

# *Chirp!* Project Report

ITU BDSA 2023 Group 11

Andreas Bartholdy Christensen anbc@itu.dk

Marcus Andreas Aandahl maraa@itu.dk

Villads Grum-Schwensen vilg@itu.dk

## **1 Design and Architecture of *Chirp!***

### **1.1 Domain model**

#### **1.1.1 ER Diagram**

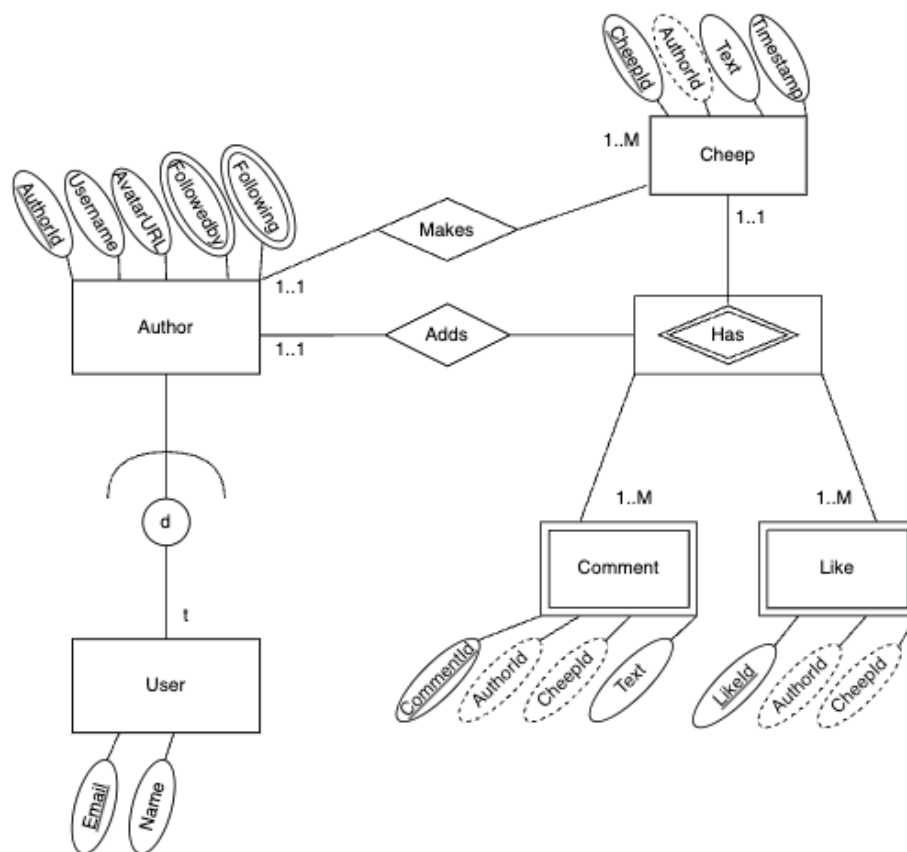


Figure 1: ER Diagram

The ER diagram is an illustration of the entities and their relations in the dataset. The author entity can make cheeps and add comments and likes while they're weak entities depending on a cheeps existence.

