Docker and Kubernetes Workshop for Beginners: Hands-On with a Simple Web App

Workshop Prerequisites

- A computer with Python installed (version 3.8+).
- Install Docker Desktop (free from docker.com—download and run the installer).
- Install Minikube (a simple way to run Kubernetes locally; download from kubernetes.io/docs/tasks/tools/).
- A code editor like VS Code or Notepad.
- Basic terminal/command line access (we'll use simple commands).

Time Estimate: 90–120 minutes, with breaks for trying things yourself.

Key Concepts We'll Cover:

- **Docker:** Think of it as a way to package your app into a "box" (container) that runs the same everywhere, without worrying about different computers.
- **Kubernetes (K8s):** Like a manager for multiple Docker boxes—it handles starting, stopping, and scaling them.

Let's dive in!

1 How to Generate the Dummy Project Using LLM

In a real workshop, we'd code everything live together. But to make it fun and interactive, we'll use an AI tool (like Grok, ChatGPT, or any LLM) to generate the basic code for our dummy project. This saves time and shows how AI can help developers.

Why use an LLM? It helps create clean, working code quickly, so we can focus on Docker and Kubernetes. You'll input a prompt, and the LLM will output the files—copy them into your editor.

Exact Prompt to Use: Copy-paste this into an LLM (e.g., chat.x.ai or chatgpt.com):

> Create a simple beginner-level Python Flask web application that serves a 'Hello World' message at the root URL ('/') and a simple JSON API at '/api' that returns {'message': 'Welcome to the workshop!'. Include only the necessary files: app.py (the main app code) and requirements.txt (for dependencies). Add comments in the code to explain each part. Keep it under 50 lines total, and make sure it's runnable with 'python app.py'. Do not include any Docker or Kubernetes code—just the app itself.

What to Expect from the LLM:

- It will output two files: `app.py` and `requirements.txt`.
- Save them in a new folder on your computer, e.g., `C:\workshop\hello-app` (Windows) or `~/workshop/hello-app` (Mac/Linux).
- If the output isn't perfect, tweak the prompt (e.g., add "fix any errors").

Now, let's assume you've generated it—here's what the code should look like (for reference; generate it live in class!):

1.1 app.py

```python
# Import Flask: This is the main library for building the web app.
from flask import Flask, jsonify

# Create the app instance: This is like the "heart" of your web server.
app = Flask(__name__)

# Define the root route ('/'): This handles visits to the main page.
@app.route('/')
def hello_world():
   return 'Hello World from our workshop!' # Simple text response.

# Define the API route ('/api'): This returns JSON data.
@app.route('/api')
def api():
   return jsonify({'message': 'Welcome to the workshop!'}) # JSON response.

# Run the app: This starts the server when you run the file.
if __name__ == '__main__':
   app.run(debug=True, host='0.0.0.0', port=5000) # Debug mode for easy
       testing; listens on all interfaces.
```

1.2 requirements.txt

```
Flask==3.0.3 # The web framework we need.
```

Hands-On Tip: In class, let's all input the prompt now and compare outputs. If yours differs slightly, that's okay—as long as it runs!

2 Creating the Project Step by Step

Now that we have the code from the LLM, let's set it up and run it locally (without Docker yet). This helps you understand the app before containerizing it.

Why do this? It shows the app works on its own, so you can see how Docker "wraps" it without changing the code.

1. **Create a Project Folder:**
   - Open your terminal (Command Prompt on Windows, Terminal on Mac/Linux).
   - Run: `mkdir hello-app` (creates a folder).
   - `cd hello-app` (go into it).

2. **Save the Generated Files:**
   - Copy `app.py` and `requirements.txt` into this folder using your editor.
3. **Install Dependencies:**
   - Run: `pip install -r requirements.txt`
   - Why? This installs Flask so your app can run.
4. **Run the App Locally:**
   - Run: `python app.py`
   - Why? This starts the web server.
   - Open a browser and go to `http://localhost:5000`—you should see “Hello World from our workshop!”
   - Try `http://localhost:5000/api`—it shows JSON: `{"message": "Welcome to the workshop!"}`
   - Stop it with Ctrl+C in the terminal.

