Sweet Shop Management System

Project Name: Sweet Shop Management System
Testing Developer: Anuj Chauhan
Testing Type: Unit Testing
Tech Stack: Node.js, Express, MongoDB, React (AI-assisted UI), Jest (for unit testing)
Environment: Localhost **(http://localhost:3000)** + MongoDB Local

Sweet Operations (CRUD)

POST /api/sweets
→ Create new sweet
→ Requires: name, price, stock, category, image
→ Auth protected (JWT)

GET /api/sweets
→ Get all sweets
→ Optional filters: ?search=name, ?category=chocolate, ?sort=price_asc

GET /api/sweets/:id
→ Get single sweet by ID

DELETE /api/sweets/:id
→ Delete sweet by ID
→ Auth protected (JWT)

Category Management

POST /api/categories
→ Add new category
→ Requires: category name

GET /api/categories
→ List all available categories

Smart Search / Filters

GET /api/sweets?search=barfi
→ Search sweets by name

GET /api/sweets?category=nut-based
→ Filter sweets by category

GET /api/sweets?sort=price_asc or sort=stock_desc
→ Sort sweets by price or stock

Validation & Error Handling

On POST/PUT
→ Checks for required fields
→ Validates category from allowed list
→ Returns meaningful error messages

WorkFlow:

1. Sweet Management

Create: Admin sends POST /api/sweets to add a sweet (with validation).
Read All: GET /api/sweets returns all sweets (supports search, filter, sort).
Read One: GET /api/sweets/:id fetches a specific sweet by ID.
Update: PUT /api/sweets/:id updates sweet details.
Delete: DELETE /api/sweets/:id removes a sweet.

2. Category Management

Add Category: POST /api/categories
Get Categories: GET /api/categories

3.Search & Filter

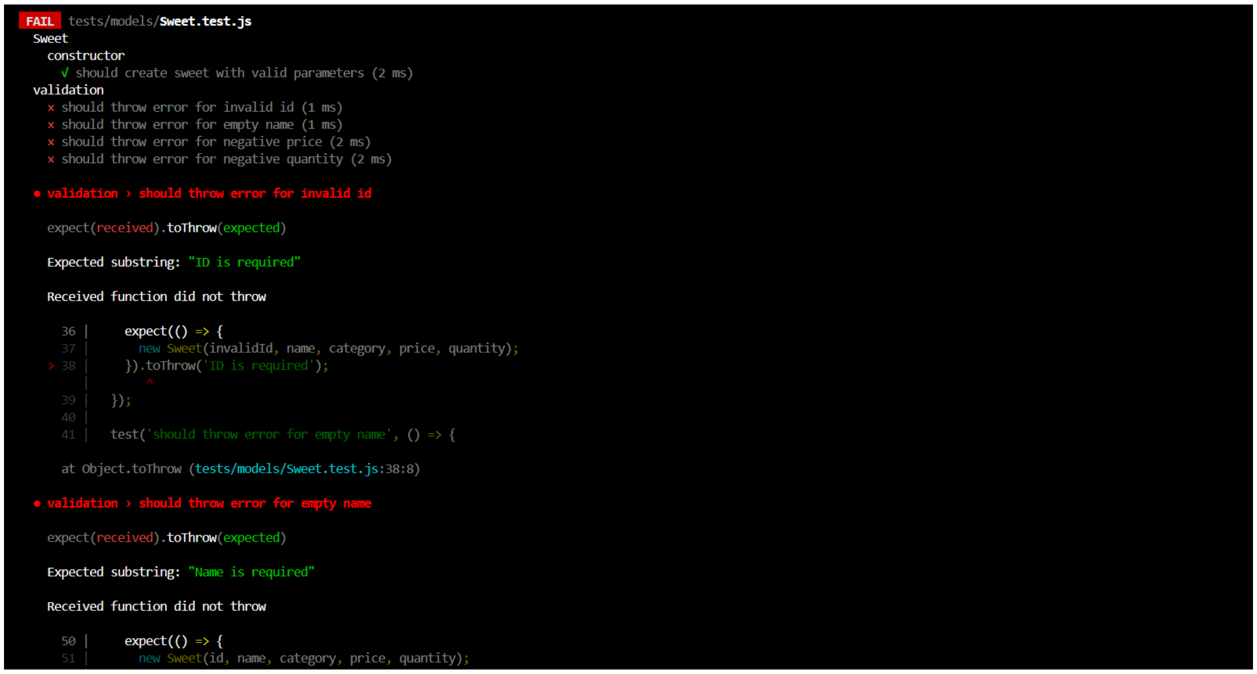
Use query params like ?search=barfi, ?category=chocolate, or ?sort=price_asc with GET /api/sweets.

