**Cloud Vendor BackEnd Application:**

**1st Video:**

Create a Spring Boot project using Spring Initializr.

Add dependencies such as Spring Web,Spring Data JPA,MySQL Driver in the project.

These dependencies will be automatically reflected in **pom.xml**

Import it in IDE

DELETE

PUT

POST

GET

Cloud Vendor Info. Services

Cloud Vendor Properties

Vendor Id,

Vendor Name,

Vendor Address,

Vendor Phone Number

**Created CRUD Operation REST APIs Without DB Involvement.**

**Tested Each REST API endpoint using POSTMAN.**

**Tried to use ResponseEntity<V> instead of simple Object type as return type.**

**Custom Exception Handling in Cloud Vendor Application:**

Extending RunTime Exception makes our custom exception as unchecked exception

**public** **class** CloudVendorNotFoundException **extends** RuntimeException{

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**public** CloudVendorNotFoundException(String message) {

**super**(message);

}

**public** CloudVendorNotFoundException(String message, Throwable cause) {

**super**(message,cause);

}

By extending **RuntimeException**, you simplify the exception handling code in your application. You don't need to clutter your method signatures with **throws** declarations for exceptions that are unlikely to be caught and recovered from at the point of method invocation.

}

**CloudVendorServiceImpl.java:**

@Override

**public** CloudVendor getCloudVendor(String vendorId) {

Optional<CloudVendor> cloudVendorDetail = cloudVendorRepo.findById(vendorId);

**if**(cloudVendorDetail.isEmpty() && ***LOGGER***.isErrorEnabled()) {

***LOGGER***.info(String.*format*("No Cloud Vendor With VendorId=%s", vendorId));

**throw** **new** CloudVendorNotFoundException("Cloud Vendor Not Found For VendorId:"+vendorId);

}

**return** (cloudVendorDetail.isPresent())?cloudVendorDetail.get():**null**;

}

**CloudVendorGlobalExeptionHandler.java:**

@RestControllerAdvice

**public** **class** CloudVendorGlobalExceptionHandler {

@ExceptionHandler(CloudVendorNotFoundException.**class**)

**public** ResponseEntity<String> handleCloudVendorNotFoundException(CloudVendorNotFoundException cloudVendorNotFoundException){

**return** **new** ResponseEntity<>(cloudVendorNotFoundException.getMessage(),HttpStatus.***NOT\_FOUND***);

}

}

@RestControllerAdvice- It makes the class as global exception handler for all the controllers.

We can also use **@ControllerAdvice** only.

When you use **@ControllerAdvice**, the assumption is that you are handling exceptions for both regular web controllers (returning views) and REST controllers (returning data). The methods within **@ExceptionHandler** can return different types of responses, such as **ModelAndView** for views and **ResponseEntity** for data.

@ExceptionHandler:

It is basically used within a controller or controller advice class (**@ControllerAdvice)**

It is used at the method-level

When used within a controller, it handles exceptions specific to that controller

When used within a controller advice class, it handles exceptions globally across multiple controllers

@ExceptionHandler(CloudVendorNotFoundException.**class**)

**public** ResponseEntity<String> handleCloudVendorNotFoundException(CloudVendorNotFoundException cloudVendorNotFoundException){

**return** **new** ResponseEntity<>(cloudVendorNotFoundException.getMessage(),HttpStatus.***NOT\_FOUND***);

}

**Custom Response Handling in REST APIs:**

Generic Response:

{

    "vendorId": "C2",

    "vendorName": "vendor 2",

    "vendorAddress": "Magarpatta City, Pune",

    "vendorPhoneNumber": "8230001908"

}

Custom Response: Often we need to display or return the REST API Response as below and for that purpose, we need to perform custom response handling:

{

    “data”:{

"vendorId": "C2",

    "vendorName": "vendor 2",

    "vendorAddress": "Magarpatta City, Pune",

    "vendorPhoneNumber": "8230001908"

},

"httpStatus": "OK”,

”message”: "Requested Vendor Details are given here”

