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

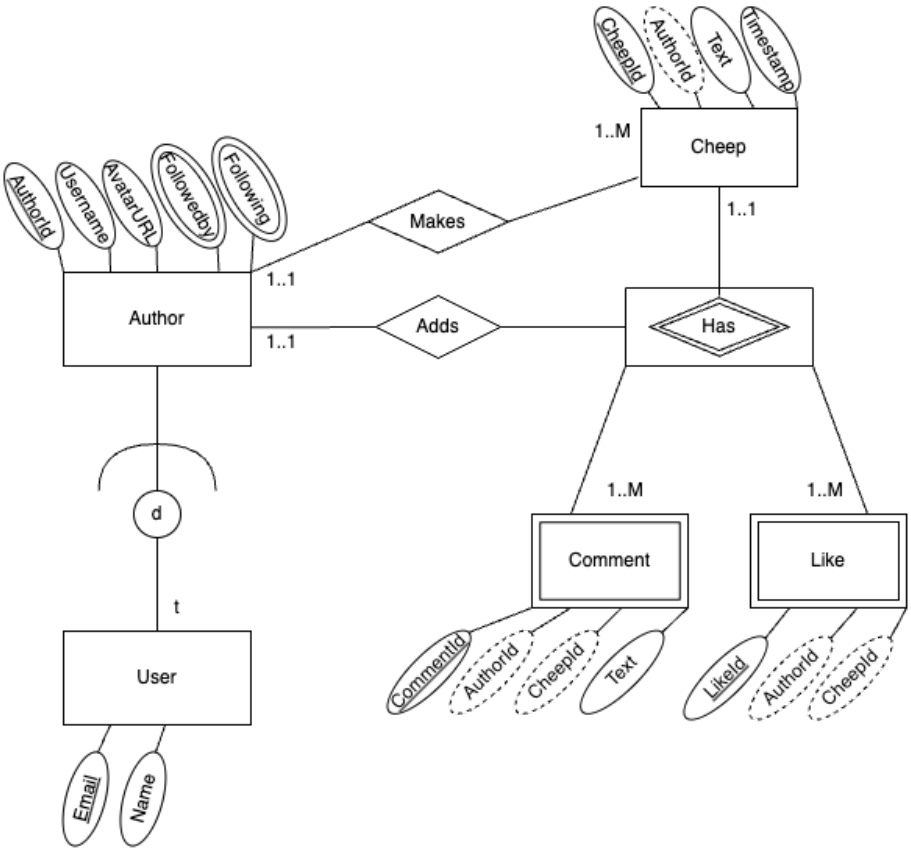

Figure 1: ER Diagram

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

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

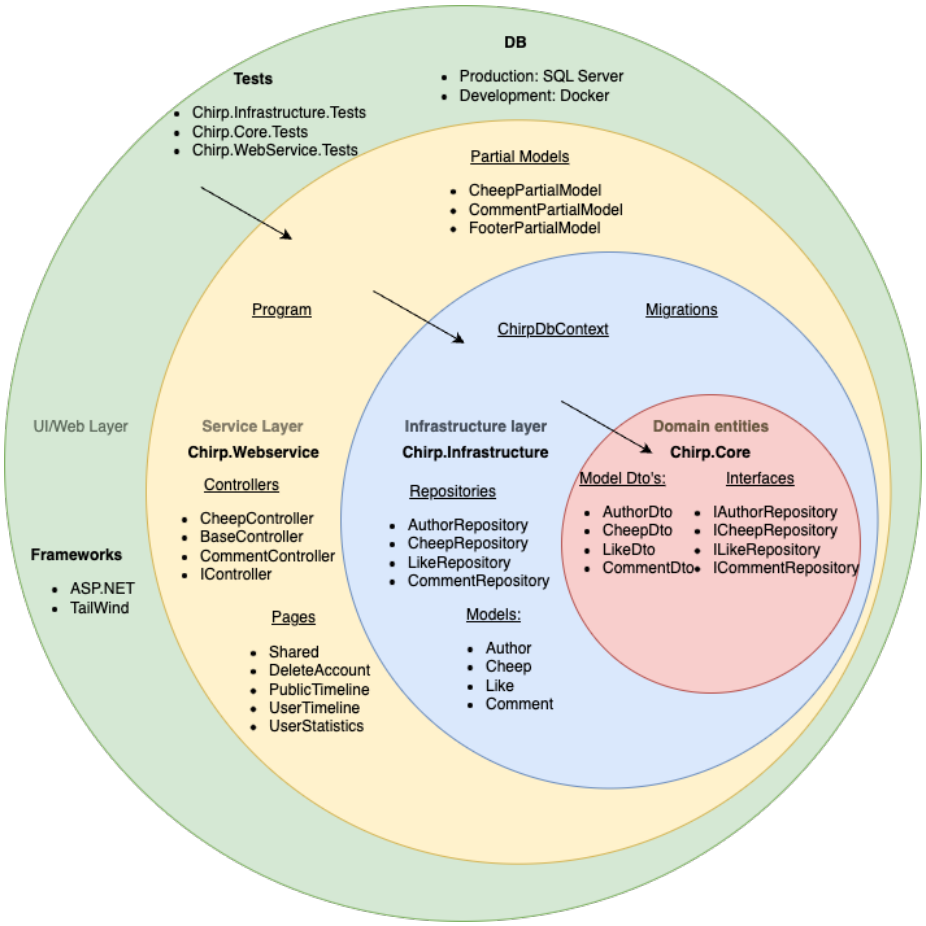

Figure 2: Onion Architecture

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

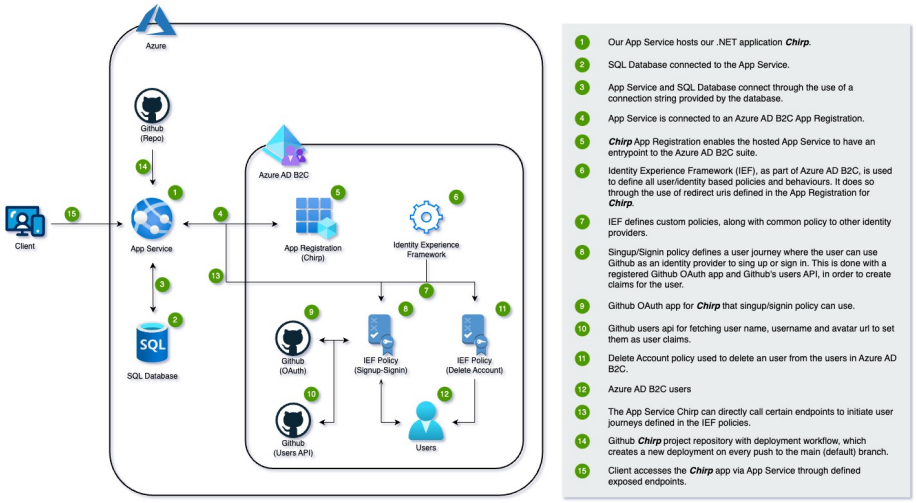

Figure 3: Cloud Architecture

This diagram shows the cloud architecture of how the clients and different Azure services communicate, in order to serve the said clients.

1.4 User activities

We have created the following diagrams to illustrate typical user activities/journeys through Chirp. We strove to create a user-flow that is smooth and functional. Therefore the register/login process is handled with OAuth and Github. This removes the need for a complicated registration (assuming the user has a Github account).

This can be seen in the diagrams after the user press the "Login" button. This starts the OAuth process. If the user has already been logged in to Chirp before, a Login press will simply handle the login and automatically redirect to the timeline without any further action from the user.

To ease the understanding of the diagrams, the following picture explains symbols and color definitions:

– INDSÆT BILLEDE AF DEFINITIONER –

– INDSÆT DIAGRAMMER AF USER ACTIVITIES –

1.5 Sequence of functionality/calls trough *Chirp!*

The sequence diagram illustrates the functionality of the *Chirp!* application. The diagram contains 5 boundary objects; PublicTimeline, OAuth Login, UserTimeline, 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

This diagram shows the usual flow starting from a pull request (PR) to the deployed application. As shown, we manage to build, test, release, and deploy our application through the use of Github Workflows/Actions.

2.1.1 Test and Coverage

Starting from when a PR is created with the default branch (`main`) as its base, the workflows `Build and Test` and `Code coverage` are run every time the PR is changed (fx with a new commit). These workflows, although similar, differentiate by the `Code coverage` workflow also providing a sticky comment on the PR

