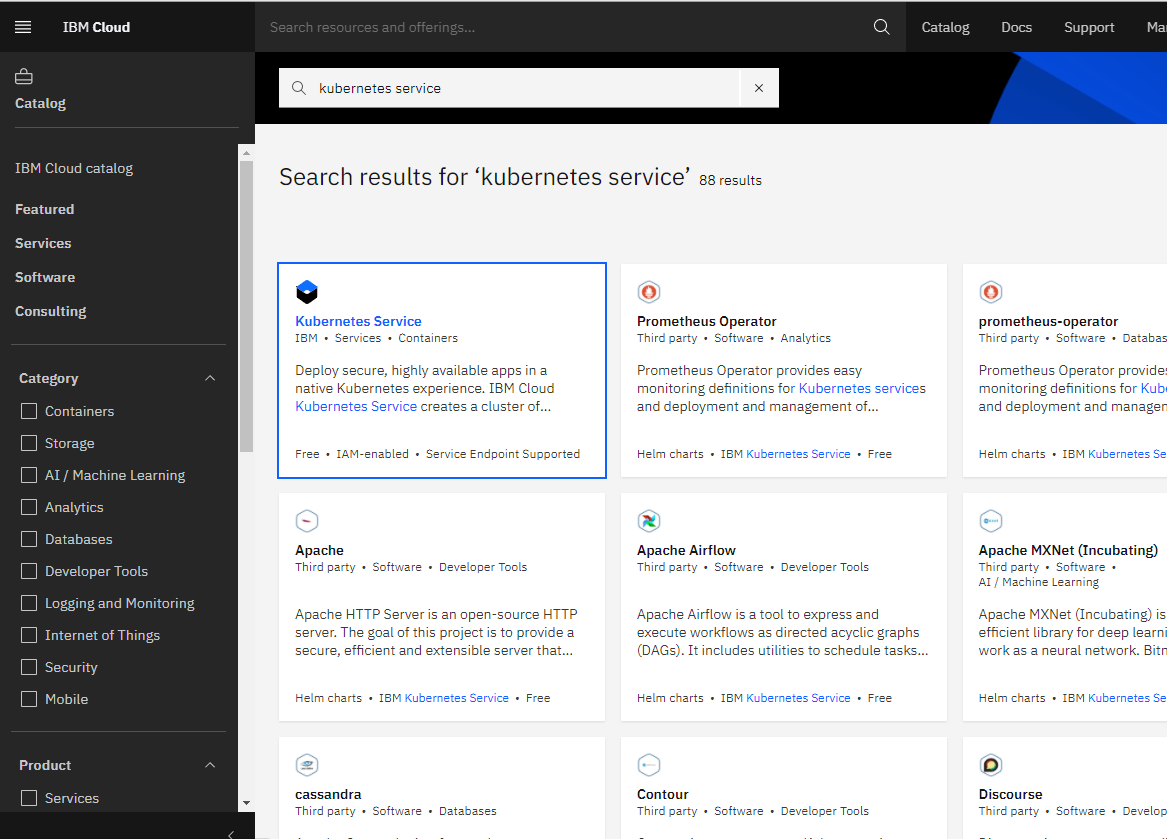
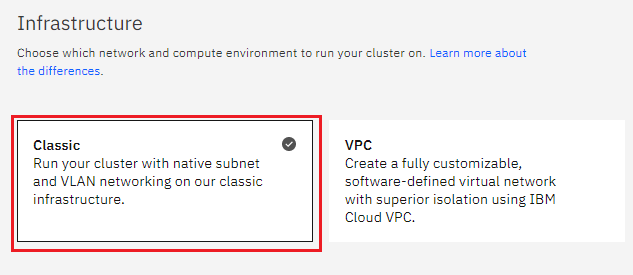
# Installing ASP.NET on IBM Cloud

