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 below 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.

```
<img src="./images/uml/Domain-model-Chirp.Core.drawio.png" alt="Chirp Core" width="100%" />
```

Architecture — In the small

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.

Onion

Architecture of deployed application

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).

Deployed_application

User activities

The diagram 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.

The diagram below shows a more detailed view of one slightly more elaborate scenario of a user journey through Chirp, in which a user logs in and sends a cheep.

OBS : insert diagram!

Sequence of functionality/calls trough *Chirp!*

The diagram below illustrates the flow of messages and data through the chirp application, 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 .....

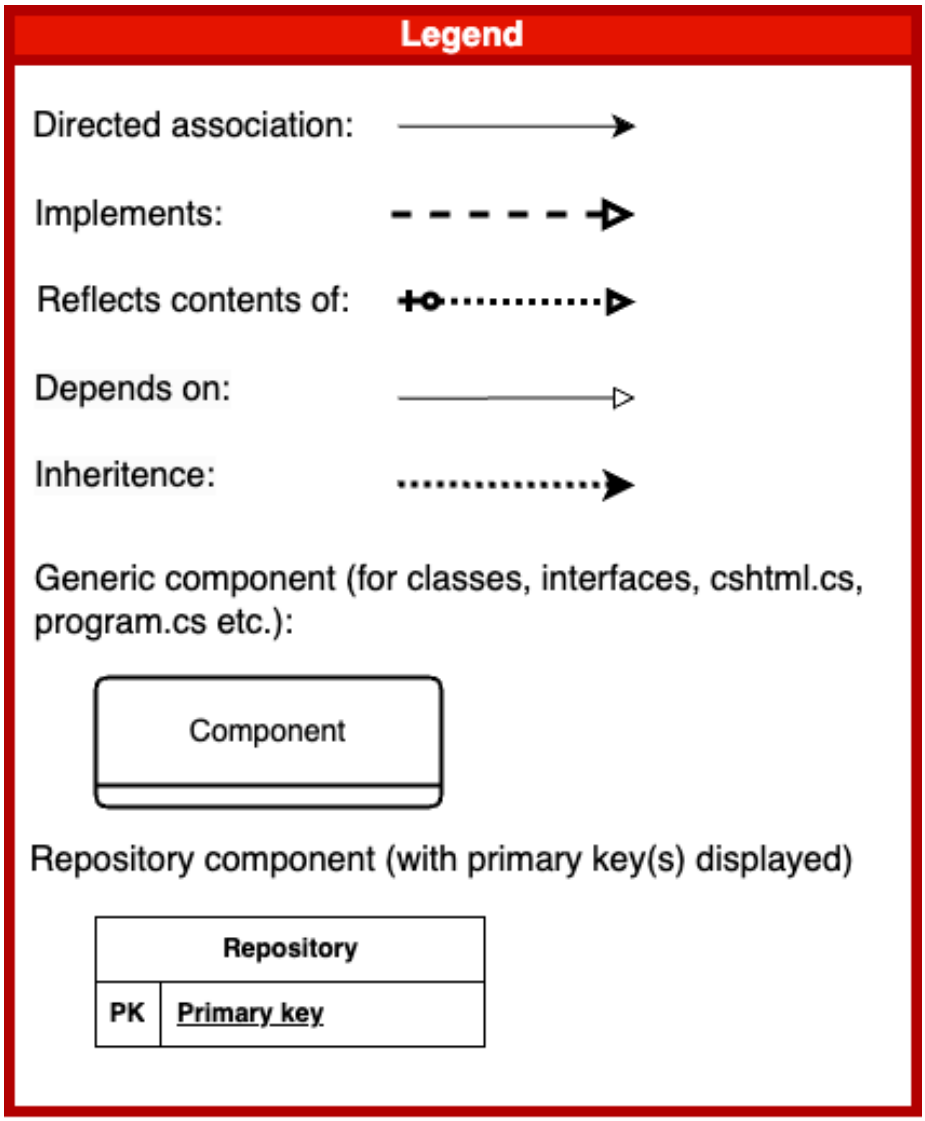

Figure 1: Legend

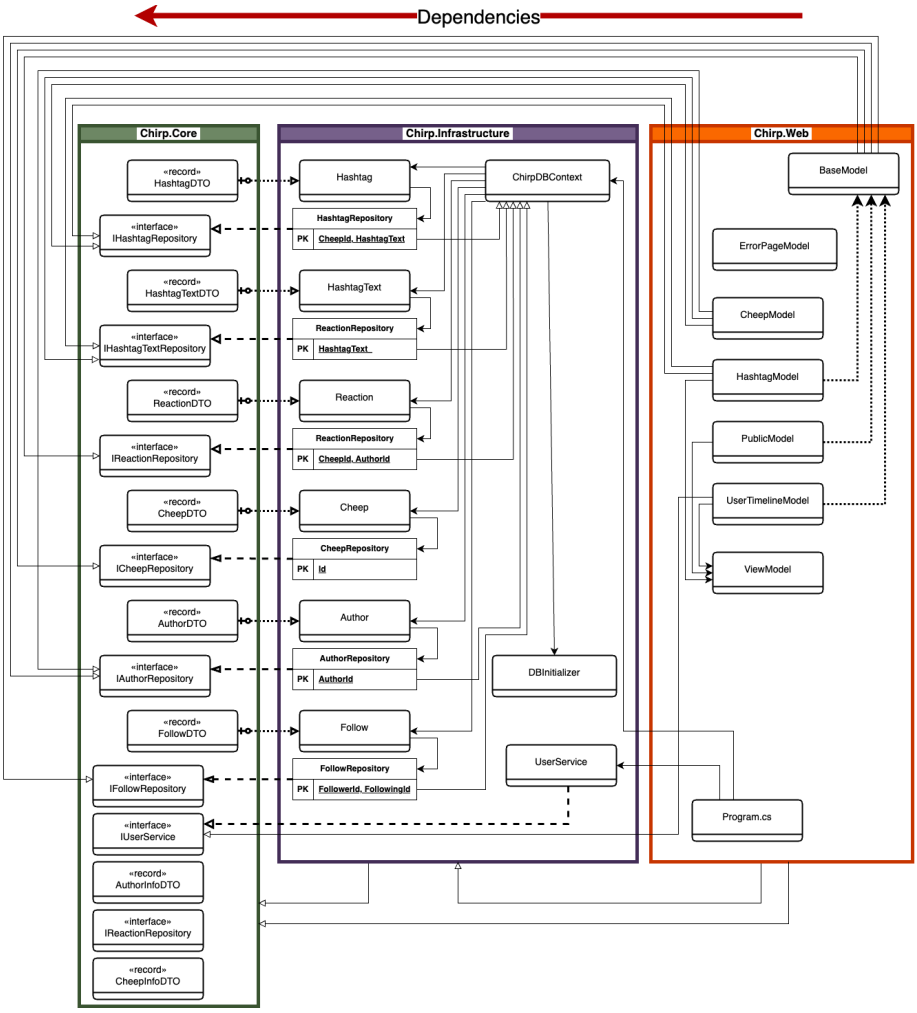

