Republic of Tunisia
Ministry of Higher Education and Scientific Research
University of Tunis
Tunis Business School

EcoVista: Cloud-Native Ecotourism Platform

Multi-Container Microservices Deployment using Red Hat OpenShift

Written By

Saja Moussa

Supervised By

Prof. Manel Abdelkader

IT460 Project Report
Academic Year 2025–2026

Contents

1 Introduction

EcoVista is a cloud-native ecotourism web application designed to connect travelers interested in discovering natural touristic destinations in Tunisia. The platform allows users to explore destinations, discover and publish activities, join travel groups, and share reviews. The application promotes sustainable tourism and social interaction between travelers with common interests.

The primary objective of this project is not only functional development, but also the design, containerization, and deployment of a multi-container microservices application using Red Hat OpenShift, in accordance with the IT460 project requirements.

2 Microservices Architecture

The EcoVista application follows a **microservices architecture**, where each core business function is implemented as an independent service. This design improves scalability, fault isolation, and maintainability.

2.1 Implemented Microservices

2.1.1 Authentication Service

Handles user login and registration functionality.

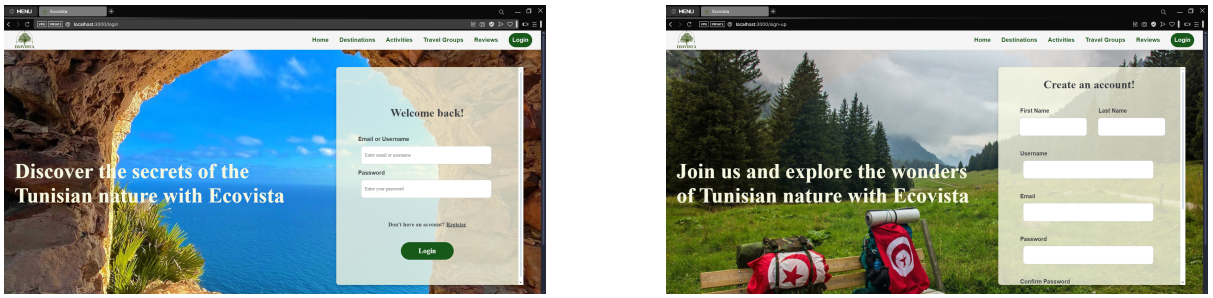


Figure 1: Authentication Service: Login (left) and Registration (right)

2.1.2 User Service

Manages user profiles and personal information.

2.1.3 Activity Service

Manages activities available in destinations.

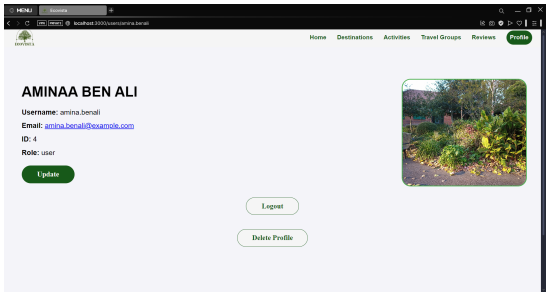


Figure 2: User Service: Main Page (left) and Add/Edit Profile (right)

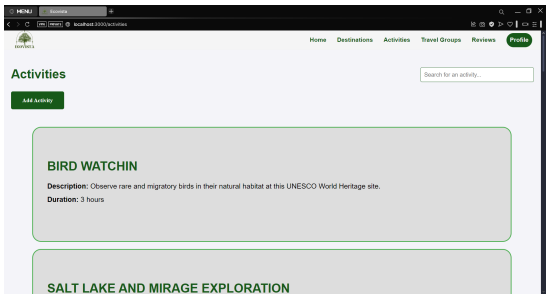


Figure 3: Activity Service: Main Page (left) and Add Activity (right)

2.1.4 Destination Service

Manages touristic locations and their details.