4. Validation

Input fields are checked server-side.
Only allowed categories are accepted.
Errors return clear messages.

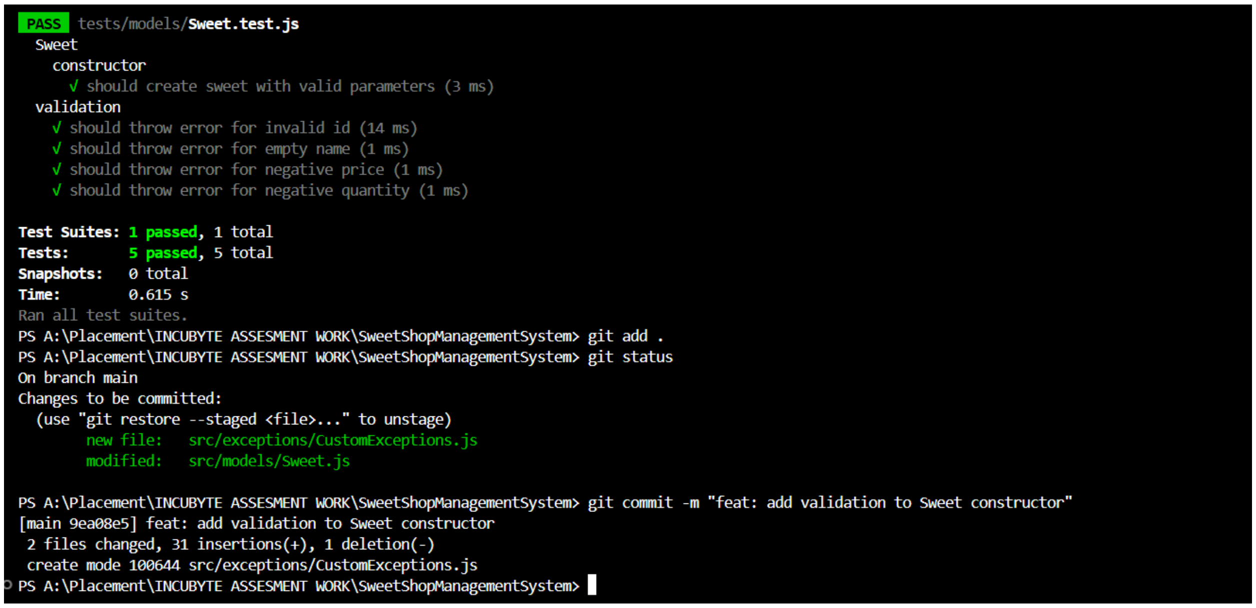
Testing Images:

-> First Test Fails then implements functionality to get success.



