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 already be 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 plan of the day to day learning objectives, though this could vary depending on the audience.

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

Conventions used in the training manual:

- Green color is for things you need to type in the terminal or a yaml file that you may need to create using an editor
- Blue is what you need to see as a result of the green command executed
- <> indicates that you need to replace/substitute what is inside the angle brackket
- **Black color** is typically instructions describing what you are to do in the green color

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 file system 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 file system, it creates a new layer, leaving the image layers intact. This reduces duplication of data and ensures the image stays unchanged.

Docker Workflow

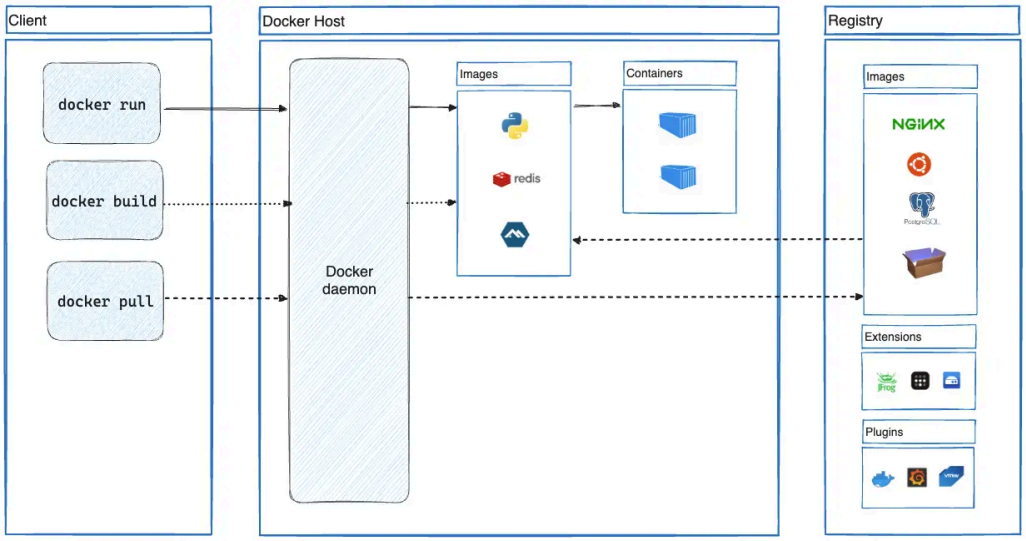
1. **Create a Dockerfile**: Write a Dockerfile that specifies the environment and instructions to build an image.
2. **Build the Image**: Use the docker build command to create an image from the Dockerfile.
3. **Run a Container**: Use the docker run command to create and start a container from the image.
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. The more the better.

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. The more the better.

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 below. Don't forget the dot (.) at the end of the line.

   ```
   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. Its free, you can create one at hub.docker.com
4. **Docker CLI access**: You should be able to run docker commands in your terminal.
5. Python3: Make sure Python is installed, or install it as follows:

```
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  # or 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!"}
```

Now that the code is working as expected, you can dockerize it.

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:

To find the Container ID:
docker ps

To stop the Container:

docker stop <container-id>

To delete/remove the container:

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:

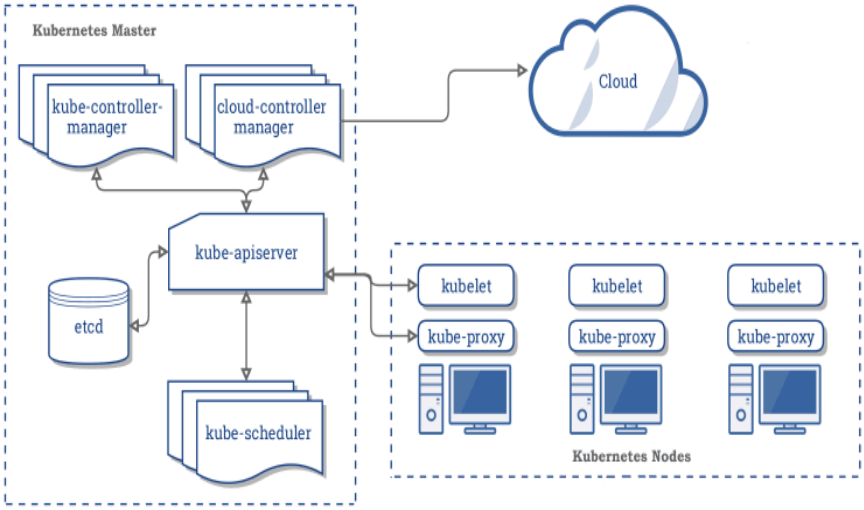
docker rmi <your-username>/fastapi-docker-demo:latest

Conclusion

In this lab, take note of the ports used, which 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: The 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 the 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 and kubens can be quite helpfu in managing context and namespacesl. Kubectx allows a user to quickly switch between contexts and namespaces without having to use the full kubectl config use-context command.Kubens

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://b86aaf4ab9f31c11ebac3ee70d52403c33753511dde9263ca09b8f7a1687d589
    Image:          nginx
    Image ID:
docker-pullable://nginx@sha256:eb3320e2f9ca409b7c0aa71aea3cf7ce7d018f03a372564dbdb023646958770b
    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

Explain 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
```

Expected Output:

Starting to serve on 127.0.0.1:8001

Then 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

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: **Y**ou 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 pod-example.yaml or the yaml below.

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 manifest pod-multi-container-example.yaml or create a new one yourself with the below yaml.

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.

pod-multi-container-labels.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. Create a file called myapp-with-sidecar.yaml and paste the following content.

```
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="true",host="10.1.4.138",name="elasticsearch-6467ffc4b5-snnnc"} 1
elasticsearch_breakers_overhead{breaker="fielddata",cluster="docker-cluster",es_client_node="true",es_data_node="true",es_ingest_node="true",es_master_node="true",host="10.1.4.138",name="elasticsearch-6467ffc4b5-snnnc"} 1.03
elasticsearch_breakers_overhead{breaker="in_flight_requests",cluster="docker-cluster",es_client_node="true",es_data_node="true",es_ingest_node="true",es_master_node="true",host="10.1.4.138",name="elasticsearch-6467ffc4b5-snnnc"} 2
elasticsearch_breakers_overhead{breaker="model_inference",cluster="docker-cluster",es_client_node="true",es_data_node="true",es_ingest_node="true",es_master_node="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.

Certified Kubernetes Application Developer (CKAD)

By Damian Igbe, PhD

Day 2

Domain 1.2: Choose and Use the Right Workload Resource (Deployment, DaemonSet, CronJob, etc.)

In Kubernetes, workloads are the resources responsible for running your applications. They define the application's deployment and execution patterns, scaling, and availability. As a Kubernetes Application Developer (CKAD), one of your key responsibilities is to choose the appropriate workload resource for your application based on your requirements.
In this section of the CKAD training, we will explore the different types of workload resources available in Kubernetes and how to use them effectively.

1. What are Workload Resources in Kubernetes?

A workload resource in Kubernetes is a resource type that helps you run your application containers in a Kubernetes cluster. It defines the desired state for your application (e.g., how many replicas of a container to run, whether it should be scheduled on specific nodes, etc.) and allows Kubernetes to manage the lifecycle of the application according to that state. There are several types of workload resources in Kubernetes, and each has specific use cases. Some of the most common workload resources are:

- Deployment
- DaemonSet
- StatefulSet
- ReplicaSet
- Job
- CronJob

2. Deployments: Managing Stateless Applications

A Deployment is the most common workload resource used in Kubernetes for stateless applications. It provides declarative updates to applications, meaning you can describe the desired state of your application, and Kubernetes will automatically manage the deployment to match that state.

Use Cases:

1. Stateless applications where replicas of a pod can be spread across nodes for load balancing.
2. Rolling updates and rollbacks for zero-downtime deployment.

Key Features:

1. Scaling: Automatically adjusts the number of replicas as required.
2. Rolling Updates: Kubernetes automatically rolls out updates to your application without downtime.

3. Rollback: If something goes wrong with the new deployment, you can easily roll back to the previous version.

Example: To create a simple Deployment for a web application, you can use the following YAML. Name teh file deploy-example.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-web-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-web-app
  template:
    metadata:
      labels:
        app: my-web-app
    spec:
      containers:
        - name: my-web-container
          image: nginx
          ports:
            - containerPort: 80
```

This YAML file describes a deployment with three replicas of the my-web-app container, ensuring high availability.

Use Kubernetes Primitives to Implement Common Deployment Strategies (e.g., Blue/Green or Canary)

In Kubernetes, deploying applications efficiently and safely is critical to maintaining high availability and minimizing downtime. One of the primary ways to achieve this is by using deployment strategies such as Blue/Green and Canary deployments. These strategies allow you to roll out updates incrementally, minimizing risk and enabling better control over the release process.
As a Kubernetes Application Developer (CKAD), you must understand how to implement these strategies using Kubernetes primitives such as Deployments, Services, Labels, and Selectors.

1. Blue/Green Deployment Strategy

Blue/Green Deployment is a deployment strategy where two environments (the "Blue" and "Green" environments) are maintained. The "Blue" environment is the live, production version of your application, while the "Green" environment contains the updated version of the application.

How it Works:

1. Initially, the "Blue" environment (the live version) serves all the traffic.
2. The "Green" environment is deployed and tested in parallel with the "Blue" environment.
3. Once the "Green" environment is tested and verified, the traffic is switched to the "Green" environment, making it the new production.
4. The "Blue" environment can be kept as a fallback in case of issues with the new deployment.

Advantages:

1. Minimized risk: If there is an issue with the "Green" deployment, you can quickly revert to the "Blue" version.
2. Zero downtime: With traffic routed between two separate environments, you can achieve a seamless switch.

Implementation in Kubernetes:

To implement Blue/Green in Kubernetes, you can use two separate Deployments (one for Blue, one for Green) and a Service that directs traffic to the active environment.

Step 1: Define Blue and Green Deployments

Blue Deployment (Current live version). Create blue.yaml file. Replace <my-app:blue> with any image of choice like nginx:latest

```
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
  name: app-blue
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
      version: blue
  template:
    metadata:
      labels:
        app: my-app
        version: blue
    spec:
      containers:
        - name: my-app
          image: <my-app:blue>
          ports:
            - containerPort: 80
```

Green Deployment (New version to be deployed). Create green.yaml file. Replace <my-app:green> with any image of choice like httpd:latest

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-green
spec:
  replicas: 0  # Initially set to 0 replicas until switch
  selector:
    matchLabels:
      app: my-app
      version: green
  template:
    metadata:
      labels:
        app: my-app
        version: green
    spec:
      containers:
        - name: my-app
          image: <my-app:green>
          ports:
            - containerPort: 80
```

Step 2: Define a Service to Route Traffic

```
apiVersion: v1
kind: Service
metadata:
  name: my-app-service
spec:
  selector:
    app: my-app
    version: blue  # Initially routes to the blue version
  ports:
    - port: 80
      targetPort: 80
```

Step 3: Switch Traffic to the Green Deployment

Once you have tested the "Green" deployment, you can modify the Service to point to the new version:

```
apiVersion: v1
kind: Service
metadata:
  name: my-app-service
spec:
  selector:
    app: my-app
    version: green  # Switch traffic to green version
  ports:
    - port: 80
      targetPort: 80
```

Step 4: Scale Down/Remove Blue Deployment (Optional)

After the traffic is fully switched to the "Green" environment, you can scale down or delete the "Blue"
Deployment:

```
kubectl scale deployment app-blue --replicas=0
```

2. Canary Deployment Strategy

Canary Deployment is a strategy where a new version of the application is deployed to a small subset of users before being rolled out to the entire user base. This allows you to monitor the performance of the new version in a production environment with minimal risk.

How it Works:

1. A small portion of traffic is routed to the new version (the "canary").
2. The rest of the traffic continues to go to the stable version.
3. If the new version performs well (i.e., no critical issues are found), more traffic is gradually shifted towards it until it is fully deployed.
4. If issues are detected, the new version can be rolled back or adjusted.

Advantages:

- Incremental rollout: Allows testing in a real-world environment with minimal risk.
- Easy rollback: If something goes wrong, the rollout can be stopped, and the traffic can be directed back to the stable version.

Implementation in Kubernetes:

To implement Canary deployment in Kubernetes, you typically use a Deployment and incrementally change the number of replicas for the new version. This is managed by adjusting the replica count for both the stable and canary versions.

Step 1: Define the Stable Version (e.g., stable) and Canary Version (e.g., canary) Deployments. Create file stable.yaml. Replace <my-app:stable> with an image of choice like nginx image.

Stable Version Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-stable
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
      version: stable
  template:
    metadata:
      labels:
        app: my-app
        version: stable
    spec:
      containers:
        - name: my-app
          image: <my-app:stable>
          ports:
            - containerPort: 80
```

Canary Version Deployment

Replace <my-app:canary> with an image of choice like the nginx image.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-canary
spec:
  replicas: 1  # Start with a small number of replicas for the canary
  selector:
    matchLabels:
      app: my-app
      version: canary
  template:
    metadata:
      labels:
        app: my-app
        version: canary
    spec:
      containers:
        - name: my-app
          image: <my-app:canary>
          ports:
            - containerPort: 80
```

Step 2: Define a Service

The Service will initially route traffic to both the stable and canary versions. As the rollout progresses, you can gradually change the traffic distribution. Create service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: my-app-service
spec:
  selector:
    app: my-app
  ports:
    - port: 80
      targetPort: 80
```

Step 3: Gradually Increase the Canary Replicas

Initially, the canary version will have a small number of replicas (e.g., 1 replica). Over time, you can gradually scale up the canary deployment and scale down the stable version to increase traffic to the new version:

```
kubectl scale deployment app-canary --replicas=3  # Increase canary replicas

kubectl scale deployment app-stable --replicas=2  # Scale down stable version
```

Step 4: Monitor and Adjust

While the canary deployment is running, monitor its performance. If it's working as expected, continue scaling it up. If any issues arise, you can easily scale it back down or roll it back to the stable version.

Step 5: Complete the Deployment

Once the canary version is fully tested, you can scale it to the desired number of replicas, ensuring that all traffic is now routed to the new version.

3. Comparison of Blue/Green and Canary Deployment

Feature	Blue/Green Deployment	Canary Deployment
Traffic Management	Entire traffic switches from Blue to Green once the new version is ready.	Gradually shifts a small portion of traffic to the new version.
Rollback	Rollback to the Blue version is instantaneous.	Rollback can be done by scaling down the canary replicas.
Risk Level	Low risk once the Green environment is validated.	Low risk with gradual exposure of the new version.
Deployment Complexity	Simpler, but requires managing two complete environments.	More complex, requires monitoring and gradual rollout.
Use Case	Ideal for major version upgrades with minimal downtime.	Ideal for testing a new version with a subset of users.

4. Conclusion

As a Kubernetes Application Developer (CKAD), understanding how to implement Blue/Green and Canary deployments will empower you to release applications safely and with minimal risk. These strategies help you manage application updates in a controlled manner, ensuring high availability and reducing the impact of issues that may arise with new versions.

Both strategies use Kubernetes primitives like Deployments, Services, Labels, and Selectors, allowing for flexibility in managing how updates are rolled out across your cluster.

Blue/Green deployments are excellent for switching between major versions with minimal downtime. Canary deployments are useful for incrementally rolling out updates and monitoring their impact on production environments. By mastering these strategies, you can ensure more resilient, reliable, and efficient application rollouts in Kubernetes.

Understand How to Perform Rolling Updates with Deployments

1. Understanding Rolling Updates

Rolling updates allow you to update an application without downtime by gradually replacing old pods with new ones. This process ensures that some instances of the application remain available while new versions are deployed.

How Rolling Updates Work:

Step 1: When you update the Deployment (e.g., change the Docker image), Kubernetes will create new pods with the updated configuration.

Step 2: It will slowly scale down the old pods and scale up the new ones, maintaining the desired number of pods running throughout the process.

Step 3: The new version of the pods will only be rolled out if they pass health checks, preventing the deployment of a failing version.

Step 4: If necessary, the update can be rolled back to a previous stable version.

Advantages of Rolling Updates:

1. Zero Downtime: Rolling updates help ensure that at least some replicas of the application are always running, preventing downtime during updates.
2. Safe Gradual Rollout: If issues occur with the new version, you can pause or roll back the update.
3. Version Control: You can easily revert to a previous stable version if necessary.

2. Creating a Deployment

To use a Deployment in Kubernetes, you create a YAML manifest that defines the deployment's configuration, including the Docker image, number of replicas, and other parameters.

Example of a simple Deployment manifest for an application. Create my-app-deployment-1.yaml. Replace <my-app:v1> image with any of your choice like nginx:latest.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app-deployment
spec:
  replicas: 3
```

```
 selector:
  matchLabels:
   app: my-app
 template:
  metadata:
   labels:
    app: my-app
  spec:
   containers:
    - name: my-app-container
      image: <my-app:v1>  # Version 1 of the app
      ports:
       - containerPort: 80
```

In this example:

1. The deployment is named my-app-deployment.
2. Three replicas are defined, meaning Kubernetes will maintain three pods running at all times.
3. The image: my-app:v1 defines the version of the Docker image being used for the application.

To apply the update, run:

```
kubectl apply -f my-app-deployment-1.yaml
```

```
kubectl get deployments
```

3. Performing a Rolling Update

Now that we have a deployment with nginx:latest image, we can perform a rolling update to another image like nginx:stable. Kubernetes automatically performs a rolling update when you change the Deployment's specification, such as updating the image version. The update can be done via the command line (kubectl) or by modifying the Deployment YAML file.

Step-by-Step Guide to Perform a Rolling Update:

Open the previous file and update the Deployment with a New Docker Image: You can modify the Deployment by changing the Docker image tag (e.g., from v1 to v2 or from :lates to :stable in our own case of using nginx). The rolling update will be initiated when the Deployment is updated.

Example: You can use the above file my-app-deployment-2.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: my-app-deployment
spec:
```

```
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: my-app-container
          image: nginx:latest  # Updated to version 2 of the app
          ports:
            - containerPort: 80
```

To apply the update, run:

```
kubectl apply -f my-app-deployment-2.yaml
```

Alternatively, you can update the image directly using kubectl set image:

```
kubectl set image deployment/my-app-deployment my-app-container=nginx:stable
```

Kubernetes Starts the Rolling Update:

Kubernetes will replace the old pods (running v1) with new pods (running v2) one by one.

- It ensures that the desired number of replicas (3 in this case) are running throughout the update process.
- It monitors the health of the new pods (via liveness and readiness probes) before scaling down the old ones.

Verify the Rolling Update: You can track the progress of the rolling update with the following command:

```
kubectl rollout status deployment/my-app-deployment
```

This will show the status of the rolling update, such as "deployment rolling update complete."

Rollback if Necessary: If the rolling update encounters issues, Kubernetes can roll back to the previous stable version of the deployment. This can be done with the following command:

```
kubectl rollout undo deployment/my-app-deployment
```

This command will restore the previous version of the deployment and revert the changes.

4. Customizing Rolling Update Behavior

Kubernetes provides several options for customizing the behavior of rolling updates. You can define how quickly the update proceeds, how many pods can be unavailable at a time, and other parameters. These options are specified in the Deployment spec under the rollingUpdate strategy.

Example of Customizing Rolling Update Strategy:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  strategy:
    type: RollingUpdate
    rollingUpdate:
       maxSurge: 1        # Maximum number of pods that can be created above the desired number of pods
       maxUnavailable: 1  # Maximum number of pods that can be unavailable during the update
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: my-app-container
          image: nginx:latest  # Updated image version
          ports:
            - containerPort: 80
```

In this example:

1. maxSurge: 1 allows one additional pod to be created during the update, ensuring that the overall pod count can temporarily exceed the desired replicas.
2. maxUnavailable: 1 ensures that no more than one pod can be unavailable at any time during the update.

5. Troubleshooting Rolling Updates

Occasionally, rolling updates can encounter issues, such as pods failing health checks or misconfigured resources. Here are some common troubleshooting steps:

Check Deployment Status: This command provides information about the current state of the update, whether it's progressing or has stalled.

```
kubectl rollout status deployment/my-app-deployment
```

Check Pod Logs. If the new pods aren't starting correctly, check their logs for errors.

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

Revert to the Previous Version: to roll back to the previous stable version use

```
kubectl rollout undo
```

6. Conclusion

As a Kubernetes Application Developer (CKAD), understanding how to configure and manage Deployments and perform Rolling Updates is essential to deploying applications safely and efficiently. Rolling updates allow you to update applications with minimal disruption and no downtime, which is crucial for production-grade environments.

1. Deployments are used to manage the lifecycle of your applications and ensure that the desired state is always maintained.
2. Rolling updates enable you to gradually replace old versions of an application with new ones, while ensuring that the service remains available.
3. Customizing rolling update parameters like maxSurge and maxUnavailable helps fine-tune the update process for your specific needs.

By mastering these concepts, you'll be able to implement seamless and safe application updates in your Kubernetes clusters.

2. Managing Stateful Applications with StatefulSets

A StatefulSet is used for stateful applications that require unique network identifiers, stable storage, and ordered deployment. It is ideal for applications like databases or distributed file systems that need persistent state across pod restarts.
Use Cases:

- Stateful applications such as databases (e.g., MySQL, MongoDB), message queues (e.g., Kafka), or distributed file systems (e.g., GlusterFS).
- Applications that require stable, persistent storage and network identity.

  Key Features:

  Stable, unique pod names: Each pod in a StatefulSet gets a unique, stable name (e.g., my-app-0, my-app-1).

  1. Persistent storage: Allows each pod to be associated with a persistent volume claim (PVC).
  2. Ordered deployment and scaling: Pods are created and terminated in a specific order.

     Example: A StatefulSet for running a stateful web application:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: my-db
spec:
  serviceName: "my-db"
  replicas: 3
  selector:
    matchLabels:
      app: my-db
  template:
    metadata:
      labels:
        app: my-db
    spec:
      containers:
        - name: my-db-container
          image: mysql:5.7
          volumeMounts:
            - name: db-data
              mountPath: /data/db
  volumeClaimTemplates:
    - metadata:
        name: db-data
      spec:
        accessModes: ["ReadWriteOnce"]
        resources:
          requests:
            storage: 1Gi
```

This example ensures that each pod in the StatefulSet has its own persistent storage for database data.

LAB walkthrough: Create a StatefulSet

PVCs and PVs can be automatically created based on VolumeClaimTemplates.

Note volumeClaimTemplates represents a type of template that the system uses to create PVCs. The number of PVCs equals the number of replicas that are deployed for the StatefulSet application. The configurations of these PVCs are the same except for the PVC names.

1. Use the following template to create a file named statefulset.yaml.

Deploy a Service and a StatefulSet, and provision two pods for the StatefulSet.

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
```

```
    app: nginx
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: nginx
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  selector:
    matchLabels:
      app: nginx
  serviceName: "nginx"
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: disk-ssd
          mountPath: /data
  volumeClaimTemplates:
  - metadata:
      name: disk-ssd
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: "standard"
      resources:
        requests:
          storage: 2Gi
```

- replicas: the parameter is set to 2 in this example. This indicates that two pods are created.
- mountPath: the path where you want to mount the disk in the container.

- accessModes: the access mode of the StatefulSet.
- storageClassName: the parameter is set to standard in this example. This indicates that a standard storage class was created and used.
- storage: specifies the storage capacity that is required by the application.

2. Run the following command to create a StatefulSet:

   kubectl create -f statefulset.yaml

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

   kubectl get pod

   Expected output:

```
NAME    READY   STATUS    RESTARTS   AGE
web-0   1/1     Running   0          6m
web-1   1/1     Running   0          6m
```

4. Run the following command to query the PVCs:

   kubectl get pvc

   Expected output:

```
NAME             STATUS   VOLUME                    CAPACITY   ACCESS MODES   STORAGECLASS      AGE
disk-ssd-web-0   Bound    d-2zegw7et6xcg6nbojuoo    2Gi        RWO            alicloud-disk-ssd   7m
disk-ssd-web-1   Bound    d-2zefbrqggvkd10xb523h    2Gi        RWO            alicloud-disk-ssd   6m
```

Verify that the PVCs are scaled out together with the StatefulSet application

1. Run the following command to scale out the StatefulSet application to three pods:

   kubectl scale sts web --replicas=3

   Expected output:

```
statefulset.apps/web scaled
```

2. Run the following command to view the pods after the StatefulSet application is scaled out:

   kubectl get pod

   Expected output:

```
NAME
```

```
web-0    1/1   Running   0    34m
web-1    1/1   Running   0    33m
web-2    1/1   Running   0    26m
```

3. Run the following command to view the PVCs after the StatefulSet application is scaled out:

```
kubectl get pvc
```

Expected output:

```
NAME          STATUS   VOLUME                  CAPACITY   ACCESS MODES   STORAGECLASS   AGE
disk-ssd-web-0   Bound   d-2zegw7et6xcg6nbojuoo   2Gi        RWO            standard       35m
disk-ssd-web-1   Bound   d-2zefbrqggvkd10xb523h   2Gi        RWO            standard       34m
disk-ssd-web-2   Bound   d-2ze4jx1zymn4ngj3pic2   2Gi        RWO            standard       27m
```

The output indicates that three PVCs are provisioned for the StatefulSet application after the StatefulSet application is scaled to three pods.

Verify that the PVCs remain unchanged after the StatefulSet application is scaled in.

1. Run the following command to scale the StatefulSet application to two pods:

```
kubectl scale sts web --replicas=2
```

Expected output:
statefulset.apps/web scaled

2. Run the following command to view the pods after the StatefulSet application is scaled in:

```
kubectl get pod
```

Expected output:

```
NAME    READY   STATUS    RESTARTS   AGE
web-0   1/1     Running   0          38m
web-1   1/1     Running   0          38m
```

Only two pods are deployed for the StatefulSet application.

3. Run the following command to view the PVCs after the StatefulSet application is scaled in:

```
kubectl get pvc
```

Expected output:

```
NAME           STATUS   VOLUME                  CAPACITY   ACCESS MODES   STORAGECLASS   AGE
disk-ssd-web-0 Bound    d-2zegw7et6xc96nbojuoo  2Gi        RWO            standard       39m
disk-ssd-web-1 Bound    d-2zefbrqggvkd10xb523h  2Gi        RWO            standard       39m
disk-ssd-web-2 Bound    d-2ze4jx1zymn4n9j3pic2  2Gi        RWO            standard       39m
```

After the StatefulSet application is scaled to two pods, the StatefulSet application still has three PVCs. This indicates that the PVCs are not scaled in together with the StatefulSet application.

Verify that the PVCs remain unchanged when the StatefulSet application is scaled out again

When the StatefulSet is scaled out again, verify that the PVCs remain unchanged.

1. Run the following command to scale out the StatefulSet application to three pods:

   ```
   kubectl scale sts web --replicas=3
   ```

   Expected output:

   ```
   statefulset.apps/web scaled
   ```

2. Run the following command to view the pods after the StatefulSet application is scaled out:

   ```
   kubectl get pod
   ```

   Expected output:

   ```
   NAME    READY   STATUS    RESTARTS   AGE
   web-0   1/1     Running   0          1h
   web-1   1/1     Running   0          1h
   web-2   1/1     Running   0          8s
   ```

3. Run the following command to view the PVCs after the StatefulSet application is scaled out:

   ```
   kubectl get pvc
   ```

Expected output:

```
NAME          STATUS   VOLUME                CAPACITY   ACCESS MODES   STORAGECLASS   AGE
disk-ssd-web-0   Bound    d-2zegw7et6xc96nbojuoo   2Gi        RWO            standard       50m
disk-ssd-web-1   Bound    d-2zefbrqggvkd10xb523h   2Gi        RWO            standard       50m
disk-ssd-web-2   Bound    d-2ze4jx1zymn4n9j3pic2   2Gi        RWO            standard       50m
```

The newly created pod uses an existing PVC.

Verify that the PVCs remain unchanged when the pod of the StatefulSet application is deleted

1. Run the following command to view the PVC that is used by the pod named web-1:

   ```
   kubectl describe pod web-1 | grep ClaimName
   ```

   Expected output:
   ```
   ClaimName:  disk-ssd-web-1
   ```

2. Run the following command to delete the pod named web-1:

   ```
   kubectl delete pod web-1
   ```

   Expected output:
   ```
   pod "web-1" deleted
   ```

3. Run the following command to view the pod:

   ```
   kubectl get pod
   ```

   Expected output:
   ```
   NAME                READY   STATUS    RESTARTS   AGE
   web-0               1/1     Running   0          1h
   web-1               1/1     Running   0          25s
   web-2               1/1     Running   0          9m
   ```

The recreated pod uses the same name as the deleted pod.

4. Run the following command to view the PVCs:

   kubectl get pvc

   Expected output:

```
NAME          STATUS   VOLUME                      CAPACITY   ACCESS MODES   STORAGECLASS   AGE
disk-ssd-web-0   Bound    d-2zegw7et6xc96nbojuoo   2Gi        RWO            standard       1h
disk-ssd-web-1   Bound    d-2zefbrqggvkd10xb523h   2Gi        RWO            standard       1h
disk-ssd-web-2   Bound    d-2ze4jx1zymn4ngj3pic2   2Gi        RWO            standard       1h
```

The recreated pod uses an existing PVC.

Verify that the StatefulSet application supports persistent storage

1. Run the following command to view the file in the /data path:

   kubectl exec web-1 -- ls /data

   Expected output:
   lost+found

2. Run the following command to create a file named statefulset in the /data path:

   kubectl exec web-1 -- touch /data/statefulset

3. Run the following command to view the file in the /data path:

   kubectl exec web-1 -- ls /data

   Expected output:
   lost+found

   statefulset

4. Run the following command to delete the pod named web-1:

   kubectl delete pod web-1

   Expected output:
   pod "web-1" deleted

5. Run the following command to view the file in the /data path:

   kubectl exec web-1 -- ls /data

Expected output:
lost+found

statefulset

The statefulset file still exists in the /data path. This indicates that data is persisted to the disk.

Lab Ends: Congratulations, you have a working Statefulsets for managing your database.

3. DaemonSets: Running One Pod per Node

A DaemonSet ensures that a copy of a specific pod is running on all (or some) nodes in the cluster. This is particularly useful for node-level services such as monitoring agents, log collectors, or networking tools that need to run on each node.

Use Cases:

1. Cluster-wide services that need to be run on every node (e.g., logging agents, monitoring agents).
2. Single-application per node deployment.

Key Features:

1. Ensures that each node gets a pod (or a set of pods).
2. Pods can be scheduled only on certain nodes by using node selectors, affinities, or taints and tolerations.
3. Automatically adds new pods to newly added nodes to the cluster.

Example: To run a DaemonSet that deploys a logging agent on every node. Create a file called daemonsets.yaml file and put the content below.

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd
spec:
  selector:
    matchLabels:
      app: fluentd
  template:
    metadata:
      labels:
        app: fluentd
    spec:
      containers:
        - name: fluentd
          image: fluent/fluentd:v1.12-debian-1
          volumeMounts:
            - name: varlog
              mountPath: /var/log
      volumes:
        - name: varlog
          hostPath:
            path: /var/log
```

This example runs a logging agent (fluentd) on each node, ensuring that logs are collected across all nodes in the cluster. Note: since you are using AIO docker desktop, you will only be able to run the pod in one node.

To apply the file, run:

```
kubectl apply -f daemonsets.yaml

kubectl get pods
kubectl get daemonset
```

4. CronJobs: Running Jobs on a Schedule

A CronJob is a workload resource that runs jobs on a scheduled basis, similar to a cron job in Unix-based systems. This is useful for tasks that need to be executed periodically, such as backups, report generation, or cleanup tasks.

Use Cases:

1. Periodic tasks like backups, database cleanups, or scheduled data processing.
2. Batch jobs that need to run at specific times or intervals.

Key Features:

1. Cron-like syntax for specifying the schedule (e.g., 0 0 * * * to run at midnight every day). You can use this link for creating your schedule https://crontab.guru/

2. Job management: Automatically creates a new job at each scheduled time and ensures that the job completes successfully.
3. Concurrency policies: Control whether multiple jobs can run concurrently or if the next job should wait for the previous one to finish.

Example 1. Create a file called cronjob.yaml file and put the content below.

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: hello
            image: busybox
            args:
            - /bin/sh
            - -c
            - date; echo Hello from the Kubernetes cluster
          restartPolicy: OnFailure
```

You can apply the file using:

```
kubectl apply -f cronjob.yaml
```

```
kubectl get pods
kubectl get cronjob
```

Example 2: A CronJob that runs a backup every day at midnight. This is an example only as we don't have the backup.sh script here.

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: backup-cronjob
spec:
  schedule: "0 0 * * *"  # Run at midnight every day
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: nginx
              image: backup-image
              command: ["/bin/sh", "-c", "backup.sh"]
          restartPolicy: OnFailure
```

This cron job runs a backup script every day at midnight.

5. Jobs: Running One-Time Tasks

A Job is a Kubernetes workload used for one-time, batch processing tasks that can be completed in the background, such as processing a queue, migrating data, or running a report.

Use Cases:

1. One-off tasks like database migrations or data processing.
2. Batch jobs that can run to completion and do not need to persist.

Key Features:

1. Completion tracking: Jobs are considered complete when the specified number of pods successfully terminate.
2. Parallel jobs: You can run multiple jobs concurrently or sequentially depending on your use case.

Example: A Job for running a database migration. Create a file called job.yaml file and put the content below.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: jobhello
spec:
  template:
    spec:
      containers:
      - name: jobhello
```

```
    image: busybox
    args:
    - /bin/sh
    - -c
    - date; echo Hello from the Kubernetes cluster
  restartPolicy: OnFailure
```

You can apply the file using:

```
kubectl apply -f job.yaml
```

```
kubectl get pods
kubectl get job
```

6. Conclusion: Choosing the Right Workload Resource

As a Kubernetes Application Developer (CKAD), choosing the correct workload resource for your application is a critical skill. Here's a summary of when to use each type:

- Deployment: For stateless applications that need scaling and rolling updates.
- DaemonSet: For running a pod on every node (e.g., monitoring, logging).
- StatefulSet: For stateful applications requiring stable storage and identity.
- CronJob: For running jobs on a scheduled basis (e.g., backups, periodic tasks).
- Job: For one-off tasks that run to completion.

Selecting the correct workload resource helps optimize application performance, scalability, and maintainability, and ensures that your applications run efficiently in a Kubernetes cluster.

Domain 2.3: Use the Helm Package Manager to Deploy Existing Packages

Helm is the most widely used package manager for Kubernetes applications. It simplifies the process of deploying, configuring, and managing Kubernetes resources by enabling you to use Helm charts, which are pre-configured Kubernetes manifests bundled together for easy installation and management. As a Kubernetes Application Developer (CKAD), mastering Helm will help you streamline your deployment process and simplify the management of complex applications.

In this section, we'll cover how to use Helm to deploy existing Helm charts (pre-packaged applications) on a Kubernetes cluster.

1. What is Helm?

Helm is a tool that helps you manage Kubernetes applications through Helm charts. A Helm chart is a collection of files that describe a related set of Kubernetes resources, such as Deployments, Services, ConfigMaps, and more. Helm packages these resources together, allowing you to deploy and manage them as a single unit.

Key Concepts:

1. **Helm Chart:** A package of pre-configured Kubernetes resources.
2. **Release:** An instance of a Helm chart deployed to a Kubernetes cluster.
3. **Repository:** A location where Helm charts are stored and shared (e.g., Helm Hub, Artifact Hub).
4. **Helm Client:** The command-line tool used to interact with Helm charts and repositories.

2. Installing Helm

Before using Helm to deploy packages, you need to install it on your local machine. Here's how you can install Helm:

Install Helm (on macOS/Linux):

On macOS using Homebrew

```
brew install helm
```

On Linux (via script)

```
curl https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3 | bash
```

Install Helm (on Windows):

You can use Chocolatey or Windows Subsystem for Linux (WSL) to install Helm on Windows:

```
choco install kubernetes-helm
```

or

winget install helm

3. Helm Repositories

Helm charts are stored in repositories, which are collections of Helm charts. By default, Helm includes the Helm Hub repository, but you can add your own or use third-party repositories.

Add a Helm Repository:

To add a new repository (e.g., the official stable repository), use the following command:

helm repo add stable https://charts.helm.sh/stable

Once added, you can update your local Helm chart repository cache:

helm repo update

4. Deploying Existing Helm Charts

To deploy an existing Helm chart to your Kubernetes cluster, you can follow these steps. We'll use the nginx-ingress chart from the official Helm repository as an example.

Step-by-Step:

1. Search for Helm Charts: To search for a chart in a Helm repository, use the helm search command:

    helm search repo nginx-ingress

This command will return a list of charts related to nginx-ingress available in your Helm repositories.

2. Install the Helm Chart:

To deploy the chart, use the helm install command followed by the release name (an identifier for the deployed application) and the name of the chart.

    helm install my-ingress stable/nginx-ingress

In this example:

- my-ingress is the release name.
- stable/nginx-ingress is the Helm chart you're installing from the stable repository.

- Helm will automatically create the necessary Kubernetes resources (Deployments, Services, ConfigMaps, etc.) based on the chart and deploy them to your Kubernetes cluster.

3. Verify the Deployment:

After installation, you can check the status of your release:

```
helm list
```

This will show you all the releases installed in your cluster.
You can also check the individual Kubernetes resources created by the chart, such as Pods and Services:

```
kubectl get pods

kubectl get services
```

4. Check the Resources Created by the Helm Chart:

To view the Kubernetes resources created by Helm, use the following command:

```
helm get all my-ingress
```

This will show you detailed information about the release, including the manifests of the deployed resources.

Install nginx from bitnami

helm repo add bitnami1 https://charts.bitnami.com/bitnami

```
helm search repo bitnami | grep nginx
```

helm install bitnami/nginx --generate-name #or helm install nginx bitnami/nginx

Expected Output:

```
NAME: nginx
LAST DEPLOYED: Mon Feb 24 21:43:42 2025
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
CHART NAME: nginx
CHART VERSION: 19.0.0
APP VERSION: 1.27.4
Did you know there are enterprise versions of the Bitnami catalog? For enhanced secure software supply chain features, unlimited pulls from Docker, LTS support, or application customization, see Bitnami Premium or Tanzu Application Catalog. See https://www.arrow.com/globalecs/na/vendors/bitnami for more information.
```

```
** Please be patient while the chart is being deployed **
NGINX can be accessed through the following DNS name from within your cluster:
    nginx.default.svc.cluster.local (port 80)
To access NGINX from outside the cluster, follow the steps below:
1. Get the NGINX URL by running these commands:
  NOTE: It may take a few minutes for the LoadBalancer IP to be available.
        Watch the status with: 'kubectl get svc --namespace default -w nginx'
    export SERVICE_PORT=$(kubectl get --namespace default -o jsonpath="{.spec.ports[0].port}" services nginx)
                export SERVICE_IP=$(kubectl get svc --namespace default nginx -o jsonpath='{.status.loadBalancer.ingress[0].ip}')
    echo "http://${SERVICE_IP}:${SERVICE_PORT}"
WARNING: There are "resources" sections in the chart not set. Using "resourcesPreset" is not recommended for production. For production installations, please set the following values according to your workload needs:
  - cloneStaticSiteFromGit.gitSync.resources
  - resources
+info https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/
⚠ SECURITY WARNING: Original containers have been substituted. This Helm chart was designed, tested, and validated on multiple platforms using a specific set of Bitnami and Tanzu Application Catalog containers. Substituting other containers is likely to cause degraded security and performance, broken chart features, and missing environment variables.
Substituted images detected:
  - docker.io/bitnami/nginx:1.27.4-debian-12-r1
⚠ WARNING: Original containers have been retagged. Please note this Helm chart was tested, and validated on multiple platforms using a specific set of Tanzu Application Catalog containers. Substituting original image tags could cause unexpected behavior.
Retagged images:
  - docker.io/bitnami/nginx:1.27.4-debian-12-r1
```

Check your deployments:

```
helm list
kubectl get deploy
kubectl get pods
kubec get svc
kubectl get ep
```

Install mysql from Bitnami

```
helm install mysql bitnami/mysql
```

Expected output:

```
NAME: mysql
LAST DEPLOYED: Mon Feb 24 21:48:57 2025
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
CHART NAME: mysql
CHART VERSION: 12.2.2
```

```
APP VERSION: 8.4.4
Did you know there are enterprise versions of the Bitnami catalog? For enhanced secure software supply chain features, unlimited pulls from Docker, LTS support, or application customization, see Bitnami Premium or Tanzu Application Catalog. See https://www.arrow.com/globalecs/na/vendors/bitnami for more information.
** Please be patient while the chart is being deployed **
Tip:
  Watch the deployment status using the command: kubectl get pods -w --namespace default
Services:
  echo Primary: mysql.default.svc.cluster.local:3306
Execute the following to get the administrator credentials:
  echo Username: root
        MYSQL_ROOT_PASSWORD=$(kubectl get secret --namespace default mysql -o jsonpath="{.data.mysql-root-password}" | base64 -d)
To connect to your database:
  1. Run a pod that you can use as a client:
kubectl run mysql-client --rm --tty -i --restart='Never' --image docker.io/bitnami/mysql:8.4.4-debian-12-r0 --namespace default --env MYSQL_ROOT_PASSWORD=$MYSQL_ROOT_PASSWORD --command -- bash
  2. To connect to primary service (read/write):
     mysql -h mysql.default.svc.cluster.local -uroot -p"$MYSQL_ROOT_PASSWORD"
WARNING: There are "resources" sections in the chart not set. Using "resourcesPreset" is not recommended for production. For production installations, please set the following values according to your workload needs:
  - primary.resources
  - secondary.resources
+info https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/
https://github.com/bitnami/charts/tree/main/bitnami/nginx/#installing-the-chart
```

5. Customizing the Helm Chart During Deployment

Helm allows you to customize the deployment of a Helm chart by overriding its default values using a values.yaml file or command-line flags. Using

--set Flag:

You can modify specific values directly through the --set flag:

```
helm install ingress stable/nginx-ingress --set replicaCount=3
```

This will override the default replica count to 3 instead of the default value specified in the chart.

You can see variables you can control from the files itself at:

https://github.com/bitnami/charts/blob/main/bitnami/nginx-ingress-controller/values.yaml

https://github.com/bitnami/charts/tree/main/bitnami/nginx-ingress-controller

https://bitnami.com/stack/nginx-ingress-controller/helm

Using a Custom values.yaml File:

Alternatively, you can create a custom values.yaml file and provide it during installation:

Put the following in values.yaml

```
controller:
    replicaCount: 3
```

Then, install the chart with your custom values:

```
helm install my-ingress stable/nginx-ingress -f custom-values.yaml
```

This approach is useful when you need to provide more complex configuration changes.

6. Upgrading an Existing Release

If you want to upgrade an existing Helm release (e.g., to a new version of the chart or with new custom values), you can use the helm upgrade command:

```
helm upgrade my-ingress stable/nginx-ingress --set replicaCount=5
```

This will update the my-ingress release with the new chart version and custom values.

7. Rolling Back a Release

If an upgrade or configuration change causes issues, you can roll back to a previous version of the release using the helm rollback command:

```
helm rollback my-ingress 1
```

In this example, 1 refers to the revision number of the release. You can view all revisions of a release with:

```
helm history my-ingress
```

This will show you the history of changes made to the release, including the revision numbers.

8. Uninstalling a Helm Release

To remove a Helm release, you can use the helm uninstall command:

```
helm uninstall my-ingress
```

This will delete all Kubernetes resources associated with the my-ingress release.

9. Benefits of Using Helm for Deployments

1. Reusability: Helm charts allow you to easily reuse application templates across multiple projects or clusters.
2. Simplified Management: Helm packages and manages complex Kubernetes applications, making it easier to handle updates, rollbacks, and configuration management.
3. Versioning: Helm supports versioned releases, making it easy to track and roll back application changes.
4. Customization: Helm allows you to easily customize application configurations without modifying the underlying YAML files manually.

10. Conclusion

Using Helm to deploy existing packages is an essential skill for a Kubernetes Application Developer (CKAD). It simplifies application deployment, management, and upgrades, especially for complex applications with multiple Kubernetes resources.

By mastering Helm, you can:

1. Easily deploy pre-configured Kubernetes applications (Helm charts).
2. Customize and manage your deployments using values files or command-line overrides.
3. Simplify updates and rollbacks using Helm's release management capabilities.

With Helm, you can focus on building and developing applications rather than worrying about the complexity of Kubernetes YAML files.

Domain 2.4: Kustomize

Kustomize is a powerful tool built into Kubernetes for customizing Kubernetes resource YAML files. It enables you to manage configurations in a more flexible and reusable manner by allowing you to define a "base" configuration and customize it per environment without modifying the original resource files.

In this section, we will dive into how Kustomize works, how to use it for customizing Kubernetes resources, and why it is essential for a Kubernetes Application Developer (CKAD) to understand it.

1. What is Kustomize?

Kustomize is a tool that helps Kubernetes users manage their configurations more efficiently. Unlike Helm, which uses templates, Kustomize works by creating a customization layer on top of existing Kubernetes manifests. Kustomize allows you to reuse the same base configuration files across different environments (e.g., development, staging, production) with environment-specific overlays.

Key features of Kustomize:

1. No Templating: Kustomize does not require any templating logic (like Helm), making it simpler to understand.
2. Customization: It enables you to customize resource definitions (e.g., replicas, image names, labels, etc.) per environment using overlays.
3. Layered Configuration: You can create a "base" set of configurations and then apply customizations on top of it for specific environments.

Installing Kustomize

- You can install kustomize but you can also use kubectl -k. The -k invokes kustomize as it is inbuilt into the kubectl command.

- You can also use kubectl kustomize to see the yaml files that would be generated without actually creating the files

2. Kustomize Workflow

The Kustomize workflow revolves around two main concepts:

- Base: The core configuration files (e.g., deployments, services) that are common across all environments.
- Overlay: Environment-specific customizations on top of the base configuration (e.g., changing the number of replicas or the image version for a specific environment).

The overall process looks like this:

1. Create a base configuration with reusable Kubernetes manifests.
2. Define environment-specific overlays that modify the base configuration.
3. Apply the kustomized resources using the kubectl command.

3. Kustomize Directory Structure

A typical Kustomize project might have the following directory structure:

```
my-app
├── base
│   ├── deployment.yaml
│   ├── kustomization.yaml
│   └── service.yaml
└── overlay
    ├──    └── prod
        ├── deployment.yaml
        └── kustomization.yamldev
    │   ├── deployment.yaml
    │   └── kustomization.yaml
```

1. **Base Directory:** Contains the common resources that will be reused across all environments.
2. **Overlays Directory:** Contains the environment-specific customizations.
3. **kustomization.yaml File**: This file is used by Kustomize to know which resources and customizations to apply.

4. Using Kustomize: Basic Example

Let's walk through a simple example of how to use Kustomize to customize Kubernetes manifests.

Step 1: Create Base Configuration

In your base/ directory, create a deployment.yaml file for your application:

base/deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 1
  selector:
    matchLabels:
```

```
    app: my-app
 template:
  metadata:
   labels:
    app: my-app
  spec:
   containers:
    - name: my-app
      image: my-app:1.0
      ports:
       - containerPort: 80
```

Next, create a kustomization.yaml file in the base directory to define the resources:

```
# base/kustomization.yaml
resources:
 - deployment.yaml
```

This kustomization.yaml tells Kustomize to manage the deployment.yaml file.

Step 2: Create Overlay for Development (dev)

In the overlays/dev/ directory, create a kustomization.yaml file:

overlays/dev/kustomization.yaml

```
resources:
 - ../../base
patches::
 - deployment.yaml
```

Here, the resources field includes the base configuration (../../base), and we also provide a patch for the deployment.yaml file to make environment-specific changes. Now, create the deployment.yaml patch for development:

```
# overlays/dev/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: my-app
spec:
 replicas: 2  # Increase replicas for the dev environment
 template:
  spec:
   containers:
    - name: my-app
      image: my-app:dev  # Use the dev version of the image
```

Step 3: Create Overlay for Production (prod)

In the overlays/prod/ directory, create a kustomization.yaml file:

```
# overlays/prod/kustomization.yaml
resources:
 - ../../base
patches:
 - deployment.yaml
```

And the deployment.yaml patch for production:

```
# overlays/prod/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: my-app
spec:
 replicas: 5  # Increase replicas for the prod environment
 template:
  spec:
   containers:
    - name: my-app
      image: my-app:latest  # Use the latest production image
```

5. Apply Kustomized Resources

Once you've created the base and overlays, you can deploy your resources to a Kubernetes cluster.

For the Development Environment:

To apply the resources for the dev environment, navigate to the overlays/dev directory and run:

```
cd overlays/dev
```

See the files that would be generated:

```
kubectl kustomize
```

Create the files:

```
kubectl apply -k .
```

This will combine the base and the dev overlay and apply the configuration to your cluster. The resulting deployment will have 2 replicas and use the my-app:dev image.

For the Production Environment:

Similarly, for production:

```
cd overlays/prod
```

See the files that would be generated:

```
kubectl kustomize
```

Create the files:

```
kubectl apply -k .
```

This will apply the production-specific customizations, such as 5 replicas and using the my-app:latest image.

6. Advantages of Using Kustomize

1. No Templating Logic: Unlike Helm, Kustomize doesn't require complex templating, which means it's easier to use for simple customization.
2. Environment-Specific Configuration: You can easily apply environment-specific customizations (e.g., different numbers of replicas, different images) without modifying the base files.
3. Reusability: Kustomize encourages reusable configurations. You can define a base configuration that can be used in different environments, reducing duplication.
4. Declarative and Version-Control Friendly: Since Kustomize works with plain YAML files, it's easier to version control, and changes are clear and declarative.

7. Kustomize vs Helm

Kustomize:	Helm:
Primarily used for customizing existing YAML files (no templating).	Uses templating to generate Kubernetes YAML files from templates.
Works by layering and patching base configurations for different environments.	Suitable for more complex, reusable packages with variables, values files, and dependencies.
Ideal for simpler scenarios where you don't need the complexity of Helm charts.	Provides additional functionality like dependency management and versioning.

8. Conclusion

Kustomize is a powerful tool for Kubernetes developers, especially for CKADs who need to manage Kubernetes resources in a clean, environment-specific manner. By allowing you to customize resource definitions across multiple environments (like

dev, staging, and prod) without modifying the original YAML files, Kustomize is a great option for managing Kubernetes resources in a maintainable way. With Kustomize, you can:

1. Create a base configuration and layer environment-specific customizations on top.
2. Easily apply customizations without modifying the base files.
3. Use simple declarative YAML files, which makes it version control-friendly and easier to understand.

By mastering Kustomize, you can streamline your Kubernetes deployment process and manage configurations more effectively, making it an essential tool in your toolkit as a Kubernetes Application Developer (CKAD).

Domain 4.1: Discover and Use Resources That Extend Kubernetes (CRDs, Operators)
As a Kubernetes Application Developer (CKAD), you will often need to extend the native capabilities of Kubernetes to meet the specific needs of your application or workload. This is where Custom Resources (CRDs) and Operators come into play. These resources allow you to customize and automate application management, creating a richer, more flexible Kubernetes ecosystem. In this section, we'll explore how to discover and use Custom Resource Definitions (CRDs) and Operators, and how they help extend Kubernetes functionality.

1. What Are Custom Resources (CRDs)?

- A Custom Resource (CR) is an extension of the Kubernetes API that allows you to add your own API objects, outside of the standard built-in Kubernetes resources (such as Pods, Deployments, and Services).
- Custom Resource Definitions (CRDs): A CRD is a way to define your own custom resources in Kubernetes. Once a CRD is created, you can manage instances of the custom resource just like any other Kubernetes object (e.g., Pods, Services, etc.) using kubectl.

Example use case: You might create a custom resource for a database configuration (e.g., DatabaseConfig) or for an application-specific resource like a CacheCluster.

2. How to Discover and Use CRDs

a) Checking for Available CRDs in Your Cluster

To discover available CRDs in your cluster, you can use the following kubectl command:

```
kubectl get crds
```

This will list all the custom resource definitions that are available in the cluster.

b) Creating a Custom Resource Definition (CRD) To extend Kubernetes with a new custom resource, you first need to define the CRD. A CRD is simply a YAML file that specifies the kind, metadata, and specifications of the custom resource.

Here is an example of a CRD definition for a custom resource called CacheCluster:
Create this file cachecluster-crd.yaml and put the below content into the file.

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: cacheclusters.example.com
spec:
  group: example.com
```

```
  names:
    kind: CacheCluster
    listKind: CacheClusterList
    plural: cacheclusters
    singular: cachecluster
  scope: Namespaced
  versions:
  - name: v1
    served: true
    storage: true
    schema:
      openAPIV3Schema:
        type: object
        properties:
          spec:
            type: object
            properties:
              size:
                type: integer
                description: "Size of the cache"
                example: 3
              engine:
                type: string
                description: "Cache engine (e.g., Redis)"
                example: "redis"
```

- Kind: Defines the custom resource's name (CacheCluster).
- Scope: Defines whether the resource is namespaced or cluster-scoped. In this case, it's namespaced.
- Spec: Specifies the fields the resource will have, such as size and engine in the example.

Once the CRD is defined, you apply it to your Kubernetes cluster:

```
kubectl apply -f cachecluster-crd.yaml
```

After the CRD is applied, Kubernetes will start recognizing the custom resource.

c) Creating and Managing Custom Resources

Once the CRD is applied, you can create instances of your custom resource (in this case, CacheCluster):

Save this as my-cache-cluster.yaml and apply it to your cluster:

```
kubectl apply -f my-cache-cluster.yaml
```

You can now manage this resource like any other Kubernetes resource:

Get the resource:

```
kubectl get cacheclusters
```

Describe the resource:

```
kubectl describe cachecluster my-cache-cluster
```

Delete the resource:

```
kubectl delete cachecluster my-cache-cluster
```

3. What Are Kubernetes Operators?

An Operator is a method of packaging, deploying, and managing a Kubernetes application. It uses Custom Resources (CRs) to extend the Kubernetes API and make the application management more automated, handling lifecycle events such as installation, scaling, backups, and upgrades.

Operators are typically written as controllers (which watch for changes to resources and take actions accordingly). They are very useful for managing stateful applications that have complex lifecycle management needs.

4. How Operators Work with CRDs

Operators use CRDs to define the resources they manage. The operator watches for changes to these CRDs, and when a change occurs, it takes actions to reconcile the desired state of the resource with the actual state. For example, if you have a CacheCluster custom resource, an operator would ensure that the right number of Redis instances are running and automatically scale them up or down based on the size field in the CacheCluster spec.

5. How to Discover, Install, and Use Operators

a) Discovering Available Operators

You can discover available Operators through OperatorHub, a catalog of Kubernetes Operators. To browse OperatorHub, visit the website:

https://operatorhub.io

Additionally, many Kubernetes distributions like OpenShift or Rancher have integrated operator catalogs.

b) Installing Operators

Operators are typically installed as Deployments or StatefulSets within a cluster. You can install Operators either manually by applying a YAML file or through Helm.

Example Installation via YAML: Suppose you want to install a Redis operator, which can manage Redis clusters using the RedisCluster custom resource.

First, find the Operator's installation YAML (often provided in the official documentation of the Operator).

You can install the Redis operator with:

```
kubectl apply -f redis-operator.yaml
```

Here are the Instructions from this link:

https://operatorhub.io/operator/redis-operator

Install Redis Operator on Kubernetes

1. Install Operator Lifecycle Manager (OLM), a tool to help manage the Operators running on your cluster.

   ```
   curl -sL https://github.com/operator-framework/operator-lifecycle-manager/releases/download/v0.31.0/install.sh | bash -s v0.31.0
   ```

2. Install the operator by running the following command:

   ```
   kubectl create -f https://operatorhub.io/install/redis-operator.yaml
   ```

   This Operator will be installed in the "operators" namespace and will be usable from all namespaces in the cluster.

3. After install, watch your operator come up using next command.

   ```
   kubectl get csv -n operators
   ```

c) Using Operators

Once the operator is installed, you can create instances of the custom resource it manages. For example, if you're using a Redis operator, you might define a RedisCluster custom resource to create a Redis cluster:

Create file my-redis-cluster.yaml with this content:

```
apiVersion: redis.redis.opstreelabs.in/v1beta1
kind: RedisCluster
metadata:
  name: redis-cluster
spec:
  clusterSize: 3
  clusterVersion: v7
  securityContext:
    runAsUser: 1000
```

```
  fsGroup: 1000
persistenceEnabled: true
kubernetesConfig:
  image: 'quay.io/opstree/redis:v7.0.5'
  imagePullPolicy: IfNotPresent
redisExporter:
  enabled: true
  image: 'quay.io/opstree/redis-exporter:v1.44.0'
  imagePullPolicy: IfNotPresent
storage:
  volumeClaimTemplate:
    spec:
      accessModes:
        - ReadWriteOnce
      resources:
        requests:
          storage: 1Gi
```

Apply it using:

```
kubectl apply -f my-redis-cluster.yaml
```

The operator will then create the necessary Redis Pods and manage the cluster lifecycle automatically.

```
kubectl get rediscluster
```

```
kubectl get pods -n operators
```

You will see that a rediscluster has been created and the pods are running in the operator namespace.

d) Updating Operators and Custom Resources

- Updating the Operator: To update the operator, you would update the Deployment or StatefulSet running the operator, either through a new version of the YAML or a Helm chart.
- Updating the Custom Resource: To update a custom resource (e.g., RedisCluster), modify the resource definition and apply the changes using kubectl apply.

Example 2: Deploying ArgoCD through Operators

1. Go to this link https://operatorhub.io/operator/argocd-operator
2. Click install to see the instructions
3. Install Operator Lifecycle Manager (OLM), a tool to help manage the Operators running on your cluster.

```
curl -sL
https://github.com/operator-framework/operator-lifecycle-manager/releases/do
wnload/v0.31.0/install.sh | bash -s v0.31.0
```

4. Install the operator by running the following command:

   ```
   kubectl create -f https://operatorhub.io/install/argocd-operator.yaml
   ```

   This Operator will be installed in the "operators" namespace and will be usable from all namespaces in the cluster.

5. After install, watch your operator come up using next command.

   ```
   kubectl get csv -n operators
   ```

6. Deploy a basic Argo CD cluster by creating a new ArgoCD resource in the namespace where the operator is installed. Put the following in argocd-basics.yaml

   ```
   apiVersion: argoproj.io/v1alpha1
   kind: ArgoCD
   metadata:
    name: example-argocd
    namespace: operators
   spec: {}
   ```

7. Create the objects from the file

   ```
   kubectl create  -f  argocd-basics.yaml
   ```

8. See the objects created

   ```
   kubectl get cm,secret,deploy -n operators
   ```

9. Obtain the admin password from the secret

   ```
   kubectl -n operators get secret example-argocd-cluster -o
   jsonpath='{.data.admin\.password}' | base64 -d
   ```

   Note: if your password appears like below, dont' use the last character '%' in the password:

   ```
   kYhzd10ZL6bXHRnET8BwV3UcJWaNOqMe%
   ```

10. Use port-forward to access the dashboard

    ```
    kubectl -n operators  port-forward service/example-argocd-server 8443:443
    ```

    The server UI should be available at https://localhost:8443/.

11. Login with username: admin and the password obtained above

Now you sould have your ArgoCD cluster up and running.

6. Benefits of CRDs and Operators

1. Automated Management: Operators enable the automation of complex application lifecycle management tasks such as installation, configuration, backup, scaling, and upgrades.
2. Custom Resource Management: CRDs allow you to define custom resources that meet the unique needs of your application.
3. Seamless Integration with Kubernetes: Both CRDs and Operators are first-class citizens in the Kubernetes ecosystem. They integrate seamlessly into Kubernetes' native management tools (kubectl, Helm, etc.).
4. Declarative Model: Both CRDs and Operators use the declarative model of Kubernetes, which ensures that the desired state of the application is continuously maintained by the Kubernetes controller.

7. Conclusion

In this section, we have covered how to discover and use resources that extend Kubernetes, specifically focusing on Custom Resource Definitions (CRDs) and Operators.

- CRDs allow you to create custom resources that extend Kubernetes' capabilities and define your own objects for managing applications.
- Operators provide an automated way to manage the lifecycle of complex applications by leveraging CRDs to handle the installation, configuration, scaling, and maintenance of stateful applications.

Mastering these concepts will significantly enhance your ability to work with Kubernetes, enabling you to build more flexible, scalable, and automated systems that go beyond the built-in Kubernetes resources.

Domain 1.4: Utilize Persistent and Ephemeral Volumes

In Kubernetes, volumes are essential for providing storage to containers within Pods. Containers are typically ephemeral, meaning that when a container is terminated or rescheduled, any data stored in its filesystem is lost. To manage data more effectively, Kubernetes offers two primary types of volumes:

- Ephemeral and
- Persistent.

As a Kubernetes Application Developer (CKAD), understanding when and how to use these different types of volumes is crucial for building scalable and resilient applications. This section will explain the difference between ephemeral and persistent volumes, provide use cases, and show how to configure them.

1. Ephemeral Volumes

- Ephemeral volumes are temporary storage that exists for the duration of a Pod.
- Once the Pod is deleted, the data stored in these volumes is also deleted.
- Ephemeral volumes are ideal for scenarios where data persistence is not required or where data can be re-created easily (e.g., temporary files, caches, or scratch space).

Common Ephemeral Volume Types:

1. emptyDir: A simple empty directory that is created when a Pod is scheduled on a node and exists as long as the Pod is running. You cannot directly list an "emptyDir" volume in Kubernetes because it is a temporary, ephemeral volume created on the node where the pod is running and is not managed as a separate entity that can be listed independently; you can only see its existence within the context of a pod's manifest when defining it as a volume type.

Key points about emptyDir:

- Created on Pod assignment: An emptyDir volume is created when a pod is assigned to a node.
- Transient data: Data stored in an emptyDir is deleted when the pod is removed from the node, even if the container restarts within the pod.
- Accessing in Pod: To see the emptyDir volume, you would need to access the pod itself and list the files within the mounted path specified in the pod manifest.

2. configMap: Stores configuration data that can be injected into Pods as files.
3. secret: Stores sensitive information, such as passwords or tokens, which are injected into Pods.
4. downwardAPI: Provides metadata about the Pod, such as labels, annotations, or resource usage metrics.
5. projected: A volume that aggregates multiple sources into one volume, such as a mix of secrets, config maps, or downward API data.

Use Cases for Ephemeral Volumes:

1. Temporary files: Storing files that don't need to persist beyond the life of a Pod (e.g., logs, caches).
2. Configuration data: Using configMap or secret volumes to inject configuration files or environment variables into Pods.
3. Shared memory: Using emptyDir for sharing files between containers within the same Pod (e.g., a web server and a sidecar container).

Example: emptyDir Volume

Here's an example of using an emptyDir volume in a Pod to store temporary data:
File: ephemeral-volume-example

```
apiVersion: v1
kind: Pod
metadata:
  name: ephemeral-volume-example
spec:
  containers:
    - name: main-container
      image: busybox
      command: ["sh", "-c", "echo 'Hello World' > /data/hello.txt; sleep 3600"]
      volumeMounts:
        - mountPath: /data
          name: emptydir-volume
  volumes:
    - name: emptydir-volume
      emptyDir: {}
```

In this example:

- The emptyDir volume is created for the Pod, and the main container writes data to the /data directory.
- The data stored in emptyDir will only exist for as long as the Pod is running. If the Pod is deleted, the data is lost.

ConfigMaps

File: pod-with-configmap-vol.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: log-config
data:
  username: k8s-admin
  access_level: "1"
---
apiVersion: v1
```

```
kind: Pod
metadata:
 name: configmap-pod
spec:
 containers:
  - name: test
    image: busybox:1.28
    command: ['sh', '-c', 'echo "The app is running!" && tail -f /dev/null']
    volumeMounts:
      - name: config-vol
        mountPath: /etc/config
 volumes:
  - name: config-vol
    configMap:
      name: log-config
      #items:
      #  - key: username
      #    path: keys
       # - key: access_level
        #   path: keys
#https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/
```

Secrets

File: pod-with-secret-vol.yaml

```
apiVersion: v1
kind: Secret
metadata:
 name: test-secret
data:
 username: bXktYXBw
 password: Mzk1MjgkdmRnN0pi
---
apiVersion: v1
kind: Pod
metadata:
 name: secret-test-pod
spec:
 containers:
  - name: test-container
    image: nginx
    volumeMounts:
      # name must match the volume name below
      - name: secret-volume
```

```
      mountPath: /etc/secret-volume
      readOnly: true
 # The secret data is exposed to Containers in the Pod through a Volume.
 volumes:
  - name: secret-volume
    secret:
     secretName: test-secret
#https://kubernetes.io/docs/tasks/inject-data-application/distribute-credentials-secure/#set-posix-permissions-for-secret-keys
```

2. Persistent Volumes (PVs)

- Unlike ephemeral volumes, persistent volumes are designed to store data that needs to persist beyond the life of a Pod. This makes them ideal for applications that require durable storage, such as databases or file storage.
- Lifecycle independent of Pods: PVs exist independently of Pods and retain data even if a Pod is deleted or rescheduled.
- Persistent Volumes can be backed by various storage systems such as:
    - local disks,
    - hostPath
    - cloud-based block storage (e.g., AWS EBS, GCE Persistent Disk),
    - or network-attached storage (e.g., NFS, GlusterFS, CEPH, iSCSI).
- Access Modes: PVs can have different access modes, determining how they can be mounted by Pods. These include:
    - ReadWriteOnce (RWO): The volume can be mounted as read-write by a single node.
    - ReadOnlyMany (ROX): The volume can be mounted as read-only by many nodes.
    - ReadWriteMany (RWX): The volume can be mounted as read-write by many nodes.

Use Cases for Persistent Volumes:

1. Databases: Storing the data of stateful applications, such as databases, that require persistent storage.
2. Shared storage: Enabling multiple Pods to access the same storage for read/write purposes (e.g., file sharing).
3. Long-term data storage: For applications that need to store user data, logs, or backups.

Key Concepts for Persistent Volumes:

1. PersistentVolume (PV): A resource in the cluster that represents a piece of storage. It's provisioned by an administrator or dynamically provisioned using StorageClasses.
2. PersistentVolumeClaim (PVC): A request for storage by a user/application. A PVC is bound to a PV that meets its storage requirements.
3. StorageClass: Defines the type of storage to be used (e.g., standard, SSD, NFS) and allows for dynamic provisioning of PVs.

3. How PVs, PVCs, and Storage Classes Work Together

PV and PVC:

When an application requires storage, a PersistentVolumeClaim (PVC) is created by the user or application, specifying the desired storage size, access mode, and storage class (optional). Binding a PVC to a PV:
Kubernetes searches for an available Persistent Volume (PV) that meets the requirements specified in the PVC. If a matching PV exists, Kubernetes binds the PVC to the PV. If no suitable PV exists and the PVC has a storageClassName, Kubernetes may trigger the dynamic provisioning of a new PV based on the specified

StorageClass.

A StorageClass is a Kubernetes resource that defines the type of storage to be used for dynamic provisioning. Storage Classes specify which storage backends can be used (e.g., AWS EBS, GCP Persistent Disks, Ceph, etc.), as well as parameters such as IOPS (Input/Output Operations Per Second), replication, and encryption.

Dynamic Provisioning:

When dynamic provisioning is enabled (via a StorageClass), the provisioner defined in the StorageClass will automatically provision a PV that meets the criteria specified in the PVC. For example, if you use AWS as the provisioner, a new EBS volume will be created and bound to the PVC. Releasing and Reclaiming a PV:
Once the PVC is no longer needed (e.g., the application is deleted), the PVC is deleted, and the bound PV becomes available. The reclaim policy defined in the PV (either Retain, Delete, or Recycle) determines what happens to the data on the PV and whether the storage resource is deleted or retained for future use.

Key Features of Storage Classes:

- Dynamic Provisioning: When a PVC specifies a storageClassName, Kubernetes will use the associated Storage Class to provision a PV automatically if no matching PV exists.
- Provisioning Parameters: You can define parameters for different storage backends (e.g., for AWS, GCP, etc.).
- Reclaim Policy: The StorageClass can define the reclaim policy for PVs (whether to delete the volume, retain it, or recycle it).

Example Storage Class YAML:

Here's an example of a Storage Class definition:
File: storage-class.yaml

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
```

```
metadata:
  name: standard
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
  iopsPerGB: "10"
reclaimPolicy: Retain
```

This configuration:

- Provisioner: The storage will be provisioned by the kubernetes.io/aws-ebs provisioner.
- Parameters: Defines specific parameters for the AWS EBS volumes, such as the volume type (gp2) and IOPS per GB.
- Reclaim Policy: The Retain policy will ensure that the volume is not deleted when the PVC is deleted.

4. Example Walkthrough of PV, PVC and StorageClasses

Persistent Volume

Here's an example that demonstrates how to use a PersistentVolume (PV) and PersistentVolumeClaim (PVC):

PersistentVolume (PV) definition:

File: pv.yaml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: my-pv
spec:
  capacity:
    storage: 1Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  #storageClassName: standard
  hostPath:
    path: /mnt/data
```

This configuration:

- Storage: 1Gi of storage.

- Access Mode: Mounted as read-write by a single node (ReadWriteOnce).
- Reclaim Policy: The PV is retained after it is released from a PVC (Retain), meaning the data remains available.
- Storage Class: The standard class will be used for dynamic provisioning.

PersistentVolumeClaim (PVC) definition:

- Claims resources from available PVs: PVCs match and bind to available PVs that satisfy the requested storage capacity and access modes.
- Dynamic provisioning: When a PVC is created, Kubernetes can dynamically provision a PV if there is no available one that matches the PVC's requirements. This is managed through Storage Classes.
- Storage Class Awareness: PVCs can specify a storageClassName, which determines how the underlying PV is provisioned (either dynamically or statically).

File: pvc.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  #storageClassName: standard
```

This configuration:

- Storage: 1Gi of storage.
- Access Mode: Mounted as read-write by a single node (ReadWriteOnce).
- Reclaim Policy: The PV is retained after it is released from a PVC (Retain), meaning the data remains available.
- Storage Class: The standard class will be used for dynamic provisioning.
- The PVC specifies a storage request of 1Gi and is bound to a PV that can fulfill these requirements.
- The accessModes and storageClassName should match the PV's definition.

Pod Using PVC:

File: pod-with-pvc.yaml

```
apiVersion: v1
kind: Pod
metadata:
```

```
 name: pvc-pod
spec:
 containers:
  - name: main-container
    image: busybox
      command: ["sh", "-c", "echo 'Persistent storage' > /mnt/data/storage.txt; sleep
3600"]
    volumeMounts:
     - mountPath: /mnt/data
       name: pvc-storage
 volumes:
  - name: pvc-storage
    persistentVolumeClaim:
     claimName: my-pvc
```

In this setup:

1. The Pod uses a PersistentVolumeClaim (PVC) to request storage from the PV.
2. The container writes data to /mnt/data/storage.txt, which is backed by the persistent volume.

Reclaim Policies

The Reclaim Policy is an important setting in both PVs and Storage Classes that dictates what happens to the persistent volume when the associated PVC is deleted.

- Retain: When the PVC is deleted, the PV will not be deleted. The data remains on the disk, and the PV will go into the Released state. It must be manually cleaned and reused or deleted.
- Delete: When the PVC is deleted, the PV is also deleted, and the underlying storage is cleaned up (for dynamic provisioning).
- Recycle: The PV is scrubbed (data is wiped) and made available for reuse (this reclaim policy is deprecated and not widely used).

Example of Reclaim Policy:

```
apiVersion: v1
kind: PersistentVolume
metadata:
 name: my-pv
spec:
 capacity:
  storage: 10Gi
 persistentVolumeReclaimPolicy: Retain
```

Access Modes

The accessMode defines how a PV can be accessed by Pods:

- ReadWriteOnce (RWO): The volume can be mounted as read-write by a single node.
- ReadOnlyMany (ROX): The volume can be mounted as read-only by many nodes.
- ReadWriteMany (RWX): The volume can be mounted as read-write by many nodes.

Example Access Mode:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
 name: my-pvc
spec:
 accessModes:
  - ReadWriteOnce
 resources:
  requests:
   storage: 5Gi
```

Use Case Examples

a. Shared Filesystem with ReadWriteMany You can use a ReadWriteMany (RWX) access mode if you need to mount a volume that allows multiple nodes to read/write to the volume, useful for shared storage scenarios like file systems (e.g., NFS). b. Cloud Volumes for Stateful Apps If you're running a stateful application (like a database) in your Kubernetes cluster, you might use a ReadWriteOnce (RWO) volume, such as an AWS EBS volume or GCP Persistent Disk, which is tied to a single node but ensures data persistence.

Difference Between Ephemeral and Persistent Volumes

Feature	Ephemeral Volumes	Persistent Volumes
Lifetime	Tied to the lifecycle of a Pod.	Exists independently of Pods and can persist beyond Pod lifecycle.
Use Case	Temporary storage for caches, logs, or scratch data.	Durable storage for stateful applications like databases.
Data Loss	Data is lost when the Pod is deleted.	Data persists even if the Pod is deleted.

Backends	Typically local (e.g., emptyDir, configMap, secret).	Can be backed by cloud storage, NFS, or other persistent storage solutions.
Access Modes	Read/write for a single Pod/container.	Access modes like ReadWriteOnce, ReadOnlyMany, or ReadWriteMany for multiple Pods.

5. Best Practices for Using Volumes

- Use Ephemeral Volumes for Temporary Data: If your application generates temporary data that doesn't need to persist after the Pod terminates, use ephemeral volumes (e.g., emptyDir, configMap, or secret).
- Use Persistent Volumes for Stateful Applications: For applications that require data persistence (e.g., databases, logging systems), use persistent volumes backed by cloud storage or networked file systems.
- Claim Resources Efficiently: When using Persistent Volumes, ensure that your PVCs request appropriate storage resources based on your application's needs, and be mindful of storage classes and capacity.
- Security Considerations: When using sensitive data, consider using secret volumes for injecting secrets (e.g., passwords, API keys) into Pods securely.

6. Conclusion

As a Kubernetes Application Developer (CKAD), understanding how to use both ephemeral and persistent volumes is critical for building applications that are both scalable and resilient. By using ephemeral volumes for temporary data and persistent volumes for data that needs to be stored long-term, you can ensure your applications are both performant and reliable. Understanding these concepts and how they interconnect is critical for successfully managing storage in Kubernetes and ensuring that your applications have the persistent storage they need, while maintaining flexibility and scalability.

- Persistent Volumes (PVs) provide a way to abstract storage resources in the cluster, making it easy to manage storage.
- Persistent Volume Claims (PVCs) allow users to request storage without worrying about how it's provisioned.
- Storage Classes define how storage should be dynamically provisioned, with specific parameters based on your environment (e.g., cloud provider).

Make sure to:

- Use ephemeral volumes for temporary data that doesn't need to survive beyond the Pod.
- Use persistent volumes for stateful applications or when you need data persistence across Pod restarts or node failures.
- Proper volume management is a key component of Kubernetes application development, and mastering it will help you build efficient and effective cloud-native applications.

Domain 5.2: Provide and Troubleshoot Access to Applications via Services

In Kubernetes, Services are an abstraction that defines how to expose and access Pods in a network. Services provide a stable IP address or DNS name for accessing Pods, abstracting the complexity of dynamically changing Pod IP addresses as Pods are created, destroyed, or rescheduled across nodes.
In this section, we'll cover:

- How to create and configure Services in Kubernetes.
- How to troubleshoot Service access issues.
- Common Service types and their use cases.
- Key concepts and tools used in diagnosing Service-related problems.

1. Overview of Kubernetes Services

Kubernetes Services expose Pods to network traffic and are essential for providing reliable, stable access to applications. By default, Pods are ephemeral (i.e., they can be created and destroyed dynamically), so Services provide an abstraction that helps ensure communication with Pods, even if their IPs change.

Types of Services:

There are four main types of Kubernetes Services:

- ClusterIP (default): Exposes the service on a cluster-internal IP. This makes the Service accessible only from within the cluster.
- NodePort: Exposes the service on a static port on each node's IP. This allows access to the Service from outside the cluster.
- LoadBalancer: Provisions a load balancer (if supported by the cloud provider) to expose the Service externally.
- ExternalName: Maps the service to an external DNS name, allowing Kubernetes resources to access external services using a DNS alias.

Key Concepts:

- ClusterIP is the most common type and is used when the service does not need to be accessed externally.
- Endpoints are the set of Pods that are selected by a Service's label selector.

2. Creating a Kubernetes Service

Here's an example of how to create a Service in Kubernetes. Service Example: Exposing an Application Using ClusterIP This YAML file creates a Service of type ClusterIP that exposes a Pod running an HTTP server.

File: clusterip.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: my-app-service
spec:
  selector:
```

```
  app: my-app
 ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
```

Explanation:

- selector: Selects the Pods labeled app=my-app to associate with the Service.
- ports: The Service exposes port 80 and forwards traffic to Pods on port 8080.

Service Example: Exposing an Application Using NodePort

A NodePort service exposes your application outside the Kubernetes cluster. Below is an example of a NodePort service:
File: nodeport.yaml

```
apiVersion: v1
kind: Service
metadata:
 name: my-app-nodeport
spec:
 selector:
  app: my-app
 ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
    nodePort: 30007
 type: NodePort
```

Explanation:

- nodePort: 30007: The service will be accessible externally on port 30007 on every node in the cluster.
- type: NodePort: This type exposes the service on a static port across all nodes in the cluster.

3. Troubleshooting Service Access

Sometimes, access to services in Kubernetes may not work as expected. There are several common reasons for Service-related issues. Here are steps to troubleshoot these problems:

Common Issues:

Service not accessible externally: The most common reason for this is that the service type is incorrectly set or the NodePort or LoadBalancer is not properly configured.

- Pods not selected by the Service: The Service's label selector may not match the Pods you want to expose.
- Service not routing traffic to Pods: Misconfigured targetPort or issues with the underlying Endpoints can result in traffic not reaching the Pods.
- DNS issues: If the application cannot be accessed using the Service name, there may be issues with DNS resolution inside the cluster.

Troubleshooting Steps:

1. Check the Service Configuration Verify that the Service is created and configured correctly.

   kubectl get svc my-app-service

Check if the Service is of the correct type (ClusterIP, NodePort, LoadBalancer) and if the correct ports are exposed.

2. Check Service Endpoints A Service relies on Endpoints to route traffic to Pods. If no Endpoints are associated with the Service, traffic won't reach the Pods.

   kubectl get endpoints my-app-service

If no Endpoints are listed, check if the Pod label selector is correct and whether Pods matching the selector are running.

3. Verify Pod Labels and Selectors Ensure that the labels of the Pods match the Service's label selector.

kubectl get pods --show-labels

For the Service to work correctly, the Pods should have labels that match the selector in the Service definition.

4. Check Pod Logs If the Pods are selected correctly but the service still doesn't respond, check the Pod logs to ensure the application inside the Pods is functioning correctly.

   kubectl logs <pod-name>

Check if there are any issues with the application that might be preventing it from accepting traffic on the specified port.

5. Verify Network Connectivity You can use the kubectl exec command to troubleshoot network connectivity to the Pods from other Pods.

   kubectl exec -it <pod-name> -- curl <service-name>:<service-port>

This command tests whether a Pod can reach the Service and port. This is useful for diagnosing network issues between Pods.

6. Investigate NodePort and LoadBalancer Services If using a NodePort or LoadBalancer service, check that the service is exposed properly.
   NodePort: Ensure that the port is accessible on all the nodes in the cluster.

   kubectl get svc my-app-nodeport

LoadBalancer: If using a LoadBalancer service, verify if the cloud provider has provisioned the external load balancer and that the IP is accessible.
kubectl describe svc my-app-lb

Check if the EXTERNAL-IP is listed and whether it's reachable from outside the cluster.

7. DNS Resolution If the application is not reachable by its Service name, ensure that the DNS service in Kubernetes is running correctly. You can check if DNS resolution is working properly by querying the DNS from within a Pod:

   kubectl exec -it <pod-name> -- nslookup my-app-service

4. Best Practices for Exposing Applications via Services

1. Use the Correct Service Type:
- Use ClusterIP for internal applications.
- Use NodePort or LoadBalancer for external access.
- ExternalName is useful for redirecting traffic to external services via DNS.
2. Match Labels Correctly: Ensure that the label selectors on the Service match the labels on the Pods to ensure correct routing of traffic.
3. Health Checks and Probes: Always implement readiness and liveness probes in your Pods. These help ensure that Pods are ready to handle traffic before the Service routes requests to them.
4. Limit Port Exposure: Only expose the necessary ports for each application. This minimizes the attack surface and limits the potential for unauthorized access.
5. Use Namespace Isolation: If your application spans multiple namespaces, ensure that the appropriate namespace selectors are used in the Service definition.

5. Conclusion

Providing and troubleshooting access to applications in Kubernetes using Services is a fundamental part of managing Kubernetes environments. Services provide a stable interface to access Pods, abstracting away the complexities of pod IP address management. By understanding the types of services available and troubleshooting common issues, you can ensure that your applications are reliably exposed to the network and that access problems can be quickly diagnosed and fixed.

Key takeaways:

- Services expose Pods and enable stable access to applications.
- Use the correct Service type based on your use case (ClusterIP, NodePort, LoadBalancer, ExternalName).
- Troubleshoot by verifying Service configurations, labels, endpoints, and Pod health.
- Leverage DNS and network debugging tools to identify connectivity issues.

Domain 5.3: Use Ingress Rules to Expose Applications

In Kubernetes, Ingress is an API object that manages external access to services within a cluster, typically HTTP and HTTPS traffic. Ingress allows you to expose HTTP/S-based applications in a Kubernetes cluster to the outside world, providing advanced routing, SSL termination, load balancing, and host-based or path-based routing.
In this section, we will cover:

- How to define and configure Ingress rules.
- Common use cases for Ingress in Kubernetes.
- How to troubleshoot Ingress-related issues.
- Best practices when using Ingress to expose applications.

1. Overview of Kubernetes Ingress

Ingress is a powerful Kubernetes resource that allows you to configure external HTTP/S access to your services. Unlike Services, which provide simple access to a group of Pods within the cluster, Ingress allows you to define fine-grained routing rules and handle SSL termination, path-based routing, and more.

Components of Ingress:

- Ingress Resource: The configuration for how external HTTP/S traffic should be routed to Services inside the cluster.
- The Ingress is a Kubernetes resource that lets you configure an HTTP load balancer for applications running on Kubernetes, represented by one or more Services. Such a load balancer is necessary to deliver those applications to clients outside of the Kubernetes cluster.
- The Ingress resource supports the following features:
    - Content-based routing:
    - Host-based routing. For example, routing requests with the host header foo.example.com to one group of services and the host header bar.example.com to another group.
    - Path-based routing.
      For example, routing requests with the URI that starts with /serviceA to service A and requests with the URI that starts with /serviceB to service B. TLS/SSL termination for each hostname, such as foo.example.com. See the Ingress User Guide to learn more about the Ingress resource.
- Ingress Controller: A controller that implements the rules defined in the Ingress resource. It is responsible for managing and directing traffic based on Ingress rules. Popular controllers include NGINX, Traefik, and HAProxy.
- The Ingress Controller is an application that runs in a cluster and configures an HTTP load balancer according to Ingress resources. The load balancer can be a software load balancer running in the cluster or a hardware or cloud load balancer running externally. Different load balancers require different Ingress Controller implementations.
- In the case of NGINX, the Ingress Controller is deployed in a pod along with the load balancer.

Benefits of Using Ingress:

- URL-based Routing: Route traffic to different services based on the URL path (e.g., /app1 to one service, /app2 to another).
- Host-based Routing: Route traffic to services based on the host domain (e.g., example.com vs app.example.com).
- SSL Termination: Terminate SSL/TLS at the Ingress controller, offloading the burden from individual services.
- Path-based Routing: Route traffic to different services based on request paths.

2. How to Define Ingress Rules

Ingress rules are defined using a Kubernetes Ingress resource. A basic Ingress configuration looks like this: Example: Basic Ingress Resource

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
 name: my-app-ingress
 namespace: default
spec:
 rules:
  - host: my-app.example.com
    http:
     paths:
      - path: /
       pathType: Prefix
       backend:
        service:
         name: my-app-service
         port:
          number: 80
```

Explanation:

- host: The domain name (my-app.example.com) that will be used to route traffic to this Ingress.
- http: This section defines HTTP routing rules.
- paths: Defines URL paths and the backend service to forward traffic to. In this case, any traffic to / will be directed to my-app-service on port 80.
- pathType: Defines how the path should be interpreted (Prefix matches the path and everything that starts with it, Exact matches only the exact path).

Example: Ingress with Path and Host-based Routing

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
 name: multi-path-ingress
```

```
  namespace: default
spec:
  rules:
    - host: example.com
      http:
        paths:
          - path: /app1
            pathType: Prefix
            backend:
              service:
                name: app1-service
                port:
                  number: 80
          - path: /app2
            pathType: Prefix
            backend:
              service:
                name: app2-service
                port:
                  number: 80
```

Explanation:

- Traffic to example.com/app1 will be routed to app1-service, while traffic to example.com/app2 will be routed to app2-service.

Example: Ingress with SSL Termination

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-ssl-ingress
  annotations:
    nginx.ingress.kubernetes.io/ssl-redirect: "true"
spec:
  rules:
    - host: secure-app.example.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: secure-app-service
                port:
                  number: 443
  tls:
```

```
- hosts:
    - secure-app.example.com
  secretName: my-ssl-secret
```

Explanation:

- This configuration enables SSL termination at the Ingress controller for the host secure-app.example.com.
- The SSL certificate is stored in a Secret (my-ssl-secret), which is used by the Ingress controller to handle HTTPS traffic.
- The nginx.ingress.kubernetes.io/ssl-redirect annotation ensures HTTP requests are redirected to HTTPS.

3. Setting Up an Ingress Controller

In order to use Ingress, you need to have an Ingress controller running in your cluster. Kubernetes does not come with a built-in Ingress controller, so you will need to deploy one. Here's an example of how to deploy the NGINX Ingress Controller:

Deploy NGINX Ingress Controller Create the NGINX Ingress Controller using Helm or kubectl. To install NGINX via Helm, run:

```
helm install nginx-ingress ingress-nginx/ingress-nginx --namespace kube-system
```

Alternatively, to deploy using kubectl:

```
kubectl apply -f
https://raw.githubusercontent.com/kubernetes/ingress-nginx/main/deploy/static
/provider/cloud/deploy.yaml
```

This will create all necessary resources for NGINX to act as your Ingress controller. Once deployed, verify that the controller is running:

```
kubectl get pods -n kube-system
```

You can check if the controller has an external IP address by running:

```
kubectl get svc -n kube-system
```

Lab Walkthrough:

1. Install the Ingress controller using helm

```
helm upgrade --install ingress-nginx ingress-nginx --repo
https://kubernetes.github.io/ingress-nginx --namespace ingress-nginx
--create-namespace
```

Expected output:
The ingress-nginx controller has been installed.

It may take a few minutes for the load balancer IP to be available.

You can watch the status by running:

```
kubectl get service --namespace ingress-nginx ingress-nginx-controller --output wide --watch
```

An example Ingress that makes use of the controller (Don't create this at this point)

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
 name: path-based-ingress
 annotations:
   nginx.ingress.kubernetes.io/rewrite-target: /
spec:
 ingressClassName: nginx
 rules:
   - host: localhost
     http:
       paths:
         - path: /app1
           pathType: Prefix
           backend:
             service:
               name: app1
               port:
                 number: 80
         - path: /app2
           pathType: Prefix
           backend:
             service:
               name: app2
               port:
                 number: 80
```

2. Check that ingress controller is running

```
kubectl get pods --namespace ingress-nginx
```

Expected output:

```
NAME                                        READY   STATUS    RESTARTS   AGE
ingress-nginx-controller-5f649bbff8-gs8j2   1/1     Running   0          89s
```

3. Check the Ingress Controller and take note of the EXTERNAL-IP. You must see 'localhost'. If you dont see localhost, check and delete any service of type 'LoadBalancer'.

```
kubectl get service ingress-nginx-controller --namespace=ingress-nginx
```

Expected output:

```
NAME                       TYPE           CLUSTER-IP      EXTERNAL-IP   PORT(S)                      AGE
ingress-nginx-controller   LoadBalancer   10.107.184.189   localhost     80:31210/TCP,443:30943/TCP   4m49s
```

4. Create deployment and service objects. Put the information in deploy1.yaml

```
 apiVersion: apps/v1
kind: Deployment
metadata:
 name: nginx
spec:
 replicas: 1
 selector:
  matchLabels:
    app: nginx
 template:
  metadata:
    labels:
      app: nginx
  spec:
    containers:
    - name: nginx
      image: nginx
      ports:
      - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
```

```
 name: nginx
spec:
 type: NodePort
 selector:
  app: nginx
 ports:
  - port: 80
    targetPort: 80
```

Expected output:
User "demo-user" set.

5. create the objects

   kubectl apply -f deploy1.yaml

   Expected output:

   deployment.apps/nginx created
   service/nginx created

6. Create ingress resource. Put the following in ingress.yaml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
 name: path-based-ingress
 annotations:
  nginx.ingress.kubernetes.io/rewrite-target: /
spec:
 ingressClassName: nginx
 rules:
  - host: localhost
    http:
     paths:
      - path: /app1
        pathType: Prefix
        backend:
          service:
            name: app1
            port:
              number: 80
```

7. Create the ingress resource

   kubectl apply -f ingress.yaml

Expected output.
ingress.networking.k8s.io/example-ingress created

8. check the deployment resources created

```
kubectl get pods
```

Expected output.
NAME READY STATUS RESTARTS AGE

nginx-7c5ddbdf54-7ccnq 1/1 Running 0 2m11s

9. Check the service object

```
kubectl get svc
```

Expected output. Your output may differ
nginx NodePort 10.106.222.193 <none> 80:30175/TCP 58s

10. Test ingress.

```
curl localhost/app1
```

```
 <!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>
```

11. Create another path by using Apache image

Create deployment and service objects. Put the information in deploy2.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: apache
spec:
 replicas: 1
 selector:
   matchLabels:
     app: apache
 template:
   metadata:
     labels:
       app: apache
   spec:
     containers:
     - name: apache
       image: httpd
       ports:
       - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
 name: apache
spec:
 type: NodePort
 selector:
   app: apache
 ports:
   - port: 80
     targetPort: 80
```

12. create the objects

```
kubectl apply -f deploy2.yaml
```

Expected output:

```
deployment.apps/apache created
service/apache created
```

13. Modify the ingress resource. Add to the ingress.yaml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
 name: path-based-ingress
 annotations:
  nginx.ingress.kubernetes.io/rewrite-target: /
spec:
 ingressClassName: nginx
 rules:
  - host: localhost
    http:
     paths:
      - path: /app1
        pathType: Prefix
        backend:
          service:
            name: app1
            port:
              number: 80
      - path: /app2
        pathType: Prefix
        backend:
          service:
            name: app2
            port:
              number: 80
```

14. Create the ingress resource

```
kubectl apply -f ingress.yaml
```

Expected output.

```
ingress.networking.k8s.io/example-ingress configured
```

15. check the deployment resources created

```
kubectl get pods
```

Expected output.

```
NAME                     READY   STATUS    RESTARTS     AGE

apache-644bc8865b-7gz8c  1/1     Running   0            11s

nginx-7c5ddbdf54-7ccnq   1/1     Running   0            2m11s
```

16. Check the service object

```
kubectl get svc
```

Expected output. Your output may differ

```
nginx          NodePort    10.106.222.193   <none>       80:30175/TCP      58s

apache         NodePort    10.96.150.54     <none>       80:31125/TCP      4m
```

17. Test ingress for both applications

```
curl localhost/app2

curl localhost/app1
```

4. Troubleshooting Ingress Issues

If you're experiencing issues with your Ingress configuration, there are several key areas to check:

1. Verify Ingress Resource Configuration

Check if the Ingress resource is properly created and configured.

```
kubectl get ingress my-app-ingress -o yaml
```

Ensure that the host, paths, and backend service names are correctly defined.

2. Check Ingress Controller Logs

The Ingress controller is responsible for routing the traffic based on your rules. If traffic is not being routed correctly, check the logs of your Ingress controller.
For NGINX Ingress, you can run:

```
kubectl logs <nginx-ingress-pod> -n kube-system
```

Look for errors related to configuration, missing services, or SSL issues.

3. Verify Service and Pod Availability Ensure that the backend Service and Pods are running and reachable by the Ingress controller.

```
kubectl get svc my-app-service

kubectl get pods -l app=my-app
```

Verify that the service exists and that there are Pods available to handle requests.

4. Check DNS Resolution Ensure that the DNS for the Ingress host is correctly set up. For example, if you're using my-app.example.com, make sure that the DNS record points to the Ingress controller's external IP address or LoadBalancer IP. Use nslookup or dig to verify:

```
nslookup my-app.example.com
```

5. SSL/TLS Issues If you're using SSL, verify that the SSL certificate is correctly installed and the TLS Secret is present.

```
kubectl get secret my-ssl-secret
```

If there are issues with SSL/TLS, check the Ingress controller logs for SSL-related errors.

6. Best Practices for Using Ingress

- Use Annotations for Fine-grained Control: Many Ingress controllers (like NGINX) support annotations for additional configuration, such as SSL redirection, rewrite-targets, rate limiting, etc. Make sure to utilize them when needed.
- Leverage Path and Host-Based Routing: Organize your services logically by exposing multiple applications through the same domain using path-based or host-based routing. This reduces the need for multiple public IPs or LoadBalancers.
- Secure Ingress with SSL: Always use SSL/TLS for secure communication. Use Kubernetes Secrets to manage SSL certificates and configure your Ingress controller to terminate SSL/TLS traffic.
- Monitor and Log Traffic: Enable logging and monitoring in the Ingress controller to troubleshoot issues and monitor traffic patterns.
- Consider Using a Centralized Ingress Resource: If you have multiple namespaces and applications, consider using a centralized Ingress resource or controller for consistent routing policies across all applications.

6. Conclusion

Ingress provides a powerful way to expose Kubernetes applications to the outside world. It offers advanced features like path-based routing, SSL termination, and host-based routing, which makes it ideal for complex applications with multiple services. By understanding how to define Ingress resources and troubleshoot issues, you can ensure that your Kubernetes applications are accessible and functioning as intended.

Key Takeaways:

- Ingress allows advanced HTTP/S routing for Kubernetes applications.
- Ingress controllers are needed to implement Ingress rules, with NGINX being a popular choice.
- Ensure DNS, SSL, and Service configurations are correctly set up.

Troubleshoot

- issues by checking logs, resources, and DNS configuration.
- Follow best practices for securing and organizing your Ingress resources.

Certified Kubernetes Application Developer (CKAD)

By Damian Igbe, PhD

Day 3

Domain 5.1: Demonstrate Basic Understanding of NetworkPolicies

NetworkPolicies in Kubernetes are used to control the communication between Pods. They define the rules and policies that control how groups of Pods are allowed to communicate with each other and with other network endpoints outside the Kubernetes cluster. By defining NetworkPolicies, you can restrict traffic to and from Pods based on various factors, such as namespace, IP address, and labels.
In this section, we'll cover the basics of NetworkPolicies:

- What NetworkPolicies are
- How they work in Kubernetes
- How to define and apply NetworkPolicies
- Common use cases and best practices for NetworkPolicies

1. What are NetworkPolicies?

A NetworkPolicy in Kubernetes is an API object that defines a set of rules governing how Pods communicate with each other and with external services. By default, Pods are open to communication from any other Pod within the same namespace, and they can initiate connections to external services. However, NetworkPolicies allow you to restrict this communication to only the Pods and external services you specify. NetworkPolicies are enforced by network plugins that support the NetworkPolicy API, such as Calico, Cilium, or Weave.

2. How Do NetworkPolicies Work?

NetworkPolicies work by specifying ingress (incoming traffic) and egress (outgoing traffic) rules for Pods.

These rules can specify:

- Selectors for Pods (e.g., by labels)
- Ports that are open or closed for communication
- Namespaces that are allowed or disallowed
- IP Blocks that control access to or from specific IP ranges

By default, if a NetworkPolicy is not applied, Pods are open to any communication. Once a NetworkPolicy is applied, it acts as a firewall, allowing only traffic that matches the rules defined in the policy and rejecting all other traffic.

3. Key Concepts

Ingress and Egress:

- Ingress refers to inbound traffic to a Pod.
- Egress refers to outbound traffic from a Pod.

You can define rules for both ingress and egress to control the flow of traffic into and out of a Pod.

- Pod Selectors: Pod selectors are used to identify which Pods the policy applies to. This can be based on labels assigned to Pods.
- Policy Types: You can define the ingress and/or egress rules in a NetworkPolicy:

- Ingress: Defines what inbound traffic is allowed.
- Egress: Defines what outbound traffic is allowed.
- Namespace Selectors: You can use namespace selectors to control communication between Pods in different namespaces.

4. Basic Example of a NetworkPolicy

Let's look at a simple example of a NetworkPolicy that restricts inbound traffic to a Pod:

NetworkPolicy Example: Allow HTTP Inbound Traffic
In this example, we will create a NetworkPolicy to allow only HTTP traffic (port 80) from other Pods in the same namespace.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-http-ingress
spec:
  podSelector:
    matchLabels:
      app: my-app
  ingress:
  - ports:
    - protocol: TCP
      port: 80
  policyTypes:
  - Ingress
```

Explanation:

1. podSelector: This policy applies to Pods with the label app=my-app.
2. ingress: Specifies that only traffic on port 80 (HTTP) is allowed.
3. policyTypes: The policy is applied to ingress traffic (incoming traffic).

By applying this NetworkPolicy, Pods labeled app=my-app will only be able to accept traffic on port 80 (HTTP). Any other traffic is blocked.

5. Egress Example: Allowing Outbound Traffic

Next, let's look at a NetworkPolicy that allows a Pod to send outbound traffic only to a specific IP range.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-egress-to-db
spec:
  podSelector:
    matchLabels:
      app: my-app
  egress:
  - to:
```

```
  - ipBlock:
      cidr: 192.168.0.0/16
 policyTypes:
 - Egress
```

Explanation:

- podSelector: This policy applies to Pods with the label app=my-app.
- egress: Specifies that traffic from the Pod is allowed only to the IP range 192.168.0.0/16.
- policyTypes: The policy is applied to egress traffic (outgoing traffic).

This policy ensures that Pods labeled app=my-app can only initiate connections to IPs within the range 192.168.0.0/16. Any other outbound traffic will be blocked.

6. Combining Ingress and Egress Policies

You can combine both ingress and egress rules in a single NetworkPolicy to control both inbound and outbound traffic for a set of Pods.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-http-and-db-traffic
spec:
  podSelector:
    matchLabels:
      app: my-app
  ingress:
  - ports:
    - protocol: TCP
      port: 80
  egress:
  - to:
    - ipBlock:
        cidr: 192.168.0.0/16
  policyTypes:
  - Ingress
  - Egress
```

This policy:

- Allows HTTP inbound traffic on port 80 (ingress).
- Allows outbound traffic to the 192.168.0.0/16 IP range (egress).

7. Default Deny Policy

A default deny policy can be created by not defining any ingress or egress rules in the policy. This implicitly denies all traffic by default. You can create a NetworkPolicy that denies all traffic to Pods by not specifying any rules.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all-traffic
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  - Egress
```

Explanation:

- The empty podSelector: {} applies the policy to all Pods in the namespace.
- No ingress or egress rules are defined, meaning all traffic (both inbound and outbound) is blocked.
- This policy effectively blocks all traffic to and from the Pods unless other NetworkPolicies allow specific traffic.

Lab Walkthrough:

Part 1: Deploy a Kind Cluster

For this lab, since Docker Desktop does not have a CNI that support network policies, by default, we will quickly create a cluster using KIND. KIND is a utility for setting up a multinode kubernetes cluster on a single node, like on a laptop. You can read more about KIND here.

https://kind.sigs.k8s.io/

Follow the steps below to deploy Kind. Full instructions for different platforms can be found here: https://kind.sigs.k8s.io/docs/user/quick-start

1. For windows, try below and wait for it to finish

   ```
   winget install Kubernetes.kind
   ```

2. Test that kind is properly installed. You may need to create a new terminal to test.

   ```
   kind
   ```

3. Expected output:

```
kind creates and manages local Kubernetes clusters using Docker container 'nodes'
Usage:
  kind [command]

Available Commands:
```

```
  build       Build one of [node-image]
  completion  Output shell completion code for the specified shell (bash, zsh or fish)
  create      Creates one of [cluster]
  delete      Deletes one of [cluster]
  export      Exports one of [kubeconfig, logs]
  get         Gets one of [clusters, nodes, kubeconfig]
  help        Help about any command
  load        Loads images into nodes
  version     Prints the kind CLI version

Flags:
  -h, --help              help for kind
      --loglevel string   DEPRECATED: see -v instead
  -q, --quiet             silence all stderr output
  -v, --verbosity int32   info log verbosity, higher value produces more output
      --version           version for kind

Use "kind [command] --help" for more information about a command.
```

3. Create a file called config.yaml with the following content. Take note that the CNI is disabled.

```
#kind-config.yaml
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
- role: worker
- role: worker
networking:
  disableDefaultCNI: true
```

4. Create a cluster with the file

```
kind create cluster --config config.yaml
```

5. Check the cluster nodes

```
kubectl get nodes
```

Expected output: Take note that the pods will be 'NOT READY' until the CNI is properly configured below.

```
NAME                 STATUS   ROLES           AGE   VERSION
kind-control-plane   Not Ready   control-plane   40s   v1.25.2
kind-worker          Not Ready   <none>          40s   v1.25.2
kind-worker2         not Ready   <none>          40s   v1.25.2
```

6. Cilium supports Network Policies. Install Cilium, for your platform (e.g windows) by downloading it from here https://github.com/cilium/cilium-cli/releases

7. Install cillium. Double click it and locate the Cilium binary.

8. Install cilium CNI on the KIND cluster. It will detect kind cluster automatically.

```
cilium install
```

9. Test that Cilium is properly installed. You need to wait a few minutes for cilium to be ready.

```
cilium status --wait
```

Expected output:

```
 /¯¯\
/¯¯\__/¯¯\    Cilium:          OK
\__/¯¯\__/    Operator:        OK
/¯¯\__/¯¯\    Envoy DaemonSet: OK
\__/¯¯\__/    Hubble Relay:    disabled
   \__/       ClusterMesh:     disabled

DaemonSet              cilium                   Desired: 3, Ready: 3/3, Available: 3/3
DaemonSet              cilium-envoy             Desired: 3, Ready: 3/3, Available: 3/3
Deployment             cilium-operator          Desired: 1, Ready: 1/1, Available: 1/1
Containers:            cilium                   Running: 3
                       cilium-envoy             Running: 3
                       cilium-operator          Running: 1
                       clustermesh-apiserver
                       hubble-relay
Cluster Pods:          14/14 managed by Cilium
Helm chart version:    1.17.0
Image versions         cilium
quay.io/cilium/cilium:v1.17.0@sha256:51f21bdd003c3975b5aaaf41bd21aee23cc08f44efaa27ef
fc91c621bc9d8b1d: 3
                       cilium-envoy
quay.io/cilium/cilium-envoy:v1.31.5-1737535524-fe8efeb16a7d233bffd05af9ea53599340d3f18
e@sha256:57a3aa6355a3223da360395e3a109802867ff635cb852aa0afe03ec7bf04e545: 3
                       cilium-operator
quay.io/cilium/operator-generic:v1.17.0@sha256:1ce5a5a287166fc70b6a5ced3990aaa442496
242d1d4930b5a3125e44cccdca8: 1
```

10. Check that your cluster nodes are now ready:

```
kubectl get nodes
```

Expected output: Note that the nodes are now 'READY'

```
NAME                 STATUS   ROLES           AGE   VERSION
kind-control-plane   Ready    control-plane   2m   v1.25.2
kind-worker          Ready    <none>          2m   v1.25.2
kind-worker2         Ready    <none>          2m   v1.25.2
```

11. Congratulations, you now have a multinode cluster created with KIND with Cilium CNI enabled for network policies to work.

Part 2: Network Policies

1. Create pod1 for the lab

```
kubectl run pod1 --image nginx:latest -l app=pod1
```

Expected output:

```
pod/pod1 created
```

2. Create pod2 for the lab

```
kubectl run pod2 --image nginx:latest -l app=pod2
```

Expected output:
pod/pod1 created

3. See the details of your pod

```
kubectl get pods -o wide
```

Expected output: (yours may differ)

```
NAME   READY   STATUS    RESTARTS   AGE   IP            NODE      NOMINATED NODE
READINESS GATES
pod1   1/1     Running   0          59s   10.244.110.50   docker-desktop   <none>           <none>
pod2   1/1     Running   0          57s   10.244.110.80   docker-desktop   <none>           <none>
```

4. Before creating the network policies, test that this works

```
kubectl exec -it pod1 -- curl 10.244.110.80 --max-time 1
```

Expected output:

```
<!DOCTYPE html>

<html>

<head>

<title>Welcome to nginx!</title>
```

5. Create Network Policy

Create a file called netpol1.yaml with the following content:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      app: pod2
  policyTypes:
    - Ingress
    - Egress
```

6. Create the network policy from the file

```
kubectl apply -f netpol1.yaml
```

Expected output:
networkpolicy.networking.k8s.io/network-policy created

7. Test if the network policy is working. The policy above blocked all traffic to pod1 so you should not have access to the Nginx server.

```
kubectl exec -it pod1 -- curl 10.244.110.80 --max-time 1
```

Expected output.
curl: (28) Connection timed out after 1001 milliseconds

command terminated with exit code 28

8. Let's open the communication between pod1 and pod2

Create a file called netpol2.yaml and paste the following content.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      app: pod2
  policyTypes:
    - Ingress
    - Egress
  ingress:
    - from:
      - podSelector:
          matchLabels:
            app: pod1
  egress:
    - to:
      - podSelector:
          matchLabels:
            app: pod1
```

9. Create the network policy

```
kubectl apply -f netpol2.yaml
```

Expected output.
networkpolicy.networking.k8s.io/network-policy created

10. Test that the policy works. Now you should have access to the nginx as if there were no net policy.

```
kubectl exec -it pod1 -- curl 10.244.110.80 --max-time 1
```

Expected output:
<!DOCTYPE html>

<html>

<head>

<title>Welcome to nginx!</title>

11. Let's test if another pod can access nginx in pod1. Create another pod and test as follows.

```
kubectl run pod3 --image nginx:latest
```

12. Test from the new pod, pod3

```
kubectl exec -it pod3 -- curl 10.244.110.80 --max-time 1
```

Expected output.
curl: (28) Connection timed out after 1001 milliseconds

command terminated with exit code 28

Congratulations: Lab Ends here

Relevant Network Policy Code Snippets

1. PodSelector

```
podSelector:
  matchLabels:
    app: app1
```

2. namespaceSelector

```
namespaceSelector:
  matchLabels:
    app: app1
```

3. ipBlock

```
ipBlock:
  cidr: 10.0.0.0/24
```

4. Multiple selectors with OR

```
ingress:
  - from:
      - namespaceSelector:
          matchLabels:
            app: demo
      - podSelector:
```

```
      matchLabels:
        app: demo-api
```

5. Multiple selectors with AND

```
ingress:
  - from:
      - namespaceSelector:
          matchLabels:
            app: demo
        podSelector:
          matchLabels:
            app: demo-api
```

6. AllowedPorts range

```
ingress:
  - from:
      - podSelector:
          matchLabels:
            app: demo
    ports:
      - protocol: TCP
        port: 1000
        endPort: 3000
```

7. Default Deny all traffic to a pod

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: network-policy
spec:
  podSelector:
    matchLabels:
      app: demo
  policyTypes:
    - Ingress
    - Egress
```

8. Default Deny all traffic to all pods

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: network-policy
spec:
  podSelector: {}
  policyTypes:
```

```
  - Ingress
  - Egress
```

9. Default Deny all ingress traffic

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
 name: network-policy
spec:
 podSelector: {}
 policyTypes:
  - Ingress
```

10. Default Deny all egress traffic

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
 name: network-policy
spec:
 podSelector: {}
 policyTypes:
  - Egress
```

11. Allow all traffic to a Pod

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
 name: network-policy
spec:
 podSelector:
  matchLabels:
   app: demo
 policyTypes:
  - Ingress
  - Egress
 ingress:
  - {}
 egress:
  - {}
```

8. Best Practices for NetworkPolicies

- Start with a Default Deny: Begin by creating a default deny policy to block all traffic. Then, incrementally allow only the traffic that is required.
- Apply NetworkPolicies Early: Apply policies early in the deployment cycle to avoid security holes.
- Label Pods Consistently: Use consistent labels on Pods to easily manage and target them with NetworkPolicies.
- Keep NetworkPolicies Simple: Start with simple, clear rules. Complex policies can be harder to debug and maintain.

- Use Namespace Selectors: For cross-namespace communication, use namespaceSelector to control which namespaces can communicate with each other.

9. Conclusion

NetworkPolicies are a critical tool in securing Kubernetes clusters by controlling traffic flow between Pods and other network endpoints. They enable you to implement the least privilege model by restricting unnecessary communication and reducing potential attack surfaces. By defining ingress and egress rules, you can manage communication between Pods, external services, and the network.

Key takeaways:

- NetworkPolicies control traffic flow between Pods.
- PodSelectors and NamespaceSelectors define which Pods are affected.
- You can control both ingress (incoming) and egress (outgoing) traffic.
- Use NetworkPolicies to minimize the attack surface and ensure secure communication.

Domain 4.2: Understand Authentication, Authorization, and Admission Control

As a Kubernetes Application Developer (CKAD), it's crucial to understand the foundational concepts of authentication, authorization, and admission control in Kubernetes. These are core components of Kubernetes' security architecture, ensuring that only authorized users and processes can access the cluster and that resources are deployed in a secure and controlled manner. Let's break down these concepts in the context of Kubernetes.

1. Authentication in Kubernetes

Authentication is the process of verifying the identity of a user, application, or service trying to access the Kubernetes API. Kubernetes supports several methods of authentication:
a) Common Authentication Methods in Kubernetes

- Certificates: Kubernetes can authenticate clients via TLS client certificates. When a user or service makes a request to the API server, the client presents a certificate, and the server verifies it.
- Bearer Tokens: A bearer token is typically used in service-to-service communication. When a client makes a request, the bearer token in the request header is validated by the API server.
- OpenID Connect (OIDC): Kubernetes can be integrated with external identity providers (e.g., Google, Azure AD) via OIDC for authentication. This allows you to leverage existing user directories to authenticate users.
- Static Password Files: Kubernetes also supports simple static password files for user authentication. This method is less secure but still available for basic setups.
- Webhook: Kubernetes can authenticate users via external systems through a webhook mechanism. When a request is made, the API server can send a request to an external authentication service, which can verify the identity and respond with a success or failure status.

b) How Authentication Works in Kubernetes When a user or service interacts with the Kubernetes API, the kube-apiserver is responsible for authenticating the client. Once authenticated, the request proceeds to authorization (covered next).

2. Authorization in Kubernetes

Authorization determines whether an authenticated user has the appropriate permissions to perform a specific action on a resource. Kubernetes uses Role-Based Access Control (RBAC) as its primary authorization mechanism.

a) Key Authorization Concepts in Kubernetes

- Role-Based Access Control (RBAC): RBAC allows Kubernetes administrators to define policies that specify what actions are allowed or denied on resources based on roles. RBAC has the following core components:
    - Role: A set of permissions within a specific namespace. A Role grants access to resources within that namespace.
    - ClusterRole: A set of permissions that can be applied across all namespaces or cluster-wide.
    - RoleBinding: A binding that assigns a Role to a user, group, or service account within a namespace.

- ClusterRoleBinding: A binding that assigns a ClusterRole to a user, group, or service account across the entire cluster.

Example RBAC Role:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: read-only
rules:
  - apiGroups: [""]
    resources: ["pods"]
    verbs: ["get", "list"]
```

- ServiceAccount: Kubernetes allows workloads (like Pods) to authenticate as a ServiceAccount. A ServiceAccount is associated with specific RBAC roles, determining its permissions.
- ABAC (Attribute-Based Access Control): Although Kubernetes mainly uses RBAC for authorization, it also supports ABAC, which allows decisions based on attributes such as user attributes, resource attributes, etc. ABAC is less commonly used compared to RBAC.

b) How Authorization Works in Kubernetes After authentication, the kube-apiserver checks whether the authenticated identity is authorized to perform the requested operation. If authorized, the request proceeds to the next stage; if not, the request is denied with an HTTP 403 Forbidden response.

3. Admission Control in Kubernetes

Admission Control is a set of plugins that govern and enforce policies on requests to the Kubernetes API server. These plugins operate after authentication and authorization but before the API server writes objects to etcd. Admission control is essential for enforcing security policies, ensuring that resources conform to organizational standards, and preventing misconfigurations or harmful deployments.
a) Common Admission Controllers in Kubernetes

- NamespaceLifecycle: Ensures that the creation or deletion of resources occurs only within active namespaces.
- LimitRanger: Enforces resource limits (like CPU or memory) for pods and containers.
- ServiceAccount: Ensures that every Pod is associated with a ServiceAccount, and that ServiceAccount is properly configured.
- PodSecurityPolicy (PSP): This deprecated controller enforces security policies at the Pod level, such as restricting privileged containers or enforcing security contexts.
- ValidatingAdmissionWebhook: Allows integration with external admission control systems to validate requests dynamically.
- MutatingAdmissionWebhook: Allows modification of Kubernetes objects before they are persisted (e.g., adding default labels or annotations).

- AlwaysPullImages: Enforces that every Pod in a namespace pulls images from a remote registry rather than using locally cached images, helping prevent the use of outdated images.

b) How Admission Control Works in Kubernetes

- After authentication and authorization checks, the admission controllers evaluate the request and can:
    - Allow the request to be processed by the API server.
    - Reject the request with an error message (e.g., if a resource request exceeds limits or violates a policy).
    - Mutate the request by modifying it before saving to the cluster (e.g., adding labels or annotations).

For example, the LimitRanger admission controller ensures that Pods can only request resources within a predefined range. If a user attempts to create a Pod requesting more than the allowed resources, the controller will reject the request.

4. Example Workflow: Authentication, Authorization, and Admission Control

1. Authentication: A user submits a request to the Kubernetes API to create a Pod.
The API server verifies the user's identity (e.g., using certificates or tokens). If the authentication is successful, the request proceeds to the authorization phase.

2. Authorization: Kubernetes checks if the user has the correct permissions to create a Pod.
If the user has the necessary permissions (via RBAC or other methods), the request moves to admission control. If the user lacks permissions, they are denied with a 403 Forbidden error.

3. Admission Control: The request to create the Pod is then evaluated by admission controllers.
If the Pod spec is valid and compliant with policies (such as resource limits), the request is allowed. If there are any violations (e.g., exceeding resource limits), the request is rejected.

5. Practical Steps to Configure Authentication, Authorization, and Admission Control

a) Enabling and Configuring Authentication
Kubernetes supports multiple authentication methods. When setting up a cluster, you can configure which methods to use (certificates, tokens, OIDC, etc.) in the API server's flags or config file.

b) Configuring RBAC for Authorization
RBAC roles are defined in YAML files. For example, you can create roles for users or service accounts and bind them to specific resources with RoleBinding and ClusterRoleBinding objects.

Example of a RoleBinding:

```
apiVersion: rbac.authorization.k8s.io/v1
```

```
kind: RoleBinding
metadata:
  name: read-only-binding
  namespace: default
subjects:
  - kind: User
    name: "johndoe"
    apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: read-only
  apiGroup: rbac.authorization.k8s.io
```

c) Setting Up Admission Controllers

Admission controllers are enabled via the --enable-admission-plugins flag in the Kubernetes API server configuration. Some admission controllers come pre-enabled by default, but you may need to enable others manually, depending on your needs. For example, enabling the PodSecurityPolicy (deprecated) admission controller:

```
kube-apiserver --enable-admission-plugins=PodSecurityPolicy
```

d) Viewing and Debugging Authentication and Authorization
You can view the roles and bindings in the cluster using:

```
kubectl get roles,rolebindings --all-namespaces

kubectl get clusterroles,clusterrolebindings
```

You can debug authentication and authorization issues by looking at the API server logs, which often contain detailed error messages.

Lab Walkthrough:

1. Check if RBAC is enabled

```
kubectl api-versions | grep rbac
```

Expected output:
rbac.authorization.k8s.io/v1

For information only: If not enabled, it can be enabled through configuration:

```
kube-apiserver --authorization-mode=RBAC
```

2. Create a Service Account

```
kubectl create serviceaccount demo-user
```

Expected output:
serviceaccount/demo-user created

3. Create an authorization token for your Service Account:

```
TOKEN=$(kubectl create token demo-user)
```

4. Configure kubectl with your Service Account

```
kubectl config set-credentials demo-user --token=$TOKEN
```

Expected output:
User "demo-user" set.

5. Set a new context. Make sure the --cluster is set to the cluster you are using. For example, if you have KIND cluster and Docker-desktop, ensure to use the cluster you are currently using.

```
kubectl config set-context demo-user-context --cluster=docker-desktop --user=demo-user
```

Expected output:
Context "demo-user-context" created.

6. Check your current context so you can switch back. Write this down.

```
kubectl config current-context
```

Expected output. Yours may differ

docker-desktop

7. Switch to the demo-user-context

```
kubectl config use-context demo-user-context
```

Expected output.
Switch to context "demo-user-context".

8. check if you can see pods

```
kubectl get pods
```

Expected output.
Error from server (Forbidden): pods is forbidden: User "system:serviceaccount:default:demo-user" cannot list resource "pods" in API group "" in the namespace "default"

9. Switch back to the default context and test that it's working

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

Expected output. Your output may differ

Switched to context "docker-desktop".

10. Create a role. Put the following in a file and name it role.yaml

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: demo-role
  namespace: default
rules:
  - apiGroups:
      - ""
    resources:
      - pods
    verbs:
      - get
      - list
      - create
      - update
```

11. Create the role

```
kubectl apply -f role.yaml
```

Expected output.

role.rbac.authorization.k8s.io/demo-role created

12. Create a RoleBinding. Put the following in rolebinding.yaml

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: demo-role-binding
  namespace: default
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: demo-role
subjects:
  - namespace: default
    kind: ServiceAccount
    name: demo-user
```

13. Create the rolebinding

```
kubectl apply -f rolebinding.yaml
```

Expected Output:
rolebinding.rbac.authorization.k8s.io/demo-role-binding created

14. Verify your service account has been granted the Role's Permissions

```
kubectl config use-context demo-user-context
```

Expected output:
Switched to context "demo-user-context".

15. Verify that the get pods command now runs successfully:

```
kubectl get pods
```

Expected output: Your output may vary
NAME READY STATUS RESTARTS AGE

nginx 1/1 Running 0 20s

16. Create a pod

```
kubectl run nginx --image=nginx:latest
```

Expected output: Your output may vary
pod/nginx created

17. Test that the pod is running

```
kubectl get pods
```

Expected output: Your output may vary
NAME READY STATUS RESTARTS AGE

nginx 1/1 Running 0 30s

18. delete will not be successful since this is not included in the role that was created.

```
kubectl delete pod nginx.
```

Expected Output:

Error from server (Forbidden): pods "nginx" is forbidden: User "system:serviceaccount:default:demo-user" cannot delete resource "pods" in API group "" in the namespace "default"

19. Switch back to the previous context and user

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

Testing RBAC permissions using kubectl auth can-i

You can test the permissions of the current context:

```
kubectl auth can-i --list --namespace=foo
```

```
kubectl auth can-i get pods --namespace=foo
```

```
kubectl auth can-i get pods
```

```
kubectl auth can-i get pods --all-namespaces
```

Check to see if service account "foo" of namespace "dev" can list pods in the namespace "prod" You must be allowed to use impersonation for the global option "--as"

```
kubectl auth can-i list pods --as=system:serviceaccount:dev:foo -n prod
```

```
kubectl auth can-i delete pods --as=system:serviceaccount:default:demo-user -n default
```

6. Conclusion

As a Kubernetes Application Developer, understanding authentication, authorization, and admission control is crucial to ensuring the security and proper functioning of your applications and the cluster. Here's a quick summary:

- Authentication ensures that only valid users or services can access the cluster.
- Authorization determines what an authenticated user or service is allowed to do, based on roles and permissions.
- Admission Control enforces policies and validates or modifies requests before they are applied to the cluster.

Together, these three mechanisms help secure the Kubernetes ecosystem by ensuring that only authorized users and processes can modify resources, while also enforcing best practices and organizational policies.

Domain 3.2: Implement Probes and Health Checks

In Kubernetes, probes and health checks are essential components for maintaining the health and availability of applications running in your clusters. These mechanisms help Kubernetes determine whether an application is running correctly, and they play a vital role in ensuring high availability and reliability. As a Kubernetes Application Developer (CKAD), understanding and properly implementing probes and health checks in your application deployment is crucial for creating resilient applications that self-heal and remain stable under varying conditions.

1. What Are Probes and Health Checks in Kubernetes?

Kubernetes uses probes to periodically check the health of the containers running in Pods. If a container fails a health check, Kubernetes can automatically restart or replace it, ensuring minimal downtime and reducing the likelihood of failure. There are three primary types of health checks (or probes) in Kubernetes:

Startup Probe: The startup probe checks whether the application within the container has started properly. This probe is useful when containers require significant startup time. If the startup probe fails, Kubernetes will kill the container, assuming it has not started successfully. This typically has to be successful before Liveness and Readiness Probes are used. This is useful for applications that require time to start, especially legacy applications.

Liveness Probe: This probe checks whether the container is still running. If the liveness probe fails, Kubernetes will kill the container and restart it. This is useful for situations where the container is alive but stuck in a non-functional state.

Readiness Probe: This probe checks if the container is ready to serve traffic. A failing readiness probe will prevent traffic from being routed to the container, but it will not kill it. This is useful during startup or when the application is not yet ready to handle requests. This probe works with the service object. A pod that fails the Readiness probe will be removed from the service endpoint.

Choose the right probe for the right purpose:

- Use readiness probes for containers that need time to initialize before receiving traffic.
- Use liveness probes to detect and restart containers that become unresponsive.
- Use startup probes to detect if containers take longer to start than expected

2. Why Are Probes Important?

1. **Self-Healing:** Probes allow Kubernetes to automatically restart failed containers, ensuring that your application remains available even when something goes wrong.
2. **Service Discovery:** By using readiness probes, Kubernetes ensures that only healthy and ready containers receive traffic, avoiding sending requests to containers that are not yet prepared to serve.

3. **Resource Efficiency:** Liveness probes help in identifying containers that are stuck or in a failed state, ensuring resources are not wasted on containers that are not functioning properly.
4. **Improved User Experience:** By ensuring that only healthy containers serve traffic, probes help avoid situations where users may experience downtime or degraded performance.

3. Health Check Methods

Health checks can be implemented in several ways:

- HTTP-based Health Checks: Kubernetes sends HTTP requests to a specified path (such as /healthz or /readiness) and expects a 200 OK response to determine if the application is healthy.

- TCP Socket Checks: Kubernetes can check if a specific port is open and accepting connections. If a connection can be established, the container is considered healthy.

- Command-based Health Checks: Kubernetes executes a specified command inside the container. If the command exits with a status of 0, the container is considered healthy. Otherwise, it's considered failed.

Example of Command-based Health Check:

```
livenessProbe:
  exec:
    command:
      - "/bin/sh"
      - "-c"
      - "cat /tmp/healthy"
  initialDelaySeconds: 5
  periodSeconds: 10
```

The command returns 0 if the file /tmp/healthy exists, which indicates the container is healthy.

4. Types of Health Checks (Probes)

1. Liveness Probe

The liveness probe determines whether your application is still running and healthy. If it fails, Kubernetes will restart the container.

Use cases:

1. If an application gets stuck and cannot recover on its own (e.g., a deadlock or infinite loop).
2. If the application is non-responsive but can still be running.

How it works:

1. Kubernetes will periodically call the endpoint you specify (e.g., an HTTP endpoint) to check if the application is healthy. If the probe fails,
2. Kubernetes restarts the container.

This pod will work because nginx's root directory / is created whenever nginx is started. If you modify the path to something that does not exist, like say /error.html, the test will fail, and the pod will keep restarting. You can describe the pod to see what it says. You will see something like:

Example of Liveness Probe:
File: liveness.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
spec:
  containers:
  - name: example-container
    image: nginx
    livenessProbe:
      httpGet:
        path: /
        port: 80
      initialDelaySeconds: 5
      periodSeconds: 10
      failureThreshold: 3
```

Parameters:

1. initialDelaySeconds: How long to wait before performing the first probe.
2. periodSeconds: How often to run the probe.
3. failureThreshold: The number of consecutive failures before considering the container unhealthy.

2. Readiness Probe

The readiness probe determines whether the application inside the container is ready to serve traffic. Unlike the liveness probe, if the readiness probe fails, the container remains running but will not receive traffic until it passes.

Use cases:

1. If your application needs time to initialize (e.g., loading large datasets, establishing connections, etc.).
2. If your application becomes temporarily unresponsive due to high load or resource constraints.

This pod will work because nginx's root directory /index.html is created whenever nginx is started and ready. If you modify the path to something that does not exists, like say /error.html, the test will fail and the pod will not come on. You can describe the pod to see what it says. You will see something like:

Readiness probe failed: HTTP probe failed with statuscode: 404

Example of Readiness Probe:

File: readiness.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: readiness-httpget
spec:
  containers:
  - name: readiness-httpget
    image: nginx:latest
    readinessProbe:
      httpGet:
        path: /index.html
        port: 80
      initialDelaySeconds: 5
      periodSeconds: 10
      failureThreshold: 3
```

This pod will work because nginx runs on port 80 so the tcpsockert test will pass. If you modify the tcpSocker port to say 8080, the test will fail and the pod will never come online. You can describe the pod to see what it says. You will see something like: Readiness probe failed: dial tcp 10.1.5.1:8080: connect: connection refused

File: pod-readiness-tcp.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: readiness-tcpsocket
spec:
  containers:
  - name: readiness-tcpsocket
    image: nginx
    ports:
    - containerPort: 80
    readinessProbe:
      tcpSocket:
        port: 80
      initialDelaySeconds: 15
      periodSeconds: 10
```

Combine both

File: both.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
```

```
  name: combine-readiness-liveness
spec:
  replicas: 1
  selector:
    matchLabels:
      app: both
  template:
    metadata:
      labels:
        app: both
    spec:
      containers:
      - name: both-container
        image: nginx:latest
        ports:
        - containerPort: 80
        readinessProbe:
          httpGet:
            path: /index.html
            port: 80
            scheme: HTTP
          initialDelaySeconds: 5
          periodSeconds: 10
          timeoutSeconds: 5
        livenessProbe:
          httpGet:
            path: /
            port: 80
            scheme: HTTP
          initialDelaySeconds: 5
          periodSeconds: 10
          timeoutSeconds: 5
```

3. Startup Probe

The startup probe helps in scenarios where your container may take time to start (e.g., loading large configurations or running initial setup tasks). If the startup probe fails, Kubernetes assumes the container has failed to start and will kill it.

Use cases:

1. When the container takes a long time to start (e.g., databases, large applications).
2. When the application needs time to establish external dependencies, like database connections or external API calls.

Example of Startup Probe:

File: startup.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
spec:
  containers:
  - name: example-container
```

```
image: my-app:latest
startupProbe:
  httpGet:
    path: /startup
    port: 8080
  initialDelaySeconds: 10
  periodSeconds: 5
  failureThreshold: 5
```

Parameters:

1. initialDelaySeconds: How long to wait before performing the first probe.
2. failureThreshold: Number of failures before considering the container as failed.

5. Best Practices for Health Checks

Set appropriate initialDelaySeconds: Allow the application enough time to start and initialize before the probes begin. Setting an initial delay too short can result in false failures.
Choose the right probe for the right purpose:

- Use readiness probes for containers that need time to initialize before receiving traffic.
- Use liveness probes to detect and restart containers that become unresponsive.
- Use startup probes to detect if containers take longer to start than expected.
- Monitor probe results: Use Kubernetes' monitoring and logging tools to track the success or failure of health probes. This helps in identifying issues early.
- Test probes locally: Before deploying to Kubernetes, test your health check endpoints locally to ensure they work as expected.
- Avoid over-tightening probe thresholds: Set reasonable thresholds for failures and delays to prevent unnecessary restarts or blocking of traffic. Too aggressive health checks can cause unnecessary container restarts.

6. Conclusion

Implementing probes and health checks is essential for maintaining the health and availability of your applications in Kubernetes. As a Kubernetes Application Developer (CKAD), you must know how to use liveness, readiness, and startup probes effectively to ensure your applications remain resilient, self-healing, and able to serve traffic without downtime.

By correctly setting up health checks:

1. Your containers will be monitored and restarted automatically when necessary.
2. Kubernetes will route traffic only to ready containers.
3. You'll be able to identify and resolve issues early, improving your application's overall reliability.
4. Always test your probes, monitor the results, and adjust them based on the specific needs of your application!

Domain 4.3: Understand Requests, Limits, and Quotas

In Kubernetes, managing resources effectively is critical for ensuring the stability, performance, and efficiency of applications running in the cluster. As a Kubernetes Application Developer (CKAD), it is essential to understand

- resource requests,
- limits, and
- quotas,

as these directly influence how workloads (Pods and containers) are

- scheduled,
- executed, and
- constrained within the cluster.

This section will cover the core concepts related to resource management and demonstrate how you can define and configure resource requests, limits, and quotas for containers and pods to ensure that your applications are properly allocated resources.

1. Resource Requests and Limits

In Kubernetes, resources like CPU and memory (RAM) are allocated to containers within a Pod. Each container in a Pod can specify requests and limits for these resources.

a) Resource Requests

A request is the amount of a particular resource that the container is guaranteed to receive when it is scheduled onto a node. When a container is scheduled, Kubernetes ensures that enough resources are available on the node to meet the requested amount.

- CPU Requests:

The amount of CPU requested by the container. Kubernetes will schedule the container onto a node only if that node has enough available CPU.

- Memory Requests:

The amount of memory the container requests. Similarly, the container will only be scheduled onto a node if there is sufficient memory available.

Example:
File: requests.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: resource-request-example
spec:
  containers:
  - name: my-container
    image: my-image
```

```
  resources:
    requests:
      memory: "64Mi"
      cpu: "250m"
```

In this example, the container requests 64 MB of memory and 250 milli-CPU (0.25 CPU core).

Understanding CPU units

The CPU resource is measured in *CPU* units. One CPU, in Kubernetes, is equivalent to:

- 1 AWS vCPU
- 1 GCP Core
- 1 Azure vCore
- 1 Hyperthread on a bare-metal Intel processor with Hyperthreading

Fractional values are allowed. A Container that requests 0.5 CPU is guaranteed half as much CPU as a Container that requests 1 CPU. You can use the suffix m to mean milli. For example 100m CPU, 100 milliCPU, and 0.1 CPU are all the same. Precision finer than 1m is not allowed.

CPU is always requested as an absolute quantity, never as a relative quantity; 0.1 is the same amount of CPU on a single-core, dual-core, or 48-core machine.

b) Resource Limits

A limit is the maximum amount of a resource that a container can use. If a container tries to exceed its specified limit, Kubernetes may throttle the container's CPU usage or kill the container if it exceeds its memory limit (and it will be restarted, depending on the restart policy).

- CPU Limits: If a container tries to exceed the CPU limit, Kubernetes will throttle the container's CPU usage to prevent it from consuming too much.
- Memory Limits: If a container exceeds its memory limit, Kubernetes will terminate the container and restart it (OOMKill), as memory consumption is a critical resource for the node.

Example:
File: limits.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: resource-limit-example
spec:
  containers:
  - name: my-container
    image: my-image
    resources:
      limits:
        memory: "128Mi"
```

```
    cpu: "500m"
```

In this example, the container can use up to 128 MB of memory and 500 milli-CPU (0.5 CPU core), but it is not allowed to consume more than that.

c) Requests vs. Limits:

- Requests ensure that the container has the required resources to run.
- Limits ensure that the container cannot use more resources than the specified maximum, preventing one container from starving other containers on the same node.

Important: If you don't specify limits, containers can use as much resource as is available on the node, which could lead to resource contention or instability.

2. Resource Quotas

Resource Quotas are a way to limit the amount of resources that can be consumed by a specific namespace within a Kubernetes cluster. This allows administrators to enforce policies that prevent one team or application from consuming all the available resources in the cluster.
A ResourceQuota object can define limits on resources like CPU, memory, number of Pods, number of Persistent Volume Claims (PVCs), and more.

a) Common ResourceQuota Constraints

- CPU: Limits the total CPU usage in the namespace.
- Memory: Limits the total memory usage in the namespace.
- Pods: Limits the number of Pods that can be created within the namespace.
- PersistentVolumeClaims: Limits the number of Persistent Volume Claims that can be created.

b) Example of a ResourceQuota

File: quota.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: example-quota
  namespace: my-namespace
spec:
  hard:
    requests.cpu: "4"
    requests.memory: "8Gi"
    limits.cpu: "8"
    limits.memory: "16Gi"
    pods: "10"
    persistentvolumeclaims: "5"
```

In this example:

- The namespace my-namespace can request up to 4 CPUs and 8 GiB of memory in total.
- The namespace is allowed to create up to 10 Pods and 5 PersistentVolumeClaims.
- The maximum limits for CPU and memory are set at 8 CPUs and 16 GiB of memory.

c) ResourceQuota and Namespace Limits

When a ResourceQuota is defined for a namespace, Kubernetes will enforce the limits on the total resource usage for that namespace. This means if you exceed the quota, Kubernetes will prevent new resources from being created until resources are freed up or the quota is adjusted.

3. Best Practices for Resource Requests, Limits, and Quotas

As a Kubernetes Application Developer, understanding how to configure requests, limits, and quotas is essential for the stability and efficiency of your applications. Here are some best practices:

a) Always Define Requests and Limits

Always specify both requests and limits for containers. If only one is defined (e.g., only a request but no limit), it can lead to unexpected behaviors, such as excessive resource consumption or throttling.

Requests should reflect the minimum resources necessary for your application to run, while limits should reflect the maximum acceptable resources your application should consume.

b) Monitor Resource Usage
Continuously monitor your containers and Pods for resource usage to ensure that you are not over- or under-allocating resources. Use Kubernetes tools like kubectl top to view resource usage in your cluster. Example:

```
kubectl top pod
```

c) Optimize Resource Requests

Setting resource requests too high can lead to overprovisioning, meaning your cluster may run out of resources. On the other hand, setting requests too low may lead to performance issues. Use monitoring tools like Prometheus to track resource usage and adjust requests and limits accordingly.

d) Use Resource Quotas to Control Resource Usage
Enforce ResourceQuotas to ensure that no single team or application uses more than its fair share of resources. For multi-tenant clusters, use ResourceQuotas to avoid resource contention between teams or applications.

4. How to View and Manage Resources

a) View Resources in the Cluster
You can check the resource requests and limits of containers and Pods using kubectl describe:

```
kubectl describe pod <pod-name>
```

This command will provide detailed information about the resource requests and limits for each container in the Pod.

b) View Resource Quotas in a Namespace
You can view the existing ResourceQuota objects in a specific namespace using the following command:

```
kubectl get resourcequota -n <namespace>
```

This will display the resource quotas in the specified namespace, along with the usage and available resources.

5. Troubleshooting Resource Issues

When resource issues arise, such as Pods being OOMKilled (Out of Memory) or Pods being throttled due to CPU limits, you can troubleshoot these problems with the following steps:

a) Check Resource Usage
Use kubectl top pod to monitor how much CPU and memory your Pods are using.

```
kubectl top pod
```

b) Check Pod Events
If Pods are being killed due to resource constraints, you can inspect their events to get more details:

```
kubectl describe pod <pod-name>
```

Look for events such as "OOMKilled" or "Memory Pressure".

c) Adjust Resource Requests and Limits
If your application is resource-intensive, consider increasing the resource requests and limits. Conversely, if your application is using fewer resources than expected, you can scale back the requests and limits.

6. Conclusion

Understanding resource requests, limits, and quotas is crucial for maintaining a healthy, efficient, and stable Kubernetes environment. As a Kubernetes Application Developer (CKAD), your role is to ensure that resources are allocated properly to your workloads, avoiding over- or under-provisioning.

To summarize:

- Requests: The minimum amount of resources that a container is guaranteed to receive.
- Limits: The maximum amount of resources a container can use.
- Resource Quotas: Limits on resources within a namespace to prevent one team or workload from consuming excessive resources.

By carefully managing these resources, you help ensure that Kubernetes clusters remain efficient, responsive, and predictable, while also preventing potential issues related to resource contention.

Domain 4.4: Understand ConfigMaps

In Kubernetes, ConfigMaps provide a way to manage configuration data for applications in a centralized and decoupled manner. As an Application Developer (CKAD), understanding how to use ConfigMaps is essential for managing configuration in a flexible and environment-agnostic way.

What is a ConfigMap?

A ConfigMap is an API object in Kubernetes that allows you to store configuration data in key-value pairs. You can then inject this data into your containers as:

- environment variables,
- command-line arguments, or
- configuration files.

By using ConfigMaps, you can separate configuration from your application code, making your applications more portable and easier to manage.
ConfigMaps are useful for storing configuration data that might change based on the environment (e.g., development, staging, production) but should remain consistent across deployments.
1. Creating a ConfigMap There are several ways to create a ConfigMap in Kubernetes:

a) Using kubectl from a File If you have a file containing configuration data, you can create a ConfigMap from that file using the kubectl create configmap command.

Example:
kubectl create configmap my-config --from-file=config.txt

This command creates a ConfigMap named my-config using the contents of config.txt.

b) Using kubectl from Literal Values You can create a ConfigMap directly from the command line by specifying key-value pairs. Example:

kubectl create configmap my-config --from-literal=key1=value1 --from-literal=key2=value2

This creates a ConfigMap named my-config with two entries: key1=value1 and key2=value2.

c) Using a YAML Manifest You can define a ConfigMap in a YAML manifest and apply it using kubectl apply. Example ConfigMap YAML:
File: configmaps.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-config
data:
  name: ckad
```

```
  day: thursday
```

Apply it with:
kubectl apply -f configmaps.yaml

2. Using ConfigMaps in Pods Once you've created a ConfigMap, you can inject its data into Pods in several ways:

a) As Environment Variables You can reference the keys from a ConfigMap as environment variables in your containers. Example:

File: cm-env-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-env-example
spec:
  containers:
  - name: my-container
    image: nginx
    envFrom:
    - configMapRef:
        name: my-config
```

In this example, all keys in the my-config ConfigMap are injected into the container as environment variables.
b) As Environment Variables (Individual Keys) If you want to specify individual keys, you can reference them explicitly. Example:

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-env-example
spec:
  containers:
  - name: my-container
    image: nginx
    env:
    - name: KEY1
      valueFrom:
        configMapKeyRef:
          name: my-config
          key: key1
```

In this case, only the value of key1 from the my-config ConfigMap is set as the KEY1 environment variable in the container.

c) As Command-line Arguments You can pass the values from a ConfigMap as command-line arguments to the container's entrypoint. Example:

```
apiVersion: v1
kind: Pod
metadata:
```

```
  name: configmap-args-example
spec:
  containers:
  - name: my-container
    image: nginx
    command:
    - "/bin/sh"
    - "-c"
    - "echo $(KEY1)"
    env:
    - name: KEY1
      valueFrom:
        configMapKeyRef:
          name: my-config
          key: key1
```

Here, the value of key1 in the my-config ConfigMap is passed as a command-line argument to the container, and the container will print it.
d) As Configuration Files You can mount the ConfigMap as a file inside the container. This is useful when the configuration is a set of files or needs to be structured in a specific way. Example:

File: cm-vol-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-volume-example
spec:
  containers:
  - name: my-container
    image: nginx
    volumeMounts:
    - name: config-volume
      mountPath: /etc/config
      readOnly: true
  volumes:
  - name: config-volume
    configMap:
      name: my-config
```

In this case, the my-config ConfigMap will be mounted as files in the /etc/config directory within the container. Each key in the ConfigMap becomes a separate file with the corresponding value as the file content.

3. Updating ConfigMaps You can update a ConfigMap at any time. However, updating a ConfigMap will not automatically trigger a Pod restart. You need to either manually restart the Pods or use strategies like rolling updates to ensure that the new configuration is applied.

a) Updating a ConfigMap Using kubectl You can update a ConfigMap by reapplying the new configuration using kubectl apply.

```
kubectl apply -f cm-vol-pod.yaml
```

b) Triggering Pod Restarts After updating a ConfigMap, you may want to restart the Pods that use the ConfigMap to pick up the new configuration. You can do this by deleting the Pods, and Kubernetes will automatically recreate them.

```
kubectl delete pod <pod-name>
```

Alternatively, if you're using a Deployment, you can trigger a rolling update to apply the new configuration:

```
kubectl rollout restart deployment <deployment-name>
```

Lab Walkthrough

1. Put this information in cm.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: demo-config
data:
  database_host: "192.168.0.1"
  debug_mode: "1"
  log_level: "verbose"
```

2. Create the configmaps

```
kubectl appply -f cm.yaml
```

Expected output:
configmap/demo-config created

3. List the configmaps

```
kubectl get configmaps
```

Expected output: (yours may differ)
NAME DATA AGE

demo-config 3 5m

4. Inspect the confimaps

```
kubectl describe configmap demo-config
```

Expected output:

```
Name:         demo-config
Namespace:    default
Labels:       <none>
Annotations:  <none>
Data
====
database_host:
----
192.168.0.1
debug_mode:
----
1
log_level:
----
verbose

BinaryData
====

Events:  <none>
```

5. Get the information as a json

```
kubectl get configmap demo-config -o jsonpath='{.data}' | jq
```

Expected output:

```
{
  "database_host": "192.168.0.1",
  "debug_mode": "1",
  "log_level": "verbose"
}
```

6. Mount ConfigMaps as ENV. put the following in a file called cm-pod.yaml and apply it

```
apiVersion: v1
kind: Pod
metadata:
  name: demo-pod
spec:
  containers:
    - name: app
      command: ["/bin/sh", "-c", "printenv"]
      image: busybox:latest
      envFrom:
        - configMapRef:
            name: demo-config
```

7. Apply it Expected output:

```
kubectl apply -f cm-pod.yaml
```

Expected output.
pod/demo-pod created

8. check logs

```
kubectl logs pod/demo-pod
```

Expected output.

```
...
database_host=192.168.0.1
debug_mode=1
log_level=verbose
...
```

9. Mount ConfigMaps as Command Line Arguments

Create a file called demo-pod-1.yaml and paste the following content.

```
apiVersion: v1
kind: Pod
metadata:
  name: demo-pod
spec:
  containers:
    - name: app
      command: ["demo-app", "--database-host", "$(DATABASE_HOST)"]
      image: demo-app:latest
      env:
        - name: DATABASE_HOST
          valueFrom:
            configMapKeyRef:
              name: demo-config
              key: database_host
```

10. Mount ConfigMaps as Volumes. save to demo-pod-2.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: demo-pod
spec:
```

```
 containers:
   - name: app
     command: ["ls", "/etc/app-config"]
     image: demo-app:latest
     volumeMounts:
       - name: config
         mountPath: "/etc/app-config"
         readOnly: true
 volumes:
   - name: config
     configMap:
       name: demo-config
```

Expected Output:

pod/demo-pod created

11. Check the values.

```
kubectl logs pod/demo-pod
```

Expected output:
database_host
debug_mode
log_level

4. Best Practices for Using ConfigMaps

As a Kubernetes Application Developer (CKAD), you should consider these best practices when working with ConfigMaps:

a) Use ConfigMaps for Non-sensitive Data ConfigMaps are intended for non-sensitive configuration data (like feature flags, configuration parameters, etc.). For sensitive data (like passwords, API keys), use Secrets instead.

b) Version Your ConfigMaps For better management and to ensure that changes in the configuration are traceable, consider versioning your ConfigMaps. You can append version numbers to the ConfigMap names or store them in a source control system.

c) Use ConfigMap for Environment-Specific Configuration You can use different ConfigMaps for different environments (e.g., config-dev, config-prod) and inject the appropriate one into your Pods based on the environment.

d) Separate Configuration from Code Decouple your application's code from its configuration. By using ConfigMaps, you make your applications more portable, as configurations can be modified without changing the application code.

e) Avoid Hardcoding ConfigMap Names Whenever possible, avoid hardcoding ConfigMap names in your code. Instead, use environment variables or Helm charts to make it easier to manage configurations for multiple environments.

5. Troubleshooting ConfigMap Issues

If your application is not behaving as expected after configuring or updating a ConfigMap, consider these troubleshooting steps:

a) Check ConfigMap Content You can view the content of a ConfigMap using:

```
kubectl describe configmap <configmap-name>
```

b) Check Pod Logs If the container is not behaving as expected, check its logs to see if there are any errors related to the configuration:

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

c) Verify Volume Mounts If you are mounting the ConfigMap as a file, ensure that the volume mount path is correct and that the files are available inside the container.

6. Conclusion

ConfigMaps are a powerful and flexible way to manage configuration data in Kubernetes. As a Kubernetes Application Developer (CKAD), you need to understand how to create, manage, and use ConfigMaps to inject configuration into your applications.

To summarize:

1. ConfigMaps store non-sensitive configuration data as key-value pairs.
2. They can be used in Pods as environment variables, command-line arguments, or mounted as files.

Best practices include versioning, separating configuration from code, and using ConfigMaps for environment-specific settings.

By understanding and using ConfigMaps effectively, you can make your Kubernetes applications more dynamic, flexible, and easier to manage across different environments.

Domain 4.6: Create & Consume Secrets

In Kubernetes, Secrets are a crucial way to store sensitive information such as passwords, API tokens, SSH keys, and other credentials in a secure and encrypted manner. These Secrets can be consumed by Pods and containers to access secure data without hard-coding it into your application code or configuration files.

In this section, we will cover the following topics:

- How to create Kubernetes Secrets.
- How to consume Secrets in a Pod.
- Best practices for handling Secrets securely.

1. What are Kubernetes Secrets?

Kubernetes Secrets store sensitive information, such as:

- Passwords
- OAuth tokens
- SSH keys
- TLS certificates

Secrets are stored in the cluster and can be used by Pods and other resources in the cluster to access sensitive information without exposing it in your configuration files. The key advantage of using Secrets is that they are **encrypted at rest**, and their values can be managed securely.

2. Creating Kubernetes Secrets

Secrets can be created in Kubernetes in several ways:

- manually via kubectl,
- through YAML files,
- or using external tools (e.g., Helm, Kustomize).

a) Create Secrets Using kubectl Command

You can create Secrets directly from the command line using ***kubectl create secret***. There are various ways to create Secrets depending on how you want to provide the secret data.

Example 1: Create a Secret from Literal Values

```
kubectl create secret generic my-secret \
  --from-literal=username=admin \
  --from-literal=password=secretpassword
```

This creates a Secret named my-secret with two keys: ***username and password***, and their respective values.

Example 2: Create a Secret from a File

If you have a file containing sensitive information, you can create a Secret from the file content.

```
kubectl create secret generic my-secret --from-file=my-ssh-key=/path/to/ssh/keyfile
```

This creates a Secret named my-secret with a key my-ssh-key containing the contents of the file /path/to/ssh/keyfile.

Example 3: Create a Secret from Multiple Files

```
kubectl create secret generic my-secret \
  --from-file=ssh-private-key=/path/to/private_key \
  --from-file=ssh-public-key=/path/to/public_key
```

This creates a Secret with two files, ***ssh-private-key*** and ***ssh-public-key***, containing the respective SSH keys.

b) Create Secrets Using YAML
Secrets can also be defined in a YAML manifest. This is useful for version control and automation.
Example:
File: secrets.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
type: Opaque
data:
  username: YWRtaW4=    # Base64 encoded string for 'admin'
  password: c2VjcmV0cGFzc3dvcmQ=    # Base64 encoded string for 'secretpassword'
```

- type: Opaque: This indicates the secret is of a general type and not associated with any specific Kubernetes use case (e.g., Docker registry secrets).
- data: The sensitive data in a Secret must be base64 encoded. In this example, the values for username and password are encoded strings for admin and secretpassword.

To apply the YAML manifest:

```
kubectl apply -f secrets.yaml
```

3. Consuming Secrets in a Pod

Once the Secret is created, it can be consumed by a Pod in two primary ways:

1. Environment Variables
2. Mounted as Volumes

a) Consume Secrets as Environment Variables

You can expose a Secret as an environment variable inside a container. This method is useful when your application needs to access the Secret as a simple environment variable.

Example: Consume a Secret as an environment variable in a Pod
File: secret-env-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-demo
spec:
  containers:
  - name: my-container
    image: nginx
    env:
    - name: SECRET_USERNAME
      valueFrom:
        secretKeyRef:
          name: my-secret
          key: username
    - name: SECRET_PASSWORD
      valueFrom:
        secretKeyRef:
          name: my-secret
          key: password
```

In this example:

1. The username and password keys from the Secret my-secret are injected into the container as environment variables (SECRET_USERNAME and SECRET_PASSWORD).
2. The application inside the container can now access the values through environment variables.

b) Consume Secrets as Mounted Volumes

Secrets can also be mounted as volumes, which allows your application to access them as files within the container. This is useful when you need to pass a file-based Secret (like an SSL certificate, SSH key, etc.).

Example: Consume a Secret as a mounted volume
File: secret-vol-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-volume-demo
spec:
  containers:
  - name: my-container
    image: nginx
    volumeMounts:
    - name: secret-volume
      mountPath: /etc/secrets
  volumes:
```

```
 - name: secret-volume
   secret:
     secretName: my-secret
```

In this example:

1. The Secret my-secret is mounted as a volume at /etc/secrets inside the container.
2. The container will have access to all the keys in the Secret as files under /etc/secrets, e.g., /etc/secrets/username and /etc/secrets/password.

Lab Walkthrough

1. Encode the secrets

```
echo -n 'admin' | base64
echo -n 'password' | base64
```

2. Create the demo-secret.yaml with this content

```
apiVersion: v1
kind: Secret
metadata:
  name: demo-secret
type: Opaque
data:
  username: YWRtaW4=
  password: cGFzc3dvcmQ=
```

or

```
apiVersion: v1
kind: Secret
metadata:
  name: demo-secret
type: Opaque
stringData:
  username: admin
  password: password
```

Apply it:

```
kubectl -n secrets-demo apply -f demo-secret.yaml
```

Expected output:
secret/demo-secret created

3. Mount Secret as Env Variables

File: secret-test-evn-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
```

```
  name: env-pod
spec:
  containers:
    - name: secret-test
      image: nginx
      command: ['sh', '-c', 'echo "Username: $USER" "Password: $PASSWORD"']
      env:
        - name: USER
          valueFrom:
            secretKeyRef:
              name: database-credentials
              key: username.txt
        - name: PASSWORD
          valueFrom:
            secretKeyRef:
              name: database-credentials
              key: password.txt
```

4. Create the file

```
kubectl -n secrets-demo apply -f secret-test-evn-pod.yaml
```

Expected output:
```
secret/demo-secret created
```

5. Describe the secret

```
kubectl -n secrets-demo describe pod env-pod
```

Expected output:

6. Mount the volumes

File: secret-test-volume-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: volume-test-pod
spec:
  containers:
  - name: secret-test
    image: nginx
    volumeMounts:
    - name: secret-volume
      mountPath: /etc/config/secret
  volumes:
  - name: secret-volume
    secret:
      secretName: database-credentials
```

7. Apply it

```
kubectl -n secrets-demo apply -f secret-test-volume-pod.yaml
```

8. check logs

```
kubectl -n secrets-demo logs env-pod
```

9. Describe the secrets

```
kubectl -n secrets-demo describe pod env-pod
```

10. See the volumes

```
kubectl -n secrets-demo exec volume-test-pod -- cat /etc/config/secret/username.txt
```

```
kubectl -n secrets-demo exec volume-test-pod -- cat /etc/config/secret/password.txt
```

```
kubectl -n secrets-demo exec volume-test-pod -- ls /etc/config/secret
```

4. Best Practices for Managing Secrets

- Avoid Hard-coding Secrets: Never hardcode sensitive data like passwords, API tokens, or certificates directly into your code or Kubernetes YAML files.
- Use kubectl to read secrets: Avoid printing Secrets in plain text to the console. Use kubectl get secret my-secret -o yaml or kubectl describe secret my-secret to view Secrets in a secure way.
- Limit Access: Only allow the Pods or services that need to access a particular Secret to do so. Use Role-Based Access Control (RBAC) to control access to Secrets.
- Encrypt Secrets at Rest: Kubernetes supports encrypting Secrets at rest. Ensure that your Kubernetes cluster has this enabled.
- Use External Secrets Management: If you need to integrate with enterprise-grade secrets management solutions, consider using tools like HashiCorp Vault, AWS Secrets Manager, or Azure Key Vault.

- Base64 Encoding Is Not Encryption: Secrets are base64 encoded, not encrypted. Ensure that your Kubernetes cluster uses encryption at rest to protect Secret data.

5. Deleting a Secret

To delete a Secret that is no longer required:

```
kubectl delete secret my-secret
```

This removes the Secret from the cluster. Make sure that any applications or resources depending on it are updated or deleted before removing a Secret.

6. Conclusion

As a Kubernetes Application Developer (CKAD), understanding how to securely create and consume Secrets is vital for building safe and efficient applications. Whether you choose to pass Secrets as environment variables or mount them as volumes, ensuring that they are handled securely and not exposed in your code is crucial for maintaining the confidentiality of sensitive data.

1. Creating Secrets: Use kubectl, YAML manifests, or external tools.
2. Consuming Secrets: Inject them as environment variables or mount them as volumes in your Pods.
3. Best Practices: Implement security measures like encryption at rest, and limit access to Secrets using RBAC.

By mastering these concepts, you can securely manage sensitive data in your Kubernetes clusters and build more secure, reliable applications.

Certified Kubernetes Application Developer (CKAD)

By Damian Igbe, PhD

Day 4

Encrypting with SOPS and Age

With the ever-growing need to safeguard secrets and sensitive data, the challenge is to find tools that offer a balance between security and usability. One such pairing that stands out is Mozilla's SopS in combination with AGE for encryption. In this blog post, we'll dive deep into the benefits of this duo, touching on simplicity, version control, encryption, and even offer a hands-on example.

What is SopS?

SopS is a secret management tool developed by Mozilla. It's designed for encrypting secrets in files that can be stored in a version control system like GIT. Rather than encrypting the entire file, SopS encrypts only the specific parts of a file that contain secrets, leaving the rest of the file (like structure and comments) in plain text. This means that you can still track changes and maintain version history without compromising security. Furthermore, SopS supports a variety of secret backends i.e. cloud KMS and Hashicorp vault. SopS additionally works in a binary mode where full files can be encrypted.

Introducing AGE

AGE (Actually Good Encryption) is a simple, modern and secure encryption tool with small explicit threat models. Built on the foundations of strong cryptographic primitives, AGE focuses on being simple to use while retaining powerful encryption capabilities.

Why Pair SopS with AGE?

1. Simplicity: One of the primary benefits of using AGE is its simplicity. With a straightforward CLI and easy-to-understand encryption and decryption commands, it reduces the chance of human error. When integrated with SopS, you get the combined power of both tools intuitively.
2. Version Control Compatibility: Since SopS encrypts only the secrets and not the entire file, it plays well with version control systems. You can visualize diffs, track changes, and maintain an audit trail of modifications without revealing the actual secrets.
3. Strong Encryption: AGE boasts robust encryption capabilities. It uses modern encryption techniques, ensuring that your data remains confidential and safe from potential breaches.

Code Example: Encryption and Decryption with SopS and AGE.

Note: ensure that sops is installed and the environmental varaiable to the secret key is set. The next section explains how to do this.

First, generate an AGE keypair:

```
age-keygen -o age.key
```

This will produce a public key string that you can use for encryption and a private key saved in age.key for decryption.

Next, use SopS to encrypt a file using the AGE public key:

```
sops --age age_public_key -e secrets.yaml > secrets.enc.yaml
```

Where age_public_key is the public key string generated in the first step and secrets.yaml is your plaintext file.

To decrypt the file, use:

```
sops --age age.key -d secrets.enc.yaml
```

The encrypted file (secrets.enc.yaml) can be safely stored in version control, while the original secrets remain safe and sound.

Lab Walkthrough

Objective

- Use **SOPS** (secrets OperationS) to manage secrets.
- Encrypt secrets using **AGE (Actually Good Encryption)** encryption.
- Store secrets securely in **Kubernetes** and access them via **Kubernetes Secrets**.
- Decrypt secrets dynamically in your Kubernetes deployments.

Prerequisites

1. **Kubernetes** cluster
2. **kubectl** command-line tool installed and configured to interact with your cluster.
3. **SOPS** installed: A tool to manage encrypted secrets files.
4. **AGE** installed: A simple and secure encryption tool.
5. **Helm** installed (optional, for better management).
6. Basic knowledge of Kubernetes.

Step 1: Install sops and age

Install sops

Linux:

```
curl -Lo sops
https://github.com/mozilla/sops/releases/download/v3.7.3/sops-v3.7.3-linux-amd64
chmod +x sops
sudo mv sops /usr/local/bin/
```

Install age

Linux:

```
curl -Lo age.tar.gz
https://github.com/FiloSottile/age/releases/download/v1.0.0/age-linux-amd64.tar.gz
tar -xvf age.tar.gz
chmod +x age
sudo mv age /usr/local/bin/
```

Step 2: Generate an Age Key Pair

AGE encrypts secrets using a public/private key pair. Let's generate the keys.

```
age-keygen -o key.txt
```

This command generates a new key pair and outputs it to key.txt. The public key will be used for encryption, and the private key will be used for decryption.

You will see output like:

```
# created: 2025-03-12T00:00:00Z
# public key: age1xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
<private_key_content>
```

Store the private key securely, as it will be required later for decryption. You can move it to ~/.sops/

Export the key to an ENV and ensure that it works

```
SOPS_AGE_KEY_FILE=~/.sops/key.txt
echo $SOPS_AGE_KEY_FILE
~/.sops/key.txt
```

you can reference here for different environments
https://github.com/getsops/sops

Step 3: Encrypt a Secret Using AGE

Now that you have your public key (from the key.txt file), let's encrypt a secret. For example, let's say you want to encrypt a Kubernetes password.

1. Create a file with the secret, e.g., db-password.txt:

```
echo "mysecretpassword" > db-password.txt
```

2. Encrypt the file using AGE:

```
age -r <public_key> -o db-password.txt.age db-password.txt
```

Where <public_key> is the public key from the key.txt file generated earlier. After running this, the file db-password.txt.age will be the encrypted version of your password.

You can delete the plaintext secret file for security:

```
rm db-password.txt
```

Step 4: Create a Kubernetes Secret for the Encrypted Data

We can now create a Kubernetes secret containing the encrypted data. This ensures the secret is stored securely in Kubernetes.

```
kubectl create secret generic db-password --from-file=db-password.txt.age
```

This command creates a Kubernetes secret named db-password, which contains the encrypted password file (db-password.txt.age).

Verify the secret is created:

```
kubectl get secret db-password -o yaml
```

You should see something like:

```
apiVersion: v1
data:
  db-password.txt.age: <encrypted_content_here>
kind: Secret
metadata:
  name: db-password
  namespace: default
type: Opaque
```

The db-password.txt.age will contain the encrypted password.

Step 5: Use SOPS for YAML-based Secret Management

Instead of manually encrypting secrets, you can use SOPS to manage your secrets as YAML files that are both human-readable and encrypted. This is very useful when working with Kubernetes manifests.

1. Create a secret manifest, e.g., secret.yaml:

```
apiVersion: v1
kind: Secret
metadata:
  name: db-password
type: Opaque
data:
  db-password.txt.age: <Base64 encoded encrypted secret>
```

2. Encrypt the YAML file using SOPS:

```
sops --encrypt --age <public_key> secret.yaml > secret-encrypted.yaml
```

The secret-encrypted.yaml file will now contain the encrypted version of your secrets.

To decrypt it:

```
sops --decrypt secret-encrypted.yaml
```

Step 6: Decrypt Secrets at Runtime with a Kubernetes Deployment

In most cases, you will not store plaintext secrets directly in your code. Instead, you can use Kubernetes to decrypt them when needed.

1. Create a Kubernetes Deployment manifest that will access the secret:

File: deployment-with-secret.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: secret-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: secret-app
  template:
    metadata:
      labels:
        app: secret-app
    spec:
      containers:
      - name: secret-app
        image: nginx
        env:
        - name: DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: db-password
              key: db-password.txt.age
```

This Deployment will mount the db-password.txt.age file from the Kubernetes secret as an environment variable.

To apply the deployment:

```
kubectl apply -f deployment-with-secret.yaml
```

Step 7: Use SOPS and AGE in CI/CD Pipelines

You can automate the process of encrypting and decrypting secrets using CI/CD tools like **Jenkins**, **GitLab CI**, or **GitHub Actions**. Here's how you can implement the decryption step dynamically during your pipeline execution.

1. **Decrypt secrets in your CI/CD pipeline**:
    - Store the private key securely (preferably using your CI/CD secrets manager).
    - Use age to decrypt the secrets and inject them into your application as environment variables.

```
age -d -i /path/to/private/key.db-password.txt.age
```

2. Make sure you handle the private key and the decrypted secrets securely to avoid accidental exposure.

Step 8: Clean Up

Finally, clean up your Kubernetes resources when you are done:

```
kubectl delete secret db-password
kubectl delete deployment secret-app
```

Conclusion

This lab demonstrates how to securely manage and encrypt Kubernetes secrets using **SOPS** and **AGE**. It walks you through the process of encrypting secrets, storing them in Kubernetes, and making them available to your applications securely. This workflow ensures your sensitive data is protected at rest, while still being accessible to your applications when needed.

Domain 4.8: Understand Application Security (SecurityContexts, Capabilities, etc.)

Application security in Kubernetes focuses on securing containers and their environments to minimize vulnerabilities and risks. This involves configuring and controlling various security features to ensure that applications run with the least privilege, restrict unnecessary permissions, and protect the integrity of both the host and the containers.

In this section, we'll cover:

- SecurityContext: What it is, and how to use it for controlling security settings at the Pod or container level.
- Linux Capabilities: Understanding and configuring Linux capabilities for containers to limit the permissions granted to a container.
- Pod Security Policies (PSP): How to enforce security policies at the Pod level (if enabled).
- Other Key Security Features: read-only file systems, seccomp, and AppArmor.

1. What is a SecurityContext?

A SecurityContext is a Kubernetes resource used to define security-related settings for Pods and containers. You can specify a SecurityContext at two levels:

1. Pod-level SecurityContext: This applies to all containers in the Pod.
2. Container-level SecurityContext: This applies to individual containers in the Pod. SecurityContext allows you to configure various security settings such as user and group IDs, privilege escalation, and more. By configuring the SecurityContext, you can ensure that containers run with the appropriate security configuration, reducing the risk of exploitation.

Common SecurityContext Settings

- runAsUser: Specifies the user ID (UID) to run the container as.
- runAsGroup: Specifies the group ID (GID) to run the container as.
- fsGroup: The group ID to apply to files created by the container in volumes.
- privileged: If true, gives the container extended privileges. By default, containers do not run in privileged mode.
- allowPrivilegeEscalation: If set to false, it prevents a container from gaining more privileges than its parent process.
- readOnlyRootFilesystem: If set to true, it mounts the root filesystem as read-only.
- capabilities: Controls which Linux capabilities to add or drop from the container.

Example of Pod-level SecurityContext
File: pod-level-sec-context.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: secure-pod
spec:
  securityContext:
```

```
    runAsUser: 1000
    runAsGroup: 3000
    fsGroup: 2000
  containers:
  - name: secure-container
    image: nginx
    securityContext:
      allowPrivilegeEscalation: false
      readOnlyRootFilesystem: true
```

In this example:

- The Pod-level SecurityContext specifies that the Pod will run with a user ID of 1000, group ID of 3000, and any files created in volumes will be associated with the group ID 2000.
- The Container-level SecurityContext ensures the container cannot escalate privileges (allowPrivilegeEscalation: false) and the root filesystem is mounted as read-only (readOnlyRootFilesystem: true).

2. Linux Capabilities

Linux capabilities are a set of fine-grained access controls that allow you to grant or remove specific privileges from processes. Containers by default have a limited set of Linux capabilities to minimize security risks. However, some applications require additional capabilities. Key Capabilities

- NET_ADMIN: Allows the process to configure networking interfaces.
- SYS_ADMIN: Grants a range of system-level administrative capabilities, such as mounting file systems.
- DAC_OVERRIDE: Allows overriding file read, write, and execute permissions.

How to Modify Capabilities

In Kubernetes, you can add or remove Linux capabilities from a container using the securityContext.capabilities field. For example, if you want to add the NET_ADMIN capability, you can do so like this:

File: pod-linux-capability.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: capability-pod
spec:
  containers:
  - name: capability-container
    image: nginx
    securityContext:
      capabilities:
        add:
        - NET_ADMIN
```

In this example, the container is granted the NET_ADMIN capability, which allows it to modify networking settings.

Best Practice:

Always try to remove unnecessary capabilities to adhere to the principle of least privilege. Adding capabilities should only be done when absolutely necessary.

3. Pod Security Policies (PSP)

Pod Security Policies (PSPs) are a set of policies that control the security features available to Pods in a cluster. PSPs allow you to define rules around:

- What security context settings are allowed.
- What privileges containers can request.
- Restrictions on the use of host network, volumes, and namespaces.

Key PSP Settings:

- privileged: Whether privileged mode containers are allowed.
- runAsUser: Controls which user IDs are allowed to run.
- allowPrivilegeEscalation: Defines if privilege escalation is allowed.
- hostNetwork: Restricts Pods from using the host network.
- hostPID: Restricts Pods from using the host's process ID namespace.
- readOnlyRootFilesystem: Enforces read-only root filesystem for Pods.

PSP Example:

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: restricted-psp
spec:
  privileged: false
  runAsUser:
    rule: MustRunAsNonRoot
  allowPrivilegeEscalation: false
  volumes:
    - configMap
    - secret
  hostNetwork: false
  readOnlyRootFilesystem: true
```

In this example, the PSP:

- Disallows privileged containers.
- Requires containers to run as non-root.
- Disallows privilege escalation.
- Restricts volumes to only ConfigMaps and Secrets.
- Disallows the use of the host network.
- Enforces read-only root filesystem for containers.

Note: PSP is deprecated and will be removed in Kubernetes 1.25+. It is being replaced with Pod Security Admission (PSA).

4. Other Security Features

a) Seccomp Seccomp (Secure Computing Mode) is a Linux kernel feature that allows you to filter system calls that containers can make, limiting the attack surface. By using seccomp profiles, you can enforce what system calls a container is allowed to invoke.
File: seccomp.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: seccomp-pod
spec:
  containers:
  - name: seccomp-container
    image: my-container
    securityContext:
      seccompProfile:
        type: RuntimeDefault
```

In this example, the container uses the default seccomp profile, which restricts it from making potentially dangerous system calls.

b) AppArmor AppArmor is another Linux security module that provides mandatory access control (MAC) by restricting the capabilities of containers and their access to the underlying system. You can configure AppArmor profiles to restrict what a container can access or do.

```
apiVersion: v1
kind: Pod
metadata:
  name: apparmor-pod
spec:
  containers:
  - name: apparmor-container
    image: my-container
    securityContext:
      appArmorProfile: "runtime/default"
```

This example ensures that the container is constrained by the runtime/default AppArmor profile.

c) User Namespaces. Kubernetes can isolate the user namespace of containers, allowing containers to run with different user IDs and group IDs on the host. This helps to isolate containers and reduce the risks associated with running containers as root.

5. Best Practices for Application Security

- Run containers as non-root users: Always configure your containers to run as non-root users unless absolutely necessary.
- Limit the privileges of containers: Use SecurityContexts to limit capabilities and prevent privilege escalation.
- Use read-only root filesystem: Configure containers to use a read-only root filesystem to prevent modifications to container files.

- Use seccomp and AppArmor profiles: Implement seccomp and AppArmor to limit system calls and interactions with the host OS.
- Use RBAC for authorization: Implement Role-Based Access Control (RBAC) to restrict access to Kubernetes resources, ensuring only authorized entities can access sensitive resources.
- Audit security configurations: Regularly audit security settings like SecurityContexts, PSPs, and user privileges to ensure they adhere to the principle of least privilege.

6. Conclusion

In Kubernetes, application security is a vital part of ensuring that your workloads run in a controlled and secure environment. By leveraging tools like SecurityContexts, Linux capabilities, Pod Security Policies, seccomp, and AppArmor, you can minimize the risk of vulnerabilities and ensure your containers adhere to security best practices.

By implementing the principles of least privilege, restricting unnecessary capabilities, and utilizing advanced security features, you ensure that your Kubernetes applications are robust and secure, both inside the container and in the broader Kubernetes cluster.

Key takeaways:

- Always use SecurityContexts to define specific security configurations for Pods and containers.
- Limit container privileges using Linux capabilities and ensure Pod Security Policies are enforced.
- Leverage seccomp and AppArmor to restrict container interactions with the host.
- Continuously apply security best practices to safeguard your Kubernetes workloads.

Lab walkthrough: Understanding the need for controlling security in Containers

1. Run this command to see the possibilities of security context

```
kubectl explain pod.spec.containers.securityContext
```

2. Create a pod with privileged access. Put the following in a file called security-context-example.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-example
spec:
  containers:
  - image: busybox
    name: busybox
    args:
    - sleep
    - "3600"
    securityContext:
      privileged: true
```

3. Apply the file

```
kubectl apply -f security-context-example.yaml
```

4. Open a shell in the pod:

```
kubectl exec -n security-context-example -it -- /bin/sh
```

3. Mount the host's hard drive and read the kubelet kubeconfig file that is created during cluster initialization:

```
mkdir /hostfs
mount /dev/vda1 /hostfs
```

4. Now you can read the kubelet config file

```
cat /hostfs/kubelet/config.yaml
```

The kubeconfig file can be used to gain admin access to the Kubernetes cluster. Be careful with the containers you give privileged access to. You can control this with SecurityContext and Linux Capabilities.

Consider using a PodSecurityPolicy to lock down container privileges.

LAB Walkthrough 2: Set the security context for a Pod

To specify security settings for a Pod, include the securityContext field in the Pod specification. The securityContext field is a PodSecurityContext object. The security settings that you specify for a Pod apply to all Containers in the Pod. Here is a configuration file for a Pod that has a securityContext and an emptyDir volume:

File: security-context.yaml

```
apiVersion: v
kind: Pod
metadata:
 name: security-context-demo
spec:
 securityContext:
  runAsUser: 1000
  runAsGroup: 3000
  fsGroup: 2000
  supplementalGroups: [4000]
 volumes:
 - name: sec-ctx-vol
  emptyDir: {}
 containers:
 - name: sec-ctx-demo
  image: busybox:1.28
  command: [ "sh", "-c", "sleep 1h" ]
  volumeMounts:
  - name: sec-ctx-vol
   mountPath: /data/demo
  securityContext:
   allowPrivilegeEscalation: false
```

- In the configuration file, the runAsUser field specifies that for any Containers in the Pod, all processes run with user ID 1000.
- The runAsGroup field specifies the primary group ID of 3000 for all processes within any containers of the Pod. If this field is omitted, the primary group ID of the containers will be root(0). Any files created will also be owned by user 1000 and group 3000 when runAsGroup is specified.
- Since fsGroup field is specified, all processes of the container are also part of the supplementary group ID 2000. The owner for volume /data/demo and any files created in that volume will be Group ID 2000.
- Additionally, when the supplementalGroups field is specified, all processes of the container are also part of the specified groups. If this field is omitted, it means empty.

1: Create the Pod:

```
kubectl apply -f security-context.yaml
```

2. Verify that the Pod's Container is running:

```
kubectl get pod security-context-demo
```

3. Get a shell to the running Container:

```
kubectl exec -it security-context-demo -- sh
```

4. In your shell, list the running processes:

```
ps
```

The output shows that the processes are running as user 1000, which is the value of runAsUser:

```
PID   USER     TIME  COMMAND
    1 1000      0:00 sleep 1h
    6 1000      0:00 sh
...
```

5. In your shell, navigate to /data, and list the one directory:

```
cd /data
```

```
ls -l
```

The output shows that the /data/demo directory has group ID 2000, which is the value of fsGroup.

```
drwxrwsrwx 2 root 2000 4096 Jun  6 20:08 demo
```

6. In your shell, navigate to /data/demo, and create a file:

```
cd demo
```

```
echo hello > testfile
```

List the file in the /data/demo directory:

```
ls -l
```

The output shows that testfile has group ID 2000, which is the value of fsGroup.

```
-rw-r--r-- 1 1000 2000 6 Jun  6 20:08 testfile
```

7. Run the following command:

```
id
```

The output is similar to this:

```
uid=1000 gid=3000 groups=2000,3000,4000
```

From the output, you can see that gid is 3000 which is same as the runAsGroup field.

If the runAsGroup was omitted, the gid would remain as 0 (root) and the process will be able to interact with files that are owned by the root(0) group and groups that have the required group permissions for the root (0) group. You can also see that groups contains the group IDs which are specified by fsGroup and supplementalGroups, in addition to gid.

8. Exit your shell:

```
exit
```

Set the security context for a Container

To specify security settings for a Container, include the securityContext field in the Container manifest. The securityContext field is a SecurityContext object. Security settings that you specify for a Container apply only to the individual Container, and they override settings made at the Pod level when there is overlap. Container settings do not affect the Pod's Volumes.

Here is the configuration file for a Pod that has one Container. Both the Pod and the Container have a securityContext field:

File: security-context-2.yaml

```
apiVersion: v1
kind: Pod
metadata:
 name: security-context-demo-2
spec:
 securityContext:
   runAsUser: 1000
 containers:
 - name: sec-ctx-demo-2
   image: busybox:1.28
   command: [ "sh", "-c", "sleep 1h" ]
   securityContext:
     runAsUser: 2000
     allowPrivilegeEscalation: false
```

1. Create the Pod:

```
kubectl apply -f security-context-2.yaml
```

2. Verify that the Pod's Container is running:

```
kubectl get pod security-context-demo-2
```

3. Get a shell into the running Container:

```
kubectl exec -it security-context-demo-2 -- sh
```

4. In your shell, list the running processes:

```
ps aux
```

The output shows that the processes are running as user 2000. This is the value of runAsUser specified for the Container. It overrides the value 1000 that is specified for the Pod.

```
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
2000         1  0.0  0.0   4336   764 ?        Ss   20:36   0:00 /bin/sh -c node server.js
2000         8  0.1  0.5 772124 22604 ?        Sl   20:36   0:00 node server.js
...
```

5. Exit your shell:

```
exit
```

Domain 3.5: Debugging in Kubernetes

Debugging is a critical skill for a Kubernetes Application Developer (CKAD). Kubernetes is a powerful platform for deploying and managing containerized applications, but it introduces complexity, especially when issues arise. Whether it's a misconfigured deployment, a failed Pod, or a networking issue, knowing how to debug and troubleshoot effectively is essential. This section will cover the tools and techniques you can use to debug applications and Kubernetes resources.

1. Key Debugging Concepts in Kubernetes

When debugging in Kubernetes, it's important to understand the following key concepts: Pod Lifecycle: Pods can go through various states like Pending, Running, Succeeded, or Failed. Understanding the lifecycle helps identify where a problem might be occurring.

1. Kubernetes Resources: Debugging Kubernetes resources (like Pods, Deployments, Services, and ConfigMaps) involves checking their **configurations, status, and logs**.
2. Cluster Context: Debugging often requires understanding the cluster context, including node status, kubelet logs, and network configuration.
3. Container Behavior: Problems with containers themselves (such as crashes, memory limits, or network timeouts) often manifest in the container logs.

2. Common Debugging Scenarios in Kubernetes

As an application developer, you'll face several common issues when working with Kubernetes. Here are some of the most frequent ones: Pods Not Starting: Pods that are stuck in the Pending or CrashLoopBackOff state. Service Unavailability: Pods are running but cannot be accessed via the Kubernetes Service. Configuration Issues: Incorrect ConfigMaps, Secrets, or environment variables affecting the

behavior of the application. Resource Constraints: Pods or containers being terminated or restarted due to memory or CPU constraints.

3. Debugging Tools and Techniques

Kubernetes provides a variety of tools and commands to debug these issues. Here's a breakdown of the most useful debugging techniques:

1. kubectl describe

The kubectl describe command provides detailed information about Kubernetes resources. It's extremely useful for debugging issues with Pods, Services, Deployments, and more. Pods: To describe a Pod, use the following command:
kubectl describe pod <pod-name>

This will give you details about the Pod, including events, container status, resource usage, and any error messages.
Example output: If a Pod is stuck in Pending or CrashLoopBackOff, kubectl describe pod will often provide insight into what's going wrong (e.g., resource allocation, failed pull image, etc.).

2. kubectl logs

Logs are one of the most effective ways to understand what's happening inside a container. If a Pod is crashing or behaving unexpectedly, check its logs: Basic logs:

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

Stream logs in real-time:

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

Previous logs: If the container crashed and was restarted, you can view logs from the previous instance:

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

Logs often provide the most direct insight into why a container is failing, such as uncaught exceptions, resource issues, or missing dependencies.

3. kubectl get

The kubectl get command is used to retrieve information about Kubernetes resources. It can be helpful to view the current state of resources and identify any issues. Pods: To get information about the status of Pods in a namespace:

```
kubectl get pods -n <namespace>
```

Deployment: To check the status of a deployment:

```
kubectl get deployment <deployment-name> -n <namespace>
```

Services: To check the Services running in the cluster:

```
kubectl get svc -n <namespace>
```

The output will show the current state, such as whether the Pods are running, and if there are any issues with resource allocation.

4. kubectl top

The kubectl top command is used to monitor resource usage (CPU and memory) for nodes and Pods. It can help identify if your application is running out of resources. Monitor nodes:

```
kubectl top nodes
```

Monitor Pods:

```
kubectl top pods -n <namespace>
```

If a Pod is experiencing memory or CPU issues, it may be terminated or fail to start. kubectl top helps identify these problems.

5. kubectl port-forward

If you need to debug an application from inside a Pod, the kubectl port-forward command allows you to access an application running inside a Kubernetes cluster without exposing it externally. Example command:

```
kubectl port-forward pod/<pod-name> 8080:80
```

This forwards port 8080 on your local machine to port 80 inside the container. This is useful for debugging an application's behavior from your local machine without exposing it to the public internet.

6. kubectl exec

The kubectl exec command allows you to execute commands directly inside a running container. This is useful for inspecting the state of an application or running debugging tools in the container. Example command:

kubectl exec -it <pod-name> -c <container-name> -- /bin/bash

This opens an interactive terminal session inside the container, allowing you to run commands such as ps, top, curl, etc., to investigate the issue.

7. Checking Events

Kubernetes events provide insights into what's happening in the cluster, including Pod creation, failure, and resource allocation issues. You can view events using:

kubectl get events -n <namespace>

Events help identify why a Pod might be stuck in the Pending state, why it was evicted, or why a deployment failed to rollout.

4. Troubleshooting Common Issues

Here are some common issues and how to debug them:

1. Pod Stuck in Pending

Check the Pod's events: Often, Pods stuck in the Pending state are waiting for resources (like CPU or memory). Use

kubectl describe pod <pod-name>

to see if there are any warnings about resource availability.
Check resource requests/limits: If your Pod is requesting more resources than are available, it may not be scheduled. Check if the Pod has resource requests/limits set appropriately.

2. Pod Crashes or Restarts (CrashLoopBackOff)

View logs:

Use kubectl logs <pod-name>

to see what's happening in the container. Look for errors like missing files, connection issues, or configuration problems.

- Check for resource issues: Make sure the container isn't being killed due to resource limits (memory/CPU). Use kubectl top pods to monitor usage.
- Check readiness/liveness probes: A misconfigured probe can cause Kubernetes to restart the Pod frequently. Review the health check configurations.

3. Service Not Accessible

- Verify Service: Use kubectl describe svc to ensure the Service is correctly pointing to the Pods and has the correct ports.
- Check Network Policies: Ensure there are no network policies blocking access to the service.
- Check Pod Logs: If the application inside the Pod is failing to respond to requests, check the logs of the container to ensure the application is running and listening on the expected port.

4. Misconfigured Environment Variables or ConfigMaps

- Inspect ConfigMap/Secret: Use kubectl describe configmap or kubectl describe secret to ensure that the correct environment variables are being set.
- Check Pods' Environment: Use kubectl exec to inspect the environment inside the container to ensure that variables are correctly passed.

5. Debugging Network Issues

Networking problems often occur when Pods are unable to communicate with each other, a service, or the outside world. Common network issues include DNS resolution problems, service misconfiguration, or blocked ports.

- Test Connectivity: Use kubectl exec to ping other Pods or services to verify connectivity.
- Check Network Policies: Ensure that Kubernetes network policies are not unintentionally blocking traffic.
- Check DNS Resolution: If Pods cannot resolve domain names, it might indicate a DNS issue. Check the CoreDNS logs using kubectl logs -n kube-system .

6. Conclusion

Debugging is a crucial skill for a Kubernetes Application Developer (CKAD). Kubernetes provides several powerful tools and commands like

kubectl describe, kubectl logs, kubectl exec, and kubectl top

to help you troubleshoot and identify issues with Pods, containers, and other resources.
By familiarizing yourself with these tools and techniques, you will be better equipped to identify, diagnose, and resolve issues in your Kubernetes-based applications, ensuring smooth operation in a cloud-native environment.

Domain 4.7: Understand Service Accounts

In Kubernetes, Service Accounts are used to provide an identity for processes that run in Pods. When applications in Pods need to interact with the Kubernetes API or other resources within the cluster, they use a ServiceAccount to authenticate and authorize themselves. Each ServiceAccount is associated with a set of API access credentials, typically tokens, that the system uses to authenticate requests made by Pods. Understanding Service Accounts is crucial for building secure Kubernetes applications, as they define how Pods interact with the Kubernetes API and other resources in the cluster.
In this section, we will cover the following topics:

- What Service Accounts are and why they are important.
- How to create and configure Service Accounts.
- How to associate Service Accounts with Pods.
- Best practices for using Service Accounts securely.

1. What is a Kubernetes Service Account?

A ServiceAccount is a Kubernetes resource that provides an identity for a Pod and is used to authenticate and authorize API requests on behalf of the Pod. It's essentially a "service user" that can be assigned specific permissions (via RBAC policies) to interact with other resources within the Kubernetes cluster.

Key Points:

- Service Accounts are tied to namespaces. Each namespace can have its own set of Service Accounts.
- A ServiceAccount is associated with an automatically generated ServiceAccount Token (JWT) that is used for authentication.
- Service Accounts can be configured to interact with the Kubernetes API, make calls to other services, or access secrets.

2. Why are Service Accounts Important?

Service Accounts provide a way to manage the security and identity of applications running in your Kubernetes clusters. They help in:

- Access Control: Service Accounts are used with Role-Based Access Control (RBAC) to define what a Pod can and cannot do in the Kubernetes environment.
- Isolation and Least Privilege: By using distinct Service Accounts for different types of workloads, you can minimize the risk of exposing sensitive data or giving unnecessary permissions.
- Automation: Service Accounts are key for applications that need to automatically interact with the Kubernetes API or other resources in the cluster, such as creating Pods, accessing secrets, etc.

3. Default Service Account

Every Kubernetes namespace automatically has a default ServiceAccount. When you create a Pod and don't explicitly specify a ServiceAccount, Kubernetes automatically assigns the default ServiceAccount of the namespace to the Pod.
Example of Using Default Service Account
If you create a Pod without specifying a ServiceAccount, it will use the default one:

File: pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: default-service-account-pod
spec:
  containers:
  - name: my-container
    image: nginx
```

This Pod will automatically use the default ServiceAccount from the same namespace.

4. Creating and Using Service Accounts

a) Creating a Service Account
You can create a Service Account using a simple YAML manifest or the kubectl command. Here's an example of a ServiceAccount YAML manifest:
File: service-account.yaml

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: my-service-account
```

To create the ServiceAccount:

```
kubectl apply -f service-account.yaml
```

Alternatively, you can create a ServiceAccount directly via kubectl:

```
kubectl create serviceaccount my-service-account
```

b) Associating a Service Account with a Pod
Once a ServiceAccount is created, you can associate it with a Pod by specifying it in the Pod's serviceAccountName field:
File: pod2.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-service-account
spec:
  serviceAccountName: my-service-account
  containers:
  - name: my-container
    image: nginx
```

This Pod will use the my-service-account ServiceAccount for authentication and authorization when interacting with the Kubernetes API. If you don't explicitly set

the serviceAccountName, the Pod will use the default ServiceAccount in the namespace.

5. Service Account Tokens

When a Pod uses a ServiceAccount, Kubernetes automatically creates a ServiceAccount Token. This token is used by the Pod to authenticate with the Kubernetes API.
Kubernetes automatically mounts the token as a file inside the Pod, typically at /var/run/secrets/kubernetes.io/serviceaccount/token.

The token is in JWT (JSON Web Token) format and can be used for authenticating API requests.
Example: Accessing the ServiceAccount Token Inside a Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-token
spec:
  serviceAccountName: my-service-account
  containers:
  - name: my-container
    image: nginx
    command: ["cat", "/var/run/secrets/kubernetes.io/serviceaccount/token"]
```

This Pod will print the ServiceAccount token when it is run.

6. Role-Based Access Control (RBAC) with Service Accounts

Service Accounts are often used in combination with Role-Based Access Control (RBAC) to define what a ServiceAccount (and, by extension, the Pod using that ServiceAccount) can do in the cluster.

a) Create a Role
Here's an example of creating a Role that allows access to Pods within a namespace:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list"]
```

This Role gives the ServiceAccount permission to get and list Pods in the default namespace.

b) Create a RoleBinding
A RoleBinding associates the Role with a ServiceAccount:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: pod-reader-binding
  namespace: default
subjects:
- kind: ServiceAccount
  name: my-service-account
  namespace: default
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

This binding allows the my-service-account ServiceAccount to read Pods in the default namespace.

7. Service Account Best Practices

1. Use the Least Privilege Principle: Grant only the minimum permissions required for the ServiceAccount to perform its tasks. This minimizes the risk of exposure.
2. Use Separate Service Accounts for Different Workloads: Avoid using the default ServiceAccount for every Pod. Create distinct ServiceAccounts for different types of workloads or applications to reduce the blast radius of compromised credentials.
3. Avoid Sharing Service Accounts Across Pods: When multiple Pods need different levels of access, create separate ServiceAccounts with the appropriate permissions for each.
4. Audit Service Account Usage: Regularly audit the permissions and roles granted to ServiceAccounts, and ensure they are not over-privileged.
5. Secure the Service Account Tokens: Ensure tokens are used securely, and access to the file paths where tokens are stored (e.g., /var/run/secrets/kubernetes.io/serviceaccount/) is restricted.
6. Rotate Service Account Tokens: Kubernetes automatically handles the expiration of tokens, but for critical workloads, consider integrating with external tools for more fine-grained control and rotation.

8. Deleting a Service Account

To delete a ServiceAccount:

```
kubectl delete serviceaccount my-service-account
```

Make sure to check if any Pods are using the ServiceAccount before deletion, as removing it may affect those Pods' ability to interact with the Kubernetes API.

9 . Conclusion

Service Accounts are a vital component in Kubernetes security, providing a way to authenticate and authorize Pods and workloads within a cluster. By associating Service Accounts with appropriate RBAC roles, you can enforce strict access control policies and reduce the risk of over-privileged access.
In this section, you have learned:

- How to create and use Service Accounts.
- How to associate Service Accounts with Pods.
- The role of RBAC in controlling access to Kubernetes resources via Service Accounts.
- Best practices for securing and managing Service Accounts in your cluster.

By following best practices and using Service Accounts appropriately, you can secure your Kubernetes applications and ensure that they interact with the Kubernetes API and other resources in a controlled and authorized manner.

Domain 3.1: Understand API Deprecations

As Kubernetes evolves, the APIs and features it provides can change over time. API deprecations are an essential aspect of this evolution. Understanding how Kubernetes manages deprecations and how to handle them in your application development is crucial for ensuring that your workloads remain stable and compatible with future versions of Kubernetes.
In this section, we'll explore what API deprecations are, why they occur, and how to handle them effectively as a Kubernetes Application Developer (CKAD).

1. What Are API Deprecations?

In Kubernetes, API deprecation refers to the process of marking an API or API version as outdated and signaling that it will be removed in a future release. The deprecation process is Kubernetes' way of ensuring backward compatibility while allowing users time to update their applications and configurations before older versions of APIs are removed.

Why Do API Deprecations Happen?

- Improved Features: New versions of APIs may provide more efficient, secure, or feature-rich implementations.
- Code Cleanup: Over time, certain API versions may no longer be needed as Kubernetes evolves, and these old versions are deprecated.
- Enhanced Performance & Usability: The Kubernetes team may optimize APIs, change field names, or restructure resources to improve performance, usability, and flexibility. When an API is deprecated, it is typically still available for some time but may show warnings about its deprecation. Eventually, it will be removed in a future release (often a major version).

2. How Kubernetes Marks APIs as Deprecated

Kubernetes marks an API as deprecated in the following ways:

- Deprecation Warnings in Logs: When using a deprecated API, Kubernetes emits warning messages indicating the API is deprecated and should be updated.
- Documentation: Kubernetes documentation provides clear guidance on which API versions are deprecated and what their replacements are.
- API Versioning: Kubernetes has multiple versions of APIs to ensure backward compatibility. Deprecated APIs are often maintained for a while in a version before they are completely removed.

For example, if you are using an older version of an API, such as extensions/v1beta1 for resources like Deployments, you might receive a warning about deprecation in the logs:

W0311 16:10:23.123456 1 warnings.go:67] extensions/v1beta1 deployments is deprecated in v1.19.0+, use apps/v1 instead.

3. Common Kubernetes APIs That Are Deprecated

Over the years, Kubernetes has deprecated several APIs. Some of the most notable deprecated APIs include:

extensions/v1beta1 -> apps/v1 (Deployments, ReplicaSets, etc.):

The extensions/v1beta1 API for resources like Deployments and ReplicaSets was deprecated in favor of the new API group apps/v1. Example:

```
apiVersion: extensions/v1beta1  # Deprecated
kind: Deployment
```

extensions/v1beta1 -> networking.k8s.io/v1 (Ingress):

The extensions/v1beta1 API for Ingress resources was also deprecated in favor of networking.k8s.io/v1.

Example:

apiVersion: networking.k8s.io/v1 # New

kind: Ingress

apiextensions.k8s.io/v1beta1 -> apiextensions.k8s.io/v1 (CustomResourceDefinitions):

The v1beta1 version of CustomResourceDefinitions (CRDs) was deprecated in favor of v1.

batch/v1beta1 -> batch/v1 (CronJobs):

The batch/v1beta1 version of CronJobs was deprecated in favor of batch/v1.

4. How to Handle API Deprecations

As a Kubernetes Application Developer (CKAD), it's important to be proactive about handling API deprecations. Here's how you can manage deprecated APIs:

1. Stay Updated with Kubernetes Release Notes

Kubernetes releases new versions regularly (every 3 months), and each release contains a changelog that lists deprecated APIs and their replacements. Staying on top of these changes will allow you to plan your upgrades and refactor your

application accordingly. You can find release notes on the official Kubernetes GitHub page and on Kubernetes documentation.

2. Update Your YAML Manifests

Whenever you encounter deprecation warnings in your Kubernetes environment, you should update your YAML files to use the newer API versions. For example, if you're using extensions/v1beta1 for Deployments, you should switch to apps/v1.

```
apiVersion: apps/v1  # New API version
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: my-app
          image: my-app:latest
```

3. Use API Version Migrations

If the deprecated API is still functional, Kubernetes will generally provide a migration path. Kubernetes' API reference documentation and release notes will help you understand how to migrate from deprecated versions to newer, stable ones. For example:
Migrate from extensions/v1beta1 to apps/v1 for Deployments.

Migrate from extensions/v1beta1 to networking.k8s.io/v1 for Ingress.

4. Test Changes in Staging Environments

Before deploying changes that involve deprecated API migrations to production, test your application thoroughly in a staging environment. This ensures that the new API versions don't break your application and that the transition is smooth.

5. Leverage Kubernetes Features for Compatibility

Kubernetes supports multi-version compatibility for a certain period. This means you can run multiple versions of APIs for a while (e.g., apps/v1 and extensions/v1beta1) in the same cluster. However, you should plan to update deprecated APIs before the versions are removed completely in future Kubernetes releases.

6. Deprecation Timeline

Kubernetes provides a deprecation timeline in its release notes. Kubernetes uses a two-year deprecation policy:

- A feature marked as deprecated will be supported for two more major versions (around 6 months each).
- After the deprecation period ends, the feature is removed.

For instance:

Kubernetes 1.14 marked extensions/v1beta1 APIs (like Deployments and Ingress) as deprecated.

- They were fully removed in Kubernetes 1.22.
- The deprecation timeline is announced in the release notes, so it's important to stay informed about upcoming changes and plan your application updates accordingly.

7. Benefits of Understanding API Deprecations

- Future-proofing: By addressing deprecated APIs early, you ensure that your applications remain compatible with future Kubernetes versions.
- Security: Newer API versions often come with security improvements, bug fixes, and better performance.
- Stability: By updating deprecated APIs, you avoid the risk of them being removed in future versions, which could break your workloads.
- Improved Maintainability: Using up-to-date API versions ensures that your Kubernetes manifests are aligned with the current best practices and Kubernetes standards.

8. Conclusion

As a Kubernetes Application Developer (CKAD), staying on top of API deprecations is a critical skill to ensure that your applications remain compatible with newer versions of Kubernetes. By:

1. Regularly reviewing release notes for deprecated APIs.
2. Updating your YAML configurations to use current APIs.
3. Testing changes in non-production environments.

You can ensure your Kubernetes applications are future-proof and continue to work seamlessly as Kubernetes evolves. Pro tip: Utilize tools like kubectl deprecations and check Kubernetes' release notes to track deprecations and plan your updates accordingly.

Domain 3.4: Use Built-in CLI Tools to Monitor Kubernetes Applications

As a Kubernetes Application Developer (CKAD), it's essential to be able to monitor the applications you deploy on Kubernetes. Monitoring helps you track the health, performance, and availability of applications, and identify any issues in real time. Kubernetes provides built-in CLI tools that can help you monitor applications and gain insight into the status of your resources. In this section, we'll focus on some of the most commonly used Kubernetes CLI tools and commands for monitoring your applications and how to use them effectively.

1. kubectl - The Kubernetes Command-Line Interface

The kubectl CLI is the primary tool for interacting with Kubernetes clusters. It can be used to inspect and monitor various components of the cluster and applications running in it.

2. Common kubectl Commands for Monitoring Kubernetes Applications

1. Checking Pod Status

You can use kubectl to get information about your Pods, such as their current state, events, and resource usage. This is useful for monitoring the health and status of the applications within your Pods. View all Pods in a specific namespace:

```
kubectl get pods -n <namespace>
```

to show additional details like the node the pod is running on. View detailed information about a specific Pod, use:

```
kubectl get pods -o wide
```

This command gives you detailed information about the Pod, including events, container status, resource usage, and more.

```
kubectl describe pod <pod-name>
```

View the logs of a specific Pod:

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

Use -f to stream the logs in real time:

```
kubectl logs -f <pod-name>
```

View logs for all containers in a Pod (if there are multiple containers):

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

2. Monitoring Deployments

A Deployment is a higher-level object in Kubernetes used to manage the lifecycle of Pods. You can use kubectl to monitor the status of Deployments. View all Deployments:

```
kubectl get deployments
```

Get detailed information about a specific Deployment:

```
kubectl describe deployment <deployment-name>
```

This provides insights into the Deployment's status, including the number of replicas, pods being used, and any ongoing issues. Check the rollout status of a Deployment:

```
kubectl rollout status deployment/<deployment-name>
```

This command helps you track the progress of a rolling update and verify that the deployment is working as expected. Undo a Deployment rollout:
If something goes wrong, you can rollback to the previous version:

```
kubectl rollout undo deployment/<deployment-name>
```

3. Monitoring Services

A Service in Kubernetes exposes your application to the network. Monitoring Services helps ensure that your application is accessible and traffic is being routed correctly. View all Services in a namespace:

```
kubectl get services
```

Get detailed information about a specific Service:

```
kubectl describe service <service-name>
```

This will show details about the service, such as the IP address, ports, and which Pods are being targeted.

4. Monitoring Namespaces

Namespaces are a way to partition resources in a Kubernetes cluster. Monitoring namespaces helps you get a clearer understanding of resource allocation and application isolation. List all namespaces:
kubectl get namespaces

Get resources (Pods, Services, Deployments) within a specific namespace:

```
kubectl get all -n <namespace>
```

5. Monitoring Resource Usage

Kubernetes also provides resource metrics, such as CPU and memory usage, to help you monitor how efficiently your applications are running. View resource usage for Pods:
To use this command, you need to have the metrics-server installed in your cluster.

```
kubectl top pod
```

This command shows the current CPU and memory usage for all Pods in the default namespace. You can also specify a namespace:

```
kubectl top pod -n <namespace>
```

View resource usage for Nodes:

```
kubectl top node
```

This command shows the CPU and memory usage for each node in the cluster.

6. Monitoring Events

Kubernetes logs events that provide insights into what's happening in the cluster, such as Pod failures, resource allocation issues, or deployment status changes.
View all events in a namespace:
kubectl get events -n <namespace>

This command shows all events in a specific namespace. Use it to track errors, warnings, or changes to your application or resources.

View events in real time:

```
kubectl get events --watch
```

This allows you to stream events as they occur in your cluster, which is helpful for troubleshooting live issues.

3. Monitoring with kubectl Plugins

Kubernetes supports the use of plugins that can extend the functionality of kubectl. Some plugins are designed specifically for monitoring and troubleshooting.

1. kube-ps1: Provides Kubernetes context information in your shell prompt.
2. kubectl-gotmpl: Allows you to use Go templating for more flexible queries and outputs.
3. kubectl-debug: Provides an easy way to debug running Pods by creating a temporary container. You can install these plugins to enhance your Kubernetes monitoring experience.

4. Troubleshooting Application Failures with kubectl

When things go wrong, you can use kubectl to troubleshoot the issue. Inspect the state of Pods:

```
kubectl get pods -o wide
```

This command shows which node the Pods are running on and helps you diagnose if there is a resource issue (e.g., resource limits, node failures).
Describe the Pod to get detailed information about the error:

```
kubectl describe pod <pod-name>
```

This provides detailed event logs and error messages for the container and helps you identify why the Pod might not be running properly.
Access a shell inside a running Pod (for debugging purposes):

```
kubectl exec -it <pod-name> -- /bin/sh
```

This allows you to interact with the application running inside the container and inspect logs, files, or processes.

5. Leveraging Labels and Selectors Kubernetes allows you to use labels and selectors to filter resources in your cluster. Labels can be added to Pods, Services, and other resources to group them based on common attributes. Get Pods with a specific label:

```
kubectl get pods -l app=<label-name>
```

View resources by label in a specific namespace:

```
kubectl get all -l app=<label-name> -n <namespace>
```

Using labels and selectors is helpful when you need to monitor or troubleshoot a subset of resources in a large cluster.

6. Conclusion

As a Kubernetes Application Developer (CKAD), being proficient with kubectl is essential for monitoring the health, performance, and availability of your applications in a Kubernetes cluster. By using the built-in CLI tools, you can quickly gain insight into the state of your Pods, Deployments, Services, and other resources, allowing you to troubleshoot and optimize your applications effectively.
Some key commands to remember:

1. kubectl get — View resources (Pods, Deployments, Services, etc.).
2. kubectl describe — Get detailed resource information and events.
3. kubectl logs — View application logs.
4. kubectl top — Monitor resource usage.
5. kubectl exec — Access running Pods for debugging.

Mastering these tools will empower you to ensure your applications are running smoothly in your Kubernetes environment, enabling you to detect and resolve issues quickly.

Domain 3.4: Utilize Container Logs

As a Kubernetes Application Developer (CKAD), one of your core responsibilities is ensuring the health and performance of applications running in Kubernetes. Logs are a critical part of this process, as they provide detailed insights into what's happening inside containers and help you troubleshoot issues. Kubernetes provides a robust system for capturing and managing container logs, which can be accessed through various means using kubectl and integrated tools. In this section, we'll discuss how to effectively utilize container logs in Kubernetes for monitoring, debugging, and troubleshooting applications.

1. Basics of Container Logging in Kubernetes

Kubernetes stores logs from containers in a centralized location on the node filesystem. By default, logs are captured from the standard output (stdout) and standard error (stderr) streams of the containers running in your Pods. These logs are then made accessible to you through the kubectl CLI.

- Pod Logs: Logs are collected per Pod and per container within the Pod.
- Log Location: On each node, container logs are typically stored in the /var/log/containers directory. However, accessing logs through kubectl is the recommended way.

2. Accessing Container Logs Using kubectl

1. View Logs of a Specific Pod

To view the logs of a container within a specific Pod, use the following kubectl command:

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

This will retrieve the logs from the first container in the Pod. Specify the container name (if the Pod has multiple containers):
kubectl logs <pod-name> -c <container-name>

View logs for a specific namespace:

```
kubectl logs <pod-name> -n <namespace>
```

2. View Previous Logs

If a container has crashed and restarted, you can view the logs from the previous instance of the container by using the --previous flag:

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

This is useful for debugging container crashes or other issues that occur before the container restarts.

3. Stream Logs in Real-Time

To monitor logs in real time (i.e., stream the logs as they are generated), use the -f (follow) option:

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

This command keeps the connection open and continuously streams logs as they are generated, which is invaluable when debugging live issues.

3. Viewing Logs for All Containers in a Pod

If a Pod has multiple containers, you can view the logs for all containers simultaneously by using the following command:

```
kubectl logs <pod-name> --all-containers=true
```

This will display the logs for every container within the specified Pod.

4. Retrieve Logs from All Pods in a Namespace

If you need to monitor logs for multiple Pods in a namespace, you can run:

```
kubectl logs -n <namespace> --selector=<label-selector> --all-containers=true
```

The --selector= helps filter the Pods based on labels, which can be useful if you only want to view logs for a specific set of Pods (e.g., Pods with a particular application label).
The --all-containers=true flag ensures logs from all containers in the selected Pods are displayed.

5. Searching and Filtering Logs

If you're looking for specific information in your logs, you can pipe the output of kubectl logs into common Unix commands like grep to search for patterns. For example, to find all occurrences of the string "error" in the logs:

```
kubectl logs <pod-name> -c <container-name> | grep "error"
```

This can help you quickly identify issues and narrow down your troubleshooting.

6. Log Aggregation in Kubernetes

While kubectl provides access to logs, for large-scale Kubernetes environments, it's recommended to use a log aggregation solution. These solutions aggregate and centralize logs from all your Pods, Nodes, and containers, making it easier to search, analyze, and monitor logs across your entire Kubernetes cluster. Some common log aggregation tools include:

1. ELK Stack (Elasticsearch, Logstash, and Kibana)
2. Fluentd
3. Prometheus & Grafana (used for metrics but can also handle logs with integrations)
4. Loki (part of the Grafana ecosystem)

By using these tools, logs are collected, indexed, and stored in a centralized location, allowing you to search and visualize logs in real-time through dashboards.

7. Implementing Logging Best Practices

To make your logs more useful and easier to manage, you can adopt the following logging best practices:

1. Structured Logging Ensure that your applications generate logs in a structured format, such as JSON. Structured logs are easier to parse and analyze, especially when using log aggregation tools. Example JSON log entry:

```
{
  "level": "error",
  "message": "Failed to connect to the database",
  "timestamp": "2023-01-10T10:00:00Z",
  "pod_name": "my-app-12345",
  "container_name": "my-app-container"
}
```

2. Use Log Levels Use log levels such as INFO, WARN, ERROR, and DEBUG to classify the severity of logs. This makes it easier to filter and prioritize issues when viewing logs.
3. Keep Logs Short and Relevant While it's important to log enough information to understand what's going on, avoid over-logging. Logs should be concise, relevant, and avoid redundancy.
4. Set Log Rotation In large-scale Kubernetes environments, logs can grow quickly. Ensure that log rotation is configured so logs don't fill up disk space. This can be managed by tools like logrotate on the node or within the logging solution you use (such as Fluentd or the ELK stack).
5. Manage and Monitor Log Retention Implement a retention policy for logs to avoid the accumulation of large amounts of data. You can define how long logs should be stored, and ensure they're archived or deleted after a certain period.

8. Troubleshooting with Container Logs

Logs are your first line of defense when troubleshooting container-related issues. Here's how you can troubleshoot common problems using container logs:

1. Pod Crash Looping If a container within a Pod is in a crash loop, you can inspect the logs to find out why the application is failing.

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

Look for error messages or stack traces that indicate the cause of failure.

2. Application Behavior If your application isn't behaving as expected, logs can provide clues. Use kubectl logs -f to stream the logs and monitor application activity over time.
3. Networking Issues If your application is having networking issues (e.g., unable to connect to other services or databases), checking the logs can reveal errors related to network connectivity.
4. Resource Constraints If your application is failing due to resource constraints (e.g., CPU or memory limits), you can examine the logs to find out if it's hitting limits or being throttled.
5. Conclusion As a Kubernetes Application Developer (CKAD), utilizing container logs effectively is crucial for ensuring the health of your applications. Kubernetes makes it easy to access logs with the kubectl logs command, but for large-scale environments, implementing a log aggregation and analysis tool is essential for efficient monitoring.

Some best practices for utilizing container logs:

1. Use structured logging and log levels to make logs easier to parse.
2. Stream logs in real-time to monitor live events.
3. Use log aggregation tools to centralize and search logs efficiently.
4. Monitor logs for patterns and common issues such as crashes or resource limits.
5. Mastering the use of container logs will help you quickly identify and resolve issues, leading to more stable and reliable applications on Kubernetes.

Certified Kubernetes Application Developer (CKAD)

By Damian Igbe, PhD

Day 5

Understanding Common Errors in Kubernetes Deployments

Deploying applications in Kubernetes can sometimes lead to common errors. These errors typically fall into a few categories: **misconfigurations, resource issues, and Kubernetes-specific errors**. Here's a list of common issues encountered during Kubernetes application deployments, along with explanations and solutions for fixing them.

---

1. Pod CrashLoopBackOff

Error Description: A CrashLoopBackOff means that the pod is repeatedly crashing and Kubernetes is trying to restart it. This usually happens when the application inside the pod fails to start or crashes soon after starting.

Possible Causes:

- Application crashes due to incorrect configuration or missing dependencies.
- The container image is invalid or does not exist.
- Missing or incorrect environment variables.
- Insufficient resources (CPU/Memory).

How to Fix:

- Check pod logs: Use kubectl logs <pod-name> to check the logs for errors.

  kubectl logs <pod-name>

This can help identify application-specific errors.

- Check the events: Use kubectl describe pod <pod-name> to see events related to the pod, such as resource issues or configuration problems.

  kubectl describe pod <pod-name>
- Increase resources: If the application needs more memory or CPU than allocated, adjust the resource requests and limits in the deployment YAML:

```
resources:
  requests:
    memory: "64Mi"
    cpu: "250m"
  limits:
    memory: "128Mi"
    cpu: "500m"
```

- Check readiness and liveness probes: Ensure that the application's readiness and liveness probes are correctly configured. If probes fail, Kubernetes might restart the container.

```
readinessProbe:
 httpGet:
   path: /healthz
   port: 8080
 initialDelaySeconds: 3
 periodSeconds: 5
```

---

2. ImagePullBackOff

Error Description: The ImagePullBackOff status indicates that Kubernetes was unable to pull the container image from the registry.

Possible Causes:

- The image name or tag is incorrect.
- The image is private and requires authentication.
- The image doesn't exist in the registry.

How to Fix:

- Check the image name and tag: Make sure the container image name and tag are correct in the deployment YAML.

  ```
  image: nginx:1.20.0
  ```
- Check image repository access: If you're using a private Docker registry, ensure the correct credentials are provided using a Secret for Docker registry authentication:

  ```
  kubectl create secret docker-registry my-secret
  --docker-server=<your-registry> --docker-username=<username>
  --docker-password=<password> --docker-email=<email>
  ```

- Verify image existence: Ensure the image is available in the registry you are pulling from (Docker Hub, Google Container Registry, etc.).

---

3. ResourceQuotaExceeded

Error Description: A ResourceQuotaExceeded error occurs when a namespace exceeds its resource limits (e.g., memory or CPU).

Possible Causes:

- The namespace has exceeded the resource quotas set for it.
- The requested resources exceed the available limits.

How to Fix:

- Check the Resource Quota: Use the following command to check the current resource usage and quotas:

  kubectl describe quota

- Modify the Resource Quotas: If necessary, adjust the resource quotas for the namespace by editing the ResourceQuota object:

```
 apiVersion: v1
kind: ResourceQuota
metadata:
  name: resource-quota
spec:
  hard:
    requests.cpu: "10"
    requests.memory: "50Gi"
```

- Reduce the resources for the pod: If a pod is requesting too many resources, reduce its resource requests in the deployment YAML:

```
resources:
  requests:
    cpu: "100m"
    memory: "128Mi"
  limits:
    cpu: "200m"
    memory: "256Mi"
```

---

4. Service Not Exposed (404 Error or Connection Refused)

Error Description: This happens when the application is deployed, but the service is not correctly exposing it, or there is an issue connecting to the service.

Possible Causes:

- The service is misconfigured (incorrect port, selector mismatch).
- The service is not properly exposed externally (for example, a ClusterIP service when you need a LoadBalancer service).

How to Fix:

- Check the service configuration: Verify that the service selector matches the labels of the pods it should route traffic to.

```
selector:
  app: nginx
```

- Check the port mapping: Ensure that the service port and the container port are correctly specified:

```
spec:
  ports:
    - port: 80
      targetPort: 80
```

- Check the service type: If you need external access, ensure the service is of type LoadBalancer (for cloud providers) or NodePort.

```
type: LoadBalancer
```

- Check service status: Use kubectl get services to verify if the service is created correctly and has an external IP (if needed).

```
kubectl get svc
```

---

5. Pod Stuck in Pending State

Error Description: When a pod is stuck in a Pending state, Kubernetes cannot schedule the pod onto any node. This could be due to resource constraints, unsatisfied node conditions, or other scheduling issues.

Possible Causes:

- No available nodes with sufficient resources.
- Incorrect resource requests that cannot be fulfilled by the cluster.
- Missing persistent storage if the pod needs a volume.

How to Fix:

- Check node resources: Use kubectl describe node to check the available resources on each node.

  kubectl describe node <node-name>
- Check pod events: Use kubectl describe pod <pod-name> to see why the pod is stuck in the Pending state.

  kubectl describe pod <pod-name>
- Adjust resource requests: Ensure the pod resource requests are within the available resources of the nodes.

```
resources:
 requests:
   cpu: "100m"
   memory: "128Mi"
```

- Check storage availability: If the pod requires persistent storage, ensure that a PersistentVolume (PV) exists and is bound to a PersistentVolumeClaim (PVC).

---

6. Insufficient Permissions (RBAC Errors)

Error Description: RBAC (Role-Based Access Control) errors occur when a pod or user doesn't have the correct permissions to access resources in Kubernetes.

Possible Causes:

- The user or service account doesn't have the necessary Role or ClusterRole to perform the action.
- The service account associated with the pod doesn't have the correct permissions.

How to Fix:

- Check RBAC roles and bindings: Use kubectl describe to check the roles and role bindings.

  kubectl describe rolebinding <rolebinding-name>
- Create or modify the RBAC configuration: If necessary, create a Role or ClusterRole and bind it to the service account.

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: <namespace>
  name: my-role
rules:
  - apiGroups: [""]
    resources: ["pods"]
    verbs: ["get", "list", "create"]
```

---

7. Network Issues (Timeouts or Connection Refused)

Error Description: Network issues can arise when pods cannot communicate with each other or services. Common symptoms include timeouts or "connection refused" errors.

Possible Causes:

- Network policies blocking communication.
- Services or pods are not correctly exposed.
- Incorrect DNS resolution inside the cluster.

How to Fix:

- Check network policies: If network policies are being used, ensure that they allow traffic between the pods and services.

  kubectl get networkpolicy

- Check DNS resolution: Verify that the DNS is working inside the cluster using the following command to resolve service names:

  kubectl exec -it <pod-name> -- nslookup <service-name>
- Ensure correct pod communication: Make sure that services and pods are in the same namespace or that the appropriate service names are used for cross-namespace communication.

Conclusion

These are just a few of the most common errors you might encounter when deploying applications in Kubernetes. The general approach to troubleshooting involves:

- Checking logs and events to understand the issue.
- Verifying your YAML configuration files (services, deployments, resources).
- Ensuring that all required resources (like secrets, PVs, and network policies) are correctly configured.

By carefully diagnosing each error and following the best practices for resource allocation and access management, you can resolve most issues and ensure smoother deployments.

Troubleshooting and Fixing CreateContainerConfigError and CreateContainerError in Kubernetes

Both CreateContainerConfigError and CreateContainerError are errors that Kubernetes may show when a pod is having trouble starting due to issues with container creation. These errors are usually indicative of problems with the **container image, configuration, or environment.** Let's walk through each error and the steps to troubleshoot and fix them.

---

1. CreateContainerConfigError

Error Description: The CreateContainerConfigError occurs when Kubernetes fails to create the container due to a problem in the configuration (e.g., the container's settings, such as environment variables, volume mounts, or container image). This is typically seen when Kubernetes cannot correctly configure or start the container based on the pod specification.

Common Causes:

- Incorrect container image name or tag.
- Missing environment variables or secrets.
- Invalid volume mounts or file paths.
- Incorrect configuration of environment variables or command arguments in the pod definition.
- Invalid Kubernetes resource configurations (e.g., an invalid configMap, secret, or volume).

How to Troubleshoot and Fix:

1. **Check Pod Events:**
    - Use kubectl describe pod <pod-name> to see detailed information about the pod and its events.
    - Look for specific error messages related to the container creation in the Events section.

Example:

kubectl describe pod <pod-name>

Look for lines like this in the event section:

Warning FailedCreate <some timestamp> pod/<pod-name> CreateContainerConfigError

2. **Check Pod Logs:**

    - If the pod was able to start the container briefly before failing, check the pod logs to see what might have gone wrong. Sometimes, you can find specific error details.

kubectl logs <pod-name> --previous

3. **Check the Container Image:**

    - Ensure that the image name and tag in the YAML file are correct.
    - If using a private registry, make sure the appropriate credentials (image pull secrets) are in place.

image: nginx:latest

4. **Check the Configuration (Environment Variables, Volumes):**

    - If your container requires certain environment variables, make sure they are correctly set in the pod definition.
    - Verify that ConfigMaps or Secrets are correctly created and linked to the pod.
    - Ensure that volumes are properly configured, and paths exist within the container (if volumes are mounted).

Example:

```
 env:

  - name: MY_ENV_VAR
    valueFrom:
      configMapKeyRef:
        name: my-configmap
        key: mykey
volumes:
  - name: my-volume
    secret:
      secretName: my-secret
```

5. **Review the Command or Args:**

- Ensure the container's command and args are configured correctly if they are explicitly set. Misconfigured startup commands or arguments can cause a container to fail to start.

```
command:
 - /bin/sh
 - -c
 - "echo Hello"
```

6. **Check Docker Image Compatibility:**

   - If you built your own image, ensure it's built correctly and is compatible with the Kubernetes environment.
   - Check that the image has the necessary entry points defined (CMD or ENTRYPOINT in Dockerfile).

---

2. CreateContainerError

Error Description: The CreateContainerError occurs when Kubernetes tries to create the container, but the container itself fails to start due to issues in the image or the environment. This is often caused by runtime issues within the container after it's been created, such as the application inside the container failing to launch or misconfiguration.

Common Causes:

- Container image issues (e.g., missing binaries or misconfigured entrypoints).
- Permissions issues (e.g., the container doesn't have the required permissions).
- Application crashes or exceptions that prevent the container from running.
- Misconfigured resources (e.g., insufficient memory or CPU limits).
- Missing or misconfigured secrets or environment variables.

How to Troubleshoot and Fix:

1. **Check Pod Events:**

   - Just like with CreateContainerConfigError, you should start by looking at the events of the pod using kubectl describe pod <pod-name>.

   kubectl describe pod <pod-name>

Look for entries like:

Warning FailedCreate <timestamp> pod/<pod-name> CreateContainerError

2. **Check Container Logs:**
    - If the container starts and crashes, you can use kubectl logs to check the logs. If the pod is in a CrashLoopBackOff state, use the --previous flag to view the logs of the last terminated container:

    kubectl logs <pod-name> --previous

3. **Check Application Errors:**
    - If your application inside the container is crashing, inspect the application logs for errors (e.g., missing files, incorrect configurations).
    - If the container is an application server, check if it's failing to bind to the required port or if it's misconfigured in some other way.
4. **Ensure Permissions:**
    - Make sure that the container has the correct permissions. For example, if your application needs to write to a file or use a specific network port, ensure that the user running the application in the container has the necessary permissions.
    - Verify securityContext settings in your pod configuration, such as:

```
securityContext:
  runAsUser: 1000
  runAsGroup: 3000
  fsGroup: 2000
```

5. **Review Resource Limits:**
    - Ensure the pod has enough resources allocated. If the container is being killed because it exceeds its memory or CPU limits, you can adjust the resources in the pod definition:

```
resources:

  requests:
    memory: "64Mi"
    cpu: "250m"
  limits:
    memory: "128Mi"
```

```
    cpu: "500m"
```

6. **Verify Dependencies:**
    - If your container relies on specific services or other containers, ensure that those dependencies are available. For example, if your application needs a database to function, ensure that the database is running and accessible.
7. **Check EntryPoint / CMD in Dockerfile:**
    - Make sure that the Docker image you're using has the correct entry point. If the ENTRYPOINT or CMD is not set correctly, the container might not run. Check the Dockerfile or Kubernetes YAML for the correct command to run the application.

```
command: ["/bin/sh", "-c", "python app.py"]
```

General Debugging Approach:

1. **Inspect Events and Logs:**
    - Run kubectl describe pod <pod-name> to understand why the container failed.
    - Use kubectl logs <pod-name> --previous to look at logs from the last run of the container.
2. **Check Image Availability:**
    - If the container image is missing or incorrect, fix the image reference or push the correct image to the registry.
3. **Recreate the Pod:**
    - If the pod configuration has been fixed, recreate it using kubectl delete pod <pod-name>, and Kubernetes will automatically recreate it based on the deployment configuration.
4. **Check Kubernetes Version Compatibility:**
    - Sometimes, image issues can arise due to Kubernetes API changes. Make sure the image you're using is compatible with the version of Kubernetes you're running.

Conclusion:

CreateContainerConfigError and CreateContainerError can be caused by misconfigurations in your Kubernetes YAML files or issues within the container image or application itself. The key to fixing them lies in careful inspection of logs, events, configuration files, and application behavior.

By using the kubectl describe and kubectl logs commands, you can quickly identify the underlying issues. Once identified, you can adjust the configuration, ensure the correct permissions, check image integrity, and make sure the application's dependencies are available to fix these errors.

Top 20 Common Errors with Deployments

When deploying applications in Kubernetes, several errors can arise. Below are the **top 20 common errors** that Kubernetes users may encounter, along with explanations and potential fixes:

1. Pod CrashLoopBackOff

- **Cause**: The pod is repeatedly crashing after being started.
- **Fix**:
    - Check the pod logs using kubectl logs <pod-name>.
    - Inspect application errors or misconfigurations in environment variables, resources, or application code.

2. ImagePullBackOff

- **Cause**: Kubernetes cannot pull the container image from the registry.
- **Fix**:
    - Verify that the image name and tag are correct.
    - Ensure the image exists in the container registry.
    - For private registries, ensure that you have the correct credentials (imagePullSecrets).

3. Pending Pod (Resource Allocation Failure)

- **Cause**: Pods cannot be scheduled due to insufficient resources (CPU/memory) on the cluster nodes.
- **Fix**:
    - Check available resources using kubectl describe node.
    - Adjust pod resource requests and limits in the YAML.
    - Check the kubectl describe pod <pod-name> for scheduling errors.

4. Deployment Not Rolling Out

- **Cause**: The deployment is stuck, and no new pods are being created.
- **Fix**:
    - Check the status of the deployment with kubectl rollout status deployment/<deployment-name>.
    - Check for any issues in the pod's spec, such as incorrect container images or ports.

5. CreateContainerConfigError

- **Cause**: Kubernetes cannot create the container due to invalid configuration (e.g., missing environment variables or configuration files).
- **Fix**:
    - Review the pod's YAML to ensure all environment variables, volumes, and config maps are defined correctly.
    - Check for missing files, configuration, or secrets.

6. CreateContainerError

- **Cause**: The container fails to start due to an error, such as an application crash or incorrect image.
- **Fix**:
    - Check the container logs for application errors using kubectl logs <pod-name>.
    - Validate the container image and its entry point or command.

7. No Persistent Volume Claim Bound

- **Cause**: A PersistentVolumeClaim is not properly bound to a PersistentVolume.
- **Fix**:
    - Ensure the correct storageClass is used.
    - Check the status of the PVC and PV using kubectl get pvc and kubectl get pv.

8. Network Policies Block Traffic

- **Cause**: Network policies are preventing traffic from reaching pods or services.
- **Fix**:
    - Review the network policies with kubectl get networkpolicy.
    - Adjust the policies to allow necessary traffic.

9. ErrImagePull

- **Cause**: Kubernetes is unable to pull the image due to a wrong image name, tag, or access issues.
- **Fix**:
    - Ensure the image name and tag are correct.
    - Verify Docker registry access (for private registries, use imagePullSecrets).

10. Service Not Exposing the Application

- **Cause**: The service is not properly exposing the pods, or traffic is not being routed correctly.
- **Fix**:

    - Verify the service selector matches the pod labels.
    - Ensure that the service is of the correct type (e.g., LoadBalancer, NodePort, ClusterIP).

11. ResourceQuotaExceeded

- **Cause**: A namespace exceeds its resource quota (e.g., CPU, memory, or storage limits).
- **Fix**:
    - Check the resource quotas with kubectl describe quota.
    - Adjust resource requests/limits in the pod specification.

12. Port Conflicts

- **Cause**: Multiple containers are trying to bind to the same port on a node.
- **Fix**:
    - Ensure that each service/container is using unique ports or adjust the node port for the service.
    - Check kubectl get svc for port mappings.

13. CrashLoopBackOff Due to Application Misconfiguration

- **Cause**: The application inside the container fails to start due to missing configuration or incorrect environment variables.
- **Fix**:
    - Check the application logs using kubectl logs <pod-name>.
    - Ensure environment variables and application configuration are correct.

14. Incorrect Helm Chart Values

- **Cause**: Incorrect values in Helm charts lead to misconfigured deployments.
- **Fix**:
    - Review and update values.yaml with correct parameters.
    - Test the Helm deployment with helm install --dry-run.

15. Permissions Issues (RBAC)

- **Cause**: Insufficient permissions to access resources (e.g., pods, secrets, services).
- **Fix**:
    - Check the Role and RoleBinding resources (kubectl get roles and kubectl get rolebindings).
    - Grant appropriate permissions to service accounts or users.

16. Missing Secrets or ConfigMaps

- **Cause**: The pod is referencing a secret or config map that doesn't exist.
- **Fix**:
    - Ensure the secret or config map is created using kubectl create secret or kubectl create configmap.
    - Verify that the secret or config map is correctly referenced in the pod spec.

17. Timeout in Readiness and Liveness Probes

- **Cause**: The container's readiness or liveness probe times out, causing Kubernetes to mark it as unhealthy.
- **Fix**:
    - Review the probe configuration and increase the timeoutSeconds or initialDelaySeconds.
    - Ensure the container responds within the expected time.

18. Application Failures Due to Lack of Resources

- **Cause**: Pods fail due to insufficient memory or CPU, causing the application to terminate unexpectedly.
- **Fix**:
    - Adjust resource requests and limits in the pod spec.
    - Monitor cluster resources with kubectl top nodes and kubectl top pods.

19. Inconsistent Network Connectivity

- **Cause**: Pods are unable to communicate with each other or external services due to network misconfigurations.
- **Fix**:
    - Verify network policies and firewall settings.
    - Ensure that services, DNS, and ingress controllers are properly set up.

20. Invalid YAML Syntax

- **Cause**: The YAML configuration files have syntax errors that prevent Kubernetes resources from being created.
- **Fix**:
    - Use YAML validators or linters (e.g., kubectl apply --dry-run=client or online validators) to check the syntax.
    - Ensure proper indentation and structure for YAML files.

Summary of Fixes

- **Logs and Events**: Always start troubleshooting with kubectl describe <resource> and kubectl logs <pod-name>.
- **Resource Allocation**: Ensure adequate resources (memory, CPU) and proper requests and limits.
- **Container Image**: Verify the image name, tag, and registry access.
- **Configuration**: Double-check environment variables, config maps, secrets, and volume mounts.
- **Helm Charts**: If using Helm, ensure correct values are specified.
- **Network**: Ensure proper network policies and service exposure.

By addressing these common errors methodically and using the right tools, you can significantly reduce downtime and troubleshoot application deployment issues in Kubernetes more effectively.

Introduction to YAML for Kubernetes Users

YAML (Yet Another Markup Language) is widely used for configuration in Kubernetes. It is used to define and describe resources such as Pods, Services, Deployments, ConfigMaps, and more. Kubernetes uses YAML files to define the state of applications and resources within the cluster.

Kubernetes YAML files have the following general structure:

- **API Version**: The version of the Kubernetes API used to create the resource.
- **Kind**: The type of the resource being created (e.g., Pod, Deployment, Service).
- **Metadata**: Metadata about the resource (e.g., name, labels, annotations).
- **Spec**: The desired state specification of the resource.

Here, we'll go through the key aspects of YAML files used in Kubernetes, common syntax, and examples to help you become proficient in creating and managing Kubernetes resources.

---

1. Basic Structure of a Kubernetes YAML File

Here is the basic structure of a Kubernetes YAML file:

```
apiVersion: v1          # Specifies the API version
kind: Pod               # Defines the type of resource (e.g., Pod, Deployment)
metadata:
  name: my-pod          # The name of the resource
  labels:
    app: my-app         # Labels for the resource
spec:
  containers:
    - name: my-container  # The name of the container
      image: nginx:latest # The container image
      ports:
        - containerPort: 80
```

Key Sections:

- **apiVersion**: Defines the version of the Kubernetes API (e.g., v1, apps/v1).
- **kind**: Specifies the type of resource (e.g., Pod, Deployment, Service).
- **metadata**: Includes the name, labels, and annotations for the resource.
- **spec**: Contains the specification of the resource, such as container details, port configurations, etc.

2. Key Kubernetes Resources and Their YAML Definitions

Let's walk through some commonly used Kubernetes resources and their YAML configurations.

Pod

A **Pod** is the smallest deployable unit in Kubernetes. It is a collection of containers that share storage and network resources.

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
    - name: nginx-container
      image: nginx:latest
      ports:
        - containerPort: 80
```

Deployment

A **Deployment** is used to manage the deployment and scaling of pods. It ensures that a specified number of pods are running and automatically manages rolling updates.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx-container
          image: nginx:latest
```

```
    ports:
      - containerPort: 80
```

- **replicas**: Defines how many copies of the pod should run.
- **selector**: Defines how the deployment finds which pods to manage.
- **template**: Defines the pod template used by the deployment.

Service

A **Service** exposes a set of Pods to the network. It can be used for load balancing and providing access to pods.

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: ClusterIP
```

- **type**: Defines how the service is exposed (e.g., ClusterIP, NodePort, LoadBalancer).
- **selector**: Determines which pods this service targets.
- **ports**: Defines the ports for the service.

ConfigMap

A **ConfigMap** allows you to decouple environment-specific configurations from your application code.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-config
data:
  my-config-key: "my-config-value"
```

- **data**: Stores configuration data in key-value pairs.

Secret

A **Secret** is used to store sensitive information, such as passwords, OAuth tokens, or SSH keys.

```
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
type: Opaque
data:
  username: bXl1c2Vy
  password: bXlwYXNzd29yZA==
```

- **data**: Contains base64-encoded data (e.g., passwords).
- **type**: Defines the type of secret (e.g., Opaque for generic secrets).

Volume

A **Volume** provides a way for data to be shared between containers in a Pod. There are many types of volumes, such as emptyDir, hostPath, NFS, etc.

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-volume
spec:
  containers:
    - name: nginx-container
      image: nginx:latest
      volumeMounts:
        - mountPath: /data
          name: my-volume
  volumes:
    - name: my-volume
      emptyDir: {}
```

- **volumeMounts**: Defines where to mount the volume inside the container.
- **volumes**: Defines the volume and its type (e.g., emptyDir).

---

3. Understanding YAML Syntax

To work effectively with Kubernetes YAML files, it's essential to understand YAML syntax:

- **Indentation**: YAML uses spaces for indentation (not tabs). Typically, two spaces per level are used.
- **Key-Value Pairs**: Data is represented in key: value format.
- **Lists**: Lists are represented with a - (dash) followed by the list item
- **Dictionarys** are nested key-value pairs

Key-Value Pairs

- YAML files define data using key-value pairs: key: value
- The colon (:) separates the key and value, followed by a space.

Example:

```
name: my-app
version: v1.0
```

Lists

- Lists are defined using a hyphen (-).
- Each item in the list starts with a - and should be indented to the same level as the other items.

Example:

```
environments:
 - dev
 - staging
 - production

volumes:
 - name: my-volume
   emptyDir: {}
 - name: another-volume
   hostPath:
     path: /mnt/data
```

Dictionaries

- Nested dictionaries (key-value pairs inside another key-value pair) are used to represent hierarchical data.
- Indentation represents the nested structure.

Example:

```
database:
  host: db.example.com
  port: 5432
  credentials:
    username: user
    password: secret
```

- **Comments**: Comments in YAML are denoted by the # symbol.

```
# This is a comment
apiVersion: v1
```

- **Multi-line strings**: YAML allows multi-line strings using the pipe | for a block-style string or greater-than sign > for folded style strings.

Example of a block-style string:

```
message: |
  This is a multi-line
  string in YAML.
```

Example of a folded string:

```
message: >
  This is a long string
  that will be folded
  into a single line.
```

---

4. Kubernetes YAML Best Practices

1. **Keep YAML Files Modular**: For better organization, break down large configurations into smaller files.
2. **Use Descriptive Names**: Choose meaningful names for resources, labels, and annotations to easily identify their purpose.
3. **Version Control**: Store YAML files in version control systems (e.g., Git) to track changes over time.

4. **Use kubectl apply**: Instead of kubectl create, use kubectl apply -f <file>.yaml to ensure that changes to the configuration are applied declaratively.
5. **Use Templates for Reusability**: If using Helm or Kustomize, create reusable templates to standardize YAML files.
6. **Parameterize Values**: Use variables or config files to manage values that might change between environments.

---

5. Validating YAML Files

Before applying YAML files to your cluster, it's essential to validate them:

Use 'kubectl apply --dry-run': To check for syntax and validation errors without applying changes.

```
kubectl apply -f <file>.yaml --dry-run=client
```

1. **Online YAML Linters**: Use online tools (e.g., YAML Lint) to validate the syntax of your YAML file.

2. **Kubeconform**: You can use tools like https://github.com/yannh/kubeconform to validate your Kubernetes YAML files against a specific Kubernetes API schema. Once installed, use it like:

```
Kubeconform my-file.yaml
```

It will report any issues with the file and you can fix them if any.

---

6. Conclusion

Kubernetes YAML files are critical for managing resources in a Kubernetes cluster. By understanding YAML syntax and how to structure Kubernetes resource definitions, you can create, update, and maintain Kubernetes applications effectively.

You learned:

- The basic structure of Kubernetes YAML files.
- Examples of key Kubernetes resources (Pods, Deployments, Services, etc.).
- How to use YAML syntax and best practices.

With this knowledge, you can confidently write, manage, and debug your Kubernetes YAML configurations.

Lab: Using Annotations in Kubernetes

Objective:

Learn how to add annotations to Kubernetes resources and how annotations can be used for storing metadata.

Step 1: Create a Simple Pod

Let's start by creating a simple Pod. This Pod will be annotated with some metadata.

1. Create a YAML file named pod-with-annotations.yaml.

```
apiVersion: v1
kind: Pod
metadata:
  name: annotated-pod
  annotations:
    description: "This is a pod with annotations"
    created-by: "user@example.com"
spec:
  containers:
    - name: nginx
      image: nginx:latest
```

2. Apply this YAML file to create the pod:

```
kubectl apply -f pod-with-annotations.yaml
```

3. Verify the pod creation:

```
kubectl get pods
```

4. To view the annotations of the pod:

```
kubectl get pod annotated-pod -o=jsonpath='{.metadata.annotations}'
```

You should see the annotations you applied, like so:

```
{"description":"This is a pod with annotations","created-by":"user@example.com"}
```

Step 2: Adding Annotations to a Service

Now, let's add annotations to a Service in Kubernetes. This can be useful for tracking things like external documentation, maintenance information, or monitoring data.

1. Create a YAML file named service-with-annotations.yaml.

```
apiVersion: v1
kind: Service
metadata:
  name: annotated-service
  annotations:
    description: "This is a service with annotations"
    team: "dev-ops"
    purpose: "Expose nginx pod to the internet"
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: LoadBalancer
```

2. Apply this YAML file to create the Service:

```
kubectl apply -f service-with-annotations.yaml
```

3. To view the annotations of the service:

```
kubectl get service annotated-service -o=jsonpath='{.metadata.annotations}'
```

You should see:

```
{"description":"This is a service with annotations","team":"dev-ops","purpose":"Expose nginx pod to the internet"}
```

Step 3: Updating Annotations

You can also update annotations after creating resources. For example, let's add a new annotation to the existing Pod.

1. Update the pod annotations using kubectl:

```
kubectl annotate pod annotated-pod updated-by="admin@example.com"
--overwrite
```

2. Verify the updated annotations:

```
kubectl get pod annotated-pod -o=jsonpath='{.metadata.annotations}'
```

You should see the updated annotations:

```
{"description":"This is a pod with
annotations","created-by":"user@example.com","updated-by":"admin@example.com"}
```

Step 4: Use Annotations for External Tools

Annotations can be used by external tools, like monitoring and logging systems, to attach metadata. For example, you can attach version information for tracking:

1. Annotate the Pod with version information:

```
kubectl annotate pod annotated-pod version="v1.0.0" --overwrite
```

2. Verify the annotation:

```
kubectl get pod annotated-pod -o=jsonpath='{.metadata.annotations}'
```

Conclusion

In this lab, you've learned how to:

1. Add annotations to Kubernetes Pods and Services.
2. Update annotations for existing resources.
3. Retrieve annotations using kubectl.
4. Understand how annotations can be useful for attaching metadata for automation, monitoring, or documentation.

Annotations are a powerful way to store arbitrary metadata and can be used in many ways to enhance the functionality of your Kubernetes resources.

Lab: Using Annotations in Kubernetes Deployment for Rolling Updates

Objective:

When performing rolling updates in Kubernetes, the --record flag is often used to automatically record the command that triggered the change in the deployment's annotations. This is useful for keeping track of deployment history.

However, if you want to manually manage the history of rolling updates using annotations (instead of relying on --record), you can do so by manually updating the annotations on the deployment with relevant metadata each time you perform a rolling update. Here's how you can implement it:

Step-by-Step Guide: Using Annotations for Rolling Updates Instead of --record

---

Step 1: Create an Initial Deployment

We will start by creating a Kubernetes Deployment that runs an nginx container. We will annotate this deployment with a version history.

1. Create a YAML file named nginx-deployment-v1.yaml for the initial deployment:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  annotations:
    version: "v1.0.0"
    description: "Initial version of nginx deployment"
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
```

```
    labels:
      app: nginx
  spec:
    containers:
      - name: nginx
        image: nginx:1.19.0
        ports:
          - containerPort: 80
```

2. Apply this YAML file to create the Deployment:

```
kubectl apply -f nginx-deployment-v1.yaml
```

3. Verify the deployment and its annotations:

```
kubectl get deployment nginx-deployment -o=jsonpath='{.metadata.annotations}'
```

You should see the output with the annotations:

```
{"version":"v1.0.0","description":"Initial version of nginx deployment"}
```

4. kubectl annotate deploy/nginx-deployment kubernetes.io/change-cause="Image changed to 1.19.0"

5. kubectl set image deploy nginx-deployment nginx=nginx:1.24.0

6. Kubectl rollout history

Step 2: Perform a Rolling Update and Record Version History

Next, we'll perform a rolling update to upgrade the version of the nginx container. During this update, we will annotate the deployment with the new version.

1. Update the nginx-deployment to a new version (e.g., upgrade to nginx:1.20.0):

Create a new YAML file nginx-deployment-v2.yaml for the updated version:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
```

```
  annotations:
    version: "v2.0.0"
    description: "Updated nginx deployment to v2"
    updated-at: "{{timestamp}}"
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.20.0
          ports:
            - containerPort: 80
```

2. Apply the new deployment:

```
kubectl apply -f nginx-deployment-v2.yaml
```

3. Verify that the rolling update was applied:

```
kubectl get deployment nginx-deployment -o=jsonpath='{.metadata.annotations}'
```

You should see the updated annotations with the new version:

```
{"version":"v2.0.0","description":"Updated nginx deployment to
v2","updated-at":"{{timestamp}}"}
```

Note that {{timestamp}} would be replaced with the actual timestamp when you apply the YAML.

4. `kubectl annotate deploy/nginx-deployment kubernetes.io/change-cause="Image changed to 1.20.0"`

5. `kubectl set image deploy nginx-deployment nginx=nginx:1.20.0`

6. `Kubectl rollout history`

Step 3: Record Multiple Versions with Annotations

To track more detailed history, such as keeping a record of all versions and updates, we can append previous versions to the annotation. Let's update the annotations to append a version history list.

1. Update the deployment to record the version history. Modify the nginx-deployment-v2.yaml file to include an annotation that tracks the history:

File: nginx-deployment-v3.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  annotations:
    version-history: "v1.0.0,v2.0.0"
    version: "v2.0.0"
    description: "Updated nginx deployment to v2"
    updated-at: "{{timestamp}}"
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.20.0
          ports:
            - containerPort: 80
```

2. Apply the updated YAML:

```
kubectl apply -f nginx-deployment-v3.yaml
```

3. Verify the version history in the annotations:

```
kubectl get deployment nginx-deployment
-o=jsonpath='{.metadata.annotations.version-history}'
```

You should see the version history:

v1.0.0,v2.0.0

4. kubectl annotate deploy/nginx-deployment kubernetes.io/change-cause="Image changed to 1.20.0 v2.0.0"

5. kubectl set image deploy nginx-deployment nginx=nginx:1.20.0

6. Kubectl rollout history

This annotation now keeps a comma-separated list of versions, which helps you track the deployment history.

Step 4: Update Annotations for Future Versions

For each future rolling update, you should append the new version to the version-history annotation, so the history of deployments is preserved. For example, if we release version v3.0.0, we would modify the deployment YAML to include v1.0.0,v2.0.0,v3.0.0.

1. Update the nginx-deployment to version v3.0.0: nginx-deployment-v4.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  annotations:
    version-history: "v1.0.0,v2.0.0,v3.0.0"
    version: "v3.0.0"
    description: "Updated nginx deployment to v3"
    updated-at: "{{timestamp}}"
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
```

```
    - name: nginx
      image: nginx:1.21.0
      ports:
        - containerPort: 80
```

2. Apply the new version:

```
kubectl apply -f nginx-deployment-v4.yaml
```

3. Verify the updated version history:

```
kubectl get deployment nginx-deployment
-o=jsonpath='{.metadata.annotations.version-history}'
```

You should see:

V1.0.0,v2.0.0,v3.0.0

4. kubectl annotate deploy/nginx-deployment
kubernetes.io/change-cause="Image changed to 1.21.0"

5. kubectl set image deploy nginx-deployment nginx=nginx:1.21.0

6. Kubectl rollout history

Step 5: Cleanup

Once you've completed the lab, you can clean up the resources:

kubectl delete deployment nginx-deployment

Conclusion

In this lab, you've learned how to:

1. Create and annotate a Kubernetes deployment with a version number.
2. Perform rolling updates and track version history using annotations.
3. Append multiple versions to the annotations for future updates.
4. Use annotations to store metadata and deployment history.

This is a simple, manual way of tracking version history within Kubernetes deployments. In a production environment, you might want to automate this process using a CI/CD pipeline to manage the versions and annotations automatically.

Service Mesh

A **Service Mesh** in Kubernetes provides an infrastructure layer to manage the communication between microservices. It abstracts away the complexity of managing service-to-service interactions, making it easier to control, monitor, secure, and route traffic between services.

Here are the primary **uses of a service mesh in Kubernetes**:

1. Traffic Management

Service meshes enable advanced traffic routing and control between microservices. This includes:

- **Load balancing**: Distribute traffic evenly across instances of a service.
- **Routing rules**: Control traffic flow based on various conditions (e.g., headers, URL paths, versioning).
- **Canary deployments**: Gradually roll out new versions of services by routing a small percentage of traffic to the new version.
- **Blue-Green and A/B Testing**: Easily switch traffic between two versions of a service to test different releases.

For example, Istio, a popular service mesh, can route traffic dynamically based on version tags or metadata attached to services.

2. Service Discovery

Service meshes automate **service discovery** within a Kubernetes cluster. In a dynamic environment like Kubernetes where services are frequently created and destroyed, the service mesh keeps track of where each service is running and enables automatic discovery by other services. This reduces the need to manually configure service addresses and ports.

3. Observability & Monitoring

Service meshes provide deep visibility into microservices communication, including:

- **Metrics collection**: Collect application-level metrics (e.g., request count, latency, error rates) from each microservice.
- **Distributed tracing**: Trace requests as they travel across multiple services in the mesh, enabling root cause analysis when something goes wrong.
- **Logging**: Aggregated logs from all microservices to analyze system behavior.

With tools like **Prometheus**, **Grafana**, and **Jaeger** (commonly integrated with service meshes), you can easily visualize the health and performance of your services.

4. Resilience & Fault Tolerance

Service meshes help make your services more resilient to failures through:

- **Retries**: Automatically retry failed requests.
- **Timeouts**: Set timeouts for service calls to avoid hanging requests.
- **Circuit breaking**: Prevent cascading failures by automatically blocking requests to a service that is failing.
- **Rate limiting**: Control the rate of incoming requests to prevent service overload.

These features improve system stability and reduce the chances of service downtime.

5. Security

Security is one of the key features of a service mesh. It provides mechanisms to secure service-to-service communication within a Kubernetes cluster:

- **Mutual TLS (mTLS)**: Encrypt all communication between services and authenticate them to each other, ensuring that only authorized services can communicate.
- **Access control**: Define policies that restrict which services can communicate with each other (e.g., only allow certain services to communicate over HTTP, not gRPC).
- **End-to-End Encryption**: Encrypt traffic end-to-end, even if it traverses untrusted networks (e.g., between different cloud providers or hybrid environments).

Istio, for example, can enforce strong security policies and provide automatic certificate management.

6. Simplifying Microservices Communication

Microservices often involve multiple instances of services interacting with each other. Service meshes simplify this communication by abstracting the complexity of service-to-service interactions. Key benefits include:

- **Uniform API for communication**: Standardized service-to-service communication, regardless of the underlying application architecture.
- **Declarative management**: Configuring routing, retries, and other policies via simple YAML files that are easy to manage and audit.

- **Seamless scaling**: The service mesh automatically handles scaling services, adjusting routing and load balancing without requiring significant changes to application code.

7. Traffic Encryption & Data Integrity

Service meshes ensure that data in transit between services remains secure and tamper-proof. With features like mTLS and automated certificate management, the mesh:

- Encrypts traffic between services to protect sensitive data.
- Provides automatic certificate rotation to ensure the ongoing security of services.

8. Policy Enforcement

A service mesh allows for centralized enforcement of policies regarding how services should behave. These policies can include:

- **Access control**: Who can access what resources within the mesh.
- **Rate limiting**: Limiting the number of requests to prevent overload.
- **Resource quotas**: Limiting the resources (CPU, memory) allocated to each service.

This makes it easier to enforce organization-wide standards and improve governance.

9. Multi-Cluster and Hybrid Cloud Communication

Service meshes are useful in environments where applications span multiple clusters or even multiple cloud environments. They enable:

- **Multi-cluster communication**: Services in different clusters can communicate securely and efficiently, without needing complex networking configurations.
- **Hybrid cloud deployments**: Allowing services to run across on-premises and cloud infrastructure, while ensuring seamless communication and traffic routing.

10. Advanced Routing Capabilities

With a service mesh, you can:

- **Perform advanced routing** based on HTTP headers, query parameters, or other contextual information.

- **Split traffic** across multiple versions of a service (canary deployments, blue-green deployments).
- **Gracefully handle failures** with retries, circuit breakers, and fallbacks.

This helps to ensure that the user experience remains smooth even during updates, failures, or traffic shifts.

11. Application Performance Optimization

Service meshes optimize the performance of applications by:

- **Reducing latencies** through intelligent load balancing and routing decisions.
- **Improving throughput** by controlling the rate of requests and preventing bottlenecks.

Popular Service Meshes for Kubernetes

- **Istio**: One of the most popular service meshes. It provides advanced features like traffic management, security, observability, and policy enforcement.
- **Linkerd**: A simpler, lighter-weight service mesh compared to Istio. It's focused on ease of use and performance.
- **Consul**: Provides service discovery and configuration, along with service mesh capabilities for Kubernetes.
- **Kuma**: A lightweight and universal service mesh designed to handle microservices in multi-cloud and hybrid environments.

Conclusion

A **Service Mesh** in Kubernetes, such as **Istio**, adds significant value by providing capabilities that allow you to:

- Manage and secure service-to-service communication.
- Gain visibility into microservices interactions.
- Control traffic flows with advanced routing.
- Ensure the resilience and security of your microservices architecture.

By abstracting away the complexity of inter-service communication, service meshes help improve the efficiency, reliability, and security of microservices in a Kubernetes environment.