NACKADEMIN

Final Thesis

–

Microservices

A case study for building a scalable application with Kubernetes and Kafka

Class: Java19
Author: Johannes Svensson
Supervisors: Mahmud Al Hakim, Sigrun Olafsdottir
Program Managers: Lotta Holmgren, Sara Strömvall
Stockholm: 2021 January

Abstract

How are big companies like Amazon's, Netflix's and Google's backends able to handle all their high traffic? I started to do some research, one answer I got was they all have microservice that can scale efficiently. While microservices are still not that clear how to create, and most often is the monolithic architecture used, this study will show you the foundation you need to build microservices. Especially how you make your services scalable with Kubernetes and Kafka. These tools will be helping us a lot to fulfill this goal.

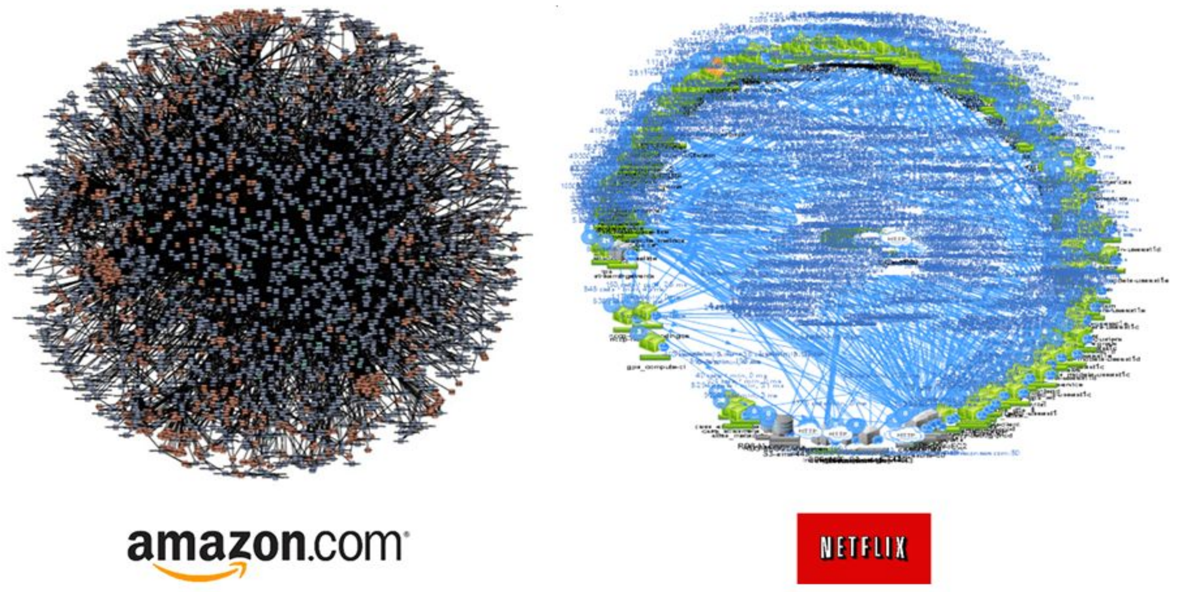
In the results of this study we will see that Loose Coupling is a big factor when building microservices and to help achieve scaling and deploying of services independently. This study also shows you with a command how easy it is to scale up or down a service when we are done.

In this study I will be building a Car Rental Service as my backend. It will be separated into four microservices (Car, Payment, Booking and User) and use both Kubernetes and Kafka.

Table of Content

1. Introduction

When you search for something on Google's search engine, watch Netflix or maybe use one of Amazon’s many services it is not only your requests that their backends handle. The fact is that they have several billions to sometimes trillions of users per day! And when you make a search on Google it is approximately 63 thousands other that do it at the same time. Have you ever wondered what companies, like these three’s, backends look like and how they can handle this kind of load? Below you will find pictures of Amazon’s and Netflix’s microservice architecture:



These two round shapes represent hundreds of microservices and how they are linked together with other parts of the application by these lines. The lines presuppose different communications, either REST calls, publish/subscribe or to a database.

