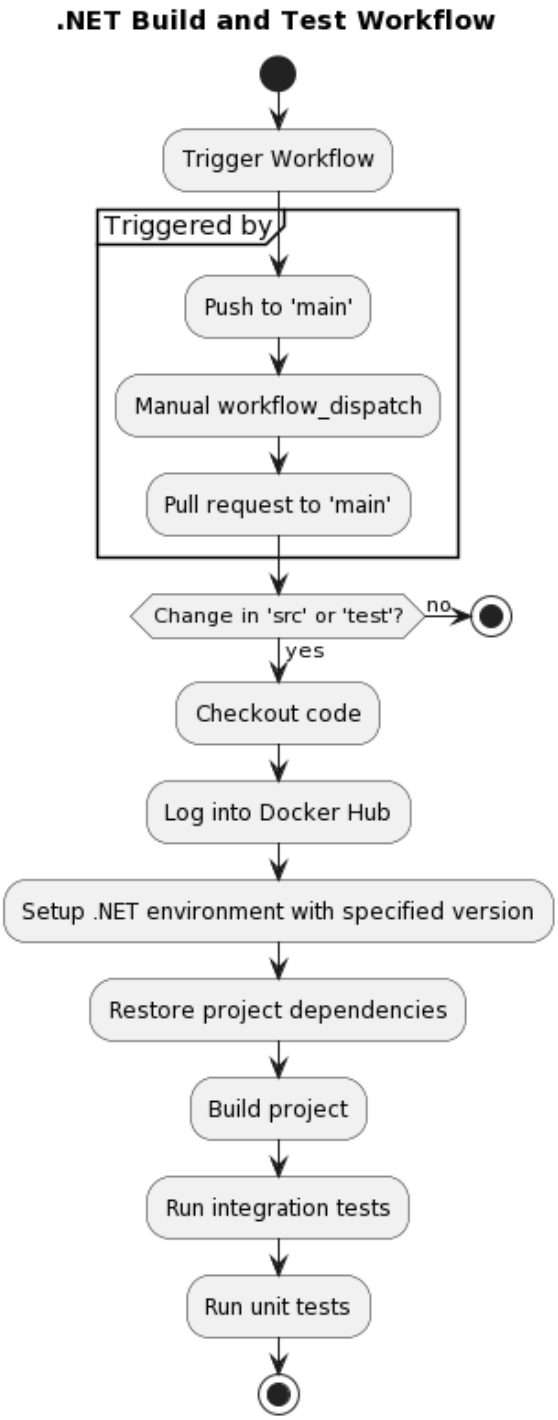


This workflow is the deploy workflow for azure

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

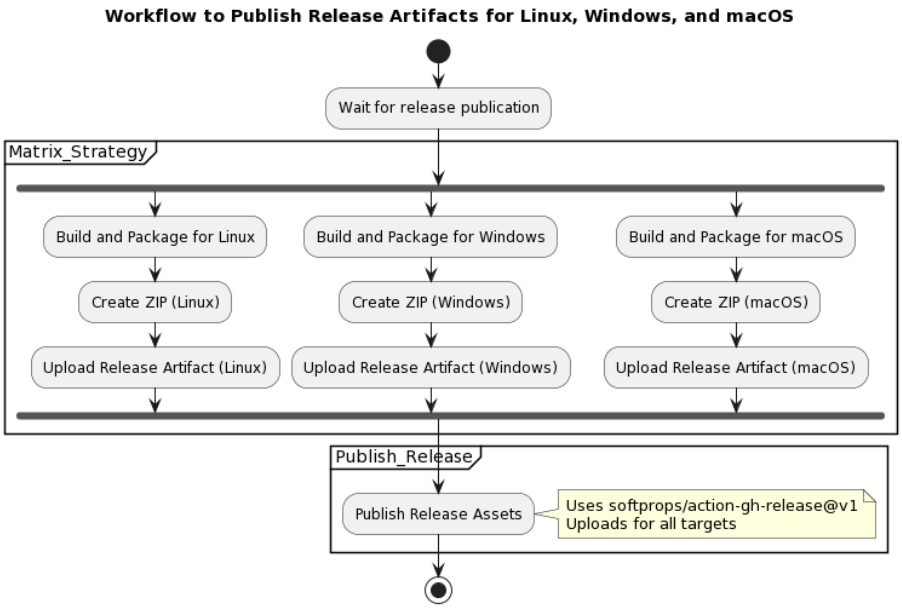
Runner dedicated to building the app.

