Lab – Understanding ConfigMaps, DaemonSets, Kubernetes Services, Secrets & Persistent Volume Claims
实验 – 理解 ConfigMaps、守护进程集、Kubernetes 服务、机密与持久卷声明



Estimated time needed: 1 hour
所需时间估计：1 小时

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.
在本实验中，您将构建并部署一个应用程序到 Kubernetes，然后理解和创建 ConfigMaps、DaemonSets、Kubernetes Services、Secrets，并进一步探索 Volumes & Persistent Volume Claims。

Objectives 目标

In this Practice Project, you will build and deploy a JavaScript application to Kubernetes using Docker.
在本实践项目中，您将使用 Docker 构建并部署一个 JavaScript 应用程序到 Kubernetes。
You will understand and create the following:
您将理解和创建以下内容：

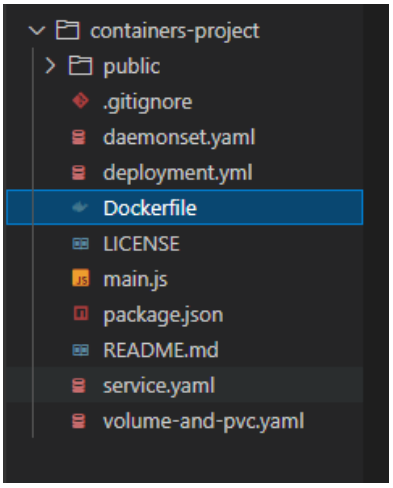
- ConfigMap
- DaemonSet
- Kubernetes Service Kubernetes 服务
- Secret
- Volumes and Persistent Volume Claims
  卷和持久卷声明

About the Dockerfile 关于 Dockerfile

1. Clone the repository containing the starter code to begin the project.
   将包含初始代码的仓库克隆到开始项目。

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

2. Open the `Dockerfile` located in the main project directory.
   打开主项目目录中的 `Dockerfile` 。



3. It's content will be as follows:
   其内容如下：

```
# 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:
以下是其中代码的解释：

1. `FROM node:14` specifies the base image to be used, which is an official Node.js runtime version 14.
   `FROM node:14` 指定要使用的基准镜像，这是一个官方的 Node.js 运行时版本 14。

2. `WORKDIR /app` sets the working directory inside the container to /app.
   `WORKDIR /app` 将容器内的工作目录设置为 /app。

3. `COPY main.js .` copies main.js from the host to the current directory (.) in the container.
   `COPY main.js .` 将 main.js 从主机复制到容器的当前目录（.）。

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.
   `COPY public/index.html public/index.html` 将 index.html 从主机上的公共目录复制到容器内的公共目录。

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.
   `COPY public/style.css public/style.css` 将主机上公共目录中的 style.css 文件复制到容器中的公共目录。

6. `EXPOSE 3000` exposes port 3000 of the container to allow connections from the outside.
   `EXPOSE 3000` 将容器的 3000 端口暴露出来，以允许外部连接。

7. `CMD ["node", "main.js"]` specifies the command to run when the container launches, which is to execute node main.js.
