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.

Architecture — In the small

In the Onion Architecture diagram bellow you'll see our applications. In the centre we have our core package. This is the lowest layer of the application. Then we move outwards for the layers with a larger impact, and end with our SQL-Server and razor pages, which interacts with our Azure application.

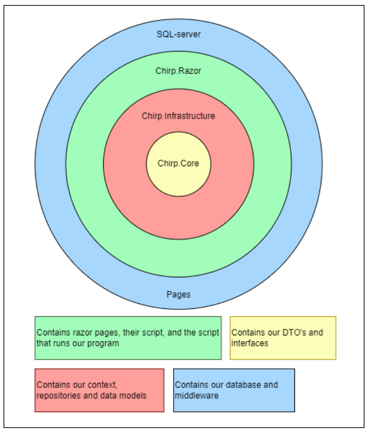

SQL-server
Chirp.Razor
Chirp.Infrastructure
Chirp.Core
Pages
Contains razor pages, their script, and the script that runs our program
Contains our DTO's and interfaces
Contains our context, repositories and data models
Contains our database and middleware

Fig. XX - Onion Architecture Diagram

These layers can be seen more detailed in our class diagram. There is one for each package, and they show everything needed to know about our classes. We've chosen to do this for more simplicity in reading the diagrams. The diagrams show each package and how the classes interact with each other. To see how they interact with the other layers, see OnionClassDiagram further down.

You will see in our repositories, that we're deleting the author at some point, this was a project demand. We had two possibilities; delete the user in the sense that they will no longer be traceable, that is make everything anonymous and delete their information, or we had the possibility of just deleting everything that the user ever touched. We chose to be sure that the user wouldn't come back complaining that their username/normal name still was in a cheep, so we deleted everything that they touched. This was also the easier approach since we could delete everything that contained that user's id or name, instead of altering everything.

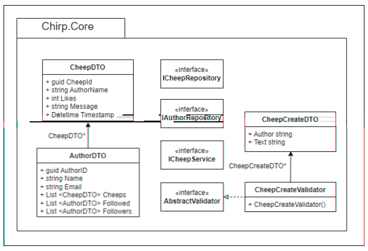


Fig. XX - ULM Class Diagram

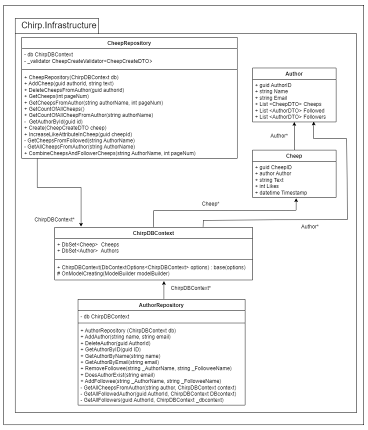

Chirp.Infrastructure
CheepRepository
- db ChirpDBContext
- _validator CheepCreateValidator<CheepCreateDTO>
+ CheepRepository(ChirpDBContext db)
+ AddCheep(guid authorId, string text)
+ DeleteCheepsFromAuthor(guid authorid)
+ GetCheeps(int pageNum)
+ GetCheepsFromAuthor(string authorName, int pageNum)
+ GetCountOfAllCheeps()
+ GetCountOfAllCheepFromAuthor(string authorName)
- GetAuthorById(guid id)
+ Create(CheepCreateDTO cheep)
+ IncreaseLikeAttributeInCheep(guid cheepId)
- GetCheepsFromFollowed(string AuthorName)
- GetAllCheepsFromAuthor(string AuthorName)
+ CombineCheepsAndFollowerCheeps(string AuthorName, int pageNum)
Author
+ guid AuthorID
+ string Name
+ string Email
+ List <CheepDTO> Cheeps
+ List <AuthorDTO> Followed
+ List <AuthorDTO> Followers
Author*
Cheep
+ guid CheepID
+ author Author
+ string Text
+ int Likes
+ datetime Timestamp
Cheep*
ChirpDBContext*
Author*
ChirpDBContext
+ DbSet<Cheep> Cheeps
+ DbSet<Author> Authors
+ ChirpDBContext(DbContextOptions<ChirpDBContext> options) : base(options)
# OnModelCreating(ModelBuilder modelBuilder)
ChirpDBContext*
AuthorRepository
- db ChirpDBContext
+ AuthorRepository (ChirpDBContext db)
+ AddAuthor(string name, string email)
+ DeleteAuthor(guid AuthorId)
+ GetAuthorByID(guid ID)
+ GetAuthorByName(string name)
+ GetAuthorByEmail(string email)
+ RemoveFollowee(string _AuthorName, string _FolloweeName)
+ DoesAuthorExist(string email)
+ AddFollowee(string _AuthorName, string _FolloweeName)
- GetAllCheepsFromAuthor(string author, ChirpDBContext context)
- GetAllFollowedAuthor(guid AuthorId, ChirpDBContext DBcontext)
- GetAllFollowers(guid AuthorId, ChirpDBContext _dbcontext)

