Show Submission Credentials

P2. Intro to Containers and Docker

Understanding Containers and Docker

- ✔ Introduction to Containers
- ✔ Hands-On Docker
- ✔ Docker Registry
- ✔ Debugging Docker Containers
- ✔ Conclusion

Introduction to Containers

Introduction to Containers

> Operating-system-level virtualization is a server virtualization method where the kernel of an operating system allows the existence of multiple isolated user-space instances, instead of just one. [Wikipedia (https://en.wikipedia.org/wiki/Operating-system-level_virtualization)]

You may already be familiar with hardware virtualization systems under which a hypervisor (https://en.wikipedia.org/wiki/Hypervisor) runs virtual machines and provides the VMs with a virtualized representation of the host machine's hardware. Under virtualization, different types of guest operating systems (e.g., Windows, Linux, and macOS) can be launched as

virtual machines on the same hardware. Common hypervisor systems include the Linux Foundation's Xen Project (http://www.xenproject.org) and Microsoft's Hyper-V (https://www.microsoft.com/en-us/cloud-platform/server-virtualization).

Under OS level virtualization, the virtualized environments are called containers. Containers may be run on either bare metal hardware or in virtual machines. Unlike hardware virtualization, OS level virtualization provides containers with certain access to the host machine's kernel.

There are a couple specific kernel level technologies that enable containers to access the host machine's kernel:

- Control groups - Control groups (https://en.wikipedia.org/wiki/Cgroups) (cgroups) are a kernel level software that provides resource isolation and access limits for a set of processes. Specific features of cgroups includes limiting the amount of resources consumed (e.g., cpu, memory). Control groups enable containers to share available hardware resources, and optionally enforce resource limits.
- Namespaces - Namespaces (https://en.wikipedia.org/wiki/Linux_namespaces) are a kernel software that is used to isolate and visualize resources. That is to say when a container is running, a set of namespaces is created specifically for that container. In the case of Docker (https://docs.docker.com/engine/understanding-docker/#/namespaces), namespaces are created for the following resources:
    - **The `pid` namespace:** The process ID to provide process isolation.
    - **The `net` namespace:** The network interface to isolate container networking.
    - **The `ipc` namespace:** Inter-process communication, which enables the sharing of information between processes.
    - **The `mnt` namespace:** The filesystem mount points, which isolates the filesystem for each container.
    - **The `uts` namespace:** The Unix Timesharing System (UTS) for isolating kernel and version identifiers.

Docker

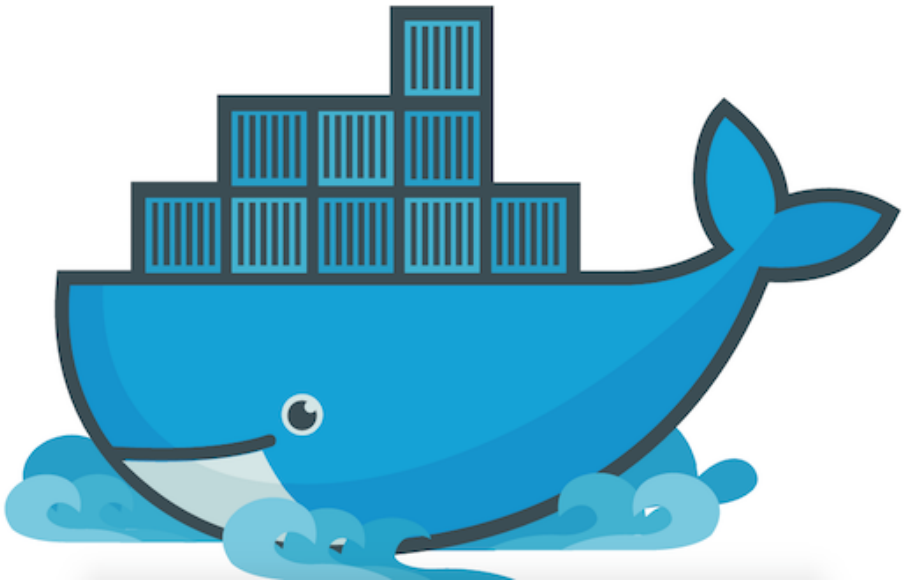

Figure 1: Docker

Docker (https://www.docker.com/) is an implementation of the Operating-system-level virtualization technology discussed above and it is the container runtime we'll explore further. A container runtime is a software that executes containers and manages container images.

