DM874 – Microservices and DevOps

Group report 1, Autumn 2020

Jakob Frydendal Donbæk Jensen
jakoj17@student.sdu.dk

Andreas Lyngholm Poulsen
andpo17@student.sdu.dk

Bjørn Angel Kjær
bjkja17@student.sdu.dk

Eivind Roslyng-Jensen
eiros15@student.sdu.dk

Md Shihab Ullah
mdull20@student.sdu.dk

Valentin Hoeck
vabro15@student.sdu.dk

Department of Mathematics and Computer Science
Faculty of Science, University of Southern Denmark, Campus Odense

Introduction

The goal of this project is to implement a platform and service that automatically tests submissions of programming assignments. Because testing is automated, the student will quickly know if their solution is made to their teachers specification, and the teacher do not need to manually run every solution handed in.

Motivation

Automatically testing code can be difficult because of the many languages, frameworks, and testing tools available. It would be simpler if we where to restrict ourselves to specific languages and testing tools, however as tools gets deprecated, and languages change, our system would become outdated, and require constant development. Not to mention the restriction could make the platform less attractive to someone who creates assignments in a supported, *and* unsupported language, and would therefore have to do manual testing anyway.
Conveying the requirements of a coding assignment can be difficult, especially if the testing is to be automated. The students could test the code themselves, but this would require that they are using the test tool correctly, and that their environment is compatible with their teachers. By creating a language and tool agnostic platform, the students only need to know how to use the website, yet can still get feedback on whether their solution is correctly structured, and the teachers tests are parsing. We also want your platform to be so that the teacher can choose what the students see. They can either completely hide the results, or give the raw output of their tests.

Result

Despite these challenges we have made a platform where teachers can create programming assignments, and their students can hand in their solutions for testing. Assignments can both be public, and for students following a specific course. The platform uses authentication to ensure the appropriate level of access to all users, and to prevent exposing the system and its components unintentionally. The teacher has full control over the environment and tests for the assignment though the use of Docker. This means that the teacher has to set up the environment and test tools beforehand, and ensure it is compatible with our system, however the added complexity gives great flexibility, and allows to use anything that will run within a non-privileged Docker

container.

Preliminaries

0.1 Lumen

Lumen is a lightweight php "micro-framework" built upon the same foundation as it older brother Laravel. It is less focused on user-facing application than building fast micro-services and is actually one of the fastest micro-frameworks available. Much of the flexibility one might know from Laravel have been removed with exception of the most fundamental components such as Eloquent, caching, routing and validation. All in the name of speed[1].

0.2 Flask

Flask is another micro-framwork, but this time for Python. It differs from Lumen by not having any database abstraction layer, validation, or any other components where pre-existing third-party libraries providing common functions. This means if any of these things is required one is required to find and install extensions[2].

0.3 Docker

Docker is an open source containerization platform. Docker lets developers package and deploy application into containers. Containerization is a technique in which a application, much like a VM, has a isolated address space and a isolated file and network space. Where it differs from VM's is that it does not simulate an entire machine but rather just the OS while still using the host's kernel. Each container then shares OS resources such as the host OS kernel and, usually, the binaries and libraries. This makes the containers much more lightweight compared to VM images and are only a couple of megabytes in size compared to the sometimes couple of GB of a VM. Not only are the less processing power wasted through overhead, but also make the containers each to share and distribute. This is made even better by dockers container image registry called Docker Hub. Here one can manage, store and share docker images[3].

These attributes makes containers portable and great tool for developing, testing and

deploying your code to a consistent operating environment.

Docker client is the primary way users interact with with docker, but in fact this is just a facade for the Docker Daemon, which actually carries out the communication.

0.4 Kubernetes

Kubernetes is an open source system for automating deployment, scaling, and management of containerized applications. It groups up sets of containers, such as docker containers, into logical units for easy management and discovery. Here it orchestrates computing, storage and networking so no direct manual container orchestration is needed in a cluster[4].

Some of the most important features includes: Seamless horizontal scaling to handle surges in workload and down scaling when the workload deflate to save money.
Self healing to automatically instantiate a new healthy replica when a container goes down due to fatal error.
Automated rollouts and rollbacks with no downtime as the old replica is kept alive and in operation until the new version is ready to take over.
Service discovery and load balancing to handle all the communication to and from the numerous micro-services that your application consists of.

