@RestController

**@RestController** is a fundamental annotation in Spring for building RESTful web services. It combines the **@Controller** and **@ResponseBody** annotations into one, making it easier to develop RESTful APIs and handle HTTP requests and responses.

@Validated

**@Validated** annotation is used in Spring applications to apply validation rules to method parameters or method return values.

It works in conjunction with the Java Bean Validation (JSR-303) framework, which defines a set of annotations like **@NotNull**, **@Size**, **@Pattern**, and more, to specify validation constraints for bean properties.

By using **@Validated**, you can trigger the validation of these constraints in your Spring application.

@Operation(summary = "Api to interact with database to add Event")

**@Operation**: This is likely a custom annotation or one from a library or framework used to **provide metadata about the operation** being performed by the API endpoint. It's not a standard Java annotation but could be part of a framework-specific set of annotation

**summary = "Api to interact with database to add Event"**: **This part is specifying a summary or description of the API operation**. In this case, **it's describing what the API does.** It suggests that this API endpoint is used to interact with a database to add an "Event." This summary is usually used for documentation purposes, making it clear to developers what the endpoint does.

In the context of building a RESTful API, such annotations are often **used to generate API documentation automatically** or to provide additional metadata to the framework, which can help with routing and validation.

@ApiResponses

**@ApiResponses**: This annotation is typically **used to describe the various responses that an API endpoint can return**. It's often used for documenting the expected responses for different scenarios.

@ApiResponses(value = {@ApiResponse(responseCode = "201", description = "Event added successfully",  
 content = {@Content(mediaType = "application/json",  
 schema = @Schema(implementation = EventResponse.class))}),  
 @ApiResponse(responseCode = "200", description = "Event already exists",  
 content = {@Content(mediaType = "application/json",  
 schema = @Schema(implementation = EventResponse.class))})})

**@ApiResponse**: This annotation defines **one specific response**. In your code, there are two of them, one for an HTTP 201 response (indicating success) and another for an HTTP 200 response (indicating success with an event already existing).

For the first **@ApiResponse**:

**responseCode = "201"**: This indicates that this response corresponds to an HTTP status code 201, which is often used to indicate that a resource has been successfully created on the server. In your context, it means that when an event is successfully added, this response code will be returned.

**description = "Event added successfully"**: This is a human-readable description of what this response means. It states that the event was added successfully.

**content = {...}**: This part specifies the content type and structure of the response. In this case, it specifies that the response content is in JSON format (**"application/json"**), and the structure of the response is defined by the schema specified in the **@Schema** annotation, which references the **EventResponse** class. This likely indicates that the response will be a JSON object conforming to the structure defined in the **EventResponse** class.

For example, if **EventResponse** is a Java class representing event data and has fields like **eventId**, **eventName**, **eventDate**, and so on, the **schema** annotation is indicating that the response should include these fields with their respective data types, possibly in a JSON format if the API produces JSON responses.

For the second **@ApiResponse**:

**responseCode = "200"**: This indicates that this response corresponds to an HTTP status code 200, which is a generic success code. In your context, it means that the request was successful (as there was no error), but the event already exists.

**description = "Event already exists"**: This is a human-readable description of what this response means. It states that the event already exists.

**content = {...}**: Similar to the first response, it specifies that the response content is in JSON format (**"application/json"**), and the structure of the response is defined by the **EventResponse** class, meaning that it will likely return a JSON object conforming to the structure defined in **EventResponse**

@PostMapping(value = DocumentGeneratorEventStoreConstants.*SLASH* + DocumentGeneratorEventStoreConstants.*EVENT\_ENDPOINT*, consumes = MediaType.APPLICATION\_JSON\_VALUE)

**@PostMapping**: This annotation is used to specify that the annotated method (usually a controller method) should handle HTTP POST requests. In RESTful API design, a POST request is typically used to create or add a new resource on the server.

**value = DocumentGeneratorEventStoreConstants.SLASH + DocumentGeneratorEventStoreConstants.EVENT\_ENDPOINT**: This part defines the URL path where this POST request should be handled. It seems that the path is being constructed dynamically by concatenating values from

**DocumentGeneratorEventStoreConstants**. Here's what's happening:

**DocumentGeneratorEventStoreConstants.SLASH**: This likely represents a constant that contains a forward slash (**/**). It's used to separate parts of the URL path.

**DocumentGeneratorEventStoreConstants.EVENT\_ENDPOINT**: This likely represents another constant that defines the endpoint or resource name. When combined with the slash, it forms the complete path where the POST request should be sent. For example, if **SLASH** is **/** and **EVENT\_ENDPOINT** is **events**, then this would map to a URL path like **/events**.

**consumes = MediaType.APPLICATION\_JSON\_VALUE**: This part specifies the media type that the HTTP request's body should be in for this endpoint to be able to process it. In this case, it indicates that the endpoint consumes JSON data (**"application/json"**). This means that when a client sends a POST request to this endpoint, the request body is expected to be in JSON format.

In summary, the **@PostMapping** annotation is used to map an HTTP POST request to a specific URL path, and it specifies that the request should contain JSON data. The actual URL path is constructed using constants from **DocumentGeneratorEventStoreConstants**. This annotation is commonly used to create new resources or perform some operation that requires data to be sent in JSON format to the server.

public ResponseEntity<EventResponse> saveEvent(@Valid @RequestBody final EventRequest eventRequest)  
 throws JsonProcessingException {  
 String correlationId = GenericUtil.*sanitizeValues*(eventRequest.getCorrelationId());  
 *LOG*.info("Document Event data correlation id: {}", correlationId);

**public ResponseEntity<EventResponse> saveEvent**: This is a method declaration. It specifies that the method is public, returns a **ResponseEntity** containing an object of type **EventResponse**, and is named **saveEvent**. In a RESTful API context, this method is likely responsible for handling a POST request to create or save an event.

**@Valid**: This is a method parameter annotation. It's typically used in conjunction(joining)with Spring's validation framework (e.g., Hibernate Validator) to indicate that the **eventRequest** object should be validated against defined constraints. If the request doesn't satisfy these constraints, a validation exception may be thrown

**@RequestBody final EventRequest eventRequest**: This annotation tells the framework that the data for this method is expected to come from the HTTP request body. It's commonly used for deserializing JSON or other data formats sent by clients into a Java object. **EventRequest** is likely a class representing the structure of the request data.