> A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. [Docker.com (https://www.docker.com/what-docker)]

The container runtime, Docker Engine (https://www.docker.com/products/docker-engine) provides interoperability between different host operating systems, which allows your container to run anywhere the Docker Engine can be installed.

> A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings. [Docker.com (https://www.docker.com/what-docker)]

A container image is the software artifact that the developer builds, and Docker uses container images to create container instances. Developers can select from a large set of base images, which are built with common packages required so that the developers do not have to build everything from scratch.

Comparing Containers and Virtual Machines

From a practical standpoint, containers and virtual machines may appear similar: they both provide isolation and enable the sharing of resources across multiple virtualized environments.

Below we explore the characteristics of deploying applications to virtual machines:

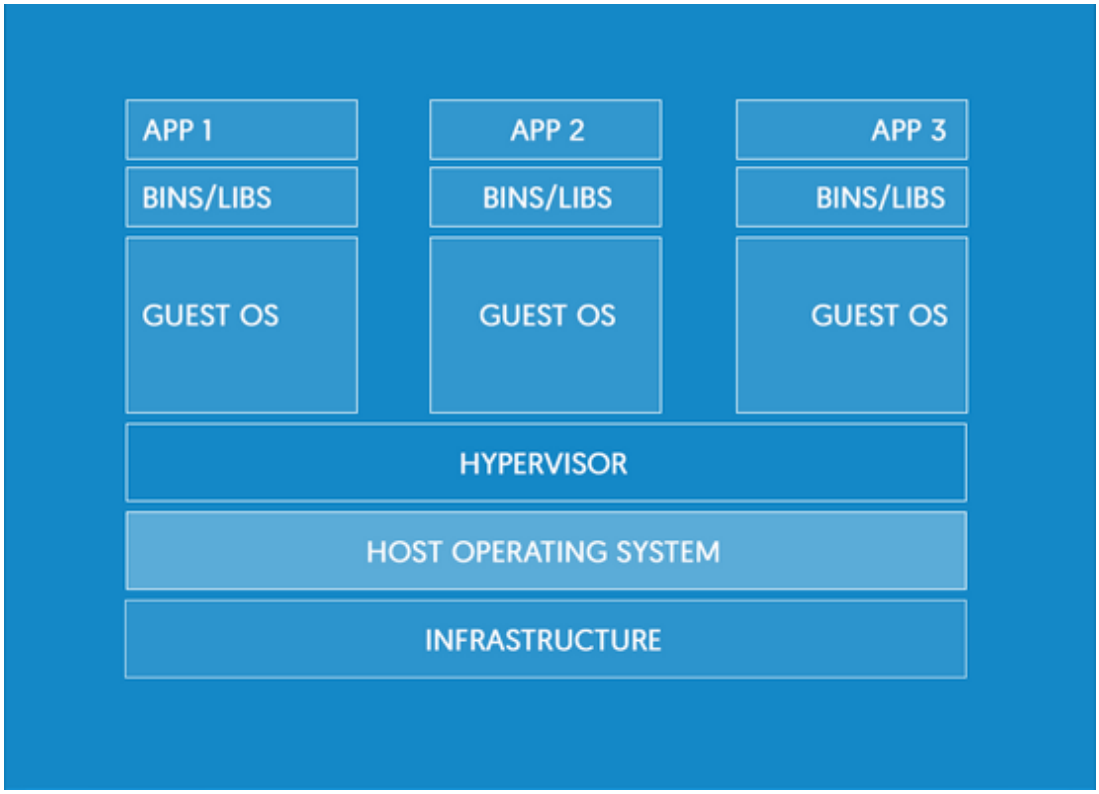

Figure 2 - Applications running on virtual machines.

- Each virtual machine (VM) has a full copy of the guest operating system.
  - One host machine can host different guest OSs at the same time.
  - Additional disk space is required to store the applications and dependencies, plus the full copy of the guest OS.
- The hardware is shared by the VMs.
- The unit of isolation is a virtual machine.
  - A VM cannot directly interact with the host or the other VMs.
- Virtual machines have a startup time on the order of minutes.
  - Similarly, building new virtual machine images is on the order of minutes.
- A single virtual machine can run multiple applications.