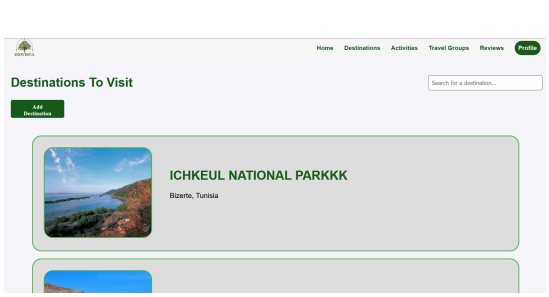


Figure 4: Destination Service: Main Page (left) and Add Destination (right)

2.1.5 Travel Group Service

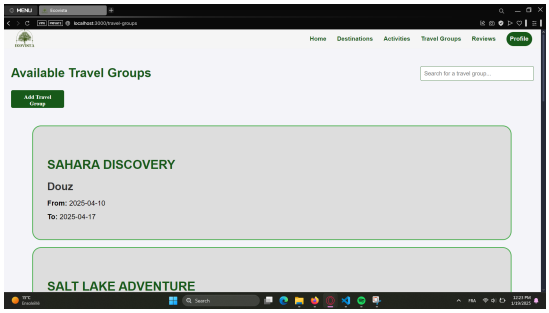
Enables users to find or create travel groups.

2.1.6 Review Service

Allows users to post and view reviews of destinations.

2.1.7 Additional Services

- **Frontend Service**: React-based user interface that integrates all microservices
- **MySQL Database Service**: Persistent relational database for data storage



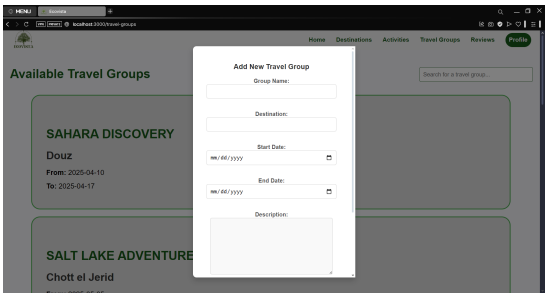
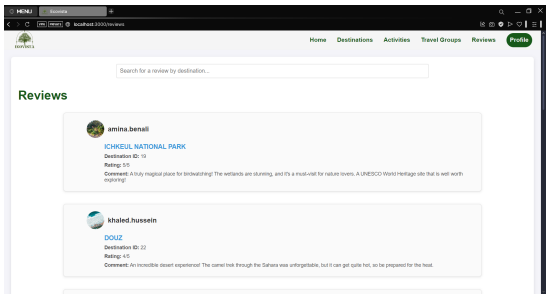


Figure 5: Travel Group Service: Main Page (left) and Create Group (right)



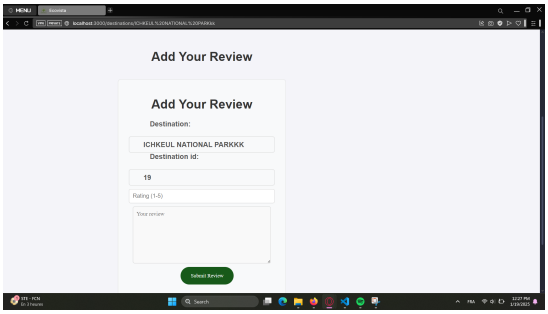


Figure 6: Review Service: Main Page (left) and Add Review (right)

Each service runs in its own container and communicates over the network using HTTP REST APIs.

3 Containerization with OpenShift

Each microservice was containerized using Docker and OpenShift-compatible images. Dedicated Dockerfiles were created for every service, ensuring environment consistency and isolated dependencies.

3.1 Docker Compose Architecture

The development environment utilizes Docker Compose to orchestrate multiple containers. Each Dockerfile represents a single microservice, and Docker Compose manages the communication between all services and the MySQL database.

