Technical University of Denmark

Booking System

Group 0
3th December 2021

s195469
Martin
Maartensson
Gitlab:Martin
Mårtensson

s141479
Anthony
Haidari
Gitlab:Anthai87

s176492
Mohamad
Abdulfatah
Ashmar
Gitlab:M0-AR

s174548
Jens
Jensen
Gitlab:Jens
Jensen

62582 Complex Systems and DevOps Fall 21

Contents

Introduction

Everything as a service is the biggest trend of the past decade. This paper will provide documentation on how bookUs is being designed, developed, published and maintained. It also provides an overview of stakeholders and requirements for the project.

BookUs is a software as a service, which will help any service provider, who has a service that can be booked by it's customers. For this iteration of the product we have decided to implement use cases for restaurant table bookings. In the future this is to be scaled so that the service providers can create the services themselves and then users can book them on a different platform. Admin and user are the main role of this system.

1 Problem and Vision Statement

BookUs want to be the Netflix of anything booking. The biggest software as a service for every service provider who wants to provide a booking for their customers. Every service provider who provides anything that can be booked should know our name and use our software to create their services and let their customers book them.

Right now we observe a movement with all kind of software services, from tax management services to vacation planners. And every restaurant or office leasing company has their own booking service set up, and it's all different from each other. This is why we want to create a concept that generalizes the booking experience for both provider and customer and makes it easy to be both the booker and the bookie. A service provider should be able to customize and create their own booking services in the admin part of the application while a customer should be able to search, select and book a service from their mobile or PC.

This project will only deal with the admin part of the project and how it's setup in terms of our development pipeline. We will also explain a bit about our thoughts on the full development stack, frontend to backend.

2 Epics and User Stories

2.0.1 Epic sign up and sign in

- As a manager I want a sign up page so that I can create a user in the system
- As a manager I want a login page so that I can login and use the service

2.0.2 Epic Dashboard

- As a manager I want to create a service, so that I can provide my services and a user can book one of it.
- As a manager I want a list to view bookings, so that I can know who register to my service.(To see a list of users, who book manager's service)
- As a manager I want an ability to manage bookings
- As a manager I want a dashboard to manage bookable

2.0.3 OUT OF SCOPE

- Admin who manage users in the system and everything else as well (not created in this semester)
- Users who dont manage anything except their own bookings and information (app course)

2.1 Epic Manage bookings with crud

- As a manager I want a button to delete a booking

Product overview

Need	Feature according to User Stories	Priority
Sign Up and Sign In	Create user in the system	5
	Log in and use the service	5
Dashboard	Create a service	4
	List view of bookings	4
	Manage bookings	4
	Manage bookable	4
	Delete bookings	4
Manage users	Admin manage users in the system	-

Table 1: Features estimation

Sprint 2

1 Prepare Sprint Backlog

User Story - Create a service

As a manager I want to create a service, so that I can provide my services.

Use case - Create a service

Primary actor: Manager
Secondary actor: None
Description: A manager can create a new service.
Precondition: Manager should be identified and authenticated.
Basic flow

1. Manager clicks on button to create a new service.
2. BookUs provides a field to enter the name of the service.
3. Manager enters the name of service.
4. Manager clicks on plus button to provide the attributes of the service.
5. BookUs provides two fields for key and value, where key could be text or number.
6. Manager enters the name and value of attribute.

   Manager repeats steps 4-6 until attributes of service done.

Alternate flow

From setp 3 of basic flow:

1. BookUs finds another service has same name.
2. BookUs displays a message that the service's name already exist.

Postcondition: Service is created.

2 Devops Pipeline

The group went through a lot of different options for deployment stategies which will be explained in the next section.

2.1 The choice of deployment strategy

In this section we will explain a bit about the different strategies we have tried for deployment and why we have chosen the one we are using over the others.

2.1.1 Heroku

This option was very easy to setup, however it didn't provide alot of flexibility like eg. connecting to a database on the local network.

2.1.2 Kubernetes

Kubernetes seemed to be the best option since it provides a lot of flexibility and control - Also it would be possible to use traefik as an ingress controller and dynamically create domain names on the fly. However it was very difficult to setup and the group didn't manage to make a working system in time for the hand in.

2.1.3 Caprover

The group tried to set up caprover which was surprisingly easy and we also successfully created domains on the fly by using a wildcard DNS record. The only problem with caprover was that it was too high level and when some errors occured we could not figure out why they occured.

