# *Chirp!* Project Report

ITU BDSA 2023 Group 12

Markus Brandt Højgaard mbrh@itu.dk
Rasmus Søholt Rasmussen rhra@itu.dk
Allan Sigge Andersen asia@itu.dk
Mads Voss Hvelplund mhve@itu.dk
Lukas Brandt Pallesen lupa@itu.dk

## Contents

# 1 Design and Architecture of *Chirp!*

## 1.1 Domain model

Our Application builds on two main concepts: Cheeps and Authors.

### 1.1.1 Cheep

A Cheep is the understanding of a post, which holds a message from the author of the post. It has a reference to its author and a knowledge of when it was posted.

### 1.1.2 Author

An Author represents the user in our program. The user can write posts (Cheeps), which is why they are named Author. An Author has references to all their Cheeps, as well as their name and email. It is possible to follow an Author; therefore, the Author also has references to the Authors who follow it and the Authors it follows.

Our domain model consists of two data entities, which depict the attributes of a 'Cheep' and an 'Author' in the context of our application. Figure 1 shows the two classes, 'Cheep' and 'Author,' with their fields and cardinality relationships between each other and within themselves. The 'Cheep' class has a one-to-one relationship with an 'Author' class, and an 'Author' class has a zero-to-many relationship with the 'Cheep' class. Furthermore the 'Author' class has a zero-to-many relationship with authors who follow it and a zero-to-many relationship with authors whom it follows.

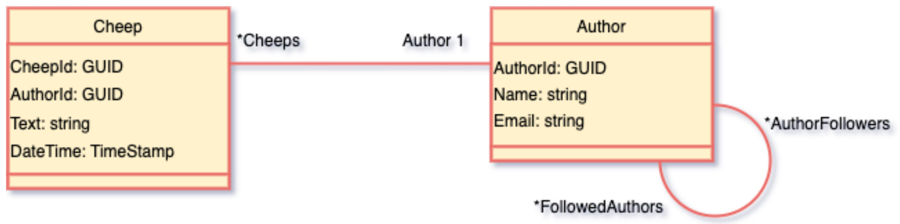


Figure 1: Illustration of the *Chirp!* data model as UML class diagram.

## 1.2 Architecture — In the small

Our *Chir!* application is written following the clean (or onion) architecture. This clean architecture of *Chirp!* can be seen visualized in figure 2. The idea with building *Chirp!* with a clean architecture, is that it implements both the principle of Domain Driven Development (DDD) and Dependency Inversion. (Microsoft 2023b)

When looking at figure 2, dependencies must be read going inwards toward the center of the illustration. In there we have the *Chirp.Core* package. Besides building *Chirp!* with clean architecture, we also make use of the repository pattern to allow for abstraction and organization of data handling. In the *Chirp.Core* package we hold two repository-pattern specific classes. The Data Transfer Objects (DTO), and the repository interfaces. This follows the DDD and the repository pattern, placing an abstraction of the business logic at the very center of our application.

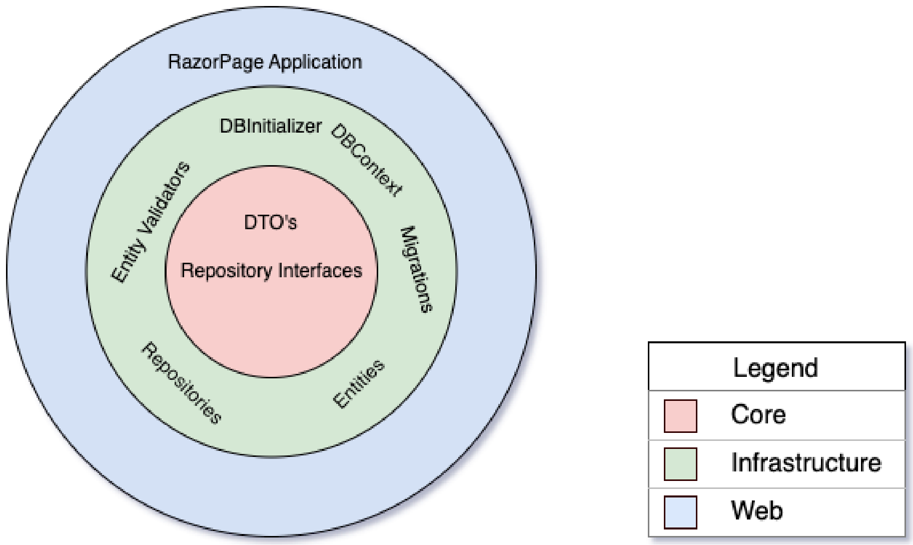


Figure 2: Clean Architecture

The next layer in the architecture is the data layer which manifests in *Chirp!* as the *Chirp.Infrastructure* package. Here lies all classes related to the data handling and database management. Following dependencies going towards the center, this package depends on the *Chirp.Core* package, as *Chirp.Infrastructure* holds the implementation of the repository interfaces. This dependency shows how the architecture implements the Dependency Inversion principle, which states that implementations should depend on abstractions and not the other way around.

The outermost layer is the entry point for our *Chirp!* application. In the code this is the *Chirp.Web* package, which holds a RazorPage Application. This package depends on both the *Chirp.Core* and the *Chirp.Infrastructure* package. It is also in this class the Program.cs file resides, which is responsible for configuring the application, including the configuration for dependency injection. Here it configures implementation from the *Chirp.Infrastructure* layer to be used when abstractions from the *Chirp.Core* layer is requested.

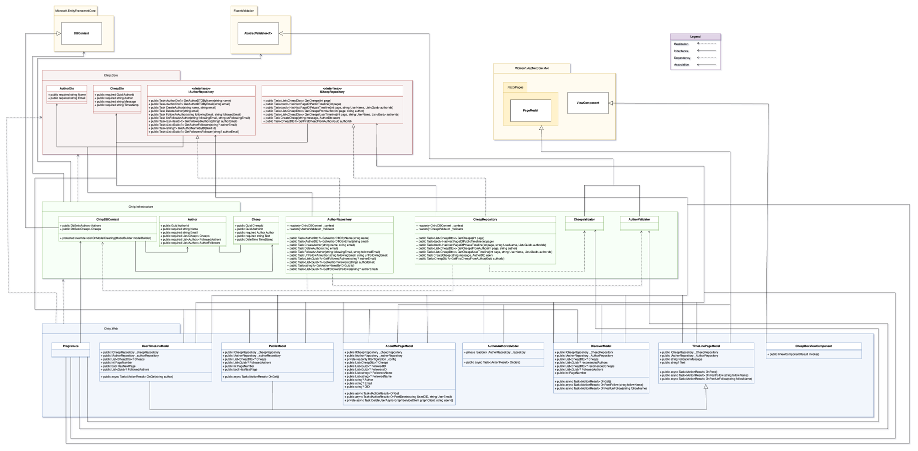


Figure 3: Package Diagram

For a more in depth visualization of which classes reside in which of the packages mentioned, see the figure 3, a complete UML package diagram for *Chirp!*.

