DevOps projectreport

IT UNIVERSITY OF COPENHAGEN

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 in connection with the 2021 'DevOps, Software Evolution and Software Evolution' spring elective course at the IT-University of Copenhagen.
The main purpose of this project was to refactor a small social-media project, orignally named 'ITU-MiniTwit' written in Python into the programming language of choice (which became C#).

Thereafter, various DevOps tools, such as virtualization, automated deployment, logging/monitoring, and security analysis, which all allow for continuous integration/continuous deployment, were applied to our 'PythonKindergarten' application. This report reflects and discusses the DevOps process that we as a group undertook.

2 Lessons Learned Perspective

2.1 Most Important 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 a lot of time into understanding the system. One of the takeaways from this was how difficult it can be to understand and further develop old code when there is a lack of documentation. This is an important lesson because this situation often takes place in the real world.

To challenge us as a DevOps team, we decided to also evolve the existing functionality further, by adding new features. Examples of this are the application's new design and the chat-functionality. Although these functionalities were not our main focus area, they still provided a useful learning experience in continuous deployment and a training in how to release small features often.

2.1.2 Maintenance and Operation

Regarding maintenance and operation, one of the primary 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 the '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 below.

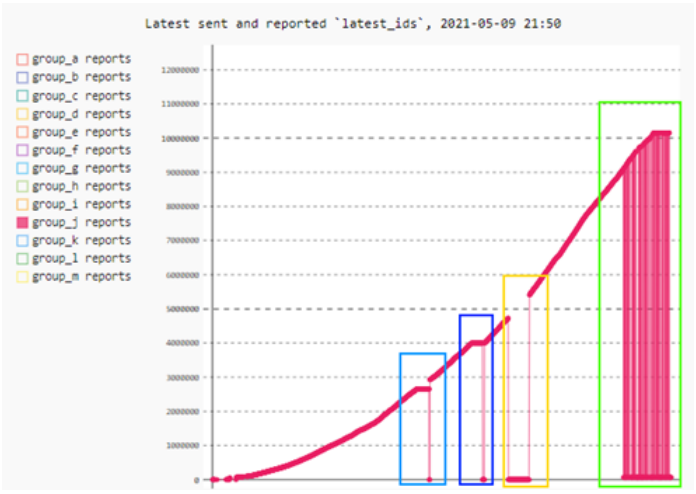

Figure 1: The graph showing "latest", an integer which is updated for each completed job sent from the simulator, thus keeping track of the current simulation progress.

All these different downtime 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, was due to a simulator-side crash. Although logging helped us out in the last down period, we realized too late that the system was down, resulting in a longer period of downtime. The major lesson learned from the two down periods was that we should have implemented some sort of alerting that would send alerts to all the team members in case of a failure. This was something we started to implement in Kibana. Unfortunately, due to the lack of time at the end of the simulation, we did not have time to implement this.

2.1.3 DevOps way of working

Throughout the project, we as a group have kept the "three ways" characterizing DevOps in mind: flow, feedback, and continual learning and experimentation [1].

Principles of Flow

While working on our system, we tried to limit the work in progress by releasing it, to the production environment as often as possible. This enabled us to have a clear idea of what had changed since last time. Which also made it easier for us to roll back and immediately locate bugs if the system was to crash.

The Principles of Feedback

With tools to monitor our system, we had constant feedback on how our system was performing. This was useful to get to see the problems as they occured, and to quickly act upon them. As we attempted to release quickly and efficiently, bugs sometimes slipped through the quality gates in our CI pipelines. When bugs did slip through, we as a team gathered to solve them in unity, enabling us to share knowledge and experiences. This was different from what we had done in other projects, that did not follow the DevOps principles, where it often was a single person who solved the issue.

The Principles of Continual Learning and Experimentation

On our bi-weekly group meetings, we shared learning points and challenges we had stumbled into, when working individually since the last meeting. We have also used the wiki tool and the issue tracker/project-board on Github to document as many issues and changes as possible, as well as making it easy to get an overview of the project's progression.