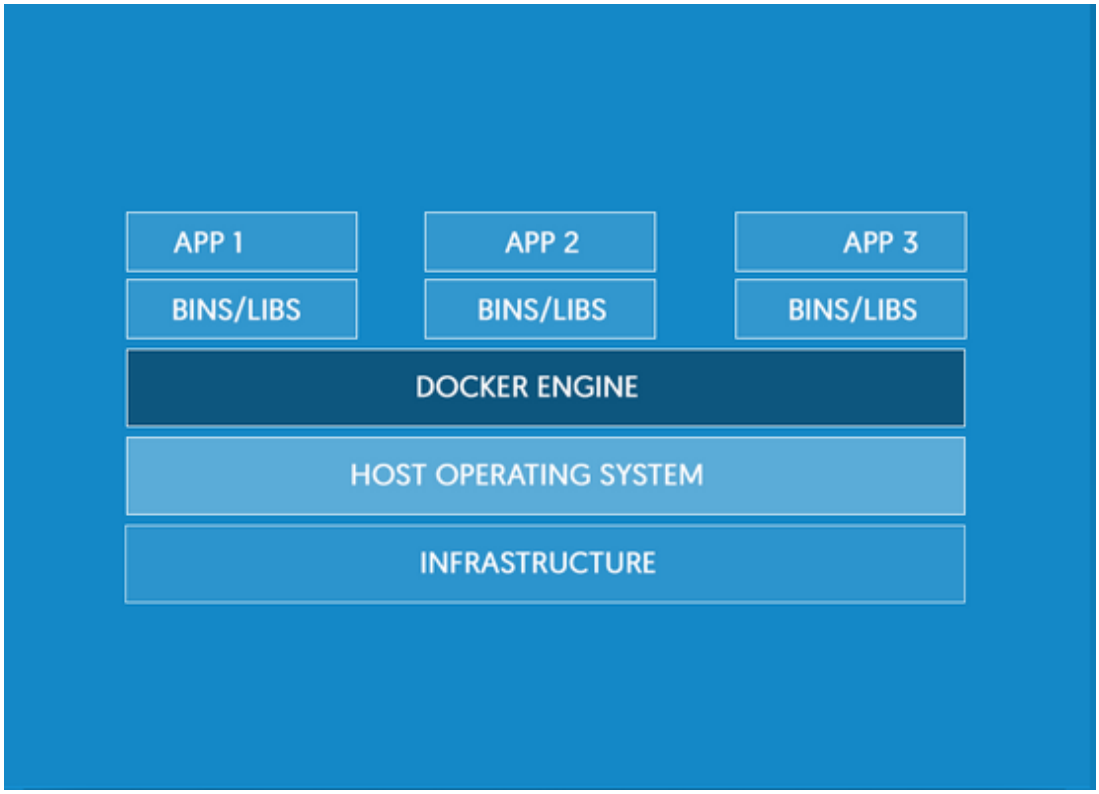

Figure 3 - Applications running in containers.

- Each container has only the necessary software to run the application.
  - Containers can be run on any system (either bare metal or virtual) where the Docker Engine is installed.
  - Only the application and dependencies account for disk space overhead.
- The host machine's kernel is shared by other containers.
- The unit of isolation is a container.
  - The isolation between containers and between the host machine and containers is enabled by cgroups and namespaces as discussed in the previous section.
- Containers have a start up time on the order of seconds.
  - Similarly, building new container images is on the order of seconds to minutes.
- A container only runs a single application.
  - To run multiple applications, you may deploy multiple containers together and the containers can communicate over a network.

It is important to understand that virtual machines and containers are not mutually exclusive, for example, developers can install Docker Engine and run containers on virtual machines.

Hands-On Docker

Hands-On Docker

To better understand how Docker works, let us start with the basic operations. Here you will learn to deploy simple applications using Docker containers.

Installation

We need to install Docker on a VM to get started.

First, provision an `Ubuntu 18.04 LTS` VM on any cloud platform that you prefer. **Please make sure you open the inbound HTTP port 80 and SSH port 22.**

Please remember to tag your resources using `project:containers`.

