

PKWAd goes Kubernetes

- [PKWAd goes Kubernetes](#)
 - [Introduction](#)
 - [Problems](#)
 - [Multiple containers as Stateful-Services](#)
 - [Images from private Registry \(GitLab\)](#)
 - [Allow access to services](#)
 - [Create Ingress](#)
 - [Conclusion](#)
 - [Useful links](#)

Introduction

Our project involves an application with microservice-architecture which contains different applications/tasks. The core of the project are the crawler built in python, which fetch data from Austrias biggest used car exchanges. Those cars are then grouped (for comparability) and stored inside a database (MariaDB). Additionally there are different services for an Event-Queue, Proxy-Management, Offer-Checker, Statservice and a Search-Engine, which is currently under development.

The basic idea of this project is now moving the existing project from docker swarm to Kubernetes. This move should yield a significant advantage in terms of scalability, deployment and availability. In terms of scalability the current state of the project would already be enough in terms of crawling-speed, even though, there is never enough speed 😅.

The main concerns in the current state are the event-queue (RabbitMQ) and the database (MariaDB). Both of the services represent a current state and are not that easy to replicate. For this problem docker swarm does not provide a solution and even if they provide a solution it's very tedious and not intuitive. Despite the immediate benefits when migrating to Kubernetes in the form of redundancy it also provides a more future-prove architecture with the rapid developments on the open-source-project and the wide spread of the orchestration-software.

Problems

Multiple containers as Stateful-Services

Multiple services and containers need to be treated as Stateful-services. Especially the database (MySQL) is critical, since many services depend on it. Furthermore rabbitMQ has to be a stateful-service. This is not a problem as itself, but needs some additional configuration. The basic-YAML-file that provides this sort of

Images from private Registry (GitLab)

First add the registry to minikube

```
minikube addons configure registry-creds
```

Next up we create a docker secret

```
kubectl create secret docker-registry regcred --docker-server=regis
```

Lastly we have to add the following code to the deployment-files inside the spec-section (before containers)

```
imagePullSecrets:
- name: regcred
```

Allow access to services

One Service we would want to access from outside is rabbitmq for example. Turns out exposing the ports to the host-machine is not that easy with statefulsets and the docker-driver in particular on macos. So when using the docker-driver with minikube we couldn't expose the ports. Getting the actual IP and Port for the service, the command

```
minikube service rabbitmq-0 --url
```

yielded the response, that the terminal needs to be open to run it. Despite doing this, the URL is never coming up.

To overcome this problem we switched the driver for minikube to **hyperkit** - this works now, as demonstrated with an example from kubernetes' website.

```
thomaslumesberger@Thomas-MacBook-Pro: ~/Documents/projects/autocheck-2/kubernetes
> docker-machine-driver-hyper...: 65 B / 65 B [-----] 100.00% 7 p/s 0s
> docker-machine-driver-hyper...: 8.35 MiB / 8.35 MiB 100.00% 11.23 MiB p/s
The 'hyperkit' driver requires elevated permissions. The following commands will be executed:
$ sudo chown root:wheel /Users/thomaslumesberger/.minikube/bin/docker-machine-driver-hyperkit
$ sudo chown ans /Users/thomaslumesberger/.minikube/bin/docker-machine-driver-hyperkit

Password:
❯ Downloading VM boot image ...
> minikube-v1.25.2.iso.sha256: 65 B / 65 B [-----] 100.00% 7 p/s 0s
> minikube-v1.25.2.iso: 237.40 MiB / 237.40 MiB [ 100.00% 17.25 MiB p/s 14s
Starting control plane node minikube in cluster minikube
Creating hyperkit VM (CPUs=2 Memory=2240MiB Disk=20480MiB) ...
❯ Preparing Kubernetes v1.23.3 on Docker 20.10.12 ...
  • Downloading kubeadm
  • Generating certificates and keys ...
  • Booting up control plane ...
  • Configuring RBAC rules ...
❯ Verifying Kubernetes components...
  • Using image gcr.io/k8s-minikube/storage-provisioner:v5
  • Enabled addons: storage-provisioner, default-storageclass
Done! minikube is now configured to use "minikube" cluster and "default" namespace by default
❯ kubectl create deployment web --image=gcr.io/google-samples/hello-app:1.0
deployment.apps/web created
❯ expose deployment web --type=NodePort --port=8888
service/web exposed
❯ kubectl get service web
NAME    TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
web     NodePort  10.103.206.147   <none>           8888:32316/TCP   4s
❯ curl -v service web --url http://192.168.64.2:32316
* Host 192.168.64.2:32316 not resolved
* connect to 192.168.64.2:32316 failed: Connection refused
* Closing connection 0
```

