Linux Web Services: Milestone 2 - Extensive Deployment Guide

1. Introduction

This document serves as a comprehensive guide for the deployment of a three-tier web application stack. The project demonstrates the progression from local development using Docker Compose to a production-like orchestration using Kubernetes (KIND) and GitOps principles with ArgoCD.

The goal is to deploy a web stack consisting of:

1. **Frontend**: A lightweight web server hosting the user interface. The frontend consists of Lighttpd
2. **Backend (API)**: A Python FastAPI application processing logic and connecting to the database.
3. **Database**: A MariaDB instance for persistent data storage.

We will explore two major phases:

1. **Containerization & Local Testing**: Using Docker and Docker Compose to validate the application logic and container builds.
2. **Orchestration**: Deploying the stack to a multi-node Kubernetes cluster, managing traffic with Ingress, and automating deployment with ArgoCD.

---

2. Theoretical Background

2.1 Docker & Docker Compose

Docker allows us to package applications and their dependencies into "containers". This ensures that the application runs consistently on any environment, solving the "it works on my machine" problem.

Docker Compose is a tool for defining and running multi-container Docker applications. It allows us to define the entire stack (frontend, api, db) in a single YAML file.

- **Why start with Docker Compose?**
  - **Simplicity**: It is easier to write one `docker-compose.yaml` than multiple Kubernetes manifests.
  - **Rapid Feedback**: The "build-run-debug" loop is much faster locally.
  - **Validation**: It proves that the containers work and can talk to each other before introducing the complexity of Kubernetes networking (Services, Ingress, DNS).

2.2 Kubernetes (K8s) & KIND

Kubernetes is an orchestration platform for automating deployment, scaling, and management of containerized applications. Unlike Docker Compose, which is meant for a single host, Kubernetes manages clusters of computers.

KIND (Kubernetes IN Docker) allows us to run a local Kubernetes cluster by using Docker containers as "nodes". This is perfect for learning and testing Kubernetes without expensive cloud infrastructure.

---

3. Step 1: Containerization and Docker Compose

In this phase, we define the environment for each component and run them together. We do this to test the logic for the containers and docker images, making sure they behave as they should.

3.1 The Backend (API)

The API is built with **FastAPI**, a modern, fast (high-performance) web framework for building APIs with Python.

File: `api/requirements.txt` This file lists the Python libraries required by our application.

```
fastapi==0.104.1
uvicorn==0.24.0
mysql-connector-python==8.1.0
pydantic==2.4.2
```

File: `api/app.py` This is the main application logic. It defines the API endpoints and handles database connections.

```
# this section is used for imports of various dependancies used in the fastAPI API
from fastapi import FastAPI, HTTPException, Response
from fastapi.middleware.cors import CORSMiddleware
import mysql.connector
import os
import socket
import datetime
from pydantic import BaseModel
from typing import Dict, Any

# Defining the fastAPI app
app = FastAPI(title="Milestone2 API", version="1.0.0")

# CORS middleware
'''
this block is used to describe Cross Origin Resource Sharing. CORS is used to
allow requests to be made to various different resources defined inside the CORS
middleware block
Because this is a testing environment, and it will not be used in production, I've
decided to allow all traffic to go through, even tho this is not a best practice
and should NEVER be used inside a production environment.
'''
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# Database configuration
```

```
'''
In this brief section of code, all variables from the dockerfile following below
are imported into the python script. These variables are used to connect to the
MariaDB database
'''
DB_CONFIG = {
    'host': os.getenv('DB_HOST'),
    'user': os.getenv('DB_USER'),
    'password': os.getenv('DB_PASSWORD'),
    'database': os.getenv('DB_NAME'),
    'port': os.getenv('DB_PORT')
}

def get_db_connection():
    """Create database connection"""
    return mysql.connector.connect(**DB_CONFIG)

def init_database():
    """Fucntion used to initialise the database, should it not exist or should
there not be any data inside the database, as to not throw an error when loading
the webpage but always display a name"""
    try:
        # Try to establish a connection to the database
        conn = get_db_connection()
        cursor = conn.cursor()

        # Create table if not exists
        cursor.execute("""
            CREATE TABLE IF NOT EXISTS users (
                id INT AUTO_INCREMENT PRIMARY KEY,
                name VARCHAR(255) NOT NULL DEFAULT 'John Doe',
                created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
            )
        """)

        # Insert default user if table is empty
        cursor.execute("SELECT COUNT(*) FROM users")
        if cursor.fetchone()[0] == 0:
            cursor.execute("INSERT INTO users (name) VALUES ('John Doe')")
        # Commit the SQL query to create and populate the database should it not
exist
        conn.commit()
        # close the connection to the database
        cursor.close()
        conn.close()

        # a debug print to let administrators or maitenance know the database was
initialised (Fallback incase init.sql failed)
        print("Database initialized successfully")
    # capturing exeptions meaning should the connection fail, or throw an error
    except Exception as e:
        # debug print for maitenance or sysadmins to let them know what exactly
happened
        print(f"Database initialization failed: {e}")
```

```python
# Define what the FastAPI app should do on startup
@app.on_event("startup")
async def startup_event():
    """Initialize database on startup"""
    init_database()


# Endpoint 1: root endpoint that retuns in JSON format a message (used as some
kind of 'sanity check')
@app.get("/")
async def root():
    return {"message": "Milestone2 API is running"}


# Endpoint 2: This one is used for the healthcheck inside the dockerfile
@app.get("/health")
async def health_check():
    """Health check endpoint"""
    try:
        # try to get a connection to the database
        conn = get_db_connection()
        conn.close()

        # returns the status of the container as being healthty is a connection is
made
        return {"status": "healthy", "database": "connected"}
    # capture exeptions should the connection fail
    except Exception as e:

        # let the healthcheck know that the database connection failed, and also
provide a http page with an error code when accessing the API directly
        raise HTTPException(status_code=503, detail=f"Database connection failed:
{e}")


# Endpoint 3: Getting the user from the database
@app.get("/user")
async def get_user():
    """Get user name from database"""
    try:
        # try to get a connection to the database
        conn = get_db_connection()
        cursor = conn.cursor(dictionary=True)
        # execute a SQL query to get the first name inside the database
        cursor.execute("SELECT name FROM users WHERE id = 1")

        # load the result of the query inside a variable
        result = cursor.fetchone()
        # close the connection to the database
        cursor.close()
        conn.close()

        # If there is a name, return the name
```

```
        if result:
            return {"name": result['name']}
        # if there is no name, return error 404 with the text that indicates that
the database is empty
        else:
            raise HTTPException(status_code=404, detail="User not found")
    # capture exeptions to let sysadmins know the connection to the database
failed
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Database error: {e}")

# Endpoint 4: Getting the container ID -> used in the multi-node clusters to show
that it actually changes the node being used(nodes in KIND are just docker
containers)
@app.get("/container_id")
async def get_container_id(response: Response):
    """Get container ID (hostname)"""
    response.headers["Connection"] = "close"
    # Prevent browser/proxy caching so each refresh actually hits the backend
    response.headers["Cache-Control"] = "no-store, no-cache, must-revalidate, max-
age=0"
    response.headers["Pragma"] = "no-cache"
    response.headers["Expires"] = "0"

    #return the hostname using the socket library in python
    return {"container_id": socket.gethostname(), "timestamp":
datetime.datetime.utcnow().isoformat() + "Z"}

# Endpoint 5: allowing the changing of the name in the database using a POST
request
@app.put("/user/{name}")
async def update_user(name: str):
    """Update user name"""
    try:
        conn = get_db_connection()
        cursor = conn.cursor()
        cursor.execute("UPDATE users SET name = %s WHERE id = 1", (name,))
        conn.commit()
        cursor.close()
        conn.close()
        # let the user know the change was successful
        return {"message": "User updated successfully", "name": name}
    # capture exeptions to let sysadmins know the connection to the database
failed
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Database error: {e}")
# only allow the app to run when it is being run as the main file and not as a
library -> security to not let it loop over itself, creating a memory overflow or
a stack error
if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

- **Explanation**: The code initializes a FastAPI app. Crucially, it reads database credentials from **Environment Variables** (`os.getenv`). This is a best practice (The 12-Factor App) because it allows us to change configuration without changing the code.

File: `api/Dockerfile` This file tells Docker how to build the API image.

```
# using the official python image for simplicity
FROM python:3.9-slim

WORKDIR /app

# Install system dependencies
RUN apt-get update && apt-get install -y \
    gcc \
    && rm -rf /var/lib/apt/lists/*

# Copy requirements and install Python dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy application code
COPY app.py .

# Create non-root user
RUN useradd -m -u 1000 apiuser && chown -R apiuser:apiuser /app
USER apiuser

# Health check
HEALTHCHECK --interval=30s --timeout=10s --start-period=5s --retries=3 \
    CMD curl -f http://localhost:8000/health || exit 1
# expose the port being used by the API
EXPOSE 8000

# command that runs when the container starts
CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "8000"]
```

- `FROM python:3.9-slim`: Uses a lightweight Python base image.
- `WORKDIR /app`: Sets the working directory inside the container.
- `RUN ...`: Installs system dependencies (gcc) needed for some Python packages.
- `COPY ...`: Moves files from our host to the container.
- `USER apiuser`: Runs the process as a non-root user. This is a critical security practice to prevent potential container breakouts.
- `HEALTHCHECK`: Docker will periodically run this command to see if the app is alive.
- `CMD`: The command that runs when the container starts (`uvicorn` is the server).

3.2 The Frontend

The frontend is a static HTML page served by **Lighttpd**, a very lightweight web server.

File: `frontend/index.html` The main interface for the user.

```
<!DOCTYPE html>
<html lang="en">
<!-- basic HTML and inline JS -->
<head>
  <meta charset="UTF-8" />
  <meta http-equiv="X-UA-Compatible" content="IE=edge" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Milestone 2</title>
</head>

<body>
  <h1><span id="user">Loading...</span> has reached milestone 2!</h1>
  <p>Container ID: <span id="container_id">Loading...</span></p>

  <script>
    // Fetch user from API
    fetch("http://localhost/api/user")
      .then((res) => res.json())
      .then((data) => {
        // Get user name
        const user = data.name;
        // Display user name
        document.getElementById("user").innerText = user;
      })
    //   catching errors and logging them to the console
      .catch((error) => console.error("Error fetching user:", error));

    // Fetch container ID from API
    fetch("http://localhost/api/container_id")
      .then((res) => res.json())
      .then((data) => {
        // Get container ID
        const containerId = data.container_id;
        // Display container ID
        document.getElementById("container_id").innerText = containerId;
      })
    //   catching errrors and logging them to the console
      .catch((error) => console.error("Error fetching container ID:", error));
  </script>
</body>

</html>
```

File: `frontend/lighttpd.conf` Configuration for the web server.

```
; define the root folder being used, here we will store all webpages
server.document-root = "/var/www/localhost/htdocs"
; defining what port the lighttpd service will open and run from
```

```
server.port = 80

; defining the user and group the service should run as
server.username = "lighttpd"
server.groupname = "lighttpd"

; modules being used by the lighttpd service
server.modules = (
    ; handles default index files
    "mod_indexfile",
    # provide access control rules
    "mod_access",
    ; allows for the remapping of URL paths to different filesystem locations
    "mod_alias",
    ; enables redirection via regex (regular expression)
    "mod_redirect",
)
; loook for index.html when accessing the root directory
index-file.names = ("index.html")

; how to display files using their extensions
mimetype.assign = (
    ".html" => "text/html",
    ".js" => "application/javascript",
    ".css" => "text/css",
)
; disallow files with the following extensions to be run as static webpages
static-file.exclude-extensions = ( ".php", ".pl", ".fcgi" )
```

File: `frontend/Dockerfile`

```
# using a more maintained and more stable image, the official Lighttpd image was
more then 2y old
FROM alpine:3.16

# Install lighttpd and curl for health checks
RUN apk add --no-cache lighttpd curl

# Copy custom lighttpd configuration
COPY lighttpd.conf /etc/lighttpd/lighttpd.conf

# Copy web content

COPY index.html /var/www/localhost/htdocs/

# Create a simple health check
RUN echo '#!/bin/sh' > /healthcheck.sh && \
    echo 'curl -f http://localhost/ || exit 1' >> /healthcheck.sh && \
    chmod +x /healthcheck.sh

# Expose port 80
```

```
EXPOSE 80

# Start lighttpd in foreground mode
CMD ["lighttpd", "-D", "-f", "/etc/lighttpd/lighttpd.conf"]
# run the healthcheck every 30 seconds, starting 5 seconds after the container is
running, with a 3s failure time, and 3 retries before marking the container as
unhealthy
HEALTHCHECK --interval=30s --timeout=3s --start-period=5s --retries=3 \
  CMD /healthcheck.sh
```

- `FROM alpine:3.16`: Alpine is an extremely small Linux distribution (approx 5MB), making our image very efficient.
- `apk add`: Alpine's package manager (like `apt` or `yum`).
- `CMD ["lighttpd", "-D", ...]`: Runs Lighttpd in the foreground (`-D`) so the container keeps running.

3.3 The Database

We use a standard **MariaDB** image and initialize it with a SQL script.

File: `db/init.sql`

```
-- Initialize database
CREATE DATABASE IF NOT EXISTS milestone2;
USE milestone2;

-- Create users table
CREATE TABLE IF NOT EXISTS users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL DEFAULT 'Test Test',
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Insert initial data
INSERT IGNORE INTO users (id, name) VALUES (1, 'Len Vogels');
```

- **Explanation**: This script runs automatically when the database container starts for the first time, creating our schema and initial data.

3.4 Orchestrating with Docker Compose

File: `docker-compose.yaml`

```
# version is not really necessairy anymore
version: '3.8'


# let the docker engine know what services (containers) to make
services:
#   frontend container
```

```
  frontend:
#   look for the buildfile in the ./frontend directory
    build: ./frontend
    # give the container a name
    container_name: lv-ms2-frontend
    # map the exposed ports
    ports:
      - "8080:80"
    # only allow the frontend container to start when the api container exists
    depends_on:
      - api
    # define that the container needs to use the network used to communicate
between the containers
    networks:
      - ms2-network
    #   persistence of the contaienr, if it should turn unhealty it will
automatically restart, and only stop with that behaviour when stopped by the user
    restart: unless-stopped
# APi container
  api:
#   look for the buildfile inside the ./api container
    build: ./api
    # give the container a name
    container_name: lv-ms2-api
    # map the ports being used by the container
    ports:
      - "8000:8000"
    #   defining the environment variables needed to access the backend, these are
used in the python script above
    environment:
      - DB_HOST=db
      - DB_USER=root
      - DB_PASSWORD=milestone2
      - DB_NAME=milestone2
      - DB_PORT=3306
    # only allow the API container to start when the database container is healty
    depends_on:
      db:
        condition: service_healthy

    # define that the container needs to use the network used to communicate
between the containers
    networks:
      - ms2-network
    #   persistence of the contaienr, if it should turn unhealty it will
automatically restart, and only stop with that behaviour when stopped by the user
    restart: unless-stopped
    # running a healthcheck every 30s, with failure time of 10s with 3 retries,
starting 40s after creation of the container
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8000/health"]
      interval: 30s
      timeout: 10s
      retries: 3
```

```
      start_period: 40s
# creation of the database container
  db:
#   using the official mariadb image
    image: mariadb:11
    # naming the container
    container_name: lv-ms2-db
    # defining environment variables to access the mariadb root user in order to
make the database
    environment:
      MYSQL_ROOT_PASSWORD: milestone2
      MYSQL_DATABASE: milestone2
# defining volumes used by the container
    volumes:
    # defining a named volume mounted to the place where mariadb stores the
database files for persistent memory, even after container destruction
      - db_data:/var/lib/mysql
    #   mounts a read only file to the mariadb initialisation directory, this will
only run if db_data does not exist
      - ./db/init.sql:/docker-entrypoint-initdb.d/init.sql:ro
    # define that the container needs to use the network used to communicate
between the containers
    networks:
      - ms2-network
    #   persistence of the contaienr, if it should turn unhealty it will
automatically restart, and only stop with that behaviour when stopped by the user
    restart: unless-stopped
    # running a healthcheck every 10s, with failure time of 5s with 10 retries,
starting 300s after creation of the container
    healthcheck:
      test: ["CMD", "mariadb-admin", "ping", "-h", "127.0.0.1", "-uroot", "-
pmilestone2"]
      interval: 10s
      timeout: 5s
      retries: 10
      start_period: 30s

# creating the network being used for the containers to internally communicate
networks:
  ms2-network:
    driver: bridge
# creating the named folume for persistence
volumes:
  db_data:
```

- `services`: Defines the containers we want to run.
- `build: ./frontend`: Tells Compose to build the image from the `./frontend` directory instead of pulling it.
- `ports`: Maps host ports to container ports (`"8080:80"` means accessing localhost:8080 goes to container port 80).
- `depends_on`: Ensures start order. The API waits for the DB to be "healthy" before starting.

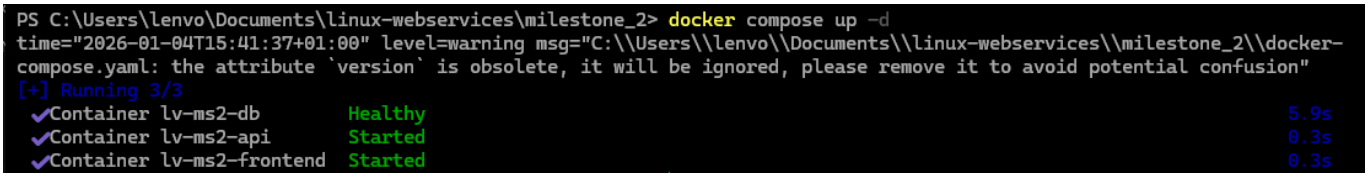
- `networks`: All services join `ms2-network` so they can talk to each other by name (e.g., API connects to `db`).
- `volumes`: `db_data` persists the database files so data isn't lost when the container stops.

3.5 Running the Stack

To start the application, run:

```
docker-compose up --build
```

- `up`: Create and start containers.
- `--build`: Force a rebuild of images (useful if you changed code).



You can now visit `http://localhost:8080` to see the application running.

4. Step 2: Kubernetes Deployment (KIND)

Now that we know the app works, we move to Kubernetes. This adds self-healing (restarting crashed pods), scaling (running multiple copies), and advanced networking. I've used KinD because my host machine is Windows based and the integration with KinD is best, because KinD uses, as the name suggests, docker to run the clusters, where each pod is a new container

4.1 Cluster Configuration

We need a cluster that supports an **Ingress Controller** (a reverse proxy that routes traffic).

File: `kind-config.yaml`

```
# tell Kind to create a cluster object
kind: Cluster
# default kind config schema
apiVersion: kind.x-k8s.io/v1alpha4
# name of the cluster
name: ms2
# nodes that should be created
nodes:
# node 1: contol plane
- role: control-plane
# patches for ingress to work properly, giving this node a tag that ingress will look for
  kubeadmConfigPatches:
  - |
    kind: InitConfiguration
    nodeRegistration:
```

```
        kubeletExtraArgs:
          node-labels: "ingress-ready=true"
  # mapping of control plane ports to host machine ports, for access on the host
  machine via localhost
    extraPortMappings:
    - containerPort: 80
      hostPort: 80
      protocol: TCP
    - containerPort: 443
      hostPort: 443
      protocol: TCP
    - containerPort: 8080
      protocol: TCP
      hostPort: 8080
  - role: worker
  - role: worker
```

- `nodes`: We define 1 control-plane (master) and 2 workers.
- `extraPortMappings`: This is crucial. It maps port 80 on your *host machine* to port 80 on the *control-plane container*. This allows the Ingress controller inside the cluster to receive traffic from your browser.
- `node-labels`: Tags the node so the Ingress controller knows where to run.

Command: Create Cluster

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

- `--config`: Tells KIND to use our custom configuration file.

```
PS C:\Users\lenvo\Documents\linux-webservices\milestone_2> kind create cluster --config kind-config.yaml
Creating cluster "ms2" ...
 ✓ Ensuring node image (kindest/node:v1.33.1) 🖼
 ✓ Preparing nodes 📦 📦 📦
 ✓ Writing configuration 📜
 ✓ Starting control-plane 🕹️
 ✓ Installing CNI 🔌
 ✓ Installing StorageClass 💾
 ✓ Joining worker nodes 🚜
Set kubectl context to "kind-ms2"
You can now use your cluster with:

kubectl cluster-info --context kind-ms2

Thanks for using kind! 😊
```

4.2 Setting up Ingress

We install NGINX Ingress Controller.

```
# applies the ingress yaml file from github to the control plane
kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-
nginx/main/deploy/static/provider/kind/deploy.yaml
```

- `kubectl apply`: The universal command to create/update resources in Kubernetes.

Wait for it to be ready:

```
#waiting for the ingress pod to be ready before continuing
kubectl wait --namespace ingress-nginx --for=condition=ready pod --
selector=app.kubernetes.io/component=controller --timeout=120s
```

Patch the Controller: We need to ensure the Ingress controller runs on the specific node we prepared in `kind-config.yaml`.

```
kubectl patch deployment -n ingress-nginx ingress-nginx-controller -p '{\"spec\":
{\"template\":{\"spec\":{\"nodeSelector\":{\"ingress-ready\":\"true\"}}}}}'```
![alt text](image-2.png)


### 4.3 Kubernetes Manifests

We define our application state using YAML files.

**1. Storage (PVC & ConfigMap)**
**File: `k8s/storage.yaml`**
```yaml

# Specifies the version of the Kubernetes API to use (v1 is for core resources)
apiVersion: v1

# Defines the type of resource: a PersistentVolumeClaim (request for storage)
kind: PersistentVolumeClaim
# Metadata section to identify and label the resource
metadata:
  # The name of this claim, which Pods will use to request this storage
  name: db-data-pvc
# The specification of the desired storage behavior
spec:
  # Defines how the volume can be mounted
  accessModes:
    # 'ReadWriteOnce' means the volume can be mounted as read-write by a single
node
    - ReadWriteOnce
  # Specifies the compute resources (storage) required
  resources:
    # The minimum amount of resources to reserve
    requests:
      # Requests exactly 1 Gigabyte of storage space
      storage: 1Gi
---
# Separator to define a new resource in the same file
# Again, using the core v1 API version
apiVersion: v1
```
```

```
# Defines a ConfigMap resource, used to store non-sensitive configuration data
kind: ConfigMap
# Metadata section for the ConfigMap
metadata:
  # The name 'db-init-script' allows us to reference this map in the Deployment
  name: db-init-script
# The actual data dictionary to store
data:
  # The key name for the file is 'init.sql'. The pipe (|) indicates a multi-line
string follows
  init.sql: |
    # SQL command to create the database if it doesn't already exist
    CREATE DATABASE IF NOT EXISTS milestone2;
    # SQL command to select the 'milestone2' database for subsequent operations
    USE milestone2;

    -- Create users table
    CREATE TABLE IF NOT EXISTS users (
        id INT AUTO_INCREMENT PRIMARY KEY,
        name VARCHAR(255) NOT NULL DEFAULT 'Test Test',
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
    );

    -- Insert initial data
    # SQL command to insert a row, ignoring errors if the ID (1) already exists
    INSERT IGNORE INTO users (id, name) VALUES (1, 'Len Vogels');
```

- `PersistentVolumeClaim` **(PVC)**: Requests 1GB of storage. K8s will provision this so our DB data persists.
- `ConfigMap`: Stores the `init.sql` script. This allows us to inject the script into the DB container without rebuilding the image.

2. Database Deployment File: `k8s/db-deployment.yaml`

```
# Specifies the apps/v1 API version, which is standard for Deployments
apiVersion: apps/v1
# Defines a Deployment resource to manage stateless (or stateful) applications
kind: Deployment
# Metadata for the Deployment itself
metadata:
  # The name of the deployment in the cluster
  name: lv-ms2-db
# Specification of the desired behavior of the Deployment
spec:
  # The number of desired Pod instances (1 database instance)
  replicas: 1
  # Defines how the Deployment identifies which Pods it manages
  selector:
    # It looks for Pods with labels matching the key-value pairs below
    matchLabels:
      # Matches Pods with the label 'app' set to 'lv-ms2-db'
```

```
      app: lv-ms2-db
  # The template used to create new Pods
  template:
    # Metadata for the Pods created by this template
    metadata:
      # Labels attached to the Pods (must match the selector above)
      labels:
        # Sets the label 'app' to 'lv-ms2-db'
        app: lv-ms2-db
    # Specification for the Pod's contents
    spec:
      # List of containers to run inside the Pod
      containers:
      # Name of the container (used for logs and internal reference)
      - name: mariadb
        # The Docker image to use (MariaDB version 11)
        image: mariadb:11
        # List of ports to expose from the container
        ports:
        # The port the container listens on (standard MySQL port)
        - containerPort: 3306
        # Environment variables to pass into the container
        env:
        # Variable name for the root password (required by MariaDB image)
        - name: MYSQL_ROOT_PASSWORD
          # The actual password value
          value: "milestone2"
        # Variable to automatically create a database on startup
        - name: MYSQL_DATABASE
          # The name of the database to create
          value: "milestone2"
        # Variable to configure the TCP port MariaDB listens on
        - name: MYSQL_TCP_PORT
          # Explicitly setting it to 3306
          value: "3306"
        # Defines where to mount volumes inside the container filesystem
        volumeMounts:
        # References the 'db-data' volume defined in the 'volumes' section
        - name: db-data
          # The path inside the container where the volume appears (standard MySQL
data path)
          mountPath: /var/lib/mysql
        # References the 'init-script' volume
        - name: init-script
          # Path where MariaDB looks for initialization scripts
          mountPath: /docker-entrypoint-initdb.d
        # Configuration for checking if the container is alive (running)
        livenessProbe:
          # Use an execution command to check health
          exec:
            # The command arguments
            command:
            # Use the shell executable
            - /bin/sh
```

```
            # Flag to execute the following string
            - -c
            # The actual command: ping the database using the mariadb-admin tool
            - "mariadb-admin ping -uroot -pmilestone2"
          # Wait 30 seconds before the first check (gives DB time to start)
          initialDelaySeconds: 30
          # Perform the check every 10 seconds
          periodSeconds: 10
          # Fail the check if it takes longer than 5 seconds
          timeoutSeconds: 5
        # Configuration for checking if the container is ready to accept traffic
        readinessProbe:
          # Use an execution command
          exec:
            # The command arguments
            command:
            # Use the shell
            - /bin/sh
            # Execute the string
            - -c
            # Same ping command to verify DB is responsive
            - "mariadb-admin ping -uroot -pmilestone2"
          # Wait 15 seconds before the first check
          initialDelaySeconds: 15
          # Perform the check every 5 seconds
          periodSeconds: 5
          # Fail if it takes longer than 5 seconds
          timeoutSeconds: 5
      # Define the underlying storage volumes available to the Pod
      volumes:
      # Name of the volume (referenced in volumeMounts above)
      - name: db-data
        # Specifies this volume uses a PersistentVolumeClaim
        persistentVolumeClaim:
          # The name of the PVC defined in the previous file
          claimName: db-data-pvc
      # Name of the second volume
      - name: init-script
        # Specifies this volume comes from a ConfigMap
        configMap:
          # The name of the ConfigMap defined in the previous file
          name: db-init-script
---
# Specifies the core v1 API for the Service resource
apiVersion: v1
# Defines a Service (provides a stable network endpoint/IP)
kind: Service
# Metadata for the Service
metadata:
  # The DNS name of the service (other Pods reach it via this name)
  name: lv-ms2-db-service
# Specification of the Service behavior
spec:
  # Defines which Pods this Service directs traffic to
```

```
  selector:
    # Targets Pods with the label 'app: lv-ms2-db'
    app: lv-ms2-db
  # List of ports managed by this Service
  ports:
  # Name for this port mapping
  - name: mysql
    # The port the Service exposes internally within the cluster
    port: 3306
    # The port on the target Container to forward traffic to
    targetPort: 3306
```

- `Deployment`: Manages the Pods. `replicas: 1` ensures 1 DB instance is running.
- `Service`: Exposes the DB to other pods.
    - `name: lv-ms2-db-service`: This becomes the DNS name. The API will connect to this host.

3. API Deployment File: `k8s/api-deployment.yaml`

```
# Standard API version for Deployments
apiVersion: apps/v1
# Defines a Deployment resource
kind: Deployment
# Metadata for the API deployment
metadata:
  # Name of the deployment
  name: lv-ms2-api
# Specs for the API deployment
spec:
  # Requests 2 identical copies of the pod for high availability
  replicas: 2
  # Selector to manage the pods
  selector:
    # Matches pods with the label 'app: lv-ms2-api'
    matchLabels:
      app: lv-ms2-api
  # Template for creating the API pods
  template:
    # Metadata for the individual pods
    metadata:
      # Labels applied to the pods
      labels:
        app: lv-ms2-api
    # Container specification
    spec:
      # List of containers
      containers:
      # Name of the container
      - name: api
        # The specific image to pull from Docker Hub
        image: lenv1891/lv-ms2-api:latest
        # Forces Kubernetes to pull the image even if it exists locally
```

```
        imagePullPolicy: Always
        # Ports exposed by the container
        ports:
        # The API listens on port 8000
        - containerPort: 8000
        # Environment variables for configuration
        env:
        # Variable for the Database Hostname
        - name: DB_HOST
          # Value matches the Service name of the Database defined earlier
          value: "lv-ms2-db-service"
        # Variable for the Database User
        - name: DB_USER
          # Value is 'root'
          value: "root"
        # Variable for the Database Password
        - name: DB_PASSWORD
          # Value matches the password set in the DB deployment
          value: "milestone2"
        # Variable for the Database Name
        - name: DB_NAME
          # Value matches the DB created in the init script
          value: "milestone2"
        # Variable for the Database Port
        - name: DB_PORT
          # Value is 3306
          value: "3306"
        # Liveness probe to check if the application is crashed/dead
        livenessProbe:
          # Uses an HTTP GET request instead of a command
          httpGet:
            # The endpoint to check
            path: /health
            # The port to send the request to
            port: 8000
          # Wait 40 seconds before first check (API startup time)
          initialDelaySeconds: 40
          # Check every 30 seconds
          periodSeconds: 30
          # Fail if response takes longer than 10 seconds
          timeoutSeconds: 10
        # Readiness probe to check if app is ready for traffic
        readinessProbe:
          # Uses HTTP GET
          httpGet:
            # The endpoint to check
            path: /health
            # The port to check
            port: 8000
          # Wait 5 seconds before first check
          initialDelaySeconds: 5
          # Check every 5 seconds
          periodSeconds: 5
          # Fail if response takes longer than 5 seconds
```

```
          timeoutSeconds: 5
---
# Separator for the Service resource
apiVersion: v1
# Defines a Service
kind: Service
# Metadata for the Service
metadata:
  # DNS name for the API service
  name: lv-ms2-api-service
# Service specification
spec:
  # Selector to find the API pods
  selector:
    app: lv-ms2-api
  # Ports configuration
  ports:
  # The port exposed by the Service inside the cluster
  - port: 8000
    # The port on the container to forward to
    targetPort: 8000
```

- `replicas: 2`: We run **2 copies** of the API for high availability. If one crashes, the other handles traffic.
- `env`: We pass the DB connection details. Note `DB_HOST` is `lv-ms2-db-service` (the K8s Service name).
- `livenessProbe`: K8s checks `/health`. If it fails, K8s kills and restarts the pod.
- `readinessProbe`: K8s checks `/health`. If it fails, K8s stops sending traffic to this pod until it recovers.

4. Frontend Deployment File: `k8s/frontend-deployment.yaml`

```
# Standard API version for Deployments
apiVersion: apps/v1
# Defines a Deployment resource
kind: Deployment
# Metadata for the Frontend deployment
metadata:
  # Name of the deployment
  name: lv-ms2-frontend
# Specs for the Frontend
spec:
  # Only 1 replica needed for the frontend
  replicas: 1
  # Selector to find the frontend pods
  selector:
    # Matches pods with label 'app: lv-ms2-frontend'
    matchLabels:
      app: lv-ms2-frontend
  # Template for creating the pods
  template:
    # Metadata for the pods
    metadata:
      # Labels applied to the pods
```

```
      labels:
        app: lv-ms2-frontend
    # Pod specification
    spec:
      # List of containers
      containers:
      # Name of the container
      - name: frontend
        # Image to use
        image: lenv1891/lv-ms2-frontend:latest
        # Always pull the latest image
        imagePullPolicy: Always
        # Ports exposed
        ports:
        # Nginx/Web server usually listens on port 80
        - containerPort: 80
        # Liveness probe to check if Nginx is running
        livenessProbe:
          # Uses 'exec' to run curl
          exec:
            # Command to run: check localhost root with curl, fail if error
            command: ["/bin/sh", "-c", "curl -f http://localhost/ || exit 1"]
          # Wait 5 seconds before start
          initialDelaySeconds: 5
          # Check every 30 seconds
          periodSeconds: 30
          # Timeout after 3 seconds
          timeoutSeconds: 3
        # Readiness probe
        readinessProbe:
          # Uses 'exec' to run curl
          exec:
            # Command to run: check localhost root
            command: ["/bin/sh", "-c", "curl -f http://localhost/ || exit 1"]
          # Wait 5 seconds before start
          initialDelaySeconds: 5
          # Check every 5 seconds
          periodSeconds: 5
          # Timeout after 3 seconds
          timeoutSeconds: 3
---
# Separator for the Service resource
apiVersion: v1
# Defines a Service
kind: Service
# Metadata for the Service
metadata:
  # Name of the service
  name: lv-ms2-frontend-service
# Service specification
spec:
  # Types: 'NodePort' exposes the service on an external port on every node
  type: NodePort
  # Selector to find frontend pods
```

```
  selector:
    app: lv-ms2-frontend
  # Ports configuration
  ports:
  # The internal port within the service
  - port: 80
    # The container port to forward to
    targetPort: 80
    # The specific external port opened on every node (30000-32767)
    nodePort: 30080
```

- Similar to API, but exposes port 80.
- `Service` **type** `NodePort`: Exposes the service on a specific port on the node (30080), though we will primarily use Ingress.

5. Ingress File: `k8s/frontend-ingress.yaml` This is the entry point for external traffic.

```
# Specifies the stable networking API version for Ingress resources
apiVersion: networking.k8s.io/v1
# Defines the resource as an Ingress (manages external access, usually HTTP/HTTPS)
kind: Ingress
# Metadata section
metadata:
  # The unique name for this Ingress resource
  name: ms2-frontend-ingress
  # Annotations to configure specific behavior of the Ingress Controller (e.g.,
NGINX)
  annotations:
    # Crucial: Rewrites the path before sending to the backend.
    # It takes the second capture group ($2) from the regex below and uses that as
the path.
    # Example: '/api/users' -> '/users' (strips the '/api' prefix)
    nginx.ingress.kubernetes.io/rewrite-target: /$2
# Specification of the routing rules
spec:
  # A list of host rules
  rules:
    # Protocol definition (HTTP)
    - http:
      # List of paths to route
      paths:
      # API routes must come first (more specific path)
      # This Regex matches "/api" followed by a slash or end-of-line ($1), then
the rest ($2)
      - path: /api(/|$)(.*)
        # Type 'ImplementationSpecific' is required when using Regex paths in
NGINX
        pathType: ImplementationSpecific
        # Defines where to send traffic matching this path
        backend:
          # The Service to forward to
```

