Chirp! Project Report

ITU BDSA 2024 Group 26

Annam Bashir Chaudhry abch@itu.dk
Gutti Torvadal gutt@itu.dk Jovana Novovic jnov@itu.dk
Torbjørn Elias Rafael Johannsen toej@itu.dk
Vicki Hauge Bjørnskov vbjo@itu.dk

1 Design and Architecture of *Chirp!*

1.1 Domain model

Here comes a description of our 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 has three entities: Author, Cheep, and Notification. Using Entity Framework Core (EF Core), these entities are mapped to tables in an SQLite database and LINQ queries are used to interact with the database. - An `Author` represents the user of the application. It inherits from ASP.NET IdentityUser, which handles user authentication and authorization. Each author has a unique username and the ability to follow other authors. - A `Cheep` is a message that an author can post. A timestamp is added to each cheep when it is created as well. - A `Notification` is sent to all followers of a cheep's author, when it is posted. If an author is tagged in a cheep by starting it with `@<Username>` a notification is sent to the tagged author as well.

Each entity has a corresponding repository that is responsible for reading and writing to the database. Additionally, each entity has a corresponding DTO (Data Transfer Object) as to only transfer the necessary data to the presentation layer.

1.2 Architecture — In the small

Illustrate the organization of your code base. That is, illustrate which layers exist in your (onion) architecture. Make sure to illustrate which part of your code is re-

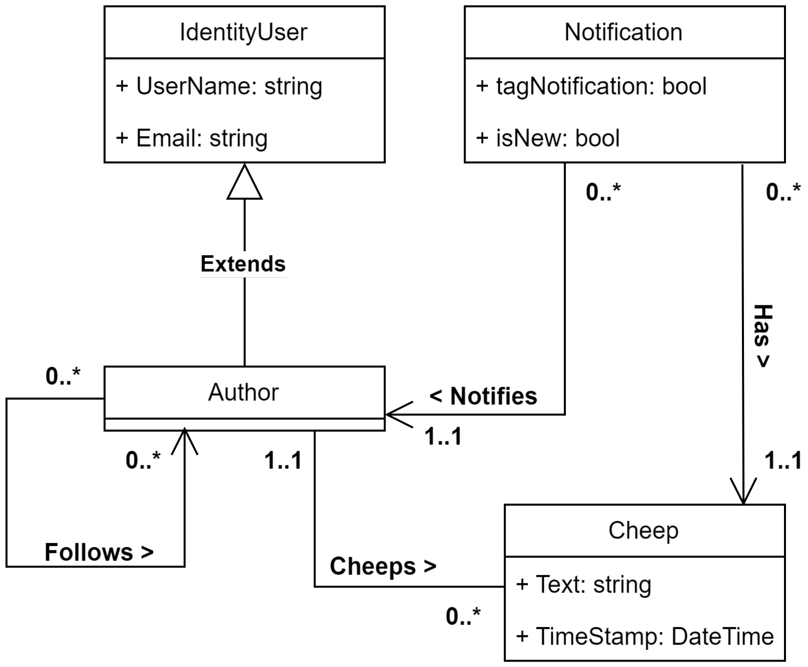
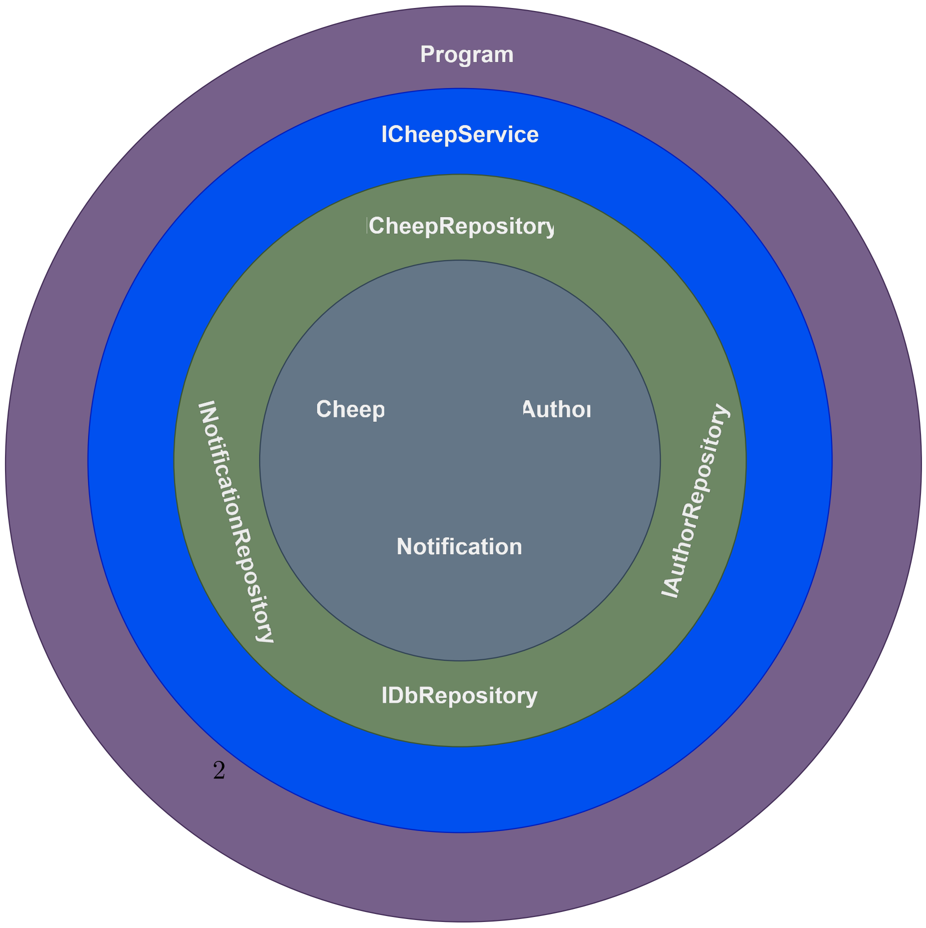


