ORACLE
University

Oracle Cloud Infrastructure Developer Professional

Activity Guide

S105662GC10

Learn more from Oracle University at ***education.oracle.com***

Table of Contents

Cloud Native Fundamentals: Explore Developer Environment in OCI

Lab 1-1 Practices

Estimated Time: 30 minutes

Setup Instructions for the Lab

IMPORTANT: If you are using a tenancy in which you do not have administrator permissions, you will need to ensure that an administrator has added you to a group and that group has been provided permissions as indicated below:

Policies

Root Level

```
        Allow group <groupname> to use cloud-shell in tenancy
```

Note: If you are using your own tenancy, simply use your initial tenancy administrator credentials to perform the exercises. In that case you will not need to add any group permission policy statements.

Get Started

Overview

In this lab exercise, you will log into your OCI tenancy and explore the OCI Console, Cloud Shell, and Code Editor. In addition, you will discover and document some key information related to your environment to include the Oracle Cloud Identifier (OCID) values of certain resources. This lab will be segmented into two practices:

a. Explore the OCI Console, locate OCIDs, and generate Auth Token.

b. Launch Cloud Shell and Code Editor, and clone a Remote Repository.

Prerequisites

An OCI tenancy and user credentials.

Explore the OCI Console, Locate OCIDs, and Generate an Auth Token

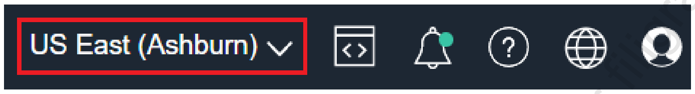
In this practice, you will log in to your own OCI tenancy, and then navigate to various pages capturing information about your user info, compartment, and region. You will also generate an authentication token, which will be used as your credential for subsequent practices.

Tasks

1. Sign in to the Oracle Cloud Infrastructure (OCI) console by using your user credentials.

2. You will need to keep track of various resource names and identifiers throughout the practices. Open a text editor (such as Notepad) to capture the following values:
   - **tenancy-name**: (i.e., mytenancyname)
   - **oci-username**: (i.e., myusername)
   - **compartment-name**: (i.e., my-compartment) ***Note***: *If you are not already using a dedicated compartment, you should create a new one to use for these practices.*

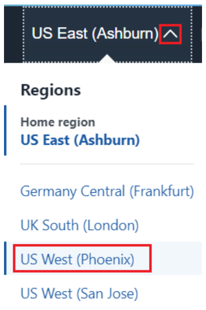
3. The current region is shown in the pane at the top right as shown here:



4. If necessary, use the drop-down menu to select the region you will be using for these exercises.

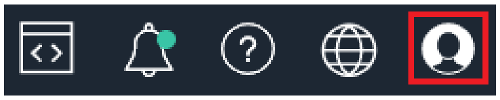
   For example – changing to **US West (Phoenix)**.

   NOTE: It is critical that you use only one region for all subsequent lab practices throughout this course.



5. After you have confirmed and selected the region you will be using for this training, open [this web page]() to locate your region, and then copy the following values to your notepad:
   - **Region Identifier**: (i.e., us-phoenix-1)
   - **Region Key**: (i.e., PHX)

6. Open the Profile menu located in the top-right pane.



   Click the link for your **Tenancy** which will open the **Tenancy Details** page.

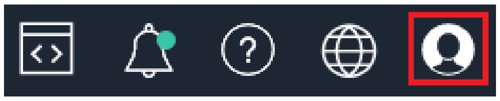
7. Under **Tenancy information**, copy the **OCID** and **Object storage namespace** values:

- **tenancy-OCID**: (ocid1.tenancy.oc1..aaaa…)
- **namespace**: (i.e., ansh81vru1zp)

NOTE: Each OCI tenancy is assigned one unique and uneditable ***namespace*** used for Object Storage and other services such as the Container Registry. The namespace is a system-generated string assigned during account creation. For some older tenancies, the namespace string may be the same as the tenancy ***name*** in all lower-case letters.

IMPORTANT: Throughout these lab practices, it is important that you DO NOT confuse tenancy ***name*** with ***namespace***. The **Namespace** value captured above will be used later in this course as the namespace value for both Object Storage and Container Registry (OCIR).

8. Once again, open the Profile menu located in the top-right pane.



Click your username link, which will open the **User Details** page.

9. Under **User information**, copy the OCID value.
   - **user-OCID**: (ocid1.user.oc1..aaaa…)

Later in this course, you will need to provide your credentials in the form of an authentication token when logging into the Container Registry (OCIR).

10. Under resources, click the **Auth Tokens** link, then under **Auth Tokens**, click **Generate Token**.

11. Enter `mytoken` as a description for your auth token.

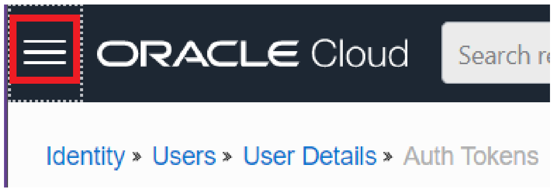
12. Click **Generate Token**. When you click the **Show** link, the auth token value is displayed.

    IMPORTANT: Copy the generated token value because you won't see the auth token again in the console. You'll need this auth token value in subsequent practices.
    - **User Auth Token:** (i.e., .i5ki95<3-c;QS(Tyw#!)

13. Close the **Generate Token** dialog box.

14. Click the three dots at the far right of the **mytoken** entry to open the Actions menu and click **Copy OCID**. Add that value to your text file.
    - **Auth Token OCID:** (ocid1.credential.oc1..aaaa…)

15. Click the menu icon in the top-left corner:



Click **Identity & Security**, then under **Identity**, click **Compartments**.

16. Locate and click the link of the compartment you will be using for these exercises to open the **Compartment Details** page. ***Note***: *If you are not already using a dedicated compartment, you should create a new one to use for these lab practices.*

    In the **Compartment Information** pane, locate and copy the **OCID** for your compartment.

    - **Compartment OCID:** (ocid1.compartment.oc1..aaaa…)

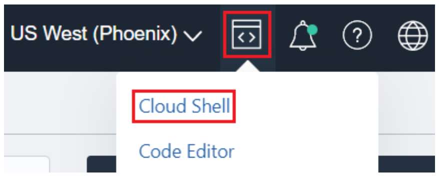
Launch Cloud Shell and Code Editor; Clone a Remote Repository

Oracle Cloud Infrastructure (OCI) **Cloud Shell** is a web browser-based terminal accessible from the Oracle Cloud Console. Cloud Shell provides access to a Linux shell with a pre-authenticated Oracle Cloud Infrastructure CLI, a pre-authenticated Ansible installation, and many other useful tools as you will discover during the practices.

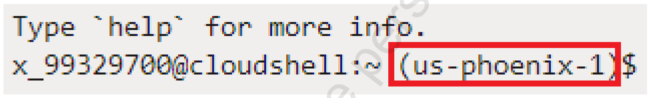
Oracle Cloud Infrastructure (OCI) **Code Editor** provides a rich, in-console editing environment that enables you to edit code and update service workflows and scripts without having to switch between the Console and your local development environment. Code Editor provides a convenient way to perform common code updates for various services, such as creating and deploying Functions, editing Terraform configurations used with Resource Manager stacks, or creating and editing an API.

Tasks

1. Launch Cloud Shell by clicking the **Developer Tools** icon and then the **Cloud Shell** link located in the top-right pane.

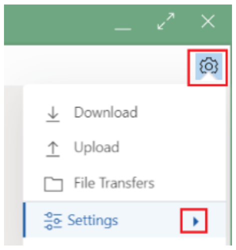


   Note: When you start Cloud Shell, the service configures your session with the currently selected region in the Console. In the default bash prompt, the region that the OCI CLI is interacting with is echoed in the Cloud Shell command line prompt.



```
Type `help` for more info.
x_99329700@cloudshell:~ (us-phoenix-1)$
```

2. Optionally, to change any settings, click the **Cloud Shell Menu** gear icon located at the far right, then click **Settings**. For example, you can change the theme from Light to Dark.



3. Execute a few OCI CLI commands:

   `$ oci -v` (the current CLI version)

   `$ oci os ns get` (the namespace value for Object Storage and OCIR)

   `$ oci iam region list --output table` (list of OCI regions)

4. Create a new directory called **labs** and then navigate to that new subdirectory.

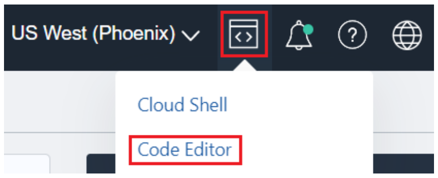
```
$ mkdir labs
$ cd labs
```

5. Clone the GitHub repository that contains the resources to be used in later practices.

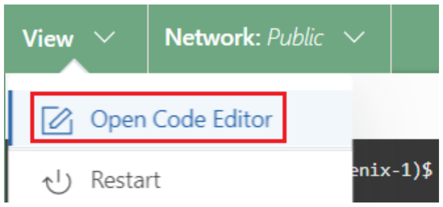
```
$ git clone https://github.com/ou-developers/oci-functions
```

To view or edit files (in addition to Cloud Shell), you also have the option to use **Code Editor**.

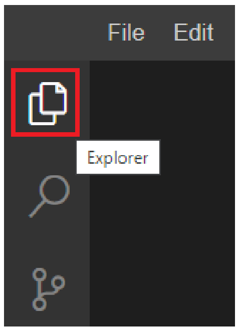
6. Open **Code Editor** using the menu icon (as shown below):

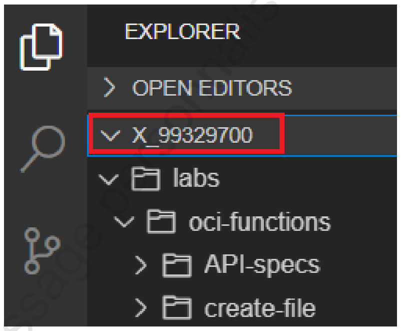


…Or you can launch from the **View** drop-down menu in Cloud Shell.



7. To view the files from the cloned repository, click the **Explorer** icon. Then click to expand your home directory labeled with your partial OCI username.

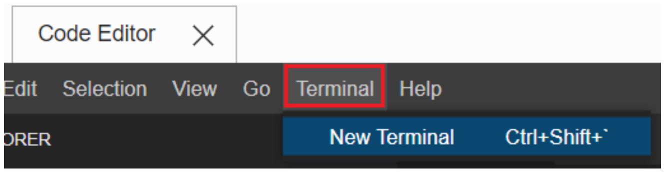




Locate any file, and then click to view its content.

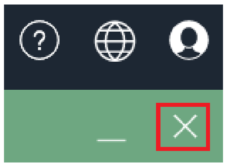
Note: Do not modify the contents of any file until directed in subsequent practice tasks.

8. You can also open additional terminal sessions using from Code Editor Menu bar.



Alternatively, you can open a terminal at a specific directory location by right-clicking any subfolder in the Explorer and select **Open in Terminal**.

9. Terminate your Code Editor and Cloud Shell sessions by clicking the **X** in the upper right-hand corner.



Congratulations! You have captured important resource identifiers, and created an Auth Token for your OCI User. You have also explored OCI Cloud Shell and Code Editor and have cloned a GitHub repository containing files that you'll use in subsequent lab practices.

Container-Based Application Development: Create a Docker Image for a Web Application

Lab 2-1 Practices

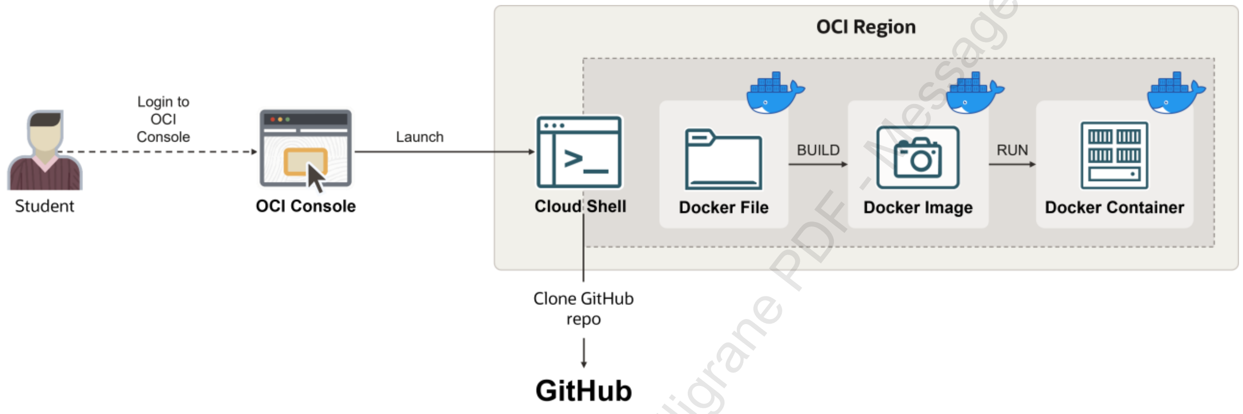
Estimated Time: 30 minutes

Get Started

Overview

There are certain ways for creating, running, and deploying applications in containers using Docker. A Docker image contains application code, libraries, tools, dependencies, and other files needed to make that application run.

In this practice, you will create a Docker image using a Dockerfile, which will further be used to build a container that can run on the Docker platform.



The practices in this lab include:

a. Access the Dockerfile.

b. Build the Docker image.

c. Run your Docker image as a container.

d. Access the web application running within the container.

Prerequisites

You have completed **Lab 1-1** (*Cloud Native Fundamentals: Explore Developer Environment in OCI*).

Assumptions

- You are signed into your Oracle Cloud Infrastructure (OCI) account using your credentials.
- You will replace the `<userID>` placeholder with your first name.

Access the Dockerfile

Retrieve the Dockerfile needed to generate the Docker image by cloning a Git repository in GitHub.

Tasks

1. Open **Cloud Shell** by clicking the icon and then the link at the right of the OCI console header.

   Note: The OCI CLI running in the Cloud Shell will execute commands against the region selected in the console's region selection menu when the Cloud Shell was started.

2. Navigate to the **lab's** directory.

   ```
   $ cd ~/labs
   ```

3. Clone the GitHub repository to your Cloud Shell VM, which contains a simple Nginx HelloWorld application that you will use to build the Docker image.

   ```
   $ git clone https://github.com/ou-developers/docker-helloworld-demo
   ```

4. Open **Code Editor**. Browse to the cloned Git repo folder (**docker-helloworld-demo**) to view the various files in the directory including application code and a Dockerfile for creating the sample nginx application.

Build the Docker Image

You’re using Cloud Shell as your development environment, which comes preinstalled with Docker.

Tasks

1. Back in the Cloud Shell terminal, navigate to the cloned directory.

```
$ cd ~/labs/docker-helloworld-demo
```

2. Check the Docker version using the following command in Cloud Shell. It will return a string with the Docker version installed.

```
$ docker -v
```

   For example: `Docker version 19.03.11-ol, build 9bb540d`

3. Check for existing Docker images in the Cloud Shell.

```
$ docker images
```

   Note: It will return an empty response as there are no images at present.

4. Create a Docker image for the sample web application using `docker build` command. This command looks for a Dockerfile within the provided PATH location.

```
$ docker build -t oci_sample_webapp_<userID>:<tag> .
```

   Where:

   - `-t` is the switch used to specify the image name.
   - Enter an image name using this format: `oci_sample_webapp_<userID>`
     - Replace *<userID>* with your first name.

       For example: `oci_sample_webapp_david`
   - A tag is used to give the image a version. Use the following tag: `1.0`
   - Use a period “**.**” as PATH at the end of the command because you are currently in the cloned directory, which contains the Dockerfile.

   For example:

```
$ docker build -t oci_sample_webapp_david:1.0 .
```

5. Upon successful build of a Docker image, verify the image in the local repository using the following command:

```
$ docker images
```

You’ll see two entries. One is the base image, and the other is the custom Docker image for the web application.

Run Your Docker Image as a Container

Your Docker image holds the application that you want Docker to run as a container.

Tasks

1. Use the `docker run` command to launch a container based on the created image.

   ```
    $ docker run -d --name webapp_<userID> -p 80:80/tcp
   oci_sample_webapp_<userID>:<tag>
   ```

   Where:

   - `-d` flag is used to run the container in the background
   - `--name` is used to assign a name to the container
   - `-p` flag is used to publish container port 80 to the host machine port 80

   For example:

   ```
    $ docker run -d --name webapp_david -p 80:80/tcp
   oci_sample_webapp_david:1.0
   ```

2. Check the container that is currently running using the `docker ps` command.

   ```
    $ docker ps
   ```

   You will see a running container with the name `webapp_<userID>`.

Access the Web Application Running Within the Container

Verify if you can access the web application that's running in your container. After you verify, stop the running container, then remove the container.

Tasks

1. Use the **curl** command to connect to the local host on port 80 to access the web application.

   ```
   $ curl -k http://127.0.0.1:80
   ```

   The output displays the webpage code, confirming that your web application is up and running.

2. Retrieve the container ID and copy it to use in your next step.

   ```
   $ docker ps
   ```

3. Stop the running container.

   ```
   $ docker stop <CONTAINER ID>
   ```

   For example:

   ```
   $ docker stop ffab54628f8f
   ```

4. Verify that the application is no longer running by using the **curl** command.

   ```
   $ curl -k http://127.0.0.1:80
   ```

   The output should return a connection refused error since the container running the application is no longer active.

5. Check the status of all the containers in the system.

   ```
   $ docker ps -a
   ```

   The status for the container should show **exited**, which means the container is stopped.

6. Delete the existing container using the **rm** flag.

   ```
   $ docker rm webapp_<userID>
   ```

   Example:

   ```
   $ docker rm webapp_david
   ```

7. Verify that the container is deleted.

```
$ docker ps -a
```

The container entry should be gone.

Congratulations! You have successfully containerized a Docker image and run the container.

Note: Do NOT delete the Docker image created in this lab, because it will be used as an artifact in an upcoming lab practice.

Container-Based Application Development: Use OCI Container Registry to Manage Images

Lab 2-2 Practices

Estimated Time: 30 minutes

Setup Instructions for the Lab

IMPORTANT: If you are using a tenancy in which you do not have administrator permissions, you will need to have the administrator add this additional policy statement:

Compartment level:

```
      Allow group <groupname> to manage repos in compartment
      <compartment name>
```

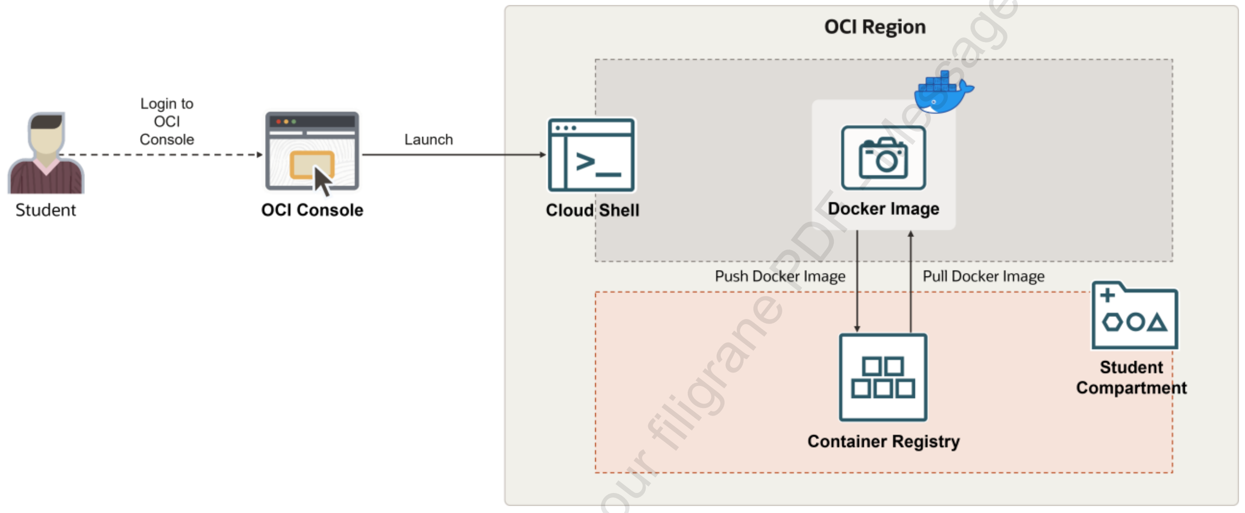
Note: If you are using your own tenancy you will not need to add any group permission policy statements.

