# Deploying and Scaling an App on Kubernetes with MinIO

**Cloud Computing Project**

Pagès, Louis César  Atkinson, Lukas  D'Aprea, Samuel
Chen, Shuang

Winter Semester 2020/2021

# Contents

# 1. Introduction

Kubernetes is an open-source platform for automating deployment, scaling, and management of containerized applications. It was open-sourced by Google in 2014 and is currently maintained by the Cloud Native Computing Foundation(CNCF). It provides users with a framework to run distributed systems resiliently and takes care of scaling and failover for the application. Kubernetes provides multiple practical functions, including service discovery and load balancing, storage orchestration, automated rollouts and rollbacks, automatic bin packing, self-healing and sensitive information management.

In this document, we are going to introduce some basic concepts of Kubernetes and provide a step-by-step guide of how to deploy a simple web application of Kubernetes. Object storage for the application will be provided by MinIO, which will be deployed on the same Kubernetes cluster.

# 2. Architecture and Concepts of Kubernetes

The basic structure of Kubernetes is called cluster, as depicted in Figure 2.1. A Kubernetes cluster consists of a set of nodes and a control plane. The nodes are worker machines running containerized applications. Each node hosts one or several pods, which are the smallest unit of the application workload. The control plane is responsible for managing the worker nodes and the pods in the cluster. In production environments, the control plane usually runs across multiple computers and a cluster usually runs multiple nodes, providing fault-tolerance and high availability.

In the following, we will provide an overview of control plane components, briefly touch upon node components and cluster addons, explain service discovery in the cluster, and present basic Kubernetes configuration objects.

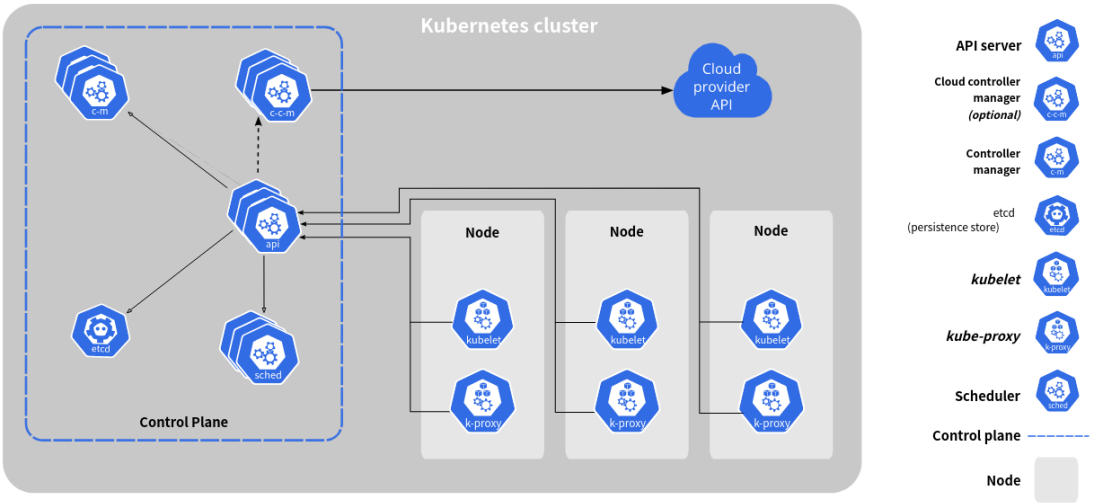


Figure 2.1.: Kubernetes Cluster[12]

## 2.1. Control Plane Components

Control Plane Components in the Kubernetes cluster are in charge of making global decisions about the cluster, and detecting and responding to cluster events. Control

plane components include API-server, persistent store, kube-scheduler, kube-controller-manager, and cloud-controller-manager [12].

The API server is the front end for the Kubernetes control plane, which is responsible for validating and configuring data for the api objects such as pods, services, replicationcontrollers and so on. The main implementation of a Kubernetes API server is kube-apiserver. It is designed to scale horizontally to deploy more instances. Users are allowed to run several instances of kube-apiserver to balance traffic.

Etcd is a consistent and highly-available key value store used for all cluster data. All Kubernetes objects are stored on etcd. In order to avoid losing master nodes and recover under disaster scenarios, it is important to periodically backup the etcd cluster data.

Kube-scheduler is the control plane component in charge of node scheduling. It monitors newly created Pods or nodes that are not assigned and will select a node for them to run on. In the process of scheduling, it usually takes a variety of factors into account, including resource requirements, hardware or software constraints, inter-workload interference, data locality, and deadlines.

Kube-controller-manager runs controller processes. The controller in Kubernetes include node controller, replication controller, endpoints controller, and service/token controllers. In theory, each controller is a separate process. However, to reduce complexity, they are all compiled into a single binary and run in a single process.

Cloud-controller-manage is responsible for embedding cloud-specific control logic. It allows users to link their cluster into the cloud provider's API, and separates out the components interacting with that cloud platform from components interacting with users' own cluster.

## 2.2. Node Components

Node components, including kubelet, kube-proxy, and container runtime, are responsible for maintaining running pods and providing the Kubernetes runtime environment. Kubelet is an agent running on each node in the cluster. Its main function is to make sure that containers are running in a Pod. Kube-proxy is a network proxy that implements part of the Kubernetes' service concept. The container runtime is a software used to run containers. Kubernetes supports several container runtimes, including Docker, containerd, and any implementation of the Kubernetes Container Runtime Interface (CRI).