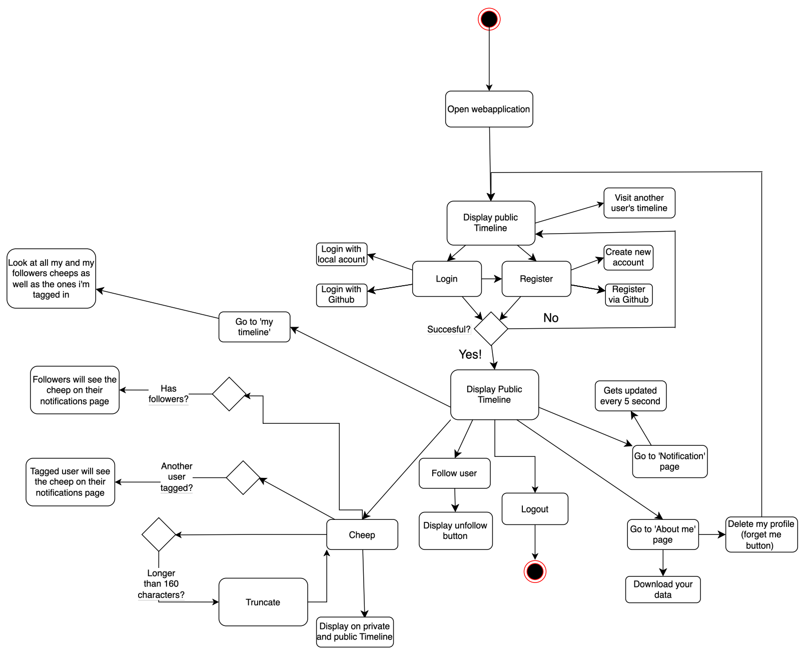
Figure 1: Illustration of the *Chirp!* data model as UML class diagram.



siding in which layer.