   `CMD ["node", "main.js"]` 指定容器启动时要运行的命令，即执行 node main.js。

Build and deploy the application to Kubernetes
构建并部署应用程序到 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.
该存储库已经包含了应用程序的代码，正如你在之前部分所观察到的。我们只需要构建 Docker 镜像并将其推送到注册中心。

You will be giving the name `myapp` to your Kubernetes deployed application.
您将给您的 Kubernetes 部署的应用程序命名为 `myapp` 。

1. Navigate to the project directory.
   导航到项目目录。

   ```
   cd containers-project/
   ```

2. Export your namespace. 导出你的命名空间。

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

3. Build the Docker image. 构建 Docker 镜像。

   ```
   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:b253aeafeaa7e0671bb60008df01de101a38a045ff7bc656e3b0fbfc7c05cca5 7.86MB / 7.86MB
 => => sha256:1d12470fa662a2a5cb50378dcdc8ea228c1735747db410bbefb8e2d9144b5452 7.51kB / 7.51kB
 => => sha256:2cafa3fbb0b6529ee4726b4f599ec27ee557ea3dea7019182323b3779959927f 2.21kB / 2.21kB
 => => sha256:2ff1d7c41c74a25258bfa6f0b8adb0a727f84518f55f65ca845ebc747976c408 50.45MB / 50.45MB
 => => sha256:3d2201bd995cccf12851a50820de03d34a17011dcbb9ac9fdf3a50c952cbb131 10.00MB / 10.00MB
 => => sha256:a158d3b9b4e3fa813fa6c8c590b8f0a860e015ad4e59bbce5744d2f6fd8461aa 776B / 776B
 => => sha256:1de76e268b103d05fa8960e0f77951ff54b912b63429c34f5d6adfd09f5f9ee2 51.88MB / 51.88MB
 => => sha256:d9a8df5894511ce28a05e2925a75e8a4acbd0634c39ad734fdfba8e23d1b1569 191.85MB / 191.85MB
 => => sha256:6f51ee005deac0d99898e41b8ce60ebf250ebe1a31a0b03f613aec6bbc9b83d8 4.19kB / 4.19kB
 => => extracting sha256:2ff1d7c41c74a25258bfa6f0b8adb0a727f84518f55f65ca845ebc747976c408
 => => sha256:5f32ed3c3f278edda4fc571c880b5277355a29ae8f52b52cdf865f058378a590 35.24MB / 35.24MB
 => => sha256:0c8cc2f24a4dcb64e602e086fc9446b0a541e8acd9ad72d2e90df3ba22f158b3 2.29MB / 2.29MB
 => => sha256:0d27a8e861329007574c6766fba946d48e20d2c8e964e873de352603f22c4ceb 450B / 450B
 => => extracting sha256:b253aeafeaa7e0671bb60008df01de101a38a045ff7bc656e3b0fbfc7c05cca5
```

4. Push the tagged image to the IBM Cloud container registry.
将标记的镜像推送到 IBM Cloud 容器注册中心。

