FPT University - October 2025

UNIT TESTING WITH AI PROMPT

BOOKING CREATION FEATURE

Presented by AI_13

OUR TEAM

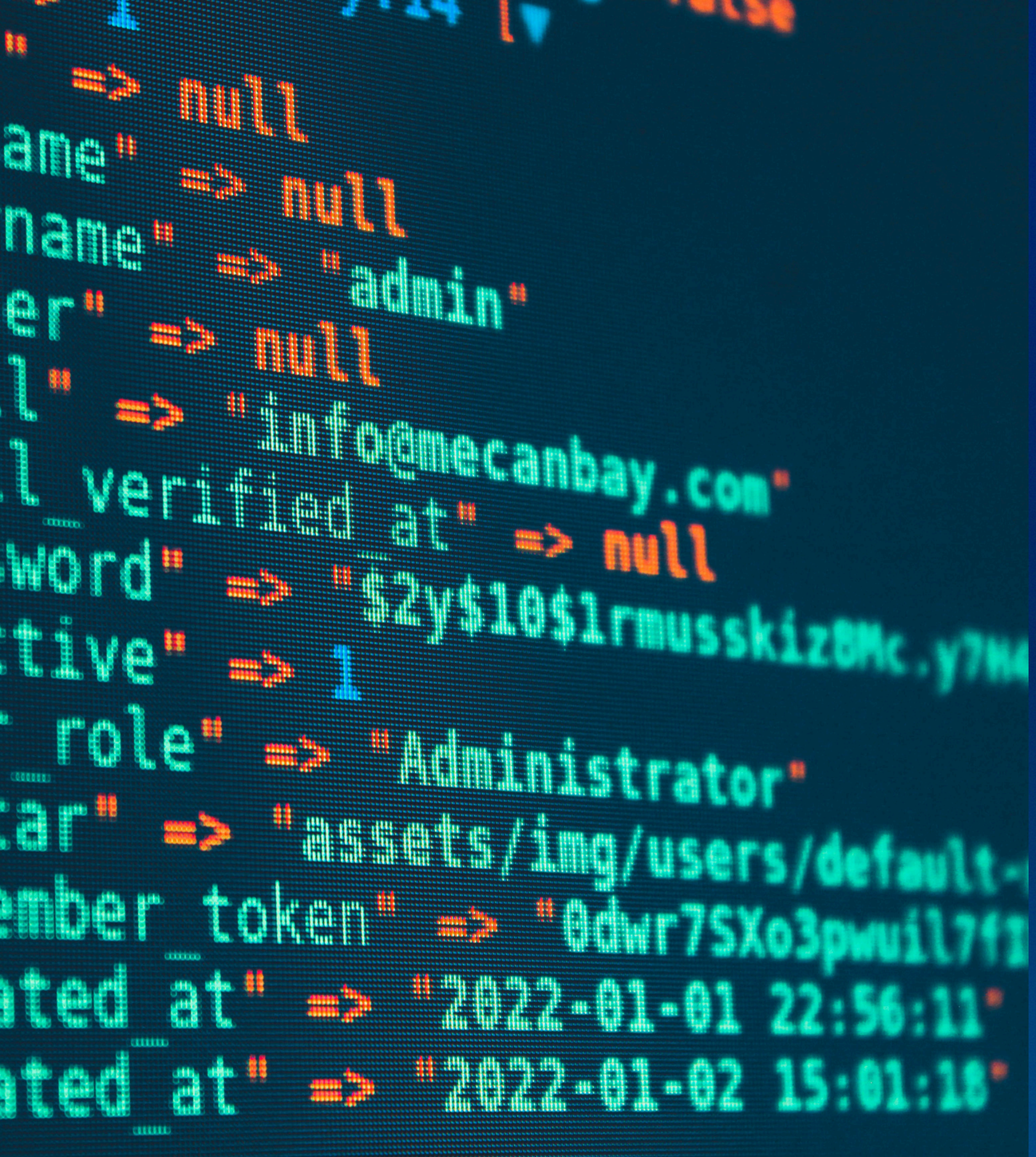
Thuy Duong

Xuan Thinh

Hoang Phuc

Gia Khiem

Phuong Uyen



OUTLINE

What We'll Be Presenting

01. Project Overview
02. Feature Analysis & Selection
03. Test Case Design
04. Generate Test Code
05. Debug Tests
06. Mocking & Optimization
07. Documentation & Demo