**throws JsonProcessingException**: This method can throw a **JsonProcessingException**. This exception typically occurs when there's an issue with JSON processing, such as when serializing or deserializing JSON data. It indicates that there might be an error when converting between JSON and Java objects

**String correlationId = GenericUtil.sanitizeValues(eventRequest.getCorrelationId());**: This line of code extracts the correlation ID from the **eventRequest** object and stores it in a local variable named **correlationId**. It appears that the **sanitizeValues** method from the **GenericUtil** class is being used to sanitize or modify the correlation ID in some way.

**LOG.info("Document Event data correlation id: {}", correlationId);**: This line logs an informational message using a logger (presumably an instance of a logging framework like Log4j or SLF4J). It prints out the correlation ID to provide information about the event data being processed. This can be useful for debugging and auditing purposes.

this method is designed to handle a POST request for creating or saving an event. It validates the incoming request data, extracts the correlation ID, and logs it for tracking and debugging purposes. The response is expected to be a **ResponseEntity** containing an **EventResponse** object, likely representing the result of the event creation or save operation.

Optional<EventResponse> optionalEventResponse = documentGeneratorEventStoreService.saveEvent(eventRequest);

**Optional<EventResponse>**:

This part declares a variable named **optionalEventResponse** with a type of **Optional<EventResponse>**.

**Optional** is a class in Java introduced to handle situations where a value might be present or absent without resorting to using **null**.

In this case, it suggests that the result of the method call **documentGeneratorEventStoreService.saveEvent(eventRequest)** can either contain an **EventResponse** or be empty (absent).Top of Form

**documentGeneratorEventStoreService.saveEvent(eventRequest)**: This is a method call to **documentGeneratorEventStoreService** with the **saveEvent** method, which seems to save an event based on the **eventRequest** parameter. The method call returns some kind of response, which is wrapped in an **Optional**.

If the event was successfully saved, it likely contains an **EventResponse**.

If there was an issue or the operation couldn't be completed, it might return an empty **Optional**.

By assigning the result of this method call to **optionalEventResponse**, you're indicating that you want to handle the response in a way that accounts for the possibility of it being empty or containing an **EventResponse**. This allows you to safely check whether the operation was successful without worrying about null references.

After assigning this result, you can use methods provided by the **Optional** class to work with the result, such as **isPresent()** to check if a value is present and **get()** to retrieve the value if it's present. You can also use methods like **ifPresent()** to execute code only if a value is present or **orElse()** to provide a default value if the **Optional** is empty. This allows for more robust and safe handling of potentially absent values in Java.

if (optionalEventResponse.isPresent()  
 && optionalEventResponse.get().getCode().equals(HttpStatus.CREATED.value())) {  
 return ResponseEntity.status(HttpStatus.CREATED).contentType(MediaType.APPLICATION\_JSON)  
 .body(optionalEventResponse.get());  
 }  
 return ResponseEntity.status(HttpStatus.OK).contentType(MediaType.APPLICATION\_JSON)  
 .body(optionalEventResponse.orElse(null));  
}

**optionalEventResponse.isPresent()**: This condition checks if the **Optional** contains a non-null value, meaning that the event operation was successful.

**optionalEventResponse.get().getCode().equals(HttpStatus.CREATED.value())**: This condition checks if the code property of the **EventResponse** object inside the **Optional** is equal to the HTTP status code for "Created," which is 201. This is an additional check to ensure that the operation corresponds to a successful event creation.

Inside the **if** block: If both conditions in the **if** statement evaluate to **true**, it means that the event creation was successful and corresponds to an HTTP 201 Created status code. In this case:

**ResponseEntity.status(HttpStatus.CREATED)**: This constructs a **ResponseEntity** with an HTTP status of 201 Created.

**.contentType(MediaType.APPLICATION\_JSON)**: Sets the response content type to JSON.

**.body(optionalEventResponse.get())**: Sets the response body to the **EventResponse** object obtained from the **Optional**.

The **else** block: If the conditions in the **if** statement evaluate to **false**, it means that either the event creation was not successful or the HTTP status code is not 201 Created. In this case:

**ResponseEntity.status(HttpStatus.OK)**: This constructs a **ResponseEntity** with an HTTP status of 200 OK (or another status code, depending on the result).

**.contentType(MediaType.APPLICATION\_JSON)**: Sets the response content type to JSON.

**.body(optionalEventResponse.orElse(null))**: Sets the response body to the **EventResponse** object if it's present in the **Optional**, or **null** if the **Optional** is empty.

In summary, this code is handling the response based on the result of an event operation. If the operation was successful and corresponds to an HTTP 201 status code, it returns a response with HTTP 201 Created. Otherwise, it returns a response with HTTP 200 OK (or another status code) to indicate the result, and it includes the **EventResponse** object in the response body if it's available.

API to interact with Database to add status for event

@Operation(summary = "API to interact with Database to add status for event.")  
@ApiResponses(value = {@ApiResponse(responseCode = "200", description = "Event status added successfully",  
 content = {@Content(mediaType = "application/json",  
 schema = @Schema(implementation = EventResponse.class))})})

**@Operation(summary = "API to interact with Database to add status for event.")**: This annotation, **@Operation**, is typically used for providing metadata and documentation about an API operation. In this case, it describes the operation with a summary. It indicates that this API operation is used to interact with a database to add a status for an event. This summary is often used in API documentation to provide a brief description of what the operation does.

**@ApiResponses(value = {...})**: This annotation is used to specify the possible responses that the API operation can return. It contains an array of **@ApiResponse** annotations.

**@ApiResponse(responseCode = "200", description = "Event status added successfully", content = {...})**: This is an **@ApiResponse** annotation that defines a specific response for the API operation.

**responseCode = "200"**: This indicates that this response corresponds to an HTTP status code 200, which typically signifies a successful request.

**description = "Event status added successfully"**: This is a human-readable description of what this response means. It states that the event status was added successfully.

**content = {...}**: This part specifies the content type and structure of the response. In this case, it specifies that the response content is in JSON format (**"application/json"**), and the structure of the response is defined by the schema specified in the **@Schema** annotation, which references the **EventResponse** class. This suggests that when the API operation is successful, it will return a JSON response conforming to the structure defined in the **EventResponse** class.

In summary, these annotations provide documentation and metadata for an API operation that adds status for an event. The operation is expected to return an HTTP 200 response with a JSON response body containing information about the event status if the operation is successful. This information can be valuable for both API developers and consumers to understand the operation and its expected outcomes.