3 Process' perspective

3.1 Interactions as developers

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

3.2 Team organization

The team strived to work in an agile manner, proceeding forward step by step, and adjusting work flows to meet ongoing challenges. Mainly, the group worked together as one big group. However, occasionally, the group divided into two subgroups to increase efficiency, whereby each subgroup focused on a selected topic. Both groups then reconvened to demonstrate and discuss their findings. Remaining work was distributed to the individual group members, assisting each other in the process.

3.3 A 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. Regretably, we experienced some problems with expenses in Travis. We therefore decided to migrate to GitHub Actions. This works just as well as Travis for us. GitHub Actions is just an easy tool to use directly from GitHub, . Since we are using GitHub to store our repository, GitHub Actions is a good choice since, it is easy to integrate, easy to duplicate workflow, works faster, and has a better ecosystem of actions in a centralised app/action store. It has also made it easy for observing the status of the pipelines, since it is all on 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.

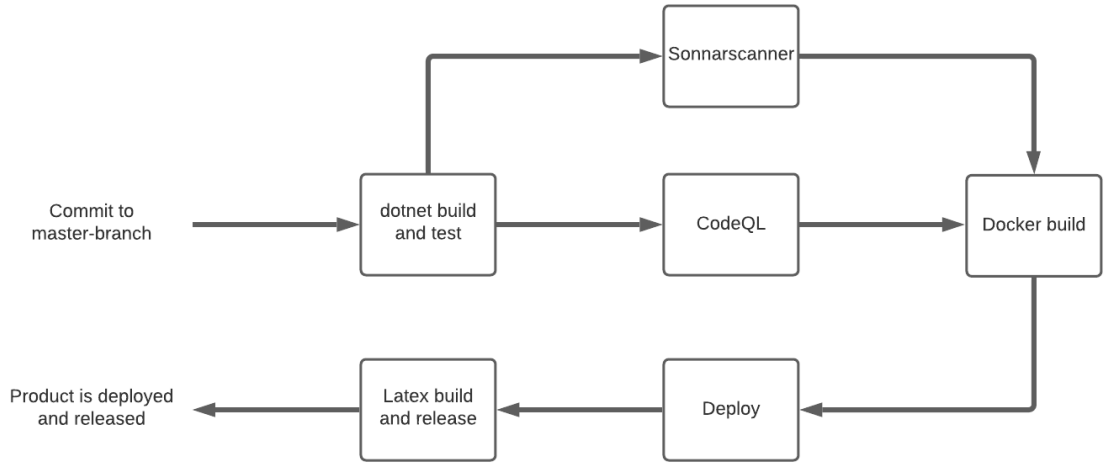

Figure 2: An illustration providing an overview of the different steps in our main CI pipeline, used on the masterbranch.

Since we are using GitHub anyway to store our repository, GitHub Actions is a good choice since it is easy to integrate, easy to duplicate workflow, and has a better ecosystem of actions in Github Marketplace, as well as a centralised app/action store. It also provides an accessible way of observing the status of the pipelines, since it is all on GitHub, whereas we had to go to another webpage for Travis.

The stages in our main workflow are *building and testing*, *CodeQL* that via autobuilds attempts to build any compiled languages, *Sonarscanner* that runs project analysis, *DockerBuild* which build the project, then *deploying* and finally *releasing* it all.

3.4 Organization of 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. Therefore we thought it would be best to keep everything in the same repository, which goes by the name of 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. [2] In this context, there are two main branches, master and develop (i.e. which we have called the 'development' branch) as well as many short-lived topic branches, which are used for implementing features. Essentially, the master branch acts as the stable branch, that is used for production code as well as releases, whereas the other branches act as the stage for code development. Upon the completion of new features developed on the topic branches, these branches were merged into development and development was later merged into master. [**?**] 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).

