Multi-Container Application Deployment on OpenShift

ChahineBA

January 29, 2025

TUNIS BUSINESS SCHOOL
UNIVERSITY OF TUNIS

Contents

1. Introduction

1.1. Project Overview

In today's rapidly evolving technological landscape, the need for scalable and efficient deployment of applications has become paramount. This project focuses on the development and deployment of a multi-container application using OpenShift, a container application platform that brings Docker and Kubernetes to the enterprise. By leveraging a microservices architecture and containerization, the project aims to deliver a robust application with seamless integration between its components.

1.2. Objectives and Significance

The primary objectives of this project are:

1. **Containerization with OpenShift**: Containerize individual components of the application using OpenShift's containerization features and utilize OpenShift-specific tools for building and managing container images.

2. **Microservices Architecture**: Design and implement the application as a set of loosely coupled microservices that align with OpenShift best practices and leverage OpenShift's support for deploying and scaling microservices.

3. **Communication Between Containers**: Implement communication mechanisms between containers, such as OpenShift Routes or services, and clearly document the communication protocols used.

4. **OpenShift Deployment Configuration**: Use OpenShift DeploymentConfigs to define and manage the application's deployment, and configure services, routes, and persistent storage within the OpenShift environment.

5. **Data Persistence**: Implement data persistence using OpenShift-compatible databases or storage solutions, ensuring that data can be stored and retrieved across container restarts within the OpenShift cluster.

6. **Scalability and Load Balancing**: Explore OpenShift's features for scaling the application horizontally and implement load balancing using OpenShift Routes or other applicable methods.

The significance of this project lies in demonstrating practical implementation of containerization and orchestration using OpenShift, which is vital for modern cloud-native applications.

1.3. Motivation

The motivation behind this project stems from the growing demand for scalable and maintainable applications. By developing a multi-container application, we can:

- Promote modularity and ease of maintenance.
- Enable independent development and deployment cycles.
- Leverage the benefits of containerization for consistent environments.
- Utilize OpenShift's orchestration capabilities for efficient resource management.

2. Application Architecture

2.1. Microservices Design

The application is designed using a microservices architecture, comprising three primary services:

1. **Login-Service**: Manages user authentication and provides a secure entry point to the application.
2. **Backend-Service**: Handles business logic and database interactions with MongoDB for data persistence.
3. **Chatbot-Service**: Offers advanced conversational AI capabilities using LangChain, Gemini, and Streamlit.

Each service is developed independently and containerized separately, promoting loose coupling and independent scalability.

2.2. Containerization with OpenShift

All services are containerized using OpenShift's containerization features. OpenShift-specific tools, such as `Source-to-Image (S2I)` and `BuildConfigs`, are utilized for building and managing container images.

3. Implementation Details

3.1. Login-Service

3.1.1. Containerization

The Login-Service is containerized using a `Dockerfile` compatible with OpenShift's build process. The service runs a Flask application for the authentication interface.

3.1.2. Deployment Configuration

An OpenShift `DeploymentConfig` is created for the Login-Service, defining replicas, strategy, and triggers for deployment.

3.2. Backend-Service

3.2.1. Data Persistence with MongoDB

The Backend-Service connects to a MongoDB instance for data persistence. MongoDB is deployed within the OpenShift cluster using a `StatefulSet` and `PersistentVolumeClaims` to ensure data durability across restarts.

3.2.2. Service Configuration

An OpenShift service is configured to expose the Backend-Service internally within the cluster, allowing communication from the Login-Service and Chatbot-Service.

3.3. Chatbot-Service

3.3.1. Advanced Conversational AI

The Chatbot-Service utilizes LangChain and Gemini models to provide advanced AI capabilities. It's built with Streamlit for the user interface.

3.3.2. OpenShift Deployment

Similar to other services, the Chatbot-Service is containerized and deployed using OpenShift `DeploymentConfig`. It is exposed using an OpenShift `Route` to allow external access.

4. Communication Between Containers

4.1. Inter-Service Communication

Communication between the services is facilitated through OpenShift services and environment variables. The services communicate over the cluster network using service names.

4.1.1. Configuration of Services

Each service has an OpenShift service definition that specifies how it can be reached by other services within the cluster.

4.1.2. Environment Variables

Environment variables are used to pass configuration details, such as service endpoints and credentials, securely to the containers.

5. OpenShift Deployment Configuration

5.1. DeploymentConfigs

`DeploymentConfigs` are used to define and manage the deployment of each service. They specify the container image, replicas, triggers, and deployment strategies.

Listing 1: Sample DeploymentConfig for Login-Service

```yaml
apiVersion: apps.openshift.io/v1
kind: DeploymentConfig
metadata:
name: login-service
spec:
replicas: 2
selector:
app: login-service
template:
metadata:
labels:
app: login-service
spec:
containers:
- name: login-container
image:
   image-registry.openshift-image-registry.svc:5000/myproject/login-service:latest
ports:
- containerPort: 5000
```

5.2. Services and Routes

`Services` are configured to expose the deployments internally, while `Routes` are used to expose services externally, enabling access from outside the OpenShift cluster.

5.3. Persistent Storage

`PersistentVolumeClaims` are used to allocate storage for the MongoDB database, ensuring data persists across pod restarts and rescheduling.