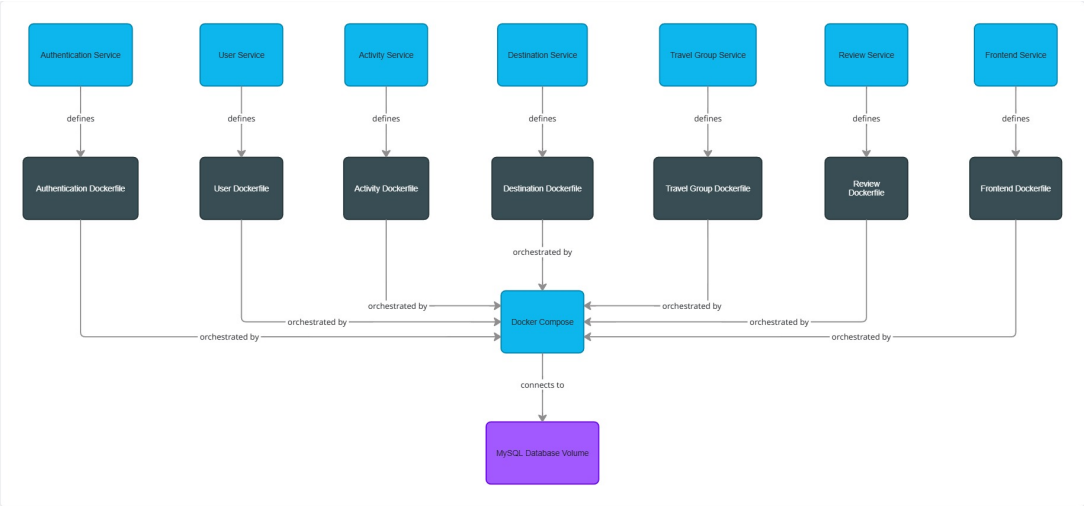


Figure 7: Docker Compose Architecture showing Dockerfiles and service communication

3.2 OpenShift Build Process

The project leverages OpenShift's Source-to-Image (S2I) build strategy through BuildConfigs, which automate the container image creation process directly from the Git repository. BuildConfigs define the source Git repository (https://github.com/sajaa45/Ecovista.git) and the target branch (cloud), enabling automated builds whenever code changes are pushed.

OpenShift containerization features leveraged include:

- Build images from Dockerfiles using OpenShift BuildConfigs
- Store images in OpenShift's internal container registry
- Manage runtime configuration via environment variables and Secrets
- Ensure compatibility with Kubernetes orchestration

This approach eliminates the need for external container registries and provides a fully integrated CI/CD pipeline within OpenShift.

4 Platform Flexibility: Kubernetes vs OpenShift Deployments

The EcoVista project demonstrates platform flexibility by maintaining two separate deployment configurations: one for pure Kubernetes (`k8s/` folder) and another for Red Hat OpenShift (`openshift/` folder). This dual-platform approach showcases a comprehensive understanding of both standard Kubernetes and enterprise OpenShift features.

4.1 Deployment Strategy Comparison

Feature	**k8s/ Folder**	**openshift/ Folder**
Portability	Works on any Kubernetes cluster (GKE, EKS, AKS, minikube)	OpenShift clusters only
Image Building	Manual process requiring external CI/CD pipeline	Automatic via BuildConfigs from Git
External Access	Requires Ingress controller setup	Automatic Routes with HTTPS
Resources Used	Standard Kubernetes objects only	OpenShift-specific resources (BuildConfigs, Routes)
Complexity	Simpler, more standard	More features, enterprise-grade
Security	Manual certificate management	Automatic SSL/TLS certificates

Table 1: Comparison between k8s/ and openshift/ deployment configurations

4.2 Pure Kubernetes Deployment (k8s/ folder)

The `k8s/` folder contains standard Kubernetes manifests that can be deployed on any Kubernetes cluster, including:

- **Google Kubernetes Engine (GKE)**: Google Cloud's managed Kubernetes service
- **Amazon Elastic Kubernetes Service (EKS)**: AWS's managed Kubernetes service
- **Azure Kubernetes Service (AKS)**: Microsoft Azure's managed Kubernetes service
- **Minikube**: Local Kubernetes for development and testing
- **Self-managed Kubernetes**: On-premise or custom installations

This configuration uses only standard Kubernetes resources:

- Deployments for pod management

- Services for internal networking
- Ingress for external access (requires separate configuration)
- PersistentVolumeClaims for data storage
- ConfigMaps and Secrets for configuration

The trade-off is that image building must be handled externally through CI/CD pipelines like Jenkins, GitLab CI, or GitHub Actions, and container images must be pushed to external registries such as Docker Hub or Google Container Registry.

