



Using Payara Micro With Kubernetes on Azure AKS

The Payara® Platform - Production-Ready,
Cloud Native and Aggressively Compatible.

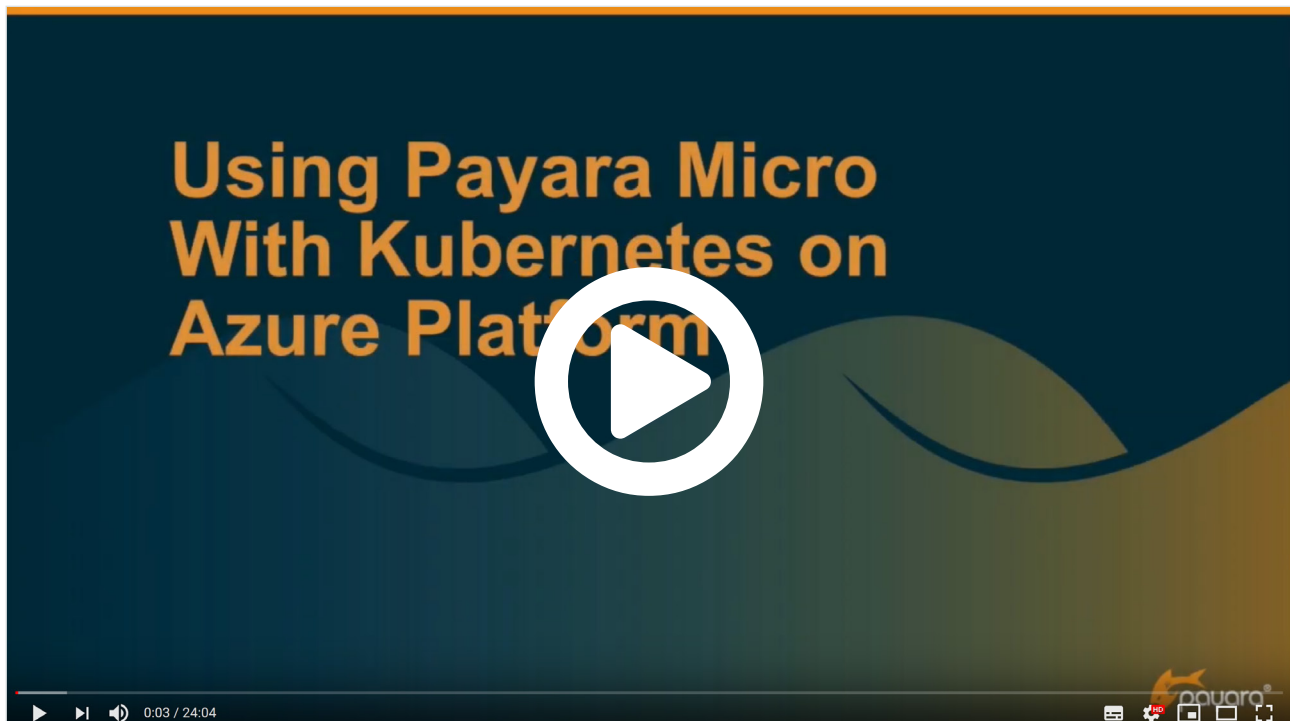
Contents

Microsoft Azure Services Related to Kubernetes	2
Microsoft Azure Kubernetes Service	2
Azure Container Registry	2
Understanding the Concepts of Microsoft Azure	3
Requirements for Following this Guide	3
Set up a Kubernetes Cluster	4
Create a Kubernetes Cluster	4
Manage Kubernetes Cluster	6
Access Kubernetes Dashboard	7
Deploy Applications to a Kubernetes Cluster	8
Sample Payara Micro Application	8
Create Docker Image	9
Push an Image to a Docker Registry	10
Attach a Private ACR Registry to a Kubernetes Cluster	13
Deploy to a Kubernetes cluster	13
Verify Application Deployment	18
Advanced Operations on the Kubernetes Cluster	19
Payara Micro is a Perfect Fit for Kubernetes and Azure	19

Kubernetes® has become the de-facto solution for container orchestration in the cloud. Kubernetes is a complex tool designed for operating hybrid platforms. If you intend to deploy a Payara® Micro Kubernetes cluster on the Microsoft® Azure cloud, you have to follow specific instructions for how to set up a Kubernetes cluster in Azure before you can use the cluster. Azure has a specific implementation mechanism for provisioning new clusters, which integrates well into the Azure tooling and infrastructure. The purpose of this guide is to showcase how to create a new Kubernetes cluster in Microsoft Azure and to set up a deployment using a sample WAR application running on Payara Micro. The contents of this guide will cover:

- What is Azure Kubernetes Service?
- What is Azure Container Registry?
- Understanding the concepts of Microsoft Azure
- Requirements for how to set up your environment
- Creating the Kubernetes cluster
- Creating a private Docker® registry in Azure linked to a Kubernetes cluster
- The description of the Payara Micro sample application
- How to prepare the Docker image to deploy in the cluster
- Provision your Kubernetes cluster with a new Deployment
- Testing the application in the cluster

[Video Guide](#)



Microsoft Azure Services Related to Kubernetes

Microsoft Azure Kubernetes Service

For creating and managing Kubernetes clusters, Microsoft Azure provides the Azure Kubernetes Service (AKS) which allows users to easily create and manage the lifecycle of a Kubernetes cluster in the Azure infrastructure without the need for maintaining the control plane components of the cluster. Azure treats every managed component as a resource. A cluster in AKS is a resource as well, and can be assigned to a resource group. A cluster can be created in any available Azure location. AKS allows you to select a specific version of Kubernetes, giving you a choice whether to use the latest version or stay with an older version which is stable for your deployment or provides features removed in the newer version. One of the main advantages of using AKS is that the health and availability of the Kubernetes control plane is guaranteed by it, lifting the burden off the back of the users. AKS will create a control plane for each Kubernetes cluster. The AKS service doesn't operate by itself in maintaining the cluster, though; the following Azure services are used in conjunction with it:

- Azure Virtual Machines power the nodes of the Kubernetes cluster as virtual instances
- Azure Virtual Network manages the networking aspects of the cluster
- Azure Load Balancer distributes the load of requests received by the cluster

AKS itself provides monitoring tools integrated into the Azure Portal. These contain insights about the status of the underlying infrastructure (e.g. CPU and memory usage for each node) and a tool to visualize metrics.

AKS integrates very well into the standard `kubectl` command and allows to run the Kubernetes Dashboard with a simple command. This makes it very easy to use the standard features of Kubernetes without interacting with Azure-specific configuration or tools. On the other hand, the Kubernetes cluster is very well integrated into Azure and can be easily connected to other Azure services like the Azure Container Registry.

Azure Container Registry

While Azure Kubernetes Service can download Docker images from the public Docker Hub registry, it's often useful to be able to use a private Docker registry to keep your resources private. Azure Container Registry allows for set up of private Docker registries hosted on Azure. This makes it easy to set up a private Docker registry and also connect it with your Kubernetes cluster so that the cluster knows where to look for Docker images.

Understanding the Concepts of Microsoft Azure

Before diving into a step by step guide, let us first explore a few basic Azure concepts first:

Resource Group

A container that holds related resources for an Azure solution like Azure Kubernetes Service or Azure Container Registry. The resource group includes those resources that you want to manage as a group. These resources are very versatile and include Virtual Machines, Networks, and IP addresses.

Subscription

You can create several subscriptions within your Azure account, each containing multiple Resource Groups. You mainly define some Subscription to have a separation in the billing and management of the resources.

Azure Portal

A web console that allows you to build, manage, and monitor everything in Azure cloud. You can access it at <https://portal.azure.com> under your Azure account. It provides a unified view of all your resources in Azure and allows you to create and manage resources in Azure cloud in a visual and user friendly way.

Azure CLI

The Azure CLI is a command-line tool providing for managing Azure resources. The CLI is suitable for scripting, querying data, and for repetitive actions where the Azure Portal is not very convenient. In general, it's a more powerful tool than Azure Portal because it also allows specifying operations and configuration which isn't available via Azure Portal yet.

Requirements for Following this Guide

In order to set up your first cluster using Azure Kubernetes Service you will first need to install the following tools in your local machine:

- The Azure CLI command-line interface (az)
 - You can install it on Windows through an installer, or on Mac® you can use the package manager Homebrew. On Linux®, there is a shell script which performs the task.

More info can be found on the page <https://docs.microsoft.com/en-us/cli/azure/install-azure-cli?view=azure-cli-latest>.

- After you install it, you can authenticate with your Azure account using the command `az login`. This will open a browser window with the Azure login form.
- Have Docker installed locally on your machine.
- The `kubectcl` command-line utility which will allow you to interact with the Kubernetes cluster. There are multiple ways to install this tool:
 - The `kubectcl` program can be installed through the Azure CLI, with a simple command `az aks install-cli`.
 - For Windows and macOS® environments, the utility will be included when installing either [Docker Desktop](#) or [Minikube](#). I personally recommend using Docker Desktop since it provides both Docker and Kubernetes management tools with one installation.
 - For most environments, you can install the utility directly using either a package manager (`apt/yum/snap` for Linux, `Brew` for macOS, `Chocolatey` for Windows), or download the utility directly to your local machine as well. Both alternatives are documented [here](#).

