DevOps Rapport

hefr sank thda jemm

May 2021

Contents

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.

The database itself is hosted on an external server. This is so whenever we are rebuilding the main server, or if it happens to go down, the database remains running. We then connect to the database server using its IP-address in connection-strings, in the main system.

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.

- C#
  Our application uses several components of C#. Firstly, Entity Framework Core. This allows for easy access to our MySQL database, and is the reason for using the repository design pattern. It is used by both the Blazor-application and the API. Secondly, we depend on ASP.Net Core, which includes a UI framework, namely Blazor. This is what our front-end is built in.
- MySQL
  MySQL is the database provider that we use. It contains all the data that we receive from the simulator.
- Prometheus
  This is our primary way of collecting metrics from our application. Prometheus

supports both data pulling and pushing. We use data pulling, where we scrape certain monitoring-specific data, such as cpu-load and amount of requests to any endpoint, every 15 seconds from our application. The data is stored in a time series key-value pair.

- Grafana
  We use Grafana as a visualizer for data that we collect from Prometheus. Prometheus supports collecting data through PromQL queries, that lets us select and aggregate data. Grafana also allows for monitoring of MySQL databases, which we also visualize with graphs and counters.
- Serilog
- Filebeat
- Elasticsearch
- Kibana
- Nginx
- Vagrant
- Docker
  Docker is used for containerizing applications. Each component runs in a seperate container. We use docker to create 'images' of our blazor-app and the API. These images are pushed to DockerHub, which is then pulled to our application server. Each image is included in a Docker Compose file, which is used with Docker Swarm to deploy a complete stack at once. Each 'image' is then considered a 'service', which allows for horizontal scaling of select parts of the application. We swapped Vagrant for Docker Machine, as Docker Machine provisions a virtual machine in the same way that Vagrant does, but allows for better ease of use, using load balancing with Docker Swarm.
- SonarCloud
- Better Code
- DigitalOcean
  We use DigitalOcean to provision droplet´s, which are virtual machines. This is where we host both our application and our database. It is possible

to assign a 'floating ip' to a droplet, which ensures that if, for any reason, we had to swap any of existing droplets with a new one, we were able to use the same ip-address.

- Github Actions
  We use Github Actions for continous deployment. It allows for several "steps", which could include testing when a push or pull request happens. We rely on Github Actions to test our code, push docker images to DockerHub and deploy these on our application server.

1.4 Important interactions of subsystems

The models and data manipulation classes are not dependant on any other part of the system. The API and frontend however, are both dependant on the data manipulation methods, as they are used to request and add data to the database. The frontend uses the data manipulations directly, as opposed to accessing them through the API's enpoints. This is because they are deployed together, due to our system being a monolithic structure. Ideally it would use the APIs endpoints, but as it wasn't needed in our case, we decided not to. [MAYBE A REASON WHY]

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, hold-

ing 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, as issues are moved to "In-progress" when a developer begins working in it. This serves to create transparency. 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 of messages and followers per user. We also have a heatmap with buckets of response times on the API.

Det her har thomas skrevet

For monitoring, we use Grafana and Prometheus. Prometheus has some standard metrics, which include average response times, total requests on each end-points etc. We use some of these metrics, such as average response times, to tell if any of our end-points are creating a bottleneck. Furthermore, Grafana allows for monitoring of a MySQL connection. We added monitoring of our database-server, where we monitor total amount of users, messages & follows. These are displayed in graphs, so it's easy to monitor the velocity of incoming requests, and helped us determine how much data we are receiving, and how our user-base is growing.

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.

Det her har thomas skrevet

During our security assessment, we used both Nmap and WMAP. It yielded no obvious security flaws, but did show the container names of the application-server. It also showed that the OpenSSH was version 7.9, but according to www.cvedetails.com, there are no known vulnerabilities of this version. The result of the WMAP scan was that port 80 was open, which was quite obvious. It seemed that no security flaws were found using WMAP either.

We made a mistake only scanning for vulnerabilities on the application server, as shortly before the simulator stopped, the database-server was hacked. This meant that all in-going data to the MySQL database was being intercepted, and dropped from our own tables. We found that we had not went through the MySQL Secure Installation (https://devanswers.co/install-secure-mysql-server-ubuntu-20-04/), and had left a couple of vulnerabilities. Also, when creating the database, we allowed all IP-addresses to connect to the 'admin' user, which allowed for access to the MiniTwit tables. This, coupled with our hardcoded connection string on a public repository, resulted in a very insecure database server. To solve this, we went through the secure installation, and only allowed for access to the MiniTwit table either through the root user, or from the floating IP-address that we had assigned our application-server.

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 our application.

As the stack deployed on our application server grew with things like logging, all of our droplet's RAM was utilized. This meant that the Kibana server wouldn't start, and we had problems with deploying our stack on the application server. We then decided to vertically scale our main server. For load balancing, we used Docker Swarm, in which we had our main server with the application

itself, logging and monitoring. We then added a worker node with less CPU and RAM, where we scaled only our blazor-application and our API to. This ensured that we had a highly-available application.

EVT NÆVN NOGET MED INCONSISTENCIES I LOGGING???

3 Lessons Learned Perspective

Describe the biggest issues, how you solved them, and which are major lessons learned with regards to:

3.1 Evolution and refactoring

We started off using an sqlite database file, as our database solution. This was not a good solution, since sqlite is in memory, and the main server would sometimes be down for redeployment or maintanence. Therefore we decided on an external server, with mysql running. After doing it, it became a lot easier, but the process of extracting the data and moving it to a new server was very difficult and time consuming. Therefore, spending time deciding what technologies and solutions we used, would have saved a lot of time down the line.

3.2 Operation

As previously mentioned, our database was hacked, since we had our connection string hardcoded in a public repository. Even though we were told it would be fine for this course, we definitely learned a lesson in security, that even though it feels unnecesessary to take safety precautions, you should expect the worst.

3.3 Maintenance

We learned that when searching for tools, it is easy to find one that seems to fit, and get stuck on that one. Especially in this course where we are recommended tools. However sometimes the tools may not work as expected, or maybe doesn't find your usecase that well. From that we have learned that it's important to research multiple tools, and weigh the pros and cons of all of them. There are multiple ways of implementing different technologies, and it can require some research to find a suitable solution.