```
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.
列出所有可用的图像。您将看到新创建的 `myapp` 图像。

```
ibmcloud cr images
```

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

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

6. Open the `deployment.yml` file located in the main project directory. It's content will be as follows:
打开位于主项目目录中的 `deployment.yml` 文件。其内容如下：

```
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.
这里是对其中代码的解释。

- `apiVersion`: apps/v1 specifies the version of the Kubernetes API being used and the resource type (Deployment).
  `apiVersion`：apps/v1 指定了所使用的 Kubernetes API 版本和资源类型（Deployment）。
- `kind`: Deployment indicates that this YAML defines a Deployment object.
  `kind`：Deployment 表示此 YAML 定义了一个 Deployment 对象。
- `metadata` section includes metadata about the Deployment, such as its name and labels.
  `metadata` 该部分包含有关 Deployment 的元数据，例如其名称和标签。
- `spec` section defines the desired state for the Deployment, including the number of replicas, update strategy, and pod template.
  `spec` 该 section 定义了 Deployment 的期望状态，包括副本数量、更新策略和 Pod 模板。
- `replicas`: 1 specifies that there should be one replica of the application running.
  `replicas`：1 指定应用应运行一个副本。
- `selector` defines how the Deployment finds which Pods to manage, using labels.
  `selector` 定义了 Deployment 如何通过标签找到要管理的 Pods。
- `strategy` specifies the update strategy for the Deployment, here using rolling updates with certain constraints.
  `strategy` 指定了 Deployment 的更新策略，此处使用带一定约束的滚动更新。
- `template` describes the Pod template used for creating new Pods.
  `template` 描述了用于创建新 Pod 的 Pod 模板。
- `containers` lists the containers within the Pod.
  `containers` 列出了 Pod 内部的容器。
- `image` specifies the Docker image to use for the container.
  `image` 指定了容器使用的 Docker 镜像。
- `imagePullPolicy`: Always ensures that the latest image is always pulled from the registry.
  `imagePullPolicy`：始终确保从注册中心拉取最新镜像。
- `name` assigns a name to the container.
  `name` 为容器分配名称。
- `ports` section specifies which ports should be exposed by the container.
  `ports` section 指定容器应暴露哪些端口。

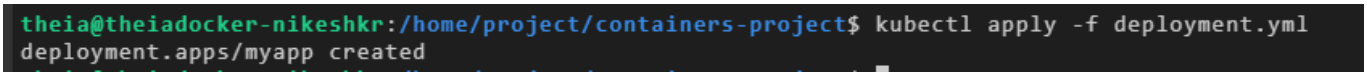
- `resources` defines resource requests and limits for the container, such as CPU.
  `resources` 定义容器的资源请求和限制，例如 CPU。

7. Replace `<your SN labs namespace>` with your actual SN labs namespace.
   将 `<your SN labs namespace>` 替换为您的实际 SN labs 命名空间。

▶ Click here for the ways to get your namespace
点击此处获取获取命名空间的途径

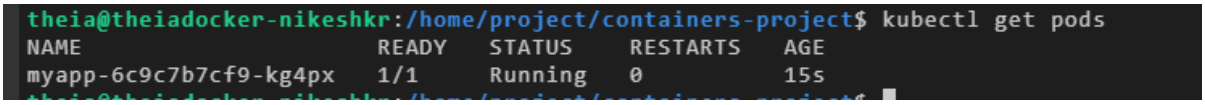
8. Apply the deployment. 应用部署。

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



9. Verify that the application pods are running and accessible.
   验证应用程序 Pod 是否正在运行且可访问。

   ```
   kubectl get pods
   ```



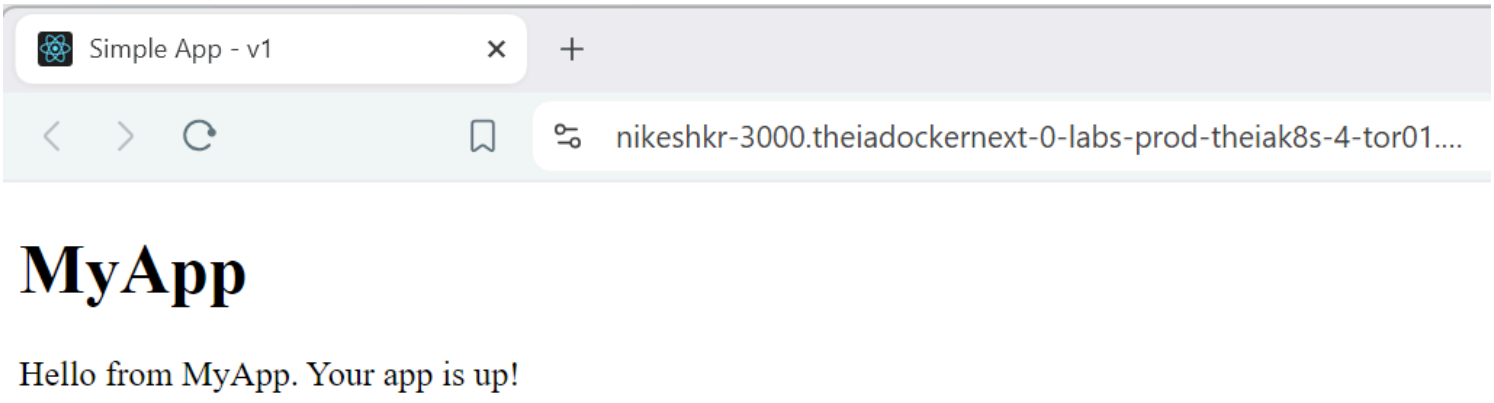
10. Start the application on port-forward:
    在端口转发上启动应用程序：

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



