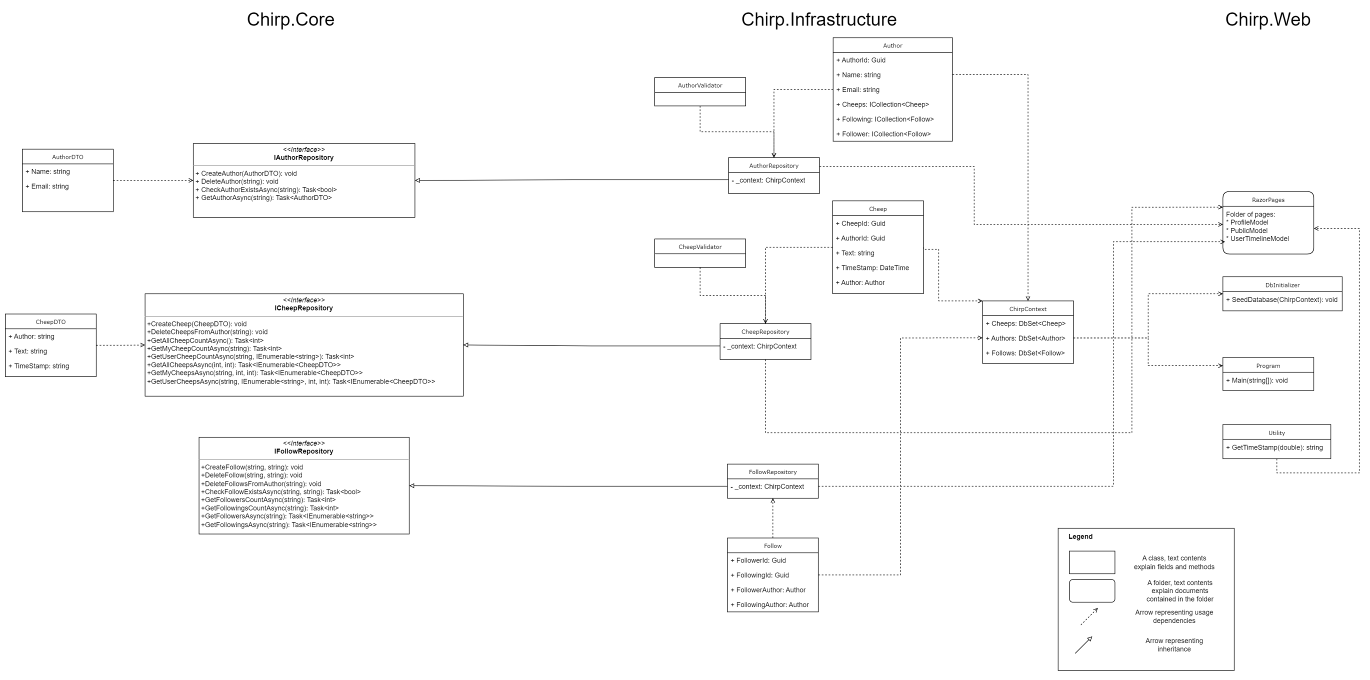