Set up a Kubernetes Cluster

Before deploying applications in Kubernetes, we'll create and configure a Kubernetes cluster in Azure Kubernetes Service and connect the `kubectcl` to it so that we can manage the Kubernetes cluster in a standard way.

Create a Kubernetes Cluster

A Kubernetes Cluster is just like the other concepts like a Docker container Registry, LoadBalancer, etc ... a resource on the Azure Platform. This means that you can create it through the Azure Portal through a wizard or the Azure CLI.

Look at the ‘Create Resources’ screen on the Portal for a new resource called “Kubernetes Service”. You need to specify the following values:

Subscription *	Pay-As-You-Go
Resource group *	myResourceGroup
Create new	
Cluster details	
Kubernetes cluster name *	azure-k8s-cluster
Region *	(Europe) West Europe
Kubernetes version *	1.13.11 (default)
DNS name prefix *	azure-k8s-cluster-dns
Primary node pool	
<p>The number and size of nodes in the primary node pool in your cluster. For production workloads, at least 3 nodes are recommended for resiliency. For development or test workloads, only one node is required. You will not be able to change the node size after cluster creation, but you will be able to change the number of nodes in your cluster after creation. If you would like additional node pools, you will need to enable the "X" feature on the "Scale" tab which will allow you to add more node pools after creating the cluster. Learn more about node pools in Azure Kubernetes Service</p>	
Node size *	Standard DS2 v2
	Change size
Node count *	3

- *Resource Group*, is a logical grouping of resources and you should select the same group as you have chosen/created during the creation of the Docker Registry.
- *Kubernetes cluster name*, the name of your cluster. An indicative name that will also be used for DNS name prefix.
- *Region*, The Region of the Data Center where the cluster will be created. If possible, this should be near your customers for network latency reasons.
- *Kubernetes version*, the version of Kubernetes which will be installed in the cluster or you can leave the default.
- *DNS name prefix*, part of the name which can be used to access the Kubernetes API Server
- *Node size*, Size (CPU and memory) of the ‘machines’ within your Kubernetes cluster. The required value depends on the requirements of your application of course. You can find more information on the sizes on this [Azure documentation page](#).
- *Node count*, the initial number of nodes in your cluster.

There are multiple tabs available on this wizard with more parameters for specific use cases. Have a look at those when you are considering the Azure Platform for your Kubernetes platform. The important parameters are on this initial screen and are enough to get you started.

When you click the 'Review + create' button, the cluster is created for you, which can take up to 5 minutes.

Instead of creating the Cluster through the Azure Portal, you can also perform this task through the Azure CLI program.

The minimal command is:

```
az aks create -n azure-k8s-cluster -g myResourceGroup
```

Manage Kubernetes Cluster

For managing a Kubernetes Cluster, you can use the kubectl program. You can use it to send info to and retrieve info from the Kubernetes API server. You can also use this kubectl program to manage the Azure Kubernetes Service (AKS). Configuring the program to connect to the correct API server is very easy and can be done through the Azure CLI program.

```
az aks get-credentials -n azure-k8s-cluster -g myResourceGroup
```

If you do not have the kubectl program installed locally on your machine, you can install it using this Azure CLI command:

```
az aks install-cli
```

Once everything is installed and linked to your Azure Kubernetes cluster, you can start issuing commands. For example, you can see the number of nodes and verify with the number you requested during creation, with the command

```
kubectl get nodes
```


