Containerizing Employees Spring Boot REST Service on AWS

SWER415 - Spring 2025

1. Introduction

This project aims to containerize a Spring Boot RESTful Employee Service and deploy it on AWS using ECS/Fargate. The service is integrated with a relational database to perform CRUD operations securely and efficiently.

2. System Overview

The system consists of a Spring Boot REST API, a MySQL database (via Amazon RDS or Docker), and AWS services for container orchestration, networking, and deployment. The project follows a microservice deployment pattern.

3. Docker Containerization

The application is containerized using Docker. A Dockerfile builds the Spring Boot JAR and exposes the application on port 8080. A Docker Compose file can be used for local integration testing with MySQL.

4. Database Integration - Scenario A (Amazon RDS)

Amazon RDS for MySQL is used as the managed database. The application.properties file is configured to connect to RDS using secure credentials. Connection String Example:

spring.datasource.url=jdbc:mysql://<rds-endpoint>:3306/employees_db

spring.datasource.username=admin

spring.datasource.password=${DB_PASSWORD}

5. AWS Deployment Instructions

Containerizing Employees Spring Boot REST Service on AWS

SWER415 - Spring 2025

1. Build Docker image: docker build -t employees-api .

2. Push to ECR: tag and push the image to Amazon ECR repository

3. Create ECS Cluster and Task Definition using AWS Fargate

4. Configure networking (VPC, subnets, security groups)

5. Deploy the service and access the public endpoint

Example Endpoint URL:

http://<ecs-service-url>/employees

6. Design Decisions Summary

- Chose Amazon RDS for managed database benefits (automatic backups, monitoring, scalability)

- Deployed with Fargate for serverless and cost-effective container hosting

- Used AWS Secrets Manager for secure environment variables

7. RDS vs Dockerized MySQL Comparison

Amazon RDS:

+ Managed service

+ Automated backups and monitoring

+ Scalable

- Higher cost

Dockerized MySQL:

+ Full control

+ Cost-effective for small scale

- Requires manual backups and maintenance

- Networking overhead

8. Challenges and Lessons Learned

- Learned to securely connect services across AWS

- Understood cost considerations with AWS services

- Explored container orchestration and image management

- Gained insights into deployment automation

9. Conclusion

This project demonstrated the end-to-end process of containerizing a Spring Boot application, integrating it with a database, and deploying it securely and efficiently on AWS.