This test class, SeatBookingControllerTest, is designed to test the behavior of the SeatBookingController in various scenarios related to seat booking. Here's an explanation of the important functions and annotations used in this test class:

Annotations

1. **@Mock**:
    - Used to create a mock object for SeatBookingService. The mock object allows you to simulate the behavior of the service without invoking actual business logic. In this case, it enables the test class to mimic different responses from the service during testing.
2. **@InjectMocks**:
    - This annotation is used to create an instance of the class under test, SeatBookingController, and inject the mocked SeatBookingService into it. It ensures that when the controller's methods are called, the mock service is used.
3. **@BeforeEach**:
    - This annotation marks the setUp() method to be run before each test method is executed. It prepares the test environment by initializing the mocks using MockitoAnnotations.openMocks(this).
4. **@Test**:
    - This annotation marks a method as a test case. Each method annotated with @Test is treated as a test that JUnit will execute.

Important Functions

1. **setUp()**:
    - This method is annotated with @BeforeEach and is used to initialize the mocks before each test case runs. It ensures that mock objects are properly created and injected into the class under test before any test methods are executed.
2. **testBookSeats_Success()**:
    - This test method simulates a successful seat booking. The service method bookSeats() is mocked to return a Booking object when called with specific parameters. It checks if the controller returns a 200 OK status with the correct Booking object in the response.
    - It also uses verify() to ensure that the service method was called exactly once.
3. **testBookSeats_SeatAlreadyBooked()**:
    - This test simulates the scenario where a seat is already booked. The service method bookSeats() is mocked to throw a SeatAlreadyBookedException. The controller's response is asserted to return 400 Bad Request with the exception message as the response body.
4. **testBookSeats_SeatLimitExceeded()**:

- In this test case, the service method is mocked to throw a SeatLimitExceededException, which simulates a scenario where a user exceeds the limit of seats they can book. The response is asserted to return 400 Bad Request with the relevant exception message.

5. **testBookSeats_CustomerNotFound()**:
    - This test simulates a case where the customer ID provided does not exist. The service method is mocked to throw a CustomerNotFoundException. The controller's response is checked to return 400 Bad Request with the message indicating the customer was not found.
6. **testBookSeats_InternalServerError()**:
    - This test handles the scenario where a generic error occurs. The service method is mocked to throw a generic Exception. The controller's response is expected to return 500 Internal Server Error with an appropriate error message.

Mockito Functions

1. **when()**:
    - Used to define the behavior of the mocked service method. It specifies what the mock should return or throw when called with specific arguments. For example:

java

Copy code

```
when(seatBookingService.bookSeats(customerId, seatIds)).thenReturn(mockBooking);
```

2. **thenThrow()**:
    - Used to specify that the mocked method should throw an exception when called. For instance:

java

Copy code

```
when(seatBookingService.bookSeats(customerId, seatIds)).thenThrow(new SeatAlreadyBookedException(...));
```

3. **verify()**:
    - This method is used to check if the mocked method was invoked a specified number of times and with the correct arguments. In this case:

java

Copy code

```
verify(seatBookingService, times(1)).bookSeats(customerId, seatIds);
```

Assertion Methods

1. **assertEquals()**:

- This method checks if two values are equal. It is used to verify the status code and the response body of the controller's response. For example:

java

Copy code

assertEquals(200, response.getStatusCodeValue());

assertEquals(mockBooking, response.getBody());

By utilizing these annotations and functions, the test class verifies that the SeatBookingController behaves as expected under various conditions (successful booking, seat already booked, seat limit exceeded, customer not found, and internal server error). The use of mocking ensures that the service layer is isolated, allowing focused testing of the controller.