IT UNIVERSITY IN COPENHAGEN

DEVOPS 2023

DevOps dudes: MiniTwit

Nikolaj Sørensen (nikso@itu.dk) Daniel Kjellberg (dakj@itu.dk)
Jeppe Lindhard (jepli@itu.dk) Johan Flensmark (jokf@itu.dk)
Benjamin Thygesen (beth@itu.dk)

Course Code: KSDSESM1KU
Course Manager: Helge Pfeiffer

24. May 2023

Contents

1 Introduction

This report documents the creation of the application named *MiniTwit* which is made by the group named *dudes* for the DevOps course at IT-University of Copenhagen. The project is hosted by GitHub at github.com/Lindharden/DevOps, and the application is running at 157.230.76.157:8080.

In section 2 we describe the system and architecture of the MiniTwit application. Here we go through the dependencies of the application and how they interact with each other. In section 3 we describe the process of creating MiniTwit. This includes the team organization, and the setups that we used in order to do DevOps. Finally in section 4 we describe the issues we ran into underway, and the main lessons we have learned.

2 System's Perspective

This section documents the architecture of the MiniTwit application. We will describe the individual dependencies of the application and why we choose them. We will also describe what the current state of the system is.

2.1 Architecture & Design

The MiniTwit application relies on a lot of dependencies, which interact with each other in many ways. Figure 1 shows all of these interactions. In figure 1 it can be seen that a pipeline of actions is initiated every time someone commits to the MiniTwit GitHub repository. When someone commits a change, the GitHub Actions initialize our CI/CD setup, which is described in section 3.3. This setup deploys the image of the application and all it's dependencies, which are described in section 2.2, to the Docker Hub. From the Docker Hub, everything is deployed to DigitalOcean which hosts our application. On DigitalOcean we use a replication setup, described in section 3.6, which ensures availability of the application.

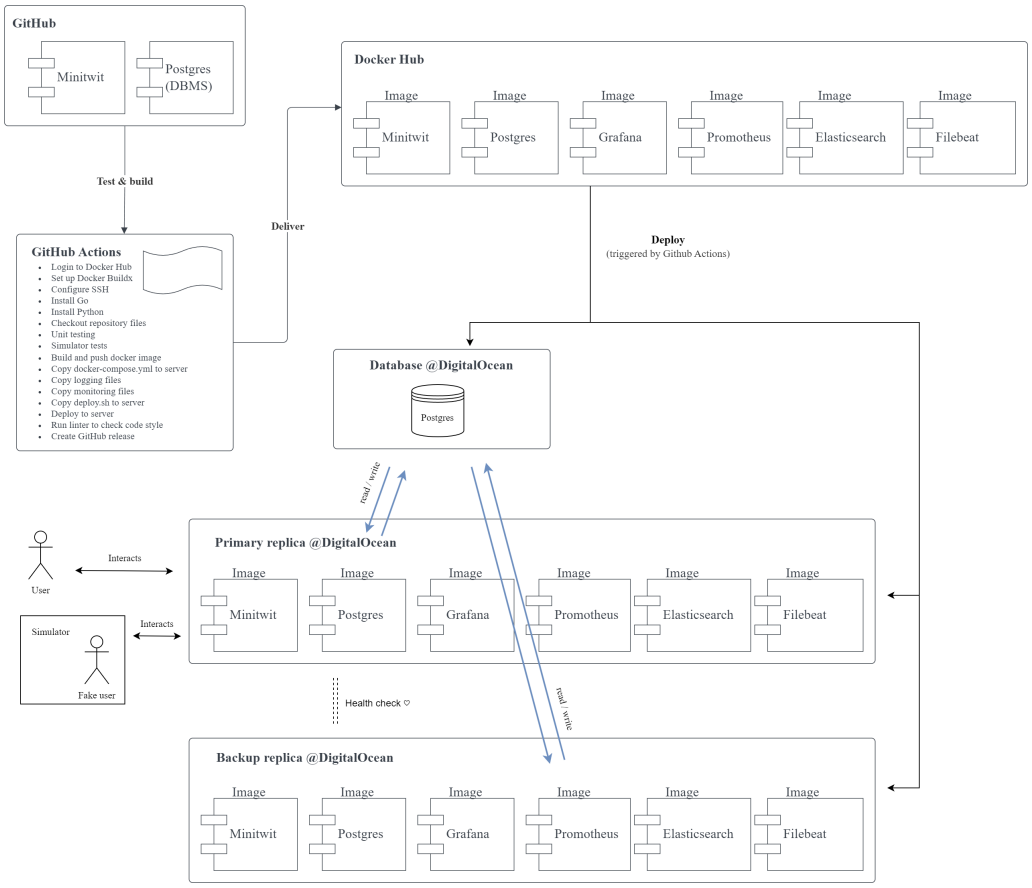

Figure 1: Architecture of the MiniTwit application. The diagram shows which dependencies are involved in the application, and how they interact with each other.

