**/Method 1: Api to interact with a database to add Event  
//Scenario 1: testSaveEventWithValidInput  
@Test  
public void testSaveEventWithValidInput() throws Exception {  
  
 // Create a sample valid event request  
 EventRequest eventRequest = new EventRequest();  
 eventRequest.setCorrelationId("123");  
  
 // Create a sample response for the service layer  
 EventResponse eventResponse = new EventResponse();  
 eventResponse.setCode(HttpStatus.*CREATED*.value());  
 eventResponse.setMessage("Event added successfully");  
 // You can set other properties of the eventResponse as needed for your test.  
  
 // Mock the service layer to return the sample response  
 *when*(DocumentGeneratorEventStoreService.*saveEvent*(*any*(EventRequest.class)))  
 .thenReturn(Optional.*of*(eventResponse));  
  
 // Perform the controller method invocation  
 ResponseEntity<EventResponse> responseEntity = eventStoreController.saveEvent(eventRequest);  
  
 // Verify the response status code  
 *assertEquals*(HttpStatus.*CREATED*, responseEntity.getStatusCode());  
  
 // Verify the response body  
 EventResponse responseBody = responseEntity.getBody();  
 *assertEquals*(HttpStatus.*CREATED*.value(), responseBody.getCode());  
 *assertEquals*("Event added successfully", responseBody.getMessage());  
}  
  
//Scenario 2: testSaveEventWithAlreadyExist  
@Test  
public void testSaveEventWithAlreadyExist() throws Exception {  
  
 // Create a sample event request with duplicate data  
 EventRequest eventRequest = new EventRequest();  
 eventRequest.setCorrelationId("123");  
  
 // Create a sample response for the service layer indicating that the event already exists  
 EventResponse eventResponse = new EventResponse();  
 eventResponse.setCode(HttpStatus.*OK*.value());  
 eventResponse.setMessage("Event already exists");  
  
 // Mock the service layer to return the sample response  
 *when*(DocumentGeneratorEventStoreService.*saveEvent*(*any*(EventRequest.class)))  
 .thenReturn(Optional.*of*(eventResponse));  
  
 // Perform the controller method invocation  
 ResponseEntity<EventResponse> responseEntity = eventStoreController.saveEvent(eventRequest);  
  
 // Verify the response status code  
 *assertEquals*(HttpStatus.*OK*, responseEntity.getStatusCode());  
  
 // Verify the response body  
 EventResponse responseBody = responseEntity.getBody();  
 *assertEquals*(HttpStatus.*OK*.value(), responseBody.getCode());  
 *assertEquals*("Event already exists", responseBody.getMessage());  
}  
  
//Scenario 3: testSaveEventWithMissingRequiredFields  
@Test  
public void testSaveEventWithMissingRequiredFields() throws Exception {  
 // Create an event request with missing required fields  
 // Missing correlationId and other required fields.  
 EventRequest eventRequest = new EventRequest();  
  
 // Create a sample response for the service layer  
 EventResponse eventResponse = new EventResponse();  
  
 *when*(DocumentGeneratorEventStoreService.*saveEvent*(*any*(EventRequest.class)))  
 .thenReturn(Optional.*of*(eventResponse));  
  
 // Perform the controller method invocation  
 ResponseEntity<EventResponse> responseEntity = eventStoreController.saveEvent(eventRequest);  
  
 // Verify the response status code  
 *assertEquals*(HttpStatus.*BAD\_REQUEST*, responseEntity.getStatusCode());  
}  
  
//Scenario 4: testSaveEventWithInvalidInputData  
@Test  
public void testSaveEventWithInvalidInputData() throws Exception {  
 // Create an event request with invalid input data (e.g., missing required fields)  
 EventRequest eventRequest = new EventRequest();  
 eventRequest.setCorrelationId(null);  
 // Missing required fields.  
  