**Step 1 provision Kubernetes Cluster**

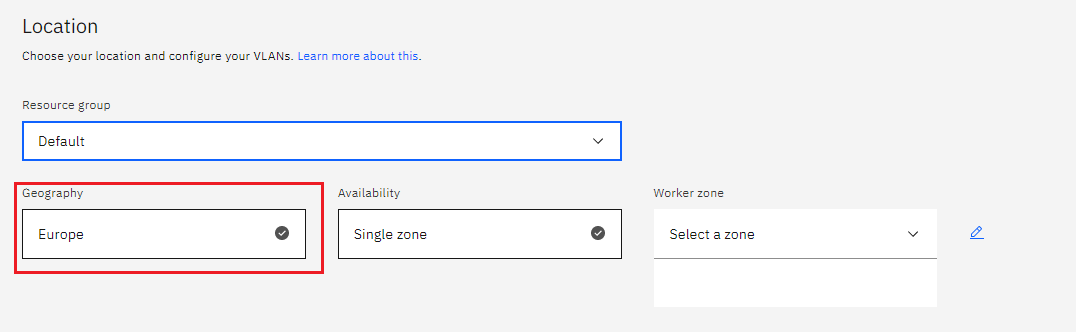
* Click the **Catalog** button on the top
* Select **Service** from the **Catalog**
* Search for **Kubernetes Service** and click on it



* You are now at the Kubernetes deployment page. You need to specify some details about the cluster
* Choose a plan **standard** or **free** , the free plan only has one worker node and no subnet, to provision a standard cluster, you will need to upgrade your account to Pay-As-You-Go
* To upgrade to a Pay-As-You-Go account, complete the following steps:
* In the console, go to Manage > Account.
* Select Account settings; and click Add credit card.
* Enter your payment information, click Next, and submit your information
* Choose **classic** or **VPC** , read the docs and choose the most suitable type for yourself



* Now choose your location settings,
* Choose **Geography** (continent)



* + Choose Single or Multizone, in single zone your data is only kept in on datacenter, on the

other hand with Multizone it is distributed to multiple zones, thus safer in an unforeseen

zone failure

* If you wish to use Multizone please set up your account with[VRF
* If at your current location selection, there is no available Virtual LAN, a new Vlan will be created for you
* Choose a Worker node setup or use the preselected one, set Worker node amount per zone
* Choose **Master Service Endpoint** , In VRF-enabled accounts, you can choose private-only to make your master accessible on the private network or via VPN tunnel. Choose public-only to make your master publicly accessible. When you have a VRF-enabled account, your cluster is set up by default to use both private and public endpoints.  
  Give desired **tags** to your cluster, for more information visit tags
* Click **create**  
  • Wait for your cluster to be provisioned  
  • Your cluster is ready for usage

**Step 2 Deploy IBM Cloud Block Storage plug-in**

The Block Storage plug-in is a persistent, high-performance iSCSI storage that you can add to your apps by using Kubernetes Persistent Volumes (PVs).

* Click the **Catalog** button on the top
* Select **Software** from the catalog
* Search for **IBM Cloud Block Storage plug-in** and click on it  
  • On the application page Click in the dot next to the cluster, you wish to use  
  • Click on Enter or Select Namespace and choose the default Namespace or use a custom one (if you get error please wait 30 minutes for the cluster to finalize)
* Give a **name** to this workspace
* Click **install** and wait for the deployment

**Step 3 For ASP.NET**

Following should be installed on the machine:

1. **IBM Cloud account**
2. **IBM CLoud CLI**
3. **Git**
4. **.NET Core 2.1.1 SDK 2.1.301**

## **Steps:**

### **Clone/download the IBM Cloud ASP.NET Core Getting Started Application**

git clone https://github.com/IBM-Cloud/get-started-aspnet-core

### **Run the ASP.NET Core app locally**

This step is to verify whether your app is running successfully locally before deployment. You can start by verifying the version of dotnet as follows:

dotnet --version

1. **Next, navigate to your App folder.**

cd get-started-aspnet-core/src/GetStartedDotnet

1. **Restore the app with the following command:**

dotnet restore

This uses NuGet to restore dependencies and project-specific tools that are specified in the project file. By default, the restoration of dependencies and tools are executed in parallel. For more info visit [Docs](https://docs.microsoft.com/en-us/dotnet/core/tools/dotnet-restore?tabs=netcore2x).

