Event Application Testing

**Event.calculatePrice()**

**Input Space Partitioning Testing for the method provided**

Method signature:

calculatePrice(int numTickets, int ticketsBookedSoFar, LocalDateTime bookingTime)

Characteristic definition:

C1: numTickets validity

* C1A: numTickets is null
* C1B: numTickets < 0
* C1C: numTickets > 0
* C1C: numTickets = 0

C2: ticketsBookedSoFar validity

* C2A: ticketsBookedSoFar is null
* C2B: ticketsBookedSoFar < 0
* C2C: ticketsBookedSoFar > 0
* C2D: ticketsBookedSoFar = 0

C3: bookingTime validity

* C3A: bookingTime is null
* C3B: bookingTime >= event.endTime
* C3C: event.startTime(minus 30days) < bookingTime < event.startTime
* C3D: bookingTime = event.startTime
* C3E: bookingTime < event.startTime(minus 30days)
* C3F: event.startTime < bookingTime < event.endTime

C4: maxTickets (object state)

* C4A: maxTickets = 0
* C4B: maxTickets > 0
* C4C: maxTickets < 0

C5: endTime (object state)

* C5A: this.endTime < now
* C5B: this.endTime > now
* C5C: this.endTime = now

C6: startTime (object state)

* C6A: this.startTime < now
* C6B: this.startTime = now
* C6C: this.startTime > now

C7: return value

* C7A: returns value = 0
* C7B: returns value < 0
* C7C: returns value >= max.double
* C7D: returns 0 < value < max.double

There it is the ISP table where I used Base Coverage to find the Expected Outputs.

A table of data with numbers and letters

AI-generated content may be incorrect.

Tests Made and Runned

A screenshot of a computer program

AI-generated content may be incorrect.

**Graph Coverage on the calculatePrice(params) function:**

Nodes:

1 2

2 3

A diagram of a network

AI-generated content may be incorrect.2 4

4 5

4 6

6 7

6 8

8 9

8 10

10 11

10 12

12 13

12 14

14 15

14 16

15 16

16 17

16 18

17 18

Initial node is 1, Ending nodes are 3 5 7 9 11 13 18

Now we did the Prime Paths to find the Test Requirements:

A screenshot of a computer

AI-generated content may be incorrect.

**LoyaltyUtils.getLoyaltyDiscount()**

**Input Space Partitioning**

Method definition:

public static String getLoyaltyLevel(int bookingCount)

Characteristics:

C1: bookingCount value

* C1A: bookingCount < 0
* C1B: bookingCount > 0
* C1C: bookingCount = 0
* C1D: bookingCount = null

C2: return value

* C2A: GOLD
* C2B: SILVER
* C2C: BRONZE
* C2D: REGULAR

A table of text with black letters

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

Logical Coverage is not that good to be made on this function, because there is no logical clause where we can test it.

**Graph Coverage:**

Graph nodes:  
1 2  
2 3  
2 4  
4 5  
4 6  
6 7  
6 8  
8 9  
8 10

Initial is 1, terminals are 3, 5, 7, 9, 10

A diagram of a network

AI-generated content may be incorrect.

Using Edge-Pair we got the following TR:

A screenshot of a computer

AI-generated content may be incorrect.

DU- Path Coverage from bookingCount (starting in 1, and ending in 2,4,6,8)

A white background with black text

AI-generated content may be incorrect.

**EventBookingService.bookEvent()**

**Input Space Partitioning with Base Choice Coverage**

Method Signature:

bookEvent(String attendeeName, String attendeeAddress, Long eventId, Long numTickets)

Characteristics:

C1: atendeeName is \_\_\_

C1A: atendeeName is null

C1B: atendeeName is empty

C1C: atendeeName is valid string

C2: atendeeAddress is \_\_\_

C2A: atendeeAddress is null

C2B: atende atendeeAddress is empty

C2C: atendeeAddress is valid string

C3: eventId is \_\_\_

C3A: eventId is null

C3B: min.Long <= eventId <= max.Long and exists in DB

C3C: min.Long <= eventId <= max.Long and does not exist in DB

C4: numTickets is \_\_\_

C4A: numTickets is null

C4B: min.Long <= numTickets <= max.Long

C5: Return Behavior

C5A: Valid booking is created and returned successfully

C5B: Exception is thrown because event is not found (RuntimeException)

C5C: InvalidArgumentException

C5D: Booking is created, but total price is 0.0 (if logic allows this edge case)

Here it is the table for base choice coverage  
A screenshot of a chart

AI-generated content may be incorrect.