As the illustration shows, the *Chirp!* application is organized in an onion architecture. The onion architecture pattern has the benefits of making the code highly modular. The dependencies exclusively go inwards, which means that the inner layers are not dependent on the outer layers. This makes it easy to replace layer implementations, allowing for a high degree of flexibility and testability.

- Core:
  - At the core are the entities of the domain model. As the program uses EF Core, the entities are also tables in an SQLite database
- Second layer:
  - In the second layer, we have our repositories, which are responsible for reading and writing to the database. Each domain model gets one repository. Additionally, we have a DBRepository, which is responsible for general database operations, that are not tied to a specific domain model. Currently, it has two methods, one for seeding the database and one for resetting it.
- Third layer:
  - The CheepService resides in the third layer and is responsible for the business logic of the application. All the razor pages have a reference to an instance of the service.
- The outermost layer:
  - The presentation layer. It has the razor pages and the controllers. This layer is responsible for tying it all together as well as rendering UI rendering ## 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. ## User activities

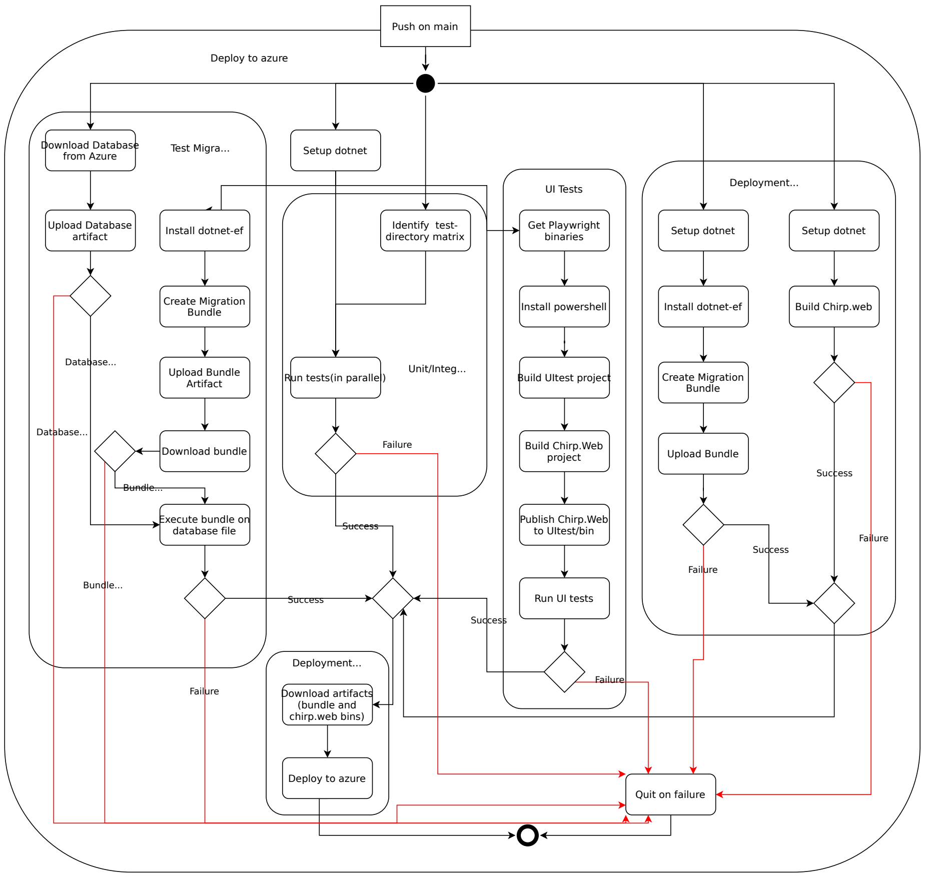

Open webapplication
Display public Timeline
Visit another user's timeline
Login with local account
Login with Github
Login
Register
Create new account
Register via Github
Succesful?
No
Yes!
Look at all my and my followers cheeps as well as the ones i'm tagged in
Go to 'my timeline'
Display Public Timeline
Gets updated every 5 second
Go to 'Notification' page
Followers will see the cheep on their notifications page
Has followers?
Tagged user will see the cheep on their notifications page
Another user tagged?
Follow user
Display unfollow button
Logout
Cheep
Longer than 160 characters?
Truncate
Display on private and public Timeline
Go to 'About me' page
Delete my profile (forget me button)
Download your data

The typical scenarios of a user journey through our Chirp! webapplication are displayed in the Diagram above. The potential user journey begins with opening the webapplication as a non-authorized user being only able to visit the public timeline, visit users from public timelines pages, login as an authorized user and register as such. In addition to the user journey in the illustration, the user can logout of the application from every page.