## 1.3 Architecture of deployed application

In this section, we will delve into the overall architecture of *Chirp!* as a deployed web application. Figure 4 shows the structure of the architecture. Our application is hosted on the cloud-based Microsoft Azure platform. The code is accessible through an Azure service called ‘Azure Web Service.’ A client can make an HTTP request to our web application, and the response will return an instance of *Chirp!* on their computer. We utilize two additional Azure services: Azure SQL Server and Azure Active Directory Business to Consumer (Azure AD B2C).

Azure SQL Server is employed to host our database by providing a connection string to the web service, which connects to our EF Core implementation. Azure AD B2C is used to authenticate users of *Chirp!* through a SignUpSignIn user flow. This flow redirects a login or signup request to GitHub, our chosen Identity provider. GitHub returns an access token when a user logs in.

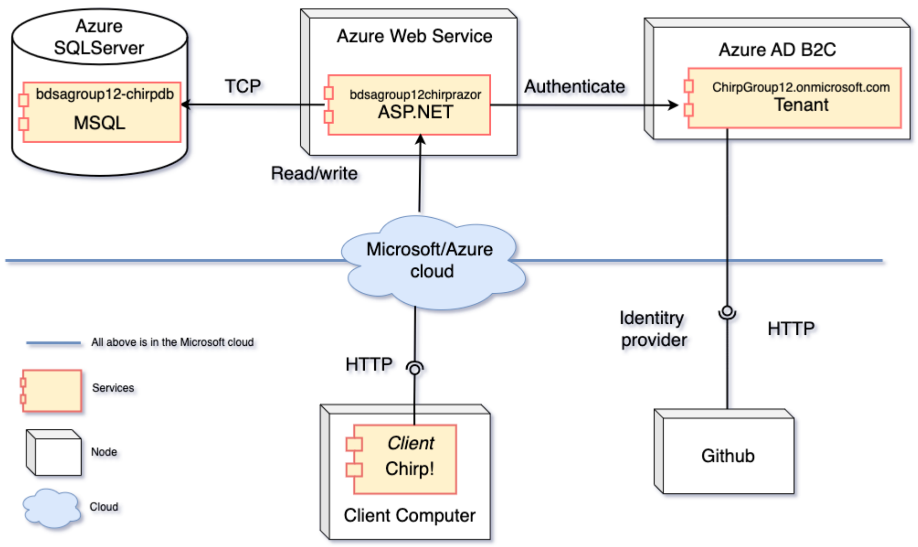


Figure 4: Illustration of the architecture of the deployed *Chirp!* app as a UML diagram.

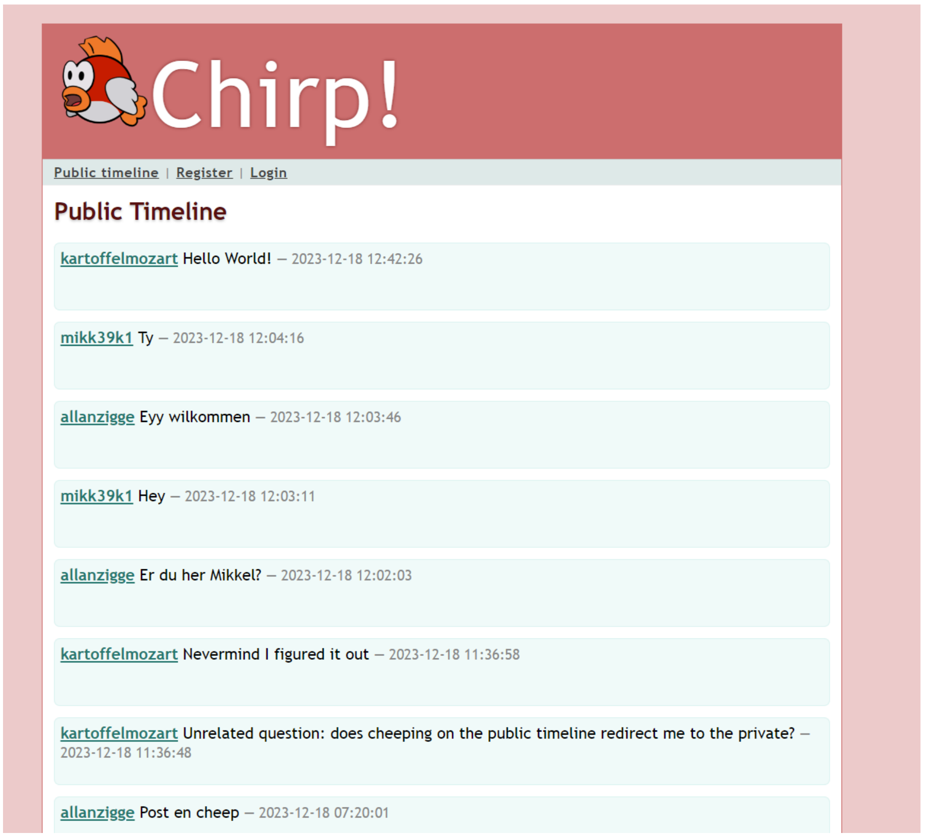
## 1.4 User activities

The gist of our application lies in two main activities; writing cheeps and reading cheeps. A **cheep** is a short message of 1-160 characters, and is not messaged to a particular person. Each cheep is publicly available for everyone to see, and there is no functionality to direct the attention of particular users onto your cheep. Lastly, the user has an option to follow other users, to get easier access to cheeps written by them. The use of our application is defined by a range of tabs, a **cheep box** and two different kinds of users - **authenticated** users and **unauthenticated** users. That is, users that are logged in and users that are not. The two kinds of users differ in the elements that they have access to on the site.

### 1.4.1 Unauthenticated

The **unauthenticated** users have access to the **Public Timeline** tab, the **Register** tab and the **Login** tab.

**1.4.1.1 Public Timeline** The **Public Timeline** contains a list of each (non-deleted) cheep ever sent, and is available to both authenticated and unauthenticated users. The cheeps are displayed on pages of 32 cheeps, and the pages can browse between using the **next** and **previous** buttons at the bottom of the page.



Public Timeline

kartoffelmozart Hello World! — 2023-12-18 12:42:26

mikk39k1 Ty — 2023-12-18 12:04:16

allanzigge Eyy wilkommen — 2023-12-18 12:03:46

mikk39k1 Hey — 2023-12-18 12:03:11

allanzigge Er du her Mikkel? — 2023-12-18 12:02:03

kartoffelmozart Nevermind I figured it out — 2023-12-18 11:36:58

kartoffelmozart Unrelated question: does cheeping on the public timeline redirect me to the private? — 2023-12-18 11:36:48