Get Started

Overview

The development to production workflow can be made simpler with the help of an Oracle-managed registry. For developers, Container Registry makes it simple to store, share, and manage container images (such as Docker images).

In this practice, you will create a Container Registry and will also perform some basic operations such as push and pull a Docker image.



The practices include:

a. Create a new Container Repository.

b. Sign in to Oracle Cloud Infrastructure Registry (OCIR) from the cloud shell.

c. Tag the Docker image.

d. Push the image to OCIR.

e. Verify if the image has been pushed.

g. Pull the image from the OCIR repository.

Prerequisites

You have completed **Lab 2-1** (*Container-Based Application Development: Create a Docker Image for a Web Application*).

Assumptions

- You are signed into your Oracle Cloud Infrastructure (OCI) account using your credentials.
- You will replace the *<userID>* placeholder with your first name.

Create a New Container Repository

Create an empty repository in your compartment and give it a name that's unique across all compartments in the entire tenancy. Having created the new repository, you can push an image to the repository using the Docker CLI.

Tasks

1. Check if you can access Oracle Cloud Infrastructure Registry (OCIR):
   a. In the Console, open the navigation menu and click **Developer Services.** Under **Containers & Artifacts**, click **Container Registry**.
   b. Choose your assigned compartment from **List Scope** in the menu on the left.
   c. Review the repositories that already exist. This practice assumes that no repositories have yet been created.
2. Click **Create Repository**.
3. Select the compartment that's assigned to you.

   Enter a name for the new repository: `oci_sample_webapp_<userID>`
   - Replace *<userID>* with your first name.

   For example: `oci_sample_webapp_david`
4. Select the **Private** option to limit access to the new repository.
5. Click **Create Repository** to create the new repository.

Sign In to OCIR from the Cloud Shell

Log in to the Container Registry (OCIR) using your username and your OCI Auth Token.

Note: You will use the value of the Auth Token (**mytoken**) you created earlier in **Lab 1-1**. If you do not have the value of your token saved, you will need to delete the token and re-create it – (refer to Lab 1-1 if necessary).

Tasks

1. Open **Cloud Shell** by clicking the icon and link at the right of the OCI console header.

2. In the cloud shell window, log in to the Container Registry (OCIR) by entering:

   ```
   $ docker login <region-key>.ocir.io
   ```

   For example:

   ```
   $ docker login phx.ocir.io
   ```

3. When prompted, enter your "full" OCI username in the format `<tenancy-name>/<oci-username>`.

   For example: `mytenancyname/myusername`

   Enter the Auth Token value you copied earlier as the password.

   Note: When you enter or paste the password in the terminal, you will not see masked characters. Press **Enter** on your keyboard to continue.

Tag the Docker Image

A tag identifies the OCIR region, tenancy, and specific repository to which you want to push the image.

This task requires the existing Docker image `oci_sample_webapp_<userID>:1.0`, which you created earlier in Lab 2-1.

Tasks

1. In the Cloud Shell terminal, run the following command to attach a tag to the image that you're going to push to the OCIR repository:

   ```
   $ docker tag oci_sample_webapp_<oci-userID>:1.0
   <region-key>.ocir.io/<namespace>/<repo-name>:<tag>
   ```

   Where:

   - `<region-key>` is the key for the OCIR region you're using (*obtained in Lab 1-1*)
   - `ocir.io` is the Oracle Cloud Infrastructure Registry name
   - `<namespace>` is the namespace string of the tenancy (*obtained in Lab 1-1*)

     For example: ansh81vru1zp
   - `<repo-name>` is the name of the new repo in OCIR you just created

     For example: `oci_sample_david`
   - `<tag>` is an image tag to give this image. Use: `latest`

   Complete example:

   ```
   $ docker tag oci_sample_david:1.0
   phx.ocir.io/ansh81vru1zp/oci_sample_webapp_david:latest
   ```

2. Validate if the new image with the tag is listed.

   ```
   $ docker images
   ```

   Note: Although two tagged images will be shown, both are based on the same image with the same image ID.

Push the Image to OCIR

After assigning a tag to the image, you use the Docker CLI to push it to your Oracle Cloud Infrastructure Registry repository.

Tasks

1. In the Cloud Shell, run the following command to push the Docker image to the OCIR repository:

   ```
   $ docker push <region-key>.ocir.io/<namespace>/<repo-name>:<tag>
   ```

   For example:

   ```
   $ docker push
   phx.ocir.io/ansh81vru1zp/oci_sample_webapp_david:latest
   ```

   You will see that the different layers of the image are pushed in turn.

Verify if the Image has Been Pushed

Verify if the image has been pushed successfully to your OCIR repository.

Tasks

1. Go back to the OCIR Service page and make sure that you are in the compartment that's allotted to you from the **List Scope** on the left menu.

   You'll see the private repository `oci_sample_webapp_<userID>` that you created.

2. Click the name of the repository that contains the image you just pushed. You'll see:
   - An image with the tag `latest`
   - A summary page that shows you the details about the repository, including who created it and when, its size, and whether it's a public or a private repository

3. Click the image tag `latest`.

   On the **Summary** page, you'll see the image size, when it was pushed and by which user, and the number of times the image has been pulled.

Pull the Image from the OCIR Repository

Perform the pull operation after deleting the existing images from the local docker repository. You will pull the same image that was previously pushed to the OCIR repository.

Tasks

1. Delete the existing web app images from the local docker repository.

   a. Open Cloud Shell, and list all the images.

   ```
   $ docker images
   ```

   b. Run the `docker rmi` command to delete both the tagged image and the original image you created earlier.

   Original image:

   ```
   $ docker rmi oci_sample_webapp_<userID>:1.0
   ```

   For example:

   ```
   $ docker rmi oci_sample_webapp_david:1.0
   ```

   Tagged image:

   ```
   $ docker rmi
   <key>.ocir.io/<namespace>/oci_sample_webapp_<userID>:latest
   ```

   For example:

   ```
   $ docker rmi
   phx.ocir.io/ansh81vru1zp/oci_sample_webapp_david:latest
   ```

   Note: If the removal fails, force it by adding the `-f` flag to the command.

2. Verify if the two web app images are deleted.

   ```
   $ docker images
   ```

3. Back in the OCI console, from the OCIR page, select the repository and the image tag that needs to be pulled.

4. Click the **Actions** drop-down menu on the image summary page and select **Copy pull command**.

   The command you copy will include the fully qualified path to the image's location in the Container Registry, in the format: `<region-key>.ocir.io/<namespace>/<repo-name>:<tag>.`

5. Execute that copied command into the Cloud Shell terminal to pull the image to the local repository.

6. Verify the pulled image from the OCIR repository.

   ```
   $ docker images
   ```

   You should see the pulled image listed within your local repository.

Congratulations! You have successfully pushed and pulled an image from your OCIR repository.

Cloud Native DevOps with Managed Kubernetes: Access an OKE Cluster from Cloud Shell

Lab 3-1 Practices

Estimated Time: 45 minutes

Setup Instructions for the Lab

IMPORTANT: If you are using a tenancy in which you do not have administrator permissions, you will need to have the administrator add these additional policy statements:

Compartment level:

```
Allow group <groupname> to manage cluster-family in compartment
<compartment name>
Allow group <groupname> to manage virtual-network-family in
compartment <compartment name>
```

Note: If you are using your own tenancy you will not need to add any group permission policy statements.

VERY IMPORTANT: An additional service access policy is still required regardless of what type of user permissions you have.

- If you are using a tenancy in which you do not have administrator permissions, you will need to have the administrator add this additional policy statement.
- If you are using your own tenancy, simply use your initial tenancy administrator credentials to add this policy statement.

Compartment level:

```
Allow service OKE to manage all-resources in compartment
<compartment name>
```

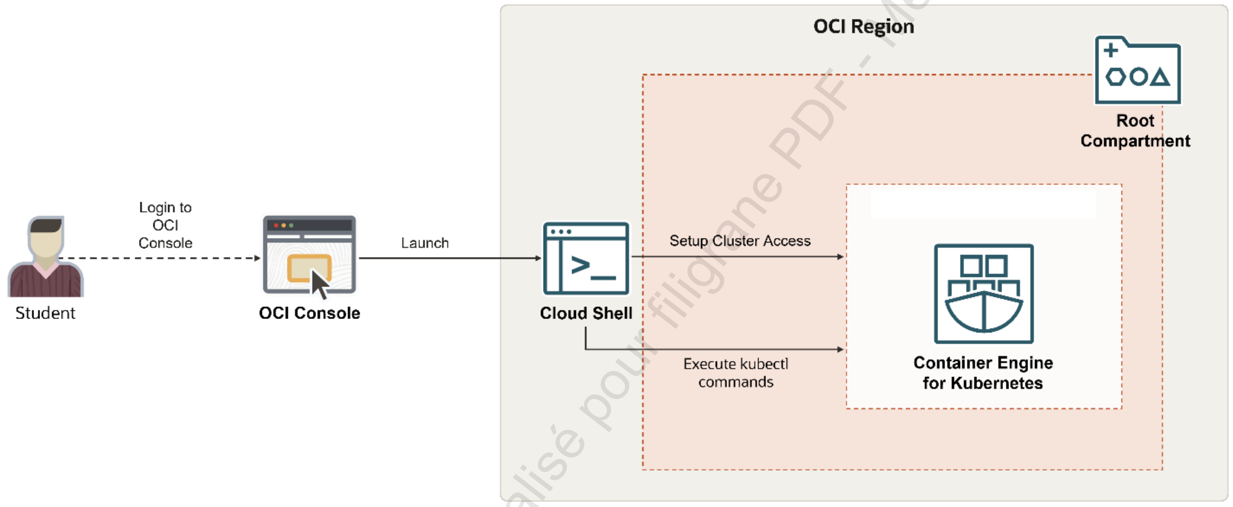
Get Started

Overview

A Kubernetes cluster is a group of nodes (machines running applications). Each node can be a physical machine or a virtual machine.

You need to set up access to your Kubernetes cluster to deploy your applications. The `kubectl` command-line client is a versatile way to interact with a Kubernetes cluster, including managing multiple clusters.

For more information on OCI Container Engine for Kubernetes (OKE), see the [OCI Container Engine Documentation.]()



In this practice, you will:

a. Create an enhanced OKE cluster

b. Create a Kubeconfig file with configuration access information to allow access to the OKE cluster from Cloud Shell.

c. Execute `kubectl` commands on an OKE cluster.

Prerequisites

You have completed **Lab 2-2** (*Cloud Native Fundamentals: Use OCI Container Registry to Manage Images*).

Assumptions

- You are signed into your Oracle Cloud Infrastructure (OCI) account using your credentials.
- You will replace the *`<userID>`* placeholder with your first name.

Create an enhanced OKE cluster

You will create a cluster with default settings and new network resources in the 'Quick Create' workflow using Container Engine for Kubernetes

Tasks

1. In the Console, open the navigation menu and click **Developer Services**. Under **Containers and Artifacts**, click **Kubernetes Clusters (OKE).**
2. Under **List Scope,** select the **compartment** in which you would like to create a cluster. Click **Create Cluster**.
3. Choose **Quick Create** and click **Launch Workflow**.
4. Fill out the dialog box:
    - **Name:** Provide a name: **OKE-Cluster**
    - **Compartment:** Choose your compartment
    - **Kubernetes Version:** Choose the most recent version
    - **Kubernetes API Endpoint:** Public Endpoint
    - **Node Type:** Managed
    - **Kubernetes Worker Nodes:** Private Workers
    - **Shape and Image:** VM.Standard.A1.Flex
        - **Image:** Leave the default selection
    - **Node Count:** 1
5. Click on **Next** to go to the Review page.
6. Review the resources to be created. Leave the Basic Cluster Confirmation **unchecked.**
7. Click on **Create cluster.**
8. On the Creating cluster and associated network resources window click on **close button** at the bottom to go back to the cluster details page.
9. Wait until the Cluster moves from *Creating* to *Active* state.

Set Up the kubeconfig File

To access a cluster using `kubectl`, you must set up a Kubernetes configuration file (commonly known as a 'Kubeconfig' file) to your client environment. For this practice, you will use your Cloud Shell VM as your client machine. The kubeconfig file provides the necessary details to access the cluster and is located (by default) at `~/.kube/config`.

The OCI CLI provides a `ce cluster create-kubeconfig` command that will create the config file (if it does not already exist), then add cluster access information to that file.

Tasks

1. On the **cluster details** page of the **OKE-Cluster** cluster you created in the previous task. Click **Access Cluster** to display the **Access Your Cluster** window.

2. Click **Cloud Shell Access**, copy the command to access the kubeconfig for your cluster via the VCN-Native public endpoint, and paste it into Notepad.

   For example:

   ```
   $ oci ce cluster create-kubeconfig --cluster-id
   ocid1.cluster.oc1.iad.xxxxxaaaziwdigokvlwhuaeslgxi6tdk473xqgodcb
   oc6nlgecsyudoxxxxx --file $HOME/.kube/config --region us-
   ashburn-1 --token-version 2.0.0  --kube-endpoint PUBLIC_ENDPOINT
   ```

   Note: *This is just an example command. Do not use this command to connect with the cluster that's created for your lab environment.*

3. Launch **Cloud Shell** and run the copied command. On successful execution, it will return a new config written to `kubeconfig` file.

4. To verify, view the newly created Kubeconfig file.

   ```
   $ view ~/.kube/config
   ```

Execute `kubectl` Commands on an OKE Cluster

Having set up the Kubeconfig file, you can start using `kubectl` to access the cluster by creating a sample deployment in the OKE cluster.

Tasks

1. Verify that **kubectl** can connect to the cluster.

   ```
   $ kubectl get nodes
   ```

   This will return the IP addresses of three worker nodes set up within this OKE cluster.

2. View the current list of namespaces in this Kubernetes cluster.

   ```
   $ kubectl get ns
   ```

3. Create a new unique namespace in the cluster to manage your resources.

   ```
   $ kubectl create ns ns-<userID>
   ```

   Where,

   - `ns-<userID>` - is a unique namespace for your group of resources within a cluster.
   - Replace `<userID>` with your first name.

   For example.

   ```
   $ kubectl create ns ns-david
   ```

4. View the cluster information.

   ```
   $ kubectl cluster-info
   ```

   It dumps relevant information regarding clusters for debugging and diagnosis.

5. Create a sample deployment in the OKE cluster.

   ```
   $ kubectl create deployment deploy-<userID> --
   image=docker.io/httpd:latest -n ns-<userID>
   ```

   Where:

   - `create deployment` creates a pod with a single running container
   - `deploy-<userID>` is the name of your deployment
   - `--image=docker.io/httpd:latest` is the image name from Docker Hub
   - `-n ns-<userID>` is the namespace where your Kubernetes objects are created

   This command will return `deployment.apps/deploy-<userID>` created.

For example.

```
$ kubectl create deployment deploy-david --
image=docker.io/httpd:latest -n ns-david
```

6. View the deployments in your namespace.

   ```
   $ kubectl get deploy -n ns-<userID>
   ```

   The output of this command will be a row with the deployment name and ready column set to 1/1 (1 pod is required and 1 pod is running). The age column indicates how long it has been deployed.

7. View the pods in your namespace:

   ```
   $ kubectl get pods -n ns-<userID>
   ```

   The output of this command is similar to `get deploy` – but provides the unique pod name for the deployment.

8. Expose your deployment using a service of the type **LoadBalancer** by using the following command:

   ```
   $ kubectl expose deployment deploy-<userID> --type=LoadBalancer
   --name=svc-<userID> --port=80 --target-port=80 -n ns-<userID>
   ```

   Where:

   - `deploy-<userID>` is the name of the deployment to be exposed
   - `--type=LoadBalancer` exposes the service externally using an OCI Load Balancer
   - `svc-<userID>` is the name of this new service
   - `--port=80 --target-port=80` is used to expose the application running within the cluster on port 80
   - `ns-<userID>` is the namespace where the Kubernetes objects are created

   For example:

   ```
   $ kubectl expose deployment deploy-david --type=LoadBalancer --
   name=svc-david --port=80 --target-port=80 -n ns-david
   ```

   This command will return: `service/svc-<userID> exposed`

9. View all the services in your namespace.

   ```
   $ kubectl get svc -n ns-<userID>
   ```

   The output of this command is a row with the service name and type set to **LoadBalancer**. It also shows both the CLUSTER-IP and EXTERNAL-IP addresses.

Note: If the EXTERNAL-IP displays as `<pending>`, simply re-execute the command a few seconds later.

10. Copy the IP address listed under the EXTERNAL-IP column and paste it into a browser to access the httpd application you deployed to the cluster earlier.

    The webpage will display: "**It Works!**"

11. Check the number of instances of pods running in your deployment.

    ```
    $ kubectl get replicaset -n ns-<userID>
    ```

    The output of this command displays the replicaset name along with the number of desired, current, and running replicas.

12. Increase the number of desired replicas to three, so that Kubernetes will start new pods for your deployment.

    ```
    $ kubectl scale --replicas=3 deployment/deploy-<userID> -n ns-
    <userID>
    ```

    On successful execution, this command will return:

    ```
    deployment.apps/deploy-<userID> scaled
    ```

13. Check if you now have three replicas running:

    ```
    $ kubectl get replicaset -n ns-<userID>
    ```

    This shows that the Load Balancer service will now balance the incoming requests among these three pods (replicaset).

14. View all the resources now available in your namespace:

    ```
    $ kubectl get all -n ns-<userID>
    ```

    This command displays all the pods, services, deployments, and replicasets within your namespace of the OKE cluster.

    Notice that the pod count has changed to 3 after the previous scaleup command.

15. View the pod logs. The `kubectl logs` command lets you inspect the logs for a particular pod.

    ```
    $ kubectl logs <podname> -n ns-<userID>
    ```

Where:

<podname> is the complete pod name to be viewed. For example, `pod/deploy-david-cd95b4455-f8plr.`

Before completing this practice, you should now need to delete all of the Kubernetes objects that you have created in your namespace.

16. Delete your deployment.

    ```
    $ kubectl delete deploy deploy-<userID> -n ns-<userID>
    ```

    On successful execution, the following command will be displayed:

    ```
    deployment.apps "deploy-<userID>" deleted
    ```

17. Delete your LoadBalancer service object:

    ```
    $ kubectl delete svc svc-<userID> -n ns-<userID>
    ```

    On successful execution, the following command will be displayed:

    ```
    service "svc-<userID>" deleted
    ```

18. Run the following command to verify that you no longer have any resources in your namespace:

    ```
    $ kubectl get all -n ns-<userID>
    ```

    Output: `No resources found in ns-<userID> namespace.`

19. Because all the resources are deleted, if you go back to your browser and hit refresh on the IP address you pasted earlier, the page will no longer respond.

Important Note: Do not delete the namespace and entry created in the kubeconfig file in this lab, because they will be required in the upcoming lab.

Congratulations! You have successfully set up the cluster access and executed various `kubectl` commands.

