Minitest 1

DevOps, Software Evolution and Software Maintenance

KSDSESM1KU

IT-University of Copenhagen

May 23, 2024

Carl Bruun	carbr@itu.dk
Christian Stender	ches@itu.dk
Nadja Brix Koch	nako@itu.dk
Theis Helth Stensgaard	thhs@itu.dk

1 System's Perspective

1.1 Design and Architecture

Design and architecture of your ITU-MiniTwit systems. All dependencies of your ITU-MiniTwit systems on all levels of abstraction and development stages.

Overview of System Design

Architecture Diagram

1.2 Technologies and tools

That is, list and briefly describe all technologies and tools you applied and depend on.

List of Technologies and Tools Utilized

- **Go**: A statically typed, compiled programming often used for backend development.
- **Go/Gorilla**: A toolkit for Golang, that provides packages for building web applications. Used for routing and session management.
- **SQLite**: A lightweight SQL database engine that is ideal for embedded database applications. Used in initial setup and for testing.
- **Docker**: A platform for shipping and running applications, ensuring consistency across different environments. Containerization is used deliberately throughout our system and in delivery.
- **Vagrant**: A tool for building and managing virtualized development environments. Used initially for VM instantiating.
- **DigitalOcean**: A cloud infrastructure provider offering scalable compute and storage solutions. Used for deploying and managing the application and database.
- **CircleCI**: A continuous integration and delivery platform that automates the build, test and deployment processes of our project.
- **CodeClimate**: A platform that provides code review, offering insights into code quality, maintainability and test coverage, helping us ensure high standards and to improve code health.
- **Prometheus**: An open-source monitoring toolkit used for monitoring metrics in our cloud environments.
- **Grafana**: An open-source analytics and monitoring platform that integrates various data sources to visualize our data and metrics.
- **PostgreSQL**: A open-source object-relational database system, known for extensibility and speed. Used after the transition from SQLite managed by DigitalOcean.
- **Promtail**: An agent that ships log file to Loki and part of the Grafana logging stack. Used for gathering logs.
- **Loki**: A log aggregation system designed to be scalable. Used with Grafana for log querying and visualization.
- **Nginx**: A high-performance web server and reverse proxy server. Used for load balancing.
- **CertBot**: A open-source tool for automatically using Let's Encrypt certificates to enable HTTPS on web servers.
- **Terraform**: An infrastrcucture as code tool that allows users to define and provision data center infrastructure using a high-level configuration language.

1.3 Subsystem interactions

Important interactions of subsystems. For example, via an illustrative UML Sequence diagram that shows the flow of information through your system from user request in the browser, over all subsystems, hitting the database, and a response that is returned to the user. Similarly, another illustrative sequence diagram that shows how requests from the simulator traverse your system.

1.4 Current state

Describe the current state of your systems, for example using results of static analysis and quality assessments.

1.5 Technology choices

MSc should argue for the choice of technologies and decisions for at least all cases for which we asked you to do so in the tasks at the end of each session. Our initial choice of language for refactoring ITU-MiniTwit was based on a detailed feature mapping of the system and a comparison of programming languages (see Appendix **??**). This analysis led us to initially select Crystal/Kemal. However, we soon discovered that Kemal's documentation was insufficient, making it challenging to work with. Consequently, we switched to Golang, which offered similar features but had much more comprehensive documentation.

2 Process' Perspective

This perspective should clarify how code or other artifacts come from idea into the running system and everything that happens on the way.

In particular, the following descriptions should be included:

A complete description of stages and tools included in the CI/CD chains, including deployment and release of your systems. How do you monitor your systems and what precisely do you monitor? What do you log in your systems and how do you aggregate logs? Brief results of the security assessment and brief description of how did you harden the security of your system based on the analysis Applied strategy for scaling and upgrades In case you have used AI-assistants during your project briefly explain which system(s) you used during the project and reflect how it supported/hindered your process.

2.1 CI/CD Pipeline

Stages of the Pipeline

Tools and Technologies

Deployment and Release Processes

Docker, feb 29

2.2 System Monitoring and Logging

Monitoring Strategies and Tools

In managing our system's health, we rely on two key tools: Prometheus and Grafana. Prometheus is a push/pull-hybrid whitebox monitoring system from which our instrumented application pushes our defined metrics, which are then pulled/scraped by the Prometheus central system... (ik færdig/skriv om)

As from the Monitoring Maturity Model by James Turnbull[**?**], our monitoring strategy combines proactive elements, such as monitoring uptime and tracking error rates metrics to anticipate and prevent issues, with reactive elements, such as tracking specific error codes and status of specific user requests to respond promptly to any issues that arise.

Key Metrics Monitored

We have defined monitoring metrics for both the API and the web client. However, given the API's consistent workload from the simulation, this has been our primary focus. We have defined a total of 14 metrics for the API:

- **http_requests_total**
- **database_accesses_total**
- **errors_total**
- **get_user_id_requests**
- **get_user_id_requests_failed**
- **follow_requests**
- **follow_requests_failed**
- **unfollow_requests**
- **unfollow_requests_failed**
- **post_message_requests**
- **post_message_requests_failed**
- **not_found**
- **bad_request**
- **internal_server_error**

These monitoring metrics are visualised in a Grafana dashboard named `Minitwit_Dashboard` as shown in Figure **??**.

