**package com.project.bos.dg.datastore.service.impl;  
  
import com.fasterxml.jackson.core.JsonProcessingException;  
import com.project.bos.dg.datastore.mapper.EventMapper;  
import com.project.bos.dg.datastore.model.entity.\*;  
import com.project.bos.dg.datastore.model.request.CreateEventStatusRequest;  
import com.project.bos.dg.datastore.model.request.EventRequest;  
import com.project.bos.dg.datastore.model.response.EventDataResponse;  
import com.project.bos.dg.datastore.model.response.EventErrorResponse;  
import com.project.bos.dg.datastore.model.response.EventResponse;  
import com.project.bos.dg.datastore.model.response.EventSummaryResponse;  
import com.project.bos.dg.datastore.repository.\*;  
import com.project.bos.dg.datastore.service.DocumentGeneratorEventStoreService;  
import jakarta.ws.rs.BadRequestException;  
import jakarta.ws.rs.ServerErrorException;  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
import org.mockito.InjectMocks;  
import org.mockito.Mock;  
import org.mockito.Mockito;  
import org.mockito.MockitoAnnotations;  
import org.springframework.dao.DataAccessException;  
import org.springframework.http.HttpStatus;  
  
import java.time.LocalDate;  
import java.util.\*;  
import java.util.concurrent.CountDownLatch;  
import java.util.concurrent.ExecutorService;  
import java.util.concurrent.Executors;  
  
import static jdk.internal.org.objectweb.asm.util.CheckClassAdapter.*verify*;  
import static org.junit.jupiter.api.Assertions.\*;  
import static org.mockito.ArgumentMatchers.*any*;  
import static org.mockito.ArgumentMatchers.*eq*;  
import static org.mockito.Mockito.\*;  
import static org.springframework.test.web.client.ExpectedCount.*never*;  
import static org.springframework.test.web.client.ExpectedCount.*times*;  
  
public class DocumentGeneratorEventStoreServiceTest {  
 @InjectMocks  
 private DocumentGeneratorEventStoreService eventStoreService;  
  
 @Mock  
 private EventStoreHelper eventStoreHelper;  
  
 @Mock  
 private EventMapper eventMapper;  
  
 @Mock  
 private ResponseMapper responseMapper;  
  
 @Mock  
 private DmEventRepository dmEventRepository;  
  
 @Mock  
 private DmEventDocumentRepository dmEventDocumentRepository;  
  
 @Mock  
 private DmEventErrorRepository dmEventErrorRepository;  
  
 @Mock  
 private DmEventNotifyRepository dmEventNotifyRepository;  
  
 @Mock  
 private DocumentRepository documentRepository;  
  
 @Mock  
 private DmEventStatusRepository dmEventStatusRepository;  
  
 //Method 1: Event creation success  
 //Scenario 1: testSaveEvent\_EventCreation\_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);  
 }  
  
 //Scenario 2: testSaveEvent\_AlreadyExists  
 @Test  
 public void testSaveEvent\_AlreadyExists() throws JsonProcessingException {  
 // Arrange  
 EventRequest eventRequest = new EventRequest();  
 eventRequest.setCorrelationId("correlationId");  
  
 EventResponse expectedResponse = new EventResponse();  
 expectedResponse.setCode(HttpStatus.*OK*.value());  
  
 *when*(eventStoreHelper.getEventDataResponse(*any*(EventRequest.class), *any*(EventResponse.class)))  
 .thenReturn(Optional.*of*(expectedResponse));  
  
 // Act  
 Optional<EventResponse> result = DocumentGeneratorEventStoreService.*saveEvent*(eventRequest);  
  
 // Assert  
 *assertTrue*(result.isPresent());  
 *assertEquals*(HttpStatus.*OK*.value(), result.get().getCode());  
 }  
  
 //Scenario 3: testSaveEvent\_EventUpdate\_Success  
 @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()));  
 }  
  
  
 //Scenario 4: 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()));  
 }  
  
 //Scenario 5: 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*());  
 }  
  
  
 //Scenario 6: 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*());  
 }  
  
 //Scenario 7: 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));  
 }  
  
 //Scenario 8: testSaveEventWithEmptyEventRequest  
 @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));  
 }  
  
 //Scenario 9: testSaveEventWithNullEventRequest  
 @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));  
 }  
  
 //Scenario 10: testSaveEventWithDataIntegrityViolationException  
 //to verify how the service handles various exceptions, both expected and unexpected.  
 @Test  
 public void testSaveEventWithDataIntegrityViolationException() throws JsonProcessingException, InterruptedException {  
 // 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*());  
 }  
  
 }  
  
 //Scenario 11: testSaveEventWithLargeCorrelationId  
 @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());  
 }  
 //Scenario 12: testSaveEventWithSmallCorrelationId  
 @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();  
 }  
  
 //Scenario 13: testSaveEventWithInvalidInput  
 //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));  
 }  
  
 //Scenario 14: testSaveEventWithMaliciousInput  
 //Provide incorrect input data to test the service's rejection of invalid or malicious requests  
 @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));  
 }  
 //Scenario 15: testIntegrationWithValidInput  
 //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());  
 }  
  
 //Scenario 16: testTimeoutHandling  
 // 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));  
 }  
  
 //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* METHOD 2\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 //1. 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;  
 }  
  
 //Scenario 1: testSaveEventStatus\_SuccessfulSave  
 @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*(-.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;  
 }  
  
  
 //Scenario 2: testSaveEventStatus\_WithEventError  
 //1. 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;  
 }  
  
 //Scenario 3: testSaveEventStatus\_WithDocumentRequest  
 //1. 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;  
 }  
  
 //Scenario 4: testSaveEventStatus\_WithEventNotifyDataRequest  
 //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;  
 }  
  
 //Scenario 5: testSaveEventStatus\_ExceptionHandling  
 //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));  
 }  
  
  
 //Scenario 6: testSaveEventStatus\_NullEmptyInputs  
 //• 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));  
 }  
  
 //Scenario 7: testConcurrentSaveEventStatus  
 //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();  
 }  
  
 //Scenario 8: testTimeoutHandling1  
 //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));  
 }  
  
//------------------------------------------------------------------------------------------------------------------  
  
 //Method 3:  
 //Scenario 1: testFetchEventByEventId\_Success  
 @Test  
 public void testFetchEventByEventId\_Success() {  
 // Mock the dependencies  
 DmEventRepository dmEventRepository = Mockito.*mock*(DmEventRepository.class);  
 DmEventDocumentRepository dmEventDocumentRepository = Mockito.*mock*(DmEventDocumentRepository.class);  
 EventMapper eventMapper = Mockito.*mock*(EventMapper.class);  
  
 // Create an instance of the service to be tested, passing the mocked dependencies  
 DocumentGeneratorEventStoreService eventService = new DocumentGeneratorEventStoreService(dmEventRepository, dmEventDocumentRepository, eventMapper);  
  
 // Mock data  
 String eventId = "1";  
 DmEvent dmEvent = new DmEvent(); // Create an appropriate DmEvent instance  
 DmEventDocument dmEventDocument = new DmEventDocument(); // Create an appropriate DmEventDocument instance  
 Document document = new Document(); // Create an appropriate Document instance  
 EventResponse eventResponse = new EventResponse(); // Create an appropriate EventResponse instance  
  