Now, run your application with the following command:

dotnet run

The application starts listening on port 5000. You will see the following message.

...

Now listening on: http://localhost:5000

Application started. Press Ctrl+C to shut down.

### **Publish the ASP.NET Core app**

To pack the application and its dependencies, create a new folder named publish for deployment to a hosting system for execution. We have to use dotnet to publish. Then it’s ready to run anywhere.

Publish the app to get a self-contained DLL using the dotnet publish command.

dotnet publish -c Release

Running publish displays some messages with a successfully published DLL at the end of the process. For our example, you can see the following message.

...

GetStartedDotnet -> /home/get-started-aspnet-core/src/GetStartedDotnet/bin/Release/netcoreapp2.0/GetStartedDotnet.dll

### **Dockerize the ASP.NET Core app**

Once the application is ready, we can make an image and put it inside a container. We need a file that contains step-by-step instructions to deploy the image inside the container to run our application anywhere. This Dockerfile is a basic file and you may only require a few lines to get started with your own image.

Go to the app folder (here GetStartedDotnet) and create a Dockerfile to define the Docker image.

cd get-started-aspnet-core/src/GetStartedDotnet

Add the contents of [Dockerfile](https://github.ibm.com/Nidhi-N-Shah/ASP.NET-CORE-App-Deployment-in-IKS/blob/master/Dockerfile) by using your favorite editor (vim, nano, etc.) and save the file.

vi Dockerfile

FROM microsoft/aspnetcore:2.0

WORKDIR /app1

COPY ./bin/Release/netcoreapp2.0/publish .

ENTRYPOINT ["dotnet", "GetStartedDotnet.dll"]

The first line we added FROM microsoft/aspnetcore:2.0 will download the aspnetcore image from the hub repository, so it actually contains the .NET Core and you don’t need to put it inside the image. You can find more repositories and versions in the Docker hub.

The Dockerfile line WORKDIR /app sets our working directory in the app folder, which is inside the container that we’re building.

Now we need to copy the contents of the publish folder into the app folder on the image COPY ./bin/Release/netcoreapp2.0/publish.

**Note:** You can find this path in the output when you run dotnet publish -c Release.

The last line in the Dockerfile is the ENTRYPOINT statement: ENTRYPOINT ["dotnet", "GetStartedDotnet.dll"]. This line tells Docker that it should run the dotnet command with GetStartedDotnet.dll as parameter.

### **Run the app on Docker (Optional)**

You can test your dockerized app by following the steps below. This section is optional for this tutorial, though.

First, build an image.

docker build -t get-started-aspnet

**Note:** You can choose any name for your app.

Running the build command displays the following message in the end.

...

Successfully tagged get-started-aspnet:latest

The following command will run an app.

docker run -d -p 8080:80 --name app get-started-aspnet

Navigate to http://localhost:8080 to access your app in a web browser.

Clean up with the following commands.

docker stop /app

docker rm /app

These above commands stop and remove the Docker container of your app, respectively. You can use them to remove your container if you no longer need it.

### **Create a Kubernetes cluster on the IKS environment**

We are now ready to create our [Kubernetes cluster](https://cloud.ibm.com/containers-kubernetes/clusters).