 // Create a sample response for the service layer (should not be called)  
 EventResponse eventResponse = new EventResponse();  
  
 *when*(DocumentGeneratorEventStoreService.*saveEvent*(*any*(EventRequest.class)))  
 .thenReturn(Optional.*of*(eventResponse));  
  
 // Perform the controller method invocation  
 ResponseEntity<EventResponse> responseEntity = eventStoreController.saveEvent(eventRequest);  
  
 // Verify the response status code  
 *assertEquals*(HttpStatus.*BAD\_REQUEST*, responseEntity.getStatusCode());  
}  
  
//Scenario 5: testSaveEventWithDatabaseError  
@Test  
public void testSaveEventWithDatabaseError() throws Exception {  
 // Create a sample event request  
 EventRequest eventRequest = new EventRequest();  
 eventRequest.setCorrelationId("123");  
  
 // Mock the service layer to throw a DataAccessException, simulating a database error  
 *when*(DocumentGeneratorEventStoreService.*saveEvent*(*any*(EventRequest.class)))  
 .thenThrow(new DataAccessException("Database error") {  
 });  
 // Perform the controller method invocation  
 ResponseEntity<EventResponse> responseEntity = eventStoreController.saveEvent(eventRequest);  
 // Verify the response status code  
 *assertEquals*(HttpStatus.*INTERNAL\_SERVER\_ERROR*, responseEntity.getStatusCode());  
 // Verify error message  
 EventResponse responseBody = responseEntity.getBody();  
 Assertions.*assertNotNull*(responseBody);  
 *assertEquals*("Database error", responseBody.getMessage());  
}**

**------------------------------------------------------------------------------------------------------**

**// Method 3 "API to retrieve event by eventId"  
  
 //Scenario 1: testRetrieveEventByEventId\_Success  
 @Test  
 public void testRetrieveEventByEventId\_Success() throws Exception {  
 // Set a sample event Response  
 EventResponse eventResponse = new EventResponse();  
 eventResponse.setCorrelationId("123");  
 eventResponse.setApplicationLabel("SampleApp");  
 eventResponse.setCode(200);  
 eventResponse.setStatus("Success");  
 eventResponse.setMessage("Event retrieved successfully");  
 eventResponse.setEventDataResponse(null);  
 eventResponse.setTime("2023-09-23T10:00:00Z");  
 eventResponse.setPath("/event/123");  
 eventResponse.setMethod("GET");  
  
 // Stub the Mock the behavior of documentGeneratorEventStoreService.fetchEventByEventId  
 *when*(documentGeneratorEventStoreService.fetchEventByEventId(*anyString*()))  
 .thenReturn(Optional.*of*(eventResponse));  
  
 // Execute and Assert  
 mockMvc.perform(*get*("/event/{eventId}", "123"))  
 .andExpect(*status*().isOk())  
 .andExpect(MockMvcResultMatchers.*content*().contentType(MediaType.*APPLICATION\_JSON*))  
 .andExpect(MockMvcResultMatchers.*jsonPath*("$.correlationId").value("123"))  
 .andExpect(MockMvcResultMatchers.*jsonPath*("$.applicationLabel").value("SampleApp"))  
 .andExpect(MockMvcResultMatchers.*jsonPath*("$.code").value(200L))  
 .andExpect(MockMvcResultMatchers.*jsonPath*("$.status").value("Success"))  
 .andExpect(MockMvcResultMatchers.*jsonPath*("$.message").value("Event retrieved successfully"))  
 .andExpect(MockMvcResultMatchers.*jsonPath*("$.eventDataResponse").doesNotExist()) // Adjust as needed  
 .andExpect(MockMvcResultMatchers.*jsonPath*("$.time").value("2023-09-23T10:00:00Z"))  
 .andExpect(MockMvcResultMatchers.*jsonPath*("$.path").value("/event/123"))  
 .andExpect(MockMvcResultMatchers.*jsonPath*("$.method").value("GET"));  
  
 // Verify that documentGeneratorEventStoreService.fetchEventByEventId was called with the correct argument  
 *verify*(documentGeneratorEventStoreService, *times*(1)).fetchEventByEventId("123");  
 }  
  
 //Scenario 2: testRetrieveEventByEventId\_NotFound  
 @Test  
 public void testRetrieveEventByEventId\_NotFound() throws Exception {  
 // Provide a non-existent eventId  
 String eventId = "nonExistentEventId";  
  
