**@Operation(summary = "API to retrieve event by eventId")  
  
@ApiResponses(value = {@ApiResponse(responseCode = "200", description = "Event retrieved successfully",  
 content = {@Content(mediaType = "application/json",  
 schema = @Schema(implementation = EventResponse.class))})})  
  
@GetMapping(value = DocumentGeneratorEventStoreConstants.*SLASH* + DocumentGeneratorEventStoreConstants.*EVENT\_ENDPOINT* + DocumentGeneratorEventStoreConstants.*SLASH* + DocumentGeneratorEventStoreConstants.*EVENT\_ID\_PATH\_PARAM*)  
  
public ResponseEntity<EventResponse> retrieveEventByEventId(  
 final HttpServletRequest httpRequest, @NotNull @PathVariable final String eventId) {  
 *LOG*.info("Getting event data from eventId:: {}", GenericUtil.*sanitizeValues*(eventId));  
  
 Optional<EventResponse> eventDataResponse = documentGeneratorEventStoreService.fetchEventByEventId(eventId);  
  
 return eventResponseMapper(httpRequest, eventDataResponse);  
}**

**DOCUMENTATION:**

**@Operation(summary = "API to retrieve event by eventId")  
/\* @Operation, Used in Spring Framework to provide information  
about a REST API operation (HTTP endpoint).  
Summary is used to provide a short description of what this API operation does,  
which is to retrieve an event by its eventId.\*/  
  
@ApiResponses(value = {@ApiResponse(responseCode = "200", description = "Event retrieved successfully",  
 content = {@Content(mediaType = "application/json",  
 schema = @Schema(implementation = EventResponse.class))})})  
/\* @ApiResponses annotation to specify possible responses for this API operation.  
@ApiResponse is used to define a specific response.  
Here, it defines a response with HTTP status code 200 (OK)  
and a description indicating that the event was retrieved successfully.  
@Content annotation specifying the media type of the response (JSON) and  
the schema definition for the response data, which is based on the EventResponse class.\*/  
  
@GetMapping(value = DocumentGeneratorEventStoreConstants.*SLASH* + DocumentGeneratorEventStoreConstants.*EVENT\_ENDPOINT* + DocumentGeneratorEventStoreConstants.*SLASH* + DocumentGeneratorEventStoreConstants.*EVENT\_ID\_PATH\_PARAM*)  
/\* @GetMapping annotation, this method handles HTTP GET requests.  
The value attribute specifies the URL path for this endpoint.  
It's constructed by concatenating several constants to form the complete URL.  
For Example: the URL will be /event/{eventId} \*/  
  
public ResponseEntity<EventResponse> retrieveEventByEventId(  
 final HttpServletRequest httpRequest, @NotNull @PathVariable final String eventId) {  
 *LOG*.info("Getting event data from eventId:: {}", GenericUtil.*sanitizeValues*(eventId));  
/\* Defines the method retrieveEventByEventId, which handles the HTTP GET request to retrieve event by its eventId.  
It takes two parameters: Final HttpServletRequest httpRequest: This parameter represents the HTTP request itself,  
allowing the method to access information about the request.  
@NotNull @PathVariable final String eventId: This parameter is annotated with @PathVariable,  
indicating that it is extracted from the URL path.  
@NotNull annotation likely enforces that the eventId cannot be null.  
Log information about the operation. It's using some kind of logging framework (LOG) to record that  
it's retrieving event data for a specific eventId. It appears to sanitize the eventId using a GenericUtil method.\*/  
  
 Optional<EventResponse> eventDataResponse = documentGeneratorEventStoreService.fetchEventByEventId(eventId);  
/\* Invokes a service method fetchEventByEventId to retrieve event data based on the eventId.  
It returns the result as an Optional<EventResponse>, which means the result may or may not be present.\*/  
  
 return eventResponseMapper(httpRequest, eventDataResponse);  
/\* Returns the result of the API operation. It calls a method eventResponseMapper with two parameters:  
httpRequest: The HTTP request object.  
eventDataResponse: The result of the event retrieval operation.  
The eventResponseMapper method is responsible for mapping the retrieved data into an appropriate response format and  
returning it as an instance of ResponseEntity<EventResponse>, which is a common way to structure HTTP responses in  
Spring applications.\*/  
  