1.1 Background

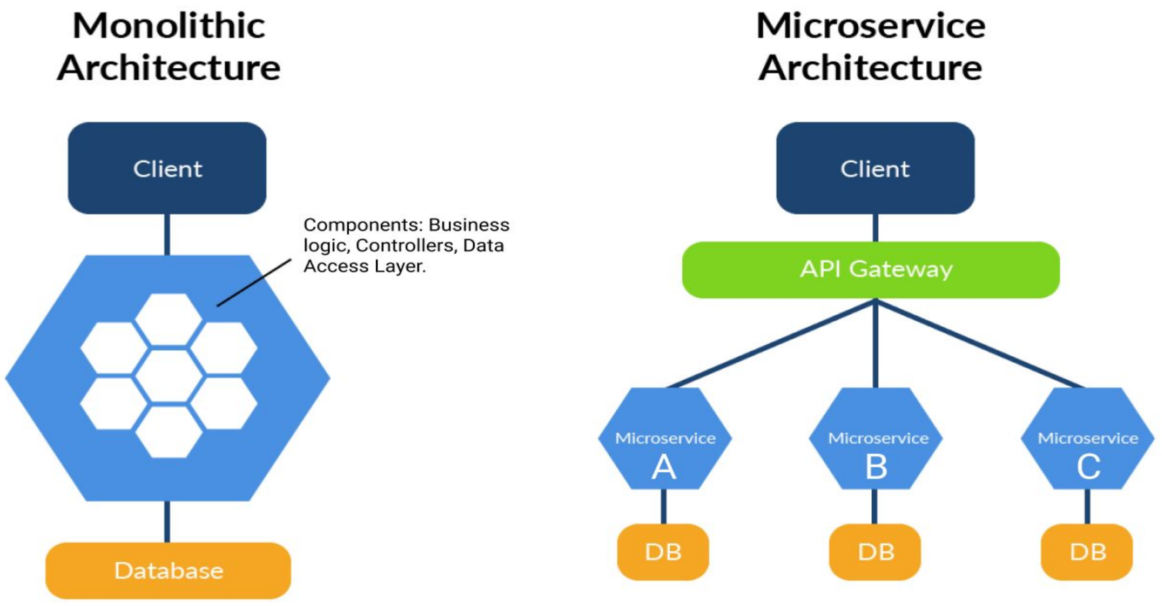
A monolithic application is what you usually always start with when building a new backend. A small team of programmers can fast and easily get a monolithic up and running in production which usually suits and fulfills all the needs of a company. The problem comes when the company grows bigger and gets more customers using their services. With a monolithic approach this can be difficult and expensive to try to scale up the application to meet the new needs of the company. While microservices entails benefits and solutions to some of monolithics disadvantages, one of the biggest benefits you get is a more effective scalable application. Many companies today, including Netflix and Amazon, changed to microservices partly for this reason.

The question is, how do you build a backend using a microservice architecture to scale services independently? There are many different ways of building microservices but in this study I am going to deepen myself in how to build a foundation of scalable microservices with the help of Kubernetes and Kafka.

1.2 Purpose

It's important to be able to scale your microservices by adding, removing or updating services without other services needing change. Isolate services, and give them loose coupling.

Why scalability is so important in an application is to be able to handle an increasing (and sometimes decreasing) number of users of that application. Because more users often means more in sales for a company, this is a crucial thing for them. A monolithic is difficult to scale because it only has one linjar dimension to do so (Looking at the picture below; by duplicating the large blue figure times itself). With microservices we get a lot more flexible scalability. In other words we can choose to scale either A, B or C independently from each other to meet the increasing pressure of the users in the best way possible.



We can also see that all microservices have their own database which helps the flexibility of the application and we get a more fault tolerant application as well (if one database crashes we still have other parts of our application working). Another thing these pictures show is that even though we have more 'blue figures' in microservices, the client only needs to know the API Gateway, not any of the microservices (gives us loose coupling[1]).

My purpose for this study, I therefore decided, is to find out how you start building a scalable microservice backend. The tools I will specifically use to help me achieve this are Kubernetes and Kafka (which I will talk more about in part: 1.4 Tools).

1.3 Question formulation

How do you build a foundation of scalable microservices?

[1]*Loose Coupling* is about detaching elements from each other to make them independent. Sam Newman describes Loose Coupling as this; "*when services are loosely coupled, a change to one service should not require a change to another.* And ...*knows as little as it needs to about the services with which it collaborates.*".

1.4 Tools

1.4.1 Description of Kubernetes

Kubernetes is one tool this study will use. Kubernetes is an open-source container orchestration platform created by Google. It automates scaling, managing and distributing of services. The core problem Kubernetes solves is to build and manage microservices clusters. It does this by helping us implement helpful tools in a very simple way, for example; Load Balancer, API Gateway etc.. It takes care of a lot of things in the background for us as well, for example if a container or Pod goes down, Kubernetes starts up a new one for us.

1.4.2 Description of Kafka

Kafka is the second tool that will be used in this study. Kafka is also an open-source solution and it is developed by Apache. Kafka is a platform for managing data streams in real time. It consists of Kafka Brokers which are servers that lets clients communicate with it (the clients in my case will be my microservices). Communication happens through subscribing/reading and publishing/writing of events to a topic. This makes clients able to read and write to topics with other services without even knowing of each other. A Kafka cluster can consist of one or more Kafka Brokers which makes it scalable and fault tolerant. In other words, if one Kafka Broker dies, another broker can take over to prevent data loss.