Managed Kubernetes: Deploy a Load-balanced Web App to an OKE Cluster

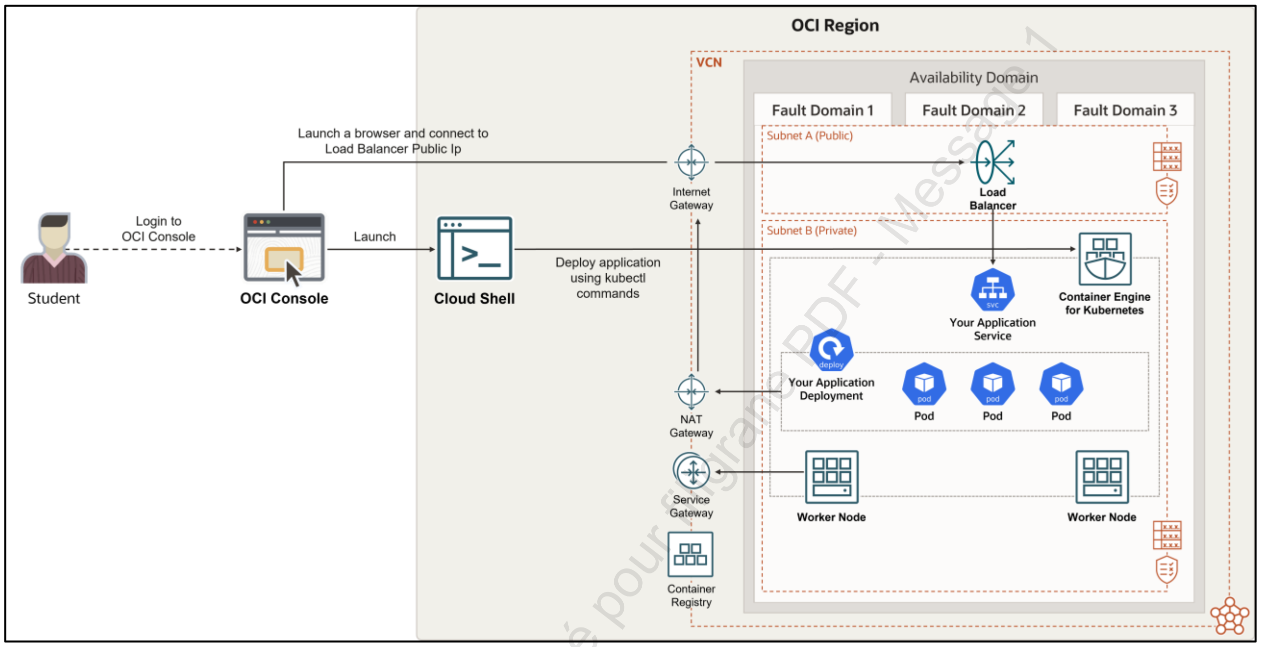
Lab 3-2 Practices

Estimated Time: 45 minutes

Get Started

Overview

In this practice, you will create a named **secret**, which contains your Oracle Cloud Infrastructure (OCI) credentials and add them to a deployment manifest. You will then use this manifest to deploy a sample Web application to your namespace in the OKE cluster, and later verify if the application is accessible.



In this practice, you will:

a. Create a Kubernetes (OKE) Secret

b. Add the secret and the image path to the deployment manifest

c. Deploy the sample web application to an OKE cluster

d. Verify that the sample web application is accessible

Prerequisites

Because you will use an existing Docker image, OCIR repository, and Kubernetes namespace from the previous lab to perform tasks for this practice, you must have already completed:

- **Lab 2-1** (*Container-Based Application Development: Create a Docker Image for a Web Application*)
- **Lab 2-2** (*Container-Based Application Development: Use OCI Container Registry to Manage Instances*)
- **Lab 3-1** (*Managed Kubernetes: Access an OKE Cluster from Cloud Shell*)

Assumptions

- You are signed into your Oracle Cloud Infrastructure (OCI) account using your credentials.
- You will replace the `<userID>` placeholder with your first name.

Create a Kubernetes (OKE) Secret

To enable Kubernetes to pull an image from an OCI Container Registry (OCIR) repository when deploying an application, you need to create a Kubernetes **secret**. The secret contains all the login details you would provide while logging in to OCIR using the `docker login` command, including your Auth Token.

Tasks

1. Open **Cloud Shell**.

2. Run the following command to create a secret:

```
$ kubectl create secret docker-registry mysecret-<userID> --docker-
server=<region-key>.ocir.io –docker-userID=”<tenancy-name>/<oci-
username>” --docker-password=”<oci-auth-token>” --docker-
email=student@example.org -n ns-<userID>
```

Where:

- `mysecret-<userID>`: A unique name for the secret

  For example: `mysecret-david` (replace `<userID>` with your first name)
- `<region-key>`: The key for the OCIR region you're using (*obtained in Lab 1-1*)
- `ocir.io` is the Oracle Cloud Infrastructure Registry name
- `<tenancy-name>` is the name of the tenancy (*obtained in Lab 1-1*)

  For example: `mytenancyname`
- `<oci-username>`: The User Name used to log in to OCI.
- `<oci-auth-token>`: Use your Auth Token value (*obtained in Lab 1-1*).
  For example: `R5kwpS-xxxxx((]51r]]`
- `ns-<userID>`: The namespace you created in Lab 3-1
- For example: `ns-david`
- `<email-address>`: Your email address

Full example:

```
$ kubectl create secret docker-registry mysecret-david --docker-server=phx.ocir.io --docker-username="mytenancyname/myusername" --docker-password="R5kwpS-xxxxx(([51r]]" --docker-email=student@example.org -n ns-david
```

You will see a confirmation message for secret creation on the screen.

3. Execute the following command to verify that the secret has been created:

```
$ kubectl get secrets -n ns-<userID>
```

For example:

```
$ kubectl get secrets -n ns-david
```

You will see the secret details displayed with the name, age, and other attributes.

Add the Secret and the Image Path to the Deployment Manifest

After the secret is created, you are required to include the name of the secret `(mysecret-`*`<userID>`*`)` and the full path of the image (`oci_sample_webapp_`*`<userID>`*`:latest`) located in your OCIR repository within the deployment manifest file. This manifest will be used for deploying the sample web application to the OKE cluster.

> **Note**: You have already pushed the image to your OCIR repository in **Lab 2-2** (*Container-based Application Development: Use OCI Container Registry to Manage Images*), which is the image you'll be using in this practice.

Tasks

1. Open **Code Editor**.
2. Within the Code Editor window, navigate to the cloned Git repo directory named `docker-helloworld-demo`, which is present in your home directory.
3. Open the `HelloWorld-lb.yaml` file and replace the placeholders with relevant values in the **Deployment** section:

   Under **metadata:**

   name: helloworld-deployment-*`<userid>`*

   For example: `helloworld-deployment-david`

   namespace: ns-*`<userid>`*

   For example: `ns-david`

   Under **containers:**

   image: `<region-key>`.ocir.io/*`<namespace>`*/*`<repo-name>`*:*`<tag>`*

   Where:

   - *`<region-key>`* is the key for the OCIR region you're using (*obtained in Lab 1-1*)
   - *`<namespace>`* is the unique namespace string (*obtained in Lab 1-1*)
   - *`<repo-name>:<tag>`* is the repo name:tag you used to push the image to OCIR

   For example: `phx.ocir.io/ansh81vru1zp/oci_sample_webapp_david:latest`

   Under **imagePullSecrets:**

   name: *`<secret-name>`*

For example: `mysecret-david`

4. Replace placeholders in the **Service** section:

   Under **metadata:**

   name: helloworld-deployment-*<userid>*

   For example: `helloworld-service-david`

   namespace: ns-*<userid>*

   For example: `ns-david`

   Once you've made all changes, your file should look similar to the following:

```
.
.
kind: Deployment
metadata:
  name: helloworld-deployment-david
  namespace: ns-david
.
.
    spec:
      containers:
      - name: helloworld
        image: phx.ocir.io/ansh81vru1zp/oci_sample_webapp_david:latest
        ports:
        - containerPort: 80
      imagePullSecrets:
      - name: mysecret-david
.
.
kind: Service
metadata:
  name: helloworld-service-david
  namespace: ns-david
.
```

5. Click **Save** from the File menu and exit the Code Editor.

Deploy the Sample Web Application to OKE Cluster

After making changes to the manifest, you are ready to deploy the application to the OKE cluster.

Tasks

1. Open Cloud Shell and navigate to the **docker-helloworld-demo** directory.

   ```
   $ cd ~/labs/docker-helloworld-demo
   ```

2. Run the following command:

   ```
   $ kubectl create -f HelloWorld-lb.yaml
   ```

   Confirmation of deployment and service creation will be displayed.

   Note: The HelloWorld Service Load Balancer is implemented as an OCI Load Balancer with a back-end set to route incoming traffic to the cluster nodes. In this case, the OKE service creates the new Load Balancer in the root compartment.

   You can see the new Load Balancer in the OCI Console by navigating to the **Load Balancers** page under **Networking**, and then selecting the root compartment from the **List Scope** menu in the left pane.

   Make a note of the overall health and public IP address for the Load Balancer. Your load balancer will be the one most recently created. Later in the next practice, you can validate that you copied the correct IP Address.

Verify That the Sample Web Application Is Accessible

Your deployment and LoadBalancer service should now be running in the OKE cluster node.

Tasks

1. Open Cloud Shell and execute the command:

   ```
   $ kubectl get services -n ns-<userID>
   ```

   For example,

   ```
   $ kubectl get services -n ns-david
   ```

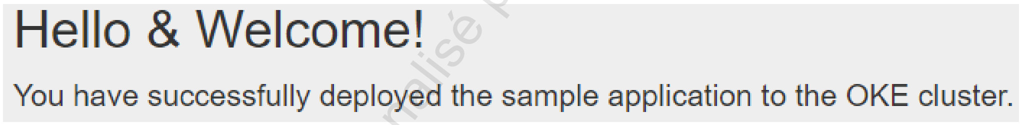
   Observe your HelloWorld-service Load Balancer details such as its External/Public IP and Port Number.

2. Open another browser tab and enter the EXTERNAL-IP into the browser's address bar to access the deployed application.

   The load balancer routes the request to available nodes in the cluster.

   In this example, you'll see only one node since the replica count is set to 1 in the Kubernetes manifest. Once the request reaches the node, you'll see a web page with the following content:

   - A message displaying:

     Hello & Welcome!

     You have successfully deployed the sample application to the OKE cluster.
   - The number of visits to the webpage from this client (increments when you refresh the page)
   - The current time
   - The Container ID of the pod hosting the web app

Now comes the fun part! Let's pretend your sample web application has suddenly gained popularity and you are now required to allocate more resources to it.

The OKE cluster is running on a single node pool with three worker nodes, thus you can easily scale your deployment.

3. To increase capacity, you can run an additional pod for your current single pod deployment.

   Execute the command:
   ```
   $ kubectl -n ns-<userID> scale --replicas=2
   deployment/<deploymentname>
   ```

   For example:
   ```
   $ kubectl -n ns-david scale --replicas=2 deployment/helloworld-
   deployment-david
   ```

   You will see a confirmation for deployment scaling on the screen.

4. Further, to see pod and deployment details, execute the command:
   ```
    $ kubectl get all -n ns-<userID>
   ```

   For example:
   ```
    $ kubectl get all -n ns-david
   ```

   Here, you will observe an additional row for the new pod that has spawned. You can identify the new pod by comparing the Container ID or the value in the Age column of the output.

   Also, the Deployment row shows '`2/2`' in the READY column, indicating the deployment is now hosted on two pods.

Before completing this practice, you now need to delete all of the Kubernetes objects that you have created in your namespace.

5. Delete your deployment and LoadBalancer service.
   ```
    $ kubectl delete -f HelloWorld-lb.yaml
   ```

   On successful execution, this command will display:
   ```
   deployment.apps "helloworld-deployment-<userID>" deleted
   service "helloworld-service-<userID>" deleted
   ```

6. Delete your Kubernetes secret.
   ```
   $ kubectl delete secret mysecret-<userID> -n ns-<userID>
   ```

   For example:
   ```
   $ kubectl delete secret mysecret-david -n ns-david
   ```

7. Execute the following command to verify that you no longer have any resources in your namespace.

```
$ kubectl get all -n ns-<userID>
```

Congratulations! You have successfully deployed a load-balanced web application to an OKE cluster using a deployment manifest file.

Serverless Functions: Build and Deploy an Oracle Function

Lab 4-1 Practices

Estimated Time: 30 minutes

Setup Instructions for the Lab

IMPORTANT: If you are using a tenancy in which you do not have administrator permissions, you will need to have the administrator add these additional policy statements:

Compartment level:

```
Allow group <groupname> to manage functions-family in
compartment <compartment-name>
Allow group <groupname> to manage virtual-network-family in
compartment <compartment name>
```

Note: If you are using your own tenancy you will not need to add any group permission policy statements.

Get Started

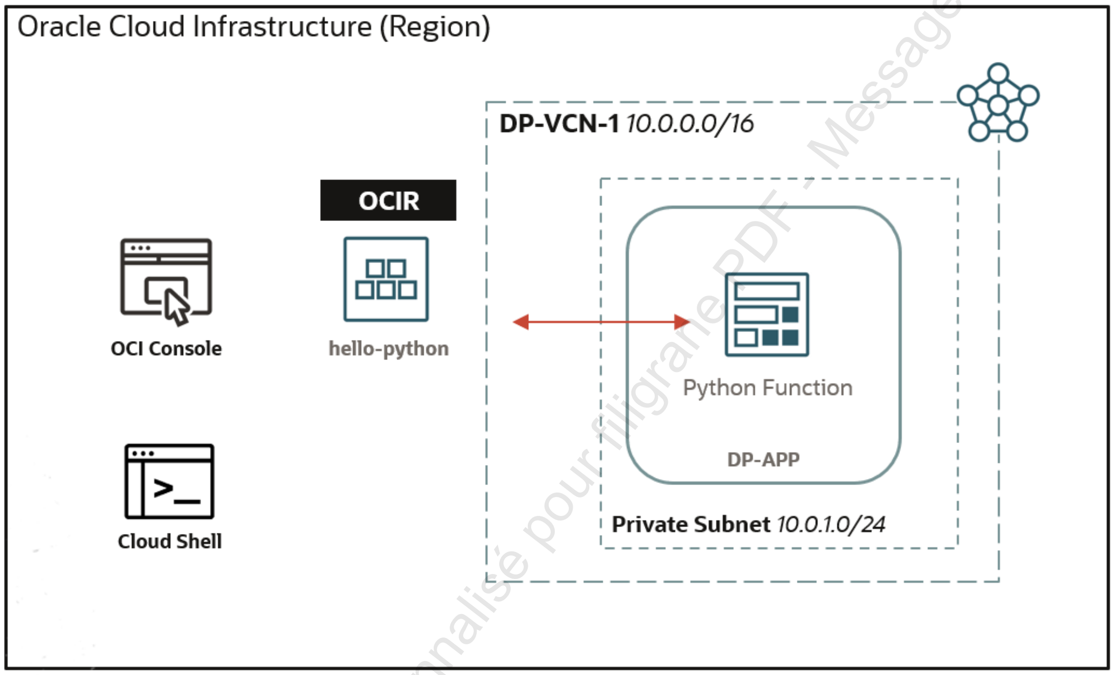
Overview

In this practice, you will build and deploy a sample serverless function written in Python to Oracle Functions. This will be segmented into three practice sections:

a. Create a Virtual Cloud Network (VCN) and a Functions application.

b. Create a private repository in OCIR and set up Cloud Shell for access.

c. Build and deploy the Function container and validate the function.



Prerequisites

You have completed **Lab 2-2** (*Container-Based Application Development: Use OCI Container Registry to Manage Images*).

Assumptions

- You are signed into your Oracle Cloud Infrastructure (OCI) account using your credentials.
- You will replace the `<userID>` placeholder with your first name.

Create a VCN and Functions Application

Serverless functions are deployed to an OCI Functions Application which must be configured to run functions in one to three subnets belonging to a specific VCN. The tasks for this part will assume that you require a new VCN.

You'll use the VCN with Internet Connectivity Wizard to create the new VCN, which will include a regional public subnet, a regional private subnet, an Internet gateway, a NAT gateway, and a service gateway. In addition, the wizard will set up basic security list rules for the two subnets.

Tasks

1. In the OCI Console, click the navigation menu, click **Networking**, and then click **Virtual Cloud Networks**.
2. Select the compartment you are using in the left column under **List Scope**.
3. Click **Start VCN Wizard**.
4. Select **Create VCN with Internet Connectivity**, and then click **Start VCN Wizard**.
5. Enter **DP-VCN** as the VCN name. Leave the default values for the remaining fields. Click **Next** and click **Create** to provision the VCN.
6. In the OCI Console, open the navigation menu and click **Developer Services**. Under **Functions**, click **Applications**.
7. Click **Create application** and specify:
    a. **Name**: `DP-APP`
    b. **VCN**: `DP-VCN`
    c. **Subnets**: `Private Subnet-DP-VCN (Regional)`
8. Click **Create**.

Create a Private Repository in OCIR and Set Up Cloud Shell for Access

Before uploading and deploying a container image containing the function code, you need to specify a private repository in the Container Registry that is within the same OCI region of the Function Application.

Tasks

1. In the Console, open the navigation menu and click **Developer Services**. Under **Containers & Artifacts**, click **Container Registry**.

2. Click the **Create repository** button.

3. For the Repository name, enter `<userID>/hello-python`

   where *<userID>* your first name which will be used later as the [repo-prefix] for Function deployments.

   For example: `david/hello-python`

4. Ensure **Access** is set to **Private**. Click **Create repository**.

5. From the navigation menu, select **Developer Services**. Go to **Functions** and click **Applications**.

6. Click **DP-APP**, and then click **Getting Started** in the left navigation pane under **Resources**.

7. Scroll down to reveal **Begin your Cloud Shell session**. Click **Launch Cloud Shell**.

   Note: The Cloud Shell environment can take up to 60 seconds to start.

In the Console, scroll down further to quickly familiarize yourself with the series of commands listed in the **Setup fn CLI on Cloud Shell** section.

In the following steps, you will execute some but not all of those listed commands. Note the **Copy** links found to the right of the listed commands. You will click these links and paste the copied commands into Cloud Shell when executing the subsequent steps.

8. Proceed to **(2) Use the context for your region**.

   Click the **Copy** link and paste to execute the `fn use context <region name>` in Cloud Shell to set your region identifier.

You'll see a message such as: "`Fn: Context` *`<region name>`* `currently in use`" or "`Now using context: <region name>`"

9. Perform **(3) Update the context with the function's compartment ID**

   Issue the `fn update context oracle.compartment-id` command to update the **fn CLI** context for your compartment.

10. Perform **(4) Provide a unique repository name prefix...**

    Edit, then issue the `fn update context registry` command to update the fn CLI context for the ***prefix*** of the repository you just created.

    Note: Replace [repo-name-prefix] with the *<userID>* you used earlier when creating the repository.

    For example:
    `fn update context registry phx.ocir.io/`*`ansh81vru1zp/`****`david`***

11. Perform **(6) Log into the Registry using the Auth Token as your password**

    Log in to the Container Registry using the listed docker login command.

    Note: You will use the value of the Auth Token (**mytoken**) you created earlier in **Lab 1-1**. If you do not have the value of your token saved, you will need to delete the token and re-create it – (refer to **Lab 1-1** if necessary).

    When prompted for the password, paste the Auth Token value in the terminal – (you will not see any masked characters) – just press the **Enter** key to continue.

    Do not execute any other commands listed in **Setup fn CLI on Cloud Shell**.

Build and Deploy the Function Container and Test Function

The function code was imported to your Cloud Shell VM when you cloned the GitHub repository in **Lab 1-1**. You will use **fn** commands to build, deploy, and invoke the function.

Tasks

1. In Cloud Shell, navigate to the directory containing the **hello-python** function code.
```
$ cd ~/labs/oci-functions/hello-python
```
2. Use the `fn deploy` command to build a container image for the function and add it to the repository. *(This may take up to 60 seconds.)*
```
$ fn -v deploy --app DP-APP
```
3. Use the `fn invoke` command to execute the function. *(This may take up to 30 seconds.)*
```
$ fn invoke DP-APP hello-python
```
   If successful, a JSON result will be returned: **{“message”: “Hello World”}**
4. To further validate the deployment, in the Console, open the navigation menu and click **Developer Services**. Under **Containers & Artifacts**, click **Container Registry**.
5. Expand the repository to view the image label and information.
6. Now navigate to **Developer Services**, and then click **Functions**.
7. Click the application link for **DP-APP**.
   a. Scroll down and notice the Image, Image digest, and Invoke endpoint for the **hello-python** function.
   b. Click the **hello-python** link. Note the General Information as well as the function metrics below.