}**

**@Test  
public void testRetrieveEventByEventId\_Success() throws Exception {  
// Arrange  
String eventId = "sample-event-id";  
HttpServletRequest httpRequest = mock(HttpServletRequest.class);  
  
// Create a sample response data  
EventResponse eventResponse = new EventResponse();  
  
// Mock the service method to return the response data  
when(documentGeneratorEventStoreService.fetchEventByEventId(eventId))  
.thenReturn(Optional.of(eventResponse));  
  
// Act  
ResponseEntity<EventResponse> result = eventController.retrieveEventByEventId(httpRequest, eventId);  
  
// Assert  
verify(documentGeneratorEventStoreService, times(1)).fetchEventByEventId(eventId);  
assertEquals(HttpStatus.OK, result.getStatusCode());  
assertEquals(eventResponse, result.getBody());  
}  
  
@Test  
public void testRetrieveEventByEventId\_NotFound() throws Exception {  
// Arrange  
String eventId = "non-existent-event-id";  
HttpServletRequest httpRequest = mock(HttpServletRequest.class);  
  
// Mock the service method to return an empty Optional (event not found)  
when(documentGeneratorEventStoreService.fetchEventByEventId(eventId))  
.thenReturn(Optional.empty());  
  
// Act  
ResponseEntity<EventResponse> result = eventController.retrieveEventByEventId(httpRequest, eventId);  
  
// Assert  
verify(documentGeneratorEventStoreService, times(1)).fetchEventByEventId(eventId);  
assertEquals(HttpStatus.NOT\_FOUND, result.getStatusCode());  
assertNull(result.getBody());  
}  
}**

**Method 3:**

**@Operation(summary = "API to retrieve error details by eventId")  
/\*This is an OpenAPI (Swagger) annotation that provides a summary or description for your API operation.  
It describes the purpose of the API endpoint, which is to retrieve error details by eventId.\*/  
  
@ApiResponses(value = {@ApiResponse(responseCode = "200", description = "Error retrieved successfully",  
 content = {@Content(mediaType = "application/json",  
 schema = @Schema(implementation = EventResponse.class))})})  
/\* @ApiResponses annotation is used to specify possible responses for this API operation.  
 When the API operation is successful, it will return a response with a 200 HTTP status code,  
 and the description is "Error retrieved successfully."\*/  
  
@GetMapping(value = DocumentGeneratorEventStoreConstants.*SLASH* + DocumentGeneratorEventStoreConstants.*EVENT\_ENDPOINT* + DocumentGeneratorEventStoreConstants.*SLASH* + DocumentGeneratorEventStoreConstants.*EVENT\_ID\_PATH\_PARAM* + DocumentGeneratorEventStoreConstants.*SLASH* + DocumentGeneratorEventStoreConstants.*ERROR\_ENDPOINT*)  
/\*@GetMapping: This annotation indicates that this method handles HTTP GET requests  
 and value is an attribute that concatenated constants to form the complete URL  
 The URL will be: /event/{eventId}/errors \*/  
  
public ResponseEntity<EventResponse> retrieveErrorsResponseByEventId(  
 final HttpServletRequest httpRequest, @NotNull @PathVariable final String eventId) {  
 *LOG*.info("Getting errors data from eventId: {}", GenericUtil.*sanitizeValues*(eventId));  
/\* This is the method signature for the API endpoint. It specifies that this endpoint will return an HTTP response  
wrapped in a ResponseEntity object. The response will contain data of type EventResponse \*/  
  