0.5 Amazon S3 Bucket

Amazon Simple Storage Service (Amazon S3) is an storage services(bucket) that provide top of the line scalability and data availability. This is achieved by storing copies of all data across multiple systems. The Amazon S3 bucket is an easy way to add storage capacity to one or more of your serves as multiple systems can mount to the same bucket. This allows for the bucket to be used as a file exchange between servers.[5]

0.6 Jenkins

Jenkins is a open-source automation tool built for continuous integration and continuous deployment. It is used to build, test your software and deepening on the specif pipeline setup, deploy your code. Jenkins achieves all this with the help of numerous plugins. Jenkins is run on ones own servers which added a bit of complexity compared

to similar tools that run on the tool providers servers.

Technical Description

0.7 Architecture

Our project is split into five services. Frontend-service, written in HTML, Javascript, and CSS. Course-management-service, written in Lumen. Admin-service, written in Lumen. Upload-service, written in Lumen. Container-service, written in Python and Shell.

What all the services have in common is that they communicate over HTTP through APIs. APIs makes it very easy to access information and functionality from each service. Our frontend is, as mentioned above, written in plain HTML with some CSS for styling, and Javascript to communicate with the APIs to fetch data.

All of the services are connected directly to the same database, the reasoning for this is explained later in this section.

The reasoning for writing the backend in Lumen is that two of our members already had experience with this. It also makes it very easy to implement security; we went with JWT, which requires each call to have a header with a Bearer token, which is generated by our login-service. This is a very secure way to implement authorization. It is also an easy way to keep persistent state, meaning that the user is able to be kept logged in, even though we are simply using plain HTML. Normally, with a monolithic architecture, a cookie or session is kept in the browser, communication directly with the backend.

All services would need to be accessible and should be able to scale automatically if traffic should suddenly increase a lot. To combat this, we went with a Kubernetes cluster. This allowed us to initialize pods, which are instances of services. If the login-service all of a sudden would be maxed out, we want to be able to replicate a new instance quickly to serve the users. This is possible with Kubernetes. Another great reason for using Kubernetes is the ease of deploying new code to the pods. This will be explained more in details in the subsection, Continuous Integration and Deployment.

We still needed a host for both the Kubernetes cluster as well as the database. For this we went with Digital Ocean. We could easily have chosen any host, as most companies offer the same service at almost the same pricing. Digital Ocean offer a very easy option to create a Kubernetes cluster as well as a dedicated database. The issue

with Kubernetes is that we need a dedicated database, which should be accessible and consistent, and Digital Ocean allows for this, out of the box. They offer backups of the system and close to no downtime.

Regarding deployability, with Lumen the database is setup using what is called migration files. Using migration files, we are able to precode the structure of the database. We are able to make modifications to the database, even in production. It is not safe to change columns, but we are always able to add columns and tables without risking losing data. For the system to work properly, we need some default roles setup. This is done through seeders. Seeders are also included in the Lumen framework. Seeders allow us to create database entries when the database is created. In our case, we create the three user types when the database is first created.

Dockerfiles are used to create an image with the relevant dependencies for our services to work. Each service might have different dependencies, the Dockerfile allow us to define it in one place and ensure that the service will run on any container when running the image. This makes deployment of the services very easy.

In our Login-service we keep an ingress.yaml file. This file is in charge of setting up the SSL certificates of all the services. For security reasons, we want to have all communication pass through HTTPS to encrypt data and ensure that the data is untampered and unsniffed. We put this file in the login-service since the system will not run correctly without this service being made at least once.

Each service has a deployment.yaml file. This file sets the name of our pod in the Kubernetes cluster and exposes the correct ports to the Ingress server.

Mentioning Ingress, we use Ingress Nginx to expose our application to the valid ports, which makes them accessible from outside the cluster itself.

0.8 Modifiability

For a micro service an important part is the modifiability. Meaning the system can adapt to change. For the system we build we will first take a look over our choices making the system easy modifiable and the choices that makes it less modifiable. Every service we made is based on a set of API calls for communication. So every call can be changed and modified later on, as long as the protocols for the API calls are kept.

Adding new services is also very easy, this is mostly based on the the communication directed to the front end. Since all calls are made from the front end. modifying the communication as mentioned with the API calls, can make the system a bit harder to modify though, since all protocols for the communication must be made before hand or modification must be made on more than one end. Still its possible, but becomes more of a challenge if specific communication is based on many services. This is one of the reason most of the communication is from front end to the services, or through data in the database. Also the database is hard to modify, due to all the services using the same database. This creates the problem of modifying the database will have an impact on many services as well. Every service could be given a database to solve the problem, but this also cost on the consistency of the system. Hence, we chose the one central database for consistency. This choice could be redundant, had we made a system to check for consistency, but would also have been a larger project than for this course. With the database being in production, it is no longer smart to change to database; with that being said, using migration files we still have the ability to add columns and tables to the database structure. This can be done without the risk of losing data or/and up-time. The system is modifiable and can be changed, though some parts should be more care full planned if modification is needed.

