IT-University of Copenhagen

DevOps, Software Evolution and Software Maintenance

Final Report

Authors:

Diyana Vladislavova Boyadzhieva
Fadi Atia Dasus
Lucas Schütt Nielsen
Sara Holst Winther
Vivian Luisa Machindano Seerup

Masters in Computer Science
May 31, 2022

IT UNIVERSITY OF COPENHAGEN

Contents

1 System's Perspective

1.1 Design of Mini-Twit Application

Our app consists of a web app made with the React for Javascript, a backend app made with the Spring framework for Java, a MongoDB database, as well as an API for the simulator written in Go. The Spring-backend exposes a REST interface for interacting with the database. The interface allows for user registration, logging in, getting and posting tweets, and so on.

The web app uses this REST interface to provide its functionality and display a graphical interface in a browser. The flow of interactions between a user and MiniTwit is illustrated in Figure 1.1

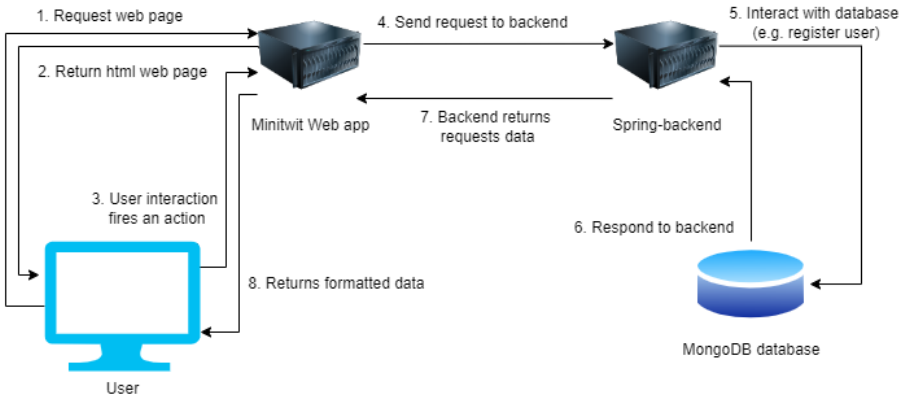

Figure 1.1: Flow of requests from a user

The Simulator API functions as an intermediate layer, translating and forwarding requests from the simulator to the Spring-backend. The flow of interactions is illustrated in Figure 1.2

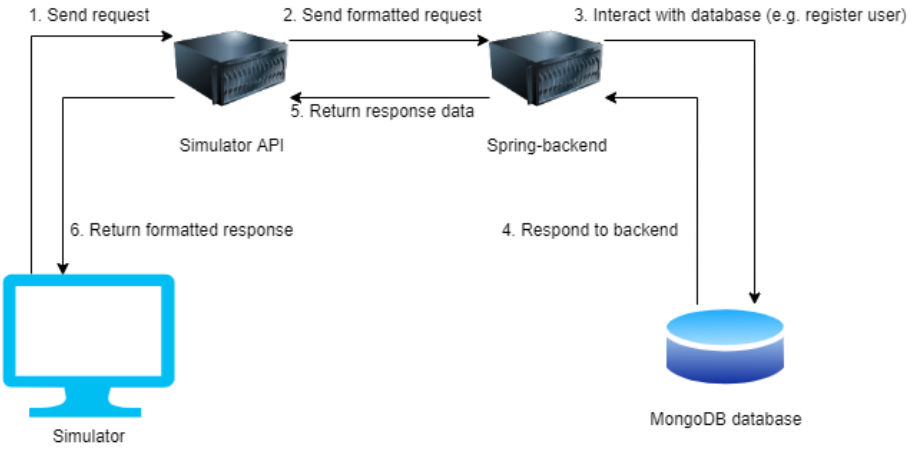

Figure 1.2: Flow of requests from the simulator

1.2 Architecture of Mini-Twit Application

The backend is designed based on the three-layer spring architecture, which is dividing the code base into three separated layers of abstraction, which help in isolating components in the codebase and facilitate SOLID and clean code principles.

