Install the IBM Cloud command-line interface (CLI)

1. Install the IBM Cloud command-line interface. After it's installed, you can access IBM Cloud from your command line with the prefix ibmcloud.
2. Log in to the IBM Cloud CLI: ibmcloud login.
   For a federated ID: If you have a federated ID, use ibmcloud login –sso to log in to the IBM Cloud CLI. Enter your user name and use the provided URL in your CLI output to retrieve your one-time passcode. You know you have a federated ID if the login fails without the –sso and succeeds with the –sso option.

Install the IBM Cloud Kubernetes Service plug-in

1. Create Kubernetes clusters and manage worker nodes by installing the IBM Cloud Kubernetes Service plug-in:
   ibmcloud plugin install container-service -r Bluemix
   Tip: The prefix for running commands by using the IBM Cloud Kubernetes Service plug-in is ibmcloud cs.
2. Verify that the plug-in is installed properly:
   ibmcloud plugin list
   The plug-in is displayed in the results as container-service.

Install the IBM Cloud Container Registry plug-in

Use this plug-in to set up your own namespace in a multi-tenant, highly available, and scalable private image registry that is hosted by IBM, and to store and share Docker images with other users. Docker images are required to deploy containers into a cluster.

Tip: The prefix for running registry commands is ibmcloud cr.

1. Manage a private image repository by installing the IBM Cloud Container Registry plug-in:
   ibmcloud plugin install container-registry -r Bluemix
2. Verify that the plug-in is installed properly:
   ibmcloud plugin list
   The plug-in is displayed in the results as container-registry.

Install the Kubernetes CLI (kubectl)

Install the version of kubectl that is the same Kubernetes version for IBM Cloud Container Service. Using an older kubectl with a newer server might produce validation errors.

Install Docker

To locally build images and push them to your registry namespace, install Docker. The Docker CLI is used to build applications into images.

Tip: The prefix for running commands by using the Docker CLI is docker.

Containers

Containers provide isolation similar to virtual machines (VMs), except provided by the OS and at the process level. Each container is a process or group of processes that are run in isolation. Typical containers explicitly run only a single process because they have no need for the standard system services. What they usually need to do can be provided by system calls to the base OS kernel.

The isolation on Linux is provided by a feature called namespaces. Each different kind of isolation, that is, user and cgroups, is provided by a different namespace.

This is a list of some of the namespaces that are commonly used and visible to the user:

PID: process IDs
USER: user and group IDs
UTS: host name and domain name
NS: mount points
NET: network devices, stacks, and ports
CGROUPS: control limits and monitoring of resources

Virtual machines versus containers

Traditional applications are run on native hardware. A single app does not typically use the full resources of a single machine. Most organizations try to run multiple apps on a single machine to avoid wasting resources. You could run multiple copies of the same app, but to provide isolation, you can use VMs to run multiple app instances (VMs) on the same hardware. These VMs have full operating system stacks that make them relatively large and inefficient due to duplication both at runtime and on disk.

Containers allow you to share the host OS. This reduces duplication while still providing the isolation. Containers also allow you to drop unneeded files such as system libraries and binaries to save space and reduce your attack surface. If SSHD or LIBC are not installed, they cannot be exploited.

Kubernetes, open-source orchestration

Kubernetes is developed by the community with the intent of addressing container scaling and management needs. So in the early days of Kubernetes, the community contributors leveraged their knowledge of creating and running internal tools, such as Borg and Omega. With the advent of the Cloud Native Computing Foundation (CNCF), the community adopted Open Governance for Kubernetes. IBM, as a founding member of CNCF, actively contributes to CNCF's cloud-native projects, along with other companies such as Google, Red Hat, Microsoft, and Amazon.

Kubernetes architecture

At its core, Kubernetes is a data store (etcd). The declarative model is stored in the data store as objects, and that means when you say you want five instances of a container, then that request is stored into the data store. This information change is watched and delegated to Controllers to take action. Controllers then react to the model and attempt to take action to achieve the desired state. The power of Kubernetes is in its simplistic model.

In the following diagram, an API server is a simple HTTP server handling create/read/update/delete (CRUD) operations on the data store. The Controller identifies the change that you want and makes that happen.

Controllers are responsible for instantiating the actual resource represented by any Kubernetes resource. These actual resources are what your app needs to allow it to run successfully.

Kubernetes resource model

Kubernetes infrastructure defines a resource for every purpose. Each resource is monitored and processed by a Controller. When you define your app, it contains a collection of these resources. Controllers will then read this collection to build your app’s actual backing instances. Some of the resources that you might work with are described in the following list. For a complete list, see Kubernetes Concepts. In this course, you will only use a few of these resources, such as pod and deployment.

Config maps: holds configuration data for pods to consume
Daemon sets: ensures that each node in the cluster runs this pod
Deployments: defines a desired state of a deployment object
Events: provides life cycle events on pods and other deployment objects
Endpoints: allows an inbound connections to reach the cluster services
Ingress: a collection of rules that allows inbound connections to reach the cluster services
Jobs: creates one or more pods and when they complete successfully, the job is marked as completed
Node: a worker machine in Kubernetes
Namespaces: multiple virtual clusters backed by the same physical cluster
Pods: the smallest deployable units of computing that can be created and managed in Kubernetes
Persistent volumes: provides an API for users and administrators to abstract details about how storage is provided from how it is consumed

Replica sets: ensures that a specified number of pod replicas are running at any given time
Secrets: holds sensitive information, such as passwords, OAuth tokens, and SSH keys
Service accounts: provides an identity for processes that run in a pod
Services: an abstraction that defines a logical set of pods and a policy by which to access them, sometimes called a microservice
Stateful sets: the workload API object that manages stateful applications

Kubernetes does not have the concept of an app. It has simple building blocks that you are required to compose. Kubernetes is a cloud native platform where the internal resource model is the same as the end user resource model.

Key resources and pods

A pod is the smallest object model that you can create and run. You can add labels to a pod to identify a subset to run operations on. When you are ready to scale your app, you can use the label to tell Kubernetes which Pod you need to scale. A pod typically represents a process in your cluster. Pods contain at least one container that runs the job and additionally might have other containers in it called sidecars for monitoring, logging, and so on. Essentially, a pod is a group of containers.

An app is typically a group of pods. Although an entire app can be run in a single pod, you usually build multiple pods that talk to each other to make a useful app. In this course, you'll see why separating the app logic and back-end database into separate pods scales better when you build an app.

Kubernetes provides a client interface through the kubectl command-line interface. Kubectl commands allow you to manage your apps and manage cluster and cluster resources by modifying the model in the data store.

You directly manipulate resources through YAML, which is a human-readable serialization language:

```
$ kubectl (create|get|apply|delete) -f myResource.yaml
```

Kubernetes app deployment workflow

The following diagram shows how apps are deployed in a Kubernetes environment.

The user deploys a new app by using the kubectl CLI. Kubectl sends the request to the API server.
The API server receives the request and stores it in the data store (etcd). After the request is written to the data store, the API server is done with the request.
Watchers detect the resource changes and send notifications to the Controller to act on those changes.
The Controller detects the new app and creates new pods to match the desired number of instances. Any changes to the stored model will be used to create or delete pods.
The Scheduler assigns new pods to a node based on specific criteria. The Scheduler decides on whether to run pods on specific nodes in the cluster. The Scheduler modifies the model with the node information.
A Kubelet on a node detects a pod with an assignment to itself and deploys the requested containers through the container runtime, for example, Docker. Each node watches the storage to see what pods it is assigned to run. The node takes necessary actions on the resources assigned to it such as to create or delete pods.
Kubeproxy manages network traffic for the pods, including service discovery and load balancing. Kubeproxy is responsible for communication between pods that want to interact.

LAB 1 – SET UP AND DEPLOY YOUR FIRST APP

1. Push an image to IBM Cloud Container Registry

Push an image to IBM Cloud Container Registry

You use the IBM Cloud Container Registry to push or pull an image.

1. Clone or download the GitHub repository associated with this course.
2. Change directory to Lab 1:
   ```
   cd "Lab 1"
   ```
3. Log in to the IBM Cloud CLI. To specify an IBM Cloud region, include the API endpoint.
   If you have a federated ID, use `ibmcloud login --sso` to log in to the IBM Cloud CLI. You know you have a federated ID if the login fails without the `--sso` and succeeds with the `--sso` option.
   ```
   ibmcloud login
   ```
4. Run `ibmcloud cr login` and log in with your IBM Cloud credentials. This will allow you to push images to the IBM Cloud Container Registry.
   Tip: The commands in this lab show the `ng` region. Replace `ng` with the region outputted from the `ibmcloud cr login` command.
5. Create a namespace in IBM Cloud Container Registry where you can store your images:
   ```
   ibmcloud cr namespace-add <my_namespace>
   ```
6. Build the example Docker image:
   ```
   docker build --tag us.icr.io/<my_namespace>/hello-world .
   ```
7. Verify the image is built:
   ```
   docker images
   ```
8. Push that image to IBM Cloud Container Registry:
   ```
   docker push us.icr.io/<namespace>/hello-world
   ```
9. Make sure that the cluster you created previously is ready to use:
   - Run `ibmcloud cs clusters` and make sure that your cluster is in Normal state.
   - Run `ibmcloud cs workers <yourclustername>` and make sure that all workers are in Normal state with a Ready status.
   - Make a note of the public IP of the worker.

You are now ready to use Kubernetes to deploy the hello-world application.

Deploy your app

1. Get the command to set the environment variable and download the Kubernetes configuration files:
   ```
   ibmcloud cs cluster-config <cluster_name_or_ID>
   ```
   When the download of the configuration files is finished, a command is displayed that you can use to set the path to the local Kubernetes configuration file as an environment variable, for example, for OS X:
   ```
   export KUBECONFIG=/Users/<user_name>/.bluemix/plugins/container-
   service/clusters/pr_firm_cluster/kube-config-prod-par02-
   pr_firm_cluster.yml
   ```
   Copy and paste the command that is displayed in your terminal to set the KUBECONFIG environment variable.
2. Run your image as a deployment:
   ```
   kubectl run hello-world --image=us.icr.io/<namespace>/hello-
   world
   ```
   This action will take a bit of time. To check the status of your deployment, run this command:
   ```
   kubectl get pods
   ```
   You should see output like this:
   ```
       => kubectl get pods
       NAME                          READY     STATUS
   RESTARTS   AGE
       hello-world-562211614-0g2kd   0/1       ContainerCreating
   0          1m
   ```
3. When the status reads Running, expose that deployment as a service, which is accessed through the IP of the worker nodes. The example for this lab listens on port 8080. Run this command:

```
kubectl expose deployment/hello-world --type="NodePort" --
port=8080
```

4. Find the port that is used on that worker node and examine your new service:
   `kubectl describe service <name-of-deployment>`
   Take note of the "NodePort:" line as `<nodeport>`.
5. Run the following command and note the public IP as `<public-IP>`.
   `ibmcloud cs workers <name-of-cluster>`

You can now access your container/service by using curl <public-IP>:<nodeport> (or a web browser).
You're done if you see this:
"Hello world! Your app is up and running in a cluster!"

LAB 2- SCALE AND UPDATE APPS: SERVICES, REPLICA SETS, AND HEALTH CHECKS

1. Scale apps with replicas

Scale apps with replicas

A *replica* is how Kubernetes accomplishes scaling out a deployment. A replica is a copy of a pod that already contains a running service. By having multiple replicas of a pod, you can ensure your deployment has the available resources to handle increasing load on your app.

1. Update the replica set:
   `kubectl edit deployment/<name-of-deployment>`
2. Edit the YAML file in the code editor of your choice.
   This configuration YAML file is the configuration of your current deployment, which you can edit to customize the configuration for more fault tolerance.
   You should see a configuration similar to the this:

```
...
spec:
  replicas: 1
  selector:
    matchLabels:
      run: hello-world
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
    type: RollingUpdate
  template:
    metadata:
      creationTimestamp: null
      labels:
        run: hello-world
...
```

3. Change the replicas number from 1 to 10 so that the configuration now reads:

```
...
spec:
  replicas: 10
  selector:
    matchLabels:
      run: hello-world
  strategy:
  rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
    type: RollingUpdate
  template:
    metadata:
      creationTimestamp: null
      labels:
        run: hello-world
...
```

21. Save your changes and exit the editor. You now have 10 replicas of the app, which provides fault tolerance.
22. Watch your changes being rolled out:
    ```
    kubectl rollout status deployment/<name-of-deployment>
    ```
    The rollout might occur so quickly that the following messages might not display:
    ```
        => kubectl rollout status deployment/hello-world
        Waiting for rollout to finish: 1 of 10 updated replicas are
    available...
        Waiting for rollout to finish: 2 of 10 updated replicas are
    available...
        Waiting for rollout to finish: 3 of 10 updated replicas are
    available...
        Waiting for rollout to finish: 4 of 10 updated replicas are
    available...
        Waiting for rollout to finish: 5 of 10 updated replicas are
    available...
        Waiting for rollout to finish: 6 of 10 updated replicas are
    available...
        Waiting for rollout to finish: 7 of 10 updated replicas are
    available...
        Waiting for rollout to finish: 8 of 10 updated replicas are
    available...
        Waiting for rollout to finish: 9 of 10 updated replicas are
    available...
        deployment "hello-world" successfully rolled out
    ```
23. After the rollout is finished, ensure your pods are running:
    ```
    kubectl get pods
    ```
    You should see output listing the 10 pods of your deployment:
    ```
        => kubectl get pods
        NAME                           READY     STATUS    RESTARTS
    AGE
        hello-world-562211614-1tqm7   1/1       Running   0
    1d
        hello-world-562211614-1zqn4   1/1       Running   0
    2m
        hello-world-562211614-5htdz   1/1       Running   0
    2m
        hello-world-562211614-6h04h   1/1       Running   0
    2m
        hello-world-562211614-ds9hb   1/1       Running   0
    2m
        hello-world-562211614-nb5qp   1/1       Running   0
    2m
        hello-world-562211614-vtfp2   1/1       Running   0
    2m
        hello-world-562211614-vz5qw   1/1       Running   0
    2m
        hello-world-562211614-zksw3   1/1       Running   0
    2m
        hello-world-562211614-zsp0j   1/1       Running   0
    2m
    ```

2. Update and roll back apps

Kubernetes allows you to use a rollout to update an app deployment with a new Docker image. This allows you to easily update the running image and also allows you to easily undo a rollout if a problem is discovered after deployment.

Before you begin, ensure that you have the image tagged with `1` and pushed:

```
docker build --tag us.icr.io/<namespace>/hello-world:1 .
```

```
docker push us.icr.io/<namespace>/hello-world:1
```

To update and roll back:

1. Make a change to your code and build a new docker image with a new tag:
   `docker build --tag us.icr.io/<namespace>/hello-world:2 .`
2. Push the image to the IBM Cloud Container Registry:
   `docker push us.icr.io/<namespace>/hello-world:2`
3. Using kubectl, update your deployment to use the latest image. You can do this in two ways:
   - Edit the YAML file again by using `kubectl edit deployment/<name-of-deployment>`
   - Specify a new image by using a single command. Using a single command is especially useful when writing deployment automation. To specify the new image, run this command:
     `kubectl set image deployment/hello-world hello-world=us.icr.io/<namespace>/hello-world:2`
     A deployment can have multiple containers in which case each container will have its own name. Multiple containers can be updated at the same time. For more information about setting images, see the kubectl reference documentation.
4. Check he status of the rollout by running one of these commands:
   - `kubectl rollout status deployment/<name-of-deployment>`
   - `kubectl get replicasets`