@PostMapping(  
 value = DocumentGeneratorEventStoreConstants.*SLASH* + DocumentGeneratorEventStoreConstants.*EVENT\_ENDPOINT* + DocumentGeneratorEventStoreConstants.*SLASH* + DocumentGeneratorEventStoreConstants.*EVENT\_ID\_PATH\_PARAM*,  
 consumes = MediaType.APPLICATION\_JSON\_VALUE)

**@PostMapping**: This annotation specifies that the annotated method should handle HTTP POST requests. In the context of a RESTful API, a POST request is typically used to create or update resources on the server.

**value**: This attribute of the **@PostMapping** annotation is used to define the URL path at which this POST request should be mapped. It specifies a **string value** that represents the URL path. In this case, the **URL path is constructed by concatenating several constants** and values together:

**DocumentGeneratorEventStoreConstants.SLASH**: This likely represents a constant that contains a forward slash (**/**). It's used to separate parts of the URL path.

**DocumentGeneratorEventStoreConstants.EVENT\_ENDPOINT**: This is likely another constant that represents an endpoint or resource name. When combined with the slashes, it forms part of the URL path.

**DocumentGeneratorEventStoreConstants.EVENT\_ID\_PATH\_PARAM**: This appears to be another constant representing a path parameter in the URL. Path parameters are placeholders in the URL that can capture values provided by the client. The value of this parameter would typically be provided dynamically in the URL when a request is made.

The concatenation of these constants and values creates a specific URL path for this POST request. The resulting URL path is where clients will send their POST requests.

**consumes**: This attribute specifies the media type or content type that the HTTP request body is expected to be in for this endpoint to be able to process it. In this case, it specifies that the request body should be in JSON format (**MediaType.APPLICATION\_JSON\_VALUE**).

In summary, this **@PostMapping** annotation is used to map an HTTP POST request to a specific URL path, which is constructed using constants and values. It also specifies that the request should contain JSON data in the request body, indicating that the endpoint expects data in JSON format when clients make POST requests to this URL.

URL: [**http://localhost:8080/events/{eventId}**](http://localhost:8080/events/%7beventId%7d).

public ResponseEntity<EventResponse> createEventStatus(@NotNull @PathVariable final String eventId,  
 @Valid @RequestBody final CreateEventStatusRequest  
 createEventStatusRequest)

**public ResponseEntity<EventResponse> createEventStatus**: This is a method declaration. It specifies that the method is public, returns a **ResponseEntity** containing an object of type **EventResponse**, and is named **createEventStatus**. In a RESTful API context, this method is likely responsible for handling a request to create an event status.

**@NotNull @PathVariable final String eventId**: These annotations are used to define a path parameter named **eventId**. Path parameters are placeholders in the URL that capture values provided by the client when making a request. The **@PathVariable** annotation indicates that this parameter is extracted from the URL path, and the **@NotNull** annotation is used to ensure that the value provided for **eventId** is not null. It's expected to be a string representing the identifier of the event for which you want to create a status.

**@Valid @RequestBody final CreateEventStatusRequest createEventStatusRequest**: These annotations are used to define a request body parameter named **createEventStatusRequest**. The **@RequestBody** annotation indicates that this parameter is expected to be provided in the request body, typically in JSON format. The **@Valid** annotation is often used in conjunction with a validation framework (e.g., Hibernate Validator) to validate the request body against defined constraints. **CreateEventStatusRequest** is likely a class that represents the structure of the request data for creating an event status.

In summary, this method is designed to handle a POST request for creating an event status. It expects two parameters:

**eventId**: A path parameter extracted from the URL representing the identifier of the event.

**createEventStatusRequest**: A request body parameter containing data in the form of a **CreateEventStatusRequest** object, which likely includes details needed to create the event status.

The method then processes these parameters to create a response, typically containing an **EventResponse** object, and wraps it in a **ResponseEntity**. The **ResponseEntity** allows you to control the HTTP status code, headers, and response body that will be returned to the client.

String correlationId = GenericUtil.*sanitizeValues*(createEventStatusRequest.getCorrelationId());  
*LOG*.info("Document Event data correlation id: {}", correlationId);

**String correlationId = GenericUtil.sanitizeValues(createEventStatusRequest.getCorrelationId());**: This line of code extracts the **correlationId** from the **createEventStatusRequest** object. The **correlationId** is likely a unique identifier associated with the event status creation request. It is being sanitized by calling the **sanitizeValues** method from the **GenericUtil** class. Sanitization typically means cleaning or validating the value to ensure it meets certain requirements or security standards.

**LOG.info("Document Event data correlation id: {}", correlationId);**: This line logs an informational message using a logger (e.g., Log4j, SLF4J). It prints out the **correlationId** to provide information about the correlation ID associated with the event data being processed. This logging can be helpful for tracking and auditing purposes.

Top of Form

Optional<EventResponse> responseMetadataOptional =  
 documentGeneratorEventStoreService.saveEventStatus(eventId, createEventStatusRequest);

**Optional<EventResponse> responseMetadataOptional = ...**: This line of code invokes a method **saveEventStatus** on the **documentGeneratorEventStoreService**. It appears to be saving the event status based on the **eventId** and **createEventStatusRequest** parameters. The method returns an **Optional<EventResponse>**, which means it may or may not produce a result.

return ResponseEntity.status(HttpStatus.OK).contentType(MediaType.APPLICATION\_JSON)  
 .body(responseMetadataOptional.orElse(null));

return ResponseEntity.status(HttpStatus.OK).contentType(MediaType.APPLICATION\_JSON).body(...);: This part of the code constructs and returns a response to the client:

ResponseEntity.status(HttpStatus.OK): It sets the HTTP response status code to 200 OK, indicating that the request was successful.

.contentType(MediaType.APPLICATION\_JSON): It sets the response content type to JSON, indicating that the response body will be in JSON format.

.body(responseMetadataOptional.orElse(null)): It sets the response body to the result of responseMetadataOptional. If responseMetadataOptional contains a value, that value (an EventResponse object) will be set as the response body. If it's empty (no value present), null will be set as the response body.

In summary, this method processes a POST request to create an event status. It sanitizes the correlationId, logs it for tracking purposes, and then calls a service method to save the event status. Finally, it constructs and returns a response with an HTTP 200 OK status code and a JSON response body containing the result, or null if there's no result to return.

