mini-x report

Course Code: BSDSESM1KU

Student	Email
Marius Thomsen	mariu@itu.dk
Marius W. S. Nielsen	mawn@itu.dk
David A. Feldner	dafe@itu.dk
Markus Grand Petersen	mgrp@itu.dk
Michael Daniel Fabricius	midf@itu.dk

1 Systems perspective

1.1 Description of the project

mini-x is a blazingly fast twitter/x clone written in Rust with the Actix framework.

1.2 Design and Architechture of the project

mini-x is designed with focus on performance, scalability and type safety. The application is structured into two main parts: the API server and the frontend client, each running concurrently on separate threads as seen in `main.rs`

1.2.1 API Server

The API server is built using Actix-Web, a blazingly fast web framework for Rust. The API server acts as the endpoint the simulation interacts with.

1.2.2 Frontend Client

`TODO: Create diagrams to show the architecture`

1.3 Dependencies of mini-x

- Rust and Actix-web:
    - The backend service is written in Rust, with Actix-web as the chosen framework due to its performance and ease of use in developing web apps. Actix-web handles the HTTP requests and routing
- PostgreSQL
    - Used for data storage in a robust and scalable way.
- Diesel.rs

    - Diesel was used for ORM to ensure safe database interactions. Diesel provides type safety and convinient DSL for rust, such that complex SQL queries can be constructed safely.
- Docker and Docker Swarm:
    - Docker is used for containerization to ensure that the app runs identically across varying environments. Docker swarm manages a cluser of Docker Engines so we can spread workload horizontally.
- ELK stack:
    - ElasticSearch, Logstash, Kibana and Beats was used for logging
- Prometheus and Grafana
    - Prometheus collects data from our api and frontend. The data is then shown in grafana for monitoring

1.4 Current state of mini-x

1.5 Important interactions of sub systems

```
Make UML Sequence diagram that shows the flow of information through your system from user requ
```

```
Make illustrative sequence diagram that shows how requests from the simulator traverse your sys
```

2 Process' persepctive

2.1 CD/CI Explanation

```
A complete description of stages and tools included in the CI/CD chains, including deployment a
```

2.2 Monitering

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

2.3 Logging

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

2.4 Security

```
Brief results of the security assessment and brief description of how did you harden the securi
```

2.4.1 Assets in our system.

In our system, there are six virtual machines hosted on Digital Ocean. Five of them hold an interest in a malicious party. Monitoring provides all endpoints

2.4.2 Assets and their value

- Application: The application has three replicas on three separate virtual machines.
- public information is found here, including usernames.
- 3 nodes worth of computing power
- Database: A single virtual machine with a backup
- All our data, hashed passwords, email, usernames, all messages
- Logging: A single VM with all our logs and errors.
- Users: The users on the application
- Provides value.

2.4.3 Threats and Risks to Assets

- Application:
- DDOS: our application can handle many requests per second depending on the endpoint.
- While our service can handle the simulator and then some. We could put all our VMs to full load with one machine running FFUF in Kali, targeting computationally heavy endpoints.
- Database:
- Injection: All fields are sanitized. The ORM we use is injection-safe. The one SQL query we have uses prepared states.
- Hashed passwords: Here we use bcrypt to encrypt them with salted hashing
- Man in the middle: We send our data from the application to the database using HTTP
- Logging:
- Verbose error messages: having better responses from your other attempt will enable better attacks.
- GDPR theft: Some user data can be acquired.
- Uptime:
- Our system is vulnerable to DDos attack affection up time. Decreased will affect the number of users.
- Users:
- Obscene content: There is no content filter, all content is allowed, which could cause users to leave
- no service: If our service is down, users leave
- no content: Without content, users don't stay

2.5 IaC Strategy

```
Applied strategy for scaling and upgrades
```

3 Lessons learned perspective

```
Describe the biggest issues,
how you solved them, and which are major lessons learned with regards to:
Evolution and refactoring, Operation and Maintenence
of your ITU-MiniTwit systems.
Link back to respective commit messages, issues, tickets, etc.
to illustrate these.
```

```
Also reflect and describe what was the "DevOps" style of your work.
For example, what did you do differently to previous development projects
and how did it work?
```

3.1 Evolution and refactoring

3.2 Operation

3.2.1 Data loss

During the project we had 2 incidents that caused data loss. The first incident happened right at the start of the simulator when we had deployed the database without a volume. We had also set up a workflow for when code was pushed to main or merged with a PR. This meant that approving any PR or pushing to main would cause our database to be deleted. Since everyone on the team was not aware of this issue, we ended up deleting our database. Because of a recent backup and quick action we quickly got up and running again - now with a volume for the database.

When we switched to having our postgres instance on its own server, we had set the password to be 'postgres' on a postgres server running on port 5432 - the default postgres port. This caused our database to be deleted by adversaries before we changed the password. Quite a silly mistake, we thought having a insecure password for a day or two would be fine, but that was not the case. Again due to backups we were able to restore some of the data.

3.2.2 Adding indexes live in production

After looking at our response time from day to day we noticed that it was higher as time went. This was a big problem for us. After hooking up our postgres database to a locally running client of mini-x with timers in the code, we could see that the database was the culprit and our application was still blazingly fast. After seeing which queries were slow we put an index on the database while it was in production and immiedietly resolved our speed issues. Some queries were still slow, and we fixed those by rewriting our ORM code to another query.

3.3 Maintenence

4 Usage of LLM's in mini-x

```
Mention LLM tools how and where we used them and which ones did they help, speed up or slow us
```

Figure List

API Squence Diagram

(Sequence Diagram)

Arhitecture Diagram(Might be contained in 3+1)

3+1 Model viewpoint

3+1 Component viewpoint

3+1 Deployment viewpoint

Dependencies Diagram (NOT REQUIRED but cool)