DevOps, Software Evolution & Software Maintenance

KSDSESM1KU

Group F

Laura Lunddahl (*laulu@itu.dk*), Louise Kahl Skafte (*lous@itu.dk*), Marcus Grattan Landberg (*magl@itu.dk*), Nanna Marcher (*nanm@itu.dk*) & Vigdís Birna Þorsteinsdóttir (*vigp@itu.dk*)

Spring 2021

Introduction

This report covers the process of designing, developing and maintaining a system throughout the course *DevOps, Software Evolution and Software Maintenance*, while aiming to learn the practices of DevOps.

The report is split into three main sections containing different perspectives: System, describing the design and architecture of Minitwit, Process, including matters related to the development and maintenance of the system and finally Lessons Learned, outlining the main obstacles of the project.

The project can be found on GitHub.

System

Design and Architecture

This section describes the design and architecture of our system through the 3+1 model (Christensen, 2007).

Figure 1 shows the *module view* of our system in a *package diagram*. The Minitwit system is built around two packages: The API, and the Web Application. Each of these two packages includes respectively four packages.

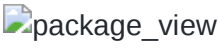

Figure 1: Package Diagram

Figure 2 shows the *component and connector view* of the system. Here it is shown that the system is accessible through the API with a simulator and the browser.

Figure 2: Component and Connector View

Figure 3 shows the *deployment view* of the system including how the system uses Digital Ocean and Docker for deployment.

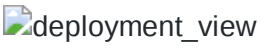

Figure 3: Deployment View

Dependencies

Figure 4 shows the direct dependencies of the system including each tool's [version] and license.

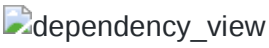

Figure 4: Dependency View

The application is implemented in Go. Go was chosen as it is good for writing web applications and working with API requests. Moreover, Go has better performance than Python which was used in the original version. The most essential tools and technologies that our system relies on are:

- Gorilla/mux: Used for matching incoming HTTP requests and calling a handler for the route that matches the URL.
- Gorilla/sessions: Used for storing sessions in cookies which is essential for the login.
- GORM: Used for Object-Relational Mapping (ORM) for decoupling the application from the DBMS. An ORM was chosen as it allows us to do basic CRUD operations while minimizing repetitive code. Additionally, it adds an abstraction layer which allows us to easily switch to another DBMS.
- Digital Ocean: Provides the load balancer and hosts the servers running the API and the application as well as the MySQL database. Digital Ocean was chosen as it provides a wide range of appropriate tools, it is easy to set up and offers student credits. We chose to migrate the database from SQLite to MySQL as it is easilier scalable and more suitable for multiple user access (Edward S., 2019).

For monitoring the system the following dependencies are used:

- Prometheus: Used for collecting the created metrics.
- Grafana: Used for visualizing metrics collected by Prometheus.

For logging the following tools have been identified as dependencies:

- Filebeat: Used for collecting logs.
- Elasticsearch: Used for storing the logs retrieved from Filebeat.
- Kibana: Used for displaying the logs stored in Elasticsearch.
- Logrus: Structured logger for Go, used for setting the format and writing the logs to Filebeat.

Subsystems

The interactions between the subsystems are shown in a *sequence diagram* in Figure 5. The identified subsystems are: Monitoring, Logging, Database, API, Backend and UI. The diagram considers a specific scenario where the simulator sends a register request. Other scenarios could also have been explored, but since the interactions would be almost identical this scenario will suffice.

Figure 5: Sequence diagram showing interaction of subsystem

License

Before choosing the desired license, the advantages and disadvantages of the different licenses were discussed. From this, we found that the copyleft GPL license fits best with our vision. The GPL license is beneficial for individual contributors and often creates a community among them. By using the GPL license we still allow users to run, modify and share the software for free, but with the restriction that their improved versions must also stay free. This may discourage some larger organizations to use it, but since our software is a social media platform, we wish to create a community where we encourage cooperation and allow users to contribute freely while avoiding larger organizations turning it into proprietary software.

The GPL license is compatible with the licenses of the used dependencies.

Current state

The static analysis tools *go vet*, *golint*, *SonarCloud* and *Codeclimate* are enabled every time a build is executed or a pull request is made. *go vet* and *golint* were chosen since they are some of the most common analysis tools for Go programs. They are part of the GitHub Action workflow, where a list of suggested fixes is displayed (see Appendix A). At the moment this mainly includes inconsistency with code style convention found from the *golint* tool. *SonarCloud* and *Codeclimate* were suggested by the course and are integrated directly in GitHub and executed every time a pull request is made. At the moment the following concerns are detected by the tools:

- Duplicate code
- Too many lines of codes in functions
- Deeply nested control flow statements

There is therefore still a need for refactoring of the code base.

Furthermore, there have been issues with monitoring and logging on the deployed version which have not been fixed. This will be described in more detail in the Process section.

Figure 6 shows a graph of the simulator status for all groups. The purple line shows the requests correctly handled by our system. The reason for the sudden change in the slope is covered in the lessons learned section.

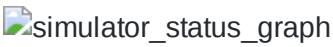

Figure 6: Simulator status graph

Process

Interaction inbetween the team

Due to the ongoing pandemic, our interaction as a team has mainly been online. We have used Zoom for meetings and group work, Facebook Messenger for asynchronous communication and GitHub projects for project overview and task delegation. Project work has been completed every Monday, Wednesday and Saturday while releasing every Sunday. When coding we have mainly used the development approach mob programming (Hammarberg, 2013) where we have worked on a specific task with one person coding and the rest of the group aiding/supervising. We have either been the entire group or divided into sub-teams depending on the size and complexity of the given task. This approach was chosen as we were unfamiliar with the subject of DevOps. We therefore wanted to be equally involved in all aspects in order to get a good understanding of the subject.

Organization of the team and repository

We chose the Git Feature Branch Workflow which is an extension of Centralized Workflow (Atlassian BitBucket, n.d.). In addition to this, we have used the mono repository model where we have a single unified source-code repository that is accessible to all team members.

The Private Small Team collaboration setup was used for contributing to the project. The repository is public but it is only the group members who have permission to contribute to the project.

We established the following commit guidelines at the beggining of the project:

- Commit regularly
- Limit subject to 50 characters
- Summarize in bullet points what has been done
- If it is a smaller fix/change e.g. fixing a typo then a single line is fine

The commit structure has not been upheld, which will make it harder to go back in older commit messages and detect exactly what changes have been made. This is not optimal and has probably been caused by the use of mob programming, which often meant that the whole group was aware of the changes included in the commits.

We used the *merging topic branches staged* as the integration/merging workflow. In order to merge a `topic` / `feature` branch into `dev`, the developer had to create a pull request. This pull request needed to be reviewed by another developer who was also in charge of accepting the pull request.

Branching strategy

We decided to use the Git-flow branching model, where we have a `master` branch for storing versions and a branch `dev` for development. When working on a specific task we create a new feature branch from `dev`. When finished, the branch is merged into `dev` and deleted. Once we have a stable version we wish to release we merge from `dev` to `master`. Figure 7 shows an extract of our network from GitHub.

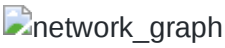

Figure 7: Extract of Network Graph from GitHub

The figure shows the `master` branch (black) and `dev` (blue) as well as three feature branches. To prevent major merge conflicts, we have sometimes merged a feature branch into another feature branch which deviates a bit from the Git-flow model. It has also happened that some changes have been made directly on `dev`.

Continous integration and continous deployment

It was chosen to use a continuous integration (CI) system where we can run automated builds. To make sure that a new feature works as intended, the pipeline has been set up to run on all pull requests to `master` and pushes on `master`. It was also chosen that we wanted to use the same tool for CI as for continuous deployment (CD).