4.3 OpenShift Deployment (openshift/ folder)

The `openshift/` folder leverages OpenShift-specific features that provide enhanced enterprise capabilities:

- **BuildConfigs**: Automate container image building directly from the Git repository (`https://github.com/sajaa45/Ecovista.git`) without external CI/CD tools
- **Routes**: Provide automatic external access with built-in SSL/TLS termination and certificate management
- **Internal Registry**: Store and manage container images within the OpenShift cluster
- **Source-to-Image (S2I)**: Build container images from source code automatically
- **Enhanced Security**: Built-in security scanning, role-based access control, and security contexts

This approach simplifies deployment by eliminating the need for:

- External container registries
- Separate CI/CD pipeline setup
- Manual certificate management
- Complex Ingress controller configuration

4.4 Use Cases for Each Deployment

4.4.1 When to Use k8s/ Folder

- Deploying to cloud providers (GKE, EKS, AKS)
- Local development with minikube or kind
- Multi-cloud or hybrid cloud strategies
- When maximum portability is required
- Working with standard Kubernetes tooling
- Cost-sensitive deployments on managed Kubernetes services

4.4.2 When to Use openshift/ Folder

- Enterprise environments with OpenShift
- When automatic image building from Git is desired
- Organizations requiring advanced security features
- Scenarios needing automatic HTTPS with managed certificates
- Red Hat ecosystem integration requirements
- When developer productivity and automation are priorities

4.5 Project Benefits of Dual Configuration

Maintaining both deployment configurations provides significant advantages:

1. **Platform Versatility**: The application can be deployed on any Kubernetes distribution or OpenShift without code changes
2. **Professional Best Practice**: Demonstrates understanding of both standard Kubernetes and enterprise OpenShift platforms
3. **Future-Proof Design**: Not locked into a single platform vendor or ecosystem
4. **Educational Value**: Clearly illustrates the differences between pure Kubernetes and OpenShift deployments
5. **Migration Flexibility**: Easy migration between platforms as organizational needs change
6. **Cost Optimization**: Can choose the most cost-effective platform for different deployment environments (development vs production)

5 Communication Between Containers

Inter-service communication is implemented using **OpenShift Services**, which provide internal DNS-based service discovery. The application exposes external endpoints via OpenShift Routes with HTTPS termination.

5.1 OpenShift and Kubernetes Communication

The following diagram illustrates the container communication architecture within OpenShift and Kubernetes:

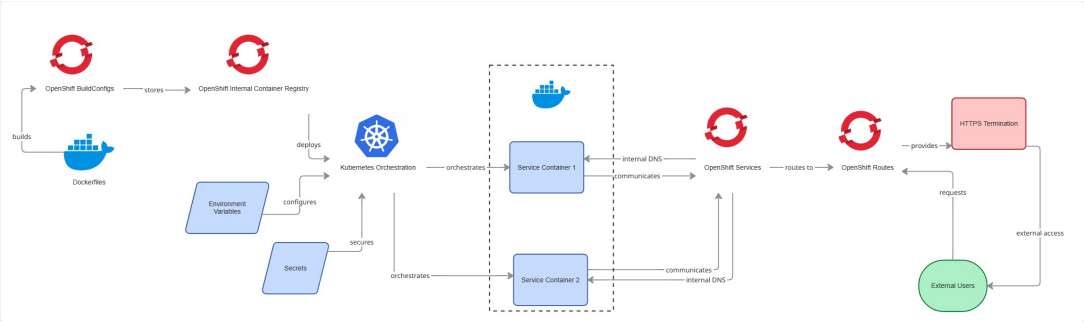


Figure 8: Container communication in OpenShift and Kubernetes environment

5.2 Communication Architecture

The internal communication flow between services is illustrated below:

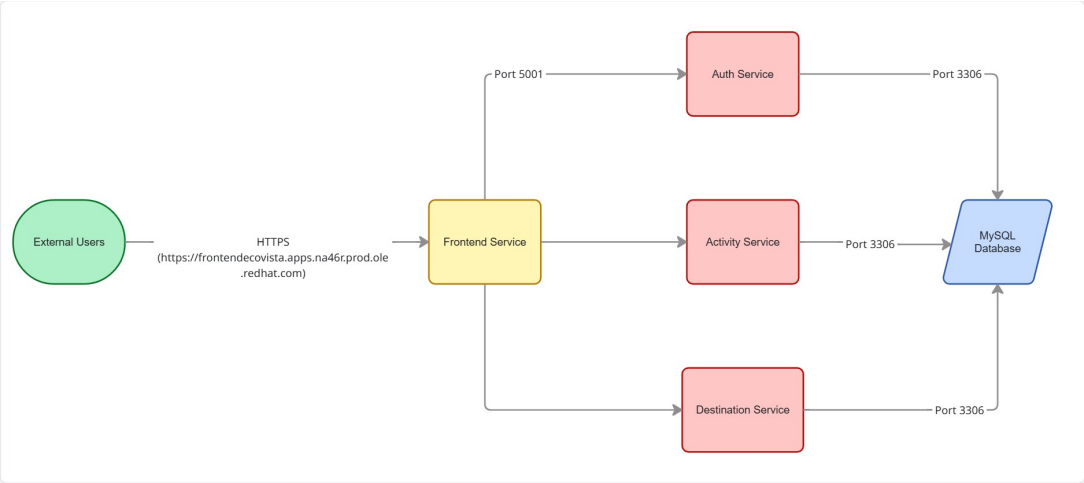


Figure 9: Internal communication architecture between microservices

The communication architecture implements the following flow: External users access the application via HTTPS through the frontend route. The frontend communicates with backend services using internal DNS names (e.g., auth-service:5001, activity-service, destination-service). All backend services connect to the MySQL database on port 3306 for data persistence.

5.3 Communication Features

- **REST API**: Each backend service exposes HTTP/JSON endpoints
- **Internal DNS**: Services are reachable via their service names (e.g., `auth-service`)
- **OpenShift Routes**: External access managed with SSL termination
- **CORS Configuration**: Cross-origin resource sharing enabled for frontend-backend communication

This approach follows OpenShift networking best practices and ensures secure, reliable communication between all application components.

6 OpenShift Deployment Configuration

The application was deployed using comprehensive OpenShift resources defined in YAML configuration files.

6.1 OpenShift Objects Used

- **Deployments / DeploymentConfigs**: Manage pod lifecycle and replica sets for all microservices
- **Services**: Enable internal networking and load balancing between pods
- **Routes**: Expose services externally via HTTPS with SSL termination
- **Secrets**: Store sensitive data such as database credentials securely
- **ConfigMaps**: Store configuration values and database initialization scripts
- **BuildConfigs**: Automate image building from Git repository

6.2 Deployment Files Structure

The project includes well-organized deployment configurations:

- `complete-deployment.yaml`: All-in-one deployment configuration for all services
- `routes.yaml`: External access configuration with HTTPS
- `secrets.yaml`: Secure credential management
- `buildconfigs-git.yaml`: Automated build process from GitHub

OpenShift automatically restarts pods in case of failure, ensuring high availability and self-healing capabilities.

7 Data Persistence

Data persistence is implemented using a **Persistent Volume Claim (PVC)** for the MySQL database, ensuring data survives pod restarts and redeployments.

7.1 Persistence Architecture

The following diagram illustrates the persistent volume claim architecture:

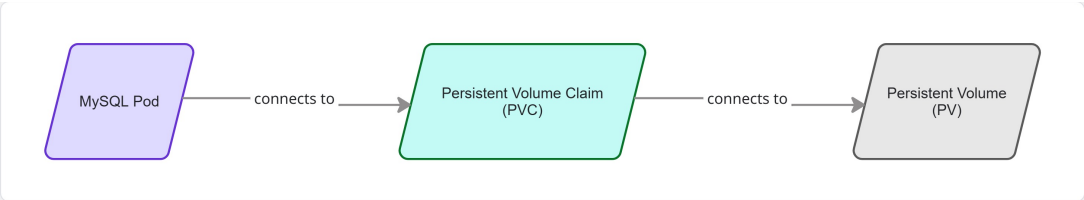


Figure 10: Persistent Volume Claim architecture for MySQL database

7.2 Implementation Details

The MySQL database uses OpenShift's persistent storage with a PersistentVolumeClaim (PVC) configuration. The PVC is configured with ReadWriteOnce access mode and requests 1Gi of storage. This ensures that the MySQL data directory is mounted to persistent storage that survives pod lifecycle events.