Run the following official convenience script (https://docs.docker.com/engine/install/ubuntu/#install-using-the-convenience-script) to install Docker Engine.

```
curl -fsSL https://get.docker.com -o get-docker.sh && sudo sh get-docker.sh
```

By default, Docker must be run as a root user. If you would like to use Docker as a non-root user, you should now consider adding your user to the "docker" group with something like:

```
# whoami: returns the current Linux user
sudo usermod -aG docker $(whoami)
```

Log out and back into the VM for the command to take effect, or you will get errors such as `docker: Got permission denied while trying to connect to the Docker daemon socket.`

Verify that Docker Engine is installed correctly by running the hello-world image.

```
docker run hello-world

# Expected output:
# ...
# Hello from Docker!
# This message shows that your installation appears to be working correctly.
# ...
```

The Docker CLI

The Docker command line (CLI) is a tool for interacting with the Docker engine. We'll start by looking at how to create a **Dockerfile** to provide the system configurations for our first container.

Deploy a Web Server on a VM

In this section, you will build a simple server inside a container.

Please download the tarball that includes a Maven project (that consists of `pom.xml` and `./src/main/java/HelloWorld.java`) and a Dockerfile.

```
wget https://clouddeveloper.blob.core.windows.net/primers/container/sample-conta
inerized-webservice.tar -O sample-containerized-webservice.tgz
tar -xvzf sample-containerized-webservice.tgz
```

The Maven project is the simplest Undertow web server, and we will explain the Dockerfile in the next section.

The structure of the tarball is as follows:

```
.
|--- Dockerfile
|--- pom.xml
|--- src
     |--- main
          |--- java
               |--- HelloWorld.java
```

Install Maven and package the Maven project as a JAR named as `demo-1.0-SNAPSHOT-jar-with-dependencies.jar` in the `target` folder:

```
sudo apt-get install -y maven
mvn clean package
```

You can run the JAR to start a web server hosted using the virtual machine.

```
sudo java -cp ./target/demo-1.0-SNAPSHOT-jar-with-dependencies.jar HelloWorld
```

Now, visit `http://<vm-public-ip>:80` to verify the web server and the server returns a plain text response `Hello World`.

Assemble an image with Dockerfile

In the previous section, you deployed a web server on a VM. Now, you will transform the VM-hosted solution and "containerize" the web server, by creating a Docker image that includes the web server JAR and running a container.

Docker Engine builds Docker images from Dockerfiles. A Dockerfile is basically a text document that contains all the commands a user could call on the command line to assemble an image.

There are several concerns to consider before you create a Dockerfile:

1. The necessary packages/software to download.
2. The files that need to be copied into the container, or any volumes that should be mounted.
3. The ports to open on the container.

Below is the Dockerfile included in the `sample-containerized-webservice.tgz` example. The sample Dockerfile uses the Ubuntu 18.04 image as the base image, installs Java Runtime Environment, copies the web server JAR, and defines the command to run the Undertow server when the container starts:

```
# Ubuntu Linux as the base image
FROM ubuntu:18.04

# Install the packages by using the default package manager in Ubuntu
RUN apt-get update && \
    apt-get -y install default-jre

# Open port 80
EXPOSE 80

# Copy the files from the host file system and ADD them to the desired directory
inside the container image
# Note that this will copy the JAR to `/demo-1.0-SNAPSHOT-jar-with-dependencies.
jar`
ADD ./target/demo-1.0-SNAPSHOT-jar-with-dependencies.jar /

# Define the command which runs when the container starts
# Note that the filepath of the JAR is `/demo-1.0-SNAPSHOT-jar-with-dependencie
s.jar`
# as in the filesystem of the image,
# NOT `./target/demo-1.0-SNAPSHOT-jar-with-dependencies.jar` as in the host file
system
CMD ["java -cp demo-1.0-SNAPSHOT-jar-with-dependencies.jar HelloWorld"]

# Use Bash as the container's entry point.
ENTRYPOINT ["/bin/bash", "-c"]
```

Both `CMD` and `ENTRYPOINT` instructions define what command gets executed when running a container. A combination of `ENTRYPOINT ["/bin/bash", "-c"]` and `CMD ["java -cp demo-1.0-SNAPSHOT-jar-with-dependencies.jar HelloWorld"]` will be translated into the following command when the container starts:

```
/bin/bash -c "java -cp demo-1.0-SNAPSHOT-jar-with-dependencies.jar HelloWorld"
```

Note that the Dockerfile only contains the necessary packages to *run* the JAR, such as Java Runtime Environment. There is no need to install Maven in the Dockerfile because the image is not in charge of building the JAR.