}

/\*\*

\* Custom Response Handling

\*/

**public** **class** CloudVendorResponseHandler {

**private** CloudVendorResponseHandler() {

// Private constructor to prevent instantiation

**throw** **new** UnsupportedOperationException("Utility class - cannot be instantiated");

}

**public** **static** ResponseEntity<Object> responseHandler(String message,Object object,HttpStatus httpStatus){

Map<String,Object> responseMap = **new** HashMap<>();

responseMap.put("message", message);

responseMap.put("data", object);

responseMap.put("httpStatus", httpStatus);

**return** **new** ResponseEntity<>(responseMap,httpStatus);

}

}

/\*\*

\* Fetching a cloud vendor using vendor Id

\*/

@GetMapping("{vendorId}")

**public** ResponseEntity<Object> getCloudVendorDetails(@PathVariable String vendorId) {

CloudVendor cloudVendor= cloudVendorService.getCloudVendor(vendorId);

**if** (cloudVendor == **null**) {

//Return a 404 Not Found status if the resource is not found

//Custom Response Handling

**return** CloudVendorResponseHandler.*responseHandler*("No Cloud Vendor Found For VendorId:"+vendorId,cloudVendor,HttpStatus.***NOT\_FOUND***);

}

// Return the cloud vendor with a 200 OK status

**return** CloudVendorResponseHandler.*responseHandler*("Cloud Vendor Details Found For VendorId:"+vendorId, cloudVendor, HttpStatus.***OK***);

}

{

    "data": {

        "vendorId": "C2",

        "vendorName": "vendor 2",

        "vendorAddress": "Magarpatta City, Pune",

        "vendorPhoneNumber": "8230001908"

    },

    "httpStatus": "OK",

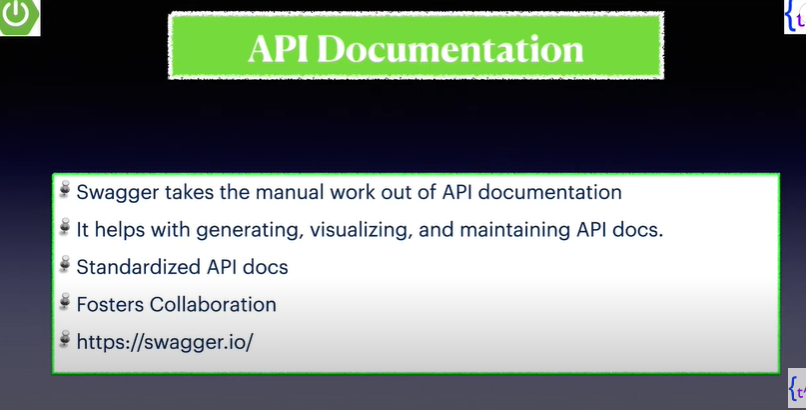
    "message": "Cloud Vendor Details Found For VendorId:C2"

}

**SWAGGER:**

Swagger is a powerful tool used for documenting RESTful APIs. It provides a structured way to describe API endpoints, request parameters, responses, and more.

Swagger supports versioning of APIs, allowing developers to manage different versions of their APIs and document changes over time

****

After generating the document, we can visualize it via swagger UI or POSTMAN.

Whenever an API is upgraded and Swagger is already integrated there, so that will automatically generate the API Document.

**Why API Documentation is so important:**

* Because whatever API is being created, it will accept the request, do some processing and generate the response.
* This Rest APIs works in request-response structure.
* This APIs needs to be consumed by some REST client such as UI Developers, Third-Party Application Devs.
* In order to make them understand what this API is expecting in request, how the request structure should be, what number of parameters it should have or exactly what url it is having, how it is generating url and giving response back to the client.
* To make the client/UI developers understand the request-response structure so that they can consume the response and use it.

