IT UNIVERSITY OF CPH

DevOps, Software Evolution and Software Maintenance, BSc - Spring 2022

Alexander Søndergaard - aleso@itu.dk
Joachim Alexander Kofoed - jkof@itu.dk
Mai Blom - lukb@itu.dk
Markus Æbelø Faurbjerg - mfau@itu.dk
Sam Al-Sapti - sals@itu.dk

BSDSESM1KU
IT-University of Copenhagen

June 1, 2022

Contents

1 System's Perspective

1.1 Design

To develop our version of MiniTwit, we have chosen to use the programming language Go. We are using a toolkit called Gorilla that includes a number of different packages, that each help with different things regarding web development. We use the library called GORM to interact with our PostgreSQL database instead of writing our own SQL queries. We use Prometheus to monitor our application - and Grafana to give us a dashboard with the Prometheus data. To log things like error messages, we use the ELK stack.

1.2 Architecture

1.2.1 Components and connectors

Our system is containerized using docker and deployed with DigitalOcean, which enables the user to access and use the application. Our API and app are connected to the database container. We use Caddy as our reverse proxy. Caddy receives traffic and redirects it to either the API, app, Grafana, or Kibana, using the same ports for all four services.

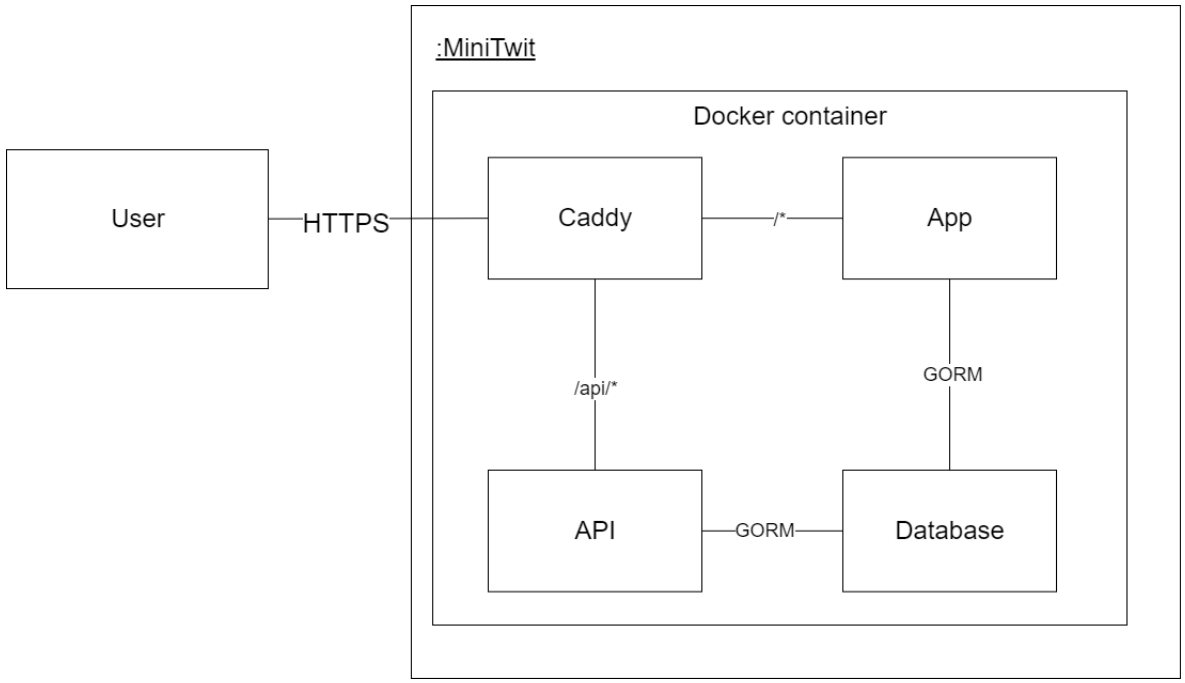

Figure 1: Diagram showing an overview of our components and connectors.

1.2.2 Module Viewpoints

Packages

The system consists of three packages: *src*, *CircleCI* and *Docker*. The packages and their dependencies are visualized in figure 2.