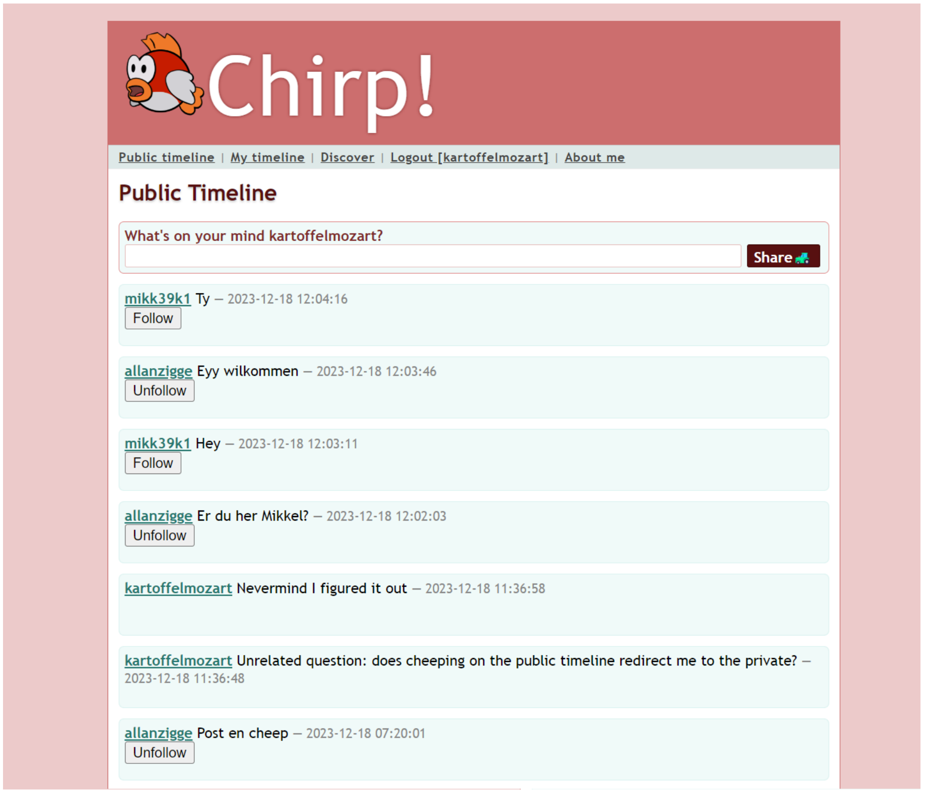
allanzigge Post en cheep — 2023-12-18 07:20:01

Figure 5: Chirp public timeline unauthenticated

**1.4.1.2 Register** If the user has not registered before, clicking the **Register** tab redirects the user to a prompt for signing in to GitHub to continue to *Chirp!*. After signing in with GitHub, the user is registered and logged in to *Chirp!*, and is now an authenticated user. If the user has registered before, this information might still be stored in cookies managed by the browser, and the user will then not be asked to sign in with GitHub - they will simply be logged in right away and immediately turn into an authenticated user. Once authenticated, the user's GitHub username is also used as the user's *Chirp!* username. Only unauthenticated users have access to the register tab.

**1.4.1.3 Login** The **Login** tab does the exact same thing as the register tab - their behavior is the same in every way.

### 1.4.2 Authenticated


Chirp!
Public timeline | My timeline | Discover | Logout [kartoffelmozart] | About me
Share

Public Timeline

What's on your mind kartoffelmozart?

mikk39k1 Ty — 2023-12-18 12:04:16
Follow

allanzigge Eyy wilkommen — 2023-12-18 12:03:46
Unfollow

mikk39k1 Hey — 2023-12-18 12:03:11
Follow

allanzigge Er du her Mikkel? — 2023-12-18 12:02:03
Unfollow

kartoffelmozart Nevermind I figured it out — 2023-12-18 11:36:58

kartoffelmozart Unrelated question: does cheeping on the public timeline redirect me to the private? — 2023-12-18 11:36:48

allanzigge Post en cheep — 2023-12-18 07:20:01
Unfollow

Figure 6: Chirp public timeline

The **authenticated** user has access to the **Public Timeline**, **My Timeline**, **Discover**, **Logout**, and **About Me** tabs.

**1.4.2.1 The follow mechanic** An authenticated user has, attached to each cheep on the public timeline, a **follow** button. Upon clicking this, the follow button is replaced by an **unfollow** button, and the user will be regarded as following the user that sent the particular cheep.

**1.4.2.2 My Timeline** The **my timeline** tab is similar to the public timeline, but here only the user's own cheeps and cheeps written by users that the user is following are displayed. One can visit other people's private timelines, but here the user's followed users' cheeps will not be showed.

**1.4.2.3 Discover** Let the user of the application be A. The **discover** tab contains the latest cheep of each user B that is deemed interesting for A. B is deemed interesting if at least two users followed by A are following B, and the users B are sorted after how many of A's followed users are following them. Here, A can browse through users that might be more relevant to A. The discover tab is only accessible to authenticated users. The users B is illustrated in the red box in figure 7

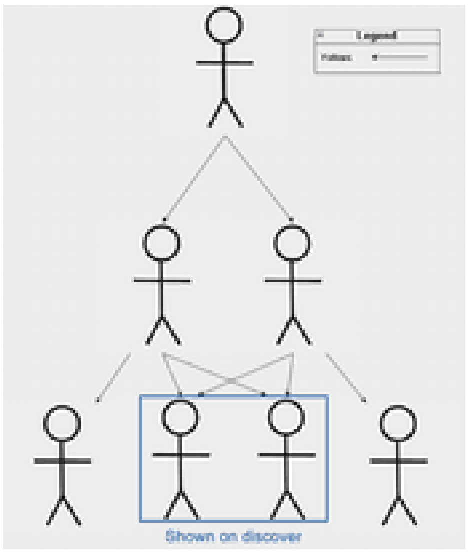


Figure 7: The discover page illustrated

**1.4.2.4 Logout** The **logout** tab turns the user into an unauthenticated user. This tab is only accessible to authenticated users.

**1.4.2.5 About me** The **about** me page displays information about the user. It:

- displays the GitHub **username** and **email** used for authentication.

- holds a **Forget Me!** button that **deletes all information** associated with the user, including sent cheeps.
- displays a list of all **cheeps written** by the user.
- displays a list of all **users followed** by the user.
- displays a list of all **users following** the user.

The about me tab is only accessible for authenticated users, and is only privately accessible.

**1.4.2.6 The cheep box** The **cheep box** is a text entry field accompanied by a **Share** button to send any text entered as a cheep. The cheep box is available on the **public timeline** page and on the **my timeline** page, and only for authenticated users.