The upper layer is where the controllers live and manage client requests. The business logic layer contains the needed services in which we apply any logic needed for managing the application. The last layer consists of the repositories which communicate with the database using ORM supported by Hibernate. The three layered architecture can be seen in Figure 1.3

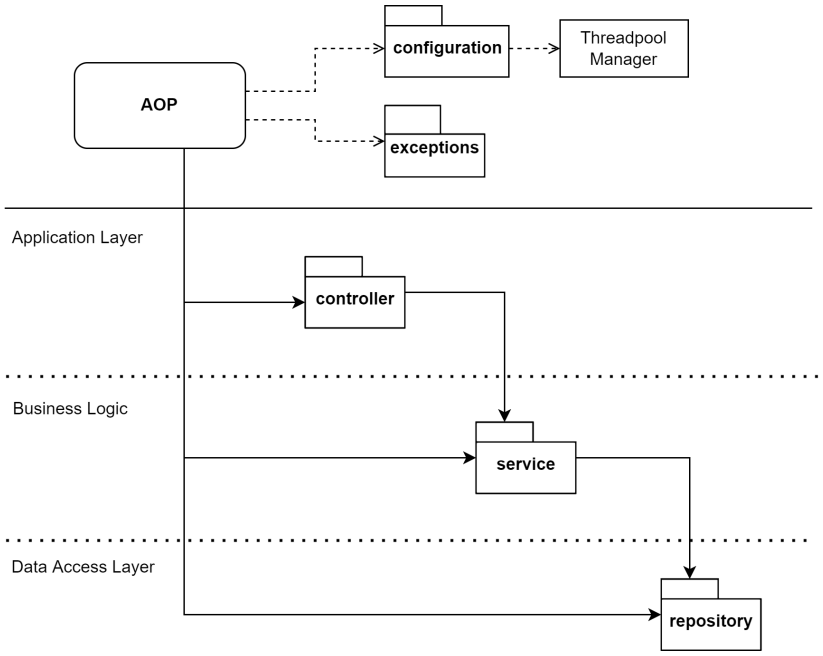

Figure 1.3: Three layered architecture shown in package diagram

The layers are connected using the dependency injection container to ensure that the code is loosely coupled, and all the dependencies are loaded to each component during the initialization phase of the server. Exception handling and other configuration, e.g., thread pool management and database configuration context load are handled using aspect-oriented programming and proxy pattern which clean the code from all the boilerplates try-catch blocks and NPE.

1.2.1 Simplified 3+1 architecture

For this project, use a simplified 3+1 architecture, in this regard we looked at the module viewpoint with a package diagram of the backend as well as an allocation viewpoint via a deployment diagram.
The concern of a module viewpoint is to look at how the functionality of the code is organized. For this we refer back to Figure 1.3 which shows a package diagram where the packages are separated into different layers to account for the three layered architecture of our system.
For the allocation viewpoint, a simple deployment diagram was created - this diagram shows how all three elements within the system are connected to the same VM, the element backend is then connected to the database. See Figure 1.4

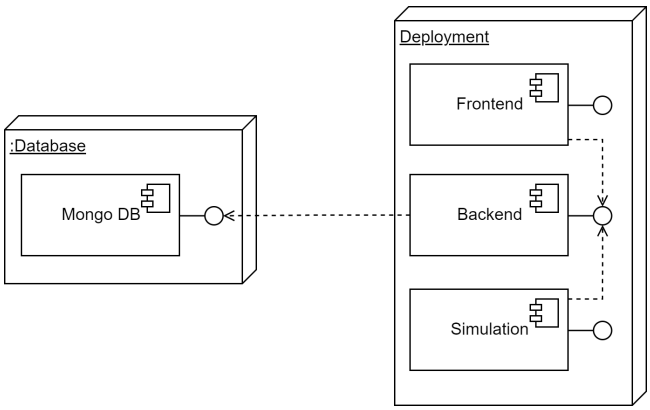

Figure 1.4: Deployment diagram of the system

1.3 Dependencies in Mini-Twit Application

In this section we will list and describe the tools used in all levels of abstraction of our system. Additional dependencies can be found in Appendix B

1.3.1 Spring Java

Spring framework offers a set of predefined projects for integration services deployed on the cloud, e.g., spring boot, spring cloud, and spring AOP. Exploring the framework was the main motivation behind choosing it for the backend.

1.3.2 React

The frontend of our ITU-MiniTwit system is developed in React. We had only 1 week to refactor a web application with a lot of functionality and the only experience some of us had was in React. We considered using Angular but that is an entire framework - everything is done in a specific way rather than flexible (like in React) and it takes considerable time to learn. Since that was not the goal of the course, we decided not to spend so much time on learning a new framework. We also considered but didn't choose Vue.JS because that is a library, designed to develop only web components rather than a full web application with state management etc.