 // Set up the behavior of mocked dependencies  
 *when*(dmEventRepository.findById(1L)).thenReturn(Optional.*of*(dmEvent)); // Simulate finding the event  
 *when*(dmEventDocumentRepository.findByEventId(1L)).thenReturn(Optional.*of*(dmEventDocument)); // Simulate finding the document  
 *when*(eventMapper.documentToDocumentDTO(document)).thenReturn(new DocumentDTO()); // Simulate mapping to DocumentDTO  
  
 // Call the method to be tested  
 Optional<EventResponse> result = eventService.fetchEventByEventId(eventId);  
  
 // Assertions  
 *assertTrue*(result.isPresent()); // Ensure that the result is present  
 *assertEquals*(eventResponse, result.get()); // Check that the result matches the expected eventResponse  
 }  
  
 //Scenario 2: testFetchEventByEventId\_EventNotFound  
 @Test  
 public void testFetchEventByEventId\_EventNotFound() {  
 // Mock data  
 String eventId = "1";  
  
 // Set up the behavior of dmEventRepository.findById to return an empty Optional  
 *when*(dmEventRepository.findById(*eq*(1L))).thenReturn(Optional.*empty*());  
  
 // Call the method to be tested  
 Optional<EventResponse> result = eventStoreService.fetchEventByEventId(eventId);  
  
 // Assertions  
 *assertFalse*(result.isPresent()); // Ensure that the result is empty  
 }  
  
 //Scenario 3: testFetchEventByEventId\_DocumentNotFound  
 @Test  
 public void testFetchEventByEventId\_DocumentNotFound() {  
 // Mock data  
 String eventId = "1";  
 DmEvent dmEvent = new DmEvent(); // Create an appropriate DmEvent instance  
  
 // Set up the behavior of dmEventDocumentRepository.findByEventId to return an empty Optional  
 *when*(dmEventRepository.findById(*eq*(1L))).thenReturn(Optional.*empty*());  
  
 // Call the method to be tested  
 Optional<EventResponse> result = eventStoreService.fetchEventByEventId(eventId);  
  
 // Assertions  
 *assertTrue*(result.isPresent()); // Ensure that the result is present  
  
 }  
  
 //Scenario 4: testFetchEventByEventId\_NullEventId  
 @Test  
 public void testFetchEventByEventId\_NullEventId() {  
 // Mock data  
 String eventId = null; // Set eventId to null  
  
 // Call the method to be tested  
 Optional<EventResponse> result = eventStoreService.fetchEventByEventId(eventId);  
  
 // Assertions  
 *assertFalse*(result.isPresent()); // Ensure that the result is empty  
 }  
  
 //Scenario 5: testFetchEventByEventId\_EmptyEventId  
 @Test  
 public void testFetchEventByEventId\_EmptyEventId() {  
 // Mock data  
 String eventId = ""; // Set eventId to empty string  
  
 // Call the method to be tested  
 Optional<EventResponse> result = eventStoreService.fetchEventByEventId(eventId);  
  
 // Assertions  
 *assertFalse*(result.isPresent()); // Ensure that the result is empty  
 }  
  
 //Scenario 6: testFetchEventByEventId\_NotFoundInTheDatabase  
 @Test  
 public void testFetchEventByEventId\_NotFoundInTheDatabase() {  
 // Mock data  
 String eventId = "1"; // Set eventId to a valid ID  
 *when*(dmEventRepository.findById(1L)).thenReturn(Optional.*empty*()); // Simulate the record not being found in the database  
  
 // Call the method to be tested and expect a server error  
 ServerErrorException exception = *assertThrows*(ServerErrorException.class, () -> {  
 eventStoreService.fetchEventByEventId(eventId);  
 });  
  
 // Optionally, you can assert the error message or error code depending on your implementation  
 *assertEquals*("Not Found In the Database", exception.getMessage()); // Replace with your error message  
 }  
  
 //Scenario 7: testFetchEventByEventId\_MapperNotFound  
 @Test  
 public void testFetchEventByEventId\_MapperNotFound() {  
 // Mock data  
 String eventId = "1"; // Set eventId to a valid ID  
 DmEvent dmEvent = new DmEvent(); // Create an appropriate DmEvent instance  
 *when*(dmEventRepository.findById(1L)).thenReturn(Optional.*of*(dmEvent)); // Simulate finding the event  
  
 // Simulate that eventMapper.documentToDocumentDTO returns null (or throws an exception)  
 *when*(eventMapper.documentToDocumentDTO(*any*())).thenReturn(null);  
  
 // Call the method to be tested and expect a server error  
 ServerErrorException exception = *assertThrows*(ServerErrorException.class, () -> {  
 eventStoreService.fetchEventByEventId(eventId);  
 });  
  
 // Optionally, you can assert the error message or error code depending on your implementation  
 *assertEquals*("Mapper not found", exception.getMessage()); // Replace with your error message  
 }  
  
 //Scenario 8: testFetchEventByEventId\_OptionalEmpty  
 @Test  
 public void testFetchEventByEventId\_OptionalEmpty() {  
 // Mock data  
 String eventId = "1"; // Set eventId to a valid ID  
 *when*(dmEventRepository.findById(1L)).thenReturn(Optional.*empty*()); // Simulate finding no event  
  
 // Call the method to be tested  
 Optional<EventResponse> result = eventStoreService.fetchEventByEventId(eventId);  
  
 // Assertions  
 *assertFalse*(result.isPresent()); // Ensure that the result is empty  
 }  
  
 //Scenario 9: testFetchEventByEventId\_InvalidEventIdFormat  
 @Test  
 public void testFetchEventByEventId\_InvalidEventIdFormat() {  
 // Mock data  
 String invalidEventId = "invalid"; // Set eventId to an invalid non-numeric string  
  
 // Call the method to be tested and expect a BadRequestException (or a suitable exception for invalid input)  
 BadRequestException exception = *assertThrows*(BadRequestException.class, () -> {  
 eventStoreService.fetchEventByEventId(invalidEventId);  
 });  
  
 // Optionally, you can assert the error message or error code depending on your implementation  
 *assertEquals*("Invalid eventId format", exception.getMessage()); // Replace with your error message  
 }  
  
 //Scenario 10: testFetchEventByEventId\_EventFoundButDocumentNotFound  
 @Test  
 public void testFetchEventByEventId\_EventFoundButDocumentNotFound() {  
 // Mock data  
 String eventId = "1"; // Set eventId to a valid ID  
 DmEvent dmEvent = new DmEvent(); // Create an appropriate DmEvent instance  
 *when*(dmEventRepository.findById(1L)).thenReturn(Optional.*of*(dmEvent)); // Simulate finding the event  
  
 // Simulate that dmEventDocumentRepository.findByEventId returns an empty Optional  
 *when*(dmEventDocumentRepository.findByEventId(1L)).thenReturn(Optional.*empty*());  
  
 // Call the method to be tested  
 Optional<EventResponse> result = eventStoreService.fetchEventByEventId(eventId);  
  
 // Assertions  
 *assertTrue*(result.isPresent()); // Ensure that the result is present  
 }  
  
