Wojciech Krzywiec Follow

Apr 5, 2020 · 14 min read · Listen

Save

Deployment of multiple apps on Kubernetes cluster — Walkthrough

With this blog post I would like to show you how you can deploy couple applications on minikube (local Kubernetes) cluster.

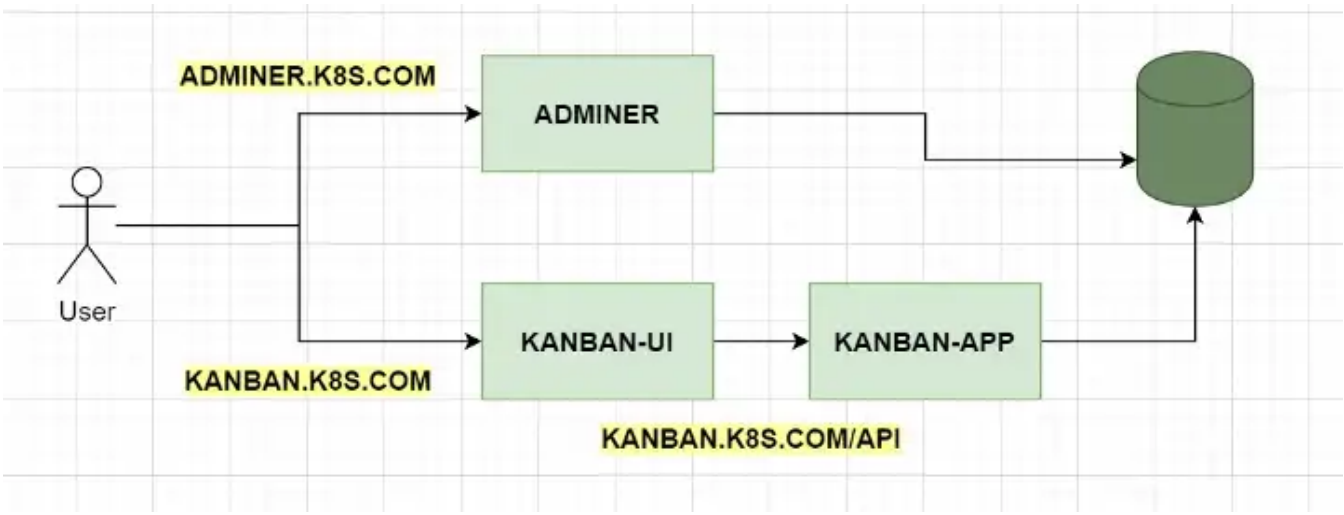
Photo by Joseph Barrientos on Unsplash

This is part one of my new series where I compare how to run applications on Kubernetes cluster using 3 approaches:

- kubectl (this one),
- Helm — *How to deploy application on Kubernetes with Helm*,
- Helmfile — *How to declaratively run Helm charts using helmfile*.

Architecture

Before making hands dirty let's see the overall architecture that we want to deploy:



It's based on my previous project — kanban-board, and include 3 services:

- database,
- backend service (*kanban-app,* written in Java with Spring Boot)
- and frontend (*kanban-ui,* written with Angular framework).

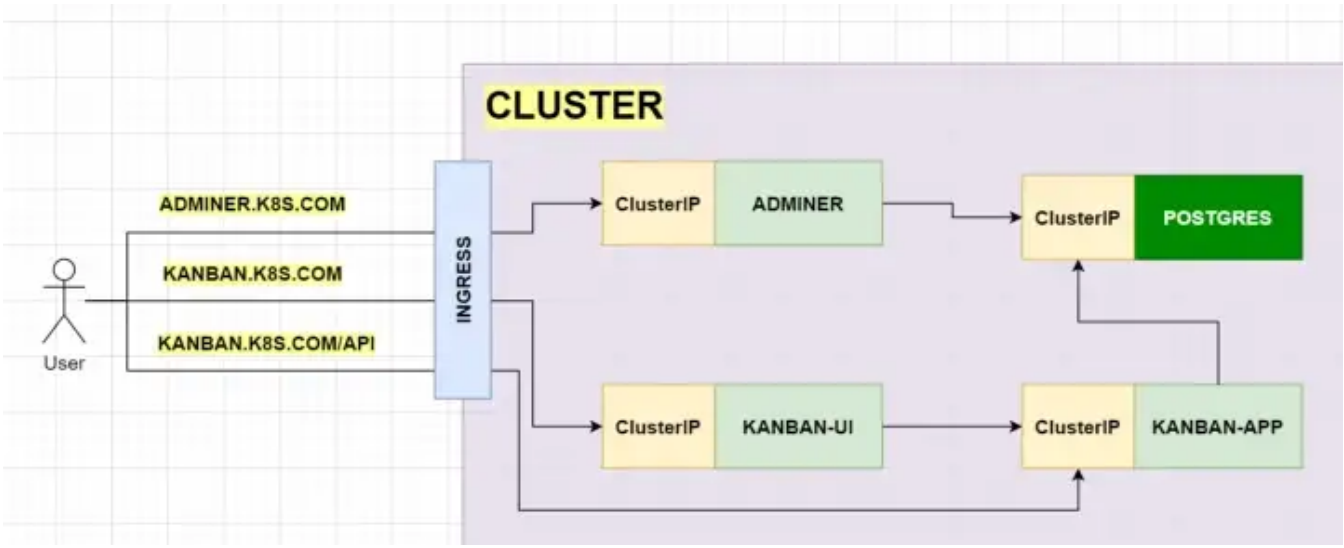
Apart from them I want to deploy the Adminer — UI application to get inside a database.