   The rollout might occur so quickly that the following messages might not be displayed:

```
    => kubectl rollout status deployment/hello-world
    Waiting for rollout to finish: 2 out of 10 new replicas have been updated...
    Waiting for rollout to finish: 3 out of 10 new replicas have been updated...
    Waiting for rollout to finish: 3 out of 10 new replicas have been updated...
    Waiting for rollout to finish: 3 out of 10 new replicas have been updated...
    Waiting for rollout to finish: 4 out of 10 new replicas have been updated...
    Waiting for rollout to finish: 4 out of 10 new replicas have been updated...
    Waiting for rollout to finish: 4 out of 10 new replicas have been updated...
    Waiting for rollout to finish: 4 out of 10 new replicas have been updated...
    Waiting for rollout to finish: 4 out of 10 new replicas have been updated...
    Waiting for rollout to finish: 5 out of 10 new replicas have been updated...
    Waiting for rollout to finish: 5 out of 10 new replicas have been updated...
    Waiting for rollout to finish: 5 out of 10 new replicas have been updated...
    Waiting for rollout to finish: 6 out of 10 new replicas have been updated...
    Waiting for rollout to finish: 6 out of 10 new replicas have been updated...
    Waiting for rollout to finish: 6 out of 10 new replicas have been updated...
    Waiting for rollout to finish: 7 out of 10 new replicas have been updated...
    Waiting for rollout to finish: 7 out of 10 new replicas have been updated...
    Waiting for rollout to finish: 7 out of 10 new replicas have been updated...
    Waiting for rollout to finish: 7 out of 10 new replicas have
been updated...
    Waiting for rollout to finish: 8 out of 10 new replicas have
been updated...
    Waiting for rollout to finish: 8 out of 10 new replicas have
been updated...
    Waiting for rollout to finish: 8 out of 10 new replicas have
been updated...
    Waiting for rollout to finish: 8 out of 10 new replicas have
been updated...
    Waiting for rollout to finish: 9 out of 10 new replicas have
been updated...
    Waiting for rollout to finish: 9 out of 10 new replicas have
been updated...
    Waiting for rollout to finish: 9 out of 10 new replicas have
been updated...
    Waiting for rollout to finish: 1 old replicas are pending
termination...
    Waiting for rollout to finish: 1 old replicas are pending
termination...
    Waiting for rollout to finish: 1 old replicas are pending
termination...
    Waiting for rollout to finish: 9 of 10 updated replicas are
available...
    Waiting for rollout to finish: 9 of 10 updated replicas are
available...
    Waiting for rollout to finish: 9 of 10 updated replicas are
available...
    deployment "hello-world" successfully rolled out
    => kubectl get replicasets
    NAME                    DESIRED   CURRENT   READY     AGE
    hello-world-1663871401   9         9         9         1h
    hello-world-3254495675   2         2         0
<invalid>
    => kubectl get replicasets
    NAME                    DESIRED   CURRENT   READY     AGE
    hello-world-1663871401   7         7         7         1h
    hello-world-3254495675   4         4         2
<invalid>
    ...
    => kubectl get replicasets
    NAME                    DESIRED   CURRENT   READY     AGE
    hello-world-1663871401   0         0         0         1h
    hello-world-3254495675   10        10        10        1m
```

5. Confirm that your new code is active:
   `curl <public-IP>:<nodeport>`
6. **Optional**: Undo your latest rollout:
   `kubectl rollout undo deployment/<name-of-deployment>`

3. Check the health of apps

Check the health of apps

Kubernetes uses availability checks (liveness probes) to know when to restart a container. For example, liveness probes can catch a deadlock where an app is running but be unable to make progress. Restarting a container in such a state can help to make the app more available despite the bugs.

Also, Kubernetes uses readiness checks to know when a container is ready to start accepting traffic. A pod is considered ready when all of its containers are ready. One use of this check is to control which pods are used as back ends for services. When a pod is not ready, it is removed from the load balancers.

In this example, an HTTP liveness probe is defined to check health of the container every five seconds. For the first 10 - 15 seconds, the `/healthz` returns a `200` response and will fail afterward. Kubernetes will automatically restart the service.

1. Open the `<username_home_directory>/container-service-getting-started-wt/Lab 2/healthcheck.yml` file with a text editor. This configuration script combines a few steps from the previous lesson to create a deployment and a service at the same time. App developers can use these scripts when updates are made or to troubleshoot issues by re-creating the pods. Make these changes in the file:
   - Update the details for the image in your private registry namespace: `image: "us.icr.io/<namespace>/hello-world:2"`
   - Note the HTTP liveness probe that checks the health of the container every five seconds:
   - `livenessProbe:`
   - `httpGet:`
   - `path: /healthz`
   - `port: 8080`
   - `initialDelaySeconds: 5`
   - `periodSeconds: 5`
   - In the `Service` section, note the `NodePort`. Rather than generating a random NodePort like you did previously, you can specify a port between 30000 – 32767. This example uses port 30072.
2. Run the configuration script in the cluster. When the deployment and the service are created, the app is available for anyone to see:
3. `kubectl apply -f <username_home_directory>/container-service-getting-started-wt/Lab 2/healthcheck.yml`
   Now that all the deployment work is done, check how everything turned out. You might notice that because more instances are running, things might run a bit slower.
4. Open a browser and check the app. To form the URL, combine the IP with the NodePort that was specified in the configuration script. To get the public IP address for the worker node, run this command:
5. `ibmcloud cs workers <cluster-name>`
   In a browser, you'll see a success message. If you do not see this message, don't worry. This app is designed to go up and down.
   For the first 10 – 15 seconds, a 200 message is returned, so you know that the app is running successfully. After those 15 seconds, a timeout message is displayed as is designed in the app.
6. Launch your Kubernetes dashboard from the Clusters page in the IBM Cloud Kubernetes Service GUI.
   In the **Workloads** tab, you can see the resources that you created. From this tab, you can continually refresh and see that the health check is working.
   In the Pods section, you can see how many times the pods are restarted when the containers in them are re-created.
   You might happen to catch errors in the dashboard, indicating that the health check caught a problem. Wait a few minutes and refresh again. You see the number of restart changes for each pod.

