**Service class name:** Document Generator Event Store Service Impl

**Method-1: Save event**

@Override  
@Transactional  
public Optional<EventResponse> saveEvent(final EventRequest eventRequest) throws JsonProcessingException {  
 String correlationId = eventRequest.getCorrelationId();  
 *LOG*.debug("Inside saveDocumentEvent method and correlationID : {}", GenericUtil.*sanitizeValues*(correlationId));  
 EventResponse eventResponse = new EventResponse();  
 try {  
 return eventStoreHelper.getEventDataResponse(eventRequest, eventResponse);  
 } catch (DataIntegrityViolationException ce) {  
 Optional<DmEvent> existEvent = findExistingEvent(eventRequest);  
 if (existEvent.isPresent()) {  
 eventMapper.updateExistingEventResponse(eventRequest, eventResponse);  
 responseMapper.mapEventTOEventResponse(existEvent.get(), eventResponse);  
 return Optional.*of*(eventResponse);  
 }  
 throw ce;  
 }  
}

Here's the documentation for the **DocumentGeneratorEventStoreServiceImpl** service class:

**Class Description**: This is a service class responsible for interacting with the document generator event store. It implements the **DocumentGeneratorEventStoreService** interface, indicating that it provides implementations for the methods defined in that interface.

**Annotations**:

**@Service**: This annotation marks the class as a Spring-managed service or component. It is used to indicate that this class should be automatically discovered and managed by the Spring Framework.

**Fields**:

The class declares several fields, each of which is annotated with **@Autowired**. These fields represent dependencies that are injected by Spring. They include repositories, mappers, and other components used within the class.

**Methods**:

**saveEvent**: This method is annotated with **@Override**, indicating that it overrides a method defined in the **DocumentGeneratorEventStoreService** interface. It saves an event based on the provided **EventRequest**. The method returns an **Optional<EventResponse>** if the event is successfully saved, and it may throw a **JsonProcessingException** if there is an issue with JSON processing.

Parameters:

**eventRequest**: The request containing event data to be saved.

**Return:**

An **Optional** containing an **EventResponse** if the event is successfully saved. The **Optional** may be empty if the save operation fails.

**Exceptions:**

**JsonProcessingException**: Thrown if there is an issue with JSON processing.

Method Logic:

Extracts the **correlationId** from the request.

Creates a new **EventResponse** object.

Calls a helper method (**eventStoreHelper.getEventDataResponse**) to process the event request and populate the **eventResponse**.

Handles a **DataIntegrityViolationException** by checking for an existing event. If an existing event is found, it updates the **eventResponse** and returns it. If no existing event is found, it rethrows the exception.

**Other Methods and Dependencies:** The class likely contains other methods and dependencies that are used in its operation, but these are not provided in the snippet.

This documentation provides an overview of the class, its purpose, its annotations, and the **saveEvent** method's behavior and usage of dependencies. It helps developers understand the role of this service class within the application.

**1. Test Event Creation:**

Create a test case to ensure that the service correctly creates a new event when provided with a valid EventRequest.

Use Mockito to mock dependencies (e.g., repositories) and verify that the service saves the event and returns the expected response.

//Method - 1 Event creation success  
 @Test  
 public void testSaveEvent\_EventCreation\_Success() throws JsonProcessingException {  
 // Create a sample EventRequest  
 EventRequest eventRequest = new EventRequest();  
 eventRequest.setCorrelationId("12345");  
  
 // Create a sample EventResponse  
 EventResponse eventResponse = new EventResponse();  
 eventResponse.setCode(Integer.*valueOf*("SUCCESS"));  
  
 // Mock the behavior of eventStoreHelper to return the eventResponse  
 *when*(eventStoreHelper.getEventDataResponse(eventRequest, eventResponse))  
 .thenReturn(Optional.*of*(eventResponse));  
  
 // Call the saveEvent method  
 Optional<EventResponse> result = DocumentGeneratorEventStoreService.*saveEvent*(eventRequest);  
  
 // Verify that the service correctly saves the event and returns the expected response  
 *assertTrue*(result.isPresent());  
 *assertEquals*("SUCCESS", result.get().getCode());  
  
 // Verify that eventStoreHelper.getEventDataResponse was called with the expected arguments  
 *verify*(eventStoreHelper, *times*(1)).getEventDataResponse(eventRequest, eventResponse);  
 }

**In this test case:**

We use JUnit and Mockito annotations to set up the test environment.

We create a sample **EventRequest** and a sample **EventResponse**.

We mock the behavior of the **eventStoreHelper** to return the sample **EventResponse** when called with the provided **EventRequest**.

We call the **saveEvent** method of the **eventStoreService** and capture its result.

We use assertions to verify that the result is as expected.

We verify that the **eventStoreHelper.getEventDataResponse** method was called with the expected arguments.

This test case covers the scenario where event creation is successful, and the service returns the expected response. You can add more test cases to cover other scenarios, such as handling exceptions or data integrity violations.

Top of Form

2. Test Event Update:

Create a test case to verify that the service updates an existing event when provided with a valid **EventRequest**.

Mock the dependencies and ensure that the service correctly updates the event and returns the updated response.

@Test  
public void testSaveEvent\_EventUpdate\_Success() throws JsonProcessingException {  
 // Create a sample EventRequest  
 EventRequest eventRequest = new EventRequest();  
 eventRequest.setCorrelationId("12345");  
  
 // Create a sample EventResponse  
 EventResponse eventResponse = new EventResponse();  
 eventResponse.setCode(Integer.*valueOf*("SUCCESS"));  
  
 // Mock the behavior of eventStoreHelper to return the eventResponse  
 *when*(eventStoreHelper.getEventDataResponse(eventRequest, eventResponse))  
 .thenReturn(Optional.*of*(eventResponse));  
  
 // Mock the behavior of findExistingEvent to return an existing event  
 *when*(dmEventRepository.findById(Long.*valueOf*(eventRequest.getCorrelationId())))  
 .thenReturn(Optional.*of*(new DmEvent()));  
  
 // Call the saveEvent method  
 Optional<EventResponse> result = DocumentGeneratorEventStoreService.*saveEvent*(eventRequest);  
  
 // Verify that the service correctly updates the event and returns the expected response  
 *assertTrue*(result.isPresent());  
 *assertEquals*("SUCCESS", result.get().getCode());  
  
 // Verify that eventStoreHelper.getEventDataResponse was called with the expected arguments  
 *verify*(eventStoreHelper, *times*(1)).getEventDataResponse(eventRequest, eventResponse);  
  
 // Verify that dmEventRepository.findByCorrelationId was called with the expected correlationId  
 *verify*(dmEventRepository, *times*(1)).findById(Long.*valueOf*(eventRequest.getCorrelationId()));  
}

In this test case:

We use JUnit and Mockito annotations to set up the test environment.

We create a sample **EventRequest** and a sample **EventResponse**.

We mock the behavior of the **eventStoreHelper** to return the sample **EventResponse** when called with the provided **EventRequest**.

We mock the behavior of **dmEventRepository.findByCorrelationId** to return an existing event when called with the provided correlationId.

We call the **saveEvent** method of the **eventStoreService** and capture its result.

We use assertions to verify that the result is as expected.

We verify that the **eventStoreHelper.getEventDataResponse** method was called with the expected arguments.

We verify that **dmEventRepository.findByCorrelationId** was called with the expected correlationId.

This test case covers the scenario where event update is successful, and the service returns the expected response. You can add more test cases to cover other scenarios, such as handling exceptions or scenarios where no existing event is found.

Top of Form

**3. Test Data Integrity Violation Handling:**

Simulate a data integrity violation (e.g., a unique constraint violation) during event creation.

Verify that the service handles the exception by checking for an existing event and returning an updated EventResponse.

Use Mockito to mock the dependencies and simulate the exception.

