מכון טכנולוגי חולון
Holon Institute of Technology

Introduction to DevOps Final Project

Requirements

This final project is divided into three parts: building a Spring Boot web DevOps server, developing Python tests to validate the server, and orchestrating the application with Docker in the Play with Docker environment. Each part includes essential DevOps practices, tools, and concepts, forming a cohesive continuous deployment pipeline project.

1. You may choose one of the dates for the project presentation.
2. You may submit the final project on your own or in groups of two or three students.
3. You may use ChatGPT/other LLM as long as you mention the parts in which you used the generated code/assisted in a generated example.

Part 1: DevOps Server - Continuous Integration and Deployment (CI/CD) Automation Server

Develop a Spring Boot application that serves as a Continuous Integration and Deployment (CI/CD) automation server. This server will manage CI/CD jobs, log job statuses, and provide various endpoints for CRUD operations on CI/CD jobs. The project will cover unit testing, integration testing, configurations, logging, dependency injection, and more.

1. Set up a Spring Boot Project: Initialize a new Spring Boot project with dependencies: Spring Web, Spring Data JPA, H2 Database, and Lombok. Configure application properties to set up H2 database.
2. Create a CI/CDJob Entity: Define a CI/CDJob entity class with fields like id, jobName, status, createdAt, updatedAt, and jobType. Annotate with `@Entity` and `@Table
3. Create a Repository Interface: Create a CI/CDJobRepository interface extending JpaRepository. Include custom JPA repository methods to find jobs by status, jobType, and date range.

גולומב 52, ת.ד 305, חולון 58102
טלפון 03-5026528, פקס' 03-5026733
52 Golomb St., Holon 58102 Israel
www.hit.ac.il Tel. 972-3-502-6528, Fax. 972-3-502-6733
הפקולטה למדעים
המחלקה למדעי המחשב
Faculty of Sciences
Department of Computer Science

4. Implement a Service Class: Implement CI/CDJobService with methods to handle CRUD operations. Use `@Service` annotation and inject CI/CDJobRepository.
5. Develop a Controller Class: Create CI/CDJobController with endpoints for CRUD operations:

@` -GetMapping("/jobs")`: Retrieve all jobs.

@` -PostMapping("/jobs")`: Create a new job.

@` -GetMapping("/jobs/{id}")`: Retrieve a job by ID.

@` -PutMapping("/jobs/{id}")`: Update a job.

@` -DeleteMapping("/jobs/{id}")`: Delete a job.

@` -GetMapping("/jobs/status/{status}")`: Retrieve jobs by status.

@` -GetMapping("/jobs/jobType/{jobType}")`: Retrieve jobs by job type.

@` -GetMapping("/jobs/date-range")`: Retrieve jobs by a date range.

6. Create a Configuration Class: Create a DatabaseSeeder class with `@Configuration` annotation. Define a CommandLineRunner bean to seed the database with initial CI/CD jobs.
7. Implement Password Encoding: Use BCryptPasswordEncoder to securely encode passwords.
8. Create Data Transfer Objects (DTOs): Define DTOs for creating and updating CI/CD jobs with Lombok annotations for getters and setters.

Testing

Standard Unit Tests (@Test): Test individual methods in the service class for specific input-output scenarios. Examples: testAddJob(), testGetJob.()

Parameterized Tests (@ParameterizedTest): Test methods with multiple input parameters. Examples: testAddJobWithVariousStatuses(), testGetJobByDifferentIds.()

Nested Tests (@Nested): Group related test cases for better organization. Examples: CI/CDJobServiceTests with nested classes for CRUD operations.

Exception Tests (assertThrows): Verify that methods throw expected exceptions. Examples: testDeleteNonExistentJob(), testUpdateJobWithInvalidData.()

Integration Tests (@SpringBootTest): Test the interaction between multiple components. Example: CI/CDJobControllerTest class with @SpringBootTest.

Logging (Logger): Configure and use Logger for logging information and debugging. Example: private static final Logger logger = LoggerFactory.getLogger(CI/CDJobController.class.;(

Part 2: Python Tests with/without Flask

Automated CI/CD Job Tester

Develop Python tests to validate the Continuous Integration and Deployment (CI/CD) automation server. This part involves writing integration tests, logging, and using fixtures to set up the test environment.

1. Set up the Python Environment Create a virtual environment and install necessary packages
2. Configure the Logger: Configure logging to output log messages with timestamps and log level
3. Create a Logging Utility Function: Create a utility function to log HTTP response status and body.
4. Define Test Fixtures (@pytest.fixture): Define fixtures to set up and tear down test resources. Example: setup fixture with module scope.

5. Write Test Functions: Write test functions to verify the functionality of specific endpoints:

-test_get_all_jobs(): Ensure retrieval of all CI/CD jobs works correctly.

-test_create_job(): Ensure creation of a new CI/CD job works correctly.

-test_get_job_by_id(): Ensure retrieval of a specific CI/CD job by ID works correctly.

-test_update_job_status(): Ensure updating the status of a CI/CD job works correctly.

-test_delete_job(): Ensure deletion of a CI/CD job works correctly.

Part 3: Advanced Docker Project with Play with Docker

Continuous Deployment Pipeline Orchestration

In this part, we'll orchestrate the Continuous Integration and Deployment (CI/CD) automation server and the Python tests using Docker Compose in the Play with Docker environment. This project will involve creating Dockerfiles, defining services, managing volumes, custom networking, and scaling services.

1. Access Play with Docker: Start a new session in Play with Docker.
2. Create a Docker Compose File: Create and edit docker-compose.yml to define services for the CI/CD automation server, Redis, PostgreSQL, and the tester.

Include the following services:

-cd-server: Spring Boot application.

-redis: Caching service.

-db: PostgreSQL database.

-tester: Python testing service.

3. Define Docker Compose details such as service dependencies, network configurations, and volume mappings for persistent data storage.
4. Create Dockerfile for the Continuous Integration and Deployment Server**: Define a Dockerfile for the Spring Boot application, including necessary dependencies and configurations.

5. Create Dockerfile for the Tester: Define a Dockerfile for the Python tests, specifying the required dependencies.
6. Build and Launch the Application: Use Docker Compose to build and launch the multi-container application.
7. Access and Verify: Access the CI/CD automation server on the defined port. Ensure the Python tests are executed correctly.

גולומב 52, ת.ד 305, חולון 58102
טלפון 03-5026528, פקס' 03-5026733
52 Golomb St., Holon 58102 Israel
www.hit.ac.il Tel. 972-3-502-6528, Fax. 972-3-502-6733
הפקולטה למדעים
המחלקה למדעי המחשב
Faculty of Sciences
Department of Computer Science