#### 1.4.3 Sending a cheep

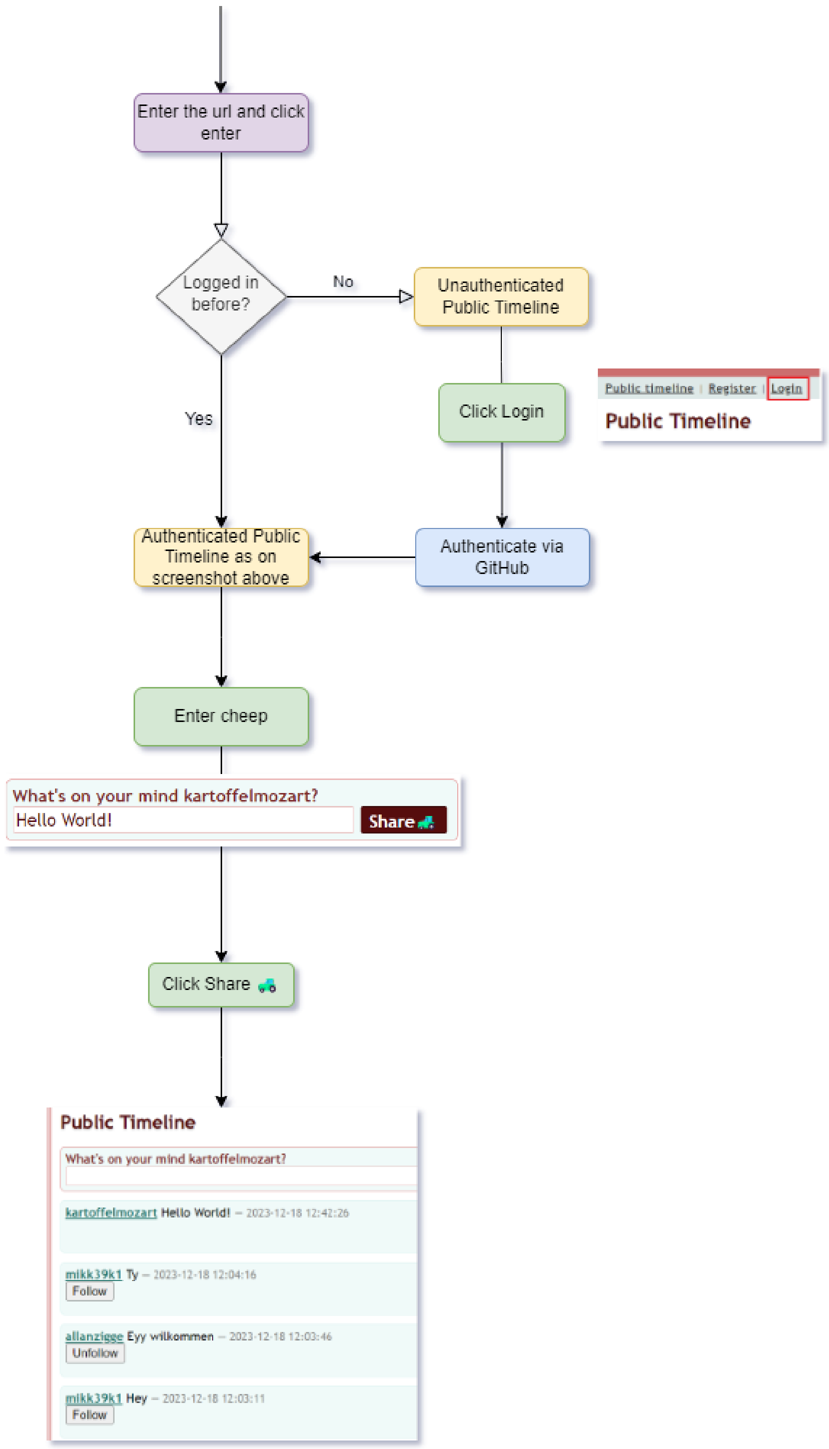
See figure 8 for a user flow diagram showing a typical scenario of a user logging in and sending a cheep.

## 1.5 Sequence of functionality/calls through *Chirp!*

In this section, we will examine the sequence of calls made during a user journey, from an unauthorized user to an authorized user. Figure 9 shows a client computer requesting an HTTP call in their internet browser by calling the root URL of *Chirp!*. The request is received by our Azure-deployed Web Application, which, through the use of Microsoft Identity, checks if the HTTP request has the needed access token to be authorized. Since the user is not authorized, the HTTP response is a limited version of *Chirp!*, as shown in the previous section *User Activities*.

The next step is for a user to press 'Register/Login.' This action triggers an ASP controller, 'Account,' to generate a URL pointing to the Azure Active Directory Business to Consumer Tenant's (Azure Tenant) SignUpSignIn user flow using the ID, secret, and information of our Azure Tenant provided by the connection strings from the appsettings.json. (Microsoft 2023a) Since we have chosen GitHub as the identity provider, our Azure Tenant sends a GET request to a GitHub OAuth App we have created in our GitHub repository. The OAuth App provides the user with a login dialog, which the user fills in. If the authentication is successful, the OAuth App provides our Azure Tenant with an access_token through the callback URL provided to the OAuth App. In the Azure Tenant, we have selected some claims for which information about the user is needed in our Web Application. (Github 2023)

When the callback URL with the access token arrives in the Azure Tenant, it checks if the access_token have the necessary claims. Since we need the user's email, and GitHub doesn't provide this, our Azure Tenant needs to provide it for us. If the user exists as a user in the Azure Tenant, then the user's email is


Enter the url and click enter
Logged in before?
No
Unauthenticated Public Timeline
Click Login
Public timeline | Register | Login
Public Timeline
Yes
Authenticate via GitHub
Authenticated Public Timeline as on screenshot above
Enter cheep
What's on your mind kartoffelmozart?
Hello World!
Share
Click Share
Public Timeline
What's on your mind kartoffelmozart?
Follow
Unfollow
Follow

kartoffelmozart Hello World! — 2023-12-18 12:42:26

mikk39k1 Ty — 2023-12-18 12:04:16

allanzigge Eyy wilkommen — 2023-12-18 12:03:46

mikk39k1 Hey — 2023-12-18 12:03:11

Figure 8: Sending a cheep user flow diagram

provided there, and it can be fetched from there. Otherwise, the Azure Tenant sends a dialog to the client where the user can fill in the email input. Now the Azure Tenant can create a user with sufficient claims and send the access token to the App service through the callback URL provided in the connection strings of the App Service.

Using Microsoft Identity, our web application now sets properties like name, email, and isAuthenticated to the values provided by the access token. Before accessing our *Chirp!* web application, the login redirects to an AuthorAuthenticator. Our user is at this point stored in the Azure Tenant but may not yet be stored in our SQL Server database, which we need to ensure.

