

Lab - Understanding ConfigMaps, DaemonSets, Kubernetes Services, Secrets & Persistent Volume Claims



Estimated time needed: 1 hour

In this lab, you will build and deploy an application to Kubernetes, then understand and create ConfigMaps, DaemonSets, Kubernetes Services, Secrets, and further explore Volumes & Persistent Volume Claims.

Objectives

In this Practice Project, you will build and deploy a JavaScript application to Kubernetes using Docker. You will understand and create the following:

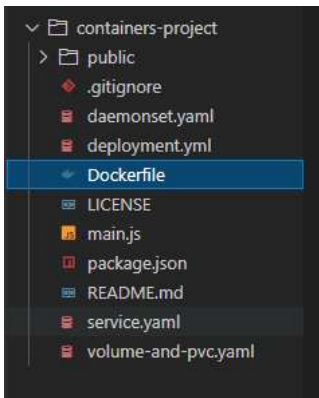
- ConfigMap
- DaemonSet
- Kubernetes Service
- Secret
- Volumes and Persistent Volume Claims

About the Dockerfile

1. Clone the repository containing the starter code to begin the project.

```
git clone https://github.com/ibm-developer-skills-network/containers-project.git
```

2. Open the Dockerfile located in the main project directory.



3. It's content will be as follows:

```
# Use an official Node.js runtime as a parent image
FROM node:14
# Set the working directory in the container
WORKDIR /app
# Copy the application files to the working directory
COPY main.js .
COPY public/index.html public/index.html
COPY public/style.css public/style.css
# Make port 3000 available to the world outside this container
EXPOSE 3000
# Run the application when the container launches
CMD ["node", "main.js"]
```

Here is an explanation of the code in it:

1. FROM node:14 specifies the base image to be used, which is an official Node.js runtime version 14.
2. WORKDIR /app sets the working directory inside the container to /app.
3. COPY main.js . copies main.js from the host to the current directory (.) in the container.
4. COPY public/index.html public/index.html copies index.html from the public directory on the host to the public directory in the container.
5. COPY public/style.css public/style.css copies style.css from the public directory on the host to the public directory in the container.
6. EXPOSE 3000 exposes port 3000 of the container to allow connections from the outside.

7. CMD ["node", "main.js"] specifies the command to run when the container launches, which is to execute node main.js.

Build and deploy the application to Kubernetes

The repository already has the code for the application as you have observed in the earlier section. We are just going to build the docker image and push to the registry.

You will be giving the name myapp to your Kubernetes deployed application.