## 2.3. Addons

Addons use Kubernetes resources (DaemonSet, Deployment, etc) to implement cluster features. Examples of Addons are dashboard, container resource monitoring, cluster-level logging and so on.

## 2.4. Service Discovery in Kubernetes

Within a Kubernetes cluster, service discovery can be performed via DNS.[4]

The `name` of a service corresponds directly to a hostname. For pods, a hostname is synthesized since there might be multiple pods within a deployment or stateful set. For a pod in a stateful set, an index is appended to the set name. E.g. the pods in the stateful set `myset` will be named `myset-0`, `myset-1`, and so on. Pod replicas in a deployment instead append a random hash to their hostname.

The fully qualified domain name (FQDN) of a pod or service consists of the cluster domain suffix (e.g. `cluster.local`), a type (e.g. `svc`), the namespace (e.g. `default`), the deployment name, and the hostname. So a container named `foo` in a stateful set `bar` would by default be accessible within the cluster as `foo.bar.default.svc.cluster.local`. Lookup is also possible without a FQDN (e.g. looking up a service with name `some-service`).

Pods can have different DNS policies (field `dnsPolicy` in a pod spec). The default setting `ClusterFirst` enables lookup of such FQDNs in Kubernetes' DNS service. Alternative settings like `Default` inherit the node's DNS setting, or select manual DNS configuration with `None`.

## 2.5. Kubernetes objects

Kubernetes objects are used to set up the Kubernetes environment. These objects represent persistent entities which have a functionality and serve a role in the Kubernetes environment.

They can be created, updated, deleted by a so called configuration file, in a `.yaml` file. Using such a file makes it possible to implement *infrastructure as code*. With `kubectl apply`, the cluster can be brought into the state specified by the configuration file. It is also possible to interact with the objects directly e.g. with `kubectl set`. This is discussed in detail in section 3.3 and section 6.

For each object, essential fields must be included in this configuration file in order to a have meaningful configuration. Common fields include the `apiVersion`, `kind`, `metadata`, and `spec`.

`apiVersion` This is the revision of the Kubernetes API specification for that object. Different versions might be support different object properties. API objects are versioned independently from the Kubernetes controller.

`kind` The type of the object, e.g. `Service` or `Pod`.

`metadata` A description of the object itself (i.e. name, labels). This description is mostly useful for organizing configuration objects, but some fields can also serve as configuration parameters.

`spec` This sub-object describe the state of the object, its characteristics. For example, a `Pod` object might list the containers that shall run within the Pod.

In the following, we describe the most important types of Kubernetes configuration objects that are relevant for our project.

### 2.5.1. Pod[1]

This is the smallest that can be created in a Kubernetes environment. It hosts at least one container and serves as a logical platform for its functionality. The container must be specified, under the `spec` field using `container`. Then, container image of the app is declared with `image`. It is most of the time not instantiated individually but rather within the declaration of a workload manager object such as `StatefulSet` or `Deployment`.

### 2.5.2. Deployment[2]

Deployment allows us to describe the main characteristics of an application deployment like the number of pods, the container and image to use by the pods. The main idea is to make it easier to deploy the application and to apply updates. Most important characteristic of this object is the ability to scale our application by adjusting the number of pods according to the workload.

### 2.5.3. StatefulSet[3]

This object is similar to a deployment object. The difference is that pods created here can only have unique identity meaning they have persistent identifier, potentially persistent storage, and cannot be treated interchangeably. This configuration is used mostly if we need persistent storage for our application.

[1] https://kubernetes.io/docs/reference/kubernetes-api/workloads-resources/pod-v1/
[2] https://kubernetes.io/docs/reference/kubernetes-api/workloads-resources/deployment-v1/
[3] https://kubernetes.io/docs/reference/kubernetes-api/workloads-resources/stateful-set-v1/

### 2.5.4. Service[4]

A Service allows to export our application to the (internal) network. It handles the connections and therefore the workload going to our app by distributing requests to the pods. As pods are not persistent resources, Service is always aware of their state and knows how and when to send them requests.

A service is not directly represented by a physical resource such as a container. Instead, it makes the services provided by one or more containers discoverable via some hostname and port, as discussed in section 2.4.

The different types of services are `NodePort`, `ClusterIP`, and `LoadBalancer` depending on how the service can be accessed. Later, we will use the `NodePort` type which exposes a port on all cluster nodes on which a pod of the service runs. We also use the `LoadBalancer` type, which requires an external load balancer.

### 2.5.5. Ingress[5]

To expose a service to the outside, it is possible to use a Service of type NodePort, which will make the service accessible from the node's public IP address. This does not permit for load balancing, except via DNS.

A more flexible approach is to use an *ingress controller* [9]. The controller can be configured via the `Ingress` API object. This makes it possible to provide a clear external entry point into the cluster, to forward traffic to services according to the configured rules, and to perform load balancing among the pods providing a service.

The ingress is generally provided by the cloud environment on which the cluster is running, e.g. the AWS load balancer, or a separately running HAproxy instance [10]. For the purpose of this project, Nginx is used as an ingress controller [17].

[4] https://kubernetes.io/docs/reference/kubernetes-api/services-resources/service-v1/
[5] https://kubernetes.io/docs/reference/kubernetes-api/services-resources/ingress-v1/

# 3. Kubernetes setup

To use Kubernetes, we first need a cluster which we will supply via Minikube. To interact with that cluster, we need the `kubectl` tool.