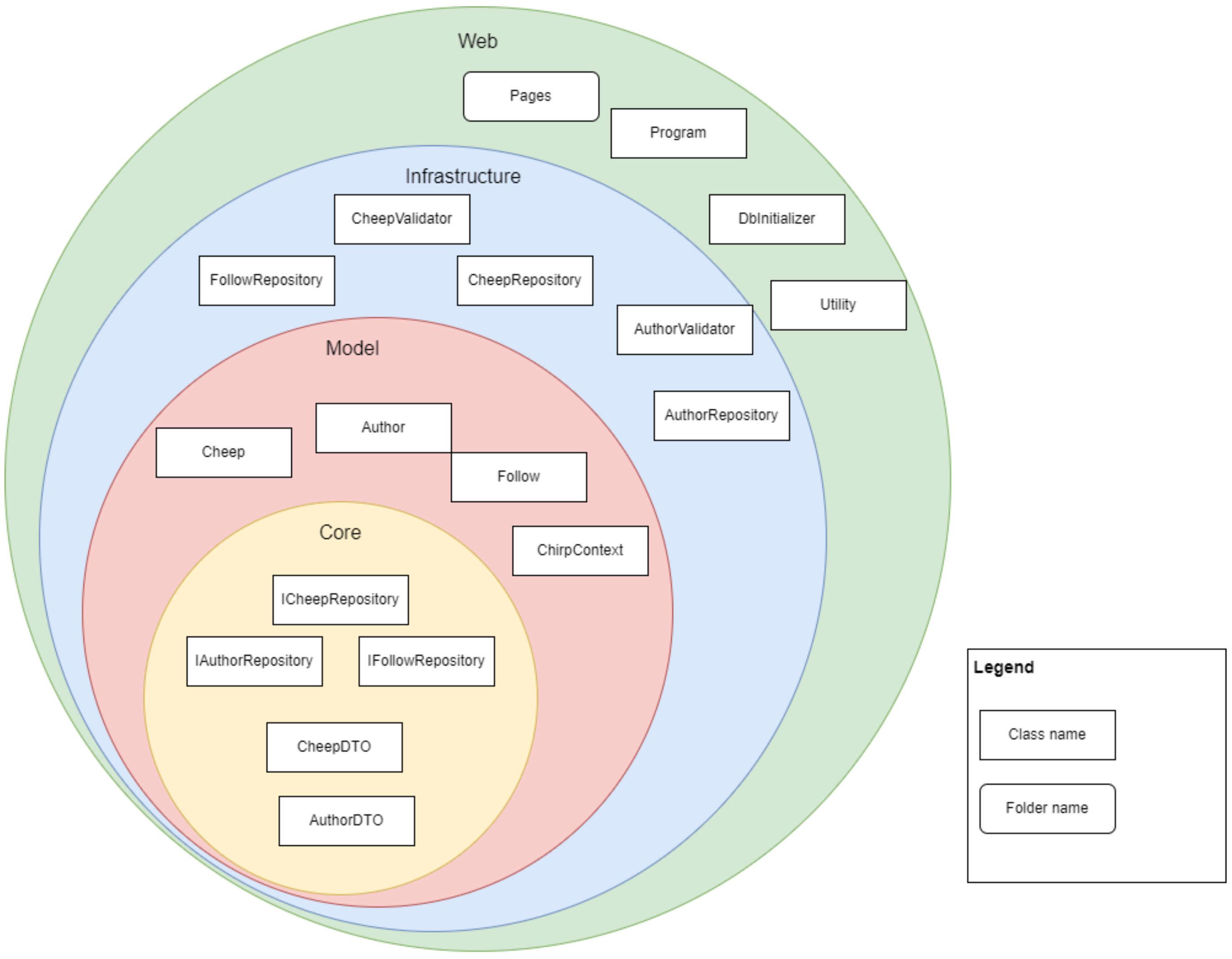
Design and Architecture of *Chirp!*