1. Navigate to the project directory.

```
cd containers-project/
```

2. Export your namespace.

```
export MY_NAMESPACE=sn-labs-$USERNAME
```

3. Build the Docker image.

```
docker build . -t us.icr.io/$MY_NAMESPACE/myapp:v1
```

```
theia@theiadocker-nikeshkr:/home/project/containers-project$ docker build . -t us.icr.io/$MY_NAMESPACE/myapp:v1
[+] Building 78.4s (11/11) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 464B
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load metadata for docker.io/library/node:14
=> [auth] library/node:pull token for registry-1.docker.io
=> [1/5] FROM docker.io/library/node:14@sha256:a158d3b9b4e3fa813fa6c8c590b8f0a860e015ad4e59bbce5744d2f6fd8461aa
=> => resolve docker.io/library/node:14@sha256:a158d3b9b4e3fa813fa6c8c590b8f0a860e015ad4e59bbce5744d2f6fd8461aa
=> => sha256:b253aaefaa7e0671bb60008df01de101a38a045ff7bc656e3b0fbfc7c05cca5 7.86MB / 7.86MB
=> => sha256:1d12470fa662a2a5cb50378dc8ea228c1735747db410bbefb8e2d9144b5452 7.51kB / 7.51kB
=> => sha256:2cfa3fbb0b6529ee4726b4f599ec27ee557ea3dea7019182323b3779959927f 2.21kB / 2.21kB
=> => sha256:2ff1d7c41c74a25258bfa6f0b8adb0a727f84518f55f65ca845ebc747976c408 50.45MB / 50.45MB
=> => sha256:3d2201bd995cccfc12851a50820de03d34a17011dcbb9ac9fdf3a50c952cbb131 10.00MB / 10.00MB
=> => sha256:a158d3b9b4e3fa813fa6c8c590b8f0a860e015ad4e59bbce5744d2f6fd8461aa 776B / 776B
=> => sha256:1de76e268b103d05fa8960e0f77951ff54b912b63429c34f5d6adfd09f5f9ee2 51.88MB / 51.88MB
=> => sha256:d9a8df5894511ce28a05e2925a75e8a4acbd0634c39ad734fd8ba8e23d1b1569 191.85MB / 191.85MB
=> => sha256:6f51ee005deac0d99898e41b8ce60ebf250ebe1a31a0b03f613aec6bb9b83d8 4.19kB / 4.19kB
=> => extracting sha256:2ff1d7c41c74a25258bfa6f0b8adb0a727f84518f55f65ca845ebc747976c408
=> => sha256:5f32ed3c3f278edda4fc571c880b5277355a29ae8f52b52cdf865f058378a590 35.24MB / 35.24MB
=> => sha256:0c8cc2f24a4dcb64e602e086fc9446b0a541e8acd9ad72d2e90df3ba22f158b3 2.29MB / 2.29MB
=> => sha256:0d27a8e86132900574c6766fba946d48e20d2c8e964e873de352603f22c4ceb 450B / 450B
=> => extracting sha256:b253aaefaa7e0671bb60008df01de101a38a045ff7bc656e3b0fbfc7c05cca5
```

4. Push the tagged image to the IBM Cloud container registry.

```
docker push us.icr.io/$MY_NAMESPACE/myapp:v1
```

```
theia@theiadocker-nikeshkr:/home/project/containers-project$ docker push us.icr.io/$MY_NAMESPACE/myapp:v1
The push refers to repository [us.icr.io/sn-labs-nikeshkr/myapp]
9d70a531f5e7: Pushed
69bde8258173: Pushed
f64b818c1238: Pushed
a78dd8fb16a7: Pushed
0d5f5a015e5d: Pushed
3c777d951de2: Pushed
f8a91dd5fc84: Pushed
cb81227abde5: Pushed
e01a454893a9: Pushed
c45660adde37: Pushed
fe0fb3ab4a0f: Pushed
f1186e5061f2: Pushed
b2dba7477754: Pushed
v1: digest: sha256:1da35085f4ac8c563114646c6113c2c6f3d1254c0c37c4b05a06ffaa7cba46d7 size: 3042
```

5. List all the images available. You will see the newly created myapp image.

```
ibmcloud cr images
```

```
theia@theiadocker-nikeshkr:/home/project/containers-project$ ibmcloud cr images
Listing images...
```

Repository	Digest	Namespace	Created	Size	Tag	Security status
us.icr.io/sn-labs-nikeshkr/myapp	1da35085f4ac	sn-labs-nikeshkr	5 minutes ago	350 MB	v1	-
us.icr.io/sn-labsassets/categories-watson-nlp-runtime	6b01b1e5527b	sn-labsassets	2 years ago	3.1 GB	latest	-
us.icr.io/sn-labsassets/classification-watson-nlp-runtime	dbd407898549	sn-labsassets	2 years ago	4.0 GB	latest	-
us.icr.io/sn-labsassets/concepts-watson-nlp-runtime	1a4744f10f560	sn-labsassets	2 years ago	2.2 GB	latest	-