//Method -1 Testcase-5 testDataIntegrityViolationHandling  
@Test  
public void testDataIntegrityViolationHandling() throws JsonProcessingException {  
 // Create a sample EventRequest  
 EventRequest eventRequest = new EventRequest();  
 eventRequest.setCorrelationId("12345");  
  
 // Mock the behavior of eventStoreHelper to throw a DataIntegrityViolationException  
 *when*(eventStoreHelper.getEventDataResponse(eventRequest, new EventResponse()))  
 .thenThrow(new DataIntegrityViolationException("Unique constraint violation"));  
  
 // Mock the behavior of findExistingEvent to return an existing event  
 *when*(dmEventRepository.findById(Long.*valueOf*(eventRequest.getCorrelationId())))  
 .thenReturn(Optional.*of*(new DmEvent()));  
  
 // Call the saveEvent method  
 Optional<EventResponse> result = DocumentGeneratorEventStoreService.*saveEvent*(eventRequest);  
  
 // Verify that the service correctly handles the DataIntegrityViolationException  
 *assertTrue*(result.isPresent());  
 *assertEquals*("SUCCESS", result.get().getCode()); // Ensure the event was updated  
  
 // Verify that eventStoreHelper.getEventDataResponse was called with the expected arguments  
 *verify*(eventStoreHelper, *times*(1)).getEventDataResponse(eventRequest, new EventResponse());  
  
 // Verify that dmEventRepository.findByCorrelationId was called with the expected correlationId  
 *verify*(dmEventRepository, *times*(1)).findById(Long.*valueOf*(eventRequest.getCorrelationId()));  
}

In this test case:

We use JUnit and Mockito annotations to set up the test environment.

We create a sample **EventRequest**.

We mock the behavior of the **eventStoreHelper** to throw a **DataIntegrityViolationException** when called with the provided **EventRequest**, simulating a data integrity violation.

We mock the behavior of **dmEventRepository.findByCorrelationId** to return an existing event when called with the provided correlationId.

We call the **saveEvent** method of the **eventStoreService** and capture its result.

We use assertions to verify that the result is as expected.

We verify that the **eventStoreHelper.getEventDataResponse** method was called with the expected arguments.

We verify that **dmEventRepository.findByCorrelationId** was called with the expected correlationId.

This test case covers the scenario where a data integrity violation occurs during event creation, and the service correctly handles the exception by updating an existing event and returning an updated **EventResponse**. You can add more test cases to cover other scenarios, such as handling exceptions when no existing event is found.

Top of Form

**4. Test Invalid JSON Request Handling:**

Create a test case to confirm that the service throws a JsonProcessingException when provided with an invalid JSON request in the EventRequest.

Ensure that the exception is caught and handled appropriately.

//Method -1 testcase 6 testInvalidJsonRequestHandling  
@Test  
public void testInvalidJsonRequestHandling() throws JsonProcessingException {  
 // Create a sample invalid JSON request that cannot be deserialized  
 EventRequest invalidEventRequest = new EventRequest();  
 String invalididField="1234";  
 invalidEventRequest.setCorrelationId("12345");  
 invalidEventRequest.setApplicationLabel("SomeEventType"); // This field does not exist in EventRequest class  
  
 // Mock the behavior of eventStoreHelper to throw a JsonProcessingException  
 *when*(eventStoreHelper.getEventDataResponse(*eq*(invalidEventRequest), *any*()))  
 .thenThrow(JsonProcessingException.class);  
  
 // Call the saveEvent method with the invalid request  
 *assertThrows*(JsonProcessingException.class, () ->DocumentGeneratorEventStoreService .*saveEvent*(invalidEventRequest));  
  
 // Verify that eventStoreHelper.getEventDataResponse was called with the invalid request  
 *verify*(eventStoreHelper, *times*(1)).getEventDataResponse(*eq*(invalidEventRequest), *any*());  
}

In this test case:

We use JUnit and Mockito annotations to set up the test environment.

We create a sample invalid **EventRequest** by including a field (**invalidField**) that does not exist in the **EventRequest** class to ensure that JSON deserialization fails.

We mock the behavior of the **eventStoreHelper** to throw a **JsonProcessingException** when called with the invalid **EventRequest**.

We call the **saveEvent** method of the **eventStoreService** with the invalid request and use **assertThrows** to verify that a **JsonProcessingException** is thrown.

We verify that the **eventStoreHelper.getEventDataResponse** method was called with the invalid request.

This test case covers the scenario where an invalid JSON request is provided to the service, and the service correctly throws a **JsonProcessingException**. You can add more test cases to cover other scenarios, such as handling other exceptions or valid JSON requests.

**5. Test Transaction Rollback:**

Trigger an exception (other than data integrity violation) within the service method and check if the transaction is rolled back correctly.

Use Mockito to mock dependencies and simulate the exception.

Verify that no data is saved to the database in the presence of the exception.

//Method -1 testTransactionRollback  
 @Test  
 public void testTransactionRollback() throws JsonProcessingException {  
 // Create a sample EventRequest  
 EventRequest eventRequest = new EventRequest();  
 eventRequest.setCorrelationId("12345");  
  
 // Mock the behavior of eventStoreHelper to throw an exception other than DataIntegrityViolationException  
 *when*(eventStoreHelper.getEventDataResponse(eventRequest, new EventResponse()))  
 .thenThrow(RuntimeException.class);  
  
 // Call the saveEvent method  
 *assertThrows*(RuntimeException.class, () ->DocumentGeneratorEventStoreService .*saveEvent*(eventRequest));  
  
 // Verify that eventStoreHelper.getEventDataResponse was called with the expected arguments  
 *verify*(eventStoreHelper, *times*(1)).getEventDataResponse(eventRequest, new EventResponse());  
  
 // Verify that dmEventRepository.save (or any database save operation) was NOT called  
 *verify*(dmEventRepository, *never*()).save(*any*());  
 }

In this test case:

We use JUnit and Mockito annotations to set up the test environment.

We create a sample **EventRequest**.

We mock the behavior of the **eventStoreHelper** to throw a **RuntimeException** (or any other exception other than **DataIntegrityViolationException**) when called with the provided **EventRequest**, simulating an exception.

We call the **saveEvent** method of the **eventStoreService** and use **assertThrows** to verify that the expected exception is thrown.

We verify that the **eventStoreHelper.getEventDataResponse** method was called with the expected arguments.

We verify that any database save operation (e.g., **dmEventRepository.save**) was NOT called, ensuring that the transaction was rolled back correctly.

This test case covers the scenario where an exception (other than data integrity violation) occurs within the service method, and the service correctly rolls back the transaction, ensuring that no data is saved to the database in the presence of the exception. You can add more test cases to cover other scenarios, such as handling other types of exceptions or successful execution.

**6. Test Dependency Interaction**:

Mock the dependencies (e.g., repositories, mappers) used in the service class.

Create test cases to verify that the service correctly interacts with these dependencies, including making the expected method calls.

Use Mockito to verify interactions and stub method responses.

//Method -1 testcases 6 testDependencyInteraction  
@Test  
public void testDependencyInteraction() throws JsonProcessingException {  
 // Create a sample EventRequest  
 EventRequest eventRequest = new EventRequest();  
 eventRequest.setCorrelationId("12345");  
  
 // Create a sample EventResponse  
 EventResponse eventResponse = new EventResponse();  
 eventResponse.setCode(Integer.*valueOf*("SUCCESS"));  
  
 // Mock the behavior of eventStoreHelper to return the eventResponse  
 *when*(eventStoreHelper.getEventDataResponse(eventRequest, eventResponse))  
 .thenReturn(Optional.*of*(eventResponse));  
  
 // Mock the behavior of dmEventRepository to return an existing event  
 *when*(dmEventRepository.findById(Long.*valueOf*(eventRequest.getCorrelationId())))  
 .thenReturn(Optional.*of*(new DmEvent()));  
  
 // Mock the behavior of eventMapper and responseMapper  
  
 // Call the saveEvent method  
 Optional<EventResponse> result = DocumentGeneratorEventStoreService.*saveEvent*(eventRequest);  
  
 // Verify that the service correctly interacts with its dependencies  
 *assertTrue*(result.isPresent());  
 *assertEquals*("SUCCESS", result.get().getCode());  
  
 // Verify interactions with dependencies  
 *verify*(eventStoreHelper, *times*(1)).getEventDataResponse(eventRequest, eventResponse);  
 *verify*(dmEventRepository, *times*(1)).findById(Long.*valueOf*(eventRequest.getCorrelationId()));  
 *verify*(eventMapper, *times*(1)).updateExistingEventResponse(eventRequest, eventResponse);  
 *verify*(responseMapper, *times*(1)).mapEventTOEventResponse(*any*(), *eq*(eventResponse));  
}

In this test case:

We use JUnit and Mockito annotations to set up the test environment.

