Chirp! Project Report

ITU BDSA 2025 Group 13

Dimitri Alessandro Nielsen dini@itu.dk
Karl Karl Theodor Ruby krub@itu.dk
Lukas Shaghashvili-Johannessen lush@itu.dk
Matthias Nyman Nielsen mnni@itu.dk
Matthias Schoenning Nielsen mscn@itu.dk

1 Design and Architecture of *Chirp!*

1.1 Domain model

1.2 Architecture — In the small

Below is a onion architecture diagram to illustrate the overall architecture of the *Chirp* application. The diagram also illustrates dependencies, where the outer circles depend on the inner circles. *Dependencies are illustrated as red arrows.*

1.2.1 Domain entities

In yellow is the center of the architecture as *Chirp.Core*. This layer stores the most fundemental parts of the codebase. In this project *Chirp.Core* stores the *Cheep* and *ChirpUser* domain model.

1.2.2 Repository layer

In red is the infrastructure layer of the codebase. This layer is responsible for retrieving domain relevant information from the database.

1.2.3 Service layer

In orange is the service layer. This layer is responsible for translating the domain models into *DTO*'s (Data Transfer Object) and connect requests to the ui. This layer therefore acts as a binder between the infrastructure and the ui layer. When a user request is received the service layer handles that requests, retrieves information from the infrastructure layer, and translates the information received into DTO's. These DTO's are then used by the UI to display information and data to the user.

1.2.4 UI layer

In blue is the UI layer. Here the UI is displayed to the user via `.cshmtl` pages. Here *page models* sent user requests to the service layer and decide the state which to display for the user. The state can change over the lifetime of the application, for example, when the user is logged in. Logging in changes the formatting of the pages, which the *page models* are responsible for handling. ## Architecture of deployed application

1.3 User activities

1.4 Sequence of functionality/calls trough *Chirp!*

2 Process

2.1 Build, test, release, and deployment

2.1.1 Versioning

Before the lecturers introduced us to semantic versioning and told us it was a requirement, we used CalVer[1]. CalVer was initially chosen, as it uses the calendar date for versioning, and seemed to be a good way to coordinate our weekly releases. [2]

Once we switched to semantic versioning we decided that it would make sense to automate this process. The tool we used to automate this was Release Please from Google. Release Please continuously monitors the git history of a project through a GitHub action. The action identifies commits which use the conventional commit standard and generates a release log based on changes in its own branch. The action also opens a pull request, which when merged, merges the changelog into main and creates a new release with the changelog added as a description. This helped give us an, and potential users, an overview over what has changed between releases.

In addition to this, Release Please also automatically computes the next version number based on the `feat`, `fix`, and `feat!` tags from conventional commit. This was nice, as we didn't have to consider what our next release number should be.

One issue we faced with this was we ended up with a rather high major version (5.x.y). The reason for this was our failure to consider what was actually a breaking change. We followed the convention of tagging any breaking API change as a breaking change, which would make release please update the major version[3]. However, these breaking API changes were often only breaking for internal APIs, for many major releases, no user-facing APIs changed. We

[1] CalVer
[2] The initial release tag was deleted and tagged again using semver
[3] Lecture slides on Semantic Versioning

should not have considered these internal API changes as breaking, since, for the end user, these changes were not breaking. What we should have considered a breaking change should be the switch from a **CLI** to a **web page**, and potentially **the addition of identity**. This would mean that *Chirp!* would be on **v3.x.y** or **v2.x.y** depending on whether the addition of identity was considered breaking, not **v5.x.y**.

2.1.2 Deployment

Whenever we deploy our code to GitHub, a number of GitHub Actions scripts will be run. These can be found the .github/workflow directory. - coverage.yml: Runs a code coverage test and fails upon not reaching the set threshold - format.yml: Runs dotnet format, which maintains a certain code standard in our code. These formatting commits have been attributed to our group member natthias - main_bdsagroup13chirprazor.yml: Builds and deploys our code to our Azure Webapp instance, using our GitHub Secrets to access login information. This file was auto-generated by Azure and afterwards customized for our needs by a group member. - release.yml: Builds and publishes our project to our GitHub repository, with builds for different operating systems.

2.1.3 Linear git history

Initially, we did not enforce a linear git history as a requirement for our project, but it was a soft requirement to attempt to keep the history linear. However, after #87381, we decided to enforce a linear history on the project through the git repo settings.

There are many reasons for enforcing a linear history. For one, having a linear history makes managing and maintaining a project easier. It is much easier to understand when a certain feature was added, as each commit is a patch, which cleanly applies atop the previous commit.

It also allows for external tools, such as `git bisect` to traverse the commit graph easier. Since a linear history is just a line, it is much easier for `git bisect` to perform a binary search and find when a regression was introduced. On the other hand, when the commit graph is a large interleaving of commits and merge commits, it can be difficult to perform a binary search. This does not mean it is impossible to perform a binary search on a non-linear history, but the cognitive load on the developer is increased.

It also allows for easier code review. Since each commit cleanly applies atop the previous, there is no extra noise generated by merge commits. It is simply a set of patches to apply to the main branch.

Finally, one of the primary reason we used a linear git history was for external tooling. We used an action called release-please for automated changelog generation. This tool recommended a linear history to ease parsing of the commit

graph[4].

The workflow for a linear git history is rather simple. When a commit is ready to be merged with the main branch, you `git rebase origin/main`, changes the base of the branch to be the current *HEAD* of the main branch. This might introduce merge conflicts, which you resolve as normal. Once the conflicts have been resolved, and the PR has been approved, it can be merged with `git merge --ff-only`.

Having a linear git history is one way to manage a project which comes with it's set of benefits. However, having a non-linear history has its own set of benefits. Chiefly, some metadata is lost when a branch is rebased, since the commit the branch was initially based on has changed. In addition to this, a non-linear history can make the history of long-lived feature branches more clear, however, since we used trunk-based development this was not a concern for us.

2.2 Team work

2.2.1 Trunk-based development

2.3 How to make *Chirp!* work locally

2.4 How to run test suite locally

3 Ethics

3.1 License

3.2 LLMs, ChatGPT, CoPilot, and others

[4]Release Please documentation about linear history