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.

Team work

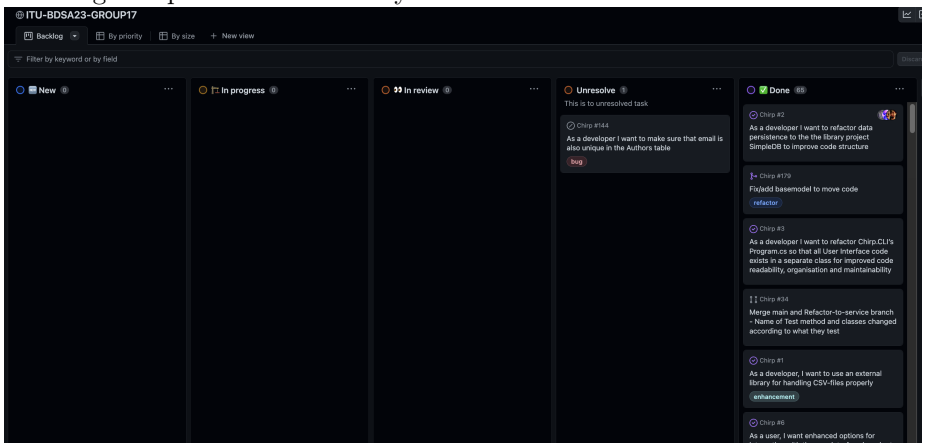
Throughout the project we have used GitHub issues to help structure the collaboration on the features of the Chirp application by multiple developers.

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, hashtags, and ...

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.

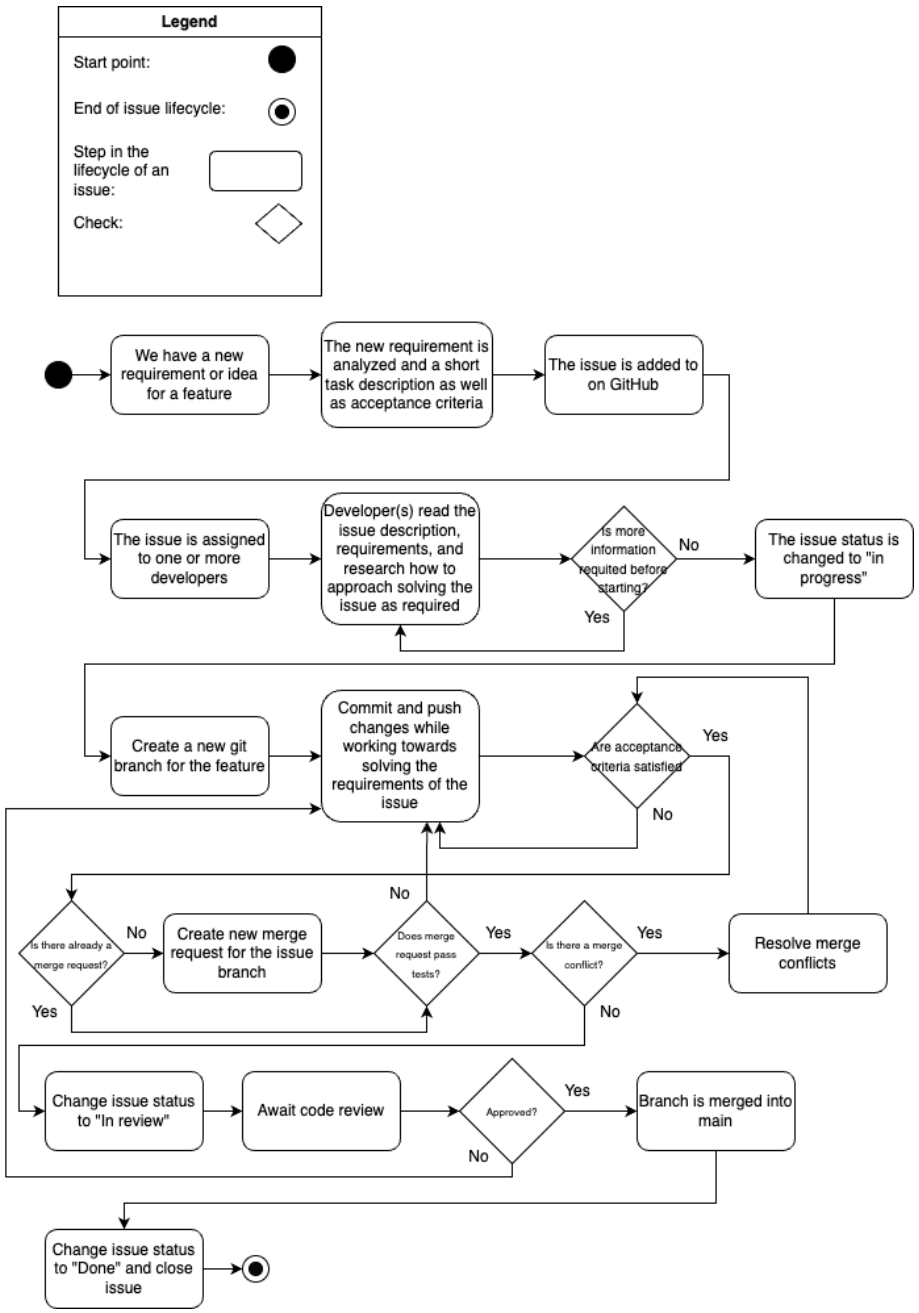
We were able to complete all the feature we want for our application. There were of course many feature we can implement, comment a cheep as well as share a cheep to name a few, but those were never in our original plan since we only focus on those feature we could make.