0.9 Continuous Integration and Deployment

Normally when we are talking about modifiable, we might also want to discuss how we deliver the changes to the services. We chose Jenkins as our tool to solve this issue. There are many tools to choose from, but we found that Jenkins would make the most sense as one of our group members already had experience with the tool, how to set it up, and use it. We did not use the plain, old, version of Jenkins, but the plugin Blue Ocean, which allows for a programmable pipeline. Using Jenkinsfiles, we were able to define stages. In each of these stages, we solved a single problem. The better the stages, the easier it is to debug when a pipeline pass-through would fail. We went with the stages; Build Test, Push, and Deploy.

0.9.1 Build & Test

In this phase we built the Docker file. How the Docker file is build, is defined in each of the micro services.
For the testing, it is automatically done with the Docker build. For our APIs, use did

unit testing using PHPUnit. The tests are setup in each project and the build will fail, if one of the tests are not passed.

0.9.2 Push

When the Docker file is built, we move into the phase of the pipeline, where the code is being pushed to Docker Hub. For this project, we simply chose to go with a public version, even though it would make our product vulnerable for unauthorized access as one could simply spin up a version of our image and examine it for mistakes. It would also give full access to our API structure, which we might not want to be public. With this being said, moving it to a private version would simply add additional billings and we did not want this. For every commit, we pushed both a version as well as a latest to Docker Hub.

0.9.3 Deploy

For the deployment, we simply need to push the new image to our Kubernetes cluster. The way we have setup our services in the cluster, we simply need to apply our deployment.yaml and ingress.yaml(only for our login service). The last step to finish the pipeline is to set the new image of our service. When this is done, the cluster automatically terminates the old pod and spins up a new one, with the new image.

This concludes our pipeline and how we do continuous integration and deployment. We found this very easy and quick to use when developing the project and is also very fast and easy for new developers to take over when the initial setup is finished, which is done automatically with the deployment.yaml file.

0.10 Solution Testing

0.10.1 Executing tests

When the teacher creates an assignment, they upload a docker image containing the environment and test tools for the students solution. Other than a few specific requirements, the teacher can use any tool and environment they please. The students upload their solution as a zip archive, which will be extracted and copied to a fixed location within the image before a container is spawned from it. The results of the test are to

be written to a specific file within the container, which then gets copied out when the container exits. The content is then stored as the result, and is available to the student and teacher.

0.10.2 Shared storage

When the docker images, and student solutions, are uploaded, they are saved using shared storage. Both the upload and testing parts of the system mounts the storage as a virtual drive, so they are completely agnostic to the technology used. The system currently uses AWS S3 for file storage. We have chosen this solution, because of the speed, scalability, and availability of AWS, and because it is fairly cheap. S3 is integrated into the system by running a daemonset in kubernetes, which mounts the S3 bucket as a FUSE filesystem on the host server. Containers which needs access to the bucket is then configured though Kubernetes to bind mount the FUSE mount point on host with a directory within the container. This solution was chosen because it disconnects the services from the shared storage solution, and because it was quick to implement.

0.10.3 Initializing tests

Upon upload of an assignment solution, a service is called over HTTP with parameters containing the solution ID, location of docker image and solution archive within the shared storage, and optionally a timeout for how long the test are allowed to run for before aborting (Default 15 minutes). The request is received though a Flask server, specifically Green Unicorn, which was chosen because it did not require configuration. When a request is received, it checks that the parameters are OK, and that it can connect to the database. It then starts a new thread, as to not block the server, and returns OK to the caller. The new thread sets the "status" field in the database for the assignment to "testing started", then executes a shell script that runs the container, and copies the result out of the container upon completion. If The timeout is reached, the script returns a non-zero value, and the python thread sets the status to "timed out" and exits. Otherwise it finds the file with the result of the test, and copies it into the database, as well as setting the status field to "Completed". The result in the DB is stored as "LONG TEXT" which puts an upper limit on the outputs size to 4GB, which we have agreed is sufficient.

0.10.4 Arbitrary code execution

Allowing the end user to execute code on our servers presents two major challenges. Creating the environment for the code, and ensuring the system is safe and secure against the code it runs. This is mostly solved by using Docker. The environment for the code execution can be given as a Docker image, allowing almost anything to run. Using Docker also creates separation between the code and host environment for additional safety. Using Docker creates a new problem however. Kubernetes manages pods, which are staticly defined, but we want to run arbitrary images defined at run-time. This is solved by bind mounting the docker socket of the host, to the containers responsible for testing the students solutions. This way, Docker commands from inside the container is communicating with the Docker daemon on the host, which allows it to spawn and work with sibling containers, almost exactly as if it was running on the host. This approach unfortunately this means Kubernetes is not directly aware of these containers.