2.1.4 Self hosted docker

In the end the group decided to do a self hosted (linux) solution with NGINX, Docker and GitlabRunners. as seen in figure 8 on page 14. On gitlab we have 3 repositories:

- Frontend
- Backend
- AuthenticationService

Each repository has a Dockerfile and a .gitlab-ci.yml. The .gitlab-ci.yml file describes the development pipeline which is divided into different stages like :

- Build
- Test
- Release
- Deploy

Each stage has one or more jobs eg. test can have a job for unit tests, a job for integration tests and or a job to check the code coverage. At the moment each stage only includes one job. Each stage runs in its own docker container (gennerated by the Gitlab-Runner) and every time a stage is complete the container is removed. If a stage is failing then the pipeline will end and the following stages will not begin. All the console output from runners is saved as logs on the gitlab repositories under pipelines, and is inspected if the build fails. This solution seems to be the best of all the above.

2.2 The stages of the pipeline

In this section we will explain how the flow of the pipeline is set up and how each stage is implemented. The stages in the CI/CD and deployment is very similar for both the frontend and backend application, so the following explanation is for all our deployments - both Frontend, Backend and OAuthServer

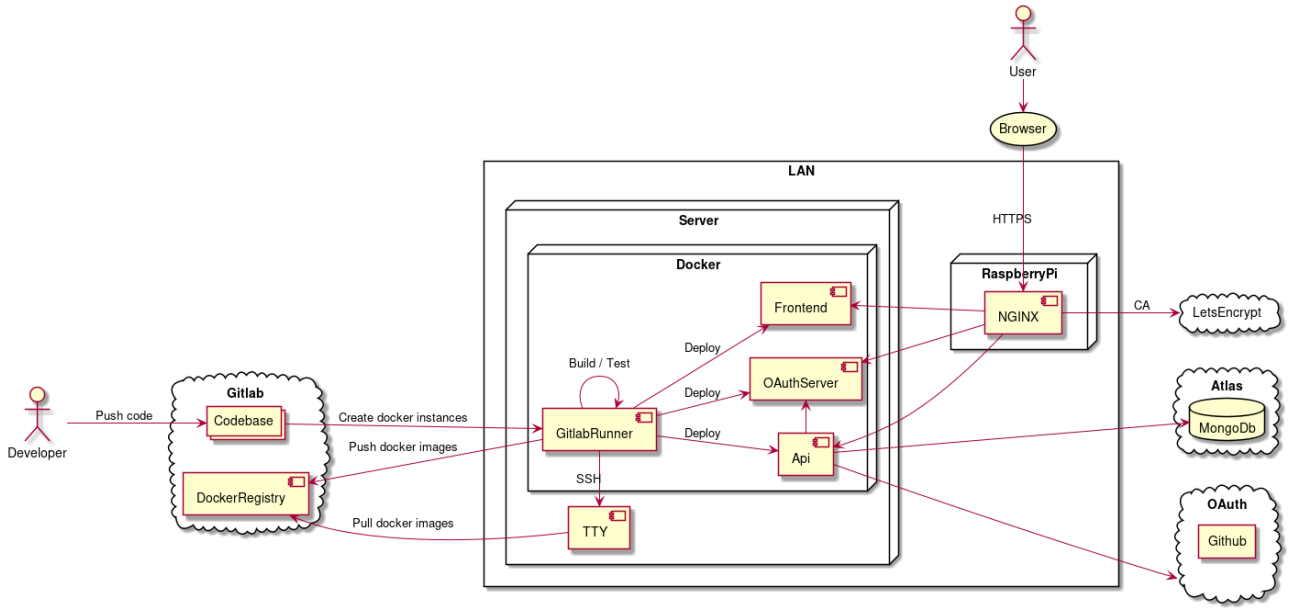

Figure 3: Deployment diagram over systemet

2.2.1 Build Stage

This is the simplest stage where the application is only beeing build - nothing more.

2.2.2 Test Stage

The test stage will simply run the tests on the application, if a test will fail it will print it to the console and terminate as failed.

2.2.3 Release Stage

The release stage is where the Dockerfile is used for the first time. Here we use a docker container in a runner to build a docker image from the Dockerfile, if the docker image is successfully built it will be pushed to the DockerRegistry on Gitlab and the job will complete succesfully.

2.2.4 Deploy Stage

This stage is setup with a rule so that it will only run on the main branch. This makes it possible to run tests and so on, on other branches without disturbing the production build with features that should not be deployed. The responsibility for this stage is very simple - deploy the latest release. An alpine container is used to SSH into the production server - from here the server will remove the old deployed docker container and replace it with the latest version, fetched from the gitlab DockerRegistry.

2.3 The components in the deployment

In this section we will talk a bit about the components in the deployment environment.

2.3.1 Gitlab

Gitlab works very similarly to Github, however there are some features in gitlab that makes it very easy for us to tailor our deployment pipeline how we want it. This is for instance a docker registry in gitlab that is private but we can use built in environment variables to access the registry from within a gitlab-runner or even a remote server we connect to in a CI/CD job.[1]. Another thing is the gitlab runners that allow us to use our own resources for CI/CD which gives us more power and speed than if we use the very limited free runners avaiable to us in Github or even Gitlab. Gitlab also gennerates a nice diagram (figure 9) showing the stages of the deployment pipeline and each job under here. It's not super important - however it gives some extra overview which is nice to have when something fails.