Domain model

Provide an illustration of your domain model. Make sure that it is correct and complete. In case you are using ASP.NET Identity, make sure to illustrate that accordingly.


Chirp.Core
Chirp.Infrastructure
Chirp.Web
AuthorDTO
+ Name: string
+ Email: string
<<Interface>>
IAuthorRepository
+ CreateAuthor(AuthorDTO): void
+ DeleteAuthor(string): void
+ CheckAuthorExistsAsync(string): Task<bool>
+ GetAuthorAsync(string): Task<AuthorDTO>
CheepDTO
+ Author: string
+ Text: string
+ TimeStamp: string
<<Interface>>
ICheepRepository
+CreateCheep(CheepDTO): void
+DeleteCheepsFromAuthor(string): void
+GetAllCheepCountAsync(): Task<int>
+GetMyCheepCountAsync(string): Task<int>
+GetUserCheepCountAsync(string, IEnumerable<string>): Task<int>
+GetAllCheepsAsync(int, int): Task<IEnumerable<CheepDTO>>
+GetMyCheepsAsync(string, int, int): Task<IEnumerable<CheepDTO>>
+GetUserCheepsAsync(string, IEnumerable<string>, int, int): Task<IEnumerable<CheepDTO>>
<<Interface>>
IFollowRepository
+CreateFollow(string, string): void
+DeleteFollow(string, string): void
+DeleteFollowsFromAuthor(string): void
+CheckFollowExistsAsync(string, string): Task<bool>
+GetFollowersCountAsync(string): Task<int>
+GetFollowingsCountAsync(string): Task<int>
+GetFollowersAsync(string): Task<IEnumerable<string>>
+GetFollowingsAsync(string): Task<IEnumerable<string>>
Author
+ AuthorId: Guid
+ Name: string
+ Email: string
+ Cheeps: ICollection<Cheep>
+ Following: ICollection<Follow>
+ Follower: ICollection<Follow>
AuthorValidator
AuthorRepository
- _context: ChirpContext
Cheep
+ CheepId: Guid
+ AuthorId: Guid
+ Text: string
+ TimeStamp: DateTime
+ Author: Author
CheepValidator
CheepRepository
- _context: ChirpContext
ChirpContext
+ Cheeps: DbSet<Cheep>
+ Authors: DbSet<Author>
+ Follows: DbSet<Follow>
FollowRepository
- _context: ChirpContext
Follow
+ FollowerId: Guid
+ FollowingId: Guid
+ FollowerAuthor: Author
+ FollowingAuthor: Author
RazorPages
Folder of pages:
* ProfileModel
* PublicModel
* UserTimelineModel
DbInitializer
+ SeedDatabase(ChirpContext): void
Program
+ Main(string[]): void
Utility
+ GetTimeStamp(double): string
Legend
A class, text contents explain fields and methods
A folder, text contents explain documents contained in the folder
Arrow representing usage dependencies
Arrow representing inheritance

Architecture — In the small



Above is the general architechture of the Chirp application shown as layers in an onion structure. Here is four layers shown although our code only reflects three. We have detached a part of the Infrastructure layer and shown it as the Model layer. This is because our code effectvily works this way - there are no dependencies going from the Model layer to the Infrastructure layer, and thus the dependency rule of the onion architechture is upheld.

Architecture of deployed application

Illustrate the architecture of your deployed application. Remember, you developed a client-server application. Illustrate the server component and to where it is deployed, illustrate a client component, and show how these communicate with each other. En snak om azure og hvordan det er sat op. Her ville vi også snakke om det hvis vi var gået tilbage til en SQLite server, men istedet skal vi måske bare nævne i en sidebemærkning, at vi løb tør for credits så vi gjorde det hele igen?

User activities

Illustrate typical scenarios of a user journey through your Chirp! application. That is, start illustrating the first page that is presented to a non-authorized user, illustrate what a non-authorized user can do with your Chirp! application, and finally illustrate what a user can do after authentication.

Make sure that the illustrations are in line with the actual behavior of your application. Skriv evt teksten først, og sæt først screenshots ind lige inden vi afleverer?

Sequence of functionality/calls trough *Chirp!*

With a UML sequence diagram, illustrate the flow of messages and data through your Chirp! application. Start with an HTTP request that is send by an unauthorized user to the root endpoint of your application and end with the completely rendered web-page that is returned to the user.

Make sure that your illustration is complete. That is, likely for many of you there will be different kinds of "calls" and responses. Some HTTP calls and responses, some calls and responses in C# and likely some more. (Note the previous sentence is vague on purpose. I want that you create a complete illustration.)

Illustration of the Chirp! the process of going onto the website as a UML sequence diagram.

Missing Features & Bugs

While we have been able to incorporate most of Chirps intended features, we find it important to highlight its missing elements and the bugs that remain.