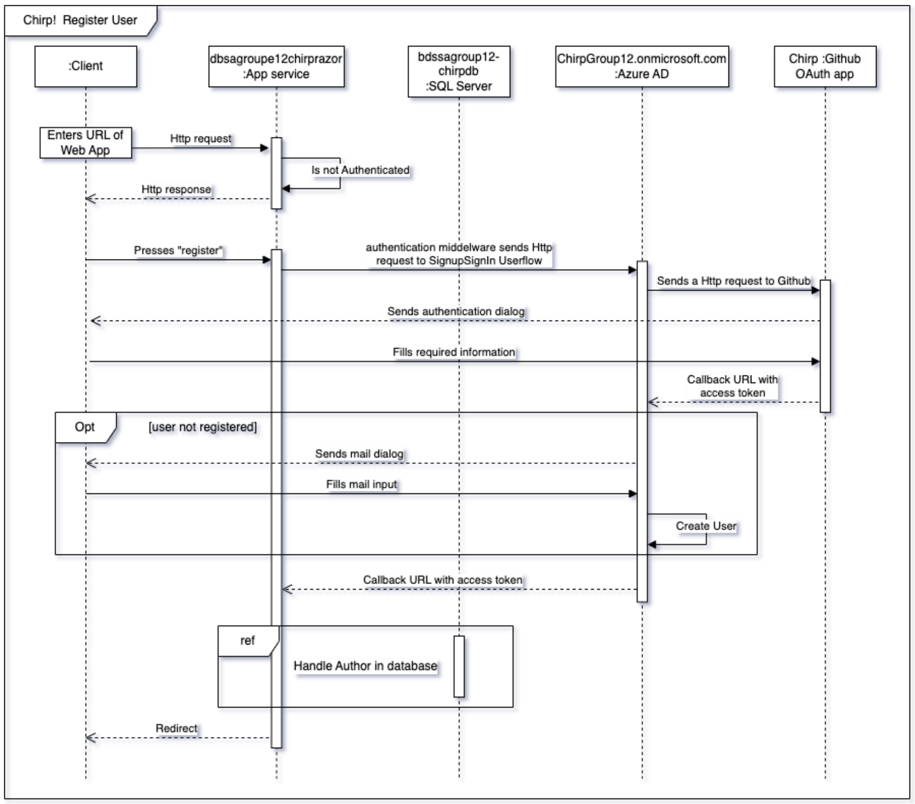

Chirp! Register User
:Client
dbsagroupe12chirprazor :App service
bdssagroup12-chirpdb :SQL Server
ChirpGroup12.onmicrosoft.com :Azure AD
Chirp :Github OAuth app
Enters URL of Web App
Http request
Is not Authenticated
Http response
Presses "register"
authentication middelware sends Http request to SignupSignIn Userflow
Sends a Http request to Github
Sends authentication dialog
Fills required information
Callback URL with access token
Opt
[user not registered]
Sends mail dialog
Fills mail input
Create User
Callback URL with access token
ref
Handle Author in database
Redirect

Figure 9: Sequence diagram of register as a User on *Chirp!*.

Therefore, as shown in figure 10, when landing on the AuthorAuthenticator page, a GET request is made to the PageModel, and a method on our Author-Repository is called to find the Author, associated with the users email. Since the AuthorRepository has the ChirpDBContext injected as a dependency in its constructor, it now calls GetFirstORDefaultAsync, with the users email as argument, on the ChirpDBContext's Authors DbSet. EF Core now uses the connection string provided to the ChirpDBContext from the Azure SQL Server, to make a TCP connection to the MSQL database and is now able to query it to find the Author record with the given email and return it back to the ChirpDBContext.

ChirpDBContext then returns the Author record, which the AuthorRepository returns as an AuthorDTO. If the AuthorDTO is not null, then the user is stored as an Author in the ChirpDBContext. If, however, it is null, then the Author needs to be created in the MSQL database.

The AuthorAuthenticate model now calls a CreateAuthor method on the AuthorRepository. With the Users name and email provided in the arguments it creates an Author entity and adds it to the ChripDBContext Authors DbSet. Now EF Core inserts it in the MSQL database.

The AuthorAuthenticate model concludes its GET method by redirecting the user to the public timeline page.

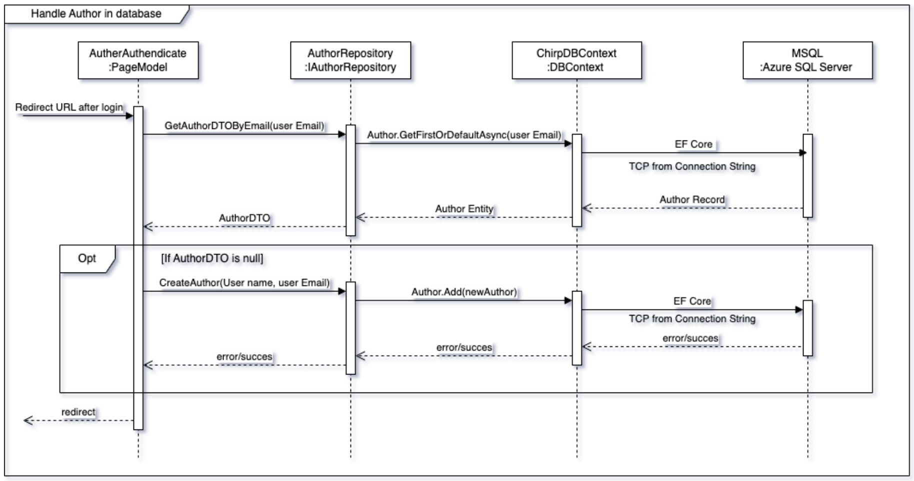

Handle Author in database
AutherAuthendicate :PageModel
AuthorRepository :IAuthorRepository
ChirpDBContext :DBContext
MSQL :Azure SQL Server
Redirect URL after login
GetAuthorDTOByEmail(user Email)
Author.GetFirstOrDefaultAsync(user Email)
EF Core
TCP from Connection String
Author Record
Author Entity
AuthorDTO
Opt
[If AuthorDTO is null]
CreateAuthor(User name, user Email)
Author.Add(newAuthor)
EF Core
TCP from Connection String
error/succes
error/succes
error/succes
redirect

Figure 10: Sequence diagram of creating a user on the Azure SQL Server if User does not exist.

## 2 Process

### 2.1 Build, test, release, and deployment

During development of the application, we have used GitHub workflows to automate some processes. It is defined in a YAML file and set to run on a trigger event. We have used one for Build and testing the application, one for creating a release on GitHub and one for deploying our .NET app to Azure and the schemas for our database. Workflows consist of one or more jobs, which have a sequence of steps that has to be executed. The diagrams below illustrate the steps and jobs of each workflow.

### 2.1.1 Build and test workflow

This workflow is triggered when a branch is pushed or a pull request is created. It locates the source code and sets up a .NET Core environment. Then it downloads the missing, if any dependencies are missing, builds the application and runs the test suite. This workflow is useful streamlining reviewing pull requests. See figure 11

### 2.1.2 Release workflow

Whenever a new tag is pushed into main with our format, the release workflow is triggered. Again the .NET Core environment is set up, and then we add the artifact to our release with softprops/action-gh-release. Initially this streamlining helped to make releases with executables on GitHub, whenever our main was given new tags. However, we have later switched to only release source code, as we have evolved *Chirp!* into a web application. We manually had to check if Main built and passed the tests before given new tags, but could have been included. See figure 12

### 2.1.3 Deployment workflow

This workflow consist of to jobs: 'BuildAndTest' and 'deploy'. Jobs can be run concurrently, but we need 'BuildAndTest' to run successfully, before we want to bother with deploying, hence the key word 'needs', which means we only run the 'deploy' when 'BuildAndTest' is done. The workflow can be triggered manually on GitHub or by push to main. Again it sets up a .Net Core environment, restores the dependencies, builds and runs our test suite. Create and upload application artifacts to the GitHub actions workflow system with the name '.net-app'. Then the new migrations are bundled together and uploaded with the name 'efbundle'. Then the second part of the workflow called 'deploy', takes care of uploading the new web-app to Azure and deploy the new migrations to the Azure database. See figure 13

## 2.2 Team work

We managed to implement all the functionality we aspired to implement during this project. A lot of features could have been added, but we decided on some core features that we felt like we could manage within the time frame, and managed those. Had we more time, we could have discussed which other features we might implement. Examples:

- A character count above the cheep box, denoting how many characters can yet be used, up to 160.
- Profile pictures, fetched from GitHub.

