Pradip Bohora

Design and Develop Decentralized Microservices Architecture with Docker Container

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor’s Thesis

27 August 2021

Abstract

Author:	Pradip Bohora
Title:	Design and Develop Decentralized Microservices Architecture with Docker Container
Number of Pages:	
Date:	27 August 2021
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Professional Major:	Software Engineering
Supervisors:	Janne Salonen, Head of the Major

The purpose of this thesis was to study the process of developing the decentralized microservices and as a prototype, develop an application with standalone services, create the image of the services using the docker and deploy it in the Kubernetes cluster. The objectives of the study were to find the differences between monolith applications and microservices. Also, the benefits of microservices over monolithic application were studied and how to implement the event bus in microservices to keep the data synced between the services was researched.

The challenges of storing data in the services and the solution for database to create a self-contained service was studied. Besides, the service was dockerized. It was deployed in the Kubernetes cluster provided by the cloud service such as Google cloud platform. Different objects of Kubernetes such as pods, services, deployments and ingress-nginx for routine traffic to different pods was studied.

An application with microservices was created using NodeJS, NextJs, Docker, Kubernetes, ingress-nginx and nats streaming server. Some third-party modules were also used to develop the application successfully.

To summarize, microservices are the best solution for the large-scale application with huge number of users. The independently scalable feature makes it easy to scale certain service on demand which is faster and cost efficient.

Keywords: Microservices, Docker, Kubernetes, Decentralized microservices architecture

Contents

List of Abbreviations

1 Introduction 1

2 Microservices 2

2.1 Differences between monolith and microservices 2

2.2 Decentralized Architecture 3

2.2.1 Sync communication between the services 4

2.2.2 Async communication between the services 5

2.3 Deployment Issues 6

3 Docker 6

3.1 Container 7

3.2 Image 8

4 Kubernetes 9

4.1 Pod 10

4.2 Deployment 10

4.3 Service 11

4.4 Load balancer and ingress controller 12

4.5 Skaffold 14

5 Project Implementation 14

5.1 Server Implementation 16

5.2 Frontend Implementation 36

6 Conclusion 38

References 38

List of Abbreviations

AWS:	Amazon Web Services.
API:	Application Programming Interface
CD:	Continuous Deployment
CI:	Continuous Integration
HTTP:	Hypertext Transfer Protocol
IP:	Internet Protocol
JSON:	JavaScript Object Notation
JWT:	JSON Web Token
NPM:	Node Package Manager
OS:	Operating System

1 Introduction

Most of the applications are built with monolithic server. A monolithic server is a server where all the code needed to run the whole application are inside of one code base and it is deployed as one discrete unit. So, in the monolith server when a request comes from the browser, it will go through some middleware to process the request. Also, it goes off to the route, and the router after inspecting the request sends it a specific feature to further process or query the data and send back the data as response that frontend application can utilize. The whole application has database, all the routings and middleware in a single codebase. So, monolith is easy to understand and implement but there is a huge drawback of monolithic application. Suppose one of the features is not functioning currently or it has failed, then the whole application goes offline.

In microservices application each of the features is an independent server. The feature has all the routings, middleware, and its own database. So, the nice thing about microservice is that if for some reason any of the features or services inside the application crashes or disappears, the other services are going to work normally as each of the services is self-contained and does not necessarily require other services to work correctly.

Furthermore, microservices are independently scalable. If there is an increase in the demand of a certain feature, the resources of the demanded feature could be scaled up easily rather than scaling up the entire application. The microservices has a small codebase and as the whole application is split off into small features or services, it is easy to maintain, deploy and keep the code clean.

This thesis was carried out with the aim of studying the process of a microservices application and deploying it into the Kubernetes cluster of google cloud platform. A prototype microservices application was developed using NodeJS, NextJS, Docker, Kubernetes, ingress-nginx and nats streaming server to demonstrate the implantation of microservices. The thesis focuses on different tools used to develop a standalone service, the implementing of the event bus for sharing the

information among the services and different Kubernetes components that assist in developing and deploying the microservices application.

2 Microservices

Microservices is the architectural design where each function works as an independent service. A single microservice contains all the routing, middleware, business logic and its own independent database. So, in the microservices architecture all the features are split off and wrapped inside their own personalized services.

The main advantage or purpose of the microservices architecture is that even if any of the features in the large application fails, the other services work as smoothly as before, unharmed and not affected. Each service has its own database also known as database-per-service which is a big challenge in microservices architecture design. Therefore, one service should not be accessing data from another service database.

The key principles of the microservices-oriented architectural design are resilience, composability, elasticity. These key principles must be taken into consideration while designing microservices architecture. Otherwise, we could end up with a monolith application split that produces problems rather than an elegant solution.

2.1 Differences between monolith and microservices

Monolith is an architectural design where the whole application is built in a single unit. In the monolith application all the features have the common shared database, routing, and middleware. Therefore, it is obvious that all the features are dependent on one another. If one of the features fails, then the whole application will be affected.
Some of the advantages of monolith application are

- Easy to understand and design and develop.

- Easy to debug and to test. As all the features are under one service it is easy to run the end-to-end testing.
- Easy to deploy. Monolith runs under a single unit so there is only one deployment that one must handle.
- In monolith application it is easy to handle logging, caching, and performance monitoring. In a monolithic application, this area of functionality concerns only one application so it is easier to handle it.

In microservices design all the features have their own independent service. Each feature work like a fully developed application and does not communicate directly with any other features.

Advantages of microservices are

- Microservices design can be deployed independently so it allows more team autonomy. Each service is independent so the team can work autonomously use any selected programming language and database.
- Microservices can be scaled independently.
- In a large application the failure of a single service or feature does not affect the whole application.

2.2 Decentralized Architecture

Microservice design follow the decentralized architecture where each of the features is independent and makes its own decision. The decentralized architecture provides independent and autonomous services but there also some challenges that come along with it.

One of the challenges is data management between the services. Since the data storage in microservices is very different than in the monolith application, there are some complications on accessing the data over the services. There are two main things that should be considered while designing the database in microservices. First of all, each service should have its own database if it needs

one. Secondly, one service is never going to reach out to another service database.