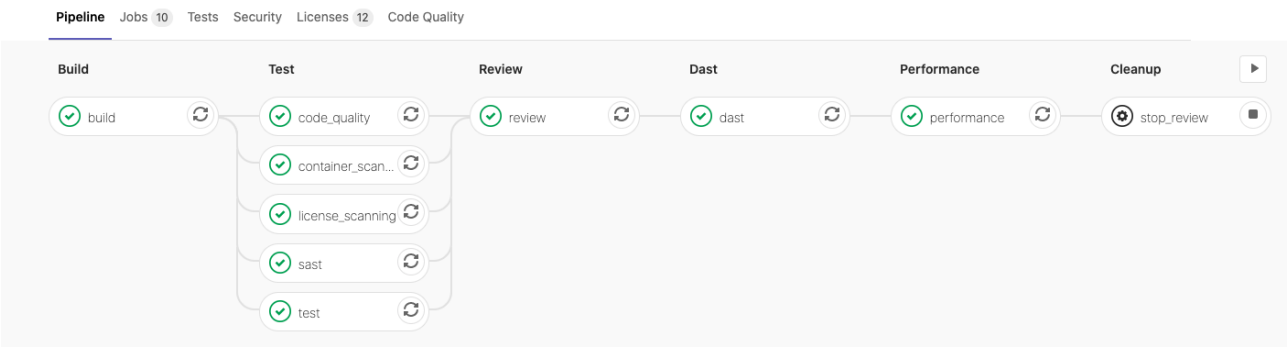

Figure 4: Pipeline diagram example

2.4 Gitlab Runner

A gitlab runner is just a docker container running on the groups server, the runner has access to create other docker containers on the same machine it is deployed on, and has a reverse connection to Gitlab, which allows us to trigger the gitlab runner to run jobs for us every time we start a CI/CD pipeline.

2.5 Self hosted server

The self hosted server is just a physical machine with an installation of ubuntu 20.04 with docker and SSH. The machine is responsible for holding the gitlab-runner and the deployed applications. The machine itself is behind a NAT and a firewall so it can not be accessed from the outside world. To access the deployed applications we are using NGINX

2.6 NGINX

NGINX is a reverse proxy application which is used to serve the different applications on the ubuntu server behind the firewall and NAT. The NGINX application is serving the ports 80 and 443 and is managing the logic of routing when someone tries to access a specific domain addressed to the public ip of the system. If someone connects to port 80 they will be redirected to port 443 which is where all the data is TLS encrypted. NGINX also manages the TLS encryption so all data going through NGINX will be encrypted, the certificates are generated with certbot which is a tool created for and by LetsEncrypt.

[1] We do that in each deployment job in our .gitlab-ci.yml file

2.7 LetsEncrypt

LetsEncrypt is the Certificate Authority for our TLS certificates on the server. Each domain has its own certificate, and when someone tried to connect to the domain then the browser will accept the domain because it is authorized by LetsEncrypt which is approved by most browsers used today.

2.8 Future releases and continuous development

For the future we want to implement a service template for service providers so they can customize and create their own services. This would propably mean a greater overhaul of the react application, but our backend is pretty scale able and can handle this kind of integration.

3 Frontend

3.1 React

The frontend part of the BookUs application is made in react as a SPA – a thick client, meaning we only transfer data to the client but routing between different sites and showing components is all handled on the client side.

3.1.1 Routing

For routing between the pages in our react application we have used the react-router-dom, which takes in a path as input and then links to a specific component. In addition to this we run a go file that takes in a url as input and then forwards it to the react application. The go file gets the request from Nginx.

3.1.2 MVW

The idea of a Model View Whatever is that it keeps track on all our views and update them with the correct data. It takes models, like a user model, and then binds them in a template and shows it as views. The Model Views job is to keep track of the views and a model view like MVVM keeps track of both models and views and can subscribe to changes on both entities. In our project we use a classic model view controller because our controller gets an input and update a model which is then used in a template to create a view for the user.

3.1.3 Authentication with cookies

For authentication a state will be maintained in the front end, this state is provided by a AuthenticationProvider which is functioning kind of like a class in java. The AuthenticationProvider has a method to check weather the backends current cookie is valid without even knowing how the cookie looks like, by calling and endpoint in the API and checking the response status code. If the code is 401 then the user is not authenticated and if it is 201 then he is. The AuthenticationProvider also has another function which is used to authorize the user with a post request.

The state provided by the AuthorizationProvider is used different places in the application to show or hide different components and fields.