Here it is an instance of a Mock, where we use hardcoded behavior of the repositories so we can test the method with ISP (Base Choice Coverage)

A screenshot of a computer

AI-generated content may be incorrect.

Tests created:

A screenshot of a computer program

AI-generated content may be incorrect.

**Mockito Behavior**

Also we introduced Mockito Behavior Testin when we were testing the functionality of the method, its cases and returns. We mocked the behaviors of the Service and Repository layers, allowing us to simulate specific scenarios and control the return values.

We used the following methods:

* verify(...) – a method that is checking interactions with your mocks
* when(…).then(…) – this method is setting behavior to the mocks

**EventService.isEventForAdultsOnly()**

**Logic Coverage applied to the method**

Method signature: public boolean isEventForAdultsOnly(Event event)

Predicate to test: (isBigEvent || isLocationOverused) && !isBadRated && isAtNight

Clauses:

a - isBigEvent

b - isLocationOverused

c - isBadRated

d - isAtNight

**Predicate written with clauses: (a | b) & !c & d**

A table with multiple letters

AI-generated content may be incorrect.

We are going to make tests, using all 3 possible coverages:

* RACC

A screenshot of a computer

AI-generated content may be incorrect.

**We get 7,8,9,11,15 rows as test cases rows**

A screenshot of a computer

AI-generated content may be incorrect.

* GACC

A screenshot of a computer

AI-generated content may be incorrect.

**We get 1,4,7,11,15 rows as test cases rows**

A screenshot of a computer

AI-generated content may be incorrect.

* CACC

A screenshot of a calculator

AI-generated content may be incorrect.

**We get 7,9,11,12,15 rows as test cases rows**

A screenshot of a computer

AI-generated content may be incorrect.

This **parameterized test** **class** demonstrates an excellent approach to testing complex business logic. The key insight is that instead of writing 10 separate test methods that would largely duplicate code, we created it as data-driven testing structure that:

1. Encapsulates test scenarios in a custom TestCase class
2. Provides easy way to test with different combinations of inputs
3. Maintains a single test method that handles all scenarios consistently

This pattern is especially useful when testing methods with multiple conditional branches, as it helps achieve comprehensive path coverage while avoiding code duplication and making the expected behavior transparent to anyone reading the tests.

**EventService.getDynamicPricesForAllEvents()**

**Graph Coverage**

Graph Nodes:

A diagram of a network

AI-generated content may be incorrect.1 2

2 10

2 3

3 4

3 6

6 8

6 7

8 9

7 9

4 5

9 5

5 2

Initial node is 1, and terminal node is 10

A screenshot of a computer code

AI-generated content may be incorrect.Now we are going to do Prime Paths and we get around 29 TR:

Now we are going to write the test cases:

**Test Cases for Prime Path Coverage**

**TC1: Valid Event – Not Sold Out**

Test Path: 1 → 2 → 3 → 6 → 8 → 9 → 5 → 2 → 10

Covers TR:

[1,2,3,6,8,9,5,2,10]

[3,6,8,9,5,2,10]

**TC2: Valid Event – Sold Out**

Test Path: 1 → 2 → 3 → 6 → 7 → 9 → 5 → 2 → 10

Covers TR:

[3,6,7,9,5,2,10]

[6,7,9,5,2,10]

**TC3: Null or Malformed Event**

Test Path: 1 → 2 → 3 → 4 → 5 → 2 → 3 → 6 → 8 → 9 → 5 → 2 → 10

Covers TR:

[3,4,5,2,3,6,8]

[4,5,2,3,6,8,9]

[1,2,3,4,5,2,3,6,8]

**TC4: Multiple Events – Sold Out and Not Sold Out**

Test Path: 1 → 2 → 3 → 6 → 7 → 9 → 5 → 2 → 3 → 6 → 8 → 9 → 5 → 2 → 10

Covers TR:

[6,7,9,5,2,3,6,8]

[7,9,5,2,3,6,8]

[5,2,3,6,7,9,5,2]

