Certified Kubernetes Application Developer (CKAD)

By Damian Igbe, PhD

About Certified Kubernetes Application Developer (CKAD) Training

The overall objective for this training is to prepare candidates for certifications in **Certified Kubernetes Application Developer Training (CKAD)** Examinations using the official training curriculum. The course requires that you have a fundamental knowledge of how container runtimes (e.g., Docker) works.

Docker has taken the world of DevOps by storm. Docker makes it easy to adopt the DevOps practices. As applications architectures change from Monolith to the Microservices, Docker plays a major role in deploying Microservices applications. The ability to package each individual microservice in a Docker image, deploy the image to a Registry and then run the microservices application is what Docker provides.

Kubernetes helps with running, deploying and maintaining a container-based application on diverse distributed nodes called the Kubernetes Cluster. Kubernetes helps with the scheduling of containers to the Kubernetes cluster, orchestrating communication between the containers, maintaining the volume infrastructure of the Kubernetes Cluster, etc. In fact, all aspects of the lifecycle of a microservices application are handled by Kubernetes.

One of the most desirable features of Docker and Kubernetes is that they are portable across different IT platforms. As businesses migrate from on-premises to the cloud environments, there is a need for the applications to work with minimal effort. Docker and Kubernetes make this possible.

Course Prerequisite

Kubernetes foundation course or equivalent professional experience. A knowledge of the following would be beneficial.

- Hands-on experience using containers
- Hands-on experience with Orchestration applications
- Knowledge of typical multi-tier architectures: web servers, application servers, load balancers, and storage
- Foundational knowledge of Linux
- Knowledge of RESTful Web Services, XML, JSON, YAML
- Familiarity with the software development lifecycle

Intended Audience

This course is intended for IT professionals who may alreadybe familiar with Containers and Container Orchestration.

Course Objectives

This training follows the outline presented by Cloud Native Computing Foundation (CNCF) for the Certified Kubernetes Administration Certification (CKAD). The outline can be downloaded here:

https://github.com/cncf/curriculum/blob/master/CKAD_Curriculum_v1.32.pdf

Here is a breakdown of the day to day learning.

Day 1

- Introduction
- Demo
- Docker (Define, build and modify container images)
- Intro to K8S (kubectl basics)
- Understand multi-container Pod design patterns (e.g. sidecar, init and others)

Day 2:

- Choose and use the right workload resource (Deployment, DaemonSet, ReplicaSet, Jobs and CronJobs)
- Use Kubernetes primitives to implement common deployment strategies (e.g. blue/ green or canary)
- Understand Deployments and how to perform rolling updates

Day 3

- Demonstrate basic understanding of Network Policies
- Provide and troubleshoot access to applications via services
- Use Ingress rules to expose applications

Day 4

- Use the Helm package manager to deploy existing packages
- Kustomize
- Operators
- Implement probes and health checks
- pv/pvc
- Discover and use resources that extend Kubernetes (CRD)
- Understand authentication, authorization and admission control
- Understanding and defining resource requirements, limits and quotas
- Understand ConfigMaps
- Create & consume Secrets
- Understand ServiceAccounts
- Understand Applications (SecurityContexts, Capabilities etc)

Day 5

- Understand API deprecations
- Implement probes and health checks
- Use provided tools to monitor Kubernetes applications
- Utilize container logs
- Debugging in Kubernetes

Exam Practice

Chapter 1: Introduction to Containers

What are containers?

Containers are lightweight, portable, and self-sufficient units that package an application and all its dependencies, allowing it to run consistently across various environments. They isolate the application from the underlying host system and other applications, ensuring that the app behaves the same, whether running on a developer's machine, a testing environment, or in production.

In essence, containers allow you to **package an application along with its dependencies (libraries, configurations, etc.)** into a single, executable unit that can be run anywhere—without worrying about compatibility or environmental differences.

Key Characteristics of Containers

1. **Lightweight**: Containers share the host operating system's kernel, which makes them much lighter and faster than virtual machines (VMs). They don't require an entire OS to run, just the application and necessary libraries.

2. **Portable**: Since containers package everything needed to run an app (including runtime, libraries, environment variables, etc.), they can run anywhere: locally, on any cloud provider, or even in on-premises data centers.

3. **Isolation**: Containers provide isolation for processes running within them. This means that the app running inside one container doesn't interfere with the app running inside another container, even if they're on the same host.

4. **Consistency**: Containers ensure that an application runs in the exact same way regardless of where it is deployed. The same container image can be used across different development, testing, and production environments.

5. **Scalability**: Containers are designed to be scaled easily. For instance, if you need to run multiple instances of a container (like in a microservices architecture), it's simple to create and manage multiple containers.

Components of a Container

1. **Container Image**: The "blueprint" for a container, which includes the application, libraries, environment variables, and runtime. It's a snapshot of the application and its environment.

2. **Container**: An instance of a container image running on a container engine (like Docker). Containers can be started, stopped, or paused as needed.

3. **Container Engine**: Software that runs and manages containers. Docker is the most popular container engine, but there are others, such as containerd, Podman, etc.

4. **Container Registry**: A place where container images are stored and shared. Popular registries include **Docker Hub** (public), **Amazon Elastic Container Registry (ECR)**, and **Google Container Registry (GCR)**.

How Containers Differ from Virtual Machines (VMs)

Containers and virtual machines (VMs) are both technologies used to isolate applications and provide a consistent runtime environment, but they work in fundamentally different ways:

- **VMs**:
    - Run a full operating system (OS) and include the application and all necessary dependencies.
    - Each VM runs its own complete OS, which makes it more resource-heavy and slower to start.
    - Virtualization is provided by a hypervisor (e.g., VMware, Hyper-V, KVM).
- **Containers**:
    - Share the host OS kernel and only include the application and its dependencies, making them much more lightweight.
    - Containers are faster to start and stop because they don't need to boot up an entire OS.
    - Containers are managed by container engines (like Docker) and don't require a hypervisor.

Container Advantages over VMs:

1. **Efficiency**: Containers use fewer resources because they share the OS kernel and do not require a separate guest OS.
2. **Faster Start-Up**: Containers can start almost instantly, whereas VMs need to boot up an entire operating system.
3. **Portability**: Containers encapsulate everything needed for an app to run, making it easier to move between environments (from local to cloud to production).

How Containers Work

1. **Container Engine**: A container engine (like Docker) is responsible for creating, managing, and running containers. It uses **container images** (pre-built blueprints) to start containers. The engine also handles networking, storage, and other configurations needed to run the container.

2. **Container Image**: A container image contains everything needed to run an application: the application code, libraries, system tools, settings, and dependencies. The image is read-only and reusable.

3. **Running the Container**: When you run a container, it creates a writable layer on top of the image. This layer contains any changes made by the application while running (e.g., file writes or configurations). When the container is stopped or deleted, this writable layer can be discarded unless explicitly saved as a new image.

Example: Docker Containers

Docker is the most widely used container platform. Here's a simple example to demonstrate how containers work with Docker:

1. **Create a Dockerfile**: This file defines the steps to build a Docker image.

```
        # Use a base image (Ubuntu)
FROM ubuntu:latest

# Install necessary packages (e.g., curl)
RUN apt-get update && apt-get install -y curl

# Set a command to run
CMD ["echo", "Hello, World!"]
```

2. **Build the Image**: Using the docker build command, you can build an image from the Dockerfile.

```
docker build -t my-ubuntu-container .
```

3. **Run the Container**: Once the image is built, you can run a container from the image:

```
docker run my-ubuntu-container
```

Output:

```
Hello, World!
```

4. **Check Running Containers**: You can view all running containers with:

```
docker ps
```

Common Use Cases for Containers

1. **Microservices Architecture**: Containers are ideal for running microservices because they allow you to easily package and deploy each service independently.
2. **Continuous Integration (CI) and Continuous Deployment (CD)**: Containers are commonly used in CI/CD pipelines to ensure that code runs consistently across different stages (dev, test, prod).
3. **Development and Testing**: Developers use containers to test their applications in isolated environments, ensuring that the app will behave the same way when it's deployed.
4. **Cloud-Native Applications**: Containers are essential in cloud-native environments, where applications need to be quickly scaled and moved across various infrastructure providers.

Summary of Container Benefits:

- **Isolation**: Containers isolate applications from each other and from the host system.
- **Portability**: Applications packaged in containers run consistently on any environment that supports the container engine.
- **Efficiency**: Containers are lightweight and share the host OS kernel, which makes them resource-efficient.
- **Scalability**: Containers can be easily scaled up or down in response to changing demand.

Containers are fundamental to modern application deployment, making it easier to develop, ship, and run applications reliably across different environments.

Chapter 2: Docker Overview & Architecture

Docker is an open-source platform that automates the deployment, scaling, and management of applications inside lightweight containers. Containers allow applications to run in isolated environments, ensuring they work the same regardless of the system or environment they are running on. Docker simplifies packaging and distribution of software by bundling the application and all of its dependencies into a single, portable container image.

Docker uses **containerization technology**, which is distinct from traditional virtual machines (VMs), to provide a more efficient way to manage applications. Containers share the same operating system kernel but run in isolated user spaces, making them much more lightweight and faster than VMs.

Key Concepts of Docker

1. **Container**: A lightweight, standalone executable package that includes everything needed to run a piece of software: the code, runtime, libraries, environment variables, and configurations.

2. **Image**: A read-only template used to create containers. Docker images are the building blocks of containers, and they define the application's environment, libraries, and configurations. Images are often shared through registries.