6. Open the deployment.yml file located in the main project directory. It's content will be as follows:

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
  labels:
    app: myapp
spec:
  replicas: 1
  selector:
    matchLabels:
      app: myapp
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - image: us.icr.io/<your SN labs namespace>/myapp:v1
        imagePullPolicy: Always
        name: myapp
        ports:
        - containerPort: 3000
          name: http
      resources:
        limits:
          cpu: 50m
        requests:
          cpu: 20m

```

Here is an explanation of the code within it.

- `apiVersion: apps/v1` specifies the version of the Kubernetes API being used and the resource type (Deployment).
- `kind: Deployment` indicates that this YAML defines a Deployment object.
- `metadata` section includes metadata about the Deployment, such as its name and labels.
- `spec` section defines the desired state for the Deployment, including the number of replicas, update strategy, and pod template.
- `replicas: 1` specifies that there should be one replica of the application running.
- `selector` defines how the Deployment finds which Pods to manage, using labels.
- `strategy` specifies the update strategy for the Deployment, here using rolling updates with certain constraints.
- `template` describes the Pod template used for creating new Pods.
- `containers` lists the containers within the Pod.
- `image` specifies the Docker image to use for the container.
- `imagePullPolicy: Always` ensures that the latest image is always pulled from the registry.
- `name` assigns a name to the container.
- `ports` section specifies which ports should be exposed by the container.
- `resources` defines resource requests and limits for the container, such as CPU.

7. Replace `<your SN labs namespace>` with your actual SN labs namespace.

▼ Click here for the ways to get your namespace

1. Run the command `oc project`, and use the namespace mapped to your project name

```

theia@theiadocker-nikeshkr:/home/project/containers-project$ oc project
Using project "sn-labs-nikeshkr" from context named "nikeshkr-context" on server "https://c1.us-east.containers.cloud.ibm.com:443"
theia@theiadocker-nikeshkr:/home/project/containers-project$

```

2. Run the command `ibmcloud cr namespaces` and use the one which shows your sn-labs-username

```

theia@theiadocker-nikeshkr:/home/project/containers-project$ ibmcloud cr namespaces
Listing namespaces for account 'QuickLabs - IBM Skills Network' in registry 'us.icr.io'...

Namespace
sn-labs-nikeshkr
sn-labsassets

OK
theia@theiadocker-nikeshkr:/home/project/containers-project$

```

8. Apply the deployment.

```
kubectl apply -f deployment.yml
```

```
theia@theiadocker-nikeshkr:/home/project/containers-project$ kubectl apply -f deployment.yml
deployment.apps/myapp created
```

9. Verify that the application pods are running and accessible.

```
kubectl get pods
```

```
theia@theiadocker-nikeshkr:/home/project/containers-project$ kubectl get pods
NAME                READY   STATUS    RESTARTS   AGE
myapp-6c9c7b7cf9-kg4px 1/1     Running   0           15s
```

10. Start the application on port-forward:

```
kubectl port-forward deployment.apps/myapp 3000:3000
```

```
theia@theiadocker-nikeshkr:/home/project/containers-project$ kubectl port-forward deployment.apps/myapp 3000:3000
Forwarding from 127.0.0.1:3000 -> 3000
Forwarding from [::1]:3000 -> 3000
Handling connection for 3000
```

11. Launch the app on Port 3000 to view the application output.

12. You should see the message Hello from MyApp. Your app is up!.



13. Stop the server before proceeding further, by pressing CTRL + C.

Exercise 1: ConfigMap

In this exercise, you will cover how to set up a ConfigMap to manage configuration data for the myapp application.

1. The following command is the syntax (with placeholder values) for creating a ConfigMap named myapp-config.

```
kubectl create configmap myapp-config --from-literal=env-var1=value1 --from-literal=env-var2=value2
```

- This command creates a ConfigMap named myapp-config that stores environment-related configuration data for the myapp application.
- The --from-literal flag indicates that the data for the ConfigMap will be provided directly on the command line.
- It specifies key-value pairs, which define environment variables and their corresponding values within the ConfigMap.

2. As an example, you can use the following key-value pairs:

```
env-var1: server-url ; value1: http://example.com
env-var2: timeout ; value2: 5000
```

3. As per these, execute the following command to create a configmap:

```
kubectl create configmap myapp-config --from-literal=server-url=http://example.com --from-literal=timeout=5000
```

```
theia@theiadocker-nikeshkr:/home/project/containers-project$ kubectl create configmap myapp-config --from-literal=server-url=http://example.com --from-literal=timeout=5000
configmap/myapp-config created
```

- Here, the **server-url** is set to http://example.com and **timeout** is set to 5000.
- **server-url** might store the URL of an external server that the application needs to communicate with, and **timeout** might represent the maximum time the application waits for a response before timing out.

4. Verify the successful creation of the ConfigMap by executing the below command:

```
kubectl get configmap myapp-config
```

```
theia@theiadocker-nikeshkr:/home/project/containers-project$ kubectl get configmap myapp-config
NAME          DATA   AGE
myapp-config  2       21s
```

- The output here indicates the presence of a ConfigMap named myapp-config, which contains two key-value pairs and was created 26 seconds ago.

Exercise 2: DaemonSets

In this exercise, you'll create a DaemonSet to ensure that a pod runs on each node in the cluster, including the nodes where the 'myapp' pods are deployed. Creating a DaemonSet enhances the availability and fault tolerance of your application (myapp) by ensuring that it runs on every node in the cluster, providing redundancy and load distribution.

1. Open the `daemonset.yaml` file located in the main project directory. It's content will be as below:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: myapp-daemonset
  labels:
    app: myapp
spec:
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
        - name: myapp-container
          image: us.icr.io/<your SN labs namespace>/myapp:v1
          ports:
            - containerPort: 3000
              name: http
      tolerations:
        - key: node-role.kubernetes.io/master
          effect: NoSchedule
```

Given below is an explanation of the code in it:

- `apiVersion: apps/v1`: This line specifies the Kubernetes API version that this YAML file adheres to. In this case, it uses the API version for Kubernetes applications, which is `apps/v1`.
- `kind: DaemonSet`: This line specifies the Kubernetes resource type being defined in this YAML file. In this case, it is a `DemonSet`. `DemonSet` ensures that all (or some) nodes run a copy of a specific pod.

```
metadata:
  name: myapp-daemonset
  labels:
    app: myapp
```

- The above lines specify metadata about the `DemonSet`. It gives the `DemonSet` a name (`myapp-daemonset`) and attaches labels to it. Labels are key-value pairs used to organize and select subsets of objects. In this case, the label `app: myapp` is attached to this `DemonSet`.

```
spec:
  selector:
    matchLabels:
      app: myapp
```

- The above lines define the selector for the `DemonSet`. It specifies how the `DemonSet` identifies which pods it should manage. Here, it selects pods with the label `app: myapp`.

```
spec:
  containers:
    - name: myapp-container
      image: us.icr.io/<your SN labs namespace>/myapp:v1
      ports:
        - containerPort: 3000
          name: http
```

- The above lines define the specification for the containers within the pods. It specifies the name of the container (`myapp-container`), the Docker image to use, and the ports to expose. In this case, it exposes port `3000` with the name `http`.

```
tolerations:
  - key: node-role.kubernetes.io/master
    effect: NoSchedule
```

- The above lines define tolerations for the `DemonSet` pods. Tolerations allow pods to be scheduled onto nodes with matching taints. Here, it tolerates the taint with the key `node-role.kubernetes.io/master` and the effect `NoSchedule`, meaning it can be scheduled on nodes with the master role.

2. Apply `daemonset.yaml` to your Kubernetes cluster using the following command:

```
kubectl apply -f daemonset.yaml
```

```
theia@theiadocker-nikeshkr:/home/project/containers-project$ kubectl apply -f daemonset.yaml
daemonset.apps/myapp-daemonset created
```

3. After applying the `DemonSet`, verify its status to ensure that it has been successfully deployed and is running as expected. Use the following command to check the status of `DemonSets`:

```
kubectl get daemonsets
```



```
theia@theiadocker-nikeshkr:/home/project/containers-project$ kubectl get daemonsets
NAME          DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR   AGE
myapp-daemonset 6          6         6        6            6           <none>          23s
```

This command shows information about the DaemonSets in the Kubernetes cluster.

For example, in the provided output:

DESIRED : 6 indicates the desired number of DaemonSet pods.
 CURRENT : 6 signifies the current number of DaemonSet pods.
 READY : 6 represents the number of DaemonSet pods that are ready.
 UP-TO-DATE : 6 implies that all DaemonSet pods are up-to-date.
 AVAILABLE : 6 denotes that all DaemonSet pods are available for service.
 NODE SELECTOR : <none> means there is no node selector specified.
 AGE : 23s indicates the age of the DaemonSet since its creation, which is **23 seconds**.

