IT-UNIVERSITETET I KBH

DevOps, Software Evolution and Software Maintenance, Spring 2023

BSDSESM1KU

MiniTwit Project

Group H

Florian Micliuc
flmi@itu.dk

Martin Kirkegaard
marki@itu.dk

Aidan Stocks
aist@itu.dk

Thomas Schauser
hasc@itu.dk

João da Silva
joap@itu.dk

System's Perspective

Throughout this course we were given the task of continuously improving and adding features to a legacy code base. The code base in question was an aging mini version of Twitter called *Minitwit*.

Technologies

Refactoring

Minitwit was originally written in Python 2 using Flask. The goal was to get the website running on a modern framework, this was done by firstly using the library 2to3, which converts Python 2 code into Python 3 code. Lastly to complete the refactoring we rewrote the code to run using the Django framework.

Containerization

Containerization is a term in software development, which references to bundling application components into container images, which can then be run in isolation on a machine. To implement this technology we decided to use Docker. Using containerization solves the issues of manually installing dependencies and issues with running an application on different operating systems. By containerizing our application, the environment used locally closely resembles the environment on our production server.

Continuous Integration and Continuous Deployment

DevOps is a development method where the time from the testing environment to production is rapidly shortened using automation to ensure high quality. Since we are using GitHub as our version control tool, we can use its features for testing before deploying. GitHub has a built in CI/CD tool where you can build a pipeline of checks that must be completed for the branches to be merged.

Cloud

Since we are creating a web app, we need to host it somewhere. We used an Infrastructure-as-a-Service (IaaS) provider called Digital Ocean, due to it being the recommend provider from the course. Digital Ocean allowed us to create multiple server instances (Droplets), to allow for both horizontal and vertical scaling.

Monitoring

Monitoring is the act of getting information about the system. An important aspect of monitoring is choosing what to monitor, since it doesn't make sense to log and measure every part of the system, because this would take an unreasonable amount of space and complexity. Therefore you need to be selective with what metrics you monitor, to give only the insights you require. This can be anything from the technical aspect of uptime/availability, to more subjective things such as how the users rate your service. The tool of choice for this task were Prometheus and Grafana.

Software quality

When writing a piece of software, one can do it fast or one can do it right. If you write something fast to get it to production, then you take on a bit of technical debt, which in itself is not bad as long as it is paid back.

The flawed problem lies in the definition which builds upon the term software quality. Software quality can be hard to measure, however one can use guidelines and tools when writing to assist the developer in making the right decisions.

These tools that analyze code before execution are referred to as static analysis tools.

We used 3 static analysis tools:

1. SonarCloud performs clean code checks. It is implemented directly into our CI/CD pipeline, where it shows potential bugs amongst being a clean code quality gate.
2. Snyk is a tool which shows potential vulnerabilities in used dependencies. Snyk creates pull requests on GitHub to update vulnerable dependencies. We also use Snyk as a quality gate for our Docker images and requirement.txt files.
3. MegaLinter provides consistency to our code, since there are bunch of ways of writing the same code which are all syntactically correct. MegaLinter automatically formats our code to adhere to coding styles.

Logging

We decided to use Django's built-in logging system, as this is robust enough and offers sufficient functionality for our use-case.
The reason for having robust logging is to log errors and exceptions. As an example, when combining it with monitoring we can see if we get certain errors and thus analyze the cause.

Penetration Testing

Websites are often targeted by malicious attacks. In our case, this can come in the form of gaining access to our Droplet, acquiring information regarding our users, or even hijacking a user's account.
To test how vulnerable our server was to these attacks we used a tool called **ZAP** (*Zed Attack Proxy*), which in summary conducts a series of simulated attacks and scans the application's infrastructure to detect possible weaknesses and how the system could potentially be breached.

Swarm

Docker Swarm Branch
load balancing if one node is doing too much work, another node can help. It also has replication. Replcation is when you are running identical instances, so if the main instance crashes the secondary can take over.

