COMP 3550

8.2 — WRITING INTEGRATION & ACCEPTANCE TESTS

Week 8: Advanced Testing

WHAT MAKES A TEST ‘INTEGRATION’ LEVEL?

Definition:

- An **integration test** verifies that two or more real components correctly interact with each other in a real runtime environment.

WHAT MAKES A TEST ‘INTEGRATION’ LEVEL?

Definition:

- An **integration test** verifies that two or more real components correctly interact with each other in a real runtime environment.

Key Characteristics:

- Tests multiple modules together
  - e.g., Service ⇔ Repository, Controller ⇔ Service
- Uses real dependencies, not mocks/stubs
- Do not mock the component you're testing
  - if you do, it's not a true integration test
- Often runs against a test database, filesystem, or HTTP endpoint

WHAT MAKES A TEST ‘INTEGRATION’ LEVEL?

Definition:

- An **integration test** verifies that two or more real components correctly interact with each other in a real runtime environment.

Key Characteristics:

- Tests multiple modules together
  - e.g., Service ⇔ Repository, Controller ⇔ Service
- Uses real dependencies, not mocks/stubs
- Do not mock the component you're testing
  - if you do, it's not a true integration test
- Often runs against a test database, filesystem, or HTTP endpoint

Example Scenarios:

- OrderService saving to a real (test) database
- LoginController calling real validation logic and auth services
- File parser reads actual test file and produces results

DB INTEGRATION TEST EXAMPLE

Key Elements of a Good DB Integration Test:

Setup

- Create test database (in-memory)
- Insert or prepare sample data

Action

- Call actual RecipeService.save() or .findById() methods
- No mocks — real persistence logic runs

Assertions

- Confirm record is saved, retrieved, updated, or deleted as expected
- Check DB contents or returned object state

DB INTEGRATION TEST EXAMPLE

```
@BeforeEach
void setup() {
    dataSource = new H2Database().start();
    recipeRepo = new RecipeRepository(dataSource);
    recipeService = new RecipeService(recipeRepo);
}

@Test
void savesRecipeCorrectly() {
    Recipe r = new Recipe("Lasagna", List.of("pasta", "cheese"));
    recipeService.save(r);

    Recipe saved = recipeService.findByName("Lasagna");
    assertEquals("Lasagna", saved.getName());
    assertTrue(saved.getIngredients().contains("cheese"));
}
```

ACCEPTANCE TESTS

Does the system do what the user expects?

A test that verifies whether a feature behaves as intended, based on user stories, requirements, or scenarios.

Key Characteristics:

- Focuses on outcomes, not implementation
- Often tests end-to-end, from input to output
- Written from the user's perspective ("Given → When → Then" style)
- May use tools like Cucumber, JBehave, or plain code + test frameworks
- Often automated in agile or CI/CD pipelines

USER STORY:

"AS A USER, I WANT TO SAVE A RECIPE SO I CAN VIEW IT LATER."

This simulates user behavior across the full stack (controller, service, DB).

```
@Test
void userCanSaveAndRetrieveRecipe() {
    client.submitRecipe("Chili", List.of("beans", "beef"));
    var recipe = client.getRecipe("Chili");

    assertEquals("Chili", recipe.getName());
    assertTrue(recipe.getIngredients().contains("beef"));
}
```

TEST NAMING & SYNTAX CONVENTIONS

Code-Based Syntax

- Common usage in Unit/Integration Tests
- Starts with test (JUnit 4, Python, etc.)
  - testUserCanSaveAndRetrieveRecipe()
  - testThrowsErrorWhenUsernameIsEmpty()
- Clear what feature or behavior is being tested
- Helps test runners auto-detect tests
- Avoid vague names like test1() or checkStuff()

Reccomended Format:

test[Condition]_When[Trigger]_Then[ExpectedOutcome]

TEST NAMING & SYNTAX CONVENTIONS

Behavior-Driven (BDD) / Gherkin Syntax:

- Used in tools like Cucumber, SpecFlow, and human-readable test cases
- Common usage in Acceptance/UI Level Tests

Feature: Save Recipes
  Scenario: User saves a valid recipe
    Given the user is on the "New Recipe" page
    When they submit "Chili" with ingredients
    Then the recipe should appear in their recipe list

- Great for acceptance tests tied to user stories
- Easily reviewed by non-developers (QA, PMs, etc.)
- Focused on behavior, not code structure

ORGANIZING LARGER TESTS

- Organize test files to mirror your main code structure:
- Group by feature/module, not just by type of test (unit, integration, etc.)
- Use consistent naming: UserServiceTest, UserIntegrationTest, etc.

```
/src
  └── main
      └── java
          └── recipe/
              └── RecipeService.java

/src
  └── test
      └── java
          └── recipe/
              └── RecipeServiceTest.java
```

TEST SUITES (GROUPING TESTS)

- Combine related tests into a suite to run together
- Useful for:
  - Smoke tests before deployment
  - Regression test packs
  - Running only fast unit tests in CI
- More in COMP 4550

SETUP & TEARDOWN BEST PRACTICES

- Use @BeforeEach / @AfterEach for isolated state per test
- Use @BeforeAll / @AfterAll only for expensive shared setup (e.g., test DB boot)

```
@BeforeEach
void setup() {
    db.clear();
    userService = new UserService(db);
}
```

- Keep tests independent: one test’s data shouldn’t affect another
- Clean up test artifacts (files, connections, temp data)

PROJECT PAUSE & REFLECT

Choose one feature you finished this week and write a skeleton acceptance test for it.