The *src* package contains the system logic relevant to user actions in the app and API. The app handles user inputs from the MiniTwit website, whereas the API handles HTTP requests through exposed endpoints. *src* also contains the monitoring and the controller packages.

The *monitoring* package contains two things. First and foremost, the monitoring infrastructure is set up with the Prometheus package. The Prometheus Promauto functionality is used to define what metrics to track. We chose it because it sets up an easily extendable code structure. Secondly, the function MiddlewareMetrics starts a timer and measures CPU usage before serving the http request it is handling. After having served the HTTP request it increments the request count and records the duration of the HTTP request.

The *controller* simply defines the structs for the database, to connect to the database, query a users ID from a username and hash passwords.
For a full overview of the entire *src* package, see appendix A.1

The *Docker* package is not tied to MiniTwit functionality. Its purpose is to contain the instructions for initialization of our different Docker containers when we deploy the system. This means that it contains a variety of file types. Naturally, it contains the .Dockerfile for the app and API. Furthermore, it contains .yml files with launch settings for our monitoring tools. The */grafana* directory also includes .json files that define the UI for its dashboard.
These files are run at every system initialization, building the containers, based on the data carried in the package.

Finally, the *CircleCI* package contains the functionality for CI/CD. It is linked to our GitHub repository, meaning that every time we merge a pull request into the MiniTwit repository's main branch, the setup in the *Docker* package is run.

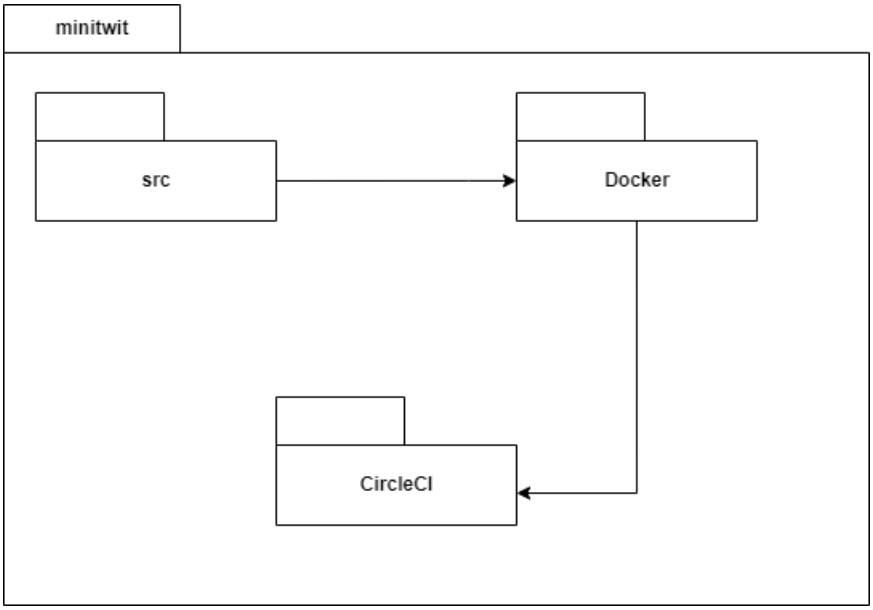

Figure 2: This diagram shows the outermost layer of the Minitwit packages and their dependencies.

Src

The overview of our *src* package is seen in figure 3 which visualizes the dependencies of our *API* and *App*.

