Name : Dev Gaonkar Div: D15C Roll no.: 12

Adv. Dev Ops Assignment No. 2

05/05

Create a REST API with the Server

Ans.

1. Set Up AWS and IAM User:

- Create AWS Account: Sign up for an AWS account and create an IAM user with AdministratorAccess. Configure the AWS CLI with the access keys for authentication.

2. Install Tools:

- Install Node.js and the Serverless Framework globally. The Serverless Framework simplifies Lambda and API Gateway deployement.

3. Create a Serverless Project:

- Create a new Serverless project using the AWS Node.js template. This generates the necessary files like serverless.yml for configuration and handler.js for defining Lambda functions.

4. Define Lambda Functions:

- In the handler file, write the Lambda functions to handle API requests, such as fetching or creating user data. These small functions will be trigered by HTTP requests routed through API Gateway.

5. Configure API Gateway:

- In serverless.yml, specify the events (like GET or POST) that will trigger the Lambda functions. This setup links API Gateway routes to your Lambda functions, defining paths like /user for different operations.

6. Deploy to AWS:

- Deploy the project using the Serverless Framework. It will automatically create and configure AWS resources such as Lambda functions, API Gateway, and permissions.

7. Test and Monitor:

- Use the provided API Gateway URL to test your API. AWS CloudWatch logs help monitor and debug any issue in the Lamda functions or API Gateway requests.

Through this experiment, I learned to create and deploy a serverless REST API using AWS Lamda and the Serverless Framework, configuration API Gateway, writing Lambda functions, and managing cloud resources efficiently with scalable, serverless architecture.

Q.2 Case Study: Using SonarQube and SonarCloud for Code Quality Analysis.

1. Introduction

In this case study, we set up SonarQube and SonarCloud to evaluate the quality of various projects, including Python, Node.js, and Java, ensuring high standards by detecting code smells, vulnerabilities, and bugs.

2. Creating a SonarQube Profile.

To begin, SonarQube was installed on a local machine, accessed via 'http://localhost:9000', and configured by logging in as an admin. A new quality profile was created, tailored specifically for Python projects. The profile focused on detecting common issues such as code duplication, maintainability concerns, and security vulnerabilities.

3. Analyzing a Python Project with SonarQube

A large Python project was analyzed using the SonarQube sanner, which was configured as follows:

```
sonar-scanner -Dsonar.projectKey=python-project -Dsonar.sources=./src
```

The analysis report highlighted 15 critical bugs and several code smells, many related to poor code formatting, unused variables, and security vulnerabilities. SonarQube provided recommendations to address these issues, leading to improved code maintainability and security.

4. Analyzing a Node.js Project

For a Node.js project, the SonarQube scanner was integrated via the "sonar-project.properties" file. The scan revealed 12 vulnerabilities and 20 code smells, with issues related to error handling and the use of deprecated functions. Recommendations from SonarQube led to restructuring the code to enhance performance and security.

5. SonarCloud Integration for Java Projects

A Java project hosted on GitHub was analyzed using SonarCloud. The repository was linked with SonarCloud, and SonarLint was installed in VSCode for real-time feedback during development. Continuous integration with Github Actions enabled ongoing monitoring of code quality as new commits were made. The analysis identified 10 major bugs along with minor issues, contributing to more robust and secure code.

6. Conclusion.

The use of SonarQube and Sonar Cloud enhanced the code quality of Python, Node.js, and Java projects by identifying and resolving critical issues. These tools provided actionable feedback, improving maintainability, security, and overall code robustness.

Q. 3 Terraform 'self-serve' Infrastructure Model

① Terraform Modules for self-serve infrastructure

• Create Terraform modules that codify the standards for deploying common resources like VPCs, EC2 instances, and S3 buckets.

Example module for an EC2 instance:

ec2 - module / main.tf :

```
variable "instance-type" {
  default = "t2.micro"
}
resource "aws_instance" "example" {
  ami = "ami - 12345678"
instance-type = var.instance_type
tags = {
        Name = "example - instance"
    }
}
```

ec2-module / outputs.tf :

```
Output "instance_id" {
  value = aws_instance.example.id
}
```

Teams can now use this module to deploy EC2 instance with

```
module "ec2" {
  source = "./ec.2-module"
  instance-type = "t2.medium"
}
```

(2) Terraform Cloud Integration with Service Now.

- You can integrate Terraform Cloud with Service Now to automate the infrastructure request process.
- Using Terraform's API-driven approach, Service Now can trigger Terraform runs based on ticket approvals, automating resource deployment.

Example workflow

1) A product team submits a request in Service Now for new infrastructure.
2) The request triggers a Terraform Cloud updates the Service Now ticket with the status and resource details.