Congratulations! You deployed the second version of the app. You had to use fewer commands, learned how health check works, and edited a deployment, which is great!

LAB 3- DEPLOY AN APP WITH WATSON SERVICES

Deploy the Watson app

1. Log in to IBM Cloud Container Registry:
   ```
   ibmcloud cr login
   ```
2. Change the directory to `Lab 3/watson`.
3. Build the `watson` image:
   ```
   docker build -t us.icr.io/<namespace>/watson ./watson
   ```
4. Push the `watson` image to IBM Cloud Container Registry:
   ```
   docker push us.icr.io/<namespace>/watson
   ```

Tip: If you run out of registry space, remove the previous lab's images with this example command:
`ibmcloud cr image-rm us.icr.io/<namespace>/hello-world:2`

5. Change the directory to `Lab 3/watson-talk`.
6. Build the `watson-talk` image:
   `docker build -t /<namespace>/watson-talk ./watson-talk`
7. Push the `watson-talk` image to IBM Cloud Container Registry:
   `docker push us.icr.io/<namespace>/watson-talk`
8. In the watson-deployment.yml file, update the image tag with the registry path to the image that you created in the following two sections:

```
    spec:
      containers:
        - name: watson
          image: "us.icr.io/<namespace>/watson"
          # change to the path of the watson image you just pushed
          # ex: image: "http://us.icr.io//<namespace>/watson"
...
    spec:
      containers:
        - name: watson-talk
          image: "us.icr.io/<namespace>/watson-talk"
          # change to the path of the watson-talk image you just
pushed
          # ex: image: "us.icr.io/<namespace>/watson-talk"
```

2. Create an instance of the IBM Watson service via the CLI

Create an instance of the IBM Watson service via the CLI

To use the Watson Tone Analyzer (the IBM Cloud service for this app), you must first request an instance of the Watson service in the org and space where you set up your cluster.

If you need to check what space and org you are currently using, run `ibmcloud login`. Then, use `ibmcloud target --cf` to select the space and org you used for the previous labs.

1. After you set your space and org, run `ibmcloud cf create-service tone_analyzer standard tone`, where `tone` is the name you will use for the Watson Tone Analyzer service.
   Tip: When you add the Tone Analyzer service to your account, you might see a message saying that the service is not free. If you limit your API calls, this course does not incur charges from the Watson service.
2. Ensure the service `tone` was created:
   `ibmcloud cf services`

3. Bind the Watson service to your cluster

Bind the Watson service to your cluster

1. Bind the service to your cluster. This command will create a secret for the service.
   `ibmcloud cs cluster-service-bind <name-of-cluster> default tone`
2. Verify the secret was created:
   `kubectl get secrets`

4. Create pods and services

Create pods and services

Now that the service is bound to the cluster, you must expose the secret to your pod so that it can use the service. To do this, create a secret data store as a part of your deployment configuration. This has been done for you in the file watson-deployment.yml:

```
    volumeMounts:
            - mountPath: /opt/service-bind
              name: service-bind-volume
      volumes:
        - name: service-bind-volume
          secret:
            defaultMode: 420
            secretName: 2a5baa4b-a52d-4911-9019-69ac01afbb7f-key0
            # from the kubectl get secrets command above
```

1. Build the app by using the YAML file:

  - `cd "Lab 3"`
  - `kubectl create -f watson-deployment.yml`

    This command exposes the secret to your pod so that it can use the service.

2. Verify the pod was created:

   ```
   kubectl get pods
   ```

5. Putting it all together: Run the app and service

Run the app and service

So far, you've created pods, services, and volumes in the previous labs. The last step is run the app and the Tone Analyzer service.

1. Open the Kubernetes dashboard and explore the new objects or use the following commands:
2. `kubectl get pods`
3. `kubectl get deployments`
4. `kubectl get services`
5. Get the public IP for the worker node to access the app:
6. `ibmcloud cs workers <name-of-cluster>`
7. Now that the you have the container IP and port, open a web browser and enter the following URL to analyze the text and see output:

   ```
   http://<public-IP>:30080/analyze/"Today is a beautiful day"
   ```

If you can see JSON output, congratulations! You are done with this lab!

DOCUMENT

An introduction to containers

Hey, are you looking for a containers 101 course? Check out our Docker Essentials.

Containers allow you to run securely isolated applications with quotas on system resources. Containers started out as an individual feature delivered with the linux kernel. Docker launched with making containers easy to use and developers quickly latched onto that idea. Containers have also sparked an interest in microservice architecture, a design pattern for developing applications in which complex applications are broken down into smaller, composable pieces which work together.

