Class Name: DocumentGeneratorEventStoreController

Method 1 - saveEvent

```
    @Operation(summary = "Api to interact with database to add Event")
    @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))})})
    @PostMapping(value = DocumentGeneratorEventStoreConstants.SLASH
            + DocumentGeneratorEventStoreConstants.EVENT_ENDPOINT, consumes =
MediaType.APPLICATION_JSON_VALUE)
    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);

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

        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));
    }
```

Method explanation: This method is a Spring Boot controller method designed to handle POST requests related to saving an event:

Annotations

- @Operation(summary = "Api to interact with database to add Event"): This is an OpenAPI annotation that provides metadata for API documentation. It summarizes what the API endpoint does.

- @ApiResponses(...): Specifies the possible HTTP response codes and their associated descriptions. This also includes the schema for the response body. In this case, two responses are expected: one with a 201 HTTP status code (Event added successfully), and another with a 200 status code (Event already exists).

- @PostMapping(...): Specifies that this method handles POST requests. The `value` indicates the URL mapping, and `consumes = MediaType.APPLICATION_JSON_VALUE` specifies that the request body should be in JSON format.

Method Signature

- public ResponseEntity<EventResponse> saveEvent(@Valid @RequestBody final EventRequest eventRequest) throws JsonProcessingException: The method signature indicates that it returns a `ResponseEntity<EventResponse>` and takes a valid `EventRequest` object as input. `@Valid` triggers validation checks on the `EventRequest` object. The method may throw a `JsonProcessingException`.

Method Body

Pre-processing & Logging

- `String correlationId = GenericUtil.sanitizeValues(eventRequest.getCorrelationId());`: Sanitizes the correlation ID from the request body. This is generally done to remove malicious or unwanted characters.

- `LOG.info("Document Event data correlation id: {}", correlationId);`: Logs the sanitized correlation ID for tracking.

Service Call & Business Logic

- `Optional<EventResponse> optionalEventResponse = documentGeneratorEventStoreService.saveEvent(eventRequest);`: Calls the `saveEvent` method from a service class. This likely performs the business logic for saving the event and returns an `Optional<EventResponse>`.

Response Creation

- `if (optionalEventResponse.isPresent() && optionalEventResponse.get().getCode().equals(HttpStatus.CREATED.value()))`: Checks if the `Optional` object has a value and if the code in that value is 201 (created).

- `return ResponseEntity.status(HttpStatus.CREATED).contentType(MediaType.APPLICATION_JSON).body(optionalEventResponse.get());`: If the condition is met, it returns a `ResponseEntity` with a 201 status code, indicating that the event was created successfully.

- `return ResponseEntity.status(HttpStatus.OK).contentType(MediaType.APPLICATION_JSON).body(optionalEventResponse.orElse(null));`: If the above conditions aren't met, it returns a 200 status code, indicating that the event already exists. The response body will be either the `EventResponse` or `null` if the `Optional` is empty.

The above covers each line in detail.

Unit Test:

Writing unit tests for Spring Boot controller methods usually involves the use of mocking frameworks like Mockito and testing utilities provided by Spring Boot itself. Below is a skeleton test code snippet using JUnit and Mockito that tests the `saveEvent` method

```
import com.fasterxml.jackson.core.JsonProcessingException;
import org.junit.jupiter.api.Test;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import java.util.Optional;

import static org.mockito.Mockito.*;
import static org.junit.jupiter.api.Assertions.*;

public class YourControllerTest {

    @InjectMocks
    private YourController yourController;

    @Mock
    private DocumentGeneratorEventStoreService documentGeneratorEventStoreService;

    @Mock
    private GenericUtil genericUtil;

    @Test
    public void testSaveEvent_created() throws JsonProcessingException {
        // Arrange
```

```
        EventRequest eventRequest = new EventRequest();
        eventRequest.setCorrelationId("testCorrelationId");

        EventResponse eventResponse = new EventResponse();
        eventResponse.setCode(HttpStatus.CREATED.value());

        when(genericUtil.sanitizeValues(anyString())).thenReturn("testCorrelationId");
        when(documentGeneratorEventStoreService.saveEvent(any(EventRequest.class)))
                .thenReturn(Optional.of(eventResponse));

        // Act
        ResponseEntity<EventResponse> result = yourController.saveEvent(eventRequest);

        // Assert
        assertEquals(HttpStatus.CREATED, result.getStatusCode());
        assertEquals(eventResponse, result.getBody());
    }

    @Test
    public void testSaveEvent_alreadyExists() throws JsonProcessingException {
        // Arrange
        EventRequest eventRequest = new EventRequest();
        eventRequest.setCorrelationId("testCorrelationId");

        EventResponse eventResponse = new EventResponse();
        eventResponse.setCode(HttpStatus.OK.value());

        when(genericUtil.sanitizeValues(anyString())).thenReturn("testCorrelationId");
        when(documentGeneratorEventStoreService.saveEvent(any(EventRequest.class)))
                .thenReturn(Optional.of(eventResponse));

        // Act
        ResponseEntity<EventResponse> result = yourController.saveEvent(eventRequest);

        // Assert
        assertEquals(HttpStatus.OK, result.getStatusCode());
        assertEquals(eventResponse, result.getBody());
    }

    @Test
    public void testSaveEvent_noResponse() throws JsonProcessingException {
        // Arrange
        EventRequest eventRequest = new EventRequest();
        eventRequest.setCorrelationId("testCorrelationId");
```

```
        when(genericUtil.sanitizeValues(anyString())).thenReturn("testCorrelationId");
        when(documentGeneratorEventStoreService.saveEvent(any(EventRequest.class)))
                .thenReturn(Optional.empty());

        // Act
        ResponseEntity<EventResponse> result = yourController.saveEvent(eventRequest);

        // Assert
        assertEquals(HttpStatus.OK, result.getStatusCode());
        assertNull(result.getBody());
    }
}
```

Here's what the code is doing:

1. **@InjectMocks** and **@Mock** annotations are used to inject mock objects into the controller class that you are going to test.
2. Three test methods are written to test the different scenarios:

   - testSaveEvent_created: This test verifies that the controller method returns a 201 status code when an event is created.
   - testSaveEvent_alreadyExists: This test verifies that the controller method returns a 200 status code when an event already exists.
   - testSaveEvent_noResponse: This test verifies that the controller method returns a 200 status code and a `null` body when the service returns an empty `Optional`.

3. when(...).thenReturn(...) statements set up the mock behavior.
4. The `**assertEquals**` and `**assertNull**` methods from JUnit are used to verify if the actual result matches the expected result.

Please replace `YourController` with the actual name of your controller and modify the `EventRequest` and `EventResponse` classes to reflect their actual structures.

This is a basic template; we may need to add more comprehensive cases based on the business logic.

****

Method 2 - retrieveDocumentsByDocumentIds

```
@Operation(summary = "API to retrieve document details by documentId/documentIds")
```

```
@ApiResponses(value = {@ApiResponse(responseCode = "200", description = "response
Retrieved successfully",
      content = {@Content(mediaType = "application/json",
            schema = @Schema(implementation = DocumentIdsResponse.class))})})
@PostMapping(value = DocumentGeneratorEventStoreConstants.SLASH
      + DocumentGeneratorEventStoreConstants.INVOICE
      + DocumentGeneratorEventStoreConstants.SLASH
      + DocumentGeneratorEventStoreConstants.DOCUMENTS)
public ResponseEntity<DocumentIdsResponse> retrieveDocumentsByDocumentIds(
      final HttpServletRequest httpRequest,
      @Valid @RequestBody final DocumentIdsRequest documentIds,
      @NotNull @RequestHeader(BOSConstants.CORRELATION_ID_HEADER)
      final String correlationId,
      @NotNull @RequestHeader(BOSConstants.APPLICATION_LABEL_HEADER)
      final String applicationLabel) {
  List<DocumentResponse> documentResponse = documentGeneratorEventStoreService
        .fetchDocumentsByDocumentIds(documentIds);
  return new ResponseMapper().createResponseEntity(documentResponse, correlationId,
applicationLabel);
}
```

let's dive into each line of the method `retrieveDocumentsByDocumentIds`.

Annotations

- `@Operation(summary = "API to retrieve document details by documentId/documentIds")`: This annotation is part of the OpenAPI framework and provides metadata about the API. It indicates that this API retrieves document details based on document IDs.