To enter one of these two UI apps user will need to type one of following URLs in the web browser:

- *kanban.k8s.com*
- *adminer.k8s.com*

The picture above is simplified, just for you to understand the main idea behind this project. Unfortunately it doesn't contain any information of what kind of

Kubernetes Objects we need to create.



If you don't know what some of these objects are, like *Ingress* or *ClusterIP*, don't worry. I'll explain all of that in a minute 😉.

Install Docker, kubectl & minikube

First you need to install all necessary dependencies. Here are links to official documentations which are covering most of popular OSes:

- Docker (container deamon),
- kubectl (a CLI tool to interact with cluster),
- minikube (locally installed, lightweight *Kubernetes* cluster).

Start minikube

Once you've got everything installed you can start the *minikube* cluster by running the CLI command in terminal:

```
$ minikube start
😄  minikube v1.25.2 on Ubuntu 20.04 (amd64)
✨  Automatically selected the docker driver
👍  Starting control plane node minikube in cluster minikube
🚜  Pulling base image ...
🔥  Creating docker container (CPUs=2, Memory=2200MB) ...
🐳  Preparing Kubernetes v1.23.3 on Docker 20.10.12 ...
    ▪ kubelet.housekeeping-interval=5m
    ▪ 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: default-storageclass, storage-provisioner
🏄  Done! kubectl is now configured to use "minikube" cluster and
"default" namespace by defaul
```

To check the status of the cluster:

```
$ minikube status
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
```

To check that `kubectl` is properly configured:

```
$ kubectl cluster-info
Kubernetes master is running at https://127.0.0.1:32768
KubeDNS is running at
https://127.0.0.1:32768/api/v1/namespaces/kube-system/services/kube-
dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl
cluster-info dump'.
```

Modify hosts file

To make the http://adminer.k8s.com & http://kanban.k8s.com work you need to edit the **hosts** file on your PC.

The location of it depends on the OS:

- Linux (Ubuntu)
- Windows 10
- Mac

When you find it, add following lines:

```
127.0.0.1   adminer.k8s.com
127.0.0.1   kanban.k8s.com
```

It will map your `localhost` IP address to both hostnames and makes them accessible after running the `minikube tunnel` command.

Add Adminer

Finally everything is set up and we can start with deploying applications. First one will be *Adminer* app.

In *Kubernetes* world the smallest deployable object is a **Pod**. It can hold one or more (Docker, cri-o) containers and also has some metadata information (e.g name, labels) that are needed. Sometimes *Pods* can be treated as single applications, because they usually have only one single container inside.

But we won't create *Pods* in this exercise 😜. Not directly at least 😉.

The problem with them is that if you're creating them manually you won't be able to easily scale their number. Also if your application inside the *Pod* crushes your *Pods* also crushes and there is no mechanism to restart it again.

Luckily there is a **Deployment** for help 🍾.

In order to create it for *Adminer* you need to have a file called **adminer-deployment.yaml** which is defined as follows:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: adminer
  labels:
    app: adminer
    group: db
spec:
  replicas: 1
  selector:
    matchLabels:
      app: adminer    # indicates which Pods (with which labeles) need be handled by this Deployment
  template:
    metadata:         # indicates the metadata that will be added to each Pod
      labels:
        app: adminer
        group: db
    spec:
      containers:     # defines a running container inside the Pod
        - name: adminer
          image: adminer:4.7.6-standalone
          ports:
            - containerPort: 8080   # which port of the container is exposed to the Pod
          env:
            - name: ADMINER_DESIGN
              value: pepa-linha
            - name: ADMINER_DEFAULT_SERVER
              value: postgres
          resources:
            limits:
              memory: "256Mi"
              cpu: "500m"
```

adminer-deployment.yaml hosted with ❤ by **GitHub** **view raw**

First section is responsible for defining of what kind of object we're creating (`apiVersion` , `kind`) followed by some metadata including name & labels (`metadata`).

Next section — `spec` — is called specification where we define specifications of a *Deployment:*

- `replicas` — indicates how many *Pods* of the same type will be created,

- `selector.matchLabels` —defines how *Deployment* will find *Pods* that it needs to take care of, in this case it will look for a Pod which is labeled with `app: adminer` ,
- `template.metadata` — tells what metadata will be added to each *Pod*, in this case all of them will have `labels : app: adminer , group: db .`
- `template.spec.containers` — is a list of containers that will be inside a *Pod*. In this case I put only one container, which is based on `adminer:4.7.6-standalone` Docker image and exposes `containerPort: 8080` . Moreover with `env` section we inject environment variable to the container to configure *Adminer UI* (full documentation can be found here). And finally we decide how much RAM and CPU an will require.

Now you can run following command in the terminal:

```
$ kubectl apply -f adminer-deployment.yaml
deployment.apps/adminer created
```

To check if everything is ok you can run:

```
$ kubectl get deployments
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
adminer   1/1     1            1           30s

$ kubectl describe deployment adminer
... many details about the Deployment ...

