DevOps projectreport

Joachim Køcher Kelsen (jokk@itu.dk) Victor Nordestgaard (vino@itu.dk)
Isabella Drest Rasmussen (iras@itu.dk) Anne Siemkowicz (asie@itu.dk)
Bjørnar Haugstad Jåtten(bjja@itu.dk)

19 May 2021

Contents

1 Introduction

This project report is written for the DevOps, Software Evolution and Software Evolution spring elective course in 2021. The

2 Lessons Learned Perspective

2.1 Biggest issues

2.1.1 Evolution and refactoring

The main issue we had with the refactoring of the old code was the lack of documentation. A large part of the group had no proficiency with neither Python nor Flask, and we were therefore forced to invest much time into understanding the system. One of the takeaways from this was how difficult it could be to understand and further develop old code when there is a lack of documentation – which is something that often can happen in the real world.

To challenge us as a DevOps team, we decided to also evolve the old software further, by adding new features. Examples of this are the new design and the chat-functionality. Although these functionalities were not the focus, it was a useful learning experience and training in continuous deployment and releasing small features often.

2.1.2 Maintenance and operation

Regarding maintenance and operation, one of the largest issues was that we often had some periods with long downtime. For the sake of simplicity, we have narrowed it down to four different major down periods in this report. These are marked with blue (from now on referred to as first), dark blue (from now on referred to as second), yellow (from now on referred to as third), and green (from now on referred to as the last) in figure 1.

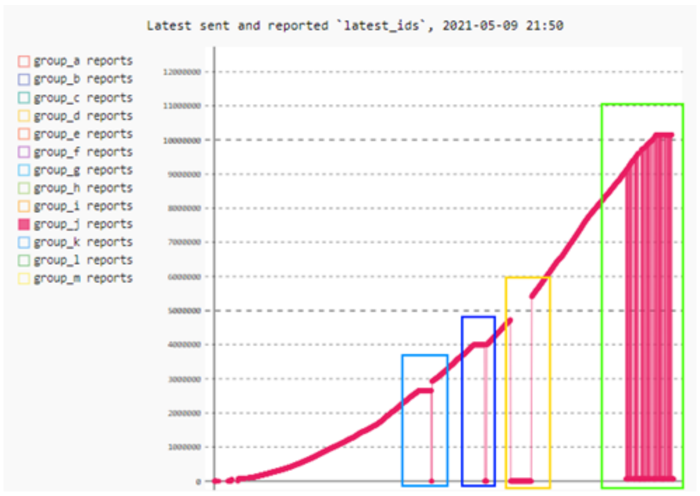


Figure 1: The graph showing "latest", an integer which is updated for each completed job sent from the simulator

All these different down periods had different solutions, and we have decided to discuss the two down periods we learned the most from, the first down period and the second one. The first down period was due to a crash with our database software (PostgreSQL), and in the second down period, it was due to a crash simulator-side. Although logging helped us out in the last down period, we realized too late that the system was down, resulting in a longer down period. The major lesson learned from the two down periods was that we should have implemented some sort of alerting that would sent to all the team members in case of a failure. This was something we started to implement in Kibana, however, due to lack of time in the end of the simulation, we did not have time to implement it.

2.1.3 DevOps way of working - CONSIDER REWRITING

Throughout the project, we as a group have kept the "three ways" characterizing DevOps in mind; flow, feedback, and continual learning and experimentation !!!!CITE devopshandbook!!!!. Looking at these principles, one of the main differences from other group work is centered around the feedback principle. In earlier projects, we developed tests as a part of the development, but they were rarely executed. This resulted in hours of development, where the newly developed code was not tested, leaving us to scrap non-working code and rewrite it. During this project, after the setup of our pipelines, we have been able to continuously test our project. We have also used tools such as Sonarcloud and CodeQL, which has led us to not only develop better code, but also ensure that the code had enough test-coverage and no security risks before it was deployed. This is where the DevOps way of working impacted our work the most, as it was a huge improvement from how we formerly have been thinking about testing.