1.5 Methodology

We will start building the Car Rental backend service (See Appendices: 6.3 *Car Rental Events and Entities*). I chose to first build it in a monolithic approach as this is usually what most companies start with, also it's fast to get feedback on what and how things work in the project. With a monolith we now have a foundation to start to separate the logic into different microservices. This Car Rental will have four microservices, one for managing bookings, one for payments, another for all the cars and last one is going to handle all our users. We are going to add Kafka (and Zookeeper, which is necessary to maintain our Kafka) to handle two topics, with two different events happening in our application; first one is when a new successful payment (PaymentSuccesfulEvent.java) happens and the other is when a booking gets canceled (BookingCanceledEvent.java). Our Payment Service will be sending the PaymentSuccesfulEvent and Booking Service will be consuming these to create a new booking into the database. BookingCanceledEvent will be sent from Booking Service and consumed by Payment Service to make a repayment of that order. (See a diagram of these events in Appendices: *6.2 Microservice Architecture*).

The User Service will handle user registrations and be generating JWT tokens for successful logged in users, to give them access to all parts of our application.

The Car Service will only show us all cars and give us knowledge of when they are available. To do this, Car Service needs to fetch data from Booking Service to get the dates when the cars are booked. (See a diagram of this http call in Appendices: *6.2 Microservice Architecture*).

Now that we have our four microservices working together we will start to implement Kubernetes to help us manage all these microservies and Kafka for our communication. An important point to add here is that Kubernetes is a big service to run on your own computer, but it is possible, I will be running it on my local machine in a docker container (but preferably if you can run it in the cloud). Before we can start making Kubernetes yaml files we need to make docker images of everything. Every service (including Kafka and Zookeeper) needs a Deployment yaml file, which is a blueprint for a Pod. Every Pod then needs a Service yaml file to have their own Service being linked to it, it will handle the access to the container running in the Pod.

To implement a API Gateway we need a so called Ingress, for it to work we need a Ingress Controller, which we get from a third party; I chose the NGINX Ingress Controller. This is a traffic management solution for cloud-native apps in Kubernetes[2]. In our Ingress we can then add rules to our path and to which Service the traffic will be directed to.

And lastly we will add some so called ConfigMaps and Secret to our Cluster. These will store host url and database connection url, this will be used by our Pods to get the environment variable they need.

While I was building this on my computer I chose to use just one database for the whole application instead of having one database per microservice (which is the recommended, because as Chris Richardson says: database per microservice… "*Helps ensure that the services are loosely coupled*"[3]). This is because my computer couldn't run all database replicas at the same time. But in my GitHub repository under yamls_kube/mongo/ directory it exists all the yamls to deploy all the databases you need. For the database I chose MongoDB but you can have whatever database that fits you. I also chose Java and Spring Boot for all my services, but the beauty of microservices is that you can choose any language you want. Different microservices can have different languages.

Now after having made all the yaml files written, the last thing to do is to apply them to your Kubernetes cluster and everything will start up. We should have an architecture looking like the diagram in the Appendices: *6.1 Kubernetes Architecture.*

1.5.1 How To Scale a Service

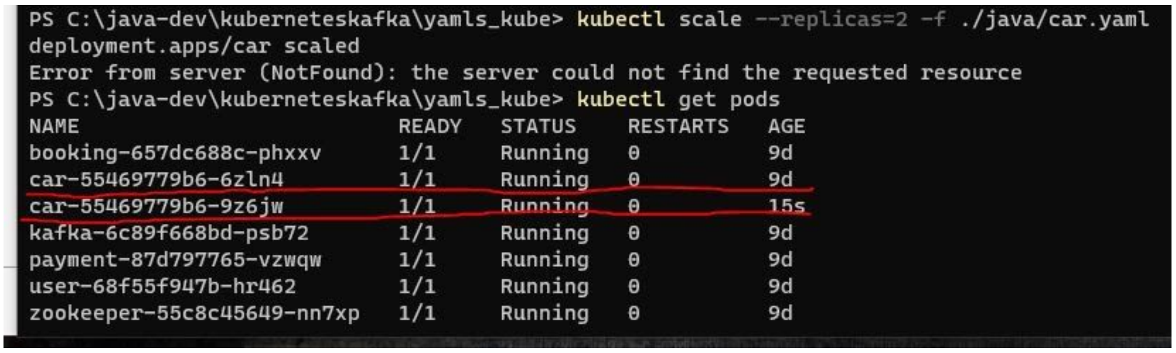
We can now scale our application, this is done by changing the value: "replicas" in any Deployment file by running the command: *"$ kubectl scale --replicas=X -f foo.yaml"* Where X is a number representing how many Pods Kubernetes will start up. (See an example of this in Results: *2.1 Loose Coupling)*.