3.2 Directory Structure

There are different way to organize our React application, because of the freedom that react gives. In BookUs application we decide to structure our react application by considering to make the system scalable. At the beginning we assume our application will use the following structure:

- Assets: To have static files like images, logo, etc.
- Components: Reusable components
- Services: JavaScript modules
- Store: Redux or Mobx
- Utils: To have helpers methods or constants variables.
- Views: Which has pages

We can take a dashboard as an example. It was not necessary to use Redux or Mobx, because we are just displaying the API's data in one component, so for this purpose we make JavaScript modules, which has Axios to fetch data and store locally in browser local-store.

4 Backend

4.1 Golang

We used Golang to write our backend and implement a RESTful API.

4.1.1 Architecture

When we make the first backend, we try to implement The Clean Architecture by Robert C. Martin (Uncle Bob). The idea is to divide the software into layers. Each layer has one business rules and interface. This class diagram an overview about Dashboard's service work and for every interface there is CRUD opreation.

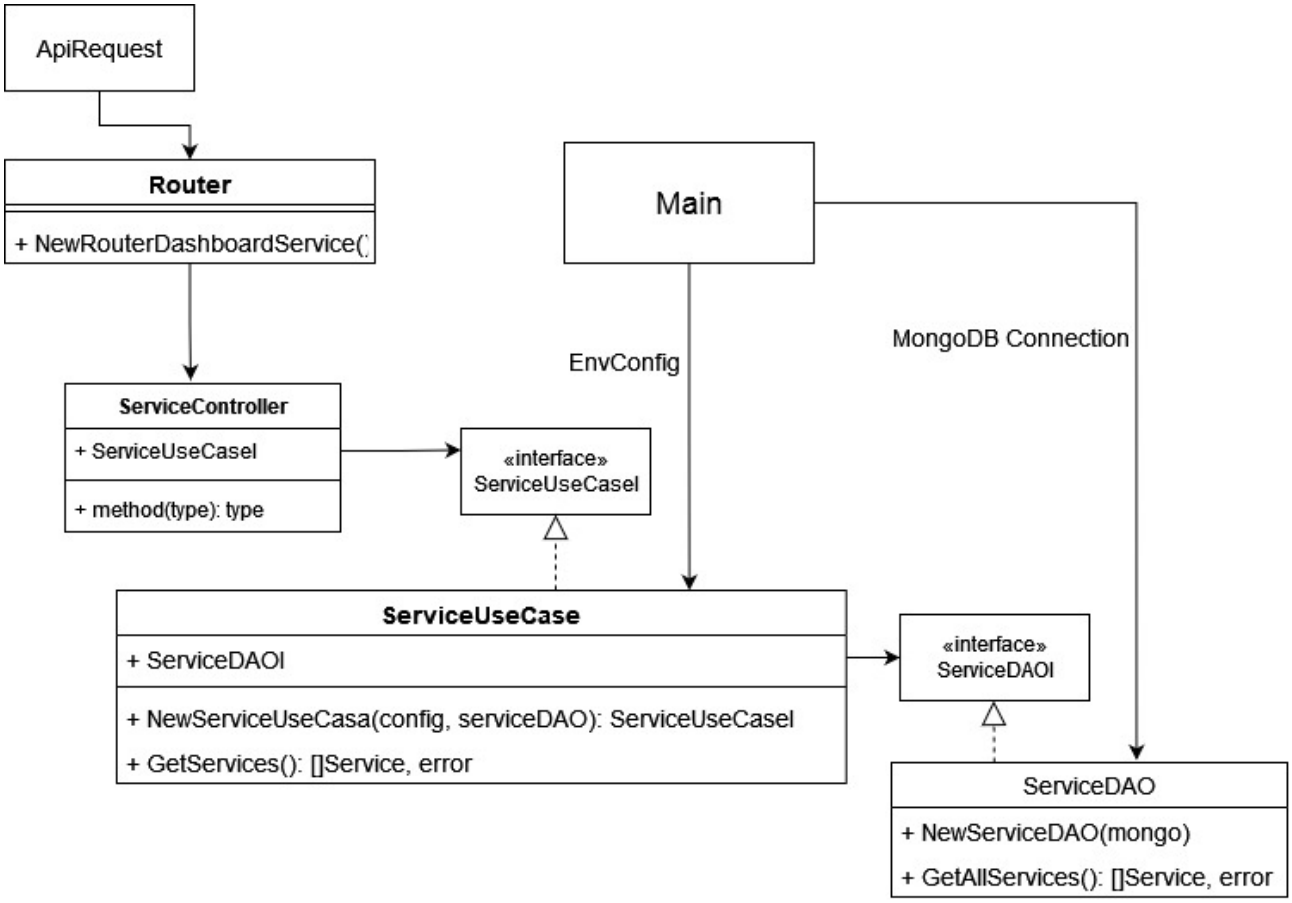

Figure 5: Old Backend Class Diagram

4.2 ORM

At first we used Mongodbs Atlas cloud based solution for storing all the data from the application. We later in the process wanted to create a local mongodb as a docker container on the server. However we changed approach and instead designed an ORM solution to store objects like users and bookings in a file on our server. Golang has a package called Gorm, that makes it possible to map objects and designate keys and constrains to the object fields and then save them in a file. Making it a locale database on the server makes it easier to test.