 // Stub the Mock behavior of documentGeneratorEventStoreService.fetchEventByEventId  
 *when*(documentGeneratorEventStoreService.fetchEventByEventId(eventId))  
 .thenReturn(Optional.*empty*());  
  
 // Execute and Assert  
 mockMvc.perform(*get*("/event/{eventId}", eventId)  
 .contentType(MediaType.*APPLICATION\_JSON*))  
 .andExpect(*status*().isNotFound()); // Expect HTTP status 404 (Not Found)  
  
 // Verify that documentGeneratorEventStoreService.fetchEventByEventId was called with the correct argument  
 *verify*(documentGeneratorEventStoreService, *times*(1)).fetchEventByEventId(eventId);  
 }  
  
 //Scenario 3: testRetrieveEventByEventId\_EmptyEventId  
 @Test  
 public void testRetrieveEventByEventId\_EmptyEventId() throws Exception {  
 // Provide an empty eventId  
 String eventId = "";  
  
 // Execute and Assert  
 mockMvc.perform(*get*("/event/{eventId}", eventId)  
 .contentType(MediaType.*APPLICATION\_JSON*))  
 .andExpect(*status*().isBadRequest()) // Expect HTTP status 400 (Bad Request)  
 .andExpect((ResultMatcher) *content*().contentType(MediaType.*APPLICATION\_JSON*))  
 .andExpect((ResultMatcher) *jsonPath*("$.error").value("Invalid input")) // Verify the error message  
 .andExpect((ResultMatcher) *jsonPath*("$.message").value("Event ID must not be empty")); // Verify the specific error message  
 }  
  
 //Scenario 4: testRetrieveEventByEventId\_NullEventId  
 @Test  
 public void testRetrieveEventByEventId\_NullEventId() throws Exception {  
 // Provide a null eventId  
 String eventId = null;  
  
 // Execute and Assert  
 mockMvc.perform(*get*("/event/{eventId}", eventId)  
 .contentType(MediaType.*APPLICATION\_JSON*))  
 .andExpect(*status*().isBadRequest()) // Expect HTTP status 400 (Bad Request)  
 .andExpect((ResultMatcher) *content*().contentType(MediaType.*APPLICATION\_JSON*))  
 .andExpect((ResultMatcher) *jsonPath*("$.error").value("Invalid input")) // Verify the error message  
 .andExpect((ResultMatcher) *jsonPath*("$.message").value("Event ID must not be null")); // Verify the specific error message  
 }  
  
 //Scenario 5: testRetrieveEventByEventId\_InvalidCharacters  
 @Test  
 public void testRetrieveEventByEventId\_InvalidCharacters() throws Exception {  
 // Provide an eventId with invalid characters (e.g., special characters)  
 String eventId = "@#%^$\*&"; //Invalid characters in eventId  
  
 // Execute and Assert  
 mockMvc.perform(*get*("/event/{eventId}", eventId)  
 .contentType(MediaType.*APPLICATION\_JSON*))  
 .andExpect(*status*().isBadRequest()) // Expect HTTP status 400 (Bad Request)  
 .andExpect((ResultMatcher) *content*().contentType(MediaType.*APPLICATION\_JSON*))  
 .andExpect((ResultMatcher) *jsonPath*("$.error").value("Invalid input")) // Verify the error message  
 .andExpect((ResultMatcher) *jsonPath*("$.message").value("Event ID contains invalid characters")); // Verify the specific error message  
 }  
  
 //Scenario 6: testRetrieveEventByEventId\_ErrorHandling  
 @Test  
 public void testRetrieveEventByEventId\_ErrorHandling() throws Exception {  
 // Provide a valid eventId  
 String eventId = "123";  
  
 // Simulate an exception thrown during event retrieval  
 *when*(documentGeneratorEventStoreService.fetchEventByEventId(eventId))  
 .thenThrow(new RuntimeException("Internal server error"));  
  