 // Scenario 11: testFetchEventByEventId\_MappingFailure  
 @Test  
 public void testFetchEventByEventId\_MappingFailure() {  
 // Mock data  
 String eventId = "1"; // Set eventId to a valid ID  
 DmEvent dmEvent = new DmEvent(); // Create an appropriate DmEvent instance  
 *when*(dmEventRepository.findById(1L)).thenReturn(Optional.*of*(dmEvent)); // Simulate finding the event  
  
 // Simulate that eventMapper.documentToDocumentDTO throws an exception  
 *when*(eventMapper.documentToDocumentDTO(*any*())).thenThrow(RuntimeException.class);  
  
 // Call the method to be tested and expect a ServerErrorException (or a suitable exception for mapping failure)  
 ServerErrorException exception = *assertThrows*(ServerErrorException.class, () -> {  
 eventStoreService.fetchEventByEventId(eventId);  
 });  
  
 // Optionally, you can assert the error message or error code depending on your implementation  
 *assertEquals*("Mapping failed", exception.getMessage()); // Replace with your error message  
 }  
  
//------------------------------------------------------------------------------------------------------------------------  
  
 //Method 4: fetchErrorsByEventId  
  
 //Scenario 1: testFetchErrorsByEventIdSuccess  
 @Test  
 public void testFetchErrorsByEventId\_Success() {  
 // Arrange  
 String eventId = "123";  
 List<DmEventError> eventErrors = new ArrayList<>();  
  
 // Mock the repository to return an empty list of errors  
 *when*(dmEventErrorRepository.findByEventId(Long.*parseLong*(eventId))).thenReturn(Optional.*of*(eventErrors));  
  
 // Act  
 Optional<EventResponse> result = eventStoreService.fetchErrorsByEventId(eventId);  
  
 // Assert  
 *assertTrue*(result.isPresent());  
  
 // Validate the EventResponse and its properties  
 EventResponse eventResponse = result.get();  
 *assertNotNull*(eventResponse);  
  
 // Validate the EventDataResponse and its properties  
 EventDataResponse eventDataResponse = eventResponse.getEventDataResponse();  
 *assertNotNull*(eventDataResponse);  
 *assertEquals*(eventId, eventDataResponse.getEventId());  
  
 // Validate that the list of error responses is empty  
 *assertTrue*(eventDataResponse.getEventErrorResponses().isEmpty());  
 }  
  
 //Scenario 2:  
 @Test  
 public void testFetchErrorsByEventId\_EmptyErrors() {  
 // Arrange  
 String eventId = "123";  
 List<DmEventError> emptyEventErrors = new ArrayList<>();  
  
 // Mock the repository to return an empty list of errors  
 *when*(dmEventErrorRepository.findByEventId(Long.*parseLong*(eventId))).thenReturn(Optional.*of*(emptyEventErrors));  
  
 // Act  
 Optional<EventResponse> result = eventStoreService.fetchErrorsByEventId(eventId);  
  
 // Assert  
 *assertTrue*(result.isPresent());  
  
 // Validate the EventResponse and its properties  
 EventResponse eventResponse = result.get();  
 *assertNotNull*(eventResponse);  
  
 // Validate the EventDataResponse and its properties  
 EventDataResponse eventDataResponse = eventResponse.getEventDataResponse();  
 *assertNotNull*(eventDataResponse);  
 *assertEquals*(eventId, eventDataResponse.getEventId());  
  
 // Validate that the list of error responses is empty  
 *assertTrue*(eventDataResponse.getEventErrorResponses().isEmpty());  
 }  
  
 //Scenario 3:testFetchErrorsByEventId\_WithErrors  
 @Test  
 public void testFetchErrorsByEventId\_WithErrors() {  
 // Arrange  
 String eventId = "123";  
  
 // Create a list of actual errors  
 List<DmEventError> actualEventErrors = new ArrayList<>();  
 DmEventError error1 = new DmEventError(); // Create and populate error objects as needed  
 DmEventError error2 = new DmEventError();  
 actualEventErrors.add(error1);  
 actualEventErrors.add(error2);  
  
 // Mock the repository to return the list of actual errors  
 *when*(dmEventErrorRepository.findByEventId(Long.*parseLong*(eventId))).thenReturn(Optional.*of*(actualEventErrors));  
  
 // Act  
 Optional<EventResponse> result = eventStoreService.fetchErrorsByEventId(eventId);  
  
 // Assert  
 *assertTrue*(result.isPresent());  
  
 // Validate the EventResponse and its properties  
 EventResponse eventResponse = result.get();  
 *assertNotNull*(eventResponse);  
  
 // Validate the EventDataResponse and its properties  
 EventDataResponse eventDataResponse = eventResponse.getEventDataResponse();  
 *assertNotNull*(eventDataResponse);  
 *assertEquals*(eventId, eventDataResponse.getEventId());  
  
 // Validate the list of error responses  
 List<EventErrorResponse> errorResponses = eventDataResponse.getEventErrorResponses();  
 *assertNotNull*(errorResponses);  
 *assertFalse*(errorResponses.isEmpty());  
 *assertEquals*(actualEventErrors.size(), errorResponses.size());  
  
 // Validate each EventErrorResponse  
 for (int i = 0; i < actualEventErrors.size(); i++) {  
 EventErrorResponse expectedErrorResponse = eventMapper.mapDmEventErrorToEventErrorResponse(actualEventErrors.get(i));  
 EventErrorResponse actualErrorResponse = errorResponses.get(i);  
 *assertEquals*(expectedErrorResponse.getErrorCode(), actualErrorResponse.getErrorCode());  
 *assertEquals*(expectedErrorResponse.getErrorMessage(), actualErrorResponse.getErrorMessage());  
 // Add more assertions as needed for each error response field.  
 }  
 }  
  
 //Scenario 4: testFetchErrorsByEventIdEvent\_NotFound  
 @Test  
 public void testFetchErrorsByEventIdEvent\_NotFound() {  
 // Arrange  
 String eventId = "123";  
  
 // Mock the repository to return an empty Optional, indicating no event found  
 *when*(dmEventErrorRepository.findByEventId(Long.*parseLong*(eventId))).thenReturn(Optional.*empty*());  
  
 // Act  
 Optional<EventResponse> result = eventStoreService.fetchErrorsByEventId(eventId);  
  
 // Assert  
 *assertTrue*(result.isPresent());  
  
 // Validate the EventResponse and its properties  
 EventResponse eventResponse = result.get();  
 *assertNotNull*(eventResponse);  
  
 // Validate the EventDataResponse and its properties  
 EventDataResponse eventDataResponse = eventResponse.getEventDataResponse();  
 *assertNotNull*(eventDataResponse);  
 *assertEquals*(eventId, eventDataResponse.getEventId());  
  
 // Ensure that the list of error responses is empty  
 *assertTrue*(eventDataResponse.getEventErrorResponses().isEmpty());  
 }  
  
 //Scenario 5: testFetchErrorsByEventId\_DatabaseError  
 @Test  
 public void testFetchErrorsByEventId\_DatabaseError() {  
 // Arrange  
 String eventId = "123";  
  
 // Mock the repository to throw a DataAccessException (simulating a database error)  
 *when*(dmEventErrorRepository.findByEventId(Long.*parseLong*(eventId)))  
 .thenThrow(new DataAccessException("Simulated database error") {});  
  