Related Works and Discussion

0.11 Micro Services

Using Micro Services is a way to decouple the code, so a bug in the admin service should not propagate to the other services. We can better scale the system because we can scale on individual services. It allows us to write the backend services in different languages, because not every language is equally good at something, and some people might not be experienced with a certain language. It is convenient to modify and update the code infrastructure along with accelerating delivery speed by streamlining complex and manual deployment and integration process. Furthermore, it reduces failures which can be caused due to the difference in the development and production environment. However, monolith architecture cannot provide these facilities seamlessly leading to counter-productiveness in business and development.

0.12 Microservices database patterns

All the services share a common database. when a service needs data, it can check itself, instead of having to call the other services. But having a single database could cause bottleneck issues, but it is unlikely, that we will be hit with a high amount of traffic. One serious problem is that if a service causes issues in the database, then that issue will affect the other services. An alternative is for the microservices to have their dedicated databases, then if a service needs data it has to ask the other services [6]. That can be a bit more tricky to develop, but it is a good way to decouple the code.

0.13 Authentication

We have stateless user authentication with JWT token [7]. That has the advantage that the server does not keep track of user sessions, and it works even if cookies are disabled. There are several issues, like that the tokens lifetime is only half-hour long, which is too little, and that means the user is auto logged out after 30 min. Tokens stored are vulnerable to XSS attacks, and if the attacker gets the token, the attacker can act as if he was the user, and thereby get confidential data. Instead, we could have used user sessions. Where we stored a session ID on a cookie. Each authentication request is sent with the cookie. The server then compares the session ID on the cookie with the session ID stored in memory. If it is correct, then we go on to fulfill their request. The

advantage is that the user is not suddenly kicked out of the website, but at the cost, that session stored in memory does not scale well.

0.14 Github

We used Github and git because nearly everyone uses that technology, and we have experience using it. We could also have used Gitlab, which has the same functionality as Github [8] but comes with a built-in CI/CD framework, but Github **Actions** does to a certain degree rival that feature. Gitlab can be used for the entirety of the software life cycle.

0.15 Git workflow

We used centralized workflow [9], while that is not ideal, but because we made sure, to work on different things, such that we would not have any merge errors We only have one branch, the release branch. During development that is not a problem, but if it was released, and we had thousands of users, then it would be better if we had a development branch, which updated when we push new code. And we could properly verify that everything works before we before push it onto the release branch.

0.16 Continuous integration

We used Jenkins for continuous integration because it is free and highly configurable. Instead, we could have used Github actions, which is built-in with Github, and we are using Github, and it is quite easy to set up, but the cost of using it, is too high for us We only have one branch, the release branch. During development that is not a problem, but if it was released, and we had thousands of users, then it would be better if we had a development branch, which updated when we push new code.

0.17 Database

When it comes to relational databases, there are two major options, MySQL and PostgresSQL [10]. MySQL is a purely relational database, while PostgresSQL object-relational database. Both have cloud vendors. PostgresSQL is better at handling concurrency, and it is highly extensible. PostgresSQL is not as popular as MySQL. When

it is read request, then MySQL is faster. Each time there is a new database connection PostgresSQL creates a new process, which size is up 10 MB.

0.18 Deployment

We choose to use Digital ocean to deploy and host our code [11]. DigitalOcean is simple to use. It is easy to set-up and does not cost a lot of money. It allows us to spin up a server in a fraction of time compared to the other cloud service platforms i.e, you pay as you go. It is good for small applications, such as ours. It works well with MySQL instances, which we use based on what you pay for. An alternative to Digital ocean would have been AWS CodeDeploy. It could deploy our code in its instance and auto-scale when necessary, then Kubernetes would not be needed. AWS is a behemoth and offers many different services. AWS works well with large scale application.

0.19 Containerization

We used Docker manually to sandbox user code and separate it from the rest of the system for safety and security reasons. Bad or malicious code could otherwise mess up the services and steal secrets from the system. Kubernetes does not allow for running arbitrary containers, because the pods that it manages are statically defined. Docker allows executing code in the environment that is defined by a user (anyone really). A teacher can just create an image which is the environment to run the users' (students) code and then the image is handed to the services so that we can execute code in that environment without knowing anything about it.

Kubernetes sets up the infrastructure and orchestration facilities to run the docker images though it supports other containerization technologies.