Build a Docker image

Build a docker image from the Dockerfile.

```
# --rm: Remove intermediate containers after a successful build
# --tag: Name and optionally tag the image in the 'name:tag' format
#
# Please note the `.` at the end of the docker build command, and you cannot omi
t it.
# `.` (which means the current working directory) is passed as the `PATH`,
# and all the files in the current working directory will get packaged and sent
to the Docker daemon
# not just the ones listed to ADD in the Dockerfile

docker build --rm --tag clouduser/primer:latest .
```

If the above command runs successfully, you should be able to see the container image listed:

```
docker images
# Sample output:
#
# REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
# clouduser/primer     latest              767ec272ed93        2 minutes ago
453MB
# ubuntu              18.04               a2a15febcdf3        4 weeks ago
64.2MB
# hello-world         latest              fce289e99eb9        8 months ago
1.84kB
```

Run, List and Stop Containers

Run a container with the image `clouduser/primer:latest` you just created:

```
# -d: When starting a Docker container, you must decide if you want to run
# the container in the background in a "detached" mode or in the default foregro
und mode.
# `-d` starts the container in detached mode
#
# -p: map a single port or range of container ports to the host port
# the mapping has the following format: "hostPort:containerPort"

docker run -d -p 80:80 clouduser/primer:latest
# A docker will run in the background
# Note the usage of command docker -p <hostPort>:<containerPort>
# Sample output:
# e32b87a5b9482a3899de8afa65a05efdfff093951cc3ec0576006b8b10a1d963
```

The container runs a web server that listens to the container at port 80, and the container port 80 is mapped to the host VM port 80 as per `-p 80:80`. As the host VM post 80 is open to the public, you can now access the containerized server from the Internet.

Visit `http://<vm-public-ip>:80` to verify that the web server returns a plain text response `Hello World`.

Congratulations! You have successfully built a custom Docker image and deployed a container.

If you want to stop the running container, use the following commands to list and stop the containers:

```
docker ps
# Sample output:
# CONTAINER ID        IMAGE                     COMMAND                  CREATED
STATUS              PORTS                NAMES
# e32b87a5b948        clouduser/primer:latest   "/bin/bash -c 'java …"   14 minu
tes ago      Up 14 minutes       0.0.0.0:80->80/tcp   serene_albattani

# stop a container by ID
docker stop CONTAINER_ID
```

There are many options with the docker run (https://docs.docker.com/engine/reference/run/) commands, such as the **-i** and **-t** flags (https://docs.docker.com/engine/reference/run/#foreground) to take you inside the container and enable you to interact with the container, which can be helpful for debugging. Refer to the official Docker documentation (https://docs.docker.com/) for further information.

You can find more information in the Docker documentations below: - Docker command line (https://docs.docker.com/engine/reference/commandline/cli/) - The base command for the Docker CLI (https://docs.docker.com/engine/reference/commandline/docker/)

Docker Registry

Introduction to the Docker Registry

> The (Docker) Registry is a stateless, highly scalable server side application that stores and lets you distribute Docker images. The Registry is open-source, under the permissive Apache license. [Docker Registry (https://docs.docker.com/registry/)]

A container registry is the platform to share Docker container images. You can access images from any of the public Docker repositories (i.e., Docker Hub (https://hub.docker.com/)) or store your Docker images in a private Docker registry.

You can create and maintain your own Docker registry, but there are many existing solutions for hosted registries. For example, Docker Hub provides free services to both public and private Docker image hosting. You might need to use cloud-managed container registry services in future projects in order for you to distribute your custom Docker images.

In this section, you will be exploring cloud managed Docker repositories on three cloud platforms: GCP, AWS and Azure.

Google Container Registry (GCR)

The Google Container Registry (GCR) provides private container image hosting on GCP.

Note: The instructions below include the steps to work with Google Container Registry on an Ubuntu 18.04 VM. For the fullest documentation, please refer to Google Container Registry - Pushing and pulling images (https://cloud.google.com/container-registry/docs/pushing-and-pulling)

Before you begin

First, create a GCP project, enable billing (https://cloud.google.com/billing/docs/how-to/modify-project#enable_billing_for_a_project), and enable Google Container Registry API (https://console.cloud.google.com/apis/library/containerregistry.googleapis.com?q=registry).