 Optional<EventResponse> eventDataResponse =  
 documentGeneratorEventStoreService.fetchErrorsByEventId(eventId);  
/\* This line calls the documentGeneratorEventStoreService to fetch error details for the specified eventId.  
The result is wrapped in an Optional because the event data might not exist for the given eventId.\*/  
 return eventResponseMapper(httpRequest, eventDataResponse);  
/\* This method returns the result of calling eventResponseMapper, which is responsible for mapping the data   
to an EventResponse and constructing the HTTP response.\*/   
}**

package com.project.bos.dg.datastore.controller;  
  
import com.fasterxml.jackson.core.JsonProcessingException;  
import com.project.bos.dg.datastore.constants.DocumentGeneratorEventStoreConstants;  
import com.project.bos.dg.datastore.model.request.EventRequest;  
import com.project.bos.dg.datastore.model.response.EventResponse;  
import com.project.bos.dg.datastore.service.DocumentGeneratorEventStoreService;  
import org.hamcrest.core.AnyOf;  
import org.junit.Assert;  
import org.junit.jupiter.api.Assertions;  
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.beans.factory.annotation.Autowired;  
import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;  
import org.springframework.boot.test.context.SpringBootTest;  
import org.springframework.boot.test.mock.mockito.MockBean;  
import org.springframework.http.HttpStatus;  
import org.springframework.http.MediaType;  
import org.springframework.test.web.servlet.MockMvc;  
import org.springframework.test.web.servlet.MvcResult;  
  
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.\*;  
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.\*;  
import java.util.Optional;  
import static org.mockito.Mockito.\*;  
  
@SpringBootTest  
@AutoConfigureMockMvc  
  
public class DocumentGeneratorEventStoreControllerTest {  
 @InjectMocks  
 private DocumentGeneratorEventStoreController controller;  
 @Mock  
 private DocumentGeneratorEventStoreService eventStoreService;  
  
 @BeforeEach  
 void setUp() {  
 MockitoAnnotations.*openMocks*(this);  
 }  
 @Autowired  
 private MockMvc mockMvc;  
 @MockBean  
 private DocumentGeneratorEventStoreService documentGeneratorEventStoreService;  
  
 // Method 3 "API to retrieve event by eventId"  
  
 //Test Case 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");  
  
 // Mock the behavior of documentGeneratorEventStoreService.fetchEventByEventId  
 *when*(documentGeneratorEventStoreService.fetchEventByEventId(*anyString*()))  
 .thenReturn(Optional.*of*(eventResponse));  
  
 // Perform the GET request  
 mockMvc.perform(*get*("/event/{eventId}", "123"))  
 .andExpect(*status*().isOk())  
 .andExpect(*content*().contentType(MediaType.*APPLICATION\_JSON*))  
 .andExpect(*jsonPath*("$.correlationId").value("123"))  
 .andExpect(*jsonPath*("$.applicationLabel").value("SampleApp"))  
 .andExpect(*jsonPath*("$.code").value(200L))  
 .andExpect(*jsonPath*("$.status").value("Success"))  
 .andExpect(*jsonPath*("$.message").value("Event retrieved successfully"))  
 .andExpect(*jsonPath*("$.eventDataResponse").doesNotExist()) // Adjust as needed  
 .andExpect(*jsonPath*("$.time").value("2023-09-23T10:00:00Z"))  
 .andExpect(*jsonPath*("$.path").value("/your-api-endpoint/123"))  
 .andExpect(*jsonPath*("$.method").value("GET"));  
  
 // Verify that documentGeneratorEventStoreService.fetchEventByEventId was called with the correct argument  
 *verify*(documentGeneratorEventStoreService, *times*(1)).fetchEventByEventId("123");  
 }  
  
 //Test Case 2: testRetrieveEventByEventId\_NotFound  
 @Test  
 public void testRetrieveEventByEventId\_NotFound() throws Exception {  
 // Mock the service response  
 *when*(documentGeneratorEventStoreService.fetchEventByEventId(*anyString*())).thenReturn(Optional.*empty*());  
  
