14. Serverless functions

Serverless computing and serverless functions have gained tremendous traction over the past few years due to scalability and reduced management overhead. Cloud services such as Azure Functions, AWS Lambda, and GCP Cloud Run have made it very easy for users to run their code as serverless functions.

The word **serverless** refers to any solution where you don't need to manage servers. Serverless functions refer to a subset of serverless computing where you can run your code as a function on-demand. This means that your code in the function will only run and be executed when there is a demand. This architectural style is called event-driven architecture. In an event-driven architecture, the event consumers are triggered when there is an event. In the case of serverless functions, the event consumers will be these serverless functions. An event can be anything from a message in a queue to a new object uploaded to storage, or even an HTTP call.

Serverless functions are frequently used for backend processing. A common example of serverless functions is creating thumbnails of a picture that is uploaded to storage, as shown in *Figure 14.1*. Since you cannot predict how many pictures will be uploaded and when they will be uploaded, it is hard to plan traditional infrastructure and how many servers you should have available for this process. If you implement the creation of that thumbnail as a serverless function, this function will be called on each picture that is uploaded. You don't have to plan the number of functions since each new picture will trigger a new function to be executed.

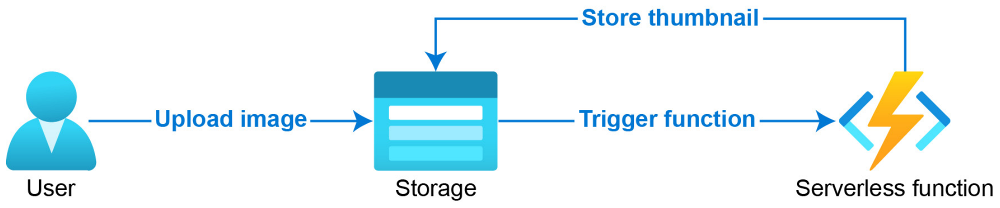


Figure 14.1: Example architecture of a serverless function to generate thumbnails of images

As you saw in the previous example, functions will automatically scale to meet increased or decreased demand. Additionally, each function can scale independently from other functions. However, this automatic scaling is just one benefit of using serverless functions. Another benefit of serverless functions is the ease of development. Using serverless functions, you can focus on writing the code and don't have to deal with the underlying infrastructure. Serverless functions allow code to be deployed without worrying about managing servers and middleware. Finally, in public cloud serverless functions, you pay per execution of the function. This means that you pay each time your functions are run, and you are charged nothing for the idle time when your functions are not run.

The popularity of public cloud serverless function platforms has caused multiple open-source frameworks to be created to enable users to create serverless functions on top of Kubernetes. In this chapter, you will learn how to deploy serverless functions on **Azure Kubernetes Service (AKS)** directly using the open-source version of Azure Functions. You will start by running a simple function that is triggered based on an HTTP message. Afterward, you will install a function **autoscaler** feature on your cluster. You will also integrate AKS-deployed applications with Azure storage queues. We will be covering the following topics:

- Overview of different functions platforms
- Deploying an HTTP-triggered function
- Deploying a queue-triggered function

Let's start this chapter by exploring the various functions platforms that are available for Kubernetes.

Various functions platforms

Functions platforms, such as Azure Functions, AWS Lambda, and Google Cloud Functions, have gained tremendous popularity. The ability to run code without the need to manage servers and having virtually limitless scale is very popular. The downside of using the functions implementation of a cloud provider is that you are locked into the cloud provider's infrastructure and their programming model. Also, you can only run your functions in the public cloud and not in your own datacenter.

A number of open-source functions frameworks have been launched to solve these downsides. There are a number of popular frameworks that can be run on Kubernetes:

- **Knative** (https://cloud.google.com/knative/): Knative is a serverless platform written in the Go language and developed by Google. You can run Knative functions either fully managed on Google Cloud or on your own Kubernetes cluster.
- **OpenFaaS** (https://www.openfaas.com/): OpenFaaS is a serverless framework that is Kubernetes-native. It can run on either managed Kubernetes environments such as AKS or on a self-hosted cluster. OpenFaaS is also available as a managed cloud service using `OpenFaaSCloud`. The platform is written in the Go language.
- **Serverless** (https://serverless.com/): This is a Node.js-based serverless application framework that can deploy and manage functions on multiple cloud providers, including Azure. Kubernetes support is provided via `Kubeless`.
- **Fission.io** (https://fission.io/): Fission is a serverless framework backed by the company Platform9. It is written in the Go language and is Kubernetes-native. It can run on any Kubernetes cluster.
- **Apache OpenWhisk** (https://openwhisk.apache.org/): OpenWhisk

is an open-source, distributed serverless platform maintained by the Apache organization. It can be run on Kubernetes, Mesos, or Docker Compose. It is primarily written in the Scala language.

Microsoft has taken an interesting strategy with its functions platform. Microsoft operates Azure Functions as a managed service on Azure and has open-sourced the complete solution and made it available to run on any system (https://github.com/Azure/azure-functions-host). This also makes the Azure Functions programming model available on top of Kubernetes.

Microsoft has also released an additional open-source project in partnership with Red Hat called **Kubernetes Event-driven Autoscaling** (**KEDA**) to make scaling functions on top of Kubernetes easier. KEDA is a custom autoscaler that can allow deployments on Kubernetes to scale down to and up from zero pods, which is not possible using the default **Horizontal Pod Autoscaler** (**HPA**) in Kubernetes. The ability to scale from zero to one pod is important so that your application can start processing events, but scaling down to zero instances is useful for preserving resources in your cluster. KEDA also makes additional metrics available to the Kubernetes HPA to make scaling decisions based on metrics from outside the cluster (for example, the number of messages in a queue).

Note

We introduced and explained the HPA in Chapter 4, Building scalable applications.

In this chapter, you will deploy Azure Functions to Kubernetes with two examples:

- An HTTP-triggered function (without KEDA)
- A queue-triggered function (with KEDA)

Before starting with these functions, the next section will consider the necessary prerequisites for these deployments.

Setting up the prerequisites

In this section, you will set up the prerequisites needed to build and run functions on your Kubernetes cluster. You need to set up an **Azure container registry** (**ACR**) and a **virtual machine** (**VM**) in Azure that will be used to develop the functions. The ACR will be used to store custom container images that contain the functions you will develop. You will also use a VM to build the functions and create Docker images, since you cannot do this from Azure Cloud Shell.

Container images and a container registry were introduced in *Chapter 1*, *Introduction to containers and Kubernetes*, in the section on *Container images*. A container image contains all the software required to start an actual running container. In this chapter, you will build custom container images that contain your functions. You need a place to store these images so that Kubernetes can pull them and run the containers at scale. You will use ACR for this. ACR is a private container registry that is fully managed by Azure.

Up to now in this book, you have run all the examples on Azure Cloud Shell. For the example in this chapter, you will need a separate VM because Azure Cloud Shell doesn't allow you to build container images. You will create a new VM in Azure to do these tasks.

Let's begin by creating an ACR.

Azure Container Registry

Azure Functions on Kubernetes needs an image registry to store its container images. In this section, you will create an ACR and configure your Kubernetes cluster to have access to this cluster:

1. In the Azure search bar, search for **`container registry`** and click on Container registries, as shown in *Figure 14.2*:

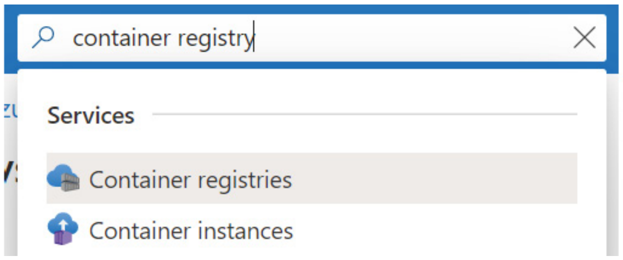
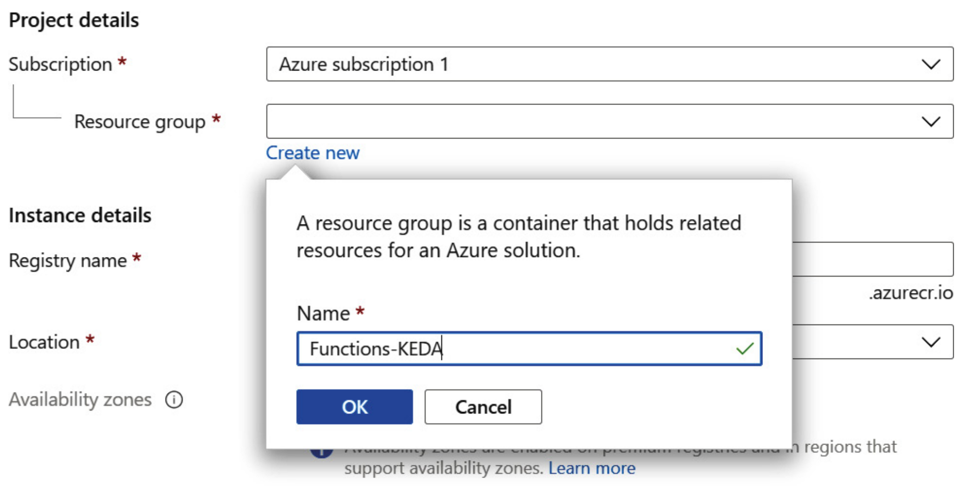
   

   Figure 14.2: Navigating to Container registry services through the Azure portal

2. Click the Add button at the top to create a new registry. To organize the resources in this chapter together, create a new resource group. To do this, click on Create new under the Resource group field to create a new resource group, and call it **`Functions-KEDA`**, as shown in *Figure 14.3*:


Project details

Subscription * Azure subscription 1

Resource group *

Create new

A resource group is a container that holds related resources for an Azure solution.