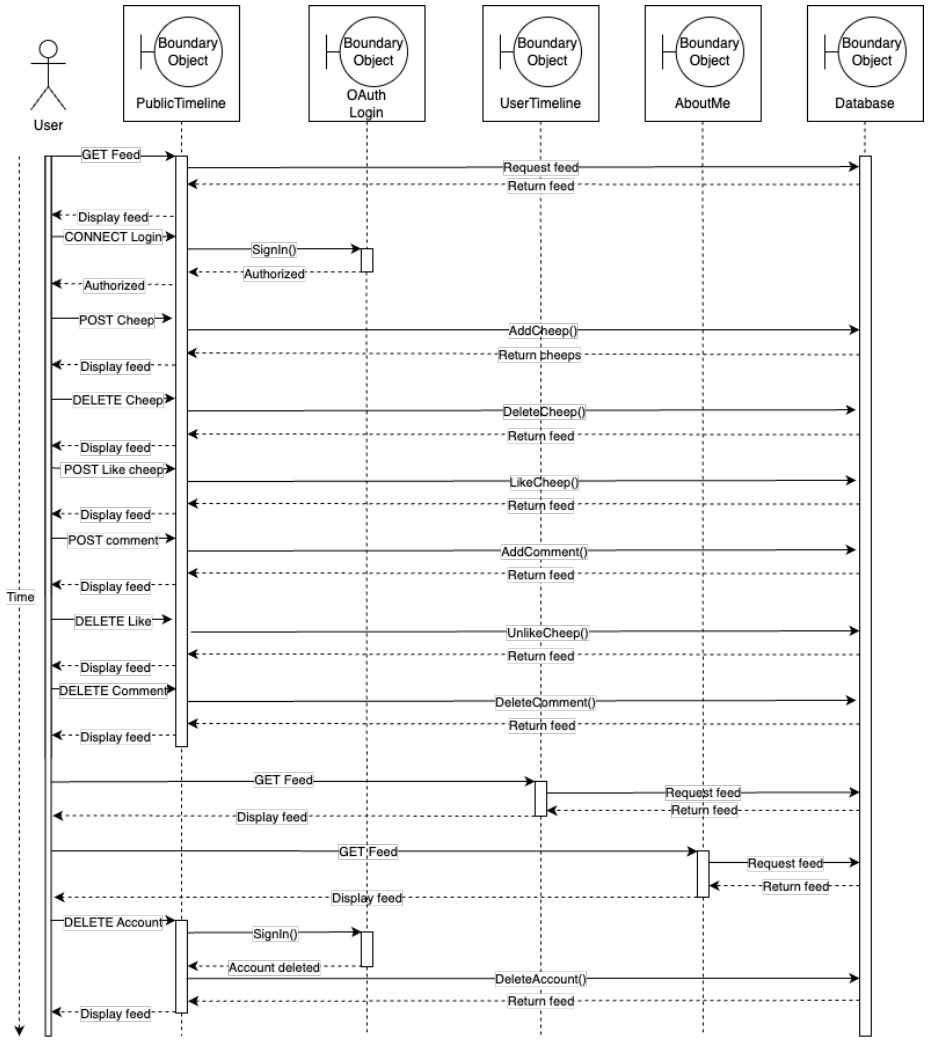

Figure 4: Functionality Sequence

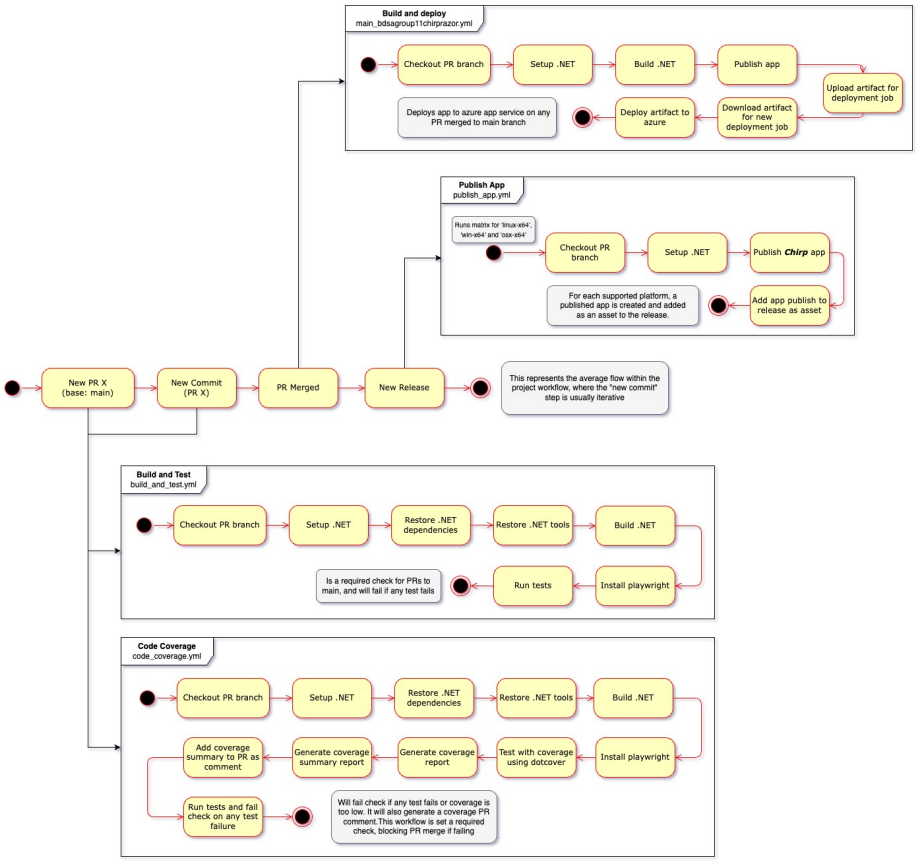

Figure 5: Workflows

with the code coverage from the tests. They both set up, restore, build and run tests. Together, they act as required checks, for which the PR merging is blocked if any of the tests fail, or if the code coverage is too low (under 60%).