We present a Minikube-based test environment because it is easy to set up and integrates various convenience tools. For local testing, it is also possible to use the Kubernetes feature of Docker Desktop [2]. For setting up a production cluster without using managed Kubernetes cloud services, Kubernetes distributions such as K3S can be used [11].

## 3.1. Install and set up kubectl

Kubectl is a Kubernetes command-line tool, which allows users to run commands against Kubernetes clusters. Kubectl provides users with the functions including deploying applications, inspecting and managing cluster resources, and viewing logs.

The steps to install kubectl on Windows are as follows. First, download release v1.20.0 or use the following curl command (URL should be on one line, without spaces):

```
curl -LO https://storage.googleapis.com/kubernetes-release/release/
 ↪  v1.20.0/bin/windows/amd64/kubectl.exe
```

Next, add the binary to the PATH.

On Ubuntu, it can be installed instead with:

```
sudo snap install --classic kubectl
```

To test correct installation, we can display to Kubernetes version:

```
kubectl version --client
```

## 3.2. Installing Minikube

Minikube runs a local Kubernetes cluster using a virtual machine or using containers [13].

For Windows users, the steps of installing Minikube are as follows.

If the Windows Package Manager is installed, use the following command to install minikube:

```
winget install minikube
```

If the Chocolatey Package Manager is installed, use the following command:

```
choco install minikube
```

Otherwise, you should firstly download and run Windows installer.

For Linux users, there are three download options.

Binary download:

```
curl -LO
 ↪  https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
sudo install minikube-linux-amd64 /usr/local/bin/minikube
```

Debian package:

```
curl -LO
 ↪  https://storage.googleapis.com/minikube/releases/latest/minikube_latest_amd64.deb
sudo dpkg -i minikube_latest_amd64.deb
```

RPM package:

```
curl -LO
 ↪  https://storage.googleapis.com/minikube/releases/latest/minikube-latest.x86_64.rpm
sudo rpm -ivh minikube-latest.x86_64.rpm
```

After installation, from a terminal with administrator access (but not logged in as root), run the following command to start minikube:

```
minikube start
```

## 3.3. Interacting with the cluster

If you already have kubectl installed, you can now use it to access your new cluster:

```
kubectl get pods -A
```

Minikube bundles the Kubernetes Dashboard, which allows you to get easily acclimated to your new environment. The dashboard provides a graphical alternative to the `kubectl` command line tool. An example of this is shown in section 6.5. The following command will start the dashboard and open it in a browser.

```
minikube dashboard
```

We can now deploy a simple example. The following commands will create a new deployment that runs the specified image, then create a service that makes this container accessible on port 8080 of the Minikube virtual machine:

```
kubectl create deployment hello-minikube --image=k8s.gcr.io/echoserver:1.4
kubectl expose deployment hello-minikube --type=NodePort --port=8080
```

The deployment will show up when you run:

```
kubectl get services hello-minikube
```

The easiest way to access this service is to let minikube launch a web browser for you:

```
minikube service hello-minikube
```

Alternatively, use kubectl to forward the port:

```
kubectl port-forward service/hello-minikube 7080:8080
```

Now, the application will be available on http://localhost:7080/

# 4. MinIO configuration

MinIO [16] is a high-performance open-source object storage with an S3-compatible API. It can handle unstructured data such as photos, videos, log files, backups, and container images with currently the maximum supported object size of 5TB. In our demo application, we use MinIO to store the images undergoing processing. In the following, we show how MinIO can be deployed on Kubernetes.

The MinIO server has two modes, standalone and distributed. In standalone mode, there is a single server process that stores data in a single directory. In this mode, the server would be started as `minio server /data`. While standalone mode is useful for local testing, it does not provide interesting features such as replication and high-availability [15].

We therefore configure MinIO in its distributed mode [3], which provides additional features. In particular, data is replicated across multiple servers in order to tolerate partial failures, thus ensuring high availability. This is similar to a software RAID-5 configuration, except on a service level rather than on a disk controller level and with different availability guarantees. As a minimal high availability configuration, we will create a cluster of four MinIO servers with one disk each. For a production deployment all servers and disks should be on different hardware systems, but for testing all will be deployed on the same node. The MinIO server might be started in distributed mode like this, with a glob-like expression that is used to create a list of domain names for the other servers in the cluster:

```
minio server http://minio-{0...3}.example.com/data
```

As the first step to writing the Kubernetes configuration, we must add a Service. Here, we create a service called `minio` that exposes port 9000, and will include all apps labeled with `app: minio`:

```yaml
---
apiVersion: v1
kind: Service
metadata:
  name: minio
  labels:
    app: minio
spec:
  clusterIP: None
```

```
  ports:
    - port: 9000
      name: minio
  selector:
    app: minio
```

To create Pods with persistent storage, we will use a `StatefulSet` as explained in section 2.5.3. The configuration is similar to a `Deployment`, but will provide predictable names for the pods and allows us to add persistent storage with `volumeClaimTemplates`. The general structure of the configuration is as follows:

```
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: minio
spec:
  selector: ...  # which Pods should be part of the Set
  serviceName: minio  # Service must exist previously
  replicas: 4
  template:
    metadata: ...
    spec: ... # MinIO container configuration
  volumeClaimTemplates: ... # storage configuration
```

The volume claim templates define persistent storage that can be mounted in containers. Here, this includes the `ReadWriteOnce` access mode, so that the volume can be mounted by a single container in read–write mode. We also apply a size limitation to 5GB. We use the following template called `data`:

```
volumeClaimTemplates:
- metadata:
    name: data
  spec:
    accessModes:
      - ReadWriteOnce
    resources:
      requests:
        storage: 5Gi
```