We containerize our code so that anyone can run it without knowing about the environment that it needs. Kubernetes provides a programmable infrastructure to run images, configure load balancers, access levels, etc. Kubernetes constantly checks the state of your deployment according to the YAML definition we use. So if a Docker container goes down, Kubernetes will spin up a new one automatically. One no longer has to go to each server where the container failed to start it up again; the orchestration will take care of that.[12]

An alternative solution could be spinning up Virtual Machine (VM) instances and run a shell script to configure the environment. However, it will not be scalable because

every time something changes, one has to take a new snapshot. And then one has to somehow organize all the different versions of those VM snapshots and still need to deploy changes in code and any dependencies to other environments respectively. [12]

The above-mentioned clauses help us to believe in the containerization process of application as we could easily do several deployments a day that take around five minutes.[12]

On the other hand, Podman could have been used instead of Docker and Docker Swarm instead of Kubernetes. But we chose to stick with Docker and Kubernetes as Docker officially supports Kubernetes. Both of them use declarative languages to define how they will run and orchestrate an app.[12] Also, they are very popular among developers and have helping blogs tutorial and suggestion all over the internet.

0.20 Server

We have used NGINX For serving the static files such as HTML, CSS, and javascript files of the front-end service without using PHP.[13] Although we could have thought of using Apache2 Webserver, NGINX serves static content much faster than Apache at high concurrency levels[14].

Also, it allows URI to interpret request which is convenient for us and better security with smaller codebase like our project[13]

We have used an official open-source Ingress Controller from NGINX Inc (now owned by F5) known as NGINX Ingress Controller. Besides simple HTTP/S routing and SSL termination use case, it has been used to expose services to external requests for managing traffic of API requests also defining and publishing the API using it. It is our all-in-one Load Balancer, Cache, API Gateway for the services inside Kubernetes. It was suitable for us because we could reuse the configuration for each service and write it simple way[15]

Alternatively, we could have used ingress-nginx which is the only open-source Ingress Controller maintained by the Kubernetes team, built on top of NGINX reverse proxy. However, default options for ingress-nginx may have performance issues at scale.[16]

As we were optimistic with NGINX, it was comfortable to use the official Ingress Controller from NGINX in Kubernetes.[16]

0.21 About Consistency, Availability and Partition-tolerance

CAP theorem applies to distributed applications, where many instances of applications are deployed. In simple words, CAP theorem states that "All systems cannot **guarantee** Consistency, Availability and Partition-Tolerance **at any given point in time**"

As per the Theorem, any distributed system can guarantee, any two out of three at all given points of time. It means that if we want our applications to be partition tolerant, then we need to compromise either Consistency or Availability. The choice here again depends on the requirement at hand.

We have used MySQL which guarantees consistency and availability making it prone to a single point of failure but an ACID complaint. [17] We could have used MongoDB or Cassandra to lean into partition-tolerance.

Since Kubernetes is optimized for running resilient, highly available workloads, it chooses Availability. State updates are propagated over time in what is known as eventual consistency.[18] On the other hand, NGINX gives priority to high availability.

To summarize, any distributed system can guarantee either Consistency or Availability. If our use case need both Consistency and Availability, then that system cannot be distributed as it will not be partition-tolerant.[17]

References

[1] https://lumen.laravel.com/.

[2] https://web.archive.org/web/20171117015927/http://flask.pocoo.org/docs/0.10/foreword.

[3] https://www.docker.com/resources/what-container.

[4] https://kubernetes.io/.

[5] https://aws.amazon.com/s3/.

[6] https://relevant.software/blog/microservices-database-management/.

[7] https://medium.com/@sherryhsu/session-vs-token-based-authentication-11a6c5ac45e4k/.

[8] https://usersnap.com/blog/gitlab-github/.

[9] https://medium.com/jamalsoliva/git-workflows-centralized-featured-branch-and-git-flow-52ebb4efd3db/.

[10] https://developer.okta.com/blog/2019/07/19/mysql-vs-postgres/.

[11] https://www.guru99.com/digitalocean-vs-aws.html.

[12] https://stackify.com/kubernetes-docker-deployments/.

[13] https://serverguy.com/comparison/apache-vs-nginx/.

[14] https://www.hostingadvice.com/how-to/nginx-vs-apache/.

[15] https://www.nginx.com/products/nginx-ingress-controller/.

[16] https://medium.com/swlh/kubernetes-ingress-controller-overview-81abbaca19ec.

[17] https://www.linkedin.com/pulse/cap-theorem-explained-pradeep-kumar.

[18] https://downey.io/blog/desired-state-vs-actual-state-in-kubernetes/.