[2] https://www.nginx.com/products/nginx-ingress-controller/
[3] https://microservices.io/patterns/data/database-per-service.html

2. Results

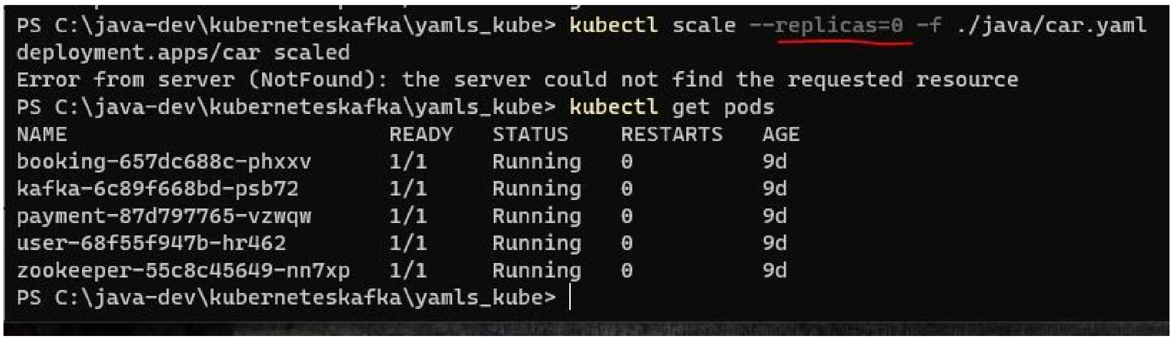
2.1 Loose Coupling

Our microservice can now be scaled and deployed independently, we can also change or add a new service without requiring to change other services. Plus if one Pod stops others will still be working regardless. This results in our microservices being Loose Coupled. As Sam Newman defines loose coupling: *"When services are loosely coupled, a change to one service should not require a change to another."*. And Ben Morris, in his article *"Why is loose coupling between services so important?"* , says a good measure for knowing if your microservice is loose coupled, is if they can deploy a service independently from others and if one service stops others will still be working.



```
PS C:\java-dev\kuberneteskafka\yamls_kube> kubectl scale --replicas=2 -f ./java/car.yaml
deployment.apps/car scaled
Error from server (NotFound): the server could not find the requested resource
PS C:\java-dev\kuberneteskafka\yamls_kube> kubectl get pods
NAME                          READY   STATUS    RESTARTS   AGE
booking-657dc688c-phxxv       1/1     Running   0          9d
car-55469779b6-6zln4          1/1     Running   0          9d
car-55469779b6-9z6jw          1/1     Running   0          15s
kafka-6c89f668bd-psb72        1/1     Running   0          9d
payment-87d797765-vzwqw       1/1     Running   0          9d
user-68f55f947b-hr462         1/1     Running   0          9d
zookeeper-55c8c45649-nn7xp    1/1     Running   0          9d
```

This picture shows that we can scale a service independently (Car Service has now 2 replicas).

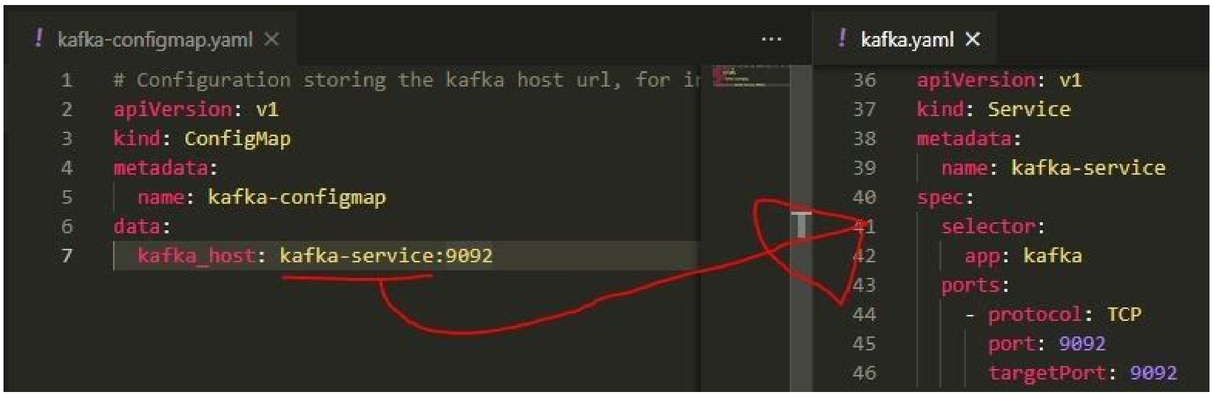


```
PS C:\java-dev\kuberneteskafka\yamls_kube> kubectl scale --replicas=0 -f ./java/car.yaml
deployment.apps/car scaled
Error from server (NotFound): the server could not find the requested resource
PS C:\java-dev\kuberneteskafka\yamls_kube> kubectl get pods
NAME                          READY   STATUS    RESTARTS   AGE
booking-657dc688c-phxxv       1/1     Running   0          9d
kafka-6c89f668bd-psb72        1/1     Running   0          9d
payment-87d797765-vzwqw       1/1     Running   0          9d
user-68f55f947b-hr462         1/1     Running   0          9d
zookeeper-55c8c45649-nn7xp    1/1     Running   0          9d
PS C:\java-dev\kuberneteskafka\yamls_kube> |
```