PROJECT OVERVIEW

Title: Unit Testing with AI – Create Booking Feature

Scope: Booking module → function **createBooking()**

Goal:

- Minimum 15 Test Cases
- ≥ 80% Code Coverage
- 180 minutes with AI prompts support

Tech: Node.js / Jest / Mock Services

FEATURE ANALYSIS & SELECTION

Core Feature: Booking Creation

Selected Reason: all booking operations depend on creation.

- Contains independent, testable logic: validations, price calculations, promotions, services.
- Easy to mock dependencies (DAOs, system settings) for deterministic tests.
- High impact: errors here affect downstream features.
- Supports boundary and error scenario testing.

FEATURE ANALYSIS & SELECTION

Functions need test:

- validateRequiredEntities()
- validateHall()
- validateEventDate()
- validateTimeSlot()
- validateDishes()
- validateDishCategories()
- validateServices()
- calculateBasePrice()
- applyPromotions()
- applyPaidServices()
- calculateTotals()

PROMPT 1

Input Prompt

"Analyze this Booking class and identify all functions that need unit testing: [Booking.js] For each function, identify:

1. Main functionality
2. Input parameters and types
3. Expected return values
4. Potential edge cases
5. Dependencies that need mocking Respond using concise bullet points for each function, covering all key details."

PROMPT 1

🧩 **1. validateRequiredEntities({ customer, eventType, hall, menu })**

Functionality: Ensures all required foreign key entities exist and have valid IDs. **Inputs:**

- `customer`, `eventType`, `hall`, `menu` → objects (each should contain an `ID` field). **Returns:**
- `void` (throws `Error` on invalid/missing entities). **Edge Cases:**
- Missing or `null` entity object.
- Missing `ID` field (`customerID`, `hallID`, etc.). **Dependencies to mock:**
- None (pure logic, no external calls).

🧩 **2. validateHall(hall)**

Functionality: Checks if booking's `tableCount` is within hall's min–max limits. **Inputs:**

- `hall`: object `{ minTable: number, maxTable: number }`. **Returns:**
- `void` (throws `Error` if invalid). **Edge Cases:**
- `tableCount` < `minTable` or > `maxTable`.
- Missing hall properties. **Dependencies to mock:**
- None.

🧩 **3. validateEventDate(systemSettings)**

Functionality: Verifies event date is valid, within allowed time range (≥ min notice hours and ≤ 1 year). **Inputs:**

- `systemSettings`: object `{ DEFAULT_MIN_BOOKING_NOTICE_HOURS?: string|number }`. **Returns:**
- `void` (throws `Error` if date invalid or out of range). **Edge Cases:**
- `eventDate` not in ISO format.
- `eventDate` before current + minNoticeHours.
- `eventDate` > 1 year ahead.
- Invalid or missing `systemSettings`. **Dependencies to mock:**
- `date-fns`: `addHours`, `addYears`, `parseISO`, `isBefore`.
- `Date.now()` (mock system time).

PROMPT 1

🧩 **4. validateTimeSlot(isHallAvailable)**

Functionality: Ensures time slot is valid and hall is available. **Inputs:**

- `isHallAvailable`: boolean. **Returns:**
- `void` (throws `Error` if invalid slot or unavailable). **Edge Cases:**
- Missing `startTime` or `endTime`.
- Invalid time slot not matching predefined slots.
- `isHallAvailable = false`. **Dependencies to mock:**
- None.

