DevOps, Software Evolution and Software Maintenance

May 2021

Course code: KSDSESM1KU

Group members

name	email
Vlad Croitoru	vlcr@itu.dk
Anders Friis Kaas	anfk@itu.dk
Rasmus Dilling Møller	rdmo@itu.dk
Jonas William Gohn	jgoh@itu.dk
Sean Wachs	sewa@itu.dk

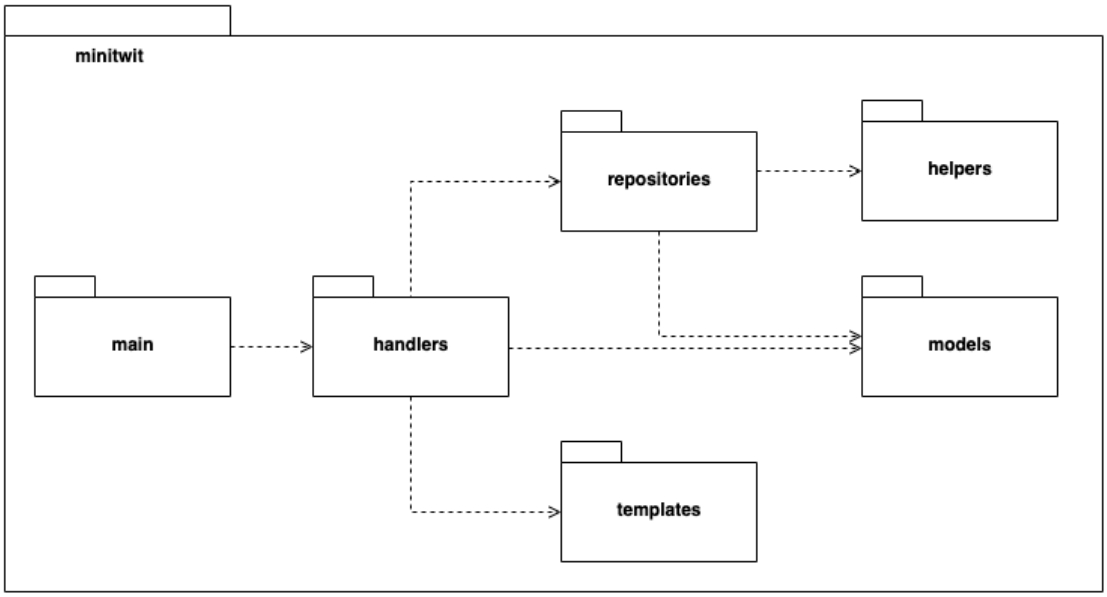
System

Overview

The core of the Minitwit application is written in Go. It is split into a frontend system that responds to HTTP requests coming in on destination port 8080 and a backend API that responds to HTTP requests received on port 8081. The frontend system responds to client GET requests with properly formatted HTML responses and is what users would interact with when they visit our website. The backend system responds only with raw JSON and is what the simulator interacts with. The two systems share a PostgreSQL database that stores user information, user following relationships, messages and the 'latest' value. The systems interface with the database through the ORM library GORM instead of through raw SQL queries. The systems are instrumented with Prometheus metrics which allows us to monitor them using Grafana.