While this picture shows that we can scale down or stop a service while the others will still be working (Car Service is not running at all now).

2.2 Load Balancer and DNS

With the Kubernetes Service we get Load Balancing and a Domain Name System. The Load Balancer handles the distribution of the network traffic evenly to all the Pods that belong to that Service. A Domain Name System or so called DNS is a hostname given to all Kubernetes Services that we can use to access instead of using each Pods own IP-address.



```yaml
# Configuration storing the kafka host url, for i
apiVersion: v1
kind: ConfigMap
metadata:
  name: kafka-configmap
data:
  kafka_host: kafka-service:9092
```

```yaml
apiVersion: v1
kind: Service
metadata:
  name: kafka-service
spec:
  selector:
    app: kafka
  ports:
    - protocol: TCP
      port: 9092
      targetPort: 9092
```

This picture shows a ConfigMap with a key-value, and the value refers to a Service DNS name. Through this DNS name we will get access to the Kafka Pod, and the Service will Load Balance the incoming traffic.

2.3 Event Driven

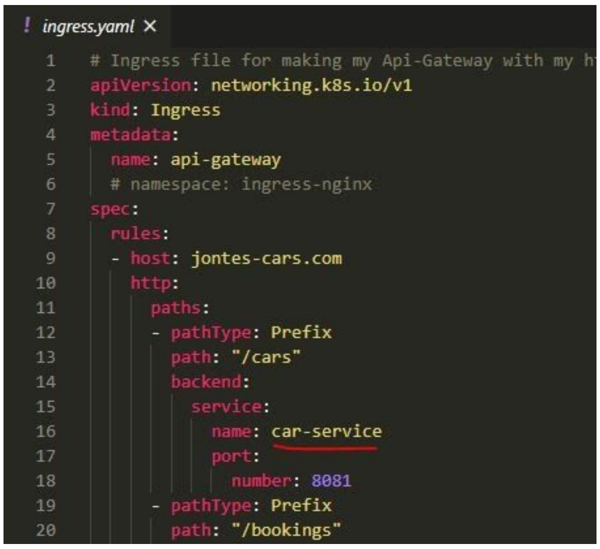
Kafka gives us an Event Driven communication between our microservices which gives us better performance and helps to decouple the microservices. As Sam Newman says in his book: *Building Microservices, "... you want to limit calls from one service to another, because beyond the potential performance problem, chatty communication can lead to tight coupling."*
We can also scale up more brokers which results in a more fault tolerant communication broker service. Having Event Driven Architecture we get a reliable communication and high resilience to failure.

2.4 Logic is Separated

Our Microservices are modelled around our business domain, this results in isolation between each business logic we have in our application.

2.5 API Gateway

The Kubernetes Ingress gives us API Gateway functionality, it exposes our endpoint routes from outside the cluster and linking them to our Kubernetes Services.



```yaml
# Ingress file for making my Api-Gateway with my h
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: api-gateway
  # namespace: ingress-nginx
spec:
  rules:
  - host: jontes-cars.com
    http:
      paths:
      - pathType: Prefix
        path: "/cars"
        backend:
          service:
            name: car-service
            port:
              number: 8081
      - pathType: Prefix
        path: "/bookings"
```

This picture shows our Ingress and our endpoints path rules we have set. Also we see it uses Services DNS name to refer to them.

3. Discussion

Based on the result we can see that our microservices have been isolated and have a good loose coupling between each other. According to Sam Newman this is his Golden Rule: *"The golden rule: can you make a change to a service and deploy it by itself without changing anything else?"*. This looks to be the biggest cause to get a good foundation for being able to scale your microservices.

Our results from having an Event Driven architecture did also help us to get this good loose coupling and also gave us that reliable communication that helps to isolate failure. As Sameer Parulkar says: *"...EDA (Event Driven Architecture) offers high resilience to failure, …if a service goes down it does not impact any of the other services in the chain. ...events can be consumed and processed later, if a service is busy or unavailable."* [4] Having this resilience is crucial when building microservices that you don't want to have any data loss with.