2.2 Dependencies

This section describes the different dependencies used in our application, and we argue for the choice of each technology.

Docker We use Docker[1] to containerize all of our dependencies, into convenient images which are far easier to transport and deploy. Without utilizing containerization, like Docker offers, it would be a struggle to keep track of every tool we use and all the technologies their depend on, and it would require a lot of steps every time we would have to run our application. Docker allows us to package everything together, and lets us run everything from one single command. The reason for using Docker, over other containerization tools such as Kubernetes, is because we already have experience using Docker from previous courses, and because it offers all the functionality we need for our application.

[1] docker.com

Go & Gin We choose to utilize Gin[2] for our web-application. Gin is an web framework written in Go/Golang, which allows for development of fast and stable web services. We choose to use Gin as it's faster than many other web development frameworks, including Flask which is the original framework that the MiniTwit application was using. We wanted our web framework to support HTML rendering, and to support the use of relational databases, as we wanted to be able to reuse as much of the original MiniTWit application as possible. Gin supports both of these features while being scalable and offering type safety, which is important for writing maintainable code, and therefore we though that Gin was very suitible for our MiniTwit application. Additionally we also saw this project as an perfect opportunity to learn Go, and to use Gin.

Gorm In Go, one of the most popular ORM libraries is Gorm[3]. We chose this library because it is one of the most mature libraries we could choose out of the possible ones for Go. Gorm has the basic ORM features we needed like creation, update, and deletion of database objects. With Gorm we could create objects in Go that has the fields needed for the database.

Vagrant

Prometheus & Grafana

Elasticsearch, Filebeat & Kibana To facilitate logging in our Minitwit application, we decided to implement an EFK stack. The EFK stack is a great fit for our Minitwit application as it provides a lightweight log shipper in Filebeat which can collect logs from the application, parse them and forward them to a storage component. It provides an efficient storage and analytics engine in Elastic search, which enables efficient handling of large data volumes and is excellent for full-text search. Lastly, it provides a simple to setup dashboard in Kibana that allows for filtering queries on the data stores in elastic search, such that we only display the data that we find most valuable.

DigitalOcean We have DigitalOcean as our cloud infrastructure provider. DigitalOcean offers a user-friendly interface for setting up and deploying Droplets (VMs) and Volumes (storage). In general, it is known for its simplicity and affordability which makes it ideal for our Minitwit application.

Github & GH Actions To facilitate our CI/CD workflow we utilize GitHub Actions because it is integrated in GitHub, which is the version control system we use for our repository. This way we don't have to involve other platforms, and we can keep everything in one place. Also, since it is integrated in GitHub, we can see the status of individual

[2] gin-gonic.com
[3] gorm.io

commits when we push them. This way we can see whether individual commits build (or contain errors), and we can see whether individual commits pass tests. Lastly, it allows for Continuous Delivery and Deployment, which means we can automate the delivery or deployment of our changes all with the same system.

2.3 Current state of the system

3 Process's Perspective

In this section we present how we organized the development process.

3.1 The team

Dudes is a *close-knit team* consisting of five Master students. We value an in-person working environment when possible, which also means that a majority of project work and general interactions have taken place in-person at school during the course hours. With our collective expertise, we have tackled most challenges collaboratively, working together to find solutions. This approach has fostered a seamless workflow, enabling efficient problem-solving and encouraging a sense of shared accomplishment within our team.

Our team has utilized GitHub Issues as a centralized tool to track and manage our tasks. By documenting each task as an issue, we have maintained a clear overview of our work and assigned responsibilities. We implemented issue templates in GitHub, providing a standardized structure for creating issues, ensuring that all essential details were included consistently. This approach enhanced clarity and efficiency in our communication, making it easier for team members to understand and address the tasks at hand.