4.3 OAuth and JWT

The api is setup so it functions as an Oauth client, for github, facebook and google. On top of that the API functions as OAuth server as well, so that other oauth clients can use this API as an authentication provider.

4.3.1 Oauth Client

The oauth client is set up with a configuration register containing the configurations for all the used oauth providers, then when a user chooses to login with eg. facebook the oauth flow will begin and the user is redirected to facebook. If the user is then succesfully authorized he will we redirected one more time to the api's oauth callback endpoint for the service (in this case facebook) The callback endpoint handler will then get a access token which can be used to access the user data from facebook on a specific URI which is found in the facebook related oauth configuration register. The server will then use the token to make a http request to that endpoint and fetch the user information which always include a unique identification value called id. Then the handler will check the database through the repository and create a new user in case no user matches the identification. At last the oauth server library will be used to generate a token pair which will then be packed into a cookie and the request will return 200 OK.

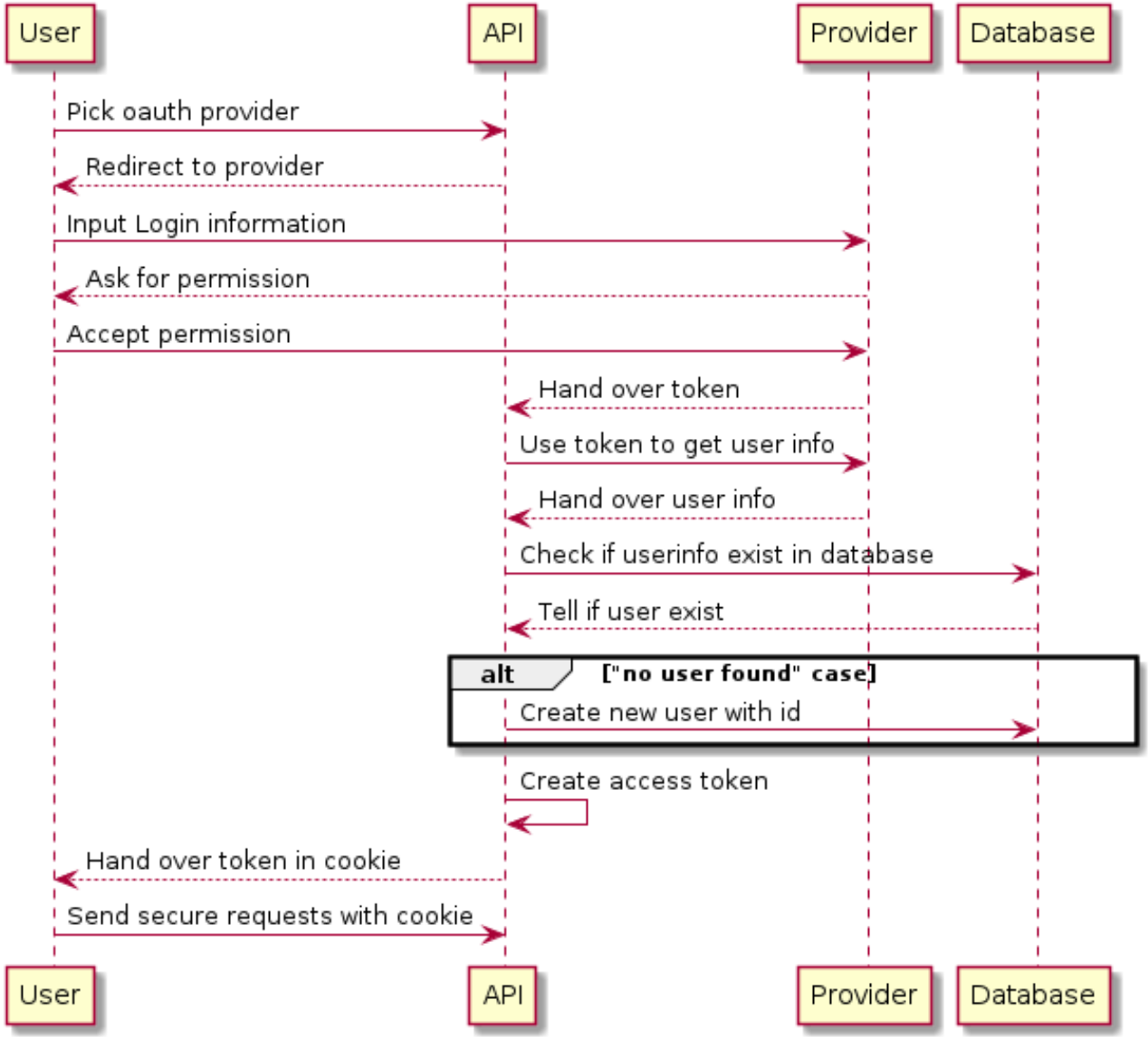

Figure 6: Oauth flow

4.3.2 Oauth Server

The API also has Oauth server functionality which makes it possible for other applications to register their app and use BookUs for authentication.