🧩 **5. validateDishes(dishesInMenu)**

Functionality: Validates selected dishes exist in menu and meet category rules. **Inputs:**

- `dishesInMenu`: array of dish objects `{ dishID, categoryID, requiredQuantity? }`. **Returns:**
- `void` (throws `Error` if missing/invalid dishes). **Edge Cases:**
- Empty or missing `this.dishes`.
- Dish ID not in menu.
- Category missing required dish count. **Dependencies to mock:**
- `validateDishCategories` (can spy/mock to isolate logic).

🧩 **6. validateDishCategories(dishesInMenu)**

Functionality: Ensures per-category minimum dish requirements are satisfied. **Inputs:**

- `dishesInMenu`: array as above. **Returns:**
- `void` (throws `Error` if category requirement unmet). **Edge Cases:**
- Category with requiredQuantity > selected dishes.
- Dish not found in menu. **Dependencies to mock:**
- None.

PROMPT 1

🧩 7. validateServices(allServices)

Functionality: Validates selected services exist and have valid quantities. **Inputs:**

- `allServices`: array of service objects `{ serviceID, price?, ... }`. **Returns:**
- `void` (throws `Error` for missing/invalid services). **Edge Cases:**
- Invalid `serviceID`.
- `quantity <= 0`.
- Empty service list. **Dependencies to mock:**
- None.

🧩 8. calculateBasePrice(hall, menu)

Functionality: Calculates base price = hall.price + (menu.price × tableCount). **Inputs:**

- `hall`: `{ price: number }`
- `menu`: `{ price: number }` **Returns:**
- Updates `this.originalPrice`. **Edge Cases:**
- Missing hall or menu price.
- `tableCount` = 0 or negative. **Dependencies to mock:**
- None.

🧩 9. applyPromotions(promotions)

Functionality: Applies promotions → updates discounts or free services. **Inputs:**

- `promotions`: array of promo objects `{ discountType, discountValue, minTable?, freeServiceID? }`. **Returns:**
- Updates `discountAmount`, `promotions`, and private `#freeServices`. **Edge Cases:**
- Empty promotion list.
- `discountType` invalid.
- `tableCount` below `minTable`.
- Multiple overlapping promotions. **Dependencies to mock:**
- None (internal logic only).

PROMPT 1

🧩 **10. applyPaidServices(allServices)**

Functionality: Adds paid service costs to `originalPrice`, ensuring no free duplicates. **Inputs:**

- `allServices`: array of `{ serviceID, price }`. **Returns:**
- Updates `services` and `originalPrice`. **Edge Cases:**
- Service marked as free (in `#freeServices`).
- Missing service info in `allServices`.
- Invalid quantity or price. **Dependencies to mock:**
- Access to private `#freeServices` (mock indirectly by calling `applyPromotions`).

🧩 **11. calculateTotals(systemSettings)**

Functionality: Computes final totals after discounts and VAT. **Inputs:**

- `systemSettings`: `{ VAT_RATE?: string|number }`. **Returns:**
- Updates `VAT` and `totalAmount`. **Edge Cases:**
- Missing VAT rate (defaults to 0.08).
- Negative total after discount.
- Large monetary values. **Dependencies to mock:**
- None (math-only logic).

✅ **Summary of Test Coverage**

Category	Functions
Validation	`validateRequiredEntities`, `validateHall`, `validateEventDate`, `validateTimeSlot`, `validateDishes`, `validateDishCategories`, `validateServices`
Calculation	`calculateBasePrice`, `applyPromotions`, `applyPaidServices`, `calculateTotals`
Mocking needed	`date-fns` (date logic), `Date.now()`, and optional spies for private sets or sub-validation calls

TEST CASE DESIGN

PROMPT 2

Input Prompt

"Generate comprehensive unit test cases for Booking's validateRequiredEntities() function:

async validateRequiredEntities() { const customer = await this.customerDAO.findById(this.customerID); const eventType = await this.eventTypeDAO.findById(this.eventTypeID); const hall = await this.hallDAO.findById(this.hallID); const menu = await this.menuDAO.findById(this.menuID);

```
    if (!customer) throw new Error("Customer record not found in database.");
    if (!eventType) throw new Error("Event type record not found in database.");
    if (!hall) throw new Error("Hall record not found in database.");
    if (!menu) throw new Error("Menu record not found in database.");
}
```

Include:

- Happy path scenarios
- Edge cases (boundary values)
- Error scenarios I just want test cases, don't generate source code. Use static data objects for customer, eventType, hall, and menu instead of mocking any DAO calls."