Access Kubernetes Dashboard

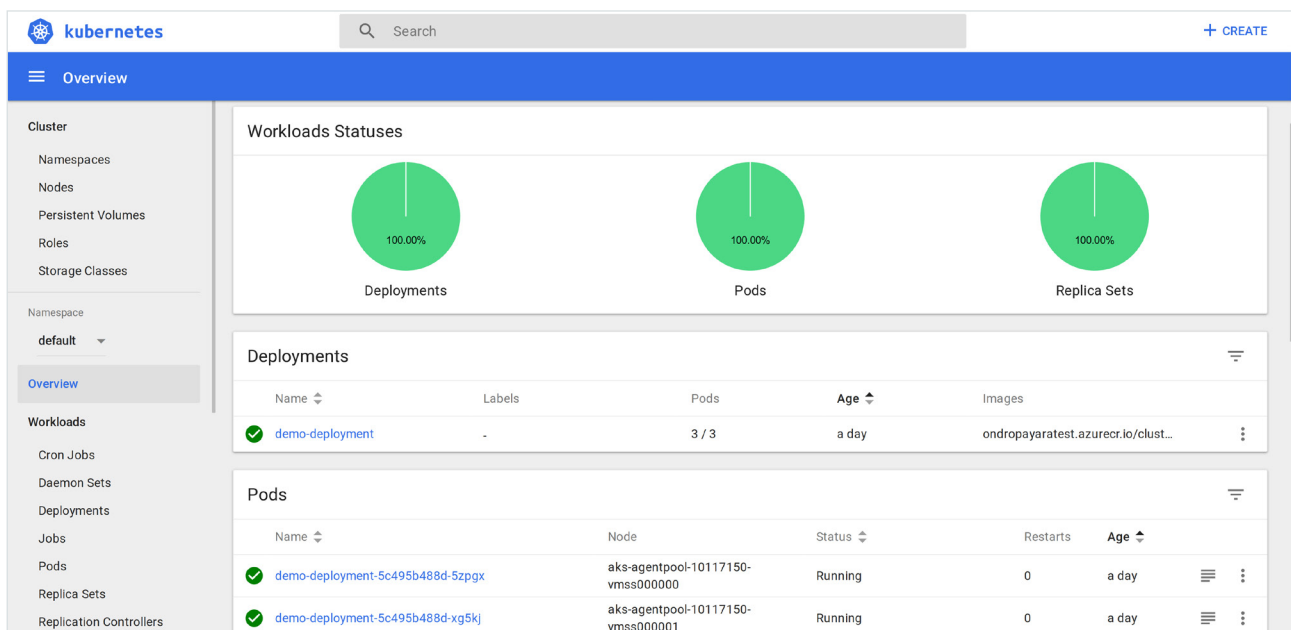
Kubernetes provides a standard Dashboard to manage resources in a Kubernetes cluster. It allows to display information and manage resources like deployments, pods and services using a web console. So its to kubectl what Azure Portal is to Azure CLI.

It's possible to enable and run the Dashboard in a standard way, using the kubectl command. But this requires several configuration steps, including setup of an authentication mechanism and accessing a very long and complicated URL in the browser. Azure CLI simplifies this a lot with just 2 simple steps.

First, run the following command to enable the Dashboard and open it in a browser:

```
az aks browse --resource-group myResourceGroup --name myAKSCluster
```

This will open the standard Kubernetes Dashboard in a browser window:



Behind the scenes, this simple command performs the following steps:

- connects to the Kubernetes cluster in AKS
- makes sure that the Dashboard is installed and enabled in the cluster
- starts a proxy that allows accessing the Dashboard as a service on localhost
- uses your Azure account to log into the Dashboard
- opens a browser window with the URL to the Dashboard proxy

Finally, you need to grant additional privileges to your Azure account. The Dashboard is deployed with insufficient privileges to access cluster resources and we need to raise the privileges. There are multiple ways to do this as described in the Azure documentation: <https://docs.microsoft.com/en-us/azure/aks/kubernetes-dashboard>. The simplest options is to bind a Kubernetes cluster role to the Kuberentes account linked to your Azure account. You can do that with the following command (you can copy and run it without modifications):

```
kubectl create clusterrolebinding kubernetes-dashboard --clusterrole=cluster-admin --serviceaccount=kube-system:kubernetes-dashboard
```

Deploy Applications to a Kubernetes Cluster

In order to deploy an application to our new Kubernetes cluster, we need to do the following steps:

- build an application WAR/EAR file
- build a Docker image with Payara Micro running our application
- push the Docker image to a Docker registry
- deploy Kubernetes resources that use the image in that Docker registry

Sample Payara Micro Application

First, we'll need a WAR/EAR application that we can run with Payara Micro. You can use any application as there are no special requirements for a Payara Micro application tor run in Kubernetes. However, in order to demonstrate the distributed capabilities of the Payara Platform and the self-managing aspects of Kubernetes, we'll use an application with the following features:

- The application will allow creating and storing data about users
 - Each user is comprised of a name, an organization, and a consecutive ID.
- The application will track the current consecutive number of users.
- The application will track the hostname of the pod's name, which runs the application instance that created the user
 - To verify that requests are routed to different instances in the cluster if there are more than 1
- The application will allow access to all stored data via any application instances, regardless which instance created the data

Our application will make use of the following set of APIs that are included in the Payara Platform in order to implement those features:

- **Jakarta EE Web API** (<https://jakarta.ee/>), which is a standard API provided by Payara Micro
- **JCache API** (<https://jcp.org/en/jsr/detail?id=107>), so that all users are cached in a distributed manner. If a user object is created by an instance that is part of the cluster, other members can access it too
- **Payara Public API** (<https://docs.payara.fish/documentation/payara-server/public-api/>), to have access to the `@Clustered` annotation which is proprietary of the Payara Platform. With this annotation, we can configure a `@Singleton` EJB as a “true” singleton, which means that only one instance of it exists across all the instances that are member of the cluster. This clustered singleton is used to generate IDs unique to the whole cluster

In order to build the application, you can find the sources in <https://github.com/payara/Payara-Examples/tree/master/cloud-providers/azure/kubernetes/payara-micro-distributed-cache> and build with the following command (Maven must be installed, more info about Maven here: <https://maven.apache.org/>):

```
mvn clean package
```

You’ll then find the built WAR application `cluster-demo.war` in the `target` directory. We’ll use this file later to deploy the application using Docker.

Create Docker Image

When you have developed your application, you should have a WAR/EAR file available by now. This application can be packaged with Payara Micro in a self-contained Docker image.

Payara provides an official Docker image for Payara Micro which is tuned for production usage but is also very useful in development. It’s pre-configured in a way that makes it easy to start an application without a custom Dockerfile. You can also build your images using the Payara Micro image as a base to remove several required configuration steps to run your application in Docker.

The Payara Micro Docker image can be found on DockerHub: <https://hub.docker.com/r/payara/micro/>

Since we need a specific Docker Image containing the application, we can build it with the following DockerFile.

Dockerfile

```
FROM payara/micro:5.193
COPY target/cluster-demo.war $DEPLOY_DIR/cluster-demo.war
CMD ["--contextRoot", "/", "/opt/payara/deployments/cluster-demo.war"]
```

Running the container produced from this image will also deploy the application on startup. The image will need rebuilding when the application changes.

The next step is to build the image from this Dockerfile. You can do it using this Docker build command:

```
docker build -t cluster-demo .
```

This will build an image and label it with the `cluster-demo` tag.

In the following section, we'll explain how to push this image to a private Docker registry powered by Azure Docker Registry. In order to do this, the tag associated with the Docker Image needs to have a format specified by the Azure Docker Registry. You can apply the required tag to your images after you build them, But it's simpler if you apply the tag right when building the image, replacing the `cluster-demo` tag in the above `docker build` command.

Push an Image to a Docker Registry

The Docker image needs to be available in a Docker registry in order to be used by a Kubernetes cluster in Azure Kubernetes Service (AKS). By default, AKS searches for Docker images in the public DockerHub repository. But you can also create a private Docker registry on the Azure platform which is preferred for your own applications which you don't like to share publicly.

You can create a registry through the Azure Portal or from the Azure CLI program. Connecting to this registry and pushing images to it is much easier using the Azure CLI.

When using the Portal, look for a new resource called "Container Registry" or ACR:

Create container registry

* Registry name

payaraTest ✓

.azurecr.io

* Subscription

Pay-As-You-Go ▼

* Resource group

(New) myResourceGroup ▼

Create new

* Location

West Europe ▼

* Admin user ⓘ

Enable Disable

* SKU ⓘ

Basic ▼

You need to specify the following values:

- *Registry name* is a name of the registry and will become part of the URL on which your private Docker Registry is hosted. This needs to be unique across the world. It's possible that your desired name is already taken by some other Azure user
- *Resource Group* is a logical grouping of resources. It makes it easy for you to find a specific resource or can be used to remove all resources in one go
- *SKU* defines the size of the registry. For most cases, the Basic option will suffice.

The same action can be performed from the Azure CLI tool:

CommandLine

```
az acr create -n payaratest -g myResourceGroup --sku Basic
```

Now that we have a Docker Registry available for our Docker images, we are ready to push to the Azure cloud.

Since we have created a private registry, some authentication is required to access it from our local Docker installation. The Azure CLI has a very useful command for this, which uses the credentials of your Azure account used for the Azure CLI command. So, after you authenticate with your Azure account (using `az login`), you can just execute the following command to authenticate Docker with your Azure account:

CommandLine

```
az acr login -n payaratest
```

The argument `-n payaratest` specifies the name of the Docker registry created in the ACR. Now our Docker CLI properly authenticated via your Azure account and we can send the image over to the `payaratest` registry in ACR.

The Docker image that we want to push to our registry in ACR needs to be labelled with a tag named according to the following format:

`<myRegistry>.azurecr.io/<name>:<version>`

The version element is optional but it is a good practice to always define it.