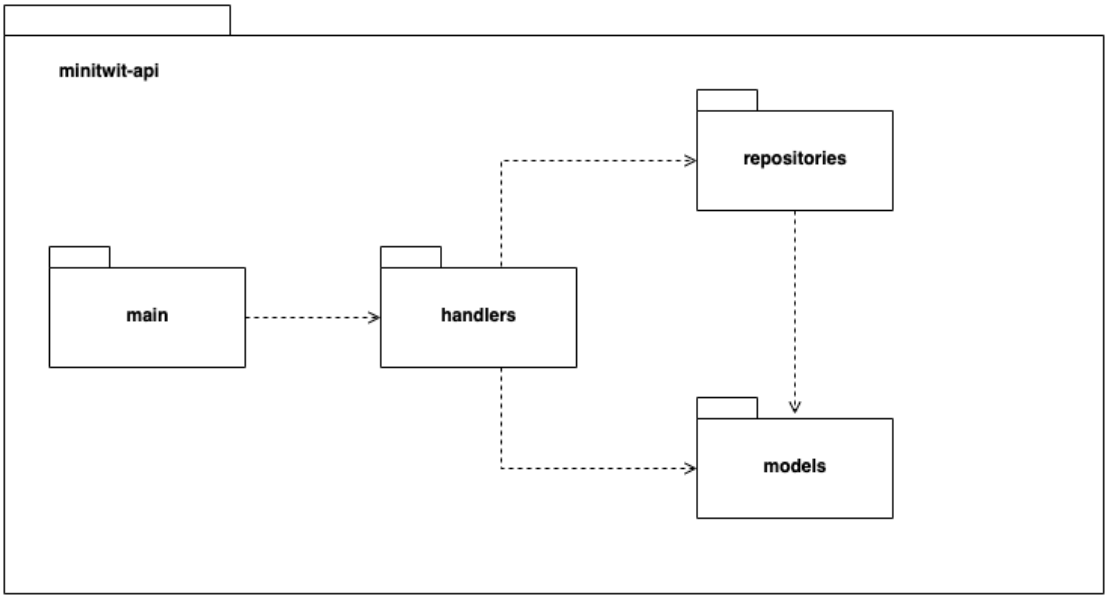
Architecture

During the course we had to rewrite both the user facing app - `minitwit` and the one used for the simulator - `minitwit-api`. The high level architecture can be provided by context diagrams.



`minitwit` has several components that can be divided logically and this is how we did it.

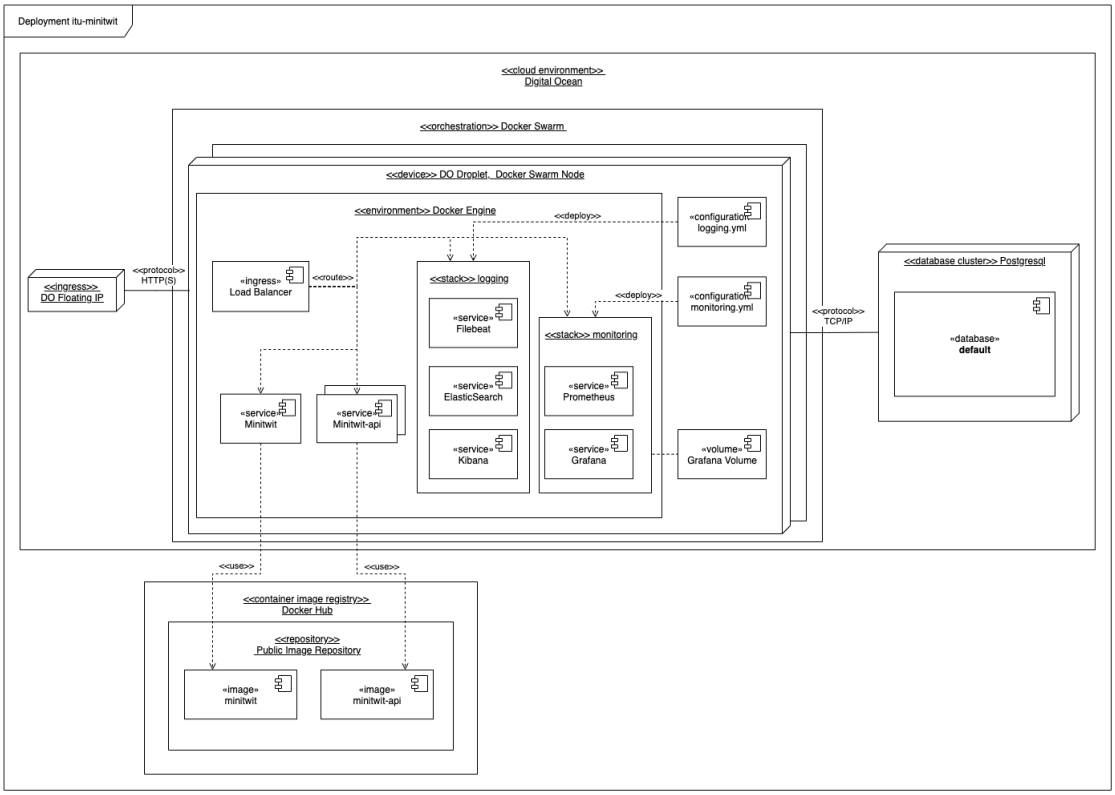
- `main` employs a chain of responsibility pattern to perform some middleware logic such as authentication and routing incoming requests to the appropriate `handlers`.
- `handlers` execute CRUD like operations by calling database/sql related procedures from `repositories` then generate an html doc based on the `templates` to respond with.
- `repositories` contain database related procedures that read or mutate data, `repositories` may aswell call certain `helper` functions to format or sanitize data and `models` that contain the mapping models of database schemas to Go structures.
- `helpers` has functions that format or sanitize data related to database procedures.
- `models` has mapping models of the database schemas to Go structures.