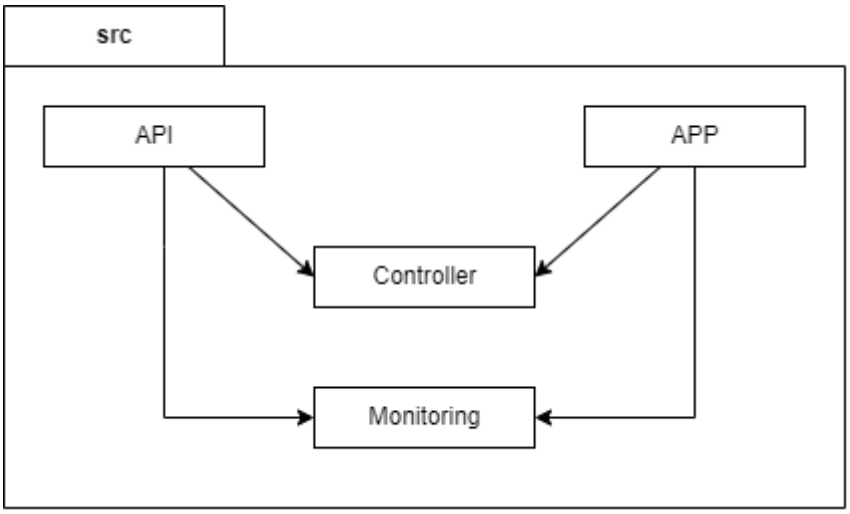

Figure 3: The diagram shows the dependencies between the API and app in the *src* package.

Class overview of App

Our App consists of the following:

- Main.go → our back-end functionality of the application
- Front-end, consisting of 4 HTML files
  - Layout.html
  - login.html
  - register.html
  - timeline.html
- a single CSS file responsible for the UI-design of our front-end

Our app exposes HTTP-endpoints for the following use cases:

- `/` → If user logged in, loading personal timeline, if not logged in, redirecting to `/public`
- `/favicon.ico` → prevents the app from thinking favicon.ico is a user
- `/public` → loads the public timeline
- `/add_message` → posting message on the web service
- `/login` → login opportunity for users
- `/register` → registration opportunity for users
- `/logout` → opportunity to log out of the web service
- `/{username}` → load the timeline of the given user
- `/{username}/follow` → logged in user wanting to follow user called {`username`}
- `/{username}/unfollow` → logged in user wanting to unfollow user called {`username`}

In order to display data in the front-end, we make use of the struct `TimelineData`, which acts as a container for the data fetched from the database. The struct `SessionData` is used to keep track of the data associated with the current session in the application. In other words, this is the struct whose field, `Flashes`, is injected into the frontend, such that nothing is directly exposed to the front-end. This hides the back-end functionality, improving security of the site as well as keeping the code less coupled, such that changes in one part do not break others. To load the data on the frontend, we use the Go package *Go/Template* which allows for Go code to be loaded directly in the HTML without the need for any JavaScript.

Class overview of API

The API makes use of HTTP requests by exposing HTTP-endpoints in order for the internal logic to handle the HTTP requests. Every endpoint has been kept according to the initial Python application. Our API has exposed endpoints for the following HTTP requests:

- Registering users (POST) /api/register - if success, returns HTTP response code 204. If failing, returns HTTP response 400.
- Following a user (POST / GET) /api/follws/{username} - the method is checking if already followed (GET) and following / unfollowing (POST) - if success, returns HTTP response code 204. If failing, returns HTTP response 404.
- Latest message (GET) /api/latest - returns the ID of the latest message from the application.
- Messages per user (GET) /api/msgs/{username} - if success, returns HTTP response code 204. If failing, returns HTTP response 405.
- Messages (feed) (GET) /api/msgs - if success, returns HTTP response code 204. If failing, returns HTTP response 405.

The API is meant to be run within a simulator environment, which acts as if the API is being accessed by real users. In our case, the API has failed quite a lot (around 90-95%), which is due to downtime of our database (4 days). We are suspecting that a lot of the users were registered during these days and therefore we had a lot of errors. The errors arising from *user does not exist* has also caused the problem that it camouflaged any downstream errors. Due to the fact that it does not follow good DevOps principles to just register users (by using the "username" property in some endpoints and therefore allowing null-values in the database, which is not a maintainable solution) when they did not exist, we were unable to find a way of resolving this in a proper manner.

1.2.3 Deployment

The deployment diagram below (see figure 4) visualizes the dependencies between software containers, stored in Docker containers, and the physical hardware that runs it. The system is run on a virtual machine hosted by DigitalOcean. The virtual machine runs Ubuntu 20.04 LTS and the OS runs the system in a Docker daemon that contains individual Docker containers and Docker networks.
The two Docker networks are separated into logical units. The first unit is the main-network, which contains the system logic, monitoring tools, and the database. The second network container, elk-network, contains the tools used to create logs. As the ELK setup does not need to communicate with any of the applications, it does not need to be in the same network. It works by Filebeat reading directly from the system directory that contains the logs from Docker containers, which is mounted as a Docker volume.