3 Process' perspective

3.1 Interactions as developers

Due to the current Covid situation, the team has been forced to meet and work online. The team typically meets on Discord in the DevOps exercise sessions, during the week, and in the weekends. Often, collaboration is done through Visual Studio Code's Live Share (which enables group/pair programming) or working through the tasks/exercises individually, helping each other along the way.

3.2 Team organization

The team strives to work in an agile manner, proceeding forward step by step, and adjusting work flows to meet ongoing challenges. Mainly, the group will work together as one big group. However, occasionally, the group splits into two to increase effectiveness, whereby each group focuses on a selected topic. Both groups will then reconvene to demonstrate their findings. Remaining work is distributed to the individual group members, assisting each other in the process.

3.3 A complete description of stages and tools included in the CI/CD chains

The tools we used for CI/CD chains were Travis and GitHub Actions. We started working with Travis, which worked very well for us, unfortunately we experienced some problems with expenses in Travis. We decided to migrate to Github Actions, due to this. We migrated to GitHub Actions, and this works just as well as Travis for us. GitHub Actions is an easy tool to use directly from GitHub. Our stages include build, deploy and test. When we deploy we have the following stages, *building and testing*, *CodeQL* that autobuild attempts to build any compiled languages, *Sonnarscanner* that runs project analysis, *DockerBuild* which build the project, then *deploying* and finally *releasing* it all.

3.4 Organization of our repository

For our project we have chosen to use the structure of mono-repository. The reason for choosing this structure, was that during this project, we were only building one system. Therefor we thought it would be best to keep everything in the same repository. This repository goes by the name PythonKindergarten.

3.5 Applied branching strategy

For development purposes, the team uses Git for Distributed Version Control and have adopted a long-running branches approach to branching. In this context, there are two main branches, master and develop (i.e. we have called it 'development' branch) as well as many short-lived topic branches, which are used for implementing features. This approach to branching works well in a centralized workflow such as ours, where the project is collaborated on in a shared repository (i.e. Github).

3.6 Applied development process and tools supporting it

The Projects Board on Github were used to categorize open tasks and organize our development work. Specifically, the team would create an issue with a description added to the comments section along with attached labels, such as 'need to have', 'group discussion', etc. These same issues were then assigned to different group members and became the tasks that needed attention on the Projects Board.

3.7 System monitoring

3.7.1 How are the system monitored (Change section title)

Monitoring of our system is done using Prometheus and Grafana, deployed using docker-compose. Prometheus acts as a data collector, periodically scraping metrics from configured targets and storing the collected data in the built-in local time series database.
Our system consists of two docker nodes, a master and a worker, which are both configured as targets in our Prometheus deployment.
By configuring our Prometheus server as a datasource within Grafana, the stored time series data can be visualized in preconfigured dashboards. Our metrics are visualized in two separate dashboards, accessible through the Grafana web interface.

3.7.2 What are the monitored metrics (Change section title)

To expose metrics in our system we use a NuGet package called prometheus-net CITE prometheusnet!.
This package allows us to expose metrics on the /metrics endpoint, which can then be scraped and stored by Prometheus.
To extend the default metrics provided by prometheus-net, we use two additional packages: prometheus-net.DotNetMetrics !!!!CITE prometheusdotnetmetrics!!!! and prometheus-net.AspNet !!!CITE prometheusasp-net!!!!.
The DotNetMetrics package provides us with general dotnet metrics, such as GC, byte allocation, lock contention and exceptions.
The AspNet package provides us with ASP.NET specific metrics, such as requests/sec, request duration and error rates.

Snapshots of our two dashboards are publicly available on the following links:
Dotnet metrics: https://tinyurl.com/pythonkindergarten-dotnet
ASP.NET (api-specific) metrics: https://tinyurl.com/pythonkindergarten-aspnet

3.8 What do you log in your systems and how do you aggregate logs?

In our system we are logging all the dependencies that we are using, a few of these are Docker, Grafana and SonarCloud.