For the pod spec template, we create a container that runs the MinIO server in distributed mode, exposes port 9000, and mounts a volume as defined above. For distributed mode, the running MinIO servers must be able to connect to each other. Here, we have `minio` containers in a `minio` service using the default Kubernetes namespace, so that the domain name of the first server in the StatefulSet will be

`minio-0.minio.default.svc.cluster.local`, as discussed for service discovery in section 2.4. In our configuration we also set environment variables with access credentials, although these could also be provided via the Kubernetes secrets mechanism.

```
template:
  metadata:
    labels:
      app: minio
  spec:
    containers:
    - name: minio
      env:
      - name: MINIO_ACCESS_KEY
        value: "minio"
      - name: MINIO_SECRET_KEY
        value: "minio123"
      image: minio/minio
      args:
      - server
      - http://minio-{0...3}.minio.default.svc.cluster.local/data
      ports:
      - containerPort: 9000
      # Each pod in the Set gets a volume mounted based on this field.
      volumeMounts:
      - name: data
        mountPath: /data
```

Together, and after adding labels/selectors in the necessary places, we obtain the following configuration:

```
---
apiVersion: v1
kind: Service
metadata:
  name: minio
  labels:
    app: minio
spec:
  clusterIP: None
  ports:
    - port: 9000
      name: minio
  selector:
    app: minio

---
apiVersion: apps/v1
kind: StatefulSet
```

```yaml
metadata:
  name: minio
spec:
  selector:
    matchLabels:
        app: minio
  serviceName: minio
  replicas: 4
  template:
    metadata:
      labels:
        app: minio
    spec:
      containers:
      - name: minio
        env:
        - name: MINIO_ACCESS_KEY
          value: "minio"
        - name: MINIO_SECRET_KEY
          value: "minio123"
        image: minio/minio
        args:
        - server
        - http://minio-{0...3}.minio.default.svc.cluster.local/data
        ports:
        - containerPort: 9000
        volumeMounts:
        - name: data
          mountPath: /data
  volumeClaimTemplates:
  - metadata:
      name: data
    spec:
      accessModes:
        - ReadWriteOnce
      resources:
        requests:
          storage: 5Gi
```

When this configuration is written to a YAML file, it can be deployed with one command:

```
kubectl create -f minio-deployment.yaml
```

Whereas the `kubectl create` command will attempt to create new resources which will only succeed once, the configuration could also be deployed with `kubectl apply` which is an idempotent operation.

We can inspect the current state of our MinIO storage by accessing our MinIO service in the Kubernetes cluster. For that, we need to port-forward the MinIO service using the following command:

```
kubectl port-forward service/minio-service 8080:9000
```

With 8080 being the port we will access on localhost and 9000 the port of the MinIO service.

In Fig. 4.1, the MinIO user interface is shown. In the left part, the available storage buckets are listed. In the right part, the files in one bucket are listed. Here, the `incoming` bucket contains one picture that was just downloaded.

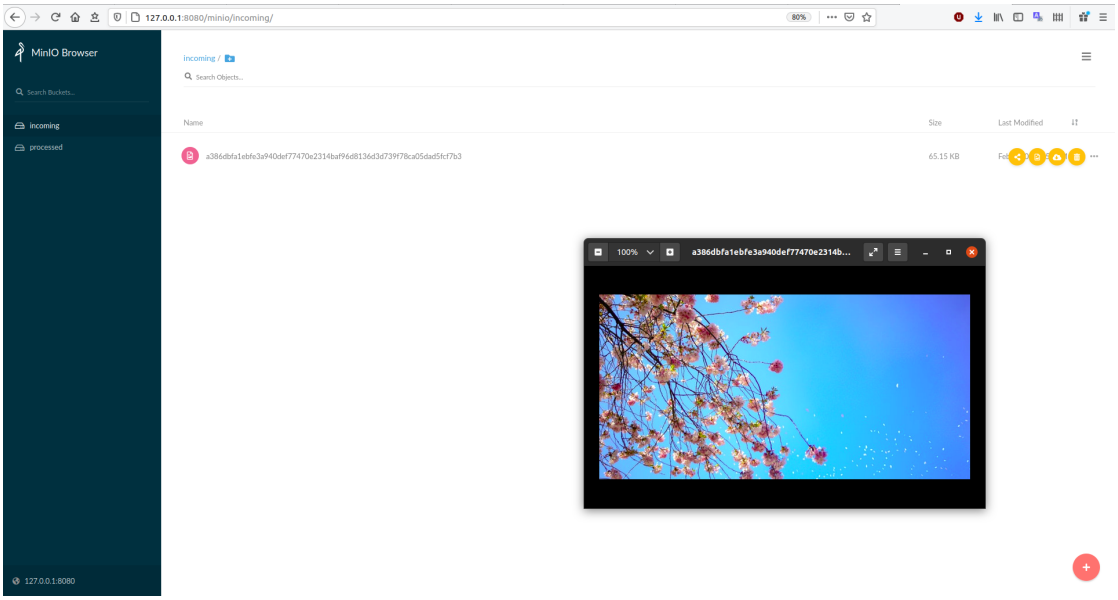


Figure 4.1.: MinIO Dashboard

# 5. Target Application

In this section, we describe the container-based application to be deployed on Kubernetes.