Quick Concept Explanation:

- Your app is like a tiny web server. Without Docker, it only runs on your machine if Python and Flask are installed. Docker will make it portable.

Hands-On Pause: Everyone try running it now. If it doesn’t work, check for typos in the code!

3 Introduction to Docker – Containerizing Your App

Docker lets you package your app, code, and dependencies into an “image” (like a blueprint), then run it as a “container” (the live version).

Simple Diagram (ASCII Art):

```
+----------------+     Build      +----------------+      Run      +----------------+
| Your Code      |  ---------->  | Docker Image   |  ----------> | Container      |
| (app.py, etc.)|               | (Blueprint)    |               | (Running App)  |
+----------------+               +----------------+               +----------------+
```

- **Why Docker?** It solves “it works on my machine” problems—runs the same on any computer.

3.1 Write the Dockerfile

Create a file called `Dockerfile` (no extension) in your `hello-app` folder.

Dockerfile Content:

```dockerfile
# Start with a base image: This is like a mini-OS with Python pre-installed.
FROM python:3.9-slim

# Set the working directory: Where your app code will live inside the container.
WORKDIR /app
```

```

# Copy requirements first: This optimizes builds by caching dependencies.
COPY requirements.txt .

# Install dependencies: Runs 'pip install' inside the container.
RUN pip install -r requirements.txt

# Copy the app code: Adds your app.py to the container.
COPY app.py .

# Expose the port: Tells Docker the app listens on port 5000.
EXPOSE 5000

# Command to run the app: Starts the server when the container runs.
CMD ["python", "app.py"]
```

Why each line?

- FROM: Bases it on Python to avoid installing it manually.
- WORKDIR: Keeps things organized.
- COPY and RUN: Ensures dependencies are installed.
- EXPOSE: Not mandatory, but good practice for documentation.
- CMD: The "start" command.

3.2 Build the Docker Image

1. In your terminal (in `hello-app` folder):
   - Run: `docker build -t hello-app:latest .`
   - Why? Builds the image named "hello-app" (tag: latest). The "." means use the current folder.
   - Output: You'll see layers being built—takes 1–2 minutes first time.
2. Check it: `docker images`—see "hello-app" listed.

3.3 Run the Container Locally

1. Run: `docker run -p 5000:5000 hello-app:latest`
   - Why? Starts the container and maps port 5000 (outside) to 5000 (inside).
   - Visit `http://localhost:5000` in your browser—same as before!
2. Stop it: Ctrl+C, or in another terminal: `docker ps` (list running), then `docker stop <container-id>`.

Hands-On Pause: Build and run your container now. Fun fact: Try running it on a friend's machine—they just need Docker!

4 Introduction to Kubernetes – Orchestrating Your Containers

Kubernetes (K8s) manages multiple containers—like a boss telling them when to start, scale, or restart.

Simple Diagram (ASCII Art):

```
+---------------+    Deploys    +---------------+     Exposes    +---------------
| Docker Image  |  ---------->  | Pod (Container|  ---------->  | Service
|               |               | in K8s)       |               | (Access Point)
+---------------+               +---------------+               +---------------
```

- **Pod:** Smallest unit—a running container.
- **Deployment:** Manages pods (e.g., restarts if one crashes).
- **Service:** Gives a stable way to access pods (like a front door).
- **Why K8s?** For apps that need to scale or be reliable beyond one container.

Setup Minikube:

1. Start Minikube: `minikube start` (creates a local K8s cluster—takes 2–5 minutes).
2. Check: `kubectl get nodes` (should show one node ready).

4.1 Push Image to a Registry (Optional but Recommended)

For K8s to use your image, push it to Docker Hub (free account needed).