PROMPT 2

✅ Test Case Matrix for `validateRequiredEntities()`

#	Scenario	Input Setup	DAO Mock Behavior	Expected Outcome
1	✅ All entities found (Happy Path)	All `*ID` properties set (e.g., `customerID=1`, `eventTypeID=2`, etc.)	Each DAO returns a valid object	Function resolves successfully (no errors thrown)
2	❌ Missing customer record	Valid IDs for all entities	`customerDAO.findById()` returns `null`; others return valid objects	Throws `"Customer record not found in database."`
3	❌ Missing event type record	Valid IDs for all entities	`eventTypeDAO.findById()` returns `null`; others valid	Throws `"Event type record not found in database."`
4	❌ Missing hall record	Valid IDs for all entities	`hallDAO.findById()` returns `null`; others valid	Throws `"Hall record not found in database."`
5	❌ Missing menu record	Valid IDs for all entities	`menuDAO.findById()` returns `null`; others valid	Throws `"Menu record not found in database."`
6	❌ All entities missing	Valid IDs	All DAO `findById()` calls return `null`	Throws first missing entity error (`"Customer record not found in database."`)
7	❌ Invalid `customerID` (e.g., undefined/null)	`customerID=null`	DAO not called or returns `null`	Throws `"Customer record not found in database."`

PROMPT 2

8	❌ Invalid `eventTypeID` (e.g., 0 or undefined)	`eventTypeID=undefined`	DAO not called or returns `null`	Throws `"Event type record not found in database."`
9	❌ Invalid `hallID`	`hallID=undefined`	DAO not called or returns `null`	Throws `"Hall record not found in database."`
10	❌ Invalid `menuID`	`menuID=undefined`	DAO not called or returns `null`	Throws `"Menu record not found in database."`
11	⚠️ DAO throws unexpected error	Valid IDs	One DAO throws (e.g., `throw new Error("DB connection failed")`)	Propagates that error (not caught by Booking)
12	⚠️ Mixed valid & missing entities (boundary validation)	Valid IDs	`customer`, `eventType` valid; `hall`, `menu` missing	Throws `"Hall record not found in database."` (first missing in order)
13	⚠️ DAO returns malformed entity (e.g., empty object)	Valid IDs	DAO returns `{}` (truthy but invalid structure)	Function resolves successfully (since only falsy check is done) — potential future enhancement
14	⚠️ All DAOs resolve slowly (async timing test)	Valid IDs	DAOs return promises that resolve after delays	Function still resolves successfully when all fulfill

GENERATE TEST CODE

PROMPT 3

Input Prompt

"I want to write unit tests for the Booking class. The class depends on DAO (data access objects) to fetch entities like Customer, Hall, Menu, etc. I don't want to hit the real database in the tests. I want to verify that Booking throws errors correctly or proceeds normally depending on the DAO output. Give me a correct test case matrix for validateRequiredEntities() function."

