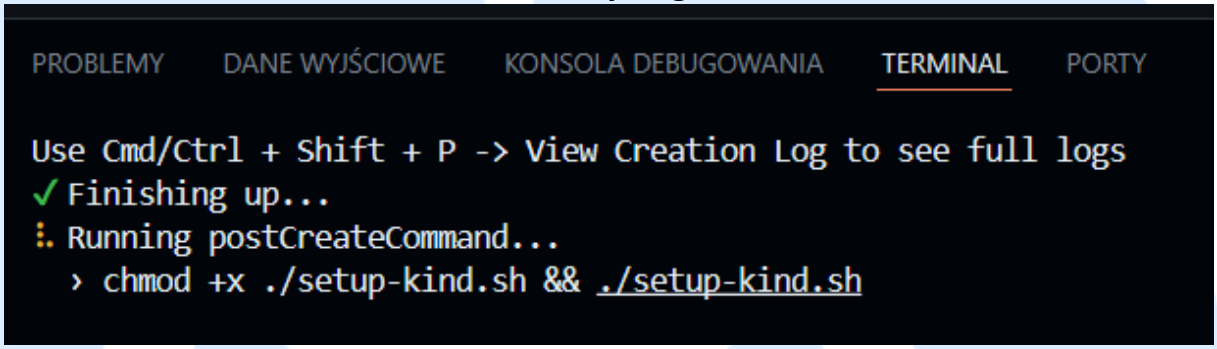
Introduction to Kubernetes

Kubernetes is an open-source platform for automating the deployment, scaling, and management of containerized applications. It allows you to run multiple cooperating containers in organized units called pods.
Instead of manually configuring the infrastructure, the user defines the desired state of the system using YAML manifests. Kubernetes automatically maintains this state by launching, restarting, or relocating containers as needed.
Key objects include: Deployment (manages replication and updates), Service (exposes applications), ConfigMap and Secret (store configuration), and Volume (stores data).
The system operates in a declarative model and provides high fault tolerance.
Thanks to autoscaling and rollback mechanisms, Kubernetes supports continuous operation and easy deployment of new application versions. It supports many environments – from local clusters (e.g., kind, minikube) to the cloud. The architecture is based on the master-worker model, where the control plane manages the cluster, and worker nodes perform tasks. Kubernetes is the foundation of modern, scalable microservices-based systems.
In this tutorial, you will practice its most important mechanisms through a practical example.

The circles indicate points you gain for completing tasks and showing them to us, they are 5, 10 and 15 points, appropriately

1. Fork and Environment Setup

1. **Fork the repository**
   – Go to GitHub: https://github.com/Ssyhaj/k8s-tutorial/tree/main
   – Click **Fork** (at the top)
2. **Create a Codespace**
   – In your fork, choose **Code → Codespaces → New codespace**.
   – Wait until VS Code launches the environment; the image build process and postCreateCommand will start a script initializing the cluster.
3. **Wait until the notification finishes and you gain terminal access**



4. **Switch the namespace**
   kubectl config set-context --current --namespace=tutorial

2. Repository Structure

k8s-tutorial/

├─ .devcontainer/

│ ├─ devcontainer.json ← image + docker-in-docker + postCreate

│ └─ Dockerfile ← installs kubectl & kind

├─ setup-kind.sh ← creates a kind cluster and namespace

├─ manifests/

│ ├─ mysql-deploy.yaml

│ ├─ mysql-svc.yaml

│ ├─ wordpress-configmap.yaml

│ ├─ wordpress-deploy.yaml

│ └─ wordpress-svc.yaml

└─ README.md ← “how to start” instructions

3. Deploying WordPress + MySQL

1. Start the cluster

bash setup-kind.sh

kubectl config set-context --current --namespace=tutorial

2. Apply MySQL manifests

Find out how to apply the files listed in the github repository. Run commands to apply all the MySQL files.