At the moment only one client is hardcoded into the API with the client id 222222 and client secret 22222222, however this could be setup the same way as a user registration system so that new

clients dynamically could be created.

The Oauth server uses the same token store as the rest of the API and can be used to reach all the same resources on the API as a user would after a normal login post request.

4.4 Frameworks

Through the process of developing the backend, multiple different frameworks has been explored, which include

- Echo
- Zap
- Logrus
- Viber
- Fiber
- Gorilla Mux

In this section we will focus on the last two frameworks in the list, which has taught us that simplicity isnt always the best solution.

Fiber seemed like a easy solution since it has a simple documentaion and comes pre packaged with a lot of middlewares and examples to work with. However it turned out that Fiber was very strict with their simplicity and they had a bug in their CORS middleware which made it almost impossible to customize the Allow-Origin header on a global scale. It was also very difficult to handle lower level http stuff, which became a problem when trying to integrate with the Oauth server library.

In the last days of the project the whole presentation layer of the application was replaced with the core http library in go and the Gorilla Mux library which helped with simple functionality like routing, method validation and extraction of variables from the url query parameters.

5 Cloud

We have two Ubuntu Servers up and running, one with docker on it to run Golang backend project and the other without docker, but with let's encrypt certificate to run React frontend project.

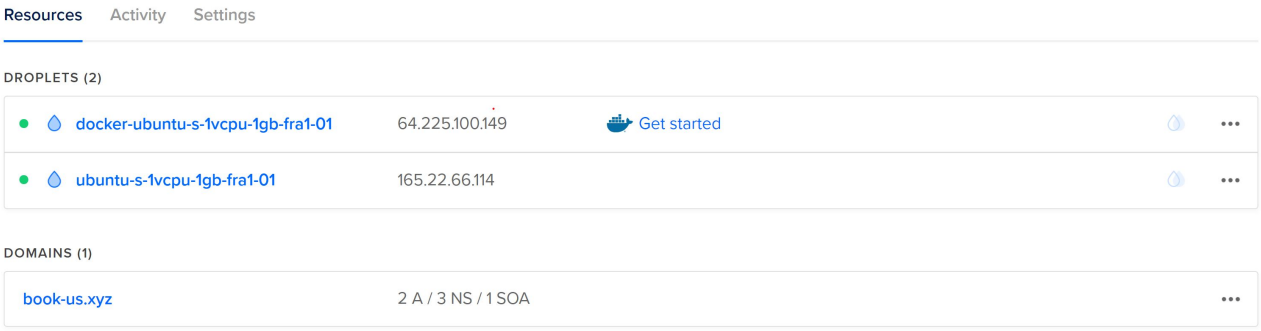

Figure 7: Digital Ocean Ubuntu Servers

An overview of one server status which provides monitoring of Bandwidth graph, CPU Usage, and Disk I/O.