`minitwit-api` is more straightforward than `minitwit` since it does not bother with generating html docs to send back to the user agent.

- `main` similarly to `minitwit`, authenticates requests and updates the latest value via middleware and then routes request to the necesarry handler functions isolated in the `handlers` component.
- `handlers` execute CRUD like operations by calling database/sql related procedures from `repositories`, may use models to map database data to Go structures.
- `repositories` contains database related procedures that read or mutate data.
- `models` contains mapping models of the database schemas to Go structures.

The deployment diagram will give a more comprehensive static view and understanding of the dynamic run time components, nodes and processes in production.



- `<<cloud environment>>` is the environment in which all the processes are deployed, in our case it is Digital Ocean but it can be any other cloud provider.
  - `<<ingress>>` in this context is meant indicate the means of accessing the deployed processes, the Floating IP service from Digital Ocean provides a static ipv4 address that we can assing to different Droplets (Virtual Machines).
  - `<<database cluster>>` is a Digital Ocean service which provides a managed database cluster, the database engine picked is Postgresql.
- `<<orchestration>>` is the mechanism used to ensure that our services can be replicated and are highly available, for this we use Docker Swarm. Consider the following subpoints to be in the context of Docker/Docker Swarm.
  - `<<device>>` is the host which our services are running on. For this we use multiple Digital Ocean Droplets that are a part of a single Docker Swarm.
  - `<<environment>>` is the execution environment. All our services are running in a containerized form using the Docker Engine Container Runtime.
  - `<<ingress>>` in this context indicates the component responsible for routing requests to the required service. This mapping is done using the Routing Mesh.
  - `<<service>>` is meant to indicate both the `service` (the desired state of a process) and the `task` (a running instance of a process). And basically represents a running Docker container.
  - `<<stack>>` is a collection of services that are deployed using `.yaml` configuration files. The clustering is done by scope so we have a monitoring stack and a logging stack.
  - `<<configuration>>` is the `.yaml` configuration file used to deploy an application stack to Docker Swarm. Initially these were docker-compose files used in development. Can be viewed in repo at './deploy/docker-swarm'.
  - `<<volume>>` is a mechanism for persisting data used by the services.
- `<<container image registry>>` is a platform facilitating storage and distribution of Docker Images via image repositories.
  - `<<repository>>` is the particular repository used to store and distribute images used in delpoyment.

Dependencies

Production environment

The production system is deployed to three nodes on DigitalOcean. One node acts as a manager and load balancer and the other two are replicated worker nodes. The nodes are managed with Docker Swarm. The runtime environment of the worker nodes are Docker containers running Ubuntu v. 20.4. DigitalOcean was chosen as the cloud provider for the following reasons:

- They have a very affordable pricing model, especially for smaller projects such as Minitwit. Additionally, the GitHub student pack offered a 100$ credit.
- They have a comprehensive set of tutorials and guides and really nice documentation.
- Their web interface is very intuitive.

We chose Docker Swarm as the cluster orchestration technology because

- As Swarm is a Docker product, it uses the standard Docker API and networking. Since we were already virtualizing our production environment using Docker and Docker Compose, the transition to Swarm was

relatively painless.
- It has a lightweight installing and is simple to deploy as Swarm mode is already included in the Docker engine.

Technologies and subsystems

On a high level, the whole system depends on the following technologies:

- **Go**

  As previously mentioned, the two main applications were written in Go. The decision to go with Go was based on the following facts:

  - Go is a fast, minimalistic and scalable language with an extensive standard library.
  - It has a strong static type system which helps to minimize bugs. This is an advantage over Python, in which the original Minitwit application was written.
  - Concurrency is an integral part of the language and is supported through goroutines and channels.
  - Programs are constructed from packages which offers clear code separation.

- **PostgreSQL**

  PostgreSQL is the database for the project. We decided to use PostgreSQL because

  - It supports concurrency while still adhering to the ACID principles.
  - It is one of the most popular DBMSs in the world. This makes it easy to find help online.
  - It is free to use.