We are also logging our simulation errors, and have divided them into errors regarding, follow, tweet, unfollow, connectionError, readTimeout and Register.

To aggregate these logs, we are using ElasticSearch and Kibana. We use ElasticSearch to store our logs in dedicated log indexes, and Kibana is used as a visualization tool for these logs in ElasticSearch. This makes it easy to keep track of our system in and quickly discover if there is anything wrong. (See apendix $_)$

3.9 Brief results of the security assessment

The identified sources are our web services, for logging, monitoring and our MiniTwit application. The servers we use to host are also listed, as well as Docker and Nginx. The threat sources are XCSS, our firewall (UFW), Docker ignoring UFW and a DDoS attack. The risk analysis consists of an exposed database connection string, which have been fixed, by storing the connection string as a Docker secret. Our private keys are stored locally on developer machines, but could by mistake be uploaded to GitHub and that would have a catastrophic impact. Because one would gain Admin rights to our servers. Dependency vulnerabilities are also possible, since we don't check versions of our dependencies. Where checking versions in eg. our pipeline, would make it easier to maintain dependencies. In regards to malicious users gaining access to user data. The impact would be minimized if we kept updates of our database. However we never got to do that.

3.10 System Scaling and load-balancing

3.10.1 Scaling

We have applied both scaling (using docker swarm) and load-balancing (using Digital Ocean's Load Balancer) in our system. Our Docker swarm consists of two servers, a master node and a dedicated worker node, each running multiple service replicas. These two server are internally load-balancing using the Docker mesh network and only one node has to be known, in order to communicate with the swarm.

3.10.2 Load-Balancing

There might be situations where every known swarm node is down, in which case the system as a whole might be unreachable, even though there might still be running nodes in the swarm.
To mitigate this, we use a Digital Ocean Load Balancer. This acts as a gateway (single entry-point) to our Docker swarm, balancing the load between each of our servers and executing health checks, which ensures that the client is always connected to a reachable swarm node.
As an added bonus, the load balancer masks the ips of our servers from clients, making our system less prone to hackers.

Using these strategies we are able to scale far beyond our current setup, simply by joining more servers to our swarm and configuring these in the DO load balancer. The only caveat is that our database is currently running on a single server, but could be migrated to a database cluster on Digital Ocean, AWS, Google Cloud or similar cloud providers.

3.11 From idea to production

Ideas starts of as issues or ideas at group meetings. Then a developer is assigned and checks out from the development branch,and thus creates a topic branch for the feature. When the feature is finished being implemented, two other developers are assigned to look through the code and when approved, it is merged into development. When other features, which form a bigger feature, are all finished and merged into development. Then the development branch is merged into the master branch. Our CI and CD pipelines are then run, and will fail If the project cannot build nor the test suite being run. It the pipelines succeed, the code will be pulled from the main server in our docker swarm. Then the main server, will notify its' worker servers, to update their containers of the running application, with the new docker image. The docker image is pulled from dockerhub.

4 System's Perspective

4.1 System Design

The developed MiniTwit system is complex and consists of many subsystems, each of which required a significant amount of decisions to construct. In this subsection we will describe the design of our system as a whole, including the rationale behind some of the decisions we have made during this project.
For the sake of clarity, we will dissect our system into different layers: frontend, API and data persistence.

1. Frontend
   The frontend of our system is what provides users access to our system through a web interface, which is developed using Blazor (wasm) and C#, as well as html and css. With blazor we are able to have our C# code compiled to webassembly and executed natively in the client browser. The frontend communicates with our api using HTTP requests, containing json payloads of serialized or deserialized data. ....

2. API
   The REST API is developed using ASP.NET and exposes endpoints for communicating with the MiniTwit service. This includes execution of actions like following and unfollowing, as well as data fetching like feeds or user profiles. The API translates frontend HTTP calls into data manipulations using Entity Framework for ORM with a code-first approach. This allows us to focus on code first and automatically construct relational database schemas based on the relations between objects in our C# code, managed by database migrations. ....