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.

When deciding to prioritize a new feature or requirement, we create an issue featuring a description of the task with a list of acceptance criteria. This issue is then assigned to one or more developers, who work on it in a designated branch until the requirements are fulfilled, a merge request is created and the feature branch is merged into the main branch of the project repository.

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



In practice, this process was not always strictly adhered to, but the diagram gives a general and idealized depiction of the process, which was mostly followed.

Even though this process was not always strictly adhered to, the work with GitHub issues but still proved a very useful tool during development.

How to make *Chirp!* work locally

Run locally

In order to run the application locally, you can either 1. clone this repository, or 2. run the release version.

Cloned 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

Releases 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</i>, for Windows users</li>
  - Chirp.Web</i>, for Mac and Linux users</li> <br>
  ```

  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 [you
```

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

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

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.

DTO Unit Tests

- `AuthorDTO_ShouldHoldProvidedValues`: Checks if the AuthorDTO object correctly assigns and retains the values provided.
- `CheepDTO_ShouldHoldProvidedValues`: Checks if the CheepDTO object correctly assigns and retains the values provided.
- `ReactionDTO_ShouldHoldProvidedValues`: Checks if the ReactionDTO object correctly assigns and retains the values provided.
- `HashtagDTO_ShouldHoldProvidedValues`: Checks if the HashtagDTO object correctly assigns and retains the values provided.

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

`dotnet test --filter Category=Unit`

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.

AuthorRepositoryTest

- `GetStatusNotNull`: Checks that the AuthorRepository is able to receive a non-null status (string).
- `GetStatusIsValid`: Checks that the AuthorRepository is able to receive a valid status, i.e., a status which equals ONLINE/OFFLINE/UNAVAILABLE.
- `SetUserStatusOnline`: Checks that the AuthorRepository is able to change the status of a user to ONLINE.
- `SetUserStatusOffline`: Checks that the AuthorRepository is able to change the status of a user to OFFLINE.
- `SetUserStatusUnavailable`: Checks that the AuthorRepository is able to change the status of a user to UNAVAILABLE.

CheepRepositoryTest

- `InsertCheepAsyncAddsCheepToDatabase`: Checks that cheeps are properly inserted into the database and are retrievable.
- `CheepOverLimitNotInserted`: Checks that a cheep over limit (i.e., over 160 characters) is NOT inserted into the database.
- `CheepUnderLimitNotInserted`: Checks that empty cheeps (i.e., cheeps with 0 characters in length) is NOT inserted into the database.