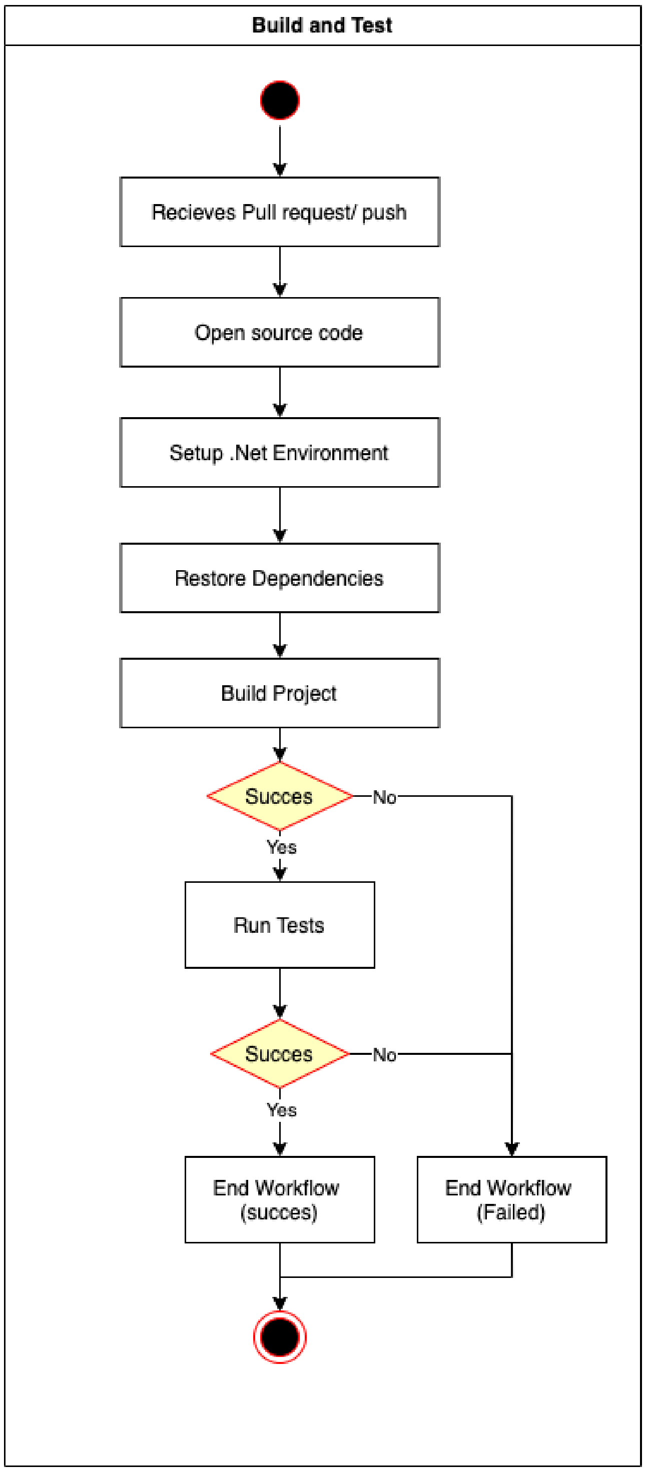


Figure 11: UML Diagram of build and test workflow

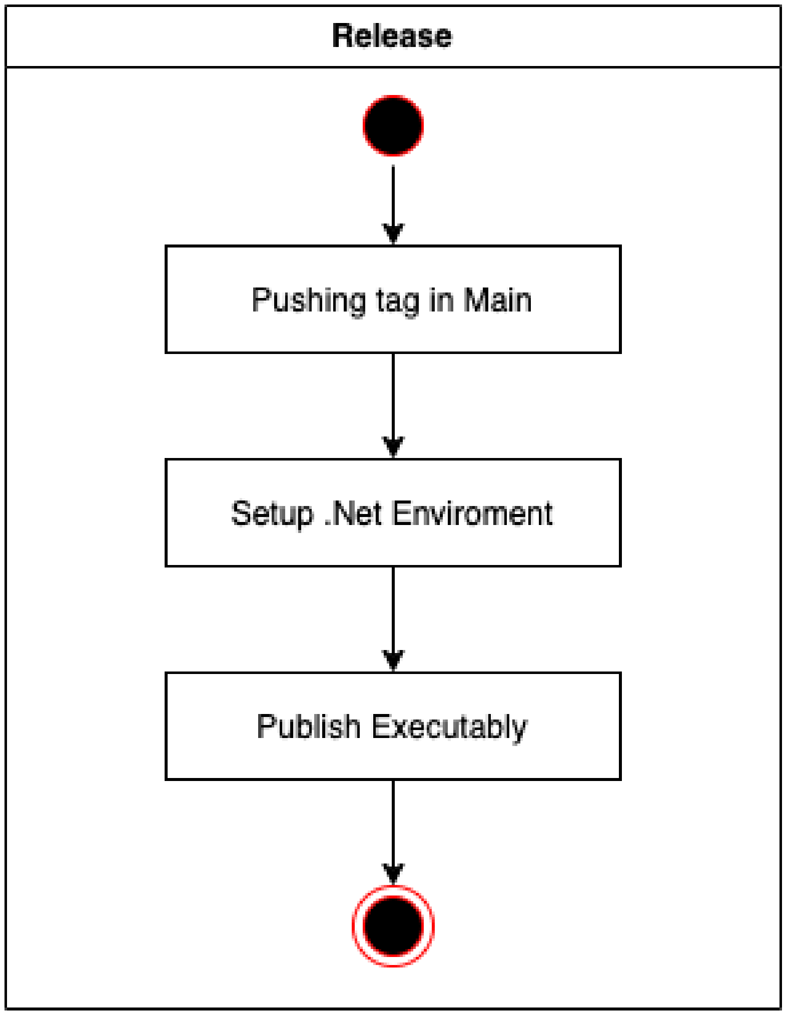


Figure 12: UML Diagram of the release workflow

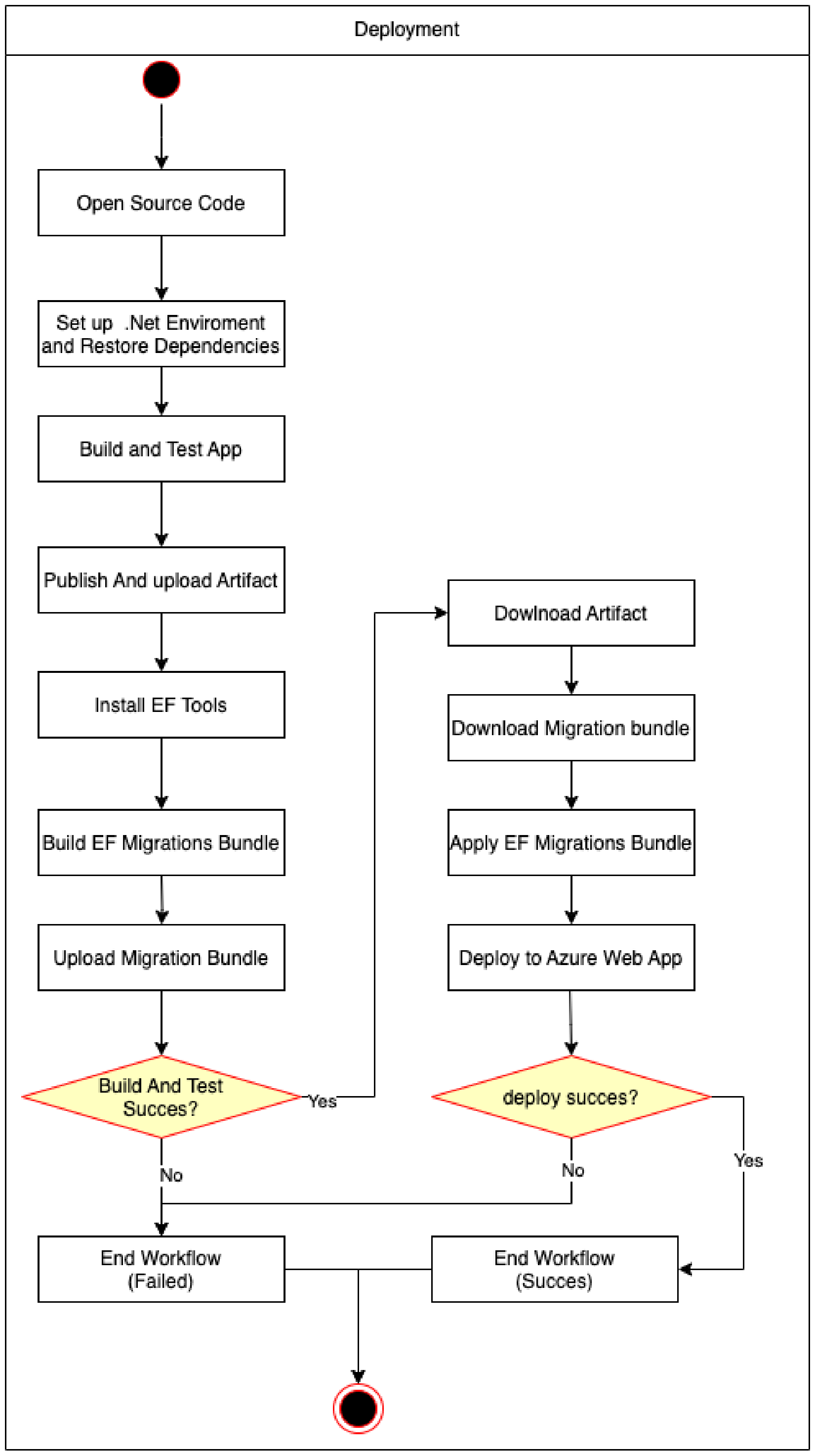


Figure 13: UML Diagram of the deploy workflow

- Functionality to reference other users in a cheep, and a page with the newest cheeps in which the user is referenced.
- Emoticon reactions to cheeps, such as happy face or tractor.
- Comments under cheeps.
- A link redirecting to a user's GitHub page.

**2.2.0.1 Work flow** We have implemented the code in short incremental steps, starting with a GitHub issue, describing a desired change in the code, who would benefit from this change, a more detailed description, and a definition of done. The issues are created to match the project description provided by our professors, or to match changes and details that we as a group or individually found relevant to implement. On creation, an issue is placed in one of three categories:

- To-do: A place for issues that contain an overall idea for the group to think on, not something that should be implemented as code.
- Backlog: An issue in the code that has no immediate solution; the group must talk about it and figure out how to turn this into an issue ready for implementation OR an issue that has a concrete solution, but should not be focused on yet, as the code base is not ready, or the issue is not relevant to work with at the moment
- Ready: Issues ready for a team member to assign themselves and begin implementation.