Congratulations! You have created a VCN and function application. You also built and deployed a sample serverless function written in Python to Oracle Functions.

Serverless Functions: Use a Custom Dockerfile for an Oracle Function

Lab 4-2 Practices

Estimated Time: 30 Minutes

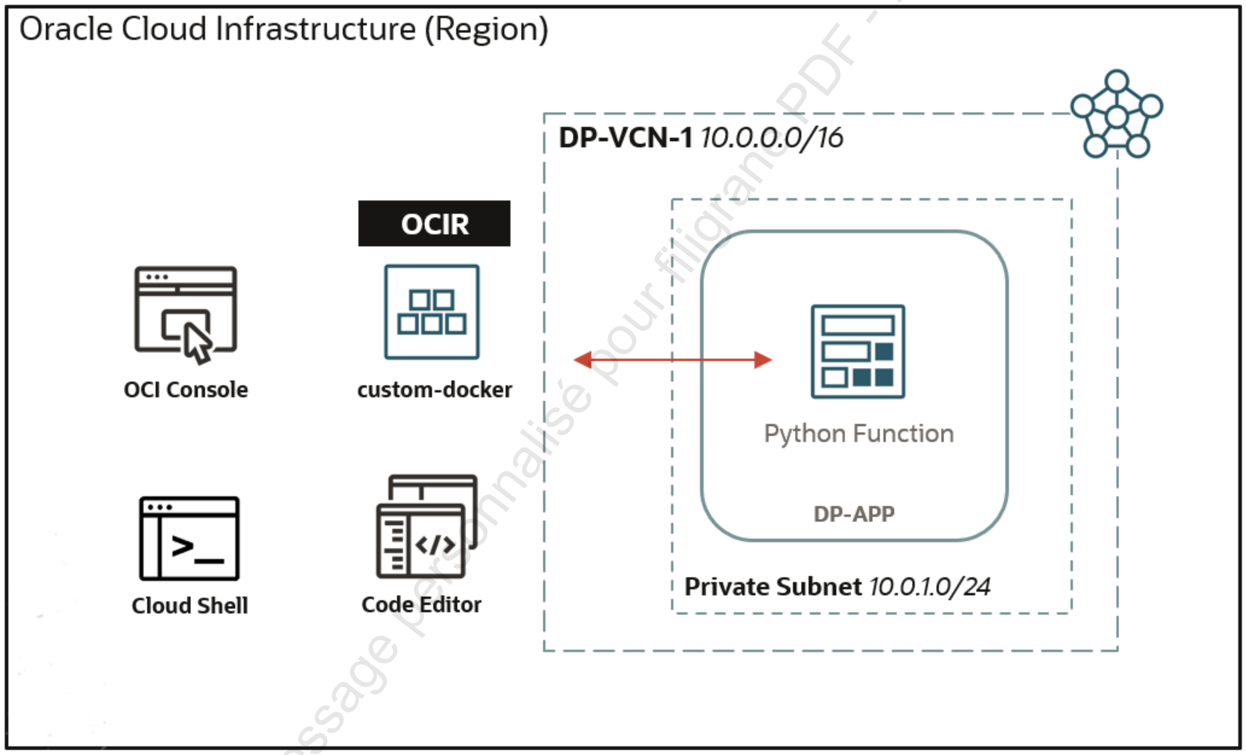
Get Started

Overview

In this practice, you will use a custom Dockerfile to instruct the Fn Server on how to build the Docker image for a function. When you develop functions, you're not directly exposed to the Docker platform, but by making a few changes to the configuration files and adding Fn files, the image can be converted into an Fn function.

In this practice, you will:

a. Create a private repository in OCIR
b. Set Function metadata using Code Editor
c. Build, deploy, and invoke the Function



Prerequisites

You have completed **Lab 4-1** (*Serverless Functions: Build and Deploy an Oracle Function*).

Assumptions

You are signed in to your Oracle Cloud Infrastructure account using your credentials.

Create a Private Repository in OCIR

Before uploading and deploying a container image containing the function code, you need to specify a private repository in the Container Registry that is within the same OCI region of the Function Application.

Tasks

1. In the Console, open the navigation menu and click **Developer Services**. Under **Containers & Artifacts**, click **Container Registry**.

2. Click the **Create repository** button.

3. For the Repository name, enter `<userID>/custom-docker`

   where *<userID>* is your first name and is also the [repo-prefix] used for Function deployments you set earlier in **Lab 4-1**

   For example: `david/custom-docker`

4. Ensure **Access** is set to **Private**. Click **Create repository**.

   A repository `<userID>/custom-docker` is created.

Set Function Metadata Using Code Editor

You're using the popular ImageMagick, which is software delivered as a binary distribution. This ImageMagick will help to process the image and find its width and height of it. All the required files are in your GitHub cloned directory. You'll review these files and then deploy the function.

Tasks

1. In your Cloud Shell, go to the directory `labs` directory that you have created in the earlier lab, and then navigate to the **custom-docker** subdirectory.
   ```
   $ cd ~/labs/oci-functions/custom-docker
   ```

2. List the files in the **custom-docker** directory.
   ```
   $ ls
   ```
   This should display `Dockerfile`, `func.yaml`, `func.js`, `package.json`, and two image files.

3. Click the **Code Editor** icon next to Cloud Shell in the Console header. Code Editor allows you to edit files and source codes present in the cloned Git directory within the cloud shell.

4. From within the **Code Editor**, navigate to the **custom-docker** directory.

   a. Open the `func.js` file and review how the function is implemented. No need to change anything in this file.

      In this code, it is taking a binary image as its argument and writes it into a temporary file. It'll use the ImageMagick library to display the width and height of the image. Because the function argument type is binary, the "inputMode" property is set to "buffer."

   b. Open `package.json` and review the file.
      ```
      {
          "name": "custom-docker",
          "version": "1.0.0",
          "description": "Function using ImageMagick that returns…
      ```
      Note: The function name indicated here will also be used to locate the repository in conjunction with the [repo-prefix-name] you set with **fn context**. When you created the repository for this function, recall that you named it `<userID>/custom-docker`.

   c. Open the `func.yaml` file and review the function metadata.

```
schema_version: 20180708
name: custom-docker
version: 0.0.19
runtime: docker
triggers:
- name: custom-docker-trigger
  type: http
  source: /custom-docker
```

d. Edit **Dockerfile** in **Code Editor**, which is at the same level as the three files (`func.yaml`, `func.js`, and `package.json`). Because this function requires Alpine, you must install the ImageMagick Alpine package by using the **`apk`** package management utility.

   Between the second pair of FROM and WORKDIR lines, insert the following statement in your Dockerfile:

```
RUN apk add --no-cache imagemagick
```

Here's an example of how your edited Dockerfile should look:

```
FROM fnproject/node:dev as build-stage
WORKDIR /function
ADD package.json /function/
RUN npm install

FROM fnproject/node
RUN apk add --no-cache imagemagick
WORKDIR /function
ADD . /function/
COPY --from=build-stage /function/node_modules/
/function/node_modules/
ENTRYPOINT ["node", "func.js"]
```

e. Save the **Dockerfile** after you edit.

f. Review the images (`image1.jpg` and `image2.jpg`) and exit from Code Editor.

With this Dockerfile, the Node.js function, its dependencies, and the "imagemagick" Alpine package will be included in an image derived from the base fnproject/node image.

Now you have everything ready to build and deploy the function.

Build, Deploy, and Invoke the Function

The function code is available in your custom-docker directory. You will use **fn** commands to build, deploy, and invoke the function.

Tasks

1. In **Cloud Shell**, make sure that you're in the `custom-docker` directory and check if you have six files in that directory.

   ```
   $ cd ~/labs/oci-functions/custom-docker
   $ ls
   ```

2. Enter the command to build the function and its dependencies as a Docker image.

   ```
   $ fn -v build
   ```

3. Enter the command to deploy the function by pushing the Docker image to the specified OCI Container Registry repository.

   ```
   $ fn -v deploy --app DP-APP
   ```

   Invoke the function and make sure it's working as expected. *(This may take up to 30 seconds.)*

   ```
   $ cat image1.jpg | fn invoke DP-APP custom-docker
   ```

   When you run the command, you should retrieve the width and height of the image.

   If successful, a JSON result will be returned: **{"width":144, "Height":144}**

4. Check the dimensions of another image. `Image2.jpg` is already in your custom-docker directory.

   ```
   $ cat image2.jpg | fn invoke DP-APP custom-docker
   ```

   If successful, a JSON result will be returned: **{"width":255, "Height":255}**

Congratulations! You have successfully built a function using a custom Dockerfile.

Serverless Streaming: Use Object Storage and OCI Streaming in a Microservice

Lab 5-1 Practices

Estimated Time: 30 minutes

Setup Instructions for the Lab

IMPORTANT: If you are using a tenancy in which you do not have administrator permissions, you will need to have the administrator add these additional policy statements:

Compartment level:

```
Allow group <groupname> to manage stream-family in compartment
<compartment-name>
Allow group <groupname> to manage object-family in compartment
<compartment-name>
```

Note: If you are using your own tenancy you will not need to add any group permission policy statements.

VERY IMPORTANT: A new dynamic group needs to be created and two policy statements must added to allow Oracle Functions access to Object Storage and Streaming. This is still required regardless of what type of user permissions you have.

- If you are using a tenancy in which you do not have administrator permissions, you will need to have the administrator create this dynamic group and add the policy statements.
- If you are using your own tenancy, simply use your initial tenancy administrator credentials to create the dynamic group and add the policy statements.

Create Dynamic Group:

Name**: FnApps-DG**

Matching Rule: **`ALL {resource.type = 'fnfunc'}`**

Add Policy Statements:

Compartment Level

```
Allow dynamic-group FnApps-DG to manage objects in compartment
<compartment-name>
Allow dynamic-group FnApps-DG to use stream-family in
compartment <compartment-name>
```

Get Started

Overview

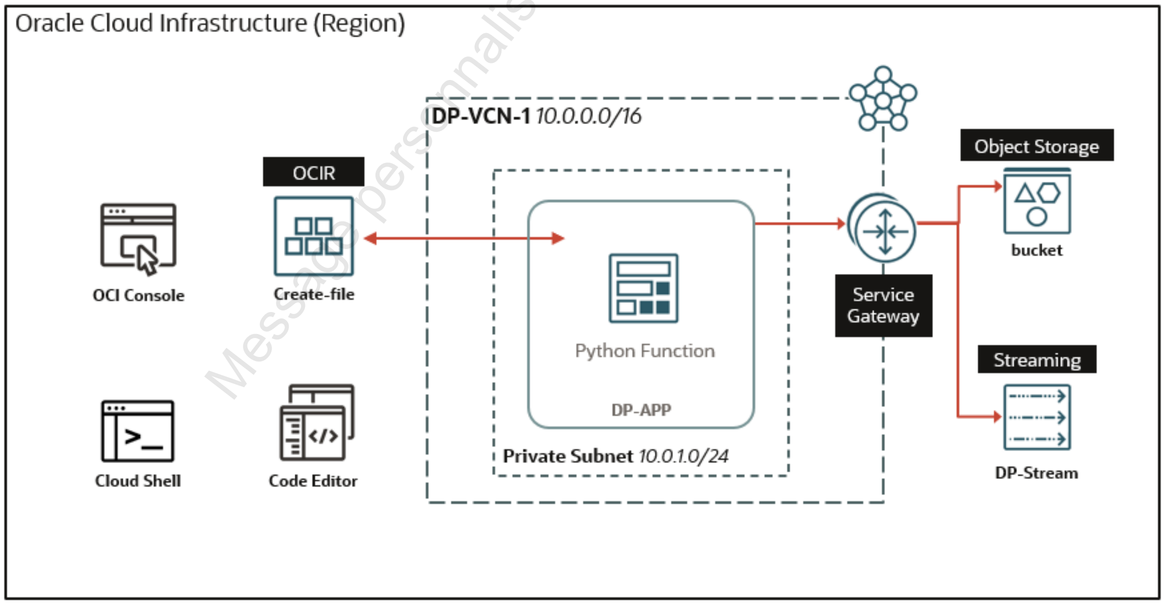
In this practice, you will work with a sample function that creates a file and then puts it into an OCI Object Storage bucket. It will also use the name and contents of that file to produce a message to an OCI Stream. The **create-file** function (written in Java) requires a JSON object input that includes three string values:

- The name of the file to be created
- The content of the file to be created
- The name of the Object Storage bucket to put the file object

The function also requires the Object Storage **namespace** associated with your tenancy that you will provide as an environment variable configuration.

To demonstrate producing messages to an OCI Stream, the function creates a message using the OCI Streaming SDK. The message will use the file name as the message key and the file contents as the message. In this practice, you will:

a. Create an OCI Stream and OCI Object Storage bucket
b. Create a private repository in OCIR
c. Build, deploy, and examine the Function code
d. Configure and test the function



Prerequisites

You have completed **Lab 4-1** (*Serverless Functions: Build and Deploy an Oracle Function*).

Assumptions

- You are signed into your Oracle Cloud Infrastructure (OCI) account using your credentials.
- You will replace the `<userID>` placeholder with your first name.

Create an OCI Stream and OCI Object Storage Bucket

In this practice, you will create a new Stream and Stream Pool that will be used as the destination for producing messages from code written in the Oracle Function that you will deploy later. You will need to make a note and copy the OCID of the stream once it has been provisioned.

You will also create a private OCI Object Storage bucket, used to store the files that will be created by the same Oracle Function.

Tasks

1. Click the navigation menu, and click **Analytics & AI.** Under **Messaging**, click **Streaming**.
2. Select the compartment that has been assigned to you in the left column under **List Scope**.
3. Click **Create Stream**.
4. For the **Stream Name**, enter `DP-Stream`.
5. Select the *Create a new stream pool* option. For the **Stream Pool Name**, enter `DP-Stream-Pool`.
6. Leave the default values for the remaining fields. Click **Create** to provision the Stream Pool and Stream.

   Note: The provisioning can take up to 60 seconds to complete.
7. Within the **Stream Information** pane, copy the **OCID** and **Messages Endpoint** values to a text file. *You will need these values in a later practice task.*
8. Click the navigation menu, and click **Storage.** Under **Object Storage & Archive Storage**, click **Buckets**.
9. Click **Create Bucket**.
10. For the **Bucket Name**, enter `<userID>-bucket` *(where <userID> is your first name)*, for example, **david-bucket**.
11. Leave the default values for the remaining fields. Click **Create**.

Create a Private Repository in OCIR

Before building and deploying a container image containing the function code, you need to create a private repository in the Container Registry that is within the same OCI region of the Function Application.

Tasks

1. In the Console, open the navigation menu and click **Developer Services**. Under **Containers & Artifacts**, click **Container Registry**.

2. Click the **Create repository** button.

3. For the Repository name, enter `<userID>/create-file`

   where *<userID>* is your first name and is also the [repo-prefix] used for Function deployments you set earlier in **Lab 4-1**

   For example: `david/create-file`

4. Ensure **Access** is set to **Private**. Click **Create repository**.

   A repository `<userID>/create-file` is created.

Build, Deploy, and Examine Function Code

The function code and related files are already staged in a subfolder of the git repository you cloned in earlier practice.

In this practice, you build and deploy that existing code. Next, you will review the logic in the code that adds a file to an OCI Object Storage bucket and produces a message to an OCI Stream.

Tasks

1. Navigate to the directory containing the **create-file** function code.
   ```
   $ cd ~/labs/oci-functions/create-file
   ```
2. Use the `fn deploy` command to build a container image for the function and add it to the repository.
   ```
   $ fn -v deploy --app DP-APP
   ```
   Note: This build and deploy process can take up to three minutes to complete.
3. Meanwhile (during the build), go ahead and open **Code Editor** by clicking the Code Editor icon located in the upper-right section of the OCI Console.
4. Navigate to `/labs/oci-functions/create-file/src/main/java/com/example/fn` and open the `WriteFileProduceMessage.java` file to examine the function's source code.
   - Lines 10-26 contain the OCI Java SDK package import statements for Authentication, Object Storage, and Streaming classes.
   - Lines 42-60 are the class constructor method that prints the env vars of the Functions container that will be used by the Authentication Provider as well as the configuration environment values needed by the function at runtime – *(you will configure these 3 later after the function is deployed)*. The `ObjectStorageClient` is also instantiated here.
   - Lines 62-87 are a simple static class used to store the data *(filename, file contents, and bucket name)* that will be sent into the function.
   - Within the function's **handle** method, lines 131-143 contain the logic for creating the file and putting the file object into the Object Storage bucket.

- The logic for producing the message to the OCI Stream starts in the **handle** method (lines 146-153), then continues within the **publishMessage** static method (lines 89-115).

5. Close (or minimize) the Code Editor window.

Configure and Test Function

Now that the function is deployed, in this practice, you will configure environment variables that are required by the function to access the Object Storage bucket and the OCI stream at runtime.

Next, you will test the function by sending one or more requests that include a filename, file contents, and the name of the bucket to use. You will validate execution success by navigating to the OCI stream to load and view the message, and then to the Object Storage bucket to see the uploaded file.

Tasks

1. In the OCI Console, navigate to **Developer Services**, under **Functions** click **Applications**, then click **DP-APP**. Under the **Resources** section, click **Functions**, then click the link for the newly deployed **create-file** function.

2. Under the **Resources** section, click **Configuration**.

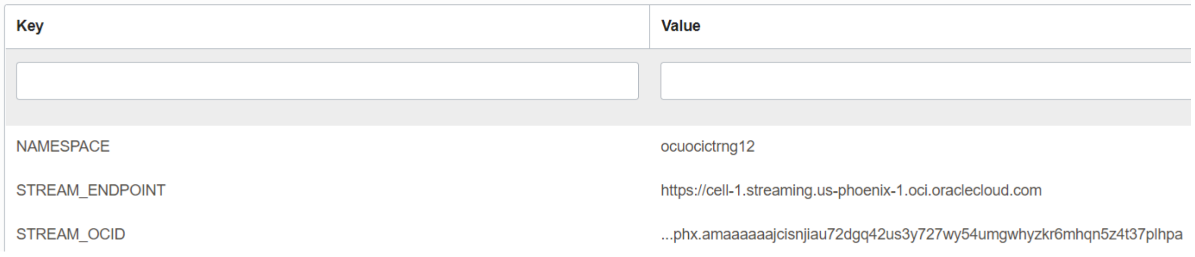
3. Add the following three configuration environment variables to the function. The **Key**s are **NAMESPACE**, **STREAM_OCID**, and **STREAM_ENDPOINT**.

   - The **Value** for NAMESPACE is the object storage namespace for your tenancy.
     Note: This value was saved earlier as ***namespace*** during Lab 1-1.
     For example: `ansh81vru1zp`

   - The **Value** for STREAM_OCID is the **OCID** of the **DP-Stream** you created and copied in an earlier task

   - The **Value** for STREAM_ENDPOINT is the **Messages Endpoint** of the **DP-Stream** you created and copied in an earlier task.

   Note: You must click the **+** icon to the right of each Key/Value pair to add them as configurations.



Key	Value
NAMESPACE	ocuocictrng12
STREAM_ENDPOINT	https://cell-1.streaming.us-phoenix-1.oci.oraclecloud.com
STREAM_OCID	...phx.amaaaaaajcisnjiau72dgq42us3y727wy54umgwhyzkr6mhqn5z4t37plhpa

4. Use `fn invoke DP-APP create-file` to execute the function. However, you will need to echo and pipe a JSON object string to the command that includes the following name/value pairs:
   - `"name":"file1.txt"` *(Any filename will work – it should be unique for each invocation.)*
   - `"bucketName":"<userID>-bucket"` *(This is the name of the bucket you created earlier in the practice.)*
   - `"content":"Sample content for file1"` *(This can be any string that will comprise the content of the file and the message to be sent to the stream.)*

   For example:

   ```
   $ echo -n '{"name":"file1.txt","bucketName":"david-bucket",
   "content":"Sample content for file1"}' | fn invoke DP-APP create-
   file
   ```

   Note: This may take up to 30 seconds to execute the first time.