"API to retrieve event by eventId

@Operation(summary = "API to retrieve event by eventId")

**@Operation(summary = "API to retrieve event by eventId")**: This annotation, **@Operation**, is used to provide metadata and documentation about the API operation. In this case, it gives a summary or brief description of what the operation does. It indicates that this API operation is used to retrieve an event by its **eventide**

@ApiResponses(value = {@ApiResponse(responseCode = "200", description = "Event retrieved successfully",  
 content = {@Content(mediaType = "application/json",  
 schema = @Schema(implementation = EventResponse.class))})})

**@ApiResponses(value = {...})**: This annotation is used to specify the possible responses that the API operation can return. It contains an array of **@ApiResponse** annotations, each representing a specific response scenario.

**@ApiResponse(responseCode = "200", description = "Event retrieved successfully", content = {...})**: This is an **@ApiResponse** annotation that defines a specific response for the API operation.

**responseCode = "200"**: This indicates that this response corresponds to an HTTP status code 200, which typically signifies a successful request.

**description = "Event retrieved successfully"**: This is a human-readable description of what this response means. It states that the event was successfully retrieved.

**content = {...}**: This part specifies the content type and structure of the response. In this case, it specifies that the response content is in JSON format (**"application/json"**), and the structure of the response is defined by the schema specified in the **@Schema** annotation. The schema references the **EventResponse** class, indicating that the response will be a JSON object conforming to the structure defined in the **EventResponse** class.

In summary, these annotations provide documentation and metadata for an API operation that retrieves an event by its **eventId**. The operation is expected to return an HTTP 200 response with a JSON response body containing information about the retrieved event. This information can be valuable for both API developers and consumers to understand the operation and its expected outcomes.

@GetMapping(value = DocumentGeneratorEventStoreConstants.*SLASH* + DocumentGeneratorEventStoreConstants.*EVENT\_ENDPOINT* + DocumentGeneratorEventStoreConstants.*SLASH* + DocumentGeneratorEventStoreConstants.*EVENT\_ID\_PATH\_PARAM*)

**@GetMapping**: This annotation specifies that the annotated method should handle HTTP GET requests. In the context of a RESTful API, a GET request is used to retrieve data from the server.

**value**: This attribute of the **@GetMapping** annotation is used to define the URL path at which this GET request should be mapped. It specifies a string value that represents the URL path. In this case, the URL path is constructed by concatenating several constants and values together:

**DocumentGeneratorEventStoreConstants.SLASH**: This likely represents a constant that contains a forward slash (**/**). It's used to separate parts of the URL path.

**DocumentGeneratorEventStoreConstants.EVENT\_ENDPOINT**: This is likely another constant that represents an endpoint or resource name.

**DocumentGeneratorEventStoreConstants.EVENT\_ID\_PATH\_PARAM**: This appears to be another constant representing a path parameter in the URL. It captures the **eventId** from the URL. The value of this parameter would typically be provided dynamically in the URL when a request is made.

The concatenation of these constants and values creates a specific URL path for this GET request. The resulting URL path is where clients will send their GET requests.

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);  
}

**public ResponseEntity<EventResponse> retrieveEventByEventId(...)**: This is the method declaration. It indicates that the method is public and returns a **ResponseEntity** containing an object of type **EventResponse**. The method is named **retrieveEventByEventId**.

**final HttpServletRequest httpRequest**: This parameter is of type **HttpServletRequest**, which represents the HTTP request being handled. It allows you to access details about the incoming request, such as headers, parameters, and the request URL. However, in the provided method, it doesn't appear to be used.

**@NotNull @PathVariable final String eventId**: These annotations are used to define a path parameter named **eventId**. Path parameters are extracted from the URL, and the **@PathVariable** annotation is used to capture them. The **@NotNull** annotation is used to ensure that the **eventId** path parameter is not null. It is expected to be a string representing the identifier of the event to retrieve.

**LOG.info("Getting event data from eventId:: {}", GenericUtil.sanitizeValues(eventId));**: This line logs an informational message using a logger (e.g., Log4j, SLF4J). It prints out the **eventId** to provide information about the event being retrieved. The **GenericUtil.sanitizeValues(eventId)** call may be used to sanitize the **eventId** before logging it.

**Optional<EventResponse> eventDataResponse = documentGeneratorEventStoreService.fetchEventByEventId(eventId);**: This line invokes the **fetchEventByEventId** method on the **documentGeneratorEventStoreService**. It is responsible for fetching the event data based on the **eventId**. The result is wrapped in an **Optional**, indicating that the event data may or may not be present.

**return eventResponseMapper(httpRequest, eventDataResponse);**: This line returns a response generated by invoking the **eventResponseMapper** method. It passes the **httpRequest** (which is not used in the method) and the **eventDataResponse** (the result of fetching event data) to the mapper. The mapper is expected to convert this data into an appropriate **ResponseEntity** and return it.

In summary, this method handles GET requests to retrieve event data by **eventId**. It constructs a URL with path parameters, extracts the **eventId** from the path, logs information about the event being retrieved, fetches the event data, and uses a response mapper to construct and return an appropriate **ResponseEntity**

API to retrieve error details by eventId

@Operation(summary = "API to retrieve error details by eventId")

**@Operation(summary = "API to retrieve error details by eventId")**: This annotation, **@Operation**, is used to provide a summary or brief description of the API operation. In this case, it describes the operation as an API to retrieve error details by **eventId**. This summary is often used in API documentation to provide context about the purpose of the operation.

@ApiResponses(value = {@ApiResponse(responseCode = "200", description = "Error retrieved successfully",  
 content = {@Content(mediaType = "application/json",  
 schema = @Schema(implementation = EventResponse.class))})})

**@ApiResponses(value = {...})**: This annotation is used to specify the possible responses that the API operation can return. It contains an array of **@ApiResponse** annotations, each representing a specific response scenario.

**@ApiResponse(responseCode = "200", description = "Error retrieved successfully", content = {...})**: This is an **@ApiResponse** annotation that defines a specific response for the API operation.

**responseCode = "200"**: This indicates that this response corresponds to an HTTP status code 200, which typically signifies a successful request.

**description = "Error retrieved successfully"**: This is a human-readable description of what this response means. It states that the error details were retrieved successfully.

**content = {...}**: This part specifies the content type and structure of the response. In this case, it specifies that the response content is in JSON format (**"application/json"**), and the structure of the response is defined by the schema specified in the **@Schema** annotation. The schema references the **EventResponse** class, indicating that the response will be a JSON object conforming to the structure defined in the **EventResponse** class.

In summary, these annotations provide documentation and metadata for an API operation that retrieves error details by **eventId**. The operation is expected to return an HTTP 200 response with a JSON response body containing information about the retrieved error details. This information can be valuable for both API developers and consumers to understand the operation and its expected outcomes.