We are currently using 4 containers as mentioned earlier, however if there is a failure in one of the containers, we manually have to go in look at the logs and make a change. When you have successfully setup a swarm, it will automatically performs actions if you have a failure. The architecture is setup in a way so there are manager nodes and worker nodes. The job of a manager node is to delicate tasks to each worker node, which could be anything from getting data from the database to reboot the node upon failure.

Terraform

https://github.com/FiveGuys-DevOps/MiniTwit/tree/feature/terraform
Terraform is a infrastructure as code tool, which means it has the ability to interact with different cloud providers using a code interface. This has the benefit of automating a process instead of having to use the GUI of a provider. If you were to have sufficiently setup Terraform with your codebase, you could launch the entire project from any computer with a single command.

Current state

The current state of the MiniTwit in this project is an in-development version. We have all the major features working, including follow/unfollow, twit, login/logout, view more, avatars, and public/private timelines. However, there are unresolved problems such as logging does not displaying as it should, the GUI loading slow (as seen in 2) and other issues. Additionally, there are also 54 unresolved security hotspots discovered by SonarCloud.

When creating a web application where multiple users can connect and share information, trust is at the core of the exchange. Therefore we tried to implement HTTPS to make user data more secure and increase our security against interception attacks.

The docker swarm could not be implemented as there is a limit on how many droplets can we have running at the same time. A ticket has been submitted to digital ocean's team, but unfortunately, no answer has been received. However, a local implementation of docker swarm can be found in the terraform branch in our git repository.

License

The license of a software is important due to it dictating how external users can use it. A limitation on this section is we only look at the python packages using the following library **pip-licenses** and command **pip-licenses –order=license**. This returns a list of all the python packages installed and the corresponding licence as seen in table 1

Since we are using libraries which are under the GNU licence, our software becomes derivative work of the dependencies and thus warrants us to also use the GNU licence and publish the changes.

Name	Version	License
distro	1.7.0	Apache Software License
django-prometheus	2.2.0	Apache Software License
importlib-metadata	4.6.4	Apache Software License
prometheus-client	0.7.1	Apache Software License
tzdata	2022.7	Apache Software License
cryptography	3.4.8	Apache Software License; BSD License
Django	4.1.7	BSD License
SecretStorage	3.3.1	BSD License
asgiref	3.6.0	BSD License
oauthlib	3.2.0	BSD License
sqlparse	0.4.4	BSD License
python-apt	2.4.0+ubuntu1	GNU GPL
PyGObject	3.42.1	GNU Lesser General Public License v2 or later (LGPLv2+)
gprof2dot	2022.7.29	GNU Lesser General Public License v3 or later (LGPLv3+)
launchpadlib	1.10.16	GNU Library or Lesser General Public License (LGPL)
lazr.restfulclient	0.14.4	GNU Library or Lesser General Public License (LGPL)
lazr.uri	1.0.6	GNU Library or Lesser General Public License (LGPL)
psycopg2	2.9.6	GNU Library or Lesser General Public License (LGPL)
wadllib	1.3.6	GNU Library or Lesser General Public License (LGPL)
PyJWT	2.3.0	MIT License
autopep8	2.0.2	MIT License
blinker	1.4	MIT License
dbus-python	1.2.18	MIT License
django-log-viewer	1.1.7	MIT License
django-silk	5.0.3	MIT License
httplib2	0.20.2	MIT License
jeepney	0.7.1	MIT License
more-itertools	8.10.0	MIT License
pycodestyle	2.10.0	MIT License
pyparsing	2.4.7	MIT License
six	1.16.0	MIT License
tomli	2.0.1	MIT License
zipp	1.0.0	MIT License
keyring	23.5.0	MIT License; Python Software Foundation License
distro-info	1.1build1	UNKNOWN
unattended-upgrades	0.1	UNKNOWN

Table 1: All Dependencies

Architecture

Figure 1 above shows a simplified version of our system architecture. We have one server (called a Digital Ocean droplet) only consisting of a database. We're running a Postgres database in a Docker container on the database server. The database stores it's data in a Docker volume, to ensure persist data.