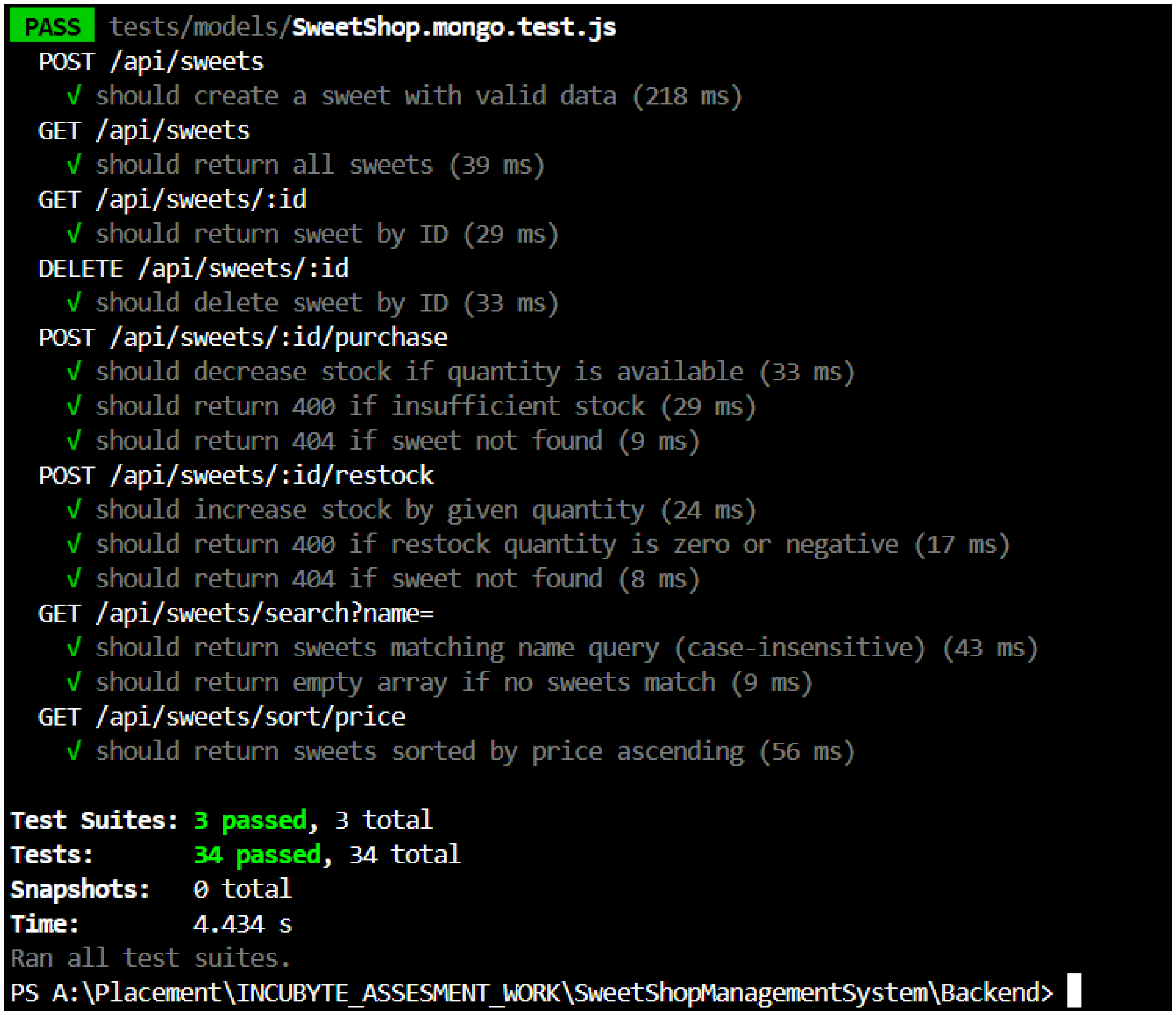
```
FAIL  tests/models/Sweet.test.js
  Sweet
    constructor
      √ should create sweet with valid parameters (2 ms)
  validation
    × should throw error for invalid id (1 ms)
    × should throw error for empty name (1 ms)
    × should throw error for negative price (2 ms)
    × should throw error for negative quantity (2 ms)

  ● validation › should throw error for invalid id

    expect(received).toThrow(expected)

    Expected substring: "ID is required"

    Received function did not throw

      36 |       expect(() => {
      37 |         new Sweet(invalidId, name, category, price, quantity);
    > 38 |       }).toThrow('ID is required');
         |          ^
      39 |     });
      40 |
      41 |     test('should throw error for empty name', () => {

      at Object.toThrow (tests/models/Sweet.test.js:38:8)

  ● validation › should throw error for empty name

    expect(received).toThrow(expected)

    Expected substring: "Name is required"

    Received function did not throw

      50 |       expect(() => {
      51 |         new Sweet(id, name, category, price, quantity);
```