Fig. XX - ULM Package Diagram of the Chirp.Infrastructure

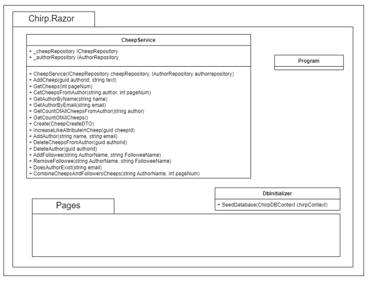

Chirp.Razor
CheepService
+ _cheepRepository ICheepRepository
+ _authorRepository IAuthorRepository
+ CheepService(ICheepRepository cheepRepository, IAuthorRepository authorrepository)
+ AddCheep(guid authorid, string text)
+ GetCheeps(int pageNum)
+ CetCheepsFromAuthor(string author, int pageNum)
+ GetAuthorByName(string name)
+ GetAuthorByEmail(string email)
+ GetCountOfAllCheepsFromAuthor(string author)
+ GetCountOfAllCheeps()
+ Create(CheepCreateDTO)
+ IncreaseLikeAttributeInCheep(guid cheepId)
+ AddAuthor(string name, string email)
+ DeleteCheepsFromAuthor(guid authorid)
+ DeleteAuthor(guid authorid)
+ AddFollowee(string AuthorName, string FolloweeName)
+ RemoveFollowee(string AuthorName, string FolloweeName)
+ DoesAuthorExist(string email)
+ CombineCheepsAndFollowersCheeps(string AuthorName, int pageNum)
Program
DbInitializer
+ SeedDatabase(ChirpDBContext chirpContext)
Pages

Fig. XX - ULM Package Diagram of the Chirp.Razor

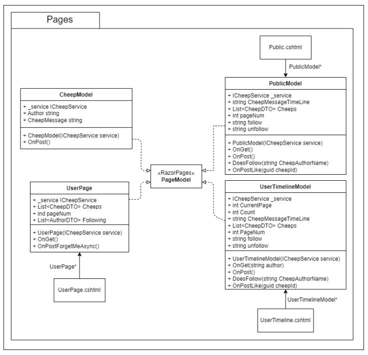


Fig. XX - ULM Package Diagram of the Chirp.Razor/Pages

The Onion Architecture (otherwise known as Clean Architecture), is great for having low coupling and high cohesion. When looking at the UML in the more specified onion diagram bellow, there is no unnecessary communication between scripts, having low coupling making the readability of the program better, even though some of the repositories contain a fair amount of methods. When moving outward you'll see the packages only use entities further in or in the same layer.

It is worth mentioning that the only way of interacting with the repositories is through their interfaces, which is an important factor in making sure the application has low coupling. The same goes for the CheepService, since every class that needs to access it uses information from the interface, and that interface uses from the other interfaces.

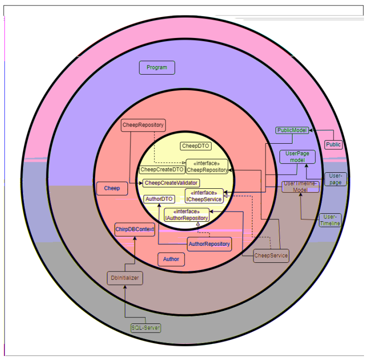


Fig.XX OnionClassDiagram

Architecture of deployed application

In the following figure a deployment diagram can be seen of our Chirp application.

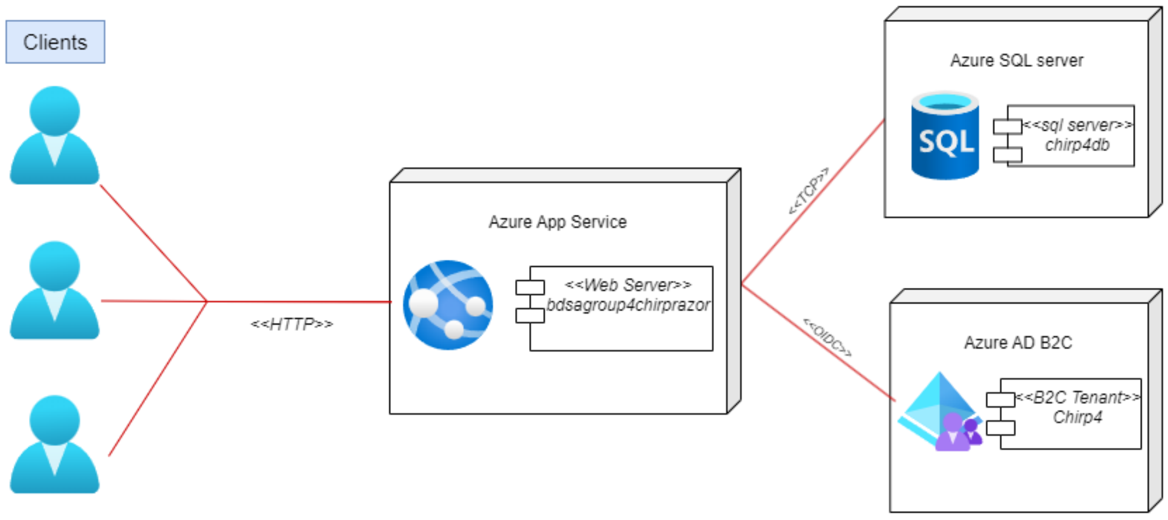


Fig.XX Deployment diagram

Chirp is a client-server application hosted on the Azure app service as a Web App. The web app is connected to a Azure SQL server where the database can be found. Furthermore the application makes use of a Azure AD B2C tenant for user-authentication. The different nodes means of communication is represented in the diagram.