**Security Warning:** The application was developed for demonstration purposes on virtual machine, and does not contain any authentication. It must not be deployed on publicly accessible systems.

## 5.1. REST API

The application is a simple REST API that provides an image processing service: it can download images from the internet, and then convert them into greyscale.

Specifically, the app will provide the following routes:

**GET /**
: Returns a JSON document with a list of routes.

**POST `/incoming`**
: Instructs the application to download an image into persistent storage. The request body must be a JSON document providing an `url`:

```
{"url": "https://example.com/image.png"}
```

The response will be a JSON document that contains the name of the stored image, which will be needed for the conversion step. For example:

```
{"path":
 ↪  "/incoming/56df5bf60b0500d5e9a6e85193ca4b52b21024b8dc5ce6f8c4882bfaacea24b1"}
```

GET `/incoming/<name>`

Retrieves a previously downloaded image. This can directly use the path from the previous step.

**POST** `/processed`

Instructs the application to convert an image to greyscale. The request body must be a JSON document providing the path to an incoming image, as from the previous steps.

The response will contain a similar object with a path, this time naming a `processed` file:

```
{"path":
 ↪  "/processed/011e8ba0a6ace928861845837e84578efbf75382d8c1595bb1aa66e33b621e4f"}
```

**GET** `/processed/<name>`

Retrieves a previously processed image. This can directly use the path from the previous step.

Note: the API accepts requests with a JSON payload only when the content type is set correspondingly. With cURL, the request for processing an image would look like:

```
curl -H 'Content-Type: application/json' http://example.com/processed --data
 ↪  '{"path":"/incoming/56df5bf60b0500d5e9a6e85193ca4b52b21024b8dc5ce6f8c4882bfaacea24b1"}'
```

## 5.2. Implementation

The REST API was implemented in Python using the Flask web framework [6]. The conversion is carried out by the ImageMagick `convert` tool [1], which is invoked as a separate process. To interface with MinIO storage, the MinIO Python SDK is used [14].

While the idea of this application and early versions of the code were substantially based on the OpenFAAS demo by Ellis [5], the code was substantially restructured and extended. This involved conversion to a simpler REST API, having the API configure MinIO storage buckets itself, as well as substantial restructuring and modernization of the Python code.

The full source code for the Python application is shown in appendix A.

## 5.3. Dockerization

To deploy the application on Kubernetes, it must be first packaged as a container image. To this end, the following Dockerfile was used:

```
FROM ubuntu:20.04
WORKDIR /app

RUN apt-get update \
  && DEBIAN_FRONTEND=noninteractive apt-get install -y --no-install-recommends
 ↪  \
     python3 python3-pip imagemagick libmagic1

COPY ./image_app/requirements.txt ./
RUN python3.8 -m pip install -r requirements.txt

# copy project
COPY ./image_app ./image_app

EXPOSE 5000

CMD ["python3.8", "./image_app/__main__.py"]
```

First, this Dockerfile installs required system prerequisites such as Python, ImageMagick, and `libmagic` for determining the MIME type of the images. To keep the image small, the `--no-install-recommends` flag excludes optional dependencies. The `DEBIAN_FRONTEND=noninteractive` environment variable prevents the package installation process from expecting human interaction.

Next, the `requirements.txt` list with Python dependencies is copied into the container and installed. This includes Flask, the MinIO client SDK, as well as `requests` for making external HTTP requests, and `python-magic` for interacting with `libmagic`.

Finally, the entire application source code is copied into the container and configured as the default command when executing the container.

## 5.4. Distribution via GitLab Container Registry

For using our container image in a Kubernetes cluster, it is necessary to store the image in a container registry. Instead of creating a public image on Docker Hub, we opted for using the Container Registry feature of a self-hosted GitLab instance [7]. This has advantages such as being independent from rate limits during testing, and being able to automatically build the image in a CI pipeline.

In our case, we used the following `.gitlab-ci.yml` pipeline to automatically rebuild the image whenever new code was pushed:

```
stages:
    - build
```

```
docker-build:
  image: docker:latest
  stage: build
  services:
    - docker:dind
  before_script:
  - echo $CI_BUILD_TOKEN | docker login -u "$CI_REGISTRY_USER"
 ↪  --password-stdin  $CI_REGISTRY
  script:
    - docker build --pull -t "$CI_REGISTRY_IMAGE" .
    - docker push "$CI_REGISTRY_IMAGE"
```

There are no special tricks involved here. The CI job (which has the application source code already checked out) logs into our private registry using a temporary token, builds the image, then pushes it into the registry. The `$CI_REGISTRY_IMAGE` variable contains an image name that is automatically derived from the Git repository name. In the following, we will assume that the image name is `pkg.example.com/kube-image-processing`.

# 6. Deploying an application on Kubernetes

In this section, we deploy the container image from section 5 on our Minikube Kubernetes environment. This will involve creating a deployment with multiple pods running this container, managing secrets to access the private container registry, exposing the API as a service, and adding an ingress to make the service accessible from outside Kubernetes.

## 6.1. Creating a Deployment

Our deployment looks as follows.

```yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-deployment
  labels:
    app: web.app
spec:
  replicas: 2
  selector:
    matchLabels:
      app: web.app
  template:
    metadata:
      labels:
        app: web.app
    spec:
      containers:
        - name: web-container
          # note the private registry - need imagePullSecret
          image: pkg.example.com/kube-image-processing
          ports:
            - containerPort: 5000
          env:
            - name: minio_hostname
              value: minio:9000
            - name: minio_access_key
              value: minio
            - name: minio_secret_key
```

```
            value: minio123
      imagePullSecrets:
        - name: kube-image-processing-regcred
```

In this YAML file, a deployment named web-deployment is created, indicated by the `.metadata.name` field. The Deployment creates two replicated Pods, indicated by the `.spec.replicas` field. The `.spec.selector` field defines how the Deployment finds which Pods to manage. Here, we select a label that is defined in the Pod template (`app: web.app`).