   If successful, a result message is returned in two lines:

   - `Successfully produced message to Stream with Key=<filename> Message=<message content>`
   - `Successfully put to Object Storage with Filename=<filename> Bucket=<bucket name>`

5. To validate that the message was sent to the Stream, in the Console, open the navigation menu and click **Analytics & AI**. Under **Messaging**, click **Streaming**. Click the **DP-Stream** link.

6. Click the **Load Messages** button to view information about recently published messages.

   Note: If you are too slow (longer than 60 seconds), you may need to send another request to the function so you can return to see this information in the OCI Console.

7. To validate that the file object was sent to the bucket, in the Console, open the navigation menu and click **Storage**. Under **Object Storage & Archive Storage**, click **Buckets**.

   Click the **<userID>-bucket** link.

8. Scroll down under **Objects** to see the files that have been put into your bucket.

9. Feel free to execute additional invokes to the function (using different data values) to observe the functionality.

Congratulations! You deployed and configured a function that both puts objects into an OCI Object Storage bucket as well as publishes messages to an OCI Stream.

Serverless Events: Create Event Rule to Trigger a Function

Lab 7-1 Practice

Estimated Time: 30 minutes.

Setup Instructions for the Lab

IMPORTANT: If you are using a tenancy in which you do not have administrator permissions, you will need to have the administrator add these additional policy statements:

Compartment level:

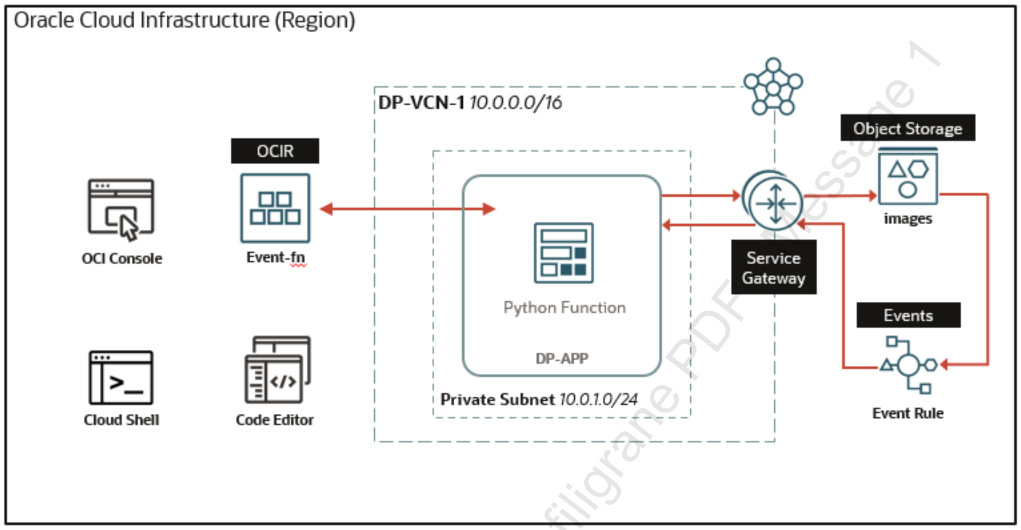
```
Allow group <groupname> to manage log-groups in compartment
<compartment-name>
Allow group <groupname> to manage cloudevents-rules in
compartment <compartment-name>
```

Note: If you are using your own tenancy you will not need to add any group permission policy statements.

Get Started

Overview

In this lab exercise, you will work with an example function designed to extract image graphic metadata that is automatically triggered upon image files being uploaded to a given OCI Object Storage bucket. When triggered, the **event-fn** function (written in Java) expects a `CloudEvent` JSON object string.



This lab will be segmented into five practice sections:

a. Create a public OCI Object Storage bucket to be used for the image files and enable it for emitting Object events.

b. Create a private repository in OCIR.

c. Build, deploy, and examine the function code.

d. Create an Event Rule in OCI Events that fires when an image file is uploaded to the public bucket and then triggers the deployed function.

e. Test the event-driven use case by uploading an image file to the bucket.

Prerequisites

You have completed **Lab 5-1** (*Serverless Streaming: Use Object Storage and OCI Streaming in a Microservice*).

Assumptions

- You are signed into your Oracle Cloud Infrastructure (OCI) account using your credentials.
- You will replace the *`<userID>`* placeholder with your first name.

Create a Public OCI Object Storage Bucket

You'll create a new Object Storage bucket in your compartment configured to be used for the image files and enable it for emitting Object events. Although you would normally use a private bucket, for this practice, you will change the bucket's visibility to public. This is only because our example function does not contain the additional authentication code logic for accessing a private bucket.

Tasks

1. Click the navigation menu and click **Storage.** Under **Object Storage & Archive Storage**, click **Buckets**.

2. Select the compartment that has been assigned to you in the left column under **List Scope**.

3. Click **Create Bucket**.
   - **Bucket Name**: `<userID>-images` (where <*userID*> is your first name)

     For example, `david-images`
   - **Emit Object Events**: Select the check box.

4. Leave the default values for the remaining fields. Click **Create**.

5. Select the newly created bucket and click the **Edit Visibility** button.

6. Select the **Public** option, leave **Allow users to list objects from this bucket** selected, and then click **Save Changes**.

   Note: This is the location where you will upload image files using the **Upload** button.

   DO NOT upload any files yet.

Create a Private Repository in OCIR

Before building and deploying a container image containing the function code, you need to create a private repository in the Container Registry that is within the same OCI region of the Function Application.

Tasks

1. In the Console, open the navigation menu and click **Developer Services**. Under **Containers & Artifacts**, click **Container Registry**.
2. Click the **Create repository** button.
3. For **Compartment**, ensure that your assigned compartment is selected.
4. For the Repository name, enter `<userID>/event-fn`

   where *<userID>* is your first name and is also the [repo-prefix] used for Function deployments you set earlier in **Lab 4-1**

   For example: `david/event-fn`
5. Ensure **Access** is set to **Private**. Click **Create repository**.

   A repository `<userID>/event-fn` is created.

Build, Deploy, and Examine Function Code

The function code and related files are already staged in a subfolder of the git repository you cloned in an earlier lab exercise.

In this practice, you build and deploy that existing code. Next, you will review the logic in the code that:

- Converts the received `CloudEvent` to `Map` data
- Retrieves the image file from the Object Storage
- Extracts the metadata from the image file

Finally, you will enable logging for the **DP-APP** application to allow for viewing the metadata information printed to stdout from the function code.

Tasks

1. Navigate to the directory containing the **event-fn** function code.
   ```
   $ cd ~/labs/oci-functions/event-fn
   ```
2. Use the `fn deploy` command to build a container image for the function and add it to the repository.
   ```
   $ fn -v deploy --app DP-APP
   ```
   Note: This build and deploy process can take up to three minutes to complete.
3. Meanwhile (during the build), go ahead and open **Code Editor**.
4. Navigate to `/labs/oci-functions/event-fn/src/main/java/com/example/fn` and open the `HandleImageMetadata.java` file to examine the function's source code.
   - **Line 7** is the import statement for the `CloudEvent` class which conforms to the CNCF Cloud Events specification.
   - Within the function's **handleRequest** method, **lines 17-19** contain the logic for creating an `ObjectMapper` that converts the `CloudEvent` JSON object message into `Map` data.
   - **Lines 23-25** build the URL for the new image file referenced from the received `CloudEvent`.
   - **Line 27** retrieves the image file from Object Storage followed by **line 28** which uses the `ImageMetadataReader` class to capture all of the image file's metadata.

> **Note:** For a real-world use case, at this point, you would add additional logic to do something with that metadata (such as updating a database or invoking some other downstream system). Instead, **line 29** simply prints the metadata to stdout.

In a moment, you will enable logging so that you can see that output when testing this function.

5. From the Console, open the navigation menu and click **Developer Services**. Under **Functions**, click **Applications**.

6. Select your application **DP-APP**. In the **Resources** section, click **Logs**, then toggle the **Enable Log** button for Function Invocation Logs.

7. Accept the default values and click **Enable Log**.

Create an Event Rule in OCI Events

You will create an Event Rule in OCI Events that fires when an image file is uploaded to the public bucket you created earlier, then triggers the **events-fn** function you recently deployed.

Tasks

1. Click the navigation menu and click **Observability & Management.** Under **Events Service**, click **Rules**.

2. Click **Create Rule**.
   - **Display Name**: `image-events`
   - **Description**: `Invokes a function when an image is uploaded`
   - **Rule Conditions**:
     - **Condition**: `Event Type`
     - **Service Name**: `Object Storage`
     - **Event Type**: `Object-Create`

3. Click the **+Another Condition** button. For **Condition**, select `Attribute`.
   - **Attribute Name**: `bucketName`
   - **Attribute Values**: type in the name of your bucket.

     For example: `david-images`

     Note: You will need to press the Enter key to enter this value.

4. Within the **Actions** pane, for **Action Type** select `Functions`.
   - **Function Compartment**: Expand the root compartment, then locate and select your assigned compartment.
   - **Function Application**: `DP-APP`
   - **Function**: `event-fn`

5. Click **Create Rule**.

Test the Event-Driven Use Case

Now that the function is deployed and the Event rule has been configured, in this practice, you will test this functionality. After downloading sample image files to your local computer, you will upload these files to your bucket. You will then verify that the function was executed by viewing the log messages in the OCI Logging service.

Tasks

1. Return to open the **Code Editor** by clicking the Code Editor icon located in the upper-right section of the OCI Console.

2. Navigate to `~/labs/oci-functions/event-fn`.
   - Right-click `oci-events-image.png` and click **Download**. (Save this file to a convenient location on your local computer.)
   - Right-click `oci-functions-image.png` and click **Download**. (Save this file to a convenient location on your local computer.)

3. In the Console, click the navigation menu, then click **Storage.** Under **Object Storage & Archive Storage**, click **Buckets**.

4. Click the link for your `<`*`userID`*`>-images` bucket.

5. Under the **Objects** section, click the **Upload** button.

6. Within the *Choose Files from your Computer* section, drag and drop those two image files (`oci-events-image.png` & `oci-functions-image.png`) then click **Upload**.

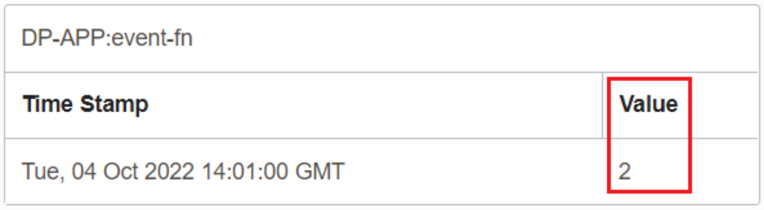
7. After they are uploaded, click **Close**.

8. To verify that the OCI Events service has executed the **event-fn** function, go to the navigation menu, and click **Developer Services**. Under **Functions** click **Applications**.

9. Click **DP-APP**, and then under **Functions**, click the **event-fn** link.

10. In the **Metrics** section, under **Invocations**, use the **Options** drop-down to select **Table View**.

Note: The entry for **DP-APP:event-fn** should indicate a **Value** of 2.



DP-APP:event-fn

Time Stamp	Value
Tue, 04 Oct 2022 14:01:00 GMT	2

11. To view the log entries, go back to **DP-APP** or click the DP-APP link in the breadcrumbs located in the upper-left of the page.

12. Click **Logs** under the **Resources** section on the left pane.

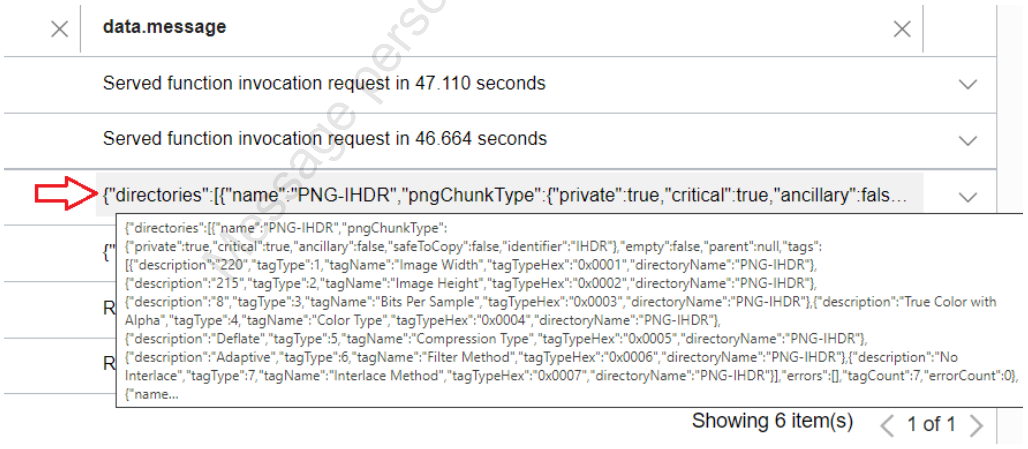
13. Click the link for the **DP_APP_invoke** log under **Log Name**.

    Note: This will open the OCI Logging service directly to the **DP_APP_invoke** log page.

14. In the **Explore Log** section, you should see entries under **data.message** that was added as the result of the two function invocations:
    - `Received function invocation request`
    - `{"directories":[{"name":"PNG-IHDR",…`
    - `Served function invocation request in <> seconds`

    Note: If it has been more than 5 minutes since you uploaded the image files. You will need to change the **Filter by time** value to the *Past 15 minutes* or *Past hour*.

15. To view the image file metadata that was sent to `stdout` from the function, hover your mouse over the log message as shown:



data.message

```
Served function invocation request in 47.110 seconds
Served function invocation request in 46.664 seconds
{"directories":[{"name":"PNG-IHDR","pngChunkType":{"private":true,"critical":true,"ancillary":fals...
```

```
{"directories":[{"name":"PNG-IHDR","pngChunkType":
{"private":true,"critical":true,"ancillary":false,"safeToCopy":false,"identifier":"IHDR"},"empty":false,"parent":null,"tags":
[{"description":"220","tagType":1,"tagName":"Image Width","tagTypeHex":"0x0001","directoryName":"PNG-IHDR"},
{"description":"215","tagType":2,"tagName":"Image Height","tagTypeHex":"0x0002","directoryName":"PNG-IHDR"},
{"description":"8","tagType":3,"tagName":"Bits Per Sample","tagTypeHex":"0x0003","directoryName":"PNG-IHDR"},{"description":"True Color with
Alpha","tagType":4,"tagName":"Color Type","tagTypeHex":"0x0004","directoryName":"PNG-IHDR"},
{"description":"Deflate","tagType":5,"tagName":"Compression Type","tagTypeHex":"0x0005","directoryName":"PNG-IHDR"},
{"description":"Adaptive","tagType":6,"tagName":"Filter Method","tagTypeHex":"0x0006","directoryName":"PNG-IHDR"},{"description":"No
Interlace","tagType":7,"tagName":"Interlace Method","tagTypeHex":"0x0007","directoryName":"PNG-IHDR"}],"errors":[],"tagCount":7,"errorCount":0},
{"name...
```

Congratulations! You created an Event rule in the OCI Events service that automatically fires when a new image file is uploaded to an OCI Object Storage bucket, then sends that `CloudEvent` message to an Oracle Function.

API Management: Create an API Gateway Deployment

Lab 08-1 Practice

Estimated Time: 30 Minutes

Setup Instructions for the Lab

IMPORTANT: If you are using a tenancy in which you do not have administrator permissions, you will need to have the administrator add these additional policy statements:

Compartment level:

```
Allow group <group-name> to manage api-gateway-family in
compartment <compartment-name>
Allow group <group-name> to use api-certificates in compartment
<compartment-name>
```

Note: If you are using your own tenancy you will not need to add any group permission policy statements.

VERY IMPORTANT: An additional policy statement is needed to allow the API Gateway service access to Oracle Functions. This is still required regardless of what type of user permissions you have.

- If you are using a tenancy in which you do not have administrator permissions, you will need to have the administrator add this additional policy statement.
- If you are using your own tenancy, simply use your initial tenancy administrator credentials to add this policy statement.

Compartment level:

```
allow any-user to use functions-family in compartment
<compartment-name> where ALL {request.principal.type =
'ApiGateway'}
```

Get Started

Overview

Oracle Cloud Infrastructure (OCI) API Gateway makes it possible to expose OCI Functions on public endpoints that do not require complex signed HTTP requests. Any function that should be easily publicly accessible can be given such easy access by creating an API deployment on an API Gateway and associating a route in that API Deployment with an Oracle Function Backend.

In this practice, you will expose a function previously deployed to a private subnet to be accessed via the API Gateway.

The instructions will be organized into the following five practice sections:

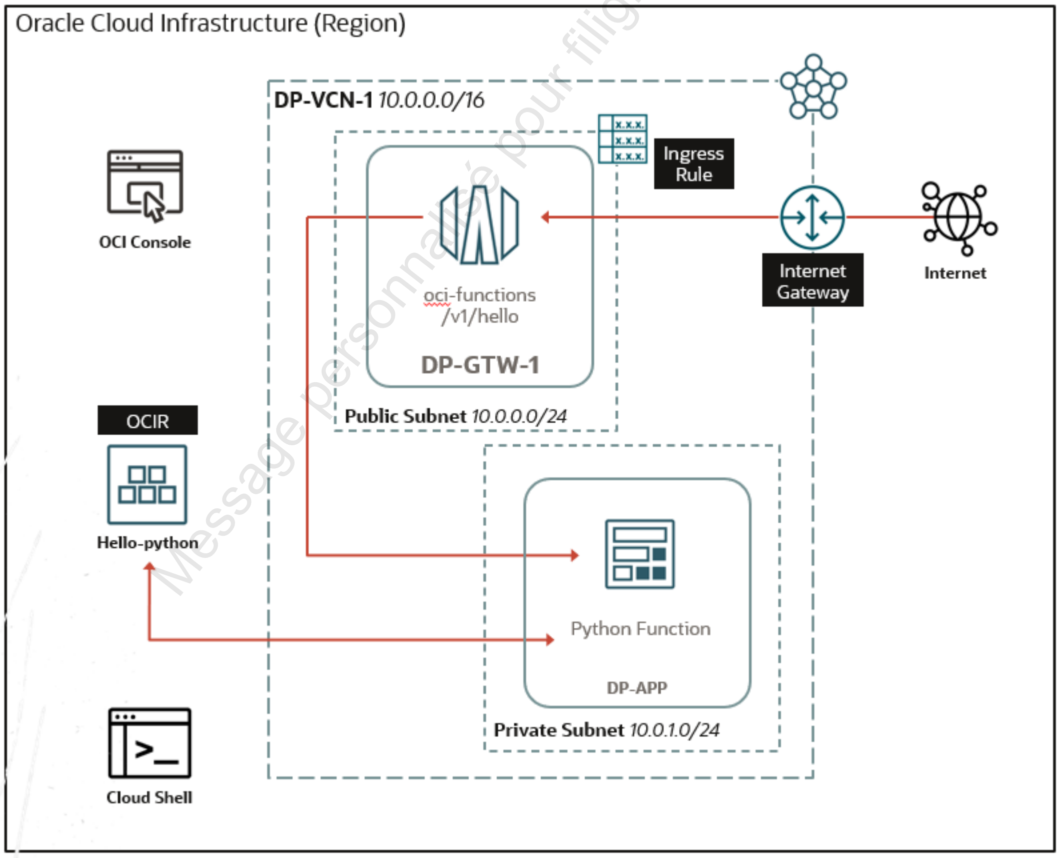
a. Create a new API Gateway.

b. Create a new API Gateway deployment.

c. Validate a policy statement that allows API Gateway to access the function.

d. Add an ingress rule for the public subnet.

e. Call the Function via your API Gateway deployment.



Prerequisites

You have completed **Lab 4-1** (*Serverless Functions: Build and Deploy an Oracle Function*).