1.3.3 MongoDB

MongoDB is a noSQL open-source document database, noSQL databases are highly flexible, allowing variations in the structure of documents and storing documents that are partially complete. After modeling the entities in the MiniTwit project, it turns out that a big object, including long text, picture or even video, is an option. That said, the decision is to be made between relational databases and document-based No-SQL databases. SQL database stores big files as a blob attached to the file system, however, blob storage in SQL server is limited to 2GB, which is not enough for a system that deals with large amounts of data, clearly MiniTwit will have this possibility in the future. Finally, storing big files in SQL Leads to memory inefficiency since all the data in SQL must go through the memory first. Mongo, on the other hand, uses BSON format to store the data in memory and it saves data in a form of a document. One document may not exceed 16MB, which is an acceptable limitation for MiniTwit, and if a document exceeds this limit, GridFS comes to the rescue which is introduced in Mongo 2.1 which is a type of document designed to handle large files that exceed 16MB. Thus, Mongo is chosen over SQL databases.

1.3.4 Azure

Azure is a cloud provider with solution for all infrastructure problems, providing services in various domains namely compute, analytics, storage and networking. Azure provides a production-ready configurable cluster as PaaS, and it was a perfect choice for the MiniTwit use case since it's a low budget project. Creating Kubernetes from scratch requires at least three VM of a specific type that match the Azure requirement for hosting the master node as well as for the worker node in the node pool. Since this project is running on a student's subscription with limited credit, it was not affordable. However, the design for such a self-managed cluster requires a master node to run Kubernetes main functionalities including the API, a worker node to run pods inside which the docker container will run, a public client to be the proxy for interacting with the controller and managing the deployment flow, see the diagram below

1.3.5 Vagrant

We choose to deploy our application in a remote server provisioned by Azure using Vagrant. As we create a remote server we understand that Vagrant is a powerful tool for VMs based environment management. With this approach we aim to have a consistent environment for deploying the docker containers in our cloud infrastructure. In choosing vagrant we considered the follows aspect:

- Combine the power of virtualization with the reach and scope of Microsoft Azure, the system benefits of both are amplified giving us speed, agility, performance, global presence, security and disaster recovery at scale.
- It's a popular tool for VMs based environment management, open source, fairly easy to learn with rich documentation and integration with other tools.

1.4 Interactions in subsystems

The diagram in Figure 1.5 shows the interaction between the subsystems of the ITU-MiniTwit system.

1. The code source consists of the react app for our front end, the spring API for the backend, the simulator for testing our endpoints and sending request throughout the course and finally the deployment folder with the vagrantfile for the deployment process. The developer makes all the relevant configurations to GitHub, Azure, Docker and GitHub Action. From the development machine the code can be push/pull from GitHub.
2. For the second stage we have the most important git repositories corresponding for the backend, frontend, simulator and deployment. After that we create 3 .yml files named docker-publish.yml for the 3 first repositories named above. Those files get triggered on push.
3. After getting triggered the GitHub Action CI/CD pipeline automatically builds the docker images and push it to Docker Hub. If the docker images are pushed successfully, another GitHub Action triggers. This action updates the docker containers on the server For details see Appendix C.1

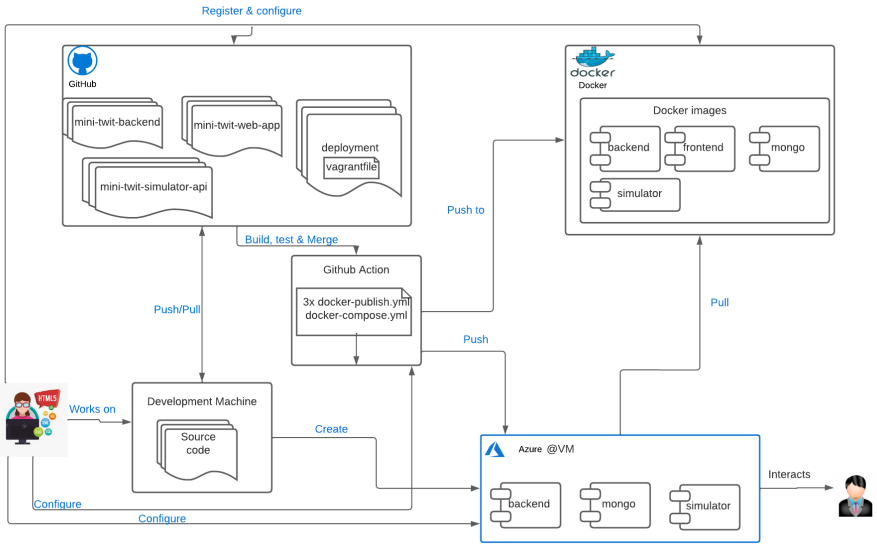

Figure 1.5: The interactions of the subsystems used in this project

1.5 The current state of the system

For the quality assessment of the code, we used online tools integrated with github.com, namely sonarcloud.io, codeclimate.com and bettercodehub.com. We got good results, especially for the backend where bettercodehub.com, for example, showed that the code is loosely coupled, the blocks are simple with no duplications. The results were slightly different for the web application where we got some duplicate code from codeclimate.com but sonarcloud.io showed no duplications for the same project and thus we decided not to take any action.