**Dependencies Needed to Integrate Swagger In Spring Boot App:**

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-swagger2</artifactId>

<version>3.0.0</version>

</dependency>

The above dependency tag was not working for me with my Spring Version

So, I decided to use the following dependency tag

<dependency>  
 <groupId>org.springdoc</groupId>  
 <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>  
 <version>2.0.2</version>  
</dependency>

Basically using **springdoc-openapi library**

Ref: https://medium.com/@berktorun.dev/swagger-like-a-pro-with-spring-boot-3-and-java-17-49eed0ce1d2f

**Swagger Demonstration-Testing API Documentation through POSTMAN:**

**Hit this to test the api documentation**

<http://localhost:8080/v3/api-docs>

**For Viewing Swagger UI, hit the below Url:**

http://localhost:8080/swagger-ui/index.html#/

**Docket bean:**

@Bean

Docket swaggerConfig() {

**return** **new** Docket(DocumentationType.***SWAGGER\_12***)

.select()

.paths(PathSelectors.*ant*("/cloudvendor/\*"))

.apis(RequestHandlerSelectors.*basePackage*("com.application.CloudVendorApp"))

.build()

.apiInfo(apiCustomData());

}

**private** ApiInfo apiCustomData() {

**return** **new** ApiInfo(

"Cloud Vendor API Application",

"Cloud Vendor Documentation",

"1.0",

"Terms of service URL",

**new** Contact("Pushpan Bhaumik","https://www.google.com", "pushpanbhaumik200@gmail.com"),

"Cloud Vendor License",

"License URL",

Collections.*emptyList*()

);

}

/\*\*

\* Fetching a cloud vendor using vendor Id

\*/

@GetMapping("{vendorId}")

@Operation(summary="Get Cloud Vendor Details By VendorID",

description = "This operation retrieves detailed information about an cloud vendor by its ID.\n"

+ "Provide the vendor ID in the path parameter."

+ "Example: /cloudvendor/{vendorId}")

**public** ResponseEntity<Object> getCloudVendorDetails(@PathVariable String vendorId) { CloudVendor cloudVendor = cloudVendorService.getCloudVendor(vendorId);

**if** (cloudVendor == **null**) {

//Return a 404 Not Found status if the resource is not found

//Custom Response Handling

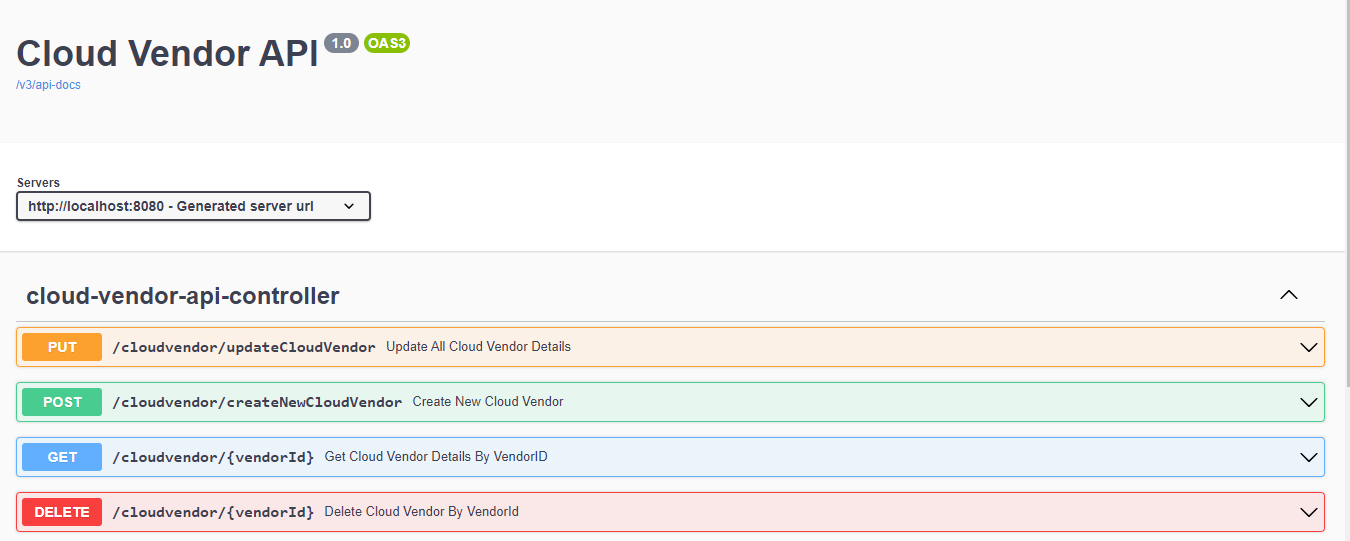
**return** CloudVendorResponseHandler.*responseHandler*("No Cloud Vendor Found For VendorId:"+vendorId, cloudVendor, HttpStatus.***NOT\_FOUND***);