Let’s take a simple example to see how data, management in microservices are challenging. There is a blog application with three simple features. There are users and the users can create the blog and read the blog written by other users. Therefore, the blog and the user will have their own database and they do not access each other’s database. Let’s imagine we want to query the blogs with users’ details who created the blog. There are two solutions to this problem. The first one is sync communication between the services and the second is event-based communication or async communication.

2.2.1 Sync communication between the services

In a sync communication process, we will create a new service that is going to communicate with other services directly using request. The request might be a plain HTTP request or a request that exchanges JSON.

The new service is going get the blog data from the blog service and users’ data from the user service. First it is going request data from the blogs service and if the blog exists then it will again make a request to the user service and get the user’s details connected with that blog.

Advantages of sync communication

- Easy to understand and implement.
- The newly created service does not need any database management.

Disadvantages of sync communication

- It introduces the dependency bet the services.
- If one the services crash, then the newly created request service will also crash
- With the bigger application it might create the web of requests.

2.2.2 Async communication between the services

In async communication world a new service will be created, but the service will also have its own database. This time the new service will have the database tables for the user and blog. So, in async communication whenever there are any changes in any of the services, then that service will emit an event with the event type and some details from the service. The event will flow into the event bus and the event bus will take that event and sent if off to any services that is interested. Our new service will have some code to handle the data coming from the event bus and to save it to the database.

There is also one important issue with async communication. What if one of the services goes down for some period? In that case our new service will never get the date event and the database of that service will not be updated. One of the best solutions for this issue would be to store events created in the past in the event bus data store. If the service is created after some time or comes online after some time, it can access all the events and update its data.

Advantages of async communication

- The services have zero dependencies with each other's.
- The newly created service will be extremely fast as it has its own database, and it does not have sent request over other services.

Disadvantages of async communication

- Data duplication. For the larger application we might end up paying extra for the database.
- Hard to understand. All the events sending data with event bus might be challenging and lot of extra work.

2.3 Deployment Issues

Running and testing the application in the local machine is easy but when it comes to deploying the application online the situation might get complicated. One easy option to deploy the application online would be to rent the virtual machines from cloud service providers such as Digital Ocean, AWS, or Microsoft, etc and make some changes to our code base and transfer all our services to the virtual machine.

The above option is easy to understand and fast but when we think about the growth of our application in the future and the traffic directed to our application routes, the situation starts to get more challenging. Let’s imagine there is a large request coming to one of our services. At some point we might need to create a second instance of that service to handle the additional requests. One easy solution for this is to create another copy of the service in the virtual machine and add the load balancer to randomly send the request to the servers.

The newly created copies of the services have different ports, and our event bus must know the ports which is very challenging and difficult to predict.

To solve this problem, a docker can be used which will make the process a lot easier. A docker has an isolated computing environment called containers which contains every element that is required to run a single program. A docker also provides images which is an immutable file that has all the commands, source code, dependencies library and middleware.

3 Docker

Docker is a software platform that provides a standard environment to build, test and deploy application quickly. It is an open platform building, shipping, and running the application across any platform or computing environment.

3.1 Container

Docker container is an isolated standard unit of software that packages the source code and its dependencies and libraries so that the application runs quickly and reliably on any computing environment. See figure 1.

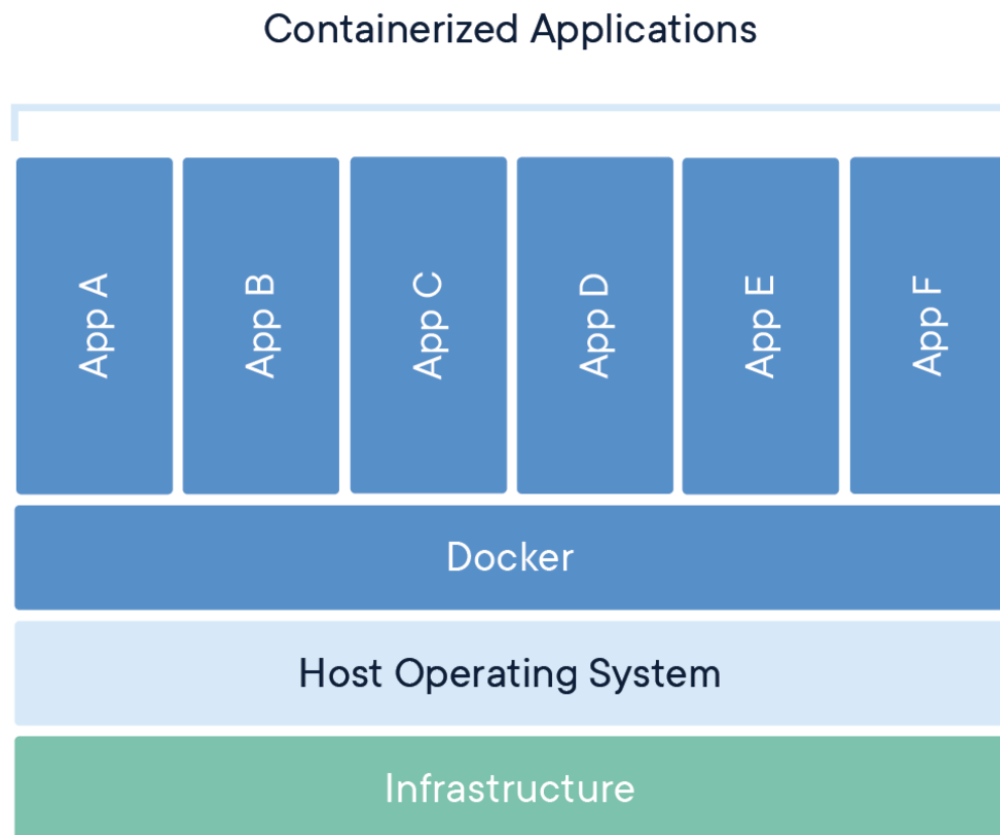

Figure 1: Containerized applications running inside docker. [16]

Docker container is a light weight, secure and executable package of software that contains everything needed to run an application. Multiple containers can run on a single machine with shared OS kernel as an isolated process. Container takes less space and can run many applications with a limited operating system compared to a virtual machine. See figure 2.