$ kubectl get pods
NAME                       READY   STATUS    RESTARTS   AGE
adminer-994865d4b-kqck5    1/1     Running   0          24m

$ kubectl describe pod adminer-994865d4b-kqck5
... many details about the Pod ...
```

> *Great! It worked! But there is a problem. How to open the Adminer page?*

To handle this problem we need to use another type of *Kubernetes* object — **Service**.

Per design *Kubernetes* is assigning the IP for each Pod, which might be problematic, because *Pods* don't live forever. Actually they are constantly created and deleted, all

the time. And for each new *Pod* new IP is assigned. And that's creates some kind of networking hell, because other applications inside the cluster would need to update the IP addresses of connected with *Pods* every time new instance is created.

Luckily *Services* are to the rescue. They solve that problem by having a single DNS name for all *Pods* handled by the *Deployment*. So no matter what IP address *Pod* have, all applications are pointing to the *Service* which do all the job - finding the right *Pod*. Plus *Services* are taking care of load balancing of the traffic if there are more than *Pod* replicas.

To create such object add new YAML file with Service definition **adminer-svc.yaml**:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: adminer
  labels:
    group: db
spec:
  type: ClusterIP
  selector:            # indicates into which pods ClusterIP provides access
    app: adminer
  ports:
    - port: 8080         # port exposed outside ClusterIP
      targetPort: 8080   # port exposed by Pod
```

adminer-svc.yaml hosted with ❤ by **GitHub** **view raw**

This one is a little bit shorter. But like the last time there is section defining the type of the object and it's metadata. Then there is a `spec` section where couple of properties are set:

- `type: ClusterIP` —indicates what type of the Service we want to deploy. There are several types, but I've decided to use **ClusterIP**. And the main reason for that is because I didn't want to expose every *Pod* outside the cluster. What *ClusterIP* does is that it exposes assigned *Pods* to other *Pod*s inside the cluster, but not outside.
- `selector` — here we say to which *Pods* this *Service* provide access, in this case it provide access to a *Pod* with `app: adminer` label.

- `ports` — indicates the mappings of the port exposed by the *Pod* to the *ClusterIP* port which will be available for other applications inside cluster.

And now we can create this Service with command:

```
$ kubectl apply -f adminer-svc.yaml
service/adminer created
```

And to check if everything is working:

```
$ kubectl get svc
NAME         TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
adminer      ClusterIP   10.99.85.149   <none>        8080/TCP   9s
kubernetes   ClusterIP   10.96.0.1      <none>        443/TCP    3m34s

$ kubectl describe svc adminer
... many details about the ClusterIP...
```

> *Okey! So am I able now to open the Adminer page?*

Of course not 😜. We need to do one more thing.

Add Ingress Controller

As it was mentioned before, *ClusterIP* exposes the app only for other apps inside the cluster. And in order to get to it from outside of it we need to use a different approach.

Here comes Ingress to the rescue, which is a gateway to our cluster. And the object that we need to create is called Ingress Controller and it's an implementation of *Ingress*.

But here is the tricky part. There are lots of *Ingress Controllers* available. Some of them are opensource, but some of them are paid one. For this project I've chosen an "official" Kubernetes Ingress Controller based on Nginx. But please be not confused with another one, also based on Nginx but created by NGINX Inc - this one is paid. Apart from these both there are also other *Ingress Controller*s available like Kong Ingress, or Traefik.

Luckily for us minikube comes with already built-in *Ingress Controller*. The only thing to do is to run following command:

```
$ minikube addons enable ingress
🌟  The 'ingress' addon is enabled
```

To make it work run following command in a separate terminal window:

```
$ minikube tunnel
✅ Tunnel successfully started

📌 NOTE: Please do not close this terminal as this process must stay alive for the tunnel to be accessible …
```

So then we can move to defining the routing rule to get inside the *Adminer* web page. Therefore we need to create an *Ingress* object defined in a file **ingress-controller.yaml**:

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-service
  annotations:
    kubernetes.io/ingress.class: nginx
spec:
  rules:
    - host: adminer.k8s.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: adminer
                port:
                  number: 8080
```

ingress-controller.yaml hosted with ❤ by GitHub view raw

As usual, first there is a definition of kind of the *Kubernetes* object we want to create. Then it's followed by `metadata` with the name of the object as usual and also add a

new section — `annotations`.

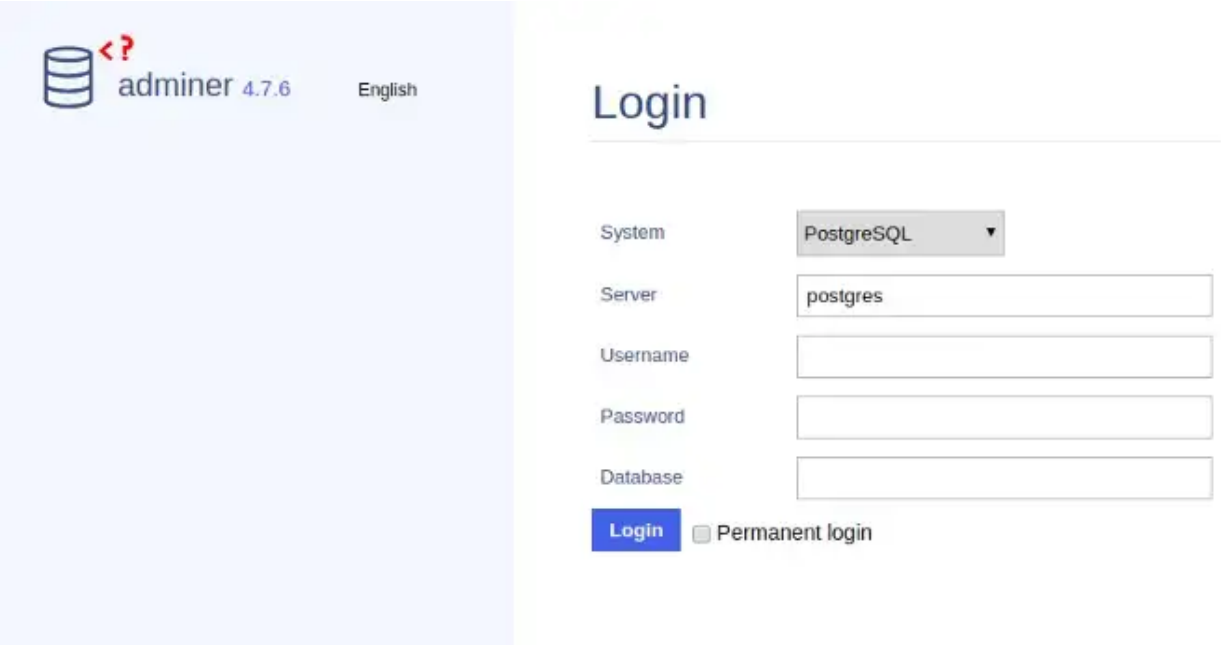
This one is very important for *Ingress* because with that we can configure its behavior. In my example, I've used the most simple one, but there are a lot more of possibilities.

And finally there is a `spec` section where we provided first rule, that all requests from the host `adminer.k8s.com` **will be routed to the ClusterIP with a name `adminer`. ClusterIP,** neither *Deployment* nor *Pod*!

After applying it into the cluster:

```
$ kubectl apply -f ingress-controller.yaml
ingress.networking.k8s.io/ingress-controller created
```

And finally after typing http://adminer.k8s.com in a web browser this page show up:



> *Awesome! But how to login to the database? Wait, but what database? We don't have any database at all!*

Add PostgreSQL database

Right, we need to set up our database. To do that we need to create another pair of *Deployment-ClusterIP*, but this time with *PostgreSQL*.

And here, again is a tricky part. Databases are not the “usual” stateless services, they store information and should not be as easily killed as “regular” *Pods* are. Even if it crashes we want to have data persisted somewhere. Therefore we need to create a space (directory) on our disk, which will be accessible by *PostgreSQL* container and will be outside the *Kubernetes* cluster.

In order to do that we need to create a new type of object called — PersistentVolumeClaim. It provides some storage located on our computer (or server) for *Pods*.

To create it once again we create a YAML file with a name — **postgres-pvc.yaml:**

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: postgres-persistent-volume-claim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 4Gi
```

postgres-pvc.yaml hosted with ❤ by **GitHub** **view raw**

Again first sections include the definition of the type of object we want to create together with some metadata. Then in the `spec` section we tell *Kubernetes* that this *Volume* has read-write access right and we want to use 4GiB of memory (I know maybe that’s too much for such small example).

And after applying it in the terminal:

```
$ kubectl apply -f postgres-pvc.yaml
persistentvolumeclaim/postgres-persistent-volume-claim created
```

And to find out if everything is ok:

```
$ kubectl get pvc
NAME       STATUS VOLUME  CAPACITY  ACCESS MODE  STORAGECLASS   AGE
postgres.. Bound  pvc-43. 4Gi       RWO          standard       40s
```

```
$ kubectl describe pvc postgres-persistent-volume-claim
... many details about the PersistentVolumeClaim...
```

Next, we should be able to create *Deployment* & *ClusterIP* for *PostgreSQL*, but first I would like to introduce new type of *Kubernetes* object, which will hold some configuration values and is called — ConfigMap.

This type of object is very useful when we want to inject environment variables to multiple containers in the *Pods*. It makes configuration of multiple *Pods/Deployments* very clean because we can have a single point of truth for our configuration. And if we decide to change it, we can do that in one place.

In this project I want keep the database configuration in *ConfigMap*, because I want to pass database config values to two Pods — one for postgres *Deployment*, and one for the backend service.