Figure 2: Architecture_in_the_small

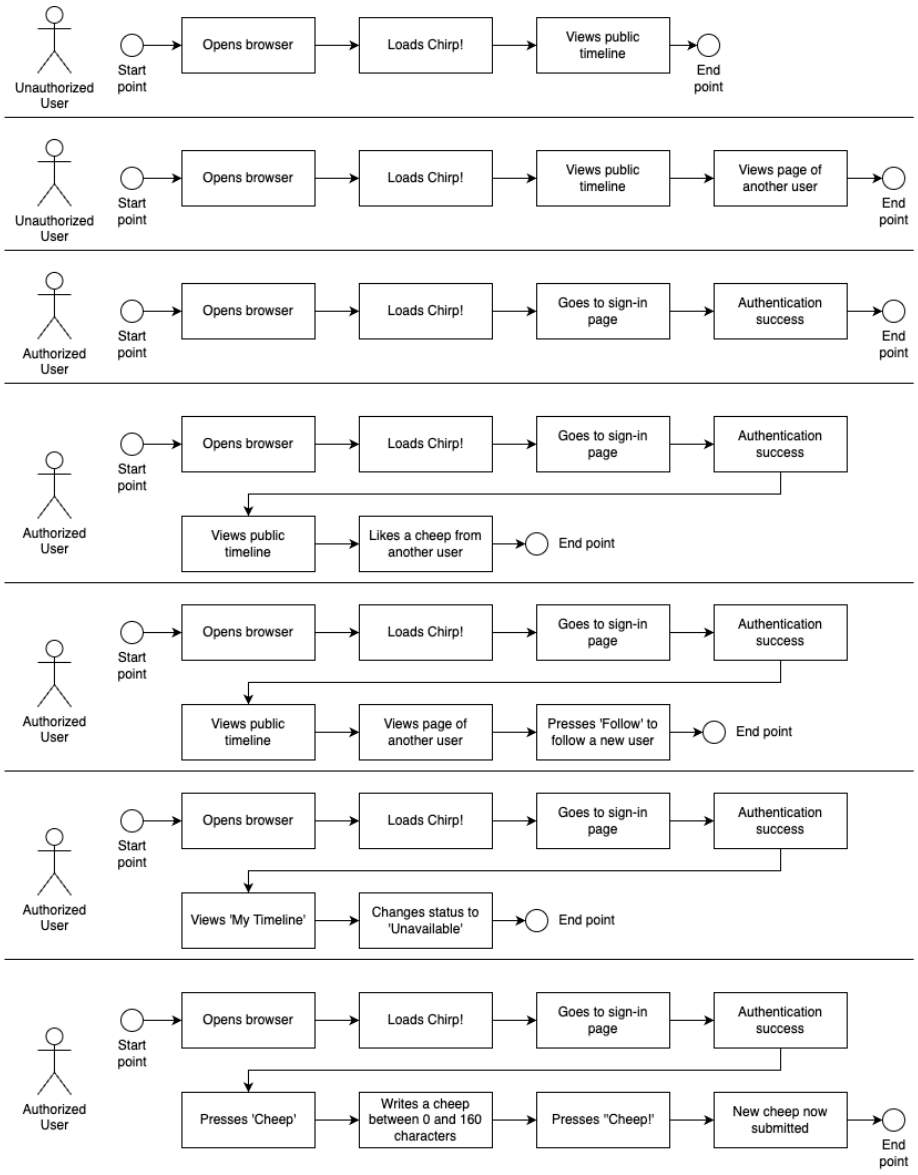

Figure 3: User_activities

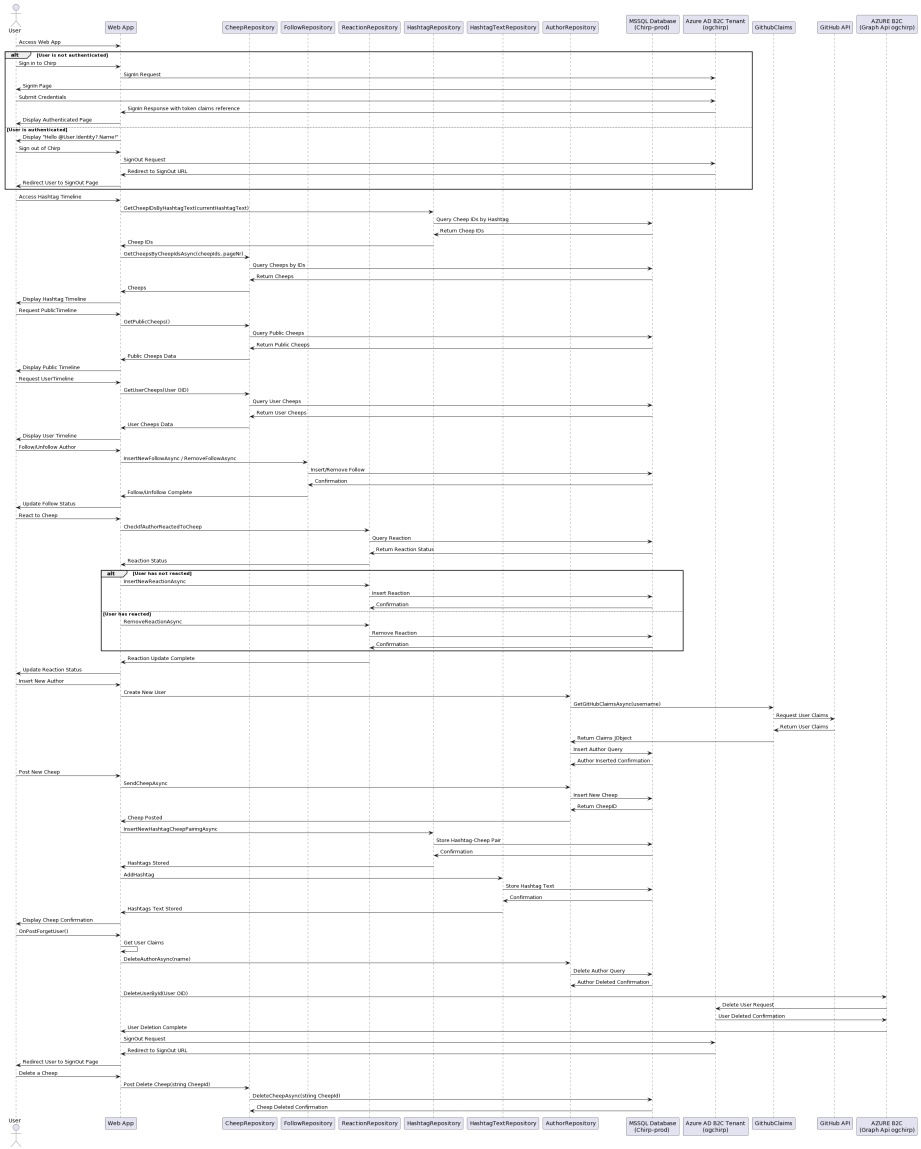

Figure 4: data_flow

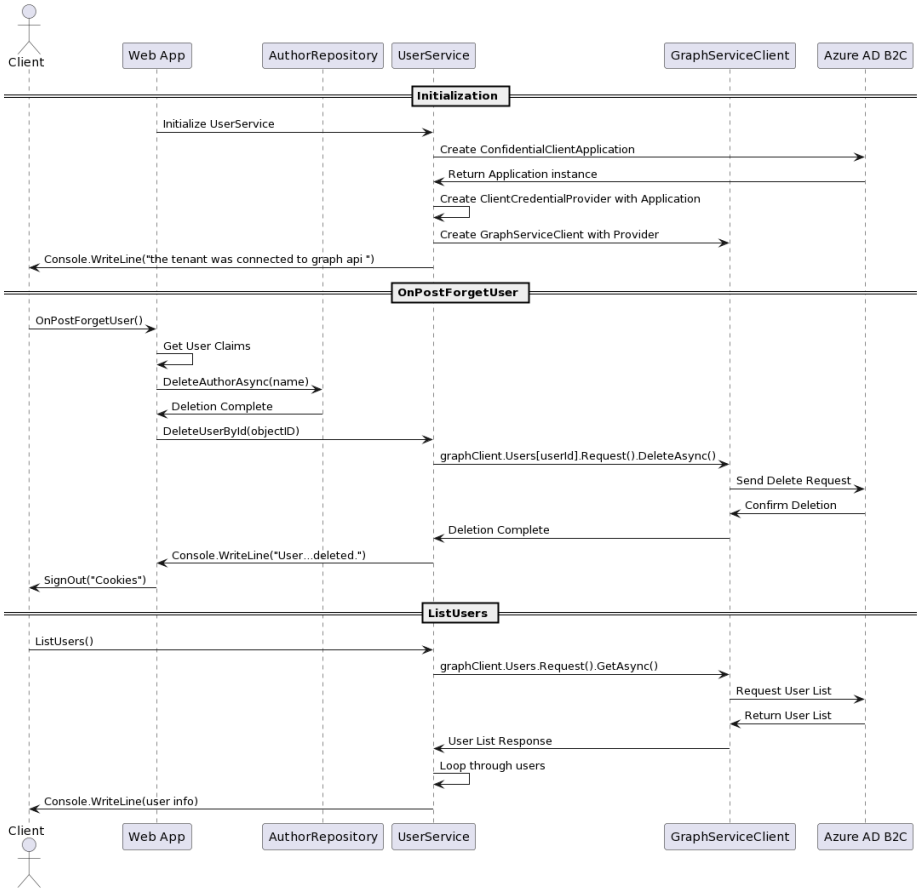

Figure 5: graph_api

Process

Build, test, release, and deployment

The UML diagram below illustrates the flow of activities in the Github Actions workflows, showing how the Chirp application is built, tested, released and deployed.

sequence_of_functionality_calls

Team work

How to make *Chirp!* work locally

Clone the repository using this git command

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

Start the program using this command

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

After you run the command you can go to `https://localhost:7102` or `https://localhost:5273`

It will then open the browser and here you can interact with the application. You can sign in by clicking on the top right corner with either your email or sign up with Github.