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.

Sequence diagram

Sequence diagram

In Figure SQD1. A sequence diagram of an unauthorized actor. Henceforth, referred to as UA, accessing our project. It shows the UA sending the HTTP get request to receive the website. After the initial request, the Chirp.Razor starts to build the HTML. Here, an asynchronous object creation message is sent through the interface in the core and onto the repository. The repository returns the same for all actors sending this request. Using Linq, the repository inquires the SQL database for the 32 most recent cheeps.

The database sends the 32 cheeps to the repository. Which inserts each cheep into a CheepDTO before returning a list of 32 CheepDTOs. This list is sent back through the system, shown in Fig SQD1. Arriving in Chirp.Razor. It is weaved into the HTML, checking the if the user is Authorized. Before the page is returned to the UA.

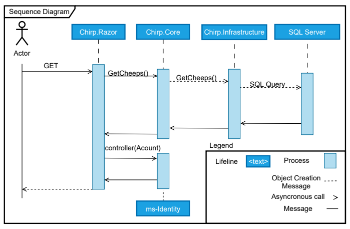


Fig. SQD1 - Sequence diagram for an unauthorized user

Figure SQD2. Show a known actor accessing our site, logging in and sending a Cheep. The first Get request is the same as seen in Fig SQD1. It deviates during the authentication step as the actor presses the login link. As they log in, Microsoft Identity redirects them to Azure OIDC. Which then redirect to GitHub.

After the actor has logged in, GitHub sends a token back to being logged on Azure. Their token is in the URL. With it confirmed, the Razor page HTML Will change.

Then the authorized user fills out the desired cheep and Chirps it. When that happens, Chirp.Razor constructs a CheepDTO and sends it through the core, where it is validated and sent to the repository. Afterwards it is committed to the database granted that Validation confirms.

Then, confirmation of success is sent back until the razorpage redirects to itself to reload.

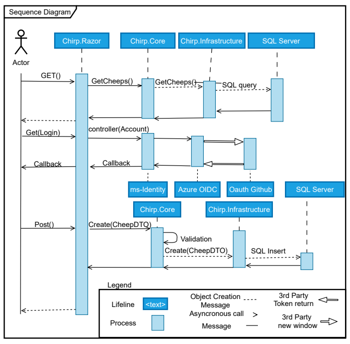


Fig. SQD2 - Sequence diagram for an unauthorized user

Process

Build, test, release, and deployment

GitHub workflows

To ensure the flow of the project, we use a tool developed by GitHub known as. GitHub Action, otherwise known as workflow. This will also entail when the workflows are activated and used.

Build and Test

The build and test workflow can be found in . The activity diagram shows how GitHub ensures what is merged into main. This workflow is run on a pull request every time a commit is made to the branch in the pull request. This is to ensure that main will still work by building the project with dotnet and tests made for the project. Because it runs the tests as well, it ensures that any incoming changes do not affect the functionality. If anything fails, it will stop and prevent the branch from merging into main.

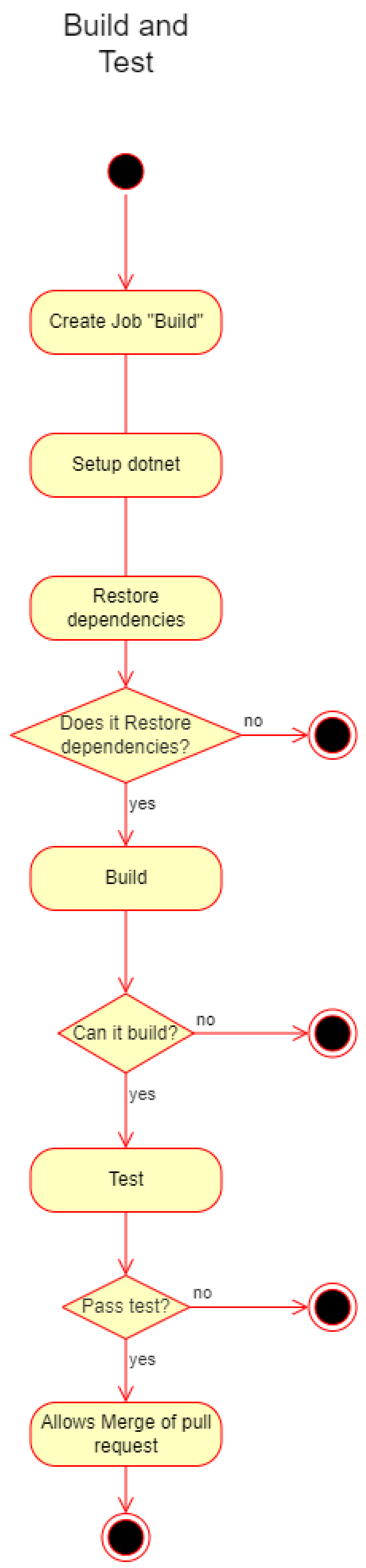
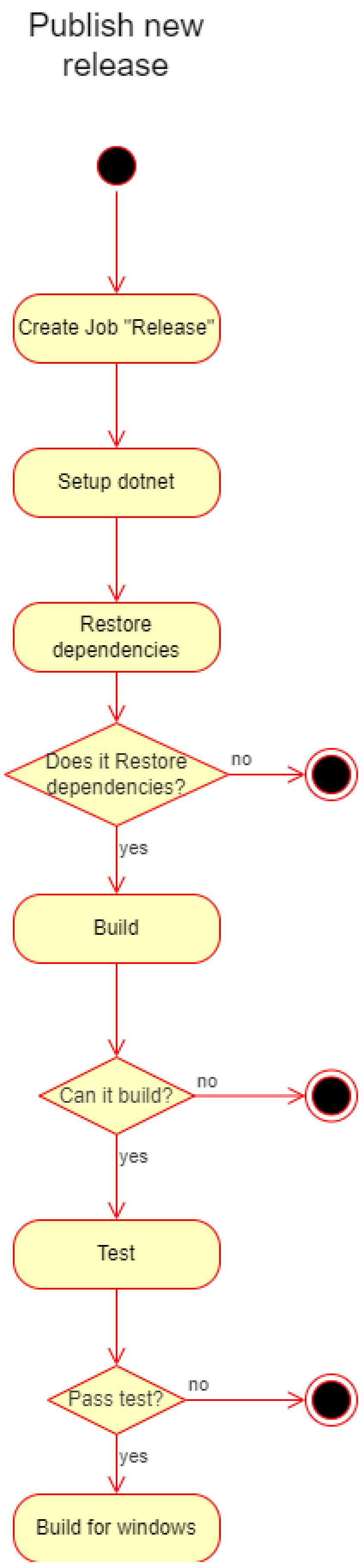