The actual process of working with long-running branches in a collaborative setting can present some challenges, especially with regards to eventual merge conflicts whenever branches are merged. This can cleary be seen in the instance where, for example, two or more developers have been working on the same code segment(s) on separate branches and alterations have been made. Generally, this can be resolved through creating pull requests, and have someone review the code.

3.6 Applied development process and tools supporting it

The Projects Board on Github was 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 labels, such as 'need to have', 'group discussion', etc. attached to it. These same issues were each classified by a tag number and then assigned to different group members. An overview of the current tasks which needed attention (with their current status of either 'To do', 'In Progress' and 'Done') would then be visualized on (either of our two) Projects Boards.

One of the main advantages in working agile is the concept of dividing tasks into sprints. Technically, in this project, every week has acted as a mini-sprint as the timespan between each lecture and the assignments that came with it stretched over a week. The flexibility of being able to shape and adjust our work flows over such short timespans in order to meet unforeseen contingencies has proven to be quite essential in fulfilling the DevOps 'three ways' of working (as discussed in section 2.1.3 above).

3.7 System monitoring

Monitoring of our system is done using Prometheus and Grafana, and is 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.

To expose metrics in our system we use a NuGet package called prometheus-net [3]. 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 [4] and prometheus-net.AspNet [5].
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 Logging

In our system we logged to ElasticSearch from our API, and it used SeriLogs to send these logs. We also logged our simulation errors, which we divided into errors regarding follow, tweet, unfollow, connectionError, readTimeout and Register. In order to aggregate these logs, we used Elastic-Search and Kibana. Additionally, we used ElasticSearch to store our logs in dedicated log indexes, while Kibana was used as a visualization tool for these same logs in ElasticSearch. This made it easy to keep track of our system and we quickly discovered if there was anything wrong.

3.9 Security assessment

The identified sources are our web services, for logging and monitoring, and our MiniTwit application. The servers we used to host our service 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 consisted of an exposed database connection string, which was clear text in our code. Our private keys were stored locally on developer machines. This could have catastrophic consequences if someone, by mistake, were to upload these keys to GitHub. Thereby, gaining admin rights to our servers. Dependency vulnerabilities were also a possibility, since we did not check versions of our dependencies. Had we been able to check versions in eg. our pipeline, it would have been easier to maintain our dependencies. In regards to malicious users gaining access to user data. The impact would be minimized if we kept backups of our database. However we never got to do that. After the security report was finalized, we started fixing some of our problems. The biggest issue was an exposed database connection string. We fixed this issue, by storing the connection string as a Docker secret. Another problem was the UFW firewall policies being ignored by Docker, which was solved by using the following command on the servers:

```
DOCKER_OPTS="--iptables=false
```

3.10 System Scaling and load-balancing

3.10.1 Scaling

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

3.10.2 Load-Balancing

Situations can occur 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 used a Digital Ocean Load Balancer. This acted as a gateway (single entry-point) to our Docker swarm, balancing the load between each of our servers and executing health checks, which helped to ensure that the client was always connected to a reachable swarm node.
We later learned, that we could have gained the same availability benefits, by using heartbeat to coordinate routing of a shared floating ip. This way we could have ensured that a shared floating ip would always point to an available node in our swarm, while leaving load balancing to the Docker routing mesh. Thus reducing costs and avoiding redundancy.

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 were 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 was 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 were formed during group meetings or when we were faced with issues, where a solution was required. Whenever an idea was accepted by the team, the process to make the idea a reality began. A developer or a team of developers was assigned to develop a feature for the idea. Work started off by creating a topic branch, which was checked out from the development branch. This ensured that all new features were present in the new feature branch. When the feature was finished, the feature branch was merged into the development branch. Then the new feature was tested with other new features, to ensure that everything was working as expected, as new features could have been merged into development in the meantime. After the manual testing was finished, the development branch was merged into the master branch. At this point, the CI pipeline began to run and a script to update the docker containers on our master server was executed. This script would pull a new image from Docker Hub and update its own docker containers. Simultaneously, the master server would notify other servers in the docker swarm, to pull the new image and update their containers. When all the servers were finished updating, the new features would be present in the production environment.