1.6 The license of the project

As we are working with open source tools we decided to license our project using the MIT License. MIT License is a permissive license and one of the most used open source licenses. It allows the public to modify, share and for commercial use without us being liable for anything. To decide on the license we analyzed our goals for this project which is to have future contributions as well as help others on their work just as we got inspired by the ITU-MiniTwit before the refactoring process. In terms of license compatibility, our top-level dependencies falls into the MIT and Apache 2.0 licenses which a both permissive open source solution high compatible. See Figure1.6.

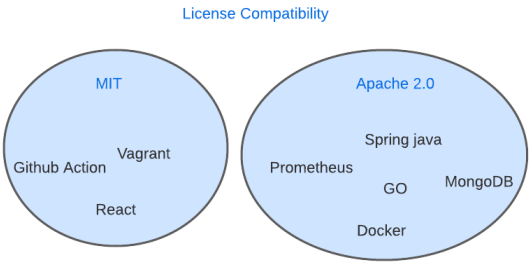

Figure 1.6: License Distribution

2 Process' Perspective

2.1 Team Organization

Within the team, we decided to create a GitHub organization which allows us to keep a collection of GitHub repositories in a shared resource. Within the group we all had different roles which allowed us to work more efficiently, this means that whilst some group members might not seem active within GitHub, that does not mean that they were not active within the project since things like the azure setup does not show up within GitHub

2.1.1 Interaction as developers

We had weekly meetings in the University where we would quickly update each other with our progress and then divide the new tasks. In some cases, we needed to spend more time to finish/plan a task, so we would schedule an extra meeting in the University, or we will make a Teams call if not all members are available to meet in person. Sometimes, we would do "pair-programming", so we would pick a task (not necessarily a programming one) and we would research the topic together and compare our findings.

2.1.2 Branching strategies

We decided to follow the Centralized flow to manage our GitHub repositories. That means we have one Main branch and we create a subbranch from it per feature. When the feature is done, we merge the subbranch with the master branch. That is because one person is typically responsible for a specific project, meaning only one person will be working on the project at a time and thus we do not need a complex GitHub flow with many different branches. Nevertheless, we did use a Develop branch for the web application but that is mostly for learning purposes – push to Develop, make sure there are no conflicts, and the app works as expected and then finally merge that with the Main branch.

2.1.3 Development process and tools

To manage our tasks, we used Trello and GitHub. In Trello we were assigning mostly non-programming tasks like Code Quality assessment, Risk Analysis, Software licensing etc. Intuitively, we were tracking the state of each task by moving from a "To Do column", to a "Done Done" column that contained "reviewed" tasks. For the programming tasks, we used GitHub issues to close on a Pull request that contains the required updates.

2.2 CI/CD chains

For the sake of learning, the project is deployed to two different environments/setups. One using a VM created using Vagrant, and another AKS cluster hosted on Microsoft Azure platform.

2.2.1 Virtual Machine created using Vagrant

Vagrant Deployment

We use Vagrant with the plugins vagrant-azure and vagrant-docker-compose.
With vagrant-azure we configure a VM in Azure, specify which ports should be open, VM size, and similar details.
Once the VM is running Vagrant will do the following.

- Setup Docker.
- Insert an ssh key as an authorized key, that will be used by our GitHub Actions workflow later.
- Transfer the docker-compose.yml file
- Use the vagrant-docker-compose plugin to setup Docker-compose, and start the app defined in docker-compose.yml

When Vagrant is finished setting up the VM, our MiniTwit app will be online.

2.2.2 AKS cluster hosted on Azure

Azure provides a production-ready configurable cluster as PaaS, and it was a perfect choice for the MiniTwit use case since it's a low budget project. Creating Kubernetes from scratch requires at least three VM of a specific type that match the Azure requirement for hosting the master node as well as for the worker node in the node pool. Since this project is running on a student's subscription with limited credit, it was not affordable. However, the design for such a self-managed cluster requires a master node to run Kubernetes main functionalities including the API, a worker node to run pods inside which the docker container will run, a public client to be the proxy for interacting with the controller and managing the deployment flow, see Figure 2.1

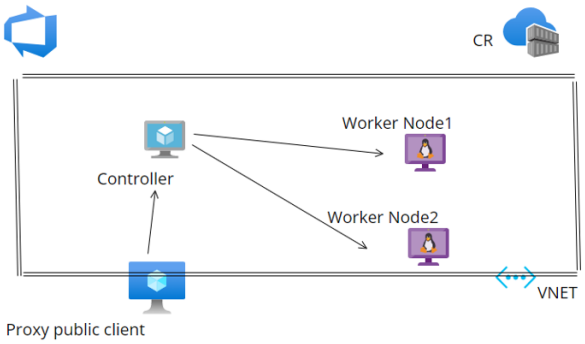

Figure 2.1: Kubernetes connect to workernodes

However, AKS cuts the cost to the absolute minimum since we only need to pay for what we use, namely the worker nodes and the master node is free.