When a team member takes responsibility for an issue, he creates a branch linked to the issue with the name of the issue and starts on implementing the change described. Often we have worked on larger/more fundamental changes in groups of 2-3 or as the whole team (mob-/pair programming). We have (with varying consistency) written the code for an issue in smaller commits of working code. When the code for an issue is done, the code is pushed to the online repository on its branch. As a security measure a Github workflow tests the code being pushed, and gives a warning on the push if the tests does not pass. A pull request is made and the code is reviewed by one or more team members, different from the author of the code. The reviewer may then decide if more work is needed from the issue assignee, or if the change is ready to merge with the main branch. Upon merge, the issue is closed and moved to Done in our project board. See figure 14 for a user flow diagram showing the process of an issue.

## 2.3 How to make *Chirp!* work locally

### 2.3.1 Prerequisites

In order to run *Chirp!* locally one should follow these steps:

Start by cloning the Chirp repository by running the following command in your terminal. After this step you should have the entire Chirp repository in a directory called Chirp.

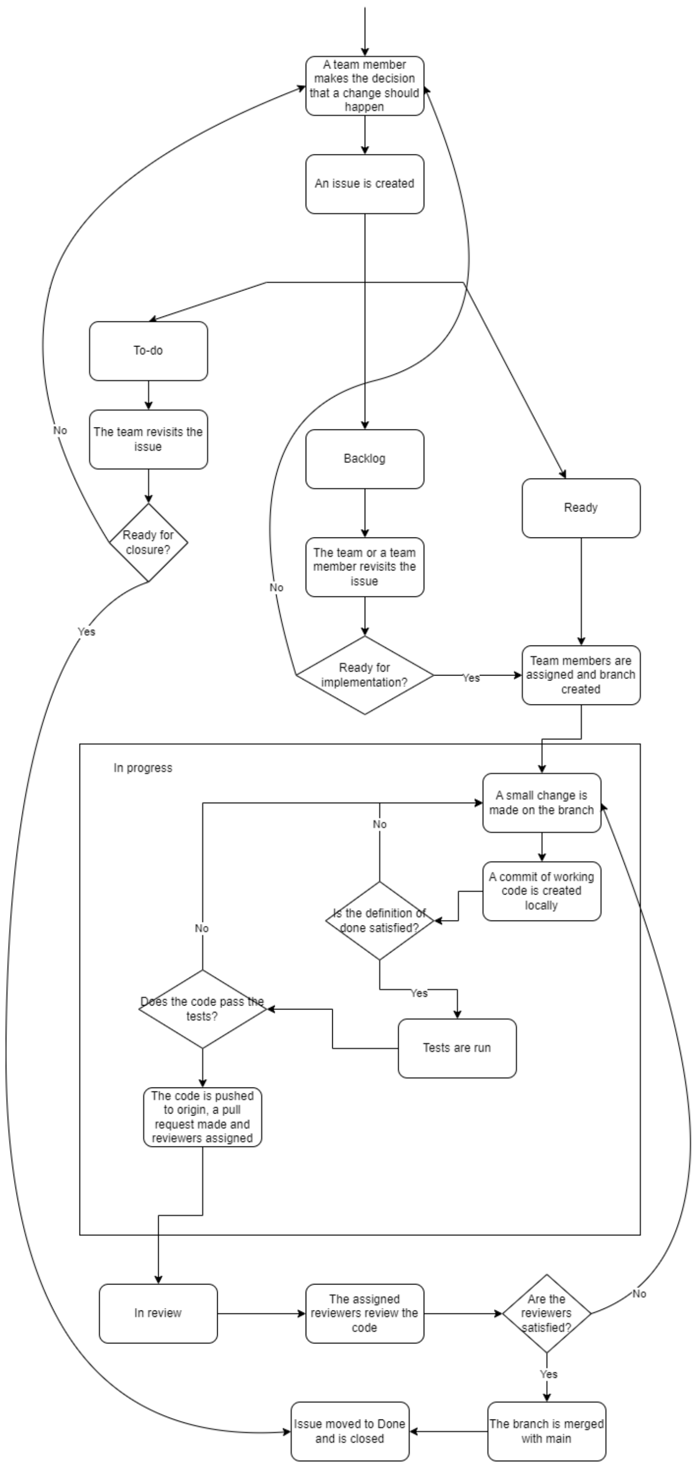

A team member makes the decision that a change should happen
An issue is created
To-do
The team revisits the issue
Ready for closure?
No
Yes
Backlog
The team or a team member revisits the issue
Ready for implementation?
No
Yes
Ready
Team members are assigned and branch created
In progress
A small change is made on the branch
A commit of working code is created locally
Is the definition of done satisfied?
No
Yes
Tests are run
Does the code pass the tests?
No
The code is pushed to origin, a pull request made and reviewers assigned
In review
The assigned reviewers review the code
Are the reviewers satisfied?
No
Yes
The branch is merged with main
Issue moved to Done and is closed

