**Controller Class :** Method 5 (RetrieveNotifyResponseByEventId)

* **Annotations**:

This code uses annotations to define various aspects of the API endpoint:

* **@Operation**: Provides a summary description for the API operation. In this case, it's a description of what this API endpoint does.
* **@ApiResponses**: Specifies the possible responses for this API operation. In this case, it defines a single response with a 200 HTTP status code and a description indicating that "notify" details were retrieved successfully. It also specifies that the response will be in JSON format and will follow the schema defined by the EventResponse class.
* **@GetMapping**: Indicates that this method should handle HTTP GET requests. The URL path for this endpoint is constructed by concatenating several constants defined in the **DocumentGeneratorEventStoreConstants** class, which likely represent parts of the URL path.
* **Method Signature**: This method is defined to take two parameters:
* **HttpServletRequest httpRequest**: An object representing the HTTP request. This is typically used to access information about the incoming request.
* **@NotNull @PathVariable final String eventId**: A path variable named **eventId** that is marked as not nullable (**@NotNull**). This means the **eventId** must be present in the URL path, and it cannot be null.
* **Logging**: This line logs a message using a logger (presumably from a logging framework like Log4j or SLF4J). It logs the message "**Getting Notify data from eventId:** " followed by the sanitized value of the **eventId**. The purpose of this log message is likely for debugging and tracking API usage.
* **Service Method Call**: This line invokes a method named **fetchNotifyDetailsByEventId** on an instance of **documentGeneratorEventStoreService**. This method is expected to return an **Optional<EventResponse>**. It appears to be retrieving "**notify**" details based on the provided **eventId**.
* **Response Mapping**: Finally, this line returns the result of calling the **eventResponseMapper** method, passing in the **httpRequest** and **eventDataResponse** as parameters. This method is responsible for mapping the retrieved data into an appropriate response format and returning it as an instance of **ResponseEntity<EventResponse>**.

**Test Case**: Total 6 test case scenarios for the controller Method 5 :

**1. Success() :** Here’s what the code is doing.

Here, we're setting up the test method. It's annotated with **@Test**, indicating that it's a **JUnit** test case. This method tests the successful retrieval of event notifications by **event ID**.

* **Create a Mock :**

**--** We create an instance of **EventResponse** and initialize a String variable **eventId** with the value "**123**". This **eventId** will be used as input for testing the controller endpoint.

* **Stub the Mock :**

**--** In this section, we're using a mocking framework (likely **Mockito**) to simulate the behavior of a service class named **documentGeneratorEventStoreService**. We're telling the mock to return an Optional containing the **eventResponse** when the **fetchNotifyDetailsByEventId** method is called with the **eventId** we provided earlier.

* **Execute and assert or simulate the MVC :**

**--** This part is testing the behavior of a Spring MVC controller. We're using the **mockMvc** object to simulate an HTTP GET request to the **/event/123/notify** endpoint with the specified **eventId**. We also set the content type to JSON.

**--** Then, we use **andExpect** to assert that the HTTP response status should be **isOk()**, indicating a successful HTTP response with a status code of **200 OK**. Finally, we use **andReturn()** to finish the request and capture the result.

* **Verify :**

- In this last part, we verify that the **fetchNotifyDetailsByEventId** method of the **documentGeneratorEventStoreService** was called exactly once with the expected **eventId**. This step ensures that the controller endpoint correctly interacts with the service class.

**Test Case**: Total 6 test case scenarios for the controller Method 5 :

**2. NotFound() :** Here’s what the code is doing.

We begin by setting up the test method. It's annotated with @Test, indicating that it's a JUnit test case. This method tests the scenario where no notifications are found for a specified event ID.

* **Create a Mock :**

**--** We create a String variable **eventId** and initialize it with the value "**123**". This **eventId** will be used as input for testing the controller endpoint.

* **Stub the Mock :**

**--** In this section, we're using a mocking framework (likely **Mockito**) to simulate the behavior of a service class named **documentGeneratorEventStoreService**. We're telling the mock to return an empty **Optional** when the **fetchNotifyDetailsByEventId** method is called with the **eventId** we provided earlier. This simulates the scenario where no event notifications are found for the given **eventId**.

* **Execute and assert or simulate the MVC :**

**--** This part is testing the behavior of a Spring MVC controller in a scenario where no event notifications are found for the provided **eventId**. We use the **mockMvc** object to simulate an HTTP GET request to the **/event/123/notify** endpoint with the specified **eventId**, setting the content type to JSON.

**--** Then, we use **andExpect** to assert that the HTTP response status should be **isNotFound()**, indicating a **404 Not Found** status code. This verifies that the controller correctly handles the scenario where no event notifications are found.

**--** We also capture the result of this **mockMvc** request in a **MvcResul**t object, which can be useful for additional assertions or analysis in more complex test scenarios.

* **Verify :**

**--** In this last part, we verify that the **fetchNotifyDetailsByEventId** method of the **documentGeneratorEventStoreService** was called exactly once with the expected **eventId**. This step ensures that the controller endpoint correctly interacts with the service class even when no event notifications are found, as expected in this test case.

**Test Case**: Total 6 test case scenarios for the controller Method 5 :

**3. EventIdIsNull() :** Here’s what the code is doing.

We start by setting up the test method. It's annotated with **@Test**, indicating that it's a **JUnit** test case. This method tests the scenario where the **eventId** is null or invalid.

* **Create a Mock :**

**--** We create a String variable **eventId** and initialize it with the value "**null**". This **eventId** represents an invalid input because it's not a proper **event ID**. The purpose of this test is to ensure that the controller handles such invalid inputs correctly.

* **Execute and assert or simulate the MVC :**

**--** In this section, we're testing the behavior of a Spring MVC controller when an **invalid** or **null** **eventId** is provided. We use the **mockMvc** object to simulate an HTTP GET request to the **/event/null/notify** endpoint with the specified **eventId**, setting the content type to JSON.

**--** Then, we use **andExpect** to assert that the HTTP response status should be **isBadRequest()**, indicating a **400 Bad Request** status code. This verifies that the controller correctly handles the scenario where the **eventId** is **null** or **invalid** and responds with a proper error status.

**--** We also capture the result of this **mockMvc** request in a **MvcResult** object, which can be useful for additional assertions or analysis in more complex test scenarios.

* **Verify :**

**--** In this last part, we verify that the **fetchNotifyDetailsByEventId** method of the **documentGeneratorEventStoreService** was not called at all. This step ensures that when the **eventId** is **null** or **invalid**, the controller does not attempt to call the service method, as it's expected behavior.

**Test Case**: Total 6 test case scenarios for the controller Method 5 :

1. **EventIdIsEmpty() :** Here’s what the code is doing.

We begin by setting up the test method. It's annotated with @Test, indicating that it's a JUnit test case. This method tests the scenario where the eventId is an empty string.

* **Create a Mock :**

**--** We create a String variable eventId and initialize it with a single space " ". This eventId represents an empty event ID, which is an invalid input. The purpose of this test is to ensure that the controller handles such invalid inputs correctly.

* **Execute and assert or simulate the MVC :**

**--** In this section, we're testing the behavior of a Spring MVC controller when an empty **eventId** is provided. We use the **mockMvc** object to simulate an HTTP GET request to the **/event//notify** endpoint with the specified **eventId**, setting the content type to JSON.

**--** Then, we use **andExpect** to assert that the HTTP response status should be **isBadRequest()**, indicating a **400 Bad Request** status code. This verifies that the controller correctly handles the scenario where the **eventId** is an empty string and responds with a proper error status.

**--** We also capture the result of this **mockMvc** request in a **MvcResult** object, which can be useful for additional assertions or analysis in more complex test scenarios.