1. Make sure you are logged into your [IBM Cloud](https://cloud.ibm.com/login?cm_sp=ibmdev-_-developer-tutorials-_-cloudreg) account by using:
2. ibmcloud login

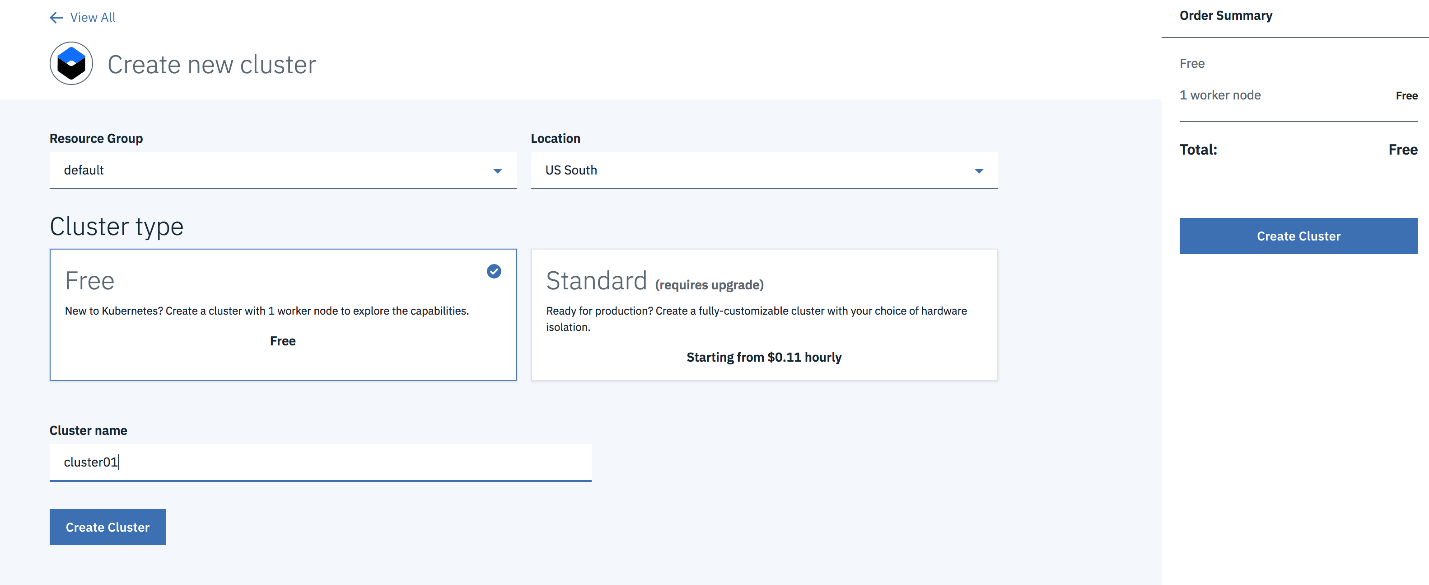
or

ibmcloud login --sso

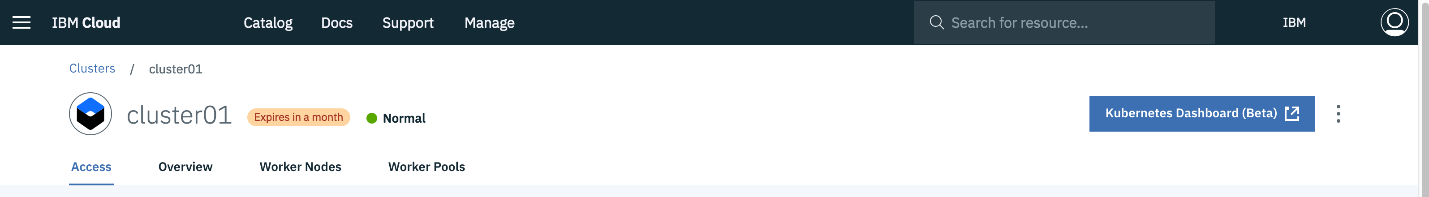
1. Create the IKS cluster for deployment.

i. Create a [Kubernetes cluster](https://cloud.ibm.com/containers-kubernetes/overview) by choosing **Cluster Type – Free**. Give a unique name to the cluster and click **Create Cluster**.