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 three 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 were able to have our C# code compiled to webassembly and executed with near-native performance within the client browser. The frontend communicates with our API using HTTP requests, containing json payloads of serialized or deserialized data. Apart from communicating with the REST API, the frontend also communicated directly with the RabbitMQ broker to send and receive private messages. This communication took place over WSS using the MQTT protocol. MQTT communication was then bound to a message queue on the broker.

2. **API**
   The REST API was 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 allowed 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. Various metrics were also exposed on the API, allowing us to monitor the system externally.

3. **Data Persistence**
   For data persistence we used multiple different database management systems. MiniTwit data such as users, followers and messages were 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 used 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 consisted of 3 servers and a load balancer, all of which were hosted on DigitalOcean. 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 was the master node in our Docker swarm and was responsible for maintaining cluster state, as well as managing service updates within the swarm. This server also hosted our Prometheus and Grafana containers, which were described in the monitoring section.

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

3. **Database Server**
   The Database server had multiple purposes, all of which related to data persistence. This server hosted our Postgres database, which was our primary means of persistence within the

system.
It also hosted our logging system, which consisted of an ElasticSearch database (for log persistence), as well as Kibana for visualization of the collected logs.
Lastly, it hosted our RabbitMQ broker.

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

4.2.1 System Deployment

Deployment of our system as a whole was 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.

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.

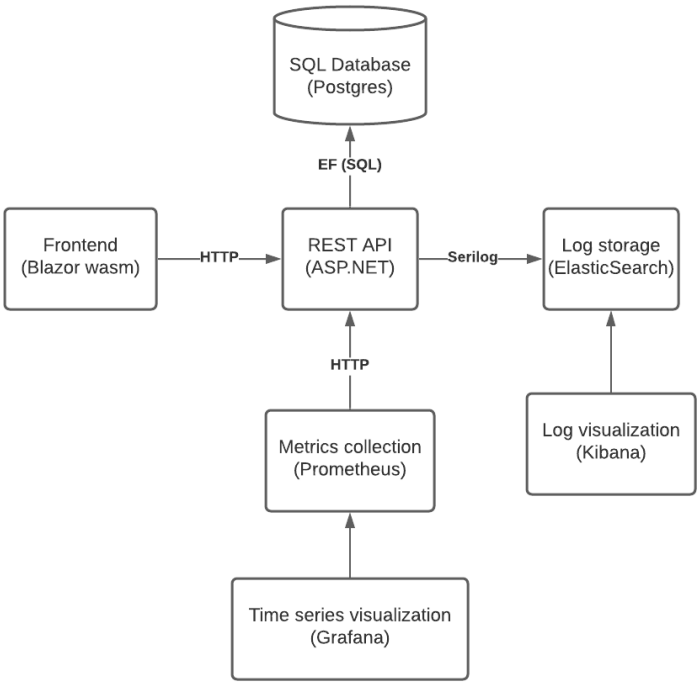

Figure 3: TODO: ADD CAPTION TO THIS IMAGE

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
- **Data Transferring over HTTP:** JSON, System.Text.Json
- **Containerization:** Docker, Docker-Compose, DockerHub, Docker Swarm
- **Pipelines (CI/CD):** Github Actions (N.B. was initially Travis until migration to Github Actions)
- **Deployment Provider:** DigitalOcean
- **SSL Certificate for application:** ZeroSSL
- **Monitoring Tools:** Prometheus, Grafana
- **Virtualization:** Vagrant
- **Logging:** Elasticsearch, Serilog, Kibana
- **Static Codecheck Tools:** SonarScanner, SonarCloud, CodeQL
- **Messaging:** RabbitMQ
- **Infrastructure Deployment:** Terraform

4.5 The current state of your system

We used two different code analysis tools and they were both executed within our CI pipeline. The first one was SonarCloud, which analysed the API and WebAssembly projects for security issues, possible bugs and general code smells.