Next up we modified the YAML-files for the services that had to be accessed from outside. This is an example for rabbitmq.

```
apiVersion: v1
kind: Service
metadata:
  name: rabbitmq
spec:
  type: NodePort
  selector:
    app: rabbitmq
  ports:
    - port: 15672
---
```

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: rabbitmq
spec:
  selector:
    matchLabels:
      app: rabbitmq
  serviceName: rabbitmq
  replicas: 2
  template:
    metadata:
      labels:
        app: rabbitmq
    spec:
      imagePullSecrets:
        - name: regcred
      containers:
        - name: rabbitmq
          image: registry.gitlab.com/lum1nanz/autocheck-2/rabbitmq
          ports:
            - name: management
              containerPort: 15672
```

To make this even better we add ingress to the cluster, so we can access the services with `http://cluster_ip/<serviceroute>`

Create Ingress

The following content is added to the beginning of `rabbitmq.yaml`

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: rabbitmq
  labels:
    name: rabbitmq
spec:
  rules:
    - http:
        paths:
          - pathType: Prefix
            path: "/rabbitmq"
            backend:
              service:
                name: rabbitmq
                port:
                  number: 15672
```

```
minikube service -n rabbitmq rabbitmq --url
```

Unfortunately this doesn't yield the success we were hoping for. The problem is, that all content is getting blocked or at least not distributed to the browser. This yields us a blank page, but the title is set correctly, so theoretically it should work. Our supposition is that the `nginx-ingress-controller` does something unexpected and therefore doesn't show the body of the html-page.

ATTENTION: In between these steps we removed Minikube from the system and went on with the standard-kubernetes installation that comes preinstalled with Docker Desktop

After quite some time trying different things out, we think we finally found the solution. We just had to follow the instructions on [Kubernetes Ingress](#) and the [Installation Guide of Nginx](#)

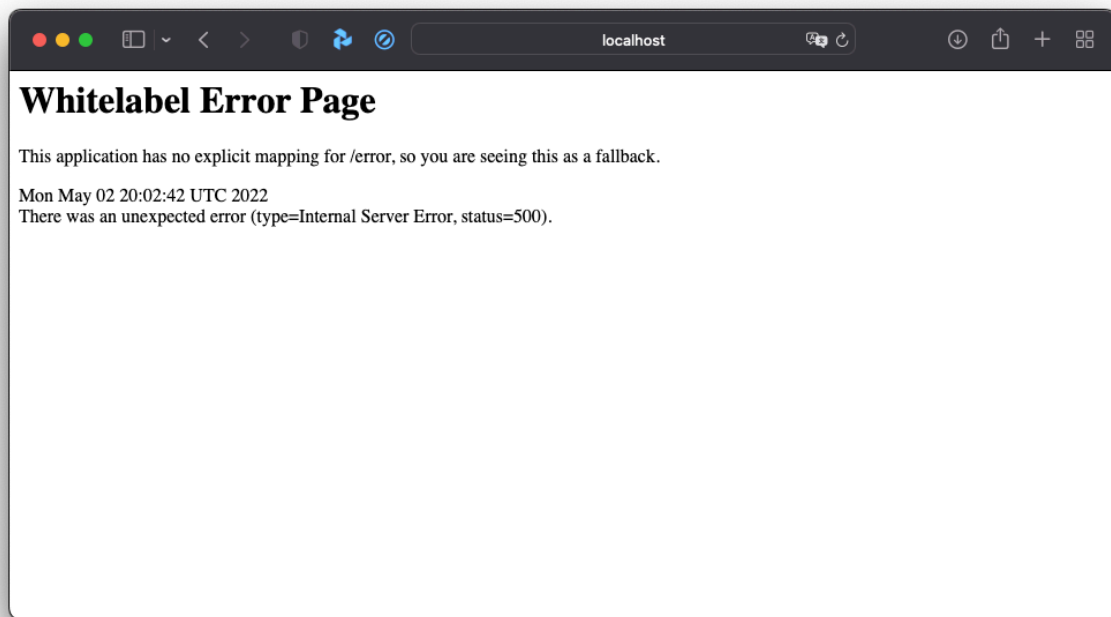
Especially important is this command, which sets up the `nginx-ingress` to work properly:

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingre
```

Starting the sample-configuration down below finally gave us the reward of hours of try-and-error, when the error-page of spring-boot finally showed up in the browser under the address `http://localhost/`.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: test-deployment
  labels:
    app: test
spec:
  replicas: 3
  selector:
    matchLabels:
      app: test
  template:
    metadata:
      labels:
        app: test
    spec:
      containers:
        - name: test
          image: stakkato95/kubernetes-shutdown:latest
          ports:
            - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: test-service
spec:
  selector:
    app: test
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-test
spec:
  ingressClassName: nginx
  rules:
    - host: localhost
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: test-service
                port:
```

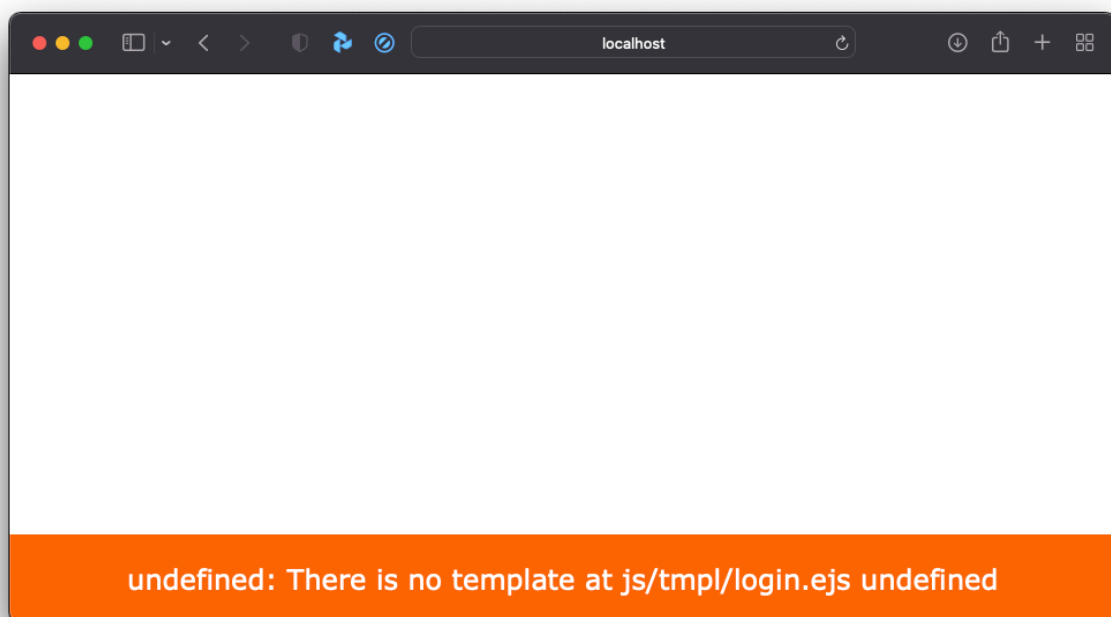
number: 80



The error-page is not what is relevant for us. We just wanted the routing via ingress to work and now it finally does.