K8s provides a rest API that facilitates the interaction with the cluster, once a request is received in the API, typically it's a deployment manifest, then k8s will store a copy of the manifest in the cluster store. Once the store confirms that the transaction is written successfully, then the scheduler will pick the manifest and assign the task to whatever free node is available in the node pool. The task immediately will get an id and it will be registered in the controller in which an event loop is running to check whether the cluster current state meets the desired state registered in the store. On the other side, the worker node has an agent configured as a listener registered in the scheduler once it receives a task it will immediately create a new pod and initialize a suitable runtime for the container to be running. The Kube-proxy job on the worker node is to manage internal networking within the node. See Figure 2.2

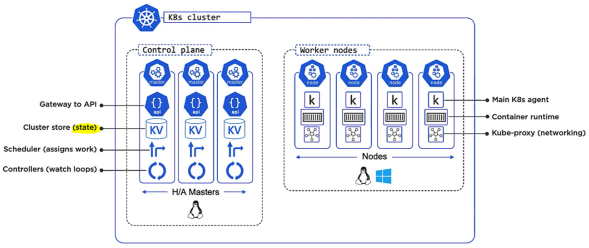

Figure 2.2: Kubernetes cluster[1]

MiniTwit infrastructure design: for the MiniTwit AKS setup, the design included two different setups. The first one is a Landing zone infrastructure which is the stateful part of the project, since it contains 1- cosmos database instance running as a service, 2- key vault to store sensitive data like connection string and active client id that allows the pipeline to push images to the CR, 3- container registry for storing docker images. The second one is the AKS cluster itself: the idea is to be able to destroy the entire cluster and rebuild it in no

time, since the cluster is a stateless by design, if anything bad happen, security wise for example a malware is installed somewhere in the services, or any other misconfiguration that might happens due to development error, then by running the pipeline we should be able recover to the initial state desired and saved in the pipeline.

Our cluster consists of the following components; An ingress controller to manage outbound routing into two services which are exposed publicly, namely the simulator and the web application. Behind each service there is exactly one pod running the application inside the container. However, the backend service, which is the busy one, is managing three pods running the spring backend application but not exposed publicly. The Node, shown in Figure 2.3 runs inside a Virtual network which facilitates isolation and works as a Lan network inside the cluster, the gateway for this network is the ingress controller.

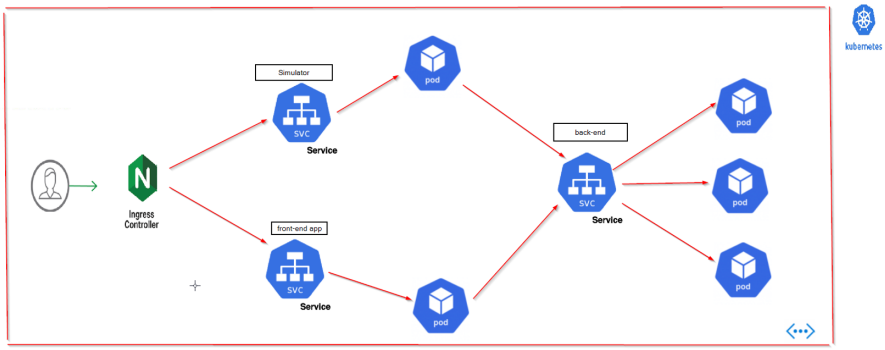

Figure 2.3: Kubernetes

2.3 Monitoring

For the MiniTwit project we have only monitored the backend spring service because it's the heart and the brain to our system. Since the service is hosted in Azure, we have used Azure monitoring features to integrate our service directly with application insight. The configuration requires an AI agent, see "`infrastructure-aks/backend/k8s/src/main/resource/AI-Agent.xml`" to be installed and packages within the backend jar file and attached to the service at the class load level. Moreover, we have exposed a set of telemetry to be exposed from spring contest to public. The telemetries we have exposed include health check and information about the service state: up/down, livenessState and readinessState, memory consumption state exposed using the metric probes and service info provided by default in spring-boot-cloud.