The `template` field contains several sub-fields. The Pods are labeled `app: web.app` using the `.metadata.labels` field. The `.template.spec` field indicates that the Pods run one container: `web-container`, which runs the image `pkg.example.com/kube-image-processing`.

In Kubernetes, a Service is an abstraction which defines a logical set of Pods and a policy by which to access them. The Service abstraction enables the frontends to be able to decouple from the backends. A Service in Kubernetes is a REST object, similar to a Pod. Like all of the REST objects, you can `POST` a Service definition to the API server to create a new instance. The name of a Service object must be a valid DNS label name[18].

## 6.2. Managing `imagePullSecrets`

One important aspect of this deployment is the `imagePullSecrets` field. This would require authentification for Kubernetes to pull a private container image with a particular namespace[19]. Our project being developed in a self-hosted GitLab and for private purposes, we pull the container image created by the CI pipeline of GitLab. We need to specify the `name` of the corresponding secret we create in our Kubernetes environment with the following command:

```
kubectl create secret docker-registry kube-image-processing-regcred
 ↪  --docker-server=pkg.example.com/kube-image-processing
 ↪  --docker-username=USERNAME --docker-password=TOKEN
```

Besides the name of the secret, 3 parameters are important. The first being the docker-server wich is simply the container image namespace. A username attached to the GitLab project with a token needs to be declared for Kubernetes to be able to pull the image. A token to access the container registry can be generated on the GitLab project space.

Here, we create the secret via the command line interface, to avoid checking in the secrets in version control. However, the secret is just another configuration object like a Pod, and could also be defined in a file.

## 6.3. Defining a Service for the REST API

The following configuration is used to define a service. This is just a logical abstraction over the pods providing the REST API, and does not perform any load balancing by itself.

```
---
apiVersion: v1
kind: Service
metadata:
  name: web-service
spec:
  selector:
    app: web.app
  type: NodePort
  ports:
    - protocol: TCP
      port: 80
      targetPort: 5000
```

This specification creates a new Service object named `web-service`, which targets TCP port 5000 on any Pod with the `app: web.app` label. The controller for the Service selector continuously scans for Pods that match its selector, and then POSTs any updates to an Endpoint object also named "web-service".

## 6.4. Adding an Ingress

For a web application on Kubernetes, the user has to be able to browse to the application in their web browser. Otherwise, the application is useless. Therefore, we need to set up access to a pod from outside and this process is referred to as Ingress. Ingress is an API object that manages external access to the service in a cluster typically HTTP. Usually, the ingress also provides a load balancer[9].

In our scenario, we first have to add an ingress controller. In Minikube, a builtin NGINX ingress controller is available but must be explicitly activated:

```
minikube addons enable ingress
```

Then, define the Ingress configuration object:

```
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: web-ingress
```

```
  annotations:
    # some examples use a rewrite here, but that seems to break routing
    kubernetes.io/ingress.class: nginx
spec:
  rules:
    - http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: web-service
                port:
                  number: 80
```

In the above configuration, all traffic is forwarded to the web-service. More complicated configurations are also possible, such as mounting the service under a path prefix, or using virtual hosts, i.e. routing requests based on the domain name of the request.

## 6.5. Applying the configuration

To make these changes take effect, they can be applied. Again, we can use `kubectl create` or `kubectl apply` depending on the desired semantics:

```
kubectl create -f imageProcessingDeployment.yml
```

To check if the deployment has been successful, we can ask Kubernetes to list all resources of a particular type.

To check if a deployment has been created:

```
kubectl get deployments
```

Also, we can check if the pods have been created:

```
kubectl get pod
```

The Ingress resource will also contain the IP address under which the application can be accessed:

```
kubectl get ingress
```

We can visualize the current state of our Kubernetes deployment by launching the Kubernetes dashboard which comes bundled with Minikube:

```
minikube dashboard
```

This will open a browser tab as shown in Fig. 6.1. We can see that our cluster is up to date and running without failures. From this dashboard, we can access details, status, and logs of all Kubernetes resources such as Pods and Services.

