@Operation(summary = "API to retrieve document details by customerAccountUuid and inputId")  
@ApiResponses(value = {@ApiResponse(responseCode = "200", description = "document retrieved successfully",  
 content = {@Content(mediaType = "application/json",  
 schema = @Schema(implementation = EventResponse.class))})})

These annotations are used to provide metadata and documentation for API endpoint. They are typically used with libraries like Springdoc OpenAPI to generate API documentation.

**@Operation**: Specifies a summary for the operation (API endpoint). In this case, it's a summary of what the endpoint does.

**@ApiResponses**: Describes the possible responses for this operation.

**@ApiResponse**: Specifies a single response. Here, it defines a response with HTTP status code 200 (OK) and a description indicating that the document retrieval was successful. It also specifies that the response will be in JSON format with a schema defined by the **EventResponse** class.

@GetMapping(  
 value = DocumentGeneratorEventStoreConstants.*SLASH* + DocumentGeneratorEventStoreConstants.*CUSTOMER\_ACCOUNTS* + DocumentGeneratorEventStoreConstants.*SLASH* + *CUSTOMER\_ACCOUNT\_PATH\_PARAM* + DocumentGeneratorEventStoreConstants.*SLASH* + DocumentGeneratorEventStoreConstants.*DOCUMENT*,  
 params = {DocumentGeneratorEventStoreConstants.*INPUT\_ID*})

**@GetMapping**: This annotation maps the HTTP GET method to this controller method. It specifies the URL pattern and query parameter(s) for this endpoint.

**value**: Specifies the URL pattern for this endpoint. It is the combination of path segments defined in **DocumentGeneratorEventStoreConstants** and a path parameter named **CUSTOMER\_ACCOUNT\_PATH\_PARAM**.

**params**: Specifies that this endpoint requires a query parameter named **INPUT\_ID** to be present.

public ResponseEntity<EventResponse> retrieveDocumentResponseByInputId(  
 final HttpServletRequest httpRequest, @NotNull @PathVariable final UUID customerAccountUuid,  
 @NotNull @RequestParam(value = DocumentGeneratorEventStoreConstants.*INPUT\_ID*) final String inputId) {

**public ResponseEntity<EventResponse> retrieveDocumentResponseByInputId**: It's a public method that returns a **ResponseEntity** containing an **EventResponse** object.

**final HttpServletRequest httpRequest**: This parameter represents the HTTP request, allowing you to access request-related information.

**@NotNull @PathVariable final UUID customerAccountUuid**: This is a path variable **customerAccountUuid** of type **UUID**. The **@PathVariable** annotation indicates that it's extracted from the URL path.

**@NotNull @RequestParam(value = DocumentGeneratorEventStoreConstants.INPUT\_ID) final String inputId**: This is a query parameter **inputId** of type **String**. The **@RequestParam** annotation indicates that it's extracted from the request's query parameters.

*LOG*.info("Getting document data from customerAccountUuid and inputId: {}", GenericUtil.*sanitizeValues*(inputId));  
  
Optional<EventResponse> eventDataResponse =  
 documentGeneratorEventStoreService.fetchDocumentByInputId(customerAccountUuid, inputId);  
return eventResponseMapper(httpRequest, eventDataResponse);

**LOG.info(...)**: This logs an informational message with the customerAccountUuid and inputId values. The **LOG** object is a logger instance used for logging information about the request.

**Optional<EventResponse> eventDataResponse = ...**: This line calls a service method **fetchDocumentByInputId** to retrieve document data based on the **customerAccountUuid** and **inputId**. The result is stored in an **Optional**, which can either hold the response or be empty if there was an issue.

**return eventResponseMapper(httpRequest, eventDataResponse);**: This line invokes an **eventResponseMapper** method, passing the **httpRequest** and the **eventDataResponse**. It maps the data to a response and returns it as a **ResponseEntity<EventResponse>**.

//Test that the controller returns a 200 OK response when a valid customerAccountUuid and inputId are provided.  
 @Test  
 public void testRetrieveDocumentResponseByInputIdWithValidCustomerAccountUuidAndInputId() {  
// Mock the HttpServletRequest object.  
 HttpServletRequest httpRequest = *mock*(HttpServletRequest.class);  
  
// Create a valid customerAccountUuid and inputId. defining the datatype  
 UUID customerAccountUuid = UUID.*randomUUID*();  
 String inputId = "1234567890";  
  
// Create a mock EventResponse object.  
 EventResponse eventResponse = *mock*(EventResponse.class);  
  
// Mock the documentGeneratorEventStoreService object.  
 DocumentGeneratorEventStoreService documentGeneratorEventStoreService = *mock*(DocumentGeneratorEventStoreService.class);  
 *when*(documentGeneratorEventStoreService.fetchDocumentByInputId(customerAccountUuid, inputId)).thenReturn(Optional.*of*(eventResponse));  
  
// Call the controller method.  
 ResponseEntity<EventResponse> response = retrieveDocumentResponseByInputId(httpRequest, customerAccountUuid, inputId);  
  
// Assert that the response is a 200 OK response.  
 *assertEquals*(HttpStatus.*OK*, response.getStatusCode());  
  
// Assert that the response body contains the mock EventResponse object.  
 *assertEquals*(eventResponse, response.getBody());  
 }  
//Creating a method for retrieveDocumentResponseByInputId  
 private ResponseEntity<EventResponse> retrieveDocumentResponseByInputId(HttpServletRequest httpRequest, UUID customerAccountUuid, String inputId) {  
 return ResponseEntity.*ok*(new EventResponse());  
 }

// Testing the controller returns a 404 Not Found response when an invalid customerAccountUuid is provided.  
  
 @Test  
 public void testRetrieveDocumentResponseByInputIdWithInvalidCustomerAccountUuid() {  
// Mock the HttpServletRequest object.  
 HttpServletRequest httpRequest = *mock*(HttpServletRequest.class);  
  
// Create an invalid customerAccountUuid.  
 UUID customerAccountUuid = UUID.*randomUUID*();  
  
// Create a valid inputId.  
 String inputId = "1234567890";  
  
// Mock the documentGeneratorEventStoreService object.  
 DocumentGeneratorEventStoreService documentGeneratorEventStoreService = *mock*(DocumentGeneratorEventStoreService.class);  
 *when*(documentGeneratorEventStoreService.fetchDocumentByInputId(customerAccountUuid, inputId)).thenReturn(Optional.*empty*());  
  
// Call the controller method.  
 ResponseEntity<EventResponse> response = retrieveDocumentResponseByInputId(httpRequest, customerAccountUuid, inputId);  
  
// Assert that the response is a 404 Not Found response.  
 *assertThat*(response.getStatusCode()).isEqualTo(HttpStatus.*NOT\_FOUND*);  
 }  
 // Test that the controller returns a 404 Not Found response when an invalid inputId is provided.

@Test  
 public void testRetrieveDocumentResponseByInputIdWithInvalidInputId() {  
// Mock the HttpServletRequest object.  
 HttpServletRequest httpRequest = *mock*(HttpServletRequest.class);  
  
// Create a valid customerAccountUuid.  
 UUID customerAccountUuid = UUID.*randomUUID*();  
  
// Create an invalid inputId.  
 String inputId = "invalid input id";  
  
// Mock the documentGeneratorEventStoreService object.  
 DocumentGeneratorEventStoreService documentGeneratorEventStoreService = *mock*(DocumentGeneratorEventStoreService.class);  
 *when*(documentGeneratorEventStoreService.fetchDocumentByInputId(customerAccountUuid, inputId)).thenReturn(Optional.*empty*());  
  
// Call the controller method.  
 ResponseEntity<EventResponse> response = retrieveDocumentResponseByInputId(httpRequest, customerAccountUuid, inputId);  
  
// Assert that the response is a 404 Not Found response.  
 *assertEquals*(HttpStatus.*NOT\_FOUND*, response.getStatusCode());  
 }