```
            service:
              # The name of our API service
              name: lv-ms2-api-service
              # The port on that service
              port:
                # Numeric port 8000
                number: 8000
        # Frontend routes come last (catch-all)
        # Matches the root path
        - path: /
          # Matches based on URL prefix (standard matching)
          pathType: Prefix
          # Backend configuration for the frontend
          backend:
            # The Service to forward to
            service:
              # The name of our Frontend service
              name: lv-ms2-frontend-service
              # The port on that service
              port:
                # Numeric port 80
                number: 80
```

- `host: lvms2.com`: The domain name we will use.
- **Routing Rules**:
  - Traffic to `lvms2.com/api/...` goes to the **API Service**.
  - Traffic to `lvms2.com/` goes to the **Frontend Service**.
- **Rewrite Target**: Removes `/api` from the path before sending it to the backend (so the backend sees `/user` instead of `/api/user`).

4.5 Manual Deployment

To deploy everything manually:

```
kubectl apply -f k8s/storage.yaml
kubectl apply -f k8s/db-deployment.yaml
kubectl apply -f k8s/api-deployment.yaml
kubectl apply -f k8s/frontend-deployment.yaml
kubectl apply -f k8s/frontend-ingress.yaml
```

```
PS C:\Users\lenvo\Documents\linux-webservices\milestone_2> kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
lv-ms2-api-6645cd8dbb-bstgd         0/1     Running   0          43s
lv-ms2-api-6645cd8dbb-vbgxt         0/1     Running   0          43s
lv-ms2-db-66f95bdf6f-n74sm          0/1     Running   0          43s
lv-ms2-frontend-647b88cb8b-2j6xj    0/1     Running   0          43s
```

5. Step 3: GitOps Deployment with ArgoCD

GitOps is a practice where the Git repository is the "source of truth". Instead of running `kubectl apply` manually, we ask an agent (ArgoCD) to sync the cluster state with the Git repo.

5.1 Installing ArgoCD

```
helm repo add argo https://argoproj.github.io/argo-helm
helm install argocd argo/argo-cd --namespace argocd --create-namespace
```

- **Helm**: A package manager for Kubernetes. It simplifies installing complex apps like ArgoCD.

5.2 The ArgoCD Application

File: `argocd-app.yaml`

```
# Specifies the Custom Resource Definition (CRD) API provided by ArgoCD
apiVersion: argoproj.io/v1alpha1
# Defines the resource as an ArgoCD 'Application'
kind: Application
# Metadata for the Application resource itself
metadata:
  # The name of this application dashboard entry in ArgoCD
  name: milestone2-app
  # The namespace where ArgoCD itself is installed (usually 'argocd')
  namespace: argocd
# Specification of the application deployment
spec:
  # ArgoCD Project to group applications under (default is capable of everything)
  project: default
  # Defines where the configuration files are stored (The "Source of Truth")
  source:
    # The URL of your GitHub repository
    repoURL: https://github.com/LVogels/milestone_2-deploy-webapp.git
    # The branch or commit to track ('HEAD' tracks the latest commit on the
default branch)
    targetRevision: HEAD
    # The specific folder inside the repo containing the .yaml files (e.g.,
k8s/*.yaml)
    path: k8s
  # Defines where the application should be deployed (The "Destination")
  destination:
    # The internal address of the Kubernetes API server (deploys to the same
cluster ArgoCD is running in)
    server: https://kubernetes.default.svc
    # The target namespace for your actual app resources (Database, API, Frontend)
    namespace: ms2
  # Defines how ArgoCD handles updates
  syncPolicy:
    # Enables automatic synchronization
    automated:
      # If true: Deletes resources in K8s if you remove them from Git (keeps it
```

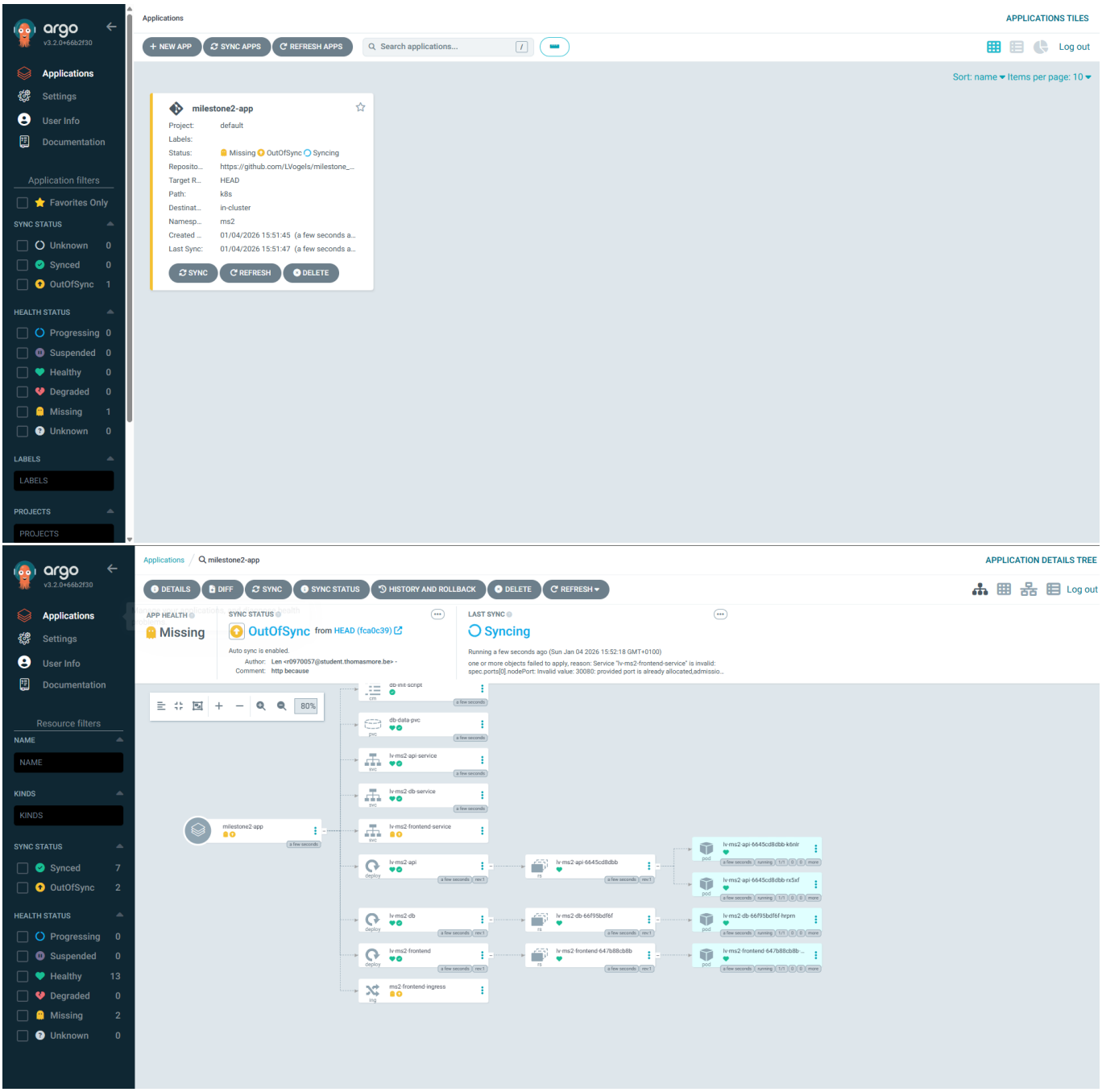
```
clean)
      prune: true
      # If true: If someone manually changes the cluster (e.g., `kubectl edit`),
ArgoCD instantly reverts it to match Git
      selfHeal: true
    # specific options for the sync process
    syncOptions:
    # Automatically creates the namespace 'ms2' if it doesn't exist yet
    - CreateNamespace=true
```

- **`source`**: Points to *this* GitHub repository and the `k8s` folder.
- **`destination`**: Tells ArgoCD to deploy into the `ms2` namespace.
- **`syncPolicy`**:
    - `automated`: ArgoCD automatically applies changes found in Git.
    - `selfHeal`: If you manually delete a pod, ArgoCD will recreate it to match Git.
    - `prune`: If you remove a file from Git, ArgoCD deletes the resource from the cluster.

5.3 Deploying via ArgoCD

```
kubectl apply -f argocd-app.yaml
```

Now, ArgoCD monitors the repository. If you push a change to the `k8s` folder, ArgoCD will automatically update the cluster.



6. Verification & Conclusion

6.1 Accessing the Application

Len Vogels has reached milestone 2!

Container ID: lv-ms2-api-6645cd8dbb-qrshb

Len Vogels has reached milestone 2!

Container ID: lv-ms2-api-6645cd8dbb-kd6rz

As seen above, the containers round-robin to balance the load between them

6.2

This project has successfully demonstrated the end-to-end process of building, containerizing, and deploying a modern three-tier web application. We have navigated the path from local development to a production-grade orchestration environment, highlighting several key technologies and concepts:

Technologies Used

- **Docker & Docker Compose**: Served as the foundation for our containerization strategy. Docker allowed us to package our FastAPI backend, Lighttpd frontend, and MariaDB database into isolated, reproducible units. Docker Compose bridged the gap for local testing, enabling us to define multi-container relationships and networking in a simple YAML format before moving to more complex orchestration.
- **Kubernetes (KIND)**: By utilizing Kubernetes in Docker (KIND), we simulated a real-world cluster environment on a local machine. This exposed us to core Kubernetes primitives such as **Pods** (atomic units of execution), **Deployments** (state management and scaling), **Services** (internal networking and load balancing), and **PersistentVolumeClaims** (stateful storage management).
- **Ingress (Nginx)**: We moved beyond simple port forwarding by implementing an Nginx Ingress Controller. This taught us about Layer 7 routing, allowing us to manage external access to our services via a single entry point (`lvms2.com`) and route traffic based on URL paths (rewriting logic for the API).
- **GitOps with ArgoCD**: The pinnacle of our automation strategy. We moved away from manual `kubectl apply` commands to a declarative GitOps workflow. ArgoCD continuously monitors our Git repository, ensuring that the live cluster state matches our desired state defined in code. This provides robust features like auto-syncing, self-healing configurations, and simplified rollback mechanisms.
- **Application Stack**:
  - **FastAPI**: Demonstrated a high-performance, lightweight Python framework for the backend.
  - **Lighttpd on Alpine**: Showcased extreme efficiency by using a minimal web server on a 5MB OS footprint.

- **MariaDB**: Provided experience with managing stateful applications in a stateless container environment using Volume mounts.

Main Learning Points

1. **The Shift to Declarative Infrastructure**: We learned the power of defining "what we want" (YAML manifests) rather than "how to get there" (scripted commands). This makes infrastructure auditable, version-controlled, and reproducible.
2. **Scalability and Resilience**: Through Kubernetes Deployments and Liveness/Readiness probes, we understood how modern systems maintain high availability. We saw how the system can automatically restart failed containers or stop sending traffic to unready pods without human intervention.
3. **Service Discovery and Networking**: The project clarified how different components in a distributed system locate and communicate with each other. From Docker networks to Kubernetes Services and Ingress rules, we mastered the flow of traffic within and into the cluster.
4. **The "Code-First" Deployment Lifecycle**: By integrating ArgoCD, we established a workflow where code commits trigger infrastructure updates. This reduces "configuration drift" and human error, mimicking the pipelines used by top-tier tech companies.

In summary, this exercise provided a holistic view of the Cloud Native landscape, equipping us with the skills to architect, deploy, and maintain robust web applications using industry-standard tools.