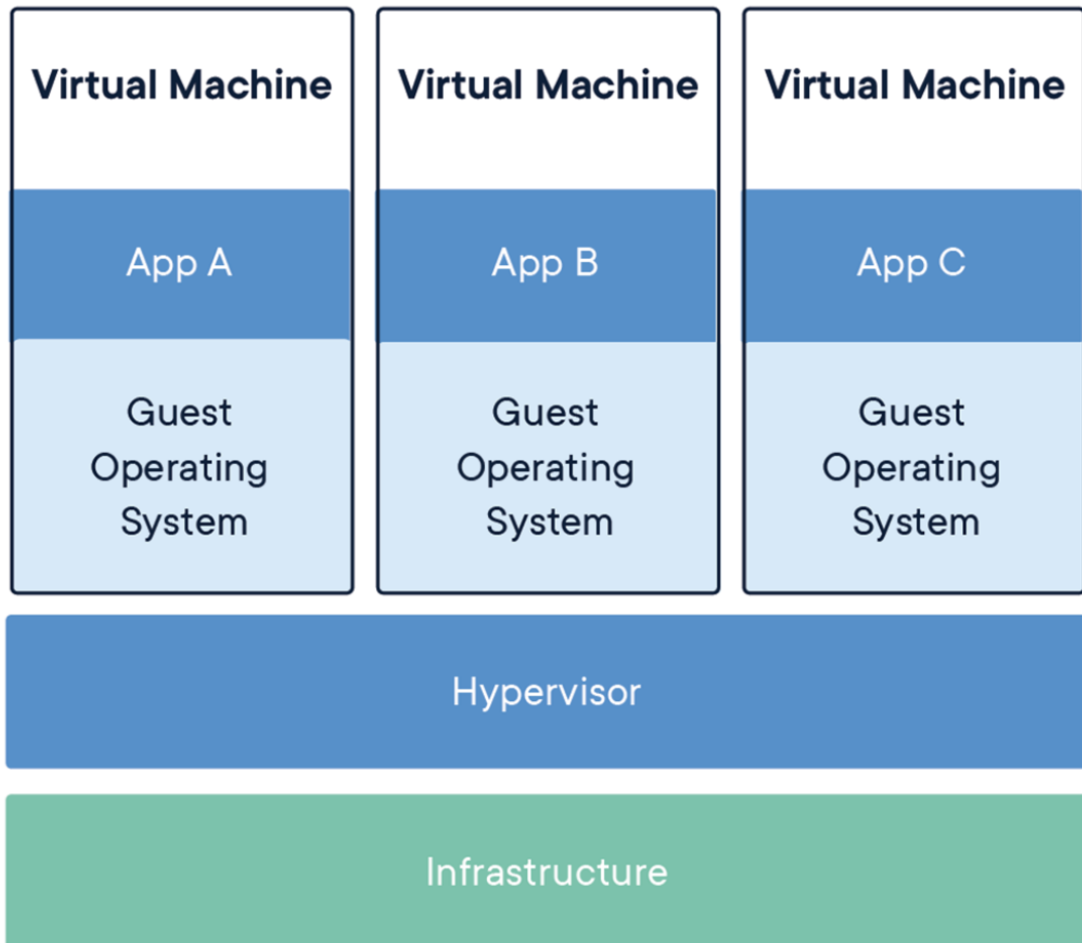

Figure 2: Applications running in virtual machine. [16]

3.2 Image

Docker image is a file that contains command to execute the code in the Docker container. It is a set of instructions to build and run the docker container. It contains all the application source codes, dependencies, tools, libraries to run the application. When the image runs, it becomes one or many instances of a container.

Images contain many layers which originate from the previous layers but each one of them is unique. The layers speed up the docker build process and increase the reusability and decrease the memory use. Docker images are a reusable

asset that can be deployed by any host. The static image can be used from one project to another. This also saves time and resources.

The docker images can be stored online in a public or private repositories. One of the popular image repositories is Docker Hub. There are two types of images in the Docker Hub. Official images are one that a docker produces and the community image is the image that was created by users. User can also create a new image from the existing image and push it to the repository with docker push command.

4 Kubernetes

Kubernetes is an open-source container orchestration platform that can be used to scale up the web hosting. Kubernetes automates the scaling of the application according to the level of the traffic in production. Kubernetes also has an advanced load balancing feature to route web traffic to different servers. It is going to handle the communication between all different containers.

It runs according to the configuration provided by us based on how each container should run and interact with each other. We always use configuration files to create different Kubernetes objects such as Pod, Deployment and Service. See figure 3.

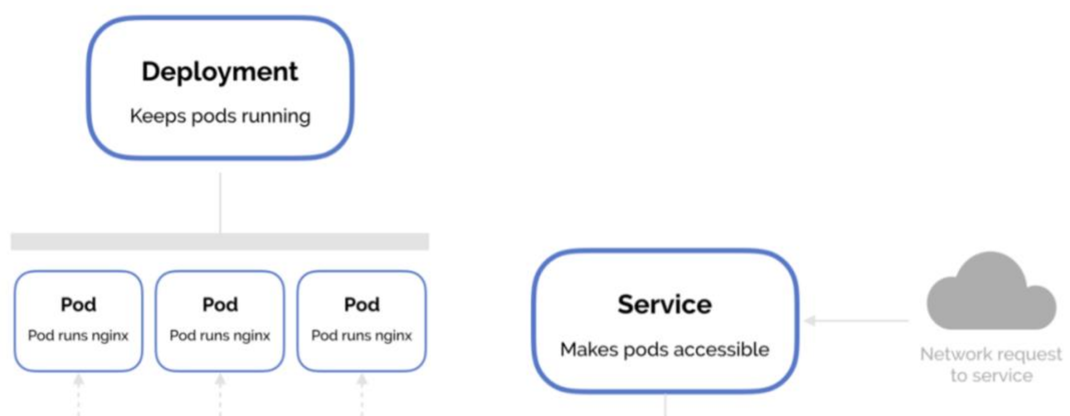

Figure 3: Kubernetes Pod, Deployment, and service. [17]

4.1 Pod

Pods are the basic unit of Kubernetes that contain one or more containers which run the application. Each pod has their one IP address that is shared by the containers that resides inside that pod. The IP address of the pod will change whenever the pod is killed or recreated. The pod must recover automatically and any data that needs to persist after the pod is killed should always be stored in the database attached to the pod.

Each pod should have a persistent storage volume, configuration information to run the containers and unique IP addresses which allows pods to communicate with each other. Most of the time the pod contains a single container.