By integrating with application insight in Azure, we have configured a test health check to send a post request to the health, livenessState and readinessState endpoints exposed from the service and monitor response time the availability state for the service. See Figure 2.4

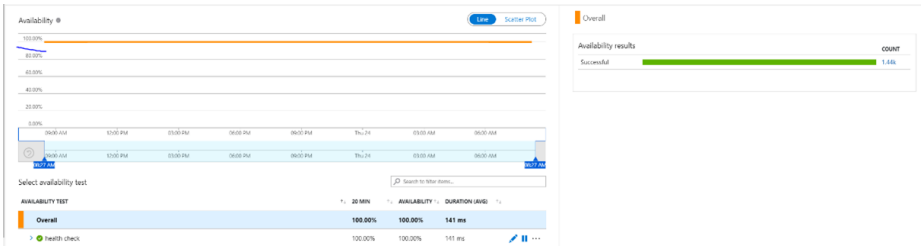

Figure 2.4: Screenshot taken of the availability of the system

Using other previously mentioned exposed metrics, we could see a live capture of system performance, which was during the entire course under 200ms on average. See Figure 2.5

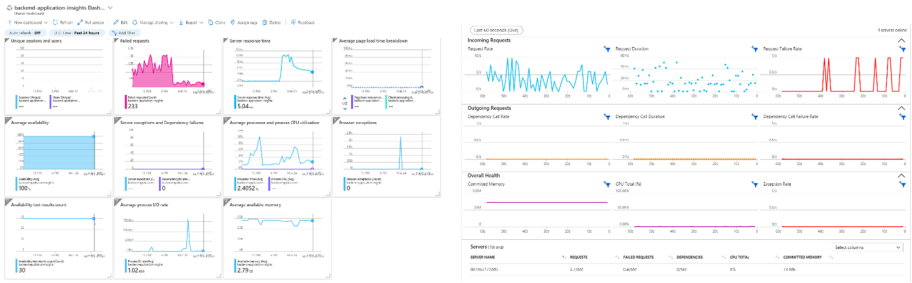

Figure 2.5: Metrics exposed in the monitoring process

As another alternative for monitoring we used Prometheus and Grafana. They are two powerful monitoring tool used for the visualization of data via charts, graphs, and maps to help us understand behaviors of the system and make decisions. In our project as we are using other monitoring approaches we decided to apply Prometheus and Grafana only on backend, as it's an important part of the system as well as for learning process. Prometheus can easily be added to a Spring API and combined with Grafana provides visualization.

2.4 Logging

Once-again, we took advantage of the application insight capabilities to store and aggregate logs coming from the backend service, with a maximum retention policy of 30 days. To achieve that we have intercepted each request coming toward our controller request handler using aspect-oriented programming to register the request in a logging agent, namely Logback which we have registered in application insight, and explicitly tracing the event triggered from the controller method handler to the business level, data access level and back again. We have set up the logging level to be configured at runtime using spring profiler and changing the log. Level attribute to be: info, trace, debug if needed. As a result, we have a control panel in Azure that shows our logging and aggregation, including response time and failed requests aggregation function. Moreover, the trace provided by logback.spring integration gave us the ability to inspect the logs and trace the route and messages after execution. See Figure 2.6.

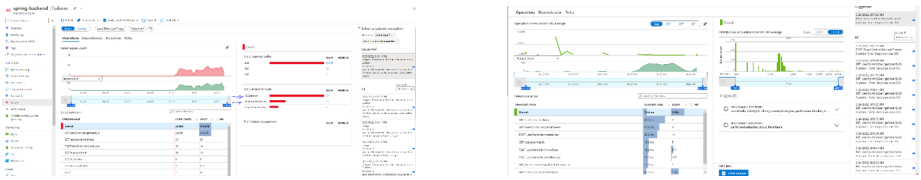

Figure 2.6: Log screenshots

The logs reveal an important issue to the developers which was a known bug in spring container dependency injection context when running on our Linux distribution version. Fortunately, the error did not cause the service to crash, and it occurred only while the cron-job in Linux was running, so we let it be since it seemed to be harmless.

2.5 Security

For the frontend, we tried to perform Cross-Site Scripting and SQL Injection attacks, which both failed because React sanitizes all inputs. We also tried to upload a file in a random input field without success because React only accepts file inputs from an Input field with the “File” type which we do not have either. On the other hand, we have broken authentication and session management because we execute the login by making a simple request that contains the plaintext username and password (thus visible to anyone that sniffs the network with a tool like Wireshark). Moreover, we manage the login state by keeping the username in local storage meaning anyone who creates such local storage key-value pair and access our application is automatically logged in with the user whose local storage’s username appears. A possible remedy for these issues will be to never send unencrypted login details to the web server and to implement some sort of token, like the JWT token that has expiration and based on it to validate the login state of a user. However, that would take time to implement and we decided it’s not of high priority as for the objective of this course. For the backend, we did some port scanning and found some possible exploits but neither of them was successful.