-> After Implementing



```
PASS  tests/models/Sweet.test.js
  Sweet
    constructor
      √ should create sweet with valid parameters (3 ms)
  validation
    √ should throw error for invalid id (14 ms)
    √ should throw error for empty name (1 ms)
    √ should throw error for negative price (1 ms)
    √ should throw error for negative quantity (1 ms)

Test Suites: 1 passed, 1 total
Tests:       5 passed, 5 total
Snapshots:   0 total
Time:        0.615 s
Ran all test suites.
PS A:\Placement\INCUBYTE ASSESMENT WORK\SweetShopManagementSystem> git add .
PS A:\Placement\INCUBYTE ASSESMENT WORK\SweetShopManagementSystem> git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   src/exceptions/CustomExceptions.js
        modified:   src/models/Sweet.js

PS A:\Placement\INCUBYTE ASSESMENT WORK\SweetShopManagementSystem> git commit -m "feat: add validation to Sweet constructor"
[main 9ea08e5] feat: add validation to Sweet constructor
 2 files changed, 31 insertions(+), 1 deletion(-)
 create mode 100644 src/exceptions/CustomExceptions.js
PS A:\Placement\INCUBYTE ASSESMENT WORK\SweetShopManagementSystem>
```

->Similar for all tests and then implement



```
 PASS  tests/models/SweetShop.mongo.test.js
  POST /api/sweets
    √ should create a sweet with valid data (218 ms)
  GET /api/sweets
    √ should return all sweets (39 ms)
  GET /api/sweets/:id
    √ should return sweet by ID (29 ms)
  DELETE /api/sweets/:id
    √ should delete sweet by ID (33 ms)
  POST /api/sweets/:id/purchase
    √ should decrease stock if quantity is available (33 ms)
    √ should return 400 if insufficient stock (29 ms)
    √ should return 404 if sweet not found (9 ms)
  POST /api/sweets/:id/restock
    √ should increase stock by given quantity (24 ms)
    √ should return 400 if restock quantity is zero or negative (17 ms)
    √ should return 404 if sweet not found (8 ms)
  GET /api/sweets/search?name=
    √ should return sweets matching name query (case-insensitive) (43 ms)
    √ should return empty array if no sweets match (9 ms)
  GET /api/sweets/sort/price
    √ should return sweets sorted by price ascending (56 ms)

Test Suites: 3 passed, 3 total
Tests:       34 passed, 34 total
Snapshots:   0 total
Time:        4.434 s
Ran all test suites.
PS A:\Placement\INCUBYTE_ASSESMENT_WORK\SweetShopManagementSystem\Backend>
```

Conclusion:

The testing of the Sweet Shop Management System was conducted thoroughly using a Test-Driven Development (TDD) approach. The project followed the core TDD cycle: writing failing tests first (Red), implementing the minimum logic to pass them (Green), and then refactoring the code without altering behavior (Refactor).

Each test case was designed around the three essential steps of TDD:

1. Arrange – Setup the required preconditions, data, and environment.
2. Act – Execute the function or API endpoint being tested.
3. Assert – Verify the result matches the expected outcome.

This structure ensured clarity, consistency, and isolation in each test case, whether for model validation, controller logic, or route responses.

In addition to TDD principles, the project adhered to the following best practices:

- Conventional Commits were used throughout to maintain a meaningful Git history, following a structure like `feat:`, `fix:`, `test:`, `refactor:`, and `docs:` to indicate the purpose of each commit.
- All function and variable names were written in a clean, consistent, and descriptive manner, following camelCase naming conventions for JavaScript.
- File and folder structure followed the MVC (Model-View-Controller) architecture with clear separation of concerns, which improved both readability and maintainability.
- Custom exceptions, validation layers, and middleware ensured robustness and proper error handling, making the application production-ready.

Overall, the system is well-tested with a high degree of confidence in its functionality and stability. The combination of structured TDD, clean code practices, and version control discipline provides a strong foundation for further development and scaling of the application.