We create a sample **EventRequest** and a sample **EventResponse**.

We mock the behavior of the **eventStoreHelper** to return the sample **EventResponse** when called with the provided **EventRequest**.

We mock the behavior of **dmEventRepository** to return an existing event when called with the provided correlationId.

We mock the behavior of **eventMapper** and **responseMapper**.

We call the **saveEvent** method of the **eventStoreService** and capture its result.

We use assertions to verify that the result is as expected.

We verify interactions with dependencies using Mockito's **verify** methods, ensuring that the service correctly interacts with its dependencies, makes the expected method calls, and stubs method responses.

This test case covers the scenario of verifying dependency interactions within the service method. You can add more test cases to cover other dependency interactions and scenarios as needed.

Top of Form

**7. Test Edge Cases:**

Create test cases for edge cases, such as providing minimal or empty EventRequest objects.

Verify that the service gracefully handles these cases and returns appropriate responses.

//Method -1 testcase -7  
@Test  
public void testSaveEventWithEmptyEventRequest() throws JsonProcessingException {  
 // Arrange  
 *when*(eventStoreHelper.getEventDataResponse(*any*(EventRequest.class), *any*(EventResponse.class)))  
 .thenReturn(Optional.*of*(new EventResponse())); // Simulate a successful response  
  
 // Act  
 Optional<EventResponse> result = DocumentGeneratorEventStoreService.*saveEvent*(new EventRequest()); // Pass an empty EventRequest  
  
 // Assert  
 *assertTrue*(result.isPresent());  
 // Add more assertions to verify the content of the response  
 *verify*(eventStoreHelper, *times*(1)).getEventDataResponse(*any*(EventRequest.class), *any*(EventResponse.class));  
}

//Method -1 testcase - 8  
 @Test  
 public void testSaveEventWithNullEventRequest() throws JsonProcessingException {  
 // Arrange  
 *when*(eventStoreHelper.getEventDataResponse(*any*(EventRequest.class), *any*(EventResponse.class)))  
 .thenReturn(Optional.*of*(new EventResponse())); // Simulate a successful response  
  
 // Act  
 Optional<EventResponse> result = DocumentGeneratorEventStoreService.*saveEvent*(null); // Pass a null EventRequest  
  
 // Assert  
 *assertTrue*(result.isPresent());  
 // Add more assertions to verify the content of the response  
 *verify*(eventStoreHelper, *times*(0)).getEventDataResponse(*any*(EventRequest.class), *any*(EventResponse.class));  
 }

**Description:**

These two test cases are used to verify the behavior of the saveEvent method within the DocumentGeneratorEventStoreService class when it receives different input scenarios.

**Test Case 1 (Method -1 testcase - 7):**

This test checks how the saveEvent method handles an empty EventRequest.

It uses mocking to simulate a successful response from the eventStoreHelper.getEventDataResponse method.

The saveEvent method is called with an empty EventRequest, and the result is asserted to be non-empty (present).

Additional assertions can be added to verify the content of the response.

It verifies that the eventStoreHelper.getEventDataResponse method was called exactly once.

**Test Case 2 (Method -1 testcase - 8):**

This test checks how the saveEvent method handles a null EventRequest.

It also uses mocking to simulate a successful response from the eventStoreHelper.getEventDataResponse method.

The saveEvent method is called with a null EventRequest, and the result is asserted to be non-empty (present).

Additional assertions can be added to verify the content of the response.

It verifies that the eventStoreHelper.getEventDataResponse method was not called because the EventRequest is null.

These test cases are essential for ensuring that the saveEvent method behaves correctly under different input conditions and interacts properly with the eventStoreHelper component. They help maintain the reliability and functionality of the DocumentGeneratorEventStoreService.

**9. Test Exception Handling:**

Write test cases to verify how the service handles various exceptions, both expected and unexpected.

Ensure that exceptions are logged appropriately, and the service provides meaningful error responses.

@Test  
public void testSaveEventWithDataIntegrityViolationException() throws JsonProcessingException {  
 // Arrange  
 EventRequest eventRequest = new EventRequest();  
 String correlationId = "correlationId";  
 eventRequest.setCorrelationId(correlationId);  
 EventResponse eventResponse = new EventResponse();  
  
 *when*(eventStoreHelper.getEventDataResponse(eventRequest, eventResponse))  
 .thenThrow(DataIntegrityViolationException.class);  
  
 // Act and Assert  
 try {  
 DocumentGeneratorEventStoreService.*saveEvent*(eventRequest);  
 *fail*("Expected DataIntegrityViolationException was not thrown.");  
 } catch (DataIntegrityViolationException ce) {  
 // Verify that the expected exception was thrown.  
 *verify*(eventStoreHelper, *times*(1)).getEventDataResponse(eventRequest, eventResponse);  
 *verify*(eventMapper, *never*()).updateExistingEventResponse(*any*(), *any*());  
 *verify*(responseMapper, *never*()).mapEventTOEventResponse(*any*(), *any*());  
 }

The **@Test** method **testSaveEventWithDataIntegrityViolationException** is a JUnit test case written to verify how the **DocumentGeneratorEventStoreService** class handles a **DataIntegrityViolationException** when it occurs during the execution of the **saveEvent** method. Let me break down the code and explain it step by step:

**Test Setup (Arrange)**:

An instance of **EventRequest** is created, and a correlation ID is set on it.

An instance of **EventResponse** is also created.

Using Mockito, the behavior of the **eventStoreHelper.getEventDataResponse** method is stubbed to throw a **DataIntegrityViolationException** when called with the provided **eventRequest** and **eventResponse**.

**Act and Assert**:

The test proceeds to call the **DocumentGeneratorEventStoreService.saveEvent(eventRequest)** method within a try-catch block.

Inside the catch block, the test checks whether a **DataIntegrityViolationException** is thrown as expected.

**Verification (Assert)**:

If a **DataIntegrityViolationException** is indeed thrown, it indicates that the service handled the exception as expected. The test then verifies the following:

**eventStoreHelper.getEventDataResponse** was called exactly once with the provided **eventRequest** and **eventResponse**.

**eventMapper.updateExistingEventResponse** and **responseMapper.mapEventTOEventResponse** were never called. This verifies that when a **DataIntegrityViolationException** occurs, the service does not execute these methods.

The purpose of this test case is to ensure that when a **DataIntegrityViolationException** is thrown, the **DocumentGeneratorEventStoreService** class appropriately handles the exception by not executing certain methods and that the exception is propagated as expected.

If the exception handling logic is correctly implemented in your **DocumentGeneratorEventStoreService** class, this test case will help you catch any regressions in that behavior during future code changes.

Top of Form

**10. Test Boundary Values:**

Create test cases with boundary values for input parameters, such as very large or very small input data, to validate the service's behavior in extreme cases.

//Method-1 Testcase:10  
 @Test  
 public void testSaveEventWithLargeCorrelationId() throws JsonProcessingException {  
 // Arrange  
 EventRequest eventRequest = new EventRequest();  
 String largeCorrelationId = generateLargeString(); // Generate a large correlation ID  
 eventRequest.setCorrelationId(largeCorrelationId);  
 EventResponse eventResponse = new EventResponse();  
  
 // Stub the behavior of eventStoreHelper  
 *when*(eventStoreHelper.getEventDataResponse(eventRequest, eventResponse))  
 .thenReturn(Optional.*of*(eventResponse));  
  
 // Act  
 Optional<EventResponse> result = DocumentGeneratorEventStoreService.*saveEvent*(eventRequest);  
  
 // Assert  
 *verify*(eventStoreHelper, *times*(1)).getEventDataResponse(eventRequest, eventResponse);  
 *assertTrue*(result.isPresent());  
 }  
//Method-1 Testcase: 11  
 @Test  
 public void testSaveEventWithSmallCorrelationId() throws JsonProcessingException {  
 // Arrange  
 EventRequest eventRequest = new EventRequest();  
 String smallCorrelationId = "123"; // A small correlation ID  
 eventRequest.setCorrelationId(smallCorrelationId);  
 EventResponse eventResponse = new EventResponse();  
  
 // Stub the behavior of eventStoreHelper  
 *when*(eventStoreHelper.getEventDataResponse(eventRequest, eventResponse))  
 .thenReturn(Optional.*of*(eventResponse));  
  
 // Act  
 Optional<EventResponse> result = DocumentGeneratorEventStoreService.*saveEvent*(eventRequest);  
  
 // Assert  
 *verify*(eventStoreHelper, *times*(1)).getEventDataResponse(eventRequest, eventResponse);  
 *assertTrue*(result.isPresent());  
 }  
  
 // Helper method to generate a large string  
 private String generateLargeString() {  
 StringBuilder sb = new StringBuilder();  
 for (int i = 0; i < 1000; i++) {  
 sb.append("A"); // Append a large number of characters  
 }  
 return sb.toString();  
 }

We have two test methods, testSaveEventWithLargeCorrelationId and testSaveEventWithSmallCorrelationId, which test boundary cases for the correlationId parameter.

testSaveEventWithLargeCorrelationId sets a very large correlationId, simulating an extreme input boundary.

testSaveEventWithSmallCorrelationId sets a very small correlationId, simulating another extreme input boundary.

We stub the behavior of eventStoreHelper to return an Optional to simulate a successful response.

We use the verify method to ensure that the eventStoreHelper method was called as expected.

We assert that the result is present in both cases, indicating a successful response.

You can create additional boundary test cases for other input parameters, testing extreme values to validate the service's behavior in different scenarios.

**11. Test Negative Scenarios:**

Provide incorrect input data to test the service's rejection of invalid or malicious requests.

@Test  
public void testSaveEventWithInvalidInput() throws JsonProcessingException {  
 // Arrange  
 EventRequest eventRequest = new EventRequest();  
 eventRequest.setCorrelationId(null); // Provide invalid input, e.g., a null correlationId  
 EventResponse eventResponse = new EventResponse();  
  
 // Stub the behavior of eventStoreHelper to throw DataIntegrityViolationException  
 *when*(eventStoreHelper.getEventDataResponse(eventRequest, eventResponse))  
 .thenThrow(DataIntegrityViolationException.class);  
  
 // Act and Assert  
 *assertThrows*(DataIntegrityViolationException.class, () -> {  
 DocumentGeneratorEventStoreService.*saveEvent*(eventRequest);  
 });  
  
 // Verify that eventStoreHelper was called  
 *verify*(eventStoreHelper, *times*(1)).getEventDataResponse(eventRequest, eventResponse);  
  
 // Verify that eventMapper and responseMapper were never called  
 *verify*(eventMapper, *never*()).updateExistingEventResponse(*any*(), *any*());  
 *verify*(responseMapper, *never*()).mapEventTOEventResponse(*any*(DmEvent.class), *eq*(eventResponse));  
}

//Method-1 Testcase: 13  
 @Test  
 public void testSaveEventWithMaliciousInput() throws JsonProcessingException {  
 // Arrange  
 EventRequest eventRequest = new EventRequest();  
 eventRequest.setCorrelationId("'; DROP TABLE events; --"); // Provide malicious input  
 EventResponse eventResponse = new EventResponse();  
  
 // Stub the behavior of eventStoreHelper to throw DataIntegrityViolationException  
 *when*(eventStoreHelper.getEventDataResponse(eventRequest, eventResponse))  
 .thenThrow(DataIntegrityViolationException.class);  
  
 // Act and Assert  
 *assertThrows*(DataIntegrityViolationException.class, () -> {  
 DocumentGeneratorEventStoreService.*saveEvent*(eventRequest);  
 });  
  
 // Verify that eventStoreHelper was called  
 *verify*(eventStoreHelper, *times*(1)).getEventDataResponse(eventRequest, eventResponse);  
  
 // Verify that eventMapper and responseMapper were never called  
 *verify*(eventMapper, *never*()).updateExistingEventResponse(*any*(), *any*());  
 *verify*(responseMapper, *never*()).mapEventTOEventResponse(*any*(DmEvent.class), *eq*(eventResponse));  
 }

We have two test methods, **testSaveEventWithInvalidInput** and **testSaveEventWithMaliciousInput**, to simulate negative scenarios.

In **testSaveEventWithInvalidInput**, we provide an invalid input by setting **correlationId** to **null**.

In **testSaveEventWithMaliciousInput**, we provide a malicious input as the **correlationId**.

We stub the behavior of **eventStoreHelper** to throw a **DataIntegrityViolationException** when called.

We use **assertThrows** to verify that the service throws a **DataIntegrityViolationException** in response to invalid or malicious input.

We verify that **eventStoreHelper** was called and that **eventMapper** and **responseMapper** were never called, indicating that the service rejected the input without further processing.

These test cases help ensure that the service properly handles negative scenarios and rejects invalid or malicious input as expected.

**13. Test Integration:**

Perform integration tests to ensure that the service functions correctly as part of an integrated system.

@Test  
public void testIntegrationWithValidInput() throws JsonProcessingException {  
 // Arrange  
 EventRequest eventRequest = new EventRequest();  
 eventRequest.setCorrelationId("validCorrelationId");  
 EventResponse eventResponse = new EventResponse();  
  
 // Stub the behavior of eventStoreHelper  
 *when*(eventStoreHelper.getEventDataResponse(eventRequest, eventResponse))  
 .thenReturn(Optional.*of*(eventResponse));  
  
 // Act  
 Optional<EventResponse> result = DocumentGeneratorEventStoreService.*saveEvent*(eventRequest);  
  
 // Assert  
 *verify*(eventStoreHelper, *times*(1)).getEventDataResponse(eventRequest, eventResponse);  
 *assertTrue*(result.isPresent());  
}

We use the **@SpringBootTest** annotation to create a Spring application context for integration testing.

We use **@Autowired** to inject the real **DocumentGeneratorEventStoreService** bean.

We use **@MockBean** for the **EventStoreHelper**, **EventMapper**, and **ResponseMapper** to replace them with mock instances while keeping the rest of the application context intact.

Inside the test method, we provide valid input, stub the behavior of the **eventStoreHelper**, and perform the actual call to **eventStoreService.saveEvent**.

We use Mockito to verify that the mocked **eventStoreHelper** was called as expected.

You can add more integration test cases to cover various scenarios, including edge cases, invalid inputs, and any interactions with other components. Integration tests ensure that your service functions correctly when integrated into a larger system.

**14. Test Timeout Handling:**

Simulate long-running operations or timeouts and verify that the service handles them appropriately, possibly using timeouts in your test framework.

@Test  
public void testTimeoutHandling() throws JsonProcessingException {  
 // Arrange  
 EventRequest eventRequest = new EventRequest();  
 eventRequest.setCorrelationId("validCorrelationId");  
 EventResponse eventResponse = new EventResponse();  
  
 // Stub the behavior of eventStoreHelper to simulate a long-running operation  
 *when*(eventStoreHelper.getEventDataResponse(eventRequest, eventResponse))  
 .thenAnswer(invocation -> {  
 Thread.*sleep*(5000); // Simulate a 5-second delay  
 return Optional.*of*(eventResponse);  
 });  
  
 // Act and Assert  
 *assertTimeout*(Duration.*ofSeconds*(3), () -> {  
 Optional<EventResponse> result = DocumentGeneratorEventStoreService.*saveEvent*(eventRequest);  
 *assertFalse*(result.isPresent()); // Ensure that the result is not present due to timeout  
 });  
  
 // Verify that eventStoreHelper was called  
 *verify*(eventStoreHelper, *times*(1)).getEventDataResponse(eventRequest, eventResponse);  
  
 // Verify that eventMapper and responseMapper were never called  
 *verify*(eventMapper, *never*()).updateExistingEventResponse(*any*(), *any*());  
 *verify*(responseMapper, *never*()).mapEventTOEventResponse(*any*(DmEvent.class), *eq*(eventResponse));  
}

We use the **assertTimeout** method to set a timeout of 3 seconds for the test.

Inside the test, we stub the behavior of **eventStoreHelper** to simulate a long-running operation using **Thread.sleep(5000)** (simulating a 5-second delay).

We then call **eventStoreService.saveEvent** within the **assertTimeout** block and expect the operation to time out within 3 seconds.

We verify that the result is not present after the timeout, indicating that the service handled the timeout appropriately.

We also verify that **eventStoreHelper** was called as expected and that **eventMapper** and **responseMapper** were never called because the operation timed out.

This test simulates a timeout scenario and ensures that your service handles timeouts correctly. You can adjust the timeout duration and test parameters as needed to match your specific use case.

**Method-2: Save event Status**

@Transactional  
public Optional<EventResponse> saveEventStatus(final String eventId,  
 final CreateEventStatusRequest createEventStatusRequest) {  
 *LOG*.debug("Inside saveEventStatus method and eventId : {}", GenericUtil.*sanitizeValues*(eventId));  
 EventResponse eventResponse = new EventResponse();  
 Long eventID = Long.*valueOf*(eventId);  
  
 DmEventStatus dmEventStatus = new DmEventStatus();  
 String eventStatus = createEventStatusRequest.getEventStatusRequest().getEventStatus().getStatus();  
 dmEventStatus.setEventId(eventID);  
 dmEventStatus.setStatus(eventStatus);  
 dmEventStatus = dmEventStatusRepository.save(dmEventStatus);  
 setEventStatus(dmEventStatus, eventResponse);  
 eventResponse.getEventDataResponse().setEventId(eventId);  
  
 EventStatusRequest.EventStatusDetail eventStatusDetail  
 = createEventStatusRequest.getEventStatusRequest().getEventStatus();  
  
 if (Optional.*ofNullable*(eventStatusDetail.getEventErrorRequest()).isPresent()) {  
 saveEventError(eventStatusDetail.getEventErrorRequest(), eventID, dmEventStatus);  
 }  
  
 if (Optional.*ofNullable*(eventStatusDetail.getDocumentRequest()).isPresent()) {  
 saveDocumentDetails(eventStatusDetail.getDocumentRequest(), eventID, eventResponse);  
 } else if (Optional.*ofNullable*(eventStatusDetail.getEventNotifyDataRequest()).isPresent()) {  
 saveEventNotify(eventStatusDetail.getEventNotifyDataRequest(), eventID);  
 }  
  
 eventMapper.updateEventRequestToEventResponse(createEventStatusRequest, eventResponse);  
 return Optional.*of*(eventResponse);  
}

Here's a breakdown of what the code does:

1. **LOG.debug(...)**: This is a logging statement using the **LOG** object, presumably for debugging purposes. It logs information related to the method and the **eventId**.
2. **EventResponse eventResponse = new EventResponse();**: Initializes an **EventResponse** object, which appears to be a custom class for representing the response of this method.
3. **Long eventID = Long.valueOf(eventId);**: Converts the **eventId** (which is likely a string) into a **Long** to work with it as a numeric value.
4. **DmEventStatus dmEventStatus = new DmEventStatus();**: Creates a new instance of **DmEventStatus**, which seems to be an entity or model class representing event status information.
5. **String eventStatus = createEventStatusRequest.getEventStatusRequest().getEventStatus().getStatus();**: Retrieves the event status from the **createEventStatusRequest** object.
6. **dmEventStatus.setEventId(eventID);**: Sets the **eventId** in the **dmEventStatus** object.
7. **dmEventStatus.setStatus(eventStatus);**: Sets the **eventStatus** in the **dmEventStatus** object.
8. **dmEventStatus = dmEventStatusRepository.save(dmEventStatus);**: Persists the **dmEventStatus** object into a database using a repository (likely a Spring Data JPA repository).
9. **setEventStatus(dmEventStatus, eventResponse);**: Calls a method **setEventStatus** to populate the **eventResponse** with information from **dmEventStatus**.
10. **eventResponse.getEventDataResponse().setEventId(eventId);**: Sets the **eventId** in the **eventResponse** object.
11. The code then checks if there are additional properties within the **eventStatusDetail** and calls corresponding methods (**saveEventError**, **saveDocumentDetails**, or **saveEventNotify**) to handle them if they are present.
12. **eventMapper.updateEventRequestToEventResponse(createEventStatusRequest, eventResponse);**: Updates the **eventResponse** by mapping properties from the **createEventStatusRequest** object.

Finally, the method returns an **Optional** containing the **eventResponse**.

Overall, this code seems to be responsible for saving event status information into a database and generating an **EventResponse** object based on the input parameters. It handles various optional properties in the input and may perform additional actions depending on their presence. The use of **@Transactional** suggests that this method is expected to run as a single database transaction to ensure data consistency.

**Positive Scenario - Successful Save:**

* Test the method when it is expected to execute successfully.
* Mock the dependencies such as dmEventStatusRepository and eventMapper to return expected values.
* Provide valid input values to eventId and createEventStatusRequest.

private DocumentGeneratorEventStoreServiceImpl service;  
 @BeforeEach  
 public void setUp() {  
 MockitoAnnotations.*initMocks*(this);  
  
 service = new DocumentGeneratorEventStoreServiceImpl();  
 service.dmEventStatusRepository = dmEventStatusRepository;  
 service.eventMapper = eventMapper;  
 service.*LOG* = logger;  
 }  
 @Test  
 public void testSaveEventStatus\_SuccessfulSave() {  
 // Create a sample eventId and createEventStatusRequest  
 String eventId = "123";  
 CreateEventStatusRequest createEventStatusRequest = createSampleRequest();  
  
 // Create a sample DmEventStatus object  
 DmEventStatus sampleDmEventStatus = createSampleDmEventStatus();  
  
 // Create a sample EventResponse object  
 EventResponse sampleEventResponse = createSampleEventResponse();  
  
 // Mock the behavior of dependencies  
 *when*(dmEventStatusRepository.save(*any*(DmEventStatus.class))).thenReturn(sampleDmEventStatus);  
  
 // Mock the behavior of the void method eventMapper.updateEventRequestToEventResponse  
 //To mock a void method, you can use the doNothing() method.  
 *doNothing*().when(eventMapper).updateEventRequestToEventResponse(*any*(CreateEventStatusRequest.class), *any*(EventResponse.class));  
  
  
 // Call the method you want to test  
 Optional<EventResponse> result = service.saveEventStatus(eventId, createEventStatusRequest);  
  
 // Assertions  
 *assertEquals*(Optional.*of*(sampleEventResponse), result);  
  
 // Verify that dmEventStatusRepository.save and eventMapper.updateEventRequestToEventResponse were called with expected arguments  
 *verify*(dmEventStatusRepository, *times*(1)).save(*any*(DmEventStatus.class));  
 *verify*(eventMapper, *times*(1)).updateEventRequestToEventResponse(*eq*(createEventStatusRequest), *any*(EventResponse.class));  
 }  
 // Helper methods to create sample objects  
 private CreateEventStatusRequest createSampleRequest() {  
 // Implement this method to create a sample request  
 CreateEventStatusRequest request = new CreateEventStatusRequest();  
 // Set request properties as needed for the test  
 return request;  
 }  
  
 private DmEventStatus createSampleDmEventStatus() {  
 // Implement this method to create a sample DmEventStatus  
 DmEventStatus dmEventStatus = new DmEventStatus();  
 // Set DmEventStatus properties as needed for the test  
 return dmEventStatus;  
 }  
  
 private EventResponse createSampleEventResponse() {  
 // Implement this method to create a sample EventResponse  
 EventResponse eventResponse = new EventResponse();  
 // Set EventResponse properties as needed for the test  
 return eventResponse;  
 }  
}

1. **private DocumentGeneratorEventStoreServiceImpl service;**: This line declares an instance of the class under test, **DocumentGeneratorEventStoreServiceImpl**, named **service**.
2. **@BeforeEach**: This is a JUnit annotation that indicates a setup method to be executed before each test method is run.
3. **public void setUp()**: This is the setup method, which is executed before each test case. It is responsible for preparing the test environment.
4. **MockitoAnnotations.initMocks(this);**: This line initializes mock objects using Mockito's **initMocks** method. It prepares the annotated fields marked with **@Mock** for mocking behavior.
5. **service = new DocumentGeneratorEventStoreServiceImpl();**: Here, an instance of the **DocumentGeneratorEventStoreServiceImpl** class is created.
6. **service.dmEventStatusRepository = dmEventStatusRepository;**: This sets the **dmEventStatusRepository** field in the **service** instance with the mock **dmEventStatusRepository**.
7. **service.eventMapper = eventMapper;**: Similarly, this sets the **eventMapper** field in the **service** instance with the mock **eventMapper**.
8. **service.LOG = logger;**: This sets the **LOG** field in the **service** instance with the mock **logger**. This allows the test to capture and verify log statements.
9. **@Test**: This is a JUnit annotation that marks a method as a test case.
10. **public void testSaveEventStatus\_SuccessfulSave()**: This is the actual test method. It is responsible for testing the positive scenario where the **saveEventStatus** method is expected to execute successfully.
11. Inside the test method, you do the following:
12. Create sample input values such as **eventId**, **createEventStatusRequest**, **sampleDmEventStatus**, and **sampleEventResponse**.
13. Mock the behavior of the **dmEventStatusRepository.save** method to return **sampleDmEventStatus** when called with any **DmEventStatus** argument.
14. Mock the behavior of the **eventMapper.updateEventRequestToEventResponse** method using **doNothing()** since it's a void method.
15. Call the **saveEventStatus** method on the **service** instance with the sample input values.
16. Assert that the result matches the expected **Optional<EventResponse>**, which is **sampleEventResponse**.
17. Verify that **dmEventStatusRepository.save** and **eventMapper.updateEventRequestToEventResponse** methods were called with the expected arguments and the expected number of times.
18. The test class also includes helper methods (**createSampleRequest**, **createSampleDmEventStatus**, and **createSampleEventResponse**) to create sample objects with test data.
19. Overall, this test class sets up a testing environment, mocks the behavior of dependencies, executes the method under test, and verifies that the expected behavior occurs. It's a typical structure for writing JUnit tests.

Top of Form

**2.Save Event Error Scenario:**

* Test the method when an event error request is present in createEventStatusRequest.
* Mock the dependencies such as dmEventStatusRepository and eventMapper to return expected values.
* Ensure that the method correctly calls saveEventError with the event error request.
* @Test  
   public void testSaveEventStatus\_WithEventError() {  
   // Create a sample eventId and createEventStatusRequest with an event error request  
   String eventId = "123";  
   CreateEventStatusRequest createEventStatusRequest = createSampleRequestWithEventError();  
    
   // Create a sample DmEventStatus object  
   DmEventStatus sampleDmEventStatus = createSampleDmEventStatus();  
    
   // Create a sample EventResponse object  
   EventResponse sampleEventResponse = createSampleEventResponse();  
    
   // Mock the behavior of dependencies  
   *when*(dmEventStatusRepository.save(*any*(DmEventStatus.class))).thenReturn(sampleDmEventStatus);  
   //when(eventMapper.updateEventRequestToEventResponse(any(CreateEventStatusRequest.class), any(EventResponse.class))).thenReturn(sampleEventResponse);  
   *doNothing*().when(eventMapper).updateEventRequestToEventResponse(*any*(CreateEventStatusRequest.class), *any*(EventResponse.class));  
    
    
   // Call the method you want to test  
   Optional<EventResponse> result = service.saveEventStatus(eventId, createEventStatusRequest);  
    
   // Assertions  
   *assertEquals*(Optional.*of*(sampleEventResponse), result);  
    
   // Verify that dmEventStatusRepository.save and eventMapper.updateEventRequestToEventResponse were called with expected arguments  
   *verify*(dmEventStatusRepository, *times*(1)).save(*any*(DmEventStatus.class));  
   *verify*(eventMapper, *times*(1)).updateEventRequestToEventResponse(*eq*(createEventStatusRequest), *any*(EventResponse.class));  
    
   // Verify that saveEventError was called with the event error request  
   *verify*(service, *times*(1)).saveEventError(*eq*(createEventStatusRequest.getEventStatusRequest().getEventStatus().getEventErrorRequest()), *eq*(sampleDmEventStatus.getEventId()), *eq*(sampleDmEventStatus));  
   }  
    
   // Helper methods to create sample objects  
   private CreateEventStatusRequest createSampleRequestWithEventError() {  
   // Implement this method to create a sample request with an event error  
   CreateEventStatusRequest request = new CreateEventStatusRequest();  
  // EventStatusRequest.EventStatusDetail eventStatusDetail = new EventStatusRequest.EventStatusDetail();  
    
   // EventStatusRequest.EventErrorRequest eventErrorRequest = new EventStatusRequest.EventErrorRequest();  
   // eventStatusDetail.setEventErrorRequest(eventErrorRequest);  
   //request.setEventStatusRequest(eventStatusDetail);  
   // Set other request properties as needed for the test  
   return request;  
   }

**3.Save Document Details Scenario**:

* Test the method when a document request is present in createEventStatusRequest.
* Mock the dependencies such as dmEventStatusRepository and eventMapper to return expected values.
* Ensure that the method correctly calls saveDocumentDetails with the document request.
* @Test  
   public void testSaveEventStatus\_WithDocumentRequest() {  
   // Create a sample eventId and createEventStatusRequest with a document request  
   String eventId = "123";  
   CreateEventStatusRequest createEventStatusRequest = createSampleRequestWithDocumentRequest();  
    
   // Create a sample DmEventStatus object  
   DmEventStatus sampleDmEventStatus = createSampleDmEventStatus();  
    
   // Create a sample EventResponse object  
   EventResponse sampleEventResponse = createSampleEventResponse();  
    
   // Mock the behavior of dependencies  
   *when*(dmEventStatusRepository.save(*any*(DmEventStatus.class))).thenReturn(sampleDmEventStatus);  
   // when(eventMapper.updateEventRequestToEventResponse(any(CreateEventStatusRequest.class), any(EventResponse.class))).thenReturn(sampleEventResponse);  
    
   *doNothing*().when(eventMapper).updateEventRequestToEventResponse(*any*(CreateEventStatusRequest.class), *any*(EventResponse.class));  
   // Call the method you want to test  
   Optional<EventResponse> result = service.saveEventStatus(eventId, createEventStatusRequest);  
    
   // Assertions  
   *assertEquals*(Optional.*of*(sampleEventResponse), result);  
    
   // Verify that dmEventStatusRepository.save and eventMapper.updateEventRequestToEventResponse were called with expected arguments  
   *verify*(dmEventStatusRepository, *times*(1)).save(*any*(DmEventStatus.class));  
   *verify*(eventMapper, *times*(1)).updateEventRequestToEventResponse(*eq*(createEventStatusRequest), *any*(EventResponse.class));  
    
   // Verify that saveDocumentDetails was called with the document request  
   *verify*(service, *times*(1)).saveDocumentDetails(*eq*(createEventStatusRequest.getEventStatusRequest().getEventStatus().getDocumentRequest()), *eq*(sampleDmEventStatus.getEventId()), *eq*(sampleEventResponse));  
   }  
    
   // Helper methods to create sample objects  
   private CreateEventStatusRequest createSampleRequestWithDocumentRequest() {  
   // Implement this method to create a sample request with a document request  
   CreateEventStatusRequest request = new CreateEventStatusRequest();  
  // EventStatusRequest.EventStatusDetail eventStatusDetail = new EventStatusRequest.EventStatusDetail();  
  // EventStatusRequest.DocumentRequest documentRequest = new EventStatusRequest.DocumentRequest();  
  // eventStatusDetail.setDocumentRequest(documentRequest);  
  // request.setEventStatusRequest(eventStatusDetail);  
   // Set other request properties as needed for the test  
   return request;  
   }

Here's an explanation of the code for documentation:

**@Test Annotation:**

This annotation marks the following method as a JUnit test case, which will be executed when the test suite is run.

**public void testSaveEventStatus\_WithDocumentRequest() { ... }** Method:

This method contains the actual test case for the "Save Document Details Scenario."

**Sample Data Preparation:**

It starts by creating sample data, including an **eventId**, a **createEventStatusRequest** with a document request, a sample **DmEventStatus**, and a sample **EventResponse**. These samples will be used for testing.

**Mocking Dependencies:**

The behavior of the **dmEventStatusRepository** and **eventMapper** dependencies is mocked to control their behavior during the test. Specifically:

**dmEventStatusRepository.save(any(DmEventStatus.class))** is mocked to return the sample **DmEventStatus** when **save** is called.

**eventMapper.updateEventRequestToEventResponse(any(CreateEventStatusRequest.class), any(EventResponse.class))** is mocked using **doNothing()** because it's a void method. You're telling Mockito to do nothing when this method is called.

**Method Invocation:**

The **saveEventStatus** method of the **service** object is called with the sample data, simulating the behavior in your application.

Assertions:

The test asserts that the result of calling **saveEventStatus** matches the expected result, which is an **Optional** containing the sample **EventResponse**.

**Verification:**

The test verifies that specific methods were called with the expected arguments:

**dmEventStatusRepository.save(any(DmEventStatus.class))** is verified to have been called once.

**eventMapper.updateEventRequestToEventResponse(eq(createEventStatusRequest), any(EventResponse.class))** is verified to have been called once with the provided arguments.

**saveDocumentDetails** is verified to have been called once with specific arguments related to the document request.

**Helper Method:**

**createSampleRequestWithDocumentRequest** is a helper method that should be implemented to create a sample **CreateEventStatusRequest** with a document request. In your current code, this method is commented out, and you should implement it to create the desired sample request.

Overall, this test case is designed to ensure that the **saveEventStatus** method behaves correctly when a document request is present in the input, and it verifies that the method interacts with its dependencies as expected.

**4.Save Event Notify Scenario:**

* Test the method when an event notify data request is present in createEventStatusRequest.
* Mock the dependencies such as dmEventStatusRepository and eventMapper to return expected values.
* Ensure that the method correctly calls saveEventNotify with the event notify data request.
* @Test  
  public void testSaveEventStatus\_WithEventNotifyDataRequest() {  
   // Create a sample eventId and createEventStatusRequest with an event notify data request  
   String eventId = "123";  
   CreateEventStatusRequest createEventStatusRequest = createSampleRequestWithEventNotifyDataRequest();  
    
   // Create a sample DmEventStatus object  
   DmEventStatus sampleDmEventStatus = createSampleDmEventStatus();  
    
   // Create a sample EventResponse object  
   EventResponse sampleEventResponse = createSampleEventResponse();  
    
   // Mock the behavior of dependencies  
   *when*(dmEventStatusRepository.save(*any*(DmEventStatus.class))).thenReturn(sampleDmEventStatus);  
   *doNothing*().when(eventMapper).updateEventRequestToEventResponse(*any*(CreateEventStatusRequest.class), *any*(EventResponse.class));  
   // Mock the behavior of saveEventNotify  
   *when*(dmEventNotifyRepository.save(*any*(DmEventNotify.class))).thenReturn(new DmEventNotify());  
    
   // Call the method you want to test  
   Optional<EventResponse> result = service.saveEventStatus(eventId, createEventStatusRequest);  
    
   // Assertions  
   *assertEquals*(Optional.*of*(sampleEventResponse), result);  
    
   // Verify that dmEventStatusRepository.save and eventMapper.updateEventRequestToEventResponse were called with expected arguments  
   *verify*(dmEventStatusRepository, *times*(1)).save(*any*(DmEventStatus.class));  
   *verify*(eventMapper, *times*(1)).updateEventRequestToEventResponse(*eq*(createEventStatusRequest), *any*(EventResponse.class));  
    
   // Verify that saveEventNotify was called with the event notify data request  
   *verify*(dmEventNotifyRepository, *times*(1)).save(*any*(DmEventNotify.class));  
  }  
    
   // Helper methods to create sample objects  
   private CreateEventStatusRequest createSampleRequestWithEventNotifyDataRequest() {  
   // Implement this method to create a sample request with an event notify data request  
   CreateEventStatusRequest request = new CreateEventStatusRequest();  
   EventStatusRequest.EventStatusDetail eventStatusDetail = new EventStatusRequest.EventStatusDetail();  
    
  // EventStatusRequest.EventNotifyDataRequest eventNotifyDataRequest = new EventStatusRequest.EventNotifyDataRequest();  
  // eventStatusDetail.setEventNotifyDataRequest(eventNotifyDataRequest);  
  // request.setEventStatusRequest(eventStatusDetail);  
    
   // Set other request properties as needed for the test  
   return request;  
   }

This code block is a JUnit test method for the **saveEventStatus** method in the **DocumentGeneratorEventStoreServiceImpl** class. It is specifically testing the scenario where an event notify data request is present in the **createEventStatusRequest**. Let's break down the key elements of this test method for documentation:

**@Test Annotation:** The **@Test** annotation indicates that this method is a JUnit test case.

**Test Method Name:** The method name is **testSaveEventStatus\_WithEventNotifyDataRequest**, which describes the purpose of the test: to verify the behavior of **saveEventStatus** when there is an event notify data request.

**Test Data Setup:**

It creates a sample **eventId** and a **createEventStatusRequest** object with an event notify data request using the **createSampleRequestWithEventNotifyDataRequest** helper method.

It creates sample objects **sampleDmEventStatus** and **sampleEventResponse** that are expected to be returned by mocked dependencies and used for assertions.

**Mocking Dependencies:**

The test mocks the behavior of dependencies using Mockito:

**dmEventStatusRepository.save** is mocked to return **sampleDmEventStatus** when called with any **DmEventStatus** object.

**eventMapper.updateEventRequestToEventResponse** is mocked to do nothing when called with any **CreateEventStatusRequest** and **EventResponse** objects.

**dmEventNotifyRepository.save** is mocked to return a new **DmEventNotify** object when called with any **DmEventNotify** object.

**Method Invocation:** The test then calls the **saveEventStatus** method on the **service** object with the sample **eventId** and **createEventStatusRequest**.

**Assertions:**

It asserts that the result of the **saveEventStatus** method is equal to **Optional.of(sampleEventResponse)**, indicating that the method returned the expected response.

**Verification:**

It verifies that **dmEventStatusRepository.save** was called exactly once with any **DmEventStatus** object.

It verifies that **eventMapper.updateEventRequestToEventResponse** was called exactly once with the expected **createEventStatusRequest** and any **EventResponse** object.

It verifies that **dmEventNotifyRepository.save** was called exactly once with any **DmEventNotify** object.

**Helper Methods:**

The **createSampleRequestWithEventNotifyDataRequest** method is a helper method used to create a sample **CreateEventStatusRequest** with an event notify data request. In this test, the event notify data request is commented out to simulate the presence of an event notify data request.

This test case covers the scenario where the **saveEventStatus** method is expected to handle and save event notify data when it is present in the request. It uses mocking to isolate the method under test and verify its interactions with dependencies.