 // Act  
 Optional<EventResponse> result = eventStoreService.fetchErrorsByEventId(eventId);  
  
 // Assert  
 *assertFalse*(result.isPresent()); // Expecting an empty result  
 }  
  
 //Scenario 6: testFetchErrorsBy\_InvalidEventId  
 @Test  
 public void testFetchErrorsBy\_InvalidEventId() {  
 // Arrange  
 String invalidEventId = "invalid\_id"; // Invalid non-numeric event ID  
  
 // Act and Assert  
 try {  
 eventStoreService.fetchErrorsByEventId(invalidEventId);  
  
 // Fail the test if no exception is thrown  
 *fail*("Expected NumberFormatException, but no exception was thrown.");  
 } catch (NumberFormatException e) {  
 // Expected NumberFormatException, do nothing  
 }  
 }  
  
 //Scenario 7: testFetchErrorsBy\_EmptyEventId  
 @Test  
 public void testFetchErrorsBy\_EmptyEventId() {  
 // Arrange  
 String emptyEventId = ""; // Empty event ID  
  
 try {  
 // Act  
 eventStoreService.fetchErrorsByEventId(emptyEventId);  
  
 // Fail the test if no exception is thrown  
 *fail*("Expected IllegalArgumentException, but no exception was thrown.");  
 } catch (IllegalArgumentException e) {  
 // Expected IllegalArgumentException, do nothing  
 }  
 }  
  
 //Scenario 8: testFetchErrorsBy\_NullEventId  
 @Test  
 public void testFetchErrorsBy\_NullEventId() {  
 // Arrange  
 String nullEventId = null; // Null event ID  
  
 try {  
 // Act  
 eventStoreService.fetchErrorsByEventId(nullEventId);  
  
 // Fail the test if no exception is thrown  
 *fail*("Expected IllegalArgumentException, but no exception was thrown.");  
 } catch (IllegalArgumentException e) {  
 // Expected IllegalArgumentException, do nothing  
 }  
 }  
  
 //Scenario 9: testFetchErrorsBy\_MaxEventId  
 @Test  
 public void testFetchErrorsBy\_MaxEventId() {  
 // Arrange  
 long maxEventId = Long.*MAX\_VALUE*; // Maximum possible event ID  
 String eventId = String.*valueOf*(maxEventId);  
  
 // Create a list of mock event errors  
 List<DmEventError> mockEventErrors = new ArrayList<>();  
  
 // Mock the repository to return the list of mock event errors  
 *when*(dmEventErrorRepository.findByEventId(maxEventId)).thenReturn(Optional.*of*(mockEventErrors));  
  
 // Act  
 Optional<EventResponse> result = eventStoreService.fetchErrorsByEventId(eventId);  
  
 // Assert  
 *assertTrue*(result.isPresent());  
 }  
  
 //Scenario 10:  
 @Test  
 public void testFetchErrorsWithCustomException() {  
 // Arrange  
 String eventId = "123";  
  
 // Mock the repository to return a custom exception when called  
 *when*(dmEventErrorRepository.findByEventId(Long.*parseLong*(eventId)))  
 .thenThrow(new RuntimeException("Custom exception message"));  
  
 // Act  
 try {  
 eventStoreService.fetchErrorsByEventId(eventId);  
  
 // Fail the test if no exception is thrown  
 *fail*("Expected CustomServiceException, but no exception was thrown.");  
 } catch (RuntimeException e) {  
 // Expected CustomServiceException, do nothing  
 }  
 }  
  
//------------------------------------------------------------------------------------------------------------------------  
  
 //Method 5: fetchNotifyDetailsByEventId  
  
 //Scenario 1: testFetchNotifyDetailsByEventId\_Success  
 @Test  
 public void testFetchNotifyDetailsByEventId\_Success() {  
 // Mock the eventId  
 String eventId = "123";  
  
 // Mock the eventNotifies  
 List<DmEventNotify> eventNotifies = Collections.*emptyList*();  
  
 // Stub the mock repository to return eventNotifies when findByEventId is called  
 *when*(dmEventNotifyRepository.findByEventId(Long.*parseLong*(eventId))).thenReturn(Optional.*of*(eventNotifies));  
  
 // Call the service method  
 Optional<EventResponse> result = eventStoreService.fetchNotifyDetailsByEventId(eventId);  
  
 // Assertions  
 *assertTrue*(result.isPresent()); // Check if the result is present (not empty)  
  
 EventResponse eventResponse = result.get(); // Get the actual EventResponse object  
  
 *assertNotNull*(eventResponse); // Check if the EventResponse is not null  
 *assertNotNull*(eventResponse.getEventDataResponse()); // Check if EventDataResponse is not null  
 *assertFalse*(eventResponse.getEventDataResponse().getEventNotifyResponses().isEmpty()); // Check if the list of EventNotifyResponses is not empty  
 }  
  
 //Scenario 2: testFetchNotifyDetailsByEventId\_EmptyEventResponse  
 @Test  
 public void testFetchNotifyDetailsByEventId\_EmptyEventResponse() {  
 // Mock the eventId  
 String eventId = "123"; // the variable eventId is being initialized  
  
 // Stub the mock repository to return an empty Optional when findByEventId is called  
 *when*(dmEventNotifyRepository.findByEventId(Long.*parseLong*(eventId))).thenReturn(Optional.*empty*());  
  
 // Act, Call the service method  
 Optional<EventResponse> result = eventStoreService.fetchNotifyDetailsByEventId(eventId);  
  
 // Assert  
 *assertFalse*(result.isPresent()); // Check if the result is not present (empty)  
 }  
  
 //Scenario 3: testFetchNotifyDetailsByEventId\_NotFound  
 @Test  
 public void testFetchNotifyDetailsByEventId\_NotFound() {  
 // Mock the eventId  
 String eventId = "123";  
  
 // Stub the mock, the repository to return an empty result when findByEventId is called  
 *when*(dmEventNotifyRepository.findByEventId(Long.*parseLong*(eventId))).thenReturn(Optional.*empty*());  
  
 // Call the service method  
 Optional<EventResponse> result = eventStoreService.fetchNotifyDetailsByEventId(eventId);  
  
 // Assertions  
 *assertFalse*(result.isPresent());  
 }  
  
 //Scenario 4: testFetchNotifyDetailsByEventId\_InternalServerError  
 @Test  
 public void testFetchNotifyDetailsByEventId\_InternalServerError() {  
 // Mock the eventId  
 String eventId = "123";  
  
 // Stub the mock, the repository to throw an exception when findByEventId is called  
 *when*(dmEventNotifyRepository.findByEventId(Long.*parseLong*(eventId))).thenThrow(new RuntimeException("Database error"));  
  
 // Call the service method  
 Optional<EventResponse> result = eventStoreService.fetchNotifyDetailsByEventId(eventId);  
  
 // Handle the exception or assert accordingly  
 *assertFalse*(result.isPresent()); // or handle the exception appropriately  
 }  
  
 //Scenario 5: testFetchNotifyDetailsByEventIdEmpty\_InvalidEventId  
 @Test  
 public void testFetchNotifyDetailsByEventIdEmpty\_InvalidEventId() {  
 // Mock the eventId  
 String eventId = "invalid"; // Invalid eventId  
  