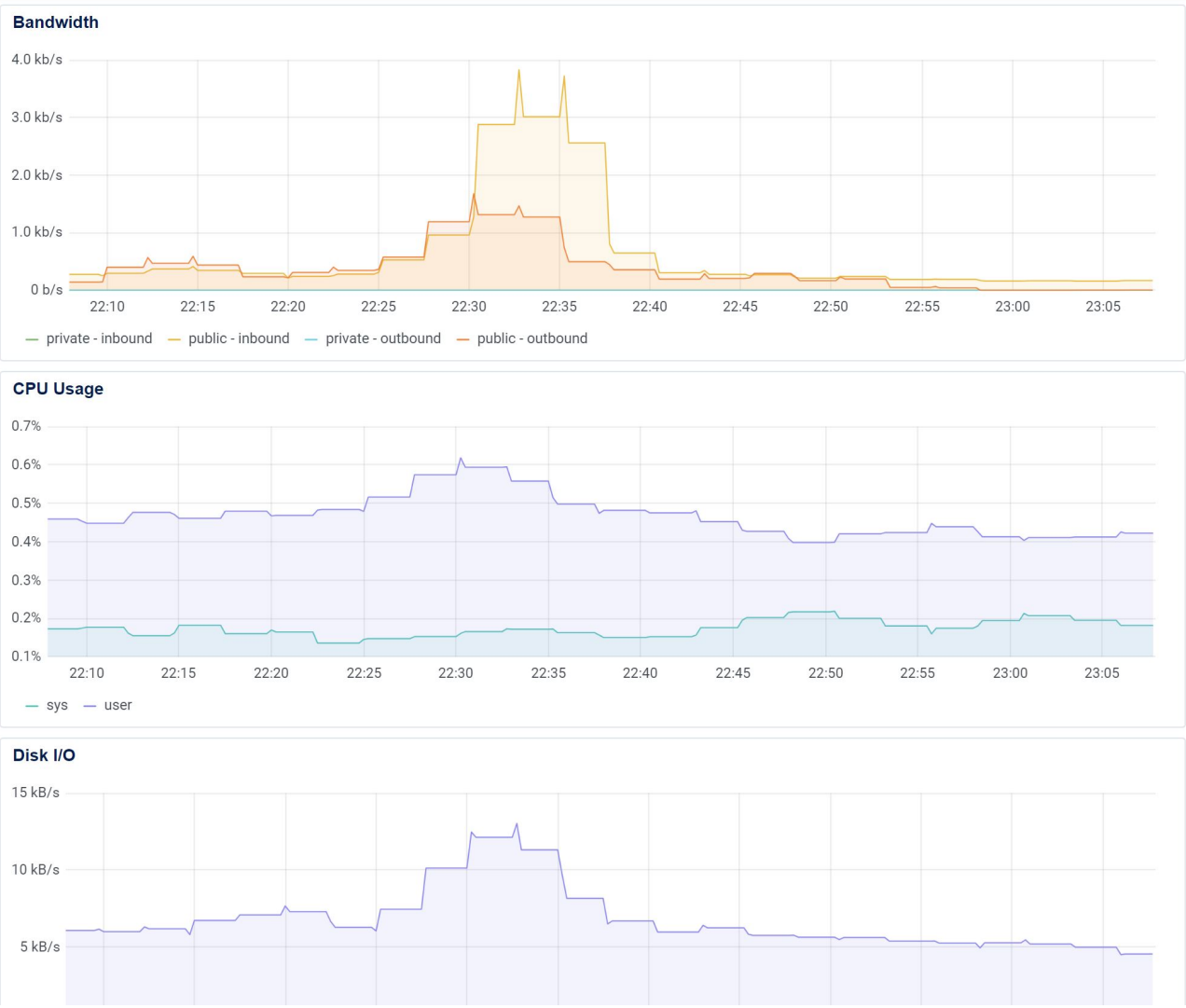

Figure 8: Overview of how the status in one server.

6 Test

6.1 Frontend

We have made a selenium test for testing the functionality of our react application and more specific the adding of bookings on the dashboard page.

```
import unittest
import time
from selenium import webdriver
from selenium.webdriver.common.keys import Keys

class PythonOrgSearch(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Firefox()

    def test_search_in_python_org(self):
        driver = self.driver
        driver.get("https://dev.comit.dev")
        #self.assertIn("React", driver.title)
        elem = driver.find_element_by_xpath("/html/body/div/div/nav/ul/a[2]"
        ).click()

        form_xpath = "/html/body/div[2]/div[3]/div/div[2]/form/div"

        for i in range(100):
            elem = driver.find_element_by_xpath(
            "/html/body/div/div/div/div[2]/main/div/div[2]/div/div/div[2]/di
            v[1]/div[2]/button"
            ).click()
            elem = driver.find_element_by_xpath(f"{form_xpath}/div[1]/div[1]
            /div/input")
            elem.send_keys(f"hello{str(i).zfill(3)}")
            elem = driver.find_element_by_xpath(f"{form_xpath}/div[1]/div[2]
            /div/input")
            elem.send_keys(f"hello{str(i).zfill(3)}")
            elem = driver.find_element_by_xpath(f"{form_xpath}/div[1]/div[3]
            /div/input")
            elem.send_keys(f"hello{str(i).zfill(3)}")
            elem = driver.find_element_by_xpath(f"{form_xpath}/div[2]/div[1]
            /div/input")
            elem.send_keys("Dec/10/2021")
            elem = driver.find_element_by_xpath(f"{form_xpath}/div[2]/div[2]
            /label/span[1]/span[1]/input").click()
            elem = driver.find_element_by_xpath(f"{form_xpath}/div[2]/div[3]
            /div/button").click()

    def tearDown(self):
        pass
        #self.driver.close()

if __name__ == "__main__":
    unittest.main()
```

6.2 Backend

Here we have a CRUD Dashboard's service testing in Postman. The api come from Docker Ubuntu Serever, which come from Digitial Ocean cloud infrastructure provider. Successfully we can create, read, update, and delete service from/to the database throw our backend.