2.6 Scaling and load balancing

Our tomcat server (spring backend) is configured to accept 100 requests at once and queue them in a thread pool consisting of 10 threads per instance, and the container hosting the server is given almost all available RAM from the VM approximately: 2 GM, so we have not needed to scale out the application during the simulation process. However, we were using the vagrant setup on the VM for almost the entire course, and we switched to AKS in the final 2 weeks. Using k8s we have a replica set configured to be replicated 3 times. To manage networking and load balancing, we have configured a routing service that has a static IP address to manage networking, which works as a gateway in front of the pods, see Figure 2.7

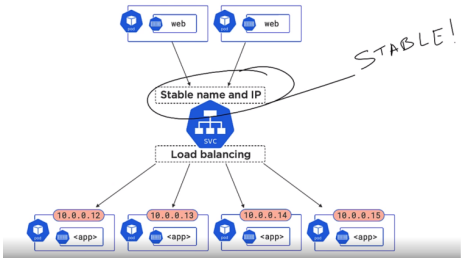

Figure 2.7: Load balancing [1]

In the current cluster we are performing a rolling update deployment and it works as follows: the scheduler in the master issue a new task to any available node from the node pool, once the task is received and acknowledged by a node, one of the controller watch loop, namely the replica controller, will register the desired state in the deployment manifest and compare it with the current cluster sate to make sure that the cluster is running exactly what is specified in the manifest. The replica set controller will watch, in a loop, the liveliness and liveness probe and the readiness probe for any new created pods and it will align with the networking service or with the load balancer service to start routing some traffic to the new pod, while keeping some of the traffic to the old pods. Once the pod could handle the traffic correctly, the replica controller will kill one of the old pods and replace it with the new one. The same process will continue until all the old pods are killed and replaced with new pods.

A GitHub repositories

- Mini-twit-web-app: `https://github.com/DEV-OPS-Group-b/new-mini-twit-web-app`
- Mini-twit-simulator-api: `https://github.com/DEV-OPS-Group-b/mini-twit-simulator-api`
- Mini-twit-backend: `https://github.com/DEV-OPS-Group-b/mini-twit-backend`
- Infrastructure-aks: `https://github.com/DEV-OPS-Group-b/infrastructure-aks`
- Deployment: `https://github.com/DEV-OPS-Group-b/deployment`
- Vagrant: `https://github.com/DEV-OPS-Group-b/vagrant`

B Dependencies

B.1 Docker

Docker is a software development platform and virtualization technology that makes it easy the development and deployment of systems inside of a virtual containerized environments. As we work in a team docker helps us avoid any problem with compatibility issues, making the system simple, quick development, easy to maintain and deploy.

B.2 Go

For the simulator API we decided to program in Go which is another popular programming language. It's stable and offers good mechanisms for concurrent and parallel developments. We chose Go mostly for the learning process as we find it interesting to explore different tools and technologies in the system. It's also pretty light-weight, which suits the rather simple API.

B.3 GitHub Actions

GitHub Actions is a tool provided by GitHub for creating a CI/CD pipeline, it helps us automate the process of building, testing and merge of the changes in the code. In our case helps us build a strong pipeline that updates docker images and assist the deployment process as well.

B.4 Prometheus and Grafana

Prometheus and Grafana are both powerful tools used for monitoring systems data, it helps to understand the behavior of the systems for better decisions process.

C Pipelines

C.1 GitHub Actions Pipeline

The pipeline uses GitHub Actions to trigger workflows when code is pushed to our repositories, and Docker Hub to store our Docker images.

There is a GitHub workflow named docker-publish in the backend, frontend, and simulator repositories. There are small differences between them when building the Docker images. Secrets such as docker login credentials are stored in GitHub Secrets, and are only available as environment variables to the workflows in our repositories.