3. Apply WordPress manifests

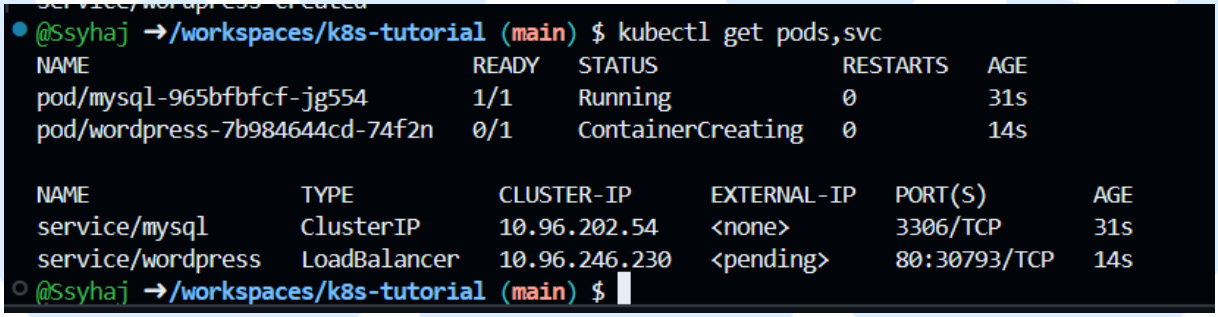
Now do the same for the WordPress files.

4. Verification ( )

kubectl get pods,svc

kubectl describe deployment wordpress

kubectl logs -l app=wordpress



```
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=wordpress
  Containers:
   wordpress:
    Image:      wordpress:6.2
    Port:       80/TCP
    Host Port:  0/TCP
    Environment Variables from:
      wp-config    ConfigMap  Optional: false
    Environment:   <none>
    Mounts:        <none>
  Volumes:         <none>
  Node-Selectors:  <none>
  Tolerations:     <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Available      True    MinimumReplicasAvailable
  Progressing    True    NewReplicaSetAvailable
OldReplicaSets:  <none>
NewReplicaSet:   wordpress-7b984644cd (1/1 replicas created)
Events:
  Type    Reason             Age   From                   Message
  ----    ------             ----  ----                   -------
  Normal  ScalingReplicaSet  75s   deployment-controller  Scaled up replica set wordpress-7b984644cd to 1
```

```
@Ssyhaj →/workspaces/k8s-tutorial (main) $ kubectl logs -l app=wordpress
WordPress not found in /var/www/html - copying now...
Complete! WordPress has been successfully copied to /var/www/html
No 'wp-config.php' found in /var/www/html, but 'WORDPRESS_...' variables supplied; copying 'wp-config-docker.php' (WORDPRESS_DB_HOST WORDPRESS_DB_NAME WORDPRESS
_DB_PASSWORD WORDPRESS_DB_USER WORDPRESS_PORT WORDPRESS_PORT_80_TCP WORDPRESS_PORT_80_TCP_ADDR WORDPRESS_PORT_80_TCP_PORT WORDPRESS_PORT_80_TCP_PROTO WORDPRESS_
SERVICE_HOST WORDPRESS_SERVICE_PORT)
AH00558: apache2: Could not reliably determine the server's fully qualified domain name, using 10.244.0.6. Set the 'ServerName' directive globally to suppress t
his message
AH00558: apache2: Could not reliably determine the server's fully qualified domain name, using 10.244.0.6. Set the 'ServerName' directive globally to suppress t
his message
[Mon May 19 06:16:00.504038 2025] [mpm_prefork:notice] [pid 1] AH00163: Apache/2.4.56 (Debian) PHP/8.0.30 configured -- resuming normal operations
[Mon May 19 06:16:00.504580 2025] [core:notice] [pid 1] AH00094: Command line: 'apache2 -D FOREGROUND'
@Ssyhaj →/workspaces/k8s-tutorial (main) $
```

4. Exploration

– Open the service URL in your browser (VS Code will show the "LoadBalancer Ingress").
– Log in to WordPress, e.g., create a "Hello K8s" page.

Step 1 – Make sure the service exists and is running:
kubectl get svc

You should see something like:

NAME	*TYPE*	*CLUSTER-IP*	*EXTERNAL-IP*	*PORT(S)*	*AGE*
wordpress	*ClusterIP*	*10.96.200.10*	*<none>*	*80/TCP3m*	
mysql	*ClusterIP*	*10.96.100.25*	*<none>*	*3306/TCP*	*3m*

The wordpress service is an internal ClusterIP service. You need to access it using port forwarding.