2.1.2 Deployment

Once a PR has been merged to the main branch, a deployment is initiated through the `Build and deploy` workflow. The workflow starts by setting up .NET, build the project, publishes the application, creates an artifact, and then deploys it to Azure App Services.

2.1.3 Publishing

When a new release is created, the `Publish app` workflow runs. Running as a matrix with common runtime identifiers (RID), it sets up .NET, publishes the application, and appends the published application as a `.zip` asset for the release. In our case, it creates 3 `.zip` files (one for each RID), specific for the commit at which the workflow was run at.

2.2 Team work

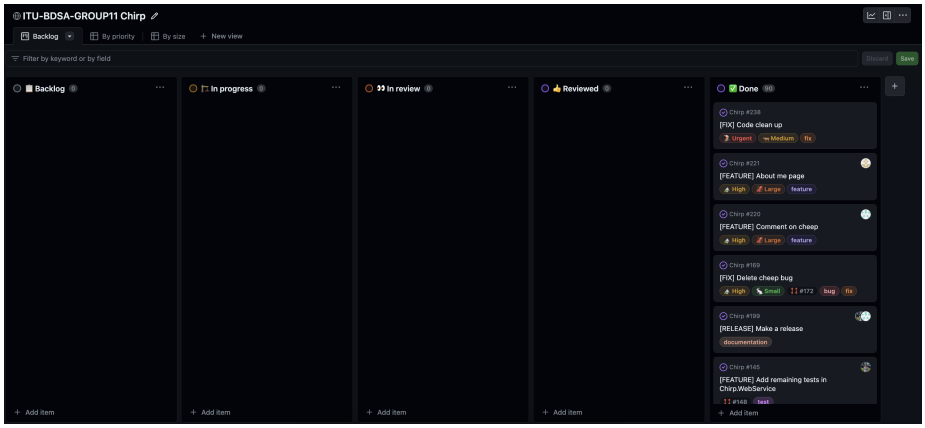

Figure 6: Project Board

The project board helps the team to organise issues with status, adding assignees and clarify acceptance criteria. When an issue is made, it is important that the creator of the issue is clear in the instruction and gives a perspective of why an issue is necessary. Additionally, conversations, questions and updates can be commented at each issue.

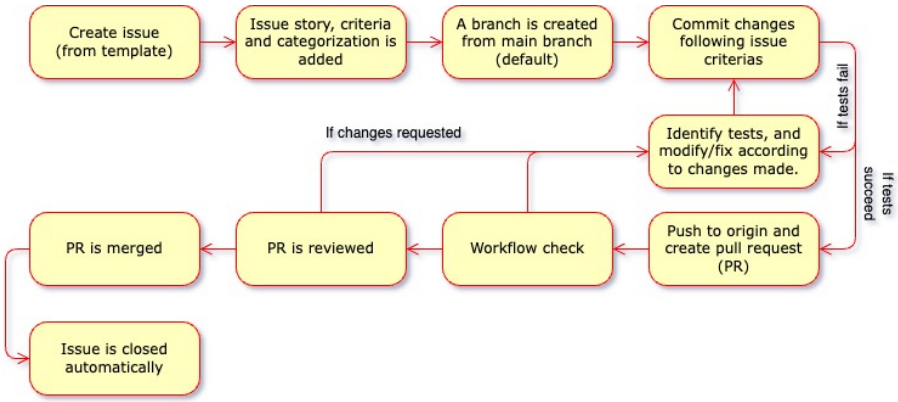

Figure 7: Teamwork Flow

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 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 a 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 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

We have licensed our application under the MIT License. This license provides access free-of-charge to any user using our product. The license is permissive and is provided "as is", which ensures no liability in terms of unwanted program behaviour, damage or claim.

The license is compliant with Tailwind, as Tailwind is also licensed under MIT (See src/Chirp.WebService/wwwroot/css/output.css for license specification at start of file).

3.2 LLMs, ChatGPT, CoPilot, and others

We have utilised AI tools in the development of Chirp, however only when we felt appropriate or necessary. Primarily we have used ChatGPT as a tool to troubleshoot code logic, if we have not been able to assess the faults ourselves.

We are using the Tailwind CSS Framework to quickly develop UI components for Chirp. Though we are able to write HTML and CSS ourselves, we have used the Vercel V0 tool to speed up development process, as it can generate HTML with Tailwind styling applied. Though this has been used extensively in the UI implementation, we found it appropriate as this course is not a UI-focused course, and we therefore did not wish to prioritize UI over more critical development issues.

V0 is currently in a closed beta, and can therefore not be publicly accessed.