 // Stub the mock repository to return an empty Optional when findByEventId is called  
 *when*(dmEventNotifyRepository.findByEventId(Long.*parseLong*(eventId))).thenReturn(Optional.*empty*());  
  
 // Act, Call the service method  
 Optional<EventResponse> result = eventStoreService.fetchNotifyDetailsByEventId(eventId);  
  
 // Assertions  
 *assertTrue*(result.isPresent());  
 *assertNotNull*(result.get().getEventDataResponse());  
 }  
  
 //Scenario 6: testFetchNotifyDetailsByEventIdEmpty\_EventIdIsNull  
 @Test  
 public void testFetchNotifyDetailsByEventIdEmpty\_EventIdIsNull() {  
 // Mock the eventId  
 String eventId = null; // Set eventId as null  
  
 // Stub the mock repository to return an empty Optional when findByEventId is called  
 *when*(dmEventNotifyRepository.findByEventId(Long.*parseLong*(eventId))).thenReturn(Optional.*empty*());  
  
 // Act, Call the service method  
 Optional<EventResponse> result = eventStoreService.fetchNotifyDetailsByEventId(eventId);  
  
 // Assertions  
 *assertFalse*(result.isPresent());  
 }  
  
 //Scenario7 : testFetchNotifyDetailsByEventIdEmpty\_EventIdIsEmpty  
 @Test  
 public void testFetchNotifyDetailsByEventIdEmpty\_EventIdIsEmpty() {  
 // Mock the eventId  
 String eventId = " "; // Set eventId as empty string  
  
 // Stub the mock repository to return an empty Optional when findByEventId is called  
 *when*(dmEventNotifyRepository.findByEventId(Long.*parseLong*(eventId))).thenReturn(Optional.*empty*());  
  
 // Act, Call the service method  
 Optional<EventResponse> result = eventStoreService.fetchNotifyDetailsByEventId(eventId);  
  
 // Assertions  
 *assertFalse*(result.isPresent());  
 }  
  
 //Scenario 8: fetchNotifyDetailsByEventId\_NoEventNotifies  
 @Test  
 public void fetchNotifyDetailsByEventId\_NoEventNotifies() {  
 // Mock the eventId  
 String eventId = "123";  
  
 // Stub the mock repository to return an empty Optional when findByEventId is called  
 *when*(dmEventNotifyRepository.findByEventId(Long.*parseLong*(eventId))).thenReturn(Optional.*of*(Collections.*emptyList*()));  
  
 // Act, Call the service method  
 Optional<EventResponse> result = eventStoreService.fetchNotifyDetailsByEventId(eventId);  
  
 // Assertions  
 *assertFalse*(result.isPresent());  
 }  
  
//-------------------------------------------------------------------------------------------------------------------------------------------  
  
 //Method 6: fetchNotifyDetailsByEventId  
 //Scenario 1: testFetchEventSummaryByDateRange\_Success  
 @Test  
 public void testFetchEventSummaryByDateRange\_Success() {  
 // Mock input parameters  
 String type = "reconcile";  
 String startDate = "2023-01-01";  
 String endDate = "2023-01-10";  
  
 // Mock data for the repository  
 LocalDate dateStart = LocalDate.*parse*(startDate);  
 LocalDate dateEnd = LocalDate.*parse*(endDate);  
 List<Map<String, Object>> dbEvents = new ArrayList<>();  
 List<Map<String, Object>> dbCounts = new ArrayList<>();  
  
 // Stub the repository methods  
 *when*(dmEventRepository.findEventsByStartDateAndEndDate(dateStart, dateEnd)).thenReturn(dbEvents);  
 *when*(dmEventRepository.findDocumentEventsByStartDateAndEndDate(dateStart, dateEnd)).thenReturn(dbEvents);  
 *when*(dmEventRepository.findTotalEventCountByStartDateAndEndDate(dateStart, dateEnd)).thenReturn(dbCounts);  
  
 // Call the service method  
 Optional<EventSummaryResponse> result = eventStoreService  
 .fetchEventSummaryByDateRange(type, startDate, endDate);  
  
 // Assertions  
 *assertTrue*(result.isPresent());  
 *assertNotNull*(result.get().getEventControlResponse());  
 }  
  
 //Scenario 2: testFetchEventSummaryByDateRange\_TypeIsEmpty  
 @Test  
 public void testFetchEventSummaryByDateRange\_TypeIsEmpty() {  
 // Mock input parameters  
 String type = ""; // Empty type  
 String startDate = "2023-01-01";  
 String endDate = "2023-01-10";  
  
 // Call the service method  
 Optional<EventSummaryResponse> result = eventStoreService  
 .fetchEventSummaryByDateRange(type, startDate, endDate);  
  
 // Assertions  
 *assertFalse*(result.isPresent()); // The result should be an empty Optional  
 }  
  
 //Scenario 3: testFetchEventSummaryByDateRange\_TypeIsNull  
 @Test  
 public void testFetchEventSummaryByDateRange\_TypeIsNull() {  
 // Mock input parameters  
 String type = null; // Type is null  
 String startDate = "2023-01-01";  
 String endDate = "2023-01-10";  
  
 // Call the service method  
 Optional<EventSummaryResponse> result = eventStoreService  
 .fetchEventSummaryByDateRange(type, startDate, endDate);  
  
 // Assertions  
 *assertFalse*(result.isPresent()); // The result should be an empty Optional  
 }  
  
 // Scenario 4: testFetchEventSummaryByDateRange\_StartDateIsEmpty  
 @Test  
 public void testFetchEventSummaryByDateRange\_StartDateIsEmpty() {  
 // Mock input parameters  
 String type = "reconcile";  
 String startDate = ""; // Empty startDate  
 String endDate = "2023-01-10";  
  
 // Call the service method  
 Optional<EventSummaryResponse> result = eventStoreService  
 .fetchEventSummaryByDateRange(type, startDate, endDate);  
  
 // Assertions  
 *assertFalse*(result.isPresent()); // The result should be an empty Optional  
 }  
  
 //Scenario 5: testFetchEventSummaryByDateRange\_StartDateIsNull  
 @Test  
 public void testFetchEventSummaryByDateRange\_StartDateIsNull() {  
 // Mock input parameters  
 String type = "reconcile";  
 String startDate = null; // StartDate is null  
 String endDate = "2023-01-10";  
  
 // Call the service method  
 Optional<EventSummaryResponse> result = eventStoreService  
 .fetchEventSummaryByDateRange(type, startDate, endDate);  
  
 // Assertions  
 *assertFalse*(result.isPresent()); // The result should be an empty Optional  
 }  
  
 //Scenario 6: testFetchEventSummaryByDateRange\_EndDateIsEmpty  
 @Test  
 public void testFetchEventSummaryByDateRange\_EndDateIsEmpty() {  
 // Mock input parameters  
 String type = "reconcile";  
 String startDate = "2023-01-01";  
 String endDate = ""; // Empty endDate  
  
 // Call the service method  
 Optional<EventSummaryResponse> result = eventStoreService  
 .fetchEventSummaryByDateRange(type, startDate, endDate);  
  
 // Assertions  
 *assertFalse*(result.isPresent()); // The result should be an empty Optional  
 }  
  
 //Scenario 7: testFetchEventSummaryByDateRange\_EndDateIsNull  
 @Test  
 public void testFetchEventSummaryByDateRange\_EndDateIsNull() {  
 // Mock input parameters  
 String type = "reconcile";  
 String startDate = "2023-01-01";  
 String endDate = null; // EndDate is null  
  
 // Call the service method  
 Optional<EventSummaryResponse> result = eventStoreService  
 .fetchEventSummaryByDateRange(type, startDate, endDate);  
  
 // Assertions  
 *assertFalse*(result.isPresent()); // The result should be an empty Optional  
 }  
  
 //Scenario 8: testFetchEventSummaryByDateRange\_AllInputsAreEmpty  
 @Test  
 public void testFetchEventSummaryByDateRange\_AllInputsAreEmpty() {  
 // Mock input parameters  
 String type = ""; // Empty type  
 String startDate = ""; // Empty startDate  
 String endDate = ""; // Empty endDate  
  
 // Call the service method  
 Optional<EventSummaryResponse> result = eventStoreService  
 .fetchEventSummaryByDateRange(type, startDate, endDate);  
  
 // Assertions  
 *assertFalse*(result.isPresent()); // The result should be an empty Optional  
 }  
  
 //Scenario 9: testFetchEventSummaryByDateRange\_AllInputsAreNull  
 @Test  
 public void testFetchEventSummaryByDateRange\_AllInputsAreNull() {  
 // Mock input parameters  
 String type = null; // Null type  
 String startDate = null; // Null startDate  
 String endDate = null; // Null endDate  
  
 // Call the service method  
 Optional<EventSummaryResponse> result = eventStoreService  
 .fetchEventSummaryByDateRange(type, startDate, endDate);  
  
 // Assertions  
 *assertFalse*(result.isPresent()); // The result should be an empty Optional  
 }  
  
 //Scenario 10: testFetchEventSummaryByDateRange\_InvalidDateRange  
 @Test  
 public void testFetchEventSummaryByDateRange\_InvalidDateRange() {  
 // Mock input parameters  
 String type = "reconcile";  
 String startDate = "2023-01-10"; // End date is before start date  
 String endDate = "2023-01-01";  
  
 // Call the service method  
 Optional<EventSummaryResponse> result = eventStoreService  
 .fetchEventSummaryByDateRange(type, startDate, endDate);  
  
 // Assertions  
 *assertFalse*(result.isPresent()); // The result should be an empty Optional  
 }  
  
 //Scenario 11: testFetchEventSummaryByDateRange\_EmptyDBEvents  
 @Test  
 public void testFetchEventSummaryByDateRange\_EmptyDBEvents() {  
 // Mock input parameters  
 String type = "reconcile";  
 String startDate = "2023-01-01";  
 String endDate = "2023-01-10";  
  
 // Mock data for the repository  
 LocalDate dateStart = LocalDate.*parse*(startDate);  
 LocalDate dateEnd = LocalDate.*parse*(endDate);  
 List<Map<String, Object>> emptyDBEvents = new ArrayList<>();  
 List<Map<String, Object>> dbCounts = new ArrayList<>();  
  
 // Stub the repository methods  
 *when*(dmEventRepository.findEventsByStartDateAndEndDate(dateStart, dateEnd)).thenReturn(emptyDBEvents);  
 *when*(dmEventRepository.findDocumentEventsByStartDateAndEndDate(dateStart, dateEnd)).thenReturn(emptyDBEvents);  
 *when*(dmEventRepository.findTotalEventCountByStartDateAndEndDate(dateStart, dateEnd)).thenReturn(dbCounts);  
  
 // Call the service method  
 Optional<EventSummaryResponse> result = eventStoreService  
 .fetchEventSummaryByDateRange(type, startDate, endDate);  
  
 // Assertions  
 *assertTrue*(result.isPresent());  
 *assertNotNull*(result.get().getEventControlResponse());  
 *assertEquals*(0, result.get().getEventControlResponse().getEventControlDatas().size());  
 }  
  
 //Scenario 12: testFetchEventSummaryByDateRange\_EmptyDBCounts  
 @Test  
 public void testFetchEventSummaryByDateRange\_EmptyDBCounts() {  
 // Mock input parameters  
 String type = "reconcile";  
 String startDate = "2023-01-01";  
 String endDate = "2023-01-10";  
  
 // Mock data for the repository  
 LocalDate dateStart = LocalDate.*parse*(startDate);  
 LocalDate dateEnd = LocalDate.*parse*(endDate);  
 List<Map<String, Object>> dbEvents = new ArrayList<>();  
 List<Map<String, Object>> emptyDBCounts = new ArrayList<>();  
  
 // Stub the repository methods  
 *when*(dmEventRepository.findEventsByStartDateAndEndDate(dateStart, dateEnd)).thenReturn(dbEvents);  
 *when*(dmEventRepository.findDocumentEventsByStartDateAndEndDate(dateStart, dateEnd)).thenReturn(dbEvents);  
 *when*(dmEventRepository.findTotalEventCountByStartDateAndEndDate(dateStart, dateEnd)).thenReturn(emptyDBCounts);  
  
 // Call the service method  
 Optional<EventSummaryResponse> result = eventStoreService  
 .fetchEventSummaryByDateRange(type, startDate, endDate);  
  
 // Assertions  
 *assertTrue*(result.isPresent());  
 *assertNotNull*(result.get().getEventControlResponse());  
 *assertEquals*(0, result.get().getEventControlResponse().getEventControlDatas().size());  
 }  
  
 //Scenario 13: testFetchEventSummaryByDateRange\_TypeIsInvalid  
 @Test  
 public void testFetchEventSummaryByDateRange\_TypeIsInvalid() {  
 // Mock input parameters  
 String type = "invalidType"; // Type is something other than "reconcile"  
 String startDate = "2023-01-01";  
 String endDate = "2023-01-10";  
  
 // Call the service method  
 Optional<EventSummaryResponse> result = eventStoreService  
 .fetchEventSummaryByDateRange(type, startDate, endDate);  
  
 // Assertions  
 *assertFalse*(result.isPresent()); // Expecting an empty Optional  
 }  
  
 //Scenario 14: testFetchEventSummaryByDateRange\_StartDateAndEndDateBothSame  
 @Test  
 public void testFetchEventSummaryByDateRange\_StartDateAndEndDateBothSame() {  
 // Mock input parameters  
 String type = "reconcile";  
 String startDate = "2023-01-10";  
 String endDate = "2023-01-10"; // Same as startDate  
  
 // Mock the behavior of dmEventRepository  
 *when*(dmEventRepository.findEventsByStartDateAndEndDate(*any*(LocalDate.class), *any*(LocalDate.class)))  
 .thenReturn(new ArrayList<>());  
 *when*(dmEventRepository.findDocumentEventsByStartDateAndEndDate(*any*(LocalDate.class), *any*(LocalDate.class)))  
 .thenReturn(new ArrayList<>());  
 *when*(dmEventRepository.findTotalEventCountByStartDateAndEndDate(*any*(LocalDate.class), *any*(LocalDate.class)))  
 .thenReturn(new ArrayList<>());  
  
 // Call the service method  
 Optional<EventSummaryResponse> result = eventStoreService  
 .fetchEventSummaryByDateRange(type, startDate, endDate);  
  