- `@ApiResponses(value = {...})`: Also an OpenAPI annotation, it defines the possible responses that this API can return. In this case, it says that a successful call will return a 200 status code along with JSON data matching the schema of `DocumentIdsResponse`.

- `@PostMapping(...)`: This is a Spring annotation indicating that this method handles HTTP POST requests. The `value` attribute specifies the URL path pattern for this API endpoint.

Method Signature

- `public ResponseEntity<DocumentIdsResponse> retrieveDocumentsByDocumentIds(...)`: The method returns a `ResponseEntity` containing an instance of `DocumentIdsResponse`. The `***ResponseEntity***` is a Spring framework class used to define HTTP responses, including status code and body.

Parameters

- `*final HttpServletRequest httpRequest*`: The HTTP request object for obtaining details about the request like headers, client info, etc.

- `@Valid @RequestBody final DocumentIdsRequest documentIds`: This denotes that the request payload (body) should be automatically mapped to an object of type `DocumentIdsRequest`. The `@Valid` annotation ensures that the object meets certain validation constraints.

- `@NotNull @RequestHeader(BOSConstants.CORRELATION_ID_HEADER) final String correlationId`: Extracts a mandatory (non-null) request header named `CORRELATION_ID_HEADER` into a `String` variable.

- `@NotNull @RequestHeader(BOSConstants.APPLICATION_LABEL_HEADER) final String applicationLabel`: Similar to `correlationId`, this extracts another mandatory header into a `String`.

Business Logic
- `List<DocumentResponse> documentResponse = documentGeneratorEventStoreService.fetchDocumentsByDocumentIds(documentIds);`: Calls a service method `fetchDocumentsByDocumentIds` to fetch documents based on the document IDs given in the request. The result is stored in `documentResponse`.

Response
- `return new ResponseMapper().createResponseEntity(documentResponse, correlationId, applicationLabel);`: The ResponseMapper is likely a utility class used for consistently creating HTTP responses. This line creates a new response entity and returns it.

The method effectively handles an HTTP POST request to retrieve documents based on their IDs, validating the request, and delegating the data retrieval to a service class, finally returning the data as part of an HTTP response.

Writing Unit Test for the Method

Writing a unit test case for the method `**retrieveDocumentsByDocumentIds**` could involve mocking its dependencies, creating test data, and then verifying that it behaves as expected. Here's a simple example using JUnit and Mockito in a Spring environment:

```
import static org.mockito.Mockito.*;
import static org.hamcrest.Matchers.*;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

@RunWith(SpringRunner.class)
```

```
@WebMvcTest(YourController.class)
public class YourControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private DocumentGeneratorEventStoreService documentGeneratorEventStoreService;

    @Test
    public void testRetrieveDocumentsByDocumentIds() throws Exception {
        // Create test data
        DocumentIdsRequest documentIds = new DocumentIdsRequest();
        // Populate documentIds as needed

        List<DocumentResponse> documentResponseList = new ArrayList<>();
        // Populate documentResponseList as needed

        // Mocking service call

when(documentGeneratorEventStoreService.fetchDocumentsByDocumentIds(any(DocumentId
sRequest.class)))
                .thenReturn(documentResponseList);

        // Execute the API call and validate the response
        mockMvc.perform(post("/yourApiEndpoint")
                .contentType(MediaType.APPLICATION_JSON)
                .header("CORRELATION_ID_HEADER", "some-correlation-id")
                .header("APPLICATION_LABEL_HEADER", "some-application-label")
                .content(new ObjectMapper().writeValueAsString(documentIds)))
                .andExpect(status().isOk())
                .andExpect(content().contentType(MediaType.APPLICATION_JSON))
                .andExpect(jsonPath("$", hasSize(documentResponseList.size())));
    }
}
```

- `@WebMvcTest(YourController.class)`: Indicates that this test focuses on the MVC components of a Spring application.

- MockMvc mockMvc: A mock environment for server-side HTTP tests.

- @MockBean: Mocks the `DocumentGeneratorEventStoreService` bean used in the controller. This allows us to set expectations on its behavior without interacting with the actual database.

- when(...).thenReturn(...): Mockito's way of specifying what mock methods should return when called.

- mockMvc.perform(...): Simulates an HTTP request to the API.

- andExpect(...): Checks that the response meets certain conditions.

By following these examples,you may need to adapt it to better fit your specific circumstances.