Install Google Cloud SDK to the Ubuntu 18.04 VM:

```
# Add the Cloud SDK distribution URI as a package source:
echo "deb [signed-by=/usr/share/keyrings/cloud.google.gpg] https://packages.clou
d.google.com/apt cloud-sdk main" | sudo tee -a /etc/apt/sources.list.d/google-cl
oud-sdk.list

# Import the Google Cloud public key
curl https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key --keyr
ing /usr/share/keyrings/cloud.google.gpg add -

# Update and install the Cloud SDK
sudo apt-get update && sudo apt-get install -y google-cloud-sdk

# Run gcloud init to get started:
gcloud init
```

Configured Docker to use gcloud as a credential helper.

```
gcloud auth configure-docker
# gcloud's Docker credential helper can be configured but it will not work until
this is corrected.
# ...
# Do you want to continue (Y/n)?  Y
# Docker configuration file updated.
```

Tag and Push an Image to the Google Container Registry

Docker uses the image tag to determine what repository the image belongs to. To push any local image to the Google Container Registry, you need to first tag the image with the Google Container Registry name and then push the image.

The command to tag your images is as follows:

```
docker tag [SOURCE_IMAGE] [HOSTNAME]/[PROJECT-ID]/[IMAGE]:[TAG]
```

Now, you need to determine the registry name `[HOSTNAME]/[PROJECT-ID]/[IMAGE]:[TAG]`, which includes a hostname, a GCP project ID, and a name with an optional tag, and you need to choose a concrete value for each of them:

1. Choose a hostname, which specifies the region of the registry's storage. You can find a list of hostnames and related information listed in the GCR documentation (https://cloud.google.com/container-registry/docs/pushing-and-pulling#tag_the_local_image_with_the_registry_name). Here we will use `us.gcr.io`, which hosts the image in the United States.
2. Choose a project ID for your GCP project. Please note a project ID is different from a project name.
3. Choose a (remote) image name, which can be different from the local image name on your local machine. In our example, the local source image is tagged as `clouduser/primer:latest`. Here we will use `clouduser/gcr-demo` as the remote image name.

To sum up, the command to tag the local image you built previously with a Google Container Registry name is as follows:

```
GCP_PROJECT_ID=your_gcp_id
docker tag clouduser/primer:latest us.gcr.io/${GCP_PROJECT_ID}/clouduser/gcr-dem
o
```

Push the image to GCR.

```
docker push us.gcr.io/${GCP_PROJECT_ID}/clouduser/gcr-demo
```

Now you can view the image you pushed in your Google Container Registry using the GCP web console (https://console.cloud.google.com/gcr/).

For future use, you can pull this image with the command:

```
# Similar to the docker push process with GCR,
# before you run docker pull, you need to set up Docker and gcloud
docker pull us.gcr.io/${GCP_PROJECT_ID}/clouduser/gcr-demo
```

Verify that the image is successfully pulled and you can run a container with the image.

```
docker images

docker run -d -p 80:80 us.gcr.io/${GCP_PROJECT_ID}/clouduser/gcr-demo
```

Troubleshooting GCR Authentication

Problem

`gcloud auth configure-docker` is the recommended solution to authenticate to the Container Registry, however, in some environments `gcloud auth configure-docker` will fail to authenticate and you need to use another more advanced authentication solution.

Symptom

After you run `gcloud auth configure-docker`, you still get an "unauthorized" error when you run `docker pull` or `docker push` with GCR:

```
Error response from daemon: unauthorized: You don't have the needed permissions
to perform this operation, and you may have invalid credentials. To authenticate
your request, follow the steps in: https://cloud.google.com/container-registry/d
ocs/advanced-authentication
```

Solution

Use the following command to authenticate Docker commands by passing a short-lived access token as a password to Container Registry.

```
gcloud auth print-access-token | docker login -u oauth2accesstoken --password-st
din https://us.gcr.io
# Expected output:
# ...
# Login Succeeded
```

Azure Container Registry (ACR)

Install Azure CLI.