3.2 Version control setup

Our version control system setup revolves around a monorepo approach, providing a centralized repository to manage our codebase. We have chosen this approach to consolidate all related code in a single repository, promoting the ease of code reuse and streamlining collaboration. To enhance our workflow and issue tracking, we utilize GitHub's built-in issues extensively. The issues created in GitHub follows predefined templates, defined with GitHub issue templates, in order to ensure consistent issue management and to establish guidelines for issue creation, assignment, and resolution. These templates provide a structured framework for issue creation, ensuring all necessary information is captured for the respective type of issue, whether it be bug reports, feature requests, or security vulnerabilities. Additionally, we have implemented a well-defined branching strategy, consisting of a main branch as the primary branch, and subbranches designated for the development of single features. The subbranches are named according to a set of naming schemas in order to provide clarity and organization. The naming schemas consists of a predefined tag followed

by a relevant name, such as "feature/feature-name" or "bug/bug-name". Before merging any changes into the main branch, a set of checks is performed to ensure code quality and stability. This branching strategy enables seamless feature development, bug fixing, and code maintenance, ensuring a streamlined development process and promoting collaboration within our team. To track releases and provide visibility into the version history, we utilize the GitHub Releases feature. This feature is configured to automatically create a new release for each deployment, ensuring the latest release is always up-to-date with the deployed version. In addition to the automatic releases, we also implemented a weekly release schedule. This manual process allows us to consolidate any significant and relevant features or updates that have been developed during the week. By creating weekly releases, we provide a clear and structured approach to showcasing our progress and informing about the latest developments.

3.3 CI/CD

For our CI/CD setup, we leveraged the capabilities of GitHub Actions to automate the verification process when code is merged into the main branch. GitHub Actions allows us to define custom workflows that encompasses various stages of our pipeline. Within these workflows, we incorporated essentials steps to ensure code quality and reliability.

Firstly, the actions execute unit tests to validate the functionality of our codebase, ensuring that it meets the expected behavior. Secondly, we integrated static analysis tools into the workflow, specifically Snyk, CodeQL, and SonarCloud. These third-party services thoroughly analyze our code to detect potential security vulnerabilities, code smells, and many other potential issues, assisting in maintaining a high level of code quality. Lastly, once all the checks pass successfully, GitHub automatically builds a Docker image of our project. This image is then deployed to our hosting service, ensuring that the latest version of our application is readily available to our users.

To further enhance our development workflow, we integrated GitHub Actions with the pull request feature. This integration enables us to run tests and analysis on proposed changes before merging them into the main branch, minimizing the risk of introducing faulty code into the main branch.

The integration capabilities and the support for automated tests provided by GitHub Actions were key factors in our decisions to choose this CI/CD solution. Our setup streamlines the development process, automates quality checks, and ensures our code is validated and ready for deployment upon introducing changes.

3.4 Monitoring & logging

The monitoring system we have implemented for Minitwit is Prometheus for the following reasons. Prometheus supports a wide range of metrics. Prometheus works well with the Go programming language. Prometheus can easily be integrated with dashboard tools such

as Grafana. Prometheus can alert the team should any metrics surpass certain thresholds. We have implemented application monitoring and infrastructure monitoring through the following metrics:

- **Number of tweet requests per hour** - This helped us determine the amount of load our application is under. We will be able to see when something critical is happening to the application, which is arguably the most important feature.
- **Error codes** - This metric shows us the total amount of errors which the system reports. This metric can be used to determine whether everything runs as expected. If the dashboard displays an unusual amount of errors, this might be a hint that something is going on with the system.
- **Response time** - This metric tracks how many milliseconds a user has to wait for for a request response. This is useful to see if any network issues arise and also to ensure that we uphold the expected average response time of <50ms included in our API SLA.
- **Hardware load** - This metric reports system load, CPU and memory usage, disk I/O and amount of network transmit and receive requests. This can be useful for determining whether everything runs as expected and whether the hosting server has sufficient resources to handle the incoming load.

We use a Grafana dashboard to display the data collected for each metric through Prometheus, which can be seen in figure 2. The dashboard is configured through a JSON file.[4] Both Prometheus and Grafana have been integrated into the application through the *docker-compose.yml* configuration file.

