Exercise - Deploy the ratings API

100 XP

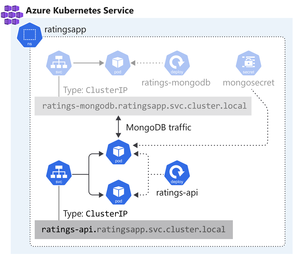
20 minutes

The Fruit Smoothies' ratings website consists of several components. There's a web frontend, a document database that stores captured data, and a RESTful ratings API that allows the web frontend to communicate with the database. The development team is using MongoDB as the document store database of choice for the ratings website.

In the previous unit, you deployed MongoDB using Helm. You'll continue your deployment and deploy the ratings API. The ratings API is a Node.js application written by using the Express framework. It stores and retrieves items and their ratings in a MongoDB database. Recall that you already created an Azure Container Registry instance.

In this exercise, you will:

- ✓ Create a Kubernetes deployment for the RESTful API
- ✓ Create a Kubernetes service to expose the RESTful API over the network



Create a Kubernetes deployment for the ratings API

A Kubernetes deployment gives you a way to provide declarative updates for Pods. You describe the desired state of the workload in a deployment manifest file, and use `kubectl` to submit the manifest to the Deployment Controller. The Deployment Controller in turn actions the desired state of the defined workload, for example, deploy a new Pod, increase the Pod count, or decrease the Pod count.

1. Create a manifest file for the Kubernetes deployment called `ratings-api-deployment.yaml` by using the integrated editor.

   bash
   ```
   code ratings-api-deployment.yaml
   ```

   > Tip
   >
   > Azure Cloud Shell includes an integrated file editor. The Cloud Shell editor supports features such as language highlighting, the command palette, and a file explorer. For simple file creation and editing, launch the editor by running `code .` in the Cloud Shell terminal. This action opens the editor with your active working directory set in the terminal. To directly open a file for quick editing, run `code <filename>` to open the editor without the file explorer. To open the editor via UI button, select the `{}` editor icon on the toolbar. This action opens the editor and defaults the file explorer to the `/home/<user>` directory.

2. Paste the following text in the file.

   YAML
   ```
   apiVersion: apps/v1
   kind: Deployment
   metadata:
     name: ratings-api
   spec:
     selector:
       matchLabels:
         app: ratings-api
     template:
       metadata:
         labels:
           app: ratings-api # the label for the pods and the deployments
       spec:
         containers:
         - name: ratings-api
           image: <acrname>.azurecr.io/ratings-api:v1 # IMPORTANT: update with your own repository
           imagePullPolicy: Always
           ports:
           - containerPort: 3000 # the application listens to this port
           env:
           - name: MONGODB_URI # the application expects to find the MongoDB connection details in
             valueFrom:
               secretKeyRef:
                 name: mongosecret # the name of the Kubernetes secret containing the data
                 key: MONGOCONNECTION # the key inside the Kubernetes secret containing the data
           resources:
             requests: # minimum resources required
               cpu: 250m
               memory: 64Mi
             limits: # maximum resources allocated
               cpu: 500m
               memory: 256Mi
           readinessProbe: # is the container ready to receive traffic?
             httpGet:
               port: 3000
               path: /healthz
           livenessProbe: # is the container healthy?
             httpGet:
               port: 3000
               path: /healthz
   ```

3. In this file, update the `<acrname>` value in the `image` key with the name of your Azure Container Registry instance.
4. Review the file, and note the following points:

   - `image`: You'll create a deployment with two replicas running the image you pushed to the Azure Container Registry instance you created previously, for example, `acr4229.azurecr.io/ratings-api:v1`. The container listens to port `3000`. The deployment and the pods are labeled with `app=ratings-api`.
   - `secretKeyRef`: The ratings API expects to find the connection details to the MongoDB database in an environment variable named `MONGODB_URI`. By using `valueFrom` and `secretKeyRef`, you can reference values stored in `mongosecret`, the Kubernetes secret that was created when you deployed MongoDB.
   - `resources`: Each container instance is given a minimum of 0.25 cores and 64 Mb of memory. The Kubernetes Scheduler looks for a node with available capacity to schedule such a pod. A container might or might not be allowed to exceed its CPU limit for extended periods. But it won't be killed for excessive CPU usage. If a container exceeds its memory limit, it could be terminated.
   - `readinessProbe` and `livenessProbe`: The application exposes a health check endpoint at `/healthz`. If the API is unable to connect to MongoDB, the health check endpoint returns a failure. You can use these probes to configure Kubernetes and check whether the container is healthy and ready to receive traffic.

5. To save the file, select `Ctrl+S`. To close the editor, select `Ctrl+Q`. You can also open the `...` action panel in the upper-right of the editor. Select **Save**, and then select **Close editor**.
6. Apply the configuration by using the `kubectl apply` command. Recall that you've deployed the MongoDB release in the `ratingsapp` namespace, so you will deploy the API in the `ratingsapp` namespace as well.

   bash
   ```
   kubectl apply \
     --namespace ratingsapp \
     -f ratings-api-deployment.yaml
   ```

   You'll see an output like this example.

   output
   ```
   deployment.apps/ratings-api created
   ```

7. You can watch the pods rolling out using the `-w` flag with the `kubectl get pods` command. Make sure to query for pods in the `ratingsapp` namespace that are labeled with `app=ratings-api`. Select `Ctrl+C` to stop watching.

   bash
   ```
   kubectl get pods \
     --namespace ratingsapp \
     -l app=ratings-api -w
   ```

   In a few seconds, you'll see the pods transition to the `Running` state. Select `Ctrl+C` to stop watching.

   output
   ```
   NAME                          READY   STATUS    RESTARTS   AGE
   ratings-api-564446d9c4-6rvvs  1/1     Running   0          42s
   ```

   If the pods aren't starting, aren't ready, or are crashing, you can view their logs by using `kubectl logs <pod name> --namespace ratingsapp` and `kubectl describe pod <pod name> --namespace ratingsapp`.