Viewable Profiles

It remains that the Profile a given user has is only viewable by the user itself. If wanting to view another user's profile, the user will simply be redirected to the timeline of the given user. As such, when entering a users profile, it looks almost identical to the timeline page, with the exception of the Followers and Following information. This is a bit atypical to the usual profile page, which usually has more customization (e.g. profile pictures, a status, etc.). It remains that a seperate user profile does exist, but whether or not it fits the criteria of a normal profile is up to the individual.

Ability to See Information Kept By Chirp

As of now, the user information is displayed under Profile -> My Information section of the user that is logged in. However, it is not explicitly stated that this is the data kept by Chirp itself. This omitted transparency does not give the user a clear indication, that this information is saved and used. In hindsight, this would be changed to be explicitly stated within the My Information box, and thus conform to GDPR regulations.

Forget Me Feature

When wanting to delete an account, the user merely gets removed from all three seperate repositories: AuthorRepository, CheepRepository, and FollowRepository. The caveat, however, is that the personal information related to the user is still kept in our Azure B2C Tenant. This information includes the claims from the B2C Sign Up and Sign In user flow: Display Name, Email Addresses, Given Name, and Identity Provider Access Token. Hence, the user information is not completely removed, and should this application go into commercial use, it'd be essential to implement deletion fully in terms of GDPR laws and ethics.

Bugs

Proper Redirects

In terms of user experience, when wanting to follow and unfollow a given user, the original user will always be redirected back to the first page of cheeps. This is a minor detail, but for regular user usage with enough repetitions, it might cause more annoyance. Should the user try to follow multiple people based on their cheeps from a given webpage, they would have to go back, and find the original cheep of user again.

Time and Place for User Authentication

This is more of a situational bug, and as such, only something noticable when a specific chain of events happen. Should the user delete their account, not only will logging in again happen without having to re-register(given the problem with account deletion highlighted in the Forget Me Feature chapter), but the user will not be registered in our database, until they choose to either cheep or follow someone. This could be fixed by letting a user be registered in the database upon sign up, rather than when an interaction is done with the website.

Errors of Cheeping

Currently when a Cheep is posted, it only checks whether or not the cheep is within the correct parameters, i.e. there is text and there is below 160 characters. However, when posting an empty cheep, or a cheep that exceeds the character limit, the website crashes. To combat this issue, we'd implement an if-else statement, that displays an error messages when either scenario takes place.

Process

Build, test, release, and deployment

Illustrate with a UML activity diagram how your Chirp! applications are build, tested, released, and deployed. That is, illustrate the flow of activities in your respective GitHub Actions workflows.

Describe the illustration briefly, i.e., how your application is built, tested, released, and deployed.

Team work

Show a screenshot of your project board right before hand-in. Briefly describe which tasks are still unresolved, i.e., which features are missing from your applications or which functionality is incomplete.

Briefly describe and illustrate the flow of activities that happen from the new creation of an issue (task description), over development, etc. until a feature is finally merged into the main branch of your repository.

Our typical flow when it comes to adding new features, was to first write an issue. If the feature was from our weekly project work, we would base the description and acceptance criteria on that. We then added the issue to the project board and when someone started working on the issue, they assigned themselves to it and moved it to "In Progress". Then it came to the implementation itself. We would usually be two or three people working together, and after we learned about pair programming, we began doing it that way. How long it took to make varied greatly. When we had problems, we would frequently look up guides or consult a TA. When the feature was finished, we would always make sure that it worked locally before pushing it to main.

Our commit graph is not equal but there can be several different reasons for this, such as a member spending a lot of time working on Azure or being sick. The frequency at which each person commits also varies, as some people commit a lot while working on a feature, while others do it less, so that also has an influence.