The result for separating the logic into business domains helped give us a better isolation between our services. To model around business domain is actually one of Sam Newman's eight "*Principles of Microservices*". In these Principles we can also see: "Deploy Independently", "Culture of Automation" and "Isolate Failure" which are three other things our microservices can now do with the help of Kubernetes and Kafka.

There are many ways to separate your application into microservices depending on your business domain, but to be able to make them scale and also deploy independently, we need that loose coupling and isolation between them. This can we see now clearly that we have got from our results. I think there's a reason why Sam Newman has that Golden Rule.

Based on the results of getting an API Gateway, Load Balancer and DNS these three don't necessarily give us better scalability but they are crucial when building a good foundation of scalable microservice. Because only if we took away one of them, our program would not be able to scale or be deployed independently efficiently, so they are needed.

One thing this study could have been improved with is by deploying the Kubernetes cluster in the cloud. Because in the cloud we can add a lot more to the cluster, in example one database for each service and we could have tried Kubernetes autoscaling tools. Having a Kubernetes cluster on your local machine has it's constraints, but I do think we have fulfilled our purpose anyway with this solution.

Auto scaling could actually be added to our application at this moment, to get an even more effective scalability. Kubernetes has three tools we can do this by; Cluster Autoscaler, Horizontal Pod Autoscaler and Vertical Pod Autoscaler.

[4] https://www.redhat.com/en/blog/importance-event-driven-architecture-digital-world

4. Conclusions

The conclusions from this study is that with our microservices now being independent and loosely coupled we can both scale and deploy safely and with the help from the API Gateway, Load Balancer, DNS and other helpful managing tools from kubernetes we have got a good environment for when scaling these microservices. With the mixture of all the results we got we did reach our goal of this study, to build a foundation of scalable microservices.

A future research on this can be how to auto scale your Kubernetes cluster. Here we also try to deploy it to a Cloud Service as mentioned in the Discussion, and try the three different autoscalers from Kubernetes (*Cluster Autoscaler, Horizontal Pod Autoscaler and Vertical Pod Autoscaler*). This combined with Metrics Server tool and in our Deployment files adding "resources.request.memory and cpu", as well as "resources.limits.memory and cpu", to see how effective scaling we can get from just the help of Kubernetes. With this we could then make more tests and stress test our building methods.

5. References

Sam Newman. ***Building Microservices.*** 2014. O'Reilly.

Sam Newman. "***Principles of Microservices***". https://samnewman.io/talks/principles-of-microservices/

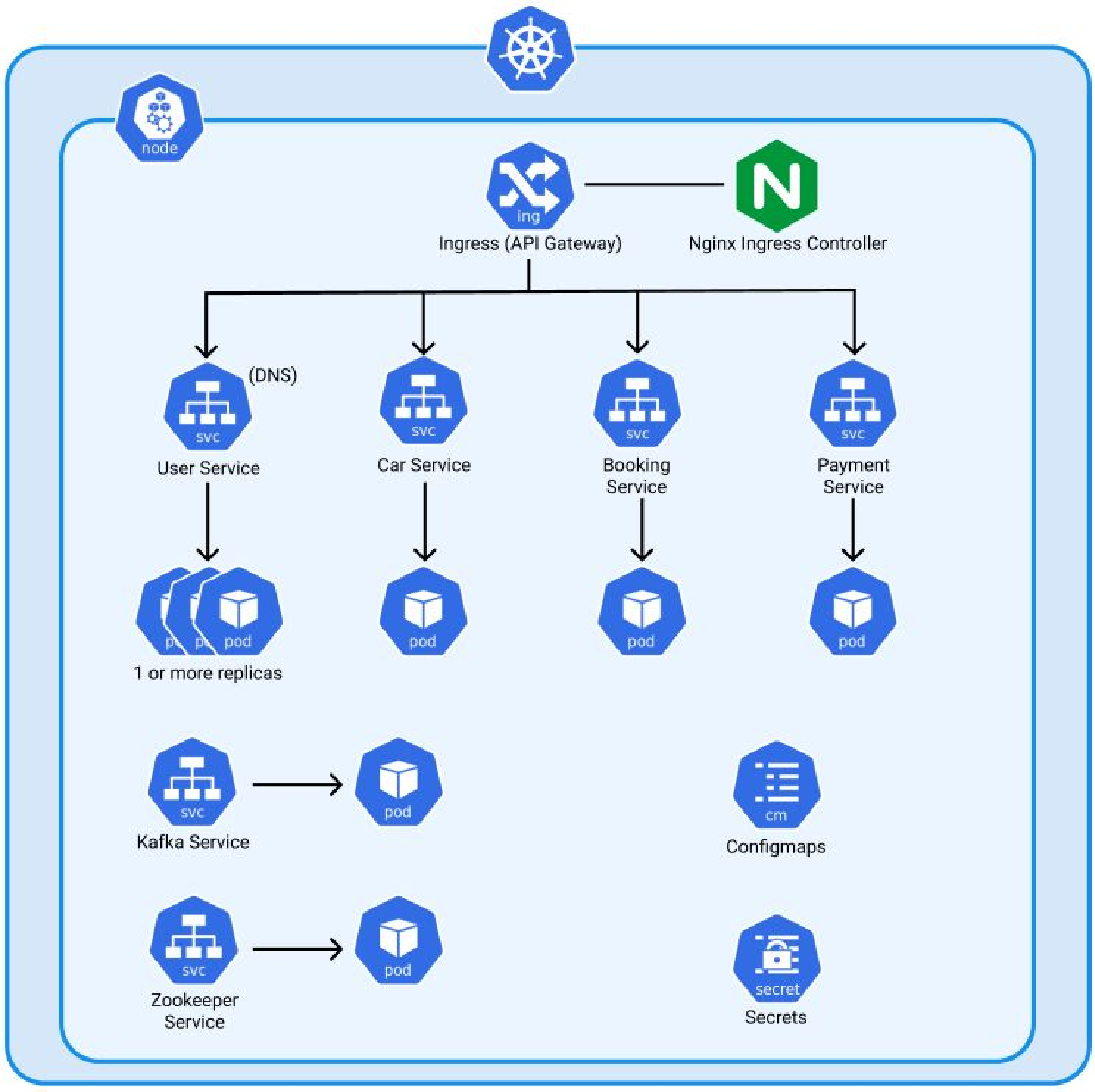
Ben Morris. *"**Why is loose coupling between services so important?**" (Article). 2019. https://www.ben-morris.com/why-is-loose-coupling-between-services-so-important/*