11. Launch the app on Port `3000` to view the application output.
11. 在端口 `3000` 上启动应用程序以查看应用程序输出。

12. You should see the message `Hello from MyApp. Your app is up!.`
    你应该看到消息 `Hello from MyApp. Your app is up!` 。



13. Stop the server before proceeding further, by pressing `CTRL + C`.
在继续之前，通过按 `CTRL + C` 停止服务器。

Exercise 1: ConfigMap 练习 1：ConfigMap

In this exercise, you will cover how to set up a ConfigMap to manage configuration data for the myapp application.
在这个练习中，你将学习如何设置一个 ConfigMap 来管理 myapp 应用程序的配置数据。

1. The following command is the syntax (with placeholder values) for creating a ConfigMap named `myapp-config`.
以下命令是创建名为 `myapp-config` 的 ConfigMap 的语法（带有占位符值）。

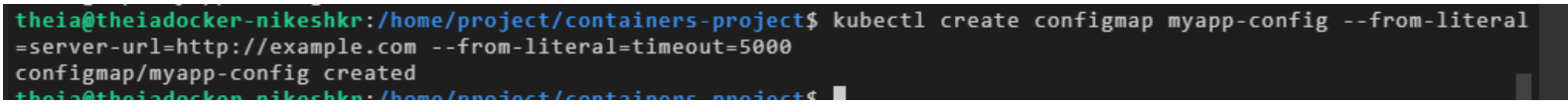
```
kubectl create configmap myapp-config --from-literal=env-var1=value1 --from-literal=env-var2=value2
```

- This command creates a ConfigMap named `myapp-config` that stores environment-related configuration data for the myapp application.
该命令创建了一个名为 `myapp-config` 的 ConfigMap，用于存储 myapp 应用程序的环境相关配置数据。
- The `--from-literal` flag indicates that the data for the ConfigMap will be provided directly on the command line.
`--from-literal` 标志表示 ConfigMap 的数据将直接在命令行上提供。
- It specifies key-value pairs, which define environment variables and their corresponding values within the ConfigMap.
它指定了键值对，这些键值对定义了 ConfigMap 中的环境变量及其相应的值。

2. As an example, you can use the following key-value pairs:
例如，您可以使用以下键值对：
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:
根据这些，执行以下命令创建一个 configmap：

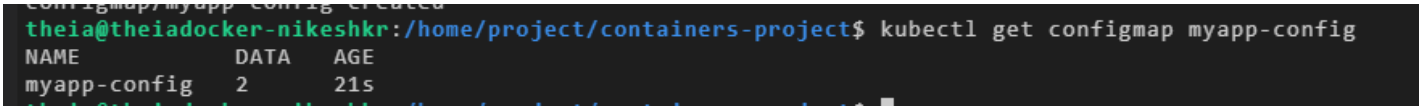
```
kubectl create configmap myapp-config --from-literal=server-url=http://example.com --from-literal=timeout=5000
```



- Here, the **server-url** is set to `http://example.com` and **timeout** is set to `5000`.
  这里，server-url 设置为 `http://example.com` ，timeout 设置为 `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.
  server-url 可能存储应用程序需要与之通信的外部服务器的 URL，timeout 可能表示应用程序在超时前等待响应的最大时间。

4. Verify the successful creation of the ConfigMap by executing the below command:
   通过执行以下命令验证 ConfigMap 的创建是否成功：

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



- The output here indicates the presence of a ConfigMap named `myapp-config`, which contains two key-value pairs and was created `26 seconds` ago.
  此处输出表明存在一个名为 `myapp-config` 的 ConfigMap，其中包含两个键值对，创建于 `26 seconds` 之前。

Exercise 2: DaemonSets 练习 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.
在这个练习中，你将创建一个 DaemonSet，以确保每个节点（包括部署了 'myapp' pod 的节点）上都运行一个 pod。创建 DaemonSet 可以通过确保它在集群的每个节点上运行来提高应用程序的可用性和容错能力（ `myapp` ），从而提供冗余和负载均衡。

1. Open the`daemonset.yaml` file located in the main project directory. It's content will be as below:
   打开位于主项目目录中的 `daemonset.yaml` 文件。其内容如下：

   ```
   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:
以下是其中代码的解释：