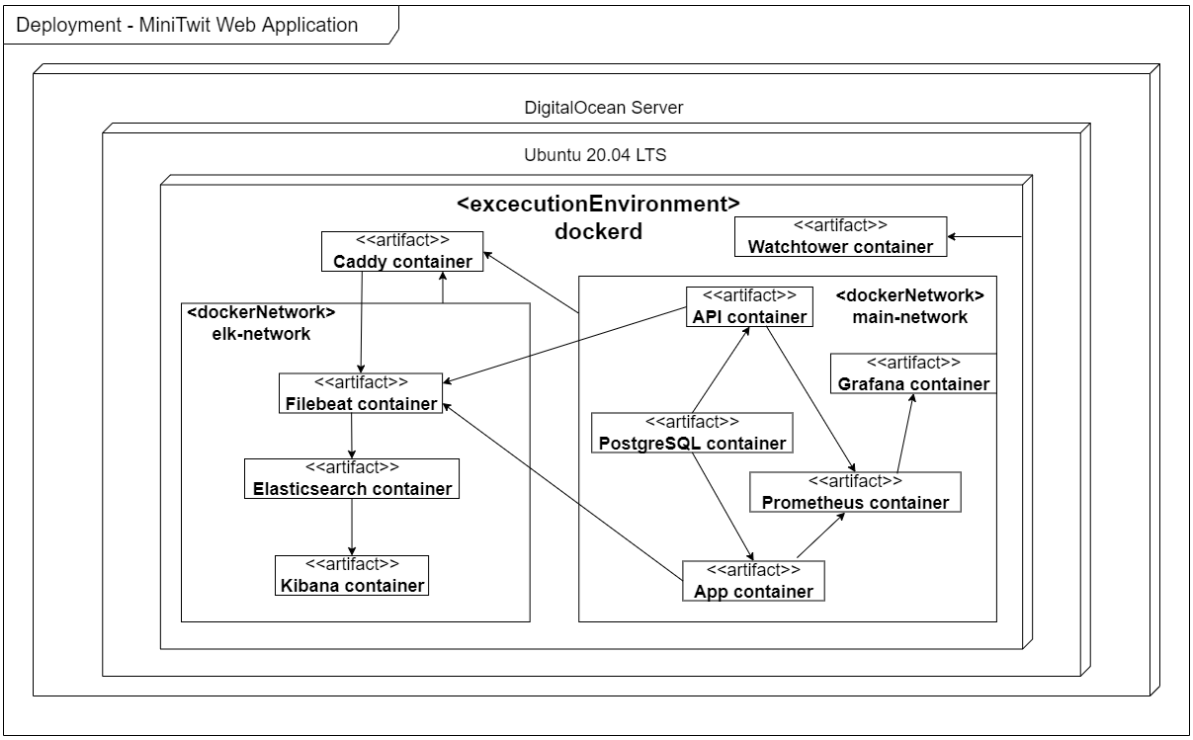

Figure 4: The deployment diagram highlights how the software elements are mapped to the hardware that it is hosted on.

1.3 Dependencies

- Go 1.18.x
- DigitalOcean: Our hosting service.
- Docker: Service for containerizing software.
- Docker Compose: Tool that gives instructions to Docker on what images to initialize when deploying the system.
- CircleCI: A platform for continuous integration and deployment.
- PostgreSQL: The relational database system
- Gorilla/mux: A http request router and dispatcher which we use for matching incoming request to their respective handlers.
- GORM: Go library used for creation of relational database schema and migration, as well as providing CRUD operations.
- Prometheus: Our monitoring software.
- Grafana: An analytics and visualization tool which is hooked up to Prometheus.
- ELK: A Docker Image providing the following services:
  - Elasticsearch: used for its index log data.
  - Filebeat: Collects logging data and forwards it to Elasticsearch.
  - Kibana: Visualization and navigation tool for the data stored in Elasticsearch.

- Caddy: Proxy and web server service. We use it for reverse proxying.
- Watchtower: A service for checking updates for Docker images and updating Docker containers.

