DevOps Rapport

hefr sank thda jemm

May 2021

1 System's Perspective

1.1 Design of your ITU-MiniTwit systems

As our project is split up into multiple parts, we are using different design patterns.

One important design pattern is the Repository Pattern, which is easily used together with Entity Framework. The business logic is also split up between users and messages, making it simple to get an overview of the methods.

Our API is built with the REST framework in mind. This means exposing simple GET/POST endpoints, whose only purpose is to send the call to the appropriate repository, which then handles the request, and returns the result.

1.2 Architecture of your ITU-MiniTwit systems

The architecture of our system is a monolithic architecture, meaning it is entirely self-contained, and it is run together as one. Although it's a monolithic structure, the different functionalities of the system is split up into different folders. The main functionalities being the backend, frontend and database manipulation functions.

1.3 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.

1.4 Important interactions of subsystems

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

We have used different tools for static code analysis, but the ones giving us the most insightful results are SonarCloud, and BetterCode.

According to BetterCode, our project fulfills all 10 out of 10 requirements, which can also be seen on the badge of our projects `README.md`.

SonarCloud on the other hand does highlight a few more problems. This is mostly regarding commented out code, and unused fields. SonarCloud also shows some security hotspots, including hard-coded connection-strings, which will be mentioned further in section 2.9.

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

2 Process' perspective

2.1 How do you interact as developers?

Within the group we have created a Discord server, which has been our main form of communication. This has both been used for sharing documents, holding meetings, and peer programming.

Furthermore, we have used the "issues" feature on GitHub, using the provided Kanban board. This has also been used as a means of communication - communicating tasks to do, and how far along with the individual tasks, that we were.

2.2 How is the team organized?

Our team consists of 4 developers, and uses the "Centralized Workflow", where we have one mono-repository, in which every developer synchronizes their work with. By the usage of "issues" on GitHub, work on the same modules or features by several developers at the same time is limited. This is to prevent merge conflicts and duplication of work. The centralized workflow allows for us to have no power imbalances, and for each to provide value to the project evenly.

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

Our CI/CD chain consists of a deployment stage, and multiple code analysis stages. These are all run through Github Actions, in our project repository.

The deployment stage consists of 2 steps, which are pushing and deploying. The first step, firstly builds the docker containers with a `docker-compose` script, and then pushes them up to `DockerHub`. The next step then uses an SSH key to access our `DigitalOcean` droplet, and runs a deployment script, which pulls the images, and runs `docker-compose up`.

The deployment script is run on every push to our main branch, and not every release. This is due our branch strategy, which ensures that code pushed to that branch has been has been tested and verified.

2.4 Organization of your repositor(ies)

That is, either the structure of of mono-repository or organization of artifacts across repositories. In essence, it has to be be clear what is stored where and why.

We are using a mono-repository where both the API and the Blazor-application is kept. This is to ensure that all modules that co-exist are versioned correctly, and will be able to run together, as an entity. This is very useful, as we have adopted GitHub Actions, where the API and Blazor is built from files from our repository, and pushed to Docker Hub. This simplifies the process of handling different versions of applications, and also simplifies the CI/CD chain, as only one repository is needed to collaborate and execute the program.

2.5 Applied branching strategy.

We will be using a variant of the Github Flow. This involves creating an issue for every feature/fix/change needed to be done. Then creating a branch from 'development' with a name along the lines of `feature/issue-num/descriptive-name`. Then closing the issue when the branch is merged back into `development`.

When the code on `development` has been tested and verified that it works, it can then be merged into main, which is deployed via our CI/CD chain.

2.6 Applied development process and tools supporting it

We used the tools within Github a lot. For tasks we used Githubs Issue feature. Furthermore we created labels, which allowed us to get a better overview of what was smaller bug fixes, and what was bigger features. We also used Githubs projects board, in order to track what issues were open, closed, in progress, and in review.

2.7 How do you monitor your systems and what precisely do you monitor?

For monitoring we use Grafana and Prometheus. We monitor the total number of messages and users in the form of a graph and a count of the average number og messages and followers per user. We also have a heatmap with buckets of

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

2.9 Brief results of the security assessment.

We tried running nmap and wmap on our droplet, but it did not return any vulnerabilities to try out.

Although shortly after, our database droplet was hacked, and all the data was deleted. This was due to us having hardcoded the connectionstring to the database in our code. Our code analysis also highlighted this problem as a high risk, but a TA had told us it was alright, so we didn't think much of it.

2.10 Applied strategy for scaling and load balancing.

We tried out hands at using Docker-Swarm to scale and load balance our system. This involved creating docker-machines, to host out application.

3 Lessons Learned Perspective

3.1 Evolution and refactoring

3.2 Operation

3.3 Maintenance