Fig.XX Build and test activity diagram

Publish and release

This workflow is made to automate the creation of a GitHub release when a tag is added (Appendix?). It will create a release of the tag. But first, the workflow builds a version for Windows, MacOS and Linux. After that, it will zip the files and add them to the release if a release was made.



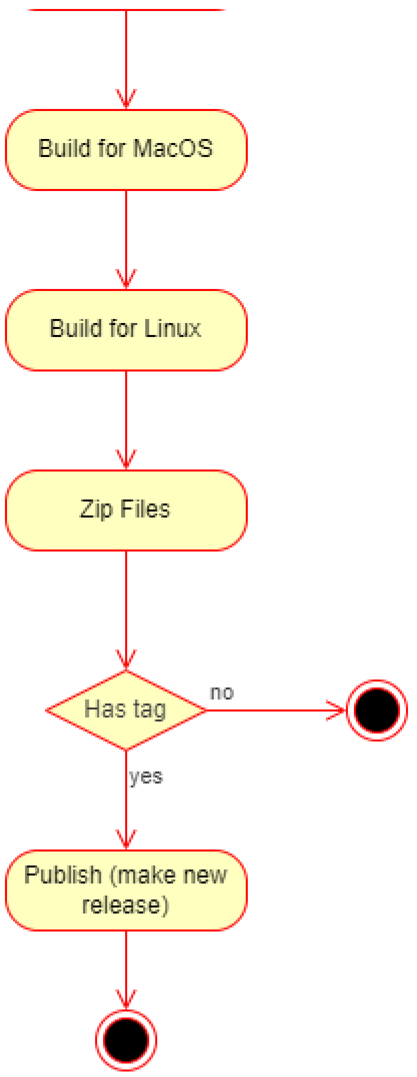
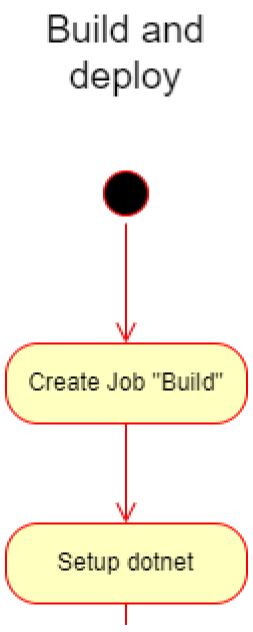


Fig.XX Publish new release activity diagram

Build and deploy

This workflow can be seen here (Appendix?). The workflow is made so it will build the program and run the "publish" command to build a version for Linux to be run on the Azure web app. After the publish command, it uploads the artifacts so the next job can use the files. The deploy job will download the artefact and use the files to deploy to our Azure web app.