This tool showed that our test coverage was 50%, which was below SonarCloud's default expectation of 80% and 90%.

Other than that a lot of informational messages were present under the category "Code Smells".

Additionally, there were a few informational security messages, which we looked through. However, they did not need to be acted on, because they were all false positives. Reports on "bad" exception handling also appeared, for example, by throwing top level exceptions eg. Exception.

The second tool we used was called CodeQL, which was used to assess the security of our project. For example, hard coded credentials or other vulnerable data, which definitely shouldn't be visible to the public. Nothing of concern was found here, just false positives in our testing suite.

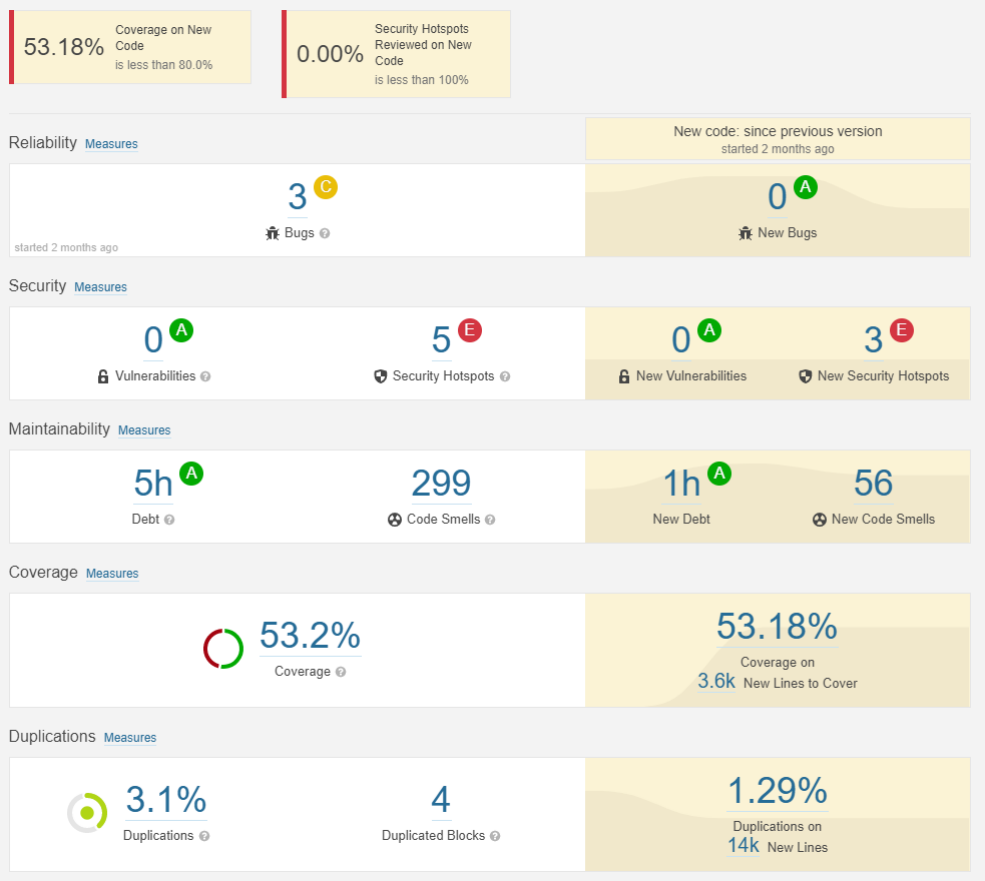

Figure 4: The latest report on static code analysis from Sonar Cloud

4.6 Licensing

We have agreed to use the Apache 2.0 License for our application in order to have the highest degree of compatibility with our direct dependencies. To ensure that we have not violated any patents on our dependencies, we chose the most permissive of licenses.

This license allows anyone to modify our software and to redistribute it, without any obligations to pay royalties.

5 References

References

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

[2] Helge Pfeiffer. *Session 02: Version control systems (Git), branching strategies, and collaborative development workflows*.