The definition of the *ConfigMap* is in the **postgres-config.yaml** file and is as follows:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: postgres-config
  labels:
    group: db
data:
  POSTGRES_DB: kanban
  POSTGRES_USER: kanban
  POSTGRES_PASSWORD: kanban
```

postgres-config.yaml hosted with ❤ by **GitHub** **view raw**

Except the usual sections — `apiVersion` , `kind` and `metadata` there is a new one instead of `spec` — `data` . It's were there are pairs of keys & values for environment variables that we will be injecting to the containers.

To create thisobject we need to run the command:

```
$ kubectl apply -f postgres-config.yaml
configmap/postgres-config created
```

And to check it we can run commands:

```
$ kubectl get configmap
NAME              DATA   AGE
postgres-config   3      2m31s

$ kubectl describe configmap postgres-config
... many details about the ConfigMap...
```

Now we can move on to the definition of PostgreSQL *Deployment* — **postgres-deployment.yaml**:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgres
  labels:
    app: postgres
    group: db
spec:
  replicas: 1
  selector:
    matchLabels:
      app: postgres
  template:
    metadata:
      labels:
        app: postgres
        type: db
    spec:
      volumes:                        # indicates which PVC are available for this Deployment
        - name: postgres-storage
          persistentVolumeClaim:
            claimName: postgres-persistent-volume-claim
      containers:
        - name: postgres
          image: postgres:9.6-alpine
          ports:
            - containerPort: 5432
          envFrom:
            - configMapRef:
                name: postgres-config
          volumeMounts:                # indicates which Volume (from spec.template.spec.volumes) should be used
            - name: postgres-storage               # name of the Volume
              mountPath: /var/lib/postgresql/data   # path inside the container
          resources:
            limits:
              memory: "256Mi"
              cpu: "500m"
```

postgres-deployment.yaml hosted with ❤ by GitHub view raw

As most of the parts where already discussed I'll skip them and move to new ones:

- `spec.template.spec.volumes` — here we're adding created PVC to the *Deployment*, so all containers inside of it will be able to use it,

- `spec.template.spec.containers[0].image` — here we specify what Docker image we want to use for our database,
- `spec.template.spec.containers[0].envFrom` — indicates from which *ConfigMap* we want to inject environment variables,
- `spec.template.spec.containers[0].volumeMounts` — tells *Kubernetes* which *Volume* to use (defined in the `spec.template.spec.volumes` section) and map it to a particular folder inside the container — basically all data inside the `mountPath` will be stored outside the cluster.

Similarly, we define the *ClusterIP* with a file **postgres-svc.yaml**:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: postgres
  labels:
    group: db
spec:
  type: ClusterIP
  selector:
    app: postgres
  ports:
    - port: 5432
      targetPort: 5432
```

postgres-svc.yaml hosted with ❤ by **GitHub** **view raw**

There is nothing new here, except the port mapping which is specific for PostgreSQL.

To create both objects we can run:

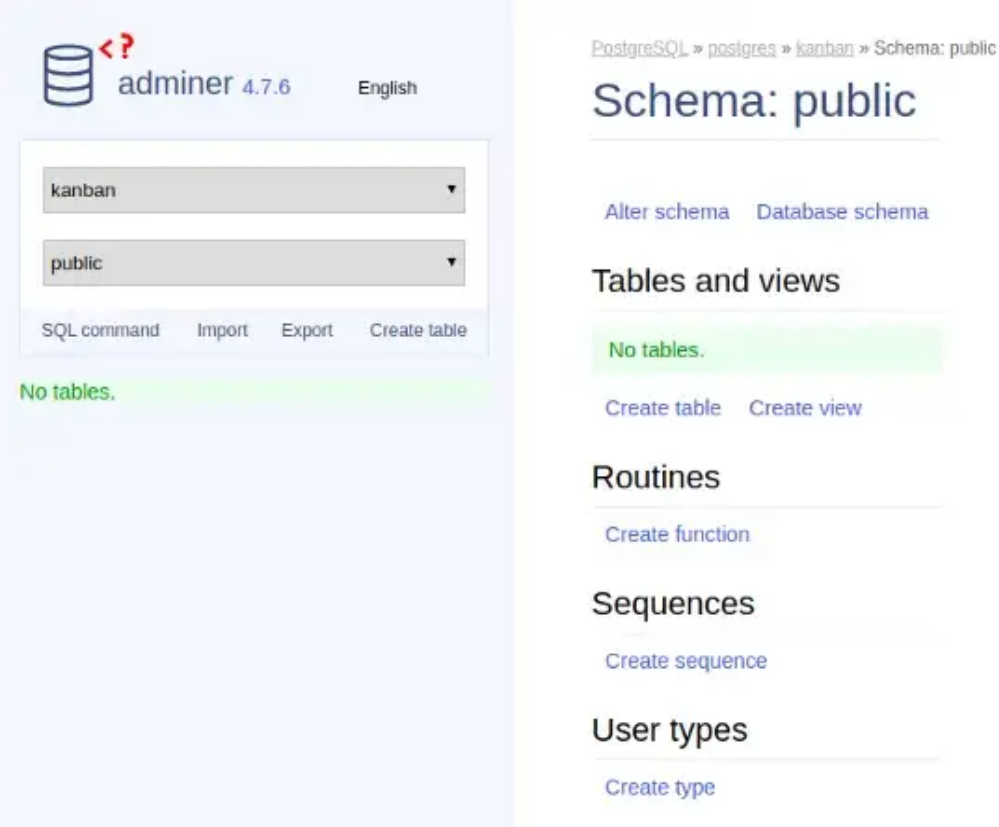
```
$ kubectl apply -f postgres-deployment.yaml
deployment.apps/postgres created

$ kubectl apply -f postgres-svc.yaml
service/postgres created
```

And now if you go to the *Adminer* once again, type following credentials:

```
System:   PostgreSQL
Server:   postgres
Username: kanban
Password: kanban
Database: kanban
```

You should be able to login to a page:



Awesome! The database is set up, so we can move on to kanban-app (backend) and kanban-ui (frontend) services.

Add kanban-app

First let's provide all necessary definitions for backend service. As it was for *Adminer,* we need also to have create *Deployment* and *Service* for backend service.

Therefore, the **kanban-app-deployment.yaml** file looks as follows:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kanban-app
  labels:
    app: kanban-app
    group: backend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: kanban-app
  template:
    metadata:
      labels:
        app: kanban-app
        group: backend
    spec:
      containers:
        - name: kanban-app
          image: wkrzywiec/kanban-app:k8s
          ports:
            - containerPort: 8080
          envFrom:
            - configMapRef:
                name: postgres-config
          env:
            - name: DB_SERVER
              value: postgres
          resources:
            limits:
              memory: "256Mi"
              cpu: "500m"
```

kanban-app-deployment.yaml hosted with ❤ by **GitHub** **view raw**

In the container specification I provided my own Docker image which I've published on Docker Hub. It exposes port `8080` and uses some of the environment variables located either in *ConfigMap* (`envFrom.configMapRef`) or from manually added environment variable only for this *Deployment*— `env` .

Next, we define the **kanban-app-svc.yaml** file:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: kanban-app
  labels:
    group: backend
spec:
  type: ClusterIP
  selector:
    app: kanban-app
  ports:
    - port: 8080
      targetPort: 8080
```

kanban-app-svc.yaml hosted with ❤ by **GitHub** **view raw**

There are no new things there in compare to previous *Services*.

To apply both definitions we need to run the commands:

```
$ kubectl apply -f kanban-app-deployment.yaml
deployment.apps/kanban-app created