1.4 Important interactions of subsystems

We use Docker Compose to run multiple Docker containers for our system, such as MiniTwit, ELK, Prometheus, etc. For automating the process of updating images, we use Watchtower.

For logging we have set up our Docker Compose file to create containers for the three ELK images; Elasticsearch, Filebeat, and Kibana. Filebeat is the container that observes and ships data to Elasticsearch. Kibana then receives the data from Elasticsearch and visualizes it.

When new features are pushed to the GitHub repository's main branch, CircleCI triggers and re-deploys the system.

We monitor our application with Prometheus, which collects metrics by scraping from a selection of HTTP endpoints. These metrics are then stored on the Prometheus server, and can be exported or visualized with Grafana.

1.5 Current state of our system

SecureGo notified that the .md5 hashing algorithm is weak. Our system still uses it, as it is a requirement for Gravatar URLs.

At the end of development, we have judged our system according to our chosen metrics, described in section 2.2.

- Maintainability - The code base has been left in a maintainable state. It is easy to navigate and understand. The packages divide the system in logical units.
- Testability - We did not manage to implement unit testing in time. However, we do static analysis on all branches and also build and deploy for each pull request to main.
- Portability - The portability of the system is difficult to estimate without unit or integration testing. Again, the static analysis and clean conventions combat this somewhat, though we are not completely satisfied yet.
- Reusability - We have managed to make our code reusable and avoiding redundant code. An example is our controller, which has methods used by both our API and app.
- Technical debt - During the course of developing and maintaining Minitwit, we tried to minimize our accrued technical debt. The most mention-worthy debt, is that our Docker swarm is stopping our monitoring and logging from working. As such, the production branch, at the date of delivery, is not using swarm, to prioritize monitoring.

1.6 License compatibility

Our MiniTwit application is licensed under the AGPLv3 license. After running the tool *lichen* on our code, we found no license violations. AGPLv3 is the most copy-left license that we know of, meaning that almost all other FOSS licenses are compatible with it.

2 Process' Perspective

2.1 Team

To organize our team, we make a weekly plan each Tuesday. The plan depends on the current hangups of the project and the new tasks of the week.
We split up the team in subgroups, depending on the complexity of the task we are taking on. Each subgroup starts their work immediately and coordinates freely until next week. During the week, we use our Discord server to keep each other updated on progress with the tasks.

2.2 CI/CD

We have implemented CI/CD with CircleCI. Our CircleCI's configuration includes a list of jobs. Those jobs are: static analysis, build, deploy, and workflows.

In our continuous integration chain, we use different tools and metrics to ensure the quality of our extensions to the system.

- For static analysis we use a tool called `golangci-lint`, which runs the following linters in parallel:
  - `gosec` is a tool to enhance security by inspecting the source code for vulnerabilities, by scanning the Go AST.
  - `gofmt` is a tool which formats the Go source code, such that for example, unnecessary parentheses are removed.
  - `staticcheck` is a tool which, by using static analysis, is able to locate bugs and find performance issues, while offering simplification and enforcement of style rules.
  - `gosimple` is a tool, with the purpose of simplifying source code.
  - `unused` is a tool to ensure that unused variables, functions constants and types are removed.
  - `typecheck` is a tool, which ensures that our variables are of the correct types.
- For Quality assessment systems we have settled on a list of metrics, that serves as the standard we want to set for our system.
  - `Reliability` - Static analysis tools (described above) are used to ensure reliability within our system.
  - `Maintainability` - Our code should be easy to read. For example, instead of doing fancy one-liners, code should be written out in longer form for better readability. All variable names should be easy to understand. This makes the code less messy and easier to maintain.
  - `Testability` - We want to have unit tests for our code, such that, whenever we push any new code to our repository, we know that it does not break anything.
  - `Portability` - Our code should be as portable as possible. This means, that we should develop with the API in mind, such that we avoid platform specific solutions.

- `Reusability` - Code should be reusable, to avoid code duplicated or redundant code.
- `Technical debt` should be minimized as much as possible. It is unrealistic to avoid completely, but steps should always be made to combat it.