After you successfully sign in into the *Chirp!* application you can now do one of the following feature we have implemented

- Sending a Cheep by clicking the blue box in the top right corner that says Cheep
- Delete your own Cheep
- Follow another user
- Unfollow a user you follow
- Go to another user and see their Cheeps only, by clicking on the name above their Cheep post
- Go to your timeline by clicking on the "My Timeline" in the navigation bar to see your information and your cheeps and in your profile you can
  - Set your status by choosing either online, offline or unavailable
  - Clicking on Forget, to remove yourself from the application
- Liking a Cheep by clicking on the thumbs up icon in a Cheep
- Removing a Cheep that you liked by clicking on the thumbs up icon

- When a Cheep has a # following a text, you can then click on the hashtag, it will then go to the hashtag page with all the Cheep that includes that hashtag, as well as displaying available hashtag that has been Cheeped. The order is descending by popularity.
- Sign out of the application

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.

CheepRepositoryTest

- `InsertCheepAsyncAddsCheepToDatabase`: Checks that cheeps are properly inserted into the database and are retrievable.

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
```

End to end 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

Using LLMs has been both a help and a curse. Most of the time the code that was Generated by ChatGPT will not work according to what we wanted, and sometime give us more debug to do than if we google the problem ourselves.

With co-pilot we used it 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, e.g. when we write many insert method in to our database in `DbInitializer.cs`

In conclusion using LLM is a helpful tool to help simple task or understanding error. It is not applicable to use for complex task, but it is another addition for a developers toolbox.