Our main server is the one that users (which are simulated) connect to. It uses 3 Docker containers:

1. Our Django server, which handles the actual MiniTwit server. This container connects to our postgres database on a separate Droplet. Django stores its logs in a Docker volume 'logs', which we can then be used for monitoring logs.
2. Prometheus, which receives metrics from the Django container.
3. Grafana, which contains our dashboard used for minitoring our MiniTwit application. The data Grafana shows is fetched from the Prometheus container.

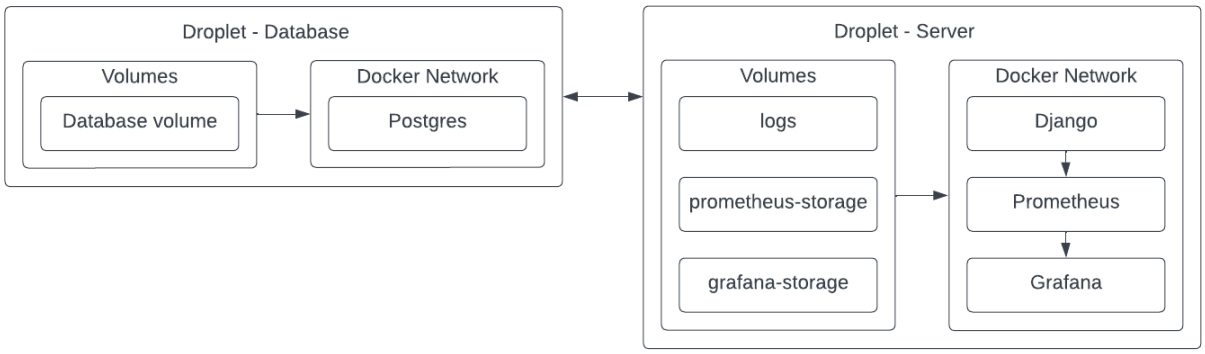

Figure 1: Simplified system architecture

Process' Perspective

Working as a team

To work together as a team we decided on using GitHub. This was due to how we were already familiar with it, and because it allowed us to use its many useful features such as branching for implementing different features, and GitHub actions and workflow files. We created a GitHub organization for the repository so that everyone had the same permissions when it came to accessing and modifying sensitive parts of it (such as GitHub secrets).

Throughout the course we would meet up at least once a week on the day of the lectures. We would often meet some hours before it started to finish up features we were behind on, and leave at the end of the exercises sessions. On some occasions, if we felt we were particularly behind on something, we would meet up more times during the week, often in person but sometimes also online using Discord to communicate and share screens.

Our team was organized in a way to encourage multiple people working on different tasks. Due to the nature of the course being about implementing many different features this was doable, and to make this task-based approach more manageable for ourselves we made extensive use of GitHub's "issues" page. Using the issues page we had 3 main sections for listing different tasks:

1. Todo: new tasks we have been given by the course or tasks we recognized ourselves that needed to be done. If the task was complex we would write descriptions of what should be done, sometimes with links to lecture tutorials or other comments on how to approach the task.
2. In Progress: tasks that one or more people are currently working on. We would often use the "assignee" feature to show who was working on these tasks.
3. Done: Tasks that had been fully completed.

When a task began getting worked on, we would create a new branch specifically for it. This made it easier to avoid merge conflicts when implementing whatever feature or fix it was for, and kept the

main branch "complete" in the sense that there wouldn't be unfinished changes on it during feature development. Using branches also allowed us to use CI/CD chains on pushes to branches that didn't deploy to the website server but made local checks. Furthermore, using branches this way allows us to make pull requests through GitHub, and use the nice UI for solving merge conflicts and reviewing each others pull requests before merging to main.