Assumptions

- You are signed into your Oracle Cloud Infrastructure (OCI) account using your credentials.
- You will replace the `<userID>` placeholder with your first name.

Create a New API Gateway

You can use a single API Gateway to link multiple back-end services (such as load balancers, compute instances, and Oracle Functions) into a single consolidated API endpoint.

You will create an API Gateway that will later be used to create an API Gateway deployment to call one or more functions**.**

Tasks

1. Open the navigation menu and click **Developer Services**. Under **API Management**, click **Gateways**.
1. Select your assigned compartment from the **Compartment** drop-down list.
2. Click **Create Gateway.**
3. Fill in the following information to define your API Gateway:
   - **Name:** `MyAPIGateway`
   - **Type:** `Public`
   - **Compartment:** `<your-compartment-name>`
   - **Virtual Cloud Network:** `DP-VCN`

     This is the VCN that you created in the *Serverless Functions: Build and Deploy an Oracle Function* practice (Lab 4-1).
   - **Subnet:** `Public Subnet-DP-VCN`
4. Click **Create**. Wait a few minutes for API Gateway to be created.

Create a New API Gateway Deployment

Having used the API Gateway service to create an API Gateway, you can now create an API Deployment that invokes serverless functions defined in Oracle Function.

You will create a new API Deployment for your API Gateway named oci-functions (using **/v1** as the path prefix) in `MyAPIGateway` and create a new Route (using **/hello** as the path; selecting **GET** as the method) that invokes the **hello-python** function.

Tasks

1. On the **Gateways** page, click the name of the API gateway you just created, for example, MyAPIGateway.
2. On the **Gateway Details** page, select **Deployments** from the **Resources** list and then click **Create Deployment**.
3. Click **From Scratch** and fill in the **Basic Information** section:
    - **Name**: `oci-functions`
    - **Path Prefix**: `/v1`

      *Note that the deployment path prefix you specify must be preceded by one or multiple forward slashes but must not end with it. It can include alphanumeric uppercase and lowercase characters, and special characters like $ - _ . + ! * ' ( ) , % ; : @ & =, and must not include parameters and wildcards.*
    - **Compartment:** `<your-compartment-name>`
4. Click **Next** to display the **Authentication** page and select **No Authentication** to give unauthenticated access to all routes in the API deployment.
5. Click **Next** to enter details of the routes in the API deployment and edit **Route 1** to specify the first route in the API deployment that maps a path and one or more methods to a back-end service:
    - **Path**: `/hello`
    - **Methods**: `GET`
    - Select **Add a single backend**.
    - **Type**: `Oracle Functions`
    - **Application**: `DP-APP`
    - **Function Name**: `hello-python`

Where:

- ***Path** refers to the path for the API calls using the listed methods to the back-end service*
- ***Methods** refer to one or more methods accepted by the back-end service*
- ***Type** refers to the type of the back-end service*
- ***Application** refers to the name of the application in Oracle Functions that contains the function*
- ***Function Name** refers to the name of the function in Oracle Functions*

6. Click **Next** to review the details you entered for the new API deployment.

7. Click **Create** to create the new API deployment.

Note that it can take a few minutes to create the new API deployment.

Validate a Policy Statement That Allows API Gateway to Access the Function

Verify that an IAM policy statement is present in your compartment that allows API Gateway to access Oracle Functions in your compartment.

Tasks

1. Open the navigation menu and click **Identity & Security**. Under **Identity**, click **Policies**.
2. Ensure you are in your assigned compartment, then click the existing policy link.
3. Verify that the following policy statement is present:

```
allow any-user to use functions-family in compartment
[compartment-name] where ALL {request.principal.type =
'ApiGateway'}
```

   Note: This requirement is also listed in the **Setup Instructions for the Lab** shown in the beginning of this document.

Add an Ingress Rule for the Public Subnet

You will create a new stateful CIDR Ingress Rule that allows TCP HTTPS traffic (port 443) from all IP addresses and ports in the default Security List for the virtual network. It will also allow you access from your Cloud Shell.

Tasks

1. Open the navigation menu, click **Networking**, and then click **Virtual Cloud Networks**.
2. Click the VCN that you created earlier, for example, **DP-VCN.**
3. Under **Resources**, click **Security Lists**.
4. On the **Security List** page, click the **Default Security List for DP-VCN** link.
5. Click **Add Ingress Rule**. Choose whether it's a stateful or stateless rule. By default, rules are stateful unless you specify otherwise. Enter the other basic information:
   - **Source Type**: `CIDR`
   - **Source CIDR**: `0.0.0.0/0`
   - **IP Protocol**: `TCP`
   - **Destination Port**: `443`
6. Click **Add Ingress Rule**. The rule will be added and you can see that in the Ingress Rules table.

Call the Function via Your API Gateway Deployment

With your API Gateway and deployment created, you can now call the Function via your API Gateway deployment.

You will use `curl` to call the function via your API Gateway deployment.

Tasks

1. To determine the deployment endpoint, navigate back to **Developer Services**, under **API Management**, and click **Gateways**.
2. Select your API Gateway **MyAPIGateway.**
3. Under the **Resources** section, click the **Deployments** link. Copy the **Endpoint URL** for **oci-functions** deployment.
4. Click the **Cloud Shell** icon at the right of the OCI console header to launch it.
5. In Cloud Shell, execute `curl -k -X GET` {endpoint-url}/`hello` to invoke the function. (Be sure to append /hello to the URL you copied earlier.)

   For example:

   ```
   curl -k -X GET https//mcflc33ojnikxwnkzb674ssxiy.apigateway.us-phoenix-1.oci.customer-oci.com/oci-functions/hello
   ```

   To create the URL for `curl`, add your deployment path to your endpoint.

   On successful execution, it will return: **{“message”: “Hello World”}.**

Congratulations! You have successfully exposed an OCI Function on a public endpoint for easy access.

API Management: Manage APIs with API Gateway

Lab 8-2 Practices

Estimated Time: 30 minutes

Get Started

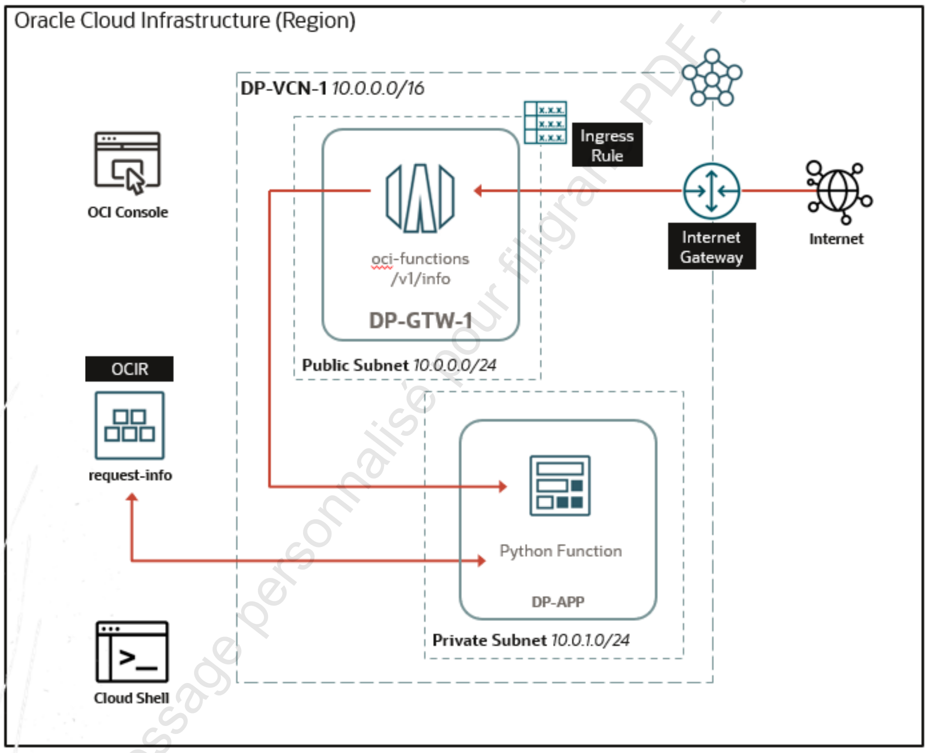
Overview

In this practice, you will create and test a new API Gateway deployment route that applies additional request and response policies. This practice will be segmented into three practice sections:

a. Deploy a new back-end function to an existing Fn Application that will be used for the new API Gateway deployment route.

b. Create a new route to access that function to an existing API Gateway deployment and test.

c. Update various policies on the route and test the results of those configurations.



Prerequisites

You have completed **Lab 8-1** (*API Management: Create an API Gateway Deployment*).

Assumptions

- You are signed into your Oracle Cloud Infrastructure (OCI) account using your credentials.
- You will replace the `<userID>` placeholder with your first name.

Deploy a Function to an Existing Fn Application

To facilitate the essential tasks for this practice, you will deploy another function to your existing Fn application used in earlier practice.

You'll create a new private repository to store the function image once created. You'll then create the image by deploying the function to the Fn application.

Tasks

1. In the Console, open the navigation menu and click **Developer Services**. Under **Containers & Artifacts**, click **Container Registry**.

2. Click the **Create repository** button.

3. For the Repository name, enter `<userID>/request-info`

   where *<userID>* is your first name and is also the [repo-prefix] used for Function deployments you set earlier in **Lab 4-1**.

   For example: `david/request-info`

4. Ensure Access is set to **Private**. Click **Create repository**.

5. Navigate to the directory containing the **request-info** function code.
   ```
   $ cd ~/labs/oci-functions/request-info
   ```

6. Use the `fn deploy` command to build a container image for the function and add it to the OCIR repository. *(This may take up to 60 seconds).*
   ```
   $ fn -v deploy --app DP-APP
   ```

7. Use the `fn invoke` command to execute the function. You will want to pipe this command to the **jq** command to view the JSON response in a more readable format. *(This may take up to 30 seconds).*
   ```
   $ fn invoke DP-APP request-info | jq
   ```

If successful, a JSON result will be returned listing the HTTP request information to include headers and configuration information of the function.

Create a New Route on an Existing API Gateway Deployment

Before uploading and deploying a container image containing the function code, you need to specify a private repository in the Container Registry that is within the same OCI region of the Function Application.

Tasks

1. In the Console, open the navigation menu and click **Developer Services**. Under **API Management**, click **Gateways**.

2. Click the link for your gateway: **MyAPIGateway**

3. Under **Resources** in the left pane, click **Deployments**.

4. Click the **oci-functions** deployment link, and then click the **Edit** button.

5. You'll leave the basic information, API request, and logging properties the same – click **Next**.

6. You'll also continue to use **No Authentication** for this deployment – click **Next**.

7. Minimize the **Route 1** box, and then click the **+ Another route** box below.

8. Edit the following information for Route 2:
   - **Path:** `/info`
   - **Methods:** `ANY`
   - Select **Edit added single backend**.
   - **Backend Type:** `Oracle Functions`
   - **Application:** `DP-APP`
   - **Function name:** `request-info`

   Click **Next**. Click **Save changes**.

9. To test this new deployment route, open Cloud Shell (if not already open) and export a variable named **APIGW** with the value of your deployment **Endpoint**.

   Hint: Copy the **Endpoint** located in the Deployment information box (for example):

```
$ export APIGW=https://mcflc33ojnikxwnkzb674ssxiy.apigateway.us-phoenix-1.oci.customer-oci.com/v1
```

10. Invoke the **/info** path on the deployment using the **`curl`** command, then pipe to the **`jq`** command to view the JSON response in a more readable format. *(This may take up to 30 seconds).*

    ```
    $ curl $APIGW/info | jq
    ```

    If successful, a JSON result will be returned listing the HTTP request information to include headers and configuration information of the function.

11. Notice the difference in some of these values as compared to invoking the function directly. To compare, execute a direct call to the function as before.

    ```
    $ fn invoke DP-APP request-info | jq
    ```

Create New Policies on an API Gateway Deployment Route

Now that you have a defined route in our deployment to invoke the back-end function, you can experiment with adding some policies to the deployment and to the route. In this section, you will:

- Add an API Request Policy to configure Rate Limiting on the entire deployment
- Add Route Request Policies for `/info` that appends a new HTTP header to the request and a new HTTP query parameter to the request
- Add a Route Response Policy for `/info` that appends a new HTTP header to the response
- Test and validate the updates to the deployment

Tasks

1. In the Console, open the navigation menu and click **Developer Services**. Under **API Management**, click **Gateways**.
2. Click the link for your gateway: **MyAPIGateway**
3. Under **Resources** in the left pane, click **Deployments**.
4. Click the **oci-functions** deployment link, then click the **Edit** button.
5. Scroll down under API request policies, located **Rate Limiting** – click **Add**.
6. Enter **1** for the *Number of requests per second* and choose **Total** for the *Type of rate limit*.

   Click **Apply changes**.
7. Click **Next**.
8. We'll continue to use **No Authentication** for this deployment – click **Next**.
9. Minimize the Route 1 box. Within the Route 2 box, scroll down and expand **Show route request policies**.
10. Scroll down to **Header transformations** – click **Add**.

11. Define the following values:
    - **Action:** `Set`
    - **Behavior:** `Append`
    - **Header name:** `oulab-extra-header`
    - **Values:** Type in your name (or any other value) – press **Enter**.

    Click **Apply changes**.

12. Next within the Query parameter transformations box, click **Add**.

13. Define the following values:
    - **Action:** `Set`
    - **Behavior:** `Append`
    - **Query parameter name:** `oulab`
    - **Values:** Type in the day of the week (for example, `Monday`) – press **Enter**.

    Click **Apply changes**.

14. Moving on – now expand **Show route response policies**.

15. Within the Header transformations box, click **Add**.

16. Define the following values:
    - **Action:** `Set`
    - **Behavior:** `Append`
    - **Header name:** `oulab-extra-header`
    - **Values:** Type: `hello from the gateway` – press **Enter**.

    Click **Apply changes**.

17. Click **Next**. Click **Save changes**.

    Note: The updates may take up to 60 seconds to complete.

18. To test these configuration changes, return to Cloud Shell. If the session expired, you need to export a variable named **APIGW** with the value of your deployment **Endpoint**:

    Hint: Copy the **Endpoint** located in the Deployment information box (for example):

```
$ export APIGW=https://mcflc33ojnikxwnkzb674ssxiy.apigateway.us-phoenix-1.oci.customer-oci.com/v1
```

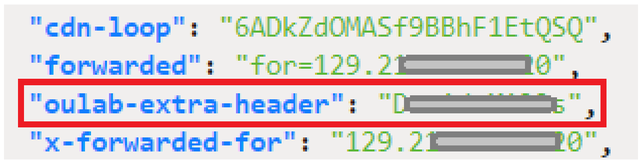
19. Invoke the **/info** path on the deployment using the **`curl`** command, then pipe to the **`jq`** command to view the JSON response in a more readable format. *(This may take up to 30 seconds.)*

```
$ curl $APIGW/info | jq
```

If successful, a JSON result will be returned listing the HTTP request information to include headers and configuration information of the function.

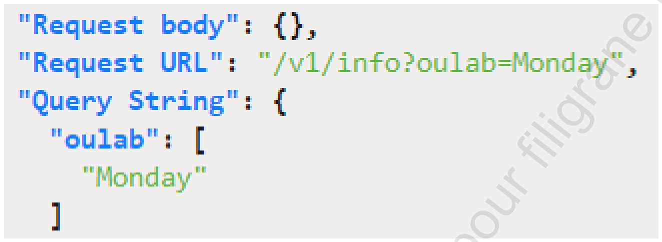
20. If your configuration was done correctly, you should see the following:

- An appended request header with a value. For example:



```
"cdn-loop": "6ADkZdOMASf9BBhF1EtQSQ",
"forwarded": "for=129.21[redacted]0",
"oulab-extra-header": "D[redacted]s",
"x-forwarded-for": "129.21[redacted]20",
```

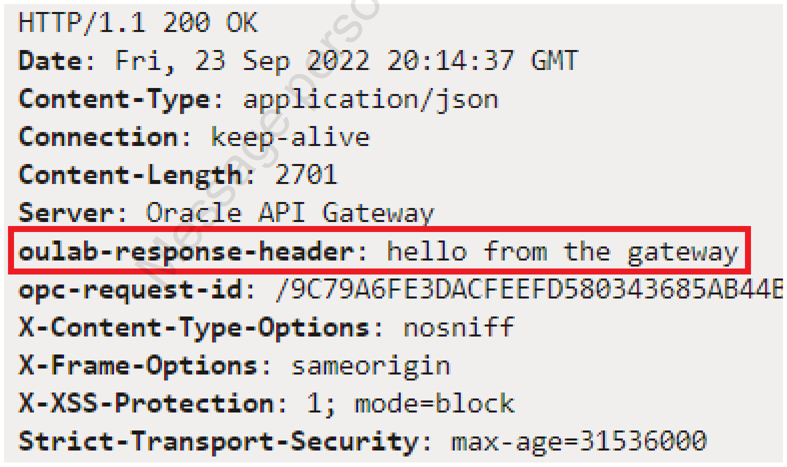
- An appended query string to the Request URL. For example:



```
"Request body": {},
"Request URL": "/v1/info?oulab=Monday",
"Query String": {
  "oulab": [
    "Monday"
  ]
```

21. To view the appended response header, invoke the endpoint again using the curl -i flag (however, this time do not attempt to pipe to the jq command).

```
$ curl -i $APIGW/info
```



```
HTTP/1.1 200 OK
Date: Fri, 23 Sep 2022 20:14:37 GMT
Content-Type: application/json
Connection: keep-alive
Content-Length: 2701
Server: Oracle API Gateway
oulab-response-header: hello from the gateway
opc-request-id: /9C79A6FE3DACFEEFD580343685AB44E
X-Content-Type-Options: nosniff
X-Frame-Options: sameorigin
X-XSS-Protection: 1; mode=block
Strict-Transport-Security: max-age=31536000
```

Congratulations! You have created and tested a new route configuration on an API Gateway deployment that included rate limiting as well as request and response policies.

API Management: Create a Deployment for a Mock API

Lab 8-3 Practices

Estimated Time: 30 Minutes

Get Started

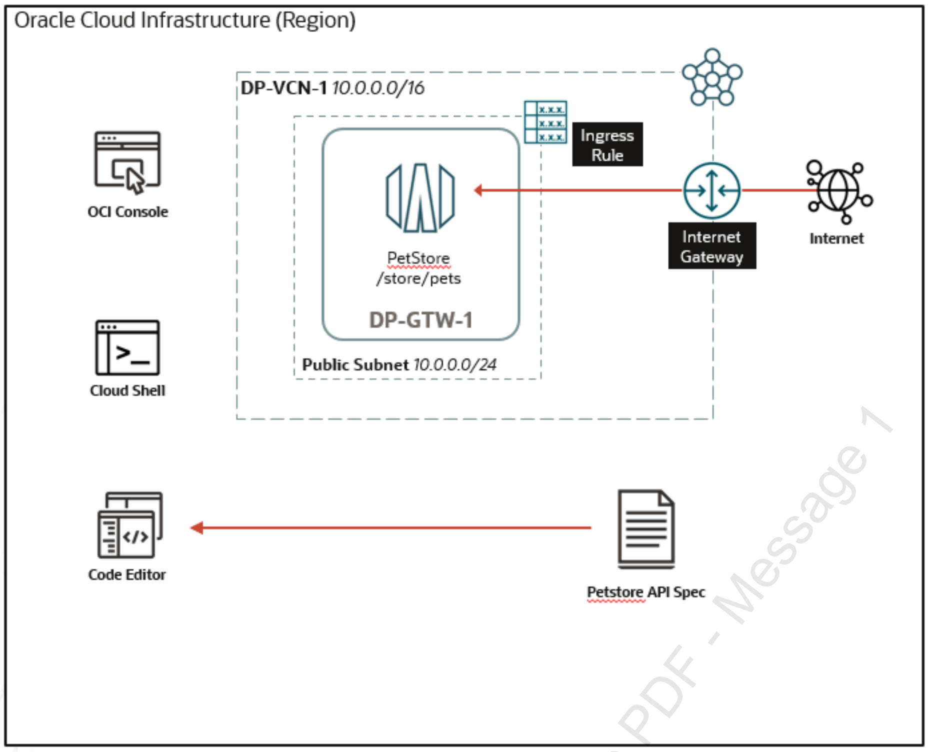
Overview

The introduction of an API-first approach coincides with the rise of microservices. API-first means that you design each of your APIs around a contract written in an API description language such as OpenAPI for consistency, reusability, and broad interoperability. Starting with an API contract allows you to better plan the design of your API and get feedback from stakeholders before writing any code.