Name * Functions-KEDA

Instance details

Registry name * .azurecr.io

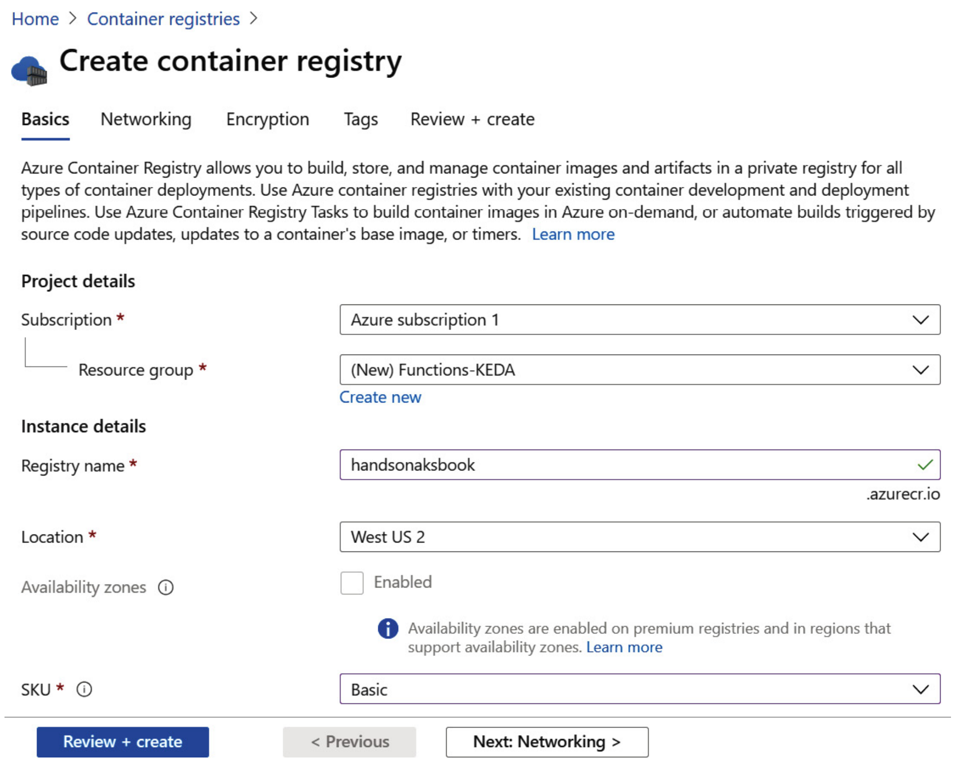
Location *

Availability zones

Availability zones are enabled on premium registries and in regions that support availability zones. Learn more

Figure 14.3: Creating a new resource group

Provide the details to create the registry. The registry name needs to be globally unique, so consider adding your initials to the registry name. It is recommended to create the registry in the same location as your cluster. To reduce spending for the demo, you can change SKU to Basic. Select the Review + create button at the bottom to create the registry, as shown in *Figure 14.4*:


Home > Container registries >

Create container registry

Basics | Networking | Encryption | Tags | Review + create

Azure Container Registry allows you to build, store, and manage container images and artifacts in a private registry for all types of container deployments. Use Azure container registries with your existing container development and deployment pipelines. Use Azure Container Registry Tasks to build container images in Azure on-demand, or automate builds triggered by source code updates, updates to a container's base image, or timers. Learn more

Project details

Subscription * : Azure subscription 1

Resource group * : (New) Functions-KEDA

Create new

Instance details

Registry name * : handsonaksbook .azurecr.io

Location * : West US 2

Availability zones : ☐ Enabled

Availability zones are enabled on premium registries and in regions that support availability zones. Learn more

SKU * : Basic

Review + create | < Previous | Next: Networking >

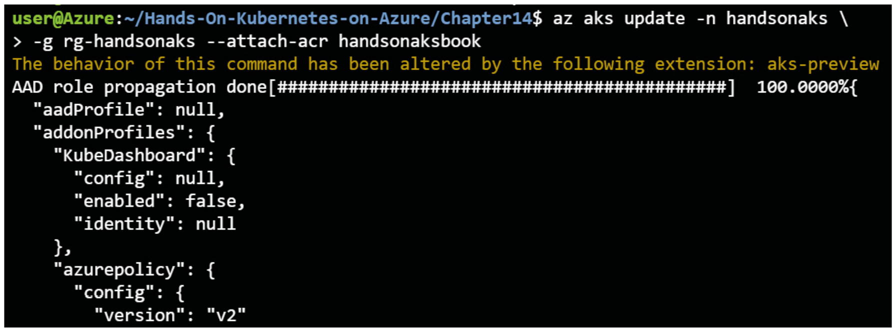
Figure 14.4: Providing details to create the registry

In the resulting pane, click the Create button to create the registry.

3. Once your registry is created, open Cloud Shell so that you can configure your AKS cluster to get access to your container registry. Use the following command to give AKS permissions to your registry:

```
az aks update -n handsonaks \
-g rg-handsonaks --attach-acr <acrName>
```

This will return an output similar to *Figure 14.5*. The figure has been cropped to show only the top part of the output:



```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter14$ az aks update -n handsonaks \
> -g rg-handsonaks --attach-acr handsonaksbook
The behavior of this command has been altered by the following extension: aks-preview
AAD role propagation done[############################################]  100.0000%{
  "aadProfile": null,
  "addonProfiles": {
    "KubeDashboard": {
      "config": null,
      "enabled": false,
      "identity": null
    },
    "azurepolicy": {
      "config": {
        "version": "v2"
```

Figure 14.5: Allowing AKS cluster to access the container registry

You now have an ACR that is integrated with AKS. In the next section, you will create a VM that will be used to build the Azure functions.

Creating a VM

In this section, you will create a VM and install the tools necessary to run Azure Functions on this machine:

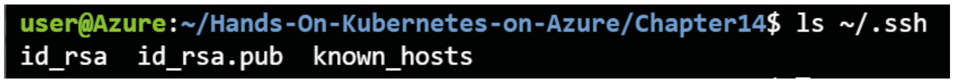
- The Docker runtime
- The Azure CLI
- Azure Functions
- Kubectl

  Note

  To ensure a consistent experience, you will be creating a VM on Azure that will be used for development. If you prefer to run the sample on *your local machine, you can install all the required tools locally.*

Let's get started with creating the VM:

1. To ensure this example works with the Azure trial subscription, you will need to scale down your cluster to one node. You can do this using the following command:
   az aks scale -n handsonaks -g rg-handsonaks --node-count 1
2. To authenticate to the VM you are going to create, you'll need a set of SSH keys. If you followed the example in *Chapter 9*, *Azure Active Directory pod-managed identities in AKS* in the *Setting up a new cluster with AAD pod-managed identity* section, you will already have a set of SSH keys. To verify that you have SSH keys, run the following command:
   ls ~/.ssh
   This should show you the presence of an SSH private key (**`id_rsa`**) and a public key (**`id_rsa.pub`**), as shown in *Figure 14.6*:



```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter14$ ls ~/.ssh
id_rsa  id_rsa.pub  known_hosts
```

Figure 14.6: Verifying SSH keys are present

If you do not have these keys already available, you will need to generate a set of SSH keys using the following command:

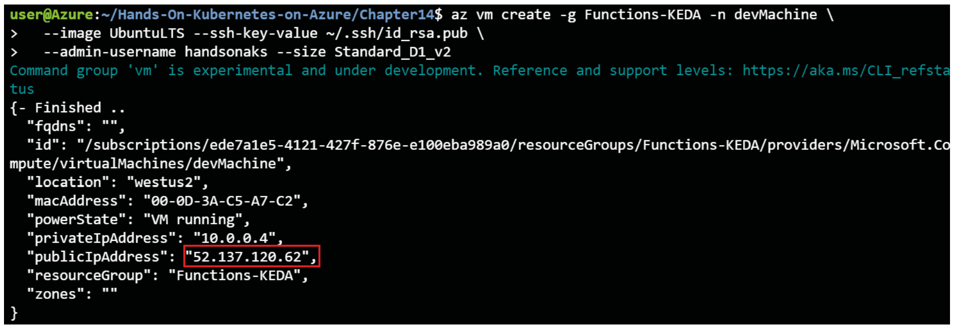
```
ssh-keygen
```

You will be prompted for a location and a passphrase. Keep the default location and input an empty passphrase.

3. You will now create the VM. You will create an Ubuntu VM using the following command:

```
az vm create -g Functions-KEDA -n devMachine \
  --image UbuntuLTS --ssh-key-value ~/.ssh/id_rsa.pub \
  --admin-username handsonaks --size Standard_D1_v2
```

4. This will take a couple of minutes to complete. Once the VM is created, Cloud Shell should show you its public IP, as displayed in *Figure 14.7*:



```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter14$ az vm create -g Functions-KEDA -n devMachine \
>   --image UbuntuLTS --ssh-key-value ~/.ssh/id_rsa.pub \
>   --admin-username handsonaks --size Standard_D1_v2
Command group 'vm' is experimental and under development. Reference and support levels: https://aka.ms/CLI_refsta
tus
{- Finished ..
  "fqdns": "",
  "id": "/subscriptions/ede7a1e5-4121-427f-876e-e100eba989a0/resourceGroups/Functions-KEDA/providers/Microsoft.Co
mpute/virtualMachines/devMachine",
  "location": "westus2",
  "macAddress": "00-0D-3A-C5-A7-C2",
  "powerState": "VM running",
  "privateIpAddress": "10.0.0.4",
  "publicIpAddress": "52.137.120.62",
  "resourceGroup": "Functions-KEDA",
  "zones": ""
}
```

Figure 14.7: Creating the development VM

Connect to the VM using the following command:

```
ssh handsonaks@<public IP>
```

You will be prompted about whether you trust the machine's identity. Type `yes` to confirm.

