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.

Containerized

Containerized is a term in software development, which references the fact that you isolate part of your software into containers. To implement this technology we decided to use Docker. Using such technologies is beneficial when writing in teams, since you can keep all the moving parts isolated, thus solving the exponentially difficult problem of dependencies. By forcing everyone to use the dependencies within the container, this enables the container to be run on any computer/server having docker installed.

We setup a total of 4 Docker containers each responsible for one specific task as seen below:

1. Django server - one of the things we are hosting the ip
2. Database - postgres container
3. Prometheus container, collecting data from Django container, how many requests monitoring
4. Grafana visualization of performance i.e. monitoring

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 built in system for testing before deploying. GitHub has a built in CI/CD tool where you can build a pipeline of checks that have to be completed for the branches to be merged.

Cloud

Since we are creating a web app, we need to host it somewhere, since we do not want our development computers to function as the server. By having a centralized structure, that is, a singular place for hosting, we only had to know one provider, and everyone could connect to it from anywhere. We used a Infrastructure-as-a-Service (IaaS) provider called Digital Ocean, due to them having a nice UX-design and allows for easy 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. 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 Grafana for *INSERT* tasks and Silk for *INSERT*.

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 *debt*, which in itself is not bad as long as it is paid back. This term of writing something fast at the expense of quality is called technical debt.

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 many actions. It is implemented directly into our CI/CD pipeline, where it shows bugs amongst other things.
2. Snyk is a tool which shows potential vulnerabilities in the code. This can be anything from creating pull requests on GitHub to update dependencies, to showing where in the code these vulnerabilities lie.
3. Megalinter provides consistency to our code. Since there are bunch of ways of writing the same code that are all syntactically correct, this tool creates consistency throughout our code base.

Logging

We decided to use Django's implementation for logging, as implementation and integration-wise this is robust enough and offers the same functionality as other third-party tools. In addition, we tried to limit the extensive use of third-party tools as these can lead to potential security risks.

Penetration Testing

When it comes to handling something valuable, there will always be people trying to get their hands on it. In our case, it can come in the form of infiltrating our system in order to acquire confidential information regarding our users, or even hijacking a user's account directly.
In order to discover what vulnerabilities are possibly exploitable in our system, a series of Pen Tests (Short for *Penetration Testing*) were executed.

The tool used to conduct this testing was **ZAP** (*Zed Attack Proxy*), which in summary conducts a series of fake attacks and scans the application's infrastructure to detect possible weaknesses and how the system could potentially be breached.

Swarm

F O R T H E S W A R M

Terraform

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 display as it should, the GUI loads too slow and other code problems. Additionally, there are also 54 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. Therefor we tried to implement HTTPS such that data were

Licence

The licence of a software is important due to it dictating how external users can use your software. 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 shows a simplified version of our system architecture. We have one server only consisting of a database. We're running a Postgres database in Docker container on the database server. The database stores it's data in a Docker volume, to ensure persist data.
On our main server, the one the users/simulations connects to, we run a container 3 Docker containers. One is our Django server, which handles the actual MiniTwit server. This container connect to our postgres database on a separate Droplet. Django stores it's logs in a Docker volume 'logs', which we can then use for monitoring logs.
Alongside, we run another Docker container running Prometheus. This receives metrics from the Django container, used for monitoring. On top of that we run a container running Grafana. Grafana contains our dashboard used for monitoring our MiniTwit application. The data Grafana shows is fetched from our Prometheus container.

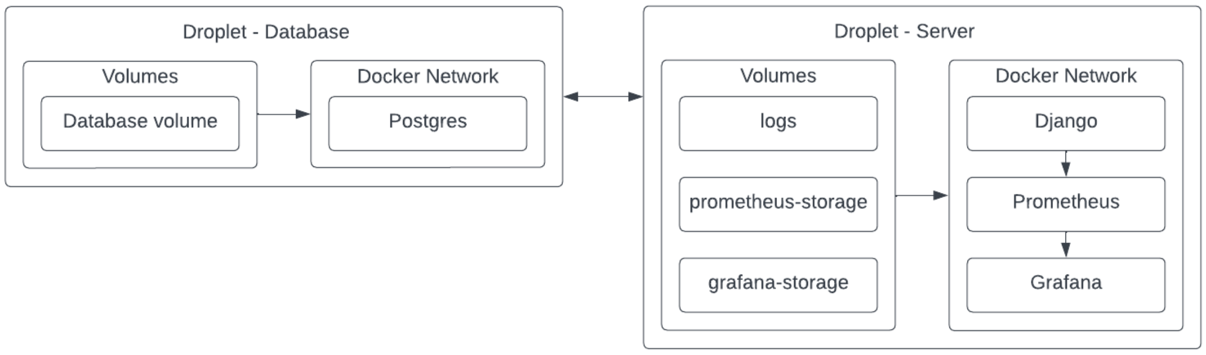

Figure 1: Simplified system architecture

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 we met up more times during the week, often in person but also sometimes 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.

0.1 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.

0.2 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:

- epic stats

0.3 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.

0.4 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 server, the job:

- Logs into our Docker Hub (login details saved in GutHub secrets)
- Builds and pushes our Minitwit docker image
- Configures SSH keys on webserver 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).

0.5 Security Assessment

Impenetrable.

0.6 Scaling and Load Balancing

Scaled far and wide.

0.7 AI-assistance

To speed up the development process, we used a few AI-tools such as GitHub Copilot and Chat-GPT. They where 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

0.8 Evolution and Refactoring

0.9 Operation and Maintenance

We have 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 also 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 and could not follow or twit. 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, now 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. The reason that we concluded was that 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.