Note: For more details, see [Creating a Kubernetes cluster in IBM Cloud](https://cloud.ibm.com/docs/containers/container_index.html#clusters).



ii. It will take some time. It is ready to use if you see the following:



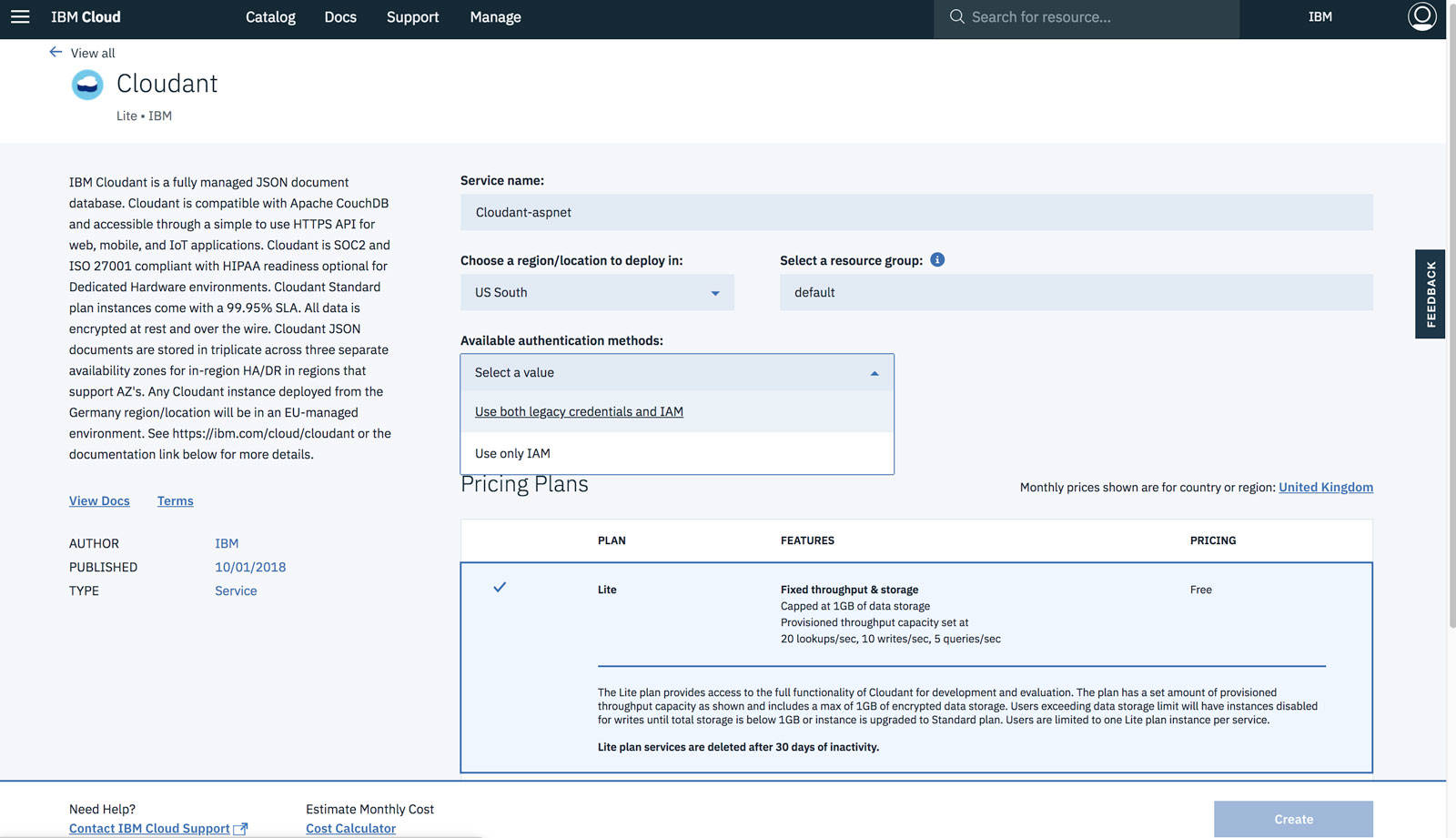
1. It’s time to deploy your containerized application to the Kubernetes cluster. From now on, you’ll use the kubectl command line.
2. Follow the instructions in the **Access** tab to set up your kubectl CLI and get access to your cluster.
3. On running the kubectl get nodes command, you should see something like the following.
4. NAME STATUS AGE VERSION
5. 10.76.197.43 Ready 1d v1.10.8+IKS