FollowRepositoryTest

- `GetFollowerIDsByAuthorIDAsync_ReturnsCorrectFollowerIDs`: Checks if the correct follower IDs are returned for a given author ID.
- `GetFollowingIDsByAuthorIDAsync_ReturnsCorrectFollowingIDs`: Checks if the correct following IDs are returned for a given follower ID.
- `InsertNewFollowAsync_InsertsFollowSuccessfully`: Checks that a new follow relationship is successfully inserted into the database.
- `RemoveFollowAsync_RemovesFollowSuccessfully`: Checks that a follow relationship is removed as expected.
- `GetFollowerCountByAuthorIDAsync_ReturnsCorrectCount`: Checks if the correct follower count is returned for an author.
- `GetFollowingCountByAuthorIDAsync_ReturnsCorrectCount`: Checks if the correct count of followings is returned for an author.

HashtagRepositoryTest

- `GetCheepIDsByHashtagText_GetsCheepIDsTiedToHashtag`: Checks if cheep ID's tied to a hashtag gets retrieved
- `InsertNewCheepHashtagPairingAsync_InsertsANewHashtagWithCorrectCheepIdAndHashtagText`: Checks if a new hashtag-cheep pairing is correctly inserted.
- `GetPopulalarHashtags_Returns10PopularHashtags`: Checks if the method returns the top 10 popular hashtags based on frequency.

HashtagTextRepositoryTest

- `AddHashtag_AddsHashtagToDatabase`: Checks if a new hashtag is added to the database.
- `AddHashtag_WillNotAddTheSameHashtagMoreThanOnce`: Checks that duplicate hashtags are not added to the database.
- `RemoveHashtag_RemovedSpecifiedHashtagTextIfItExist`: Checks if the specified hashtag text is removed from the database.

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
```

Note: As you may notice in our test folder we have more integration tests than unit tests. The reason is that unit test which is testing in the `Chirp.Core` package have only a few methods compared to the integration test, which is testing in the `Chirp.Infrastructure` package. Normally you have more unit test than integration test.

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

Ethics

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.

LLMs, ChatGPT, CoPilot, and others

We have used LLMs in two ways: For aid in writing code and for aid in understanding the overall concepts of different frameworks, architectures and concepts. In both cases, this came with both advantages and disadvantages.

When using LLMs (Primarily ChatGPT) for gaining a basic understanding of for instance Entity Framework Core, Docker or Onion architecture, the ability to ask direct questions can be a powerful tool in gaining familiarity with these concepts. On the other hand, LLMs are not always a reliable source of information, meaning that the answers provided by ChatGPT and the knowledge gained had to be approached with a level of scepticism, which was at times more frustrating than helpful. Thus, using LLMs in this way did not mean, that we did not also need to seek out more reliable sources of information and documentation. It is also possible that using LLMs in this way at all leads to a more superficial understanding of the core concepts at play, as any question that arises may be quickly answered by the LLM, without the need to seriously engage with ones own confusions or lack of understanding of the area.

The use of LLMs in generating or helping with the writing of code has also been both helpful and brought certain disadvantages. Most of the time the code that was generated by ChatGPT did not work according to what we wanted, and sometimes the work with debugging code which relied on help from LLMs ended up being more work than it was to just research and properly understand the problem ourselves. ChatGPT was mostly for explaining errors or explaining the code, and did prove helpful in the debugging process in this regard. We have also used GitHub co-pilot for error handling for our code, but it was quite minimal use. It has the feature to autocomplete our code when we write, but frequently the code it suggest is in no use, the only time it was been effective is when we need to write something that was repeating or very predictable, e.g. when we write insert methods in to our database in `DbInitializer.cs`. co-pilot also was helpful when writing tests, although we made a point out of not relying too much on it in order to make sure that we fully understood the tests and ensured that they properly tested what needed testing.

In conclusion the use of LLMs has been a useful tool to help with simple repetitive tasks or explaining, analyzing and understanding errors in the code and less helpful in understanding core concepts and ideas, and solving and aiding in complex complex tasks. Overall it is just another addition for a developers toolbox.