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. Docker Containerization

The application is containerized using Docker. A Dockerfile builds the Spring Boot JAR and exposes the application on port 8080.

## 3. 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.

spring.datasource.url=jdbc:mysql://database1.c474hxyw4lbq.us-east-1.rds.amazonaws.com:3306/database1

spring.datasource.username=admin

spring.datasource.password=dU9to|![S<nXpwx>ym[#:wfF6fT7

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

## 4. AWS Deployment Instructions

1. Build Docker image: docker build -t payroll-service .  
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  
  
Endpoint URL:  
http://3.230.0.156:8080/employees

## 5. 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

## 6. Technology Decisions and Comparisons

### 6.1 . RDS vs Dockerized MySQL Comparison

| **Aspect** | **Amazon RDS (Managed)** | **Dockerized MySQL (Self-Managed)** |
| --- | --- | --- |
| **Setup Time** | Faster and easier (GUI) | Requires manual setup and network configuration |
| **Maintenance** | Automatic | Backup and updates need to be managed manually |
| **Scalability** | Flexible (scale without downtime) | Difficult and manual |
| **Monitoring** | CloudWatch integrated | External tools are needed |
| **Security** | IAM, VPC, and SSL integrated | Everything must be set up manually |
| **Cost Efficiency** | Slightly higher but less effort | Cheaper but higher risk |

### 6.2AWS Service Choice: Fargate vs EC2

We chose to use AWS Fargate because it allows us to deploy our containerized Spring Boot REST service without managing the underlying infrastructure. It simplifies the deployment process, automatically handles scaling, and integrates seamlessly with other AWS services like Amazon RDS, IAM, and CloudWatch. This makes it an ideal solution for our project, as it reduces operational overhead and lets us focus on application development rather than server management.

## 7. 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

## 8. 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.