 // Assertions  
 *assertTrue*(result.isPresent()); // Expecting a non-empty Optional  
 *assertNotNull*(result.get());  
 *assertNotNull*(result.get().getEventControlResponse());  
 }  
  
 //Scenario 15: testFetchEventSummaryByDateRange\_NoEventSummary  
 @Test  
 public void testFetchEventSummaryByDateRange\_NoEventSummary() {  
 // Mock input parameters  
 String type = "reconcile";  
 String startDate = "2023-01-01";  
 String endDate = "2023-01-10";  
  
 // Mock data for the repository  
 LocalDate dateStart = LocalDate.*parse*(startDate);  
 LocalDate dateEnd = LocalDate.*parse*(endDate);  
 List<Map<String, Object>> dbEvents = new ArrayList<>();  
 List<Map<String, Object>> dbCounts = new ArrayList<>();  
  
 // Stub the repository methods  
 *when*(dmEventRepository.findEventsByStartDateAndEndDate(dateStart, dateEnd)).thenReturn(dbEvents);  
 *when*(dmEventRepository.findDocumentEventsByStartDateAndEndDate(dateStart, dateEnd)).thenReturn(dbEvents);  
 *when*(dmEventRepository.findTotalEventCountByStartDateAndEndDate(dateStart, dateEnd)).thenReturn(dbCounts);  
  
 // Call the service method  
 Optional<EventSummaryResponse> result = eventStoreService  
 .fetchEventSummaryByDateRange(type, startDate, endDate);  
  
 // Assertions  
 *assertTrue*(result.isPresent());  
 *assertNotNull*(result.get().getEventControlResponse());  
 *assertNotNull*(result.get().getEventControlResponse().getEventControlDatas());  
 *assertTrue*(result.get().getEventControlResponse().getEventControlDatas().isEmpty());  
 }  
  
 //Scenario 16: testFetchEventSummaryByDateRange\_InternalServerError  
 @Test  
 public void testFetchEventSummaryByDateRange\_InternalServerError() {  
 // Mock input parameters  
 String type = "reconcile";  
 String startDate = "2023-01-01";  
 String endDate = "2023-01-10";  
  
 // Mock data for the repository  
 LocalDate dateStart = LocalDate.*parse*(startDate);  
 LocalDate dateEnd = LocalDate.*parse*(endDate);  
  
 // Stub the repository methods to throw an exception  
 *when*(dmEventRepository.findEventsByStartDateAndEndDate(dateStart, dateEnd)).thenThrow(new RuntimeException("Internal Server Error"));  
  
 // Call the service method  
 *assertThrows*(RuntimeException.class, () -> eventStoreService  
 .fetchEventSummaryByDateRange(type, startDate, endDate));  
 }  
  
 //Scenario 17: testFetchEventSummaryByDateRange\_NoEventControlResponse  
 @Test  
 public void testFetchEventSummaryByDateRange\_NoEventControlResponse() {  
 // Mock input parameters  
 String type = "reconcile";  
 String startDate = "2023-01-01";  
 String endDate = "2023-01-10";  
  
 // Mock data for the repository  
 LocalDate dateStart = LocalDate.*parse*(startDate);  
 LocalDate dateEnd = LocalDate.*parse*(endDate);  
 List<Map<String, Object>> dbEvents = new ArrayList<>();  
 List<Map<String, Object>> dbCounts = new ArrayList<>();  
  
 // Stub the repository methods  
 *when*(dmEventRepository.findEventsByStartDateAndEndDate(dateStart, dateEnd)).thenReturn(dbEvents);  
 *when*(dmEventRepository.findDocumentEventsByStartDateAndEndDate(dateStart, dateEnd)).thenReturn(dbEvents);  
 *when*(dmEventRepository.findTotalEventCountByStartDateAndEndDate(dateStart, dateEnd)).thenReturn(dbCounts);  
  
 // Call the service method  
 Optional<EventSummaryResponse> result = eventStoreService  
 .fetchEventSummaryByDateRange(type, startDate, endDate);  
  
 // Assertions  
 *assertTrue*(result.isPresent());  
 *assertNull*(result.get().getEventControlResponse()); // No event control data is retrieved  
 }  
  
 //Scenario 18: testFetchEventSummaryByDateRange\_NotFound  
 @Test  
 public void testFetchEventSummaryByDateRange\_NotFound() {  
 // Mock input parameters  
 String type = "reconcile";  
 String startDate = "2023-01-01";  
 String endDate = "2023-01-10";  
  
 // Mock data for the repository  
 LocalDate dateStart = LocalDate.*parse*(startDate);  
 LocalDate dateEnd = LocalDate.*parse*(endDate);  
  
 // Stub the repository methods to return empty lists  
 *when*(dmEventRepository.findEventsByStartDateAndEndDate(dateStart, dateEnd)).thenReturn(new ArrayList<>());  
 *when*(dmEventRepository.findDocumentEventsByStartDateAndEndDate(dateStart, dateEnd)).thenReturn(new ArrayList<>());  
 *when*(dmEventRepository.findTotalEventCountByStartDateAndEndDate(dateStart, dateEnd)).thenReturn(new ArrayList<>());  
  
 // Call the service method  
 Optional<EventSummaryResponse> result = eventStoreService  
 .fetchEventSummaryByDateRange(type, startDate, endDate);  
  
 // Assertions  
 *assertTrue*(result.isEmpty()); // To simulate a "Not Found" scenario.  
 }  
//----------------------------------------------------------------------------------------------------------------------------  
  
 //Method 7: fetchDocumentByInputId  
 //Scenario 1: shouldFetchDocumentByInputIdSuccessfullyWhenDocumentIsFound  
 @Test  
 void shouldFetchDocumentByInputIdSuccessfullyWhenDocumentIsFound() {  
 // Create a valid customer account UUID and input ID  
 UUID customerAccountUuid = UUID.*randomUUID*();  
 String inputId = "my-input-id";  
  
 // Create a valid document object  
 Document document = new Document();  
 document.setCustomerAccountUuid(customerAccountUuid);  
 document.setInputId(inputId);  
  
 // Simulate the successful retrieval of the document and method returns a valid Document object  
 *when*(documentRepository.findByCustomerAccountUuidAndInputId(customerAccountUuid, inputId))  
 .thenReturn(Optional.*of*(document));  
  
 // Call the fetchDocumentByInputId() method  
 Optional<EventResponse> eventResponse = eventStoreService  
 .fetchDocumentByInputId(customerAccountUuid, inputId);  
  
 // Assert the results  
 *assertTrue*(eventResponse.isPresent());  
 *assertEquals*(HttpStatus.*OK*, eventResponse.get().getCode());  
 }  
  
 private Optional<EventResponse> fetchDocumentByInputId(UUID customerAccountUuid, String inputId) {  
 return null;  
 }  
  
 //Scenario 2: shouldReturnEmptyOptionalObjectWhenDocumentIsNotFound  
 @Test  
 void shouldReturnEmptyOptionalObjectWhenDocumentIsNotFound() {  
 // Create a valid customer account UUID and input ID  
 UUID customerAccountUuid = UUID.*randomUUID*();  
 String inputId = "my-input-id";  
  