- **GORM**

  GORM is our object-relational mapping (ORM) library that we use to abstract our database into objects in Go. We chose GORM because

  - It is a full-featured ORM library for Go with many different different association types.
  - It integrates well with our PostgreSQL database.
  - It has built-in support for Prometheus.
  - It has a rich, extensive documentation which makes bug-hunting less painful.

- **GitHub Actions**

  CI/CD is handled with GitHub Actions. Actions was chosen for the following reasons:

  - As Actions is a GitHub product, it integrates very well with our GitHub repository.
  - It is relatively straight-forward to setup compared to other CI/CD solutions. Many common processes ('Actions') have already been implemented by GitHub themselves or other developers.
  - It is free to use up to 2000 minutes per month.

- **Prometheus/Grafana**

  The Go applications are both instrumented with Prometheus metrics, which are then visualized in Grafana. These two technologies were chosen based on the following reasons:

- The combination of Prometheus and Grafana is an industry standard for monitoring Go applications. Additionally, it is relatively straightforward to connect Prometheus to Grafana.
- Prometheus delivers metrics without creating time lag on performance.
- They both exist as Docker images in Docker Hub, which makes deployment easy.
- Grafana offers very customizable dashboards for visualizing application performance.

- **SonarQube**

  The codebase is statically analyzed by SonarQube. The reasons we went with SonarQube are

  - SonarQube is very feature-packed. It has support for identification of duplicated code, unit testing, code complexity, code smells and much more.
  - It integrates well with popular IDEs such as Eclipse and Visual Studio.

Go libraries

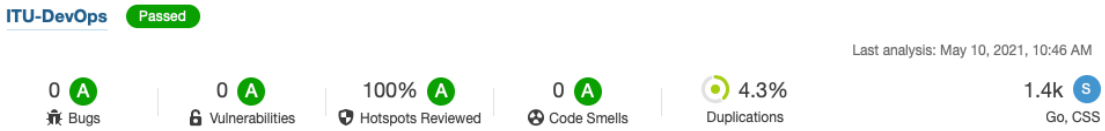
On the lowest level, our Go applications depend on the following libraries:

- **fmt** - used for string formatting and basic I/O.
- **errors** - useful functions for error handling and error manipulation .
- **os** - used to interface with the operating system. We use this to obtain environment variables.
- **encoding/json** - used to encode code objects as JSON strings.
- **strconv** - conversion to and from string representations
- **strings** - string manipulation
- **time** - used for function execution timing.
- **net/http** - used to listen for HTTP requests and serve responses.
- **log** - used for basic logging of our application. We log system errors and HTTP requests and responses.
- **gorilla/mux** - used to implement request routing. We use mux to direct specific requests to appropriate handler functions.
- **gorilla/sessions** - provides a cookie system.
- **godotenv** - used to read environment variables from a file.
- **gorm** - Gorm is the ORM library that we use to abstract the database to objects in code.
- **postgres** - used by Gorm to interact with the database.
- **prometheus** - prometheus is our metrics system that stores information about numbers of requests per endpoint as well as function execution times.
- **promhttp** - this submodule of prometheus is used to expose an endpoint called /metrics that is used by Grafana.
- **bcrypt** - used to hash passwords.
- **html/template** - used for generating valid HTML from templates

Current state of system

Code base

We use the static analysis service SonaryQube for automatic code review. After removing unnecessary files and refactoring the system, we received the the following ratings from SonarQube:



This indicates that our system is relatively bug-free.

License

After reviewing all our dependencies, we found the following six different licenses that our license had to comply with:

- PostgrSQL License
  - a permissive license that allows anyone to use, copy, modify and distribute PostgreSQL.
- MIT License
  - A very permissive license with high license compatibility.
- Apache License 2.0
  - a permissive, free software license.
- AGPLv3
  - a free software license. However, this has a strong copyleft clause, which requires us to use AGPLv3 for our project as well.
- GNU Lesser General Public License
  - General permissive license
- BSD-3-Clause License - very permissive license.

Because of the copyleft clause of AGPLv3, we used that as our license.

Process' perspective

A description and illustration of:

- How do you interact as developers?

We interacted with each other mainly through Teams voice and instant chat where we planned meetings and aligned tasks. After our meetings we would use the built-in Github features of the kanban board and issues section.

- How is the team organized?

Our team is organized as a flat hierarchy where we would debate on issues and taking joint decisions. There was no clear manager or leader during this process but due to the relatively small scope of the project this did not feel like an issue.