The final step of our CI/CD chain is that we have a Discord bot set up that publishes a message to our Discord server, whenever there is a new commit to the repository. This way, all developers are encouraged to inspect commits and stay up to date with the code base.

2.3 Repositories

We chose to use a mono-repository structure hosted on a single GitHub repository. The repository includes all of our code for both the API as well as the MiniTwit web-application. We chose to keep the API and application in the same repository since they are both quite small systems, and because they share several dependencies and functions.

2.4 Branching strategy

Our branching strategy is trunk-based development. We have a main branch which, contains a working state of our application. Whenever we want to make a new feature or fix a bug, we create a new branch from our main branch, where we work on the feature. When the work is done, we make a pull request from the work branch into our main branch which then needs to be approved by another team member before it gets merged into main.

2.5 Development process

We add issues to our Github, following the courses weekly additions to the project. All individual issues are added to a To Do List, that tracks their completion. This is to create a quick overview of our progress with the project and maintain a backlog. For weekly tasks, we post a message on our discord, that tracks who is responsible for what.

2.6 Monitoring

To monitor our application, we use the tool Prometheus that monitors the metrics we want to track. We use the package PromAuto to setup the metrics we want to track with Prometheus. The constructors in PromAuto, in contrast to the Prometheus package, returns *Collectors* that are already registered to a registry. It creates functions on 2 levels: top level functions, that return *Collectors* registered to the global registry, and Factory type methods that returns *Collectors* that are registered with the constructor, that the factory were created with.
We have chosen to track the following three metrics for our app and API

- The CPU load
- The total number of processed HTTP requests
- The average duration of requests

To visualize our data, we use the tool Grafana, which uses the Prometheus data and visualizes it through a web endpoint. The Grafana dashboard can be seen at A.2.

2.7 Logging

We use the ELK stack for logging. The information we log includes:

- Caddy access/error logs
- App and API error messages
- Log messages from the ELK components themselves

The log data is read directly from the directory that contains the logs from the Docker containers, which is mounted as a Docker volume on the Filebeat container. Filebeat then parses the data and sends it to Elasticsearch over an internal network connection. The web-frontend we use to look at our logs is Kibana.

2.8 Security

To estimate the security standing of our system, we sat down as a team and completed a security assessment. We started by doing a risk identification. We concluded that our assets include the user data stored on our servers as well as a web server with almost constant uptime. The related threat sources would thus be unauthorized access to sensitive data, and infrastructure provider downtime that would make our service unavailable. We visualized these risk factors in a risk matrix (see figure 5) based on their impact and how likely they are to happen.

	Rare	Unlikely	Possible	Likely	Certain
Catastrophic		SSH key leak			
Critical	DigitalOcean outage Physical server attack				
Marginal					
Negligible		Login credentials			
Insignificant	Github outage CircleCI outage				

Figure 5: The Risk matrix visualizes threats based on their severity and likelihood.

The most crucial risk of provider downtime is DigitalOcean, since we host our application with them, which is why we assess downtime on their end as much more crucial

than downtime on Github or CircleCI.
In relation to keeping sensitive user data safe, we identified multiple possible risk scenarios. The first risk is a physical server attack on DigitalOcean. Since we cannot improve security on their end, the best course of action is to have another provider setup ready. Our personal risk factors are leaked login credentials or an SSH key leak. To improve the security, we have implemented two-factor authentication.

After pentesting our system with Zaproxy, we found out that we had a number of vulnerabilities, although none of them were too high of a risk. One of the medium-risk issues that Zaproxy found was regarding some missing HTTP headers. We chose to fix this since it was a larger risk than the others. Figure 6 shows the results from running Zaproxy on the newest version of our application. We unfortunately did not have time to fix the remaining issues, however, the risks proposed by Zaproxy would be an obvious next thing to do in regards to improving the security.

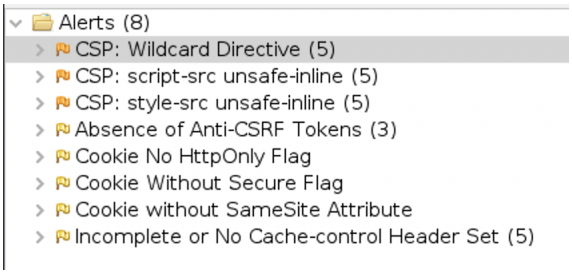

Figure 6: Results from Zaproxy based on the newest version of our application.