## 1.2 Architecture — In the small

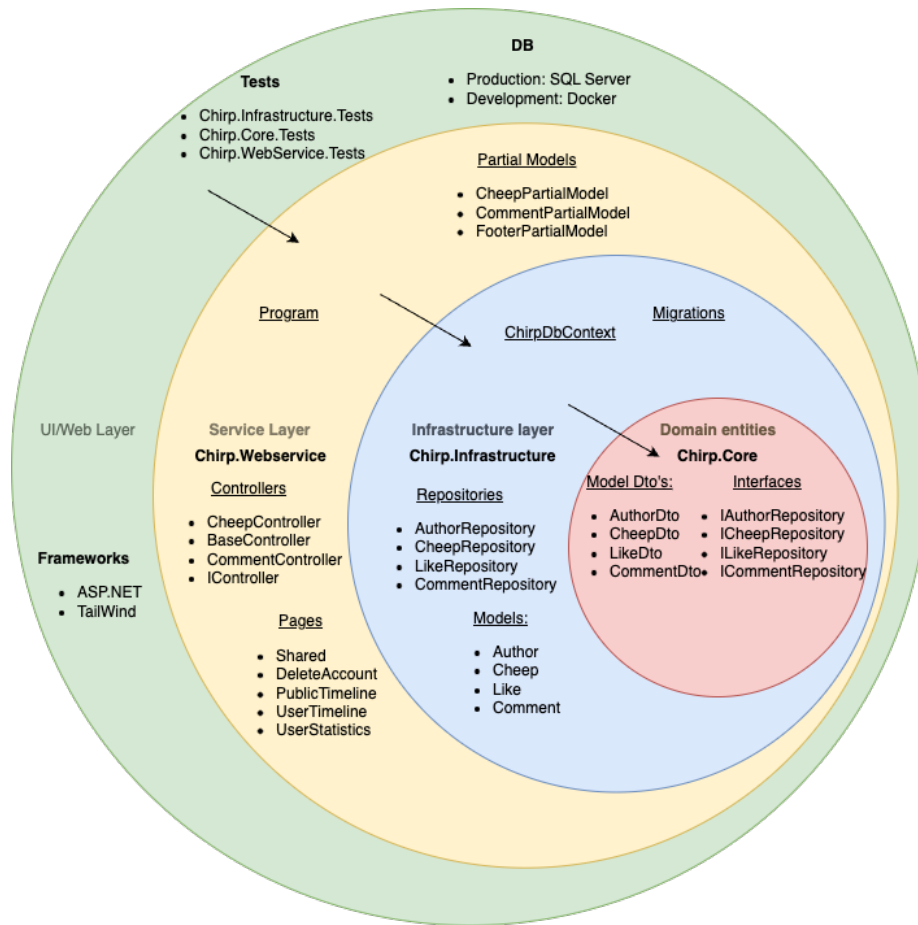


Figure 2: Onion Architecture

In the development of our *Chirp!* application, we used the Onion Architecture pattern. This architecture is composed of four integral layers: Domain, Infrastructure, Service and Web.

**Domain Layer:** Positioned at the core of the architecture, the Domain layer serves as a foundation of the application. The layer contains essential elements such as Data Transfer Objects (DTOs) and interfaces. This could also be the position of primary business logic if the application needed any.

**Infrastructure Layer:** This layer serves as a bridge between core domain logic and the practical implementation. The infrastructure layer consist of models that mirrors the DTOs and repositories built upon the interfaces. The layer also includes the DbContext for database interactions and Migrations scripts to version control the entities of the database.

**Service Layer:** Tasked to generate the user interface and functionality to improve user experience. The layer contains pages, controllers, partial models and the program file. The service layer translate operations and data from the infrastructure layer and the domain layer to facilitate user interactions and experience.

**Web Layer:** The outer layer in the architecture represents servers, frameworks and tests to support the application. This layer is crucial for deploying the application in a web and development environment.

Our approach of layering the application ensures that *Chirp!* is built on a solid architectural foundation that increases extendability and sustainability for future development.

### 1.3 Architecture of deployed application

#### 1.4 User activities

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

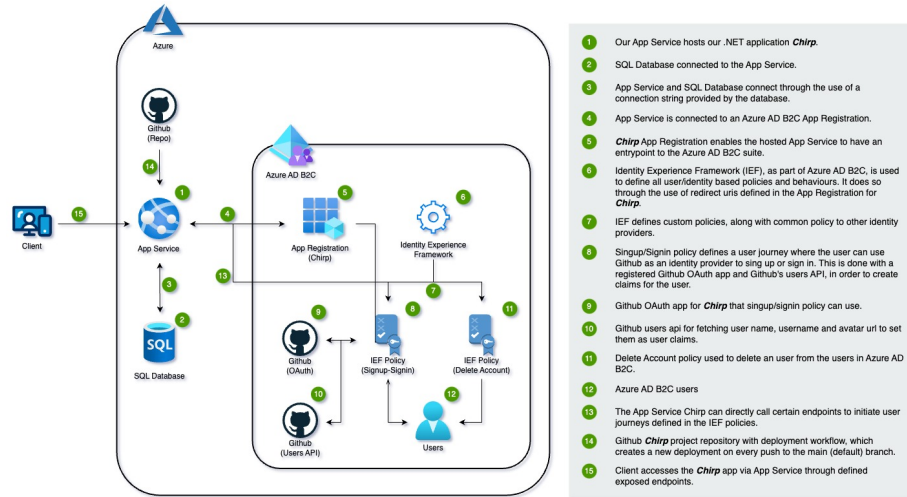


Figure 3: Cloud Architecture

The sequence diagram illustrates the functionality of the *Chirp!* application. The diagram contains 5 boundary objects; PublicTimeline, OAuth Login, User-Timeline, AboutMe and the database. These boundaries will be interacted with depending on the HTTP requests from the user. This particular sequence represents an unauthorized user that starts with a request to get the feed through a HTTP GET request, potentially for the first time. After authorization, all the features will be available which is represented by HTTP requests from the user following the timeline downwards. The sequence concludes with a final deletion of the user account, whereas all the accounts information is removed from the tenant through OAuth. Whereas cheeps, likes, comments etc. is deleted from the database and an unauthorized view of *Chirp!* will be displayed.