- `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`.
  `apiVersion: apps/v1` ： 这一行指定了该 YAML 文件遵循的 Kubernetes API 版本。在这种情况下，它使用的是 Kubernetes 应用的 API 版本，即 `apps/v1` 。

- `kind: DaemonSet`: This line specifies the Kubernetes resource type being defined in this YAML file. In this case, it is a DaemonSet. DaemonSet ensures that all (or some) nodes run a copy of a specific pod.
  `kind: DaemonSet` : 这一行指定了本 YAML 文件中定义的 Kubernetes 资源类型。在这种情况下，它是一个 DaemonSet。DaemonSet 确保所有（或部分）节点运行一个特定 Pod 的副本。

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

- The above lines specify metadata about the DaemonSet. It gives the DaemonSet 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 DaemonSet.
  以上行指定了 DaemonSet 的元数据。它为 DaemonSet 赋予一个名称（ `myapp-daemonset` ），并将其附加了标签。标签是键值对，用于组织和选择对象子集。在此情况下，标签 `app: myapp` 被附加到此 DaemonSet。

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

- The above lines define the selector for the DaemonSet. It specifies how the DaemonSet identifies which pods it should manage. Here, it selects pods with the label `app: myapp`.
  以上行定义了 DaemonSet 的选择器。它指定了 DaemonSet 如何识别它应管理的 Pod。在此，它选择带有标签 `app: myapp` 的 Pod。

```
    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`.
  以上行定义了 Pod 内容器的规范。它指定了容器名称（`myapp-container`）、要使用的 Docker 镜像以及要暴露的端口。在此情况下，它以名称 `http` 暴露端口 `3000`。

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

- The above lines define tolerations for the DaemonSet 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.
  上述行定义了 DaemonSet Pod 的容忍度。容忍度允许 Pod 被调度到具有匹配污点的节点上。在这里，它容忍具有键 `node-role.kubernetes.io/master` 和效果 `NoSchedule` 的污点，这意味着它可以在具有 master 角色的节点上被调度。

2. Apply `daemonset.yaml` to your Kubernetes cluster using the following command:
   使用以下命令将 `daemonset.yaml` 应用于您的 Kubernetes 集群：

```
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 DaemonSet, 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 DaemonSets:
   应用 DaemonSet 后，验证其状态以确保它已成功部署并按预期运行。使用以下命令检查 DaemonSet 的状态：

```
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.
此命令显示 Kubernetes 集群中 DaemonSet 的信息。

For example, in the provided output:
例如，在提供的输出中：

`DESIRED : 6` indicates the desired number of DaemonSet pods.
`DESIRED : 6` 表示期望的 DaemonSet pod 数量。
`CURRENT : 6` signifies the current number of DaemonSet pods.
`CURRENT : 6` 表示当前的 DaemonSet pod 数量。

`READY : 6` represents the number of DaemonSet pods that are ready.
`READY : 6` 表示已就绪的 DaemonSet pod 数量。
`UP-TO-DATE : 6` implies that all DaemonSet pods are up-to-date.
`UP-TO-DATE : 6` 表示所有 DaemonSet 实例都已更新到最新版本。
`AVAILABLE : 6` denotes that all DaemonSet pods are available for service.
`AVAILABLE : 6` 表示所有 DaemonSet 实例均可用于服务。
`NODE SELECTOR : <none>` means there is no node selector specified.
`NODE SELECTOR : <none>` 表示未指定节点选择器。
`AGE : 23s` indicates the age of the DaemonSet since its creation, which is **23 seconds**.
`AGE : 23s` 表示 DaemonSet 自创建以来的时间，为 23 秒。

Exercise 3: Kubernetes services
练习 3：Kubernetes 服务

In this exercise, you will create a Kubernetes Service to expose your application within the cluster.
在这个练习中，你将创建一个 Kubernetes 服务来在集群中暴露你的应用程序。

1. Open the `service.yaml` file located in the main project directory. It's content will be as below:
   打开位于主项目目录中的 `service.yaml` 文件。其内容如下：