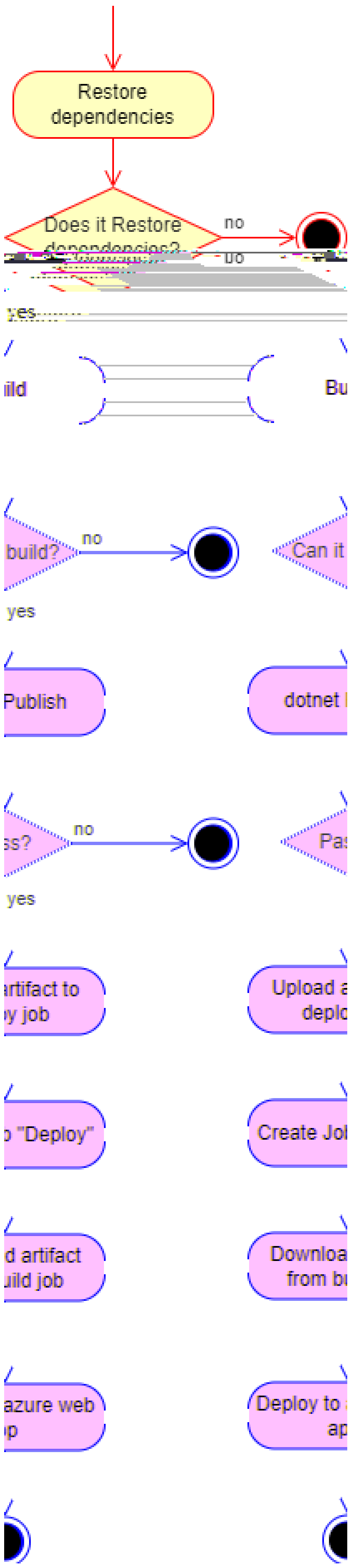


Fig.XX Build and deploy activity diagram

....

Before putting anything into the workflow actions, we create test manually to run on the computer with the "dotnet test" command. There has been created an activity diagram showing this. For most test we try to implement it going how we expect the method or feature to behave, and after we've concluded that it works, we create a test to challenge this method. By example we can look at the Create(CreateCheepDTO)'s tests in the unit tests.
This can be found in the infrastructure tests in the tests for Cheep Repository. We start by testing that what we want it to will work, and then we challenge it, by giving it some input that should throw validation exceptions. When we know both of these will pass, we can then move onto the workflows.

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.

How to make *Chirp!* work locally

How to make Chirp! work locally

prerequisites:

1. download .NET
2. IDE of your choice

1. Clone the repository

Follow this link: github.com/ITU-BDSA23-GROUP4

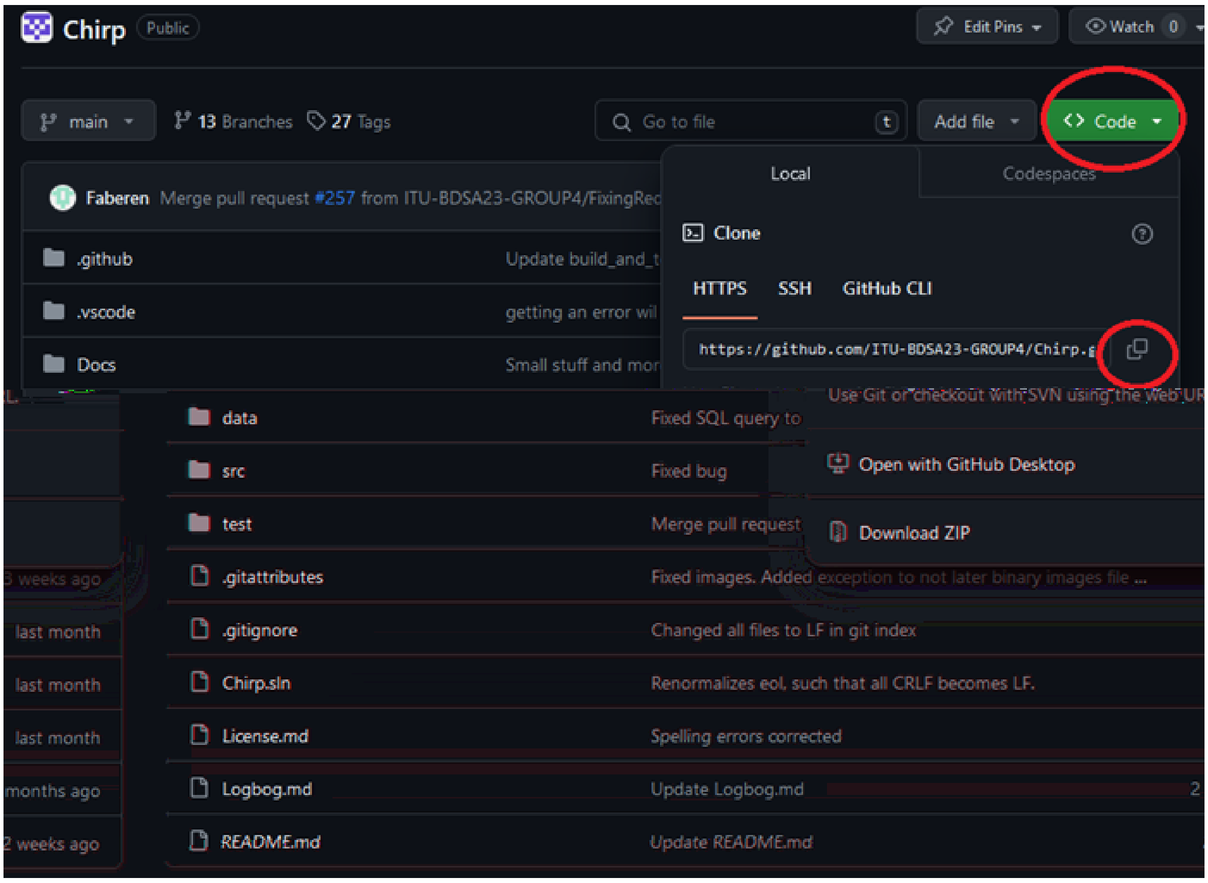


Fig.XX Cloning

copy the url and run the following command in your terminal where you want to clone the repository to.

```
git clone https://github.com/ITU-BDSA23-GROUP4/Chirp.git
```