Figure 14: The life of an issue.

```
git clone https://github.com/ITU-BDSA23-GROUP12/Chirp.git
```

Next you move to your newly cloned repository's root directory:

```
cd Chirp
```

Before being able to build, test or run the *Chirp!* application you need to make sure you have docker installed, and then start an mssql using the following command:

```
docker run -e "ACCEPT_EULA=Y" -e "MSSQL_SA_PASSWORD=<YourStrong@Password>" \
   -p 1433:1433 --name sqlServer1 --hostname sqlServer1 \
   -d \
   mcr.microsoft.com/mssql/server:2022-latest
```

Next you need to add two user-secrets. One is used to establish connection to your newly created local database, the other is used to connect with our azure management app to handle deletion requests. Start by moving from the root directory to the Chirp.Web package:

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

Then run the following commands.:

```
dotnet user-secrets init
```

```
dotnet user-secrets set "ConnectionStrings:DefaultConnection":
   "Server=127.0.0.1,1433; Database=Master; User Id=SA;
   <YourStrong@Password>; TrustServerCertificate=true"
```

```
dotnet user-secrets set "ConnectionStrings:DeleteSecret":
   "n4x8Q~ULLEGdgNVvcSoSqwoHFJOAgyYfyroMQchJ"
```

Now you should be ready to build, test and run the *Chirp!* application.

**Note** We are aware that during normal circumstances one should never include user-secrets in source control. Given that this is educational purposes, we have chosen to include the above-mentioned secret, in order for the reader to be able to test deletion functionality locally.

### 2.3.2 Building the project

To build the entire project make sure you are in the root directory (Chirp) and run the following command:

```
dotnet build
```

A prompt should appear in your terminal, saying build success, 0 warnings, 0 errors.

If you wish to only build individual packages within the application, then first move to the given package before running the build command. E.g.:

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

```
dotnet build
```

### 2.3.3 Running the application

To run the *Chirp!* application locally you have to move to the Chirp.Web package (/src/Chip.Web). When in the Chirp.Web package, run the following command:

```
dotnet run
```

This should prompt a message in your terminal including the line: "info: Microsoft.Hosting.Lifetime[14] Now listening on: https://localhost:7028"

*Chirp!* is now running locally on your machine, and you can access it by going to: https://localhost:7028 in your browser of choice.

## 2.4 How to run test suite locally

### 2.4.1 Prerequisites

The test suite of *Chirp!* includes User Interface Test made with Playwright. In order to run these you need to make sure you have the needed browsers and dependencies installed.:

First move to 'Chirp/test/Chirp.Web.Test'. Then run the following command:

```
pwsh bin/Debug/net8.0/playwright.ps1 install
```

If pwsh is not available, you have to install PowerShell.

### 2.4.2 Running test suite

The test suite of *Chirp!* is divided in two test-projects. 'Chirp.Infrastructure.Tests' and 'Chirp.Web.Test'. In order to run all test suites of the *Chirp!* application, you simply need to be located in the root directory (Chirp) and run the following command:

```
dotnet test
```

First this should run the UI Tests located in the 'Chirp.Web.Test' project, and should return a prompt with 3 tests passed. After that it will run the integrationtests from the 'Chirp.Infrastructure.Tests' project, which should return a prompt with 26 tests passed.

**Note** As mentioned the 'Chirp.Web.Test' project holds user interface tests made using playwright. Upon review of the test class in this project it will be clear that there are more tests than the 3 being run. This is because we have five UI tests where we use a GitHub 'test account' to test logged-in features, these have been a great help during development. The problem is that in order for them to run on a new computer, the GitHub 'test account' will have to verify the new device with an email code. Therefore, we have chosen to comment these test out, in order for the reader to be able to run our test suite. If you wish to run the test suite including these five test, then get in contact with any member of the team, and we will then help verify you device.

# 3 Ethics

## 3.1 License

It became clear to us that choosing a license is important to us as developers. This is to avoid legal issues regarding copyrights and to protect ourselves from liabilities. Also, it is always good practice to add a license to guide other developers to what they can do with our code. This is of course even more relevant if the application were to stay and generate profit, and in this case we would need to consider another license.

When choosing a license we generally have to choose between permissive(copyright) and protective(copyleft) licenses, with different protective grades. We are not a community where the code needs contributors, and we do not have a desire to ensure openness. In our group we agree that anyone can re-use our code, however we do not want that anyone has permission to patent our code. Therefore, we have chosen the MIT license because of its simplicity and permissive nature. The Apache license grants patent rights from contributors, which is what we do not want.

Our dependencies are all licensed by MIT or Apache 2.0, and therefore compatible with MIT. List of all our dependencies:

- .NET MIT
- Fluent validation Apache 2.0
- Entity Framework MIT
- ASP.NET core MIT

- Microsoft graph MIT
- Microsoft identity MIT

- Microsoft.data.sqlite MIT
- Azure Identity MIT

## 3.2 LLMs, ChatGPT, CoPilot, and others

A large level model is used for language understanding and generation. We have used ChatGPT and Co-Pilot in this project. We used these to expedite the coding in certain areas.

We tried asking ChatGPT when hitting an error that we could not figure out why we received. ChatGPT is often helpful with both finding but also explain what the error is to understand *why* we get the error. This has been great to get a better understanding of what we need to fix to make good and functional code.

Furthermore ChatGPT has proven to be a solid tool for discovering keywords and to figure out what documentation needs to be delved into. ChatGPT is also good at comprehending how components work together, whereas documentation for each part (being Azure, .NET or even GitHub) is excellent at focusing on the specific domain. When "integrating" these domains, ChatGPT has been helpful with sharing its insights and filled the gaps between documentation. This has definitely saved us time.

Co-Pilot were used on the fly, as we were coding it would suggest what we might write, and often it was right and expedited the coding. It was also helpful, as we could write a comment stating what we want and Co-Pilot will then suggest the code for this. We have always been careful when using these tools as they may be wrong, inaccurate, etc. and researched upon information it gave that we were going to use. The way these tool were used in the process, made the code writing a bit faster, and sometimes **way** faster to debug.

## References

Github. 2023. "Authenticating to the REST API with an OAuth App." https://docs.github.com/en/apps/oauth-apps/building-oauth-apps/authenticating-to-the-rest-api-with-an-oauth-app.

Microsoft. 2023a. "AccountController Class." https://learn.microsoft.com/en-us/dotnet/api/microsoft.identity.web.ui.areas.microsoftidentity.controllers.accountcontroller?view=msal-model-dotnet-latest.

———. 2023b. "Common Web Application Architectures." https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures.