DevOps, Software Evolution & Software Maintenance Rapport

Course code: KSDSESM1KU

Group O - Oh sorry

Repository: group-o

Christoffer Tofteng (chtof@itu.dk)
German Alexander Garcia Angus (gega@itu.dk)
Hristiyana Hristova Toteva (hrto@itu.dk)
Lasse Sonn (lson@itu.dk)
Mikkel Munkholm Blak Nilsson (muni@itu.dk)
Nohely Gedeon (noho@itu.dk)

May 2023

Contents

1 Introduction

2 System's Perspective

2.1 Design of the systems

The design of our system follows a relatively simple structure. The system has been separated into smaller components in order to maintain a loosely coupled application. These components consist of a front-end system, a remote server, a database, a monitoring system, a logging system, as well as a simulator to simulate real users.

2.2 Architecture of the systems

Our version of the MiniTwit application consists of a web application, an API, and a database. The application is inspired by Twitter and similar to Twitter, it allows users to create a profile, log in, post short messages to a public timeline, and follow and unfollow other users. The web application is supported by our API, which has endpoints to handle those functionalities. The front-end was developed using the JavaScript React.js library with Node.js as the runtime environment. The API was built with C# using the .Net framework. A detailed discussion of why we chose these technologies for our system can be found in the section named Technologies.

2.3 Dependencies

- **C#**
  We use C# and the .NET framework to handle all the API and data manipulation in our application.
- **React**
  The React library is used to visualize our data through the front-end.
- **npgsql**
  The database for our project is NPGSQL which is a .NET version of a postgres database.
- **Prometheus**
  Our way of getting the monitoring data is with Prometheus which ...
- **Grafana**
  To visualize our monitoring data we are using Grafana to show different metrics.
- **SeriLog**
  SeriLog is a C# logging tool which creates logs and formats them before sending them to ElasticSearch.

- **ElasticSearch**
  This tool is a data ingestion toool used to index the logs send from SeriLog in order to allow for easier access.
- **Kibana**
  Kibana is used to query the logs from ElasticSearch and visualize the data from the log.
- **filebeat**
  Filebeat is a logshipper used to ship the logs to elasticsearch
- **nginx**
  We are using the proxy service nginx for port forwarding in relation to our logging.
- **Docker**
- **Vagrant**
- **DigitalOcean**
- **GitHub Actions**

2.4 Important interactions of subsystems

Given that the front-end is built independently from our API with the front-end being React and the API being C# the front-end accesses endpoints of the API in order to visualize the data and send new data if a new tweet or user is created.

2.5 Current state of the system

2.6 Licensing

As there was no explicit license from the original minitwit application we copied we see this program as our own software and therefore will have to license it ourselves. If we take a look at the denpendencies we use in the project and their licenses we see the following.

- prometheus
  Apache 2.0 license
- grafana
  GNU AGPLv3 license

- SeriLog
  Apache 2.0 license
- ElasticSearch, Kibana and filebeat
  Elastic License and Server Side Public License (SSPL) (Elastic License v2, or ELv2)
- nginx
  2-clause license (”Simplified BSD License” or ”FreeBSD License”
- .NET (C#)
  MIT license
- npgsql
  PostgreSQL License
- React
  MIT license
- Vagrant
  MIT license

Looking at the licences above the GNU AGPLv3 license seems like the strongest and since it is free it makes sense to go with the same[1]. The GNU AGPLv3 is a strong copyleft license which we feel like makes sense for the project in case other people wants to help and make changes or use it as an inspiration for their own work then they have a starting platform.

3 Process' perspective

3.1 Developer interactions

For internal communication in the group we have created a Slack workspace to ask questions, arrange meetings and share documents. We have used teams to facilitate our meetings as teams is also used in the class and has a better.

We had biweekly scrum-like meetings each Tuesday and Friday to give an update on what tasks you had done, which you had started on, and if you needed any help with an ongoing task.

We kept track of work in progress and future work by using Githubs issues function. In the beginning of the project we also used the projects option as a sort of kanban board[2] in order to track when the application was refactored and which steps where still needed.

We have used pull requests in order to have some quality assurance and outside perspective of work when a task was done.

[1] https://github.com/Group-O-Minitwit/minitwit-group-o/blob/main/LICENSE.md
[2] kanban

3.2 Team organization

We are a team of 6 student developers each with different time schedules and therefore we have split us into smaller teams responsible for backend and front-end according to our schedules. We also made use of pair programming and

3.3 Stages and tools included in the CI/CD chain

For the most part our CI/CD pipeline follows the blueprint we were given, where every time changes are pushed to the main branch, the pipeline builds, delivers and deploys the application to DigitalOcean. However in the process of configuring the pipeline, we encountered some minor issues. Issues that were entirely due to lack of understanding the concept and how the different components worked together. Such is the learning process.

When the remote server was originally set up, the `remote_files` directory containing the deploy.sh and docker-compose.yml needed by the server was not copied over to the server, which meant that the workflow could not execute the deploy command and this caused the workflow to fail. To resolve this issue, we added a step called "Copy `remote_files` directory to server" right after the SSH is configured and just before deploying to the server. This step, as the name suggests, copies the `remote_files` directory and its contents to a specified location in the remote server every time the workflow runs. That way the server always has the latest version of the files.

3.4 Repository organization

3.5 Applied branching strategy

3.6 Applied development process and tools supporting it

3.7 Monitoring

Our MiniTwit application in C# .NET now supports monitoring with Prometheus and Grafana as a Dashboard. To achieve this, we Dockerized the application and configured the main settings for Prometheus and Grafana in a YAML file. This allowed us to access the localhost ports for Grafana, Prometheus, and its metrics, and build our dashboard.
Monitoring is an essential aspect of managing IT systems and applications, providing feedback on their performance and quality of service. It can be manual or automated, with the latter usually driven by configuration management tooling. Our application's support for Prometheus and Grafana allows us to perform white-box monitoring, focusing on what's inside the application.
We were able to check the performance of our application and its core metric types, including counter, gauge, histogram, and summary. Grafana and Prometheus are commonly used to monitor key metrics, such as HTTP requests in progress, HTTP requests total, and response size, to identify potential bottlenecks, performance issues, and trends.

Other metrics we monitored include CPU time usage, garbage collector run time, maximum file descriptors, and remote read queries. These metrics provide valuable insights into our application's performance and help identify potential issues that we can address to improve its overall performance.
In summary, by adding support for Prometheus and Grafana, we were able to monitor our MiniTwit application more effectively, gain valuable insights into its performance, and identify potential issues that we can address to improve its overall quality of service.

3.8 Logging

To implement logging in our system, we decided to go with part of the EFLK stack. Namely, ElasticSearch, filebeat and Kibana and replacing Logstash with .NET's own logging library called Serilog. With this setup we are logging all API activity so that we through Kibana can visualize all the data we get.

We decided to drop Logstash as it is has high resource consumption which is not desirable for our system where we can risk getting a lot of API calls due to a large amount of users.

3.9 Security assessment

The output from the Metasploit Framework's Web Application Scanning module (`wmap_vulns`) after scanning a target IP address (157.230.79.99) for vulnerabilities in directories and files seems like there were no vulnerabilities found. The scanner tested various modules to check for SSL, web server, and file/directory vulnerabilities, but there were no positive results indicating that the target site was vulnerable to any of them.

Some of the tests did not receive a response from the target site, which could indicate that the site is blocking requests or the scanner is being detected and blocked. However, without further information or analysis, it's not possible to determine the exact reason why the site did not respond.

The output shows that the scanner found three directories (/Login/, /login/, and /register/) and two files (/register and /login) on the target. For each directory or file found, the scanner performed an HTTP GET request and received a response code.

The response codes are 405 (Method Not Allowed) for the directories, indicating that the HTTP method used (GET) is not supported for those directories. For the files, the response codes are 404 (Not Found), indicating that the files were found but could not be accessed.

Overall, the output suggests that the target has directories and files that may be of interest to an attacker, but the access to them is restricted.

3.10 Applied strategy for scaling and load balancing.

For scaling the application we decided to go with Docker swarm. Docker swarm is a framework that provides both scalability and reliability via replication and

load balancing.

3.10.1 Implementing scaling

Implementing docker swarm with our application wasn't a difficult task, since we were already using docker-compose. The docker-compose file was with only a few tweaks able to be used as the configuration for the Docker swarm cluster. When using the "`docker stack deploy ...`" it was possible with only small adjustments to the docker-compose.yml file to reuse it to create the cluster. The small adjustments primarily included adding replicas configuration to services that needed scaling. This was most importantly our API. The API had up until that point been running as a single container in the docker-compose setup. This resulted in the service being slow, and at times did not process requests, correctly. Another service that we considered upgrading was the database, but after taking it through we concluded that this was a more difficult task than expected. It would require us, to make sure, the two replicas got the same updates, simultaneously, to keep the consistency of the application. This, of course, would not help with scaling the application, since both databases would have to process the same queries, thus doing the same amount of work, as with no replicas.

3.10.2 Implementing availability

Implementing availability is to make sure that all services is constantly running. To begin with we did this with a restart policy

3.11 AI assistance

4 Lessons Learned Perspective

4.1 Evolution and refactoring

4.2 Operation

4.3 Maintenance

5 Conclusion

References

[1] author name. title. year. url.