8. Check the status of the deployment.

   bash
   ```
   kubectl get deployment ratings-api --namespace ratingsapp
   ```

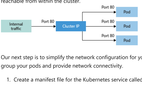
   The deployment should show that one replica is ready.

   output
   ```
   NAME          READY   UP-TO-DATE   AVAILABLE   AGE
   ratings-api   1/1     1            1           2m
   ```

Create a Kubernetes service for the ratings API service

A service is a Kubernetes object that provides stable networking for Pods by exposing them as a network service. You use Kubernetes Services to enable communication between nodes, pods, and users of your application, both internal and external, to your cluster. A Service, just like a node or Pod, gets an IP address assigned by Kubernetes when you create them. Services are also assigned a DNS name based on the service name, and a TCP port.

A *ClusterIP* allows you to expose a Kubernetes service on an internal IP in the cluster. This type makes the service only reachable from within the cluster.



Our next step is to simplify the network configuration for your application workloads. You'll use a Kubernetes service to group your pods and provide network connectivity.

1. Create a manifest file for the Kubernetes service called `ratings-api-service.yaml` by using the integrated editor.

   bash
   ```
   code ratings-api-service.yaml
   ```

2. Paste the following text in the file.

   YAML
   ```
   apiVersion: v1
   kind: Service
   metadata:
     name: ratings-api
   spec:
     selector:
       app: ratings-api
     ports:
     - protocol: TCP
       port: 80
       targetPort: 3000
     type: ClusterIP
   ```

3. Review the file, and note the following points:

   - `selector`: The selector determines the set of pods targeted by a service. In the following example, Kubernetes load balances traffic to pods that have the label `app: ratings-api`. This label was defined when you created the deployment. The controller for the service continuously scans for pods that match that label to add them to the load balancer.
   - `ports`: A service can map an incoming `port` to `targetPort`. The incoming port is what the service responds to. The target port is what the pods are configured to listen to. For example, the service is exposed internally within the cluster at `ratings-api.ratingsapp.svc.cluster.local:80` and load balances the traffic to the ratings-api pods listening on port `3000`.
   - `type`: A service of type `ClusterIP` creates an internal IP address for use within the cluster. Choosing this value makes the service reachable only from within the cluster. Cluster IP is the default service type.

4. To save the file, select `Ctrl+S`. To close the editor, select `Ctrl+Q`.
5. Apply the configuration by using the `kubectl apply` command, and use the `ratingsapp` namespace.

   bash
   ```
   kubectl apply \
     --namespace ratingsapp \
     -f ratings-api-service.yaml
   ```

   You'll see an output like this example.

   output
   ```
   service/ratings-api created
   ```

6. Check the status of the service.

   bash
   ```
   kubectl get service ratings-api --namespace ratingsapp
   ```

   The service should show an internal IP where it would be accessible. By default, Kubernetes creates a DNS entry that maps to `[service name].[namespace].svc.cluster.local`, which means this service is also accessible at `ratings-api.ratingsapp.svc.cluster.local`. Notice how `CLUSTER-IP` comes from the Kubernetes service address range you defined when you created the cluster.

   output
   ```
   NAME          TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
   ratings-api   ClusterIP   10.2.0.182   <none>        80/TCP    60s
   ```

7. Finally, let's validate the endpoints. Services load balance traffic to the pods through endpoints. The endpoint has the same name as the service. Validate that the service points to one endpoint that corresponds to the pod. As you add more replicas, or as pods come and go, Kubernetes automatically keeps the endpoints updated. Run the `kubectl get endpoints` command to fetch the endpoint information.

   bash
   ```
   kubectl get endpoints ratings-api --namespace ratingsapp
   ```

   You'll see a similar output like the example below. Notice how the `ENDPOINTS` IPs come from the `10.240.0.0/16` subnet you defined when you created the cluster.

   output
   ```
   NAME          ENDPOINTS             AGE
   ratings-api   10.240.0.11:3000      1h
   ```

You've now created a deployment of the **ratings-api** and exposed it as an internal (ClusterIP) service.

- **Deployment/ratings-api**: The API, running a replica, which reads the MongoDB connection details by mounting **mongosecret** as an environment variable.
- **Service/ratings-api**: The API is exposed internally within the cluster at **ratings-api.ratingsapp.svc.cluster.local:80**.

Summary

In this exercise, you created a Kubernetes deployment for the *ratings-api* by creating a deployment manifest file and applying it to the cluster. You've also created a Kubernetes service for the *ratings-api* by creating a manifest file and applying it to the cluster. You now have a *ratings-api* endpoint that is available through a cluster IP over the network.

Next, you'll use a similar process to deploy the Fruit Smoothies ratings website.

Next unit: Exercise - Deploy the ratings front end

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue.