Microsoft

Book three

Azure Kubernetes Service Workshop

Creation date
09/29/2020

Revision date
09/29/2020

Corporate Headquarters
Microsoft Corporation
1 Microsoft Way,
Redmond, WA 98052
www.microsoft.com

Submission/Revision History

Revision	Authors	Released date	Comments
1.0	Mauricio Zaragoza	09/29/2020	Initial release

Technical Review History

Examine	Examiner (s)	Date	Comments
1.0 Within group	Reggie Snead Bhavin Doshi	09/29/2020	Comments
2.0 Outside group		09/29/2020	Comments

Table of contents

Azure Kubernetes Service Workshop

In this workshop, you will go through tasks to deploy a multi-container application to Kubernetes on Azure Kubernetes Service (AKS).

Learning objectives.

In this workshop, you will:

- Create an Azure Kubernetes Service cluster
- Choose the best deployment options for your Pods
- Expose Pods to internal and external network users
- Configure SSL/TLS for Azure Kubernetes Service ingress
- Monitor the health of an Azure Kubernetes Service cluster
- Scale your application in an Azure Kubernetes Service cluster

Prerequisites

- Knowledge of Kubernetes and its concepts
- Access to an Azure subscription

Scope

This document will address the challenges and solutions for the website monitoring of the Aura web applications health. Other PwC organizations, teams, projects, and structures are out of the scope of this document.

Definitions

Table 1: Terms and Acronyms used in this document

Term	Acronym	Definition	Comments

Conventions

Table 2: Key Terms and Symbols used in this document

Term	Definition

Note	Notes give you additional information that will help you obey the instructions written in the procedural steps.
	Red squares or rectangles shows you where to enter and / or to examine the necessary information (data, IP addresses, etc.).
Bold	Text written in bold in procedural steps indicate data that must be typed and / or menu selections that must selected or clicked on with the mouse pointer.
	Red arrows indicate where to point and click your mouse pointer. Reference to these in screenshots and other parts of the document.

Document updates

This document contains information about Microsoft Azure and other Microsoft technologies.
Due to the dynamics of the modern world, these technologies are evolving and changing all the time.

Microsoft is adding and deprecating features to Azure every day, increasing quotas, changing limits, releasing new products, changing, and improving architectures, etcetera.

In general, Microsoft is modifying the characteristics of the Azure services to accommodate customers' requirements and needs quite often. For this reason, the details about the features and products contained in this document might be enhanced or changed at some point in time.

Most of the concepts that are outlined in this guide are general concepts that are not expected to change drastically soon.

You need to be aware of the dynamics of Azure and use this document as reference. Please complement the information in this guide with the most current Azure documentation that is available online.

Values, limits, names, capabilities, features, costs, billing models, regions, etc. are valid at the time this document was published.

The document release date is in the **Revision** section, and a comprehensive list of online resources are published in the **References** section. These 2 portions need to be modified accordingly when Microsoft updates any of the functionalities described in this white paper.


Azure

Introduction

Scenario description.

Imagine you are an IT engineer at Fruit Smoothies, a nationwide chain of smoothie shops. The company's development team developed a new website that allows users to rate the company's different smoothy flavors. Fruit Smoothies has outlets worldwide with a large follower base and the company expects many fans to visit the new website. You want to make sure when you deploy the ratings website, you can scale the site quickly when needed.

The ratings website consists of several components. There is a web frontend, a document database that stores captured data, and a RESTful API. The API allows the web frontend to communicate with the database. Since the ratings website may store sensitive data, an additional requirement is to protect the site with an HTTPS certificate.

Fruit Smoothies wants to use Kubernetes as their compute platform. The development teams already use containers for application development and deployment and using an orchestration platform will introduce many benefits. Kubernetes is a portable, extensible, open-source platform for automating the deployment, scaling, and management of containerized workloads. Kubernetes abstracts away complex container management and provides declarative configuration to orchestrate containers in different compute environments. This orchestration platform gives the same ease of use and flexibility as with Platform as a Service (PaaS) and Infrastructure as a Service (IaaS) offerings.

In this module, you will go through tasks to deploy a multi-container application to Kubernetes on Azure Kubernetes Service (AKS).

You will use your Azure subscription to deploy the resources in this workshop. To estimate the expected costs for these resources, see the preconfigured Azure Calculator estimate of the resources that you'll deploy.

Learning Objectives

In this workshop, you will:

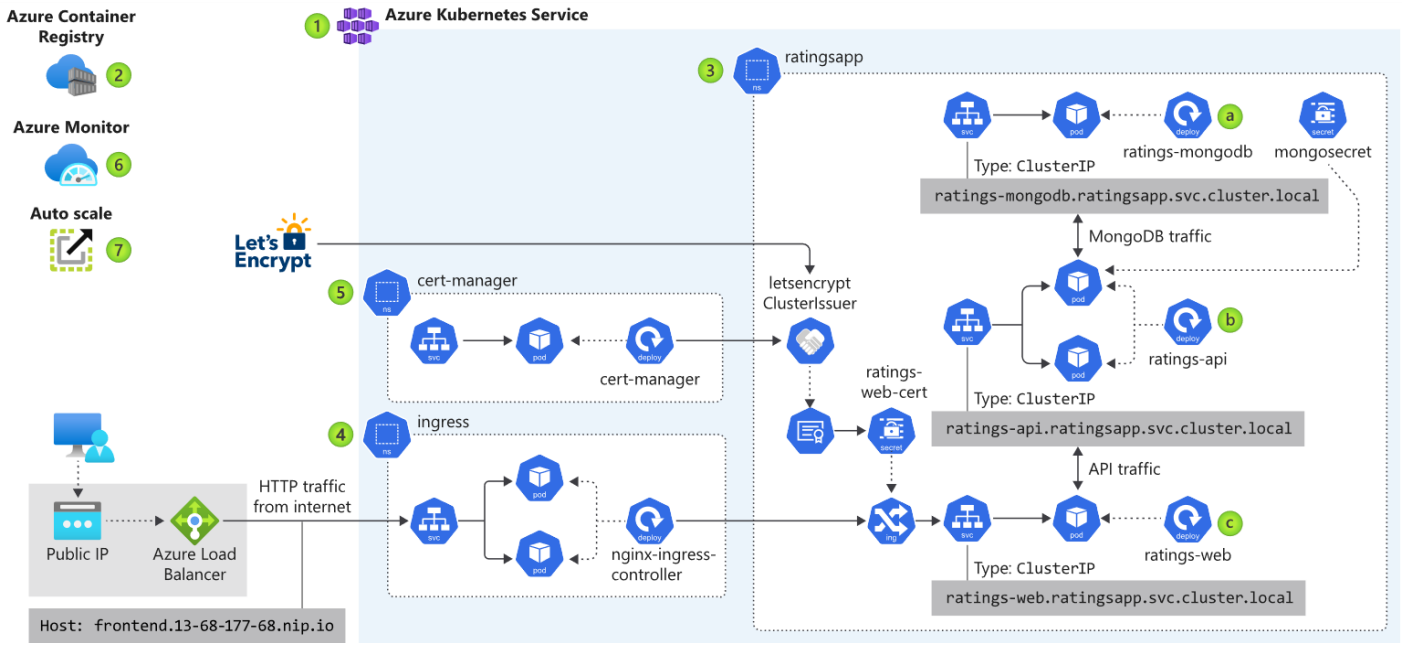
- Create an Azure Kubernetes Service cluster
- Choose the best deployment options for you Pods
- Expose Pods to internal and external network users
- Configure SSL/TLS for Azure Kubernetes Service ingress
- Monitor the health of an Azure Kubernetes Service cluster
- Scale your application in an Azure Kubernetes Service cluster

Prerequisites

- Familiarity with Kubernetes and its concepts. If you're new to Kubernetes, start with the basics of Kubernetes .
- An Azure subscription to deploy resources in.
- Familiarity with Azure Cloud Shell .
- A GitHub account.

Application architecture

Our goal is to deploy an Azure managed Kubernetes service, Azure Kubernetes Service (AKS), that runs the Fruit Smoothies ratings website in the following series of exercises.



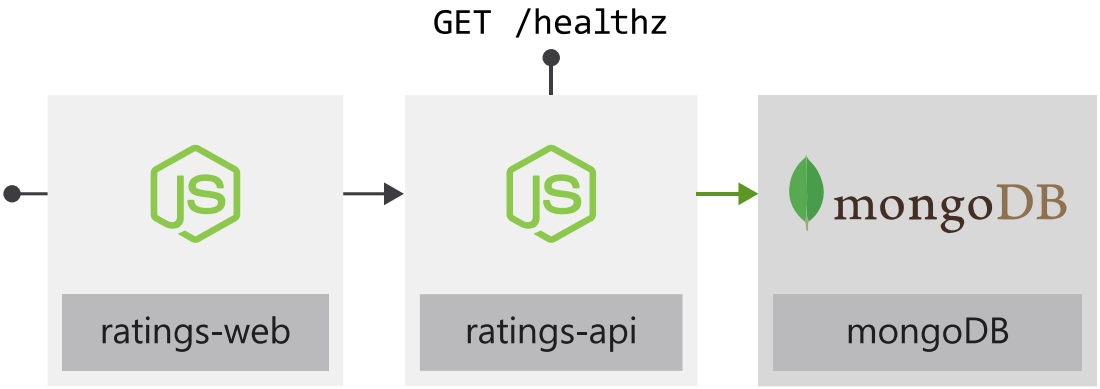
There are several tasks that you will complete to show how Kubernetes abstracts away complex container management and provides you with declarative configuration to orchestrate containers.

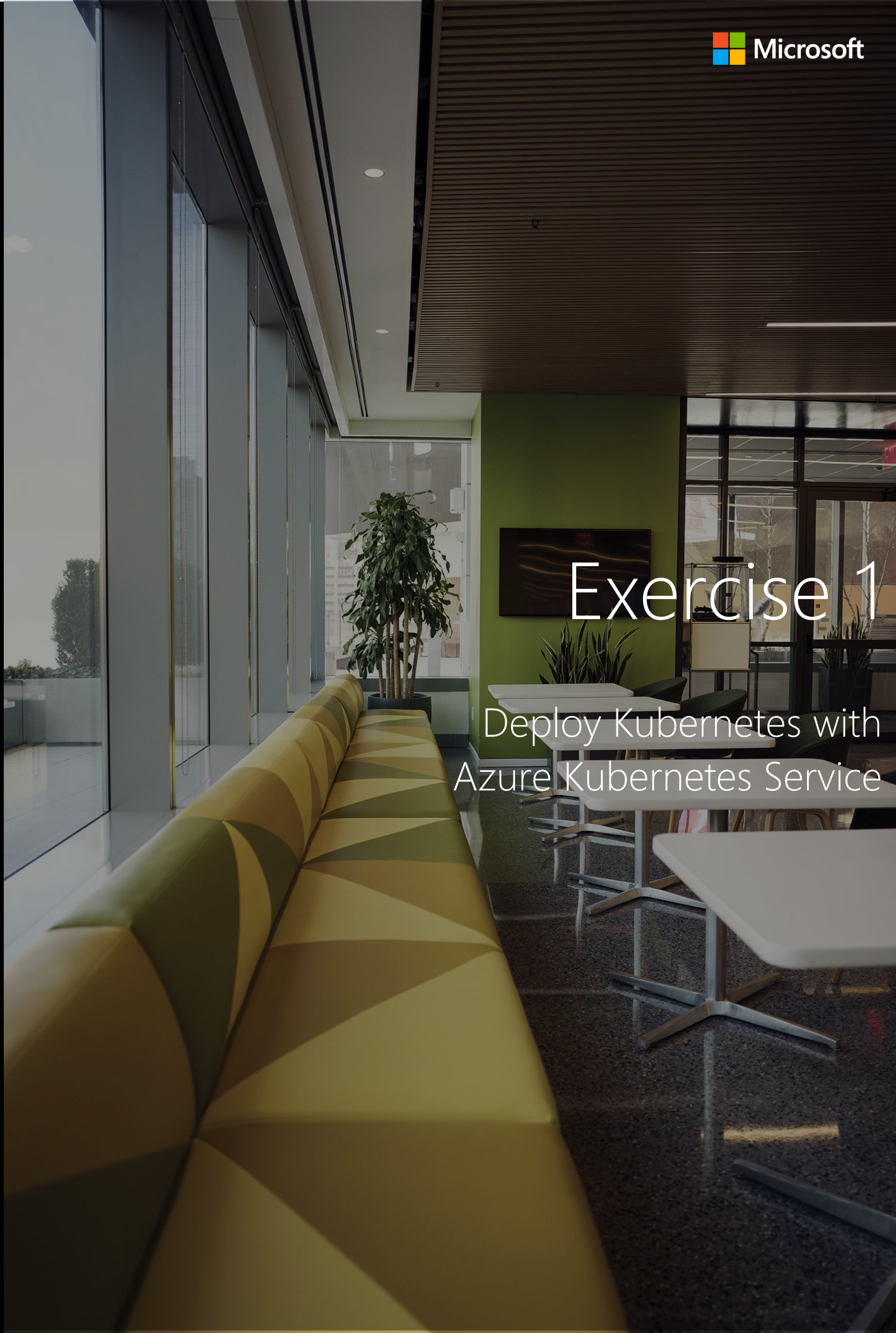
1. Use AKS to deploy a Kubernetes cluster.
2. Configure an Azure Container Registry to store application container images.
3. Deploy the three ratings application components.
4. Deploy the Fruit Smoothies website document database using Helm 3.
5. Deploy the Fruit smoothie's RESTful API using deployment manifests.
6. Deploy the Fruit smoothie's website frontend using deployment manifests.
7. Deploy Azure Kubernetes ingress using Helm 3.
8. Configure SSL/TLS on the controller using `cert-manager`.
9. Configure Azure Monitor for containers to monitor the Fruit Smoothies website deployment.
10. Configure cluster auto-scaler and horizontal pod auto-scaler for the Fruit Smoothies cluster.

Source code

The application consists of two components: the API and the front end. Both components are written in Node.js. The API stores data in a MongoDB database.

Component	Link
An API `ratings-api`	GitHub repo
A front-end `ratings-web`	GitHub repo




Microsoft
Exercise 1
Deploy Kubernetes with
Azure Kubernetes Service

Exercise - Deploy Kubernetes with Azure Kubernetes Service

Fruit Smoothies wants to use Kubernetes as their compute platform. The development teams already use containers for application development and deployment and using an orchestration platform will help them rapidly build, deliver, and scale their application.

To do this, you need to deploy the foundation of your Kubernetes environment.

In this exercise, you will:

- ✓ Create a new resource group.
- ✓ Configure cluster networking.
- ✓ Create an Azure Kubernetes Service cluster.
- ✓ Connect to the Kubernetes cluster by using `kubectl`.
- ✓ Create a Kubernetes namespace.

> ⓘ Important
>
> You need your own Azure subscription to run this exercise and you may incur charges. If you don't already have an Azure subscription, create a free account before you begin..

Create a new resource group

1. Sign-in to Azure Cloud Shell with your Azure account. Select the Bash version of Cloud Shell.

   Azure Cloud Shell

2. We are going to reuse some values throughout the deployment scripts. For example, you need to choose a region where you want to create a resource group, such as **East US**. If you select a different value, remember it for the rest of the exercises in this module. You may need to redefine the value between Cloud Shell sessions. Run the following commands to record these values in Bash variables.

   Azure CLI

   ```
   REGION_NAME=eastus
   RESOURCE_GROUP=aksworkshop
   SUBNET_NAME=aks-subnet
   VNET_NAME=aks-vnet
   ```

> ⓘ Note
>
> You can use the **Copy** button to copy commands to the clipboard. To paste, right-click on a new line in the Cloud Shell window and select **Paste** or use the `Shift+Insert` keyboard shortcut (⌘+V on macOS)..

1. You can check each value using the `echo` command, for example, `echo $REGION_NAME`.
2. Create a new resource group with the name **aksworkshop**. Deploy all resources created in these exercises in this resource group. A single resource group makes it easier to clean up the resources after you finish the module.

Azure CLI

```
az group create \
    --name $RESOURCE_GROUP \
    --location $REGION_NAME
```

Configure networking

We have two network models to choose from when deploying an AKS cluster. The first model is *Kubenet networking*, and the second is *Azure Container Networking Interface (CNI) networking*.

What is Kubenet networking?

Kubenet networking is the default networking model in Kubernetes. With Kubenet networking, nodes get assigned an IP address from the Azure virtual network subnet. Pods receive an IP address from a logically different address space to the Azure virtual network subnet of the nodes.

Network address translation (NAT) is then configured so that the pods can reach resources on the Azure virtual network. The source IP address of the traffic is translated to the node's primary IP address and then configured on the nodes. Note, that pods receive an IP address that is "hidden" behind the node IP.

What is Azure Container Networking Interface (CNI) networking?

With Azure Container Networking Interface (CNI), the AKS cluster is connected to existing virtual network resources and configurations. In this networking model, every pod gets an IP address from the subnet and can be accessed directly. These IP addresses must be unique across your network space and calculated in advance.

Some of the features you will use require you to deploy the AKS cluster by using the *Azure Container Networking Interface networking* configuration.

For a more detailed comparison, see the **Learn more** section at the end of this module.

Let's create the virtual network for your AKS cluster. We will use this virtual network and specify the networking model when we deploy the cluster.

1. First, create a virtual network and subnet. Pods deployed in your cluster will be assigned an IP from this subnet. Run the following command to create the virtual network.

Azure CLI

```
az network vnet create \
    --resource-group $RESOURCE_GROUP \
    --location $REGION_NAME \
    --name $VNET_NAME \
    --address-prefixes 10.0.0.0/8 \
    --subnet-name $SUBNET_NAME \
    --subnet-prefixes 10.240.0.0/16
```

2. Next, retrieve, and store the subnet ID in a Bash variable by running the command below.

```
Azure CLI

SUBNET_ID=$(az network vnet subnet show \
    --resource-group $RESOURCE_GROUP \
    --vnet-name $VNET_NAME \
    --name $SUBNET_NAME \
    --query id -o tsv)
```

Create the AKS cluster

With the new virtual network in place, you can go ahead and create your new cluster. There are two values you need to know before running the `az aks create` command. The first is the version of the latest, non-preview, Kubernetes version available in your selected region, and the second is a unique name for your cluster.

1. To get the latest, non-preview, Kubernetes version you use the `az aks get-versions` command. Store the value that returns from the command in a Bash variable named `VERSION`. Run the command below the retrieve and store the version number.

```
Azure CLI

VERSION=$(az aks get-versions \
    --location $REGION_NAME \
    --query 'orchestrators[?!isPreview] | [-1].orchestratorVersion' \
    --output tsv)
```

2. The AKS cluster name must be unique. Run the following command to create a Bash variable that holds a unique name.

```
Bash

AKS_CLUSTER_NAME=aksworkshop-$RANDOM
```

3. Run the following command to output the value stored in `$AKS_CLUSTER_NAME`. Note this for later use. You'll need it to reconfigure the variable in the future, if necessary.

```
Bash

echo $AKS_CLUSTER_NAME
```

4. Run the following `az aks create` command to create the AKS cluster running the latest Kubernetes version. This command can take a few minutes to complete.

Azure CLI

```
az aks create \
--resource-group $RESOURCE_GROUP \
--name $AKS_CLUSTER_NAME \
--vm-set-type VirtualMachineScaleSets \
--node-count 2 \
--load-balancer-sku standard \
--location $REGION_NAME \
--kubernetes-version $VERSION \
--network-plugin azure \
--vnet-subnet-id $SUBNET_ID \
--service-cidr 10.2.0.0/24 \
--dns-service-ip 10.2.0.10 \
--docker-bridge-address 172.17.0.1/16 \
--generate-ssh-keys
```

Let's review the variables in the previous command:

- `$AKS_CLUSTER_NAME` specifies the name of the AKS cluster.
- `$VERSION` is the latest Kubernetes version you retrieved earlier.
- `$SUBNET_ID` is the ID of the subnet created on the virtual network to be configured with AKS.

Note the following deployment configuration:

- `--vm-set-type`: We're specifying that the cluster is created by using virtual machine scale sets. The virtual machine scale sets enable you to switch on the cluster auto-scaler when needed.

- `--node-count`: We're specifying that the cluster is created with two nodes. The default node count is three nodes. However, if you are running this exercise using a free trial account, cluster creation may fail due to quota limits if left at the default setting.

- `--network-plugin`: We're specifying the creation of the AKS cluster by using the CNI plug-in.

- `--service-cidr`: This address range is the set of virtual IPs that Kubernetes assigns to internal services in your cluster. The range must not be within the virtual network IP address range of your cluster. It should be different from the subnet created for the pods.

- `--dns-service-ip`: The IP address is for the cluster's DNS service. This address must be within the *Kubernetes service address range*. Don't use the first IP address in the address range, such as 0.1. The first address in the subnet range is used for the *kubernetes.default.svc.cluster.local* address.

- `--docker-bridge-address`: The Docker bridge network address represents the default *docker0* bridge network address present in all Docker installations. AKS clusters or the pods themselves do not use *docker0* bridge. However, you have to set this address to continue supporting scenarios such as *docker build* within the AKS cluster. It is required to select a classless inter-domain routing (CIDR) for the Docker bridge network address. If you do not set a CIDR, Docker chooses a subnet automatically. This subnet could conflict with other CIDRs. Choose an address space that does not collide with the rest of the CIDRs on your networks, which includes the cluster's service CIDR and pod CIDR.

Test cluster connectivity by using kubectl

kubectl is the main Kubernetes command-line client you use to interact with your cluster and is available in Cloud Shell. A cluster context is required to allow *kubectl* to connect to a cluster. The context contains the cluster's address, a user, and a namespace. Use the `az aks get-credentials` command to configure your instance of *kubectl*.

1. Retrieve the cluster credentials by running the command below.

   Azure CLI

   ```
   az aks get-credentials \
       --resource-group $RESOURCE_GROUP \
       --name $AKS_CLUSTER_NAME
   ```

2. Let's look at what was deployed by listing all the nodes in your cluster. Use the `kubectl get nodes` command to list all the nodes.

   Bash

   ```
   kubectl get nodes
   ```

3. You'll see a list of your cluster's nodes. Here's an example.

   Output

   ```
   NAME                                STATUS   ROLES   AGE     VERSION
   aks-nodepool1-29333311-vmss000000   Ready    agent   2m36s   v1.17.9
   aks-nodepool1-29333311-vmss000001   Ready    agent   2m34s   v1.17.9
   ```

Create a Kubernetes namespace for the application

Fruit Smoothies want to deploy several apps from other teams in the deployed AKS cluster as well. Instead of running multiple clusters, the company wants to use the Kubernetes features that let you logically isolate teams and workloads in the same cluster. The goal is to provide the least number of privileges scoped to the resources each team needs.

What is a namespace?

A namespace in Kubernetes creates a logical isolation boundary. Names of resources must be unique within a namespace but not across namespaces. If you don't specify the namespace when you work with Kubernetes resources, the *default* namespace is implied.

Let's create a namespace for your ratings application.

1. List the current namespaces in the cluster.

   Bash

   ```
   kubectl get namespace
   ```

   You'll see a list of namespaces similar to this output.

   Output

   ```
   NAME              STATUS   AGE
   default           Active   1h
   kube-node-lease   Active   1h
   kube-public       Active   1h
   kube-system       Active   1h
   ```

2. Use the `kubectl create namespace` command to create a namespace for the application called **ratingsapp**.

   Bash

   ```
   kubectl create namespace ratingsapp
   ```

   You'll see a confirmation that the namespace was created.

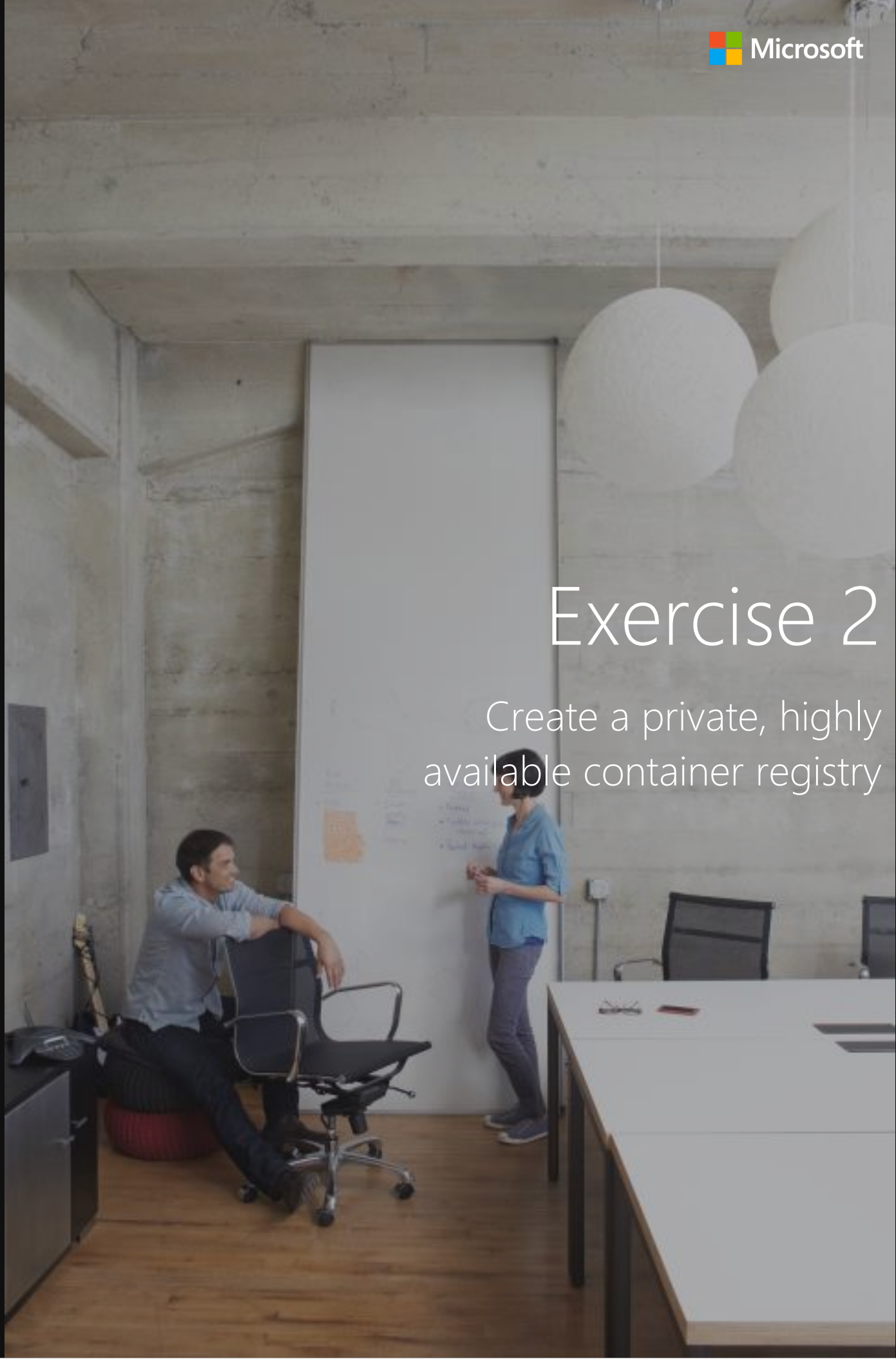
   Output

   ```
   namespace/ratingsapp created
   ```

Summary

In this exercise, you created a resource group for your resources. You created a virtual network for your cluster to use. You then deployed your AKS cluster, including the Azure CNI networking mode. You then connected to your cluster with *kubectl* and created a namespace for your Kubernetes resources.

Next, you'll create and configure an Azure Container Registry (ACR) instance to use with your AKS cluster and store your containerized ratings app.



Exercise 2

Create a private, highly available container registry

Exercise - Create a private, highly available container registry

The Fruit Smoothies software development and operations teams made the decision to containerize all newly developed applications. Containerized applications provide the teams with mutual benefits. For example,

- The ease of managing hosting environments
- The guarantee of continuity in software delivery
- The efficient use of server hardware
- The portability of applications between environments.

The teams made the decision to store all containers in a central and secure location and the decision made is to use Azure Container Registry.

In this exercise, you will:

- ✓ Create a container registry by using the Azure CLI
- ✓ Build container images by using Azure Container Registry Tasks
- ✓ Verify container images in Azure Container Registry
- ✓ Configure an AKS cluster to authenticate to an Azure Container Registry

Create a container registry

Azure Container Registry is a managed Docker registry service based on the open-source Docker Registry 2.0. Container Registry is private and hosted in Azure. You use it to build, store, and manage images for all types of container deployments.

Container images can be pushed and pulled with Container Registry by using the Docker CLI or the Azure CLI. You can use Azure portal integration to visually inspect the container images in the container registry. In distributed environments, the Container Registry geo-replication feature can be used to distribute container images to multiple Azure datacenters for localized distribution.

Azure Container Registry Tasks can also build container images in Azure. Tasks use a standard Dockerfile to create and store a container image in Azure Container Registry without the need for local Docker tooling. With Azure Container Registry Tasks, you can build on-demand or fully automate container image builds by using DevOps processes and tooling.

Let's deploy a container registry for the Fruit Smoothies environment.

1. The container registry name must be unique within Azure and contain between 5 and 50 *alphanumeric* characters. For learning purposes, run this command from Azure Cloud Shell to create a Bash variable that holds a unique name.

```bash
ACR_NAME=acr$RANDOM
```

2. You use the `az acr create` command to create the registry in the same resource group and region as your Azure Kubernetes Service (AKS) cluster. For example, **aksworkshop** in **East US**.

   Run the command below to create the ACR instance.

```azure-cli
az acr create \
    --resource-group $RESOURCE_GROUP \
    --location $REGION_NAME \
    --name $ACR_NAME \
    --sku Standard
```

   You'll see a response similar to this JSON example when the command completes.

```json
{
  "adminUserEnabled": false,
  "creationDate": "2019-12-28T01:33:23.906677+00:00",
  "id": "/subscriptions/00000000-0000-0000-0000-
000000000000/resourceGroups/aksworkshop/providers/Microsoft.ContainerRegistry/registries/acr4229",
  "location": "eastus",
  "loginServer": "acr4229.azurecr.io",
  "name": "acr4229",
  "networkRuleSet": null,
  "policies": {
    "quarantinePolicy": {
      "status": "disabled"
    },
    "retentionPolicy": {
      "days": 7,
      "lastUpdatedTime": "2019-12-28T01:33:25.070450+00:00",
      "status": "disabled"
    },
    "trustPolicy": {
      "status": "disabled",
      "type": "Notary"
    }
  },
  "provisioningState": "Succeeded",
  "resourceGroup": "aksworkshop",
  "sku": {
    "name": "Standard",
    "tier": "Standard"
  },
  "status": null,
  "storageAccount": null,
```

```
  "tags": {},
  "type": "Microsoft.ContainerRegistry/registries"
}
```

Build the container images by using Azure Container Registry Tasks

The Fruit Smoothies rating app makes use of two container images, one for the front-end website and one for the RESTful API web service. Your development teams use the local Docker tooling to build the container images for the website and API web service. A third container is used to deploy the document database provided by the database publisher and will not be stored the database container in ACR.

You can use Azure Container Registry to build these containers using a standard Dockerfile to provide build instructions. With Azure Container Registry, you can reuse any Dockerfile currently in your environment, which includes multi-staged builds.

Build the ratings-api image

The ratings API is a Node.js application that's built using Express, a Node.js web framework. The source code is on GitHub and already includes a Dockerfile, which builds images based on the Node.js Alpine container image.

Here, you'll clone the repository and then build the Docker image using the included Dockerfile. Use the built-in ACR functionality to build and push the container image into your registry by running the `az acr build` command.

1. Clone the repository to your Cloud Shell.

   Bash
   ```
   git clone https://github.com/MicrosoftDocs/mslearn-aks-workshop-ratings-api.git
   ```

2. Change into the newly cloned directory.

   Bash
   ```
   cd mslearn-aks-workshop-ratings-api
   ```

3. Run `az acr build`. This command builds a container image by using the Dockerfile. Then it pushes the resulting image to the container registry.

Azure CLI

```
az acr build \
    --resource-group $RESOURCE_GROUP \
    --registry $ACR_NAME \
    --image ratings-api:v1 .
```

> **Note**
>
> Don't forget the period `.` at the end of the preceding command. It represents the source directory that contains the **Dockerfile**. In this case, it's the current directory. Because you didn't specify the name of a file with the `--file` parameter, the command looks for a file called **Dockerfile** in the current directory.

After a few minutes, you'll see a response similar to this example.

Output

```
2019/12/28 02:04:11 Successfully pushed image: acr4229.azurecr.io/ratings-api:v1
2019/12/28 02:04:11 Step ID: build marked as successful (elapsed time in seconds: 240.205952)
2019/12/28 02:04:11 Populating digests for step ID: build...
2019/12/28 02:04:13 Successfully populated digests for step ID: build
2019/12/28 02:04:13 Step ID: push marked as successful (elapsed time in seconds: 33.293102)
2019/12/28 02:04:13 The following dependencies were found:
2019/12/28 02:04:13
- image:
    registry: acr4229.azurecr.io
    repository: ratings-api
    tag: v1
    digest: sha256:b35cc14b16e3a4f51b86d0ed61f74dcfabb00f63e015ed33ec1fe7f48c55abda
  runtime-dependency:
    registry: registry.hub.docker.com
    repository: library/node
    tag: 13.5-alpine
    digest: sha256:a5a7ff4267a810a019c7c3732b3c463a892a61937d84ee952c34af2fb486058d
  git: {}

Run ID: ca2 was successful after 4m41s
```

Make a note of the pushed image registry and name, for example, `acr4229.azurecr.io/ratings-api:v1`. You'll need this information when you configure the Kubernetes deployment.

Build the ratings-web image

The ratings front end is a Node.js application that was built by using the Vue JavaScript framework and WebPack to bundle the code. The source code is on GitHub and already includes a Dockerfile, which builds images based on the Node.js Alpine image.

The steps you follow are the same as before. Clone the repository and then build the docker image using the included Dockerfile using the `az acr build` command.

1. First, change back to the home directory.

   Bash

   ```
   cd ~
   ```

2. Clone the *ratings-web* repo.

   Bash

   ```
   git clone https://github.com/MicrosoftDocs/mslearn-aks-workshop-ratings-web.git
   ```

3. Change into the newly cloned directory.

   Bash

   ```
   cd mslearn-aks-workshop-ratings-web
   ```

4. Run `az acr build`. This command builds a container image by using the Dockerfile. Then it pushes the resulting image to the container registry.

   Azure CLI

   ```
   az acr build \
       --resource-group $RESOURCE_GROUP \
       --registry $ACR_NAME \
       --image ratings-web:v1 .
   ```

   In a few minutes, you'll see a response similar to this example.

   Output

   ```
   2019/12/28 02:09:51 Successfully pushed image: acr4229.azurecr.io/ratings-web:v1
   2019/12/28 02:09:51 Step ID: build marked as successful (elapsed time in seconds: 26.612936)
   2019/12/28 02:09:51 Populating digests for step ID: build...
   2019/12/28 02:09:53 Successfully populated digests for step ID: build
   2019/12/28 02:09:53 Step ID: push marked as successful (elapsed time in seconds: 35.571607)
   2019/12/28 02:09:53 The following dependencies were found:
   2019/12/28 02:09:53
   - image:
       registry: acr4229.azurecr.io
       repository: ratings-web
       tag: v1
   ```

```
    digest: sha256:ae4bab55e74d057e48b05b45761eef8d1c71874d9cfeeef6e0c3c1178f01f0f2
  runtime-dependency:
    registry: registry.hub.docker.com
    repository: library/node
    tag: 13.5-alpine
    digest: sha256:a5a7ff4267a810a019c7c3732b3c463a892a61937d84ee952c34af2fb486058d
  git: {}

Run ID: ca3 was successful after 1m9s
```

Make a note of the pushed image registry and name, for example, `acr4229.azurecr.io/ratings-web:v1`. Use this information when you configure the Kubernetes deployment.

Verify the images

1. Run the following command in Cloud Shell to verify that the images were created and stored in the registry.

Azure CLI

```
az acr repository list \
    --name $ACR_NAME \
    --output table
```

The output from this command looks similar to this example.

Output

```
Result
-----------
ratings-api
ratings-web
```

The images are now ready for use.

Configure the AKS cluster to authenticate to the container registry

We need to set up authentication between your container registry and Kubernetes cluster to allow communication between the services.

Let's integrate the container registry with the existing AKS cluster by supplying valid values for **AKS_CLUSTER_NAME** and **ACR_NAME**. You can automatically configure the required service principal authentication between the two resources by running the `az aks update` command.

Run the following command.

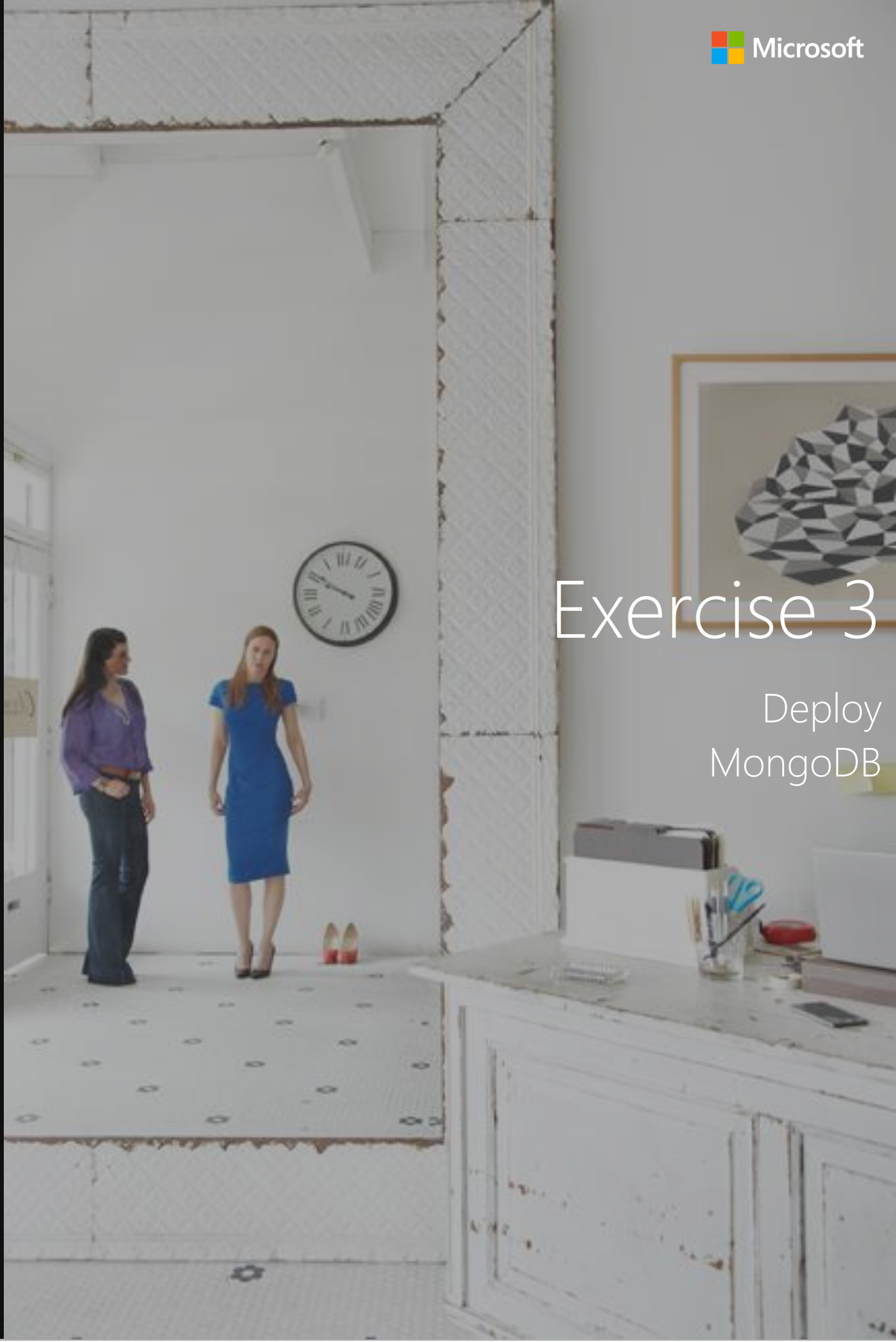
Azure CLI

```
az aks update \
    --name $AKS_CLUSTER_NAME \
    --resource-group $RESOURCE_GROUP \
    --attach-acr $ACR_NAME
```

Summary

In this exercise, you created a container registry for the Fruit Smoothies application. You then built and added container images for the *ratings-api* and *ratings-web* to the container registry. You then verified the container images and configured your AKS cluster to authenticate to the container registry.

Next, you'll take the first step to deploy your ratings app. The first component you'll deploy is MongoDB as your document store database, and you'll see how to use the HELM package manager for Kubernetes.



Microsoft

Exercise 3

Deploy MongoDB

Exercise - Deploy MongoDB

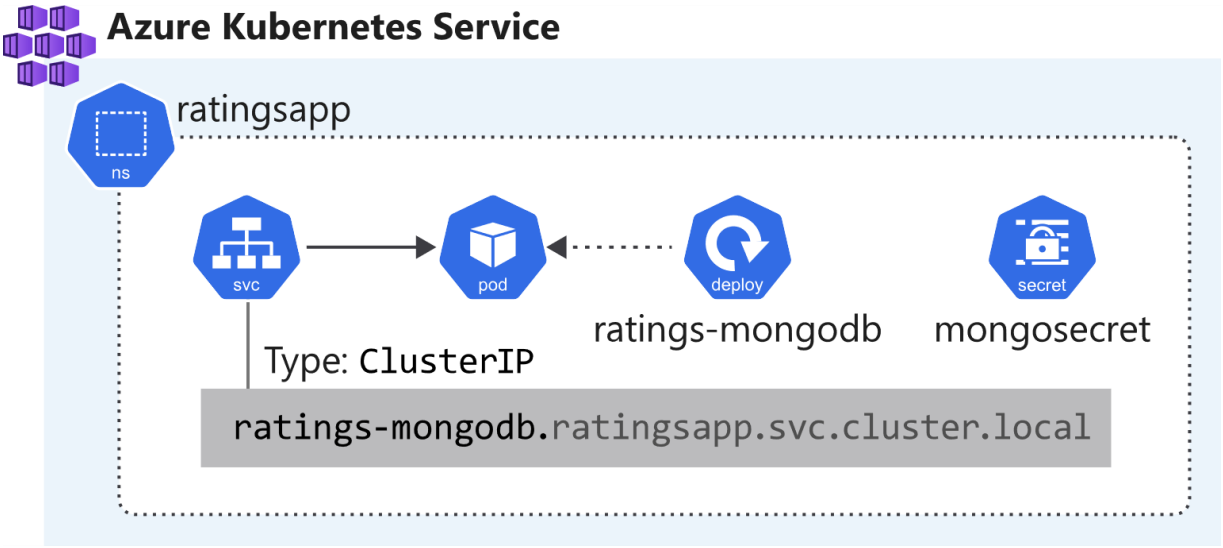
The Fruit Smoothies' ratings website consists of several components. There's a web frontend, a document database that stores captured data, and a RESTful API that allows the web frontend to communicate with the database. The development team is using MongoDB as the document store database of choice for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

In this exercise, you'll deploy MongoDB to the Azure Kubernetes Service (AKS) cluster using Helm. You'll also see how to use a Kubernetes secret to store the MongoDB connection username and password.

This example architecture deploys MongoDB on the cluster for the application to use to store data. While this is acceptable for test and development environments, it's not recommended for production environments. For production, it's recommended to store your application state and data in a scalable data storage platform, such as CosmosDB.

In this exercise, you will:

- ✓ Configure the Helm stable repository
- ✓ Install the MongoDB chart
- ✓ Create a Kubernetes secret to hold database credentials



Add the Helm bitnami repository

Helm is an application package manager for Kubernetes. It offers a way to easily deploy applications and services using charts.

The Helm client is already installed in the Azure Cloud Shell and can be run with the `helm` command. Helm provides a standard repository of charts for many different software packages. Helm has a chart for MongoDB that is part of the official Helm *bitnami* charts repository.

1. Configure the Helm client to use the stable repository by running the `helm repo add` command below.

   Bash

   ```
   helm repo add bitnami https://charts.bitnami.com/bitnami
   ```

2. You can now list the charts to install by running the `helm search repo` command. Notice how you can list all charts from the stable channel in the command below.

   Bash

   ```
   helm search repo bitnami
   ```

   You'll see a list of the available charts, like this example.

   Output

   ```
   NAME                          CHART VERSION   APP VERSION   DESCRIPTION
   bitnami/bitnami-common                0.0.8           0.0.8                 Chart with custom
   templates used in Bitnami cha...
   bitnami/airflow                       6.1.8           1.10.10               Apache Airflow is a
   platform to programmaticall...
   bitnami/apache                        7.3.15          2.4.43                Chart for Apache
   HTTP Server
   bitnami/cassandra                     5.3.3           3.11.6                Apache Cassandra is
   a free and open-source dist...
   ...
   ```

Install a Helm chart

A Helm chart is a collection of files that describe a related set of Kubernetes resources. You can use a single chart to deploy something simple, like a memcached pod, or something complex, like a full web app stack with HTTP servers, databases, and caches.

Helm charts are stored in Helm chart repositories. The official chart repository is maintained on GitHub. The Helm Hub provides a way to discover and view documentation of such charts.

You're now ready to install the MonogoDB instance. Recall from earlier, that you configured your cluster with a `ratingsapp` namespace. You'll specify the namespace as part of the `helm install` command, and a name for the database release. The release is called `ratings` and is deployed into the `ratingsapp` namespace.

1. Run the `helm install` command below. Make sure to replace `<username>` and `<password>` with appropriate values and note them for later use.

   Keep in mind that the MongoDB connection string is a URI. You have to escape special characters using a standard URI escape mechanism when choosing special characters in the username or password.

   Bash

   ```
   helm install ratings bitnami/mongodb \
       --namespace ratingsapp \
       --set auth.username=<username>,auth.password=<password>,auth.database=ratingsdb
   ```

   You provide parameters with the `--set` switch and a comma-separated list of `key=value` pairs. Pay attention to the `mongodbUsername`, `mongodbPassword`, and `mongodbDatabase` parameters and their values, which set the username, password, and database name, respectively. The application expects that the database is called **ratingsdb**. The `helm install` command is a powerful command with many capabilities.

2. After the installation is finished, you should get an output similar to this example. Make a note of the MongoDB host, which should be `ratings-mongodb.ratingsapp.svc.cluster.local`, if you used the same parameters.

   Output

   ```
   NAME: ratings
   LAST DEPLOYED: Thu Apr 30 14:15:58 2020
   NAMESPACE: ratingsapp
   STATUS: deployed
   REVISION: 1
   TEST SUITE: None
   NOTES:
   ** Please be patient while the chart is being deployed **

   MongoDB can be accessed via port 27017 on the following DNS name from within your cluster:
   ratings-mongodb.ratingsapp

   To get the root password run:
   ```

```
export MONGODB_ROOT_PASSWORD=$(kubectl get secret --namespace ratingsapp ratings-mongodb -o
jsonpath="{.data.mongodb-root-password}" |  base64 --decode)

To get the password for "chris" run:

export MONGODB_PASSWORD=$(kubectl get secret --namespace ratingsapp ratings-mongodb -o
jsonpath="{.data.mongodb-password}" | base64 --decode)

To connect to your database run the following command:

kubectl run --namespace ratingsapp ratings-mongodb-client --rm --tty -i --restart='Never' --image
docker.io/bitnami/mongodb:4.2.6-debian-10-r13 --command -- mongo admin --host ratings-mongodb --
authenticationDatabase admin -u root -p $MONGODB_ROOT_PASSWORD

To connect to your database from outside the cluster execute the following commands:

kubectl port-forward --namespace ratingsapp svc/ratings-mongodb 27017:27017 &
mongo --host 127.0.0.1 --authenticationDatabase admin -p $MONGODB_ROOT_PASSWORD
```

Keep in mind that you can easily remove a Helm release by running the `helm uninstall` command. The full command is `helm uninstall ratings --namespace ratingsapp`. In this exercise, uninstalling a chart should only be necessary if you made a mistake specifying a non-escaped username or password.

Create a Kubernetes secret to hold the MongoDB details

In the previous step, you installed MongoDB using Helm, with a specified username, password, and database name. Now you'll store these details in a Kubernetes secret. This step ensures that you don't leak secrets by hard coding them into configuration files.

Kubernetes has a concept of secrets. Secrets let you store and manage sensitive information, such as passwords. Putting this information in a secret is safer and more flexible than hard coding it in a pod definition or a container image.

The ratings API expects to find the connection details to the MongoDB database in the form of `mongodb://<username>:<password>@<endpoint>:27017/ratingsdb`. Replace `<username>`, `<password>`, and `<endpoint>` with the ones you used when you created the database, for example, `mongodb://ratingsuser:ratingspassword@ratings-mongodb.ratingsapp:27017/ratingsdb`.

1. Use the `kubectl create secret generic` command to create a secret called `mongosecret` in the `ratingsapp` namespace. A Kubernetes secret can hold several items and is indexed by a key. In this case, the secret contains only one key, called `MONGOCONNECTION`. The value is the constructed connection string from the previous step. Replace `<username>` and `<password>` with the ones you used when you created the database.

   Bash

   ```
   kubectl create secret generic mongosecret \
       --namespace ratingsapp \
       --from-literal=MONGOCONNECTION="mongodb://<username>:<password>@ratings-mongodb.ratingsapp:27017/ratingsdb"
   ```

2. Run the `kubectl describe secret` command to validate that the secret.

   Bash

   ```
   kubectl describe secret mongosecret --namespace ratingsapp
   ```

   The output from this command looks similar to this example.

   Output

   ```
   Name:         mongosecret
   Namespace:    ratingsapp
   Labels:       <none>
   Annotations:  <none>

   Type:  Opaque

   Data
   ====
   ```

```
MONGOCONNECTION:  98 bytes
```

You now have an AKS cluster with a configured MongoDB database in a namespace called `ratingsapp`. In this namespace, you'll find the following resources:

- **Deployment/ratings-mongodb**: A deployment represents one or more identical pods managed by the Kubernetes Deployment Controller. This deployment defines the number of replicas (pods) to create for MongoDB. The Kubernetes Scheduler ensures that if pods or nodes encounter problems, additional pods are scheduled on healthy nodes.

- **Pod/ratings-mongodb-{random-string}**: Kubernetes uses pods to run an instance of MongoDB.

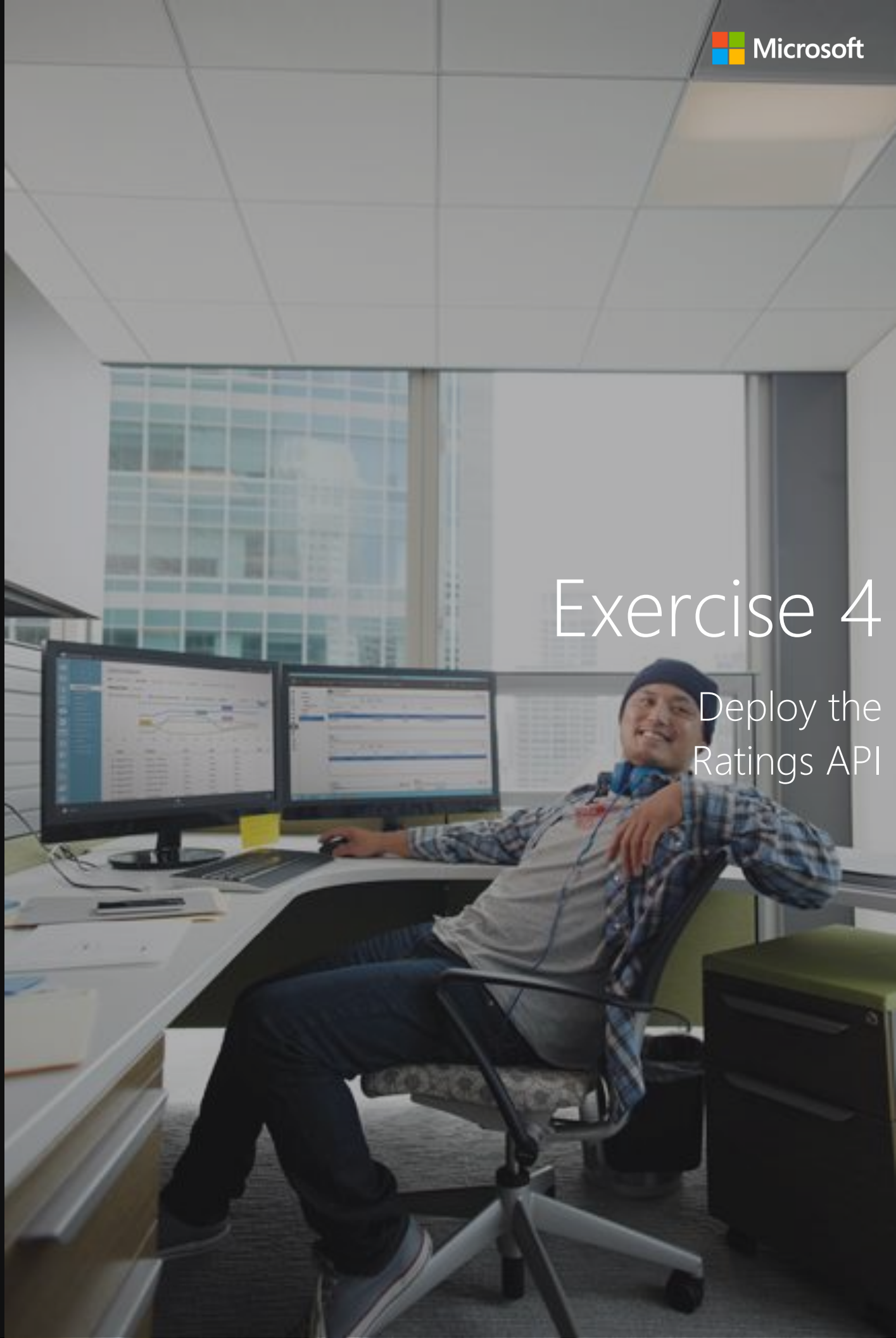
- **Service/ratings-mongodb**: To simplify the network configuration, Kubernetes uses services to group a set of pods and provide network connectivity logically. Connectivity to the MongoDB database is exposed via this service through the DNS name **ratings-mongodb.ratingsapp.svc.cluster.local**.

- **Secret/mongosecret**: A Kubernetes secret is used to inject sensitive data into pods, such as access credentials or keys. This secret holds the MongoDB connection details. You'll use it in the next unit to configure the API to communicate with MongoDB.

Summary

In this exercise, you configured the Helm stable repository, then used a Helm chart to deploy MongoDB to your cluster. You then created a Kubernetes secret to hold database credentials.

Next, you'll deploy the Fruit Smoothies ratings-api to your AKS cluster.



Exercise 4

Deploy the Ratings API

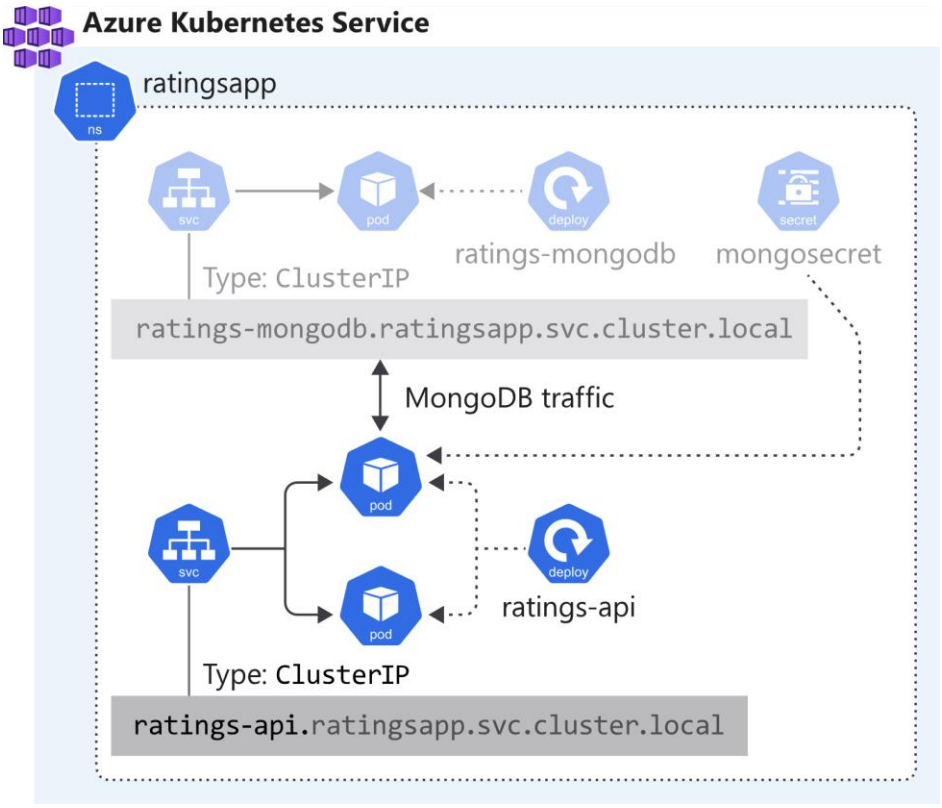
Exercise - Deploy the ratings API

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. Open this YAML file (click the link). Copy all the text and paste it to the editor.

```yaml
# DO NOT COPY THE YAML FROM HERE. USE THE DOWNLOAD LINK ABOVE.
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
this environment variable
          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 a replica 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 pod is 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.

```
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 pod transition to the `Running` state. Select `Ctrl+C` to stop watching.

```
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
```

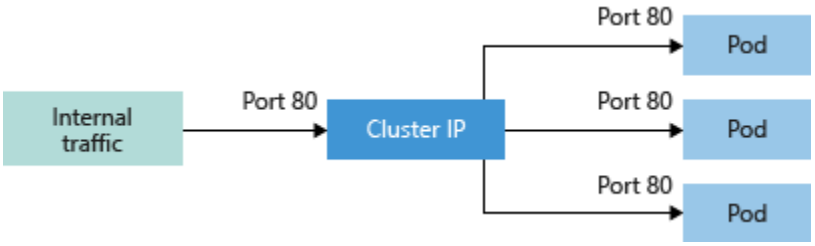
The deployment should show that one replica is ready.

```
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. Open this YAML file (click the link). Copy all the text and paste it to the editor.

   YAML
   ```yaml
   # DO NOT COPY THE YAML FROM HERE. USE THE DOWNLOAD LINK ABOVE.
   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   10.2.0.102   <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
   ```

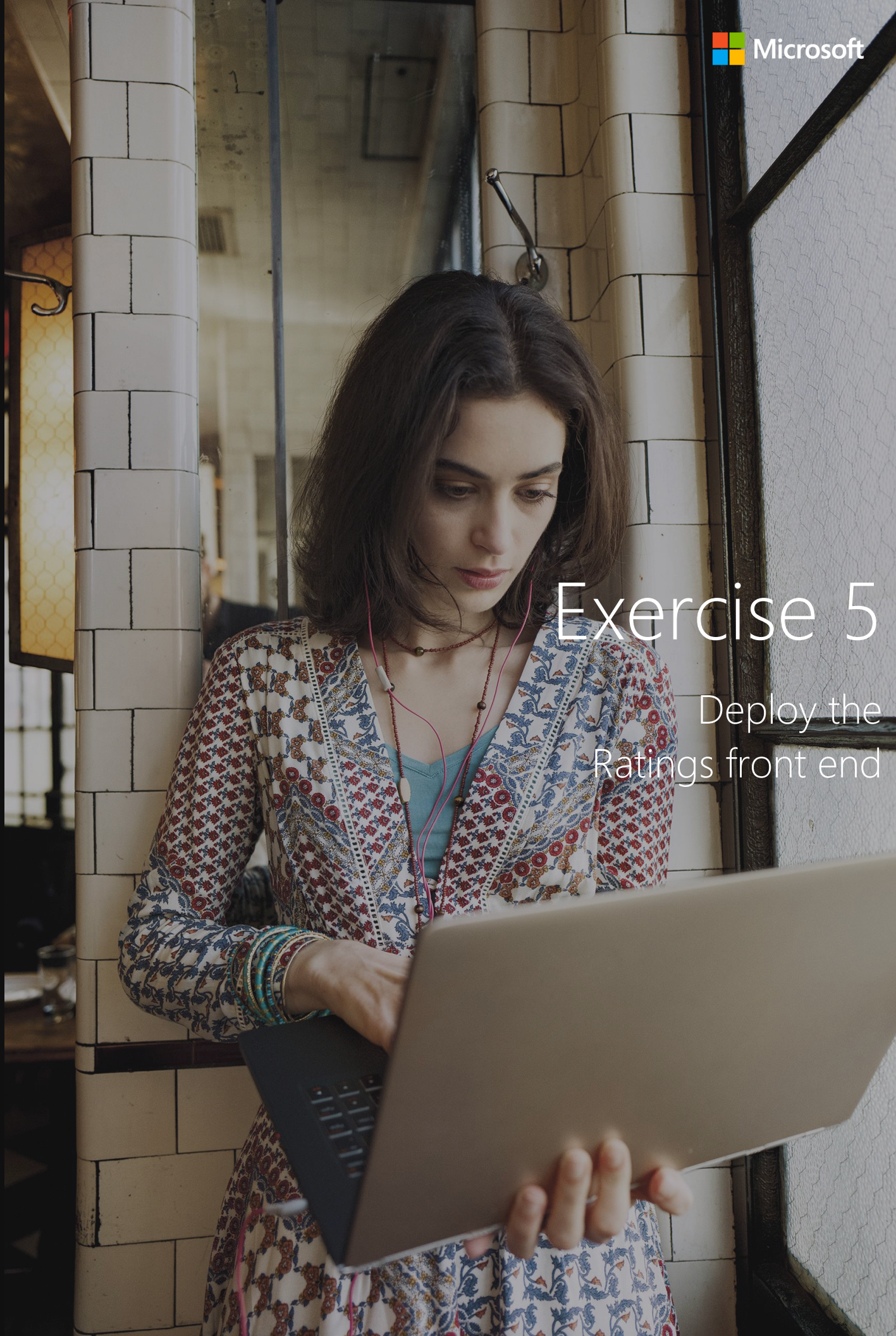
You've now created a deployment of the **ratings-api** and exposed it as an internal (ClusterIP) service.

- **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.



Microsoft

Exercise 5

Deploy the Ratings front end

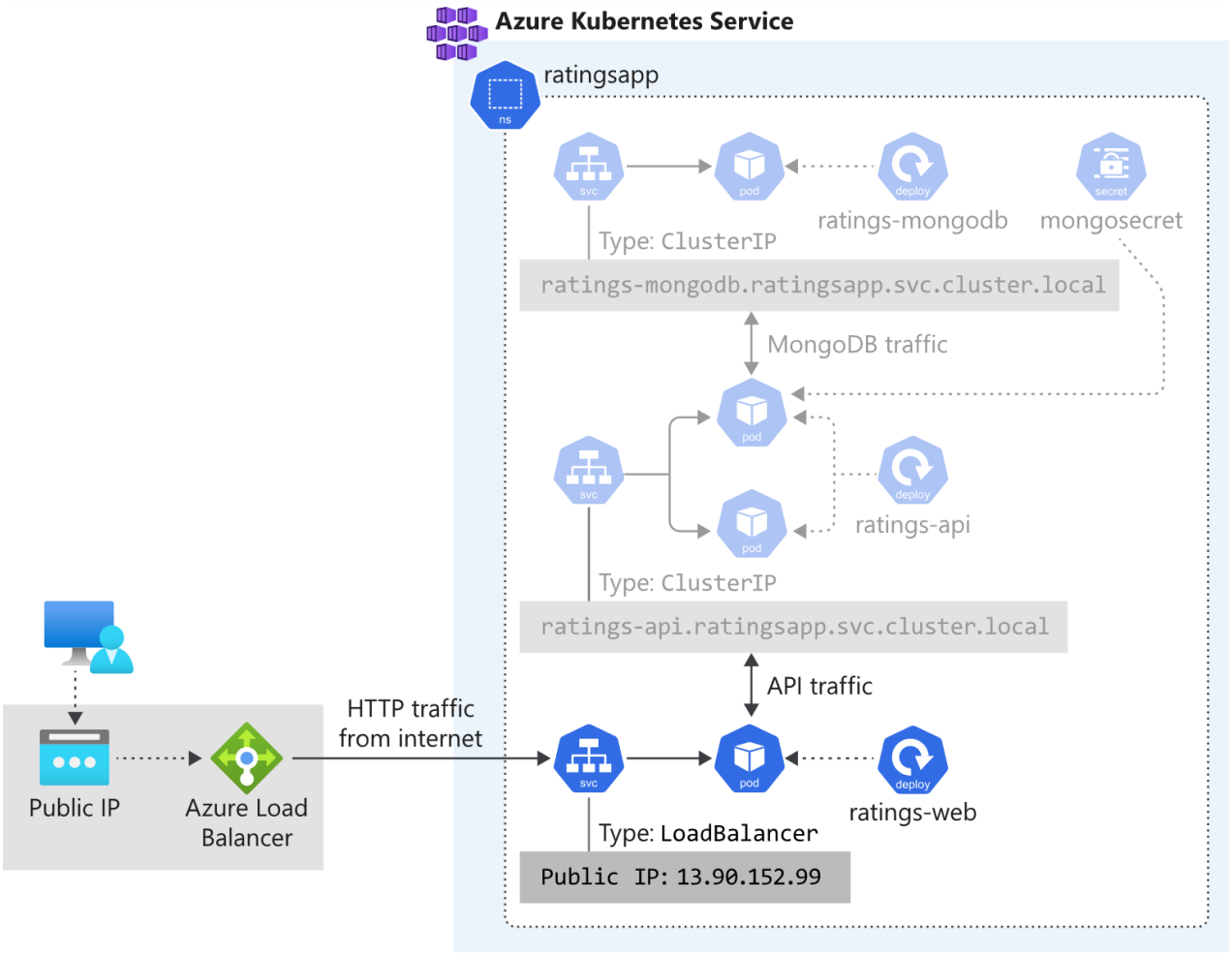
Exercise - Deploy the ratings front end

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 the ratings API. You'll continue your deployment and deploy the ratings web front end. The ratings web front end is a Node.js application. Recall that you've already created an Azure Container Registry instance. You used it to build a Docker image of the front end and store it in a repository.

In this exercise, you will:

- ✓ Create a Kubernetes deployment for the web front end
- ✓ Create a Kubernetes service manifest file to expose the web front end as a load-balanced service
- ✓ Test the web front end



Create a Kubernetes deployment for the ratings web front end

Let's start by creating a deployment for the ratings front end.

1. Create a file called `ratings-web-deployment.yaml` by using the integrated editor.

```Bash
code ratings-web-deployment.yaml
```

2. Open this YAML file (click the link). Copy all the text and paste it to the editor.

```YAML
# DO NOT COPY THE YAML FROM HERE. USE THE DOWNLOAD LINK ABOVE.
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ratings-web
spec:
  selector:
    matchLabels:
      app: ratings-web
  template:
    metadata:
      labels:
        app: ratings-web # the label for the pods and the deployments
    spec:
      containers:
      - name: ratings-web
        image: <acrname>.azurecr.io/ratings-web:v1 # IMPORTANT: update with your own repository
        imagePullPolicy: Always
        ports:
        - containerPort: 8080 # the application listens to this port
        env:
        - name: API # the application expects to connect to the API at this endpoint
          value: http://ratings-api.ratingsapp.svc.cluster.local
        resources:
          requests: # minimum resources required
            cpu: 250m
            memory: 64Mi
          limits: # maximum resources allocated
            cpu: 500m
            memory: 512Mi
```

3. In the `image` key update, the value replaces `<acrname>` with the name of your Container Registry instance.

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

   - `image`: You'll create a deployment running the image you pushed in the Container Registry instance you created earlier, for example, `acr4229.azurecr.io/ratings-web:v1`. The container listens to port `8080`. The deployment and the pods are labeled with `app=ratings-web`.

   - `env`: The ratings front end expects to connect to the API endpoint configured in an `API` environment variable. If you used the defaults and deployed the ratings API service in the `ratingsapp` namespace, the value of that should be `http://ratings-api.ratingsapp.svc.cluster.local`.

   - `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.

5. To save the file, select `Ctrl+S`. To close the editor, select `Ctrl+Q.`

6. Apply the configuration by using the `kubectl apply` command and deploy the application in the `ratingsapp` namespace.

   Bash

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

   You'll see an output like this example.

   Output

   ```
   deployment.apps/ratings-web 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-web`. Select `Ctrl+C` to stop watching.

   Bash

   ```
   kubectl get pods --namespace ratingsapp -l app=ratings-web -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-web-fcc464b8d-vck96    1/1     Running   0          37s
   ```

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-web --namespace ratingsapp
```

The deployment should show that one replica is ready.

```
Output

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

Create a Kubernetes service for the ratings web front end

Your next step is to simplify the network configuration for your application workloads. Use a Kubernetes service to group your pods and provide network connectivity.

You'll use a Kubernetes *LoadBalancer* instead of a *ClusterIP* for this service. A *LoadBalancer* allows you to expose a Kubernetes service on a public IP in the cluster. The type makes the service reachable from outside the cluster.

1. Create a file called `ratings-web-service.yaml` by using the integrated editor.

   ```
   Bash
   code ratings-web-service.yaml
   ```

2. Open this YAML file (click the link). Copy all the text and paste it to the editor.

   ```yaml
   YAML
   # DO NOT COPY THE YAML FROM HERE. USE THE DOWNLOAD LINK ABOVE.
   apiVersion: v1
   kind: Service
   metadata:
     name: ratings-web
   spec:
     selector:
       app: ratings-web
     ports:
     - protocol: TCP
       port: 80
       targetPort: 8080
     type: LoadBalancer
   ```

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

   - `selector`: The set of pods targeted by a service is determined by the selector. In the following example, Kubernetes load balances traffic to pods that have the label `app: ratings-web`. The 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 externally at port `80` and load balances the traffic to the ratings-web pods listening on port `8080`.

- `type`: A service of type `LoadBalancer` creates a public IP address in Azure and assigns it to Azure Load Balancer. Choosing this value makes the service reachable from outside the cluster.

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 to deploy the service in the `ratingsapp` namespace.

```
Bash

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

You'll see an output like this example.

```
Output

service/ratings-web created
```

6. Next, let's check the status of the service. It takes a few minutes for the service to acquire the public IP. Run the `kubectl get service` command with a *watch* by adding the `-w` flag to see it update in real time. Select `Ctrl+C` to stop watching.

```
Bash

kubectl get service ratings-web --namespace ratingsapp -w
```

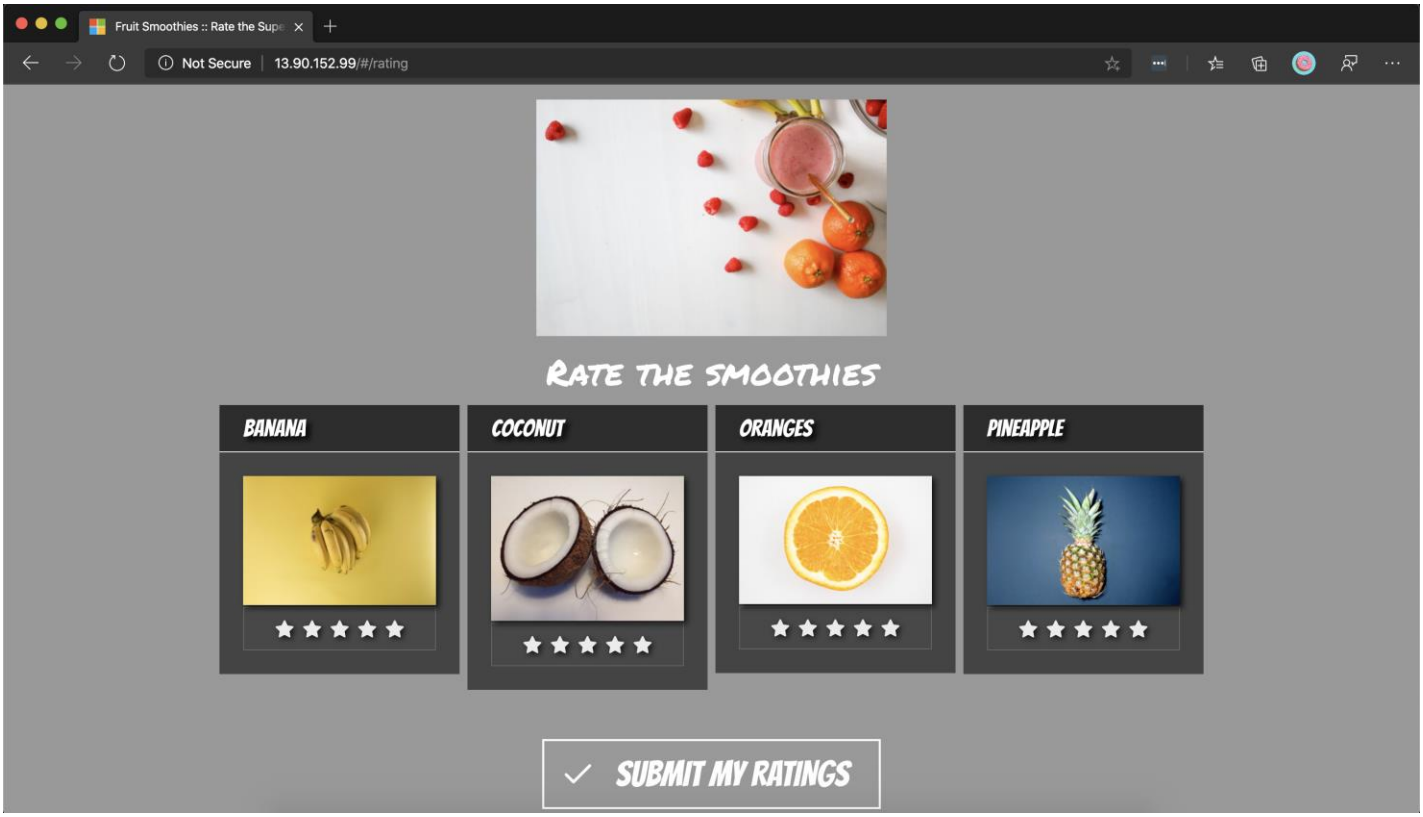
The service shows `EXTERNAL-IP` as `<pending>` for a while until it finally changes to an actual IP.

```
Output

NAME          TYPE           CLUSTER-IP   EXTERNAL-IP    PORT(S)         AGE
ratings-web   LoadBalancer   10.2.0.112   <pending>      80:32747/TCP    11s
ratings-web   LoadBalancer   10.2.0.112   13.90.152.99   80:32747/TCP    5m
```

Make note of that EXTERNAL-IP, for example, `13.90.152.99`. You'll use the address to access the application.

Test the application

Now that the ratings-web service has a public IP, open the IP in a web browser, for example, at http://13.90.152.99, to view and interact with the application.

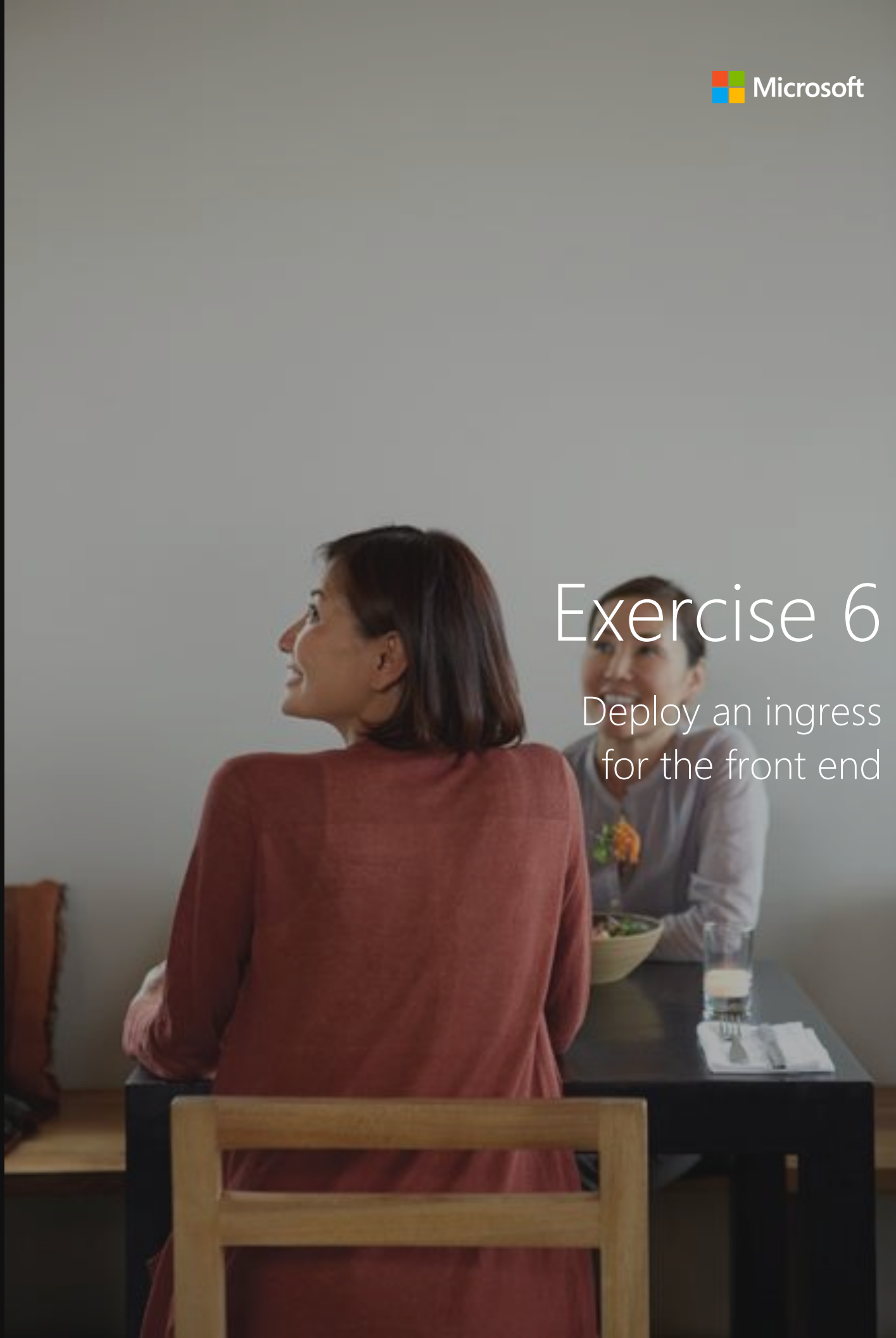


Summary

In this exercise, you created a deployment of **ratings-web** and exposed it to the internet through a LoadBalancer type service.

- **Deployment/ratings-web**: The web front end.
- **Service/ratings-web**: The load-balanced service, which is exposed on Azure Load Balancer through a public IP.

Next, we'll improve the network accessibility of the application by using Ingress.



Exercise 6

Deploy an ingress for the front end

Exercise - Deploy an ingress for the front end

In the previous units, you exposed the Fruit Smoothies' ratings website and RESTfull API in two different ways for allowing access to each instance. The API is exposed via a ratings-api service using a *ClusterIP* that creates an internal IP address for use within the cluster. Recall, choosing this value makes the service reachable only from within the cluster. The website is exposed via a ratings-web service using a *LoadBalancer* that creates a public IP address in Azure and assigns it to Azure Load Balancer. Recall, choosing this value makes the service reachable from outside the cluster.

Even though the load balancer exposes the ratings website via a publicly accessible IP, there are limitations that you need to consider.

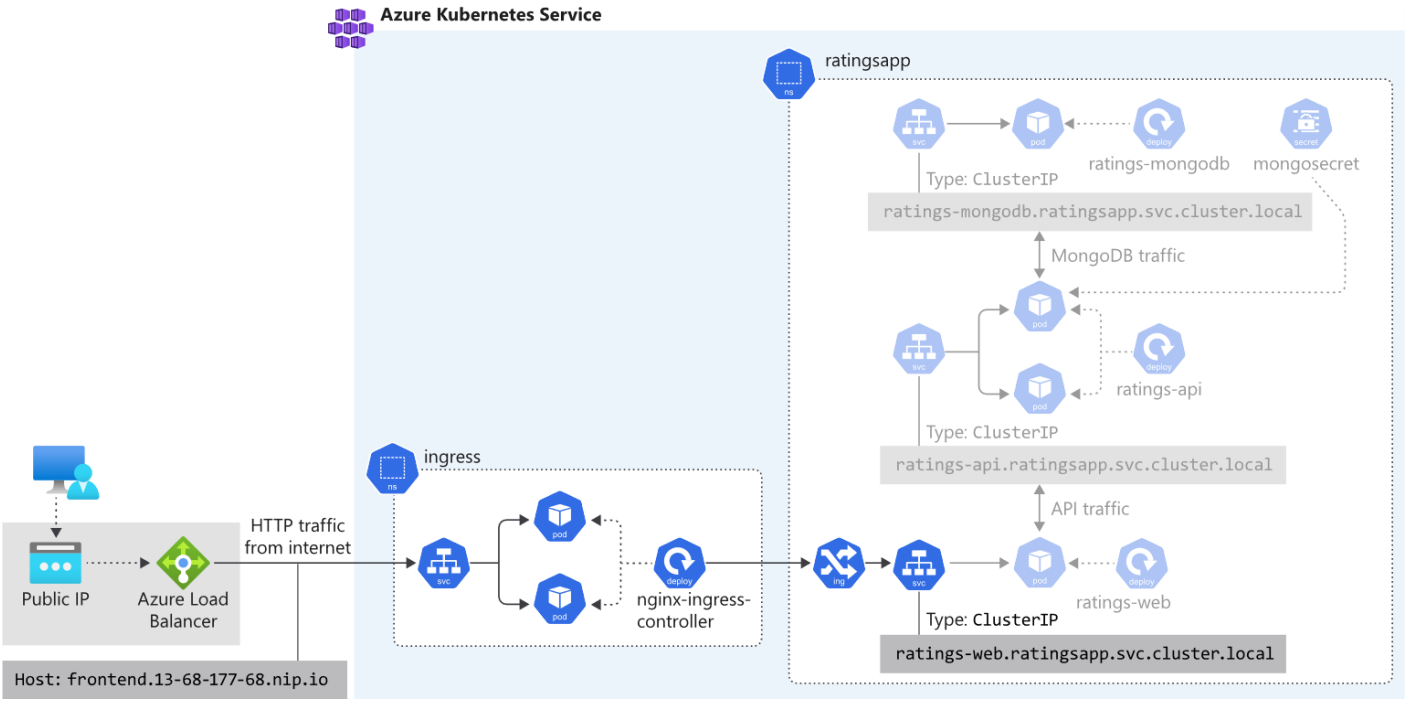
Let's assume the Fruit Smoothies' development team decides to extend the project by adding a video upload website. Fans of Fruit Smoothies can submit videos of how they're enjoying their smoothies at home, at the beach, or work. The current ratings website responds at `FruitSmoothies.com`. When you deploy the new video site, you want the new site to respond at `fruitsmoothies.com/videos` and the ratings site at `fruitsmoothies.com/ratings`.

If you continue to use the load balancer solution, you'll need to deploy a separate load balancer on the cluster and map its IP address to a new fully qualified domain name (FQDN), for example, `videos.fruitsmoothies.com`. To implement the required URL-based routing configuration, you'll need to install additional software outside of your cluster.

The extra effort is that a Kubernetes load balancer service is a Layer 4 load balancer. Layer 4 load balancers only deal with routing decisions between IPs addresses, TCP, and UDP ports. Kubernetes provides you with an option to simplify the above configuration by using an ingress controller.

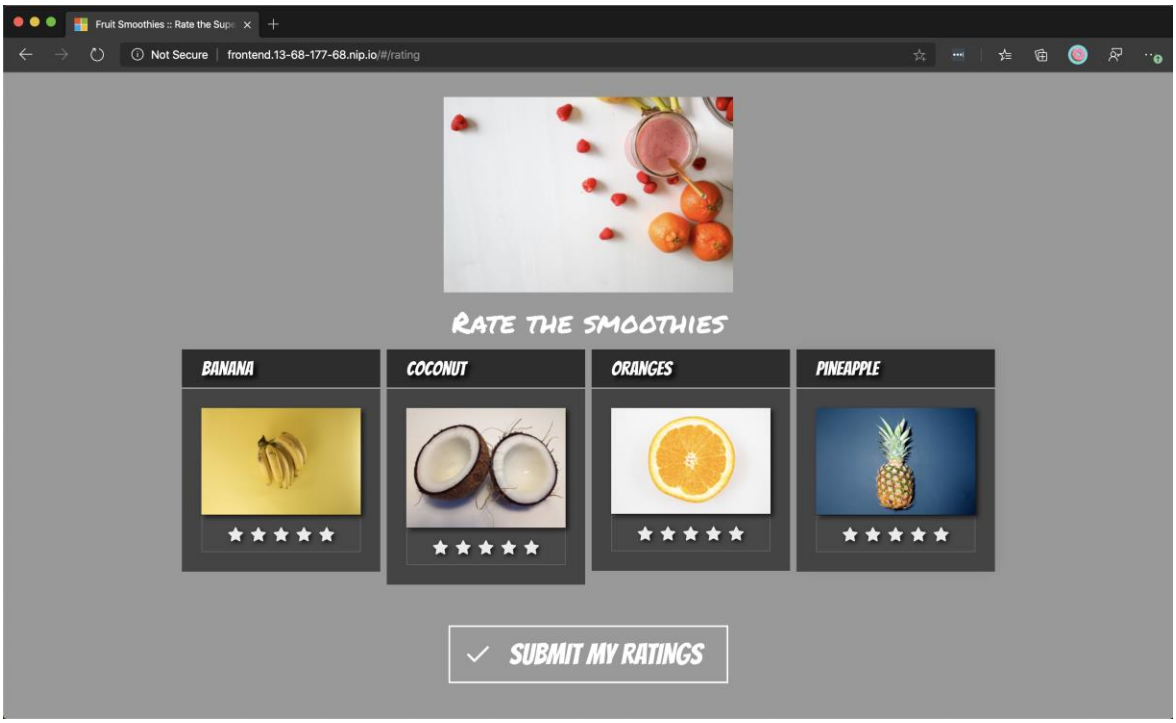
In this exercise, you will:

- ✓ Deploy a Kubernetes ingress controller running NGINX
- ✓ Reconfigure the ratings web service to use ClusterIP
- ✓ Create an Ingress resource for the ratings web service
- ✓ Test the application



Deploy a Kubernetes ingress controller running NGINX

A Kubernetes ingress controller is software that provides layer 7 load balancer features. These features include reverse proxy, configurable traffic routing, and TLS termination for Kubernetes services. You install the ingress controller and configure it to replace the load balancer. With the ingress controller, you can now do all load balancing, authentication, TSL/SSL, and URL-based routing configuration without the need for extra software outside of the cluster.



There are several options for running Kubernetes ingress on Azure Kubernetes Service (AKS), such as Azure Application Gateway, Ambassador, HAProxy, Kong, NGINX, and Traefik. The ingress controllers are exposed to the internet by using a Kubernetes service of type LoadBalancer. The ingress controller watches and implements Kubernetes ingress resources, which create routes to application endpoints. Here, you'll deploy a basic Kubernetes ingress controller by using NGINX. Then you'll configure the ratings front-end service to use that ingress for traffic.

NGINX ingress controller is deployed as any other deployment in Kubernetes. You can either use a deployment manifest file and specify the NGINX ingress controller image or you can use an nginx-ingress Helm chart. The NGINX helm chart simplifies the deployment configuration required for the ingress controller. For example, you don't need to define a configuration mapping or configure a service account for the NGINX deployment. Here, you'll use a Helm chart to install the ingress controller on your cluster.

1. Start by creating a namespace for the ingress.

```bash
kubectl create namespace ingress
```

2. Configure the Helm client to use the stable repository by running the `helm repo add` command below.

```bash
helm repo add stable https://kubernetes-charts.storage.googleapis.com/
```

3. Next, install the NGINX ingress controller. NGINX ingress is part of the stable Helm repository you configured earlier when you installed MongoDB. You'll install two replicas of the NGINX ingress controllers are deployed with the `--set controller.replicaCount` parameter for added redundancy. Make sure to schedule the controller only on Linux nodes as Windows Server nodes shouldn't run the ingress controller. You specify a node selector by using the `--set nodeSelector` parameter to tell the Kubernetes scheduler to run the NGINX ingress controller only on Linux-based nodes.

```bash
helm install nginx-ingress stable/nginx-ingress \
    --namespace ingress \
    --set controller.replicaCount=2 \
    --set controller.nodeSelector."beta\.kubernetes\.io/os"=linux \
    --set defaultBackend.nodeSelector."beta\.kubernetes\.io/os"=linux
```

4. After the installation is finished, you'll see an output similar to this example.

```
NAME: nginx-ingress
LAST DEPLOYED: Mon Jan  6 15:18:42 2020
NAMESPACE: ingress
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
The nginx-ingress controller has been installed.
It may take a few minutes for the LoadBalancer IP to be available.
You can watch the status by running 'kubectl --namespace ingress get services -o wide -w nginx-ingress-controller'
```

5. Next, let's check the public IP of the ingress service. It takes a few minutes for the service to acquire the public IP. Run the following command with a *watch* by adding the `-w` flag to see it update in real time. Select `Ctrl+C` to stop watching.

```bash
kubectl get service nginx-ingress-controller --namespace ingress -w
```

The service shows `EXTERNAL-IP` as `<pending>` for a while until it finally changes to an actual IP.

Output

```
NAME                       TYPE           CLUSTER-IP    EXTERNAL-IP      PORT(S)                      AGE
nginx-ingress-controller   LoadBalancer   10.2.0.162    13.68.177.68     80:32010/TCP,443:30245/TCP   3m30s
```

Make a note of that EXTERNAL-IP, for example, 13.68.177.68.

Reconfigure the ratings web service to use ClusterIP

There's no need to use a public IP for the service because we're going to expose the deployment through ingress. Here, you'll change the service to use `ClusterIP` instead of `LoadBalancer`.

1. Edit the file called `ratings-web-service.yaml` by using the integrated editor.

   Bash

   ```
   code ratings-web-service.yaml
   ```

2. Replace the existing content with the text in this YAML file (click the link). Note the change of the service `type` to `ClusterIP`.

   YAML

   ```yaml
   # DO NOT COPY THE YAML FROM HERE. USE THE DOWNLOAD LINK ABOVE.
   apiVersion: v1
   kind: Service
   metadata:
     name: ratings-web
   spec:
     selector:
       app: ratings-web
     ports:
     - protocol: TCP
       port: 80
       targetPort: 8080
     type: ClusterIP
   ```

3. To save the file, select `Ctrl+S`. To close the editor, select `Ctrl+Q`.

4. You can't update the value of `type` on a deployed service. You have to delete the service and re-create it with the changed configuration. Run the following command to delete the service.

```bash
kubectl delete service \
    --namespace ratingsapp \
    ratings-web
```

Then, run the following command to re-create the service.

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

Create an Ingress resource for the ratings web service

In order for your Kubernetes Ingress controller to route requests to the ratings-web service, you will need an Ingress resource. The Ingress resource is where you specify the configuration of the Ingress controller.

Each Ingress resource will contain one or more Ingress rules, which specify an optional host, a list of paths to evaluate in the request, and a backend to route the request to. These rules are evaluated to determine the route that each request should take.

Let's set up an Ingress resource with a route to the ratings-web service.

1. Edit the file called `ratings-web-ingress.yaml` by using the integrated editor.

   ```bash
   code ratings-web-ingress.yaml
   ```

2. Open this YAML file (click the link). Copy all the text and paste it to the editor.

   ```yaml
   # DO NOT COPY THE YAML FROM HERE. USE THE DOWNLOAD LINK ABOVE.
   apiVersion: networking.k8s.io/v1beta1
   kind: Ingress
   metadata:
     name: ratings-web-ingress
     annotations:
       kubernetes.io/ingress.class: nginx
   ```

```
spec:
  rules:
  - host: frontend.<ingress ip>.nip.io # IMPORTANT: update <ingress ip> with the dashed public IP of your
                                       # ingress, for example frontend.13-68-177-68.nip.io
    http:
      paths:
      - backend:
          serviceName: ratings-web
          servicePort: 80
        path: /
```

In this file, update the `<ingress ip>` value in the `host` key with the *dashed* public IP of your ingress that you retrieved earlier, for example, `frontend.13-68-177-68.nip.io.` This value allows you to access the ingress via a host name instead of an IP address. In the next unit, you'll configure SSL/TLS on that host name.

> ⓘ Note
>
> In this example, you use nip.io, which is a free service that provides wildcard DNS. You can use alternatives such as xip.io or sslip.io. Alternatively, you can use your domain name and set up the proper DNS records.

To save the file, select `Ctrl+S`. To close the editor, select `Ctrl+Q`.

3. Apply the configuration by using the `kubectl apply` command and deploy the ingress route file in the `ratingsapp` namespace.

Bash

```
kubectl apply \
    --namespace ratingsapp \
    -f ratings-web-ingress.yaml
```

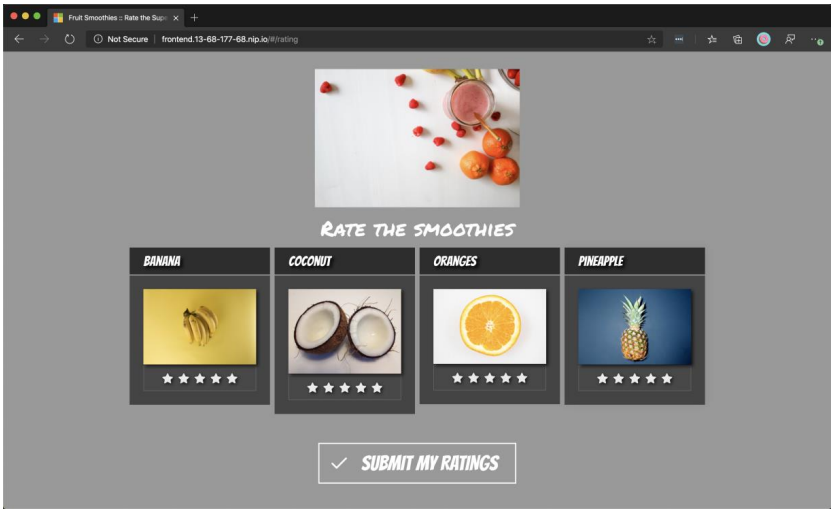
You'll see an output similar to this example.

Output

```
ingress.networking.k8s.io/ratings-web-ingress created
```

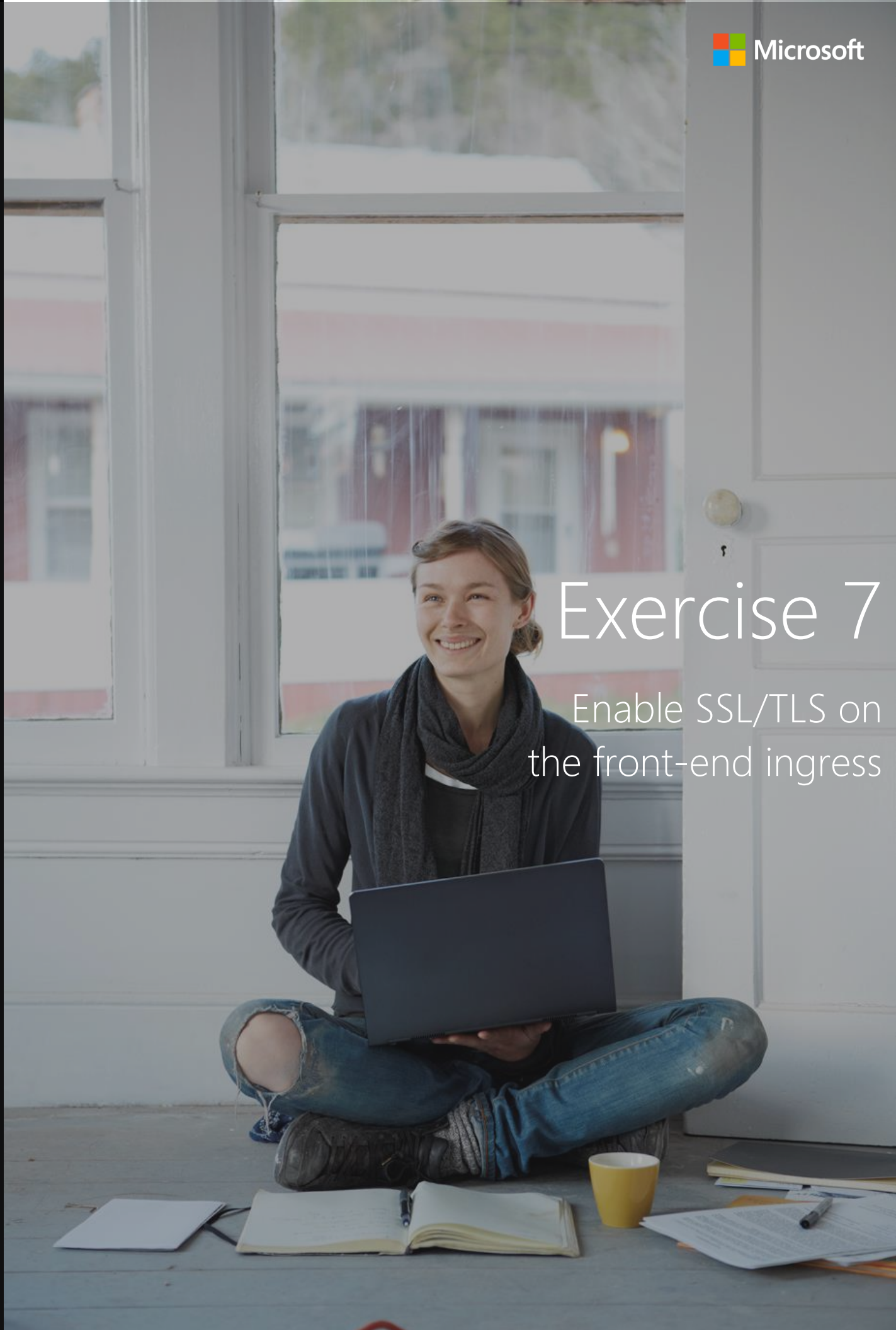
Test the application

Open the host name you configured on the ingress in a web browser to view and interact with the application. For example, at http://frontend.13-68-177-68.nip.io.



Summary

In this exercise, you deployed an NGINX Ingress controller and updated the **ratings-web** service to be accessible only from within the cluster. You then created an Ingress resource with a route to reverse proxy the deployment of the **ratings-web** service through a host name.



Microsoft

Exercise 7

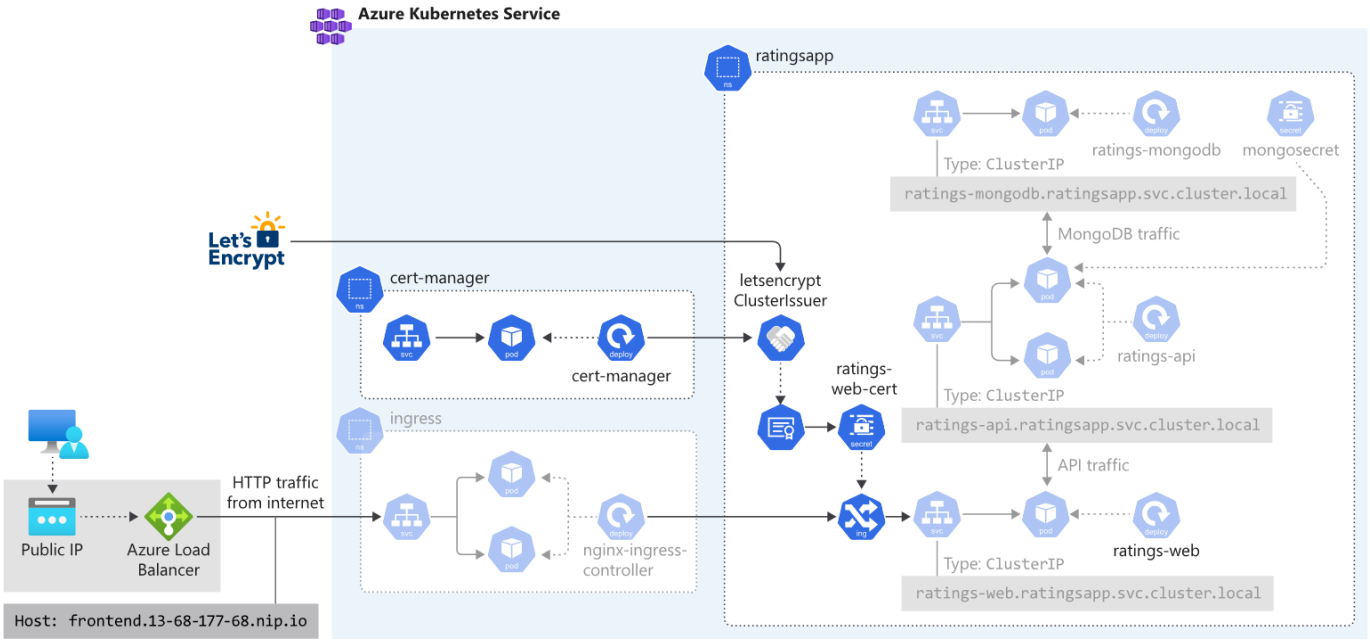
Enable SSL/TLS on the front-end ingress

Exercise - Enable SSL/TLS on the front-end ingress

The online security and privacy of user data is a primary concern for Fruit Smoothies as a company. It's important the ratings website allows HTTPS connections to all customers. NGINX ingress controller supports TLS termination and provides several ways to retrieve and configure certificates for HTTPS. This exercise demonstrates how to use *cert-manager*, which provides automatic *Let's Encrypt* certificate generation and management functionality.

In this exercise, you will:

- ✓ Deploy cert-manager by using Helm
- ✓ Deploy a ClusterIssuer resource for Let's Encrypt
- ✓ Enable SSL/TLS for the ratings web service on Ingress
- ✓ Test the application



Deploy cert-manager

cert-manager is a Kubernetes certificate management controller that makes it possible to automate certificate management in cloud-native environments. cert-manager supports various sources including Let's Encrypt, HashiCorp Vault, Venafi, simple signing key pairs, or self-signed certificates. You'll use `cert-manager` to ensure your website's certificate is valid and up to date, and attempt to renew certificates at a configured time before the certificate expires.

cert-manager uses Kubernetes custom resources. A Kubernetes custom resource is an object that allows you to extend the Kubernetes API or to introduce your API into a cluster. You use custom resource definition (CRD) files to define your object kinds and the API Server manage the lifecycle of the object.

Here, you'll use Helm to install cert-manager and then configure it to use Let's Encrypt as the certificate issuer.

1. Let's start by creating a namespace for cert-manager.

   Bash

   ```
   kubectl create namespace cert-manager
   ```

2. You'll use the Jetstack Helm repository to find and install cert-manager. First, you'll add the Jetstack Helm repository by running the code below.

   Bash

   ```
   helm repo add jetstack https://charts.jetstack.io
   helm repo update
   ```

3. Next, run the following command to install cert-manager by deploying the cert-manager CRD.

   Bash

   ```
   kubectl apply --validate=false -f https://raw.githubusercontent.com/jetstack/cert-manager/release-0.14/deploy/manifests/00-crds.yaml
   ```

4. Install the cert-manager Helm chart

   Bash

   ```
   helm install cert-manager \
       --namespace cert-manager \
       --version v0.14.0 \
       jetstack/cert-manager
   ```

5. You'll see output similar to the example below when the installation completes.

```
Output

NAME: cert-manager
LAST DEPLOYED: Tue Jan  7 13:11:19 2020
NAMESPACE: cert-manager
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
cert-manager has been deployed successfully!
```

6. Verify the installation by checking the `cert-manager` namespace for running pods.

```
Bash

kubectl get pods --namespace cert-manager
```

You'll see that the `cert-manager`, `cert-manager-cainjector`, and `cert-manager-webhook` pod is in a `Running` state. It might take a couple of minutes to provision the web hook required for the TLS assets.

```
Output

NAME                                       READY   STATUS    RESTARTS   AGE
cert-manager-5c6866597-zw7kh               1/1     Running   0          2m
cert-manager-cainjector-577f6d9fd7-tr77l   1/1     Running   0          2m
cert-manager-webhook-787858fcdb-nlzsq      1/1     Running   0          2m
```

Deploy a ClusterIssuer resource for Let's Encrypt

Cert-manager will ensure that your website's certificate is valid and up to date, and even attempt to renew certificates at a configured time before the certificate expires. However, you need to set up a *ClusterIssuer* before you can begin the certificate issuing process. The cluster issuer acts as an interface to a certificate-issuing service such as Let's Encrypt.

Let's Encrypt is a nonprofit Certificate Authority that provides TLS certificates. Let's Encrypt allows you to set up an HTTP server and have it automatically obtain a browser-trusted certificate. The process of retrieving and installing a certificate is fully automated without human intervention and managed by running a certificate management agent on the webserver. For more information about Let's Encrypt, see the *learn more* section at the end of this module.

1. Edit the file called `cluster-issuer.yaml` by using the integrated editor.

   Bash
   ```
   code cluster-issuer.yaml
   ```

2. Open this YAML file (click the link). Copy all the text and paste it to the editor.

   YAML
   ```yaml
   # DO NOT COPY THE YAML FROM HERE. USE THE DOWNLOAD LINK ABOVE.
   apiVersion: cert-manager.io/v1alpha2
   kind: ClusterIssuer
   metadata:
     name: letsencrypt
   spec:
     acme:
       server: https://acme-v02.api.letsencrypt.org/directory
       email: <your email> # IMPORTANT: Replace with a valid email from your organization
       privateKeySecretRef:
         name: letsencrypt
       solvers:
       - http01:
           ingress:
             class: nginx
   ```

   In the `email` key, you'll update the value by replacing `<your email>` with a valid certificate administrator email from your organization.

3. To save the file, select `Ctrl+S`. To close the editor, select `Ctrl+Q`.

4. Apply the configuration by using the `kubectl apply` command. Deploy the cluster-issuer configuration in the `ratingsapp` namespace.

Bash

```
kubectl apply \
    --namespace ratingsapp \
    -f cluster-issuer.yaml
```

You'll see an output similar to this example.

Output

```
clusterissuer.cert-manager.io/letsencrypt created
```

Enable SSL/TLS for the ratings web service on Ingress

The last part of the configuration is to configure the Kubernetes Ingress file for the ratings web service to enable SSL/TLS.

1. Edit the file called `ratings-web-ingress.yaml` by using the integrated editor.

   Bash

   ```
   code ratings-web-ingress.yaml
   ```

2. Replace the existing content with the text in this YAML file (click the link). Note the addition of the `cert-manager.io/issuer` annotation and the new `tls` section.

   YAML

   ```
   # DO NOT COPY THE YAML FROM HERE. USE THE DOWNLOAD LINK ABOVE.
   apiVersion: networking.k8s.io/v1beta1
   kind: Ingress
   metadata:
     name: ratings-web-ingress
     annotations:
       kubernetes.io/ingress.class: nginx
       cert-manager.io/cluster-issuer: letsencrypt
   spec:
     tls:
       - hosts:
         - frontend.<ingress ip>.nip.io # IMPORTANT: update <ingress ip> with the dashed public IP of your ingress, for example frontend.13-68-177-68.nip.io
         secretName: ratings-web-cert
     rules:
     - host: frontend.<ingress ip>.nip.io # IMPORTANT: update <ingress ip> with the dashed public IP of your ingress, for example frontend.13-68-177-68.nip.io
       http:
         paths:
         - backend:
             serviceName: ratings-web
             servicePort: 80

           path: /
   ```

In this file, update the `<ingress ip>` value in the `host` key with the *dashed* public IP of the ingress you retrieved earlier, for example, frontend.13-68-177-68.nip.io. This value allows you to access the ingress via a host name instead of an IP address.

3. To save the file, select `Ctrl+S`. To close the editor, select `Ctrl+Q`.

4. Apply the configuration by using the `kubectl apply` command. Deploy the updated Kubernetes ingress file in the `ratingsapp` namespace.

```
Bash

kubectl apply \
    --namespace ratingsapp \
    -f ratings-web-ingress.yaml
```

You'll see an output similar to this example.

```
Output

ingress.networking.k8s.io/ratings-web-ingress configured
```

5. Verify that the certificate was issued.

```
Bash

kubectl describe cert ratings-web-cert --namespace ratingsapp
```

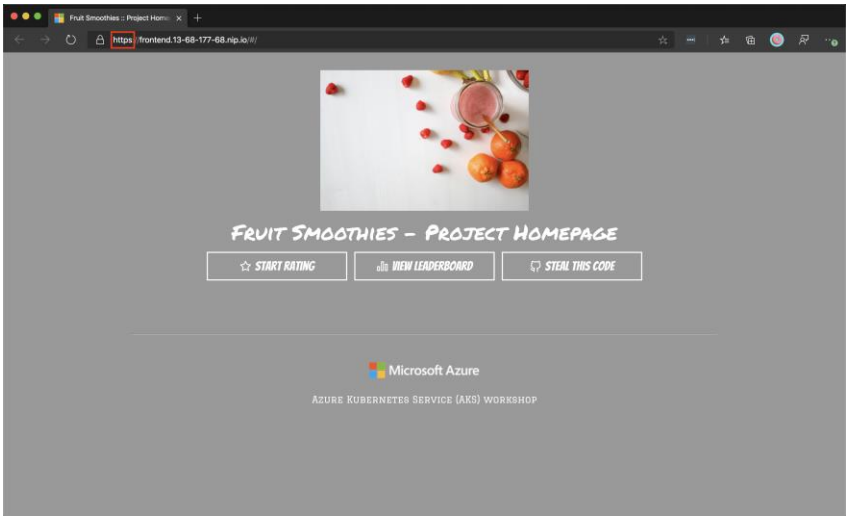
You'll get an output similar to this example.

```
Output

Name:         ratings-web-cert
Namespace:    ratingsapp
API Version:  cert-manager.io/v1alpha2
Kind:         Certificate

[..]

Spec:
  Dns Names:
    frontend.13-68-177-68.nip.io
  Issuer Ref:
    Group:      cert-manager.io
    Kind:       ClusterIssuer
    Name:       letsencrypt
  Secret Name:  ratings-web-cert
Status:
  Conditions:
    Last Transition Time:  2020-01-07T22:27:23Z
```

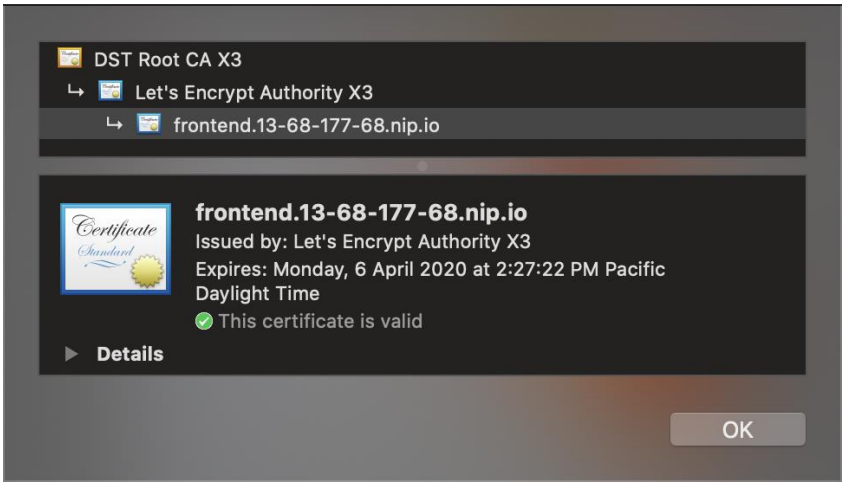
```
    Message:               Certificate is up to date and has not expired
    Reason:                Ready
    Status:                True
    Type:                  Ready
  Not After:               2020-04-06T21:27:22Z
Events:
  Type    Reason        Age   From          Message
  ----    ------        ----  ----          -------
  Normal  GeneratedKey  36s   cert-manager  Generated a new private key
  Normal  Requested     36s   cert-manager  Created new CertificateRequest resource "ratings-web-
cert-1603291776"
  Normal  Issued        34s   cert-manager  Certificate issued successfully
```

Test the application

Open the host name you configured on the ingress in a web browser over SSL/TLS to view and interact with the application. For example, at https://frontend.13-68-177-68.nip.io.



Verify that the front end is accessible over HTTPS and that the certificate is valid.



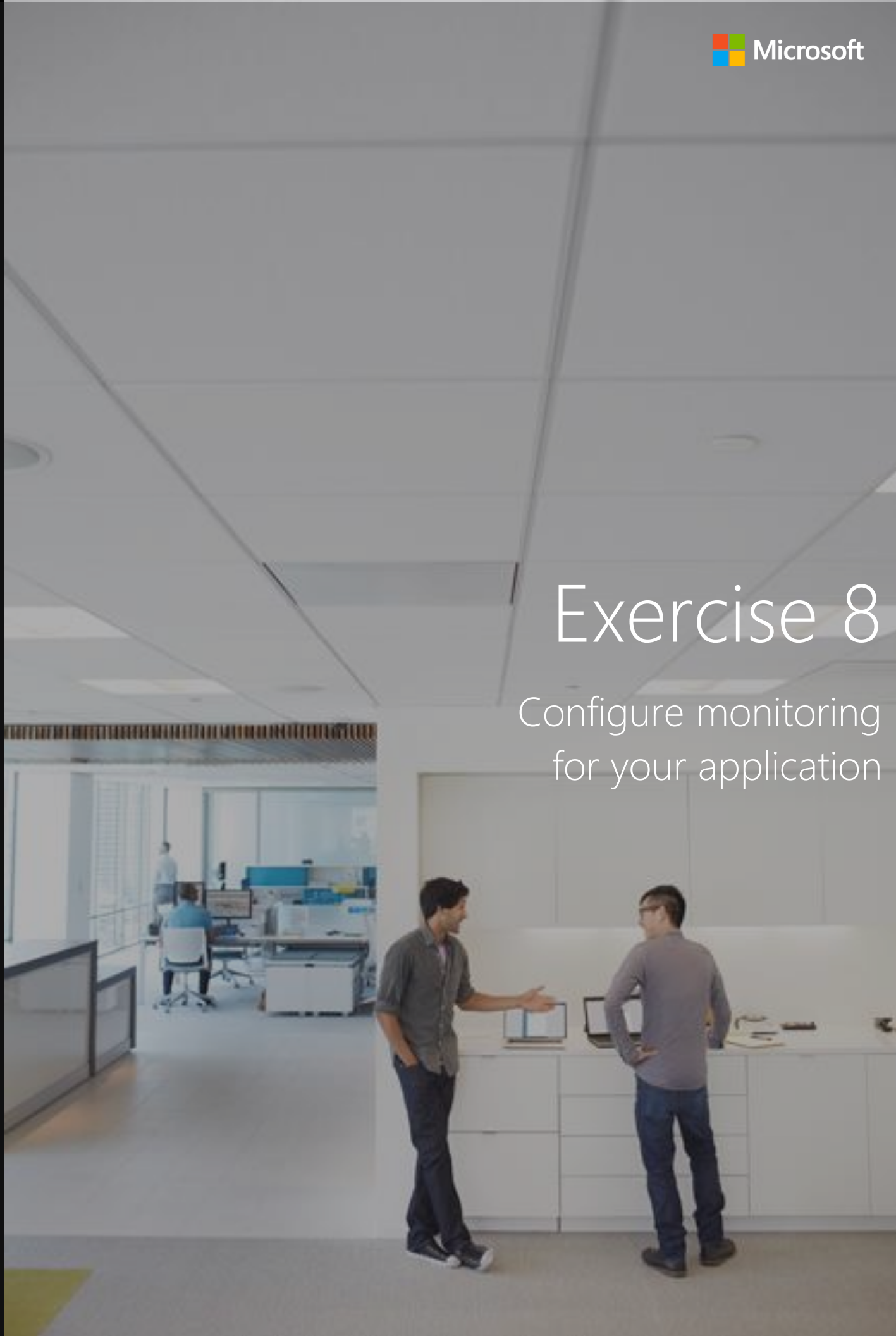
Summary

In this exercise, you deployed cert-manager and configured it to issue Let's Encrypt certificates automatically. You then configured the ingress you created earlier to serve encrypted TLS/SSL traffic through the generated certificates. Next, you'll configure monitoring for your AKS cluster.

Microsoft

Exercise 8

Configure monitoring for your application



Exercise - Configure monitoring for your application

The success of Fruit Smoothies' marketing campaign is the ongoing performance of the ratings website. The performance is depended on your cluster's performance and relies on the fact that you can monitor the different components in your application, view logs, and get alerts whenever your application goes down or some parts of it fail. You can use a combination of available tools to set up alerting capabilities for your application.

In this exercise, you will:

- ✓ Create a Log Analytics workspace
- ✓ Enable the AKS monitoring add-on
- ✓ Inspect the AKS event logs and monitor cluster health
- ✓ Configure Kubernetes RBAC to enable live log data
- ✓ View the live container logs and AKS events

Create a Log Analytics workspace

Azure Monitor for containers is a comprehensive monitoring solution for Azure Kubernetes Service. This solution gives you insight into the performance of your cluster by collecting memory and processor metrics from controllers, nodes, and containers.

You use Log Analytics in Azure Monitor to store monitoring data, events, and metrics from your AKS cluster and the applications. First, you'll pre-create the Log Analytics workspace in your assigned environment resource group.

1. You need a unique name for the workspace. Run the command below in Cloud Shell to generate a name similar to **aksworkshop-workspace-12345**.

   Bash

   ```
   WORKSPACE=aksworkshop-workspace-$RANDOM
   ```

2. Run the `az resource create` command to create the workspace in the same resource group and region as your Azure Kubernetes Service (AKS) cluster. For example, **aksworkshop** in **East US**.

   Bash

   ```
   az resource create --resource-type Microsoft.OperationalInsights/workspaces \
           --name $WORKSPACE \
           --resource-group $RESOURCE_GROUP \
           --location $REGION_NAME \
           --properties '{}' -o table
   ```

Enable the AKS monitoring add-on

Once the workspace is ready, you can integrate the Azure Monitor add-on and enable container monitoring on your AKS cluster.

1. You need to provide the resource ID of your workspace to enable the add-on. Run the following command to retrieve and store the workspace ID in a Bash variable named `WORKSPACE_ID.`

   Azure CLI

   ```
   WORKSPACE_ID=$(az resource show --resource-type Microsoft.OperationalInsights/workspaces \
       --resource-group $RESOURCE_GROUP \
       --name $WORKSPACE \
       --query "id" -o tsv)
   ```

2. Next, enable the monitoring add-on by running the `az aks enable-addons` command.

   Bash

   ```
   az aks enable-addons \
       --resource-group $RESOURCE_GROUP \
       --name $AKS_CLUSTER_NAME \
       --addons monitoring \
       --workspace-resource-id $WORKSPACE_ID
   ```

   > Note
   >
   > It might take some time to establish monitoring data flow for newly created clusters. Allow at least 5 to 10 minutes for data to appear for your cluster

Inspect the AKS event logs and monitor cluster health

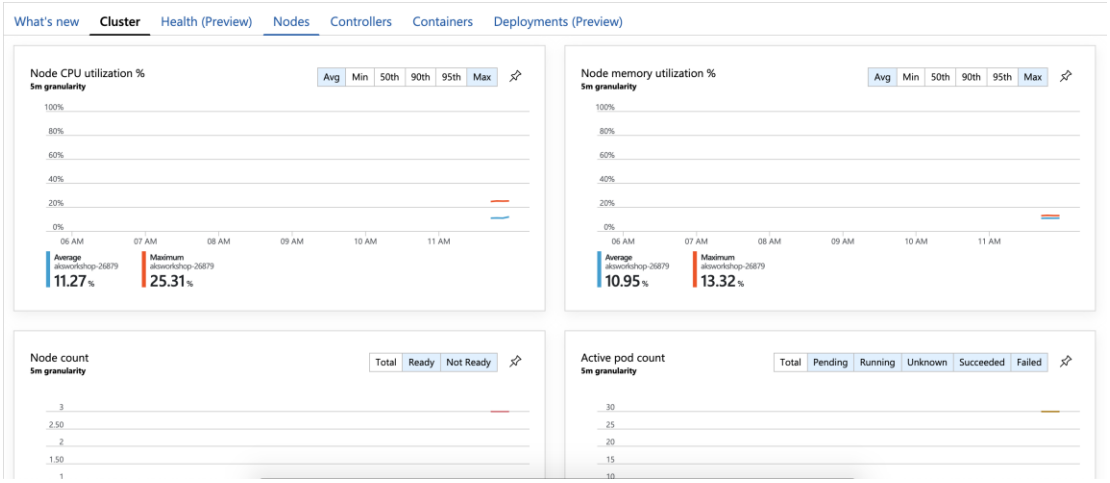
We view utilization reports and charts for your cluster in the Azure portal by using Azure Monitor. Azure Monitor gives you a global perspective of all containers deployed across subscriptions and resource groups. From here, you can track containers that are monitored and those containers that aren't monitored. You can also inspect each container's statistics individually.

Let's look at the steps you need to take to get a detailed view of the health of nodes and pods in a cluster.

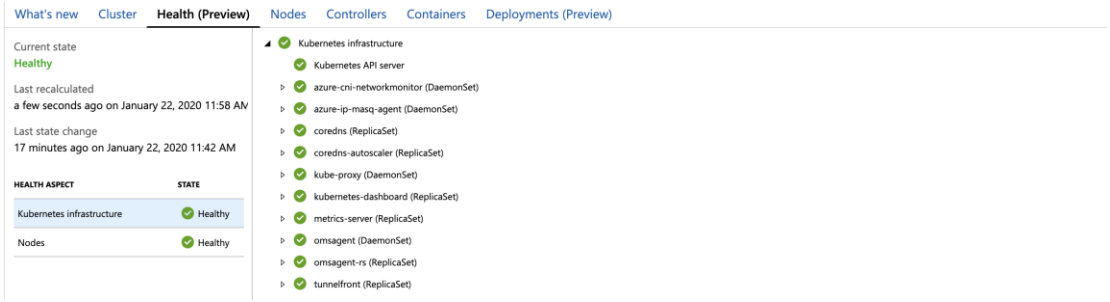
1. Sign in to the Azure portal.

   Azure Portal

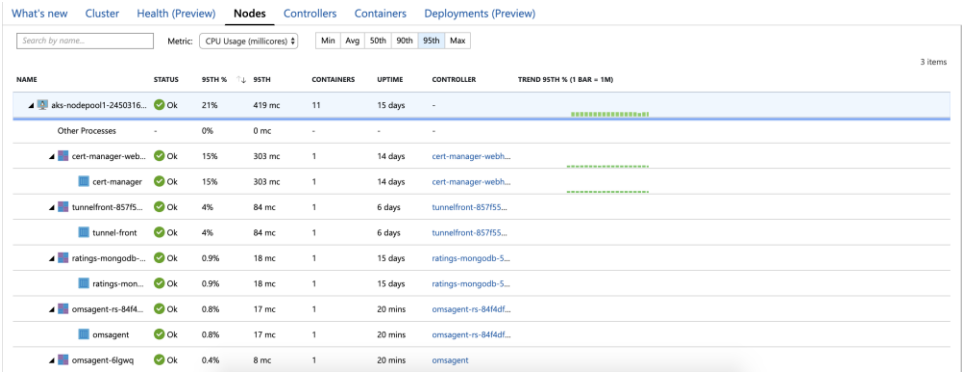
2. Select **Azure Monitor** from the left pane in the Azure portal.
3. Under the **Insights** section, select **Containers** to see a list of all clusters that you have access to.
4. Select the **Cluster** tab at the top of the view to check the cluster utilization. Notice how this view is again a high-level view that provides you a view on the cluster, nodes, controllers, and containers.



5. Select the **Health** tab at the top of the view to get a view on how the AKS infrastructure services of the cluster are doing.



6. Select the **Nodes** tab at the top of the view to get a detailed view of your nodes' health and pods in the cluster.



Configure Kubernetes RBAC to enable live log data

In addition to the high-level overview of your cluster's health, you can also view live log data of specific containers.

To enable and set permissions for the agent to collect the data, first, create a *Role* that has access to pod logs and events. Then you'll assign permissions to users by using *RoleBinding*.

What is role-based access control (RBAC)?

We use role-based access control (RBAC) in Kubernetes as a way of regulating access to resources based on the roles of individual users within your organization. RBAC authorization uses a set of related paths in the Kubernetes API to allow you to dynamically configure policies. The RBAC API defines four Kubernetes objects:

- Role
- ClusterRole
- RoleBinding
- ClusterRoleBinding

What is a Kubernetes Role?

The RBAC Role and ClusterRole objects allow you to set up rules that represent a set of permissions. The main difference between a Role and a ClusterRole is that a Role is used with resources in a specific namespace and ClusterRole is used with non-namespace resources in a cluster. You'll see how to define a ClusterRole later in the exercise.

What is a Kubernetes RoleBinding?

We use a role binding to grant the permissions defined in a role to a user or set of users. A role binding contains the list of users, groups, or service accounts, and a reference to the role being granted. Like the Role and ClusterRole, a RoleBinding grants permission within a specific namespace and the ClusterRoleBinding grants access to the cluster. You'll use a ClusterRoleBinding bind your ClusterRole to all the namespaces in your cluster.

In this exercise, you'll set up *ClusterRoles* and *ClusterRoleBindings* that aren't limited to a specific namespace. You configure *CusterRoles* to define permissions on namespaced resources given within individual namespaces or across all namespaces. *CusterRoles* are also used to describe permissions on cluster-scoped resources. You then use the *ClusterRoleBindings* to grant permissions across a whole cluster.

1. Create a file called `logreader-rbac.yaml` by using the integrated editor in Cloud Shell.

   Bash

   ```
   code logreader-rbac.yaml
   ```

2. Open this YAML file (click the link). Copy all the text and paste it to the editor.

YAML

```yaml
# DO NOT COPY THE YAML FROM HERE. USE THE DOWNLOAD LINK ABOVE.
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
    name: containerHealth-log-reader
rules:
- apiGroups: ["", "metrics.k8s.io", "extensions", "apps"]
  resources:
  - "pods/log"
  - "events"
  - "nodes"
  - "pods"
  - "deployments"
  - "replicasets"
  verbs: ["get", "list"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
    name: containerHealth-read-logs-global
roleRef:
    kind: ClusterRole
    name: containerHealth-log-reader
    apiGroup: rbac.authorization.k8s.io
subjects:
- kind: User
  name: clusterUser
  apiGroup: rbac.authorization.k8s.io
```

3. To save the file, select `Ctrl+S`. To close the editor, select `Ctrl+Q`.

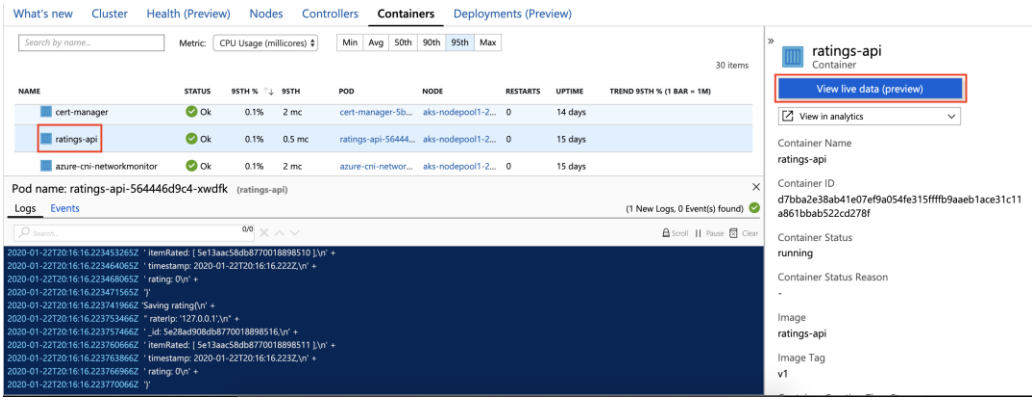
4. Apply the configuration by using the `kubectl apply` command.

Bash