 // Execute and Assert  
 mockMvc.perform(*get*("/event/123", eventId)  
 .contentType(MediaType.*APPLICATION\_JSON*))  
 .andExpect(*status*().isInternalServerError()) // Expect HTTP status 500 (Internal Server Error)  
 .andExpect((ResultMatcher) *content*().contentType(MediaType.*APPLICATION\_JSON*))  
 .andExpect((ResultMatcher) *jsonPath*("$.error").value("Internal Server Error")) // Verify the error message  
 .andExpect((ResultMatcher) *jsonPath*("$.message").value("An error occurred during event retrieval")); // Verify the specific error message  
  
 // Verify that the service method was called once with the provided eventId  
 *verify*(documentGeneratorEventStoreService, *times*(1)).fetchEventByEventId(eventId);  
 }  
  
//------------------------------------------------------------------------------------------------------------------------  
  
 //Method 4: API to retrieve error details by eventId  
 //Scenario 1: testRetrieveErrorsResponseByEventId\_Success  
 @Test  
 public void testRetrieveErrorsResponseByEventId\_Success() throws Exception {  
 // Sample event response for testing  
 EventResponse EventResponse = new EventResponse();  
 EventResponse.setCorrelationId("123");  
 EventResponse.setApplicationLabel("SampleApp");  
 EventResponse.setCode(200);  
 EventResponse.setStatus("Success");  
 EventResponse.setMessage("Error retrieved successfully");  
 EventResponse.setEventDataResponse(null);  
 EventResponse.setTime("2023-09-23T10:00:00Z");  
 EventResponse.setPath("/event/123/errors");  
 EventResponse.setMethod("GET");  
  
 // Mock the service to return the sample event response  
 *when*(documentGeneratorEventStoreService.fetchErrorsByEventId("123"))  
 .thenReturn(Optional.*of*(EventResponse));  
  
 // Perform the GET request  
 mockMvc.perform(*get*("/events/123/errors"))  
 .andExpect(*status*().isOk())  
 .andExpect(MockMvcResultMatchers.*content*().contentType(MediaType.*APPLICATION\_JSON*))  
 .andExpect(MockMvcResultMatchers.*jsonPath*("$.correlationId").value("123"))  
 .andExpect(MockMvcResultMatchers.*jsonPath*("$.applicationLabel").value("SampleApp"))  
 .andExpect(MockMvcResultMatchers.*jsonPath*("$.code").value(200L))  
 .andExpect(MockMvcResultMatchers.*jsonPath*("$.status").value("Success"))  
 .andExpect(MockMvcResultMatchers.*jsonPath*("$.message").value("Error retrieved successfully"))  
 .andExpect(MockMvcResultMatchers.*jsonPath*("$.eventDataResponse").doesNotExist())  
 .andExpect(MockMvcResultMatchers.*jsonPath*("$.time").value("2023-09-23T10:00:00Z"))  
 .andExpect(MockMvcResultMatchers.*jsonPath*("$.path").value("/event/123/errors"))  
 .andExpect(MockMvcResultMatchers.*jsonPath*("$.method").value("GET"));  
 }  
  
 //Scenario 2: testRetrieveErrorsResponseByEventId\_InvalidEventId  
 @Test  
 public void testRetrieveErrorsResponseByEventId\_InvalidEventId() throws Exception {  
 // Mock the service to return an empty Optional (indicating that the eventId does not exist)  
 *when*(documentGeneratorEventStoreService.fetchErrorsByEventId("invalidId"))  
 .thenReturn(Optional.*empty*());  
  
 // Perform the GET request with an invalid eventId  
 mockMvc.perform(*get*("/events/invalidId/errors"))  
 .andExpect(*status*().isNotFound()) // Verify that the response status code is 404 (Not Found)  
 .andExpect(MockMvcResultMatchers.*content*().contentType(MediaType.*APPLICATION\_JSON*))  
 .andExpect((ResultMatcher) *jsonPath*("$.error").value("Event not found")) // Verify an appropriate error message  
 .andExpect((ResultMatcher) *jsonPath*("$.message").value("The specified event ID 'invalidId' does not exist."));  
 }  
  