Show more

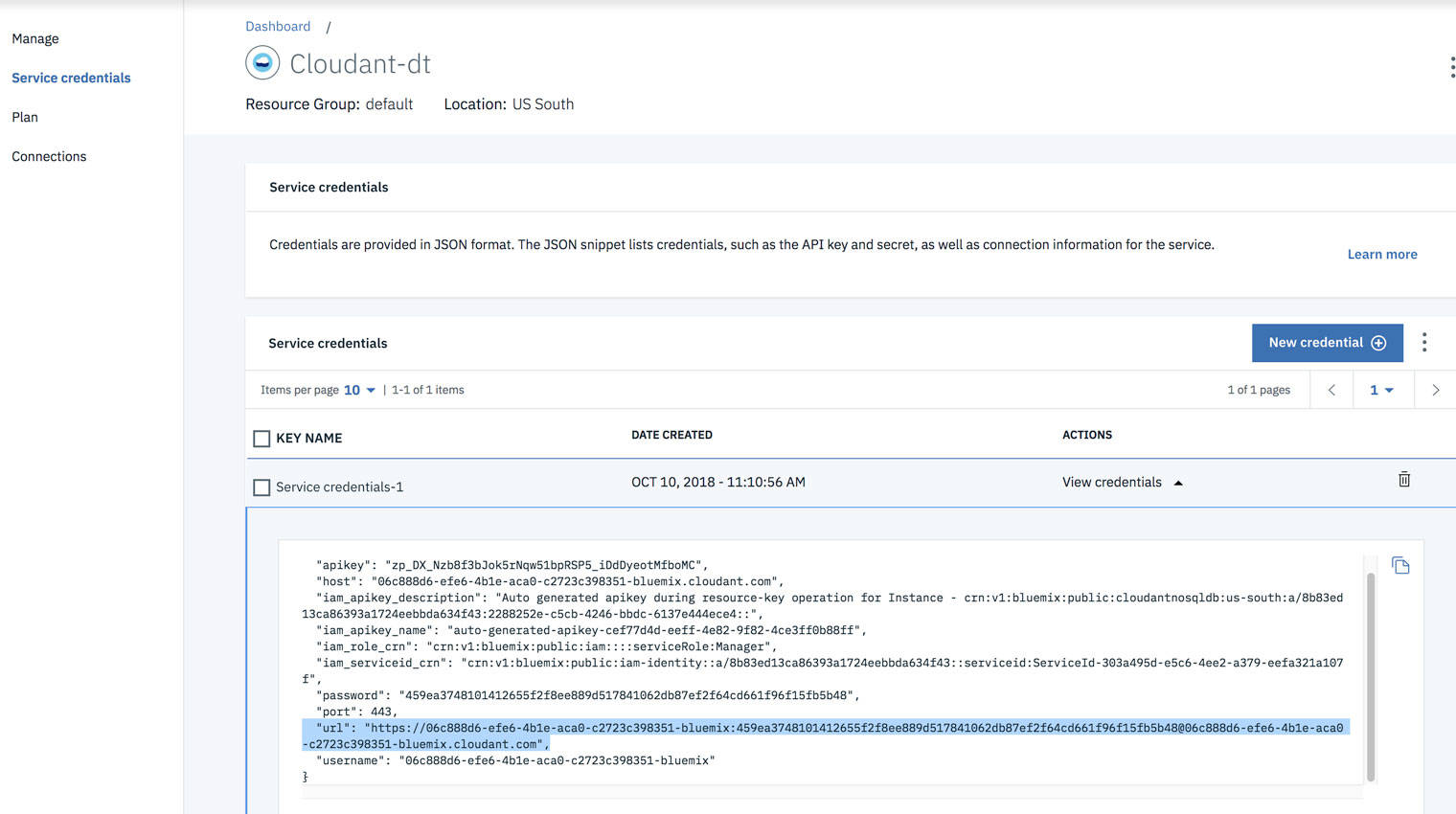
### **Create a Cloudant database in the IKS Cluster**