At first, we chose to use Travis CI as our CI/CD tool, as it is easy to integrate with GitHub, and because it offers a lot of automated features. Lastly, it is also a cloud-based solution which means that we do not need to run and maintain a server for it ourselves.

As Travis CI did not provide infinite run credits we chose to change our CI/CD tool to GitHub Actions (see issue 89). GitHub Actions provides a large catalog of available to use actions which makes the process of configuring it much easier. Another advantage of GitHub Actions is that we now have everything in one place.

Our CI/CD chain consists of two jobs: build and deploy. Each of these jobs consists of multiple steps that use different action features. The CI/CD chain does not include continuous releases.

Applied development process

As our repository is located in GitHub we also chose to make use of the GitHub project board to keep track of outstanding tasks. Here we have created a Kanban board that consists of three columns *To do*, *Doing* and *Done*. When encountering new tasks they were immediately added and assigned to one or more team members. This ensured that we maintained an overview of who was responsible for each task as well as whether they were in progress or completed. For some issues a checklist was added in case it consisted of multiple subtasks. Figure 8 shows an extract of the Kanban board.

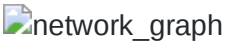

Figure 8: Extract of the Kanban board

Monitoring

For monitoring the system, the group has used Grafana and Prometheus. Prometheus is a storage backend that collects metrics from monitored targets, while Grafana is a visualization layer for the monitored data. Below we have listed the metrics that have been monitored for the application. These can be grouped into two categories, technical and business. The technical metrics are related to the performance and hardware information of the system, while the business category consists of statistical information related to the usage of the system. Figure 9 shows the Grafana dashboard.

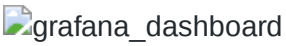

Figure 9: Screenshot from our Grafana dashboard

- Technical:
  - CPU load
  - Response time for a user to register
  - Response time for a user to send a message
  - Response time for a user to retrieve messages
  - Number of HTTP request
- Business:
  - Amount of users registered in your system
  - Average amount of followers a user has
  - Number of times a user unfollows another user
  - Number of times a user follows another user
  - Average amount of posts per user

Logging

For the aggregation of logging in our system, the Elasticsearch-Filebeat-Kibana (EFK) stack is used. The EFK stack was chosen since it is open source and was recommended in the course. The processing of logs is shown in figure 10. We chose to send the data directly from Filebeat to Elasticsearch (removing Logstash from the stack) since it requires less parsing and transformation power.

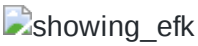

Figure 10: Processing of logs with EFK

Currently we are only logging errors, i.e. when an error happens with a database query or when an error occurs in the API. The plan was to extend this to also log successes.

At the moment, the logging does not work on the deployed version. We have been experiencing the error *"Failed to poll for work: [cluster_block_exception] index [.kibana_task_manager] blocked by: [FORBIDDEN/12/index read-only / allow delete (api)]; :: {\"path\"* and have not been able to find a proper solution to fix the error, therefore it only works locally. A screenshot of the Kibana dashboard is shown in figure 11.

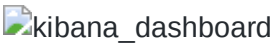

Figure 11: Kibana dashboard

Security assessment

When identifying the risks that the system is vulnerable to we started by identifying assets and threat sources and use that information to construct risk scenarios. The assets in the system that are possible targets for an attacker are most of the parts of the system, for example:

- The web application
- API
- Database
- Github
- Added tools such as Kibana, Prometheus, Grafana

Possible threat sources that could be used to attack the application are:

- SQL injection
- Cross-site scripting
- DDOS Attack
- Bad Authentication

The scenarios that that were constructed and could be done by an attacker are following:

1. Cross site scripting (XXS) through the form when a message is being posted.
2. SQL injection when registering a new user or logging into the application.
3. Cracking the password to our droplet and ssh to the machine and delete everything or install malicious software.
4. Perform a DoS (Denial of service) attack on our web application, making it impossible for users to view the site.
5. Cracking the password to our database and delete or access information. Since our GitHub repository is public and includes all other information about the database it would only require bruteforcing the password.

When analysing the risks each scenario was given a grade for likelihood and impact. Figure 11 shows a risk matrix that was used to prioritise the risk of scenarios. The detailed ranking can be seen in the risk_analysis.md file but below the scenarios have been listed in a descending order based on the ratings they got along with the actions that should be taken to prevent them.

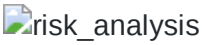

Figure 11: Risk analysis matrix

- Scenario 2: The Go package we are currently using in our program already sanitizes the input that is being used to query the database so as it is now, the input is already parameterized which is a good defence against SQL injections.
- Scenario 5: Change the password of the database so it is long and complicated making it difficult to brute force. Check if it is possible to add two-factor authentication.
- Scenario 3: Change the password to the droplet so it is long and complicated, making it difficult to brute force.
- Scenario 4: By adding some scaling to our setup, if the main system is down due to a DoS attack we will be able to start the backup system so we have the application up and running. Other than that, there is not much you can do when it comes to this type of attack.
- Scenario 1: Go has some packages that can be used to defend against XXS attacks. We will look into adding the functionality provided by those packages to prevent this type of attack.

Scaling and load balancing

The system has been scaled vertically (up) and thereafter horizontally (out). The vertical scaling was applied since the implemented logging tools required more CPU than what the machine was originally created with. A need for applying horizontal scaling was however still necessary as the service that the system provides requires a system that is always available. It also removes the single point of failure that only having a single machine presents. This is necessary as it is a social media system that needs to be running at all times in contrary to e.g. an internal system that can be taken down during the night for maintenance.

Load balancing has been set up through Digital Ocean. This load balancer was chosen since the virtual machines are also hosted at Digital Ocean. This made the load balancing easy to set up. The load balancer is dependent on a health check that relies on a certain status code response from the system. If the status code does not match what the load balancer is expecting it will declare the system as down. Digital Ocean does not provide us with the possibility to change this expected status code, and as the status code we must respond with to the simulator does not match it, the system is constantly down.

Lessons Learned

As most of the introduced subjects and tools were new to us, most parts have been quite challenging. Nevertheless, some challenges were of greater significance.

Read Timeout

The biggest and most time-consuming challenge has been that our system was too slow at responding to the simulator’s requests which resulted in numerous ReadTimeout errors (see issue 44).

Attempting to solve the issue we tried multiple things including separating the Web Application from the API, switching from MySQL to Postgres as well as switching where the database was hosted. At this point, we were just trying to implement possible fixes without actually knowing the source of the problem.

After having narrowed it down to being slow queries from the database, we wished to profile the network (see issue 73). This was not possible as the database was hosted on ITU's servers which we did not have the permissions to profile. Later, when implementing the load balancer in Digital Ocean we realized that the system had been stored on a server in the default location, which is New York, while the database was located in Copenhagen and the simulator in Frankfurt. Moving the server to Frankfurt and migrating the database to a MySQL database hosted on DigitalOcean solved the issue.

Separating API from Web Application

As mentioned above, in an attempt to solve the ReadTimeout, we moved the API and the Web Application into two separate docker images (see issue 82). As this was done late in the project, the general structure was not designed for this. This resulted in a large amount of duplicate code which could have been prevented if we had spent more time on choosing a proper architecture at the beginning of the project.

CI/CD setup

We found it difficult to set up the CI/CD pipeline. This was mainly due to not having done enough research prior to this, especially as setting it up also required understanding the relation between Docker, Digital Ocean and Vagrant. Once the CI/CD worked as intended the workload of deploying manually was removed.

Logging and monitoring

It would have been beneficial to prioritize setting up the logging and monitoring earlier in the project. It took us a long time to implement them correctly. In the meantime, we had issues that a functional logging system could have helped us detect and possibly fix.

Load balancer

After having set up the load balancer we learned that it is valuable to have two running droplets. This makes it possible to do maintenance on one while still having the other droplet running, so it does not affect the availability of the system.

Reflection on DevOps style

One of the approaches from the DevOps Handbook (Kim et al, 2016) that we have used in our project is having an emphasis on limiting the amount of tasks in progress. We have tried to focus on helping each other completing ongoing tasks before starting a new one. Furthermore, we tried to reduce batch sizes by continuously deploying whenever a task was completed. This was more straightforward once the CI/CD pipeline worked. Another thing also mentioned earlier is the use of a Kanban board, which has helped make our work visible. Further aspects are covered in Week5.md.

The main difference from previous projects has been the focus on the health/uptime/state of our system rather than solely on the functionality of the system.

Conclusion

When working on the application, it has become apparent how important it is to work in a structured way when building software and to organize when multiple developers are working together. We found that it is important to consider the design of the system and how different objects depend on each other as a loosely coupled system will be easier to maintain and modify. Lastly, we experienced the value of automating the process as much as possible and how it will relieve the workload in the long run.

References

Atlassian BitBucket (n.d.). Git Feature Branch Workflow [Accesed on 2021-05-17].

Christensen (2007). An Approach to Software Architecture Description Using UML (Revision 2.0).

Edward S. (2019). sqlite vs mysql whats the difference [Accessed 2021-02-27].

Hammarberg M. (2013). Mob-programming [Accessed on 2021-05-12].

Kim et al (2016). DevOps Handbook part 1 (First Edition). IT Revolution Press.

Appendix

A - Static analysis output from API

Output from running go vet and golint
db/dbhandler.go:9:2: a blank import should be only in a main or test package, or have a comment justifying it
db/dbhandler.go:14:5: exported var DB should have comment or be unexported
db/dbhandler.go:32:1: exported function GetDB should have comment or be unexported
dto/follower.go:8:6: exported type Follower should have comment or be unexported
dto/follower.go:9:2: struct field WhoId should be WhoID
dto/follower.go:10:2: struct field WhomId should be WhomID
dto/follower.go:13:1: exported function IsFollowing should have comment or be unexported
dto/follower.go:13:18: func parameter whoId should be whoID
dto/follower.go:13:29: func parameter whomId should be whomID
dto/follower.go:18:1: exported function FollowUser should have comment or be unexported
dto/follower.go:18:17: func parameter whoId should be whoID
dto/follower.go:18:28: func parameter whomId should be whomID
dto/follower.go:27:18: func parameter whoId should be whoID
dto/follower.go:27:29: func parameter whomId should be whomID

Output from running go vet and golint
dto/follower.go:37:1: exported function GetFollowers should have comment or be unexported
dto/follower.go:37:19: func parameter whoId should be whoID
dto/follower.go:47:1: exported function UnfollowUser should have comment or be unexported
dto/follower.go:47:19: func parameter whoId should be whoID
dto/follower.go:47:30: func parameter whomId should be whomID
dto/follower.go:56:1: exported function GetTotalNumberOfFollowerEntries should have comment or be unexported
dto/message.go:10:6: exported type Message should have comment or be unexported
dto/message.go:11:2: struct field MessageId should be MessageID
dto/message.go:12:2: struct field AuthorId should be AuthorID
dto/message.go:18:1: exported function AddMessage should have comment or be unexported
dto/message.go:18:17: func parameter authorId should be authorID
dto/message.go:27:1: exported function GetTotalNumberOfMessages should have comment or be unexported
dto/timeline.go:11:6: exported type Timeline should have comment or be unexported
dto/timeline.go:13:2: struct field UserId should be UserID
dto/timeline.go:18:2: struct field MessageId should be MessageID
dto/timeline.go:19:2: struct field AuthorId should be AuthorID
dto/timeline.go:28:1: exported function GetPrivateTimeline should have comment or be unexported
dto/timeline.go:28:25: func parameter userId should be userID
dto/timeline.go:38:1: exported function GetPublicTimeline should have comment or be unexported
dto/timeline.go:48:1: exported function GetUserTimeline should have comment or be unexported
dto/timeline.go:48:22: func parameter profileUserId should be profileUserID
dto/user.go:8:6: exported type User should have comment or be unexported
dto/user.go:9:2: struct field UserId should be UserID
dto/user.go:16:1: exported function GetUserID should have comment or be unexported
dto/user.go:27:1: exported function GetUser should have comment or be unexported
dto/user.go:37:1: exported function GetUserById should have comment or be unexported
dto/user.go:37:6: func GetUserById should be GetUserByID
dto/user.go:37:18: func parameter userId should be userID