3. Data Persistence
   For data persistence we use use multiple difference database management systems. MiniTwit data such as users, followers and messages are stored in a postgres relational database, which integrates well with Entity Framework. Logging data are stored in an ElasticSearch database configured with lifetime policies to keep the size under control. Lastly we use RabbitMQ for temporarily storing user private messages in message queues, interfaced by MQTT over WS, to allow direct communication from our blazor frontend. ....

4.2 System Architecture

Our system as a whole consists of 3 servers and a load balancer, all of which are hosted on digital ocean. Each server is running an NGINX server, which has been configured to serve as a reverse proxy for routing of subdomain access, e.g. master.pythonkindergarten.tech, slave.pythonkindergarten.tech, monitor.pythonkindergarten.tech etc.
Below is an overview of the 3 servers, as well as their function and purpose within our system.

1. Master Server
   The Master server is the master node in our Docker swarm and is responsible for maintaining cluster state, as well as managing service updates within the swarm. This server also hosts our Prometheus and Grafana containers, which were described in the monitoring section.

2. Worker Server
   The Worker server simply contributes to the pooled resources within our swarm. It is a Docker swarm worker node, which runs multiple replicas of our service, hosting both API and frontend.

3. Database Server
   The Database server has multiple purposes, all of which relates to data persistence. This server hosts our Postgres database, which is our primary means of persistence within the system.
   It also hosts our logging system, which consists of an ElasticSearch database (for log persistence), as well as Kibana for visualization of the collected logs.
   Lastly, it hosts our RabbitMQ broker.

4. Load Balancer
   The load balancer is hosted on Digital Ocean and allows us to balance incoming traffic on pythonkindergarten.tech between each of our docker swarm servers.