When working together on the same problem we often used the VS Code "Live Share" extension, which allowed multiple people to work on one persons instance of VS Code. This allowed us to easily do pair programming, or simply help each other with problems.

More on Repository Structure

We used a mono-repository, and apart from the usual *.github* files we have 5 files/folders:

- API_Spec: Simulator files for testing application.
- PenTests: PDF's of security reports.
- src: Minitwit application files.
- tools: Prometheus file and deployment scripts.
- Other loose files such as MegaLinter files.

We have this structure to keep things organized, but we did sacrifice some organization to keep default file-path structures that some tools use. For example there is a loose MegaLinter file in the root folder. There are also some outdated/unused files in the repository such as our Vagrant file which we ended up not using but kept in case we wanted to use it eventually.

System Monitoring

We use Prometheus with Grafana, and Silk to do various monitoring. With Grafana we have a dashboard accessible at port 3000, which displays:

- Total CPU seconds - line graph of total CPU seconds for the past 24 hours.
- Process Virtual Memory - a line graph of how much virtual memory is being used over the past 24 hours.
- Newly inserted models - a line graph showing the total number of follower and message requests for the past 24 hours.
- Total HTTP requests - a bar chart of the total number of HTTP requests over the past 24 hours.

With Silk we monitor how long it takes for different HTTP requests and queries to be met. On our silk dashboard we display four main sections:

- Summary - summary stats such as total number of requests, average time per request, average time spent communicating with the database per request.

- Most Time Overall - Information on the requests that take the most time to complete. For each request displays overall time taken, time taken on queries, and total number of queries.
- Most Time Spent in Database - Similar to above but for requests that take the most time to communicate with the Database.
- Most Database Queries - Similar to above but for requests that have done the most queries.

A screenshot of this page can be seen on Figure 2 in the Appendix.

Logging

Logging was done using Django's log rotation implementation. We log everything that Django logs, which uses Python defined log levels:

- DEBUG: Low level system information for debugging purposes
- INFO: General system information
- WARNING: Information describing a minor problem that has occurred.
- ERROR: Information describing a major problem that has occurred.
- CRITICAL: Information describing a critical problem that has occurred.

These logs get written to a file called *info.log*. If the log file exceeds 5MB then the file is renamed to *info.log.1*, and a new *info.log* file is created for the most recent logs to be written to. When this new *info.log* file is full, *info.log.1* is renamed to *info.log.2*, and *info.log* renamed to *info.log.1* etc. to have these "rolling" logs up until *info.log.5*.

At *info.log.5*, if the logs are full and a new *info.log* file needs to be made, it simply gets deleted. This method is done so that we retain enough logs to be useful in case something goes wrong, but don't end up with our logs infinitely increasing over time and requiring unreasonable amounts of space.

We display the logs on the Django admin page, so that we can access them without SSH'ing into the DigitalOcean droplet.

CI/CD chains

When pushing to a branch we have a GitHub workflow file that makes the following checks:

- Starts localhost version of server.
- Performs simulation test.
- Performs Snyk security evaluation check on docker image. Check fails if Snyk finds any "high severity" issues that are fixable.
- Shuts down localhost version of server.

If any of the checks fail the person who pushed can know that their changes break at least the local version of the server.

On pull-requests multiple quality gates / checks are made using various tools before things are merged to main:

- SonarCloud quality gate - this checks all files for bugs, code smells, vulnerabilities, security hotspots and duplication. We have this configured to the SonarCloud defaults. This is a particularly nice quality gate because on a failure to pass the quality gate we can go to SonarCloud's website to see what problems it found and choose to either fix them or tell SonarCloud to ignore them if we deem them not a problem.
- Snyk quality gate - differently than before, this time Snyk is used to check all *requirements.txt* files. This gate fails only when the PR is adding a dependency with issues.
- MegaLinter - Used once to initially format code to standard styles. Then used to print MegaLinter report for all merges, showing how many "errors" it detects to tell us if we should run it again.