- A complete description of stages and tools included in the CI/CD chains.
  - That is, including deployment and release of your systems.

The CICD pipelines are implemented with Github Actions. Initially, there was just a single workflow which was building the docker images, pushing them to the Docker Hub repository and then running a docker compose file on the host machine while also forcing the new images to be pulled.

As the project evolved, we ended up with 5 workflows:

- `deploy-minitwit` - consists of two jobs `build-minitwit` and `deploy-minitwit`. `build-minitwit` builds the docker images for the minitwit service then pushes them to docker hub. `deploy-minitwit` uses a marketplace action appleboy/ssh-action to ssh into the Swarm Manager node, pull the latest image and trigger a service update. The workflow gets triggered on push to the development (default) branch or can be triggered manually on any branch.

- `deploy-minitwit-api` - consists of two jobs `build-minitwit-api` and `deploy-minitwit-api`. The steps for these jobs are exactly like the ones from above but for the minitwit-api service. The workflow gets triggered on push to the development (default) branch or can be triggered manually on any branch.

- `ci formatting` - consists of two jobs `format-minitwit` and `format-minitwit-api`. Both of these jobs use a marketplace action (Jerome1337/gofmt-action@v1.0.4) [https://github.com/marketplace/actions/check-code-formatting-using-gofmt] to perform formatting (go fmt) on the root directories and fails on code not meeting the formatting standards. The workflow gets triggered on push and pull request to the development (default) branch.

- `format` - consists of `generate-report`, a job that generates a pdf report from the markdown files, uploads the pdf as an artefact and then commits and pushes the pdf report to 'report/build'. The workflow gets triggered on push to development (default) branch or branches matching the 'docs/*' pattern.

- `sonarcloud` - enabled by installing the `SonarCloud` Github App. Performs linting and checks for bugs, vulnerabilities, code smells and security hotspots. The cicd chain get triggered on pull requests to any branch.

- Organization of your repositor(ies).
    - That is, either the structure of of mono-repository or organization of artifacts across repositories.
    - In essence, it has to be be clear what is stored where and why.

- Applied branching strategy.

A feature based branching strategy has been used for this project, that is anytime a team member desires to add a new feature, a new branch with the name of that feature is created. Once the changes has been implemented a pull-request is made and reviewed by another team member before it is merged with the development branch, which functions as the main branch.

- Applied development process and tools supporting it
    - For example, how did you use issues, Kanban boards, etc. to organize open tasks

Distribution of tasks among the team members has been done through GitHub's issues and a Kanban board. When new tasks came up they were added as issues on GitHub and assigned a team member. These issues were tracked on the Kanban board using a standard layout containing sections for TODO, In Progress, Under Review and Done. By using both Github Projects and Github Issues we have seamless integration between issues and kanban tasks which gives us a better overview of the project status.

- How do you monitor your systems and what precisely do you monitor?

Monitoring is done with Prometheus, where various metrics is defined in the application. Prometheus scapes the application for these metrics once every 5 seconds. These metrics are pulled by Grafana which has a built-in customizable dashboard for visualizing these metrics. Specificaly we monitor the following targets:

- `minitwit_ui_usertimeline_requests` - asdfadsf
- `minitwit_ui_personaltimeline_requests` - asdfadsf
- `minitwit_ui_unfollow_requests` - asdfadsf
- `minitwit_ui_follow_requests` - asdfadsf
- `minitwit_ui_addmessage_requests` - asdfadsf
- `minitwit_ui_homepage_requests` - asdfadsf
- `minitwit_ui_register_requests` - asdfadsf
- `minitwit_ui_login_requests` - asdfadsf
- `minitwit_ui_logout_requests` - asdfadsf
- `minitwit_ui_total_requests` - asdfadsf
- `minitwit_ui_register_requests` - asdfadsf
- `minitwit_api_register_requests` - asdfadsf
- `minitwit_api_messages_requests` - asdfadsf
- `minitwit_api_messages_per_user_requests` - asdfadsf
- `minitwit_api_follow_requests` - asdfadsf
- `minitwit_api_total_requests` - asdfadsf
- `minitwit_api_latest_execution_time_in_ns` - asdfadsf
- `minitwit_api_register_execution_time_in_ns` - asdfadsf
- `minitwit_api_messages_execution_time_in_ns` - asdfadsf
- `minitwit_api_messages_per_user_execution_time_in_ns` - asdfadsf
- `minitwit_api_follow_execution_time_in_ns` - asdfadsf
- `minitwit_api_authentication_middleware_execution_time_in_ns` - asdfadsf
- `minitwit_api_latest_middleware_execution_time_in_ns` - asdfadsf

- What do you log in your systems and how do you aggregate logs?