## 2 Process

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

### 2.2 Team work

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

#### 2.3.1 Clone Github repository

To make *Chirp!* work locally, first clone the repository with the following command if you have SSH keys set up for Github:

```
git clone git@github.com:ITU-BDSA23-GROUP11/Chirp.git
```

otherwise, if you don't have SSH keys set up for Github, the following command

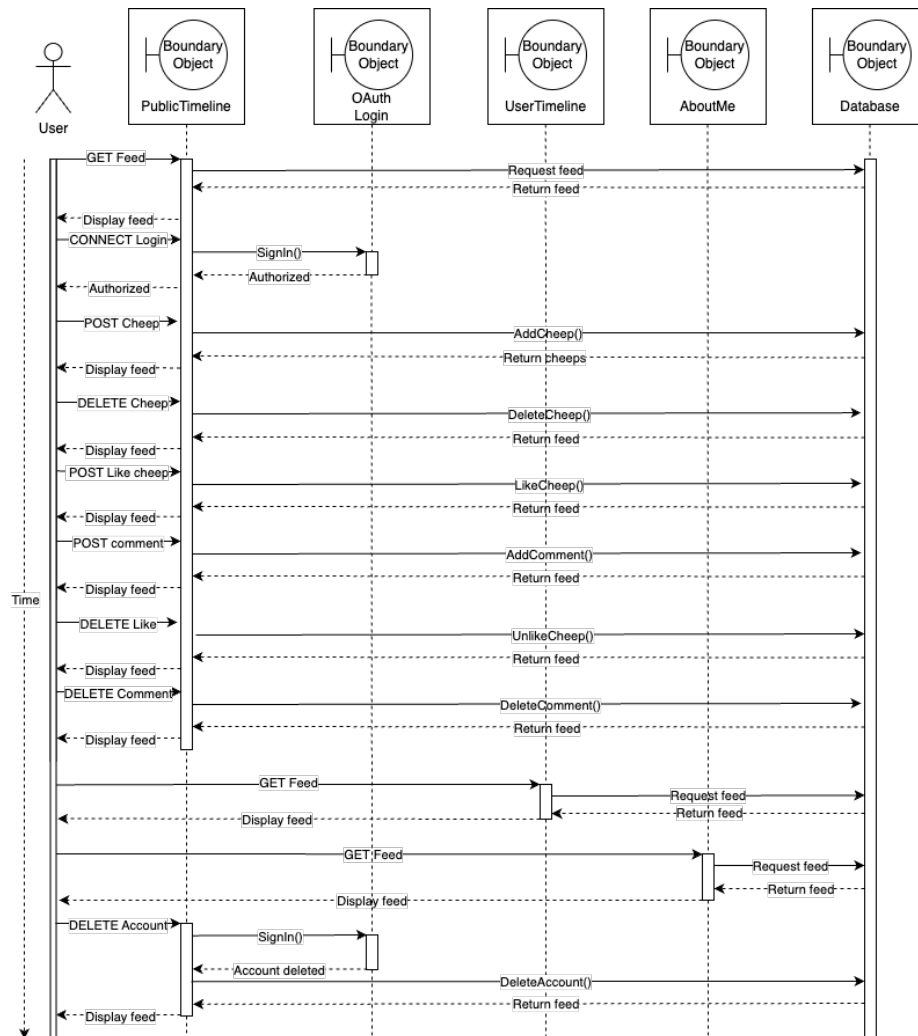


Figure 4: Functionality Sequence

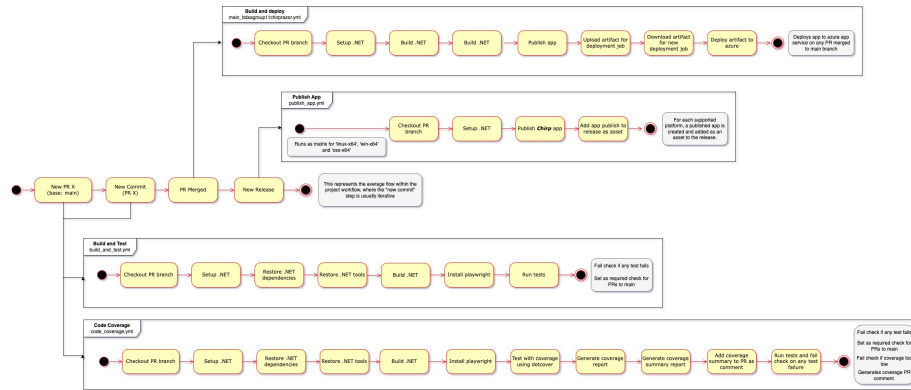


Figure 5: Workflows

can be used:

```
git clone https://github.com/ITU-BDSA23-GROUP11/Chirp.git
```

### 2.3.2 Install .NET

Thereafter, in order to set up the project, the main dependency you need is .NET 7.0. It can be downloaded from the from the *'Download .NET 7.0'* website. > *Make sure to download .NET 7.0 and not .NET 8.0, as **Chirp** will not work otherwise.*

### 2.3.3 Set up Sql Server with Docker

Here are the steps to set up the sql server with docker:

#### 2.3.3.1 1. Pull docker image Run the following to pull the docker image

```
docker pull mcr.microsoft.com/azure-sql-edge
```

#### 2.3.3.2 2. Run the image in a container Replacing <YOUR\_DB\_PASSWORD> with a strong password (requires 1 upper case, 1 lower case, 1 number, and no special characters), run

```
docker run -e "ACCEPT_EULA=Y" \
-e "MSSQL_SA_PASSWORD=<YOUR_DB_PASSWORD>" \
-p 1433:1433 --name azure-sql-server \
-d mcr.microsoft.com/azure-sql-edge
```

#### 2.3.3.3 3. Init secrets If not done yet, run the following to create a secrets file

```
dotnet user-secrets init --project ./src/Chirp.WebService
```

**2.3.3.4 4. Add DB password secret** Add the DB secret by running the following command, replacing <YOUR\_DB\_PASSWORD> with the strong password you generated earlier

```
dotnet user-secrets set "DB:Password" "<YOUR_DB_PASSWORD>" \
  --project ./src/Chirp.WebService
```

### 2.3.4 Run the project

After the project is set up, it can now be run.

When running the profile, make sure to run with the https profile. This can be done with the following command:

```
dotnet run --launch-profile https --project src/Chirp.WebService
```

We would however recommend running it through an IDE, such as Rider, which automatically detects launch profiles.

Furthermore, since the project needs to be run on HTTPS, a certificate will be needed.