If you already have an image available, you can give it the correct tag so that it can be pushed. Otherwise, you can build it and supply the expected tag name.

For example, when you already have a Docker Image called *cluster-demo* available locally, the following command labels it with the expected tag name:

CommandLine

```
docker tag cluster-demo payaratest.azurecr.io/cluster-demo:v1
```

Or you can specify the tag name directly when you build the Docker Image with a command similar to:

CommandLine

```
docker build -t payaratest.azurecr.io/cluster-demo:v1 .
```

And the last step we need to perform is to push the Image using the expected tag name:

CommandLine

```
docker push payaratest.azurecr.io/cluster-demo:v1
```

This will upload all the layers of the Docker image which haven't been uploaded to the registry yet. So it can take some time when you push a new version of the image.

Attach a Private ACR Registry to a Kubernetes Cluster

If we push our Docker images to a registry in the Azure Container Registry (ACR), we need to link this Registry to our cluster so that Kubernetes can pull images from this registry automatically in a secure way.

To link the Docker Registry with the Kubernetes Cluster, execute the following Azure CLI command

```
az aks update -n azure-k8s-cluster -g myResourceGroup --attach-acr payaratest
```

This will attach a registry called `payaratest` to a cluster called `azure-k8s-cluster` in the `myResourceGroup` group.

Deploy to a Kubernetes cluster

Once the Kubernetes cluster is set up, you can use it in a standard way using the `kubectl` command line tool.

To run Dockerized Payara Micro application on the Azure Kubernetes Service (AKS), we need to define the Kubernetes Deployment and Kubernetes Service resources.

deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: demo-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: cluster-demo
```

```
template:
  metadata:
    labels:
      app: cluster-demo
  spec:
    containers:
      - name: py-micro-demo
        image: payaratest.azurecr.io/cluster-demo:v1
        imagePullPolicy: Always
        ports:
          - containerPort: 8080
        args:
          - "--clustermode"
          - "kubernetes"
```

This YAML file describes the deployment. Some of the important parts in this configuration file are:

- spec/replicas defines the number of instances running your application.
- spec/template/spec/containers/image defines which Docker image will be used for the containers. In this case, it points to an image in our ACR registry
- spec/template/spec/containers/ports/containerPort: The port on which the application is accessible
- spec/template/spec/containers/args: Arguments for Payara Micro. You can also define them in the Docker image itself with a CMD command. If you want to make the Docker Image more generic, you can define some Kubernetes specific options this way (e.g. Kubernetes cluster mode)

You should also have a look at defining the readiness and liveness probes so that a pod is added correctly to the load balancing algorithm. You can read more about it in this blog <https://blog.payara.fish/scaling-payara-micro-applications-with-kubernetes>.

It's important to note here that we define the cluster mode of the Data Grid as Kubernetes. So Payara Micro will use the Kubernetes API server to discover the other instances of our application so that they automatically join the same Data Grid. For this, we need to define some permissions so that Payara Micro can access the Kubernetes API.

rbac.yaml

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: default-cluster
roleRef:
```



```
apiGroup: rbac.authorization.k8s.io
kind: ClusterRole
name: view
subjects:
- kind: ServiceAccount
  name: default
  namespace: default
```

This grants some view permission to the Cluster information to the Service Account. This way, the DataGrid can automatically create a cluster and we can share information between the different instances as we are doing within our example application.

The above 2 files can be applied to the Azure Kubernetes service:

```
kubectl apply -f rbac.yaml
kubectl apply -f deployment.yaml
```

Now that we have deployed our application, we can verify if it was successful and we can see our application's pods created by the deployment:

```
kubectl get pods
kubectl log cluster-demo-xxxxx
```

The list of available pods should list 3 pods created by our deployment because the deployment specifies to have 3 replicas of our application:

Kubectl get pods

NAME	READY	STATUS	RESTARTS	AGE
demo-deployment-766bd6967f-mmzrl	1/1	Running	0	37s
demo-deployment-766bd6967f-ps2xg	1/1	Running	0	37s
demo-deployment-766bd6967f-wm7rk	1/1	Running	0	37s

Looking at the cluster-demo logs, we see that the 3 instances are joined into a single cluster of the Data Grid. After some time, each of the logs should contain lines similar to this:

kubectl log

```
Payara Data Grid State: DG Version: 35 DG Name: development DG Size: 3
```

```
Instances: {
  DataGrid: development Instance Group: MicroShoal Name: Busy-Nibbler Lite:
false This: true UUID: fc69e74b-c7bd-44cc-ac9c-f794a85e3dd0 Address:
/10.244.0.9:6900
  DataGrid: development Lite: false This: false UUID: cdbf836a-b9bf-4648-a2ad-
e963e01bed2a Address: /10.244.2.3:6900
  DataGrid: development Lite: false This: false UUID: d722a33e-c7e4-4be4-be6f-
3c4197125b61 Address: /10.244.1.4:6900
}]]
```