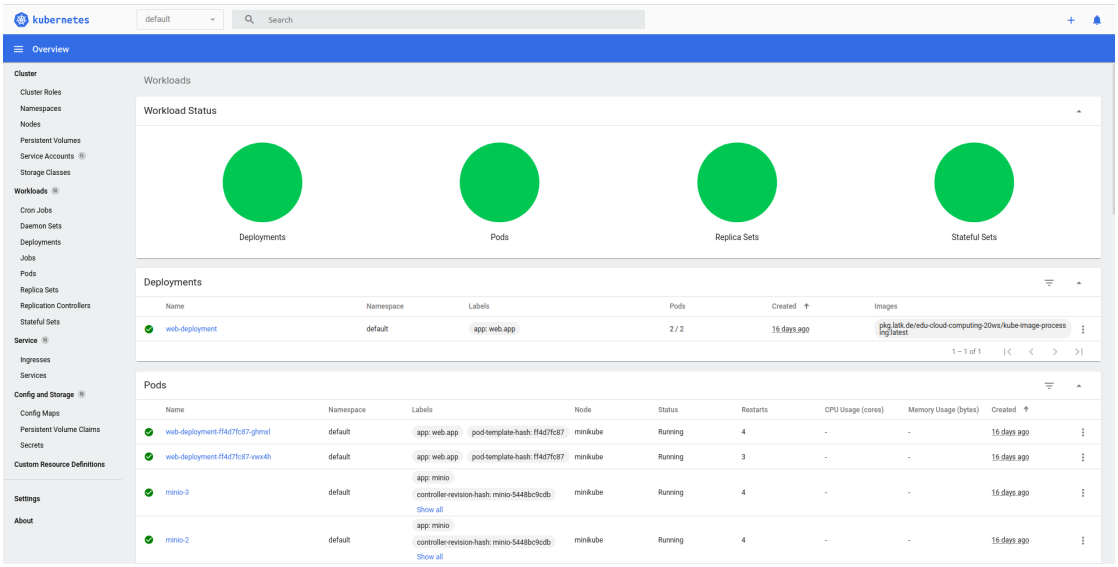


Figure 6.1.: Kubernetes dashboard

# 7. Scaling

In the previous chapters, we have created a deployment and exposed it publicly via a Service. Kubenetes also allows us to scale the application, which can efficiently keep up with user demand when the traffic increases. Scaling can be accomplished by changing the number of replicas in a `Deployment`, which will increase the number of Pods.

The number of replicas can be changed by editing the configuration and then running `kubectl apply` to bring the cluster into the specified state. Alternatively, the number of replicas can be set directly via the command line. For example, the following command will increase the number of REST API servers to three:

```
kubectl scale --replicas=3 deployment/web-deployment
```

Of course, running multiple instances of an application requires a mechanism to balance traffic. In Kubernetes, this can be achieved by `Services`, which have an integrated load-balancer to distribute network traffic to all Pods of an exposed Deployment. By continuously monitoring the running Pods using endpoints, `Services` can ensure the traffic is sent only to available Pods.

# 8. Conclusion

In this project, we have explained essential Kubernetes concepts and have applied them to deploy a simple web application, using a MinIO deployment on the same cluster as persistent object storage. Kubernetes does introduce substantial up front complexity through its configuration mechanism (e.g. separating the service concept from deployments and pods). But once that hurdle is overcome, the well designed object system has an attractive uniformity to it, and has very good support for scripting and automation.

Of course, a production Kubernetes deployment will necessarily be more involved than the presented Minikube demo. This would involve deploying a Kubernetes cluster on multiple machines, and then configuring the deployments to be distributed over those machines in an appropriate manner. For example, the different MinIO replicas should not share hardware in order to maximize availability. Given the complexity in maintaining actual hardware in a cluster, it understandable that managed Kubernetes clusters rented out by cloud service providers enjoy great popularity.

It is also worth pointing out that the presented application was fairly simple: just two kinds of services. Larger-scale systems will definitely run into substantial repeated configuration, and will therefore avoid using the raw Kubernetes configuration system. Instead, reusable components might be provides as *operators* that define new configuration resource types. Higher-level configuration systems like Helm Charts [8] use configuration templates to inject variables at configuration-time, thus making it possible to build reusable configuration packages.

# References


[1] *Convert Between Image Formats*. URL: https://www.imagemagick.org/script/convert.php (visited on 02/04/2021).

[2] *Deploy on Kubernetes*. URL: https://docs.docker.com/docker-for-windows/kubernetes/ (visited on 02/04/2021).

[3] *Distributed MinIO Quickstart Guide*. URL: https://docs.min.io/docs/distributed-minio-quickstart-guide.html (visited on 02/04/2021).

[4] *DNS for Services and Pods*. URL: https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/ (visited on 01/15/2021).

[5] Alex Ellis. *Get storage for your Severless Functions with Minio & Docker*. Jan. 22, 2018. URL: https://blog.alexellis.io/openfaas-storage-for-your-functions/ (visited on 02/04/2021).

[6] *Flask*. URL: https://flask.palletsprojects.com/en/1.1.x/ (visited on 02/04/2021).

[7] *GitLab Container Registry*. URL: https://docs.gitlab.com/ee/user/packages/container_registry/ (visited on 02/04/2021).

[8] *Helm: THe package manager for Kubernetes*. URL: https://helm.sh/ (visited on 02/04/2021).

[9] *Ingress*. URL: https://kubernetes.io/docs/concepts/services-networking/ingress/ (visited on 02/03/2021).

[10] *Ingress Controllers*. URL: https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/ (visited on 02/04/2021).

[11] *K3S – Lightweight Kubernetes*. URL: https://k3s.io/ (visited on 02/04/2021).

[12] *Kubernetes Components*. URL: https://kubernetes.io/docs/concepts/overview/components/ (visited on 12/14/2020).

[13] *Minikube*. URL: https://minikube.sigs.k8s.io/docs/ (visited on 02/04/2021).

[14] *minio on PyPI*. URL: https://pypi.org/project/minio/ (visited on 02/04/2021).

[15] *MinIO Quickstart Guide*. URL: https://docs.min.io/docs/minio-quickstart-guide.html (visited on 02/04/2021).

[16] *MinIO: Object Storage for the Era of the Hybrid Cloud*. URL: https://min.io/ (visited on 02/04/2021).

[17] *NGINX Ingress Controller*. URL: https://kubernetes.github.io/ingress-nginx/ (visited on 02/04/2021).