In our case, using Rider as our IDE, a HTTPS certificate was added after being prompted when running *Chirp* the first time. However, a certificate can also be added with:

```
dotnet dev-certs https
```

## 2.4 How to run tests locally

### 2.4.1 Install .NET

Thereafter, in order to set up the project, the main dependency you need is .NET 7.0. It can be downloaded from the from the ‘*Download .NET 7.0*’ website. > *Make sure to download .NET 7.0 and not .NET 8.0, as **Chirp** will not work otherwise.*

### 2.4.2 Install Playwright

Tests have only one requirement, which is needed to run end-to-end tests: Playwright.

First of all, the powershell dotnet tool is needed, which can be installed with the following command:

```
dotnet tool install PowerShell --version 7.4.0
```

After running the tests the first time, and failing, the cause will be due to playwright not be installed. This can be solved by running the following command:



```
dotnet pwsh \  
  test/Chirp.WebService.Tests/bin/Debug/net7.0/playwright.ps1 \  
  install
```

Everything should now be set up in order to enable tests to run.

### 2.4.3 Run tests

To run tests, given the it is set up, simply run the following command:

```
dotnet test --verbosity normal
```

### 2.4.4 About the *Chirp* test suite

During the project, having a robust test suite, along with great coverage, was one of our focus points. Our tests are set up to reflect the structure of our source code, as to keep the structure coherent. This was further set into action with the use of a required workflow check, which failed if test coverage was under 60%.

In `Chirp.Tests.Core`, we have included anything our tests might have in common, as to keep our code DRY. This includes generated fake instances of our models (using Bogus), mocked repositories (using Moq), fixtures and application factories.

In `Chirp.Infrastructure.Tests`, we aim to cover `DbContexts`, `Models` and `Repositories`. These tests are primarily unit tests, covering the different functional components found in `Chirp.Infrastructure`, mainly based off of dotCover reports.

In `Chirp.WebService.Tests` lies a combination of unit, integration and end-to-end (E2E) tests. Whilst unit tests cover the functional aspects of our controllers and extension classes, our integration and E2E tests cover user flows an user might have foretaken.

## 3 Ethics

### 3.1 License

### 3.2 LLMs, ChatGPT, CoPilot, and others