Now we are ready for the next step, making our application available from outside of the cluster with a single endpoint. We need to define a service that exposes all the pods created by our deployment as a single endpoint via a load balancer:

service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: demo-service
spec:
  type: LoadBalancer
  ports:
    - port: 8080
  selector:
    app: cluster-demo
```

This Kubernetes service definition is quite simple. The main points consist of

- spec/type: defines the type of LoadBalancer which will be used by the Azure platform so that we have a round-robin type of calls to each of the instances
- spec/ports/port: Define the port on the pods which we want to expose via the load balancer. By default, this is also the port number to be used from the outside world
- spec/selector: Based on what criteria pods will be attached to the load balancer. The `app: cluster-demo` value is also defined in the deployment as match labels. Therefore the load balancer balance among all the pods created by the deployment

We need to apply this file to our cluster to create the load balancer service:












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

We can then verify the service by issuing the command:

```
kubectl get svc demo-service
```

Initially, you will see that there is no external IP address assigned to your service, but after some time, the external IP address is known and can be used in our testing.

All the information is accessible also through the Azure Portal. There you can see that a lot of resources are created to support our application, from loadbalancer, through IP addresses, to containers.

All resources	
Default Directory	
<div><div><div><div></div></div><div>Add</div></div><div><div><div></div></div><div>Edit columns</div></div><div><div><div></div></div><div>Refresh</div></div><div><div><div></div></div><div>Export to CSV</div></div><div><div><div></div></div><div>Assign tags</div></div><div><div><div></div></div><div>Delete</div></div><div><div><div></div></div><div>Feedback</div></div></div>	
<div><div><div>Filter by name...</div></div><div>Subscription == all</div><div>Resource group == all</div><div>Type == all</div><div>Location == all</div></div>	
Showing 1 to 11 of 11 records. <input type="checkbox"/> Show hidden types	
<input type="checkbox"/> Name ↑↓	Type ↑↓
<input type="checkbox"/>  7783f6b2-bd0d-4db8-8271-af7b03ec24a0	Public IP address
<input type="checkbox"/>  aks-agentpool-90984536-nsg	Network security group
<input type="checkbox"/>  aks-agentpool-90984536-routetable	Route table
<input type="checkbox"/>  aks-agentpool-90984536-vmss	Virtual machine scale set
<input type="checkbox"/>  aks-vnet-90984536	Virtual network
<input type="checkbox"/>  azure-k8s-cluster	Kubernetes service
<input type="checkbox"/>  ContainerInsights(defaultworkspace-1b72a038-5d25-46f4-807f-a008f...	Solution
<input type="checkbox"/>  DefaultWorkspace-1b72a038-5d25-46f4-807f-a008fddf5bef-WEU	Log Analytics workspace
<input type="checkbox"/>  kubernetes	Load balancer
<input type="checkbox"/>  kubernetes-ae042547eff1311e98630f66780e87f5	Public IP address
<input type="checkbox"/>  rubus	Container registry

Verify Application Deployment

Now that the application is deployed and accessible to the outside world via an external IP address, let's try it out with the following curl commands. If you can't use curl, you can fire similar requests with any REST client.

```
curl http://<ipaddress>:8080/data/all
curl -X POST -i http://<ipaddress>:8080/data -H 'Content-Type: application/json' -d '{"name": "Rudy De Busscher", "organization": "Payara Services Ltd."}'
```

The first request is a GET request which retrieves all data via the data/all URL. The second request is a POST request that sends data about a new user and stores them in a distributed cache powered by Payara Data Grid.

In the beginning, there are no data stored in the cache which is distributed among all application instances. So the first command should just return an empty set. If it completes without an error, we know everything is working and the application endpoints can be accessed from outside of the Azure Kubernetes Cluster.

With the second command, we actually post some data and we should get a reply with HTTP Status 201 that an entry is created. We can retrieve the data via an URL resource on which it is accessible:

```
curl http://<ipaddress>:8080/data/1
```

We get back the same information which we posted before. We can verify which of the 3 instances processed the original POST operation with the value of the `createdOnInstance` property. It contains the pod name of the instances that saved the data.

You cannot see it, but retrieving the data for that first record was handled by another instance than the one which saved it. This is due to the fact Azure LoadBalancer uses a round-robin algorithm. We can see it better when we post other data.