If the node fails in the cluster, then the controller will find that malfunctioning pod and create and replace it with new pod. If a pod has many containers, then all the containers will share their resources and dependencies.

The containers in the pod can communicate with each other by using a localhost. To communicate outside of the pod, it is required to expose the port. Kubernetes assigns each cluster a cluster IP to every pod in the cluster through which the pods can communicate with each other with the help of the service.

4.2 Deployment

According to the Kubernetes documentation “A deployment provides declarative updates for Pods and ReplicaSets.”

A deployment is a Kubernetes object that is intended to manage a set of pods. Each of those pods are going to be identical and they will be running the same configuration and the same container inside them. There are two main jobs of deployment. The first one is to make sure if any pod disappears for some reason or crashes, then the deployment is going to create that pod again. It is the primary job of the deployment.

The second job of deployment is to update the versions of the pods. For example, we create the pod with a certain container image but later we update our application image to a newer version and want to update our pods to that version and rollback to the previous versions. The deployment is going to take care of the update automatically behind the scenes.

The deployment can be created and managed by using Kubernetes command line interface. When the deployment is created, the container image of the application and the number of replicas must be specified.

4.3 Service

A Kubernetes service is a logical set that defines both the method and policy to communicate between different pods and external requests such as a HTTP request from the Frontend application. Therefore, every time we try to communicate and network, we utilize the Kubernetes service. Every pod and deployment that is created is going to have its own service.

There are three types of services that are available but only cluster IP and node port is used in most of the cases.

a. Cluster IP: It is the service that is used to communicate between the pods. Only exposes the port to the cluster so that one pod can comminate effectively with another pod in each cluster.

b. Node Port: This service that is used to communicate outside the Kubernetes cluster. It is mostly used for the development purposes only.

c. Load Balancer: Load balancer works as like the Node port. It makes the pods accessible outside the cluster. It is a better way to expose the pods to the outer world.

Whenever we are trying the establish the communication between one pod to another, we send the request to the cluster IP of another pod and the cluster IP sends request to the actual pod. A load balancer service is the best way to communicate between the frontend application and backend pods.

4.4 Load balancer and ingress controller

There could be different numbers of pods depending on the size of the application. The communication between the pods is handled by the cluster IP but for the communication between the bowser or frontend application and pods is possible with the help of load balancer. The frontend application is going to access the load balancer service which contains some logic and configuration inside of it that is going to take the incoming requests and route them to the appropriate pods. Responses come through the same load balancer and send them back to the frontend application.

Basically, the job of the load balancer service is to route the traffic to a single pod. If there are more than one pod of different services, then the load balancer is of no use. Therefore, a requirement of ingress controller is needed. The ingress controller is a pod that has a set of routing rules that distributes traffic to different pods of different service (microservices service) inside a cluster. See figure 4.

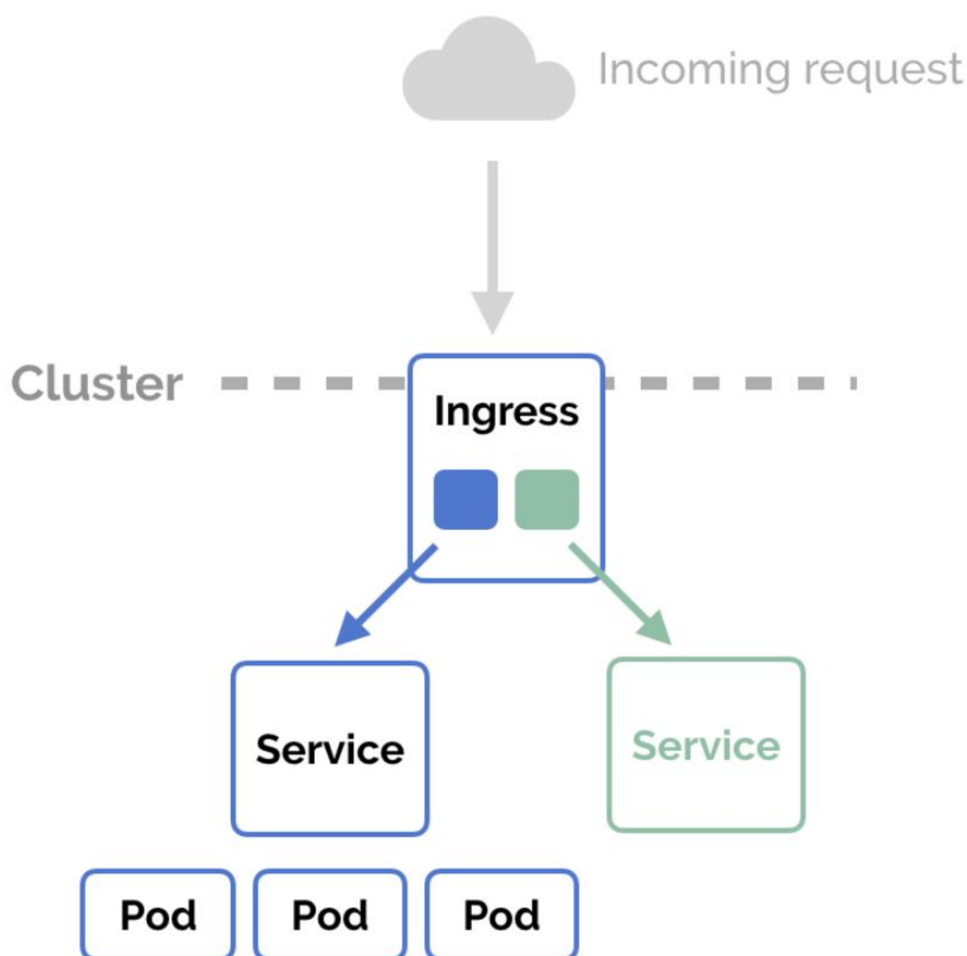

Figure 4: Ingress controller handling route.[18]

The request from the frontend is sent to load balancer and the load balancer service passes that request to the ingress controller which with the help of routing rules knows in which pod the request is targeted and sends the request the cluster IP of that specific pod. Ingress-nginx is one of the open-source projects that I utilized in my project for ingress controller. The ingress controller does not recognise the HTTP methods such as GET, POST, and PUT. Therefore, we must think about better route path names in our application to make sure the incoming request coming from the browser is passed to the correct route.