Exercise 3: Kubernetes services

In this exercise, you will create a Kubernetes Service to expose your application within the cluster.

1. Open the `service.yaml` file located in the main project directory. It's content will be as below:

```
apiVersion: v1
kind: Service
metadata:
  name: myapp-service
spec:
  selector:
    app: myapp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 3000
  type: NodePort
```

Given below is an explanation of the code in it:

`apiVersion: v1`: This line specifies the Kubernetes API version that this YAML file adheres to. In this case, it's using the **core/v1** API version, which is the most basic Kubernetes API version.

`kind: Service`: This line specifies the Kubernetes resource type being defined in this YAML file. In this case, it's a Service. A Service in Kubernetes is an abstraction that defines a logical set of pods and defines a policy for accessing them.

```
metadata:
  name: myapp-service
```

- The above lines specify metadata about the Service, giving it a name (`myapp-service`).

```
spec:
  selector:
    app: myapp
```

- These lines define the selector for the Service, specifying which pods the Service should target based on their labels. In this case, it selects pods with the label `app: myapp`.

```
ports:
  - protocol: TCP
    port: 80
    targetPort: 3000
```

- These lines define the ports configuration for the Service. It specifies the ports to expose on the Service and where to forward the traffic. Here, it exposes port 80 on the Service and forwards traffic to port 3000 on the pods.

```
type: NodePort
```

- This line specifies the type of Service as `NodePort`. It is a Kubernetes Services that exposes the Service on a specific port of each node in the cluster for clients to access it.

2. Apply this configuration to your Kubernetes cluster.

```
kubectl apply -f service.yaml
```

```
theia@theiadocker-nikeshkr:/home/project/containers-project$ kubectl apply -f service.yaml
service/myapp-service created
```

3. Retrieve all the services present.

```
kubectl get services
```

```
theia@theiadocker-nikeshkr:/home/project/containers-project$ kubectl get services
NAME          TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)          AGE
myapp-service NodePort    172.21.26.244 <none>        80:30016/TCP     25s
```

- This command will display both the 'openshift-web-console' (already existing in the lab environment) service and the `myapp-service` that you have just created.

For example, in the provided output for the `myapp-service`:

TYPE: `NodePort`: It specifies that this service is of type `NodePort`, meaning it exposes the service on a port on all nodes in the cluster.

CLUSTER-IP: `172.21.26.244`: This is the internal Cluster IP address assigned to the service. It is used for communication within the Kubernetes cluster.

EXTERNAL-IP: `<none>`: It indicates that there is no external IP address assigned to this service. External clients cannot directly access this service from outside the Kubernetes cluster.

PORT(S): `80:30016/TCP`: This indicates that port **80** is exposed internally by the service, and externally it is accessible on port **30016** using the **TCP protocol**. The format is `externalPort/protocol`.

AGE: `25s`: It shows that the service has been running for 25 seconds since its creation.

Note : Based on the availability of Cluster-IP addresses or ports this value may differ in your lab Environment.

Exercise 4: Secrets

In this exercise, you will learn how to create and manage secrets in Kubernetes to securely store sensitive information, such as passwords, tokens, and SSH keys.

1. The following command helps create a secret named `myapp-secret`, providing key-value pairs for sensitive data:

```
kubectl create secret generic <secret-name> --from-literal=<key1>=<value1> --from-literal=<key2>=<value2> ...
```

2. For example, you can use the following key-value pairs:

```
key1: username ; value1: myuser
key2: password ; value2: mysecretpassword
```

3. As per these, run the below command to create a secret:

```
kubectl create secret generic myapp-secret --from-literal=username=myuser --from-literal=password=mysecretpassword
```

```
theia@theiadocker-nikeshkr:/home/project/containers-project$ kubectl create secret generic myapp-secret --from-literal=username=myuser --from-literal=password=mysecretpassword
secret/myapp-secret created
```

- This command creates a Secret named `myapp-secret` and populates it with key-value pairs, where `username` is set to `myuser` and `password` is set to `mysecretpassword`.

