**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",

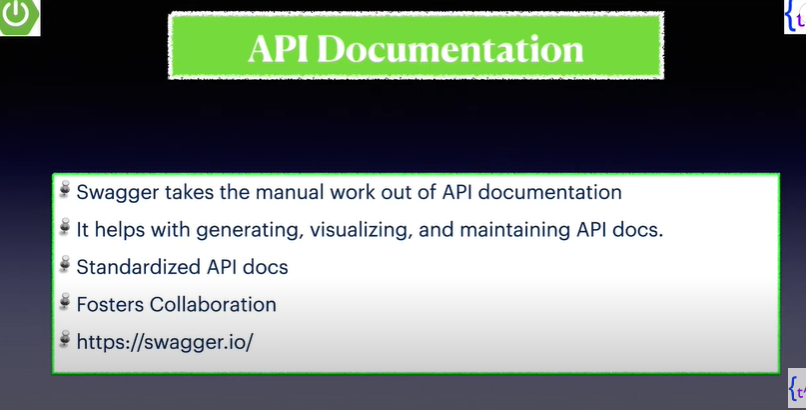
    "message": "Cloud Vendor Details Found For VendorId:C2"

}

**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>

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:**