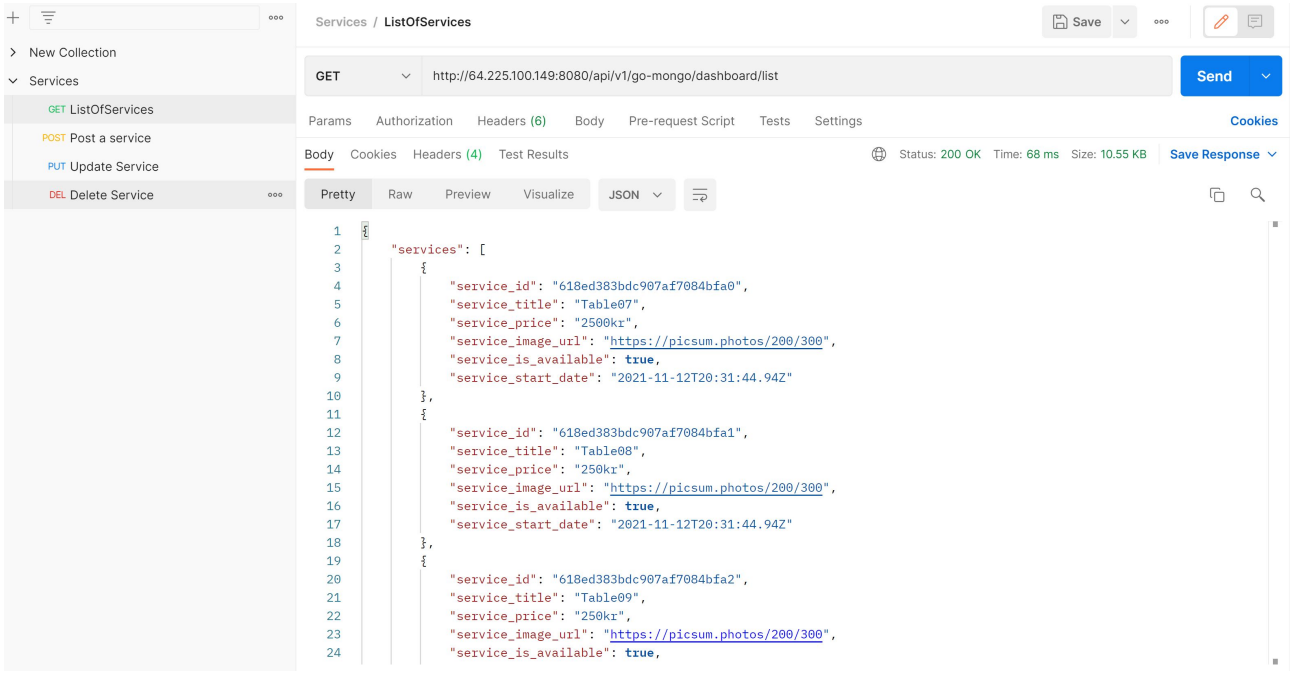

Figure 9: Dashboard's Service Test in Postman

7 Conclusion

The group started out with very high ambitions on a project which we had to simplify a lot. We have tried different frameworks and technologies which were replaced during the process.
We have explored a lot of devops technologies everything from Heroku, DigitalOcean, to gitlab CI/CD and to kubernetes which turned out to be a big mouthful. Overall it has been an education journey, which led to this final project and a single page full stack web application developed in React framework using JavaScript and Golang. This is attached to this final assignment.
We could improve our gated commits with more tests, both frontend and backend. This would secure a more stable release every time and make for a more trustworthy pipeline. If we aim to reach a 100% code coverage we can make a clean releases every release and more rapidly deploy new code or find mistakes we made along the way.
In the context of business, Devops makes it possible to innovate faster for customers, adapt to changing market better as well as grow more efficient at driving business results. Nonetheless DevOps enables developers to take ownership of the product/project and then release quicker updates to the product. Devops also enables rapid delivery and deployment, hence the quicker you can discover bugs and fix them the faster you can meet all the needs and requirements. This ensures the quality of the application and makes it possbile to reliably deliver at a more rapid pace. Under a DevOps model it also makes it easy to work across the entire application lifecycle - from development and test to deployment to operations - and improve a range of technical skills not limited to a single function.

8 References

References

[1] Document: this document is being inspired by
https://github.com/LinkedInLearning/Software-Design-Requirements-Release-2825344

[2] Dashboard, React Material UI Complete Tutorial
https://www.youtube.com/watch?v=m-2_gb_3L7Q&t=5s

[3] Backend, How to Build Rest API Using Golang and Mongo DB With the Concept of Clean Architecture
https://medium.com/learning-about-golang/golang-how-to-build-rest-api-using-golang-and-mo

[4] Backend, [Golang] How to Record System Activity with Log on Go
https://medium.com/learning-about-golang/golang-how-to-record-system-activity-with-log-on

[5] Backend, The Clean Architecture
https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

[6] Deployment: How To Set Up a Continuous Deployment Pipeline with GitLab CI/CD on Ubuntu 18.04
https://www.digitalocean.com/community/tutorials/how-to-set-up-a-continuous-deployment-pip