The objective of this Kata is to design, build, and test a full-stack Sweet Shop Management System. This project will test skills in API development, database management, frontend implementation, testing, and modern development workflows, including the use of AI tools.

Step one involves the initial repository setup. Your first few commits need to establish the environment. Remember to use the AI co-author trailer if you rely on tools for generating boilerplate code.

You start by initializing the project. Set up your backend with Node.js or TypeScript, Python, Java, or Ruby. Then handle the frontend using React, Vue, Angular, or Svelte.

Next comes the database connection. Configure a persistent one like PostgreSQL, MongoDB, or SQLite to keep things running smoothly.

Do not forget the README initialization. Create the README.md file with sections for project explanation and setup instructions. Include the mandatory My AI Usage section right there.

Step two focuses on backend TDD implementation.

The core of your evaluation rests on the Red-Green-Refactor pattern. You must write tests before any implementation for each part that follows.

That includes user authentication. For registration, you handle POST to api/auth/register. For login, it is POST to api/auth/login, and you implement JWT along the way.

Then move to sweets management, which stays protected. Implement CRUD operations for sweets. Make sure to include the specific api/sweets/search endpoint in there.

The data structure requires each sweet object to have a unique ID, name, category, price, and quantity. For admin security, ensure DELETE to api/sweets/:id works only for Admin users.

Inventory logic comes next. The purchase endpoint, POST to api/sweets/:id/purchase, must decrease the quantity properly. For restock, POST to api/sweets/:id/restock increases the quantity, and it stays restricted to Admins.

Step three covers frontend development.

Build your SPA to interact with the API. Keep a strong focus on UX and UI throughout the process.

Forms need attention first. Build login and registration interfaces that feel straightforward.

The dashboard displays sweets. Include search and filtering logic to make it useful.

Purchase logic requires the button to disable programmatically if the sweet quantity hits zero. That prevents any mix-ups.

For admin UI, create separate forms. They handle adding, updating, and deleting sweets without issues.

Step four deals with AI usage transparency. Since you use AI, your commit history matters as much as the code itself.

Commit format stays consistent. Every time you use an AI tool for code generation or debugging, include in the message Co-authored-by: AI Tool Name <AI@users.noreply.github.com>.

Documentation fits in the README.md. Reflect on how tools like Gemini or Copilot impacted your workflow. Specify exactly what they helped with in the process.

The final deliverables checklist helps before submitting your GitHub link for review. Ensure you have a clear README.md with all required sections.

Include screenshots of the application in action to show it works. Add a test report that shows high coverage and successful pass rates.

Code

```
import pytest
from unittest.mock import Mock

# Assume these are imported from the files we previously discussed
from sweet_shop.backend.models import UserCreate, UserInDB
from sweet_shop.backend.services.auth_service import UserService, AuthException
from sweet_shop.backend.security import verify_password

# ------------------------------------------------
# Pytest Fixtures for Mocking Dependencies
# ------------------------------------------------

@pytest.fixture
def mock_db_service():
    """Fixture to create a mock database session/service."""
    # Mock object simulates DB access without a real database
    return Mock()


@pytest.fixture
def auth_service(mock_db_service):
    """Fixture to instantiate the UserService with mocked DB dependency."""
    return UserService(db_dependency=mock_db_service)
```

```
# ------------------------------------------------
# RED PHASE: TESTS FOR REGISTRATION (POST /api/auth/register)
# ------------------------------------------------

def test_register_user_success(auth_service, mock_db_service):
    """TC 1.1: Should successfully register a new user and hash the password."""

    # Arrange
    mock_db_service.get_user_by_email.return_value = None

    user_data = UserCreate(
        email="test@user.com",
        username="tester",
        password="securepassword123",
    )

    # Act
    new_user = auth_service.register_user(user_data)

    # Assert
    mock_db_service.create_user.assert_called_once()

    assert new_user.email == user_data.email
    assert new_user.role == "customer"  # default role

    # Check hashed password (indirectly)
    saved_user_data = mock_db_service.create_user.call_args[0][0]
    assert saved_user_data["hashed_password"] != user_data.password
    assert verify_password(
```

```
        user_data.password, saved_user_data["hashed_password"]
    ) is True


def test_register_user_duplicate_email_fails(auth_service, mock_db_service):
    """TC 1.2: Should raise AuthException if email already exists."""

    # Arrange
    existing_user = UserInDB(
        id=1,
        email="duplicate@mail.com",
        username="dup",
        role="customer",
    )
    mock_db_service.get_user_by_email.return_value = existing_user

    user_data = UserCreate(
        email="duplicate@mail.com",
        username="new_user",
        password="pass",
    )

    # Act & Assert
    with pytest.raises(AuthException) as excinfo:
        auth_service.register_user(user_data)

    assert "Email already registered" in str(excinfo.value)
    mock_db_service.create_user.assert_not_called()
```

```
# ------------------------------------------------
# RED PHASE: TESTS FOR LOGIN (POST /api/auth/login)
# ------------------------------------------------

def test_authenticate_user_success(auth_service, mock_db_service):
    """TC 2.1: Should return the user object upon successful login."""

    # Arrange
    mock_user = UserInDB(
        id=10,
        email="admin@shop.com",
        username="admin",
        role="admin",
    )
    mock_db_service.get_user_by_email.return_value = mock_user

    # Mock password verification
    auth_service.verify_password = Mock(return_value=True)

    # Act
    authenticated_user = auth_service.authenticate_user(
        email="admin@shop.com",
        password="correct_password",
    )

    # Assert
    assert authenticated_user == mock_user
    auth_service.verify_password.assert_called_once()
```

```
def test_authenticate_user_bad_password_fails(auth_service, mock_db_service):
    """TC 2.2: Should return None for incorrect password."""

    # Arrange
    mock_user = UserInDB(
        id=11,
        email="user@shop.com",
        username="user",
        role="customer",
    )
    mock_db_service.get_user_by_email.return_value = mock_user

    auth_service.verify_password = Mock(return_value=False)

    # Act
    authenticated_user = auth_service.authenticate_user(
        email="user@shop.com",
        password="wrong_password",
    )

    # Assert
    assert authenticated_user is None



def test_authenticate_user_not_found_fails(auth_service, mock_db_service):
    """TC 2.3: Should return None if the user email is not found."""

    # Arrange
    mock_db_service.get_user_by_email.return_value = None
```

```
# Act
authenticated_user = auth_service.authenticate_user(
    email="unknown@shop.com",
    password="any_password",
)

# Assert
assert authenticated_user is None

# No password verification should occur if user does not exist
assert not hasattr(auth_service, "verify_password") or not getattr(
    auth_service.verify_password, "called", False
)
```