```
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:
下面是对其代码的解释：

`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.
`apiVersion: v1`：这一行指定了该 YAML 文件遵循的 Kubernetes API 版本。在这种情况下，它使用的是 core/v1 API 版本，这是最基础的 Kubernetes API 版本。

`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.
`kind: Service`：这一行指定了该 YAML 文件中定义的 Kubernetes 资源类型。在这种情况下，它是一个 Service。在 Kubernetes 中，Service 是一种抽象，它定义了一组逻辑 Pod 并定义了访问它们的策略。

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

- The above lines specify metadata about the Service, giving it a name (`myapp-service`).
  上述行指定了服务的元数据，为其赋予名称（ `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`.
  这些行定义了服务的选择器，指定服务应针对哪些根据其标签的 Pod。在这种情况下，它选择了具有标签 `app: myapp` 的 Pod。

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

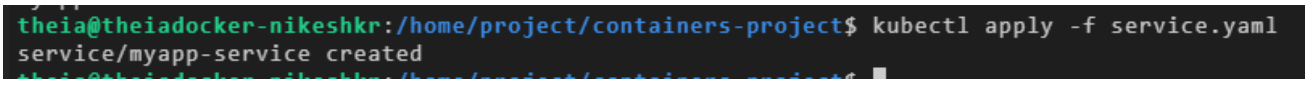
- These lines define the ports configuration for the Service. It specifies the ports to expose on the Service and where to forward the traffic. Here, it exposes port 80 on the Service and forwards traffic to port 3000 on the pods.
  这些行定义了服务的端口配置。它指定了在服务上暴露的端口以及流量转发位置。这里，它在服务上暴露端口 80，并将流量转发到 Pod 上的端口 3000。

```
  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.
  这一行指定了服务类型为 `NodePort` 。它是一个 Kubernetes 服务，在集群中每个节点的特定端口上暴露服务，以便客户端可以访问它。

2. Apply this configuration to your Kubernetes cluster.
   将此配置应用于您的 Kubernetes 集群。

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



3. Retrieve all the services present.
   检索所有现有的服务。

   ```
   kubectl get services
   ```



- 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.
  此命令将显示 'openshift-web-console'（已在实验室环境中存在）服务和您刚刚创建的 myapp-service。

For example, in the provided output for the `myapp-service`:
例如，在提供的 `myapp-service` 输出中：

`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.
`TYPE: NodePort`：它指定此服务类型为 NodePort，意味着它在集群中所有节点的端口上公开服务。

`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.
`CLUSTER-IP: 172.21.26.244`：这是分配给服务的内部 Cluster IP 地址。它用于 Kubernetes 集群内部的通信。

`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.
`EXTERNAL-IP: <none>`：它表示没有为该服务分配外部 IP 地址。外部客户端不能直接从 Kubernetes 集群外部访问此服务。

`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`.
`PORT(S): 80:30016/TCP`：这表示服务内部通过端口 80 公开，而外部可通过 TCP 协议在端口 30016 上访问。格式为 `externalPort/protocol` 。

`AGE: 25s`: It shows that the service has been running for 25 seconds since its creation.
`AGE: 25s` ： 这表明该服务自创建以来已运行了 25 秒。

Note : Based on the availability of Cluster-IP addresses or ports this value may differ in your lab Environment.
注意：根据您的实验室环境中 Cluster-IP 地址或端口的可用性，此值可能会有所不同。

Exercise 4: Secrets 练习 4：秘密

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.
在这个练习中，你将学习如何在 Kubernetes 中创建和管理密钥，以安全地存储敏感信息，例如密码、令牌和 SSH 密钥。

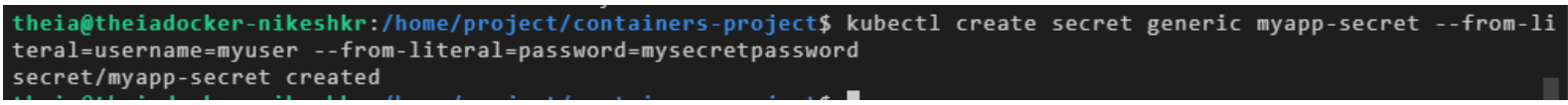
1. The following command helps create a secret named `myapp-secret`, providing key-value pairs for sensitive data:
   以下命令有助于创建一个名为 `myapp-secret` 的密钥，为敏感数据提供键值对：

   ```
   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:
   例如，你可以使用以下键值对：
   key1: `username` ; value1: `myuser`
   key2: `password` ; value2: `mysecretpassword`