After this finally worked, we got back to our deployment-files and modified them to the point where they should work and yield us a proper result. And after the deployment:

TADA 🎉



To overcome this problem, we have to declare a rewrite of the urls that are now invalid,

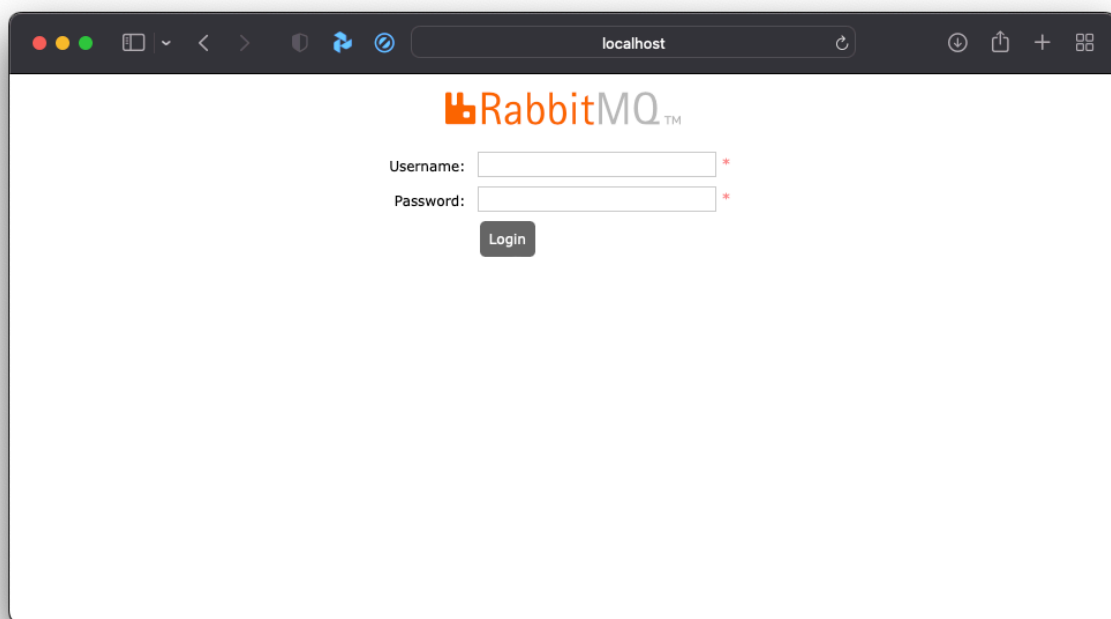
since the url is now `http://localhost/rabbitmq/` and this doesn't match with the paths in the original container. Rewriting the url basically just ment opting for this line with path:

```
- path: /rabbitmq(/!$)(.*)
```

and those inside the metadata

annotations:

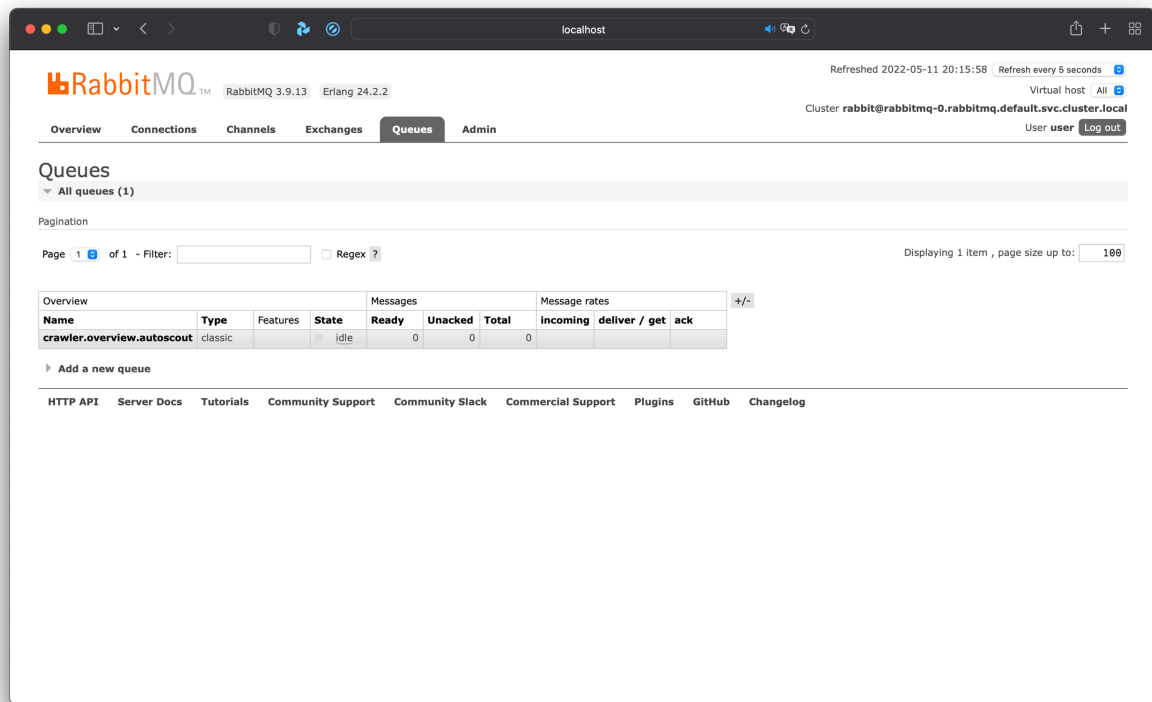
```
nginx.ingress.kubernetes.io/rewrite-target: /$2  
nginx.ingress.kubernetes.io/ssl-redirect: "false"
```



Since this shows the starting-page, we thought we were done, but turns out, that the login is not working properly. For the sake of getting it to work finally, we scraped the path `/rabbitmq/` parameter and let it just run on `/`, which finally made the page work.

Once we got that issue resolved, the next one popped up: Networking with Kubernetes.

The crawler and all other Microservices of the project need the rabbitmq-queue to receive and publish their tasks. In docker swarm, the other services are reached via their name that is specified in the docker-compose-file (e.g. rabbitmq). This can't be done so simple with Kubernetes. A huge amount of time went into finding the reason, why the networking from pod to pod did not work. To overcome the issue, we went ahead and renamed every descriptor in the rabbitmq-file to `rabbitmq`. This did the trick.



Despite looking like it worked, this wasn't the case at all. Problem was, that our crawler and services dropped the connection all the time again. This might be overcome by using a so called RabbitMQ Cluster Operator, which is currently being developed by the RabbitMQ-Team and is currently in the beta-state. This operator should not only make the configuration and deployment of the service easier, it does also provide a bunch of additional tools for the deployment, like mirroring and replication, more or less out of the box. Using this operator should yield us a more stable and reliable message-broker, make the discovery for the other services easier and should not result in a connection-loss all the time. Unfortunately we didn't have any time left to get this implemented or tried out.

Conclusion

This whole journey made clear that Kubernetes has an immense potential. With an incredible amount of customization and out-of-the-box-tools like a LoadBalancer, Ingress, etc. this is clearly the tool to go to when deploying a large-scale-application. But this all comes with a downside: At the beginning of a project or when trying to deploy it, a huge amount of time goes into configuring the Kubernetes-deployments, which can be a problem when starting out for example and trying to deploy a MVP (minimal viable product). Nonetheless it's an investment into the future of the project and might turn out in the long-run.

Useful links

[Kubernetes.io Documentation](#)

[Kubernetes Commands](#)

Docker

RabbitMQ Kubernetes Operator