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

Log in to Azure.

```
az login
```

List the subscriptions of your Azure account.

```
az account list --output table
```

Set a subscription to be the current active subscription that you want to provision a container registry with.

```
az account set --subscription <SUBSCRIPTION_ID>
```

Create a resource group.

```
az group create --name acrDemoResourceGroup --location eastus
```

Create a container registry. The registry name must be globally unique (case insensitive) within Azure and contain 5-50 alphanumeric characters. We suggest that you only use **lowercase** letters and digits and avoid uppercase letters. You can get a globally unique string that consists of lowercase letters and digits at the Online UUID Generator (https://www.uuidgenerator.net/), but you will need to make some modifications such as to remove the dashes.

```
ACR_NAME=<unique_acr_name>
az acr create --name ${ACR_NAME} --resource-group acrDemoResourceGroup --locatio
n eastus --sku Basic
```

Before pushing and pulling container images, you must log in to the container registry with the `az acr login` command.

```
az acr login --name ${ACR_NAME}
# Expected output:
# Login Succeeded
```

You can now verify the fully qualified name of your ACR login server with the following command.

```
az acr list --resource-group acrDemoResourceGroup --query "[].{acrLoginServer:lo
ginServer}" --output table

# Expected output:
AcrLoginServer
----------------------------------------
<lowercased_acr_name>.azurecr.io
```

The login server name is in the format `<lowercased_acr_name>.azurecr.io` (all lowercase). We will use `${ACR_NAME}.azurecr.io` in the following bash commands, with the assumption that your ACR name only contains lowercase letters and digits.

We will use `clouduser/acr-demo` as the example of the image name on ACR, the commands to push an image to ACR will be:

```
docker tag clouduser/primer:latest ${ACR_NAME}.azurecr.io/clouduser/acr-demo:lat
est

docker push ${ACR_NAME}.azurecr.io/clouduser/acr-demo:latest
```

For future use, you can pull this image with the command:

```
# Similar to the docker push process with ACR,
# before you run docker pull, you need to
# 1. install and log in Azure CLI
# 2. set the same subscription to be the current active subscription as you used
in docker push progress
# 3. and log in to the container registry with `az acr login`
docker pull ${ACR_NAME}.azurecr.io/clouduser/acr-demo:latest
```

Verify that the image is successfully pulled and you can run a container with the image.

```
docker images
docker run -d -p 80:80 ${ACR_NAME}.azurecr.io/clouduser/acr-demo:latest
```

Amazon Elastic Container Registry (ECR)

> Amazon Elastic Container Registry (ECR) is a fully-managed Docker container registry that makes it easy for developers to store, manage, and deploy Docker container images. [Amazon Elastic Container Registry (https://aws.amazon.com/ecr/)]

You will first access the Elastic Container Registry from the AWS web console (https://console.aws.amazon.com/ecr/repositories?region=us-east-1) with the region as `us-east-1`.

1. Click the "Create repository" button to create a repository for your images. We will use `clouduser/ecr-demo` as the example.
2. Set "Tag mutability" to be "Enable".

After you successfully create the repository, click "View push commands" and AWS will provide the Docker commands to authenticate your Docker client, and to tag and push the Docker image to the ECR registry.

Within the context of this primer, the commands to push an image to ECR will be:

```
# install AWS CLI, if it has not been installed
sudo apt-get install -y awscli

# configure AWS CLI
aws configure

# Authenticate your Docker client to your registry
eval $(aws ecr get-login --no-include-email --region us-east-1)
# Expected output:
# ...
# Login Succeeded

AWS_ID=your_aws_id
docker tag clouduser/primer:latest ${AWS_ID}.dkr.ecr.us-east-1.amazonaws.com/clo
uduser/ecr-demo:latest

docker push ${AWS_ID}.dkr.ecr.us-east-1.amazonaws.com/clouduser/ecr-demo:latest
```

For future use, you can pull this image with the command:

```
# Similar to the docker push process with ECR,
# before you run docker pull, you need to install and configure AWS CLI and
# authenticate your Docker client to your registry if you have not done so

docker pull ${AWS_ID}.dkr.ecr.us-east-1.amazonaws.com/clouduser/ecr-demo:latest
```

Verify that the image is successfully pulled and you can run a container with the image.

```
docker images

docker run -d -p 80:80 ${AWS_ID}.dkr.ecr.us-east-1.amazonaws.com/clouduser/ecr-d
emo:latest
```

Debugging Docker Containers

Debugging Docker Containers

Here is an application. You need to use the above knowledge to deploy it in the docker container. We use the web framework fastapi to build the application and we are not going to provide the source code. Here are some information about the application:

The application will listen on port 8000.

The application has three endpoints. They are `/`, `/docker` and `/k8s`.

Here is a buggy Dockerfile, please fix bugs and deploy the application in the docker container.