How to make *Chirp!* work locally

These are the following prerequisites:

- .NET 7
- Docker Container

Follow these steps to open Chirp locally:

1. Use these respective properties for instantiating the Docker Container: 1a. Password: Y0waddup
2. Open Docker and Instantiate the local database container by running this command: 'docker run -e "ACCEPT_EULA=Y" -e "MSSQL_SA_PASSWORD=yourStrong(!)Password" -p 1433:1433 --name sqlpreview --hostname sqlpreview -d mcr.microsoft.com/mssql/server:2022-latest', and type in the password from step 1 where <yourStrong(!)Password> is.
3. Open Docker and run the container.
4. Clone the 'https://github.com/ITU-BDSA23-GROUP19/Chirp' repository onto your workspace. This could be your desktop, a folder you find appropriate, or a third place.
5. Wherever you have cloned the repository to, you should now see the given repository as a folder called Chirp.
6. Open either a terminal in the respective folder through a general terminal like PowerShell or a code editor of choice, and then a terminal within.
7. In either terminal, navigate to the ~/Chirp/src/Chirp.Web/ folder.
8. In either terminal, run the "dotnet run" command.
9. You should now see a 'Building...' syntax, and the respective queries being executed.
10. In your terminal, you should now be able to find a syntax like this: "info: Microsoft.Hosting.Lifetime[14] Now listening on: https://localhost:7102".
11. Go to https://localhost:7102 in your browser, and see the Chirp app.

How to run test suite locally

These are the following prerequisites:

- .NET 7
- Docker Container

Follow these steps to test Chirp locally:

1. Use these respective properties for instantiating the Docker Container: 1a. Password: Y0waddup
2. Open Docker and Instantiate the local database container by running this command: 'docker run -e "ACCEPT_EULA=Y" -e "MSSQL_SA_PASSWORD=yourStrong(!)Password" -p 1433:1433 --name sqlpreview --hostname sqlpreview -d mcr.microsoft.com/mssql/server:2022-latest', and type in the password from step 1 where <yourStrong(!)Password> is.
3. Open Docker and run the container.
4. Wherever you have cloned the repository to, you should now see the given repository as a folder called Chirp.
5. Open either a terminal in the respective folder through a general terminal like PowerShell or a code editor of choice, and then a terminal within.
6. In either terminal, navigate to the ~/Chirp/ folder.
7. Run the 'dotnet build' command.
8. You should now see the respective projects being restored and built.
9. Run the 'dotnet test' command.
10. You should now see the respective tests being run on each respective project and the results of them.

Test Suite Tests

This is a brief overview of what kinds of test we have in our test suites and what they are testing.

####Failed Tests

When running the test suite locally, it should be highlighted that some test will fail. They all are regarding the timestamp of cheeps. The reason they fail is that the strings differ in terms of timestamps, i.e. the expected string is "... 13:15:25", but the actual value is "...13.15.25". We have made the decision to keep these, since changing them will cause the tests to fail on GitHub instead. We have prioritized the tests on GitHub, rather than the local testing.

####Chirp.Core.Tests

The Chirp.Core tests consists unit tests. The unit tests are for creation of the AuthorDTO and CheepDTO objects. The AuthorDTO tests checks whether or not it is possible to create an AuthorDTO object, and the CheepDTO tests checks similarly if it is possible to create a CheepDTO.

####Chirp.Infrastructure.Tests

The Chirp.Infrastructure tests consist unit tests matching AuthorRepository, CheepRepository and FollowRepository tests. Each repository respectively gets tested for whether the repositories function as intended. For example, for the AuthorRepository, it tests whether or not it is possible to create the repository itself, to find an existing Author in the repository, can't have an Author of the same name, to create or delete an Author in the repository, and to find a non-existing Author in repository.

####Chirp.Web.Tests

The Chirp.Web tests consist of integration tests. They each tests the way our app functions, i.e. whether or not it is possible to see the timelines themselves (both private and public), and to see the first page on either timeline correctly.

Ethics

License

State which software license you chose for your application.

We use an MIT License for our code.

LLMs, ChatGPT, CoPilot, and others

During the development of this project, we have at times used ChatGPT to figure out why the code was not working as we thought it would or how to fix functionality that did not work as intended. ChatGPT has been good at detecting where the faults have been and what we might need to change but actually doing the change and creating a solution was a bigger challenge for the LLM. Mainly we have used the information about the mistakes to further research for more in depth explanations of the APIs or method calls we have used. This way we created the solution ourselves but ChatGPT helped us understand where the fault were.