Before developing web service implementations or writing client application code to access those web service REST APIs, an architect will define the API using an API design tool such as Oracle Apiary or SwaggerHub. Once designed, an API contract is provided to both the API web service implementers as well as potential client application developers in an API specification format (typically OpenAPI or API Blueprint).

In this practice, you'll be provided an API contract that has been designed and created by an architect for a future Pet Store application to be developed later. You'll use this API document to create an Oracle Cloud Infrastructure (OCI) API Gateway Deployment, which has routes for each method identified in the API. Instead of routing to an implemented back-end service, the deployment will return a static stock response you've specified.

The purpose of this API Gateway deployment is to allow client application developers to implement and test their code logic to invoke these APIs. Once the Pet Store application developers have completed their web service REST API implementation, the API Gateway deployment configuration can then be updated to forward the requests to the Pet Store application instead of returning mock responses to client applications.



In this practice, you will:

a. Preview the PetStore OpenAPI document description using Code Editor
b. Add the API to the API Gateway service and create a new deployment
c. Edit the deployment routes to add JSON response samples
d. Validate the mock API deployment

Prerequisites

You have completed **Lab 8-1** (*API Management: Create an API Gateway Deployment*).

Assumptions

- You are signed into your Oracle Cloud Infrastructure (OCI) account using your credentials.
- You will replace the `<userID>` placeholder with your first name.

Preview the PetStore OpenAPI Document Description

Before uploading and deploying the Pet Store mock API, you need to view and understand the API document that has been defined with the OpenAPI v3 specification.

Tasks

1. Open **Code Editor**.
2. From within the **Code Editor,** navigate to the `~/labs/oci-functions/API-specs` directory.
3. Click to open the `petstoreAPIv3.yaml` file.
   a. Press **Ctrl/Cmd + Shift + P** on your keyboard to open the command palette.
   b. In the search bar, enter `OpenAPI` then select **OpenAPI:View** from the drop-down list. *You may also use a filter to search for the item*.

      After several seconds, an **OpenAPI View** window will open next to the window displaying the yaml file.
4. Review the API's four-method detailed specifications.
5. In the **Code Editor**, go to the **File** menu and click **Download** to download the `petstoreAPIv3.yaml` file to your computer.

   Save it to a convenient location because you will be uploading this file in a later task step to the API Gateway service.

Add the API to the API Gateway Service and Create a New Deployment

When using the API Gateway service, you have the option to create an API artifact. You already have an API description file (`petstoreAPIv3.yaml`) to create the artifact, which can then be used to create a new deployment on an API gateway. This deployment will include a route for each defined API resource method and pre-populate most of the deployment properties from the API description file.

Tasks

1. Go to the navigation menu and click **Developer Services**. Under **API Management**, click **APIs**.
2. Click **Create API**.
   - **Name**: `PetStoreAPI`
   - **Compartment**: `Select your` *`<assigned compartment>.`*
   - **Upload API description file (Swagger, OpenAPI 3.x specification)**: Drag and drop the previously downloaded `petstoreAPIv3.yaml` file from your computer.
3. Click **Create API** to create the artifact.

   With the uploaded file, the API is created and validated. Note that it can take a few minutes to validate the API description. Wait for **Validation: Valid** before moving to the next step.
4. On the newly created **PetStoreAPI** page, click **Deploy** to create a new API deployment.

   Most of the initial values for the API deployment specification properties shown in the **Create deployment from API wizard** are derived from the API description file.
5. In **Gateway**, select your API gateway (**MyAPIGateway**) to create and add this new deployment. This is the gateway that was created in Lab 8-1.
6. In the Basic Information section, specify:
   - **Name**: `PetStore`
   - **Path Prefix**: `/store`
   - **Compartment**: select your *`<assigned compartment>`*
7. Leave everything else to the default settings and click **Next** to review the details of the 4 routes in the Deployment.

By default, a route is created for every path and associated method that is present in the API description. Initially, each of these default routes is created with a Stock response back-end type. The relative URI path, HTTP request method, HTTP response status code, and response HTTP headers are obtained from the details in the API description file.

8. Review each route. You'll see four routes created from the description file.
   - **Path:** Corresponds to the relative URI used for that API resource
   - **Methods:** The corresponding HTTP method used for that API resource
   - **Type:** `Stock response` *(which indicates that a static response will be returned to the client instead of forwarding the request to an implemented back-end service)*

   The corresponding HTTP **Status code**, **Header name**, and **Header value** are populated for each API resource as defined by the API document. For the HTTP response **Body**, you will add a sample response message that corresponds to the API description file in later steps. Leave this blank for now.

9. Click **Next** to review the deployment details.

10. Click **Submit** to create the new API deployment.

    Note: It can take a few minutes to create the new API deployment. While it is being created, the API deployment is shown with a state of **Creating** on the **Gateway details** page. When it has been successfully created, the deployment is shown with a state of **Active**.

Edit the Deployment Routes to Add JSON Response Samples

Now that the new API Deployment has been added to the gateway, you need to further edit each route to return the corresponding body of the HTTP response. On receiving a request to the relative URI path of that deployment, the API gateway itself will serve as the mock back end, returning a stock response to include the HTTP response body that you've specified.

Tasks

1. Open the `petstoreAPIv3.yaml` file in Code Editor.
   a. Press **Ctrl/Cmd + Shift + P** on your keyboard to open the command palette.
   b. Select >**OpenAPI:View** from the drop-down list.

2. Switch to the OCI console and click **PetStore** under **Deployments** on the Gateway details page.

   Tip: You can open the OCI Console on a different tab so that you can copy-paste the response code easily by switching tabs while editing the deployment.

3. Click **Edit** on the PetStore Deployment details page.

4. Click **Routes** in the left navigation pane on the Edit deployment page.

   Tip: Collapse each of the four route boxes for more readability and easy access.

5. Expand **Route 1**.
   a. Go to Code Editor and in the **OpenAPI: View**, copy the status code 200 response sample for **findPets**. You can use the **Copy** link to copy the JSON object string in the **Response samples** box.
   b. Switch to the OCI Edit deployment page and in **Route 1**, paste the string into the **Body** text field.

```
[
  {
    "name": "string",
    "tag": "string",
    "id": 0
  }
]
```

6. Collapse **Route 1** and expand **Route 2**.

   a. Go to Code Editor again and copy the status code 201 response sample for **addPet**.

   b. Switch to the OCI Edit deployment page and in **Route 2**, paste the string into the **Body** text field.

```
{
  "name": "string",
  "tag": "string",
  "id": 0
}
```

7. Collapse **Route 2** and expand **Route 3**.
   a. Go to Code Editor again and copy the status code 200 response sample to **find pet by id**.
   b. Switch to the Edit deployment page and in **Route 3**, paste the string into the **Body** text field.

```
{
  "name": "string",
  "tag": "string",
  "id": 0
}
```

8. Collapse **Route 3** and expand **Route 4**.
   a. Go to Code Editor again to look at **deletePet**.
   b. Notice that the response sample is only for the **default** case (which is for unexpected errors). Since the HTTP response code is defined as 204, there will be no HTTP body returned when the DELETE method was successfully executed by the web service.
   c. In this case, you will leave the **Body** text field empty for **Route 4**.

9. Click **Next** and click **Save Changes** to update your API deployment.

   It will take a few seconds to complete the update. In the next practice, you will confirm the API deployment configuration by making HTTP requests to each of the API methods.

Validate the Mock API Deployment

Having deployed a mock API deployment on an API gateway, you can call the deployed API methods and verify the responses.

Tasks

1. You'll need the Endpoint URL when you call the mock API.
   a. On the **Gateway details** page, under **Deployments**, click **PetStore** that you created.
   b. On the PetStore Deployment details page, copy the **Endpoint** URL and paste it into your Notepad. You'll need this when you call your API.

2. Open **Cloud Shell**.

3. Validate the response for the **findPets** API resource using **curl**:

```
$ curl -i -X GET <Endpoint URL>/pets
```

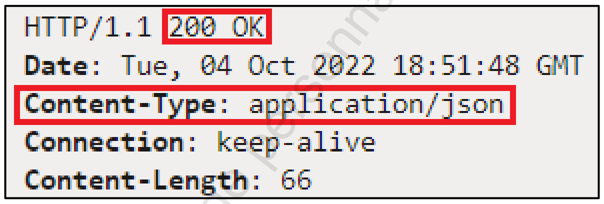
Replace *<Endpoint URL>* with the URL you copied.

For example:

```
$ curl -i -X GET
https://nmcnrqtf7dsgnxxxxxxxxxxxxx.apigateway.uk-london-
1.oci.customer-oci.com/pet/pets
```

Verify the following as defined in your Stock response:

- The HTTP Status Code is 200 and the HTTP Header Content-Type is `application/json`.



```
HTTP/1.1 200 OK
Date: Tue, 04 Oct 2022 18:51:48 GMT
Content-Type: application/json
Connection: keep-alive
Content-Length: 66
```

- The HTTP body is defined for that route.

```
[
  {
    "name": "string",
    "tag": "string",
    "id": 0
  }
]
```

4. Validate the response for the **addPet** API resource using **curl**:

   ```
   $ curl -i -X POST <Endpoint URL>/pets
   ```

   For example:

   ```
   $ curl -i -X POST
   https://nmcnrqtf7dsgnxxxxxxxxxxxxx.apigateway.uk-london-
   1.oci.customer-oci.com/pet/pets
   ```

   Verify the following as defined in your Stock response:

   - The HTTP Status Code is 201 and the HTTP Header Content-Type is `application/json`.
   - The HTTP body as defined for that route

   ```
   {
     "name": "string",
     "tag": "string"
   }
   ```

5. Validate the response for the **find pet by id** API resource using **curl**:

   ```
   $ curl -i -X GET <Endpoint URL>/pets/{id}
   ```

   For example:

   ```
   $ curl -i -X GET
   https://nmcnrqtf7dsgnxxxxxxxxxxxxx.apigateway.uk-london-
   1.oci.customer-oci.com/pet/pets/1234
   ```

   Note: You could use any value as the ID used for the path parameter. The example above uses 1234.

   Verify the following as defined in your Stock response:

   - The HTTP Status Code is 200 and the HTTP Header Content-Type is `application/json`.
   - The HTTP body is defined for that route.

   ```
   {
     "name": "string",
     "tag": "string",
     "id": 0
   }
   ```

6. Validate the response for the **deletePet** API resource using **curl**:

   ```
   $ curl -i -X DELETE <Endpoint URL>/pets/{id}
   ```

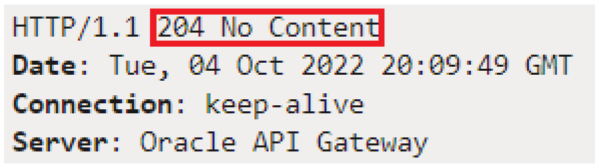
For example:

```
$ curl -i -X DELETE
https://nmcnrqtf7dsgnxxxxxxxxxxxxx.apigateway.uk-london-
1.oci.customer-oci.com/pet/pets/1234
```

Note: You could use any value as the ID used for the path parameter. The example above uses `1234`.

Verify the following as defined in your Stock response:

- The HTTP Status Code is 204



```
HTTP/1.1 204 No Content
Date: Tue, 04 Oct 2022 20:09:49 GMT
Connection: keep-alive
Server: Oracle API Gateway
```

You won't see any HTTP body response since the status code is 204. This indicates to the client of the API that the request has succeeded to delete the identified resource.

Congratulations! You have successfully created a mock API implementation, which can now be used to test client applications that need to consume the Pet Store Rest APIs. When you call an endpoint for a mock implementation, it returns the request sample that you provided in the Body field along with the appropriate HTTP status code and HTTP headers.

To review, the purpose of this API Gateway deployment is to allow client application developers to implement and test their code logic to invoke these APIs. Once the Pet Store application developers have completed their web service REST API implementation, the API Gateway deployment configuration can then be updated to forward the requests to the Pet Store application instead of returning mock responses to client applications.

Testing and Securing Cloud Native Applications: Unit Testing of Node Functions using Jest

Lab 9-1 Practices

Estimated Time: 20 minutes

Get Started

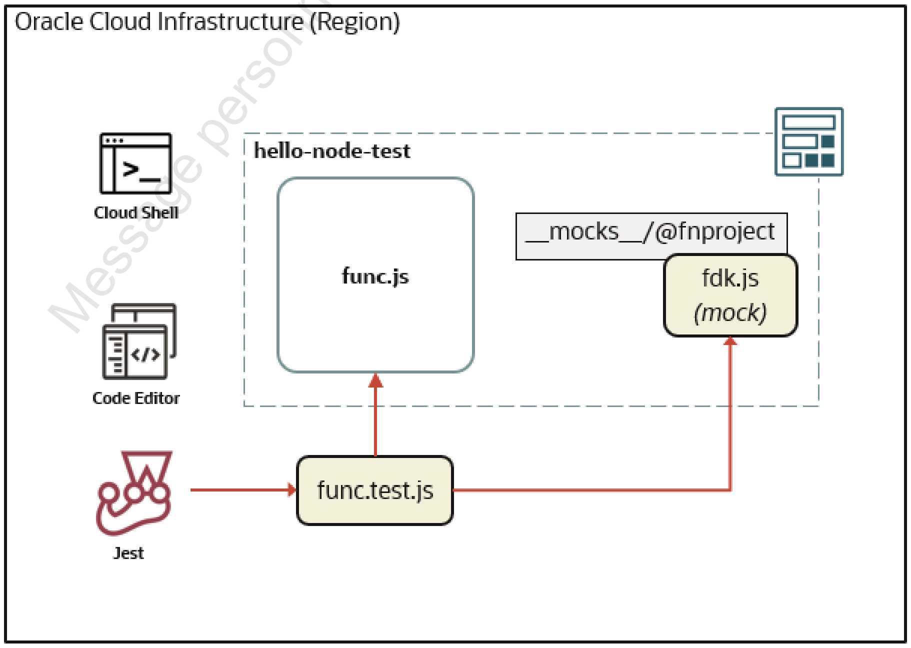
Overview

Unit tests that can be automated are essential to describe and prove the behavior of the software and to allow rapid and safe refactoring of the code. These tests should be executable without requiring the deployment of the code to a central environment and focus on the core logic. Ideally, unit testing should not require external dependencies nor the frameworks that wrap the code at run time.

This lab illustrates the use of Jest for writing tests for JavaScript-based functions and for mocking the Fn server environment using a simple example Hello World function, **hello-node-test**. Normally at run time, the function uses the Fn FDK framework to handle HTTP requests that are handed to the function for processing. However, in an automated testing scenario, we want to test outside the scope and context of the Fn framework and without any HTTP requests being made.

This can be achieved by using a mock for the FDK module. Jest allows us to define a mock in the following way:

- Create a file `(fdk.js)` in a subdirectory `(__mocks__/@fnproject)` under the function's parent directory.
- Implement the mock version of module `@fnproject/fdk` with a mock version of the function handle.
- Create a test file `(func.test.js)` that uses module `@fnproject/fdk` and executes tests against the function's implementation code `(func.js)`.



In this lab exercise, you will:

a. Examine mock function handle and Jest test code
b. Install Jest and execute unit testing

Prerequisites

You have completed **Lab 4-1** (*Serverless Functions: Build and Deploy an Oracle Function*).

Assumptions

- You are signed into your Oracle Cloud Infrastructure (OCI) account using your credentials.
- You will replace the *<userID>* placeholder with your first name.

Examine Mock Function Handle and Jest Test Code

In this practice, you will view the project files to include the mock function handle and the code that executes Jest tests.

Tasks

1. Open **Code Editor**.

2. Navigate to `/labs/oci-functions/hello-node-test` to review the project files.
   - Standard Node Function Files: `func.js`, `func.yaml`, `package.json`
   - Node Function Test Code: `func.test.js`
   - Mock Function Handler: `__mocks__/@fnproject/fdk.js`

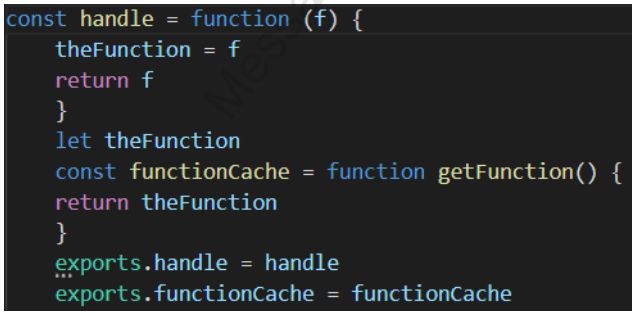
3. Open and view `package.json`.

   The following snippet has been added to `package.json`. This creates a new property at the same level as the `main` and `dependencies` which will cause `jest` to be invoked whenever the `npm test` is executed.

```
,"scripts": {
"test": "jest"
}
```

4. Open and view `__mocks__/@fnproject/fdk.js`.

   When `func.js` is required, it invokes a function handle on the fdk module and passes a function as an input parameter. This function is the actual Fn function implementation that handles the input and context objects to the Fn function. In the case of the mock fdk module, the handle function will simply retain the function reference in a local variable called `theFunction`.



```
const handle = function (f) {
    theFunction = f
    return f
    }
    let theFunction
    const functionCache = function getFunction() {
    return theFunction
    }
    exports.handle = handle
    exports.functionCache = functionCache
```

The test function (`func.test.js`) will invoke **functionCache** to retrieve the handle to this function and subsequently invoke it — just as it would be in the case of the normal invocation of the Fn function.

5. Open and view `func.test.js`.

   The test loads the object to test (**func.js**) and the mock (**@fnproject/fdk**) for the Fn FDK. The requirement of **func.js** causes the call in `func.js` to `fdk.handle()` to occur. This loads the reference to the function object defined in `func.js` in the **functionCache**. The test gets the reference to the function in the local variable **theFunction**. Two tests are defined:

   - Test #1: When the function is invoked with an object that contains a name property, does the response object contain a message property that has the specified value?
   - Test #2: When the function is invoked with a second object, (**context**) – does the response contain a **ctx** object?

6. On Line 6, change the value of the name to your name. For example:

   ```
   const name = "John Doe"
   ```

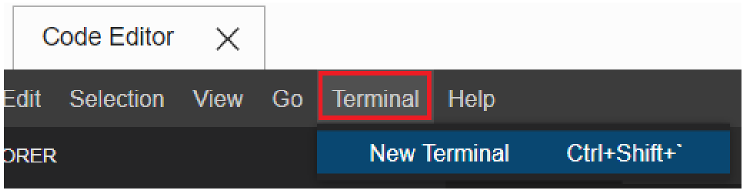
7. Use **Save** from the **File** menu to save the file.

Install Jest and Execute Unit Testing

In this practice, you will install the npm testing module (Jest), then use Jest to run the tests defined in the `func.test.js` file. After observing the test results, you will use this information to update the function code (`func.js`). Finally, you will rerun the tests to validate the successful execution of the function code.

Tasks

1. Open a terminal from the Code Editor menu bar.



2. Navigate to the directory containing the **hello-node-test** function code.

```
$ cd ~/labs/oci-functions/hello-node-test
```

3. Install the **Jest** npm testing module as a development time dependency.

```
$ npm install --save-dev jest
```

   Note: This may take up to 30 seconds to install. You can ignore any warnings for packages that are looking for `funding`.

4. Run the test.

```
$ npm test
```

5. The output should report on the tests for `func.js` (the real function implementation), but still without the runtime interference of the runtime Fn server.

   The first test passes, and the second one fails.