```
curl -X POST -i http://<ipaddress>:8080/data -H 'Content-Type: application/json' -d '{"name": "Ondrej Mihalyi", "organization": "Payara Services Ltd."}'
curl http://<ipaddress>:8080/data/all
```

Retrieving all records now, we clearly see that requests are processed by different instances or Pods. And due to the automatic clustering capabilities of the Data Grid of Payara Micro, all information is available to all instances.

Advanced Operations on the Kubernetes Cluster

All the features expected in Kubernetes can be used on an Azure Kubernetes Service cluster. The cluster is just a regular Kubernetes installation with the added value that it's also an Azure resource and is integrated with other resources in Azure. We can try out a few of the resilience and scaling aspects.

```
kubectl delete pod demo-deployment-xxxx
kubectl scale --replicas=2 deployment/demo-deployment
```

The first command removes one of the pods. This is a very unlikely operation but it simulates a more common event that the container within the pod crashes. In the event that a pod is down, another pod is recreated to satisfy the number of replicas specified in the deployment. In the case a container within a Pod dies, this can be detected by a liveness probe and the Kubernetes system will make sure that the container is restarted.

The second command scales the number of instances of our application. This can be down or up in a manual fashion as in this case or when we define the horizontal pod scaling, automatically based on CPU usage for example. Adding and removing instances doesn't affect the data stored in the DataGrid, as long as at least one instance is running. When new instances connect to already running instances in the Data Grid they can access the data in the distributed cache. Data in distributed caches are evenly distributed and replicated so they will be available even in case instances are removed.

Payara Micro is a Perfect Fit for Kubernetes and Azure

Payara Micro is an ideal platform to build elastic Java applications deployed in Kubernetes. Payara Micro can be easily deployed in Kubernetes using the official Payara Micro Docker images which can be flexibly customized for any deployment configuration. It supports elastic clustering with data grid that allows distributing session and cached data among multiple instances which can be dynamically scaled seamlessly, without any custom configuration changes. With the Kubernetes cluster mode, Payara Micro instances automatically detect all the other instances in the Kubernetes cluster and join the same data grid. Payara Micro also offers a health status endpoint that can be easily used by Kubernetes health probes to detect failed instances and restart them.

Microsoft Azure cloud offers an easy way to provision a Kubernetes cluster in the cloud and integrate it easily with other services and resources in Azure. The Azure Kubernetes Service (AKS) greatly simplifies using standard Kubernetes tooling like kubectl with Kubernetes clusters in Azure. The Azure CLI allows to install kubectl, connect it to a specified cluster, connect the cluster to a Docker registry in Azure Container Registry (ACR), install and open the Kubernetes Dashboard; all of that

with a few simple commands. AKS enables managing Kubernetes very easily and be productive with it, while offering all of the standard features expected in any Kubernetes distribution.

If you are considering lifting and shifting existing Java EE applications to Kubernetes in Azure or creating new flexible cloud-native applications for Kubernetes in Azure, Payara Micro is the perfect platform choice. Payara Micro is very well suited for Kubernetes deployments in Azure. It can take advantage of the robust collection of services provided by the Azure cloud, including Azure Service Bus, which can be very easily connected to a Payara Micro application using the cloud connector tested with Payara Platform (<https://github.com/payara/Cloud-Connectors/tree/master/AzureServiceBus>). If you need assistance, the [Payara Accelerator](#) team can provide customized consultancy services to [support customers](#) to advise on architectures and migration strategies or they can lift and shift your application onto Kubernetes and Azure for you. The [Payara Enterprise Support](#) service provides support for the Payara Platform on Azure and direct access to Payara Engineers to ask questions and assist you if you have problems in moving your application to Microsoft Azure Kubernetes Service.

Docker and the Docker logo are trademarks or registered trademarks of Docker, Inc. in the United States and/or other countries. Docker, Inc. and other parties may also have trademark rights in other terms used herein.

Kubernetes is a registered trademark of The Linux Foundation in the United States and/or other countries.

Microsoft, Azure, and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

Mac and macOS are trademarks of Apple Inc., registered in the U.S. and other countries.

The registered trademark Linux® is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a world-wide basis.



sales@payara.fish



+44 207 754 0481



www.payara.fish

Payara Services Ltd 2018 All Rights Reserved. Registered in England and Wales; Registration Number 09998946
Registered Office: Malvern Hills Science Park, Geraldine Road, Malvern, United Kingdom, WR14 3SZ