2.9 Scaling & Load balancing

We have attempted to set up horizontal scaling via Docker Swarm, however we did not succeed in integrating it completely. Our Swarm setup, which is currently on the `feature/swarm` git branch, can only run the MiniTwit application itself, so no monitoring nor logging. We believe this is due to issues with persistent storage, which is complicated to implement in a cluster setup.

3 Lessons learned perspective

In previous projects, we have not worked with an already existing code base, nor have we had to update it and implement new tools. Having to rewrite the application in a new language came with the implications of having to recreate functions with exactly the same functionality. Thus having to find dependencies to replace functionality of the python dependencies used, such as flask.

The course has highlighted that there are a wide array of different tools and solutions across programming languages. It has also made it clear that the ease of use between tools vary widely and that there is not necessarily equal support for each programming language. Additionally, we experienced that trying to replace one tool caused a ripple effect and made us look for new solutions in places that were only tangentially related to what we were trying to do in the first place. For example, one of our commit messages states: "The system now correctly directs to a HTML page, but the HTML pages have python dependencies".

This is also the first time we have had to create an API to communicate with an outside source, the simulator. Thus having to handle unknown request and adjust accordingly to errors caught by logging.

What we did differently compared to previous projects was that we took the principles of DevOps and applied them to our project. That is, we set up continuous integration, continuous deployment, implemented monitoring and logging, and scaled our system with Docker Swarm. Being aware of the DevOps process of maintaining our system also made us think about keeping our dependencies up to date. For example, we had to update to Go version 1.18 in march. We also had not experienced the concept of technical debt before. The course's structure of weekly releases, simulating an actual production product, showed us how fixing small difficulties and hangups can scale to a large task over time.

We used a CI/CD pipeline to continuously keep working code in our production branch while being able to quickly get new features or bug fixes merged in. Whenever we pushed new code to the production branch, the code was automatically built, tested, and deployed to the webserver. This worked well as we did not have to waste a lot of time for manually running tests and deploying the system.

The biggest issue for our group was downtime at the beginning of the simulation. The downtime meant that a significant amount of users, likely, did not get registered in our database. Thus, they did not exist for their respective API requests later in the simulation. This resulted in a lot of HTTP error responses, which caused a significant fluctuation on our graph of HTTP responses. Unfortunately, it took us too long to identify why we kept on having HTTP errors, where we did not expect them.

Some groups fixed this by creating users, when calls were made from users that did not already exist. What we wanted to do was to do a database transfer from another group who had the complete data set, but we never got to do this. If we could do it over, we would have prioritized this solution earlier, over creating the users that we did not get to register in time, since the first solution caused incomplete data sets for those groups.