On a successful merge two jobs are performed, one to deploy the new code to the server and another to make an automatic GitHub release. The release job is performed only if the code successfully deploys to the server.

To deploy to the Digital Ocean droplet, the job:

- Logs into our Docker Hub (login details saved in GutHub secrets)
- Builds and pushes our Minitwit docker image
- Configures SSH keys with droplet machine using GitHub secrets for SSH details.
- Removes old deployment tools (deploy.sh, docker-compose.yml, prometheus).
- Adds new deployment tools.
- Deploys new changes by running deploy.sh (which runs our docker-compose.yml).

Security Assessment

Scaling and Load Balancing

Without the docker swarm up and running we do not really have any load balancing implemented. Scaling is not automated, but the droplets can be manually scaled vertically using Digital Oceans UI.

We scaled the main droplet when we noticed the time for requests being particularly slow. We did so as it was the easiest solution at the time and we had enough credits to do so without paying out of our own pockets.

Name	Risk Level	Descriptionn
Content Security Policy (CSP) Header Not Set	Medium	Balls
Application Error Disclosure	Low	Bolas
Cookie No HttpOnly Flag	Low	Cojones
Server Leaks Version Information via "Server" HTTP Response Header Field	Low	Pelotas
X-Content-Type-Options Header Missing	Low	Marbles
User Agent Fuzzer	Informational	Schqweppes

Table 2: L + Ratio + I'm balding + HEEEEEELLP

AI-assistance

To speed up the development process, we used a few AI-tools such as GitHub Copilot and Chat-GPT. They were used not as a way of writing the entire project, but more of a way to implement smaller functionalities and trouble shooting. There is the concern of one not learning the material to the full extend, however the things we used it for were merely things one could Google anyways, so it was simply a way to speed up the troubleshooting and perform some of the easier tasks. We still implemented the vast majority of the infrastructure our self.

Lessons Learned Perspective

Development and Refactoring

During development, we encountered a problem with Vagrant. Specifically, Vagrant was not working on other sub-systems other than Ubuntu. Therefore, due to this limitation, we have decided to manually create the initial setup, which included creating the droplet and installing initial dependencies, such as docker and docker-compose. Even though, the solution worked in this project due to the scale, it would be optimal to use Vagrant to automatically create the initial setup for the project. This is especially true for large scale projects.

Operation and Maintenance

We encountered two problems when implementing logging during the maintenance phase. Firstly, we did not rotate the log files at the beginning, so the file became big enough that our droplet did not have enough space and stopped working.

This can be seen in the simulator plots as we have a drop on the 28th of March. Additionally, this has also resulted in us getting more errors because the users that were supposed to be created when the server was down, were not created, so when the simulator attempts to have them follow or twit more errors occur. The solution was to scale up the size of the droplet and introduce rotating log files with max-size to prevent this from happening.

Secondly, we did not check the log formatting. The initial implementation was done based on gathering every data point we needed, therefore the logs had the following formatting:

{asctime} {levelname} {module} {process:d} #{thread:d} {message}

2023-05-09 12:47:56,545 WARNING log 11 139829591119424 Not Found: /sim/fllws/Florentino

However, when implementing the Django-log-viewer, the requested formatting to display the logs in the proper way is:

[{levelname}] {asctime} {name}: {message}

{[INFO]} 2023-05-16 11:04:51,008 django.server: "GET /public HTTP/1.1" 200 1037

Therefore, since we did not use the correct formatting, when accessing the Django-log-viewer, the logs are in one extensive line, instead of being on separate lines.

Another encountered difficulty is that the number of equal tweets on GUI and API is zero. We concluded that the reason for this was because our GUI took too long to load, and in the time it was loading the simulator already pushed multiple tweets, therefore never achieving the parity. This is an unresolved problem in this project's MiniTwit.