**5.Error Handling Scenario:**

* Test the method when an exception is thrown, such as a **DataAccessException** during the repository save operation.
* Mock the dependencies to throw the desired exception.
* Verify that the method handles the exception appropriately, possibly by returning an empty **Optional** or logging an error.
* //Method-2 Testcase:5  
   //Error Handling Scenario:  
   //• Test the method when an exception is thrown, such as a DataAccessException during the repository save operation.  
   //• Mock the dependencies to throw the desired exception.  
   //• Verify that the method handles the exception appropriately, possibly by returning an empty Optional or logging an error.  
  @Test  
  public void testSaveEventStatus\_ExceptionHandling() {  
   // Create a sample eventId and createEventStatusRequest  
   String eventId = "123";  
   CreateEventStatusRequest createEventStatusRequest = createSampleRequest();  
    
   // Mock the behavior of dependencies to throw DataAccessException when saving  
   *when*(dmEventStatusRepository.save(*any*(DmEventStatus.class)))  
   .thenThrow(new DataAccessException("Simulated database error") {});  
    
   // Call the method you want to test  
   Optional<EventResponse> result = service.saveEventStatus(eventId, createEventStatusRequest);  
    
   // Assertions  
   *assertTrue*(result.isEmpty()); // Check that the result is an empty Optional  
    
   // Verify that dmEventStatusRepository.save was called with expected arguments  
   *verify*(dmEventStatusRepository, *times*(1)).save(*any*(DmEventStatus.class));  
    
   // Verify that eventMapper.updateEventRequestToEventResponse was not called  
   *verify*(eventMapper, *never*()).updateEventRequestToEventResponse(*any*(CreateEventStatusRequest.class), *any*(EventResponse.class));  
  }

Explanation:

1. In this test case, we set up the **DocumentGeneratorEventStoreServiceImpl** class as a test subject (**service**) and mock its dependencies (**dmEventStatusRepository** and **eventMapper**) using **@Mock** and **@InjectMocks** annotations.
2. The **setUp** method is annotated with **@BeforeEach** and initializes the Mockito annotations.
3. The **testSaveEventStatus\_ExceptionHandling** method tests the scenario where the repository throws a **DataAccessException**. We mock this behavior using **when(...).thenThrow(...)**. When the repository save operation throws an exception, the method is expected to handle it.
4. We call the **saveEventStatus** method with sample data.
5. Assertions:
6. We check that the result is an empty **Optional** using **assertTrue(result.isEmpty())**, which verifies that the method handled the exception appropriately.
7. Verification:
8. We verify that **dmEventStatusRepository.save** was called with the expected arguments.

We verify that **eventMapper.updateEventRequestToEventResponse** was not called because we expect the method to return early when the exception occurs during the save operation.

This test case verifies the error handling behavior of the **saveEventStatus** method when a **DataAccessException** is thrown during the repository save operation.

**6.Edge Cases - Null or Empty Inputs:**

* Test the method with null or empty values for eventId and createEventStatusRequest.
* Ensure that the method handles these cases gracefully, possibly returning an empty Optional or logging errors.
* //Method:2 Testcases -6  
   //• Test the method with null or empty values for eventId and createEventStatusRequest.  
  //• Ensure that the method handles these cases gracefully, possibly returning an empty Optional or logging errors.  
  @Test  
  public void testSaveEventStatus\_NullEmptyInputs() {  
   // Test with null or empty values for eventId and createEventStatusRequest  
   String nullEventId = null;  
   CreateEventStatusRequest nullRequest = null;  
   String emptyEventId = "";  
   CreateEventStatusRequest emptyRequest = new CreateEventStatusRequest();  
    
   // Call the method with null eventId and request  
   Optional<EventResponse> resultNull = service.saveEventStatus(nullEventId, nullRequest);  
    
   // Call the method with empty eventId and request  
   Optional<EventResponse> resultEmpty = service.saveEventStatus(emptyEventId, emptyRequest);  
    
   // Assertions  
   *assertTrue*(resultNull.isEmpty()); // Check that the result is an empty Optional for null inputs  
   *assertTrue*(resultEmpty.isEmpty()); // Check that the result is an empty Optional for empty inputs  
    
   // Verify that dmEventStatusRepository.save and eventMapper.updateEventRequestToEventResponse were not called  
   *verify*(dmEventStatusRepository, *never*()).save(*any*(DmEventStatus.class));  
   *verify*(eventMapper, *never*()).updateEventRequestToEventResponse(*any*(CreateEventStatusRequest.class), *any*(EventResponse.class));  
  }

**Explanation:**

1. In this test case, we set up the **DocumentGeneratorEventStoreServiceImpl** class as a test subject (**service**) and mock its dependencies (**dmEventStatusRepository** and **eventMapper**) using **@Mock** and **@InjectMocks** annotations.
2. The **setUp** method is annotated with **@BeforeEach** and initializes the Mockito annotations.
3. The **testSaveEventStatus\_NullEmptyInputs** method tests the scenario where the method is called with null or empty inputs for **eventId** and **createEventStatusRequest**.
4. We call the **saveEventStatus** method with null and empty inputs.
5. Assertions:
6. We check that the results are empty **Optional** instances using **assertTrue(resultNull.isEmpty())** and **assertTrue(resultEmpty.isEmpty())**. This verifies that the method handles null and empty inputs by returning empty **Optional** instances.
7. Verification:
8. We verify that **dmEventStatusRepository.save** and **eventMapper.updateEventRequestToEventResponse** were not called because we expect the method to return early without performing these operations when null or empty inputs are provided.

This test case covers the edge cases where the **saveEventStatus** method handles null or empty inputs gracefully.

**7.Concurrency Testing:**

Simulate concurrent requests to the method and verify that it maintains data consistency and thread safety.

@RunWith(SpringRunner.class)

@SpringBootTest

//Method -2 Testcase-7  
//Concurrency Testing:  
//Simulate concurrent requests to the method and verify that it maintains data consistency and thread safety.  
@Test  
public void testConcurrentSaveEventStatus() throws InterruptedException {  
 int numThreads = 10; // Number of concurrent threads  
 CountDownLatch latch = new CountDownLatch(numThreads);  
 ExecutorService executorService = Executors.*newFixedThreadPool*(numThreads);  
  
 // Mock the behavior of dmEventStatusRepository.save  
 *when*(dmEventStatusRepository.save(*any*(DmEventStatus.class)))  
 .thenAnswer(invocation -> {  
 // Simulate some processing time  
 Thread.*sleep*(100);  
 return invocation.getArgument(0);  
 });  
  
 for (int i = 0; i < numThreads; i++) {  
 executorService.submit(() -> {  
 try {  
 String eventId = "123";  
 CreateEventStatusRequest createEventStatusRequest = createSampleRequest();  
  
 // Call the method you want to test concurrently  
 service.saveEventStatus(eventId, createEventStatusRequest);  
  
 } catch (Exception e) {  
 e.printStackTrace();  
 } finally {  
 latch.countDown();  
 }  
 });  
 }  
  
 // Wait for all threads to complete  
 latch.await();  
  
 // Verify that dmEventStatusRepository.save was called the expected number of times  
 *verify*(dmEventStatusRepository, *times*(numThreads)).save(*any*(DmEventStatus.class));  
  
 // Clean up  
 executorService.shutdown();  
}

In this test, we create a pool of concurrent threads that call the **saveEventStatus** method with sample data. The **CountDownLatch** is used to ensure that all threads complete their execution before proceeding with verifications. The **dmEventStatusRepository.save** method is mocked to simulate some processing time and ensure that concurrent calls are handled correctly.

Make sure to adjust the number of concurrent threads and the sample data as needed for your specific testing requirements. Additionally, consider testing various scenarios, including cases where concurrent requests may result in conflicts or data consistency issues.

**Timeout Handling:**

Simulate long-running operations or timeouts and verify that the method handles them appropriately.

//Method -2 Testcase-8  
 //Timeout Handling:  
 //Simulate long-running operations or timeouts and verify that the method handles them appropriately.  
@Test  
public void testTimeoutHandling1() {  
 // Create a sample eventId and createEventStatusRequest  
 String eventId = "123";  
 CreateEventStatusRequest createEventStatusRequest = createSampleRequest();  
  
 // Mock the behavior of dmEventStatusRepository to throw a timeout exception  
 *when*(dmEventStatusRepository.save(*any*(DmEventStatus.class))).thenThrow(new DataAccessException("Timeout") {});  
  
 // Call the method you want to test  
 Optional<EventResponse> result = service.saveEventStatus(eventId, createEventStatusRequest);  
  
 // Assertions  
 *assertFalse*(result.isPresent()); // Check that the result is empty due to the timeout  
  
 // Verify that dmEventStatusRepository.save was called with expected arguments  
 *verify*(dmEventStatusRepository, *times*(1)).save(*any*(DmEventStatus.class));  
  
 // You can also verify other behaviors, such as logging or error handling, if applicable  
}

In this test, we mock the behavior of dmEventStatusRepository.save() to throw a DataAccessException with a message indicating a timeout. After calling the saveEventStatus method, we verify that the result is not present, indicating that the method appropriately handled the timeout scenario. Additionally, you can add further verifications depending on how you handle timeouts in your actual code, such as logging or error handling.