4.5 Skaffold

Skaffold is a command line tool that automates many tasks in the Kubernetes development environment. With the help of the skaffold it is very easy to manage the Kubernetes objects. It is an open-source project provided by Google.

Skaffold handles the workflow for building, pushing, and deploying the application and provides base for creating CI/CD pipeline. So, with the help of skaffold we do not have to update, create, or delete the Kubernetes objects like pod service manually whenever there is any update in the application source code.

Benefits of Skaffold

- Local Kubernetes development is fast with skaffold. The changes in the source code are detected by the skaffold and handles the pipelines of the application automatically.

- Skaffold projects can be used to share the projects with other developers, to build CI/CD pipeline and build image and render templated Kubernetes manifests to user in GitOps workflow.

- A single pluggable, declarative configuration file can handle the whole project.

5 Project Implementation

The event booking application was developed to study and develop the decentralized microservices architecture where each of the services are not dependent to any other services. The application was developed with help of NodeJS, mongoDB in the backend and NextJs and bootstrap in the frontend. Docker is used to create the images of each service and Kubernetes to manage all the services and their instances. The application was deployed in Google

Cloud Services. The skaffold is also developed by a Google team and therefore, it has a very good integration with Google cloud services.

The development tools that are used to develop and deploy the application are Vs code as code editor, git, and GitHub for the version control, NPM to manage the node modules and Google Cloud Services for Kubernetes cluster and deployment.

The application has the features like

- Users can create the account for themselves
- After creating the account, user can sign in.
- When the users sing in, they are navigated to the home page
- In the navigation bar user can see different options like creating an event, view all upcoming events, view all the events created by the user.
- The “all events page” lists all the events in the future and the user can reserve the spot by clicking the “Attend” option.
- By clicking the “view details” option, user is navigated to the detail page of that event. In the details page user can see all the information about the event and able to edit the event if that event was created by the signed in user.
- All events are visible even if the user is not signed in but event actions like attend event and edit is not available of the user is not signed in.

5.1 Server Implementation

The backend of the application is developed with Nodejs/Express. It is one of the most popular JavaScript frameworks that is used to create backend APIs. The latest version of the Node was installed globally from the Node official page which is mandatory to create the Node application.

There are three main features of the application. Each feature is a service that is interdependent and has it own routing logic, middleware, express server, and database. The services are

- Auth service
- Events service
- Booking Service

Each of the service was initialized by the command in the terminal

```
npm init -y
```

The command initializes the project by creating *package.json* file. This file contains information about the packages installed for that application and their versions, version of the application description and the commands to run and manage the application.

After initializing the services, the following command was run to install express.

```
npm install express
```

It adds the *node_modules* folder that contains the installed packages. The *src* folder was created and it contains the following sub folders: controllers, models, utils and the file inde.js. The controller has all the route of that service, models have the mongoDB database objects, utils folder has all the custom middleware

and helper function of that service. Index.js file has the express server running and the log to connect the database and all the imports like route and middleware.

Middleware

Middleware is the function that can execute any code, make change to the request and response objects, or end the request-response cycle.

Custom middleware was added in the middleware folder inside the utils folder. A separate middleware was created to handle the errors. The error handler middleware sends the errors message as a response if any error occurred during the API call.

The current user middleware compares and verify the json web token that was set in cookie when user signed in and on success sends back the information of the current user. The require auth middleware uses the current user middleware and send require authorisation message if current user is not available. See figure 5.

```
const jwt = require("jsonwebtoken");

const currentUser = (req, res, next) => {
  if (!req.session.jwt) {
    next();
  }
  try {
    const payload = jwt.verify(req.session.jwt, process.env.JWT_KEY);
    req.currentUser = payload;
  } catch (err) {}
  next();
};

module.exports = { currentUser };        @pradipbohora [6 weeks ago]
```

Figure 5: Custom middleware to get current user

All the services have the same middleware. To use this middleware in all services there were two options, one was to copy and paste the middleware file in each service and another was to create the npm package and publish it to the npm. In this project the first option was used for the middleware.

Nats streaming server

The official docker image of nats-streaming was used in the Kubernetes cluster to create the nats-streaming deployment and service. See figure 6.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nats-depl
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nats
  template:
    metadata:
      labels:
        app: nats
    spec:
      containers:
        - name: nats
          image: nats-streaming:0.22.0
          args:      You, a month ago • add
            [
              "-p",
              "4222",
              "-m",
              "8222",
              "-hbi",
              "5s",
              "-hbt",
              "5s",
              "-hbf",
              "2",
              "-SD",
              "-cid",
              "eventbooking",
            ]
---
apiVersion: v1
kind: Service
metadata:
  name: nats-srv
spec:
  selector:
    app: nats
  ports:
    - name: client
      protocol: TCP
      port: 4222
      targetPort: 4222
    - name: monitoring
      protocol: TCP
      port: 8222
      targetPort: 8222
```

Figure 6: Nats streaming server deployment config

The details of the arguments provided in the container is available in https://hub.docker.com/_/nats-streaming.

The nats-streaming server is an event bus that is going to handle the events in different services. It requires different types of events or event channels that need to be subscribed when listening the events inside different services. See figure 7.

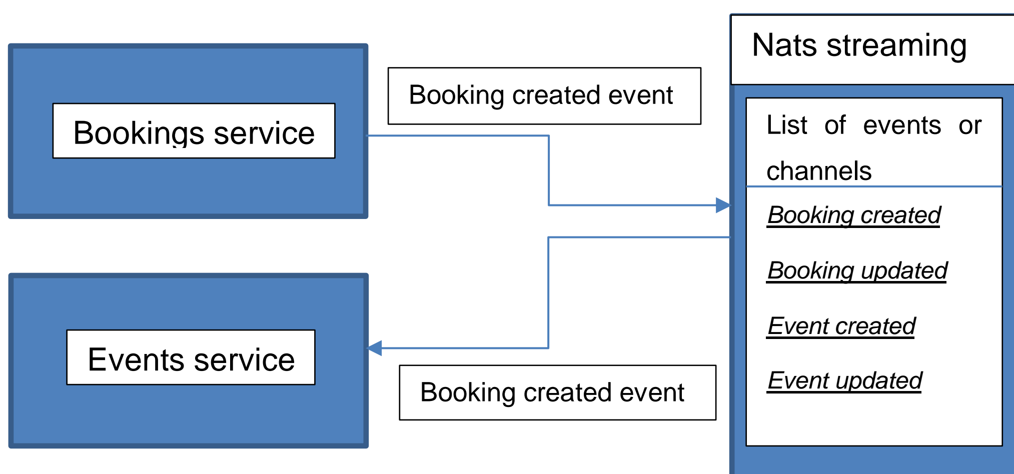

Figure 7: Nats streaming Event bus

In this project when for example a booking is created then the events will be emitted out of the booking service. That event will go over the channel “booking:created” and the event will be sent to only the services that is listening to that channel.

The events are stored in the in the memory of the nats-streaming server. Therefore, when the server is down and then comes back online it can access the event from the nats-streaming and update its database. There will not be any risk of losing data.

The base publisher and listener were created in the common folder in the root of the project. Later each of the services can use this base listener and publisher to listen and publish the event. See figure 8.