```
kubectl apply \
    -f logreader-rbac.yaml
```

View the live container logs and AKS events

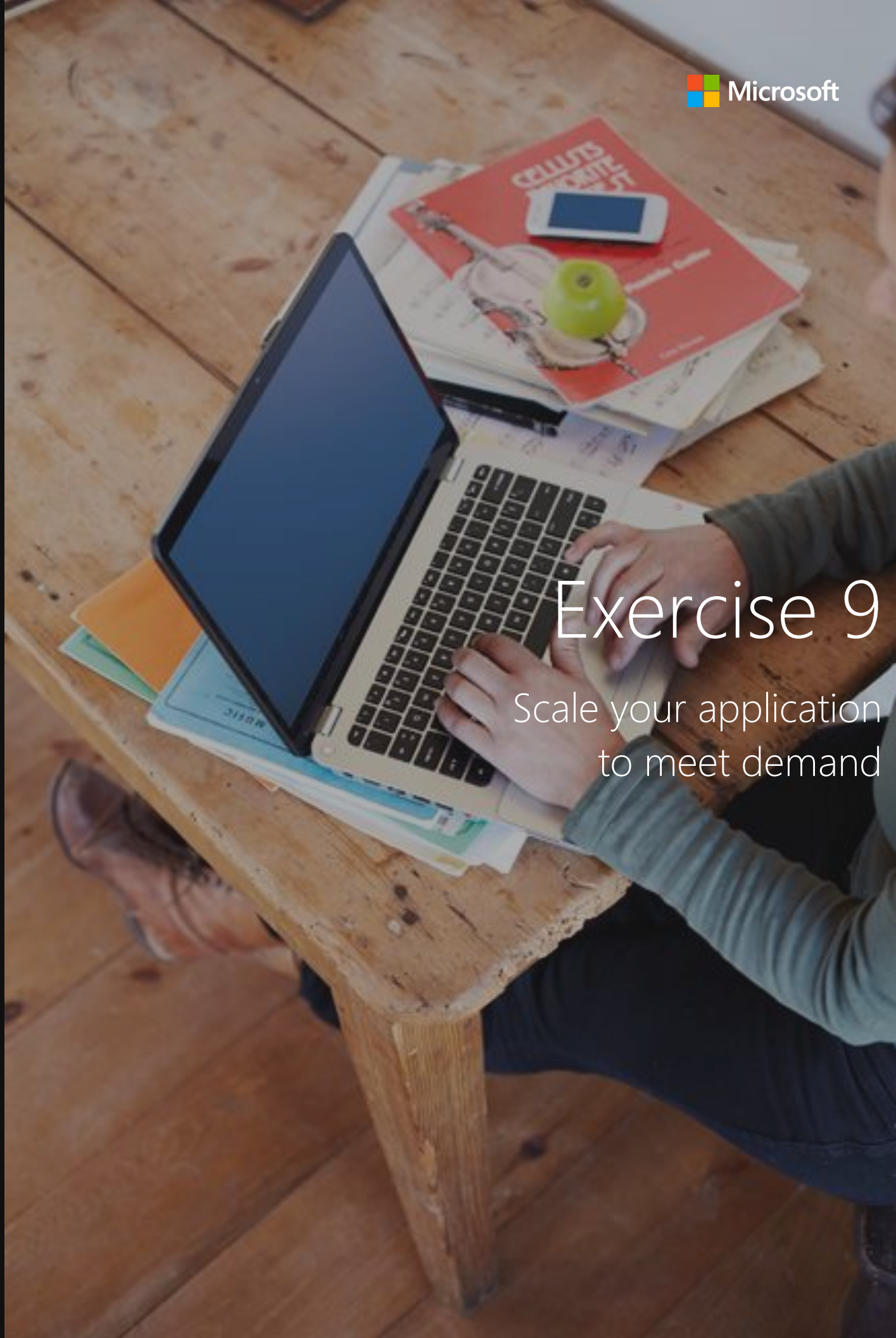
1. Switch back to the AKS cluster in the Azure portal.
2. Select **Insights** under **Monitoring**.
3. Select the **Controllers** tab, and choose a container to view its live logs or event logs. For example, choose the **ratings-api** container. The new view allows you to debug the status of the container.



Summary

In this exercise, you created a Log Analytics workspace in Azure Monitor to store monitoring and logging data for your AKS cluster. You enabled the AKS monitoring add-on to enable the collection of data, and inspected the AKS cluster health. You then used Kubernetes RBAC to enable the collection of live logging data and then viewed live log data in the Azure portal.

Next, we'll take a look at scaling the Fruit Smoothies AKS cluster.



Microsoft

Exercise 9

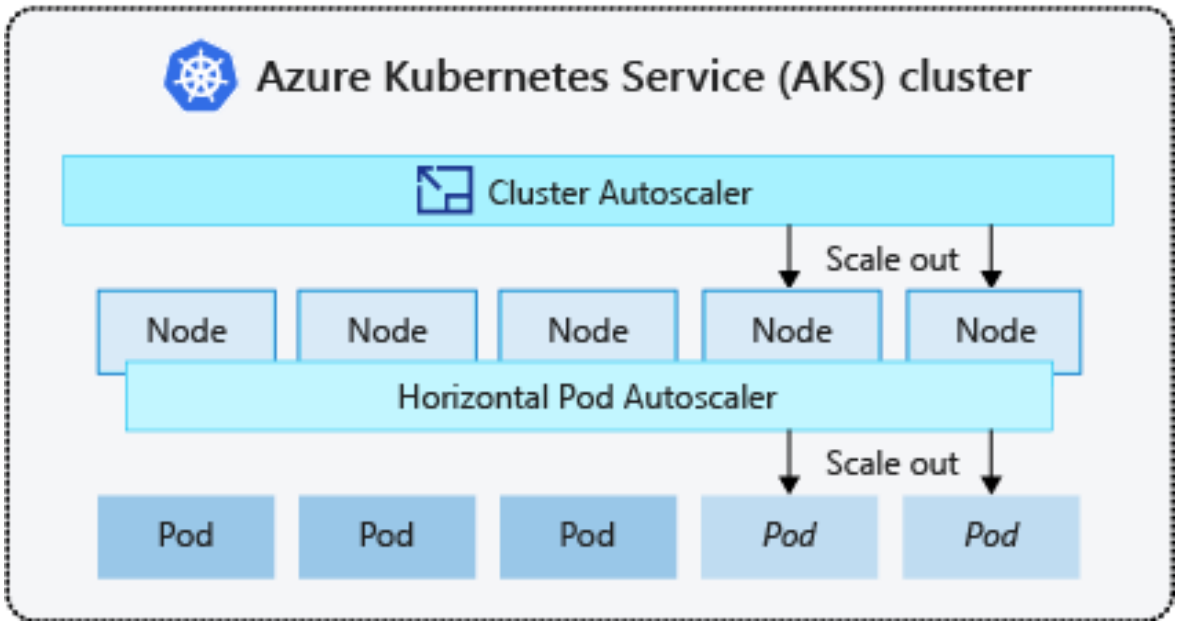
Scale your application to meet demand

Exercise - Scale your application to meet demand

Fruit Smoothies has shops worldwide with a large follower base and the expectation is that many users will use the ratings website to rate their favorite smoothy flavor. As the popularity of your application grows, the application needs to scale appropriately to manage demand changes. You have to ensure that your application remains responsive as the number of ratings increases.

In this exercise, you'll:

- ✓ Create an AKS horizontal pod autoscaler
- ✓ Run a load test with horizontal pod autoscaler enabled
- ✓ Autoscale the AKS cluster



Create the horizontal pod autoscaler

With increased traffic, the `ratings-api` container is unable to cope with the number of requests coming through. To fix the bottleneck, you can deploy more instances of that container.

We have two options to choose from when you need to scale out container instances in AKS. You can either manually increase the number of replicas in the deployment or use the horizontal pod autoscaler.

What is a horizontal pod autoscaler (HPA)?

The horizontal pod autoscaler (HPA) controller is Kubernetes control loop that allows the Kubernetes controller manager to query resource usage against the metrics specified in a *HorizontalPodAutoscaler* definition. The HPA controller calculates the ratio between a desired metric value specified in its definition file and the current metric value measured. The HPA automatically scales the number of pods up or down based on the calculated value.

HPA allows AKS to detect when your deployed pods need more resources based on metrics such as CPU. HPA can then schedule more pods onto the cluster to cope with the demand. You can configure HPA by using the `kubectl autoscale` command, or you can define the HPA object in a YAML file.

1. Create a file called `ratings-api-hpa.yaml` by using the integrated editor.

   Bash
   ```
   code ratings-api-hpa.yaml
   ```

2. Open this YAML file (click the link). Copy all the text and paste it to the editor.

   YAML
   ```yaml
   # DO NOT COPY THE YAML FROM HERE. USE THE DOWNLOAD LINK ABOVE.
   apiVersion: autoscaling/v2beta2
   kind: HorizontalPodAutoscaler
   metadata:
     name: ratings-api
   spec:
     scaleTargetRef:
       apiVersion: apps/v1
       kind: Deployment
       name: ratings-api
     minReplicas: 1
     maxReplicas: 10
     metrics:
     - type: Resource
       resource:
         name: cpu
         target:
           type: Utilization
           averageUtilization: 30
   ```

3. Review the file, and note the following points:
    - **Scale target**

      The target for scaling is the **ratings-api** deployment.
    - **Min and max replicas**

      The minimum and maximum number of replicas to be deployed.
    - **Metrics**

      The autoscaling metric monitored is the CPU utilization, set at 30%. When the utilization goes above that level, the HPA creates more replicas.
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. Deploy the HPA object in the `ratingsapp` namespace.

```
Bash

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

You'll see an output similar to this example.

```
Output

horizontalpodautoscaler.autoscaling/ratings-api created
```

> ⓘ Important
>
> For the horizontal pod autoscaler to work, you must remove any explicit replica count from your ratings-api deployment. Keep in mind that you need to redeploy your deployment when you make any changes.

Run a load test with horizontal pod autoscaler enabled

To create the load test, you can use a prebuilt image called `azch/artillery` that's available on Docker hub. The image contains a tool called artillery that's used to send traffic to the API. Azure Container Instances can be used to run this image as a container.

When it runs as a container instance set, you don't want it to restart after it has finished. Use the `--restart-policy` parameter and set the value to `Never` to prevent the restart.

1. In Azure Cloud Shell, store the front-end API load test endpoint in a Bash variable and replace `<frontend hostname>` with your exposed ingress host name, for example, https://frontend.13-68-177-68.nip.io.

   Bash

   ```
   LOADTEST_API_ENDPOINT=https://<frontend hostname>/api/loadtest
   ```

   Let's run a load test to see how the HPA scales your deployment.

2. Run the load test by using the following command, which sets the duration of the test to 120 seconds to simulate up to 500 requests per second.

   Bash

   ```
   az container create \
       -g $RESOURCE_GROUP \
       -n loadtest \
       --cpu 4 \
       --memory 1 \
       --image azch/artillery \
       --restart-policy Never \
       --command-line "artillery quick -r 500 -d 120 $LOADTEST_API_ENDPOINT"
   ```

   You might need to run this command a few times.

3. Watch the horizontal pod autoscaler working.

   Bash

   ```
   kubectl get hpa \
     --namespace ratingsapp -w
   ```

   In a few seconds, you'll see the HPA transition to deploying more replicas. It scales up from 1 to 10 to accommodate the load. Select `Ctrl+C` to stop watching.

Output

```
NAME          REFERENCE                  TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
ratings-api   Deployment/ratings-api     0%/30%     1         10        1          19m
ratings-api   Deployment/ratings-api     46%/30%    1         10        1          20m
ratings-api   Deployment/ratings-api     46%/30%    1         10        2          20m
ratings-api   Deployment/ratings-api     120%/30%   1         10        2          21m
ratings-api   Deployment/ratings-api     120%/30%   1         10        4          21m
ratings-api   Deployment/ratings-api     93%/30%    1         10        4          22m
ratings-api   Deployment/ratings-api     93%/30%    1         10        8          22m
ratings-api   Deployment/ratings-api     93%/30%    1         10        10         22m
ratings-api   Deployment/ratings-api     0%/30%     1         10        10         23m
```

Autoscale the cluster

HPA scales out with new pods as required. Eventually, the cluster runs out of resources, and you'll see scheduled pods in a pending state.

What is a cluster autoscaler?

The cluster autoscaler watches for pods that can't be scheduled on nodes because of resource constraints. The cluster then automatically increases the number of nodes in the cluster.

Let's introduce load to the cluster to force it to autoscale. We can simulate this by artificially increasing the resource `request` and `limit` for CPU in the `ratings-api` deployment to `cpu: "1000m"` and redeploy. This forces the pods to request more resources across the cluster than is actually available. We can then enable autoscaling, and increase the available nodes that are available to run pods.

1. Edit the file called `ratings-api-deployment.yaml` by using the integrated editor.

   Bash

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

2. Change the `resources.requests` and `resources.limits` for the container to be 1000m, which means one core. The section should now look like this. (Do not replace the whole file, just change the indicated values).

   YAML

   ```yaml
   resources:
     requests: # minimum resources required
       cpu: 1000m
       memory: 64Mi
     limits: # maximum resources allocated
       cpu: 1000m
       memory: 256Mi
   ```

3. Apply the configuration by using the `kubectl apply` command. Deploy the resource update in the `ratingsapp` namespace.

```
Bash

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

You'll seen an output similar to this example.

```
Output

deployment.apps/ratings-api configured
```

4. Review the new pods rolling out. Query for pods in the `ratingsapp` namespace, which are labeled with `app=ratings-api`.

```
Bash

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

You'll now see multiple pods stuck in the `Pending` state because there isn't enough capacity on the cluster to schedule those new pods.

```
Output

NAME                               READY   STATUS    RESTARTS   AGE
ratings-api-7746bb6444-4k24p       0/1     Pending   0          5m42s
ratings-api-7746bb6444-brkd8       0/1     Pending   0          5m42s
ratings-api-7746bb6444-l7fdq       0/1     Pending   0          5m42s
ratings-api-7746bb6444-nfbfd       0/1     Pending   0          5m42s
ratings-api-7746bb6444-rmvb2       0/1     Pending   0          5m42s
ratings-api-7cf598d48-7wmml        1/1     Running   0          35m
ratings-api-7cf598d48-98mwd        1/1     Running   0          12m
ratings-api-7cf598d48-clnbq        1/1     Running   0          11m
ratings-api-7cf598d48-cmhk5        1/1     Running   0          10m
ratings-api-7cf598d48-t6xtk        1/1     Running   0          10m
ratings-api-7cf598d48-vs44s        1/1     Running   0          10m
ratings-api-7cf598d48-xxhxs        1/1     Running   0          11m
ratings-api-7cf598d48-z9klk        1/1     Running   0          10m
ratings-mongodb-5c8f57ff58-k6qcd   1/1     Running   0          16d
ratings-web-7bc649bccb-bwjfc       1/1     Running   0          99m
ratings-web-7bc649bccb-gshn7       1/1     Running   0          99m
```

To solve the pending pod problem, you can enable the cluster autoscaler to scale the cluster automatically.

5. Configure the cluster autoscaler. You should see it dynamically adding and removing nodes based on the cluster utilization. Use the `az aks update` command to enable the cluster autoscaler. Specify a minimum and maximum value for the number of nodes. Make sure to use the same resource group from earlier, for example, **aksworkshop**.

   The following example sets the `--min-count` to *3* and the `--max-count` to *5*.

   Bash

   ```
   az aks update \
   --resource-group $RESOURCE_GROUP \
   --name $AKS_CLUSTER_NAME  \
   --enable-cluster-autoscaler \
   --min-count 3 \
   --max-count 5
   ```

   In a few minutes, the cluster should be configured with the cluster autoscaler. You'll see the number of nodes increase.

6. Verify the number of nodes has increased.

   Bash

   ```
   kubectl get nodes -w
   ```

   In a few minutes, you'll see some new nodes popping up and transitioning to the `Ready` state. Select `Ctrl+C` to stop watching.

   Output

   ```
   NAME                                STATUS   ROLES   AGE   VERSION
   aks-nodepool1-24503160-vmss000000   Ready    agent   50m   v1.15.7
   aks-nodepool1-24503160-vmss000001   Ready    agent   50m   v1.15.7
   aks-nodepool1-24503160-vmss000002   Ready    agent   50m   v1.15.7
   aks-nodepool1-24503160-vmss000003   Ready    agent   14s   v1.15.7
   aks-nodepool1-24503160-vmss000004   Ready    agent   21s   v1.15.
   ```

Summary

In this exercise you created a horizontal pod autoscaler and ran a load test to scale out the pods on your cluster. You then increased the compute capacity of your cluster through the cluster autoscaler, adding nodes to your AKS cluster. You now have the knowledge to ensure the Fruit Smoothies AKS environment can scale in response to fluctuations in user traffic.

Let's next wrap up what you've learned here.

Summary

Summary and cleanup

Summary and cleanup

In this workshop, you deployed a multi-container application to Azure Kubernetes Service (AKS). You used Azure Container Registry to store your container images. You deployed MongoDB with Helm and learned about key Kubernetes concepts to make deployments easier, and support communication between applications and services. You set up TSL/SSL to ensure communication is encrypted, and also set up autoscaling to handle fluctuations in traffic.

You can now use what you learned to deploy container-based applications in your environment to AKS.

Clean up resources

In this module, you created resources by using your Azure subscription. You want to clean up these resources so that there's no continued charge against your account for these resources.

1. Open the Azure portal.

   Azure Portal

2. Select **Resource groups** on the left.
3. Find the **aksworkshop** resource group, or the resource group name you used, and select it.
4. On the **Overview** tab of the resource group, select **Delete resource group**.
5. Enter the name of the resource group to confirm. Select **Delete** to delete all of the resources you created in this module.
6. Finally, run the `kubectl config delete-context` command to remove the deleted clusters context. Here is an example of the complete command. Remember to replace the name of the cluster with your cluster's name.

   Bash

   ```
   kubectl get nodes -w
   ```

   If successful, the command returns the following example output.

   Bash

   ```
   kubectl get nodes -w
   ```

Learn more

We've covered a number of concepts in this document. Visit the articles and sites below to learn more about each of the concepts.

Azure and AKS resources:

- Kubernetes core concepts for AKS
- Network concepts for applications in AKS
- Security concepts for applications and clusters in Azure Kubernetes Service (AKS)
- Azure Monitor for containers overview
- Create a Log Analytics workspace in the Azure portal

Kubernetes and Helm resources:

- Kubernetes documentation
- Kubernetes secrets
- Helm
- How to use Helm
- Helm charts
- GitHub Helm charts repository
- Helm Hub
- MongoDB Helm chart repository

Other utilities and resources:

- Let's Encrypt
- cert-manager
- Wildcard DNS services
  - nip.io
  - xip.io
  - sslip.io

References

- ✓ **Azure Kubernetes Service Workshop**
  https://docs.microsoft.com/en-us/learn/modules/aks-workshop/

Cloud Solution Architects

Customer Success

NorthEast Region

August 2020