Output from running go vet and golint
dto/user.go:47:1: exported function GetUsername should have comment or be unexported
dto/user.go:47:18: func parameter userId should be userID
dto/user.go:58:1: exported function RegisterUser should have comment or be unexported
dto/user.go:67:1: exported function GetTotalNumberOfUsers should have comment or be unexported
handler/minitwit_sim_api.go:30:1: exported function GetLatest should have comment or be unexported
handler/minitwit_sim_api.go:38:1: exported function RegisterUser should have comment or be unexported
handler/minitwit_sim_api.go:80:1: exported function Messages should have comment or be unexported
handler/minitwit_sim_api.go:100:1: exported function MessagesPerUser should have comment or be unexported
handler/minitwit_sim_api.go:114:3: var userId should be userID
handler/minitwit_sim_api.go:134:3: var userId should be userID
handler/minitwit_sim_api.go:149:1: exported function Follow should have comment or be unexported
handler/minitwit_sim_api.go:161:2: var userId should be userID
handler/minitwit_sim_api.go:178:4: var followsUserId should be followsUserID
handler/minitwit_sim_api.go:197:4: var unfollowsUserId should be unfollowsUserID
handler/structs.go:5:6: exported type Timeline should have comment or be unexported
handler/structs.go:7:2: struct field UserId should be UserID
handler/structs.go:12:2: struct field MessageId should be MessageID
handler/structs.go:13:2: struct field AuthorId should be AuthorID
handler/structs.go:19:6: exported type Message should have comment or be unexported
handler/structs.go:20:2: struct field MessageId should be MessageID
handler/structs.go:21:2: struct field AuthorId should be AuthorID
handler/structs.go:27:6: exported type Latest should have comment or be unexported
handler/structs.go:31:6: exported type Response should have comment or be unexported
handler/structs.go:36:6: exported type ApiMessage should have comment or be unexported
handler/structs.go:36:6: type ApiMessage should be APIMessage
handler/structs.go:42:6: exported type Register should have comment or be unexported
handler/structs.go:48:6: exported type Followers should have comment or be unexported
handler/structs.go:51:6: exported type FollowUser should have comment or be unexported

Output from running go vet and golint
handler/structs.go:56:6: exported type User should have comment or be unexported
handler/structs.go:57:2: struct field UserId should be UserID
helper/helper.go:10:1: exported function GravatarUrl should have comment or be unexported
helper/helper.go:10:6: func GravatarUrl should be GravatarURL
helper/helper.go:18:1: exported function CheckErr should have comment or be unexported
logging/logger.go:7:1: exported function Logging should have comment or be unexported
metrics/metrics.go:68:1: exported function RecordMetrics should have comment or be unexported
metrics/metrics.go:123:1: exported function IncrementFollows should have comment or be unexported
metrics/metrics.go:127:1: exported function IncrementUnfollows should have comment or be unexported
metrics/metrics.go:131:1: exported function IncrementRequests should have comment or be unexported
metrics/metrics.go:135:1: exported function ObserveResponseTime should have comment or be unexported