[6,8,9,5,2,3,6]

**TC5: Null + Sold Out + Valid**

Test Path: 1 → 2 → 3 → 4 → 5 → 2 → 3 → 6 → 7 → 9 → 5 → 2 → 3 → 6 → 8 → 9 → 5 → 2 → 10

Covers TR:

[4,5,2,3,6,7,9]

[4,5,2,3,6,7,9,5,2]

[8,9,5,2,3,6,7]

[9,5,2,3,6,7,9]

**TC6: All Types – Edge & Backtrack Paths**

Test Path: 1 → 2 → 3 → 4 → 5 → 2 → 3 → 6 → 7 → 9 → 5 → 2 → 3 → 4 → 5 → 2 → 10

Covers TR:

[3,4,5,2,10]

[2,3,4,5,2]

[5,2,3,4,5]

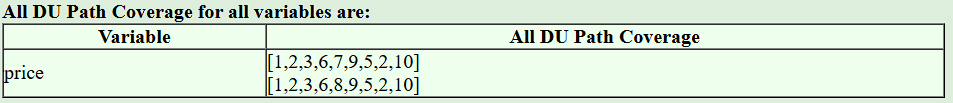
[1,2,10]

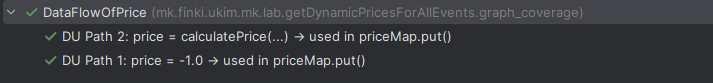
[4,5,2,3,4]

A screenshot of a computer

AI-generated content may be incorrect.

**Du Paths** coverage for the usage and definitions of **price** variable.





We inserted some parameterized tests that are going to test the method itself too.

A screen shot of a computer

AI-generated content may be incorrect.

**LoyaltyUtils. getLoyaltyDiscount**

Method Signature:

public static double getLoyaltyDiscount(Integer bookingCount)

**Input Space Partitioning with Base Choice**

Characteristics:

C1 – bookingCount

* C1A: bookingCount == null
* C1B: bookingCount <0
* C1C: bookingCount ==0
* C1D: bookingCount >0

C2 – value of return number(x)

* C2A: x==null
* C2B: x<0
* C2C: x==0.0
* C2D: 0.0<x<=0.1
* C2E: 0.1<x<=0.2
* C2F: 0.2<x<=0.3
* C2G: 0.3<x<=0.4
* C2H: 0.4<x<=0.5
* C2J: x>0.5

For best choice we choose C1D C2D – T

Below, we added the table where we see all combinations for Base Choice method.

A screenshot of a computer program

AI-generated content may be incorrect.

So we wrote all the test combinations that are possible to distinguish (total of 8)

A screen shot of a computer

AI-generated content may be incorrect.

**Graph Coverage**

On the picture below we can see the graph we got

**A diagram of a network

AI-generated content may be incorrect.**1 2

2 3

2 4

4 5

4 6

6 7

6 8

8 9

8 10

10 11

10 12

12 13

12 14

14 15

14 16

Initial node is 1, and terminal nodes are 3 5 7 9 11 13 15 16

We used **prime paths** to design our test cases, aiming to cover all important and independent paths through the graph for stronger logic coverage. We got the following Test Requirements:

**A screenshot of a graph

AI-generated content may be incorrect.**

A screenshot of a computer program

AI-generated content may be incorrect.

Then we used Du-Paths for the bookingCount variable and we got the following coverage result:

**A screenshot of a computer

AI-generated content may be incorrect.**

From this, we test the variable defs and uses with the following tests:

A screenshot of a computer program

AI-generated content may be incorrect.

A white and black list with black text

AI-generated content may be incorrect.We wrote several parameterized unit tests and pure unit tests so we can test the method logic itself too.

**Location.isValidLocation()**

Method Signature:

public boolean isValidLocation()

**Logic Coverage**

This method contains a clear structure that precisely fits for testing Logic Coverage, because has a predicate with several clauses.

Clause:

A – name.isBlank()

B – address.isBlank()

C – capacity.matches(‘’\\d+’’)

D – description.length() >= 10

**Predicate: P = !A ∧ !B ∧ C ∧ D**

Here it is the truth table provided:

A table of data with letters

AI-generated content may be incorrect.