6. To pass this unit test, the function code (`func.js`) needs to be updated to allow for an added context parameter.

   Open and view `func.js`.

7. Follow the instructions in the code comments on lines 4 and 14.

   Essentially, you will remove or comment out the first and last lines of the existing handle method (lines 3 and 13), then substitute the valid implementation by uncommenting the correct code (lines 5 and 15).

8. Use **Save** from the **File** menu to save the file.

9. Run the test again.

   ```
   $ npm test
   ```

   This time, both tests should pass.

Congratulations! You have observed unit testing on a Node function by installing `Jest` and executing two tests defined using a mock `fdk` handle method.

Testing and Securing Cloud Native Applications: Scanning Container Images in OCIR

Lab 9-2 Practices

Estimated Time: 60 Minutes

Setup Instructions for the Lab

IMPORTANT: If you are using a tenancy in which you do not have administrator permissions, you will need to have the administrator add this additional policy statement:

Compartment level:

```
Allow group <groupname> to manage vss-family in compartment
<compartment name>
```

Note: If you are using your own tenancy you will not need to add any group permission policy statements.

VERY IMPORTANT: These additional service access policy statements are still required regardless of what type of user permissions you have.

- If you are using a tenancy in which you do not have administrator permissions, you will need to have the administrator add these additional policy statements.
- If you are using your own tenancy, simply use your initial tenancy administrator credentials to add these policy statements.

Compartment Level

```
Allow service vulnerability-scanning-service to read repos in
compartment <compartment name>

Allow service vulnerability-scanning-service to read
compartments in compartment <compartment name>
```

Get Started

Overview

Imagine a software development team working to deliver a business-critical application that passes sensitive data. A developer commits code to a continuous integration and continuous delivery (CI/CD) tool kicking off a build process. Then, the CI/CD tool pushes the newly built container image to an OCI Container Registry (OCIR) repository and when ready, the new image is deployed to a production OCI Container Engine for Kubernetes (OKE) cluster.

While this CI/CD process sounds reasonable, it is missing a few key steps. Critical to shipping compliant and secure containers, system administrators need to ensure that container images have the following characteristics:

- Are free of known critical vulnerabilities that can cause an accidental system failure or result in malicious activity.
- Have not been modified since they were published to maintain their integrity.
- Are only deployed to a Kubernetes cluster and come from a trusted source.

OCI container image scanning, signing, and verification address all these secure container deployment needs.

In this lab exercise, you will:

- Create a new OCIR repository
- Enable image scanning
- Sign in to OCIR from Cloud Shell and pull a sample Docker image from Docker Hub
- Tag and push the Docker Image to the OCIR repository
- View scan results and container image scans
- View and export vulnerability reports

Prerequisites

You have completed **Lab 2-2** (*Cloud Native Fundamentals: Use OCI Container Registry to Manage Images*).

Assumptions

- You are signed into your Oracle Cloud Infrastructure (OCI) account using your credentials.
- You will replace the `<userID>` placeholder with your first name.

Create a Private Repository in OCIR

In this practice, you will create a repository in your assigned compartment and give it a name that's unique across all compartments in the entire tenancy. Having created the new repository, you can push an image to the repository using the Docker CLI.

Tasks

1. Click the navigation menu, click **Developer Services,** and then click **Container Registry**.
2. Choose your assigned compartment from **List Scope** in the menu on the left.
3. Click **Create repository**.
4. For **Compartment**, ensure that your assigned compartment is selected.
5. For the Repository name, enter `demo_image_<userID>`

   (where <*username*> is your first name).

   For example: `demo_image_david`
6. Ensure **Access** is set to **Private**. Click **Create repository**.

   A repository `demo_image_<userID>` is created.

Enable Image Scanning

When you create a new repository, image scanning is disabled by default. You can use the Console to enable image scanning for a repository by creating a new image scanner. If image scanning has already been enabled, you can use the Console to disable it.

Tasks

1. Click the navigation menu, click **Developer Services,** and then click **Container Registry**.

2. Choose your assigned compartment from **List Scope** in the menu on the left.

3. Select the newly created repository `demo_image_<userID>` from the list of repositories.

4. Click **Actions** and select **Add Scanner**.

5. In the **Add scanner to repository** pane, enter the following values:
   - **Name:** `demo_scanner_<userID>`

     For example, `demo_scanner_david`
   - **Create in compartment:** Ensure that your compartment is selected.
   - **Description:** Optionally, add a description for the scanner.

6. Configure the **Scan configuration** settings.
   - Select **Create a new scan configuration**.
   - **Name**: `Scan_Config_<userID>`

     For example, `Scan_Config_david`
   - **Create in compartment**: Ensure that your compartment is selected.

7. Click **Add Scanner**.

   Note: A scan configuration identifies which images to scan by designating the compartment where they reside.

   Now that a scanner has been created and configured, images saved to the repository will be scanned for vulnerabilities. If the repository already contains images, the four most recently saved images will be immediately scanned for vulnerabilities as soon as the scanner is created.

Sign In to OCIR from Cloud Shell and Pull a Sample Docker Image from Docker Hub

Using your previously generated Auth Token, sign in to OCIR from Docker CLI in Cloud Shell. You then will pull a Docker image (`maven:latest`) from Docker Hub which will be tagged and pushed to your OCIR repository later.

Note: You will use the value of the Auth Token (**mytoken**) you created earlier in **Lab 1-1**. If you do not have the value of your token saved, you will need to delete the token and re-create it. (Refer to Lab 1-1 if necessary.)

Tasks

1. Open **Cloud Shell**.

2. In the cloud shell window, log in to the Container Registry (OCIR) by entering:
   ```
   $ docker login <region-key>.ocir.io
   ```
   For example:
   ```
   $ docker login phx.ocir.io
   ```

3. When prompted, enter your full OCI username in the format `<tenancy-name>/<oci-username>`.

   For example: `mytenancyname/myusername`

   Enter your Auth Token value you copied earlier as the password.

   Note: When you enter or paste the password in the terminal, you will not see masked characters. Press **Enter** on your keyboard to continue.

4. Run the following command to pull a sample Docker image (`maven:latest`).
   ```
   $ docker pull maven:latest
   ```

5. Verify that the Docker pull command is successfully executed:
   ```
   $ docker images
   ```
   Note: You should see the `maven:latest` image in the list of images.

Tag and Push the Docker Image to an OCIR Repository

A tag identifies the OCIR region, tenancy, and repository to which you want to push the image.

Tasks

1. In Cloud Shell, run the following command to attach a tag to the image that you're going to push to the OCIR repository:

   ```
   $ docker tag maven:latest
   <region-key>.ocir.io/<namespace>/<repo-name>:<tag>
   ```

   Where:

   - *<region-key>* is the key for the OCIR region you're using (*obtained in Lab 1-1*)
   - `ocir.io` is the Oracle Cloud Infrastructure Registry name
   - *<namespace>* is the unique namespace string of the tenancy (*obtained in Lab 1-1*)

     For example: `ansh81vru1zp`
   - *<repo-name>* is the name of the new repo in OCIR you just created

     For example: `demo_image_david`
   - *<tag>* is an image tag to give this image. Use: `1.0`

   Complete example:

   ```
   $ docker tag maven:latest
   phx.ocir.io/ansh81vru1zp/demo_image_david:1.0
   ```

2. Validate if the new image with the tag is listed.

   ```
   $ docker images
   ```

   Note: Although two tagged images will be shown (`latest` and `1.0`), both are based on the same image with the same image ID.

3. In Cloud Shell, run the following command to push the newly tagged Docker image to your OCIR repository:

   ```
   $ docker push <region-key>.ocir.io/<namespace>/<repo-name>:<tag>
   ```

   For example:

   ```
   $ docker push phx.ocir.io/ansh81vru1zp/demo_image_david:1.0
   ```

   You will see the different layers of the image are pushed in turn.

4. To verify if the image was pushed successfully, open the navigation menu, click **Developer Services,** and then click **Container Registry**.

   a. Choose your assigned compartment from **List Scope** in the menu on the left.

   b. You'll see the private repository `demo_image_<userID>` that you created. Select the repository and you'll see:

      - An image with the tag `1.0`
      - A summary page that shows you the details about the repository, including who created it and when, its size, and whether it's a public or a private repository

5. Click the image tag `1.0`.

   On the Summary page, you'll see the image size, when it was pushed and by which user, and the number of times the image has been pulled.

View Scan Results and Container Image Scans

The results of a container image scan include the specific vulnerabilities in the CVE database that were detected in the image.

Tasks

1. While still on the **Container Registry** page, select the image tagged `1.0` and click the **Scan Results** tab. This will show you the following info:
    - Risk level
    - Issues found
    - Scan started
    - Scan completed
2. Click the three dots on the right to open the Actions menu. Select **View Details** to see the issues in more detail with the risk level associated with each of them and their descriptions.

The results of a container image scan include the specific vulnerabilities in the CVE database that were detected in the image.

3. Click the navigation menu and click **Identity & Security**. Under **Scanning**, click **Scanning Reports**.
4. Ensure that your compartment is selected in **List Scope** in the left menu.
5. Click the **Container images** tab.
6. Locate the **Risk level** filter drop-down menu. Select **All**.
7. Locate the **Scan start date** and **Scan end date** filter drop-down menus.

    By default, only the most recent scan reports are displayed. To view older reports, choose the specific start and end dates.

    Alternatively, click **Scan start date** and choose to view reports for either the **Past 7 Days** or the **Past 30 Days**.
8. Locate the **Reset** button. Click **Reset** at any time to set the risk level and date ranges back to the default values.

9. (Optional) Click the table columns to sort the container image scans by:
   - Issues found
   - Risk level
   - Scan completed

10. To view a Container image report, click the name of the Container image.
    For example: `ansh81vru1zp/demo_image_david:1.0`

11. The following details are shown for each issue that was detected in this image:
    - Issue ID
    - Risk level
    - Issue description
    - Last detected
    - First detected
    - Cause and Remediation

12. Click an **Issue ID** to view more details about a specific vulnerability.

13. Click the **View detail** button in the **Cause and remediation** column to get more information about how to address this vulnerability.

View and Export Vulnerability Reports

The results of a container image scan include the specific vulnerabilities in the CVE database that were detected in the image.

In this section, you will explore Vulnerability Reports, accessing information about specific vulnerabilities that were detected in one or more targets.

Tasks

1. Click the navigation menu and click **Identity & Security**. Under **Scanning**, click **Vulnerability Reports**.

   Note: If you are presented with a general information screen labeled Vulnerability Scanning Service; locate and click **Skip**.

2. In the left navigation pane, under **Scanning**, select **Vulnerability Reports**.
3. Ensure your compartment is selected in **List Scope** on the left menu.
4. In the left navigation pane, under **Filters**, select the Risk level, **All.**
5. Click the **Risk level** header to sort by risk level.
6. To view a description of a specific vulnerability, click **Show** in the CVE description column.
7. To view details about a specific vulnerability, click a report's **CVE ID**.

   This will result in you viewing a vulnerability report. A vulnerabilities report includes details about the affected resources and CVEs.

8. On the Vulnerabilities report page, in the left navigation pane, under **Resources**, select **Container Images** to view a list of Container images that are affected by the selected vulnerability.

Use the Console to export all vulnerabilities reports as a file in comma-separated value (CSV) format for offline analysis.

9. Click the navigation menu and click **Identity & Security**. Under **Scanning**, click **Vulnerability Reports**.
10. Ensure your assigned compartment is selected in **List Scope** in the left menu.

11. Click **Export CSV**.

Example output:

```
resultId,compartmentId,cveId,severity,state,title,lastDetected,firs
tDetected,hostCount
1234,ocid1.compartment.example123,CVE-2018-12345,HIGH,OPEN,CVE-
2018-12345,2020-12-22T12:47:18Z,2020-12-21T16:47:25Z,3
```

Monitoring and Troubleshooting Cloud Native Applications: Troubleshooting Oracle Functions

Lab 10-1 Practices

Estimated Time: 20 minutes

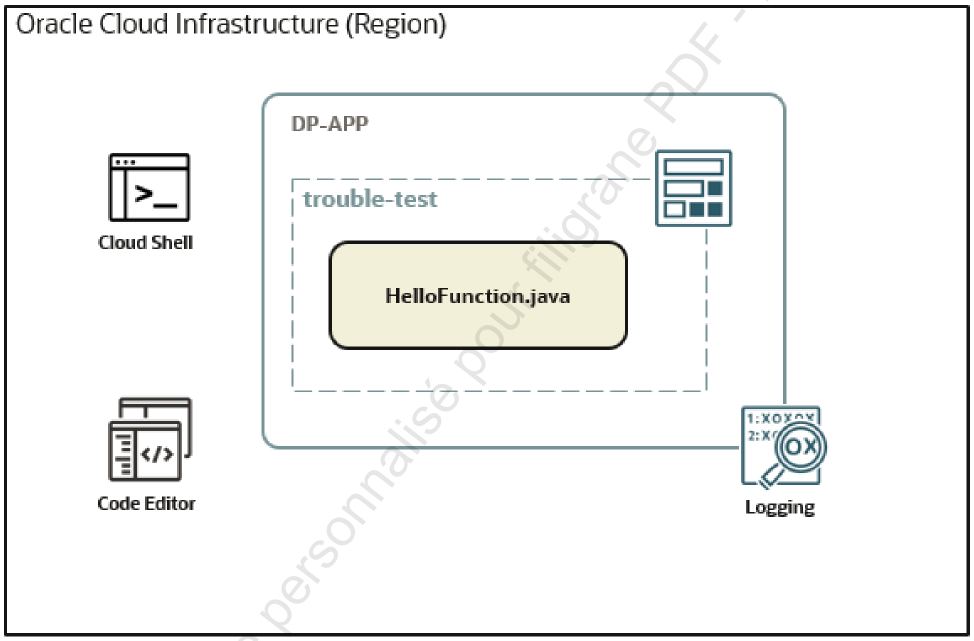
Get Started

Overview

Even if you've got excellent unit tests (for example, using the Fn Java JUnit support), things can still go wrong. Your function may throw an exception, or you may be getting back unexpected results.

In this lab exercise, you will explore some techniques and Fn features that can be used to get to the root cause of your problem. There are three practice sections:

a. Create a Private Repository in OCIR
b. Troubleshoot Fn Build Issues
c. Use OCI Logging to Troubleshoot Runtime Issues



Prerequisites

You have completed **Lab 4-1** (*Serverless Functions: Build and Deploy an Oracle Function*).

Assumptions

- You are signed into your Oracle Cloud Infrastructure (OCI) account using your credentials.
- You will replace the *`<userID>`* placeholder with your first name.

Create a Private Repository in OCIR

Before building and deploying a container image containing the function code, you need to create a private repository in the Container Registry that is within the same OCI region of the Function Application.

Tasks

1. In the Console, open the navigation menu and click **Developer Services**. Under **Containers & Artifacts**, click **Container Registry**.
2. Click the **Create repository** button.
3. For Repository name, enter `<userID>/trouble-test`

   where *<userID>* is your first name and is also the [repo-prefix] used for Function deployments you set earlier in **Lab 4-1**

   For example: `david/trouble-test`
4. Ensure **Access** is set to **Private**. Click **Create repository**.

   A repository `<userID>/trouble-test` is created.

Troubleshoot Fn Build Issues

In this practice, you will create a new Java function and explore the troubleshooting benefits of using the `--verbose` flag.

Tasks

1. Open **Cloud Shell** and navigate to the **labs** directory.

   ```
   $ cd ~/labs/
   ```

2. Create a new Java function called **trouble-test**:

   ```
   $ fn init --runtime java trouble-test
   ```

   This will create a basic "hello world" Java function in a new folder called `trouble-test`.

3. Navigate to that new folder.

   ```
   $ cd trouble-test
   ```

4. Delete the unit tests so that you can focus on troubleshooting techniques rather than keeping the tests up-to-date.

   ```
   $ rm -rf src/test
   ```

5. When you run commands such as `fn build` or `fn deploy`, you typically see "progress dots" (…) that let you know some action is taking place.

   Build the function and observe the output:

   ```
   $ fn build
   ```

   After a minute or two, you should see an output similar to:

   ```
   Function [repo-name]/trouble-test:0.0.1 built successfully.
   ```

6. To demonstrate the difference when there is an error due to the code not compiling or failing unit tests, you can change the function's code.

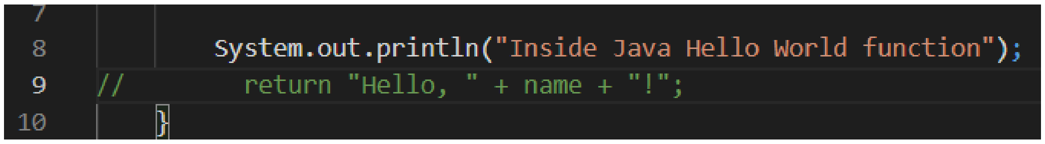
   Open **Code Editor** by clicking the Code Editor icon located in the upper-right section of the OCI Console.

7. Navigate to `/labs/trouble-test` to review the project files.

8. Open and view `src/main/java/com/example/fn/HelloFunction.java`.

9. Comment out the return statement in the HelloFunction class' **handleRequest** method by inserting `//` at the beginning of **line 9**.

Your code should look similar to the following:



```
            System.out.println("Inside Java Hello World function");
//          return "Hello, " + name + "!";
        }
```

10. Use `File > Save` from the menu bar or `<Ctrl+S>` to save the file.

11. In the Cloud Shell terminal, attempt to rebuild the function:

    ```
    $ fn build
    ```

    After a few seconds, you should see an output similar to:

    ```
    Fn: error running docker build: exit status 1
    ```

12. Because that information is not very useful, you should also notice that the output recommends using the `--verbose` flag. Do that now:

    ```
    $ fn -v build
    ```

    With verbose output, you see the entirety of the Maven build, which includes an error message that indicates a missing a return statement – (as we expected). When a build issue occurs, verbose output is the first thing you need to enable to diagnose the issue.

Use OCI Logging to Troubleshoot Run-Time Issues

In this practice, you will update the function code to produce a run-time exception, and then deploy and invoke the function. You will then leverage OCI Logging to view the Function Invocation Log for the application to view the error.

You will also enable debug mode to view additional information on the command line when invoking a function.

Tasks

1. Update the `HelloFunction.java` file so that it writes an error message and then throws an exception in the `handleRequest` method.

   Replace the entire definition of **HelloFunction** with the following code:

```
package com.example.fn;

public class HelloFunction {

    public String handleRequest(String input) {
        System.err.println("an error is about to happen");
        throw new RuntimeException("This is the runtime error");
    }
}
```

2. Use `File > Save` from the menu bar or `<Ctrl+S>` to save the file.

3. Build and deploy the function.

```
    $ fn -v deploy --app DP-APP
```

4. Invoke the function.

```
    $ fn invoke DP-APP trouble-test
```

   After a few seconds, you should see an output similar to:

```
  Error invoking function. status: 502 message: function failed
```

   Because this error message does not help in debugging the issue, we need to look at the function application log.

5. In the OCI Console, navigate to **Developer Services** > **Functions**, and then click the **DP-APP** link.

6. Under the **Resources** section in the left pane, click **Logs**.

7. The Function Invocation Logs should already be enabled from an earlier practice.

   Note: If the log is NOT enabled, enable it now by clicking the toggle switch under **Enable Log**. Then accept the default values and click **Enable Log**. Before proceeding to the next step, you will need to invoke the function again.

8. To view the log, click the **DP_APP_invoke** link under **Log Name**.

9. Review the list of entries. Locate and expand the entry:

   ```
   An error occurred in function: This is the runtime error
   ```

10. In addition, you can also enable debug mode when invoking an Oracle function with the Fn CLI to see the full details of the HTTP requests going to the Fn server and the response.

    Invoke the **trouble-test** function again adding **DEBUG=1** prior to the command:

    ```
    $ DEBUG=1 fn invoke DP-APP trouble-test
    ```

11. Take note of the details returned.

Congratulations! You have looked at the troubleshooting techniques when building and testing function code, as well as leveraging OCI logging for debugging run-time errors.