This configuration guarantees:

- Data survives pod restarts and failures
- Database state is preserved across redeployments
- Automatic volume provisioning by OpenShift
- Database initialization through ConfigMaps
- Secure password management via Secrets

Persistent storage is fully managed by OpenShift and complies with IT460 requirements.

8 Scalability and Load Balancing

The application architecture supports horizontal scaling and automatic load balancing through OpenShift's orchestration capabilities.

8.1 Horizontal Scalability

OpenShift enables horizontal scaling of microservices by adjusting the number of pod replicas in the deployment configuration. Each deployment specifies a replica count that can be modified dynamically. For example, the activity-service can be configured with 2 replicas, allowing OpenShift to distribute the workload across multiple pods.

Scaling can be performed dynamically using the OpenShift CLI with commands like: `oc scale deployment activity-service --replicas=3`, which immediately adjusts the number of running pods to meet demand.

8.2 Load Balancing

Load balancing is handled automatically at multiple levels:

- **Internal Load Balancing**: OpenShift Services distribute traffic across pod replicas using round-robin
- **External Load Balancing**: OpenShift Routes use HAProxy for external traffic distribution
- **Health Checks**: Automatic detection and removal of unhealthy pods from rotation
- **Rolling Updates**: Zero-downtime deployments with gradual pod replacement

8.3 Current Deployment

The application currently runs with 8 pods:

- 6 microservice pods (auth, activity, destination, review, travelgroup, user)
- 1 frontend pod
- 1 MySQL database pod

Although the current deployment uses one replica per service, the architecture fully supports horizontal scaling without any application-level changes.