1.3 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.) # 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.



Note: I have taken the liberty of making the lines from negative conditions red, to make the diagram more readable considering that there are a considerable amount of points of failure. In addition, some repetitive steps(like checking out the repository, cleaning up ect) have been omitted

Note that the diagram doesn't include the release process, since it wasn't a part of the process at the time of writing(Essentially we let Chirp be a web-only application).

In our case a successful deployment requires 4 parallel processes to all succeed, where 3 of these are tests. In the diagram these sub-processes are marked within the larger *Deploy to Azure* process. They are, from left to right - Test Migration: - This workflow ensures that any possible migrations the deployment might want to apply to the production database won't break anything, by mimicking the migration 1:1. Obviously this would be unfeasible in a larger application with a db of many TBs, here you would instead create a database with an identical schema and seed it with some small, representative sample of the real database. - This is important as we have one *persistant* database across Chirp's entire lifetime, instead of just pushing a *chirp.db* file with every deployment(thus resetting the database on every deployment, which seemed undesirable) - Unit/Integration tests: - By far the simplest of the test workflows. Here we just figure out what test projects exit(omitting UI-tests since they require a lot of additional set-up) and then run those test projects in parallel. - UI Tests: - The bulk of this workflow is in setting up for the UI tests. The way they are implemented, the test runner expects an up to date Chirp.Web binary in it's own bin/ folder. Whatever binary is there is what will be tested, so for accurate tests we have to make sure we export the newest version of Chirp.Web. - In addition to that, Playwright just has a lot of large dependencies(powershell and several browsers(who we cache since they take up ~500MB)) that aren't installed by default on the github actions machines. - Deployment Setup - Here we build the Chirp.Web project binaries that we want to deploy as well as the bundled migration we (might) want to apply to the production database. Note that we always push a bundle, even if there is no new migrations to apply. In that case nothing happens when you try to apply the migration on the server. Naturally we can't deploy if we fail to generate either of these artifacts, but this step should usually succeed.

If a single step fails, the entire workflow fails and nothing will be deployed.

1.3.1 Issues/Points of improvement

As one can see, there is a few instances of redundancy in the workflow. The worst offender is probably the fact that we generate the exact same migration bundle and Chirp.Web binaries twice; once for testing and once for the actual deployment. Redudancy in, say, setup dotnet is immaterial considering how little time that action usually takes to execute.

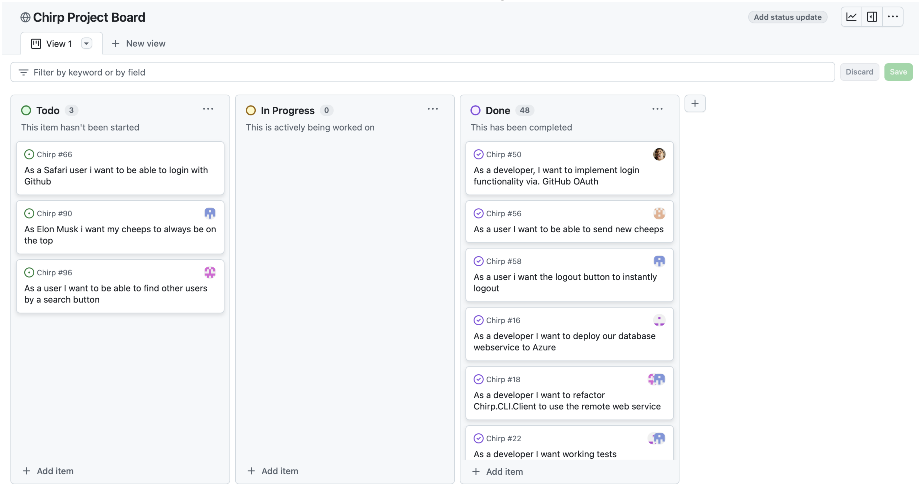
The reason for this redundancy is that the three test-workflows of the deployment workflows are, in Github actions, an entirely different workflow which is called on every push to every branch. The deployment workflow just calls this workflow and has the actual deployment action depend on it's success.

