Lingoda Case Study

Task 1: Deploy Symfony Demo App in Kubernetes:

Objective: Deploy the Symfony demo application in a Kubernetes environment.

Instructions

● Obtain the Symfony demo application (GitHub - Symfony Demo).
● Create Dockerfiles to containerize the application.
● Write Kubernetes manifest files for deploying the containerized application.
● Use production-applicable best practices where possible (and/or leave a comment on what you would do differently, given more time and a real production setting if your implementation differs).
● Ensure the application is accessible and functional.
● The deliverable should be reproducible on an empty namespace.

☑ Obtain the Symfony demo application (GitHub - Symfony Demo).

Symfony demo app repo: ***https://github.com/symfony/demo***

☑ Create Dockerfiles to containerize the application.

```
# Use the official PHP image as the base image
FROM php:cli

# Set the working directory in the container
WORKDIR /var/www/html

# Install PHP extensions and other dependencies
RUN apt-get update && apt-get install -y \
   libzip-dev \
   git \
   curl \
   && docker-php-ext-install zip pdo pdo_mysql

# Install Composer
RUN curl -sS https://getcomposer.org/installer | php --
--install-dir=/usr/local/bin --filename=composer
```

```
# Install Symfony CLI
RUN curl -sS https://get.symfony.com/cli/installer | bash && \
   mv /root/.symfony5/bin/symfony /usr/local/bin/symfony

# Copy Symfony demo app

COPY . .

# Install Symfony dependencies

RUN composer install  --no-dev --optimize-autoloader --no-scripts --no-plugins

# Expose port 8000 to the outside world
EXPOSE 8000

# Command to run the Symfony application
CMD ["symfony", "server:start", "--no-tls", "--port=8000", "--allow-http"]
```

☑ Write Kubernetes manifest files for deploying the containerized application.

Write Kubernetes manifest files for deploying the containerized application. We need to define Deployment, Service, secrets, persistent volume & persistent volume claim.

Deployment:

```
# Symfony App Deployment

apiVersion: apps/v1
kind: Deployment
metadata:
 name: symfony-demo
 namespace: case-study
spec:
 replicas: 3
```

```
 selector:
   matchLabels:
     app: symfony-demo
 template:
   metadata:
     labels:
       app: symfony-demo
   spec:
     containers:
     - name: symfony-demo
       image: nidhishd/symfony-demo3:latest
       ports:
       - containerPort: 8000
       env:
         - name: DATABASE_URL
           valueFrom:
             secretKeyRef:
               name: mysql-secret
               key: DATABASE_URL

---

# Database Deployment (MySQL Example)

apiVersion: apps/v1
kind: Deployment
metadata:
 name: mysql-db
 namespace: case-study
spec:
 replicas: 1
 selector:
   matchLabels:
     app: mysql-db
 template:
   metadata:
     labels:
       app: mysql-db
   spec:
     containers:
     - name: mysql
       image: mysql:latest
       ports:
       - containerPort: 3306
       env:
```

```
      - name: MYSQL_ROOT_PASSWORD
        valueFrom:
           secretKeyRef:
            name: mysql-credentials
             key: mysql-root-password
      - name: MYSQL_DATABASE
        valueFrom:
           secretKeyRef:
            name: mysql-credentials
             key: db-name
      - name: MYSQL_USER
         valueFrom:
          secretKeyRef:
            name: mysql-credentials
             key: db-user
       - name: MYSQL_PASSWORD
        valueFrom:
          secretKeyRef:
             name: mysql-credentials
            key: db-password

      volumeMounts:
      - name: mysql-data
        mountPath: /var/lib/mysql
    volumes:
    - name: mysql-data
      emptyDir: {}
```

Service:

```
apiVersion: v1
kind: Service
metadata:
 name: symfony-demo-service
 namespace: case-study
spec:
 selector:
   app: symfony-demo
 ports:
   - protocol: TCP
     port: 80
     targetPort: 8000
```

```
---

apiVersion: v1
kind: Service
metadata:
 name: mysql-service
 namespace: case-study
spec:
 selector:
   app: mysql-db
 ports:
   - protocol: TCP
     port: 3306
```

Secrets:

```
apiVersion: v1
kind: Secret
metadata:
 name: mysql-secret
 namespace: case-study
type: Opaque
data:
 database_url: bXlzcWw6Ly9sb2NhbGhvc3Q6MzMwNi9kYXRhYmFzZQ==

---

apiVersion: v1
kind: Secret
metadata:
 name: mysql-credentials
 namespace: case-study
type: Opaque
data:
 mysql-root-password: YWRtaW5fMTIz
 db-name: ZGF0YWJhc2U=
 db-user: YWRtaW4=
 db-password: cGFzc3dvcmQxMjM=
```

pv:

```
apiVersion: v1
```

```yaml
kind: PersistentVolume
metadata:
 name: mysql-data-pv
 namespace: case-study
spec:
 capacity:
   storage: 1Gi
 accessModes:
   - ReadWriteOnce
 persistentVolumeReclaimPolicy: Retain
 hostPath:
   path: /var/lib/mysql

---



apiVersion: v1
kind: PersistentVolume
metadata:
 name: migrations
 namespace: case-study
spec:
 capacity:
   storage: 1Gi
 accessModes:
   - ReadWriteOnce
 persistentVolumeReclaimPolicy: Retain
 hostPath:
   path: /migrations
```

pvc:

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
 name: mysql-data-pvc
 namespace: case-study
spec:
 accessModes:
   - ReadWriteOnce
 resources:
   requests:
     storage: 1Gi
```

```
---



apiVersion: v1
kind: PersistentVolumeClaim
metadata:
 name: migrations
 namespace: case-study
spec:
 accessModes:
   - ReadWriteOnce
 resources:
   requests:
     storage: 1Gi
```

- ☑ Use production-applicable best practices where possible (and/or leave a comment on what you would do differently, given more time and a real production setting if your implementation differs).

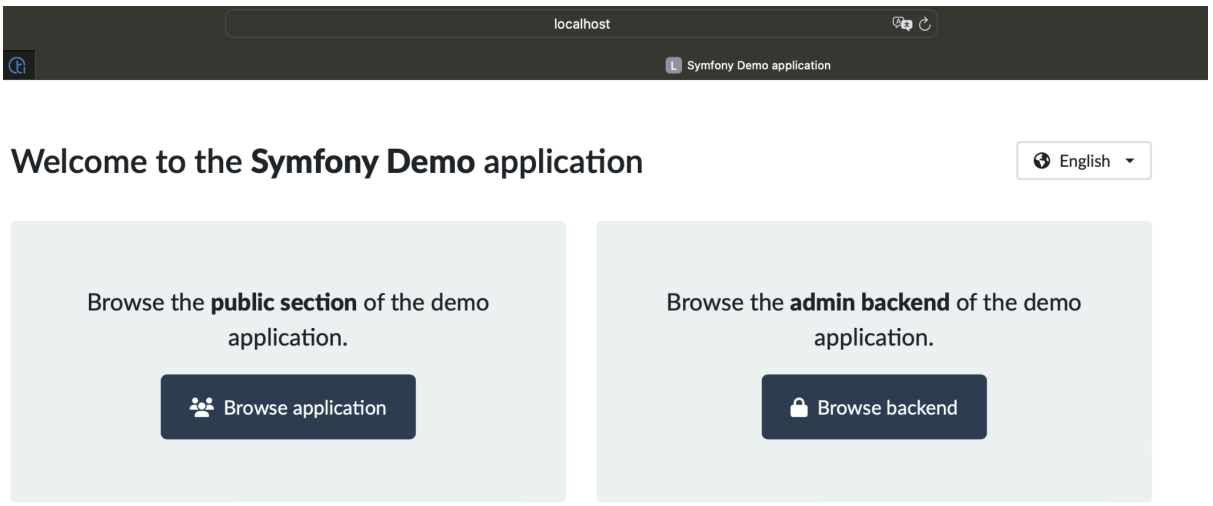
  - Implement health checks for pods to ensure they are responsive.
  - Set up logging and monitoring for observability.
  - Configure resource limits and requests for pods.
  - Consider implementing Ingress for more advanced routing and TLS termination.

- ☑ Ensure the application is accessible and functional.

I forwarded traffic from port 8000 on your local machine to port 80 on the symfony-demo-service service in your Kubernetes cluster.

kubectl port-forward service/symfony-demo-service 8000:80

The application is functional: http://localhost:8000/



☑ The deliverable should be reproducible on an empty namespace.

The kubernetes manifests are deployed in a newly created ‘case-study’ namespace.

Task 2: Implement Database Migrations:

Objective: Add the capability to run database migrations within the Kubernetes deployment from Task 1.

☑ Outline a process or create a script to handle database migrations during deployment of a new version of the demo app.

☑ Ensure that the database migrations are smoothly integrated into the deployment process without downtime or data loss.

Here's a basic outline for a script to handle database migrations during deployment:

```
#!/bin/bash

# Script to handle database migrations during Symfony deployment in Kubernetes
```

```
# Exit immediately if a command exits with a non-zero status
set -e

# Variables
APP_CONTAINER_NAME="symfony-demo"
NAMESPACE="default"

# Run migrations
echo "Running database migrations..."
kubectl exec -it $APP_CONTAINER_NAME -n $NAMESPACE -- php bin/console
doctrine:migrations:migrate --no-interaction

# Verify migrations
echo "Verifying migrations..."
kubectl exec -it $APP_CONTAINER_NAME -n $NAMESPACE -- php bin/console
doctrine:migrations:status --show-versions