1. Login: `docker login` (use your credentials).
2. Tag: `docker tag hello-app:latest yourusername/hello-app:latest`
3. Push: `docker push yourusername/hello-app:latest`

- Why? Minikube can use local images, but this teaches real-world sharing. If skipping, use `minikube image load hello-app:latest`.

4.2 Write Kubernetes Manifests

Create two YAML files in your folder.

4.2.1 deployment.yaml

```
apiVersion: apps/v1 # Version of the K8s API.
kind: Deployment # Type: A Deployment manages pods.
metadata:
  name: hello-deployment # Name for this deployment.
spec:
  replicas: 1 # Number of pods to run (start with 1).
  selector:
    matchLabels:
      app: hello-app # Labels to identify pods.
  template: # Pod template.
    metadata:
      labels:
        app: hello-app # Label for the pod.
    spec:
      containers: # List of containers in the pod.
      - name: hello-container # Container name.
        image: yourusername/hello-app:latest # Your Docker image (or local:
            hello-app:latest).
        ports:
        - containerPort: 5000 # Port the app uses inside.
```

Why? This tells K8s to run 1 copy of your container.

4.2.2 service.yaml

```
apiVersion: v1 # API version.
kind: Service # Type: A Service exposes the app.
metadata:
  name: hello-service # Name for the service.
spec:
  selector:
    app: hello-app # Matches the deployment's labels.
  ports:
  - protocol: TCP # Protocol used.
    port: 80 # External port (browser uses this).
    targetPort: 5000 # Internal pod port.
  type: LoadBalancer # Type: Makes it accessible outside Minikube.
```

Why? Without this, you can't access the pod from your browser.

4.3 Apply Manifests and Access the App

1. Apply Deployment: `kubectl apply -f deployment.yaml`
   - Why? Creates the pod.
2. Check: `kubectl get pods`—see it running.
3. Apply Service: `kubectl apply -f service.yaml`
4. Get Access URL: `minikube service hello-service -url`
   - This gives a URL like `http://192.168.49.2:XXXX`—open in browser.
   - Visit `/` and `/api`—your app is now in K8s!

Hands-On Pause: Apply and access it. Scale up: Edit replicas to 2 in `deployment.yaml`, re-apply, and check `kubectl get pods`.

5 Troubleshooting Tips and FAQs

Common Issues:

- **Docker Build Fails:** Check for typos in Dockerfile. Run `docker build ...` with `-no-cache` to rebuild fresh.
- **Container Won't Start:** Use `docker logs <container-id>` to see errors (e.g., missing dependencies).
- **K8s Pod Crashing:** `kubectl logs <pod-name>`—fix code or image.

- **Minikube Not Starting:** Ensure Virtualization is enabled in BIOS; restart Docker.

FAQs:

- Q: What's the difference between Docker and K8s? A: Docker packages; K8s manages many packages.
- Q: Why YAML? A: It's human-readable for configs.
- Q: Is this production-ready? A: No—this is local; real K8s uses clouds like AWS.

6 Exercises and Challenges

Great job! Now practice:

1. **Basic:** Change the "Hello World" message in `app.py`, rebuild Docker image, update K8s deployment, and see the change.
2. **Intermediate:** Add a new route in `app.py` (e.g., `/health` returning `{'status': 'ok'}`), rebuild, and test in K8s.
3. **Challenge:** Scale to 3 replicas in `deployment.yaml`. Use `kubectl get pods` to verify, then stress-test by refreshing the browser rapidly—K8s load-balances!
4. **Advanced:** Add environment variables to Dockerfile (e.g., `ENV MESSAGE="Custom Hello"`), use in `app.py`, and pass via K8s deployment.

Cleanup: `kubectl delete -f service.yaml -f deployment.yaml`; `minikube stop`; `docker rmi hello-app:latest`.

Thanks for joining! You've just containerized and orchestrated your first app. Practice more, and soon you'll be deploying real projects. Questions? Let's discuss!