$ kubectl apply -f kanban-app-svc.yaml
service/kanban-app created
```

Now you would want to test it, but in order to do so we need to configure the Ingress Controller so we can enter the Swagger UI page to check the API of the backend service.

We need to add a new host to the **ingress-controller.yaml** file so it will looks as follows:

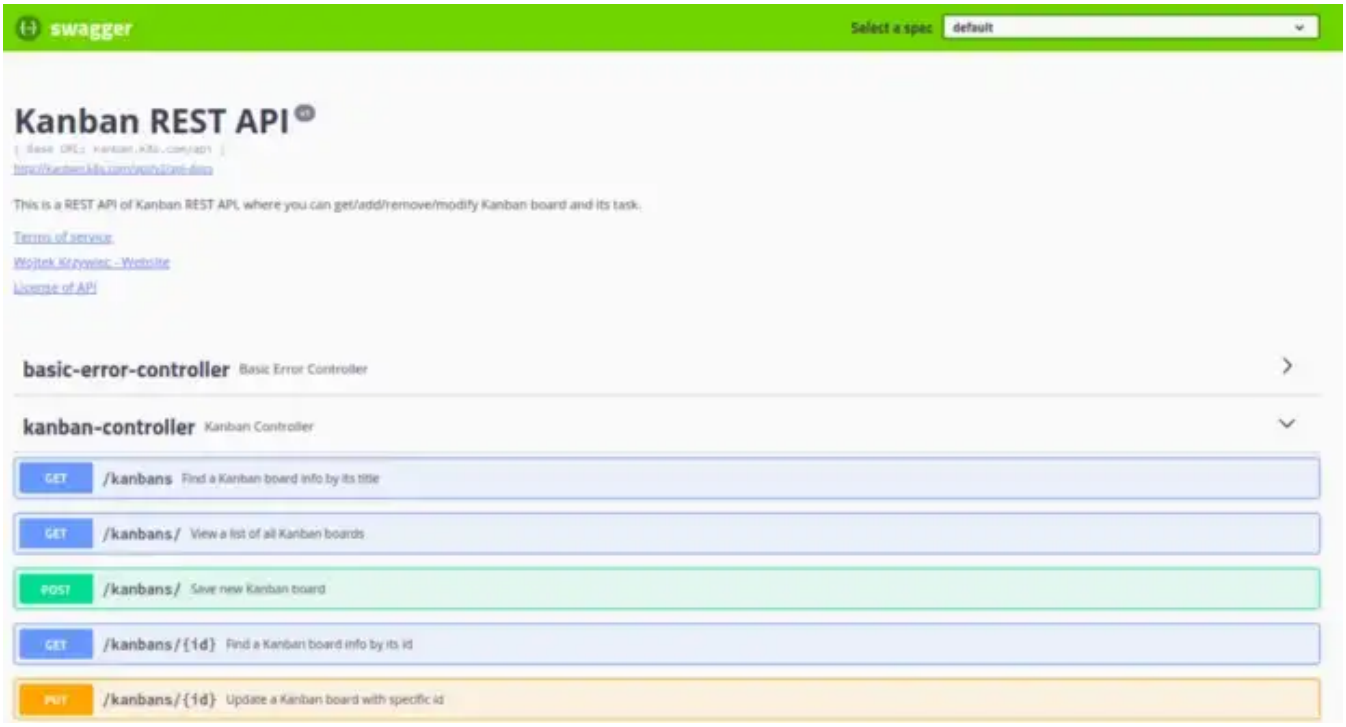
```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-service
  annotations:
    kubernetes.io/ingress.class: nginx
spec:
  rules:
    - host: adminer.k8s.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: adminer
                port:
                  number: 8080
    - host: kanban.k8s.com
      http:
        paths:
          - path: /api/
            pathType: Prefix
            backend:
              service:
                name: kanban-app
                port:
                  number: 8080
```

ingress-controller.yaml hosted with ❤ by **GitHub** **view raw**

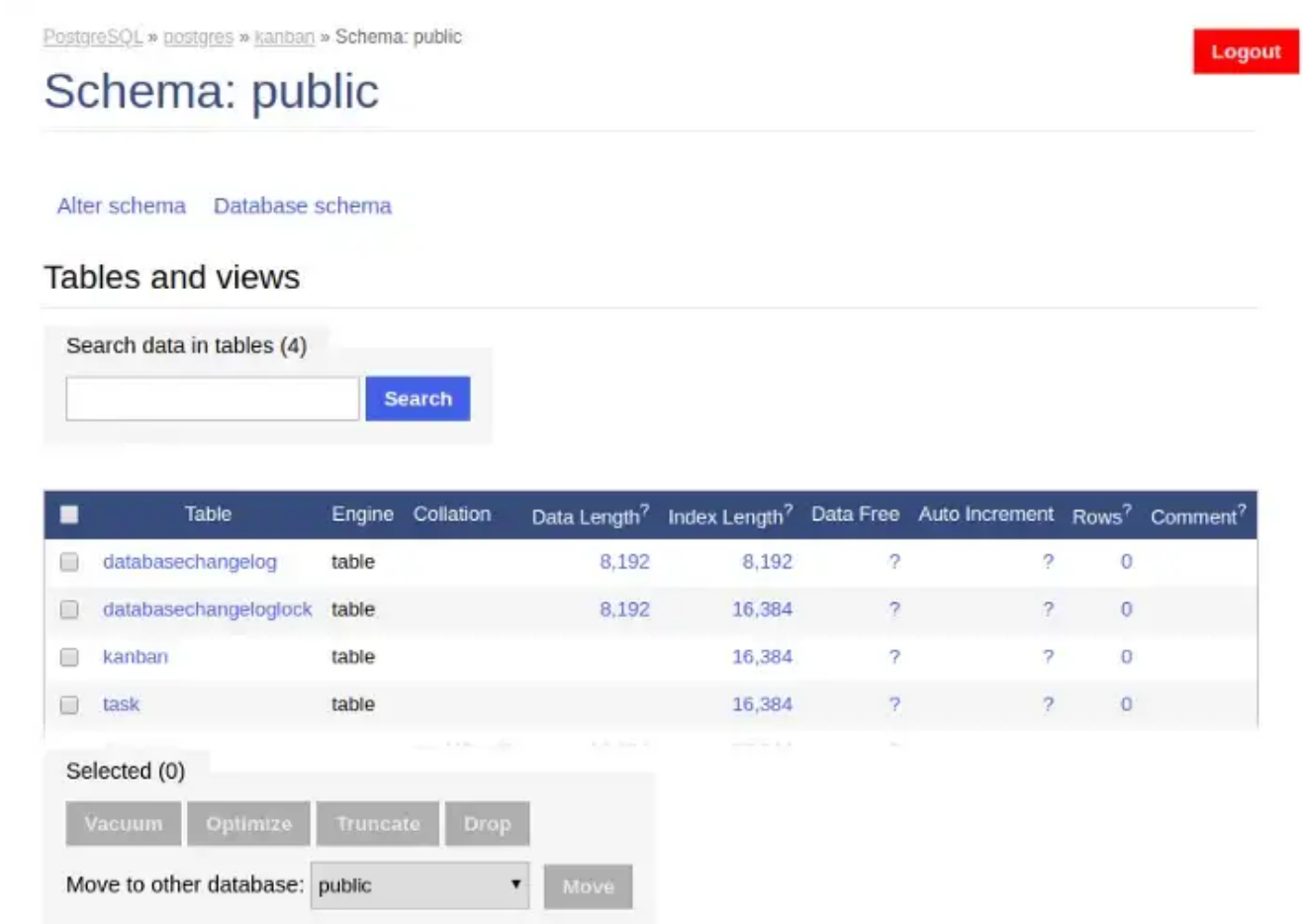
And then apply those changes to the cluster by running the command:

```
$ kubectl apply -f ingress-controller.yaml
ingress.networking.k8s.io/ingress-service configured
```

Now, if you enter the http://kanban.k8s.com/api/swagger-ui.html address in the web browser you should get the overview of the REST API that this application is providing.



You can also go to the Adminer (http://adminer.k8s.com) and check if new tables were added to the database (they were added by the Liquibase script during start up of kanban-app).



Add kanban-ui

And at last, we can add the UI application. Again, we need to define the *Deployment* and *ClusterIP*.

Here is the **kanban-ui-deployment.yaml** file

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kanban-ui
  labels:
    app: kanban-ui
    group: frontend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: kanban-ui
  template:
    metadata:
      labels:
        app: kanban-ui
        group: frontend
    spec:
      containers:
        - name: kanban-ui
          image: wkrzywiec/kanban-ui:k8s
          ports:
            - containerPort: 80
          resources:
            limits:
              memory: "256Mi"
              cpu: "500m"
```

kanban-ui-deployment.yaml hosted with ❤ by **GitHub** **view raw**

And **kanban-ui-svc.yaml** file:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: kanban-ui
  labels:
    group: backend
spec:
  type: ClusterIP
  selector:
    app: kanban-ui
  ports:
    - port: 80
      targetPort: 80
```

kanban-ui-svc.yaml hosted with ❤ by **GitHub** **view raw**

Nothing special in both files, so we can go right away to applying both of them to the cluster:

```
$ kubectl apply -f kanban-ui-deployment.yaml
deployment.apps/kanban-ui created

$ kubectl apply -f kanban-ui-svc.yaml
service/kanban-ui created
```

And again, to test it we need to expose it outside cluster. For that we need to configure *Ingress Controller*:

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-service
  annotations:
    kubernetes.io/ingress.class: nginx
spec:
  rules:
    - host: adminer.k8s.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: adminer
                port:
                  number: 8080
    - host: kanban.k8s.com
      http:
        paths:
          - path: /api/
            pathType: Prefix
            backend:
              service:
                name: kanban-app
                port:
                  number: 8080
          - path: /
            pathType: Prefix
            backend:
              service:
                name: kanban-ui
                port:
                  number: 80
```

ingress-controller.yaml hosted with ❤ by **GitHub** **view raw**

And now, if you open the address — http://kanban.k8s.com you should get this page:

You can now add Kanban boards, tasks, etc.

But one more point before the wrap up.

> *How the kanban-ui is connected with kanban-app?*

The answer to this question is in the configuration file of Nginx server, included in the Docker image of *kanban-ui* — default.conf.

```
server {
    listen 80;
    server_name kanban-ui;
    root /usr/share/nginx/html;
    index index.html index.html;

    location /api/kanbans {
        proxy_pass http://kanban-app:8080/api/kanbans;
    }

    location /api/tasks {
        proxy_pass http://kanban-app:8080/api/tasks;
    }

    location / {
        try_files $uri $uri/ /index.html;
    }
}
```

default.conf hosted with ❤ by **GitHub** **view raw**

In above example the address http://kanban-app:8080 is a DNS address of the *ClusterIP*, not the *Deployment*.

Conclusion

With this blog post I've tried to walk you through all the steps to deploy couple applications into a local *Kubernetes* cluster.

But there is one problem. How to avoid creating such great number of YAML files? And is it a single command with which we could deploy all these objects all at once?

For a second question there is a simple answer — you can run the `kubectl apply` command not on every single file but on the entire folder where they are located i.e.:

```
$ kubectl apply -f ./k8s
deployment.apps/adminer created
service/adminer created
ingress.networking.k8s.io/ingress-service created
deployment.apps/kanban-app created
service/kanban-app created
deployment.apps/kanban-ui created
service/kanban-ui created
configmap/postgres-config created
deployment.apps/postgres created
persistentvolumeclaim/postgres-persistent-volume-claim created
service/postgres created
```

But for the first question, how to avoid such boilerplate code there is no simple question. But I'll try to address it in my next post, where I'll deploy same services,

As usual here are links to my repositories, first with all Kubernetes YAML files:

wkrzywiec/k8s-helm-helmfile

Contribute to wkrzywiec/k8s-helm-helmfile development by creating an account on GitHub.

github.com

And second with the source code of kanban-app & kanban-ui:

wkrzywiec/kanban-board

This is a simple implementation of a Kanban Board, a tool that helps visualize and manage work. Originally it was first...

github.com

30th May 2022 update: *Couple of things were updated or removed to comply with the latest version of minikube (1.25) and Kubernetes (1.23) including Ingress Controller definition, editing hosts file, adding resource limits to Deployment definitions. Thanks Angelos and Arkadiusz Halicki for catching some of them!*

References

Kubernetes Documentation

Kubernetes is an open source container orchestration engine for automating deployment, scaling, and management of...

kubernetes.io

Installing Kubernetes with Mi

313 | 12 | •••

Edit This Page Minikube is a too[illegible] Kubernetes locally. Minikube runs a single-node Kubernetes...

kubernetes.io

Kubernetes Ingress 101: NodePort, Load Balancers, and Ingress Controllers

This article will introduce the three general strategies in Kubernetes for ingress, and the tradeoffs with each...

blog.getambassador.io

Studying the Kubernetes Ingress system

I have been researching how the Kubernetes Ingress system works. My use case is to setup an autoscaled Nginx cluster...

www.joyfulbikeshedding.com

Overview of kubectl

Edit This Page Kubectl is a command line tool for controlling Kubernetes clusters. kubectl looks for a file named...

kubernetes.io

kubectl apply vs kubectl create?

kubectl run = kubectl create deployment Simple, easy to learn and easy to remember. Require only a single step to make...

stackoverflow.com

Using Kubernetes to Deploy PostgreSQL

Kubernetes is an open source container orchestration system for automating deployment, scaling and management of...

severalnines.com

Dev Ops Kubernetes Cloud Computing Java Postgres

Stay tune for upcoming publication!

Subscribe to my feed to get notification for new publications

Emails will be sent to hamdi.bouhani@dealroom.co. Not you?

Subscribe