DevOps, Software Evolution and Software Maintenance

IT University of Copenhagen

Thomas Tyge Andersen
thta@itu.dk

Ask Harup Sejsbo
asse@itu.dk

Joakim Hinnerskov
jhhi@itu.dk

Petya Buchkova
pebu@itu.dk

Kasper Olsen
kols@itu.dk

May 10, 2021

Contents

1 System's Perspective

Automatic CI build test

1.1 Design of your ITU-MiniTwit systems

1.2 Architecture of your ITU-MiniTwit systems

1.3 Important interactions of subsystems

2 Process' perspective

2.1 How do you interact as developers?

There were a few channels of communication. Every monday, after the lecture we would have a "traditional" standup-meeting where each member could lay out potential issues.

After the task of the finished week were discussed, and dealt with, we would turn our focus on the coming week, and allocate which tasks should be solved by next monday. More on this later.

Since the course was organized with a lecture a week, and corresponding deadlines for the work, we would also often have a less formal standup on Sundays, in the case that certain tasks were lingering, or needed discussion.

By attempting to keep our tasks small and keeping up frequent communication on teams, we tried to keep a steady value stream going throughout the week. Thus we were also hoping to decrease the amount of lead time for tasks. This was however not always achievable because of external time pressures, and much of the processing time was often placed at the end of the week.

It's probably worth noting that we never met physically during working together, which increased the importancy of frequent communications on teams, and clear communication regarding which times one would be available.

2.2 How is the team organized?

Our team was inspired by the agile philosophies in how we organized work between us, although we did not adhere to any specific agile framework. As mentioned in the previous section, we started the week i.e. just after the lecture finished, with a meeting. This meeting served both as our "daily" stand-up, our sprint review, and the sprint planning. The start of the meeting was where our stand-up took place and everyone reported on whether they had completed their previous task, and if they were facing any impediments. During the stand-up we would then be updating the Kanban board containing our backlog items. We would then proceed to do a small review of our progress and determine if the goal of the previous sprint had been achieved. If there were any tasks from the previous sprint that we were not satisfied with, these would be carried over into the new sprint. When our review was finished, we would proceed to break down the task for the given weeks sprint into smaller subtask where possible. The members of the team would then volunteer for working on whichever backlog items that they wanted to focus on, often having a pair of people working on the same task, though pair programming was not enforced at any point.

Keeping with the typical agile approach of any developer being able to work on any part of the project, we did not separate our team into separate categories, in regard to which parts that they were going to be working on. Any one member was free to voice their intention of working on a given part of the project. With that said, once a certain member had become comfortable with a certain part of the project, they tended to be more likely to commit to solving a similar problem in the future, especially so if it was building upon their previous work.

In addition to the one meeting that we had scheduled each week after the lecture, the individual pairs of developers were also meeting ad hoc throughout the week. Here they would be working on the tasks at a rate they deemed necessary, without having to consult the entirety of the team. As mentioned in section 2.1, some of these ad hoc meetings had a tendency to be held on Sundays. This was in order for everyone to have a quick catch up on what had happend, and what still needed doing before the sprint goal had been achieved. These meetings usually only occurred if we had determined in advance that we were still missing a couple of tasks.

2.3 Stages and tools

2.4 Organization of repositories

Our project is making use of a mono repository structure. This was an obvious choice due to the structure of our application. Our application is one single entity i.e., the UI that a user would access is rendered server side and shares most of its logic with the existing API. As for our various configuration files, e.g., docker compose for the logging utilities, it makes logical sense that these would be located together with the application that makes use of them.

Had our application made use of a client- server architecture instead of a purely server side one, and argument could have been made for separating the client into a self-contained repository, though in our scenario it would not make much sense to try and separate part of the application into its own repository.

2.5 Applied branching strategy

In week 2 we discussed what branching strategy we should use. We quickly narrowed the field of possible strategies to two - Git Flow and GitHub Flow, since these were the two strategies most people had experience with within the group.

Git Flow is a strategy where the main branch is used for released code e.g. stable releases. A development branch is merged out from the main branch, which is used to branch out feature branches where actual new work is done. New features are then merged back into the development branch. Once the team is ready to make a release, the development branch is merged into a release branch, then into main and released.

GitHub Flow is more simple than Git Flow since it does not have release and development branches. New features are instead merged out from the main branch, and back into the main branch once the feature is done. This will result in that the main branch containing code that is not yet released if deployment is not automatically set up.

We initially decided to go with Git Flow, since we would work on features throughout the week, and then create a single release on Sundays. We quickly realized that releasing once a week was not satisfactory for our needs, since we often wanted features to be pushed immediately once ready, and merging back and forth between the main branch and the development branch was cumbersome.

By week 7, we decided to change our branching strategy to GitHub Flow due to the addition of static code analysis and the rewrite of API simulation tests from Python to Golang. The motivation for the change in strategy is a move towards Continuous Deployment. Our pipelines now supported a full deployment process from our main branch to our production servers with automatic releases. This means that a Git Flow strategy will only hinder our lead time from user story to production.

2.6 Applied development process and tools

As breifly touched upon in 2.2, we made use of a Kanban board to keep track of all our tasks at any given point. We treated the overall objectives given for each weeks project work as our user stories, and then proceeded to split these down further into their own product backlog items. The tool that we utiliesed to keep our backlog in is azure backlog.

2.7 Monitoring

Our Minitwit application was initially hosted on Microsoft Azure, which included some limited monitoring, like CPU usage, memory usage, and response times. But due to our application were hosted in a Docker container, we weren't for instance able to get specific metrics on how long time the different endpoints took to execute. This meant that we had to implement custom metrics programmatically.

We ended up using Prometheus for this purpose. We set up a droplet on Digital Ocean as our Server, where the client reported to. To visualize the data, we had another Digital Ocean droplet with a Grafana client running. This client tapped into the Prometheus server to get the data. The data we visualized were:

1. CPU usage
2. Memory usage
3. Number of registered users
4. Number of messages (tweets)
5. API endpoint execution duration
   (a) Average
   (b) 50th percentile
   (c) 95th percentile

When we initially deployed our application with custom metrics to Azure, we noticed that we didn't get any metrics. We later found out that Azure does not allow outbound traffic on the plan we had. To address this issue, we change our cloud provider to Digital Ocean.

2.8 Logging

We initially choose to implement the ELK-stack (Elasticsearch, Logstash, and Kibana), since it was a solid and proven stack in the industry. But due to unforeseen implications with the ELK-stack and how our application infrastructure was set up, which made the implementation more complicated than we had time for. We then decided to explore what other logging services we could use and ended up with a service called Datadog.

Datadog works by installing an agent on the server, which listens to stdout from the application. On the application level, we use a Go library called Logrus, which can specify different log levels, i.e. info, debug, warning and error. We for instance log all endpoint requests on info-level, while failed login attempts are logged as a warning since it could indicate compromised users. Application errors, like records in the database not found or when errors are thrown are logged as an error. Logrus gives us the flexibility to log whatever we want in whatever format since it can take JSON as input. We just need to serialize the data.

Figure 1: Example on logs.

2.9 Security assessment

2.10 Scaling and load balancing

3 Lessons learned perspective

3.1 Evolution and refactoring

3.2 Operation

3.3 Maintenance