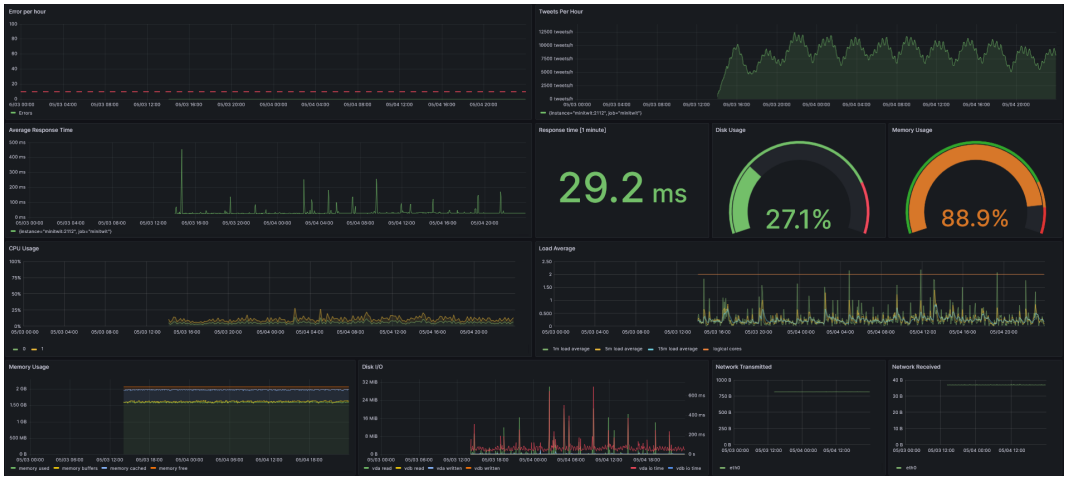

Figure 2: Grafana dashboard displaying the data collected with Prometheus.

[4] github.com/Lindharden/DevOps/blob/main/monitoring/grafana/dashboards/main.json.

The logging system we have implemented an EFK stack consisting of Elastic search, Filebeat and Kibana. We used the *Zap* package for Go to nicely format and emit log messages that Filebeat collects. We have two types of log messages, the first being the native Gin Gonic log messages that happen for every request. The second one is the log messages that we have implemented in the code, which are formatted by *Zap* in JSON as follows:

```
{
    "level":"error",
    "timestamp":"2023-03-25T13:09:38Z",
    "caller":"controllers/loginController.go:130",
    "msg":"Could not find session",
    "user":"John Doe",
    "stacktrace":"..."
}
```

The Kibana dashboard can be seen in figure 3. The EFK stack is integrated into the application through the docker container.

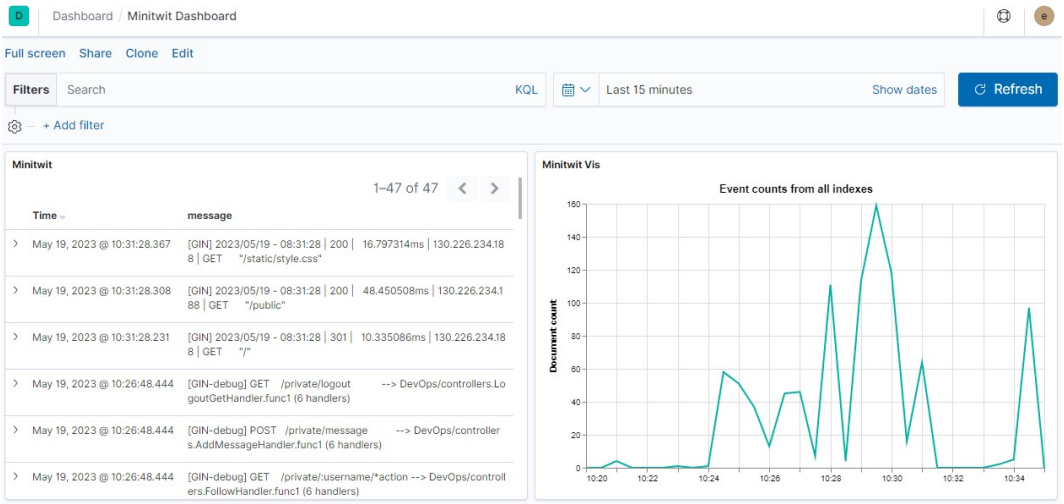

Figure 3: Kibana dashboard displaying the logs that are collected and organised through Filebeat and Elastic search.

3.5 Security assessment

In our security assessment we have considered which risks the infrastructure and application are vulnerable to. The risks we have assessed are mapped in a risk assessment matrix which can be seen in figure 4. More concretely they are:

- **Compromised Github repository** - The github repository owner uses two-factor authentication. Furthermore, all users must make a pull request and have it approved to change the code on the main branch. This makes it unlikely that the github repository is compromised. However, a compromised GitHub repository would mean that the

attacker would gain access to one of our GitHub admin accounts, which would mean they have the ability to change our code, making it a significant impact risk.