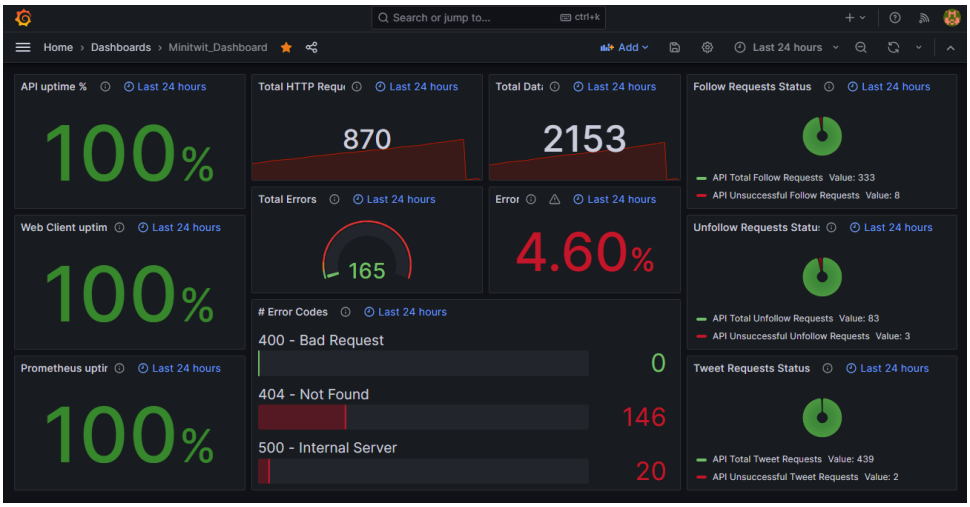

Figure 1: Minitwit_Dashboard - API Monitoring

The key metrics are categorised and displayed as follows:

Total HTTP requests and Database accesses, **Total Errors and Error Ratio**, **Error code distribution** of the total errors, and **Specific user request success/failure** (follow, unfollow, tweet).

Additionally, the visualisation shows the uptime percentage of the API, Web Client, and Prometheus, which is derived from predefined Prometheus metrics.

Log Aggregation Techniques

april 1, mention log rotation, possibly mention that we also tried setting up ELK at first

2.3 Security Assessment

Assessment Results

Security Hardening Approaches

2.4 Scaling and Upgrade Strategy

april 23

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 maintenance 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 Key Challenges Faced

Problem-Solving Approaches

- Quick to notify each other on Discord, discuss severity and decide course of action. - If one member was more involved with an issue, they would sit together with someone not involved -¿ both insider knowledge of how it is set up and should work, with a fresh mind seeing it from a different perspective.

Lessons Learned from Issue Resolution

3.2 Evolution and Refactoring

- Importance of modular solutions open for extension and easy to refactor, we experienced that often setting up something resulted in more issues than solved issues. Some of this is also due to the way the course is structured. First adapt everything to one type of database -¿ refactor everything to a new database. Same exercise with making the system scalable and going from docker to docker swarm. - Prioritisation

3.3 Operations and Maintenance

- Time consumption: bite the bullet and accept not everything can be done 100% and if things break there might not be time to fix all of it. - Importance of tooling such as monitoring; Easily see if things are working properly at different layers of the system (frontend, api...), logging; Good tool for error debugging, codeclimate;

3.4 Reflection on DevOps Practices

Log: feb9 (git branch setup, later march2 argue if static analysis removes need for manual reviews), feb25 (CI/CD, using pipelines in your project),

4 Appendix

A Choice of programming language

	Python3 Flask	Crystal Kemal	Ruby Sinatra	Golang
Our experience	Moderate experience	No experience	No experience	Moderate experience
Types	Dynamically typed	Statically typed	Dynamically typed	Statically typed
Performance	Moderate performance	High performance	Moderate performance	High performance
SQLite support	yes	yes	yes	yes
Middleware support*	yes	yes	yes	yes
Release date	2010	2016	2007	2011
Deployment	Deployed using virtual environments to manage dependencies.	Compiled into a single binary executable with all its dependencies.	Requires the presence of the Ruby runtime environment and dependencies.	Can be deployed as single binary executables.

Table 1: Programming language strengths

* Middleware in web APIs is used as a design pattern to intercept and manipulate HTTP requests. `https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-middleware`

B Choice of CI tool

As our code is stored on Github, we've eliminated Gitlab CI as as an option. Travis CI is only free for a single month (for students), and we've also eliminated this.

Github Actions	CircleCI	Our considerations
Is free ... "Cheapest for people with public repositories"	3000 minutes for free pr month	We have a public repo
Runs full pipeline automatically	Can be paused and wait for human interaction	We don't have a usecase for needing human intervention before deploying if the code passes all the tests we stup, and the CircleCI feature (even though nice) is not needed.
More than CI/CD - can also automate manual tasks like generating changelogs or versioning releases	Only CI/CD, but specialised in this.	We only need CI/CD for now.
Slower than CircleCI	Faster than Github Actions	Do we need speed?
Only Windows, MacOS and Linux	Every operating system	We only need Linux
Configuration can be split in mulitple files	Single file configuration	Cleaner setup with GHA?
Docker support is still a bit buggy on GHA, works only with Linux.	CircleCI has perfected its Docker support over the years to make it (almost) the de-facto environment for running builds.	We will use docker.
More granular control by exposing all commands. Complexity increase.	Less complex, has built in commands for often-used services. Less control	We don't know what we need yet - so maybe more control is nice, but it being easy is also nice.

Table 2: CI tool strengths.