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

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

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

6 Appendix

Scenario 3(a): Some random kid rents a botnet to DDoS our servers

	Low Risk	Medium Risk	High Risk
Catastrophic Impact			
Critical Impact			
Marginal Impact	X		
Negligible Impact			
Insignificant Impact			

Scenario 3(b): Some random kid finds our repository and the connection string to our database

	Low Risk	Medium Risk	High Risk
Catastrophic Impact			X
Critical Impact			
Marginal Impact			
Negligible Impact			
Insignificant Impact			

Figure 5: Scenario 3a and 3b

Scenario 3(c): Some random group member pushes the private key to github and some random kid finds it, accesses the servers and starts mining bitcoin.

	Low Risk	Medium Risk	High Risk
Catastrophic Impact	X		
Critical Impact			
Marginal Impact			
Negligible Impact			
Insignificant Impact			

Scenario 3(d): Some malicious hacker acquires access (using ssh keys) to our users' personal data and requires a ransom to release the data.

	Low Risk	Medium Risk	High Risk
Catastrophic Impact	X		
Critical Impact			
Marginal Impact			
Negligible Impact			
Insignificant Impact			

Figure 6: Scenario 3c and 3d

Scenario 3(e): Dependency vulnerability

	Low Risk	Medium Risk	High Risk
Catastrophic Impact			
Critical Impact	X		
Marginal Impact			
Negligible Impact			
Insignificant Impact			

Figure 7: Scenario 3e

Figure 8: Our graphs in Grafana

Figure 9: Our graphs in Grafana

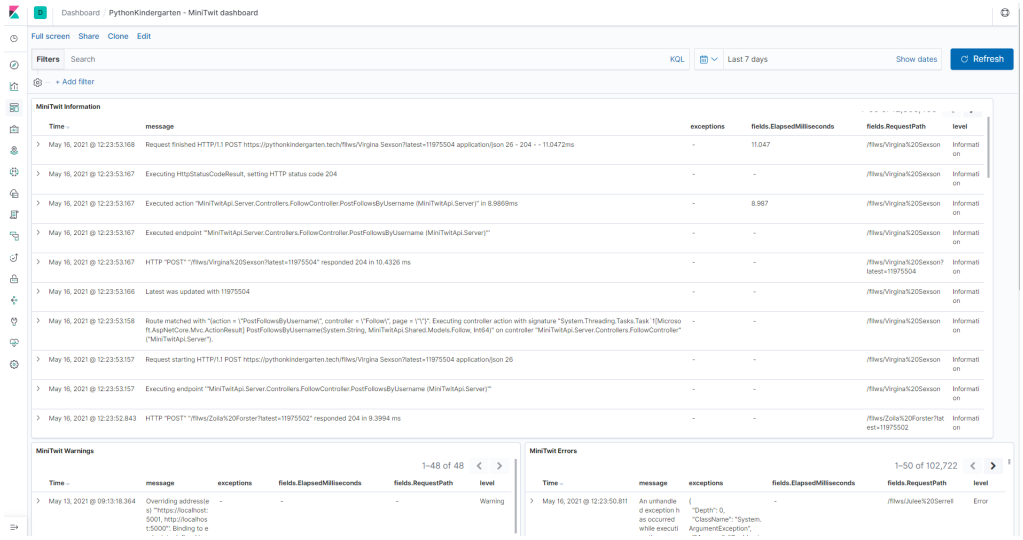

Figure 10: Current setup in Kibana

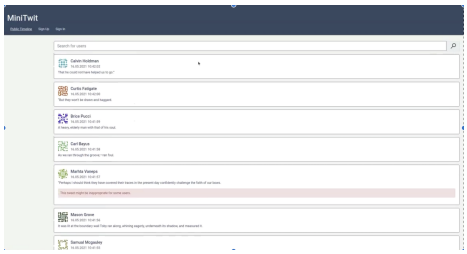

Figure 11: Current look in our MiniTwit App