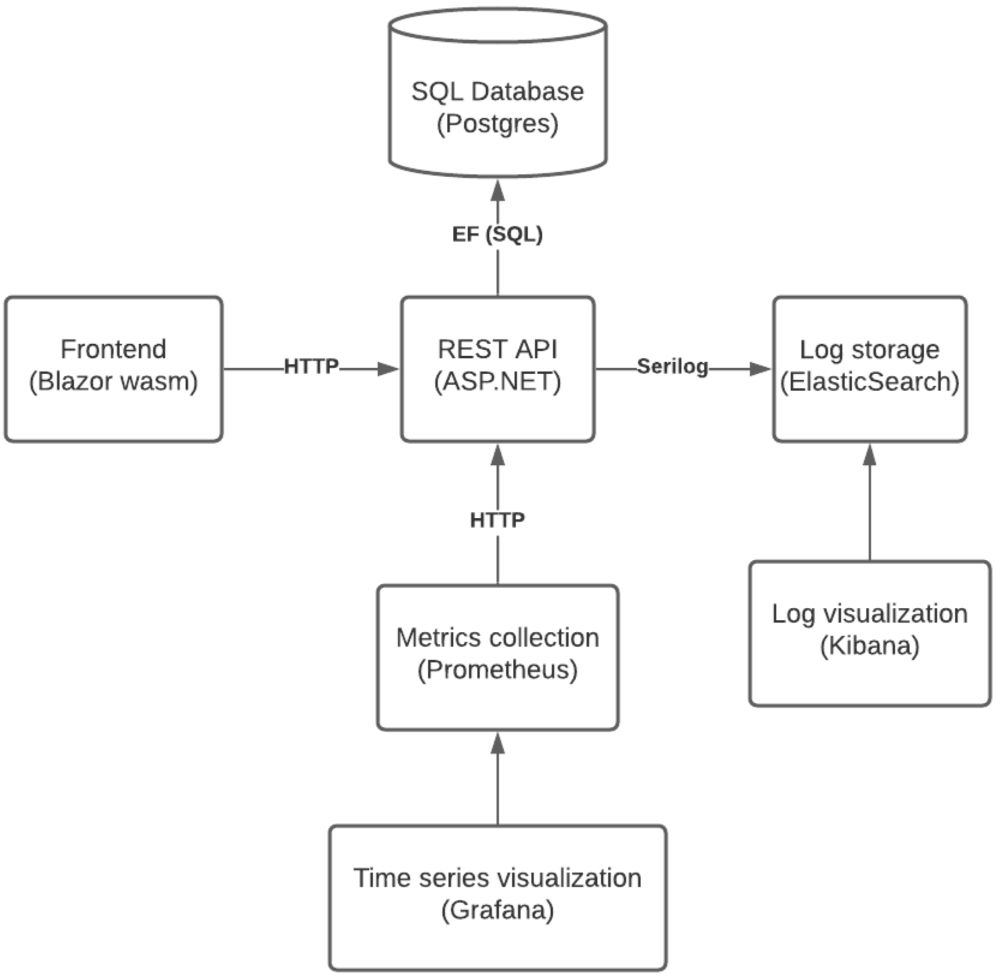
4.2.1 System Deployment

Deploment of our system as a whole has been partially automated using 'Insert Technology', such that a simple shell script will deploy most of our system architecture. The only parts not included in this automated system deployment are: something, something, something and something, due to the high setup/configuration complexity.

All dependencies of your ITU-MiniTwit systems on all levels of abstraction and development stages. (That is, list and briefly describe all technologies and tools you applied and depend on.)

4.3 Subsystem interactions

There are many important subsystems (described in previous sections), collectively making the system work as a whole. In this section we will illustrate the most important interactions between these services, as seen in the diagram below.


SQL Database (Postgres)
EF (SQL)
Frontend (Blazor wasm)
HTTP
REST API (ASP.NET)
Serilog
Log storage (ElasticSearch)
HTTP
Metrics collection (Prometheus)
Log visualization (Kibana)
Time series visualization (Grafana)

4.4 Dependencies

Below is a list of all the **dependencies** of the Pythonkindergarten MiniTwit project, classified by description:

- **Language, Runtimes, Templates and Dependency Manager:** .Net 5.0, ASP.Net Runtime 5.0, Blazor WebAssembly, WASM, NuGet
- **Project Structure:** Server, Client, Shared, Tests

- **Database:** Postgres, EntityFramework Core, EntityFramework NPSQL
- **Testing:** Xunit
- **Repository and Distributed Version Control:** Github, Git
- **Domain names and Domain service:** Pythonkindergarten.tech, .Tech Domains
- **DataTransferring over HTTP** JSON, System.Text.Json
- **Containerization:** Docker, Docker-Compose, DockerHub
- **Pipelines (CI/CD):** Github Actions (N.B. was initially Travis until migration to Github Actions)
- **Deployment Provider:** DigitalOcean
- **Server Certificate for application:** ZeroSSL
- **Monitoring Tools:** Prometheus, Grafana
- **Virtualization:** Vagrant
- **Logging:** Elasticsearch, Serilog, Kibana
- **Static Codecheck Tools:** SonarScanner, SonarCloud, CodeQL
- **Messaging:** RabbitMQ

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

We use two different code analysis tools. They are both run in our CI pipeline. The first one is SonarCloud. The C# Webassembly project and the API project are both analysed. The current state is that our test coverage is 50%. Which is below both the groups expection and Sonar clouds default expectation. Which is 90% and 80% respectively. Other than that a lot of informational messages are present.

This is due to a static code analysis tool called Roslynator, which we wanted to analyze and refactor code smells.Then there are a few informational security messages, which have been looked through however they do not need to be acted on. There are also reports on "bad" exception handling, for example throwing general exception objects.The second tool we use, is called CodeQL. It is used to assess the security of our project. For example hard coded credentials or other vulnerable data, which must not be open to the public. It has not found anything of concern, only false positives, in our testing suite.

4.6 Finally, describe briefly, if the license that you have chosen for your project is actually compatible with the licenses of all your direct dependencies.

5 References

References

[1] Gene Kim. *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations.* 2016

[2] *https://github.com/prometheus-net/prometheus-net.*

[3] *https://github.com/djluck/prometheus-net.DotNetRuntime.*

[4] *https://github.com/rocklan/prometheus-net.AspNet.*