Sameer Parulkar (*Product Marketing Director for Red Hat*). ***"The importance of event-driven architecture in the digital world."*** (Article). https://www.redhat.com/en/blog/importance-event-driven-architecture-digital-world

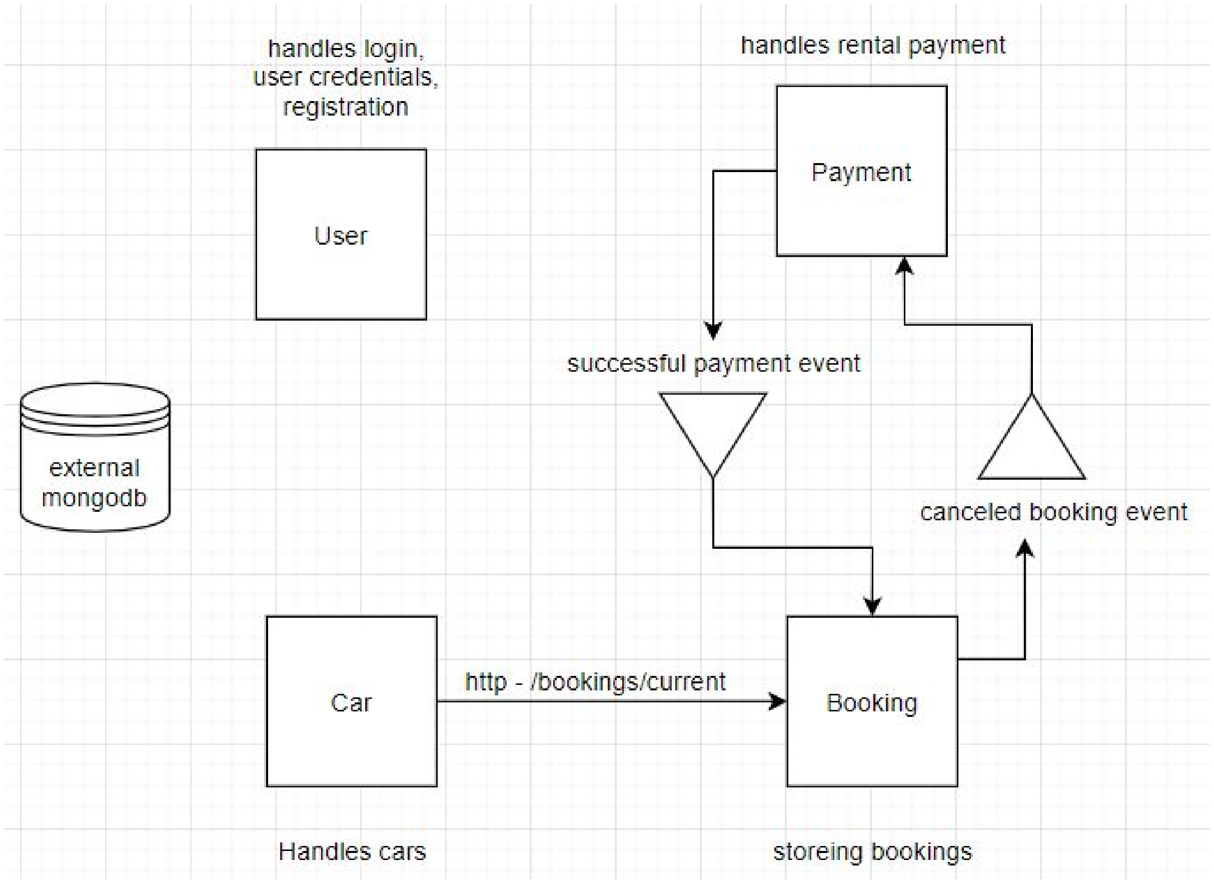
Chris Richardson. "***Pattern: Database per service***". (Article). https://microservices.io/patterns/data/database-per-service.html