 // Perform the GET request  
 mockMvc.perform(*get*("/events/{eventId}", "NotExistId123"))  
 .andExpect(*status*().isNotFound());  
 }  
  
 //Test Case 3: testRetrieveEventByEventId\_InvalidInput  
 @Test  
 public void testRetrieveEventByEventId\_InvalidInput() throws Exception {  
 // Perform the GET request with a null eventId  
 mockMvc.perform(*get*("/events/{eventId}", null))  
 .andExpect(*status*().isBadRequest());  
 }  
  
 //Test Case 4: testRetrieveEventByEventId\_InternalServerError  
 @Test  
 public void testRetrieveEventByEventId\_InternalServerError() throws Exception {  
 // Mock an exception being thrown in the service  
 *when*(documentGeneratorEventStoreService.fetchEventByEventId(*anyString*()))  
 .thenThrow(new RuntimeException("Some error occurred."));  
  
 // Perform the GET request  
 mockMvc.perform(*get*("/events/{eventId}", "123"))  
 .andExpect(*status*().isInternalServerError());//Expect Http 500 Internal Server Error here  
 }  
  
  
 /\*@Test  
 void saveNewEventTest() throws Exception {  
 EventResponse response = EventResponse.builder().status(String.valueOf(HttpStatus.CREATED.value()))  
 .build();//*TODO set rest of the fields*  
 when(documentGeneratorEventStoreService.saveEvent(Mockito.any(EventRequest.class))).thenReturn(Optional.of(response));  
 EventRequest request = mock(EventRequest.class);  
  
 mockMvc.perform(post( DocumentGeneratorEventStoreConstants.SLASH  
 + DocumentGeneratorEventStoreConstants.EVENT\_ENDPOINT, request))  
 .andExpect(status().isCreated());  
 }  
  
 @Test  
 void saveExistingEventTest() throws Exception {  
 EventResponse response = EventResponse.builder().status(String.valueOf(HttpStatus.OK.value()))  
 .build();//*TODO set rest of the fields*  
 when(documentGeneratorEventStoreService.saveEvent(Mockito.any(EventRequest.class))).thenReturn(Optional.of(response));  
 EventRequest request = mock(EventRequest.class);  
  
 mockMvc.perform(post( DocumentGeneratorEventStoreConstants.SLASH  
 + DocumentGeneratorEventStoreConstants.EVENT\_ENDPOINT, request))  
 .andExpect(status().isOk());  
 }  
  
 @Test  
 void saveExistingEventResponseNull() throws Exception {  
 when(documentGeneratorEventStoreService.saveEvent(Mockito.any(EventRequest.class))).thenReturn(Optional.empty());  
 EventRequest request = mock(EventRequest.class);  
  
 MvcResult result = mockMvc.perform(post( DocumentGeneratorEventStoreConstants.SLASH  
 + DocumentGeneratorEventStoreConstants.EVENT\_ENDPOINT, request))  
 .andExpect(status().isOk()).andReturn();  
 Assertions.assertEquals("", result.getResponse().getContentAsString());  
 }\*/  
  
 // Method 4: "API to retrieve error details by eventId"  
 // Test Case 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(*content*().contentType(MediaType.*APPLICATION\_JSON*))  
 .andExpect(*jsonPath*("$.correlationId").value("123"))  
 .andExpect(*jsonPath*("$.applicationLabel").value("SampleApp"))  
 .andExpect(*jsonPath*("$.code").value(200L))  
 .andExpect(*jsonPath*("$.status").value("Success"))  
 .andExpect(*jsonPath*("$.message").value("Error retrieved successfully"))  
 .andExpect(*jsonPath*("$.eventDataResponse").doesNotExist())  
 .andExpect(*jsonPath*("$.time").value("2023-09-23T10:00:00Z"))  
 .andExpect(*jsonPath*("$.path").value("/event/123/errors"))  
 .andExpect(*jsonPath*("$.method").value("GET"));  
 }  
  
 // Test Case 2: 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());  
 }  
  
 // Test Case 3: 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());  
 }  
}