Appendix

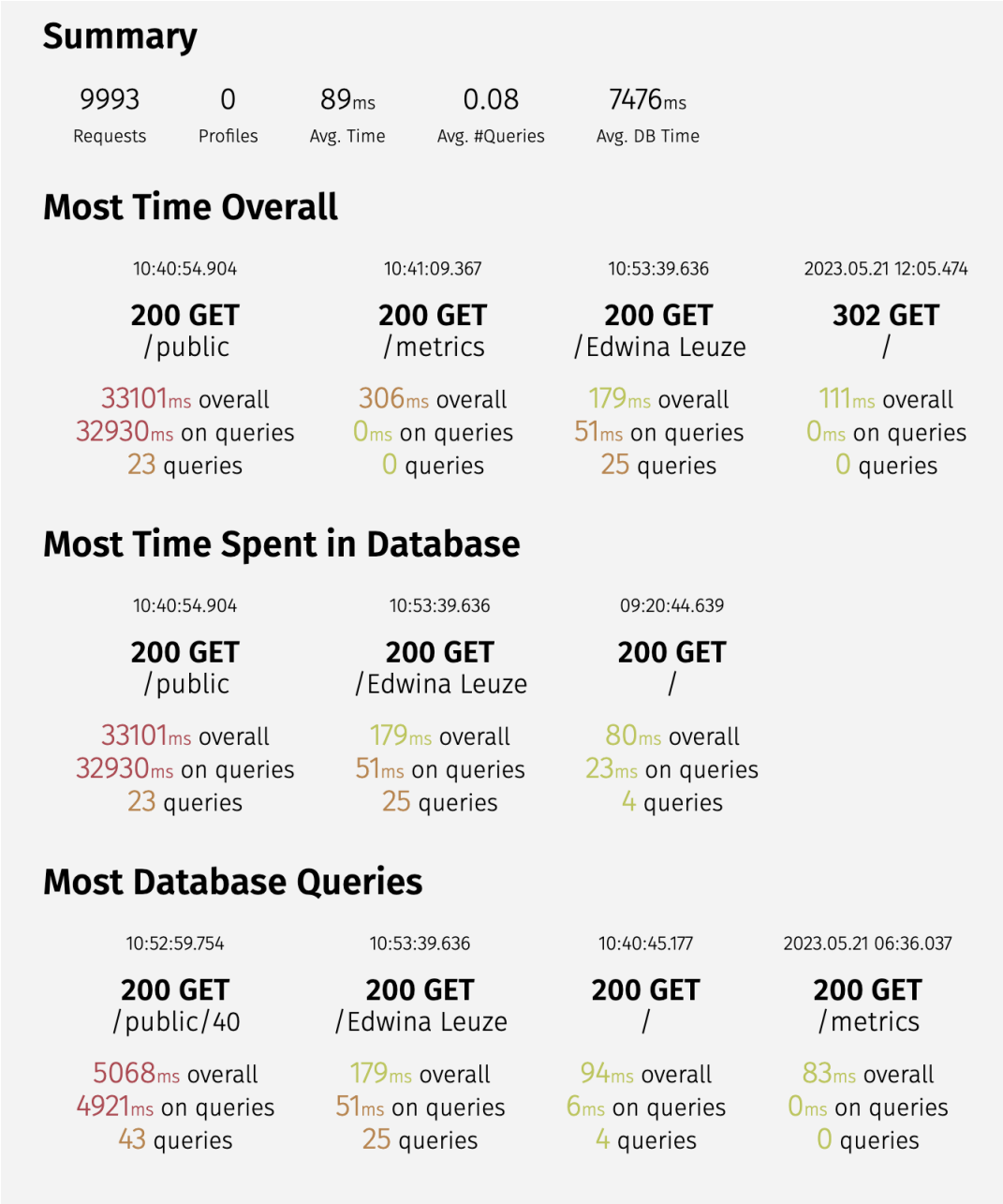

Figure 2: Summary of Query metrics