The docker-publish workflow is triggered by a push and the main flow is the following:

1. Get the newest version of the repository
2. Login to Docker Hub
3. Prepare metadata for building the Docker image
4. Build the Docker image and push it to Docker Hub
5. If the previous steps succeed: call the workflow in our deployment repository called docker-compose

The docker-compose workflow is triggered when called by another workflow: It takes 3 parameters HOST, USERNAME, and KEY HOST is the address of our server USERNAME is the user on the server setup by vagrant KEY is the ssh key

1. Get the newest version of the repository
2. Ssh into the server with the provided credentials and address. The following is run on the server:
3. Download the newest docker-compose file from our deployment repository
4. Get the newest version of all docker images used in the docker-compose file
5. Stop, update, and then restart any containers that have a newer version
6. Delete unused docker images

C.2 Pipelines for AKS clusters

we have two pipeline flows, namely, GitHub action and Bitbucket pipeline. Bitbucket is a commercial platform, however, it has the possibility of free usage up to 3 users, which was enough for experimenting with it.

In the bitbucket pipeline the following repository were hosted, including the pipelines:

1. A copy of existing services, e.g., spring backend API, React frontend web application

2. Two repositories to manage creating and destroying the infrastructure, AKS in particular.

For the backend service, the pipeline file can be found in "`infrastructure-aks/pipelines,bitbucket/backend`", the pipeline build runs on top of a maven JDK slim image as a base image sine the application runs Java. Moreover, it relies on two services to be available during the build, viz, mongo and docker. The pipeline has two steps.

1. Build and test step: in this step, maven will package the application and run through all the tests available in the project to ensure nothing is falling for each deployment finally, if all tests pass, a jar file will be created and placed in the root directory. Using spring the testing process is also controlled by a plugin "Jacoco" to manage that each feature is covered by a test ratio and aggregation test coverage ratio under which the pipeline will fail when maven runs the verification process before generating the jar file. Running the tests includes interacting with a testing database running in the pipeline, here where mongo service is used, and using spring profiler the application will pick a connection string for a local mongo instance that runs in the pipeline. To differentiate between the production database and the testing one, spring profiler will pick the connection string from a repository variable stored in bitbucket according to the active profiler configured inside spring application.property file.

2. Dockerize step: in this step the pipeline will create and publish the Docker image to a container registry hosted in Azure. To do so, the pipeline uses repository variables to authenticate a predefined client id in Azure and tag the image with the commit number and push it to Azure CR. The image will be created based on a docker file described as a layer docker file, the file can be found at "infrastructure-aks/backend-k8s/dockerfile". The idea behind the layer docker file is to produce small images "180 MB" max, and that is achieved by using a cached layer for building the image, so for instance, the dependency part is less likely to be changed compared to the actual code, hence by caching layers and transforming them forward into the next layer in the docker build, only the top layer, which in this case will be the actual code modification, will be rebuild. Thus, faster building process.

Note: the pipeline is missing an important step in which the newly created image, after pushing to the CR, must be included in the YAML deployment manifest, and sent to the Kubernetes cluster to be scheduled in a pod. This step requires a predefined service principle on the active directory level that is allowed to interact with the cluster from the pipeline. Unfortunately, in the student subscription provided by Azure, we do not have access to the active directory configuration to add this user, and as a consequence, every new push to the CR will require a manual trigger from the cluster to pull the new image and start the rolling update deployment and replace old pods with the new ones.

C.3 The infrastructure pipelines

1. 1 Creating Azure Landing Zone Infrastructure: This pipeline is used to create the infrastructure needed for the Data tier layer in our Kubernetes

setup "Azure calls it the Landing Zone area". Source:" `infrastructure-aks/minitweet-landing-zone/build_pipeline.yml`" and for the separated resource bicep files see sub-folders in "`infrastructure-aks/minitweet-landing-zone/`"
The pipeline consists of 4 steps using

- Creating the resource group, under which all other resources will be manages, and rather than including all the boilerplate code in one big yaml file, we have separated each resource in as a bicep file and reference them in the pipeline as variables.
- Creating the container registry, to store images pushed from the services pipelines
- Creating a cosmosDB instance as a service managed by Azure, to store our data
- Creating keyvault to store sensitive information

There is a reverse build/ destroy pipeline to clean up the resources when needed, source "`infrastructure-aks/minitweet-landing-zone/destroy_pipeline.yml`"

2. Creating MiniTwit AKS infrastructure "stateless zone": the same concept as above, however, this pipeline is designed to create different type of resources in Azure, and it consists of 5 tasks

   - creating the resource group in Azure to contain the sub-resources
   - creating a static service connection IP address, which will be the public IP for the cluster
   - creating a storage account, needed for issuing commands with Azure CLI towards the cluster
   - creating a virtual network inside which the cluster will operate
   - create the AKS cluster

   There is a reverse build/ destroy pipeline to clean up the resources when needed, source "`infrastructure-aks/mini-tweet-testing/destroy_pipeline.yml`"

References

[1] https://kubernetes.io/. Kubernetes Documentation — Kubernetes.