We tried GACC and RACC in the function and we got the following results:

* GACC

A table of test results

AI-generated content may be incorrect.

Tests chosen: (31,21), (21,13), (25,13), (13,15), (13,14) - (13,14,15,21,25,31)

A screenshot of a computer

AI-generated content may be incorrect.

* RACC

A screenshot of a test

AI-generated content may be incorrect.

Tests chosen: (29,30), (29,31),(25,29),(21,29),(9,25) – (9, 21, 25, 29, 31)

A screenshot of a computer program

AI-generated content may be incorrect.

We wrote some base tests too, with parameters in them. On the table bellow you can see a description for the tests and which approach we used to create them

A close-up of a list of text

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

**BookingService.addToCart()**

Method Signature:

BookingCart addToCart(String selectedEvent, Integer numTickets, String username, String address, HttpSession session)

**Input Space Partitioning with Base Choice Coverage**

Characteristics:

C1 - selectedEvent

* C1A - selectedEvent==null
* C1B - selectedEvent== “ ”
* C1C – selectedEvent != “ ”

C2 - numTicketsC2A - numTickets==null

* C2B - numTickets < 0
* C2C – numTickets== 0
* C2D– numTickets> 0

C3 - username

* C3A - username==null
* C3B - username== “ ”
* C3C – username != “ ”

C4- address

* C4A - address==null
* C4B - address == “ ”
* C4C – address!= “ ”

C5- address

* C5A - session==null
* C5B - session!= null

Best choice: C1C C2D C3C C4C C5B

Table:

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

**Graph Coverage**

0 1

1 2

A diagram of a network

AI-generated content may be incorrect.1 15

2 3

2 16

3 4

3 17

4 5

4 18

5 6

5 19

6 7

7 8

7 20

8 9

9 10

9 21

10 14

Initial node is 0, terminal nodes are 14 15 16 17 18 19 20 21

A screenshot of a computer

AI-generated content may be incorrect.Graph Coverage with **Prime Paths**

A screenshot of a test results

AI-generated content may be incorrect.

Graph Coverage with **All Du-Paths Coverage for event**

A close-up of a label

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

Same as all the methods, we wrote some base tests so we can test the functionality of the method and its behavior on different scnearios

A screenshot of a program

AI-generated content may be incorrect.

**EventService.save\_event()**

Method Signature:

void save\_event(Long id, String name, String description, double popularityScore, Long locationID, LocalDateTime from, LocalDateTime to,double basePrice, int maxTickets)

**Graph Coverage**

0 1

1 2

1 3

3 4

A diagram of a network

AI-generated content may be incorrect.3 5

5 6

5 7

7 8

7 9

9 10

9 11

11 12

11 13

13 14

13 15

15 16

15 17

17 18

18 19

19 20

20 21

21 22

22 23

Initial node is 0, and terminal are 2 4 6 8 10 12 14 16 23

A screenshot of a computer

AI-generated content may be incorrect.First we are going to introduce the realisation of GC with **Prime Paths**

A screenshot of a computer

AI-generated content may be incorrect.

And we do with **All Du-Path coverage for name** variable

A close-up of a label

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

In addition to structural coverage techniques, we also designed a set of base tests to validate method behavior under both valid and invalid input conditions. These tests help ensure that the method respond appropriately to various edge cases, including incorrect or unexpected inputs.

A screenshot of a computer program

AI-generated content may be incorrect.

LoginService.simulateBookings()

During the testing of this application, we were using several times @Mock, so we can mock a behavior to some repository mostly (as an interface) and to check some cases while the testing.

Now we chose this function to be tested with Mocks only. The method simulates a random number (0–59) of event bookings for a given user. It selects random events from the database and creates EventBooking entries using a booking repository.

The usage of mockito is in:

1. Mock EventRepository and EventBookingRepository
2. Simulate scenarios like:

* Available events
* No events
* Single event reuse
* Null usernames

1. Verify how many times bookings are saved
2. Assert correctness of fields (username, address, price)

A screenshot of a computer

AI-generated content may be incorrect.

**Spring MVC Test framework**

**EventBookingControllerTest**

The EventBookingControllerTest class verifies the behavior of the EventBookingController using the Spring MVC Test framework.