4. To verify the successful creation of the Secret, execute the following command:

```
kubectl get secret myapp-secret
```

```
theia@theiadocker-nikeshkr:/home/project/containers-project$ kubectl get secret myapp-secret
NAME          TYPE      DATA   AGE
myapp-secret  Opaque    2       22s
```

The output displays the following information about the `myapp-secret` Secret:

NAME: This column displays the name of the secret, which in this case is `myapp-secret`.

TYPE: This column specifies the type of secret. Here, it shows **Opaque**, which indicating it's a generic secret and does not have a specific type associated with it.

DATA: This column indicates the number of data entries stored within the secret. In this case, it shows **2**, indicating there are two pieces of data stored within the secret.

AGE: This column indicates how long it has been since the secret was created or last updated. In this case, it is *22 seconds**.

Exercise 5: Volumes and persistent volume claims

In this exercise, you will explore how to define volumes and persistent volume claims (PVCs) in Kubernetes to provide storage for your application.

A **PersistentVolume (PV)** is a storage resource provisioned by an administrator in the cluster that exists independently of any Pod that might use it. It is a cluster-wide resource that can be dynamically provisioned or statically defined, and has a lifecycle managed by the cluster administrator.

A **PersistentVolumeClaim (PVC)** is a request for storage by a user or a Pod which consumes **PersistentVolume(s)**. PVCs provide a way for users to request the storage resources they need. They are namespace-specific and can only request storage within their namespace, with their lifecycle managed by the user or developer who creates them.

1. Explore the file named `volume-and-pvc.yaml` that defines **both a PV and a PVC**.

2. The following code snippet creates a PV:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: myapp-volume
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /data
```

Here is an explanation of the code:

- `apiVersion: v1`: It specifies the Kubernetes API version being used.
- `kind: PersistentVolume`: It indicates that this YAML describes a `PersistentVolume` resource.
- `metadata`: It contains metadata about the `PersistentVolume`.
- `name: myapp-volume`: It specifies the name of the `PersistentVolume`, which is 'myapp-volume' in this case.
- `spec`: This defines the specification of the `PersistentVolume`.
- `capacity`: It specifies the capacity of the `PersistentVolume`.
- `storage: 1Gi`: This indicates that the `PersistentVolume` has a storage capacity of 1 gigabyte.
- `accessModes`: It defines the access modes for the `PersistentVolume`.
 - `ReadWriteOnce`: Specifies that the volume can be mounted as read-write by a single node.
 - `hostPath`: Specifies a host path volume source.
 - `path: /data`: Indicates that the `PersistentVolume` is backed by a directory on the host (/data in this case).

3. Three dashes (i.e., ---) separate the definition of the PV from the definition of the PVC.

4. The following code snippet creates a PVC:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: myapp-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

Here is an explanation of the code:

- `apiVersion: v1`: Specifies the Kubernetes API version being used for the `PersistentVolumeClaim`.
- `kind: PersistentVolumeClaim`: Indicates that this YAML describes a `PersistentVolumeClaim` resource.
- `metadata`: Contains metadata about the `PersistentVolumeClaim`.
- `name: myapp-pvc`: Specifies the name of the `PersistentVolumeClaim`, which is myapp-pvc in this case.
- `spec`: Defines the specification of the `PersistentVolumeClaim`.
- `accessModes`: Defines the access modes for the `PersistentVolumeClaim`.
 - `ReadWriteOnce`: Specifies that the volume can be mounted as read-write by a single node.
 - `resources`: Specifies the resource requirements for the `PersistentVolumeClaim`.
 - `requests`: Indicates the requested resources.
 - `storage: 1Gi`: Specifies that the `PersistentVolumeClaim` requests a storage capacity of **1 gigabyte**.

Conclusion

In this Practice Project, you started by building and deploying a Javascript application to Kubernetes using Docker. You further created and understood a `ConfigMap` to manage configuration data for the application, a `DaemonSet` to ensure that a pod runs on each node in the cluster, a `Kubernetes Service` to expose your application within the cluster, and a `Secret` to securely store sensitive information. Further, you explored how to define volumes and persistent volume claims to provide storage for your application.

Author(s)

K Sundararajan

© IBM Corporation. All rights reserved.