Watch this video to learn about production uses of containers.

Objectives

This lab is an introduction to using containers on Kubernetes in the IBM Cloud Kubernetes Service. By the end of the course, you'll achieve these objectives:

- Understand core concepts of Kubernetes
- Build a container image and deploy an application on Kubernetes in the IBM Cloud Kubernetes Service
- Control application deployments, while minimizing your time with infrastructure management
- Add AI services to extend your app
- Secure and monitor your cluster and app

Prerequisites

- A Pay-As-You-Go or Subscription IBM Cloud account

Virtual machines

Prior to containers, most infrastructure ran not on bare metal, but atop hypervisors managing multiple virtualized operating systems (OSes). This arrangement allowed isolation of applications from one another on a higher level than that provided by the OS. These virtualized operating systems see what looks like their own exclusive hardware. However, this also means that each of these virtual operating systems are replicating an entire OS, taking up disk space.

Containers

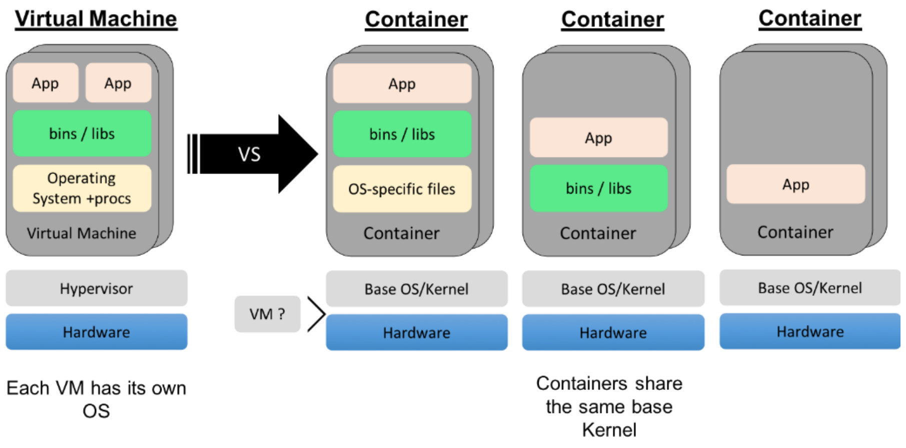
Containers provide isolation similar to VMs, except provided by the OS and at the process level. Each container is a process or group of processes run in isolation. Typical containers explicitly run only a single process, as they have no need for the standard system services. What they usually need to do can be provided by system calls to the base OS kernel.

The isolation on linux is provided by a feature called 'namespaces'. Each different kind of isolation (IE user, cgroups) is provided by a different namespace.
This is a list of some of the namespaces that are commonly used and visible to the user:

- PID - process IDs
- USER - user and group IDs
- UTS - hostname and domain name
- NS - mount points
- NET - network devices, stacks, and ports
- CGROUPS - control limits and monitoring of resources

VM vs container

Traditional applications are run on native hardware. A single application does not typically use the full resources of a single machine. We try to run multiple applications on a single machine to avoid wasting resources. We could run multiple copies of the same application, but to provide isolation we use VMs to run multiple application instances (VMs) on the same hardware. These VMs have full operating system stacks which make them relatively large and inefficient due to duplication both at runtime and on disk.



Containers allow you to share the host OS. This reduces duplication while still providing the isolation. Containers also allow you to drop unneeded files such as system libraries and binaries to save space and reduce your attack surface. If SSHD or LIBC are not installed, they cannot be exploited.

Get set up

Before we dive into Kubernetes, you need to provision a cluster for your containerized app. Then you won't have to wait for it to be ready for the subsequent labs.

1. You must install the CLIs per https://cloud.ibm.com/docs/containers/cs_cli_install.html. If you do not yet have these CLIs and the Kubernetes CLI, do [lab 0]() before starting the course.
2. If you haven't already, provision a cluster. This can take a few minutes, so let it start first: `ibmcloud ks cluster create classic --name <name-of-cluster>`
3. After creation, before using the cluster, make sure it has completed provisioning and is ready for use. Run `ibmcloud ks clusters` and make sure that your cluster is in state "deployed".
4. Then use `ibmcloud ks workers --cluster <name-of-cluster>` and make sure that all worker nodes are in state "normal" with Status "Ready".

Kubernetes and containers: an overview

Let's talk about Kubernetes orchestration for containers before we build an application on it. We need to understand the following facts about it:

- What is Kubernetes, exactly?
- How was Kubernetes created?
- Kubernetes architecture
- Kubernetes resource model
- Kubernetes at IBM
- Let's get started

What is Kubernetes?

Now that we know what containers are, let's define what Kubernetes is. Kubernetes is a container orchestrator to provision, manage, and scale applications. In other words, Kubernetes allows you to manage the lifecycle of containerized applications within a cluster of nodes (which are a collection of worker machines, for example, VMs, physical machines etc.).