```
class Publisher {
  subject;
  client;

  constructor(client) {
    this.client = client;
  }

  publish(data) {
    return new Promise((resolve, reject) => {
      this.client.publish(this.subject, JSON.stringify(data), (err) => {
        if (err) {
          return reject(err);
        }
        console.log("Event published to subject", this.subject);
        resolve();
      });
    });
  }
}

module.exports = { Publisher };
```

Figure 8: Base nats streaming publisher

Like the middleware, the base publisher and listener could be added to each of the services by copying and pasting or publishing it to the npm and later installing the package in all the services. For this module it was published in the npm with package name “common-lib-nats”. To publish the package in the npm an account was in created through the link https://www.npmjs.com/signup. Then the common module from the project was published with command

```
npm publish –access public
```

to create the client “node-nats-streaming” library was used. The client is going to connect to the nats-streaming-server and exchange the information. “Node-nats-

streaming” package was installed in each of the services and in the common module from npm with the command.

```
npm install node-nats-streaming
```

After installation each of the services is going to establish the connection with “nats-streaming-server” with the help of node-nats-streaming library.

Google cloud Platform

The google cloud platform was used to create the Kubernetes cluster for the remote development. Running the Kubernetes cluster in the local machine for development is not a bad idea, but the local machine might get slow down if there are many pods running. Therefore, all the services, deployment and skaffold were configured accordingly to run on the google cloud Kubernetes cluster.

This new project was created in the google cloud platform with name eventbooking-dev form the link https://console.cloud.google.com/home/dashboard. In that project the Kubernetes cluster was created with cluster name eventbooking-dev. To connect the remote cluster from the local machine, a google cloud SDK was installed from https://cloud.google.com/sdk/docs/quickstart and initialized with the command.

```
gcloud init
```

Before initializing gcloud user should be authenticated. Authentication can be done by running the command *gcloud auth* login in the terminal. While initializing the gcloud from the local machine there was also an option to choose the project created in the google cloud that needs to be connected to the local machine. Finally, gcloud context was installed in the local machine with command

```
gcloud container cluster get-credentials eventbooking-dev.
```

In the google cloud “build” was enabled for this project and the option was added in the scaffold.yaml to build the cluster according to the config that was set in k8s folder. See figure 9.

```
build:
  googleCloudBuild:
    projectId: eventbooking-dev
```

Figure 9: Google cloud build config in scaffold.yaml.

Auth service

The auth service handles all the functions that are related to user authentications such as sign in, sign up and sign out.

In the controller folder of the auth service that was initialized by the above command four different files were created with the names sign up, sign in, sign out and users. The sign-up file has the sign-up route which takes care of creating a new user. Before creating the routes, the mongoDB user schema with the help of mongoose was created. Mongoose was installed with following command in the auth service folder directory:

```
npm install mongoose
```

The user schema has first name, last name, password, and email.

In addition to mongoose another package called body-passer was also to installed. It is a middleware that parses the incoming request before sending it to the route handlers. And express-validator package was also installed to validate the incoming request body.

Back in the sign-up route, first the express-validator package is utilized to validate the incoming request body and rules and error messages were also set. After passing the validation the new user is created in the database.

Similarly, a sign in route which requests a body is validated first with the help of “express-validator” and passed to authenticate. After passing the authentication the token is created with the help of “json-web-token” package and that token is sent in a cookie as a response header.

Besides sign in and sign up routes there are two more routes in auth service, user route and sign out route. The sign out route is simple; it removes the cookie-session. While the user route returns the currently signed in user with the help custom middleware “currentUser”.

The file named “Dockerfile” was added to create the image of auth service. See figure 10.

```
FROm node:alpine          @pradipboh

WORKDIR /app
COPY package.json .
RUN npm install
COPY . .

CMD ["npm", "start"]
```

Figure 10: Docker file for auth service

The image could be built by running the command

```
docker build -t docker_username/auth
```

To run this command successfully, a docker command line tool was installed in the local machine.

Kubernetes deployment and service for Auth service

The “k8s” sub folder inside the infrastructure folder was created at the root of the project. The k8s folder contains all the Kubernetes deployments and services. The deployment .yaml files were created for each of the services.

The deployment file for the auth service is “auth-depyl.yaml”. In this file the apiVersion of apps/v1, a kind of deployment, was set. The metadata had the name of the deployment. The number of pods is set in the spec section. The spec section also has a selector. The purpose of the selector is to instruct the deployment to find all the pods that it is going to create. The selector has “matchLabels” with “app:auth”.

After setting up the selector, the template is set. The template provides the instructions to create each individual pod the deployment is going to create. The metadata is set with labels “app:auth”. The matchLabels in the selector is going to match the labels that we set in the template.

On the same indentation level of metadata, the spec is set. This spec is going to tell the pods how to function. In the spec the containers are designated with name and image. In addition to that the environment variables are also set in the containers. The mongoDB environment was set with “name:MONGO_URI” and “value:mongodb://auth-mongo-srv/auth”. The JWT secret key was also set in the environment. Since the JWT secret key shared by other service also in the application, the generis JWt secret was created in the Kubernetes cluster with the following command:

```
kubectl create secret generic jwt-secret --from-
literal=jwt=secret
```

After defining the deployment, the service was also defined at the end of the same file for auth service, just like in the deployment the “apiVersion”, “kind” and “metadata” was set. In the “spec” the “selector” was set to find the running pods and in the “ports”. All the ports that must be exposed were listed. See figure 11.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: auth-depl
spec:
  replicas: 1
  selector:
    matchLabels:
      app: auth
  template:
    metadata:
      labels:
        app: auth
    spec:
      containers:
        - name: auth
          image: us.gcr.io/eventbooking-dev/auth
          env:
            - name: MONGO_URI
              value: "mongodb://auth-mongo-srv/auth"
            - name: JWT_KEY
              valueFrom:
                secretKeyRef:
                  name: jwt-secret
                  key: JWT_KEY
---
apiVersion: v1
kind: Service
metadata:
  name: auth-srv
spec:
  selector:
    app: auth
  ports:
    - name: auth
      protocol: TCP
      port: 3000
      targetPort: 3000
```

Figure:11: Auth service deployment config file

There were two ports set in the “port” section, “port” and “targetPort”. The “port” is the port were the pod runs and the “targetPort” is the port where the application runs.

Events service

The event service has API routes for the creating, updating, and getting the events. The events service was also initialized with the as in auth service. The route, database schema and express server was initialized just like in auth service. Additionally, the events service has the database schemas for events and bookings. The events schema helps to populate the database with created events and the booking schema helps to populate the database with booking received from the booking service.

The event service shares the events with booking service. Whenever the user signs in and creates an event, the booking service should be able to tract that event so that the user can book that event through booking service.

Inside the src folder a new file with the name “natsWrapper” was added. In this file the nats client was initialized and added to the logic for connection. See figure 12.

```
const nats = require("node-nats-streaming");          @pradipbohora [4 w

class NatsWrapper {
  _client;

  get client() {
    if (!this._client) {
      throw new Error("Cannot access NATS client before connecting");
    }

    return this._client;
  }

  connect(clusterId, clientId, url) {
    this._client = nats.connect(clusterId, clientId, { url });

    return new Promise((resolve, reject) => {
      this.client.on("connect", () => {
        console.log("Connected to NATS");
        resolve();
      });
      this.client.on("error", (err) => {
        reject(err);
      });
    });
  }
}

module.exports = new NatsWrapper();
```

Figure 12: Nata wrapper config file

The connection was established in index.js file. To establish the connection arguments clusterId, clientId and natsUrl were passed which were set by the deployment file for event service. The event listeners and instances of publishers were added in the in the folder “serviceEvents”. See figure 13.

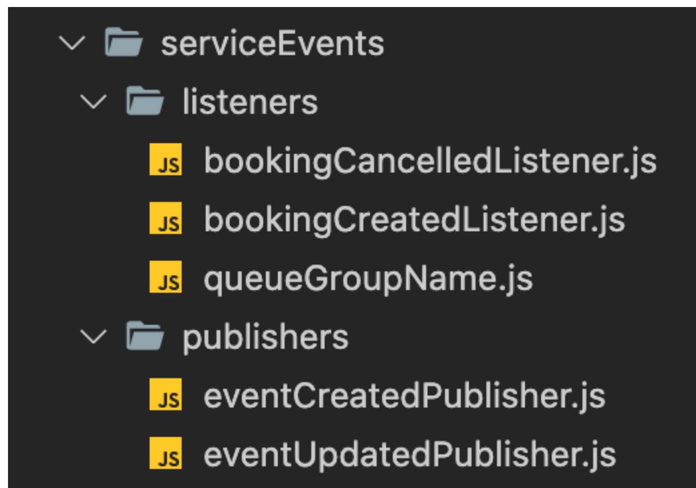

Figure 13: Folder structure of publisher and listener.

The instance of publisher was later used to the create and update events route to publish an event when the user creates or updates the event. See figure 14.

```
const savedEvent = await event.save();
new EventCreatedPublisher(natsWrapper.client).publish({
  id: event.id,
  name: event.name,        You, a month ago • add the event
  date: event.date,
  categories: event.categories,
  createdBy: event.createdBy,
  version: event.version,
});
```

Figure 14: Example of publisher used in create event route

The listener listens to the events published by the booking service and makes the changes in the booking table of the database and the publisher publishes the events whenever the user creates or updates the event. The publisher and listener were created with the help of the package “common-lib-nats” that was published in the npm. See figure 15.

```
const { Publisher } = require("common-lib-nats");

You, a month ago | 1 author (You)
class EventCreatedPublisher extends Publisher {
  subject = "event:created";
}

module.exports = { EventCreatedPublisher };
```

Figure 15: Example of event created publisher.

The Kubernetes deployment and service for the events service is like the auth service. The new environment variables for nats streaming server were also added. See figure 16.

```
env:
  - name: NATS_CLIENT_ID
    valueFrom:
      fieldRef:
        fieldPath: metadata.name
  - name: NATS_URL
    value: "http://nats-srv:4222"
  - name: NATS_CLUSTER_ID
```

Figure 16: Environment variables for nats-streaming server

Booking service

The booking service is similar in structure to events service. Like events service, booking service also has the natsWrapper for the nats connection and event

publisher and listeners for the events coming from the event service. It publishes the booking events to the event service.

Each of the services has its own database. Also, each database separate deployment was created in the k8s sub folder of infrastructure folder.

Ingress-nginx

The ingress-nignx deployment was installed from the Kubernetes documentation with the following command:

```
kubectl apply -f
https://raw.githubusercontent.com/kubernetes/ingress-
nginx/controller-
v1.0.0/deploy/static/provider/cloud/deploy.yaml
```

The ingress config file was created in the k8s folder with the name "ingress-srv.yaml". The file has the configuration to handle the incoming request from the browser. The annotations were set in the metadata to handle different path names in the routing rules. See figure 17.

```yaml
apiVersion: networking.k8s.io/v1        You, 2 months
kind: Ingress
metadata:
  name: ingress-service
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/use-regex: "true"
spec:
  rules:
    - host: eventbooking.dev
      http:
        paths:
          - path: /api/users/?(.*)
            pathType: Prefix
            backend:
              service:
                name: auth-srv
                port:
                  number: 3000
          - path: /api/events/?(.*)
            pathType: Prefix
            backend:
              service:
                name: events-srv
                port:
                  number: 3000
          - path: /api/bookings/?(.*)
            pathType: Prefix
            backend:
              service:
                name: bookings-srv
                port:
                  number: 3000
          - path: /?(.*)
            pathType: Prefix
            backend:
              service:
                name: client-srv
                port:
                  number: 3000
```

Figure 17: Ingess-nginx service config file

In the spec section the routing rules were added with the host name eventbooking.dev and different path names for events, booking, users and client (frontend application).