Step 2 – Start port-forwarding
kubectl port-forward svc/wordpress 8080:80

This command:

- Forwards port 80 from the WordPress service to localhost:8080 in your Codespace.
- The terminal will remain "occupied" until interrupted (e.g., Ctrl+C).
  IMPORTANT: Do not close the terminal with this command – it must remain active while using the application.

Step 3 – Open browser in Codespaces (If the browser does not open because of a connection timeout, you can skip to step 5)

1. At the top of Codespaces, click **PORTS**.
2. Find port 8080, click **Open in Browser**.
   Alternatively, click the link icon next to port 8080.
   The browser will open the WordPress installation page.

Step 4 – Install and log into WordPress

1. Choose your language.
2. Fill in the form:
   - Site title: **Hello K8s**
   - Username: **admin**
   - Password: **admin123**
   - Email: anything, e.g., test@example.com
3. Click **Log in** and create a new page/article.

Step 5 – Confirm it works with K8s

From the terminal, you can confirm that traffic is reaching the WordPress pod:

```
kubectl get pods -l app=wordpress -o wide
```

You will see the pod name and internal IP. This means that WordPress is running in a pod, the traffic goes through port-forwarding to the service and from there to the pod.

5. ConfigMap instead of env

1. Delete the env: section from manifests/wordpress-deploy.yaml (if still present).

wordpress-deploy.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: wordpress
spec:
  replicas: 1
  selector:
    matchLabels:
      app: wordpress
  template:
    metadata:
      labels:
        app: wordpress
    spec:
      containers:
      - name: wordpress
        image: wordpress:6.2
        ports:
        - containerPort: 80
        envFrom:
        - configMapRef:
            name: wp-config
```

2. Reapply using the same method as in **3.2/3.3**:

Check that WordPress still works – variables are now loaded from the ConfigMap.

6. Volume Experiment ( )

You need to modify the manifests/wordpress-deploy.yaml file to add a temporary volume (emptyDir) to the WordPress container. This way, any data the application writes during the pod's lifetime will be assigned to that volume — but will disappear after the pod restarts (demonstrating temporary resources without PVC).

1. Edit manifests/wordpress-deploy.yaml, under containers add:

```
        volumeMounts:

        - name: wp-content

        mountPath: /var/www/html

volumes:

- name: wp-content

  emptyDir: {}
```

2. Apply the changes

3. Connect to the pod and create a file:

```
kubectl exec -it $(kubectl get pod -l app=wordpress -o name) -- bash

cd /var/www/html && touch test.txt
```

4. Exit, verify if pod exists then delete it and verify again

What does this experiment show?

- emptyDir: {} – is a temporary volume tied to the **pod's** lifetime, not the container's.
- mountPath: /var/www/html – is the directory where WordPress stores code and data (including user files).
- After deleting the pod (kubectl delete pod) — a new one appears, but the data saved in the old one (e.g., test.txt) is gone.

7. Rolling Update + Rollback ( )

1. Trigger a WordPress update with a faulty image. You can research the command and make up your own faulty WordPress image

2. Monitor status:

kubectl rollout status deployment wordpress

kubectl get pods

kubectl describe pod

(The new pod will not start)

3. Roll back the change:

kubectl rollout undo deployment wordpress

kubectl get pods

Congratulations — you've completed a full application lifecycle on Kubernetes, from cluster startup to hands-on experiments with configuration, volumes, and updates. This showed how declarative resource management works, how Kubernetes responds to errors, and how it maintains application state. You've understood the difference between temporary and persistent data, and how to control app behavior using ConfigMaps.
Experiments using kubectl exec, logs, delete pod, and scale gave you insight into what's happening "under the hood." Rolling updates and rollbacks demonstrated Kubernetes' support for safe change deployment. These were just the basics — but highly practical.
In the next steps, you can explore topics like Ingress, StatefulSet, Helm, or monitoring. The Kubernetes documentation is an excellent resource for further learning.
For now – you have a working cluster and application you can build on.