This then also means that every push on main has the same tests run on it twice; the test workflow is triggered once directly by the push and once by the deployment workflow.

The double generation of the migration bundle and Chirp.Web binaries could be solved by having the binaries as an output of the Test workflow and input of the Deployment workflow, and the double running of the tests could be solved by either having the test workflow explicitly only trigger on *non-main* branches or by having the deployment workflow query if a successful test run on the same commit exists.

1.4 Team work

Below is an image of our project board on GitHub right before hand-in. As seen in the picture there are unfinished issues. The unfinished issues are from the wild style week that aren't implemented due to focusing on higher priority issues based on the project requirements or time constraints. On the project board, it is visible that each issue are assigned to one or more team members.



Chirp Project Board

Add status update

View 1 | + New view

Filter by keyword or by field

Discard | Save

Todo 3
This item hasn't been started

- Chirp #66
  As a Safari user i want to be able to login with Github
- Chirp #90
  As Elon Musk i want my cheeps to always be on the top
- Chirp #96
  As a user I want to be able to find other users by a search button

+ Add item

In Progress 0
This is actively being worked on

+ Add item

Done 48
This has been completed

- Chirp #50
  As a developer, I want to implement login functionality via. GitHub OAuth
- Chirp #56
  As a user I want to be able to send new cheeps
- Chirp #58
  As a user i want the logout button to instantly logout
- Chirp #16
  As a developer I want to deploy our database webservice to Azure
- Chirp #18
  As a developer I want to refactor Chirp.CLI.Client to use the remote web service
- Chirp #22
  As a developer I want working tests

+ Add item

When issues were created a person/persons were assigned for the responsibility of the new feature. The responsible person creates a branch and starts working on the issue. Once an issue is done the assigned person submits a pull request and the PR is tested and reviewed by another team member that hasn't worked on the specific issue. This part of the timeline is iterative, meaning that if the PR is not approved or fails tests, the assigned person will continuously improve it until it is approved, and the branch is successfully merged into the main branch. After this step will the issue be moved to 'Done' on our project board.

1.5 How to make *Chirp!* work locally

How to Git Clone and Run the Program:

1. Open a new terminal window at the preferred directory and run the following command: `git clone https://github.com/ITU-BDSA2024-GROUP26/Chirp.git`
2. Navigate to the Chirp directory, once the cloning has finished: `cd Chirp`
3. Navigate to the src folder, where the source code lies: `cd src`
4. Navigate to the Web folder, containing the razor pages and program: `cd Web`
5. Once in the web directory, run the program:
    1. To run the program on Windows, write following command : `& { $env:ASPNETCORE_ENVIRONMENT = "Development"; dotnet run }`

