## Overview

In this lab, we will use Kubernetes in a declarative fashion by writing resource manifests as YAML files that describe our desired state.

* Create a Deployment
* Make it more robust
* Expose it with a Service

[Overview](#_xn4i8df3o0mj)

[Instructions](#_a4vdcj9or4m)

[Task 0: YAML](#_q1eipr6h6290)

[Step 1: Explore a Deployment definition](#_z9ad9pzgs0f1)

[Task 1: Create a deployment](#_guqiq3ja6mzy)

[Step 1: Apply a definition](#_s71q4evhfyxl)

[Step 2: View actual state](#_6mpenovnyswm)

[Step 3: Test autohealing](#_4yavdch7id2i)

[Task 2: Add readiness probes](#_i2e82b4aksib)

[Step 1: Break the app](#_efzf5n84ceea)

[Step 2: Add Probes](#_o09t9mrjx710)

[Step 3: Break app again](#_x6hvpbmk5q6n)

[Task 3: Make a service object](#_uo3om5e9wbq)

[Step 1: Apply a service definition](#_ydzrs3cn9gv)

[Step 2: View round robin load balancing](#_2bxz6glv9bl5)

[Step 3: Add session affinity](#_t7n921b0pmc)

[Step 4: View always hitting the same Pod](#_8d4dbffam7rk)

[Step 5: Delete the pod](#_hiiy1vpwe6i4)

[Task 4: Cleanup](#_u9xvcp7wwuqd)

## 

## Instructions

Read this lab like a book, all text is there for a reason!

"→" denotes an action you must take

Use your favorite editor to edit files within the console. I suggest VI, nano, or emacs.

|  |
| --- |
| White boxes with black text denote commands and file contents |

|  |
| --- |
| Black boxes with green text denote example output |

### **Task 0: YAML**

Kubernetes resources are often described with YAML in text files. They can also be described with JSON but YAML is more human readable. YAML used to stand for Yet Another Markup Language (they rebranded to "YAML Ain't Markup Language" but I'm not buying it) and is similar to other markup languages that allow you to define nested objects.

#### **Step 1: Explore a Deployment definition**

The following is a Deployment object. Notice how YAML relies on whitespace to denote structure. One important thing to note is that these must be spaces and never tabs, this is a common source of syntax errors. More information on YAML syntax can be found here ([https://en.wikipedia.org/wiki/YAML#Syntax](https://en.wikipedia.org/wiki/YAML" \l "Syntax))

A Deployment object described in YAML

|  |
| --- |
| apiVersion: apps/v1  kind: Deployment  metadata:  name: nginx-deployment  labels:  app: nginx  spec:  replicas: 3  selector:  matchLabels:  app: nginx  template:  metadata:  labels:  app: nginx  spec:  containers:  - name: nginx  image: nginx:1.7.9  ports:  - containerPort: 80 |

Kubernetes objects each have their own schema of allowed attributes. They can easily be explored in the [kubernetes reference docs](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.19/) or using the 'explain' command.

→ Run the following command to see the valid Deployment attributes:

|  |
| --- |
| kubectl explain deployment |

→ Dig deeper by looking into the spec:

|  |
| --- |
| kubectl explain deployment.spec |

You can look at sub fields by adding '.' and the field name.

→ For example

|  |
| --- |
| kubectl explain deployment.spec.replicas |

### **Task 1: Create a deployment**

#### **Step 1: Apply a definition**

→ Create a file named deployment.yaml

|  |
| --- |
| vi deployment.yaml |

→ Type ":set paste", press **enter**. Then press **i**, paste the following:

|  |
| --- |
| apiVersion: apps/v1  kind: Deployment  metadata:  name: http-deployment  labels:  app: http  spec:  replicas: 3  selector:  matchLabels:  app: http  template:  metadata:  labels:  app: http  spec:  containers:  - name: http-server  image: acmeade/http-app  ports:  - containerPort: 80 |

→ Hit **ESC**, then type ':wq' and press **enter** to save the file.

→ Create Deployment

|  |
| --- |
| kubectl apply -f deployment.yaml |

|  |
| --- |
| deployment/http-deployment created |

#### **Step 2: View actual state**

→ Get deployment details

|  |
| --- |
| kubectl get deployment http-deployment |

Here we can see the desired state is 3 pods, under the UP-TO-DATE column, hopefully that is also true under your ACTUAL.

We can view the raw yaml of the object using the -o yaml flag. This will have an extra section called 'status', usually at the end, that shows the actual state of the object.

→ Get deployment details

|  |
| --- |
| kubectl get deployment http-deployment -o yaml |

We can even use jsonpath to grab individual elements of the response. This can be super useful when automating.

→ Get deployment details

|  |
| --- |
| kubectl get deployment http-deployment -o=jsonpath='{.status.availableReplicas}' |

#### **Step 3: Test autohealing**

→ List the pods

|  |
| --- |
| kubectl get pods |

→ Copy a Pods name from the output

→ Delete a pod!

|  |
| --- |
| kubectl delete pod <POD\_NAME> |

→ List the pods

|  |
| --- |
| kubectl get pods |

Notice a new pod was created. This is because we told kubernetes we always want 3 replicas and when it noticed we had only 2, it scheduled another.