# Exit successfully
exit 0
```

To smoothly integrate our database migrations into the Kubernetes deployment process for our Symfony application, we can follow the below process:

Implement Migration Script: Created a script that runs the database migrations. This script is executable within a container and can be triggered during the deployment process.

Updated Kubernetes Deployment Configuration: Modified our Kubernetes deployment configuration to include an init container responsible for running the migration scripts. This container will execute the migration process before starting the main application container.

Symfony app deployment.yaml with init-container:

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: symfony-demo
 namespace: case-study
spec:
 replicas: 1
 selector:
   matchLabels:
     app: symfony-demo
```

```yaml
template:
  metadata:
    labels:
      app: symfony-demo
  spec:
    containers:
    - name: symfony-demo
      image: nidhishd/symfony-demo3:latest
      ports:
      - containerPort: 8000
      env:
        - name: DATABASE_URL
          valueFrom:
            secretKeyRef:
              name: mysql-secret
              key: DATABASE_URL
    initContainers:
    - name: migration
      image: php:latest
      command: ["sh", "-c", "migration_script.sh"]
      volumeMounts:
      - name: migrations
        mountPath: /migrations
    volumes:
    - name: migrations
      persistentVolumeClaim:
        claimName: migrations
```

The script will run database migrations smoothly during deployment of a new version of your Symfony application in Kubernetes, ensuring that the database schema is up to date without downtime or data loss.

Doctrine.yaml:

```yaml
doctrine:
    dbal:
        # Choose your database driver
        # Options include: pdo_mysql, pdo_pgsql, pdo_sqlite, ...
        driver: pdo_mysql

        # Database connection parameters
        host: "%env(resolve:DATABASE_HOST)%"
        port: "%env(resolve:DATABASE_PORT)%"
```

```
        dbname: "%env(resolve:DATABASE_NAME)%"
        user: "%env(resolve:DATABASE_USER)%"
        password: "%env(resolve:DATABASE_PASSWORD)%"

        # Set other options as needed
        charset: utf8mb4
        # other options...

    orm:
        # Configure the ORM
        auto_generate_proxy_classes: true
        naming_strategy: doctrine.orm.naming_strategy.underscore
        auto_mapping: true
```

Kubernetes will execute the migration command in the Init Container before starting the Symfony application container. The readiness probe will ensure that the application is ready to serve traffic only after the migrations are successfully applied, ensuring zero downtime deployment.

Task 3: Scaling Concerns and Implementations

Objective:

Propose and implement relevant scaling strategies for the application deployed in Kubernetes.

☑ Identify potential scaling issues with the current deployment (consider both application and database layers).

- Application Layer:

  - High CPU or memory usage during peak times.
  - Increased response times or decreased throughput under load.
  - Instances becoming unresponsive or crashing due to resource exhaustion.

- Database Layer:

  - High CPU or memory usage on database nodes.

- Increased query latency or timeouts during high traffic.
- Database becoming a bottleneck for application performance.

☑ Implement either Vertical Pod Autoscaler (VPA) or Horizontal Pod Autoscaler (HPA) based on the identified needs.

For a Symfony application deployed in Kubernetes, the decision between Horizontal Pod Autoscaler (HPA) and Vertical Pod Autoscaler (VPA) depends on several factors specific to the application's architecture, resource usage patterns, and scalability requirements. Symfony applications are typically stateless, handling requests independently without relying on shared state across instances. Also it experiences fluctuating demand. So, we can go with Horizontal Pod Autoscaler as a best choice.
Implementation:

Sets up the HPA for the Deployment. The HPA will adjust the number of replicas based on CPU utilisation, targeting an average utilisation of 70%. It will scale between a minimum of 2 replicas and a maximum of 5 replicas.

☑ Determine appropriate metrics (CPU, memory usage, request rate, etc.) to base the scaling on.
☑ Document your choices and the rationale behind them.

CPU Usage: Symfony applications often consume CPU resources during request processing, especially if they are handling complex computations or heavy database operations. Monitoring CPU usage can help determine when to scale out to accommodate increased load and prevent performance degradation.

Memory Usage: Symfony applications may experience memory spikes due to caching, database queries, or other memory-intensive operations. Monitoring memory usage can ensure that pods have enough memory available to handle incoming requests without running out of resources.

Request Rate: Monitoring the rate of incoming HTTP requests can be valuable for scaling based on actual user demand. Symfony applications typically handle HTTP requests, so scaling based on request rate can ensure that there are enough pods to handle incoming traffic effectively.

Response Time: Monitoring response time can provide insights into application performance and user experience. Scaling based on response

time thresholds can help maintain acceptable performance levels during periods of increased load.

Since we already have deployment & service manifest for our symfony app, let's create an hpa.yaml for autoscaling our application.

hpa.yaml

```yaml
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
 name: myapp-hpa
spec:
 scaleTargetRef:
   apiVersion: apps/v1
   kind: Deployment
   name: symfony-demo
 minReplicas: 2
 maxReplicas: 5
 metrics:
 - type: Resource
   resource:
     name: cpu
     targetAverageUtilization: 70
 - type: Resource
   resource:
     name: memory
     targetAverageUtilization: 70
 - type: Pods
   pods:
     metricName: http_response_time
     targetAverageValue: 0.5s
 - type: Pods
   pods:
     metricName: http_requests_per_second
     targetAverageValue: 100
```

My repo: **https://github.com/Nidhishd/Case_Study-Lingoda** (public)

FYI:

K8S ,anifests are under the **manifest** folder.
DB migration script - **migrations/migration_script.sh**
Doctrine - **config/packages/doctrine.yaml**