3. **Dockerfile**: A text file that contains a series of instructions to build a Docker image. It defines everything from the base operating system to the application dependencies, and the commands to run when the container starts.

4. **Docker Registry**: A repository where Docker images are stored and shared. Docker Hub is the default public registry, but there are also private registries such as **Amazon ECR** and **Google Container Registry (GCR)**.

5. **Docker Engine**: The core component of Docker, which is responsible for creating and managing containers. It runs on the host operating system and interacts with the container runtime.

6. **Docker Compose**: A tool for defining and running multi-container Docker applications. Using a docker-compose.yml file, you can define services, networks, and volumes to create a complete application stack.

Docker Architecture

The Docker architecture is designed to be modular and scalable. At its core, Docker has the following components:

1. Docker Client

The **Docker Client** is the interface that a user interacts with. It can be a command-line interface (CLI) or a GUI. The Docker client sends requests to the Docker daemon (engine) through the Docker API. Common commands include:

- docker build: To build Docker images.
- docker run: To create and start containers from images.
- docker ps: To list running containers.
- docker pull: To pull images from a registry.
- docker push: To push images to a registry.

The Docker client communicates with the Docker daemon over the Docker API. It's important to note that the Docker client and daemon can be on the same machine or different machines.

2. Docker Daemon (dockerd)

The **Docker Daemon** (or dockerd) is the core component of Docker. It is responsible for managing Docker images, containers, networks, and volumes. It listens for API requests and handles the container lifecycle (start, stop, delete).

Key responsibilities of the Docker Daemon:

- **Build images** from Dockerfiles.
- **Run containers** from images.
- **Manage containers** (start, stop, pause, resume).
- **Monitor container state** (such as logging, health checks).
- **Pull/push images** to/from registries.

The Docker daemon is typically running on the host machine, and it can be managed via the Docker CLI or an API.

3. Docker Images

Docker Images are the building blocks of containers. An image is a **read-only** template that includes everything needed to run an application. It includes the operating system layers, application code, runtime, libraries, and dependencies.

Images are created from Dockerfiles, which define how the image is built. Docker images can be stored in a **Docker registry**, such as Docker Hub or a private registry.

Key image operations:

- **Build**: Create a Docker image using a Dockerfile (docker build).
- **Pull**: Download an image from a registry (docker pull).
- **Push**: Upload an image to a registry (docker push).

4. Docker Containers

Docker Containers are runtime instances of Docker images. When you run an image, Docker creates a container from it. A container is an isolated environment that contains everything needed to run the application.

Containers are **ephemeral** by default, which means they can be started and stopped without affecting the underlying system. Each container runs as an isolated process, and the filesystem used by containers is built using layers from the Docker image.

Key container operations:

- **Run**: Start a new container from an image (docker run).
- **Stop**: Stop a running container (docker stop).
- **Exec**: Execute commands inside a running container (docker exec).
- **Logs**: Retrieve logs from running containers (docker logs).

5. Docker Registries

A **Docker Registry** is a service where Docker images are stored and shared. Docker Hub is the default public registry, but you can also use private registries. Registries allow you to push and pull images as needed.

- **Docker Hub**: The public default registry provided by Docker.
- **Private Registries**: Companies often use private registries to store their custom images, such as **Amazon ECR** or **Google Container Registry (GCR)**.

Images are pulled from the registry when creating containers and pushed to the registry when updating or creating new images.

6. Docker Volumes

Docker Volumes are used for persistent data storage. By default, any data created inside a container will be lost once the container is deleted. Volumes allow data to persist even after a container is stopped or removed.

- **Mounting Volumes**: You can mount volumes to containers to share data between them or persist data beyond the life of a container.
- **Named Volumes**: Volumes managed by Docker, often used to store persistent application data.
- **Bind Mounts**: Bind a host directory or file to a container.

7. Docker Networks

Docker Networks provide isolated communication between containers. Containers on the same network can communicate with each other, while containers on different networks are isolated.

- **Bridge Network**: The default network mode, where containers on the same host can communicate with each other.
- **Host Network**: The container shares the host's network stack.
- **Overlay Network**: Allows containers on different Docker hosts to communicate, used in multi-host or multi-node setups.

Docker's Layered Architecture

Docker uses a layered approach for images and containers, which helps optimize storage and performance.

1. **Images are layered**: Each image is made up of layers. These layers are read-only and include everything from the base OS to the application code. When a container is created from an image, Docker adds a read-write layer on top of it.
2. **Layer caching**: Docker caches layers to avoid rebuilding them from scratch, improving build performance.
3. **Copy-on-write**: When a container writes to its filesystem, it creates a new layer, leaving the image layers intact. This reduces duplication of data and ensures the image stays unchanged.

Docker Workflow

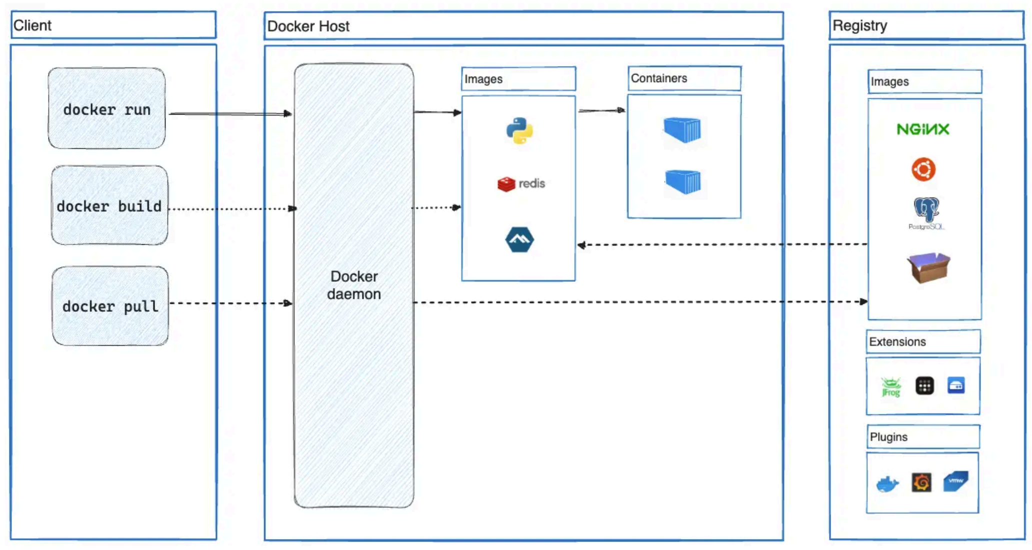
1. **Create a Dockerfile**: Write a Dockerfile that specifies the environment and instructions to build an image.
2. **Build the Image**: Use the docker build command to create an image from the Dockerfile.
3. **Run a Container**: Use the docker run command to create and start a container from the image.
4. **Push/Pull Images**: Use docker push and docker pull to push images to or pull images from a registry.
5. **Manage Containers**: Use docker ps, docker stop, docker rm, and other commands to manage container lifecycle.

Docker Architecture Diagram

Here's a simplified view of Docker's architecture:

```
+---------------------------------------------------------------+
|                  Docker Client                   |
| (CLI or GUI - User interacts with the Docker Engine)        |
+---------------------------------------------------------------+
                  | (API calls)
                  V
+---------------------------------------------------------------+
|                Docker Daemon (dockerd)              |
| (Manages containers, images, networks, and volumes)          |
+---------------------------------------------------------------+
    |                  |                   |
    V                  V                   V
+------------+      +--------------+      +---------------+
| Docker     |      | Docker       |      | Docker Volumes |
| Containers |       | Images      |       | (Data storage) |
+------------+      +--------------+      +---------------+
```

Here is the Architecture in graphical View:



Summary of Key Components:

- **Docker Client**: The interface that interacts with the Docker Daemon, typically through the command line or a GUI.
- **Docker Daemon**: The engine that handles container creation, management, and interaction with images, networks, and volumes.
- **Docker Images**: The base blueprints for containers, containing the application and all dependencies.
- **Docker Containers**: The running instances of Docker images.
- **Docker Registry**: A repository for storing and sharing Docker images (e.g., Docker Hub).
- **Docker Volumes**: Storage used for persisting data beyond the lifecycle of a container.
- **Docker Networks**: Allow containers to communicate with each other.

Docker Benefits:

- **Lightweight**: Containers share the host OS kernel, making them more efficient than VMs.
- **Portable**: Containers can be easily moved between different environments.
- **Consistent**: Containers ensure applications run consistently across different systems.
- **Scalable**: Easily scale applications by running multiple containers.
- **Faster Deployment**: Containers are faster to start and stop compared to VMs.

Docker simplifies application deployment and management, making it easier to move

Chapter 3: Installing Docker

To start using Docker, you need to install Docker Engine on your system. Below is a step-by-step guide to installing Docker on different platforms: **Windows**, **macOS**, and **Linux**.

---

1. Installing Docker on Windows

Docker on Windows runs via a utility called **Docker Desktop**. Docker Desktop provides a GUI to manage Docker containers and images, along with the Docker CLI for command-line interaction.

Prerequisites:

- Windows 10 or 11 (Pro, Enterprise, or Education) with **Hyper-V** and **Windows Subsystem for Linux (WSL 2)** enabled.
- At least 4GB of RAM.

Steps:

1. **Download Docker Desktop for Windows**:
    - Go to the Docker website: Docker Desktop for Windows.
    - Click "Download Docker Desktop."
2. **Install Docker Desktop**:
    - Run the installer after it finishes downloading.
    - Follow the installation instructions. The installer will guide you through enabling the necessary features (like WSL 2).
    - Once the installation is complete, you may be prompted to restart your machine.
3. **Enable WSL 2 and Hyper-V**:
    - Docker Desktop requires **WSL 2** for better performance, and **Hyper-V** is used for virtualization.
    - You'll be prompted to enable these during the installation. If you encounter any issues, follow these instructions:
        - **WSL 2 Installation**: Microsoft WSL 2 Installation Guide
        - **Enable Hyper-V**: You can enable it manually through the Windows Features menu.
4. **Start Docker Desktop**:

   - After the restart, launch Docker Desktop from the Start menu.
   - Docker will initialize, and you should see a Docker icon in the system tray once it's running.
5. **Verify Installation**:

Open a terminal (Command Prompt or PowerShell), and type:

```
docker --version
```

   - You should see the version of Docker installed on your machine.

---

2. Installing Docker on macOS

Docker on macOS also uses **Docker Desktop**. Like Windows, it provides a GUI and integrates with macOS through a virtualization layer.

Prerequisites:

- macOS 10.14 or later (Mojave, Catalina, Big Sur, Monterey, or Ventura).
- At least 4GB of RAM.

Steps:

1. **Download Docker Desktop for macOS**:
   - Go to the Docker website: Docker Desktop for macOS.
   - Click "Download Docker Desktop."
2. **Install Docker Desktop**:
   - Open the downloaded .dmg file and drag the Docker icon to your Applications folder.
   - Launch Docker from the Applications folder.
3. **Start Docker Desktop**:
   - After launching, Docker will start in the background, and you should see the Docker icon in your top menu bar.
   - Docker may take a moment to initialize. Once it's ready, you should see the "Docker is running" status.
4. **Verify Installation**:

Open a terminal window (you can use **Terminal.app** or iTerm2) and type:

```
docker --version
```

   -

- You should see the Docker version installed on your machine.

---

3. Installing Docker on Linux

On Linux, Docker is typically installed through the package manager. The installation process varies depending on the distribution you are using. Below, we'll go through the steps for **Ubuntu** (Debian-based) and **CentOS** (Red Hat-based).

Installing Docker on Ubuntu (Debian-based)

1. **Update the apt package index**:

```
sudo apt update
```

2. **Install required dependencies**:

```
sudo apt install \
apt-transport-https \
ca-certificates \
curl \
software-properties-common
```

3. **Add Docker's official GPG key**:

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key
add -
```

4. **Add the Docker APT repository**:

```
sudo add-apt-repository \
"deb [arch=amd64] https://download.docker.com/linux/ubuntu \
$(lsb_release -cs) \
stable"
```

5. **Update the apt package index again**:

```
sudo apt update
```

6. **Install Docker Engine**:

   ```
   sudo apt install docker-ce
   ```
7. **Start Docker and enable it to run at boot**:

   ```
   sudo systemctl start docker
   sudo systemctl enable docker
   ```

8. **Verify Docker Installation**:

To verify that Docker is running, use:

```
sudo docker --version
```

- To check Docker's status:

  ```
  sudo systemctl status docker
  ```

Installing Docker on CentOS (Red Hat-based)

1. **Update your system**:

   ```
   sudo yum update
   ```
2. **Install required dependencies**:

   ```
   sudo yum install -y yum-utils
   ```

3. **Add Docker's official repository**:

   ```
   sudo yum-config-manager --add-repo
   https://download.docker.com/linux/centos/docker-ce.repo
   ```

4. **Install Docker Engine**:

   ```
   sudo yum install docker-ce docker-ce-cli containerd.io
   ```

5. **Start Docker and enable it to run at boot**:

   ```
   sudo systemctl start docker
   ```

```
sudo systemctl enable docker
```

6. **Verify Docker Installation**:
   - To check Docker's version:
     ```
     sudo docker --version
     ```
   - To check Docker's status:
     ```
     sudo systemctl status docker
     ```

Post-installation Setup (Linux Specific)

To run Docker commands without sudo, you can add your user to the Docker group:

1. **Create the Docker group** (if it doesn't exist):

   ```
   sudo groupadd docker
   ```

2. **Add your user to the Docker group**:

   ```
   sudo usermod -aG docker $USER
   ```
3. **Log out and log back in**, or restart your session:
   - This ensures the changes take effect. Now you should be able to run Docker commands without sudo.

Verifying Docker Installation

After installing Docker, you can verify the installation by running a simple test. In the terminal, execute:

```
docker run hello-world
```

This command will:

- Download the hello-world image if it's not already available on your system.
- Run the container, which will print a "Hello from Docker!" message if everything is set up correctly.

Troubleshooting

- **Windows and macOS**:
  - If Docker Desktop does not start, try restarting the system or ensuring that **Hyper-V** (Windows) or **VirtualBox** (macOS) is installed and properly configured.
- **Linux**:
  - Make sure that Docker is running:
    ```
    sudo systemctl status docker
    ```
  - If Docker isn't starting, use:
    ```
    sudo systemctl start docker
    ```

Basic Docker Commands

- `docker --version`: Check Docker version.
- `docker pull <image>`: Download an image from Docker Hub.
- `docker images`: List available Docker images.
- `docker run <image>`: Create and start a container from an image.
- `docker ps`: List running containers.
- `docker stop <container_id>`: Stop a running container.
- `docker rm <container_id>`: Remove a stopped container.
- `docker rmi <image_id>`: Remove a Docker image.

Conclusion

Now that Docker is installed, you can start creating, managing, and running containers. Docker provides a consistent environment for development, testing, and deployment across different platforms and environments.

LAB 1: Containerize NodeJS App

step-by-step guide on how to containerize a simple Node.js application and push the Docker image to Docker Hub.

Prerequisites:

1. **Docker installed**: Make sure you have Docker installed on your local machine.
2. **Node.js application**: You need a simple Node.js app to containerize.
3. **Docker Hub account**: If you don't already have one, sign up at Docker Hub.
4. **Docker CLI access**: You need to be able to use docker commands in your terminal.
5. Install Node and npm

```
sudo apt install nodejs

sudo apt install npm

node -v
```

Step 1: Create a Simple Node.js Application

Let's create a basic Node.js app for thisLab.

Create a project directory:

```
mkdir node-docker-demo
cd node-docker-demo
```

1. **Initialize a Node.js project**:

```
npm init -y
```

2. **Install Express (for simplicity)**:

```
npm install express
```

3. **Create the application code**: Create a file called app.js and add the following code:

```
const express = require('express');

const app = express();

const port = process.env.PORT || 3000;
```

```
app.get('/', (req, res) => {

  res.send('Hello from Dockerized Node.js app!');

});

app.listen(port, () => {

  console.log(`Server running on http://localhost:${port}`);

});
```

4. **Test your app locally**: Run the app using node to ensure it works.

   ```
   node app.js
   ```
5. Open your browser and go to http://localhost:3000 to see the message "Hello from Dockerized Node.js app!"

Step 2: Create a Dockerfile

In the project directory, create a file called Dockerfile (no extension). Add the following content:

```
# Use the official Node.js image as the base image
FROM node:14

# Set the working directory inside the container
WORKDIR /usr/src/app

# Copy package.json and package-lock.json
COPY package*.json ./

# Install dependencies
RUN npm install

# Copy the rest of the app's source code
COPY . .

# Expose the port the app runs on
EXPOSE 3000

# Define the command to run the app
CMD ["node", "app.js"]
```

1. Explanation of each line:

   - FROM node:14: Using the official Node.js image as the base.

- WORKDIR /usr/src/app: Sets the working directory inside the container.
- COPY package*.json ./: Copies the package files (for dependencies).
- RUN npm install: Installs the dependencies.
- COPY . . : Copies the rest of the application code.
- EXPOSE 3000: Exposes port 3000, which is the port your app will run on.
- CMD ["node", "app.js"]: Runs the application when the container starts.

Step 3: Build the Docker Image

1. Open your terminal in the project directory (where the Dockerfile is located).

Build the Docker image using the docker build command:

```
docker build -t your-username/node-docker-demo .
```

2. Replace your-username with your Docker Hub username. This will create a Docker image with the tag your-username/node-docker-demo.

Check that your image was built successfully by listing your local Docker images:

```
docker images
```

3. **Step 4: Run the Docker Container Locally**

Run the Docker container:

```
docker run -p 3000:3000 your-username/node-docker-demo
```

1. Open your browser and navigate to http://localhost:3000.
2. You should see the message: Hello from Dockerized Node.js app!

Step 5: Push the Docker Image to Docker Hub

Log in to Docker Hub: If you're not logged in, run the following command and enter your Docker Hub credentials:

```
docker login
```

1. **Tag the image (if needed)**: If you didn't use the correct tag when building the image, you can tag it like so:

```
docker tag your-username/node-docker-demo your-username/node-docker-demo:latest
```

2. **Push the image to Docker Hub**: Push the Docker image to your Docker Hub repository with the following command:

```
docker push your-username/node-docker-demo:latest
```

3. Once the image is pushed successfully, you can verify it on your Docker Hub repository page.

Step 6: Verify the Image on Docker Hub

1. Go to your Docker Hub profile: https://hub.docker.com
2. You should see the node-docker-demo repository listed under your repositories.
3. The image will be listed there with the latest tag, and you can now share the image or use it in other environments.

Step 7: Clean Up

If you want to remove the container and images from your local system after you're done testing, you can run:

Stop and remove the container:

```
 docker ps  # To find the container ID
docker stop <container-id>
docker rm <container-id>
```

1. **Remove the image**:

```
docker rmi your-username/node-docker-demo:latest
```

Conclusion

In this lab, take note of the ports used, which is the default for Node applications.

Congratulations: You've successfully created a simple Node.js app, containerized it using Docker, and pushed it to Docker Hub. Now, you can pull and run this image anywhere.

LAB 2: Containerize Python Fast-api app

A step-by-step guide to containerizing a simple FastAPI Python application and pushing the Docker image to Docker Hub.

Prerequisites:

1. **Docker**: Make sure Docker is installed on your machine.
2. **Python 3.7+**: FastAPI requires Python 3.7 or above.
3. **Docker Hub account**: You need to have a Docker Hub account to push the image.
4. **Docker CLI access**: You should be able to run docker commands in your terminal.
5. Python3 is installed

```
sudo apt update

sudo apt install python3

python3 --version
```

Step 1: Create a Simple FastAPI Application

Create a new directory for your FastAPI project:

```
 mkdir fastapi-docker-demo

cd fastapi-docker-demo
```

1. **Set up a virtual environment (optional but recommended)**:

   ```
   python -m venv venv

   source venv/bin/activate  # For Windows, use `venv\Scripts\activate`
   ```

2. **Install FastAPI and Uvicorn**: FastAPI is the web framework, and Uvicorn is the ASGI server that serves the app.

   ```
   pip install fastapi uvicorn
   ```

3. **Create the FastAPI app**: Inside the project directory, create a file called main.py and add the following code:

```
 from fastapi import FastAPI
app = FastAPI()
@app.get("/")
def read_root():
    return {"message": "Hello, Dockerized FastAPI app!"}
```

4. **Test the application locally**: You can test your FastAPI app locally before Dockerizing it. Run it with Uvicorn:

```
uvicorn main:app --reload
```

Navigate to http://localhost:8000 in your browser, and you should see the message:

```
{"message": "Hello, Dockerized FastAPI app!"}
```

Step 2: Create a Dockerfile

Next, you'll need a Dockerfile to containerize the FastAPI application.

Create a Dockerfile in the same directory as main.py. Here's the content for the Dockerfile:

```
 # Use the official Python image from Docker Hub

FROM python:3.9-slim

# Set the working directory inside the container

WORKDIR /app

# Copy the requirements file (we'll create it next)

COPY requirements.txt .

# Install the Python dependencies
```

```
RUN pip install --no-cache-dir -r requirements.txt

# Copy the FastAPI app code

COPY . .

# Expose the port that FastAPI will run on

EXPOSE 8000

# Set the default command to run the app with Uvicorn

CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

This Dockerfile does the following:

- Uses a slim version of the Python 3.9 image as the base image.
- Sets a working directory in the container.
- Installs Python dependencies.
- Copies the application code into the container.
- Exposes port 8000, which is where FastAPI runs.
- Sets the default command to run the FastAPI app using Uvicorn.

Create a requirements.txt file: FastAPI and Uvicorn are needed to run the app, so create a requirements.txt with the following content:

```
fastapi==0.75.0

uvicorn==0.17.0
```

Step 3: Build the Docker Image

Now, let's build the Docker image from the Dockerfile.

Build the Docker image: In the project directory (where the Dockerfile is), run the following command to build the image:

```
docker build -t your-username/fastapi-docker-demo .
```

1. Replace your-username with your Docker Hub username.

Verify the image was built: To see the built image, run:

```
docker images
```

2. You should see the your-username/fastapi-docker-demo image listed there.

Step 4: Run the Docker Container Locally

Now let's test if the container works locally.

Run the Docker container:

```
docker run -p 8000:8000 your-username/fastapi-docker-demo
```

This will run the container and map port 8000 from the container to port 8000 on your host.

Test the application: Open your browser and go to http://localhost:8000. You should see the JSON response:

```
{"message": "Hello, Dockerized FastAPI app!"}
```

Step 5: Push the Image to Docker Hub

Once the image is built and tested, you can push it to Docker Hub.

Log in to Docker Hub: If you are not logged in, run the following command and enter your Docker Hub credentials:

```
docker login
```

1. **Tag the image** (optional if not done already): If you didn't tag the image with your Docker Hub username during the build step, you can do it now:

   ```
   docker tag fastapi-docker-demo
   your-username/fastapi-docker-demo:latest
   ```
2. **Push the image to Docker Hub**: Now push the image to your Docker Hub repository:

   ```
   docker push your-username/fastapi-docker-demo:latest
   ```
3. This will upload the image to Docker Hub. Depending on your internet connection and the size of the image, this could take some time.

4. **Verify the image on Docker Hub**: Go to your Docker Hub profile: https://hub.docker.com and check your repositories. You should see the fastapi-docker-demo repository, and the image should have been pushed successfully.

Step 6: Clean Up

If you want to clean up your local environment:

Stop and remove the running container: If your container is still running, find its container ID and stop it:

```
 docker ps  # To find the container ID

docker stop <container-id>

docker rm <container-id>
```

Remove the Docker image (optional): If you want to remove the image from your local system after pushing it to Docker Hub:

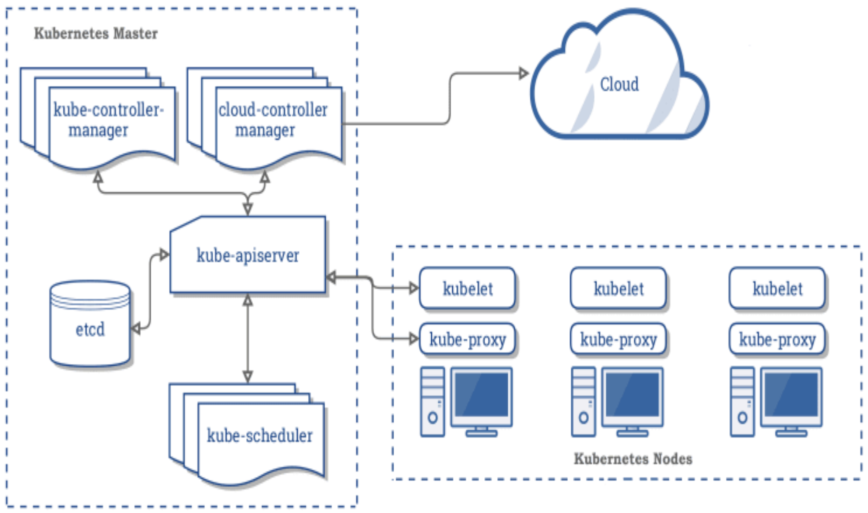
```
 docker rmi your-username/fastapi-docker-demo:latest
```

Conclusion

In this lab, take note of the ports used, whihc is the default for fast-api.

Congratulatios!!!You've now successfully containerized your FastAPI application, tested it locally, and pushed the Docker image to Docker Hub.

Chapter 4: Kubernetes Architecture



Control Plane Components:

- Kube-apiserver manager
- Kube-controller-manager
- Kube-scheduler
- Etcd
- Cloud-controller-manager

Data Plane Components:

- Kubelet
- Kube-proxy
- Docker-runtime

LAB: Enable Kubernetes in Docker Desktop

Docker Desktop should already be installed. If not installed, use the link below to install Docker Desktop. Docker Desktop installs the all-in-one (AIO) Kubernetes. This means that every component of Kubernetes including kube-api, kube-controller, kube-scheduler, etcd, kubelet, kube-proxy etc are all installed on the same node.

https://www.docker.com/products/docker-desktop/

Once Docker Desktop is installed you can enable Kubernetes by using teh following steps:

1. Click on Docker Desktop
2. Go to Settings
3. Click on Kubernetes
4. Check Enable Kubernetes
5. Click Apply and Reset

Docker Desktop will restart and enable Kubernetes. To test that Kubernetes is working, go to the CLI and type;

```
kubectl version
```

LAB: Getting started with kubectl, the utility for managing Kubernetes

The Kubernetes client, ***kubectl*** is the primary method of interacting with a Kubernetes cluster. Getting to know it is essential to use Kubernetes itself.

.

Syntax Structure

Kubectl uses a common syntax for all operations in the form of:

```
kubectl <command> <type> <name> <flags>
```

- **command** - The command or operation to perform.
  e.g. apply, create, delete, and get.
- **Type** - The resource type or object, e.g. pod, deployment, ervice
- **Name** - The name of the resource or object.
- **Flags** - Optional flags to pass to the command.

Examples

Here is a Pod manifest file. Copy and past the content in a file called mypod.yaml

mypod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
    - name: nginx
      image: nginx
      ports:
        - containerPort: 80
```

```
$ kubectl create -f mypod.yaml
$ kubectl get pods
$ kubectl get pod mypod
$ kubectl delete pod mypod
```

Context and kubeconfig

Kubectl allows a user to interact with and manage multiple Kubernetes clusters. To do this, it requires what is known as a context. A context consists of a combination of ***cluster, namespace and user.***

- **Cluster** - A friendly name, server address, and certificate for the Kubernetes cluster.
- **Namespace (optional)** - The logical cluster or environment to use. If none is provided, it will use the default ***default*** namespace.
- **User** - The credentials used to connect to the cluster. This can be a combination of client **certificate and key, username/password, or token**.

These contexts are stored in a local yaml based config file referred to as the ***kubeconfig***. For *nix based systems, the **kubeconfig** is stored in **$HOME/.kube/config**

This config is viewable without having to view the file directly.

Command

```
$ kubectl config view
```

Example

```
$ kubectl config view

apiVersion: v1
```

```
clusters:
- cluster:
    certificate-authority-data: DATA+OMITTED
    server: https://165.22.13.187:6443
  name: kubernetes
contexts:
- context:
    cluster: kubernetes
    user: kubernetes-admin
  name: kubernetes-admin@kubernetes
current-context: kubernetes-admin@kubernetes
kind: Config
preferences: {}
users:
- name: kubernetes-admin
  user:
    client-certificate-data: REDACTED
    client-key-data: REDACTED
```

kubectl config
Managing all aspects of contexts is done via the ***kubectl config*** command. Some examples include:

- See the active context with ***kubectl config current-context***.
- Get a list of available contexts with ***kubectl config get-contexts.***
- Switch to using another context with the ***kubectl config use-context <context-name>*** command.
- Add a new context with ***kubectl config set-context <context name> --cluster=<cluster name> --user=<user> --namespace=<namespace>***.

Exercise: Using Contexts

Objective: Create a new context called ***minidev*** and switch to it.

1. View the current contexts.

   ```
   $ kubectl config get-contexts
   ```

2. Take note of the current context and write it down

   ```
   $ kubectl config current-context
   ```

3. Create a new context called *minidev* within the *kubernetes* cluster with the *dev* namespace, as the *kubernetes-admin user*

```
$ kubectl config set-context minidev --cluster=docker-desktop
--user=admin-user --namespace=dev
```

3. View the newly added context.

```
 kubectl config get-contexts
```

4. Switch to the minidev context using *use-context*.

```
$ kubectl config use-context minidev
```

5. View the current active context.

```
$ kubectl config current-context
```

5. Switch back to your previous context

```
$ kubectl config use-context  docker-desktop
```

.

Summary: Understanding and being able to switch between contexts is a base fundamental skill required by every Kubernetes user. As more clusters and namespaces are added, this can become unwieldy. Installing a helper application such as kubectx can be quite helpful. Kubectx allows a user to quickly switch between contexts and namespaces without having to use the full kubectl config use-context command.

Kubectl Basics

There are several ***kubectl*** commands that are frequently used for any sort of day-to-day operations. ***get, create, apply, delete, describe, and logs***. Other commands can be listed simply with ***kubectl --help***, or kubectl <command> --help.

kubectl get

.

kubectl get fetches and lists objects of a certain type or a specific object itself. It also supports outputting the information in several different useful formats including: json, yaml, wide (additional columns), or name (names only) via the -o or –output flag.

Command

```
kubectl get <type>
kubectl get <type> <name>
kubectl get <type> <name> -o <output format>
```

.

Examples

```
$ kubectl get namespaces
NAME              STATUS   AGE
default           Active   75m
kube-node-lease   Active   75m
kube-public       Active   75m
kube-system       Active   75m

$kubectl get pod  -o wide
NAME                     READY   STATUS    RESTARTS   AGE   IP          NODE
NOMINATED NODE   READINESS GATES
nginx-7bb7cd8db5-bsz5g   1/1     Running   0          61m   10.44.0.1
ubuntu-s-1vcpu-1gb-nyc1-01   <none>           <none>
```

.

kubectl create

kubectl create creates an object from the commandline (stdin) or a supplied json/yaml manifest. The manifests can be specified with the -f or –filename flag that can point to either a file, or a directory containing multiple manifests.

Command

```
kubectl create <type> <parameters>
kubectl create -f <path to manifest>
```

Examples

```
$ kubectl create namespace dev
namespace "dev" created

$ kubectl create -f manifests/mypod.yaml
pod "mypod" created
```

.

kubectl apply

kubectl apply is similar to kubectl create. It will essentially update the resource if it is already created, or simply create it if does not yet exist. When it updates the config, it will save the previous version of it in an annotation on the created object

itself. **WARNING:** If the object was not created initially with ***kubectl apply*** it's updating behavior will act as a two-way diff.
Just like ***kubectl create*** it takes a json or yaml manifest with the ***-f flag*** or accepts input from stdin.
Command

```
kubectl apply -f <path to manifest>
```

Examples

```
$ kubectl apply -f manifests/mypod.yaml
Warning: kubectl apply should be used on resource created by either kubectl
create --save-config or kubectl apply
pod "mypod" configured
```

.

kubectl edit
kubectl edit modifies a resource in place without having to apply an updated manifest. It fetches a copy of the desired object and opens it locally with the configured text editor, set by the KUBE_EDITOR or EDITOR Environment Variables. This command is useful for troubleshooting, but should be avoided in production scenarios as the changes will essentially be untracked.
Command

```
$ kubectl edit <type> <object name>
```

Examples

```
kubectl edit pod mypod
kubectl edit service myservice
```

.

kubectl delete

kubectl delete deletes the object from Kubernetes.

Command

```
kubectl delete <type> <name>
```

Examples

```
$ kubectl delete pod mypod
pod "mypod" deleted
```

.

kubectl describe
kubectl describe lists detailed information about the specific Kubernetes object. It is a very helpful troubleshooting tool.
Command

```
kubectl describe <type>
kubectl describe <type> <name>
```

Examples

```
kubectl  describe pod nginx-7bb7cd8db5-bsz5g

Name:           nginx-7bb7cd8db5-bsz5g
Namespace:      default
Priority:       0
Node:           ubuntu-s-1vcpu-1gb-nyc1-01/134.209.208.179
Start Time:     Mon, 05 Aug 2019 01:38:57 +0000
Labels:         pod-template-hash=7bb7cd8db5
                run=nginx
Annotations:    <none>
Status:         Running
IP:             10.44.0.1
Controlled By:  ReplicaSet/nginx-7bb7cd8db5
Containers:
  nginx:
    Container ID:
docker://b86aaf4ab9f31c11ebac3ee70d52403c33753511dde9263ca09b8f7a1687d5
89
    Image:          nginx
    Image ID:
docker-pullable://nginx@sha256:eb3320e2f9ca409b7c0aa71aea3cf7ce7d018f03a3
72564dbdb023646958770b
    Port:           <none>
    Host Port:      <none>
    State:          Running
      Started:      Mon, 05 Aug 2019 01:39:05 +0000
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-wr4l5 (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Volumes:
  default-token-wr4l5:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-wr4l5
    Optional:    false
QoS Class:       BestEffort
Node-Selectors:  <none>
Tolerations:     node.kubernetes.io/not-ready:NoExecute for 300s
                 node.kubernetes.io/unreachable:NoExecute for 300s
Events:          <none>
```

.

kubectl logs

kubectl logs outputs the combined stdout and stderr logs from a pod. If more than one container exist in a pod the -c flag is used and the container name must be specified.

Command

```
kubectl logs <pod name>
kubectl logs <pod name> -c <container name>
```

Examples

```
$ kubectl logs mypod
172.17.0.1 - - [10/Mar/2018:18:14:15 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.57.0"
"-"
172.17.0.1 - - [10/Mar/2018:18:14:17 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.57.0"
"-"
```

kubectl api-resources

This will list all the Kubernetes API objects in the cluster

Expected Output:

```
NAME                 SHORTNAMES   APIVERSION          NAMESPACED   KIND
bindings                          v1                  true         Binding
componentstatuses    cs           v1                  false        ComponentStatus
configmaps           cm           v1                  true         ConfigMap
endpoints            ep           v1                  true         Endpoints
events               ev           v1                  true         Event
limitranges          limits       v1                  true         LimitRange
namespaces           ns           v1                  false        Namespace
nodes                no           v1                  false        Node
```

```
persistentvolumeclaims          pvc          v1                        true       PersistentVolumeClaim
persistentvolumes               pv           v1                        false      PersistentVolume
pods                            po           v1                        true       Pod
podtemplates                                 v1                        true       PodTemplate
replicationcontrollers          rc           v1                        true       ReplicationController
resourcequotas                  quota        v1                        true       ResourceQuota
secrets                                      v1                        true       Secret
serviceaccounts                 sa           v1                        true       ServiceAccount
services                        svc          v1                        true       Service
mutatingwebhookconfigurations                admissionregistration.k8s.io/v1   false
MutatingWebhookConfiguration
validatingwebhookconfigurations              admissionregistration.k8s.io/v1   false
ValidatingWebhookConfiguration
customresourcedefinitions       crd,crds     apiextensions.k8s.io/v1           false
CustomResourceDefinition
apiservices                                  apiregistration.k8s.io/v1         false      APIService
controllerrevisions                          apps/v1                   true       ControllerRevision
.
.
.
```

kubectl explain resource_name

Explian is like the Linux man command. It gives a short description of the API object.

For example:

kubectl explain pod

This will give a short description of the pod resource

Exercise: The Basics

Objective: Explore the basics. Create a namespace, a pod, then use the kubectl commands to describe and delete what was created.

Here is the mypod.yaml file

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
```

```
containers:
  - name: nginx
    image: nginx
    ports:
      - containerPort: 80
```

.

1. Create the dev namespace.

   ```
   kubectl create namespace dev
   ```

2. Apply the manifest manifests/mypod.yaml.

   ```
   kubectl apply -f manifests/mypod.yaml -n dev
   ```

3. Get the yaml output of the created pod mypod.

   ```
   kubectl get pod mypod -o yaml -n dev
   ```

4. Describe the pod mypod.

   ```
   kubectl describe pod mypod -n dev
   ```

5. Clean up the pod by deleting it.

   ```
   kubectl delete pod mypod -n dev
   ```

.

Summary: The kubectl *"CRUD"* commands are used frequently when interacting with a Kubernetes cluster. These simple tasks become 2nd nature as more experience is gained.

.Accessing the Cluster

Kubectl provides several mechanisms for accessing resources within the cluster remotely. For this lab, the focus will be on using ***kubectl exec*** to get a remote shell within a container, and ***kubectl proxy*** to gain access to the services exposed through the API proxy.

.

kubectl exec

kubectl exec executes a command within a Pod and can optionally spawn an interactive terminal within a remote container. When more than one container is present within a Pod, the -c or –container flag is required, followed by the container name.
If an interactive session is desired, the -i (--stdin) and -t(--tty) flags must be supplied.
Command

```
kubectl exec <pod name> -- <arg>
kubectl exec <pod name> -c <container name> -- <arg>
kubectl exec  -i -t <pod name> -c <container name> -- <arg>
kubectl exec  -it <pod name> -c <container name> -- <arg>
```

.

Exercise: Executing Commands within a Remote Pod

Objective: Use ***kubectl exec*** to both initiate commands and spawn an interactive shell within a Pod.

Here is the mypod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
    - name: nginx
      image: nginx
      ports:
        - containerPort: 80
```

.

1. Put the above information in a file called mypod.yaml and, create the Nginx Pod mypod from the manifest mypod.yaml.

   ```
   $ kubectl create -f mypod.yaml
   ```

2. Wait for the Pod to become ready (running).

   ```
   $ kubectl get pods
   ```

3. Use kubectl exec to cat the file /etc/os-release.

   ```
   $ kubectl exec mypod -- cat /etc/os-release
   ```

It should output the contents of the os-release file.

Expected Output:

```
PRETTY_NAME="Debian GNU/Linux 12 (bookworm)"
NAME="Debian GNU/Linux"
VERSION_ID="12"
VERSION="12 (bookworm)"
VERSION_CODENAME=bookworm
ID=debian
HOME_URL="https://www.debian.org/"
SUPPORT_URL="https://www.debian.org/support"
BUG_REPORT_URL="https://bugs.debian.org/"
```

4. Now use ***kubectl exec*** and supply the -i -t flags to spawn a shell session within the container.

   ```
   $ kubectl exec -i -t mypod -- /bin/sh
   ```

   If executed correctly, it should drop you into a new shell session within the nginx container.

   Expected Output:

   ```
   #
   ```

5. Note that most Linux commands may not work as this is a caled down version of linux. You try such commands as ls, pwd

   ```
   # ls
   ```

6. Exit out of the container simply by typing exit. With that the shell process will be terminated and the only running processes within the container should once again be nginx and its worker process.

.

Summary: kubectl exec is an important command to be familiar with when it comes to Pod debugging.
.

kubectl proxy
kubectl proxy enables access to both the Kubernetes API-Server and to resources running within the cluster securely using kubectl. By default it creates a connection to the API-Server that can be accessed at 127.0.0.1:8001 or an alternative port by supplying the ***-p or –port*** flag.

Command

```
kubectl proxy
kubectl proxy --port=<port>
```

Examples

```
$ kubectl proxy
Starting to serve on 127.0.0.1:8001

<from another terminal>
$ curl 127.0.0.1:8001/version
{
  "major": "",
  "minor": "",
  "gitVersion": "v1.9.0",
  "gitCommit": "925c127ec6b946659ad0fd596fa959be43f0cc05",
  "gitTreeState": "clean",
  "buildDate": "2018-01-26T19:04:38Z",
  "goVersion": "go1.9.1",
  "compiler": "gc",
  "platform": "linux/amd64"
}
```

The Kubernetes API-Server has the built in capability to proxy to running services or pods within the cluster. This ability in conjunction with the ***kubectl proxy*** command allows a user to access those services or pods without having to expose them outside of the cluster.

```
http://<proxy_address>/api/v1/namespaces/<namespace>/<services|pod>/<service_name|pod_name>[:port_name]/proxy
```

- **proxy_address** - The local proxy address - 127.0.0.1:8001
- **namespace** - The namespace owning the resources to proxy to.
- **service|pod**- The type of resource you are trying to access, either service or pod.
- **service_name|pod_name**- The name of the service or pod to be accessed.
- **[:port]**- An optional port to proxy to. Will default to the first one exposed.

Example:

```
curl http://127.0.0.1:8001/api/v1/namespaces/default/pods/mypod/proxy/
curl http://127.0.0.1:8001/api/v1/namespaces/default/services/mysvc/proxy/
```

kubectl port-forward

kubectl run web --image=nginx --labels app=web --expose --port 80

To access the service?
sudo kubectl -n default port-forward svc/web 90:80

Curl localhost:90

To access the pod directly;
kubectl -n default port-forward pod/web 8080:80

Curl localhost:8080

LAB: Exploring the Core of Kubernetes

This Lab covers the fundamental building blocks that make up Kubernetes. Understanding what these components are and how they are used is crucial to learning how to use the higher level objects and resources.

- Namespaces
- Pods
- Labels and Selectors
- Cleaning up
- Helpful Resources

.

Namespaces

Namespaces are a logical cluster or environment. They are the primary method of partitioning a cluster or scoping access.

.Exercise: Using Namespaces

Objectives: Learn how to create and switch between Kubernetes Namespaces using kubectl.
NOTE: You may have completed this already in the previous lab.

.

1. List the current namespaces

```
$ kubectl get namespaces
```

2. Create the **dev** namespace

```
$ kubectl create namespace dev
```

3. Create a new context called minidev within the kubernetes cluster as the administrator user, with the namespace set to dev.

```
$ kubectl config set-context minidev --cluster=kubernetes
--user=kubernetes-admin --namespace=dev
```

4. Switch to the newly created context.

```
$ kubectl config use-context minidev
```

.

Summary: Namespaces function as the primary method of providing scoped names, access, and act as an umbrella for group based resource restriction. Creating and switching between them is quick and easy, but learning to use them is essential in the general usage of Kubernetes.

.

Pods

A pod is the atomic unit of Kubernetes. It is the smallest *"unit of work"* or *"management resource"* within the system and is the foundational building block of all Kubernetes Workloads.

Note: These exercises build off the previous Core labs. If you have not done so, complete those before continuing.

Exercise: Creating Pods

Objective: Examine both single and multi-container Pods; including: viewing their attributes through the cli and their exposed Services through the API Server proxy.

1. Create a simple Pod called pod-example using the nginx:stable-alpine image and expose port 80. Use the manifest manifests/pod-example.yaml or the yaml below.

manifests/pod-example.yaml

```
apiVersion: v1
```

```
kind: Pod
metadata:
  name: pod-example
spec:
  containers:
  - name: nginx
    image: nginx:stable-alpine
    ports:
    - containerPort: 80
```

Command

```
$ kubectl create -f manifests/pod-example.yaml
```

2. Use kubectl to describe the Pod and note the available information.

   ```
   $ kubectl describe pod pod-example
   ```

3. Use ***kubectl proxy*** to verify the web server running in the deployed Pod.

Command

```
$ kubectl proxy &
```

URL

```
curl http://127.0.0.1:8001/api/v1/namespaces/default/pods/pod-example/proxy/
```

The default **"Welcome to nginx!"** page should be visible.

4. Using the same steps as above, create a new Pod called multi-container-example using the manifestmanifests/pod-multi-container-example.yaml or create a new one yourself with the below yaml.

manifests/pod-multi-container-example.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: multi-container-example
spec:
  containers:
  - name: nginx
    image: nginx:stable-alpine
```

```
  ports:
  - containerPort: 80
  volumeMounts:
  - name: html
    mountPath: /usr/share/nginx/html
- name: content
  image: alpine:latest
  volumeMounts:
  - name: html
    mountPath: /html
  command: ["/bin/sh", "-c"]
  args:
    - while true; do
        echo $(date)"<br />" >> /html/index.html;
        sleep 5;
      done
volumes:
- name: html
  emptyDir: {}
```

Command

```
$ kubectl create -f manifests/pod-multi-container-example.yaml
```

Note: spec.containers is an array allowing you to use multiple containers within a Pod.

5. Use the proxy to verify the web server running in the deployed Pod.

Command

```
$ kubectl proxy
```

URL

```
curl
http://127.0.0.1:8001/api/v1/namespaces/default/pods/multi-container-example/
proxy/
```

There should be a repeating date-time-stamp.

.

Summary: Becoming familiar with creating and viewing the general aspects of a Pod is an important skill. While it is rare that one would manage Pods directly within Kubernetes, the knowledge of how to view, access and describe them is important and a common first-step in troubleshooting a possible Pod failure.

.

Labels and Selectors

Labels are key-value pairs that are used to identify, describe and group together related sets of objects or resources.

Selectors use labels to filter or select objects and are used throughout Kubernetes.

.

Exercise: Using Labels and Selectors

Objective: Explore the methods of labelling objects in addition to filtering them with both equality and set-based selectors.

.

