1. **Write a spring boot application. no logic needed. just hello world is fine.**

Here's a simple Spring Boot application that prints "Hello, World!" when accessed: -

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication

@RestController

public class HelloWorldApplication {

public static void main(String[] args) {

SpringApplication.run(HelloWorldApplication.class, args);

}

@GetMapping("/")

public String helloWorld() {

return "Hello, World!";

}

}

This is a minimal Spring Boot application with a single endpoint mapped to the root URL ("/"). The @RestController annotation marks the class as a RESTful controller, and the @GetMapping annotation specifies the HTTP GET method for the helloWorld() method. When you run this application and access http://localhost:8080/ in your browser or through an HTTP client, you will see the message "Hello, World!" displayed.

1. **Connect this spring boot application to a database using testcontainers.**

To connect your Spring Boot application to a database using Testcontainers, you will need to perform the following steps:

**Step 1: Add Dependencies**

Add the necessary dependencies to your project's pom.xml file:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>org.testcontainers</groupId>

<artifactId>testcontainers</artifactId>

<version>1.15.3</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.testcontainers</groupId>

<artifactId>postgresql</artifactId>

<version>1.15.3</version>

<scope>test</scope>

</dependency>

**Step 2: Configure Database Connection**

Configure the database connection in your application.properties file:

spring.datasource.url=jdbc:tc:postgresql://hostname/databasename

spring.datasource.driver-class-name=org.testcontainers.jdbc.ContainerDatabaseDriver

spring.datasource.username=postgres

spring.datasource.password=postgres

**Step 3: Write a Test Class**

Create a test class to demonstrate the connection to the database using Testcontainers:

import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.test.context.SpringBootTest;

import org.testcontainers.containers.PostgreSQLContainer;

import org.testcontainers.junit.jupiter.Container;

import org.testcontainers.junit.jupiter.Testcontainers;

@SpringBootTest

@Testcontainers

public class HelloWorldApplicationTests {

@Container

private static final PostgreSQLContainer<?> postgresContainer = new PostgreSQLContainer<>("postgres:latest");

@Autowired

private YourRepository yourRepository;

@Test

public void testDatabaseConnection() {

// Perform any database-related operations here

// For example, you can save or retrieve data from the database

// Example:

YourEntity entity = new YourEntity();

entity.setName("John Doe");

yourRepository.save(entity);

YourEntity retrievedEntity = yourRepository.findById(entity.getId()).orElse(null);

assertNotNull(retrievedEntity);

assertEquals("John Doe", retrievedEntity.getName());

}

}

In this example, we're using the PostgreSQL container from Testcontainers to spin up a temporary PostgreSQL database for testing. The @Container annotation creates and manages the database container. You can use the postgresContainer.getJdbcUrl() method to retrieve the JDBC URL dynamically if needed.

The @Autowired annotation is used to inject an instance of your repository or any other Spring Data JPA components.

The testDatabaseConnection() method demonstrates a basic test scenario where data is saved to the database and then retrieved to ensure the connection is working correctly. Modify this method to fit your specific use case.

Make sure to replace YourRepository with the actual repository class you are using, and YourEntity with the entity class you are working with.

Now, when you run your test class, Testcontainers will automatically start a PostgreSQL container, and your Spring Boot application will connect to it for testing purposes.

1. **Write a terraform script to deploy this spring boot application to AWS Lambda**

To deploy a Spring Boot application to AWS Lambda using Terraform, you can follow these steps:

**Step 1: Set up your Terraform project**

Create a new directory for your Terraform project and initialize it by running the terraform init command. This command downloads the necessary provider plugins.

**Step 2: Write the Terraform configuration file**

Create a new file called main.tf and add the following code:

provider "aws" {

region = "us-east-1" # Replace with your desired AWS region

}

resource "aws\_lambda\_function" "spring\_boot\_app" {

function\_name = "spring-boot-lambda"

role = aws\_iam\_role.lambda\_exec\_role.arn

handler = "com.example.Application::handleRequest" # Replace with your actual handler class and method

runtime = "java11"

timeout = 60 # Maximum execution time in seconds

// Zip your Spring Boot application JAR file and upload it to S3

filename = "app.zip"

source\_code\_hash = filebase64sha256("app.zip")

// Specify the S3 bucket and object key where your application JAR is stored

s3\_bucket = "your-s3-bucket"

s3\_key = "your-s3-key"

}

resource "aws\_iam\_role" "lambda\_exec\_role" {

name = "lambda-exec-role"

assume\_role\_policy = <<EOF

{

"Version": "2012-10-17",

"Statement": [

{

"Sid": "",

"Effect": "Allow",

"Principal": {

"Service": "lambda.amazonaws.com"

},

"Action": "sts:AssumeRole"

}

]

}

EOF

}

Make sure to replace the placeholders (com.example.Application::handleRequest, your-s3-bucket, and your-s3-key) with your actual values. The handler value should point to the entry point of your Spring Boot application in the format <package.ClassName::methodName>.

**Step 3: Zip your Spring Boot application**

Create a zip file containing your Spring Boot application JAR file. Place the zip file in the same directory as the Terraform files.

**Step 4: Deploy the application**

Run the following Terraform commands in your project directory:

terraform init # Initialize Terraform

terraform plan # Preview the changes

terraform apply # Deploy the application

Review the plan and confirm the deployment when prompted.

Terraform will create the necessary AWS Lambda function and IAM role. It will also upload your Spring Boot application to an S3 bucket and configure the Lambda function to use that file.

After the deployment is successful, you can access your Spring Boot application by invoking the Lambda function using the AWS Lambda API or by triggering it with an event source.

**(I took the help of Google and other online resources for spring boot application and connecting it to database, as I did not have any prior knowledge and experience in Spring boot or Java Graal).**