2. Running and installing migrations

naviate to the root folder of the program, run the following command in your terminal.

```
--global dotnet-ef
```

naviagte to *Chirp/src/Chirp.Infrastructure*
delete all migrations file if they exists

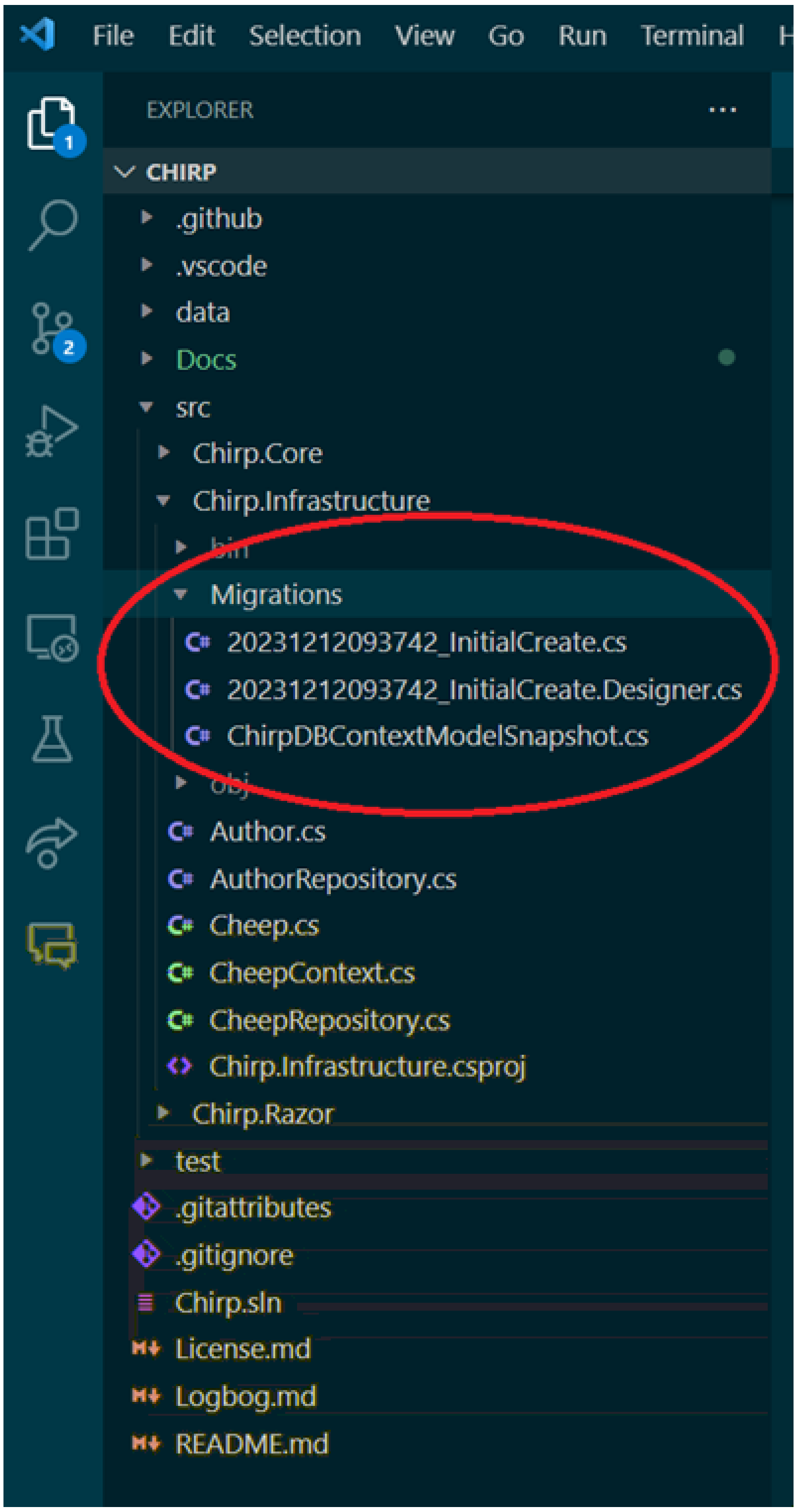


Fig.XX Deletion of migrations

then run the following command

```
dotnet ef migrations add InitialCreate
dotnet ef database update
```

3. Setting up docker

To setup the Docker container for development on own pc you need to run the following command: `docker run -e "ACCEPT_EULA=Y" -e "MSSQL_SA_PASSWORD=Admin123" -p 1433:1433 --name chirpdb --`

`hostname chirpdb -d mcr.microsoft.com/mssql/server:2022-latest`
After this the Container should have been created and a new Image can be seen in your Docker Desktop app. With the new lines of code in Program.cs it should create the database on the container. We can all just use the same command since the connectionstring is already made for this password. hostname and port.

Setup Database on docker

The last step is to create the database on the docker server to do this you are to navigate to the `Exec` on your new server.
To get there go to "Containers" and click on your container.