PROMPT 4

Input Prompt

"Create Jest unit tests for Booking's validateHall() function with these test cases:

1. Table count within range
2. Table count equals minTable
3. Table count equals maxTable
4. Below minimum table count
5. Above maximum table count
6. tableCount missing (undefined)
7. tableCount not a number (string)
8. tableCount NaN
9. Hall not found (DAO returns null) Requirements:

- Use Jest framework
- Include setup/teardown
- Use proper assertions (toEqual, toThrow)
- Add descriptive test names
- Mock any external dependencies"

PROMPT 5

Input Prompt

""Create Jest unit tests for Booking's validateTimeSlot() function with these test cases:

1. Buổi trưa — start/end auto-filled
2. Buổi tối — start/end auto-filled
3. Valid explicit times (Buổi trưa)
4. Valid explicit times (Buổi tối)
5. Lower regex boundary (00:00:00)
6. Upper regex boundary (23:59:59)
7. Invalid startTime format
8. Invalid endTime format
9. Valid format but not matching predefined slot
10. Valid times but hall unavailable

Requirements:

- Use Jest framework
- Include setup/teardown
- Use proper assertions (toEqual, toThrow)
- Add descriptive test names
- Mock any external dependencies"

PROMPT 6

Input Prompt

"Create Jest unit tests for Booking's validateDish() function with these test cases:

1. All selected dishes exist in menu
2. Single dish selection (minimum valid)
3. Empty dishes array
4. Dishes is undefined
5. Dishes is not an array (string)
6. Contains dish not in menu
7. Valid dishes but validateDishCategories fails internally

Requirements:

- Use Jest framework
- Include setup/teardown
- Use proper assertions (toEqual, toThrow)
- Add descriptive test names
- Mock any external dependencies"

PROMPT 7

Input Prompt

"Create Jest unit tests for Booking's validateDishCategories() function with these test cases:

1. All categories satisfied
2. Single category, exact required quantity
3. Extra dishes beyond required
4. Duplicate dish selection in same category

Requirements:

- Use Jest framework
- Include setup/teardown
- Use proper assertions (toEqual, toThrow)
- Add descriptive test names
- Mock any external dependencies"

PROMPT 8

Input Prompt

Create Jest unit tests for Booking's calculateBasePrice() function with these test cases:

1. Valid input — normal calculation
2. Hall price is not a number (string)
3. Menu price is not a number (null)
4. tableCount is not a number (string)
5. tableCount = 0 (edge case)
6. tableCount = 1 (minimum valid)
7. Negative tableCount (invalid value)
8. Large tableCount (stress test)
9. Decimal prices (floating-point precision)
10. Missing hall or menu object

Requirements:

- Use Jest framework
- Include setup/teardown
- Use proper assertions (toEqual, toThrow)
- Add descriptive test names
- Mock any external dependencies

PROMPT 9

Input Prompt

"Create Jest unit tests for Booking's validateServices() function with these test cases:

1. All services valid
2. Empty services array
3. this.services undefined
4. this.services not an array
5. Service quantity = 0
6. Service quantity negative
7. Service quantity not a number
8. Service not in database

Requirements:

- Use Jest framework
- Include setup/teardown
- Use proper assertions (toEqual, toThrow)
- Add descriptive test names
- Mock any external dependencies"

PROMPT 10

Input Prompt

Create Jest unit tests for Booking's applyPromotions() function with these test cases:

1. Single valid percentage promotion — correctly calculates discountAmount and stores promotion.
2. Multiple valid promotions — cumulative percentage discounts applied correctly.
3. Free service promotion — adds freeServiceID to #freeServices without changing discountAmount.
4. Mixed promotions (percentage + free service) — verifies combined logic works correctly.
5. Promotion below minTable threshold — ignored (no discount applied).
6. Empty promotions array — discountAmount = 0 and promotions = [].
7. promotions = undefined — throws or handles gracefully (TypeError expected).
8. Invalid discountType or missing fields — promotion ignored.
9. Floating discountValue (e.g., 7.5%) — computed precisely with toBeCloseTo.
10. Duplicate promotions — both processed; verify additive stacking or duplication logic.