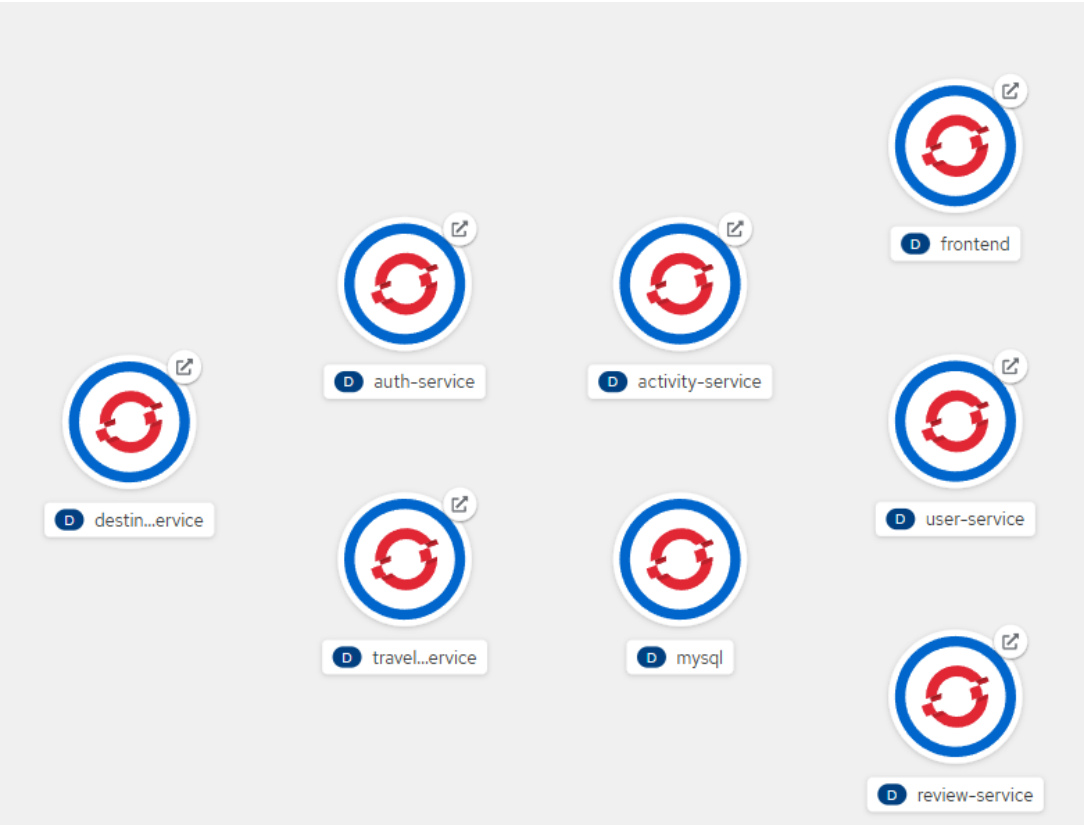


Figure 11: OpenShift Container Platform - Pod Topology View

9 Comparison with Docker Compose

Feature	Docker Compose	OpenShift
Scalability	Limited	Automatic
Self-healing	No	Yes
Public HTTPS URLs	No	Yes
Persistent Storage	Basic volumes	PVCs
Load Balancing	No	Yes
Rolling Updates	No	Yes

Table 2: Comparison between Docker Compose and OpenShift

10 Challenges and Lessons Learned

Several technical challenges were encountered during the development and deployment process, providing valuable hands-on experience with real-world cloud-native deployment scenarios.

10.1 Technical Challenges

1. **CORS Configuration**: Configuring cross-origin resource sharing between the React frontend and Flask backend services required careful setup of HTTP headers and allowed origins.

2. **Database Schema Synchronization**: Ensuring consistent database schemas across multiple microservices that share the MySQL database required coordination and migration strategies.

3. **MySQL Authentication**: Configuring secure database authentication in a containerized environment with proper credential management through OpenShift Secrets.

4. **Git Branch Configuration**: Setting up OpenShift BuildConfigs to pull from the correct Git branch (cloud) and trigger automated builds.

5. **Service Routing**: Correctly configuring internal service-to-service communication and external routing in OpenShift's networking model.

6. **Migration Path**: Transitioning from local Docker Compose development to Kubernetes and finally to OpenShift required understanding the differences in networking, storage, and configuration management.

10.2 Key Lessons Learned

- **API Design**: Microservices require careful API design and versioning to maintain loose coupling while ensuring proper integration.

- **Container Orchestration**: OpenShift and Kubernetes significantly simplify scaling, deployment, and management compared to manual container orchestration.

- **Enterprise Features**: OpenShift provides enterprise-grade features beyond plain Kubernetes, including integrated CI/CD, enhanced security, and developer-friendly tooling.

- **Development Workflow**: Starting with Docker Compose for local development, then migrating to Kubernetes, and finally deploying to OpenShift proved to be an effective incremental approach.

- **Infrastructure as Code**: Maintaining all configurations in YAML files enables version control, reproducibility, and easy disaster recovery.

11 Conclusion

This project successfully demonstrates the design, containerization, and deployment of a multi-container microservices application using Red Hat OpenShift. By leveraging Docker, Kubernetes, and OpenShift, EcoVista achieves scalability, reliability, and production-grade deployment.

11.1 Project Achievements

The EcoVista platform represents a complete implementation of modern cloud-native architecture:

- **Complete Microservices Architecture**: Six well-defined, loosely coupled services implementing single responsibility principle
- **Production-Ready Deployment**: Live application accessible at `https://frontend-ecovista.apps.na46r.prod.ole.redhat.com` with HTTPS endpoints and persistent storage
- **OpenShift Best Practices**: Proper utilization of BuildConfigs, Routes, Services, and persistent volumes
- **Scalable Design**: Each service can be independently scaled without impacting other components
- **Real-World Application**: Functional eco-tourism platform with user authentication, CRUD operations, and social features

11.2 IT460 Objectives Compliance

The project fully aligns with all IT460 objectives:

1. Containerization using OpenShift BuildConfigs and internal registry
2. Microservices architecture with six independent services
3. Container communication through OpenShift Services and Routes
4. Comprehensive OpenShift deployment configuration
5. Data persistence using Persistent Volume Claims
6. Scalability and load balancing capabilities

This implementation reflects modern enterprise container orchestration practices and demonstrates proficiency in cloud-native application development and deployment using Red Hat OpenShift.