@GetMapping(value = DocumentGeneratorEventStoreConstants.*SLASH* + DocumentGeneratorEventStoreConstants.*EVENT\_ENDPOINT* + DocumentGeneratorEventStoreConstants.*SLASH* + DocumentGeneratorEventStoreConstants.*EVENT\_ID\_PATH\_PARAM* + DocumentGeneratorEventStoreConstants.*SLASH* + DocumentGeneratorEventStoreConstants.*ERROR\_ENDPOINT*)

**@GetMapping**: This annotation specifies that the annotated method should handle HTTP GET requests. In the context of a RESTful API, a GET request is used to retrieve data from the server.

**value**: This attribute of the **@GetMapping** annotation is used to define the URL path at which this GET request should be mapped. It specifies a string value that represents the URL path. In this case, the URL path is constructed by concatenating several constants and values together:

**DocumentGeneratorEventStoreConstants.SLASH**: This likely represents a constant that contains a forward slash (**/**). It's used to separate parts of the URL path.

**DocumentGeneratorEventStoreConstants.EVENT\_ENDPOINT**: This is likely another constant that represents an endpoint or resource name.

**DocumentGeneratorEventStoreConstants.EVENT\_ID\_PATH\_PARAM**: This appears to be another constant representing a path parameter in the URL. It captures the **eventId** from the URL.

**DocumentGeneratorEventStoreConstants.ERROR\_ENDPOINT**: This is likely yet another constant that represents an endpoint or resource name, specifically for handling error-related information.

The concatenation of these constants and values creates a specific URL path for this GET request. The resulting URL path is where clients will send their GET requests.

In summary, this **@GetMapping** annotation is used to map an HTTP GET request to a specific URL path. The URL path is constructed using constants and values representing different parts of the path, including the event endpoint, event ID path parameter, and error endpoint. This construction defines the URL where clients can send GET requests to retrieve error-related information for a specific event.

URL Example : [**http://localhost:8080/events/{eventId}/error**](http://localhost:8080/events/%7beventId%7d/error).

public ResponseEntity<EventResponse> retrieveErrorsResponseByEventId(  
 final HttpServletRequest httpRequest, @NotNull @PathVariable final String eventId) {  
 *LOG*.info("Getting errors data from eventId: {}", GenericUtil.*sanitizeValues*(eventId));

**public ResponseEntity<EventResponse> retrieveErrorsResponseByEventId(...)**: This is the method signature. It indicates that the method is public and returns a **ResponseEntity** containing an object of type **EventResponse**. The method is named **retrieveErrorsResponseByEventId**.

**final HttpServletRequest httpRequest**: This parameter is of type **HttpServletRequest**, which represents the HTTP request being handled. It allows you to access details about the incoming request, such as headers, parameters, and the request URL. However, in the provided method, it doesn't appear to be used.

**@NotNull @PathVariable final String eventId**: These annotations are used to define a path parameter named **eventId**. Path parameters are extracted from the URL, and the **@PathVariable** annotation is used to capture them. The **@NotNull** annotation is used to ensure that the **eventId** path parameter is not null. It is expected to be a string representing the identifier of the event for which you want to retrieve error-related information.

**LOG.info("Getting errors data from eventId: {}", GenericUtil.sanitizeValues(eventId));**: This line logs an informational message using a logger (e.g., Log4j, SLF4J). It prints out the **eventId** to provide information about the event for which error data is being retrieved. The **GenericUtil.sanitizeValues(eventId)** call may be used to sanitize the **eventId** before logging it, ensuring that it meets certain requirements or security standards.

In summary, this method handles an HTTP GET request to retrieve error-related information for a specific event identified by **eventId**. It extracts the **eventId** from the URL path, logs information about the event being retrieved, and is expected to return a response (of type **ResponseEntity<EventResponse>**) containing the error-related data for the event. The actual retrieval and response generation logic are not provided in the code snippet you've shared.

Optional<EventResponse> eventDataResponse =  
 documentGeneratorEventStoreService.fetchErrorsByEventId(eventId);  
 return eventResponseMapper(httpRequest, eventDataResponse);  
}

**Optional<EventResponse> eventDataResponse = ...**: This line invokes a method named **fetchErrorsByEventId** on the **documentGeneratorEventStoreService** object. It is responsible for fetching error-related information for a specific event identified by the **eventId** parameter. The result is stored in an **Optional<EventResponse>**. An **Optional** is a container that may or may not contain a value, in this case, an **EventResponse** object. If the event is found, the **Optional** will contain the response data; otherwise, it will be empty.

**return eventResponseMapper(httpRequest, eventDataResponse);**: This line returns a response generated by invoking the **eventResponseMapper** method. It takes two parameters:

**httpRequest**: This is an instance of **HttpServletRequest**, representing the HTTP request. However, it doesn't appear to be used within this code snippet.

**eventDataResponse**: This is the **Optional** containing the error-related information retrieved for the event.

The **eventResponseMapper** method is likely responsible for transforming the data in **eventDataResponse** into a proper HTTP response, typically a **ResponseEntity<EventResponse>**. Depending on the contents of **eventDataResponse**, it may construct a success response if data is present or an appropriate error response if data is absent or if an error occurred during the retrieval process.

In summary, this code snippet retrieves error-related information for a specific event and returns it as an **Optional<EventResponse>**. It then passes this **Optional** to the **eventResponseMapper** method, which is responsible for mapping the data into an HTTP response to be returned to the client.

**Method-9** API to interact with Database to save document service request status

@Operation(summary = "API to interact with Database to save document service request status.")  
@ApiResponses(value = {@ApiResponse(responseCode = "200", description = "Request status saved successfully",  
 content = {@Content(mediaType = "application/json",  
 schema = @Schema(implementation = DocumentRetrievalResponse.class))})})  
@PostMapping(  
 value = DocumentGeneratorEventStoreConstants.SLASH + DocumentGeneratorEventStoreConstants.INVOICE  
 + DocumentGeneratorEventStoreConstants.SLASH + DocumentGeneratorEventStoreConstants.DOCUMENT  
 + DocumentGeneratorEventStoreConstants.SLASH  
 + DocumentGeneratorEventStoreConstants.DOCUMENT\_ID\_PATH\_PARAM,  
 consumes = MediaType.*APPLICATION\_JSON\_VALUE*)  
public ResponseEntity<DocumentRetrievalResponse> createDocumentServiceRequestStatus(  
 @NotNull @RequestHeader(BOSConstants.CORRELATION\_ID\_HEADER) final String correlationId,  
 @NotNull @RequestHeader(BOSConstants.APPLICATION\_LABEL\_HEADER) final String applicationLabel,  
 @NotNull @PathVariable final UUID documentId,  
 @Valid @RequestBody final CreateDocumentServiceRequestStatus createDocumentServiceRequestStatus) {  
 *LOG*.debug("DocumentServiceRequest data correlation id: {}", correlationId);  
 Optional<DocumentRetrievalResponse> responseMetadataOptional =  
 documentGeneratorEventStoreService.saveDocumentServiceRequestStatus(documentId,  
 createDocumentServiceRequestStatus);  
 if (responseMetadataOptional.get().getCode().equals(HttpStatus.CREATED.value())) {  
 return ResponseEntity.status(HttpStatus.CREATED).contentType(MediaType.APPLICATION\_JSON)  
 .body(responseMetadataOptional.get());  
 }  
 return ResponseEntity.status(HttpStatus.OK).contentType(MediaType.APPLICATION\_JSON)  
 .body(responseMetadataOptional.orElse(null));  
}

The code you provided is a Java method annotated with OpenAPI annotations and seems to be part of a Spring-based web application. Let's break down the functionality and purpose of this method:

**Endpoint Description**:

@Operation(summary = "API to interact with Database to save document service request status.")

This annotation provides a summary description of the API endpoint. It explains that the endpoint is used to interact with the database to save the status of a document service request.

**API Responses**:

@ApiResponses(value = {@ApiResponse(responseCode = "200", description = "Request status saved successfully",

content = {@Content(mediaType = "application/json",

schema = @Schema(implementation = DocumentRetrievalResponse.class))})})

Top of Form

This annotation defines the possible responses that the API endpoint can return. In this case, it specifies that the endpoint can return an HTTP response with a status code of 200 (OK) and a description of "Request status saved successfully." The response content is expected to be in JSON format and should conform to the schema defined by the **DocumentRetrievalResponse** class.

**HTTP Method and Path**:

@PostMapping(

value = DocumentGeneratorEventStoreConstants.SLASH + DocumentGeneratorEventStoreConstants.INVOICE

+ DocumentGeneratorEventStoreConstants.SLASH + DocumentGeneratorEventStoreConstants.DOCUMENT

+ DocumentGeneratorEventStoreConstants.SLASH

+ DocumentGeneratorEventStoreConstants.DOCUMENT\_ID\_PATH\_PARAM,

consumes = MediaType.APPLICATION\_JSON\_VALUE)

This annotation specifies that the method is intended to handle HTTP POST requests. It also defines the URL path at which this endpoint is accessible. The path is constructed using constants and path parameters, where **DocumentGeneratorEventStoreConstants** likely contains string constants for various parts of the URL. The **consumes** attribute indicates that this endpoint consumes JSON data.

**Method Parameters**:

**@NotNull @RequestHeader(BOSConstants.CORRELATION\_ID\_HEADER) final String correlationId**: This parameter is a required request header with the name specified by **BOSConstants.CORRELATION\_ID\_HEADER**. It represents the correlation ID associated with the request.

**@NotNull @RequestHeader(BOSConstants.APPLICATION\_LABEL\_HEADER) final String applicationLabel**: This parameter is a required request header with the name specified by **BOSConstants.APPLICATION\_LABEL\_HEADER**. It represents the application label associated with the request.

**@NotNull @PathVariable final UUID documentId**: This parameter is a required path variable representing the document ID extracted from the URL.

**@Valid @RequestBody final CreateDocumentServiceRequestStatus createDocumentServiceRequestStatus**: This parameter is the request body, and it is validated using the **@Valid** annotation. It likely represents the data associated with the document service request.

**Request Headers and Path Variables:**

The method expects three required parameters in the request:

**correlationId**: A correlation ID passed as a request header.

**applicationLabel**: An application label passed as a request header.

**documentId**: A document ID extracted from the URL path as a path variable.

These parameters provide context and data for processing the request.

**Request Body:** The method also expects a request body named **createDocumentServiceRequestStatus**. This request body is marked as **@Valid**, indicating that it should be validated based on validation rules defined for the **CreateDocumentServiceRequestStatus** class.

**Method Logic**:

The method logs the correlation ID associated with the request.

It then calls the **documentGeneratorEventStoreService.saveDocumentServiceRequestStatus** method, passing the **documentId** and **createDocumentServiceRequestStatus** as arguments. This method is expected to save the status of the document service request and return an **Optional<DocumentRetrievalResponse>**.

**The logic is as follows:**

If the response code in **responseMetadataOptional** is equal to **HttpStatus.CREATED.value()** (i.e., 201 Created), it returns a **ResponseEntity** with an HTTP status of 201 (Created) and the response data in the body.

Otherwise, it returns a **ResponseEntity** with an HTTP status of 200 (OK) and the response data in the body.

Depending on the response returned by **documentGeneratorEventStoreService.saveDocumentServiceRequestStatus**, the method constructs and returns an appropriate **ResponseEntity**:

If the response code is **HttpStatus.CREATED.value()** (indicating a successful creation), it returns a response with an HTTP status of 201 (Created).

Otherwise, it returns a response with an HTTP status of 200 (OK).

In summary, this method defines an API endpoint for saving the status of a document service request. It expects certain headers and path variables, and it returns an appropriate response based on the outcome of the request. It also includes OpenAPI annotations to document the endpoint's purpose and expected responses.

@Test  
public void testCreateDocumentServiceRequestStatus\_Failure() throws Exception {  
 // Arrange  
 String correlationId = "test-correlation-id";  
 String applicationLabel = "test-app-label";  
 UUID documentId = UUID.*randomUUID*();  
 CreateDocumentServiceRequestStatus requestStatus = new CreateDocumentServiceRequestStatus();  
  
 // Create a sample response data indicating failure  
 DocumentRetrievalResponse response = new DocumentRetrievalResponse();  
 response.setCode(HttpStatus.*INTERNAL\_SERVER\_ERROR*.value());  
  
 // Mock the service method to return the response data  
 *when*(documentGeneratorEventStoreService.saveDocumentServiceRequestStatus(documentId, requestStatus))  
 .thenReturn(Optional.*of*(response));  
  
 // Act  
 ResponseEntity<DocumentRetrievalResponse> result = documentGeneratorEventStoreController.*createDocumentServiceRequestStatus*(  
 correlationId, applicationLabel, documentId, requestStatus);  
  
 // Assert  
 *verify*(documentGeneratorEventStoreService, *times*(1)).saveDocumentServiceRequestStatus(documentId, requestStatus);  
 *assertEquals*(HttpStatus.*OK*, result.getStatusCode());  
 *assertEquals*(response, result.getBody());  
}

**Test Annotation**: **@Test** is a JUnit annotation that marks a method as a test method. When you run your test suite, JUnit will execute all methods annotated with **@Test**.

**Method Signature**: This method is named **testCreateDocumentServiceRequestStatus\_Failure**. It's a conventional naming pattern for test methods to describe what is being tested. In this case, it seems to be testing the failure scenario of creating a document service request status.

**Test Setup (Arrange)**: This section sets up the necessary data and mocks for the test:

**String correlationId**, **String applicationLabel**, **UUID documentId**, and **CreateDocumentServiceRequestStatus requestStatus** are variables that store test data and input values required for the test.

**DocumentRetrievalResponse response** is a sample response object that you create to simulate a failure scenario. It has an HTTP status code indicating an internal server error (500).

**when(documentGeneratorEventStoreService.saveDocumentServiceRequestStatus(...))** is using Mockito to mock the behavior of a service method (**saveDocumentServiceRequestStatus**) that's being tested. It's configured to return an **Optional** containing the **response** object when called with specific arguments.

**Test Execution (Act)**: This section executes the code that's being tested:

**ResponseEntity<DocumentRetrievalResponse> result** is the result of calling a controller method (**createDocumentServiceRequestStatus**) with the provided input parameters. This is where you trigger the code you want to test.

**Assertions (Assert)**: In this section, you verify whether the actual results match the expected results:

**verify(documentGeneratorEventStoreService, times(1)).saveDocumentServiceRequestStatus(...)** verifies that the **saveDocumentServiceRequestStatus** method of the mocked service was called exactly once with the specified arguments during the test.

**assertEquals(HttpStatus.OK, result.getStatusCode())** checks whether the HTTP status code of the response (**result.getStatusCode()**) is equal to **HttpStatus.OK**. This verifies that the HTTP response status code is as expected.

**assertEquals(response, result.getBody())** compares the response body (**result.getBody()**) to the **response** object you created earlier. This checks if the controller method returned the expected response object.

Overall, this test method is testing the failure scenario of a controller method by mocking the behavior of a service method and then verifying that the controller behaves correctly when it receives the expected inputs. If everything is set up correctly, the test will pass, indicating that the controller handles the failure scenario as expected.

Method -10 Event Mapper

ResponseEntity<EventResponse> eventResponseMapper(final HttpServletRequest httpRequest,  
 final Optional<EventResponse> eventResponse) {  
 final TransactionContext context = (TransactionContext) httpRequest  
 .getAttribute(BOSConstants.TRANSACTION\_CONTEXT);  
  
 if (eventResponse.isPresent()) {  
 eventResponse.get().setCode(HttpStatus.*OK*.value());  
 eventResponse.get().setApplicationLabel(context.getApplicationLabel());  
 eventResponse.get().setCorrelationId(context.getCorrelationID());  
 eventResponse.get().setMessage(DocumentGeneratorEventStoreConstants.*SUCCESS*);  
  
 if (!Optional.*ofNullable*(eventResponse.get().getEventDataResponse()).isPresent()) {  
 eventResponse.get().setMessage(DocumentGeneratorEventStoreConstants.*NOT\_FOUND*);  
 }  
 }  
 return ResponseEntity.*status*(HttpStatus.*OK*).contentType(MediaType.*APPLICATION\_JSON*)  
 .body(eventResponse.orElse(null));  
}

**Get Transaction Context**: It retrieves the transaction context (**TransactionContext**) from the **HttpServletRequest** object. The transaction context likely contains information about the ongoing transaction, such as application labels and correlation IDs.

final TransactionContext context = (TransactionContext) httpRequest.getAttribute(BOSConstants.TRANSACTION\_CONTEXT);

**Check if EventResponse is Present**: It checks if the **eventResponse** parameter is present (not empty) using **eventResponse.isPresent()**. If it's present, it means there is a response to be processed.

**Set Response Details**: If the **eventResponse** is present, the method sets various properties of the response:

**eventResponse.get().setCode(HttpStatus.OK.value())**: It sets the HTTP status code of the **eventResponse** to 200 (OK).

**eventResponse.get().setApplicationLabel(context.getApplicationLabel())**: It sets the application label of the **eventResponse** based on the value retrieved from the transaction context.

**eventResponse.get().setCorrelationId(context.getCorrelationID())**: It sets the correlation ID of the **eventResponse** based on the value retrieved from the transaction context.

**eventResponse.get().setMessage(DocumentGeneratorEventStoreConstants.SUCCESS)**: It sets the message of the **eventResponse** to "SUCCESS".

**Check Event Data Response**: It checks if the **eventDataResponse** property of the **eventResponse** is not null using **Optional.ofNullable(...)**. If it is null, it sets the message of the **eventResponse** to "NOT\_FOUND".

if (!Optional.ofNullable(eventResponse.get().getEventDataResponse()).isPresent()) {

eventResponse.get().setMessage(DocumentGeneratorEventStoreConstants.NOT\_FOUND);

}

**Return ResponseEntity**: Finally, it creates a **ResponseEntity** object with an HTTP status of 200 (OK), a content type of JSON, and the body set to the **eventResponse** if it's present, or **null** if it's not present.

return ResponseEntity.status(HttpStatus.OK).contentType(MediaType.APPLICATION\_JSON)

.body(eventResponse.orElse(null));

In summary, this method takes an **EventResponse** object (which likely represents some kind of response data), updates its properties based on the transaction context and the presence of **eventDataResponse**, and wraps it in a **ResponseEntity** with an appropriate HTTP status and content type before returning it. This is a common pattern for mapping a response object to an HTTP response in a Spring application.

Top of Form

**Unit Test1:**

@Test  
public void testEventResponseMapper\_Success() {  
 // Mock the attribute value from BOSConstantsUtil  
 *when*(httpRequest.getAttribute(BOSConstantsUtil.*TRANSACTION\_CONTEXT*))  
 .thenReturn("TestTransactionContext");  
  
 // Create a sample EventResponse  
 EventResponse eventResponse = new EventResponse();  
 eventResponse.setEventDataResponse("SomeEventData");  
  
 Optional<EventResponse> optionalEventResponse = Optional.*of*(eventResponse);  
  
 // Act  
 ResponseEntity<EventResponse> result = documentGeneratorEventStoreController.eventResponseMapper(httpRequest, optionalEventResponse);  
  
 // Assert  
 *assertEquals*(HttpStatus.*OK*, result.getStatusCode());  
 *assertEquals*(MediaType.*APPLICATION\_JSON*, result.getHeaders().getContentType());  
 *assertEquals*(eventResponse, result.getBody());  
 *assertEquals*("TestTransactionContext", eventResponse.getApplicationLabel());  
 *assertNull*(eventResponse.getCorrelationId()); // In this test, correlationId is not set  
 *assertEquals*("SUCCESS", eventResponse.getMessage());  
}

**Mocking the Attribute Value:**

when(httpRequest.getAttribute(BOSConstantsUtil.TRANSACTION\_CONTEXT)).thenReturn("TestTransactionContext");: This line sets up a mock behavior for the httpRequest object. It mocks the behavior of httpRequest.getAttribute(BOSConstantsUtil.TRANSACTION\_CONTEXT) to return the string "TestTransactionContext" when called.

**Creating a Sample EventResponse:**

EventResponse eventResponse = new EventResponse();: This line creates a new instance of EventResponse, which is the response object that will be passed to the eventResponseMapper method.

eventResponse.setEventDataResponse("SomeEventData");: It sets the eventDataResponse property of the eventResponse object to "SomeEventData".

Optional<EventResponse> optionalEventResponse = Optional.of(eventResponse);: It wraps the eventResponse object in an Optional.

**Act (Method Invocation):**

ResponseEntity<EventResponse> result = documentGeneratorEventStoreController.eventResponseMapper(httpRequest, optionalEventResponse);: This line calls the eventResponseMapper method with the mocked httpRequest and the optionalEventResponse created in the previous steps. The result is stored in the result variable.

**Assertions (Verification):**

assertEquals(HttpStatus.OK, result.getStatusCode());: This assertion checks that the HTTP status code of the result is equal to HttpStatus.OK, which is expected for a successful response.

assertEquals(MediaType.APPLICATION\_JSON, result.getHeaders().getContentType());: This assertion checks that the content type of the response is JSON.

assertEquals(eventResponse, result.getBody());: This assertion checks that the body of the result matches the eventResponse object that was originally passed to the method. It verifies that the eventResponse object has been correctly mapped to the response body.

assertEquals("TestTransactionContext", eventResponse.getApplicationLabel());: This assertion checks that the ApplicationLabel property of the eventResponse object has been correctly set to "TestTransactionContext".

assertNull(eventResponse.getCorrelationId());: This assertion checks that the CorrelationId property of the eventResponse object is null. This verifies that the eventResponseMapper method does not set this property in this specific test case.

Overall, this test method is designed to verify that the eventResponseMapper method produces the expected ResponseEntity when provided with specific input parameters. It checks the HTTP status code, content type, response body, and properties of the eventResponse object after the mapping.

Unit test – 2

@Test  
public void testEventResponseMapper\_NotFound1() {  
 // Mock the attribute value from BOSConstantsUtil  
 *when*(httpRequest.getAttribute(BOSConstantsUtil.*TRANSACTION\_CONTEXT*))  
 .thenReturn("TestTransactionContext");  
  
 // Create an empty (not found) Optional<EventResponse>  
 Optional<EventResponse> optionalEventResponse = Optional.*empty*();  
  
 // Act  
 ResponseEntity<EventResponse> result = documentGeneratorEventStoreController.eventResponseMapper(httpRequest, optionalEventResponse);  
  
 // Assert  
 *assertEquals*(HttpStatus.*OK*, result.getStatusCode());  
 *assertEquals*(MediaType.*APPLICATION\_JSON*, result.getHeaders().getContentType());  
 *assertNull*(result.getBody()); // Since eventResponse is not present  
 *assertEquals*("TestTransactionContext", result.getHeaders().getFirst("X-Application-Label")); // Verify the application label header  
 *assertNull*(result.getHeaders().getFirst("X-Correlation-ID")); // Verify the correlation ID header  
 *assertEquals*("NOT\_FOUND", result.getHeaders().getFirst("X-Message")); // Verify the message header  
}

**Mocking the Attribute Value:**

when(httpRequest.getAttribute(BOSConstantsUtil.TRANSACTION\_CONTEXT)).thenReturn("TestTransactionContext");: This line sets up a mock behavior for the httpRequest object. It mocks the behavior of httpRequest.getAttribute(BOSConstantsUtil.TRANSACTION\_CONTEXT) to return the string "TestTransactionContext" when called.

**Creating an Empty Optional<EventResponse>:**

Optional<EventResponse> optionalEventResponse = Optional.empty();: This line creates an empty Optional<EventResponse>. This simulates a scenario where there is no EventResponse available, indicating a "not found" response.

**Act (Method Invocation):**

ResponseEntity<EventResponse> result = documentGeneratorEventStoreController.eventResponseMapper(httpRequest, optionalEventResponse);: This line calls the eventResponseMapper method with the mocked httpRequest and the empty optionalEventResponse. The result is stored in the result variable.

**Assertions (Verification):**

assertEquals(HttpStatus.OK, result.getStatusCode());: This assertion checks that the HTTP status code of the result is equal to HttpStatus.OK, which is expected for a successful response.

assertEquals(MediaType.APPLICATION\_JSON, result.getHeaders().getContentType());: This assertion checks that the content type of the response is JSON.

assertNull(result.getBody());: This assertion checks that the body of the result is null. This is expected because the optionalEventResponse is empty, indicating that there is no response body.

assertEquals("TestTransactionContext", result.getHeaders().getFirst("X-Application-Label"));: This assertion checks that the "X-Application-Label" header of the response contains the value "TestTransactionContext". This verifies that the application label header is correctly set.

assertNull(result.getHeaders().getFirst("X-Correlation-ID"));: This assertion checks that the "X-Correlation-ID" header of the response is null. This verifies that the correlation ID header is not set because there is no response.

assertEquals("NOT\_FOUND", result.getHeaders().getFirst("X-Message"));: This assertion checks that the "X-Message" header of the response contains the value "NOT\_FOUND". This verifies that the message header is correctly set to indicate a "not found" response.

Overall, this test method is designed to verify that the eventResponseMapper method produces the expected ResponseEntity when provided with an empty optionalEventResponse, indicating a "not found" response. It checks the HTTP status code, content type, absence of a response body, and the presence and values of specific headers in the response.