1. Our example application uses the Cloudant database to save the data that was entered. The goal here is to show how the database is created and can be deployed to the IKS cluster. You can use any database for this purpose and follow the same steps of deployment that are mentioned here (or avoid these steps if your application does not use database at all).

To create an IBM Cloud Cloudant Database, create a new [Cloudant](https://cloud.ibm.com/catalog/services/cloudant) database instance. Select **Use both legacy credentials and IAM** under **Available authentication methods**.



1. Create new credentials under **Service Credentials** and copy the value of the **url** field. (See image below).



1. Create a Kubernetes secret with your Cloudant credentials. A secret is an object that contains a small amount of sensitive data, such as a password, a token, or a key. Such information might otherwise be put in a Pod specification or in an image; putting it in a Secret object allows for more control over how it is used, and reduces the risk of accidental exposure. For more info, visit [Kubernetes Secret](https://kubernetes.io/docs/concepts/configuration/secret/).
2. kubectl create secret generic cloudant --from-literal=url=<URL>

Show more

For example:

kubectl create secret generic cloudant --from-literal=url=https://username:passw0rdf@username-bluemix.cloudant.com

Show more

1. You will need this info in your deployment. You can see your secrets by using the following command.
2. kubectl get secrets

Show more

This will display all the secrets you created in their respective clusters.

### **Deploy ASP.NET Core app to an IKS cluster**

The [IBM Cloud Container Registry](https://cloud.ibm.com/kubernetes/catalog/registry) provides a multi-tenant private image registry that you can use to safely store and share your Docker images with users in your IBM Cloud account.

1. Log in to the Container Registry Service to store the Docker image that we created with Docker.

ibmcloud cr login

1. Find your container registry namespace by running the following command.

ibmcloud cr namespaces

1. If you don’t have any, create one by using following command.

ibmcloud cr namespace-add <name>

For example:

ibmcloud cr namespace-add aspnetapp-01

1. Identify your Container Registry by running the following command.

ibmcloud cr info

For example: registry.ng.bluemix.net

1. Build and tag (-t) the Docker image by running the command below, replacing REGISTRY and NAMESPACE with the appropriate values.

docker build . -t <REGISTRY>/<NAMESPACE>/myapp:v1.0.0

For example:

docker build . -t registry.ng.bluemix.net/aspnetapp-01/myapp:v1.0.0

It will display the following message in the end.

...

Successfully tagged registry.ng.bluemix.net/aspnetapp-01/app:v1.0.0

1. Push the Docker image to your [Container Registry on IBM Cloud](https://cloud.ibm.com/docs/services/Registry?topic=registry-index#index).

docker push <REGISTRY>/<NAMESPACE>/myapp:v1.0.0

1. Verify that the image was pushed successfully by running the following command.

ibmcloud cr image-list

You have set up a namespace in the IBM Cloud Container Registry and pushed a Docker image to your namespace.