* **Verify :**

**--** In this last part, we verify that the **fetchNotifyDetailsByEventId** method of the **documentGeneratorEventStoreService** was not called at all. This step ensures that when the **eventId** is an empty string, the controller does not attempt to call the service method, as it's expected behavior.

**Test Case**: Total 6 test case scenarios for the controller Method 5 :

1. **InternalServerError() :** Here’s what the code is doing.

We start by setting up the test method. It's annotated with @Test, indicating that it's a JUnit test case. This method tests the scenario where an internal server error occurs during the execution of the controller.

* **Create a Mock :**

**--** We create a String variable eventId and initialize it with the value "**123**". This **eventId** represents a valid **event ID**, and we will use it to test the controller's behavior when an internal server error occurs.

* **Stub the Mock :**

**--** In this section, we're using a mocking framework (likely **Mockito**) to simulate the behavior of a service class named **documentGeneratorEventStoreService**. We're telling the mock to return an empty **Optional** when the **fetchNotifyDetailsByEventId** method is called with the **eventId**. This simulates the scenario where the service encounters an issue and cannot retrieve the event notifications, leading to an internal server error.

* **Execute and assert or simulate the MVC :**

**--** In this part, we're testing the behavior of a Spring MVC controller when the service method encounters an internal server error. We use the **mockMvc** object to simulate an HTTP GET request to the **/event/123/notify** endpoint with the specified **eventId**.

Then, we use **andExpect** to assert the following:

**--** The HTTP response status should be **isInternalServerError()**, indicating a **500** Internal Server Error status code.

**--** The response content type should be JSON, specified as "**application/json**".

**--** The JSON response body should contain an "**error**" field with the value "**Internal Server Error**". This checks that the controller is correctly handling internal server errors and responding with an appropriate error message.

* **Verify :**

- In this last part, we verify that the **fetchNotifyDetailsByEventId** method of the **documentGeneratorEventStoreService** was called exactly once with the expected **eventId**. This step ensures that the controller correctly attempts to call the service method before the internal server error occurs.

**Test Case**: Total 6 test case scenarios for the controller Method 5 :

1. **EmptyEventResponse() :** Here’s what the code is doing.

We start by setting up the test method. It's annotated with @Test, indicating that it's a JUnit test case. This method tests the scenario where the service method returns an empty response when attempting to retrieve event notifications.

* **Create a Mock :**

**--** We create a String variable **eventId** and initialize it with the value "**123**". This **eventId** represents a valid event ID, and we will use it to test the controller's behavior when the service method returns an empty response.

* **Stub the Mock :**

**--** In this section, we're using a mocking framework (likely **Mockito**) to simulate the behavior of a service class named **documentGeneratorEventStoreService**. We're telling the mock to return an empty Optional when the **fetchNotifyDetailsByEventId** method is called with the **eventId**. This simulates the scenario where the service method cannot find any event notifications for the given **eventId** and returns an empty result.

* **Execute and assert or simulate the MVC :**

**--** In this part, we're testing the behavior of a Spring MVC controller when the service method returns an empty response. We use the **mockMvc** object to simulate an HTTP GET request to the **/event/123/notify** endpoint with the specified **eventId**.

Then, we use **andExpect** to assert the following:

**--** The HTTP response status should be **isOk()**, indicating a **200 OK** status code, as the controller successfully responds even when the service returns an empty response.

**--** The response content type should be JSON, specified as **MediaType.APPLICATION\_JSON**. We assume that the JSON response contains an "**eventId**" field with the value of the provided **eventId**, and we check this using jsonPath("$.eventId").value(eventId)".

**--** We also ensure that the "**data**" field does not exist in the response using **jsonPath("$.data").doesNotExist()**, which verifies that no data is returned when the service result is empty.

* **Verify :**

**--** In this last part, we verify that the **fetchNotifyDetailsByEventId** method of the **documentGeneratorEventStoreService** was called exactly once with the expected **eventId**. This step ensures that the controller correctly attempts to call the service method and handle its response, even when the response is empty.