1. Label the Pod pod-example with app=nginx and environment=dev via kubectl.

   ```
   $ kubectl label pod pod-example app=nginx environment=dev
   ```

2. View the labels with kubectl by passing the –show-labels flag

   ```
   $ kubectl get pods --show-labels
   ```

3. Update the **multi-container example manifest** created previously with the labels app=nginx and environment=prod then apply it via kubectl.

manifests/pod-multi-container-example.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: multi-container-example
  labels:
    app: nginx
    environment: prod
spec:
  containers:
  - name: nginx
    image: nginx:stable-alpine
    ports:
    - containerPort: 80
    volumeMounts:
    - name: html
```

```
    mountPath: /usr/share/nginx/html
- name: content
  image: alpine:latest
  volumeMounts:
  - name: html
    mountPath: /html
  command: ["/bin/sh", "-c"]
  args:
    - while true; do
        date >> /html/index.html;
        sleep 5;
      done
volumes:
- name: html
  emptyDir: {}
```

Command

```
$ kubectl apply -f manifests/pod-multi-container-example.yaml
```

4. View the added labels with *kubectl* by passing the –*show-labels* flag once again.

   ```
   $ kubectl get pods --show-labels
   ```

5. With the objects now labeled, use an equality based selector targeting the ***prod*** environment.

   ```
   $ kubectl get pods --selector environment=prod
   ```

6. Do the same targeting the ***nginx*** app with the short version of the selector flag (-l).

   ```
   $ kubectl get pods -l app=nginx
   ```

7. Use a set-based selector to view all pods where the ***app*** label is ***nginx*** and filter out any that are in the prodenvironment.

   ```
   $ kubectl get pods -l 'app in (nginx), environment notin (prod)'
   ```

.

Summary: Kubernetes makes heavy use of labels and selectors in near every aspect of it. The usage of selectors may seem limited from the cli, but the concept can be extended to when it is used with higher level resources and objects.

Domain 1.3: Understand Multi-Container Pod Design Patterns (e.g., Sidecar, Init Containers, and Others)

In Kubernetes, Pods are the smallest deployable units and can contain one or more containers. When using multi-container Pods, it's essential to understand different design patterns to ensure that the containers work together seamlessly, providing enhanced functionality or solving specific use cases. As a Kubernetes Application Developer (CKAD), one of your tasks is to design and implement efficient multi-container Pods by selecting the appropriate design patterns. In this section of the CKAD training, we will explore several multi-container Pod design patterns, their use cases, and best practices.

1. Introduction to Multi-Container Pods

A Pod in Kubernetes is a collection of one or more containers that are deployed together on the same host. The containers in the same Pod share the same network namespace, meaning they can communicate with each other using localhost. They also share storage volumes, which means they can access persistent data. While Pods often contain a single container, in some cases, it's beneficial to use multiple containers within a single Pod to enhance the functionality of your application. These multiple containers are tightly coupled and share the same lifecycle.

LAB:

In this lab exercise:

- a multi-container pod is created. Containers in a pod share the same volume and localhost.
- access the shared volume of the pods
- access the localhost of the pods

1. Use the following template to create a file named multi-container-pod.yaml.

```
apiVersion: v1
kind: Pod
metadata:
  name: multi-container-pod
spec:
  containers:
  - name: nginx-container
    image: nginx:latest
    command: ['sh', '-c', 'while true; do echo "logging" >> /opt/logs.txt; sleep 1; done']
    volumeMounts:
      - name: data
      mountPath: /opt
    ports:
    - containerPort: 80
  - name: busybox-container
    image: busybox:latest
    command: ["/bin/sh"]
    args: ["-c", "while true; do sleep 3600; done"]
    volumeMounts:
      - name: data
      mountPath: /opt
  volumes:
    - name: data
      emptyDir: {}
```

- Container 1: nginx-container
- Container 2: busybox-container
- Use emptyDir volume which is local
- The volume is mounted on both containers
- The mountPath on both containers can be different

2. Run the following command to create the multi-containers:

```
kubectl create -f multi-container-pod.yaml
```

3. Run the following command to query the deployed pods:

```
kubectl get pod
```

Expected output:

```
NAME                    READY   STATUS      RESTARTS    AGE
multi-container-pod     2/2     Running     0           3s
```

- Notice 2/2 under READY.
- This means that the pod have 2 containers and both of them are running. If one has error, you might see 1/2 under READY.

4. Access the volume in both containers to confirm that they both see the same file in the mounted location

Here we access the nginx-container

```
kubectl exec multi-container-pod -c nginx-container -- ls /opt/
```

Expected output:

```
log.txt
```

- here we are accessing nginx-container to list the /opt directory
- we could also login to the container and issue ls /opt

Here we access the busybox-container

```
kubectl exec multi-container-pod -c busybox-container -- ls /opt/
```

Expected output:

```
log.txt
```

- here we are accessing busybox-container to list the /opt directory
- we could also login to the container and issue ls /opt

1. Access the localhost in both containers to confirm that they both see the same localhost

Here we access the nginx-container

```
kubectl exec multi-container-pod -c nginx-container -- service nginx restart
kubectl exec multi-container-pod -c nginx-container -- curl localhost
```

Expected output:

```
kubectl exec [POD] [COMMAND] is DEPRECATED and will be removed in a future
version. Use kubectl exec [POD] -- [COMMAND] instead.
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>
<p>For online documentation and support please refer to
```

```
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>
<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

- here we are accessing nginx-container to list the /opt directory
- we could also login to the container and issue ls /opt

Here we access the busybox-container

```
kubectl exec -it multi-container-pod -c busybox-container
wget -qO- http://localhost
exit
or
kubectl exec -it multi-container-pod -c busybox-container -- wget -qO-
http://localhost
```

Expected output:

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>
<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>
<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

- ignore the warning sign if any
- here we are accessing nginx webserver on the localhost when inside the busybox-container

In this setup:

- The we created a multi-container pod
- access shared volumes from both
- Accessed localhost from both

9. Lab cleanup

```
kubectl delete -f multi-container.yaml
```

2. Types of Multi-Container Pod Design Patterns

Kubernetes offers several design patterns for structuring multi-container Pods. The most commonly used patterns include:

- Sidecar Pattern
- Init Container Pattern
- Ambassador Pattern
- Adapter Pattern

Let's dive into each of these design patterns in detail. You may read a technical paper here https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/45406.pdf

1. Sidecar Pattern

The Sidecar Pattern involves running a secondary container alongside the main application container to add extra functionality or support. The sidecar container typically runs alongside the main container and extends its capabilities without modifying it.

Use Cases:

1. Logging agents: A sidecar container can be responsible for aggregating logs from the main application container and sending them to a logging system.
2. Proxy servers: A sidecar can run a proxy server (e.g., Envoy or Istio) that handles service discovery, traffic routing, or load balancing for the main application.
3. Data synchronization: A sidecar container could sync data from the main container to a remote service or database.

Key Features:

1. The sidecar container is independent but still tightly coupled to the main container, meaning it can be scaled or deployed together with the main application.
2. The sidecar container shares the same network and storage volumes as the main container.

Example: A typical example of the sidecar pattern is running a log shipping agent like Fluentd in a sidecar container alongside an application container. Here's a basic example:

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-with-sidecar
spec:
  containers:
    - name: main-app
      image: nginx:latest
      volumeMounts:
        - name: logs
          mountPath: /var/log/nginx
      ports:
        - containerPort: 8080
    - name: fluentd
      image: fluent/fluentd:edge-debian
      volumeMounts:
        - name: logs
          mountPath: /var/log/nginx
  volumes:
    - name: logs
      hostPath:
        path: /var/log
```

In this example:

1. The main container (nginx:latest) runs the application.
2. The sidecar container (fluentd) handles logging by reading logs from /var/log and sending them to a logging service.

2. Init Containers Pattern

Init Containers are specialized containers that run to completion before the main containers in the Pod start. These containers are useful for performing initialization tasks such as setting up environment variables, downloading resources, or waiting for certain conditions to be met before the main application starts.

Use Cases:

1. Initialization tasks: Running setup scripts or database migrations before the application container starts.
2. Pre-checks or waits: Performing checks (e.g., checking if a database is available) before allowing the main container to run.
3. Pre-populating data: Downloading resources, configuration files, or mounting volumes before the main application runs.

Key Features:

1. Init containers run sequentially, one after another, and each must complete successfully before the next container starts.
2. They can access the same volumes as the main containers and have their own isolated environment.

LAB:

In this lab exercise, an init container waits for a database service to be available before starting the main application:

1. Use the following template to create a file named myapp-with-init.yaml.

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-with-init
spec:
  initContainers:
    - name: wait-for-db
      image: busybox
      command: ["sh", "-c", "until nc -z mysql 3306; do sleep 1; done;"]
  containers:
    - name: main-app
      image: nginx:latest
      ports:
        - containerPort: 8080
```

- initContainer: wait-for-db
- Main container : main-app

2. Run the following command to create a pod with initcontainers:

```
kubectl create -f myapp-with-init.yaml
```

3. Run the following command to query the deployed InitContainer pod:

```
kubectl get pod
```

Expected output:

```
kubectl get pod
NAME                READY   STATUS        RESTARTS      AGE
myapp-with-init         0/1    Init:0/1        0            5s
```

- Notice the status of Init:0/1.
- This means that the initContainer pod is not yet completed because it is waiting for the condition to be met before it will continue.
- The condition it is waiting is that mysql database is accessible on port 3306

4. Use the following template to create a file named mydb.yaml.

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mysql
  labels:
    app: wordpress
    component: mysql
spec:
  replicas: 1
  selector:
    matchLabels:
      app: wordpress
      component: mysql
  serviceName: mysql
  template:
    metadata:
      labels:
        app: wordpress
        component: mysql
    spec:
      containers:
      - image: mysql:oracle
        name: mysql
        env:
        - name: MYSQL_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysql
              key: password
        ports:
        - containerPort: 3306
          name: mysql
        volumeMounts:
        - name: mysql-data
        mountPath: /var/lib/mysql
  volumeClaimTemplates:
  - metadata:
      name: mysql-data
    spec:
      accessModes: [ "ReadWriteOnce" ]
      #storageClassName: standard
      resources:
        requests:
          storage: 1Gi
---
apiVersion: v1
kind: Service
metadata:
  name: mysql
  labels:
    app: wordpress
    component: mysql
spec:
  ports:
    - port: 3306
  selector:
    app: wordpress
    component: mysql
  clusterIP:
---
apiVersion: v1
kind: Secret
metadata:
  name: mysql
  labels:
    app: wordpress
    component: mysql
type: Opaque
data:
  password: c3VwZXJzZWNyZXRwYXNzd29yZA==
  # supersecretpassword%
```

5. Run the following command to create a pod with mysql running on port 3306:

```
kubectl create -f mydb.yaml
```

6. Run the following command to query the deployed pods:

```
kubectl get pod
```

Expected output:

```
NAME                READY   STATUS         RESTARTS       AGE
myapp-with-init         0/1     Init:0/1        0             34s
mysql-0              1/1     Running         0             22s
```

7. Check the pods again

```
kubectl get pods
```

Expected output:

```
NAME                READY   STATUS         RESTARTS       AGE
myapp-with-init         1/1     Running         0             77s
mysql-0              1/1     Running         0             65s
```

In this setup:

- The init container (wait-for-db) runs and waits until the database is accessible on port 3306.
- Once the init container successfully finishes, the main container (myapp:latest) starts.

8. Clean up

```
kubectl delete -f mydb.yaml
```

9. Lab complete

3. Ambassador Pattern

The Ambassador Pattern involves using a sidecar container that acts as a proxy for the main application, often to manage traffic between the application and external systems. The ambassador can handle requests, such as routing, monitoring, or security tasks, between the main application and other services.

Use Cases:

1. Service discovery: Proxy containers can help the main application discover services within the Kubernetes cluster.
2. Traffic routing: An ambassador can route traffic between services based on rules or configuration, e.g., Istio or Envoy.
3. Security/Authentication: A proxy can handle tasks like JWT authentication or SSL/TLS encryption for the application.

Key Features:

The ambassador sidecar container routes traffic to and from the main application, often performing transformations or enhancements.

LAB:

In this lab exercise:

- a multi-container pod is created. Containers in a pod share the same volume and localhost.
- access the shared volume of the pods
- access the localhost of the pods

1. Use the following template to create a file named pod-with-ambassador-pattern.yaml.

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-ambassador-pattern
  labels:
    app: main
spec:
  containers:
  - name: main-container
    image: httpd
    ports:
    - containerPort: 80
  - name: proxy-container
    image: nginx:latest
    ports:
    - containerPort: 8080
    env:
    - name: MAIN_HOST
      value: "localhost"
    - name: MAIN_PORT
      value: "80"
    volumeMounts:
    - name: nginx-config
      mountPath: /etc/nginx/nginx.conf
      subPath: nginx.conf
  volumes:
  - name: nginx-config
    configMap:
      name: nginx-config
---
apiVersion: v1
```

```
kind: ConfigMap
metadata:
  name: nginx-config
data:
  nginx.conf: |
    events {}
    http {
      server {
        listen 8080;
        location / {
          proxy_pass http://localhost:80;
        }
      }
    }
```

- Container 1: main-container
- Container 2: proxy-container
- Use configMap volume to map nginx configuration file
- The configMap volume is mounted on the proxy container at /etc/nginx/nginx.conf
- The above mount will replace the default nginx configuration

1. Run the following command to create the multi-containers:

```
kubectl create -f pod-with-ambassador-pattern.yaml
```

2. Run the following command to query the deployed pods:

```
kubectl get pod
```

Expected output:

```
NAME                          READY   STATUS    RESTARTS   AGE
pod-with-ambassador-pattern   2/2     Running   0          4s
```

- Notice 2/2 under READY.
- This means that the pod have 2 containers and both of them are running. If one has error, you might see 1/2 under READY.

3. Access the proxy/ambassador pod in both containers to confirm that they both point to the backend application.

Here we access the main container

```
kubectl exec -it ambassador-pattern-pod -- /bin/bash
apt update
apt install curl
curl localhost
curl localhost:8080
```

Expected output:

```
<html><body><h1>It works!</h1></body></html>
```

Here we access the proxy-container

```
kubectl exec -it ambassador-pattern-pod -c proxy-container -- /bin/bash
curl localhost
curl localhost:8080
```

Expected output:

```
<html><body><h1>It works!</h1></body></html>
```

Conclusion: In this setup:

1. The main app is the core application.
2. The nginx container acts as an ambassador, managing incoming requests and routing them to the main app.
3. A config map is used to pass the proxy configuration to the nginx server, to replace the default configuration
4. Lab cleanup kubectl delete -f pod-with-ambassador-pattern.yaml

4. Adapter Pattern

The Adapter Pattern involves using a sidecar container to adapt an existing service to work with your application. The adapter translates or modifies requests to meet the expectations of the main container, ensuring compatibility without modifying the application.

Use Cases:

1. Protocol translation: Adapting a service to work with your application if it expects a different communication protocol or API.
2. Data transformation: Adapting data formats (e.g., JSON to XML) for use by your application.

Key Features:

The adapter container listens for incoming requests, transforms them if needed, and passes them to the main container.

LAB:

In this lab exercise:

- Elasticsearch does not natively export Prometheus metrics

- We add a Prometheus exporter to help convert Elasticsearch metrics into prometheus' expected format. Details here https://github.com/prometheus-community/elasticsearch_exporter
- We did not alter the elasticsearch code or image but we achieved our purpose with Adapter patter
- We created a clear separation between the application and the platform

1. Use the following template to create a file named pod-with-adapter-pattern.yaml.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deployment-with-adapter-pattern
spec:
  selector:
    matchLabels:
      app.kubernetes.io/name: elasticsearch
  template:
    metadata:
      labels:
        app.kubernetes.io/name: elasticsearch
    spec:
      containers:
        - name: elasticsearch
          image: elasticsearch:7.9.3
          env:
            - name: discovery.type
              value: single-node
          ports:
            - name: http
              containerPort: 9200
        - name: prometheus-exporter
          image: justwatch/elasticsearch_exporter:1.1.0
          args:
            - '--es.uri=http://localhost:9200'
          ports:
            - name: http-prometheus
              containerPort: 9114
---
apiVersion: v1
kind: Service
metadata:
  name: elasticsearch
spec:
  selector:
    app.kubernetes.io/name: elasticsearch
```

```
ports:
  - name: http
    port: 9200
    targetPort: http
  - name: http-prometheus
    port: 9114
    targetPort: http-prometheus
```

- Container 1: elasticsearch
- Container 2: prometheus-exporter
- Prometheus is exposed on http://localhost:9114

1. Run the following command to create the multi-containers:

```
kubectl create -f pod-with-adapter-pattern.yaml
```

2. Run the following command to query the deployed pods:

```
kubectl get pod
```

Expected output:

```
NAME                          READY   STATUS              RESTARTS      AGE
elasticsearch-6467ffc4b5-snnnc   2/2     Running                0             5s
```

- Notice 2/2 under READY.
- This means that the pod have 2 containers and both of them are running. If one has error, you might see 1/2 under READY.

We deployed this as a Deployment so you can also see the deployment:

```
kubectl get deploy
```

Expected output:

```
NAME                          READY   UP-TO-DATE   AVAILABLE   AGE
deployment-with-adapter-pattern   1/1     1            1           53m
```

3. Access the the exported Elasticsearch metrics.

Here we access the main container

```
kubectl exec -it elasticsearch-6467ffc4b5-snnnc -- /bin/bash
curl localhost:9114/metrics | grep head
```

Expected output will be similar to below:

```
elasticsearch_breakers_overhead{breaker="accounting",cluster="docker-cluster",es_
client_node="true",es_data_node="true",es_ingest_node="true",es_master_node="tru
e",host="10.1.4.138",name="elasticsearch-6467ffc4b5-snnnc"} 1
elasticsearch_breakers_overhead{breaker="fielddata",cluster="docker-cluster",es_cli
ent_node="true",es_data_node="true",es_ingest_node="true",es_master_node="true",h
ost="10.1.4.138",name="elasticsearch-6467ffc4b5-snnnc"} 1.03
elasticsearch_breakers_overhead{breaker="in_flight_requests",cluster="docker-clust
er",es_client_node="true",es_data_node="true",es_ingest_node="true",es_master_nod
e="true",host="10.1.4.138",name="elasticsearch-6467ffc4b5-snnnc"} 2
elasticsearch_breakers_overhead{breaker="model_inference",cluster="docker-clust
er",es_client_node="true",es_data_node="true",es_ingest_node="true",es_master_nod
e="true",host="10.1.4.138",name="elasticsearch-6467ffc4b5-snnnc"} 1
```

You can also do the following without logging into the pod:

```
kubectl exec -it elasticsearch-6467ffc4b5-snnnc -- curl localhost:9114/metrics |
grep head
```

4. Lab cleanup kubectl delete -f pod-with-adapter-pattern.yaml

Conclusion:

In this setup:
1. The main app is the Elasticsearch.
2. The exporter container acts as an adapter, converting elasticsearch output format to Promethues format

5. Conclusion: Choosing the Right Design Pattern

Choosing the appropriate multi-container Pod design pattern is crucial for building efficient, scalable, and maintainable applications in Kubernetes. As a Kubernetes Application Developer (CKAD), here's a summary of when to use each design pattern:
1. Sidecar Pattern: Use when you need additional functionality (e.g., logging, proxy, monitoring) alongside your main application.
2. Init Containers: Use for initialization tasks that need to be completed before your main application starts (e.g., waiting for services, pre-populating data).
3. Ambassador Pattern: Use when you need a proxy to handle traffic, service discovery, or security for your application.
4. Adapter Pattern: Use to transform requests or data between services or protocols to ensure compatibility.

By understanding these patterns and their use cases, you'll be well-equipped to design and deploy multi-container Pods that work seamlessly in Kubernetes