5. You're now connected to a new VM on Azure. On this machine, we will begin by installing Docker:

   ```
   sudo apt-get update
   sudo apt-get install docker.io -y
   sudo systemctl enable docker
   sudo systemctl start docker
   ```

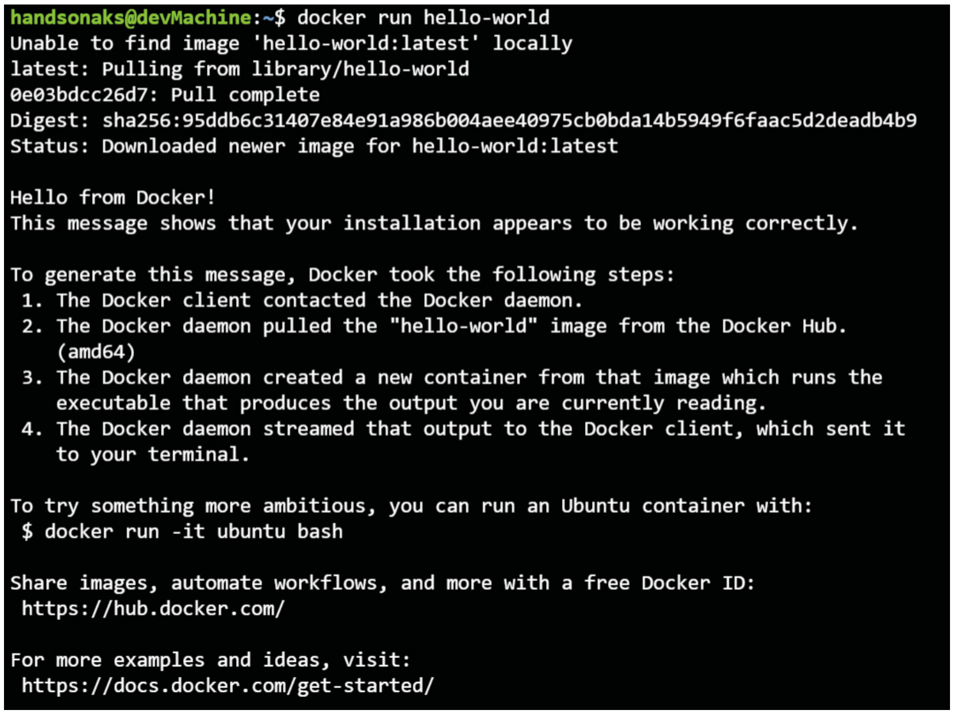
6. To make the operation smoother, add the user to the Docker group. This will ensure you can run Docker commands without `sudo`:

   ```
   sudo usermod -aG docker handsonaks
   newgrp docker
   ```

   You should now be able to run the `hello-world` command:

   ```
   docker run hello-world
   ```

   This will show you an output similar to *Figure 14.8*:



```
handsonaks@devMachine:~$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
0e03bdcc26d7: Pull complete
Digest: sha256:95ddb6c31407e84e91a986b004aee40975cb0bda14b5949f6faac5d2deadb4b9
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

Figure 14.8: Verifying Docker runs on the VM

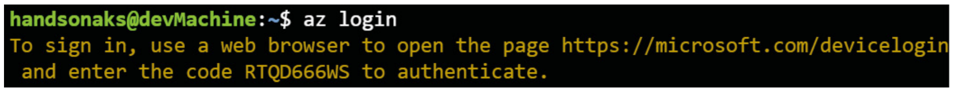
7. Next, you will install the Azure CLI on this VM. You can install the CLI using the following command:

   curl -sL https://aka.ms/InstallAzureCLIDeb | sudo bash

8. Verify that the CLI was installed successfully by signing in:

   az login

   This will display a login code that you need to enter at https://microsoft.com/devicelogin:



```
handsonaks@devMachine:~$ az login
To sign in, use a web browser to open the page https://microsoft.com/devicelogin
 and enter the code RTQD666WS to authenticate.
```

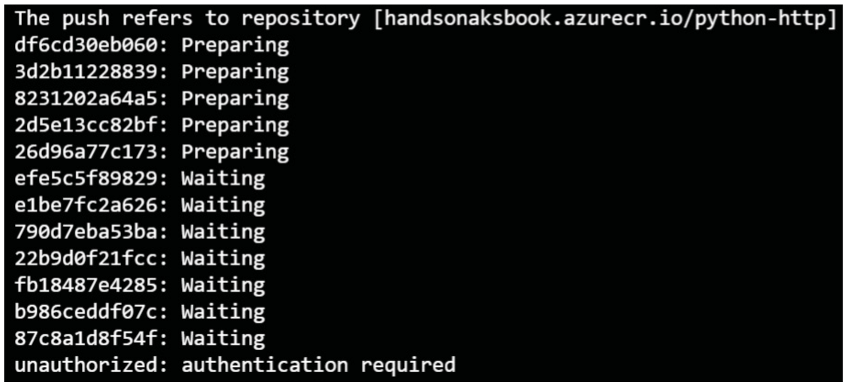
Figure 14.9: Signing in to the Azure CLI

Browse to that website and paste in the login code that was provided to you to enable you to sign in to Cloud Shell. Make sure to do this in a browser you are signed in to with the user who has access to your Azure subscription.

You can now use the CLI to authenticate your machine to ACR. This can be done using the following command:

az acr login -n <registryname>

The credentials to ACR expire after 3 hours. If you run into the following error during this demonstration, you can sign in to ACR again using the following command:



```
The push refers to repository [handsonaksbook.azurecr.io/python-http]
df6cd30eb060: Preparing
3d2b11228839: Preparing
8231202a64a5: Preparing
2d5e13cc82bf: Preparing
26d96a77c173: Preparing
efe5c5f89829: Waiting
e1be7fc2a626: Waiting
790d7eba53ba: Waiting
22b9d0f21fcc: Waiting
fb18487e4285: Waiting
b986ceddf07c: Waiting
87c8a1d8f54f: Waiting
unauthorized: authentication required
```

Figure 14.10: Potential authentication error in the future

9. Next, you'll install **`kubectl`** on your machine. The Azure CLI has a shortcut to install the CLI, which you can use to install it:

   sudo az aks install-cli

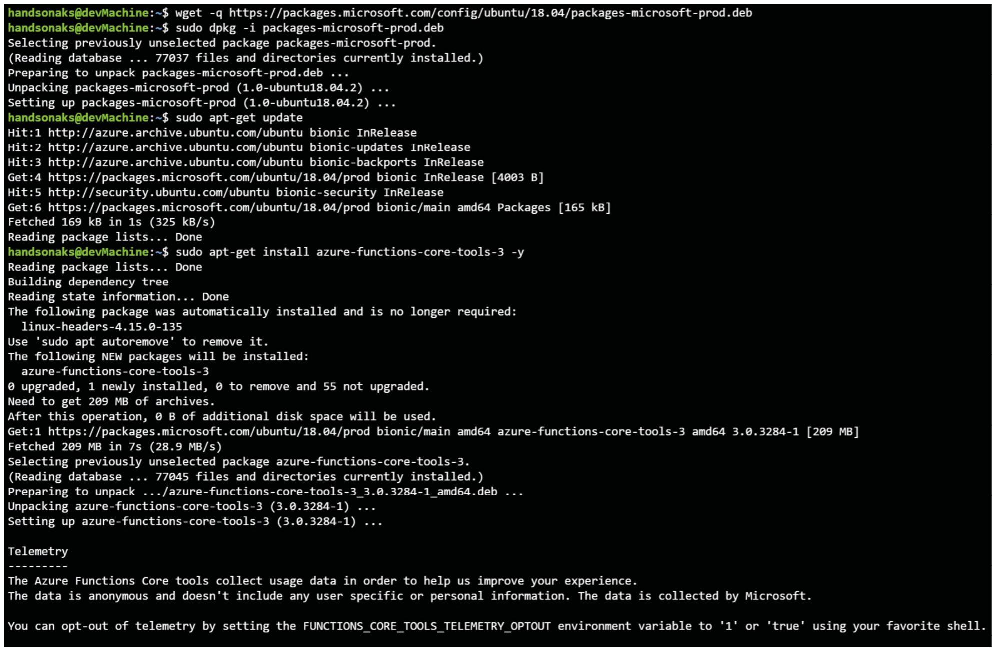
Let's verify that **`kubectl`** can connect to our cluster. For this, we'll first get the credentials and then execute a **`kubectl`** command:

```
az aks get-credentials -n handsonaks -g rg-
handsonaks
kubectl get nodes
```

10. Now, you can install the Azure Functions tools on this machine. To do this, run the following commands:

```
wget -q
https://packages.microsoft.com/config/ubuntu/18.04/packages-
microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
sudo apt-get update
sudo apt-get install azure-functions-core-tools-
3 -y
```

This will return an output similar to *Figure 14.11*:



```
handsonaks@devMachine:~$ wget -q https://packages.microsoft.com/config/ubuntu/18.04/packages-microsoft-prod.deb
handsonaks@devMachine:~$ sudo dpkg -i packages-microsoft-prod.deb
Selecting previously unselected package packages-microsoft-prod.
(Reading database ... 77037 files and directories currently installed.)
Preparing to unpack packages-microsoft-prod.deb ...
Unpacking packages-microsoft-prod (1.0-ubuntu18.04.2) ...
Setting up packages-microsoft-prod (1.0-ubuntu18.04.2) ...
handsonaks@devMachine:~$ sudo apt-get update
Hit:1 http://azure.archive.ubuntu.com/ubuntu bionic InRelease
Hit:2 http://azure.archive.ubuntu.com/ubuntu bionic-updates InRelease
Hit:3 http://azure.archive.ubuntu.com/ubuntu bionic-backports InRelease
Get:4 https://packages.microsoft.com/ubuntu/18.04/prod bionic InRelease [4003 B]
Hit:5 http://security.ubuntu.com/ubuntu bionic-security InRelease
Get:6 https://packages.microsoft.com/ubuntu/18.04/prod bionic/main amd64 Packages [165 kB]
Fetched 169 kB in 1s (325 kB/s)
Reading package lists... Done
handsonaks@devMachine:~$ sudo apt-get install azure-functions-core-tools-3 -y
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following package was automatically installed and is no longer required:
  linux-headers-4.15.0-135
Use 'sudo apt autoremove' to remove it.
The following NEW packages will be installed:
  azure-functions-core-tools-3
0 upgraded, 1 newly installed, 0 to remove and 55 not upgraded.
Need to get 209 MB of archives.
After this operation, 0 B of additional disk space will be used.
Get:1 https://packages.microsoft.com/ubuntu/18.04/prod bionic/main amd64 azure-functions-core-tools-3 amd64 3.0.3284-1 [209 MB]
Fetched 209 MB in 7s (28.9 MB/s)
Selecting previously unselected package azure-functions-core-tools-3.
(Reading database ... 77045 files and directories currently installed.)
Preparing to unpack .../azure-functions-core-tools-3_3.0.3284-1_amd64.deb ...
Unpacking azure-functions-core-tools-3 (3.0.3284-1) ...
Setting up azure-functions-core-tools-3 (3.0.3284-1) ...

Telemetry
---------
The Azure Functions Core tools collect usage data in order to help us improve your experience.
The data is anonymous and doesn't include any user specific or personal information. The data is collected by Microsoft.

You can opt-out of telemetry by setting the FUNCTIONS_CORE_TOOLS_TELEMETRY_OPTOUT environment variable to '1' or 'true' using your favorite shell.
```

Figure 14.11: Installing Functions core tools

Note

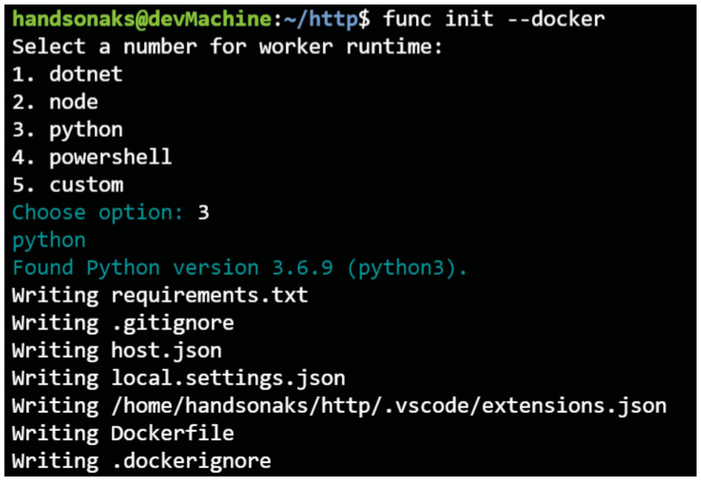
If you are running a newer version of Ubuntu than 18.04, please make sure that you download the correct **dpkg** *package by changing the URL in the first line to reflect your Ubuntu version.*

You now have the prerequisites to start working with functions on Kubernetes. You created an ACR to store custom container images, and you have a VM that will be used to create and build Azure functions. In the next section, you will build your first function, which is HTTP-triggered.

Creating an HTTP-triggered Azure function

In this first example, you will create an HTTP-triggered Azure function. This means that you can browse to the page hosting the actual function:

1. To begin, create a new directory and navigate to that directory:
   mkdir http
   cd http
2. Now, you will initialize a function using the following command:
   func init --docker
   The `--docker` parameter specifies that you will build the function as a Docker container. This will result in a Dockerfile being created. Select the Python language, which is option 3 in the following screenshot:



```
handsonaks@devMachine:~/http$ func init --docker
Select a number for worker runtime:
1. dotnet
2. node
3. python
4. powershell
5. custom
Choose option: 3
python
Found Python version 3.6.9 (python3).
Writing requirements.txt
Writing .gitignore
Writing host.json
Writing local.settings.json
Writing /home/handsonaks/http/.vscode/extensions.json
Writing Dockerfile
Writing .dockerignore
```

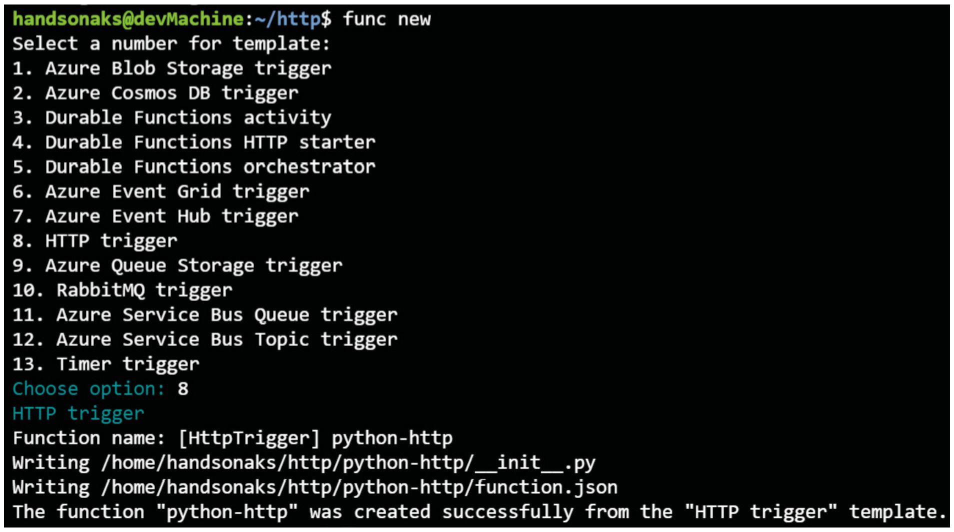
Figure 14.12: Creating a Python function

This will create the required files for your function to work.

3. Next, you will create the actual function. Enter the following command:

   func new

   This should result in an output like the following. Select the eighth option, HTTP trigger, and name the function **`python-http`**:



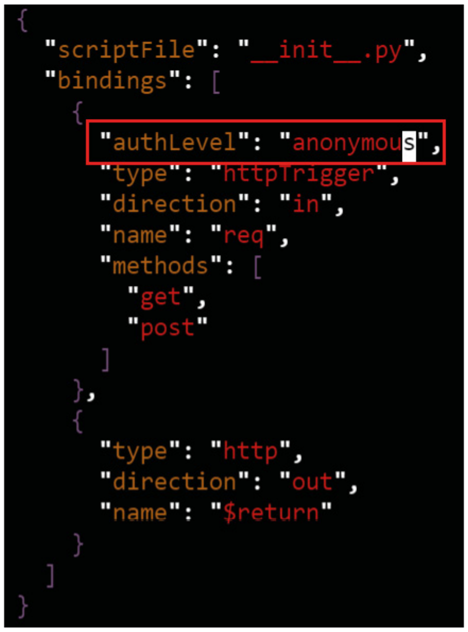
```
handsonaks@devMachine:~/http$ func new
Select a number for template:
1. Azure Blob Storage trigger
2. Azure Cosmos DB trigger
3. Durable Functions activity
4. Durable Functions HTTP starter
5. Durable Functions orchestrator
6. Azure Event Grid trigger
7. Azure Event Hub trigger
8. HTTP trigger
9. Azure Queue Storage trigger
10. RabbitMQ trigger
11. Azure Service Bus Queue trigger
12. Azure Service Bus Topic trigger
13. Timer trigger
Choose option: 8
HTTP trigger
Function name: [HttpTrigger] python-http
Writing /home/handsonaks/http/python-http/__init__.py
Writing /home/handsonaks/http/python-http/function.json
The function "python-http" was created successfully from the "HTTP trigger" template.
```

Figure 14.13: Creating an HTTP-triggered function

4. The code of the function is stored in the directory called **`python-http`**. You are not going to make code changes to this function. If you want to check out the source code of the function, you can run the following command:

   cat python-http/__init__.py

5. You will need to make one change to the function's configuration file. By default, functions require an authenticated request. You will change this to **`anonymous`** for this demo. Make the change using the **`vi`** command by executing the following command:
   vi python-http/function.json
   Replace **`authLevel`** on *line 5* with **`anonymous`**. To make that change, press *I* to go into insert mode, then remove **`function`** and replace it with **`anonymous`**:



```
{
  "scriptFile": "__init__.py",
  "bindings": [
    {
      "authLevel": "anonymous",
      "type": "httpTrigger",
      "direction": "in",
      "name": "req",
      "methods": [
        "get",
        "post"
      ]
    },
    {
      "type": "http",
      "direction": "out",
      "name": "$return"
    }
  ]
}
```

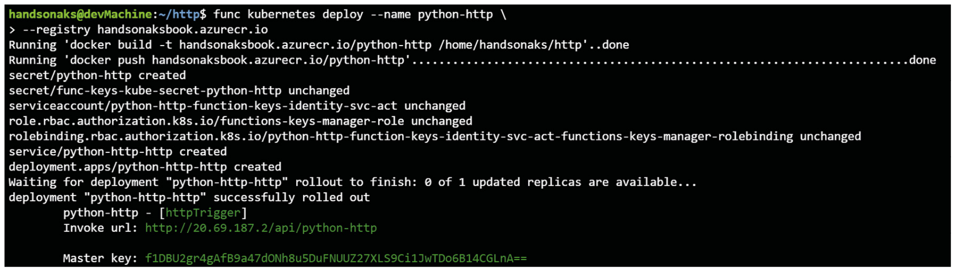
Figure 14.14: Changing the authLevel function to anonymous

Hit *Esc*, type **`:wq!`**, and then hit *Enter* to save and quit **`vi`**.

Note

You changed the authentication requirement for your function to **`anonymous`**. *This will make the demo easier to execute. If you plan to release functions to production, you need to carefully consider this setting, since this controls who has access to your function.*

6. You are now ready to deploy your function to AKS. You can deploy the function using the following command:
   func kubernetes deploy --name python-http \
   --registry <registry name>.azurecr.io
   This will cause the functions runtime to do a couple of steps. First, it will build a container image, then it will push that image to the registry, and finally, it will deploy the function to Kubernetes:



```
handsonaks@devMachine:~/http$ func kubernetes deploy --name python-http \
> --registry handsonaksbook.azurecr.io
Running 'docker build -t handsonaksbook.azurecr.io/python-http /home/handsonaks/http'..done
Running 'docker push handsonaksbook.azurecr.io/python-http'...................................................................done
secret/python-http created
secret/func-keys-kube-secret-python-http unchanged
serviceaccount/python-http-function-keys-identity-svc-act unchanged
role.rbac.authorization.k8s.io/functions-keys-manager-role unchanged
rolebinding.rbac.authorization.k8s.io/python-http-function-keys-identity-svc-act-functions-keys-manager-rolebinding unchanged
service/python-http-http created
deployment.apps/python-http-http created
Waiting for deployment "python-http-http" rollout to finish: 0 of 1 updated replicas are available...
deployment "python-http-http" successfully rolled out
        python-http - [httpTrigger]
        Invoke url: http://20.69.187.2/api/python-http

        Master key: f1DBU2gr4gAfB9a47dONh8u5DuFNUUZ27XLS9Ci1JwTDo6B14CGLnA==
```

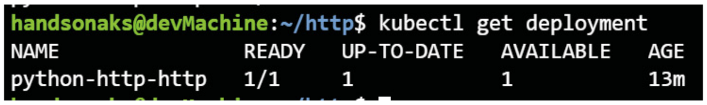
Figure 14.15: Deploying the function to AKS

You can click the Invoke url URL that is shown to get access to your function. Before doing so, however, let's explore what was created on the cluster.

7. To create the function, a regular deployment on top of Kubernetes was used. To check the deployment, you can run the following command:

   kubectl get deployment

   This will show you the deployment, as in *Figure 14.16*:



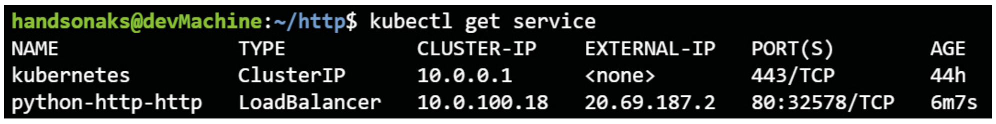
```
handsonaks@devMachine:~/http$ kubectl get deployment
NAME               READY   UP-TO-DATE   AVAILABLE   AGE
python-http-http   1/1     1            1           13m
```

   Figure 14.16: Deployment details

8. This process also created a service on top of your Kubernetes cluster. You can get the public IP of the service that was deployed and connect to it:

   kubectl get service

   This will show you the service and its public IP, as shown in *Figure 14.17*. Notice how this public IP is the same as the one shown in the output of *Step 4*.



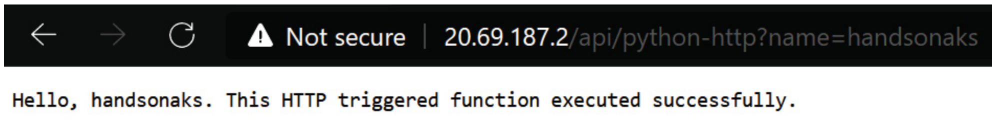
```
handsonaks@devMachine:~/http$ kubectl get service
NAME               TYPE           CLUSTER-IP    EXTERNAL-IP   PORT(S)        AGE
kubernetes         ClusterIP      10.0.0.1      <none>        443/TCP        44h
python-http-http   LoadBalancer   10.0.100.18   20.69.187.2   80:32578/TCP   6m7s
```

Figure 14.17: Getting the service's public IP

Open a web browser and browse to

`http://<external-ip>/api/python-http?`

`name=handsonaks`. You should see a web page showing you Hello, handsonaks. This HTTP triggered function executed successfully. This is shown in *Figure 14.18*:



Not secure | 20.69.187.2/api/python-http?name=handsonaks