### **Task 2: Add readiness probes**

Probes are periodic diagnostics that Kubelet will execute against a container. This helps kubernetes understand the current state of a container.

Readiness Probes tell kubernetes whether the application is ready to receive user traffic. You may have an application that takes awhile to start up. You wouldn't want user requests going to that application until it can successfully respond.

Liveness probes help kubernetes understand if the container is healthy. This way, if your application become unresponsive, kubelet can restart it or schedule a new Pod. More info on container probes in the [docs](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/" \l "container-probes).

#### **Step 1: Break the app**

Choose a pod and grab it's ip (POD\_IP). The test the app, we should get back the name of the pod it is running in.

→ List the pods

|  |
| --- |
| kubectl get pods -o wide |

→ Call the app

|  |
| --- |
| curl <POD\_IP> |

→ Then we can break it with a special url on the app:

|  |
| --- |
| curl <POD\_IP>/break |

→ Now try to hit the app again:

|  |
| --- |
| curl <POD\_IP> |

Here we can see that the app remains broken. Kubernetes can't help us because it doesn't know it's broken, it still sees the process running fine.

#### **Step 2: Add Probes**

-> Open deployment.yaml:

|  |
| --- |
| vi deployment.yaml |

-> Type ":set paste", press **enter**. Then press **i**, add the following in bold:

|  |
| --- |
| apiVersion: apps/v1  kind: Deployment  metadata:  name: http-deployment  labels:  app: http  spec:  replicas: 3  selector:  matchLabels:  app: http  template:  metadata:  labels:  app: http  spec:  containers:  - name: http-server  image: acmeade/http-app  ports:  - containerPort: 80  **livenessProbe:**  **httpGet:**  **path: /**  **port: 80**  **initialDelaySeconds: 3**  **periodSeconds: 5**  **readinessProbe:**  **httpGet:**  **path: /**  **port: 80**  **initialDelaySeconds: 5**  **periodSeconds: 3** |

→ Hit **ESC**, then type ':wq' and press **enter** to save the file.

**→ Apply your changes**

#### **Step 3: Break app again**

-> Copy a new Pod IP.

|  |
| --- |
| kubectl get pods -o wide |

-> Break the app:

|  |
| --- |
| curl <POD\_IP>/break |

-> See that after a few minutes, restarts for the pod is now one.

|  |
| --- |
| kubectl get pods -o wide |

It may take a minute for the probes to notice the pod is unhealthy

### **Task 3: Make a service object**

#### **Step 1: Apply a service definition**

Let's construct a loadbalancer for our pods.

→ Explore the service resource

|  |
| --- |
| kubectl explain service |

→ write your service in a file called **service.yaml**

→ Type ":set paste", press **enter**. Then press **i**, add the following in bold:

|  |
| --- |
| kind: Service  apiVersion: v1  metadata:  name: http-service  spec:  type: LoadBalancer  selector:  app: http  ports:  - protocol: TCP  port: 80  targetPort: 80 |

→ Hit **ESC**, then type ':wq' and press **enter** to save the file.

→ Apply our service

|  |
| --- |
| kubectl apply -f service.yaml |

|  |
| --- |
| service/my-service created |

#### **Step 2: View round robin load balancing**

Copy the service External IP. It may take awhile for it to leave pending status. Run the following command every few seconds until the EXTERNAL-IP field is populated.

→ Get service details

|  |
| --- |
| kubectl get services |

→ Copy the external IP

→ Test the service in your browser

**Go to http://<SERVICE\_IP>** in your browser. **Refresh a few times** to see that it actually cycles through which pod it sends you to. You may also notice that your app is now exposed to the internet via an external ip! This doesn't have to be the behavior if we don't want our app exposed, but it sure make it a lot easier to test in our lab :)

#### **Step 3: Add session affinity**

Being sent to random pods is good for stateless apps as it will not create a hotspot and overload a single pod. However, let's pretend that we have a stateful app and we want our users to have a consistent experience. We can set up our service to send connections from the same source ip to the same pod each time.

→ Look at session affinity options for service objects

|  |
| --- |
| kubectl explain service.spec.sessionAffinity |

→ Add session affinity in your **service.yaml.** Add the following in bold

|  |
| --- |
| ...  type: LoadBalancer  **sessionAffinity: ClientIP**  selector:  ... |

→ Apply your changes

|  |
| --- |
| kubectl apply -f service.yaml |

#### **Step 4: View always hitting the same Pod**

→ Now if you **go to http://<SERVICE\_IP>** in your browser and **refresh a few times**, you will always hit the same pod.

#### **Step 5: Delete the pod**

Lets delete the pod that we are being directed to.

→ Copy the pod name out of the browser:

→ Delete the pod

|  |
| --- |
| kubectl delete pod <POD\_NAME> |

→ Then **refresh your browser**, notice we are being sent to another pod now. The service is just doing best effort for session affinity.

### **Task 4: Cleanup**

→ Delete resources and files

|  |
| --- |
| kubectl delete all --all  rm service.yaml deployment.yaml |