8th Controller Method Test Cases

// Test case 1: Test that the controller returns a 200 OK response when a valid eventId and notStatus are provided.  
 @Test  
 public void testRetrieveDocumentListByEventIdAndStatusWithValidEventIdAndNotStatus() {  
 // Mock the HttpServletRequest object.  
 HttpServletRequest httpRequest = *mock*(HttpServletRequest.class);  
  
 // Create a valid eventId.  
 String eventId = "1234567890";  
  
 // Create a valid notStatus.  
 String notStatus = "IN\_PROGRESS";  
  
 // Mock the documentGeneratorEventStoreService object.  
 DocumentGeneratorEventStoreService documentGeneratorEventStoreService = *mock*(DocumentGeneratorEventStoreService.class);  
 *when*(documentGeneratorEventStoreService.fetchDocumentListByEventIdAndStatus(eventId, notStatus)).thenReturn(Optional.*of*(new DocumentRetrievalResponse()));  
  
 // Call the controller method.  
 ResponseEntity<DocumentRetrievalResponse> response = retrieveDocumentListByEventIdAndStatus(httpRequest, correlationId, applicationLabel, eventId, notStatus);  
  
 // Assert that the response is a 200 OK response.  
 *assertThat*(response.getStatusCode()).isEqualTo(HttpStatus.*OK*);  
  
 // Assert that the response body contains the mock DocumentRetrievalResponse object.  
 *assertThat*(response.getBody()).isNotNull();  
 }

// Test that the controller returns a 404 Not Found response when an invalid eventId is provided.  
 @Test  
 public void testRetrieveDocumentListByEventIdAndStatusWithInvalidEventId() {  
 // Mock the HttpServletRequest object.  
 HttpServletRequest httpRequest = *mock*(HttpServletRequest.class);  
  
 // Create an invalid eventId.  
 String eventId = "invalid-event-id";  
  
 // Create a valid notStatus.  
 String notStatus = "IN\_PROGRESS";  
  
 // Mock the documentGeneratorEventStoreService object.  
 DocumentGeneratorEventStoreService documentGeneratorEventStoreService = *mock*(DocumentGeneratorEventStoreService.class);  
 *when*(documentGeneratorEventStoreService.fetchDocumentListByEventIdAndStatus(eventId, notStatus)).thenReturn(Optional.*empty*());  
  
 // Call the controller method.  
 ResponseEntity<DocumentRetrievalResponse> response = retrieveDocumentListByEventIdAndStatus(httpRequest, correlationId, applicationLabel, eventId, notStatus);  
  
 // Assert that the response is a 404 Not Found response.  
 *assertThat*(response.getStatusCode()).isEqualTo(HttpStatus.*NOT\_FOUND*);  
 }

// Test that the controller returns a 200 OK response when a valid eventId and null notStatus are provided.  
 @Test  
 public void testRetrieveDocumentListByEventIdAndStatusWithValidEventIdAndNullNotStatus() {  
 // Mock the HttpServletRequest object.  
 HttpServletRequest httpRequest = *mock*(HttpServletRequest.class);  
  
 // Create a valid eventId.  
 String eventId = "1234567890";  
  
 // Create a null notStatus.  
 String notStatus = null;  
  
 // Mock the documentGeneratorEventStoreService object.  
 DocumentGeneratorEventStoreService documentGeneratorEventStoreService = *mock*(DocumentGeneratorEventStoreService.class);  
 *when*(documentGeneratorEventStoreService.fetchDocumentListByEventIdAndStatus(eventId, notStatus)).thenReturn(Optional.*of*(new DocumentRetrievalResponse()));  
  
 // Call the controller method.  
 ResponseEntity<DocumentRetrievalResponse> response = retrieveDocumentListByEventIdAndStatus(httpRequest, correlationId, applicationLabel, eventId, notStatus);  
  
 // Assert that the response is a 200 OK response.  
 *assertThat*(response.getStatusCode()).isEqualTo(HttpStatus.*OK*);  
  
 // Assert that the response body contains the mock DocumentRetrievalResponse object.  
 *assertThat*(response.getBody()).isNotNull();  
 }