The goal for the project was to implement and utilize the ELK stack for analysing and aggregating logs on Kibana, but there were manu challenges making the stack work with Docker Swarm. For the final release of

this project the ELK stack has not been fully implemented, hence there are no logs collected and available through Kibana nor Elasticsearch. Contrarily the internal logging library of Golang is being used to some extend. HTTP responses and errors are being logged locally, but not collected by ELK stack, due to our challenges with Docker Swarm. The ELK stack is implemented in the application using Filebeat to collect and ship logfiles to Elasticsearch without the L in ELK. That is, Logstash has not been included in the logging stack for this application. Kibana is supposed to fetch logging data from Elasticsearch, but when the application switched to Docker Swarm, Elasticsearch was not receiving any logs. Although the stack is not fully functional in the final application, it was working properly before transforming to a Docker Swarm cluster.

- Brief results of the security assessment.
- Applied strategy for scaling and load balancing.

In order to secure the minitwit application for large amounts of users and operations and ensure a high level of availability the system has been set up using Docker in Swarm mode. The system is operating with a single swarm manager connected to two worker nodes forming a cluster. With Docker Swarm you can simply add more replicas of already running containers and let the manager node handle the distribution of the containers across the swarm. In case of failure within one of the worker nodes, Docker is capable of detecting this failure and spinning up new containers on the failing node. Docker Swarm also comes with internal load balancing, which is used for this project. That is, the manager node is capable of routing incoming requests to the worker nodes in order to maintain the best performance possible.

Another benefit of Docker Swarm is overlay network and service discovery features that are enabled when using it as an orchestration tool. All containers launched by the manager are added with its own unique DNS name such that we can access and investigate separate containers with ease.

Lessons learned

Evolution and refactoring

Over the time that we spent developing this project, we saw the potential in using tools to assist our work. We added static code analysis, benchmarking and error logging. All of these three things had a heavy impact on the evolution of the application. We benchmarked the application both internally and externally to figure out weak points with regards to efficiency and find potential bottlenecks. We also applied logging to detect any mistakes that we could have missed. To ensure that our codebase was bug-free, consistent and understandable we also used the SonarQube static analysis tool.

All of the elements mentioned here played a big role in the evolution and refactoring of the project. The changes we implemented reflects the results of these tools and had an overall positive impact on how we matured the application.

Operation

The application was deployed and active while the simulator was pushing out messages. It was an interesting experience to observe the application and how it responded to various requests. With the monitoring tools, we had available we were able to spot issues that occurred and pinpoint which endpoints were affected. It was also helpful to see the data provided by the teachers benchmarking the performance.

Over time the application went from a single one node service to a more sophisticated manager-worker architecture that utilized load balancing and replication. By using these features we were able to scale our services horizontally to meet increasing traffic.

Monitoring

We discovered the importance of instrumenting production systems with metrics and visualizing them through graphs. Without having some sort of monitoring solution, the system is essentially a black box. As an example, after setting up Prometheus and Grafana, we discovered that one of our endpoints took on average about 10 seconds to respond to requests which were unacceptable. Through Grafana, we were able to pinpoint the responsible handler function, and we discovered that the slow execution time was due to an inefficient database query. This was not an issue during the first weeks of the system but became a problem as the size of the database tables increased. We were able to fix the issue by creating an index on a field of one of our database tables.

Maintenance

Through the use of the Docker Swarm orchestrator, we were able to establish a system in which we could do maintenance to our containers without turning off the application. Because of our usage of 2 container replicas, we can at any time take one down and push new code to it while the other is serving clients.

While pushing application updates was a pretty seamless experience, the database migration was a bit more clunky as we could not find a way to keep the service online while we swapped out the database engine.

DevOps style of work

Compared to previous development projects at ITU, our workflow has shifted significantly in this course. The workflow of DevOps has been much more structured and automated. The repository has functioned as the central component of our work by both hosting our codebase and serving as a communication hub through issues. Branching was used extensively to separate the main development codebase from new feature implementations, thus allowing the development branch to always be in a functioning state.