Requirements:

- Use Jest framework
- Include setup/teardown with beforeEach / afterEach
- Use static data objects for customer, eventType, hall, and menu (no DAO mocks)
- Apply proper assertions: toEqual, toThrow, toBeCloseTo
- Add clear and descriptive test names
- Mock or stub any external dependencies if present (none for static data)

PROMPT 11

Input Prompt

Create Jest unit tests for Booking's applyPaidServices() function with these test cases:

1. Valid single service — correctly calculates applied price and updates originalPrice.
2. Multiple valid services — verifies cumulative total and correct services array transformation.
3. Service not found in allServices — throws "Service ID not found.".
4. Service already free (exists in #freeServices) — throws "Service ID is already free.".
5. Empty this.services array — keeps originalPrice unchanged and returns empty array.
6. Quantity = 0 (edge case) — applied price is 0, no error thrown.
7. Negative quantity (invalid input) — should throw or produce NaN (verify handling).
8. Decimal price values — ensures floating-point precision with toBeCloseTo.
9. Very large quantity and high price — verifies no overflow or precision loss.
10. Missing serviceID field — throws "Service ID undefined not found.".

Requirements:

- Use Jest framework
- Include setup/teardown with beforeEach / afterEach
- Use static data objects for customer, eventType, hall, and menu (no DAO mocks)
- Apply proper assertions: toEqual, toThrow, toBeCloseTo
- Add clear and descriptive test names
- Mock or stub any external dependencies if present (none for static data)

PROMPT 12

Input Prompt

Create Jest unit tests for Booking's calculateTotals() function with these test cases:

1. Valid VAT rate (0.1) — correctly calculates VAT and totalAmount.
2. Missing VAT_RATE — defaults to 0.08 and computes totals correctly.
3. Invalid VAT_RATE string ("abc") — falls back to default 0.08.
4. Negative VAT_RATE — resets to 0.08 as fallback.
5. originalPrice not a number — throws "originalPrice must be a number.".
6. discountAmount not a number — throws "discountAmount must be a number.".
7. discountAmount greater than originalPrice — results in totalAmount = 0 (non-negative clamp).
8. Very large values for price and discount — verifies no overflow or precision loss.
9. Decimal VAT and prices — uses toBeCloseTo for floating-point precision.
10. Repeated calls with modified values — ensures recalculation and idempotent behavior.

Requirements:

- Use Jest framework
- Include setup/teardown with beforeEach / afterEach
- Use static data objects for customer, eventType, hall, and menu (no DAO mocks)
- Apply proper assertions: toEqual, toThrow, toBeCloseTo
- Add clear and descriptive test names
- Mock or stub any external dependencies if present (none for static data)

PROMPT 12

Input prompt

Create Jest unit tests for Booking's validateEventDate() function with these test cases:

1. Valid Future Date
2. Exactly Minimum Notice
3. Exactly Maximum Notice
4. Date Too Soon
5. Date Too Far
6. Invalid Input: Missing
7. Invalid Input: Bad Format
8. Invalid Input: Impossible Date 9.Invalid systemSettings: Negative Numberany valid date 10.Invalid Input: Impossible Date'2025-02-30'

Requirements:

- Use Jest framework
- Include setup/teardown
- Use proper assertions (toEqual, toThrow)
- Add descriptive test names
- Mock any external dependencies

RUN & DEBUG

```
expect(received).rejects.toThrow(expected)

Expected substring: "Event type record not found in database."
Received message:   "Customer record not found in database."

      102 |         const menu = await this.menuDAO.findById(this.menuID);
      103 |
    > 104 |         if (!customer) throw new Error("Customer record not found in database.");
          |                                ^
      105 |         if (!eventType) throw new Error("Event type record not found in
database.");
      106 |         if (!hall) throw new Error("Hall record not found in database.");
      107 |         if (!menu) throw new Error("Menu record not found in database.");

      at Booking.validateRequiredEntities (src/Booking.js:104:30)
      at Object.<anonymous> (tests/Booking.validateRequiredEntities.test.js:117:9)

  117 |         await expect(booking.validateRequiredEntities())
  118 |             .rejects
> 119 |             .toThrow("Event type record not found in database.");
      |              ^
  120 |     });
  121 |
  122 |     // 8️⃣ Invalid hallID

  at Object.toThrow (node_modules/expect/build/index.js:2155:20)
  at Object.toThrow (tests/Booking.validateRequiredEntities.test.js:119:14)
```