A Appendix

A.1 Class diagram

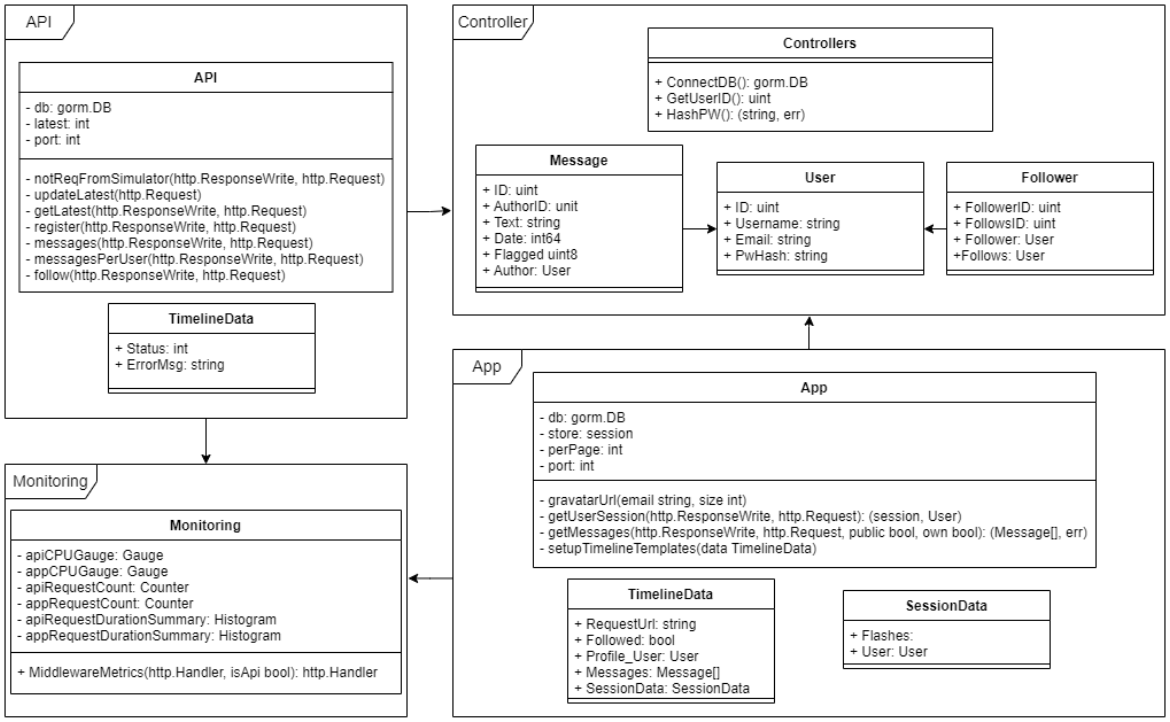

Figure A.1: Full class diagram of our application.

A.2 Grafana

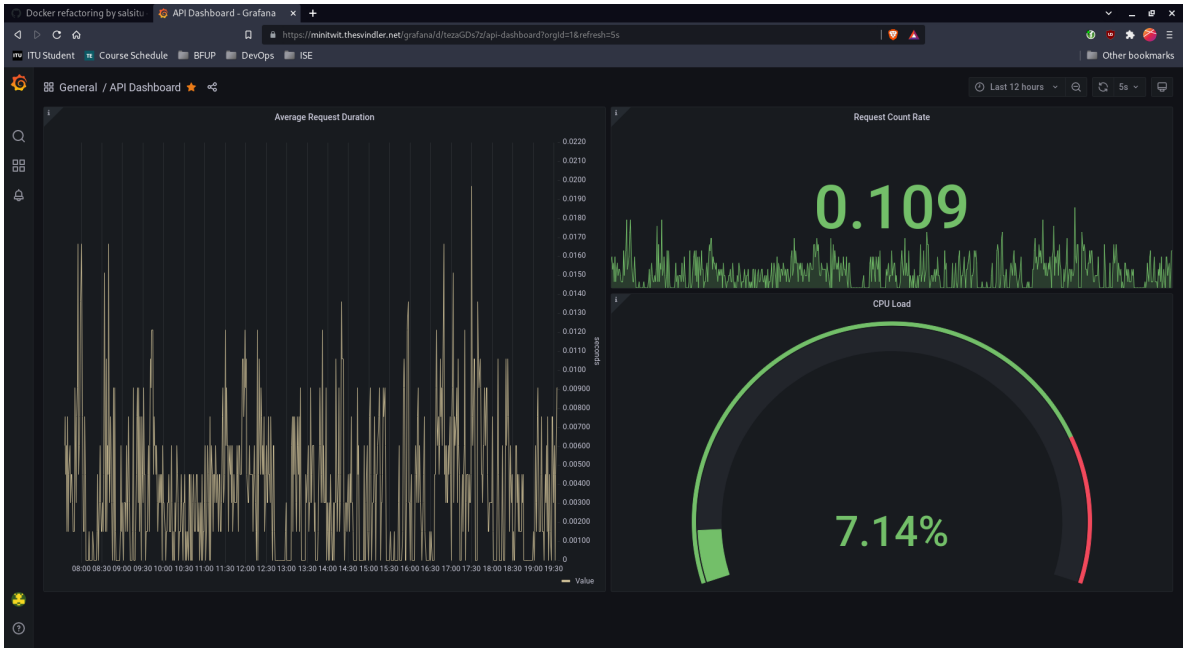

Figure A.2: Overview of our Grafana dashboard.