}

// Return the cloud vendor with a 200 OK status

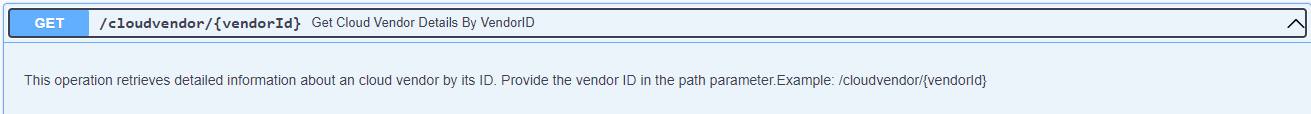
**return** CloudVendorResponseHandler.*responseHandler*("Requested Cloud Vendor Details Found For VendorId:"+vendorId, cloudVendor, HttpStatus.***OK***);

}

****

Using **description** attribute

@Operation(summary="Get Cloud Vendor Details By VendorID

****

**If we are using Open API Specification when Open API acquired Swagger,**  we need to import the following as been done in this project

**import** io.swagger.v3.oas.annotations.Operation;

**public** **class** CloudVendor {

@Id

@Schema(description = "Unique ID" , example="C2")

**private** String vendorId;

**private** String vendorName;

**private** String vendorAddress;

**private** String vendorPhoneNumber;

**public** CloudVendor() {

**super**();

}

**public** CloudVendor(String vendorId, String vendorName, String vendorAddress, String

**super**();

**this**.vendorId = vendorId;

**this**.vendorName = vendorName;

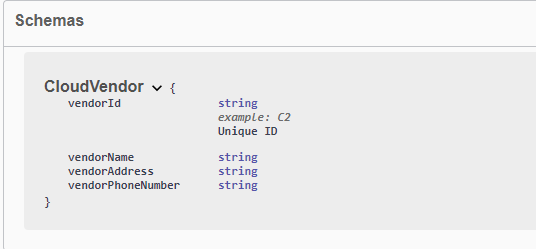
**this**.vendorAddress = vendorAddress;

**this**.vendorPhoneNumber = vendorPhoneNumber;

}

Getter/setters

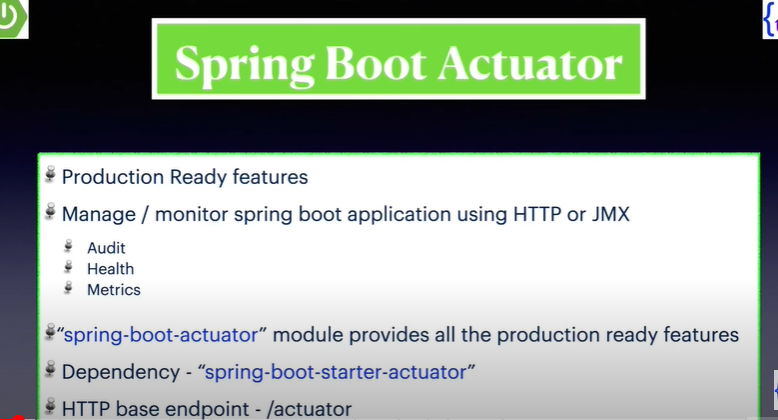
}

****

@Id

@Schema(description = "Unique ID" , example="C2")

**Spring Boot Actuator:**

****

Dependency:

<dependency>

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

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

**In the O/p console, we can observe the following to check endpoint is working:**

Exposing 1 endpoint(s) beneath base path '/actuator'

It means the default actuator services are available to use for my application using the endpoint /actuator

We can hit the below endpoint to check it:

localhost:8080/actuator

{

    "\_links": {

        "self": {

            "href": "http://localhost:8080/actuator",

            "templated": false

        },

        "health": {

            "href": "http://localhost:8080/actuator/health",

            "templated": false

        },

        "health-path": {

            "href": "http://localhost:8080/actuator/health/{\*path}",

            "templated": true

        }

    }

}

Now this comes under the default behaviour:

If we try to check the same with

localhost:8080/beans , localhost:8080/metrics

It won’t give us any data as this endpoint is not exposed under default settings

**How to include all the endpoints of an application:**

management.endpoints.web.exposure.include=\*

**Adding it in application.properties file will be enough**

Exposing 13 endpoint(s) beneath base path '/actuator'

**How to exclude a particular endpoint:**

management.endpoints.web.exposure.exclude=caches

**To check more about the health endpoint:**

management.endpoint.health.show-details=always

**Showing some selected endpoints :**

management.endpoints.web.exposure.include=health,beans,metrics

**Changing the base path of Spring-Boot Actuator:**

management.endpoints.web.base-path=/cloudvendor-actuator

**Exposing 3 endpoint(s) beneath base path '/cloudvendor-actuator'**

**Earlier the url was :**

localhost:8080/actuator

**Now after changing the base path:**

localhost:8080/cloudvendor-actuator

**Checking Health Of Other Services Using Spring Boot Actuators:**

Now, we need to use the address and port no. to check the health of the other applications or service:

**Create a Controller Class:-**

**package** com.application.CloudVendorApp.monitoring;

**import** java.io.IOException;

**import** java.net.InetSocketAddress;

**import** java.net.Socket;

**import** org.slf4j.Logger;

**import** org.slf4j.LoggerFactory;

**import** org.springframework.boot.actuate.health.Health;

**import** org.springframework.boot.actuate.health.HealthIndicator;

**import** org.springframework.core.env.Environment;

**import** org.springframework.stereotype.Controller;

@Controller

**public** **class** OtherCloudAppServiceHealthCheck **implements** HealthIndicator{

**private** **static** **final** Logger ***LOGGER*** = LoggerFactory.*getLogger*(OtherCloudAppServiceHealthCheck.**class**);

**private** **final** Environment env;

/\*\*

\* Constructor Injection

\* **@param** env

\*/

**public** OtherCloudAppServiceHealthCheck(Environment env) {

**super**();

**this**.env = env;

}

@Override

**public** Health health() {

**if**(isServiceUp()) {

**return** Health.*up*().withDetail("Other Cloud App ", "is now working").build();

}**else** {

**return** Health.*down*().withDetail("Other Cloud App ", "is down").build();

}

}

**private** **boolean** isServiceUp() {

String address = env.getProperty("othercloud.app.address");

String port = env.getProperty("othercloud.app.port");

**if**(***LOGGER***.isInfoEnabled()) {

***LOGGER***.info(String.*format*("Address=%s,Port=%s",address,port));

}

**return** isAddressReachable(address,Integer.*parseInt*(port),3000);

}

//Creating connection with the OuterCloudApp Service

**private** **boolean** isAddressReachable(String address, **int** port, **int** timeout) {

Socket isSocket = **new** Socket();

**try** {

//Connect this socket to the server with a specified timeout value

isSocket.connect(**new** InetSocketAddress(address,port),timeout);

**return** **true**;

}**catch**(IOException exception) {

//connection failed so returning false

exception.printStackTrace();

**return** **false**;

}

**finally** {

**try** {

isSocket.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

}

**application.properties:**

othercloud.app.address=127.0.0.1

othercloud.app.port=8081

**Checking the application health:-**

http://localhost:8080/cloudvendor-actuator/health

"otherCloudAppServiceHealthCheck": {

            "status": "DOWN",

            "details": {

                "Other Cloud App ": "is down"

            }

        },

No need of the other application to be configured using **Spring Boot Actuators**

**Testing the Cloud-Vendor Application:**

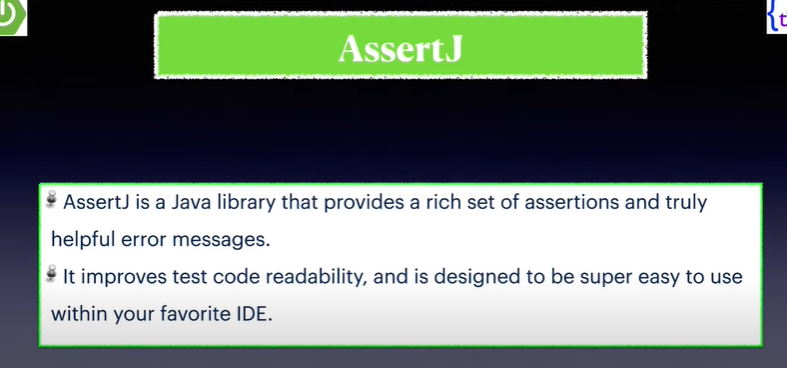
**Junit5:**

**Unit Testing:-** Unit testing is a software testing technique where individual units or components of a software application are tested in isolation to ensure that they function correctly

JUnit 5 provides a rich set of annotations such as **@Test** for defining test methods, **@BeforeEach** and **@AfterEach** for setup and teardown methods executed before and after each test, and **@DisplayName** for providing custom display names to tests.

**AssertJ:**

AssertJ provides a fluent, expressive API for writing assertions in tests

****

**Mockito:-**

Mockito is a popular mocking framework that allows developers to create mock objects for testing

It provides methods like **Mockito.mock()** to create mock objects and **when()** to define mock behaviors.

**Mockito 3.x requires Java 8 & above**

**H2 database:-**

H2 is an in-memory database that allows developers to create lightweight databases for testing purposes.

It provides fast performance as data is stored in memory, making it suitable for unit and integration tests.

* Spring Boot can auto-configure in-memory h2 database
* Very fast and open-source
* While writing unit test cases, we may need to test part of application which connects with real-time db.
* So, for testing purpose, we can’t call the db table everytime we are testing, in such case, we need to use the in-memory db

No need of adding dependencies for **junit,assertj,mockito** as while creating the spring boot application, we added the following dependency

<dependency>

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

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

Spring-Boot-Starter test already contains the dependencies for **junit,assertj,mockito**

**For h2 database: add the below dependency**

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<scope>test</scope>

</dependency>

**@BeforeEach** and **@AfterEach** annotations are used to set up and tear down the **CloudVendor** object before and after each test method runs, respectively.Used for cleanup tasks for testing

We need to configure the in-memory db as follows:-

**src/test/resources:**

**application.properties:**

spring.datasource.url=jdbc:h2://mem:db;DB\_CLOSE\_DELAY=-1

spring.datasource.username=sa

Default username,password for h2 in-memory db

spring.datasource.password=sa

spring.datasource.driver-class-name=org.h2.Driver

spring.jpa.properties.hibernate=org.hibernate.dialect.H2Dialecct

spring.jpa.hibernate.ddl-auto=create-drop

spring.jpa.show-sql=true

Notes:

**@Test** is used to signal that the annotated method is a *test* method.

**@Test** methods must not be private or static and must not return a value

The **@DataJpaTest** annotation in Spring Boot tells Spring to set up an environment for testing JPA repositories and configures a test slice with an embedded database

Repository Test Cases:

**assertThat** from AssertJ is used to test behaviors and verify the expected outcomes in your code.

/\*\*

\* Repository Layer Test Case

\* importing the assertj package for assertion

\* not the junit jupiter package

\*/

@Test

@DisplayName("Testing the repo method for finding vendors by name")

**void** testFindByVendorName() {

List<CloudVendor> cloudVendorList = cloudVendorRepository.findByVendorName("Amazon"); *assertThat*(cloudVendorList.get(0).getVendorAddress()).isEqualTo(

cloudVendor.getVendorAddress()); *assertThat*(cloudVendorList.get(0).getVendorId()).isEqualTo(cloudVendor.getVendorId());

}

/\*\*

\* Repository Layer Test Case

\*/

@Test

@DisplayName("Testing the repo method for not finding any vendor")

**void** testCloudVendorName\_NotFound() {

List<CloudVendor> cloudVendorList = cloudVendorRepository.findByVendorName("GCP");

*assertThat*(cloudVendorList).isEmpty();

}

**Testing The Service Layer:-**

* Avoiding real-time database interaction while testing service layer
* Creating a mock of the CloudVendorRepository
* By using **@Mock**, you instruct Mockito (a popular mocking framework) to create a mock implementation of the CloudVendorRepository

The purpose of **AutoCloseable** is to close all the resources the moment this class finishes it’s execution. So, release all the resources immediately

In Mockito, the when method is used to set up the behavior of mock objects.

When you mock an object, by default,all of its methods return default values (null for objects, 0 for integers, etc.).

**Problem faced while testing DELETE call:**

We need to use when and thenReturn to mock our cloudVendor instance created in the setUp method otherwise it will try to call the main database in the service layer instead of calling the in-memory H2 database.

*when*(cloudVendorRepo.findById("1")).thenReturn(Optional.*ofNullable*(cloudVendor));

@DataJpaTest

**class** CloudVendorServiceImplTest {

/\*\*

\* Avoiding real-time database interaction while testing service layer

\* Creating a mock of the CloudVendorRepository

\* By using **@Mock**, you instruct Mockito (a popular mocking framework) to create a mock implementation of the CloudVendorRepository

\*/

@Mock

**private** CloudVendorRepository cloudVendorRepo;

**private** CloudVendorService cloudVendorService;

AutoCloseable autoCloseable;

CloudVendor cloudVendor;

/\*\*

\* Executes before each test methods

\*/

@BeforeEach

**void** setUp() {

//Used to automatically initialize annotated fields in the test class

autoCloseable = MockitoAnnotations.*openMocks*(**this**);

cloudVendorService = **new** CloudVendorServiceImpl(cloudVendorRepo);

cloudVendor = **new** CloudVendor("1","Amazon","USA","79001");

cloudVendorRepo.save(cloudVendor);

}

/\*\*

\* Executes after each test methods

\* Used for cleanup tasks for testing

\*/

@AfterEach

**void** tearDown() {

**try** {

//Closes this resource, relinquishing any underlying resources

autoCloseable.close();

} **catch** (Exception e) {

e.printStackTrace();

}

cloudVendor = **null**;

cloudVendorRepo.deleteAll();

}

/\*\*

\* Service Layer Test Cases

\*/

/\*\*

\* In Mockito, the when method is used to set up the behavior of mock objects.

\* When you mock an object, by default,

\* all of its methods return default values (null for objects, 0 for integers, etc.).

\*/

@Test

@DisplayName("Testing For Cloud Vendor Creation")

**void** testCreateVendor() {

*mock*(CloudVendor.**class**);

*mock*(CloudVendorRepository.**class**);

*when*(cloudVendorRepo.save(cloudVendor)).thenReturn(cloudVendor);

*assertThat*(cloudVendorService.createVendor(cloudVendor)).isEqualTo("CloudVendor Resource Created Successfully");

}

@Test

@DisplayName("Testing For Update Cloud Vendor Functionaliy")

**void** testUpdateVendor() {

*when*(cloudVendorRepo.save(cloudVendor)).thenReturn(cloudVendor);

*assertThat*(cloudVendorService.updateVendor(cloudVendor)).isEqualTo("CloudVendor Resource Updated Successfully");

}

@Test

@DisplayName("Testing For Delete Cloud Vendor Functionaliy")

**void** testDeleteVendor\_Success() {

*mock*(CloudVendor.**class**);

*mock*(CloudVendorRepository.**class**);

// Given

*when*(cloudVendorRepo.findById("1")).thenReturn(Optional.*ofNullable*(cloudVendor));

// When

String result = cloudVendorService.deleteVendor(cloudVendor.getVendorId());

// Then

// It verifies that the deleteById is called once

*verify*(cloudVendorRepo, *times*(1)).deleteById(cloudVendor.getVendorId());

*assertEquals*("CloudVendor Deleted Successfully", result);

}

@Test

@DisplayName("Testing For Get Cloud Vendor")

**void** testGetCloudVendor() {

*when*(cloudVendorRepo.findById("1")).thenReturn(Optional.*ofNullable*(cloudVendor));

*assertThat*(cloudVendorService.getCloudVendor("1").getVendorName())

.isEqualTo(cloudVendor.getVendorName());

}

/\*\*

\* We are fetching cloud vendors

\* We are using singletonList because

\* When you use Collections.singletonList(cloudVendor),

\* it explicitly creates a List containing only one element, which is of type CloudVendor

\*/

@Test

@DisplayName("Testing GetAllCloudVendor Functionality")

**void** testGetAllCloudVendor() {

*when*(cloudVendorRepo.findAll()).thenReturn(

**new** ArrayList<CloudVendor>(Collections.*singletonList*(cloudVendor))

);

*assertThat*(cloudVendorService.getAllCloudVendor().get(0).getVendorPhoneNumber())

.isEqualTo(cloudVendor.getVendorPhoneNumber());

}

}

**Testing the Controller Layer:-**

**Annotations Used:**

@WebMvcTest(CloudVendorAPIController.**class**)

* **@WebMvcTest** is primarily used for controller layer testing. It focuses on setting up a slice of the application context relevant for Spring MVC testing.
* It does not load the complete application context, such as service layer beans, repository beans, or other non-web-related components.

/\*\*

\* Controller Layer Testing

\*/

@Autowired

**private** MockMvc mockMvc;

@MockBean

**private** CloudVendorService cloudVendorService;

CloudVendor cloudVendorOne;

CloudVendor cloudVendorTwo;

List<CloudVendor> cloudVendorList = **new** ArrayList<>();

/\*\*

\* Executes before each test methods

\*/

@BeforeEach

**void** setUp() {

cloudVendorOne = **new** CloudVendor("1","Amazon",

"USA","9000129221");

cloudVendorTwo = **new** CloudVendor("2","GCP",

"UK","9008129290");

cloudVendorList.add(cloudVendorOne);

cloudVendorList.add(cloudVendorTwo);

}

/\*\*

\* Executes after each test methods

\* Used for cleanup tasks for testing

\*/

@AfterEach

**void** tearDown() {

}

@Test

@DisplayName("Testing Get Cloud Vendor By Vendor Id")

**void** testGetCloudVendorDetails() **throws** Exception {

*when*(cloudVendorService.getCloudVendor("1"))

.thenReturn(cloudVendorOne);

**this**.mockMvc.perform(*get*("/cloudvendor/1"))

.andDo(*print*()).andExpect(*status*().isOk());

}

@Test

@DisplayName("Testing Get All Vendor Details")

**void** testGetAllVendorDetails() **throws** Exception{

*when*(cloudVendorService.getAllCloudVendor())

.thenReturn(cloudVendorList);

**this**.mockMvc.perform(*get*("/cloudvendor/getVendorDetails"))

.andDo(*print*()).andExpect(*status*().isOk());

}

@Test

@DisplayName("Testing Delete Cloud Vendor Details")

**void** testDeleteCloudVendorDetais() **throws** Exception {

*when*(cloudVendorService.deleteVendor("1"))

.thenReturn("CloudVendor Deleted Successfully");

**this**.mockMvc.perform(*delete*("/cloudvendor/1"))

.andDo(*print*()).andExpect(*status*().isOk());

}

@Test

@DisplayName("Testing Create Cloud Vendor Details")

**void** testCreateCloudVendorDetails() **throws** Exception {

ObjectMapper mapper = **new** ObjectMapper();

mapper.configure(SerializationFeature.***WRAP\_ROOT\_VALUE***, **false**);

String requestJson = mapper.writeValueAsString(cloudVendorOne);

*when*(cloudVendorService.createVendor(cloudVendorOne))

.thenReturn("CloudVendor Resource Created Successfully");

**this**.mockMvc.perform(*post*("/cloudvendor/createNewCloudVendor")

.contentType(org.springframework.http.MediaType.***APPLICATION\_JSON***)

.content(requestJson))

.andDo(*print*()).andExpect(*status*().isCreated());

}