UserChecksTest Documentation

Overview

This document details the unit tests implemented for the UserChecks class in our e-commerce application backend. The UserChecks service is responsible for validating user input data during registration and profile management, ensuring data integrity and security by verifying that email addresses, passwords, and location information meet the required standards.

Test Class Structure

Dependencies

The test class uses:

- **JUnit 5**: For test execution and assertions
- **Mockito**: To mock dependencies and simulate interactions
- **Spring Boot Test**: For integration with the Spring testing framework

Mocked Components

- **UserRepository**: Database access for user operations
- **UserService**: Service for user-related operations including checking if an email is already in use

Test Setup

Before each test, the following setup is performed:

1. Initialize mocks using MockitoAnnotations

Test Cases

1. testEmailChecks_ValidEmail

Purpose: Verify that a valid email address passes validation checks.

Test Scenario:

- **Arrange**:
    - Create a valid email address "test@gmail.com"
    - Configure userService mock to return false when checking if the email is taken
- **Act**:
    - Call userChecks.emailChecks with the valid email
- **Assert**:
    - Verify the response body is empty (indicating no errors)
    - Verify the response has 200 status code

Business Logic Verified:

- Valid email addresses are accepted by the system
- The system correctly responds with a success status when email format is valid
- The system also verifies that the email is not already in use

2. testEmailChecks_InvalidEmailFormat

Purpose: Verify that an invalid email format is rejected.

Test Scenario:

- **Arrange**:

    - Create an invalid email address "testinvalid" (missing @ and domain)
- **Act**:
    - Call userChecks.emailChecks with the invalid email
- **Assert**:
    - Verify the response body contains "Invalid email format" error message
    - Verify the response has 400 status code

Business Logic Verified:

- The system rejects malformed email addresses
- Appropriate error messages are returned to guide users
- The system uses correct HTTP status codes for validation errors

3. testPasswordChecks_ValidPassword

Purpose: Verify that a valid password passes validation checks.

Test Scenario:

- **Arrange**:
    - Create a valid password "Password1!" that meets complexity requirements
- **Act**:
    - Call userChecks.passwordChecks with the valid password
- **Assert**:
    - Verify the response body is empty (indicating no errors)
    - Verify the response has 200 status code

Business Logic Verified:

- The system accepts passwords that meet security requirements
- The password validation rules are correctly implemented

- The system responds with success status for valid passwords

4. testCityChecks_ValidCity

Purpose: Verify that a valid city name passes validation checks.

Test Scenario:

- **Arrange**:
    - Create a valid city name "İSTANBUL"
- **Act**:
    - Call userChecks.cityChecks with the valid city
- **Assert**:
    - Verify the response body is empty (indicating no errors)
    - Verify the response has 200 status code

Business Logic Verified:

- The system accepts valid city names
- City validation correctly handles non-ASCII characters (like İ)
- The system responds with success status for valid cities

5. testCityChecks_InvalidCity

Purpose: Verify that an invalid city name is rejected.

Test Scenario:

- **Arrange**:
    - Create an invalid city name "INVALIDCITY"
- **Act**:
    - Call userChecks.cityChecks with the invalid city

- **Assert**:
    - Verify the response body contains "Invalid city" error message
    - Verify the response has 400 status code

Business Logic Verified:

- The system rejects city names that are not in the allowed list
- Appropriate error messages are returned to guide users
- The system uses correct HTTP status codes for validation errors

Mocking Strategy

The tests use a consistent mocking strategy to isolate the UserChecks service from its dependencies:

1. **Mock Responses**: Configure the userService mock to return predetermined results
2. **Minimal Mocking**: Only mock the specific method calls needed for each test
3. **Response Verification**: Focus on validating the response content and status codes

Test Coverage

These tests cover the core validation functionality of the UserChecks service:

- Email format validation
- Password complexity validation
- City name validation against allowed values

Conclusion

The UserChecksTest thoroughly verifies the input validation mechanisms used in the application. The tests ensure that user-provided data is properly validated before being accepted by the system, which helps maintain data integrity and security.

These tests help enforce consistent validation rules throughout the application, ensuring that users receive clear feedback when their input doesn't meet the required standards. This improves the user experience by catching errors early in the process and providing meaningful guidance.