Booking.validateRequiredEntities() › ❌ should throw an error when eventTypeID is invalid

```
expect(received).rejects.toThrow(expected)

Expected substring: "Hall record not found in database."
Received message:   "Customer record not found in database."

      102 |         const menu = await this.menuDAO.findById(this.menuID);
      103 |
    > 104 |         if (!customer) throw new Error("Customer record not found in database.");
          |                                ^
      105 |         if (!eventType) throw new Error("Event type record not found in database.");
      106 |         if (!hall) throw new Error("Hall record not found in database.");
      107 |         if (!menu) throw new Error("Menu record not found in database.");

      at Booking.validateRequiredEntities (src/Booking.js:104:30)
      at Object.<anonymous> (tests/Booking.validateRequiredEntities.test.js:127:9)

  127 |         await expect(booking.validateRequiredEntities())
  128 |             .rejects
> 129 |             .toThrow("Hall record not found in database.");
      |              ^
  130 |     });
  131 | });

  at Object.toThrow (node_modules/expect/build/index.js:2155:20)
  at Object.toThrow (tests/Booking.validateRequiredEntities.test.js:129:14)"
```

Booking.validateRequiredEntities() › ❌ should throw an error when hallID is invalid

PROMPT 13

"Help me fix this failing unit test: FAIL tests/Booking.validateRequiredEntities.test.js Booking.validateRequiredEntities()

√ ✅ should resolve successfully when all entities are found (4 ms)

√ ❌ should throw an error when customer record is missing (13 ms)

√ ❌ should throw an error when event type record is missing (1 ms)

√ ❌ should throw an error when hall record is missing (2 ms)

√ ❌ should throw an error when menu record is missing (1 ms)

√ ❌ should throw an error when customerID is invalid

× ❌ should throw an error when eventTypeID is invalid (13 ms)

× ❌ should throw an error when hallID is invalid (3 ms)

PROMPT 13

```
// 7️⃣ Invalid eventTypeID
it("❌ should throw an error when eventTypeID is invalid", async () => {
  booking.eventTypeID = undefined;

  // All prior DAOs must return valid data
  mockCustomerDAO.findById.mockResolvedValue(mockCustomer);
  mockEventTypeDAO.findById.mockResolvedValue(null); // invalid
  mockHallDAO.findById.mockResolvedValue(mockHall);
  mockMenuDAO.findById.mockResolvedValue(mockMenu);

  await expect(booking.validateRequiredEntities())
    .rejects
    .toThrow("Event type record not found in database.");
});

// 8️⃣ Invalid hallID
it("❌ should throw an error when hallID is invalid", async () => {
  booking.hallID = undefined;

  // All prior DAOs must return valid data
  mockCustomerDAO.findById.mockResolvedValue(mockCustomer);
  mockEventTypeDAO.findById.mockResolvedValue(mockEventType);
  mockHallDAO.findById.mockResolvedValue(null); // invalid
  mockMenuDAO.findById.mockResolvedValue(mockMenu);

  await expect(booking.validateRequiredEntities())
    .rejects
    .toThrow("Hall record not found in database.");
});
```

WHY THIS WORKS

The function checks entities **in strict order**:

```
customer → eventType → hall → menu
```

So to test a specific failure, every entity **before** it in the chain must pass. Otherwise, the function short-circuits on the first failure, giving misleading errors.

✅ **After this fix:**

- All eight tests will pass.
- The error order logic will be properly verified.

OPTIMIZATION

PROMPT 14

Input Prompt

Generate a fully optimized set of Jest unit test files for the Booking class, focusing on the Booking Creation feature. The tests should be modular, reusable, maintainable, and cover all important scenarios: happy paths, boundary conditions, and error handling.

FPT University - October 2025

THANK YOU SO MUCH!

Presented by AI_13