3. As per these, run the below command to create a secret:
   根据这些信息，运行以下命令以创建一个密钥：

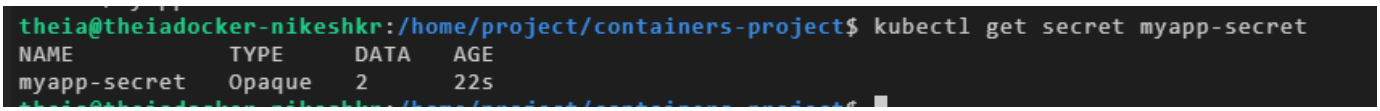
   ```
   kubectl create secret generic myapp-secret --from-literal=username=myuser --from-literal=password=mysecretpassword
   ```



- 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`.
  此命令将创建一个名为 `myapp-secret` 的密钥，并用键值对填充，其中 `username` 设置为 `myuser` ， `password` 设置为 `mysecretpassword` 。

4. To verify the successful creation of the Secret, execute the following command:
   为验证密钥的创建是否成功，执行以下命令：

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



The output displays the following information about the `myapp-secret` Secret:
输出显示了关于 `myapp-secret` 密钥的以下信息：

`NAME`: This column displays the name of the secret, which in this case is myapp-secret.
`NAME` ： 该列显示密钥的名称，此处为 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.
`TYPE` ： 该列指定密钥的类型。此处显示 Opaque，表示它是一个通用密钥，没有特定类型关联。

`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.
`DATA` ： 该列指示密钥中存储的数据条目数量。此处显示 2，表示密钥中存储了两份数据。

`AGE`: This column indicates how long it has been since the secret was created or last updated. In this case, it is *22 seconds**.
`AGE` ： 该列指示自秘密创建或最后更新以来经过的时间。在此情况下，为 22 秒*。

Exercise 5: Volumes and persistent volume claims
练习 5：卷和持久卷声明

In this exercise, you will explore how to define volumes and persistent volume claims (PVCs) in Kubernetes to provide storage for your application.
在这个练习中，你将探索如何在 Kubernetes 中定义存储卷和持久卷声明（PVC），为你的应用程序提供存储。

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.
一个 `PersistentVolume (PV)` 是由集群管理员在集群中配置的存储资源，它独立于任何可能使用它的 Pod。它是一个全局资源，可以动态配置或静态定义，其生命周期由集群管理员管理。

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.
一个 `PersistentVolumeClaim (PVC)` 是用户或消耗 PersistentVolume(s) 的 Pod 对存储的请求。PVC 提供了一种方式，让用户请求他们需要的存储资源。它们是命名空间特定的，并且只能请求其命名空间内的存储，其生命周期由创建它们的用户或开发者管理。

1. Explore the file named `volume-and-pvc.yaml` that defines **both a PV and a PVC**.
   探索名为 `volume-and-pvc.yaml` 的文件，该文件定义了一个 PV 和一个 PVC。

2. The following code snippet creates a PV:
   以下代码片段创建了一个 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:
以下是代码的解释：

- `apiVersion: v1`: It specifies the Kubernetes API version being used.
  `apiVersion: v1`：它指定了正在使用的 Kubernetes API 版本。
- `kind: PersistentVolume`: It indicates that this YAML describes a PersistentVolume resource.
  `kind: PersistentVolume`：它表示此 YAML 描述了一个 PersistentVolume 资源。
- `metadata`: It contains metadata about the PersistentVolume.
  `metadata`： 它包含有关持久卷的元数据。
- `name: myapp-volume`: It specifies the name of the PersistentVolume, which is 'myapp-volume' in this case.
  `name: myapp-volume`： 它指定了持久卷的名称，在本例中为'myapp-volume'。
- `spec`: This defines the specification of the PersistentVolume.
  `spec`： 它定义了持久卷的规范。
- `capacity`: It specifies the capacity of the PersistentVolume.
  `capacity`： 它指定了持久卷的容量。
- `storage: 1Gi`: This indicates that the PersistentVolume has a storage capacity of 1 gigabyte.
  `storage: 1Gi`： 这表示 PersistentVolume 的存储容量为 1 GB。
- `accessModes: It defines the access modes for the PersistentVolume.
  `accessModes：它定义了 PersistentVolume 的访问模式。
  - `ReadWriteOnce`: Specifies that the volume can be mounted as read-write by a single node.
    `ReadWriteOnce`： 指定该卷可以被单个节点以读写方式挂载。
  - `hostPath`: Specifies a host path volume source.
    `hostPath`： 指定了一个主机路径卷源。

- `path: /data`: Indicates that the PersistentVolume is backed by a directory on the host (/data in this case).
  `path: /data`：表示 PersistentVolume 是由主机上的一个目录支持的（在本例中为 /data）。

3. Three dashes (i.e., ---) separate the definition of the PV from the definition of the PVC.
   三个连字符（即 --- ）将 PV 的定义与 PVC 的定义分开。

4. The following code snippet creates a PVC:
   以下代码片段创建了一个 PVC：

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

Here is an explanation of the code:
以下是该代码的解释：

- `apiVersion: v1`: Specifies the Kubernetes API version being used for the PersistentVolumeClaim.
  `apiVersion: v1`：指定所使用的 Kubernetes API 版本，用于 PersistentVolumeClaim.
- `kind: PersistentVolumeClaim`: Indicates that this YAML describes a PersistentVolumeClaim resource.
  `kind: PersistentVolumeClaim`：表示此 YAML 描述了一个 PersistentVolumeClaim 资源。
- `metadata`: Contains metadata about the PersistentVolumeClaim.
  `metadata`：包含关于 PersistentVolumeClaim. 的元数据。
- `name: myapp-pvc`: Specifies the name of the PersistentVolumeClaim, which is `myapp-pvc` in this case.
  `name: myapp-pvc`：指定 PersistentVolumeClaim 的名称，此处为 `myapp-pvc` 。
- `spec`: Defines the specification of the PersistentVolumeClaim.
  `spec`：定义了 PersistentVolumeClaim. 的规范
- `accessModes`: Defines the access modes for the PersistentVolumeClaim.
  `accessModes`：定义了 PersistentVolumeClaim. 的访问模式
  - `ReadWriteOnce`: Specifies that the volume can be mounted as read-write by a single node.
    `ReadWriteOnce`：指定卷可以被单个节点以读写方式挂载。
  - `resources`: Specifies the resource requirements for the PersistentVolumeClaim.
    `resources`：指定了 PersistentVolumeClaim. 的资源需求
  - `requests`: Indicates the requested resources.
    `requests`：表示请求的资源。
  - `storage: 1Gi`: Specifies that the PersistentVolumeClaim requests a storage capacity of **1 gigabyte**.
    `storage: 1Gi`：指定 PersistentVolumeClaim 请求 1GB 的存储容量。

Conclusion 结论

In this Practice Project, you started by building and deploying a Javascript application to Kubernetes using Docker.
在这个实践项目中，你首先使用 Docker 构建并部署了一个 JavaScript 应用程序到 Kubernetes。
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.
你还创建并理解了一个 ConfigMap 来管理应用程序的配置数据，一个 DaemonSet 来确保集群中的每个节点上都运行一个 Pod，一个

Kubernetes Service 来在集群内暴露你的应用程序，以及一个 Secret 来安全地存储敏感信息。
Further, you explored how to define volumes and persistent volume claims to provide storage for your application.
此外，你探索了如何定义卷和持久卷声明来为你的应用程序提供存储。

Author(s) 作者

K Sundararajan

© IBM Corporation. All rights reserved.