 // Simulate the unsuccessful retrieval of the document  
 *when*(documentRepository.findByCustomerAccountUuidAndInputId(customerAccountUuid, inputId))  
 .thenReturn(Optional.*empty*());  
  
 // Call the fetchDocumentByInputId() method  
 Optional<EventResponse> eventResponse = eventStoreService  
 .fetchDocumentByInputId(customerAccountUuid, inputId);  
  
 // Assert the results  
 *assertFalse*(eventResponse.isPresent());  
 }  
  
 //Scenario 3: shouldThrowExceptionWhenCustomerAccountUuidIsInvalid  
 @Test  
 void shouldThrowExceptionWhenCustomerAccountUuidIsInvalid() {  
 // Create an invalid customer account UUID  
 UUID customerAccountUuid = UUID.*randomUUID*();  
  
 // Call the fetchDocumentByInputId() method  
 Optional<EventResponse> eventResponse = eventStoreService  
 .fetchDocumentByInputId(customerAccountUuid, "my-input-id");  
  
 // Assert the results (If the code throws the expected exception, the assertion will pass. If the code does not throw the expected exception, the assertion will fail)  
 *assertThrows*(IllegalArgumentException.class, eventResponse::get);  
 }  
  
 //Scenario 4: shouldThrowExceptionWhenInputIdIsInvalid  
 @Test  
 void shouldThrowExceptionWhenInputIdIsInvalid() {  
 // Create an invalid input ID  
 String inputId = null;  
  
 // Call the fetchDocumentByInputId() method  
 Optional<EventResponse> eventResponse = eventStoreService  
 .fetchDocumentByInputId(UUID.*randomUUID*(), inputId);  
  
 // Assert the results  
 *assertThrows*(IllegalArgumentException.class, eventResponse::get);  
 }  
  
 //Scenario 5: shouldFetchDocumentByInputIdSuccessfullyWhenDocumentIsDeleted  
 @Test  
 void shouldFetchDocumentByInputIdSuccessfullyWhenDocumentIsDeleted() {  
 // Create a valid customer account UUID and input ID  
 UUID customerAccountUuid = UUID.*randomUUID*();  
 String inputId = "my-input-id";  
  
 // Create a valid document object and set its deleted flag to true  
 Document document = new Document();  
 document.setCustomerAccountUuid(customerAccountUuid);  
 document.setInputId(inputId);  
 document.setDeleted(true);  
  
 // Simulate the successful retrieval of the document  
 *when*(documentRepository.findByCustomerAccountUuidAndInputId(customerAccountUuid, inputId))  
 .thenReturn(Optional.*of*(document));  
  
 // Call the fetchDocumentByInputId() method  
 Optional<EventResponse> eventResponse = eventStoreService  
 .fetchDocumentByInputId(customerAccountUuid, inputId);  
  
 // Assert the results  
 *assertTrue*(eventResponse.isPresent());  
 *assertEquals*(HttpStatus.*OK*, eventResponse.get().getCode());  
 }  
  
 //Scenario 6: shouldReturnEmptyOptionalObjectWhenInputIdIsNull  
 @Test  
 void shouldReturnEmptyOptionalObjectWhenInputIdIsNull() {  
 // Create a valid customer account UUID  
 UUID customerAccountUuid = UUID.*randomUUID*();  
  
 // Call the fetchDocumentByInputId() method with a null input ID  
 Optional<EventResponse> eventResponse = eventStoreService  
 .fetchDocumentByInputId(customerAccountUuid, null);  
  
 // Assert the results  
 *assertFalse*(eventResponse.isPresent());  
 }  
  
 //Scenario 7: shouldReturnEmptyOptionalObjectWhenCustomerAccountUuidIsNull  
 @Test  
 void shouldReturnEmptyOptionalObjectWhenCustomerAccountUuidIsNull() {  
 // Call the fetchDocumentByInputId() method with a null customer account UUID  
 Optional<EventResponse> eventResponse = eventStoreService.fetchDocumentByInputId(null, "my-input-id");  
  
 // Assert the results (assertion should pass if the eventResponse object is not present)  
 *assertFalse*(eventResponse.isPresent());  
 }  
  
 //Scenario 8: shouldReturn500StatusCodeWhenDocumentRepositoryThrowsException  
 @Test  
 void shouldReturn500StatusCodeWhenDocumentRepositoryThrowsException() {  
 // Create a valid customer account UUID and input ID  
 UUID customerAccountUuid = UUID.*randomUUID*();  
 String inputId = "my-input-id";  
  
 // Simulate the `documentRepository.findByCustomerAccountUuidAndInputId()` method throwing an exception  
 *when*(documentRepository.findByCustomerAccountUuidAndInputId(customerAccountUuid, inputId))  
 .thenThrow(new RuntimeException());  
  
 // Call the fetchDocumentByInputId() method  
 Optional<EventResponse> eventResponse = eventStoreService.fetchDocumentByInputId(customerAccountUuid, inputId);  
  
 // Assert the results  
 *assertFalse*(eventResponse.isPresent());  
 *assertEquals*(HttpStatus.*INTERNAL\_SERVER\_ERROR*, eventResponse.get().getCode());  
 }  
  
 //Scenario 9: shouldFetchDocumentByInputIdSuccessfullyWhenDocumentIsExpired  
 @Test  
 void shouldFetchDocumentByInputIdSuccessfullyWhenDocumentIsExpired() {  
 // Create a valid customer account UUID and input ID  
 UUID customerAccountUuid = UUID.*randomUUID*();  
 String inputId = "my-input-id";  
  
 // Create a valid document object and set its expired flag to true  
 Document document = new Document();  
 document.setCustomerAccountUuid(customerAccountUuid);  
 document.setInputId(inputId);  
 document.setExpired(true);  
  
 // Simulate the successful retrieval of the document  
 *when*(documentRepository.findByCustomerAccountUuidAndInputId(customerAccountUuid, inputId))  
 .thenReturn(Optional.*of*(document));  
  
 // Call the fetchDocumentByInputId() method  
 Optional<EventResponse> eventResponse = eventStoreService.fetchDocumentByInputId(customerAccountUuid, inputId);  
  
 // Assert the results  
 *assertTrue*(eventResponse.isPresent());  
 *assertEquals*(HttpStatus.*OK*, eventResponse.get().getCode());  
 }  
  
 //Scenario 10: shouldReturn404StatusCodeWhenDocumentIsNotFound  
 @Test  
 void shouldReturn404StatusCodeWhenDocumentIsNotFound() {  
 // Create a valid customer account UUID and input ID  
 UUID customerAccountUuid = UUID.*randomUUID*();  
 String inputId = "my-input-id";  
  
 // Simulate the unsuccessful retrieval of the document  
 *when*(documentRepository.findByCustomerAccountUuidAndInputId(customerAccountUuid, inputId))  
 .thenReturn(Optional.*empty*());  
  
 // Call the fetchDocumentByInputId() method  
 Optional<EventResponse> eventResponse = eventStoreService.fetchDocumentByInputId(customerAccountUuid, inputId);  
  
 // Assert the results  
 *assertFalse*(eventResponse.isPresent());  
 *assertEquals*(HttpStatus.*NOT\_FOUND*, eventResponse.get().getCode());  
 }  
  
//----------------------------------------------------------------------------------------------------  
  
 //Method 8:  
 //Scenario 1:  
}**