1. Go to the "Containers"
2. Open your Container ours is "chirpdb"

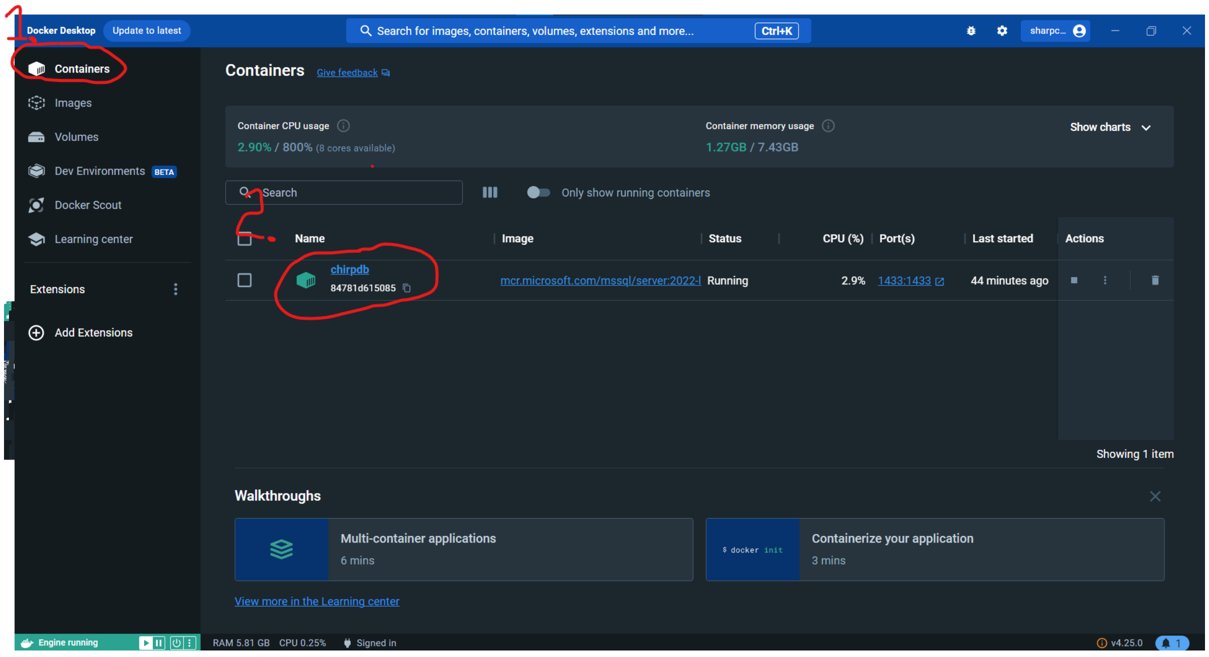


Fig.XX Container

3. Open `Exec`

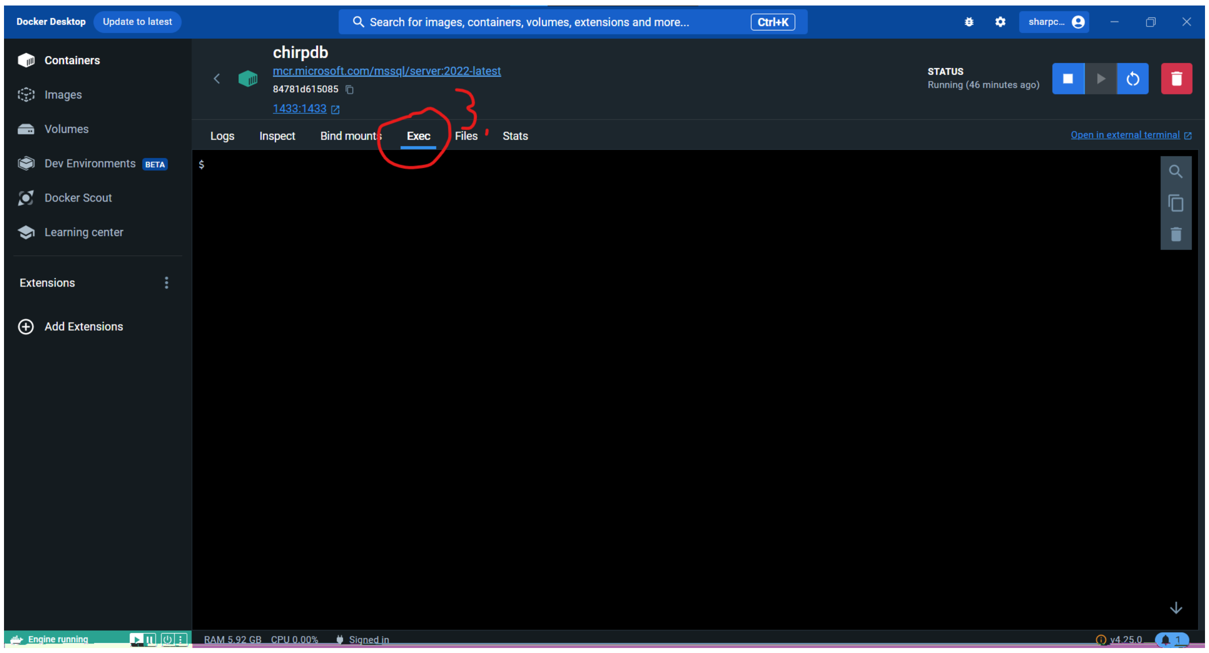


Fig.XX Exec

Her you can run bash commands on your container and look around the container.
We are here to use the MsSQL tool to make a database on this container. To do this we run this `/opt/mssql-tools/bin/sqlcmd -S localhost -U SA -P Admin123` (the `-U` is the user in our case we will just use SA which is System Admin and `-P` is the password for SA) this will gain access to the MsSQL tool. Here we can run SQL commands. Bare in mind that this is a diffrent tool the usual and have different commands.
The last part is to add the docker connectionstring to the user secrets. Navigate to src/Chirp.Razor and run command `dotnet user-secrets set "ConnectionStrings:ChirpDB" "Server=localhost,1433;Database=ChirpDB;User=SA;Password=Admin123;TrustServerCertificate=True;MultipleActiveResultSets=True;"`
You can also give your docker container another name if you want to.

4. Running the program

navigate to *src\Chirp.Razor* and run the following command

```
dotnet run
```

How to run test locally

The test suite of Chirp consists of 3 test folders each targeting their own part of the application, Infrastructure, Razor and playwright tests. All the tests are found in *Chirp/test/*

Infrastructure.Tests

No prerequisites are needed to accomplish the infrastructure test, simply cd into the *Chirp/test/Chirp.Infrastructure.Tests* folder in your terminal and run

```
dotnet test
```

Our Infrastructure tests targets our database and repositories, it creates an in memory database which all the test are run against.

```
var builder = new DbContextOptionsBuilder<ChirpDBContext>();
builder.UseSqlite("Filename=:memory:");
ChirpDBContext context = new(builder.Options);
_connection = context.Database.GetDbConnection() as SqliteConnection;
if (_connection != null)  //Takes care of the null exception
{
    _connection.Open();
}
context.Database.EnsureCreated();
```

what is tested

- AuthorRepositoryUnitTests
  This class targets our AuthorRepository. It performs unit tests for almost every method created in the repository with both correct and incorrect input. e.g. finding author by email or adding a follower.
- CheepRepositoryCreateUnitTests
  This class targets our CheepRepository. It specifically targets the methods around the creation of Cheeps. e.g. Adding a Cheep and checking if a Cheep is not empty
- CheepRepositoryUnitTests
  This class targets our AuthorRepository. It performs unit tests on liking a cheep. e.g. liking increases a Cheeps total likes.
- InMemoryDatabaseTests
  This class tests if the in memory database is created correctly which is crucial for the other classes since they all rely on it.
- RestrictedCheepsUnitTests
  This class targets the Cheepvalidator. It performs unit tests to check if a Cheep has the correct information, such as not being empty or over 160 characters and having a valid author..

Razor.Tests