Skaffold

Skaffold was installed in the local machine with the command

```
brew install skaffold
```

At the root directory of the project the file scaffold.yaml was created. The skaffold config file is responsible for the infrastructure directory.

There are also the deploy and build configurations. The deploy configuration takes care of the .yaml files in the infrastructure and if there is any change to those files, the skaffold applies those changes to the Kubernetes cluster.

The build configuration watches for the changes in the source code of the services and if there are any changes in the files that match the configuration in the services, it is going to update the pods. If they do not match the name, then it is going to rebuild the image and apply it to the pod. See figure 18.

```yaml
apiVersion: skaffold/v2alpha3          @pradipbohora [8 w
kind: Config
deploy:
  kubectl:
    manifests:
      - ./infrastructure/k8s/*
build:
  googleCloudBuild:
    projectId: eventbooking-dev
  artifacts:
    - image: us.gcr.io/eventbooking-dev/auth
      context: auth
      docker:
        dockerfile: Dockerfile
      sync:
        manual:
          - src: "src/**/*.js"
            dest: .
    - image: us.gcr.io/eventbooking-dev/events
      context: events
      docker:
        dockerfile: Dockerfile
      sync:
        manual:
          - src: "src/**/*.js"
            dest: .
    - image: us.gcr.io/eventbooking-dev/bookings
      context: bookings
      docker:
        dockerfile: Dockerfile
      sync:
        manual:
          - src: "src/**/*.js"
            dest: .
    - image: us.gcr.io/eventbooking-dev/client
      context: client
      docker:
        dockerfile: Dockerfile
      sync:
        manual:
          - src: "**/*.js"
            dest: .
```

Figure 18: Skaffold config file

The whole project could be started from the root of the project with command

```
scaffold dev.
```

5.2 Frontend Implementation

For the frontend of the application NextJS was used. NextJS is the JavaScript open-source framework was built on top of Node.js. It enables server-side rendering in react application.

When starting the project, a “client folder” was created and inside the client directory the package was generated with the command.

```
npm init -y
```

After generating the package, the dependencies were installed with following command:

```
npm install react react-dom next
```

The pages folder has all the pages of the application and the components folder the react components. The hooks folder has the useRequest react hook that helps to make a request to the backend API.

To run the front application in Kubernetes cluster the Dockerfile was created. After that the Kubernetes deployment and service config file was created in the k8s folder which is identical to the backend services deployment. See figure 19.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: client-depl
spec:
  replicas: 1
  selector:
    matchLabels:
      app: client
  template:
    metadata:
      labels:          @pradipbohora [6 weeks ago]
        app: client
    spec:
      containers:
        - name: client
          image: us.gcr.io/eventbooking-dev/client
---
apiVersion: v1
kind: Service
metadata:
  name: client-srv
spec:
  selector:
    app: client
  ports:
    - name: client
      protocol: TCP
      port: 3000
      targetPort: 3000
```

Figure 19: Client deployment config file.

The ingress-nginx config file was updated with frontend routing rules and the scaffold config file was also updated with client image.

6 Conclusion

The goal of this thesis was to study and design microservices with docker and Kubernetes. Different ways of creating microservices and ways of establishing the communication between the services was learned during the thesis. The advantages as well as the disadvantages of developing microservices were studied. In addition, the deployment of the microservices application in the Google cloud platform was done.

Microservices might look confusing and hard to understand. The microservices might also use more recourses as all the services have their own databases. It is also required to add some event bus between the services to make the data synced. In some cases, we might have to copy the whole data from one service to another which increases the use of recourses, so the cost of development could be higher. The debugging of the application could be challenging with multiple services having their own sets of logs.

However, compared to monolith application, the speed of the application is very fast as each of the features is divided into small services with its own data. Since each component is independent, it avoids shutting down the whole application if any of the features is unable to function. With the help of docker and Kubernetes the application is easy to manage. It is also easy to deploy different services with it.

References

1 Microservices Documentation. Microservices architecture. Available from: https://microservices.io/

2 Docker official documentation. Build and run docker image. Available from: https://docs.docker.com/get-started/

3 Kubernetes documentation. Understand Kubernetes. Available from: https://kubernetes.io/docs/home/

4 Docker documentation. Deploy in Kubernetes. Available from: https://docs.docker.com/desktop/kubernetes/

5 NGINX ingress controller. How it works. Available from: https://kubernetes.github.io/ingress-nginx/how-it-works/

6 Kubernetes Ingress. Terminology and ingress resources. Available form: https://kubernetes.io/docs/concepts/services-networking/ingress/

7 CSC Rahti documentation. Kubernetes and OpenShift concepts. Available from https://docs.csc.fi/cloud/rahti/concepts/

8 NextJS. About NextJS. Available from: https://nextjs.org/docs

9 Skaffold. About scaffold and getting stated. Available from: https://skaffold.dev/docs/quickstart/

10 Google cloud. Google Kubernetes engine. Available from: https://cloud.google.com/kubernetes-engine

11 Docker Hub. MongodB reference. Available from: https://hub.docker.com/_/mongo

12 Nats Docs. About nats streaming server. Available from: https://docs.nats.io/nats-streaming-server/changes

13 Docker hub. Reference and command line options. Available from: https://hub.docker.com/_/nats-streaming

14 NPM. Node-nats-sreaming. NodeJS client for nats streaming. Available from: https://www.npmjs.com/package/node-nats-streaming

15 Morgan Bruce, Paulo O Pereira. Microservices in Action[eBook]. O`Reilly Safari Online, October 2018. Available from: https://learning.oreilly.com/library

16 Docker. User containers to build, share and run your applications. Available from: https://www.docker.com/resources/what-container

17 AVM Consulting. Deployment types in Kubernetes. Available from: https://medium.com/avmconsulting-blog/deployment-types-in-kubernetes-14b70ca7ef93

18 Matthew Palmer. Kubernetes Ingress with Nginx Example. Available from: https://matthewpalmer.net/kubernetes-app-developer/articles/kubernetes-ingress-guide-nginx-example.html