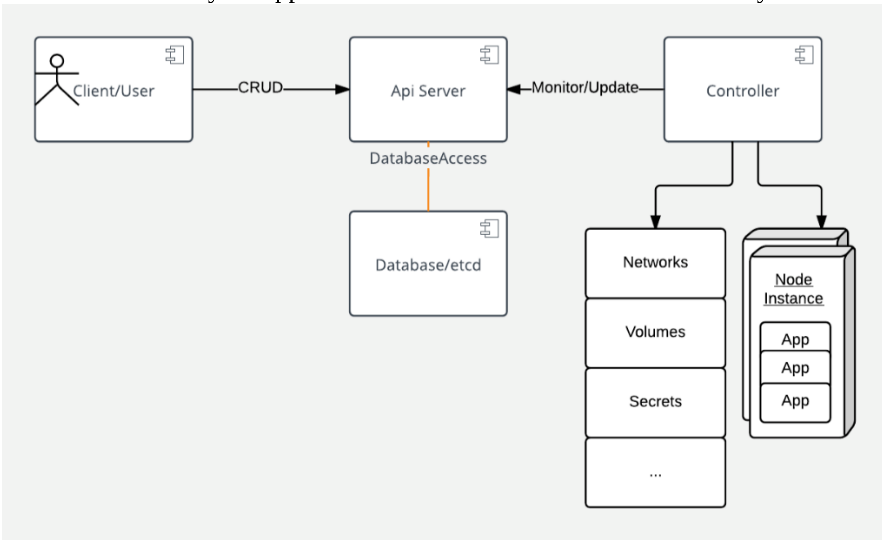
Your applications may need many other resources to run such as Volumes, Networks, and Secrets that will help you to do things such as connect to databases, talk to firewalled backends, and secure keys. Kubernetes helps you add these resources into your application. Infrastructure resources needed by applications are managed declaratively.
Fast fact: Other orchestration technologies are Mesos and Swarm.
The key paradigm of kubernetes is it's Declarative model. The user provides the "desired state" and Kubernetes will do it's best make it happen. If you need 5 instances, you do not start 5 separate instances on your own but rather tell Kubernetes that you need 5 instances and Kubernetes will reconcile the state automatically. Simply at this point you need to know that you declare the state you want and Kubernetes makes that happen. If something goes wrong with one of your instances and it crashes, Kubernetes still knows the desired state and creates a new instances on an available node.
Fun to know: Kubernetes goes by many names. Sometimes it is shortened to *k8s* (losing the internal 8 letters), or *kube*. The word is rooted in ancient Greek and means "Helmsman". A helmsman is the person who steers a ship. We hope you can seen the analogy between directing a ship and the decisions made to orchestrate containers on a cluster.

How was Kubernetes created?

Google wanted to open source their knowledge of creating and running the internal tools Borg & Omega. It adopted Open Governance for Kubernetes by starting the Cloud Native Computing Foundation (CNCF) and giving Kubernetes to that foundation, therefore making it less influenced by Google directly. Many companies such as RedHat, Microsoft, IBM and Amazon quickly joined the foundation.
Main entry point for the kubernetes project is at https://kubernetes.io/ and the source code can be found at https://github.com/kubernetes.

Kubernetes architecture

At its core, Kubernetes is a data store (etcd). The declarative model is stored in the data store as objects, that means when you say I want 5 instances of a container then that request is stored into the data store. This information change is watched and delegated to Controllers to take action. Controllers then react to the model and attempt to take action to achieve the desired state. The power of Kubernetes is in its simplistic model.
As shown, API server is a simple HTTP server handling create/read/update/delete(CRUD) operations on the data store. Then the controller picks up the change you wanted and makes that happen. Controllers are responsible for instantiating the actual resource represented by any Kubernetes resource. These actual resources are what your application needs to allow it to run successfully.

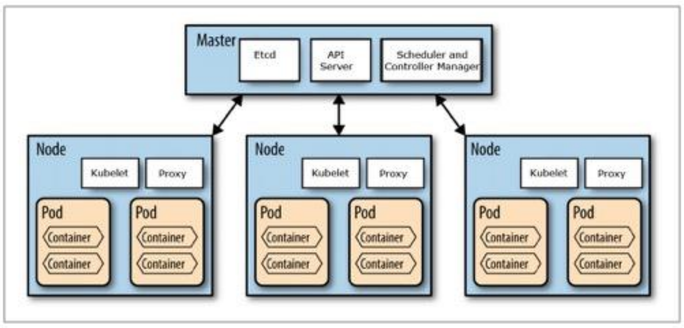


Kubernetes resource model

Kubernetes Infrastructure defines a resource for every purpose. Each resource is monitored and processed by a controller. When you define your application, it contains a collection of these resources. This collection will then be read by Controllers to build your applications actual backing instances. Some of resources that you may work with are listed below for your reference, for a full list you should go to https://kubernetes.io/docs/concepts/. In this class we will only use a few of them, like Pod, Deployment, etc.

- Config Maps holds configuration data for pods to consume.
- Daemon Sets ensure that each node in the cluster runs this Pod
- Deployments defines a desired state of a deployment object

- Events provides lifecycle events on Pods and other deployment objects
- Endpoints allows a inbound connections to reach the cluster services
- Ingress is a collection of rules that allow inbound connections to reach the cluster services
- Jobs creates one or more pods and as they complete successfully the job is marked as completed.
- Node is a worker machine in Kubernetes
- Namespaces are multiple virtual clusters backed by the same physical cluster
- Pods are the smallest deployable units of computing that can be created and managed in Kubernetes
- Persistent Volumes provides an API for users and administrators that abstracts details of how storage is provided from how it is consumed
- Replica Sets ensures that a specified number of pod replicas are running at any given time
- Secrets are intended to hold sensitive information, such as passwords, OAuth tokens, and ssh keys
- Service Accounts provides an identity for processes that run in a Pod
- Services is an abstraction which defines a logical set of Pods and a policy by which to access them - sometimes called a micro-service.
- Stateful Sets is the workload API object used to manage stateful applications.
- and more...