- **Whitebox exploits** - As our GitHub repository is public, anyone can see our code and create exploits for it. Therefore if the attacker finds a reason and an opportunity to create a whitebox exploit, it would likely happen. However, as we continuously update our code and update vulnerable packages the chance of an exploit being possible, is very low. As the attacker knows what our code base looks like they can plan their attacks towards having as large of an impact as possible. In addition, the attacker can see all the dependencies we use, meaning they can utilize any exploits found in our dependencies, making it an extensive impact risk.
- **DDoS attack** - Since our application is not that popular it is not likely that it will get DDoS'ed, but it is still possible since our application is public. A DDoS attack would mean that our application would be unreachable for a period of time. This would be a moderate inconvenience for us and our users, making it a moderate impact risk.
- **SQL injection** - Since the database abstraction layer library GORM sanitizes all keywords, an SQL injection attack is very rare, basically impossible. SQL injection would mean that an attacker performs attacks directly on our database, by sending arbitrary SQL queries to our database, making it an extensive impact risk.
- **DB leak** - The only ways our database can be leaked, is by someone compromising our code repository and performing some kind of whitebox exploit, or by performing SQL injection. As we deemed these risks as "rare", our database leaking must also be a rare event. If our database leaked, the attacker would gain access to usernames, email's and hashed passwords of our users. As none of this is sensitive information the impact would be negligible. However, the hashed passwords could be used in an brute-force attempt to identify passwords, making it a negligible impact risk.

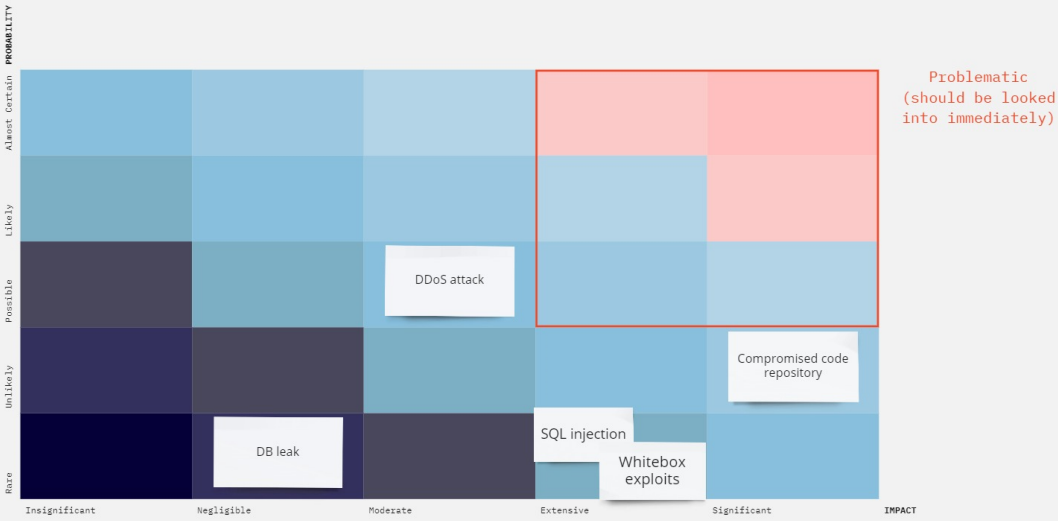

Figure 4: Risk assessment matrix. The vertical axis indicates the probability of a risk while the horizontal axis indicates the impact of a risk. Risks in the top-right corner are considered problematic and should be handled accordingly.

None of the assessed risks are placed inside the problematic area in the risk assessment matrix in figure 4. Therefore, we have not taken any actions against these risks, we have simply decided to accept them for now.

3.6 Scaling & redundancy

To ensure fault tolerance we introduced redundancy in the form of a replica. We created an additional droplet on DigitalOcean and changed our Vagrantfile accordingly to have identical instances running on each of the droplets. One of the instances act as the primary one, for which all traffic is directed to. The other instance serves as a backup. With the use of heartbeats the server can keep track of the primary replicas state, and should it fail to show signs of life, the backup replica will take over as primary.

In addition to fault tolerance, having a primary and backup replica also enables rolling updates which eliminates the inevitable downtime caused from deploying updates. We did not get around to do it for this project, but we had a plan of how we wanted to do it. In order to increase our availability, we will first be updating the backup replica and make it our primary server, then we can shut down the replica which was previously the primary one and update it as well. DigitalOcean will make this easy as it automatically switches the primary server to be the replica which is running.

3.7 AI assistance

For project work we have consulted ChatGPT for setting up Docker, however, without much benefit. For writing the report we have used ChatGPT in a couple of places to make some

paragraphs more concise, which have helped to stay within the words limit. Most of the gain, though, have been counterweighed by having to include this subsection xd kek.

4 Lessons Learned Perspective

One of the biggest issues throughout the project was to setup new services in Docker. We did not have prior experience with many of the services such as Grafana, Prometheus, and Kibana. Getting these services to work properly took us a decent amount of time. There was not a simple solution, other than trying until we got it right. Working with docker we found out it is hard to narrow down a reason to a problem as the logs are not always that elaborate.

Literature