6. Data Persistence

6.1. MongoDB Deployment

MongoDB is deployed with persistent storage. OpenShift's storage classes are used to provision storage dynamically.

6.2. Data Reliability

By using `StatefulSets` and `PersistentVolumes`, the database maintains data integrity, and data is retained even if the MongoDB pod is terminated or rescheduled.

7. Scalability and Load Balancing

7.1. Horizontal Scaling

OpenShift's `HorizontalPodAutoscaler` is used to scale the application horizontally based on CPU utilization or other metrics.

7.1.1. Scaling Policy

Policies are defined to scale the number of replicas between a minimum and maximum number based on resource utilization.

Listing 2: Sample HorizontalPodAutoscaler

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
name: login-service-hpa
spec:
maxReplicas: 5
minReplicas: 2
scaleTargetRef:
apiVersion: apps.openshift.io/v1
kind: DeploymentConfig
name: login-service
targetCPUUtilizationPercentage: 50
```

7.2. Load Balancing

OpenShift automatically load balances traffic between pod replicas. Services distribute incoming requests across available pods.

7.3. OpenShift Routes

`Routes` are used to expose services to external clients. They handle SSL termination and provide additional features like path-based routing.

8. Workflow Setup and Deployment

8.1. Continuous Integration and Continuous Deployment with GitHub Actions

8.1.1. Workflow Configuration

GitHub Actions workflows are set up for each repository to automate the build and deployment process.

- **Build Stage**: Code is checked out, and Docker images are built.
- **Push Stage**: Images are pushed to a container registry accessible by OpenShift.
- **Deploy Stage**: OpenShift's `oc` CLI tool is used to apply deployment configurations.

8.1.2. Secrets Management

GitHub repository secrets store sensitive information like OpenShift API credentials, which are accessed securely within workflows.

8.1.3. Sample Workflow File

Listing 3: GitHub Actions Workflow for CI/CD

```
name: CI/CD Pipeline

on:
push:
branches: [ main ]

jobs:
```

```
build-and-deploy:
runs-on: ubuntu-latest
steps:
- name: Checkout Code
uses: actions/checkout@v2

Copy
  - name: Log in to OpenShift
    run: oc login ${{ secrets.OPENSHIFT_SERVER }} --token=${{
        secrets.OPENSHIFT_TOKEN }} --insecure-skip-tls-verify

  - name: Build and Push Image
    run: |
      docker build -t ${{ secrets.REGISTRY_URL
         }}/myproject/login-service:${{ github.sha }} .
      docker push ${{ secrets.REGISTRY_URL }}/myproject/login-service:${{
         github.sha }}

  - name: Update OpenShift Deployment
    run: |
      oc set image dc/login-service login-container=${{
         secrets.REGISTRY_URL }}/myproject/login-service:${{ github.sha }}
```

8.2. Deployment Automation

The deployment process is automated to ensure that any changes pushed to the main branch are automatically built and deployed to the OpenShift cluster.

9. Technical Highlights

9.1. Leveraging OpenShift Features

The project makes extensive use of OpenShift features such as `BuildConfigs`,`DeploymentConfi`
`Routes`, and `PersistentVolumeClaims` to manage the application's lifecycle.

9.2. Containerization Best Practices

By containerizing each microservice, the application benefits from portability, consistency across environments, and simplified dependency management.

9.3. Microservices Architecture

The microservices architecture allows for independent development, testing, and deployment of services, leading to better scalability and maintainability.

9.4. Communication Protocols Documentation

Communication protocols between services are documented, with clear definitions of endpoints, request/response formats, and authentication methods.

10. Challenges and Solutions

10.1. Challenges Encountered

- **Complexity of Microservices**: Managing multiple services increases complexity in deployment and communication.
- **Stateful Services Deployment**: Deploying databases like MongoDB requires careful handling of storage and state.
- **Automating CI/CD**: Setting up secure and efficient CI/CD pipelines that integrate with OpenShift.

10.2. Solutions Implemented

- **Use of OpenShift Templates**: Simplified deployment configurations by using templates for similar services.
- **Persistent Storage Solutions**: Implemented persistent volumes and claims for stateful services.

- **Secure Automation**: Leveraged GitHub Actions with encrypted secrets to automate CI/CD securely.

11. Conclusion

11.1. Achievements

The project successfully met its objectives by:

- Containerizing application components using OpenShift.
- Implementing a microservices architecture aligned with best practices.
- Establishing communication between containers using OpenShift services and routes.
- Configuring deployments using OpenShift DeploymentConfigs.
- Ensuring data persistence with OpenShift-compatible storage solutions.
- Exploring scalability and load balancing using OpenShift's features.

11.2. Lessons Learned

Key takeaways from this project include:

- The effectiveness of OpenShift in managing containerized applications.
- The importance of proper configuration and documentation in microservices communication.
- Strategies for automating deployment workflows securely.

11.3. Future Work

Potential areas for future enhancement:

- Implementing advanced monitoring and logging.
- Exploring service mesh technologies for better microservices management.

- Integrating more sophisticated CI/CD tooling.
- Enhancing the chatbot with additional AI capabilities.