Kubernetes does not have the concept of an application. It has simple building blocks that you are required to compose. Kubernetes is a cloud native platform where the internal resource model is the same as the end user resource model.

Key resources

A Pod is the smallest object model that you can create and run. You can add labels to a pod to identify a subset to run operations on. When you are ready to scale your application you can use the label to tell Kubernetes which Pod you need to scale. A Pod typically represent a process in your cluster. Pods contain at least one container that runs the job and additionally may have other containers in it called sidecars for monitoring, logging, etc. Essentially a Pod is a group of containers.

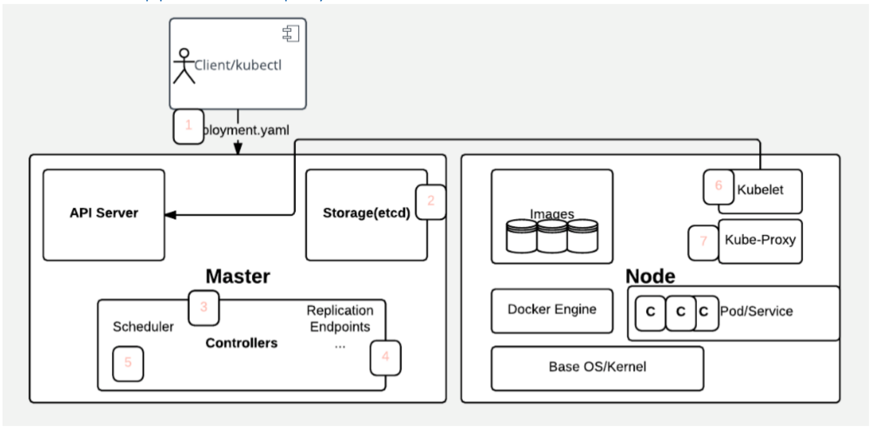
When we talk about a application, we usually refer to group of Pods. Although an entire application can be run in a single Pod, we usually build multiple Pods that talk to each other to make a useful application. We will see why separating the application logic and backend database into separate Pods will scale better when we build an application shortly.

Services define how to expose your app as a DNS entry to have a stable reference. We use query based selector to choose which pods are supplying that service.

The user directly manipulates resources via yaml: `$ kubectl (create|get|apply|delete) -f myResource.yaml`

Kubernetes provides us with a client interface through ‘kubectl’. Kubectl commands allow you to manage your applications, manage cluster and cluster resources, by modifying the model in the data store.

Kubernetes application deployment workflow



1. User via "kubectl" deploys a new application. Kubectl sends the request to the API Server.
2. API server receives the request and stores it in the data store (etcd). Once the request is written to data store, the API server is done with the request.
3. Watchers detects the resource changes and send a notification to controller to act upon it
4. Controller detects the new app and creates new pods to match the desired number# of instances. Any changes to the stored model will be picked up to create or delete Pods.
5. Scheduler assigns new pods to a Node based on a criteria. Scheduler makes decisions to run Pods on specific Nodes in the cluster. Scheduler modifies the model with the node information.
6. Kubelet on a node detects a pod with an assignment to itself, and deploys the requested containers via the container runtime (e.g. Docker). Each Node watches the storage to see what pods it is assigned to run. It takes necessary actions on resource assigned to it like create/delete Pods.
7. Kubeproxy manages network traffic for the pods – including service discovery and load-balancing. Kubeproxy is responsible for communication between Pods that want to interact.

Lab information

IBM Cloud provides the capability to run applications in containers on Kubernetes. The IBM Cloud Kubernetes Service runs Kubernetes clusters which deliver the following:

- Powerful tools
- Intuitive user experience
- Built-in security and isolation to enable rapid delivery of secure applications
- Cloud services including cognitive capabilities from Watson
- Capability to manage dedicated cluster resources for both stateless applications and stateful workloads

Lab overview

Lab 0 (Optional): Provides a walkthrough for installing IBM Cloud command-line tools and the Kubernetes CLI. You can skip this lab if you have the IBM Cloud CLI, the container-service plugin, the containers-registry plugin, and the kubectl CLI already installed on your machine.
Lab 1: This lab walks through creating and deploying a simple "hello world" app in Node.JS, then accessing that app.
Lab 2: Builds on lab 1 to expand to a more resilient setup which can survive having containers fail and recover. Lab 2 will also walk through basic services you need to get started with Kubernetes and the IBM Cloud Kubernetes Service
Lab 3: This lab covers adding external services to a cluster. It walks through adding integration to a Watson service, and discusses storing credentials of external services to the cluster.
Lab 4 (Under Construction, Paid Only, Optional): This lab will outline how to create a highly available application, and build on the knowledge you have learned in Labs 1 - 3 to deploy clusters simultaneously to multiple availability zones. As this requires a paid IBM Cloud account, skip this lab if you are sticking to the free tier.
Lab 5: This lab walks through securing your cluster and applications using network policies, and will later add leveraging tools like Vulnerability Advisor to secure images and manage security in your image registry.