You are going to fix following issues:

Base environment. You should use `Python 3.8.10` as the base image.

Expose the correct port of the container.

Set correct environment variables. You can find hints in logs of the container.

```
FROM python:3.9

EXPOSE 8080

WORKDIR /code

COPY requirements.txt /code/requirements.txt
COPY buggy-app.pyc /code/buggy-app.pyc

RUN pip install --no-cache-dir --upgrade -r /code/requirements.txt

CMD ["python3", "buggy-app.pyc"]
```

Deploy the application

Setting work directory and download application.

```
# Create work directory
$ mkdir debug-example && cd debug-example
# Download compiled application
$ wget https://clouddeveloper.blob.core.windows.net/primers/container/buggy-app.
pyc
# Create Dockerfile
$ touch Dockerfile
# Copy above buggy Dockefile into the Dockerfile we just created
# You may use Vim or other editor to paste the code
# Download the requirements.txt
$ wget https://clouddeveloper.blob.core.windows.net/primers/container/requiremen
ts.txt
```

Build and run the container

```
$ docker build -t debug-example:0.1.0 .
$ docker run -d --name=docker-debug -p 8080:8000 debug-example:0.1.0
```

We can use `docker container ls -a` to check the status of the container.

```
$ docker container ls -a
CONTAINER ID   IMAGE                 COMMAND                  CREATED
STATUS                          PORTS     NAMES
76b685eb3c62   debug-example:0.1.0   "python3 buggy-app.p…"   About a minute ago
Exited (1) About a minute ago             docker-debug
```

As we can see, the container exited with an error. We can use `docker logs` to check the error message.

```
$ docker logs docker-debug
RuntimeError: Bad magic number in .pyc file
```

The runtime error indicates that we used a wrong python version. Thus, we should change the container base image to `Python: 3.8.10` . Then we can build and run the application again.

```
$ docker rm docker-debug
docker-debug
$ docker build -t debug-example:0.2.0 .
$ docker run -d --name=docker-debug -p 8080:8000 debug-example:0.2.0
$ docker container ls -a
CONTAINER ID   IMAGE                 COMMAND                  CREATED         ST
ATUS         PORTS                                NAMES
885947d2e4e1   debug-example:0.2.0   "python3 buggy-app.p…"   8 seconds ago   Up
6 seconds   8080/tcp, 0.0.0.0:8080->8000/tcp   docker-debug
```

Now, since the container is running successfully, we can use `curl` to start our test.

```
$ curl http://localhost:8080/docker
"Aoh, something going wrong."
```

The response message indicates something went wrong in the application. Thus, we should check container logs to find out error messages.

```
$ docker logs docker-debug
INFO:     Started server process [1]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
Welcome to the Docker Debug Tutorial!
It looks like we are missing the environment variable CC_DOCKER that the program
needs.
Try to set CC_DOCKER="I LOVE CLOUD COMPUTING" in the docker.
You may start another container and add environment variable CC_DOCKER="I LOVE C
LOUD COMPUTING"
You can refer to https://docs.docker.com/engine/reference/run/

INFO:     172.17.0.1:44752 - "GET /docker HTTP/1.1" 200 OK
```

The error message indicates that we should set environment variable `CC_DOCKER` to `I LOVE CLOUD COMPUTING`. Since the environment variables are always sensitive data, we are not going to set them in the Dockerfile.

```
$ export CC_DOCKER="I LOVE CLOUD COMPUTING"
$ docker stop docker-debug && docker rm docker-debug
$ docker run -d --name=docker-debug -p 8080:8000 -e "CC_DOCKER=$CC_DOCKER" debug
-example:0.2.0
$ curl http://localhost:8080/docker
"Great! You pass the test!"
```

The message indicates that we pass a part of debug test.

Tips:

If you want to see the logs in real time, you could use the **attach** command.

```
docker attach <YOUR_CONTAINER_NAME>
```

Conclusion

Conclusion

Now that you have gained an understanding of Docker and cloud-managed container registries, you are encouraged to review the ***Kubernetes and Container Orchestration*** primer to develop practical skills of building and running containers using Docker and deploying applications with Kubernetes.

Danger

Please remember to delete all resources in this primer as you may incur charges, which include:

1. the Ubuntu 18.04 VM
2. the GCP project which includes the GCP Container Registry
3. the Azure resource group which includes the Azure Container Registry
4. the AWS Container registry