Now we are going to list the scenarios we covered with this method:

1. /cart/add – Add to Cart

This endpoint tests the functionality of adding an event to the booking cart. It simulates a valid booking by mocking selectedEvent, numTickets, username, and address, then verifies redirection to /cart/view and checks that the cart is correctly stored in the session.

1. /cart/view – View Cart

The /cart/view endpoint is tested under two conditions.

* First, it simulates a session where the cart already exists and contains booking items, *verifying that the model includes the existing cart*.
* Second, it handles the case where *no cart is present in the session, asserting that a new empty BookingCart is initialized and correctly added to the model*.

1. /cart/confirm – Confirm Cart

The /cart/confirm endpoint is tested by simulating a session that contains a populated BookingCart. The test mocks the eventBookingService.saveBooking() method to handle the booking process. It then verifies that the user is redirected to the bookingConfirmation page. Additionally, the test asserts that key session attributes—such as username, attendeeAddress, numTickets, selectedEvent, and totalPrice—are correctly set during the confirmation flow.

**Tools & Setup for testing the controllers**

* **@WebMvcTest**(EventBookingController.class) to test controller in isolation
* **MockMvc** to simulate HTTP requests/responses
* **Mockito to mock dependencies**:
  + EventService
  + EventBookingService
  + BookingCartService
* .**with(csrf())** and **.with(user(...))** for authenticated (secured requests)

**EventControllerTest**

The EventControllerTest class provides unit tests for the EventController, which manages event-related operations in the application such as listing, filtering, adding, editing, and deleting events.

|  |  |  |
| --- | --- | --- |
| Method | Purpose | Covered Scenarios |
| testGetEventsPage() | Tests the /events GET endpoint | Ensures events are loaded into the model and the correct view (listEvents) is returned. |
| testFilterList() | Tests the /events/filter\_events POST endpoint | Verifies that filtering by text and rating works, the correct model attributes are added (event\_list, bookedTickets, eventPrices), and view is listEvents. |
| testAddNewEvent() | Tests /events/add-form GET endpoint (admin only) | Verifies the add event form is returned with a list of all locations. |
| testDeleteEventFromList() | Tests /events/delete/{id} GET endpoint (admin only) | Confirms that an event can be deleted and user is redirected to /events. |
| testSaveEvent() | Tests /events/add POST endpoint | Verifies a new or updated event is saved and the user is redirected to the event list. |
| testBookEvent() | Tests /events/book\_event POST endpoint | Ensures that booking an event stores the user, ticket number, and event name in session and redirects to /eventBooking. |
| testEditEvent() | Tests /events/edit/{id} GET endpoint (admin only) | Verifies that the edit form is pre-filled with event and location data. |

**LocationControllerTest**

The LocationControllerTest class verifies the behavior of the LocationController, which manages HTTP requests for creating, listing, and deleting event locations. It uses MockMvc and Mockito to simulate and validate the controller’s behavior without starting a full web server.

|  |  |
| --- | --- |
| Test Method Name | Description |
| testLocationsPage() | Tests GET /locations endpoint. Ensures the list of locations is fetched and available as location\_list in the model. |
| testDeleteLocationFromList() | Tests GET /locations/delete/{id}. Confirms that the location is deleted and redirects to the /locations page. |
| testAddNewLocation() | Tests GET /locations/add-loc. Validates that the add-location form view is correctly returned. |
| testSaveLocation() | Tests POST /locations/add. Validates saving a location with the correct parameters, and redirection to the /locations page. |

**LoginControllerTest**

This test verifies that the login page is rendered correctly. Since the application does not implement actual login logic, this is the only test needed.

**The only scenario that is covered here is the GET /login scenario**, that is testing the user has no session on the application, to redirect him on the login page.

**LogoutControllerTest**

The LogoutControllerTest verifies the behavior of the logout route in the application. It checks whether accessing the /logout endpoint triggers a redirection to the /login page as expected. This controller does not perform actual logout logic but serves as a redirection handler post-logout. The test confirms if accessing GET /logout returns a 3xx redirection status and the redirection to /login.

***Conclusion for Spring MVC Testing Framework***

The **Spring MVC Test framework** provides a powerful and efficient way to test Spring web applications by allowing developers to simulate HTTP requests and verify controller behavior without deploying the application to a server. It supports thorough testing of request mappings, model attributes, view resolution, and session handling.