[18] *Service*. URL: https://kubernetes.io/docs/concepts/services-networking/service/ (visited on 02/03/2021).

[19] *Using GitLab as a container registry for Kubernetes*. URL: https://juju.is/tutorials/using-gitlab-as-a-container-registry#7-pull-our-container (visited on 02/02/2021).

# A. Python source code for REST API

```
from typing import Optional
from flask import Flask, request, make_response, Response
from minio import Minio  # type: ignore
from minio.error import S3Error  # type: ignore
from requests import get as http_get
import os
import subprocess
import io
import hashlib
import magic  # type: ignore

INCOMING_BUCKET = 'incoming'
PROCESSED_BUCKET = 'processed'

app = Flask(__name__)


@app.route('/')
def handle_index():
    return dict(routes=[
        dict(path='/incoming', method='POST'),
        dict(path='/processed', method='POST'),
        dict(path='/incoming/<name>', method='GET'),
        dict(path='/processed/<name>', method='GET'),
    ])


@app.route('/incoming', methods=['POST'])
def add_incoming():
    """
    Download the provided URLs and save them in a bucket.

    Request object:

    - url (str): the URL to download into the bucket.

    Response:

    - path (str): the path under which the file was stored
    """
```

```
    data = request.get_json()
    url = data['url']

    name = download_to_minio(MINIO_CLIENT, url)
    path = f'/incoming/{name}'

    response = make_response(dict(path=path), 201)
    response.headers['location'] = path
    return response


@app.route('/processed', methods=['POST'])
def add_processed():
    """
    Convert the named images to greyscale.

    The names must refer to objects in the INCOMING bucket.

    Request object:

    - path (path): the names of images to be converted in the INCOMING
      bucket.

    Response object:

    - path (str): the path of the object in the PROCESSED bucket
    """
    data = request.get_json()
    PREFIX = '/incoming/'
    incoming_path = data['path']

    if not isinstance(incoming_path, str):
        return make_response(dict(error=".path must be string"), 400)

    if not incoming_path.startswith(PREFIX):
        return make_response(
            dict(
                error=".path must start with `/incoming/`",
                path=incoming_path,
            ),
            400,
        )

    name = incoming_path[len(PREFIX):]

    processed_name = convert_image_to_greyscale(MINIO_CLIENT, name)
    processed_path = f'/processed/{processed_name}'

    response = make_response(dict(path=processed_path), 201)
```

```
    response.headers['location'] = processed_path
    return response


@app.route('/incoming/<name>')
def get_incoming(name: str):
    """Download an incoming image."""
    with MINIO_CLIENT.get_object(INCOMING_BUCKET, name) as mc_response:
        data: bytes = mc_response.read()

    return Response(data, mimetype=guess_mime_type(data))


@app.route('/processed/<name>')
def get_processed(name: str):
    """Download an incoming image."""
    with MINIO_CLIENT.get_object(PROCESSED_BUCKET, name) as mc_response:
        data: bytes = mc_response.read()

    return Response(data, mimetype=guess_mime_type(data))


def download_to_minio(mc: Minio, url: str) -> str:
    r = http_get(url)
    data = r.content

    name = derive_name_for_data(data)
    mc_put_object(mc, INCOMING_BUCKET, name, data)
    return name


def convert_image_to_greyscale(mc: Minio, incoming_name: str) -> str:
    with mc.get_object(INCOMING_BUCKET, incoming_name) as mc_response:
        data: bytes = mc_response.read()

    # pipe the image through the ImageMagick `convert` command
    proc = subprocess.run(
        ['convert', '-', '-colorspace', 'Gray', '-'],
        input=data,
        stdout=subprocess.PIPE,  # capture output
        check=True,  # exception on error
    )

    processed_data = proc.stdout
    processed_name = derive_name_for_data(processed_data)
    mc_put_object(mc, PROCESSED_BUCKET, processed_name, processed_data)
    return processed_name
```

```
def mc_put_object(
    mc: Minio,
    bucket_name: str,
    object_name: str,
    data: bytes,
    mimetype: Optional[str] = None,
    **kwargs,
) -> None:

    if mimetype is None:
        mimetype = guess_mime_type(data)

    mc.put_object(bucket_name,
                  object_name,
                  io.BytesIO(data),
                  len(data),
                  content_type=mimetype,
                  **kwargs)


def mc_ensure_bucket_exists(mc: Minio, bucket_name: str) -> None:
    try:
        mc.make_bucket(bucket_name)
    except S3Error as err:
        if err.code == 'BucketAlreadyOwnedByYou':
            return
        else:
            raise


def derive_name_for_data(data: bytes) -> str:
    hasher = hashlib.sha256()
    hasher.update(data)
    return hasher.hexdigest()


def guess_mime_type(data: bytes) -> str:
    return magic.from_buffer(data[:1024], mime=True)


if __name__ == '__main__':
    MINIO_CLIENT = Minio(
        os.environ['minio_hostname'],
        access_key=os.environ['minio_access_key'],
        secret_key=os.environ['minio_secret_key'],
        secure=False,
    )

    # create buckets, if they don't exist yet
```

```
    mc_ensure_bucket_exists(MINIO_CLIENT, INCOMING_BUCKET)
    mc_ensure_bucket_exists(MINIO_CLIENT, PROCESSED_BUCKET)

    # Start the web app.
    # It is visible externally,
    # but that is desired since the app runs in a Docker container
    app.run(host='0.0.0.0', port=5000)
```