### **Deploy your containerized application**

Once you have a running Kubernetes cluster, you can deploy your containerized application on top of it. To do so, you create a Kubernetes Deployment configuration. The Deployment instructs Kubernetes on how to create and update instances of your application. Once you create a Deployment, the Kubernetes master schedules the mentioned application instances onto individual Nodes in the cluster. A Kubernetes Deployment Controller continuously monitors those instances that were created. If the Node that’s hosting an instance goes down or is deleted, the Deployment controller replaces it. This provides a self-healing mechanism to address machine failure or maintenance.

1. To create a deployment, you will create a folder called **“kubernetes”** and create a [deployment.yaml](https://github.ibm.com/Nidhi-N-Shah/ASP.NET-CORE-App-Deployment-in-IKS/blob/master/Kubernetes/deployment.yaml) file.
2. mkdir kubernetes

Show more

vi deployment.yaml

# Update <REGISTRY> <NAMESPACE> values before use

# Replace app name instead of get-started-aspnet if you wish to use different name for your app

apiVersion: apps/v1

kind: Deployment

metadata:

name: get-started-aspnet

labels:

app: get-started-aspnet

spec:

replicas: 2

selector:

matchLabels:

app: get-started-aspnet

template:

metadata:

labels:

app: get-started-aspnet

spec:

containers:

- name: get-started-aspnet

image: <REGISTRY>/<NAMESPACE>/myapp:v1.0.0

ports:

- containerPort: 8080

imagePullPolicy: Always

env:

- name: CLOUDANT\_URL

valueFrom:

secretKeyRef:

name: cloudant

key: url

optional: true

The deployment get-started-aspnet was created, indicated by the .metadata.name field. The Deployment creates two replicated Pods, indicated by the replicas field. These replicas are needed to handle the traffic in deployment. You can keep it to 1 as well. The selector field defines how the Deployment finds which Pods to manage. However, more sophisticated selection rules are possible, as long as the Pod template itself satisfies the rule. The Pods labeled app: get-started-aspnet are using the labels field. The Pod template’s specification, or .template.spec field, indicates that the Pods run one container, get-started-aspnet, which runs the <REGISTRY>/<NAMESPACE>/myapp:v1.0.0 Docker image. Open port 8080 so that the container can send and accept traffic. Set the imagePullPolicy of the container to Always. The Secret information has been updated in the env field, like the CLOUDANT\_URL that we mentioned while creating our Secret for the Cloudant database.

1. Create a deployment by using the following command.
2. kubectl create -f kubernetes/deployment.yaml

The output will display, similar to the following message.

deployment "get-started-aspnet" created

1. By default, the pod is only accessible by its internal IP within the cluster. Create a Kubernetes Service object that external clients can use to access an application running in a cluster. The Service provides load balancing for an application.

Use the NodePort 8080 to expose the deployment.

kubectl expose deployment get-started-aspnet --type NodePort --port 8080 --target-port 8080

You will see the following message.

service "get-started-aspnet" exposed

### **Access the application**

To verify that your application is running successfully, you need to check the STATUS of your pod. It should be in a state of Running:

kubectl get pods -l app=get-started-aspnet

It will appear like the following:

NAME READY STATUS RESTARTS AGE

get-started-aspnet-68d6dc5c4-2trcl 1/1 Running 0 1m

get-started-aspnet-68d6dc5c4-qdbkt 1/1 Running 0 1m

It will show two instances as we have set two replicas in our deployment.

For access of ASP.Net Core application:

1. Identify Worker Public IP by using ibmcloud cs workers YOUR\_CLUSTER\_NAME.
2. Identify the Node Port by using kubectl describe service get-started-aspnet.
3. Access your application at http://<WORKER-PUBLIC-IP>:<NODE-PORT>/.

In this way, deployment and access of application in the IKS environment can be done.

### **Clean up**

Following command will be used to clean up the sample application:

kubectl delete deployment,service -l app=get-started-aspnet

kubectl delete secret cloudant