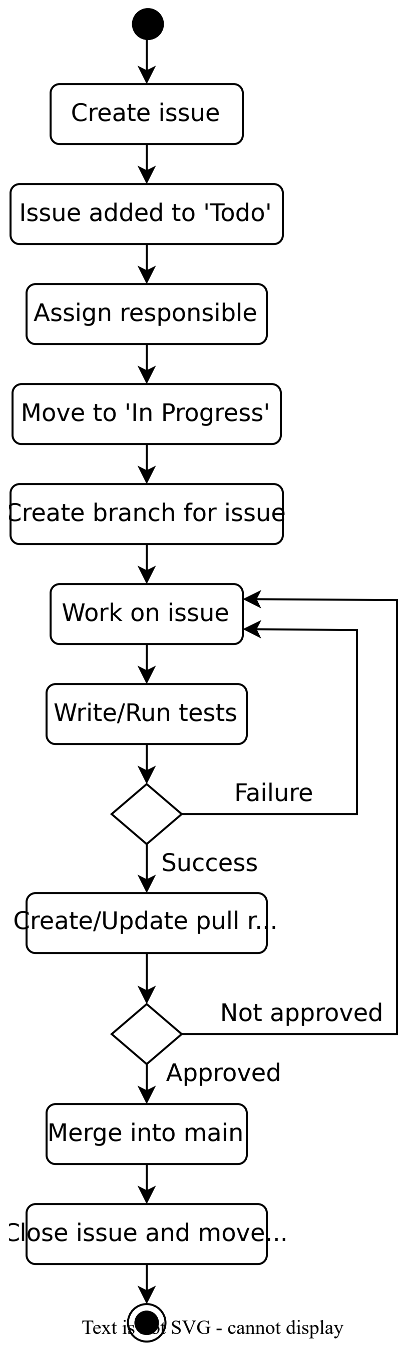


Figure 2: Illustration of issue activities

2. To run the program on MacOS and Linux, write following command:
   `ASPNETCORE_ENVIRONMENT=Development dotnet run`

Once the build has finished, this line should be visible containing a link to the localhost:

```
Now listening on: http://localhost:5273
```

Clicking on the link will direct to the locally run Chirp! application.

1.6 How to run test suite locally

List all necessary steps that Adrian or Helge have to perform to execute your test suites. Here, you can assume that we already cloned your repository in the step above.

Briefly describe what kinds of tests you have in your test suites and what they are testing.

To run test on the unit tests, you have to have powershell as well as dotnet and Playwright installed.

How to install powershell:

1. Linux: `sudo apt update && sudo apt install \-y powershell`
2. MacOS: `brew install \--cask powershell`
3. Windows: You can skip this step if you are using powershell.

How to install Playwright:

1. Go to the root directory of the project
2. Run the following command: `./tests/Web.UITest/bin/Debug/net8.0/playwright.ps1 install`

How to run test suite locally:

1. Navigate to the Chirp directory.
2. Run `scripts/setup_UI_tests.sh`
3. Ensure it has execute permissions (on Linux/MacOS)
   1. Run the command: `chmod +x scripts/setup_UI_tests.sh`
   2. Run the command: `./scripts/setup_UI_tests.sh`
4. Run the tests by using the command: `dotnet test`

The different test suites:

Our webapplication Chirp includes three test suites.

1. Repository.Tests
2. Service.Test

3. Web.UITest

The Repository.Test folder contains unit and integration tests for the functionality of the repository classes within the Chirp application.

The Service.Test folder contains unit tests for validating the functionality of the Cheepservice class within the Chirp application.

The Web.UITest folder contains UI tests that were made using Playwright. The UI tests tests the user interface, and the user interactions of our webapplication, Chirp!.

2 Ethics

2.1 License

State which software license you chose for your application.

We have chosen the standard MIT License for its simplicity and widespread use. The license is commonly used with the .NET, which is the main platform we are working with.

2.2 LLMs, ChatGPT, CoPilot, and others

During the development of our project, we used the following LLMs: ChatGPT GitHub and Copilot. ChatGPT was primarily used when we needed clarification, a better overview, boilerplate code or help understanding specific errors and bugs that we couldn't resolve within the group. It assisted us through the project development with issues and questions, where the textbook material wasn't enough to guide us. On the other hand, Copilot was more code-specific, directly assisting us in writing and completing code.

Both of the LLMs improved our productivity by saving time on repetitive tasks, such as generating boilerplate code or rewriting previously written code by the group. The biggest disadvantage we noticed when using ChatGPT, was that we had to be careful with our prompts and know exaclty what we wanted to ask, to receive relevant and helpful responses. At times, especially for obscure code-related issues, ChatGPT's answers weren't useful, requiring us to either rephrase our questions, or simply not rely on ChatGPT for that issue.