 //Scenario 3: testRetrieveErrorsResponseByEventId\_NotFound  
 @Test  
 public void testRetrieveErrorsResponseByEventId\_NotFound() throws Exception {  
 // Mock the service to return an empty Optional, simulating that the event was not found  
 *when*(documentGeneratorEventStoreService.fetchErrorsByEventId(*anyString*()))  
 .thenReturn(Optional.*empty*());  
  
 // Perform the GET request for a non-existent event  
 mockMvc.perform(*get*("/events/nonExistentEventId/errors"))  
 .andExpect(*status*().isNotFound());  
 }  
  
 //Scenario 4: testRetrieveErrorsResponseByEventId\_NullEventId  
 @Test  
 public void testRetrieveErrorsResponseByEventId\_NullEventId() throws Exception {  
 // Perform the GET request with a null eventId  
 mockMvc.perform(*get*("/events/null/errors"))  
 .andExpect(*status*().isBadRequest()) // Verify that the response status code is 400 (Bad Request)  
 .andExpect(MockMvcResultMatchers.*content*().contentType(MediaType.*APPLICATION\_JSON*))  
 .andExpect((ResultMatcher) *jsonPath*("$.error").value("Bad Request")) // Verify an appropriate error message  
 .andExpect((ResultMatcher) *jsonPath*("$.message").value("The 'eventId' path parameter is required and must not be null or empty."));  
 }  
  
 //Scenario 5: testRetrieveErrorsResponseByEventId\_EmptyEventId  
 @Test  
 public void testRetrieveErrorsResponseByEventId\_EmptyEventId() throws Exception {  
 // Perform the GET request with an empty eventId  
 mockMvc.perform(*get*("/events//errors"))  
 .andExpect(*status*().isBadRequest()) // Verify that the response status code is 400 (Bad Request)  
 .andExpect(MockMvcResultMatchers.*content*().contentType(MediaType.*APPLICATION\_JSON*))  
 .andExpect((ResultMatcher) *jsonPath*("$.error").value("Bad Request")) // Verify an appropriate error message  
 .andExpect((ResultMatcher) *jsonPath*("$.message").value("The 'eventId' path parameter is required and must not be null or empty."));  
 }  
  
 //Scenario 6: testRetrieveErrorsResponseByEventId\_SpecialCharacterEventId  
 @Test  
 public void testRetrieveErrorsResponseByEventId\_SpecialCharacterEventId() throws Exception {  
 // Perform the GET request with an eventId containing special characters  
 mockMvc.perform(*get*("/events/!#$%^&\*@e/errors"))  
 .andExpect(*status*().isBadRequest()) // Verify that the response status code is 400 (Bad Request)  
 .andExpect(MockMvcResultMatchers.*content*().contentType(MediaType.*APPLICATION\_JSON*))  
 .andExpect((ResultMatcher) *jsonPath*("$.error").value("Bad Request")) // Verify an appropriate error message  
 .andExpect((ResultMatcher) *jsonPath*("$.message").value("The 'eventId' path parameter contains invalid characters. " +  
 "Please use alphanumeric characters or hyphens."));  
 }  
  
 //Scenario 7: testRetrieveErrorsResponseByEventId\_InternalServerError  
 @Test  
 public void testRetrieveErrorsResponseByEventId\_InternalServerError() throws Exception {  
 // Mock the service to throw an exception, simulating an internal server error  
 *when*(documentGeneratorEventStoreService.fetchErrorsByEventId("event123"))  
 .thenThrow(new RuntimeException("Internal Server Error Occurred"));  
  
 // Perform the GET request  
 mockMvc.perform(*get*("/events/event123/errors"))  
 .andExpect(*status*().isInternalServerError()) // Verify that the response status code is 500 (Internal Server Error)  
 .andExpect(MockMvcResultMatchers.*content*().contentType(MediaType.*APPLICATION\_JSON*))  
 .andExpect((ResultMatcher) *jsonPath*("$.error").value("Internal Server Error")) // Verify an appropriate error message  
 .andExpect((ResultMatcher) *jsonPath*("$.message").value("Internal Server Error Occurred"));  
 }**