6. Appendices

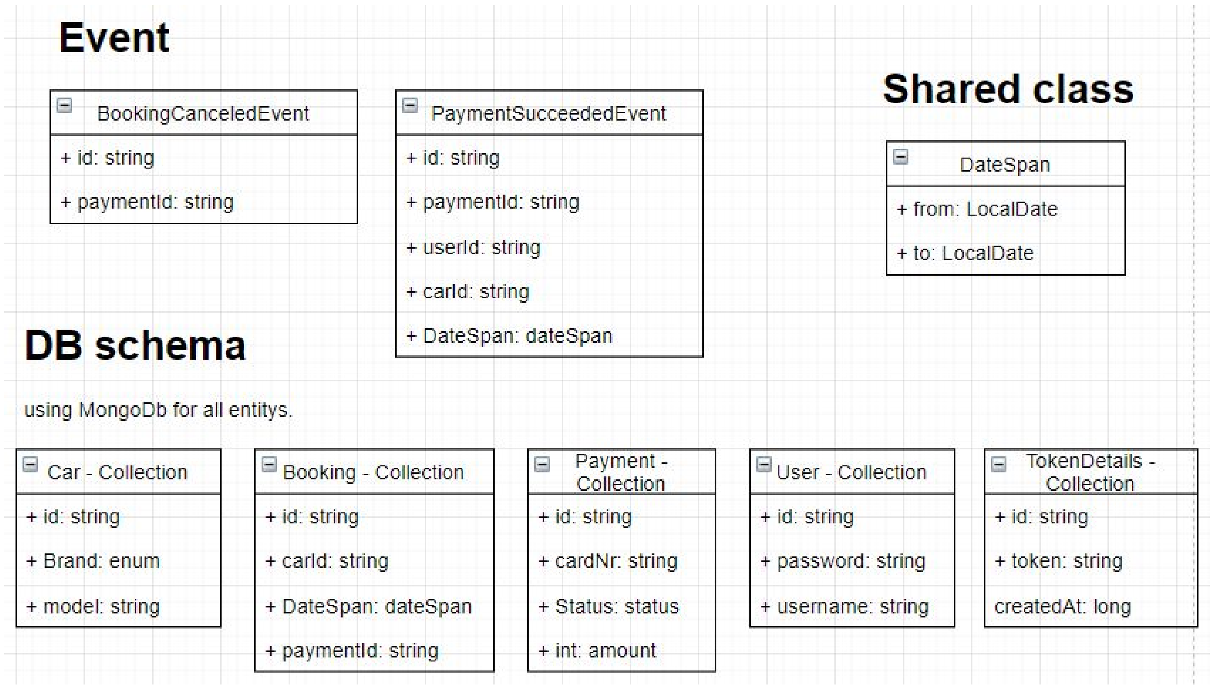
6.1 Kubernetes Architecture



6.2 Microservice Architecture



6.3 Car Rental Events and Entities


Event
BookingCanceledEvent
+ id: string
+ paymentId: string
PaymentSucceededEvent
+ id: string
+ paymentId: string
+ userId: string
+ carId: string
+ DateSpan: dateSpan
Shared class
DateSpan
+ from: LocalDate
+ to: LocalDate
DB schema
using MongoDb for all entitys.
Car - Collection
+ id: string
+ Brand: enum
+ model: string
Booking - Collection
+ id: string
+ carId: string
+ DateSpan: dateSpan
+ paymentId: string
Payment - Collection
+ id: string
+ cardNr: string
+ Status: status
+ int: amount
User - Collection
+ id: string
+ password: string
+ username: string
TokenDetails - Collection
+ id: string
+ token: string
createdAt: long