To run the tests you need to setup and download docker. A complete guide for downloading and setting up docker correctly with our application can be found here After following the guide cd into the Chirp.Razor.tests folder and run the following command

```
dotnet test
```

what is tested

The razor tests consist of one class, **IntegrationTest.cs**. The class creates a local instance of our web application using the WebApplicationFactory class. With this we can test that our applications ui functions as we expect before we deploy it to azure. The test include, testing 32 Cheeps per page, ordering of Cheeps by date and the functionality of dynamic buttons.

Playwright.tests

To run the test first download playwright with the following command

```
pwsh bin/Debug/net7.0/playwright.ps1 install
```

This install various browsers and tools to run UI tests. The browser we use is chromium based.
if you run in to issues with the version of .net replace net7.0 in the command with the correct version
if you don't have powerShell installed follow these instructions Install PowerShell

After completing these steps you can run the test with:

```
dotnet test
```

When you run the test a chromium based browser will open and the first step tries to log in. Here the automation stops and the user has to log in through Github themselves. **No passwords are saved!** After this step is completed playwright will do the rest itself.

what is tested

The playwright test differs from the razor test in that it, mimics user behavior on our live website compared to the razor test which test locally. The test navigates through different pages and interacts with the website's functionality confirming that what it interacts with is as expected in the test.

Ethics

License

We chose the MIT license for our application, with the major reason being it's open-source nature towards programming-collaboration. Furthermore all the dependencies which we use in our application are also under the MIT license except one, which encourages the collaborative nature of the programming community. A list of all our dependencies and their licenses can be found here. One of our packages is under the Apache-license, which is fine since both are permissive licenses meaning they are able to be used together. This is also stated in our license file.

LLMs, ChatGPT, CoPilot, and others

In this section we will go over the AI help that has been through out the process of creating the project.

ChatGPT

ChatGPT is one of the AI's that we have used through out the project for smaller questions. Theses questions range from C# related code, refactoring of code, setup of docker or .NET console commands. With the help from the AI we can ask questions and get quick response that can help us in the right direction. There are somethings to consider when using ChatGPT the main thing will be the validity of the response since we can't guarantee that it's right.

Github CoPilot

Github CoPilot has been used through out the project to help with speeding up the process of writing code. This range from repetitive code to unit tests. This is where CoPilot shines with helping auto completion. With the recommendation that you get from CoPilot may not work since the AI can't know the full extend of the program.