```
Hello, handsonaks. This HTTP triggered function executed successfully.
```

Figure 14.18: Output of the HTTP triggered function

You have now created a function with an HTTP trigger. Using an HTTP-triggered function is useful in scenarios where you are providing an HTTP API with unpredictable load patterns. Let's clean up this deployment before moving on to the next section:

kubectl delete deployment python-http-http

kubectl delete service python-http-http

kubectl delete secret python-http

In this section, you created a sample function using an HTTP trigger. Let's take that one step further and integrate a new function with storage queues and set up the KEDA autoscaler in the next section.

Creating a queue-triggered function

In the previous section, you created a sample HTTP function. In this section, you'll build another sample using a queue-triggered function. Queues are often used to pass messages between different components of an application. A function can be triggered based on messages in a queue to then perform additional processing on these messages.

In this section, you'll create a function that is integrated with Azure storage queues to consume events. You will also configure KEDA to allow scaling to/from zero pods in the case of low traffic.

Let's start by creating a queue in Azure.

Creating a queue

In this section, you will create a new storage account and a new queue in that storage account. You will connect functions to that queue in the next section, *Creating a queue-triggered function*.

1. To begin, create a new storage account. Search for **`storage accounts`** in the Azure search bar and select Storage accounts:

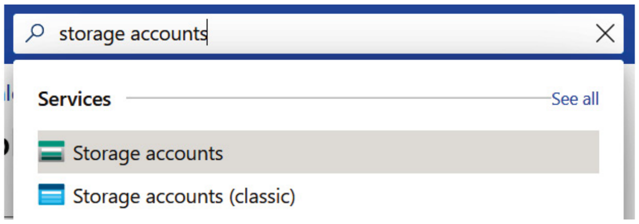
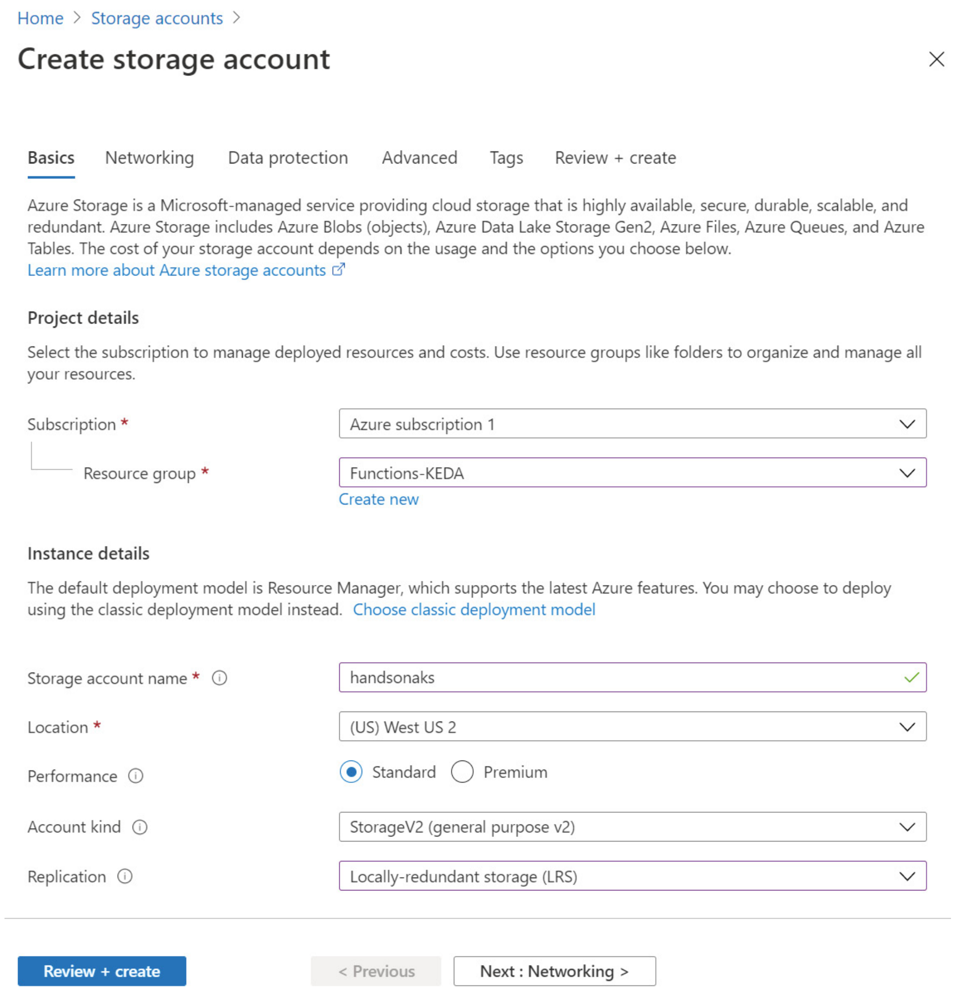


Figure 14.19: Navigating to Storage accounts service through the Azure portal

2. Click the + New button at the top to create a new storage account. Provide the details to create the storage account. The storage account name has to be globally unique, so consider adding your initials. It is recommended to create the storage account in the same region as your AKS cluster. Finally, to save on costs, you are recommended to downgrade the replication setting to Locally-redundant storage (LRS) as shown in *Figure 14.20*:



Azure Storage is a Microsoft-managed service providing cloud storage that is highly available, secure, durable, scalable, and redundant. Azure Storage includes Azure Blobs (objects), Azure Data Lake Storage Gen2, Azure Files, Azure Queues, and Azure Tables. The cost of your storage account depends on the usage and the options you choose below.
Learn more about Azure storage accounts

Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription *: Azure subscription 1
Resource group *: Functions-KEDA
Create new

Instance details

The default deployment model is Resource Manager, which supports the latest Azure features. You may choose to deploy using the classic deployment model instead. Choose classic deployment model

Storage account name *: handsonaks
Location *: (US) West US 2
Performance: [✓ Standard] Premium
Account kind: StorageV2 (general purpose v2)
Replication: Locally-redundant storage (LRS)

Figure 14.20: Providing the details to create the storage account

Once you're ready, click the Review + create button at the bottom. On the resulting screen, select Create to start the creation process.

3. It will take about a minute to create the storage account. Once it is created, open the account by clicking on the Go to resource button. In the Storage account pane, select Access keys in the left-hand navigation, click on Show keys, and

copy the primary connection string, as shown in *Figure 14.21*. Note down this string for now:

Figure 14.21: Copying the primary connection string

Note

For production use cases, it is not recommended to connect to Azure Storage using the access key. Any user with that access key has full access to the storage account and can read and delete all files on it. It is recommended to either generate a **shared access signatures** *(***SAS***) token to connect to storage or to use Azure AD-integrated security. To learn more about SAS token authentication to storage, refer to* **https://docs.microsoft.com/rest/api/storageservices/delega access-with-shared-access-signature**. *To learn more about Azure AD authentication to*

Azure Storage, please refer to https://docs.microsoft.com/rest/api/storageservices/authoi with-azure-active-directory.

4. The final step is to create our queue in the storage account. Look for **queue** in the left-hand navigation, click the + Queue button to add a queue, and provide it with a name. To follow along with this demo, use **`function`** as the queue name:

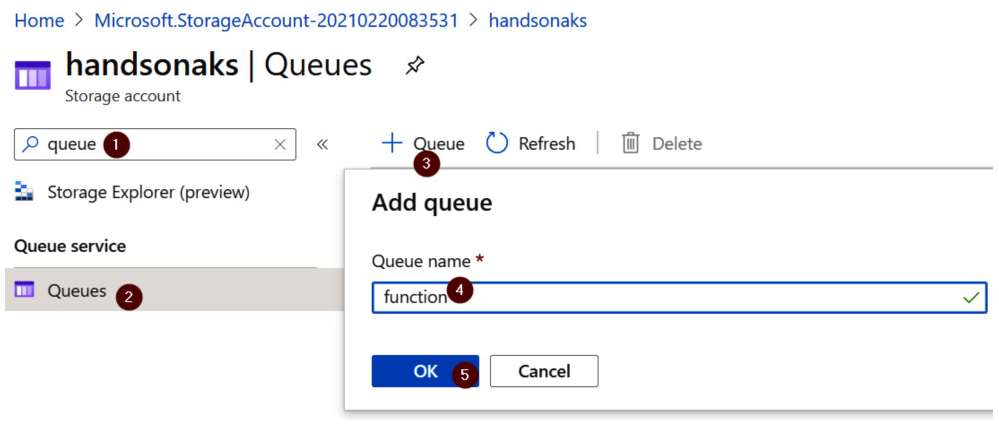


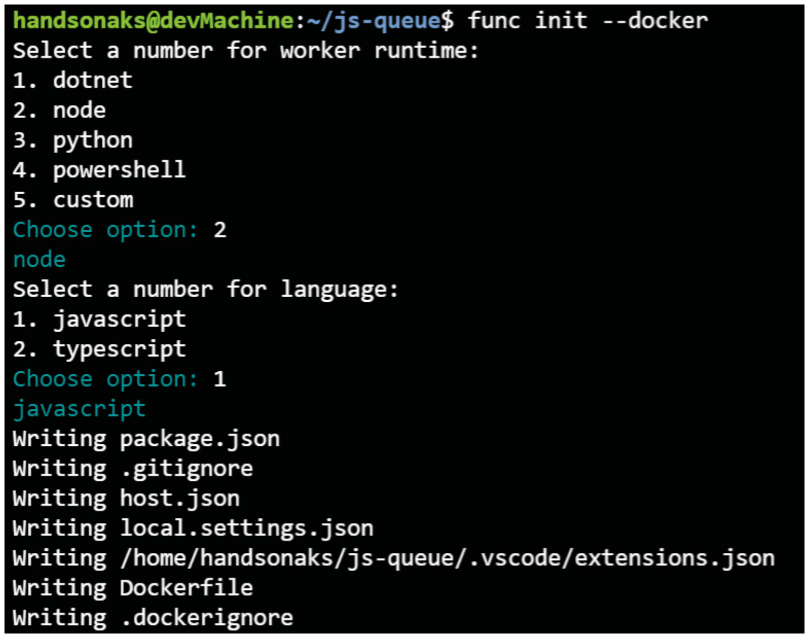
Figure 14.22: Creating a new queue

You have now created a storage account in Azure and have its connection string. You created a queue in this storage account. In the next section, you will create a function that will consume messages from the queue.

Creating a queue-triggered function

In the previous section, you created a queue in Azure. In this section, you will create a new function that will monitor this queue and remove messages from the queue. You will need to configure this function with the connection string to this queue:

1. From within the VM, begin by creating a new directory and navigating to it:
   ```
   cd ..
   mkdir js-queue
   cd js-queue
   ```
2. Now we can create the function. We will start with the initialization:
   ```
   func init --docker
   ```
   This will ask you two questions now. For the runtime, select node (option 2), and for the language, select JavaScript (option 1). This should result in the output shown in *Figure 14.23*:



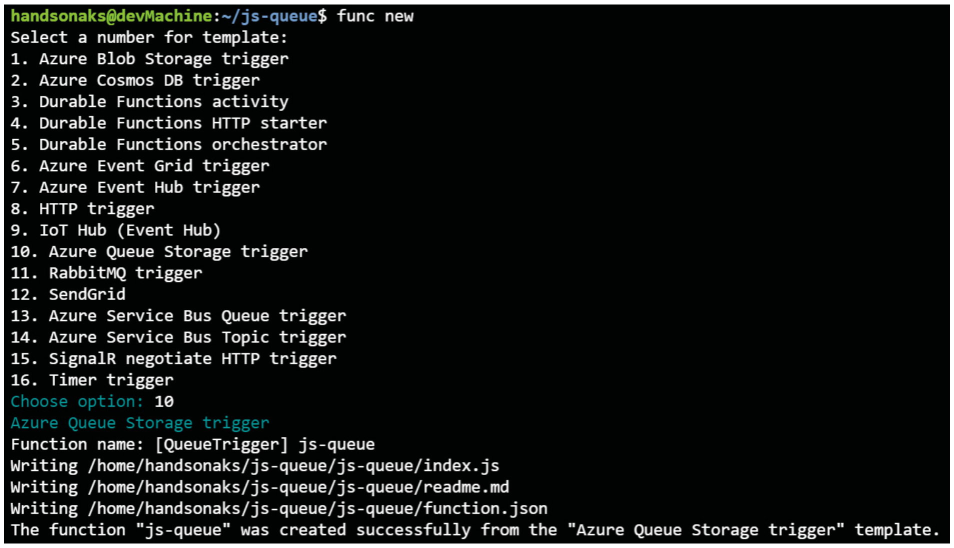
```
handsonaks@devMachine:~/js-queue$ func init --docker
Select a number for worker runtime:
1. dotnet
2. node
3. python
4. powershell
5. custom
Choose option: 2
node
Select a number for language:
1. javascript
2. typescript
Choose option: 1
javascript
Writing package.json
Writing .gitignore
Writing host.json
Writing local.settings.json
Writing /home/handsonaks/js-queue/.vscode/extensions.json
Writing Dockerfile
Writing .dockerignore
```

Figure 14.23: Initializing a new function

Following the initialization, you can create the actual function:

func new

This will ask you for a trigger. Select Azure Queue Storage trigger (option 10). Give the name **`js-queue`** to the new function. This should result in the output shown in *Figure 14.24*:



```
handsonaks@devMachine:~/js-queue$ func new
Select a number for template:
1. Azure Blob Storage trigger
2. Azure Cosmos DB trigger
3. Durable Functions activity
4. Durable Functions HTTP starter
5. Durable Functions orchestrator
6. Azure Event Grid trigger
7. Azure Event Hub trigger
8. HTTP trigger
9. IoT Hub (Event Hub)
10. Azure Queue Storage trigger
11. RabbitMQ trigger
12. SendGrid
13. Azure Service Bus Queue trigger
14. Azure Service Bus Topic trigger
15. SignalR negotiate HTTP trigger
16. Timer trigger
Choose option: 10
Azure Queue Storage trigger
Function name: [QueueTrigger] js-queue
Writing /home/handsonaks/js-queue/js-queue/index.js
Writing /home/handsonaks/js-queue/js-queue/readme.md
Writing /home/handsonaks/js-queue/js-queue/function.json
The function "js-queue" was created successfully from the "Azure Queue Storage trigger" template.
```

Figure 14.24: Creating a queue-triggered function

3. You will now need to make a couple of configuration changes. You need to provide the function you created the connection string on to Azure Storage and provide the queue name. First, open the **`local.settings.json`** file to configure the connection strings for storage:

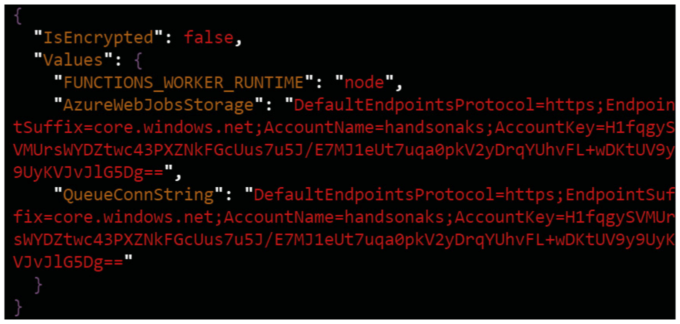
   vi local.settings.json

   To make the changes, follow these instructions:

   1. Hit *I* to go into insert mode.
   2. Replace the connection string for **`AzureWebJobsStorage`** with the connection string you copied earlier. Add a comma to the end of this line.
   3. Add a new line and then add the following text on that line:

   "QueueConnString": "<your connection string>"

   The result should look like *Figure 14.25*:



```
{
  "IsEncrypted": false,
  "Values": {
    "FUNCTIONS_WORKER_RUNTIME": "node",
    "AzureWebJobsStorage": "DefaultEndpointsProtocol=https;Endpoin
tSuffix=core.windows.net;AccountName=handsonaks;AccountKey=H1fqgyS
VMUrsWYDZtwc43PXZNkFGcUus7u5J/E7MJ1eUt7uqa0pkV2yDrqYUhvFL+wDKtUV9y
9UyKVJvJlG5Dg==",
    "QueueConnString": "DefaultEndpointsProtocol=https;EndpointSuf
fix=core.windows.net;AccountName=handsonaks;AccountKey=H1fqgySVMUr
sWYDZtwc43PXZNkFGcUus7u5J/E7MJ1eUt7uqa0pkV2yDrqYUhvFL+wDKtUV9y9UyK
VJvJlG5Dg=="
  }
}
```

Figure 14.25: Editing the local.settings.json file

1. Save and close the file by hitting the *Esc* key, type **`:wq!`**, and then press *Enter*.

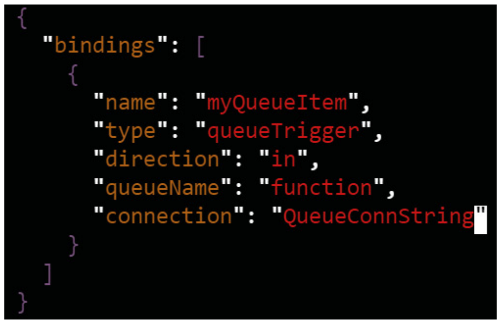
4. The next file you need to edit is the function configuration itself. Here, you will refer to the connection string from earlier, and provide the queue name we chose in the *Creating a queue* section. To do that, use the following command:

   vi js-queue/function.json

   To make the changes, follow these instructions:

   1. Hit *I* to go into insert mode.
   2. Change the queue name to the name of the queue you created (**`function`**).
   3. Next, add **`QueueConnString`** to the **`connection`** field.

   Your configuration should now look like *Figure 14.26*:



```
{
  "bindings": [
    {
      "name": "myQueueItem",
      "type": "queueTrigger",
      "direction": "in",
      "queueName": "function",
      "connection": "QueueConnString"
    }
  ]
}
```

Figure 14.26: Editing the js-queue/function.json file

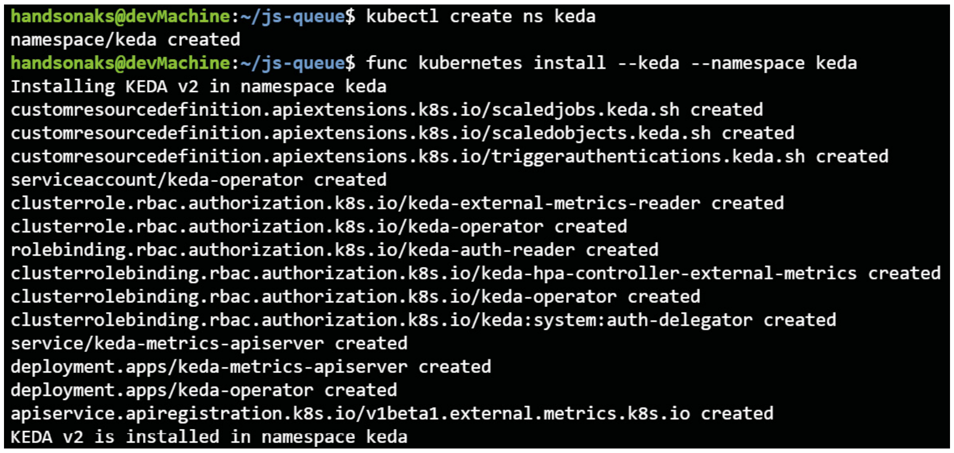
1. Save and close the file by hitting the *Esc* key, type **`:wq!`**, and then press *Enter*.

5. You are now ready to publish your function to Kubernetes. You will start by setting up KEDA on your Kubernetes cluster:

   kubectl create ns keda

   func kubernetes install --keda --namespace keda

   This should return an output similar to *Figure 14.27*:



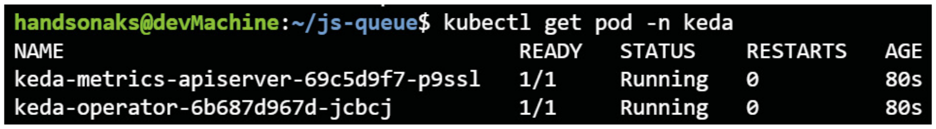
```
handsonaks@devMachine:~/js-queue$ kubectl create ns keda
namespace/keda created
handsonaks@devMachine:~/js-queue$ func kubernetes install --keda --namespace keda
Installing KEDA v2 in namespace keda
customresourcedefinition.apiextensions.k8s.io/scaledjobs.keda.sh created
customresourcedefinition.apiextensions.k8s.io/scaledobjects.keda.sh created
customresourcedefinition.apiextensions.k8s.io/triggerauthentications.keda.sh created
serviceaccount/keda-operator created
clusterrole.rbac.authorization.k8s.io/keda-external-metrics-reader created
clusterrole.rbac.authorization.k8s.io/keda-operator created
rolebinding.rbac.authorization.k8s.io/keda-auth-reader created
clusterrolebinding.rbac.authorization.k8s.io/keda-hpa-controller-external-metrics created
clusterrolebinding.rbac.authorization.k8s.io/keda-operator created
clusterrolebinding.rbac.authorization.k8s.io/keda:system:auth-delegator created
service/keda-metrics-apiserver created
deployment.apps/keda-metrics-apiserver created
deployment.apps/keda-operator created
apiservice.apiregistration.k8s.io/v1beta1.external.metrics.k8s.io created
KEDA v2 is installed in namespace keda
```

Figure 14.27: Installing KEDA on Kubernetes

This will set up KEDA on your cluster. The installation doesn't take long. To verify that the installation was successful, make sure that the KEDA pod is running in the **`keda`** namespace:

kubectl get pod -n keda

This should return an output similar to *Figure 14.28*:



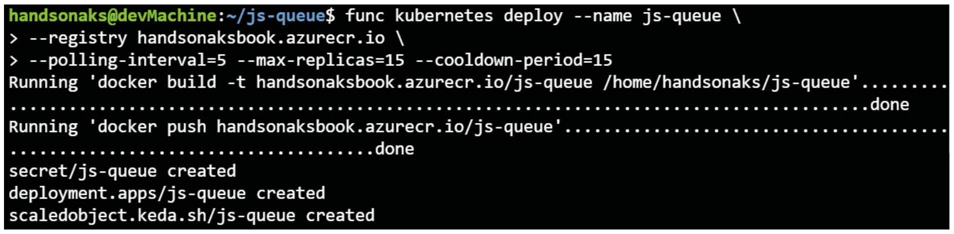
```
handsonaks@devMachine:~/js-queue$ kubectl get pod -n keda
NAME                                    READY   STATUS    RESTARTS   AGE
keda-metrics-apiserver-69c5d9f7-p9ssl   1/1     Running   0          80s
keda-operator-6b687d967d-jcbcj          1/1     Running   0          80s
```

Figure 14.28: Verifying the KEDA installation succeeded

6. You can now deploy the function to Kubernetes. You will configure KEDA to look at the number of queue messages every 5 seconds (**`polling-interval=5`**) to have a maximum of 15 replicas (**`max-replicas=15`**), and to wait 15 seconds before removing pods (**`cooldown-period=15`**). To deploy and configure KEDA in this way, use the following command:

   func kubernetes deploy --name js-queue \
   --registry <registry name>.azurecr.io \
   --polling-interval=5 --max-replicas=15 --
   cooldown-period=15

   This will return an output similar to *Figure 14.29*:



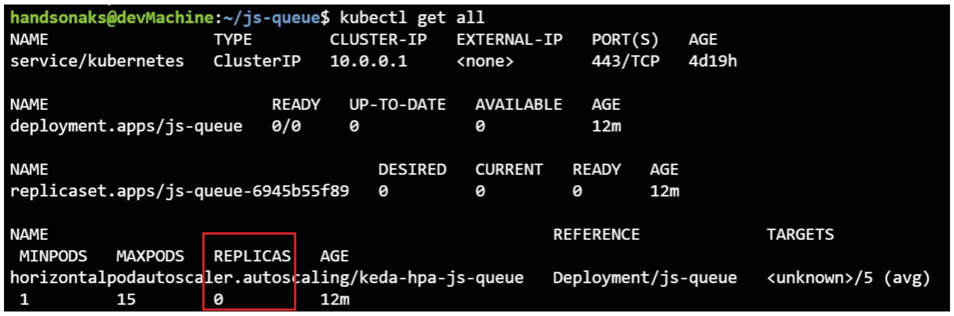
```
handsonaks@devMachine:~/js-queue$ func kubernetes deploy --name js-queue \
> --registry handsonaksbook.azurecr.io \
> --polling-interval=5 --max-replicas=15 --cooldown-period=15
Running 'docker build -t handsonaksbook.azurecr.io/js-queue /home/handsonaks/js-queue'........
...............................................................................done
Running 'docker push handsonaksbook.azurecr.io/js-queue'.....................................
....................................done
secret/js-queue created
deployment.apps/js-queue created
scaledobject.keda.sh/js-queue created
```

Figure 14.29: Deploying the queue-triggered function

To verify that the setup completed successfully, you can run the following command:

```
kubectl get all
```

This will show you all the resources that were deployed. As you can see in *Figure 14.30*, this setup created a deployment, ReplicaSet, and an HPA. In the HPA, you should see that there are no replicas currently running:



```
handsonaks@devMachine:~/js-queue$ kubectl get all
NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   10.0.0.1     <none>        443/TCP   4d19h

NAME                       READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/js-queue   0/0     0            0           12m

NAME                                  DESIRED   CURRENT   READY   AGE
replicaset.apps/js-queue-6945b55f89   0         0         0       12m

NAME                                                       REFERENCE             TARGETS
 MINPODS   MAXPODS   REPLICAS   AGE
horizontalpodautoscaler.autoscaling/keda-hpa-js-queue   Deployment/js-queue   <unknown>/5 (avg)
 1         15        0          12m
```

Figure 14.30: Verifying the objects created by the setup

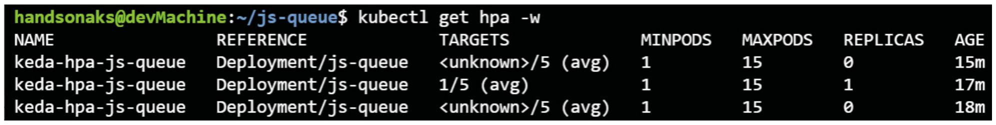
7. Now you will create a message in the queue to trigger KEDA and create a pod. To see the scaling event, run the following command:

   ```
   kubectl get hpa -w
   ```

8. To create a message in the queue, we are going to use the Azure portal. To create a new message, open the queue in the storage that you created earlier. Click on the + Add message button at the top of your screen, create a test message, and click on OK. This is shown in *Figure 14.31*:

Figure 14.31: Adding a message to the queue

After creating this message, have a look at the output of the previous command you issued. It might take a couple of seconds, but soon enough, your HPA should scale to one replica. Afterward, it should also scale back down to zero replicas:



```
handsonaks@devMachine:~/js-queue$ kubectl get hpa -w
NAME                REFERENCE             TARGETS              MINPODS   MAXPODS   REPLICAS   AGE
keda-hpa-js-queue   Deployment/js-queue   <unknown>/5 (avg)    1         15        0          15m
keda-hpa-js-queue   Deployment/js-queue   1/5 (avg)            1         15        1          17m
keda-hpa-js-queue   Deployment/js-queue   <unknown>/5 (avg)    1         15        0          18m
```

Figure 14.32: KEDA scaling from 0 to 1 and back to 0 replicas

This has shown you that KEDA enabled the Kubernetes HPA to scale from zero to one pod when there are messages in the queue, and also from one to zero pods when those messages are processed.

You have now created a function that is triggered by messages being added to a queue. You were able to verify that KEDA scaled the pods from 0 to 1 when you created a message in the queue, and back down to 0 when there were no messages left. In the next section, you will execute a scale test, and you will create multiple messages in the queue and see how the functions react.

Scale testing functions

In the previous section, you saw how functions reacted when there was a single message in the queue. In this example, you are going to send 1,000 messages into the queue and see how KEDA will first scale out the function, and then scale back in, and eventually scale back down to zero:

1. In the current Cloud Shell, watch the HPA using the following command:
   kubectl get hpa -w
2. To start pushing the messages, you are going to open a new Cloud Shell session. To open a new session, select the Open new session button in Cloud Shell:

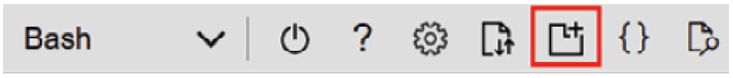


Figure 14.33: Opening a new Cloud Shell instance

To send the 1,000 messages into the queue, a Python script has been provided called **`sendMessages.py`** in *Chapter 15* of the code examples in the GitHub repo accompanying this book. To make the script work, you'll need to install **`azure-storage-queue package using pip`**:

pip install azure-storage-queue==12.1.5

Once that is installed, you will need to provide this script with your storage account connection string. To do this, open the file using:

code sendMessages.py

Edit the storage connection string on *line 8* to your connection string:

Figure 14.34: Pasting in your connection string for your storage account on line 8

3. Once you have pasted in your connection string, you can execute the Python script and send 1,000 messages to your queue:

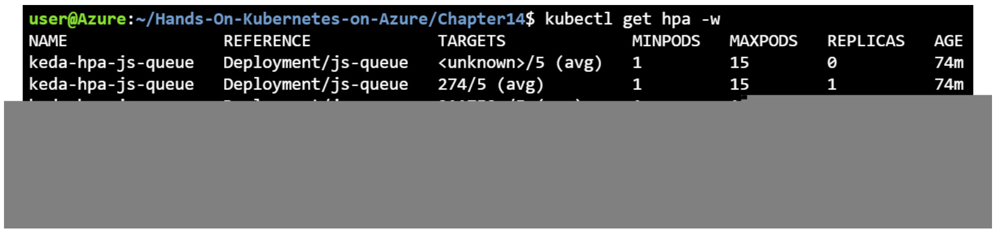
   python sendMessages.py

   While the messages are being sent, switch back to the previous Cloud Shell instance and watch KEDA scale from 0 to 1, and then watch the HPA scale to the number of replicas. The HPA uses metrics provided by KEDA to make scaling decisions. Kubernetes, by default, doesn't know about the number of messages in an Azure storage queue that KEDA provides to the HPA.

   Note

   Depending on how quickly KEDA in your cluster scales up the application, your deployment might not scale to the 15 replicas that are shown in Figure 14.29.

   Once the queue is empty, KEDA will scale back down to zero replicas:



```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter14$ kubectl get hpa -w
NAME                REFERENCE            TARGETS            MINPODS   MAXPODS   REPLICAS   AGE
keda-hpa-js-queue   Deployment/js-queue  <unknown>/5 (avg)  1         15        0          74m
keda-hpa-js-queue   Deployment/js-queue  274/5 (avg)        1         15        1          74m
```

Figure 14.35: KEDA will scale from 0 to 1, and the HPA will scale to 15 pods

As you can see in the output of this command, the deployment was scaled first from zero to one replica, and then gradually got scaled out to a maximum of 15 replicas. When there were no more messages in the queue, the deployment was scaled down again to zero replicas.

This concludes the examples of running serverless functions on top of Kubernetes. Let's make sure to clean up the objects that were created. Run the following command from within the VM you created (the final step will delete this VM; if you want to keep the VM, don't run the final step):

```
kubectl delete secret js-queue

kubectl delete scaledobject js-queue

kubectl delete deployment js-queue

func kubernetes remove --namespace keda

az group delete -n Functions-KEDA  --yes
```

In this section, you ran a function that was triggered by messages in a storage queue on top of Kubernetes. You used a component called KEDA to achieve scaling based on the number of queue messages. You saw how KEDA can scale from 0 to 1 and back down to 0. You also saw how the HPA can use metrics provided by KEDA to scale out a deployment.

Summary

In this chapter, you deployed serverless functions on top of your Kubernetes cluster. To achieve this, you first created a VM and an ACR.

You started the functions deployments by deploying a function that used an HTTP trigger. The Azure Functions core tools were used to create that function and to deploy it to Kubernetes.

Afterward, you installed an additional component on your Kubernetes cluster called KEDA. KEDA allows serverless scaling in Kubernetes. It allows deployments to and from zero pods, and it also provides additional metrics to the HPA. You used a function that was triggered on messages in an Azure storage queue.

In the next – and final – chapter of this book, you'll learn how to integrate containers and Kubernetes in a **continuous integration and continuous delivery** (**CI/CD**) pipeline using GitHub Actions.